開發人員指南

AWS Lambda

AWS Lambda: 開發人員指南

Table of Contents

什麼是 AWS Lambda？

您可以使用 AWS Lambda 在不佈建或管理伺服器的情況下執行程式碼。

Lambda 在高可用性的運算基礎設施上執行您的程式碼，並執行所有運算資源的管理，包括伺服器與作業系統維護、容量佈建與自動擴展以及記錄。使用 Lambda，您唯一需要做的就是在 Lambda 支援的其中一種語言執行期中提供您的程式碼。

您可以將您的程式碼組織為 Lambda 函數。Lambda 服務只有在需要時才會執行您的函數，並會自動擴展。只需為使用的運算時間支付費用，一旦未執行程式碼，就會停止計費。如需詳細資訊，請參閱 AWS Lambda 定價。

> **Tip**
>
> 若要了解如何建置無伺服器解決方案，請參閱無伺服器開發人員指南。

使用 Lambda 的時機

Lambda 是理想的運算服務，適用於需要快速縱向擴展的應用程式案例，並在不需要時縮減規模至零。例如，您可將 Lambda 用於：

- 檔案處理：使用 Amazon Simple Storage Service (Amazon S3)，在上傳之後即時觸發 Lambda 資料處理程序。
- 串流處理：使用 Lambda 和 Amazon Kinesis 處理即時串流資料，以進行應用程式活動追蹤、交易訂單處理、點選流分析、資料清理、日誌篩選、索引編制、社交媒體分析、物聯網 (IoT) 裝置資料遙測以及計量。
- Web 應用程式：將 Lambda 與其他 AWS 服務結合，以建置功能強大的 Web 應用程式，這些應用程式可自動擴展和縮減，並在多個資料中心的高可用性組態中
- IoT 後端：使用 Lambda 建置無伺服器後端，用於處理 Web、行動裝置、IoT 和第三方 API 請求。
- 行動後端：使用 Lambda 和 Amazon API Gateway 建置後端，用於驗證和處理 API 請求。使 AWS Amplify 用可輕鬆整合您的 iOS、安卓系統、網頁和反應原生前端。

使用 Lambda 時，您只需負責程式碼的相關操作。Lambda 會管理提供記憶體、CPU、網路和其他資源平衡的運算機群，以執行您的程式碼。由於 Lambda 管理這些資源，因此您無法登入運算執行個體

或在提供的執行時間自訂作業系統。Lambda 會代表您執行操作和管理活動，包括管理容量、監控和記錄您的 Lambda 函數。

主要功能

下列主要功能協助您開發可擴展、安全且易於擴充的 Lambda 應用程式：

環境變數

使用環境變數來調整函數的行為，而無需更新程式碼。

版本

使用版本來管理函數部署，例如，可以使用新函數進行 Beta 測試，而不會影響穩定生產版本的使用者。

容器映像

使用 AWS 提供的基礎映像或替代基礎映像，為 Lambda 函數建立容器映像，以便重複使用現有的容器工具，或部署依賴相依性的大型工作負載，例如機器學習。

圖層

封裝程式庫和其他相依項可減少部署存檔的大小，並可更快地部署程式碼。

Lambda 延伸

使用監控、觀察、安全和管理工具來增強您的 Lambda 函數。

函數 URL

將專用 HTTP(S) 端點新增至 Lambda 函數。

回應串流

設定您的 Lambda 函數 URL，將回應承載從 Node.js 函數串流回用戶端，以提高第一個位元組時間 (TTFB) 效能或傳回較大的承載。

並行和擴展控制

對生產應用程式的擴展和回應能力進行精細控制。

程式碼簽署

確認只有核准的開發人員可在 Lambda 函數中發佈未修改、受信任的程式碼

私有網路

為資料庫、快取執行個體或內部服務等資源建立私有網路。

檔案系統存取

設定函數，將 Amazon Elastic File System (Amazon EFS) 掛載到本機目錄，使您的函數程式碼能夠安全地且高度並行地存取和修改共用資源。

爪哇 SnapStart 的 Lambda

將 Java 執行期的啟動效能提升最高 10 倍，無需額外費用，且通常不需要變更函數程式碼。

開始使用 Lambda

若要開始使用 Lambda，請使用 Lambda 主控台來建立函數。您可以在幾分鐘內建立和部署函數，並在主控台中加以測試。

進行教學課程時，您會學到一些基本 Lambda 概念，例如如何使用 Lambda「事件物件」，將引數傳遞給函數。您還將學習如何從函數返回日誌輸出，以及如何在日誌中查看函數的調用日 CloudWatch 誌。

為了簡化，您可以使用 Python 或 Node.js 執行期建立函數。您可以使用這些轉譯語言，直接在主控台的內建程式碼編輯器中編輯函數程式碼。使用 Java 和 C# 等編譯語言，您必須在本機建置機器上建立部署套件，並將其上傳至 Lambda。若要了解如何使用其他執行期，將函數部署至 Lambda，請參閱the section called “其他資源和後續步驟”一節中的連結。

> **Tip**
>
> 若要了解如何建置無伺服器解決方案，請參閱無伺服器開發人員指南。

必要條件

註冊一個 AWS 帳戶

如果您沒有 AWS 帳戶，請完成以下步驟來建立一個。

若要註冊成為 AWS 帳戶

1. 開啟 https://portal.aws.amazon.com/billing/signup。
2. 請遵循線上指示進行。

   部分註冊程序需接收來電，並在電話鍵盤輸入驗證碼。

   當您註冊一個時 AWS 帳戶，將創建AWS 帳戶根使用者一個。根使用者有權存取該帳戶中的所有 AWS 服務 和資源。安全性最佳做法是將管理存取權指派給使用者，並僅使用 root 使用者來執行需要 root 使用者存取權的工作。

AWS 註冊過程完成後，會向您發送確認電子郵件。您可以隨時登錄 https://aws.amazon.com/ 並選擇我的帳戶，以檢視您目前的帳戶活動並管理帳戶。

建立具有管理權限的使用者

註冊後，請保護您的 AWS 帳戶 AWS 帳戶根使用者 AWS IAM Identity Center、啟用和建立系統管理使用者，這樣您就不會將 root 使用者用於日常工作。

保護您的 AWS 帳戶根使用者

1. 選擇 Root 使用者並輸入您的 AWS 帳戶 電子郵件地址，以帳戶擁有者身分登入。AWS Management Console在下一頁中，輸入您的密碼。

   如需使用根使用者登入的說明，請參閱 AWS 登入 使用者指南中的以根使用者身分登入。
2. 若要在您的根使用者帳戶上啟用多重要素驗證 (MFA)。

   如需指示，請參閱《IAM 使用者指南》中的為 AWS 帳戶 根使用者啟用虛擬 MFA 裝置 (主控台)。

建立具有管理權限的使用者

1. 啟用 IAM Identity Center。

   如需指示，請參閱 AWS IAM Identity Center 使用者指南中的啟用 AWS IAM Identity Center。
2. 在 IAM 身分中心中，將管理存取權授予使用者。

   若要取得有關使用 IAM Identity Center 目錄 做為身分識別來源的自學課程，請參閱《使用指南》IAM Identity Center 目錄中的「以預設值設定使用AWS IAM Identity Center 者存取」。

以具有管理權限的使用者身分登入

- 若要使用您的 IAM Identity Center 使用者簽署，請使用建立 IAM Identity Center 使用者時傳送至您電子郵件地址的簽署 URL。

  如需使用 IAM 身分中心使用者登入的說明，請參閱使用AWS 登入 者指南中的登入 AWS 存取入口網站。

指派存取權給其他使用者

1. 在 IAM 身分中心中，建立遵循套用最低權限許可的最佳做法的權限集。

   如需指示，請參閱《AWS IAM Identity Center 使用指南》中的「建立權限集」。
2. 將使用者指派給群組，然後將單一登入存取權指派給群組。

如需指示，請參閱《AWS IAM Identity Center 使用指南》中的「新增群組」。

使用主控台建立一個 Lambda 函數

在此範例中，您的函數會使用 JSON 物件，其中包含兩個標示 "length" 和 "width" 的整數值。函數會將這些值相乘以計算區域，並以 JSON 字串傳回。

您的函數也會列印計算的區域，以及其 CloudWatch 記錄群組的名稱。稍後在教學課程中，您將學習如何使用 CloudWatch Logs 來檢視函式叫用的記錄。

若要建立函數，請先使用主控台，建立基本 Hello world 函數。在下一步中，您會新增自己的函數程式碼。

若要使用主控台建立 Hello world Lambda 函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇 建立函數。
3. 選取從頭開始撰寫。
4. 在基本資訊窗格中，為函數名稱輸入 **myLambdaFunction**。
5. 針對執行期，選擇 Node.js 20.x 或 Python 3.12
6. 將架構設定為 x86_64，然後選擇建立函數。

Lambda 會建立函數，傳回 `Hello from Lambda!` 訊息。Lambda 也會為函數建立「執行角色」。執行角色是一種 AWS Identity and Access Management (IAM) 角色，可授與 Lambda 函數存取 AWS 服務 和資源的權限。對於您的函數，Lambda 建立的角色會授與寫入 CloudWatch 記錄的基本權限。

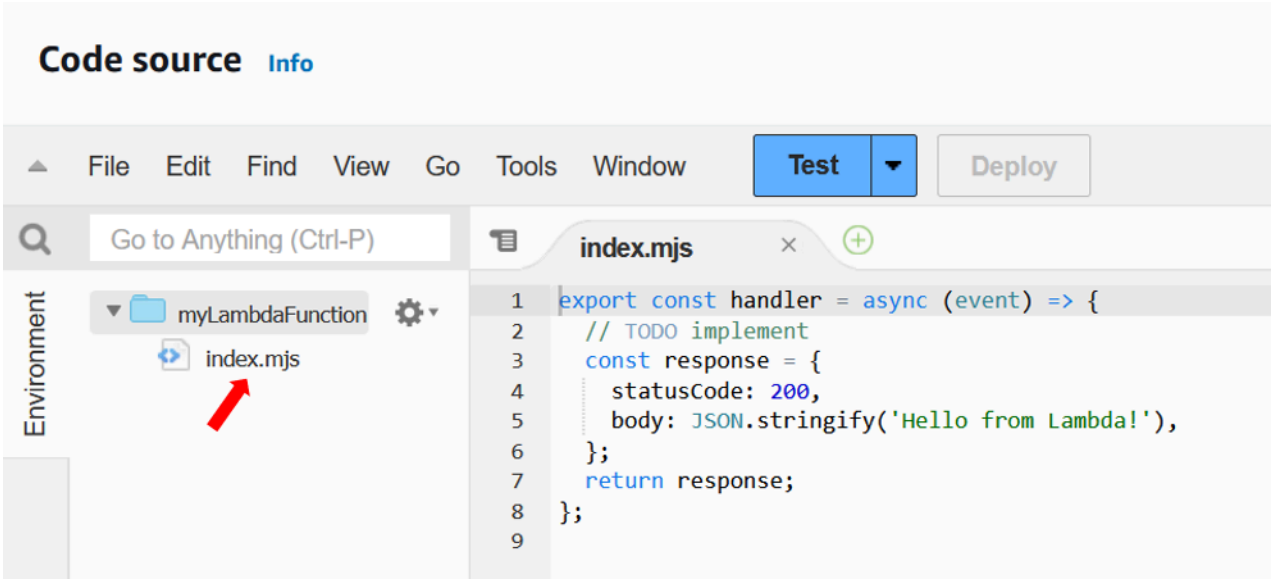
您現在要使用主控台的內建程式碼編輯器，將 Lambda 建立的 Hello world 程式碼，替換為您自己的函數程式碼。

Node.js

若要在主控台中修改程式碼

1. 選擇 程式碼 標籤。

在主控台的內建程式碼編輯器中，您應該會看到 Lambda 建立的函數程式碼。如果您在程式碼編輯器中沒看到 index.mjs 標籤，請在檔案總管中選取 index.mjs，如下圖所示。



2. 將以下程式碼貼到 index.mjs 標籤中，替換 Lambda 建立的程式碼。

```
export const handler = async (event, context) => {

  const length = event.length;
  const width = event.width;
  let area = calculateArea(length, width);
  console.log(`The area is ${area}`);

  console.log('CloudWatch log group: ', context.logGroupName);

  let data = {
    "area": area,
  };
    return JSON.stringify(data);

  function calculateArea(length, width) {
    return length * width;
  }
};
```

3. 選取部署以更新您函數的程式碼。Lambda 部署變更後，主控台會顯示橫幅，讓您知道已成功更新函數。

了解函數程式碼

在進行下一步之前，讓我們花一點時間看看函數程式碼，並了解一些重要 Lambda 概念。

- Lambda 處理常式：

  您的 Lambda 函數包含 Node.js 函數 `handler`。以 Node.js 編寫的 Lambda 函數可以包含多個 Node.js 函數，但「handler」函數始終是程式碼的進入點。當有人調用您的函數時，Lambda 會執行此方法。

  使用主控台建立 Hello world 函數時，Lambda 會自動將函數的處理常式方法名稱設定為 `handler`。請勿編輯此 Node.js 函數的名稱。如果這麼做，調用函數時，Lambda 將無法執行您的程式碼。

  若要進一步了解以 Node.js 編寫的 Lambda 處理常式，請參閱 the section called “處理常式”。

- Lambda 事件物件：

  `handler` 函數會使用兩個引數，`event` 和 `context`。Lambda 中的「事件」是一種 JSON 格式的文件，它包含供函數處理的資料。

  如果您的函數是由另一個函數調用 AWS 服務，則事件對象包含有關導致調用的事件的信息。舉例來說，如果 Amazon Simple Storage Service (Amazon S3) 儲存貯體在上傳物件時調用您的函數，該事件會包含 Amazon S3 儲存貯體的名稱和物件索引鍵。

  在此範例中，您會進入有兩個索引鍵/值組的 JSON 格式文件，在主控台中建立事件。

- Lambda 內容物件：

  您函數使用的第二個引數為 `context`。Lambda 會自動將「內容物件」傳遞至您的函數。內容物件包含有關函數調用及執行環境的資訊。

  您可以使用內容物件，基於監控目的，輸出函數調用的資訊。在此範例中，您的函數會使用`logGroupName`參數來輸出其 CloudWatch 記錄群組的名稱。

  若要進一步了解以 Node.js 編寫的 Lambda 內容物件，請參閱 the section called “Context”。

- 在 Lambda 中記錄：

  您可以透過 Node.js，使用 `console.log` 和 `console.error` 等主控台方法，將資訊傳送到函數的日誌。範例程式碼會使用`console.log`陳述式來輸出計算的區域以及函數之 CloudWatch Logs 群組的名稱。您也可以使用任何寫入 `stdout` 或 `stderr` 的記錄程式庫。

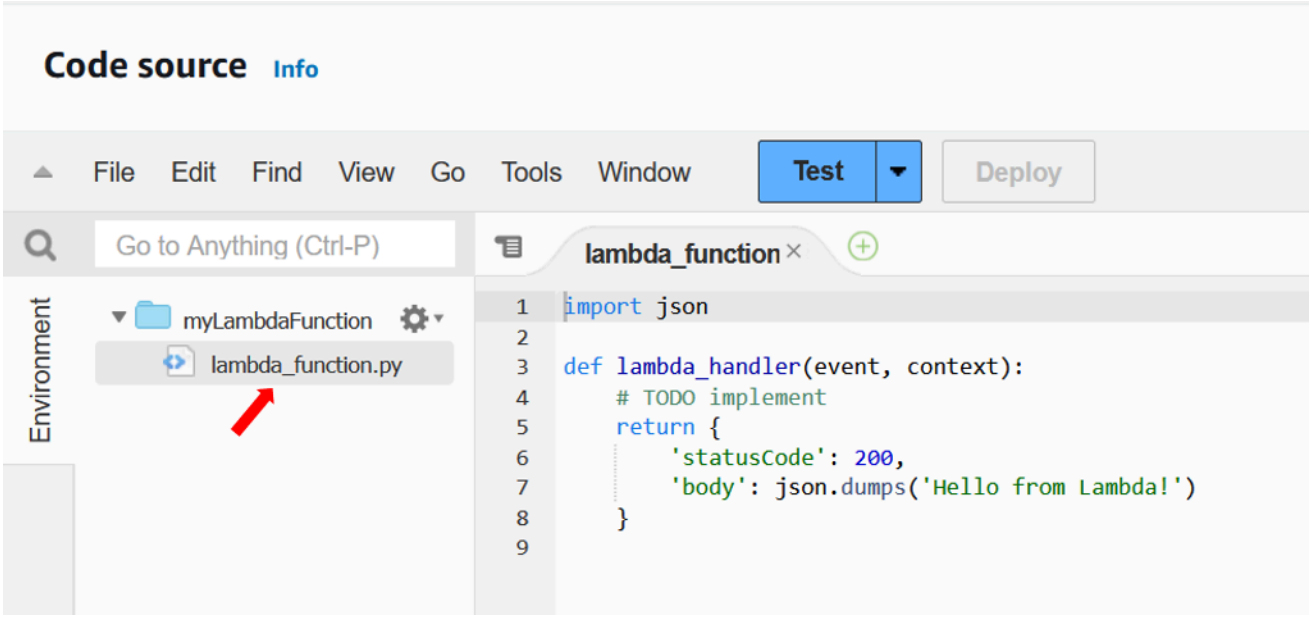
如需進一步了解，請參閱the section called “日誌”。若要了解如何在其他執行期中記錄，請參閱您有興趣之執行期的「建置方式」頁面。

Python

若要在主控台中修改程式碼

1. 選擇 程式碼 標籤。

   在主控台的內建程式碼編輯器中，您應該會看到 Lambda 建立的函數程式碼。如果您在程式碼編輯器中沒看到 lambda_function.py，請在檔案總管中選取 lambda_function.py，如下圖所示。



2. 將以下程式碼貼到 lambda_function.py 標籤中，替換 Lambda 建立的程式碼。

```
import json
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)

def lambda_handler(event, context):

    # Get the length and width parameters from the event object. The
    # runtime converts the event object to a Python dictionary
    length = event['length']
```

```
    width = event['width']

    area = calculate_area(length, width)
    print(f"The area is {area}")

    logger.info(f"CloudWatch logs group: {context.log_group_name}")

    # return the calculated area as a JSON string
    data = {"area": area}
    return json.dumps(data)

def calculate_area(length, width):
    return length*width
```

3. 選取部署以更新您函數的程式碼。Lambda 部署變更後，主控台會顯示橫幅，讓您知道已成功更新函數。

了解函數程式碼

在進行下一步之前，讓我們花一點時間看看函數程式碼，並了解一些重要 Lambda 概念。

- Lambda 處理常式：

  Lambda 函數包含 Python 函數 `lambda_handler`。以 Python 編寫的 Lambda 函數可以包含多個 Python 函數，但「handler」函數始終是程式碼的進入點。當有人調用您的函數時，Lambda 會執行此方法。

  使用主控台建立 Hello world 函數時，Lambda 會自動將函數的處理常式方法名稱設定為 `lambda_handler`。請勿編輯此 Python 函數的名稱。如果這麼做，調用函數時，Lambda 將無法執行您的程式碼。

  若要進一步了解以 Python 編寫的 Lambda 處理常式，請參閱the section called “處理常式”。

- Lambda 事件物件：

  `lambda_handler` 函數會使用兩個引數，`event` 和 `context`。Lambda 中的「事件」是一種 JSON 格式的文件，它包含供函數處理的資料。

  如果您的函數是由另一個函數調用 AWS 服務，則事件對象包含有關導致調用的事件的信息。舉例來說，如果 Amazon Simple Storage Service (Amazon S3) 儲存貯體在上傳物件時調用您的函數，該事件會包含 Amazon S3 儲存貯體的名稱和物件索引鍵。

在此範例中，您會進入有兩個索引鍵/值組的 JSON 格式文件，在主控台中建立事件。

- Lambda 內容物件：

  您函數使用的第二個引數為 `context`。Lambda 會自動將「內容物件」傳遞至您的函數。內容物件包含有關函數調用及執行環境的資訊。

  您可以使用內容物件，基於監控目的，輸出函數調用的資訊。在此範例中，您的函數會使用`log_group_name`參數來輸出其 CloudWatch 記錄群組的名稱。

  若要進一步了解以 Python 編寫的 Lambda 內容物件，請參閱the section called “Context”。

- 在 Lambda 中記錄：

  透過 Python，您可以使用 `print` 陳述式或 Python 記錄程式庫，將資訊傳送到函數的日誌。為了說明擷取內容的差異，範例程式碼會使用這兩種方法。在生產應用程式中，建議您使用記錄程式庫。

  如需進一步了解，請參閱the section called “日誌”。若要了解如何在其他執行期中記錄，請參閱您有興趣之執行期的「建置方式」頁面。

使用主控台調用 Lambda 函數

若要使用 Lambda 主控台調用函數，請先建立要傳送至函數的測試事件。事件是一種 JSON 格式文件，其中包含兩個索引鍵/值組，索引鍵 `"length"` 和 `"width"`。

若要建立測試事件

1. 在程式碼來源窗格中選擇測試。
2. 選取建立新事件。
3. 事件名稱輸入 **`myTestEvent`**。
4. 在事件 JSON 面板中，貼上下列項目以替換預設值：

```
{
  "length": 6,
  "width": 7
}
```

5. 選擇儲存。

現在，您可以測試函數，並使用 Lambda 主控台和記 CloudWatch 錄來檢視函數叫用的記錄。

若要在主控台中測試您的函數並檢視調用記錄

- 在程式碼來源窗格中選擇測試。當您的函數執行完時，您會看到回應和函數日誌顯示在執行結果標籤。您應該會看到類似下列的結果。

  Node.js

```
Test Event Name
myTestEvent

Response
"{\"area\":42}"

Function Logs
START RequestId: 5c012b0a-18f7-4805-b2f6-40912935034a Version: $LATEST
2023-08-31T23:39:45.313Z 5c012b0a-18f7-4805-b2f6-40912935034a INFO The area is
 42
2023-08-31T23:39:45.331Z 5c012b0a-18f7-4805-b2f6-40912935034a INFO CloudWatch
 log group:  /aws/lambda/myLambdaFunction
END RequestId: 5c012b0a-18f7-4805-b2f6-40912935034a
REPORT RequestId: 5c012b0a-18f7-4805-b2f6-40912935034a Duration: 20.67 ms Billed
 Duration: 21 ms Memory Size: 128 MB Max Memory Used: 66 MB Init Duration:
 163.87 ms

Request ID
5c012b0a-18f7-4805-b2f6-40912935034a
```

  Python

```
Test Event Name
myTestEvent

Response
"{\"area\": 42}"

Function Logs
START RequestId: 2d0b1579-46fb-4bf7-a6e1-8e08840eae5b Version: $LATEST
The area is 42
[INFO] 2023-08-31T23:43:26.428Z 2d0b1579-46fb-4bf7-a6e1-8e08840eae5b CloudWatch
 logs group: /aws/lambda/myLambdaFunction
END RequestId: 2d0b1579-46fb-4bf7-a6e1-8e08840eae5b
```

```
REPORT RequestId: 2d0b1579-46fb-4bf7-a6e1-8e08840eae5b Duration: 1.42 ms Billed
 Duration: 2 ms Memory Size: 128 MB Max Memory Used: 39 MB Init Duration: 123.74
 ms

Request ID
2d0b1579-46fb-4bf7-a6e1-8e08840eae5b
```

在此範例中，您使用主控台的測試功能調用程式碼。這表示您可以直接在主控台中查看函數的執行結果。當您的函數在控制台外部調用時，您需要使用 CloudWatch 日誌。

在日誌中查看函數的調用記錄 CloudWatch

1. 開啟主控台的 [記錄群組] 頁 CloudWatch 面。
2. 為函數 (`/aws/lambda/myLambdaFunction`) 選擇日誌群組名稱。這是您的函數列印至主控台的日誌群組名稱。
3. 在日誌串流標籤上，為函數調用選擇日誌串流。

   您應該會看到類似下列的輸出：

   Node.js

   ```
   INIT_START Runtime Version: nodejs:20.v13    Runtime Version ARN:
    arn:aws:lambda:us-
   west-2::runtime:e3aaabf6b92ef8755eaae2f4bfdcb7eb8c4536a5e044900570a42bdba7b869d9
   START RequestId: aba6c0fc-cf99-49d7-a77d-26d805dacd20 Version: $LATEST
   2023-08-23T22:04:15.809Z    5c012b0a-18f7-4805-b2f6-40912935034a  INFO The area
    is 42
   2023-08-23T22:04:15.810Z    aba6c0fc-cf99-49d7-a77d-26d805dacd20  INFO
    CloudWatch log group:  /aws/lambda/myLambdaFunction
   END RequestId: aba6c0fc-cf99-49d7-a77d-26d805dacd20
   REPORT RequestId: aba6c0fc-cf99-49d7-a77d-26d805dacd20    Duration: 17.77 ms
    Billed Duration: 18 ms    Memory Size: 128 MB    Max Memory Used: 67 MB    Init
    Duration: 178.85 ms
   ```

   Python

   ```
   INIT_START Runtime Version: python:3.12.v16    Runtime Version ARN:
    arn:aws:lambda:us-
   west-2::runtime:ca202755c87b9ec2b58856efb7374b4f7b655a0ea3deb1d5acc9aee9e297b072
   START RequestId: 9d4096ee-acb3-4c25-be10-8a210f0a9d8e Version: $LATEST
   ```

```
The area is 42
[INFO] 2023-09-01T00:05:22.464Z 9315ab6b-354a-486e-884a-2fb2972b7d84 CloudWatch
 logs group: /aws/lambda/myLambdaFunction
END RequestId: 9d4096ee-acb3-4c25-be10-8a210f0a9d8e
REPORT RequestId: 9d4096ee-acb3-4c25-be10-8a210f0a9d8e    Duration: 1.15 ms
 Billed Duration: 2 ms    Memory Size: 128 MB    Max Memory Used: 40 MB
```

清除

使用完範例函數時，請加以刪除。您還可以刪除存放函數日誌的日誌群組，以及主控台建立的執行角色。

刪除 Lambda 函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇 動作 、刪除 。
4. 在 刪除函數 對話方塊中輸入 delete，然後選擇 刪除 。

刪除日誌群組

1. 開啟主控台的 [記錄群組] 頁 CloudWatch面。
2. 選取函數的日誌群組 (`/aws/lambda/my-function`)。
3. 選擇 動作 、刪除日誌群組 。
4. 在 刪除日誌群組 對話方塊中，選擇 刪除 。

刪除執行角色

1. 開啟 AWS Identity and Access Management (IAM) 主控台的「角色」頁面。
2. 選取函數的執行角色，(例如 `myLambdaFunction-role-31exxmpl`)。
3. 選擇 刪除 。
4. 在 刪除角色 對話方塊中輸入角色名稱，然後選擇 刪除 。

您可以使用和 AWS Command Line Interface (AWS CLI) 自動建立和清理函數、記錄群組 AWS CloudFormation 和角色。

其他資源和後續步驟

既然您已使用主控台建立並測試簡單的 Lambda 函數，接著請採取下列後續步驟：

- 了解將相依項新增程式碼中，並使用 .zip 部署套件加以部署。請從下列連結中選擇您有興趣的語言。

  Node.js

  請參閱 the section called “部署 .zip 封存檔”

  Typescript

  請參閱 the section called “部署 .zip 封存檔”

  Python

  請參閱 the section called “部署 .zip 封存檔”

  Ruby

  請參閱 the section called “部署 .zip 封存檔”

  Java

  請參閱 the section called “部署 .zip 封存檔”

  Go

  請參閱 the section called “部署 .zip 封存檔”

  C#

  請參閱 the section called “部署套件”
- 進行教學課程使用 Amazon S3 觸發條件調用 Lambda 函數，了解如何設定 Lambda 函數以供另一個 AWS 服務調用。
- 選擇下列任一教學課程，獲得搭配其他 AWS 服務使用 Lambda 更複雜的範例。
  - 搭配 API Gateway 使用 Lambda：建立調用 Lambda 函數的 Amazon API Gateway REST API。
  - 使用 Lambda 函數存取 Amazon RDS 資料庫：使用 Lambda 函數，透過 RDS 代理將資料寫入 Amazon Relational Database Service (Amazon RDS) 資料庫。
  - 使用 Amazon S3 觸發條件建立縮圖影像：每次將影像檔案上傳到 Amazon S3 儲存貯體時，使用 Lambda 函數建立縮圖。

AWS Lambda 基金會

Lambda 函數是 Lambda 服務的主要資源。

您可以使用 Lambda 主控台、Lambda API AWS CloudFormation 或 AWS SAM. 您可以建立函數的程式碼，並使用部署套件上傳程式碼。當事件發生時，Lambda 會調用函數。Lambda 會同時執行多個函數執行個體，並由並行和擴展限制所控制。

主題

Lambda 概念

Lambda 會執行函數的執行個體來處理事件。您可以使用 Lambda API 來直接叫用您的函數，也可以設定 AWS 服務或資源來叫用您的函數。

概念

- 函數
- 觸發條件
- 事件
- 執行環境
- 指令集架構
- 部署套件
- 執行期
- Layer
- 延伸
- 並行數量
- 限定詞
- 目的地

函數

函數是您可以叫用以在 Lambda 中執行程式碼的資源。函數具有程式碼，用於處理您傳遞至函數的事件，或傳送至函數的其他 AWS 服務。

觸發條件

觸發是叫用 Lambda 函數的資源或組態。觸發程式包括您可設定以叫用函數和事件來源映射的 AWS 服務。事件來源映射是 Lambda 中的資源，它可從串流或佇列中讀取項目並叫用函數。如需詳細資訊，請參閱 了解 Lambda 函數叫用方法 和 使用來自其 AWS 他服務的事件叫用 Lambda。

事件

事件是一種 JSON 格式的文件，會包含供 Lambda 函數處理的資料。執行時間會將事件轉換為物件，再將它傳遞到您的函數程式碼。當您呼叫函數時，您決定事件的結構和內容。

Example 自訂事件 - 天氣資料

```
{
  "TemperatureK": 281,
  "WindKmh": -3,
  "HumidityPct": 0.55,
  "PressureHPa": 1020
}
```

當某項 AWS 服務叫用您的函式時，此服務會定義事件雛型。

Example 服務事件 - Amazon SNS 通知

```
{
  "Records": [
    {
      "Sns": {
        "Timestamp": "2019-01-02T12:45:07.000Z",
        "Signature": "tcc6faL2yUC6dgZdmrwh1Y4cGa/ebXEkAi6RibDsvpi+tE/1+82j...65r==",
        "MessageId": "95df01b4-ee98-5cb9-9903-4c221d41eb5e",
        "Message": "Hello from SNS!",
        ...
```

如需 AWS 服務事件的詳細資訊，請參閱 使用來自其 AWS 他服務的事件叫用 Lambda。

執行環境

執行環境為您的 Lambda 函數提供了安全且隔離的執行時間環境。執行環境會管理執行函數所需的程序和資源。執行環境為函數以及與函數關聯的任何延伸項目提供生命週期支援。

如需詳細資訊，請參閱 Lambda 執行環境。

指令集架構

指令集架構會決定 Lambda 用來執行函數的電腦處理器類型。Lambda 提供了指令集架構的選擇：

- arm64 - 64 位元的 ARM 架構，適用於 AWS Graviton2 處理器
- x86_64 - 64 位元 x86 架構，適用於 x86 處理器。

如需詳細資訊，請參閱 設定 Lambda 函數的指令集架構。

部署套件

使用部署套件部署 Lambda 函數程式碼。Lambda 支援兩種部署套件：

- 包含函數程式碼及其相依項目的 .zip 封存檔。Lambda 為您的函數提供作業系統和執行時間。
- 與開放式容器方案 (OCI) 規範相容的容器映像。您可以將函數程式碼和相依項新增至映像。還必須包含作業系統和 Lambda 執行時間。

如需詳細資訊，請參閱 Lambda 部署套件。

執行期

執行時間會提供在執行環境中執行的特定語言環境。執行時間會轉送叫用事件、內容資訊以及 Lambda 與函數之間的回應。您可以使用由 Lambda 提供的執行時間或是自行建置。如果您將程式碼封裝為 .zip 封存檔，則必須將函數設定為使用符合程式設計語言的執行時間。針對容器映像，您可以在建置映像時包含執行時間。

如需詳細資訊，請參閱 Lambda 執行期。

Layer

Lambda 層是可以包含其他程式碼或資料的 .zip 封存檔。圖層可以包含程式庫、自訂執行階段、資料或組態檔案。

利用圖層可封裝與 Lambda 函數搭配使用的程式庫和其他依存項。使用圖層可減少已上傳的部署存檔的大小，並可更快地部署程式碼。圖層還可促進程式碼共用和責任分離，以便您可以在撰寫商業邏輯時更快地重複執行。

每個函數最多可包含五個圖層。層會計入標準 Lambda 部署大小配額。在函數中包含圖層時，內容會擷取到執行環境中的 `/opt` 目錄。

根據預設，您建立的圖層對 AWS 帳戶來說是私有的。您可選擇與其他帳戶共用圖層或將圖層公開。如果您的函數使用不同帳戶發佈的圖層，在已刪除該圖層版本之後，或者撤銷該圖層的存取許可之後，您的函數可繼續使用該圖層版本。但是，您不能使用已刪除的圖層版本來建立新函數或更新函數。

部署為容器映像的函數不使用圖層。而是在您建置映像時，將偏好的執行時間、程式庫和其他相依項封裝為容器映像。

如需詳細資訊，請參閱 Lambda 層。

延伸

Lambda 擴展可讓您增強函數。例如，您可以使用延伸項目將您的函式與偏好的監控、可觀度、安全性和控管工具整合。您可以從 AWS Lambda 合作夥伴提供的廣泛工具集中選擇，也可以建立自己的 Lambda 擴展。

內部延伸項目會在執行階段程序中執行，並與執行階段共用相同的生命週期。外部延伸項目會在執行環境中做為個別的程序執行。在叫用函式之前初始化外部延伸項目，並與函式的執行階段平行執行，並在函式叫用完成後繼續執行。

如需詳細資訊，請參閱 使用 Lambda 擴充功能擴充功能擴充。

並行數量

並行數量是函式在任何指定時間服務的請求數。叫用函數時，Lambda 會佈建函數的執行個體來處理事件。當函數程式碼完成執行時，它會處理另一項請求。如果在請求仍在處理時再次呼叫該函數，則會佈建另一個執行個體，進而增加函數的並行數量。

並行數量受限於 AWS 區域層級配額。您可以設定個別函式來限制其並行數量，或讓它們可以達到特定的並行數量層級。如需詳細資訊，請參閱 為函數配置保留並發。

限定詞

當您叫用或檢視函式時，您可以包含限定詞來指定版本或別名。版本是具有數值限定詞的函式代碼和配置的不可變快照。例如 `my-function:1`。別名是版本的指標，您可以進行更新以映射至不同的版本，或分割兩個版本之間的流量。例如 `my-function:BLUE`。您可以同時使用版本和別名，為用戶端提供穩定的界面來叫用您的函數。

如需詳細資訊，請參閱 Lambda 函數版本。

目的地

目的地是一個 AWS 資源，Lambda 可在其中從非同步叫用中傳送事件。可對處理失敗的事件設定目的地。某些服務也支援處理成功的事件的目的地。

如需更多詳細資訊，請參閱 設定非同步調用的目的地。

Lambda 程式設計模型

Lambda 提供的程式設計模型對於所有執行時間通用。程式設計模型會定義程式碼與 Lambda 系統之間的介面。透過在函數組態中定義一個處理常式，將函數的進入點告知 Lambda。執行時間會將包含呼叫事件和內容的物件傳入至處理常式，例如函數名稱和請求 ID。

當處理常式完成處理第一個事件時，執行時間就會傳送至另一個。函數的類別會保留在記憶體中，因此可以重複使用在初始化程式碼的處理常式方法外宣告的用戶端和變數。為了節省後續事件的處理時間，請建立可重複使用的資源，例如初始化期間的 AWS SDK 用戶端。初始化後，函式的每個執行個體都可以處理數千個請求。

您的函數也可以存取 `/tmp` 目錄中的本機儲存。執行內容凍結時，目錄環境會凍結，所提供的暫時性可用於多重調用。如需詳細資訊，請參閱 Lambda 執行環境。

啟用AWS X-Ray追蹤時，執行時間會記錄初始化和執行的個別子區段。

執行階段會擷取函數的記錄輸出，並將其傳送至 Amazon Lo CloudWatch gs。除了記錄函式的輸出之外，執行階段也會記錄函式叫用開始和結束的項目。這包含有要求 ID、帳單期、初始化持續時間和其他詳細資料的記錄。如果函數拋出錯誤，執行時間會將該錯誤傳回給叫用者。

> **Note**
>
> 記錄會受到記CloudWatch 錄配額的限制。日誌資料可能會因節流而遺失，或在某些情況下，因函數的執行個體停止而遺失。

Lambda 會隨著需求增加執行額外的執行個體，以及隨需求降低停止執行個體，藉以擴展您的函數。此模型會導致應用程式結構的變化，例如：

- 除非另有說明，否則可能會不按順序或同時處理傳入的請求。
- 不依賴函數長期存留的執行個體，而是將應用程式的狀態存放在其他服務中。
- 使用本機儲存和類別層級物件來提高效能，但將部署套裝服務的大小和傳輸至執行環境的資料量降至最低。

如需使用慣用程式設計語言的程式設計模型實作簡介，請參閱以下章節。

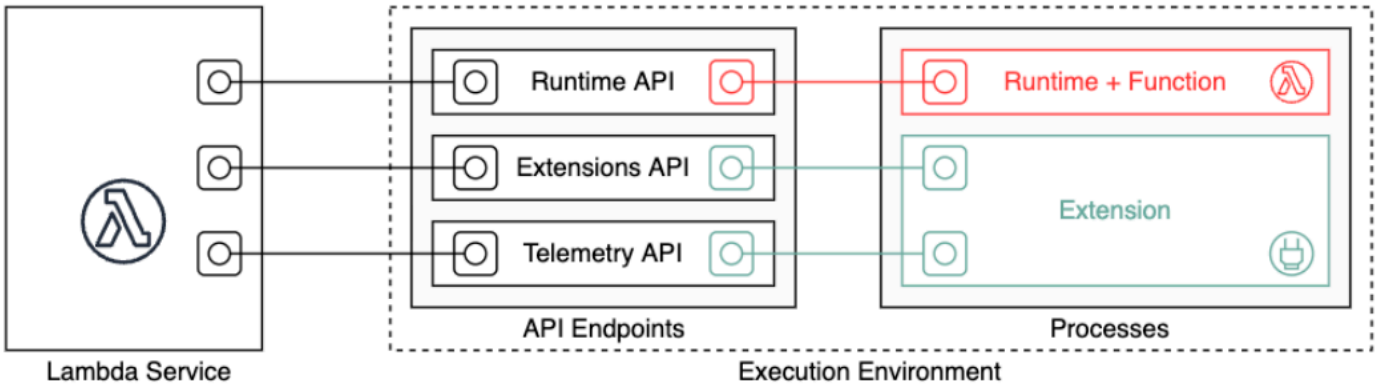
- 使用 Node.js 建置 Lambda 函數
- 使用 Python 建置 Lambda 函數

- 使用 Ruby 建置 Lambda 函數
- 使用 Java 建置 Lambda 函數
- 使用 Go 建置 Lambda 函數
- 使用 C# 建置 Lambda 函數
- 使用建置 Lambda 函數 PowerShell

Lambda 執行環境

Lambda 會在執行環境中調用您的函數，該環境可提供安全且隔離的執行時間環境。執行環境會管理執行函式所需的資源。執行環境也會提供函式執行階段的生命週期支援，以及與函式相關聯的任何外部延伸項目。

函數的執行時間會使用 Runtime API 與 Lambda 進行通訊。延伸項目會使用 Extensions API 與 Lambda 進行通訊。延伸項目還可以透過使用遙測 API 來接收函數的日誌訊息和其他遙測項目。



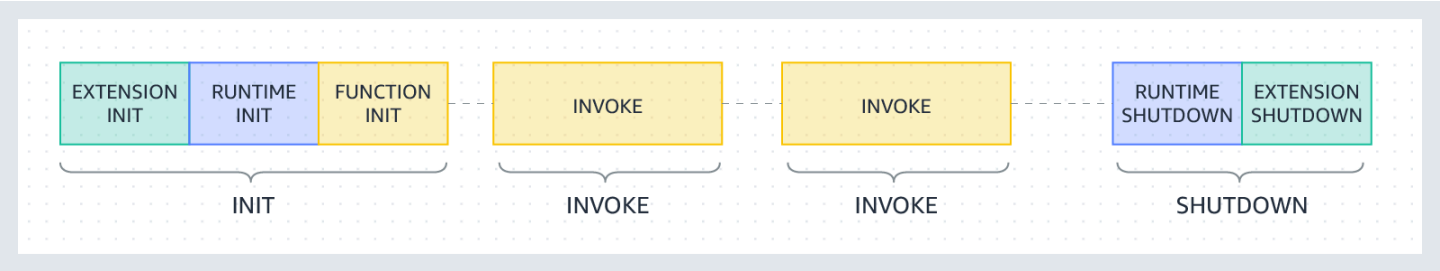
當您建立 Lambda 函數時，您將指定組態資訊，例如您的函數允許的記憶體數量與執行時間上限。Lambda 會使用此資訊來設定執行環境。

函式的執行階段和每個外部延伸項目都是在執行環境中執行的程序。許可、資源、認證和環境變數會在函式和延伸項目之間共用。

主題

- Lambda 執行環境生命週期
- 在函數中實現無狀態

Lambda 執行環境生命週期



每個階段都以 Lambda 傳送到執行階段和所有已註冊延伸項目的事件開始。執行時間和每個已註冊延伸項目都會透過傳送 Next API 請求來表示已完成。當執行時間和每個延伸項目已完成且沒有擱置的事件時，Lambda 凍結執行環境。

主題

初始化階段

在 Init 階段中，Lambda 會執行三項任務：

- 啟動所有延伸項目 (Extension init)
- Bootstrap 執行時間 (Runtime init)
- 執行該函式的靜態代碼 (Function init)
- 執行任何執行beforeCheckpoint階段掛鉤 (SnapStart 僅限 Lambda)

當執行階段和所有延伸項目透過傳送 Next API 請求發出訊號表示它們已準備就緒時，Init 階段便會結束。Init 階段限制為 10 秒。如果所有三項任務都未在 10 秒內完成，Lambda 會在第一次函數調用時以設定的函數逾時重試 Init 階段。

在 Lambda SnapStart 啟動的情況下，發佈函數版本時會發生 Init 階段。Lambda 會儲存初始化執行環境的記憶體和磁碟狀態快照、保留加密的快照，並快取以進行低延遲存取。如果您有 beforeCheckpoint 執行階段掛鉤，那麼程式碼會在 Init 階段結束時執行。

> **Note**
>
> 10 秒逾時不適用於使用佈建並行或的函數。 SnapStart對於佈建的並行 SnapStart 功能，您的初始化程式碼最多可執行 15 分鐘。時間限制為 130 秒或設定的函數逾時 (最長 900 秒)，以較長者為準。

使用佈建並行時，當您設定函數的電腦設定，Lambda 會初始化執行環境。Lambda 也可確保初始化的執行環境在調用之前隨時可用。您會發現函數調用和初始化階段之間出現差距。根據函數的執行期和記憶體組態，您也會在初始化的執行環境上第一次調用時看到變數延遲。

對於使用隨需並行的函數，Lambda 偶爾會在調用請求之前初始化執行環境。發生這種情況時，您也會注意到函數初始化和調用階段之間出現時間差。建議您不要依賴此行為。

初始化階段期間出現的失敗

如果在 `Init` 階段期間函數當機或出現逾時，Lambda 會在 `INIT_REPORT` 日誌檔中發出錯誤資訊。

Example — 逾時的 INIT_REPORT 日誌

```
INIT_REPORT Init Duration: 1236.04 ms Phase: init Status: timeout
```

Example — 延伸失敗的 INIT_REPORT 日誌

```
INIT_REPORT Init Duration: 1236.04 ms Phase: init Status: error Error Type:
 Extension.Crash
```

如果`Init`階段成功，除非啟動，`INIT_REPORT`否則 Lambda 不會發出記錄SnapStart檔。 SnapStart函數總是發出`INIT_REPORT`。如需詳細資訊，請參閱 監控 Lambda SnapStart。

還原階段 (SnapStart 僅限 Lambda)

當您第一次叫用SnapStart函數並在函數擴展時，Lambda 會從持續快照恢復新的執行環境，而不是從頭開始初始化函數。如果您有 `afterRestore()` 執行階段掛鉤，程式碼會在 `Restore` 階段結束時執行。您需支付 `afterRestore()` 執行階段掛鉤期間的費用。執行階段 (JVM) 必須載入，且 `afterRestore()` 執行階段掛鉤必須在逾時限制 (10 秒) 內完成。否則，你會得到一個 SnapStartTimeoutException. `Restore` 階段完成時，Lambda 會調用函數處理常式 (調用階段)。

還原階段期間出現的失敗

如果 `Restore` 階段失敗，Lambda 會在 `RESTORE_REPORT` 日誌檔中發出錯誤資訊。

Example — 逾時的 RESTORE_REPORT 日誌

```
RESTORE_REPORT Restore Duration: 1236.04 ms Status: timeout
```

Example — 執行期勾點失敗的 RESTORE_REPORT 日誌

```
RESTORE_REPORT Restore Duration: 1236.04 ms Status: error Error Type: Runtime.ExitError
```

如需RESTORE_REPORT 日誌的詳細資訊，請參閱 監控 Lambda SnapStart。

調用階段

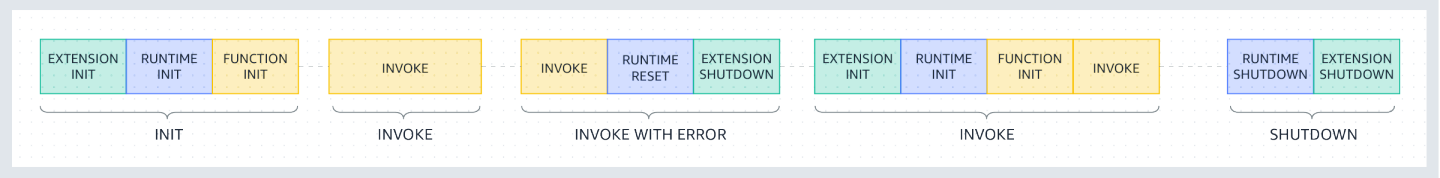
調用 Lambda 函數以回應 Next API 請求時，Lambda 會將 Invoke 事件傳送至執行時間和每個延伸項目。

該函式的逾時設定會限制整個 Invoke 階段的持續時間。例如，如果您將函式逾時設定為 360 秒，則函式和所有延伸項目都需要在 360 秒內完成。請注意，沒有獨立的調用後階段。持續時間是所有調用時間 (執行階段 + 延伸項目) 的總和，直到函式和所有延伸項目完成執行後才會計算。

調用階段會在執行階段後結束，所有延伸項目訊號都透過傳送 Next API 請求完成。

調用階段期間出現的故障

如果 Lambda 函數當機或在 Invoke 階段逾時，Lambda 會重設執行環境。下圖會說明發生調用故障時 Lambda 執行環境的行為：



在前一張示意圖中：

- 第一階段是 INIT 階段，執行期間未發生錯誤。
- 第二階段是 INVOKE 階段，執行期間未發生錯誤。
- 假設您的函數在某個時間點發生調用故障 (例如函數逾時或執行階段錯誤)。第三階段 (標記為「INVOKE WITH ERROR」) 會說明此狀況。發生此狀況時，Lambda 服務會進行重設。重設的行為會與 Shutdown 事件一樣。首先，Lambda 會關閉執行階段，然後將 Shutdown 事件傳送給每個已註冊的外部延伸項目。事件會包括關閉的原因。如果將此環境用於新的調用，Lambda 便會在下一次調用時重新初始化延伸項目和執行階段。

> **Note**
>
> 在下一個初始化階段之前，Lambda 重設不會清除 `/tmp` 目錄內容。這種行為與一般關機階段一致。

- 第四階段指的是出現調用故障後立即進入的 INVOKE 階段。在此階段中，Lambda 會透過重新執行 INIT 階段來再次初始化環境。(我們將其稱為 隱藏的初始化。) 當隱藏的插入發生時，Lambda 不會在 CloudWatch 日誌中明確報告額外的 INIT 階段。相反地，您可能會注意到 REPORT 行中的持續時間包含其他的 INIT 持續時間以及 INVOKE 持續時間。例如，假設您在中看到下列記錄檔 CloudWatch：

  ```
  2022-12-20T01:00:00.000-08:00 START RequestId: XXX Version: $LATEST
  2022-12-20T01:00:02.500-08:00 END RequestId: XXX
  2022-12-20T01:00:02.500-08:00 REPORT RequestId: XXX Duration: 3022.91 ms
  Billed Duration: 3000 ms Memory Size: 512 MB Max Memory Used: 157 MB
  ```

  在這個例子中，REPORT 和 START 時間戳記之間的差距是 2.5 秒。這與回報的 3022.91 毫秒持續時間不相符，因為它不會將 Lambda 執行的額外 INIT（隱藏的初始化）納入考量。在這個例子中，您可以推斷實際的 INVOKE 階段花費了 2.5 秒的時間。

  若要深入了解此行為，您可以使用 Lambda 遙測 API. 調用階段發生隱藏的初始化時，遙測 API 便會透過 `phase=invoke` 發送 `INIT_START`、`INIT_RUNTIME_DONE` 及 `INIT_REPORT` 事件。
- 第五階段指的是 SHUTDOWN 階段，執行期間未發生錯誤。

關閉階段

當 Lambda 即將關閉執行時間，它會將 `Shutdown` 事件傳送至每個已註冊外部延伸。延伸項目可以使用此時間進行最終清理工作。`Shutdown` 事件是對 `Next` API 請求的回應。

持續時間：整個 `Shutdown` 階段上限為 2 秒。如果執行時間或任何延伸項目沒有回應，Lambda 會透過訊號 (`SIGKILL`) 加以終止。

在函數和所有延伸項目完成之後，Lambda 會維護執行環境一段時間，並預期另一個函數調用。不過，Lambda 會每隔幾個小時終止執行環境，以允許執行階段更新和維護，即使是持續叫用的函數也是如此。您不應假設執行環境將無限期持續存在。如需詳細資訊，請參閱 在函數中實現無狀態。

再次調用該函數時，Lambda 會解凍環境以供重複使用。重複使用執行環境具有下列含義：

- 在函數處理常式方法外宣告的物件會保持初始化，於再次呼叫函數時提供額外的最佳化。例如，假設您的 Lambda 函數建立資料庫連線，而不是重建連線，那麼在後續呼叫時便會使用原始連線。建議您在程式碼中新增邏輯，在建立連線前先確認是否存在既有連線。
- 每個執行環境都會在 `/tmp` 目錄中提供 512 MB 到 10,240 MB 的磁碟空間，增量為 1 MB。執行內容凍結時，目錄環境會凍結，所提供的暫時性可用於多重調用。您可以新增額外的程式碼，確認快取是否具有您已儲存的資料。如需部署大小限制的詳細資訊，請參閱Lambda 配額。
- 如果 Lambda 重複使用執行環境，則會恢復由 Lambda 函數啟動且在函數結束時沒有完成的背景程序或回呼。請確定程式碼中的任何背景程序或回呼在程式碼存在前已完成。

在函數中實現無狀態

撰寫 Lambda 函數程式碼時，請將執行環境視為無狀態環境，假設它只存在於單一叫用中。Lambda 會每隔幾個小時終止執行環境，以允許執行階段更新和維護，即使是連續叫用的函數也是如此。在函數啟動時初始化任何必要的狀態 (例如，從 Amazon DynamoDB 表格擷取購物車)。在退出之前，請將永久性資料變更提交到亞馬遜簡單儲存服務 (Amazon S3)、DynamoDB 或 Amazon Simple Queue Service (Amazon SQS) 等耐用存放區。避免仰賴現有的資料結構、暫存檔案或跨越呼叫的狀態，例如計數器或彙總。這可確保您的函數獨立處理每次調用。

Lambda 部署套件

你的 AWS Lambda 函數的代碼由腳本或編譯的程序及其依賴關係組成。使用部署套件將函數程式碼部署到 Lambda。Lambda 支援兩種類型的部署套件：容器映像和 .zip 封存檔。

主題

容器映像

容器映像包括基礎作業系統、執行時間、Lambda 延伸項目、您的應用程式的程式碼及其相依項。您還可以將靜態資料 (如機器學習模型) 新增至映像。

Lambda 提供一組開源基礎映像，您可以將其用於建置容器映像。若要建立和測試容器映像，您可以使用 AWS Serverless Application Model (AWS SAM) 命令列介面 (CLI) 或原生容器工具，例如 Docker CLI。

將您的容器映像上傳到亞馬遜彈性容器登錄 (Amazon ECR)，這是一種受管的 AWS 容器映像登錄服務。若要將映像部署到您的函數，請使用 Lambda 主控台、Lambda API、命令列工具或 AWS 開發套件指定 Amazon ECR 影像 URL。

如需有關 Lambda 容器映像的詳細資訊，請參閱 使用容器映像檔建立 Lambda 函數。

.zip 封存檔

.zip 封存檔包含您的應用程式的程式碼及其相依項。當您使用 Lambda 主控台或工具組撰寫函數時，Lambda 會自動建立程式碼的 .zip 封存檔。

使用 Lambda API、命令列工具或 AWS SDK 建立函數時，您必須建立部署套件。如果您的函數使用已編譯的語言，或將依賴項添加到函數，則還必須創建部署包。若要部署函數的程式碼，請從 Amazon Simple Storage Service (Amazon S3) 或本機電腦上傳部署套件。

您可以使用 Lambda 主控台 AWS Command Line Interface (AWS CLI) 或將 .zip 檔案做為部署套件上傳至亞馬遜簡單儲存服務 (Amazon S3) 儲存貯體。

使用 Lambda 主控台

下列步驟示範如何使用 Lambda 主控台將 .zip 檔案上傳為您的部署套件。

若要在 Lambda 主控台中上傳 .zip 檔案

1. 開啟 Lambda 主控台中的 Functions (函數) 頁面。
2. 選取函數。
3. 在 Code Source (程式碼來源) 窗格中，選擇 Upload from (上傳來源)，然後選擇 .zip file (.zip 檔案)。
4. 選擇 Upload (上傳) 以選取您的本機 .zip 檔案。
5. 選擇儲存。

使用 AWS CLI

您可以使用 AWS Command Line Interface (AWS CLI) 上傳 .zip 檔案做為部署套件。如需語言的專屬說明，請參閱以下主題。

Node.js

使用 .zip 封存檔部署 Node.js Lambda 函數

Python

使用 .zip 封存檔部署 Python Lambda 函數

Ruby

使用 Ruby Lambda 函數的 .zip 封存檔

Java

使用 .zip 或 JAR 封存檔部署 Java Lambda 函數

Go

使用 .zip 封存檔部署 Go Lambda 函數

C#

使用 .zip 封存檔建置和部署 C# Lambda 函數

PowerShell

使用 .zip 檔案封存部署 PowerShell Lambda 函數

使用 Amazon S3

您可以使用 Amazon Simple Storage Service (Amazon S3) 將 .zip 檔案作為部署套件進行上傳。如需詳細資訊，請參閱 。

圖層

如果您使用 .zip 封存檔來部署函數程式碼，可以使用 Lambda 層作為程式庫、自訂執行時間和其他函數相依項的分配機制。Layer 讓您能夠單獨管理開發中的函數程式碼，與不會再變動的程式碼及函數所使用的資源分開。您可以將函數設定為使用您建立的圖層、 AWS 提供的圖層或來自其他 AWS 客戶的圖層。

您無法將圖層與容器影像搭配使用。而是在建置映像檔時，將您偏好的執行階段、程式庫和其他相依性封裝到容器映像檔中。

如需有關 Layer 的詳細資訊，請參閱 Lambda 層。

使用其他 AWS 服務建置部署套件

下節說明可用來封裝 Lambda 函數相依性的其他 AWS 服務。

使用 C 或 C ++ 程式庫的部署套件

如果您的部署套件包含原生程式庫，您可以使用 AWS Serverless Application Model (AWS SAM) 建置部署套件。您可以搭配使用 AWS SAM CLI `sam build` 命令`--use-container`來建立您的部署套件。此選項會在與 Lambda 執行環境相容的 Docker 映像內建置部署套件。

如需詳細資訊，請參閱 AWS Serverless Application Model 開發人員指南中的 sam 建置。

超過 50 MB 的部署套件

如果您的部署套件大於 50 MB，請將函數程式碼和相依項上傳至 Amazon S3 儲存貯體。

您可以建立部署套件，然後將 .zip 檔案上傳到您想要建立 Lambda 函數的 AWS 區域中的 Amazon S3 儲存貯體。當您建立 Lambda 函數時，請在 Lambda 主控台中指定 S3 儲存貯體名稱和物件金鑰名稱，或使用 AWS CLI。

若要使用 Amazon S3 主控台建立儲存貯體，請參閱 Amazon 簡單儲存服務使用者指南中的建立儲存貯體。

將 Lambda 搭配基礎設施即程式碼 (IaC)

Lambda 提供多種方式來部署程式碼和建立函數。例如，您可以使用 Lambda 主控台或 AWS Command Line Interface (AWS CLI) 手動建立或更新 Lambda 函數。除了這些手動選項之外，AWS 還提供了許多解決方案，可使用基礎設施即程式碼 (IaC) 部署 Lambda 函數和無伺服器應用程式。使用 IaC，您可以使用程式碼佈建和維護 Lambda 函數和其他 AWS 資源，而不是使用手動程序和設定。

大多數情況下，Lambda 函數不會單獨執行。相反地，它們是具有其他資源 (例如資料庫、佇列和儲存體) 的無伺服器應用程式的一部分。使用 IaC，您可以自動化部署程序，以快速且重複地部署並更新涉及許多不同 AWS 資源的整個無伺服器應用程式。這種方法可加快您的開發週期，使組態管理更加輕鬆，並確保您的資源每次都以相同的方式部署。

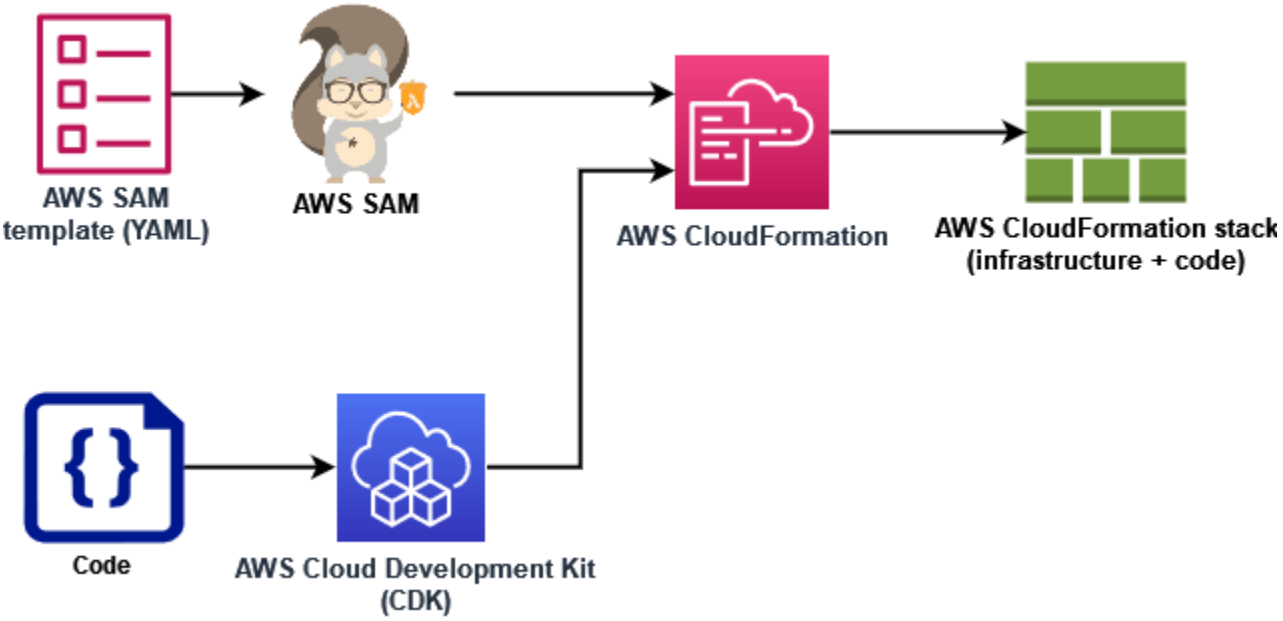
主題

用於 Lambda 的 IaC 工具

若要使用 IaC 部署 Lambda 函數和無伺服器應用程式，AWS 會提供許多不同的工具和服務。

AWS CloudFormation 是由 AWS 所提供，用於建立和設定雲端資源的第一個服務。您可以使用 AWS CloudFormation 建立文字範本來定義基礎設施和程式碼。隨著 AWS 導入更多的新服務和建立 AWS CloudFormation 範本的複雜性增加，我們推出了另外兩個工具。AWS SAM 是另一個用於定義無伺服器應用程式的範本型架構。AWS Cloud Development Kit (AWS CDK) 是使用許多常用程式設計語言中的程式碼建構模組來定義和佈建基礎設施的程式碼優先方法。

使用 AWS SAM 和 AWS CDK，AWS CloudFormation 可在幕後運作以建置和部署您的基礎設施。下圖說明這些工具之間的關係，圖表後面的段落說明它們的主要功能。



- AWS CloudFormation-使用描述您的AWS資源及其屬性的 YAML 或 JSON 範本來建立資源的模型和設定資源。 CloudFormation CloudFormation 以安全、可重複的方式佈建您的資源，讓您無需手動步驟即可經常建置基礎架構和應用程式。當您變更組態時， CloudFormation 決定更新堆疊所要執行的正確作業。 CloudFormation 甚至可以回滾更改。
- AWS Serverless Application Model (AWS SAM) - AWS SAM 是一種開放原始碼架構，用於定義無伺服器應用程式。AWS SAM 範本使用簡寫語法來定義函數、API、資料庫和事件來源映射，每個資源只需要幾行文字 (YAML)。在部署期間，AWS SAM 將 AWS SAM 語法轉換並擴充至 AWS CloudFormation 語法。因此，任何 CloudFormation 語法都可以添加到AWS SAM模板中。這提供了AWS SAM所有功能 CloudFormation，但配置行較少。
- AWS Cloud Development Kit (AWS CDK)-使用AWS CDK，您可以使用程式碼結構來定義基礎結構，並透過AWS CloudFormation佈建。 AWS CDK可讓您使用現有的 IDE TypeScript、測試工具和工作流程模式，使用 Python、Java、.NET 和 Go (在「開發人員預覽版」中) 建立應用程式基礎結構的模型。您可以獲得 AWS CloudFormation 的所有好處，包括可重複部署、輕鬆復原和漂移偵測。

AWS 還提供了一種稱為 AWS 應用程式編寫器、使用簡單的圖形介面開發 IaC 範本的服務。使用應用程式編寫器，您可以透過在視覺化畫布中拖曳、分組和連線 AWS 服務 來設計應用程式架構。然後，應用程式編寫器會從您的設計建立 AWS SAM 範本或 AWS CloudFormation 範本，供您用來部署應用程式。

在下面的 the section called “開始使用適用於 Lambda 的 IaC” 章節中，您可以使用應用程式編寫器，根據現有的 Lambda 函數為無伺服器應用程式開發範本。

開始使用適用於 Lambda 的 IaC

在本教學課程中，您可以透過從現有的 Lambda 函數建立 AWS SAM 範本，然後新增其他 AWS 資源，在應用程式編寫器中建置無伺服器應用程式，開始將 IaC 與 Lambda 搭配使用。

如果您想開始執行 AWS SAM 或 AWS CloudFormation 教學課程，以瞭解如何在不使用應用程式編寫器的情況下使用範本，您可以在本頁結尾的 the section called “後續步驟” 章節中找到其他資源的連結。

當您執行此教學課程時，您將學習到一些基本概念，例如如何在 AWS SAM 中指定 AWS 資源。您也將學習如何使用應用程式編寫器來建置可使用 AWS SAM 或 AWS CloudFormation 部署的無伺服器應用程式。

請執行下列步驟以完成本教學課程：

- 建立範例 Lambda 函數
- 使用 Lambda 主控台可檢視函數的 AWS SAM 範本
- 將函數的配置匯出到 AWS 應用程式編寫器 並根據函數的組態設計一個簡單的無伺服器應用程式
- 儲存可用來部署無伺服器應用程式的更新 AWS SAM 範本

在此 the section called “後續步驟” 區段中，您會找到可用來進一步瞭解 AWS SAM 和應用程式編寫器的資源。這些資源包括更進階教學課程的連結，會教導您如何使用 AWS SAM 部署無伺服器應用程式。

必要條件

在本教學課程中，您會使用應用程式編寫器的本機同步處理功能，將範本和程式碼檔案儲存到本機建置機器。要使用此功能，您需要一個支援檔案系統存取 API 的瀏覽器，該瀏覽器允許 Web 應用程式在本機檔案系統中讀取、寫入和儲存文件。我們建議使用 Google Chrome 或 Microsoft Edge。如需檔案系統存取 API 的詳細資訊，請參閱什麼是檔案系統存取 API？

建立 Lambda 函數

在此第一步驟中，將會建立 Lambda 函數，可用於完成本教學課程的其餘部分。為了簡化事情，您可以使用 Lambda 主控台，使用 Python 3.11 執行期來建立基本的「Hello world」函數。

若要使用主控台建立「Hello world」Lambda 函數

1. 開啟 Lambda 主控台。

2. 選擇 建立函式。
3. 保持選取從頭開始撰寫，然後在基本資訊之下的函數名稱中輸入 **LambdaIaCDemo**。
4. 針對執行期，選取 Python 3.11。
5. 選擇 建立函式。

檢視函數的 AWS SAM 範本

在您將函數組態匯出至應用程式編寫器之前，請使用 Lambda 主控台以 AWS SAM 範本的方式檢視函數目前的組態。按照本章節中的步驟操作，您將瞭解 AWS SAM 範本的剖析，以及如何定義 Lambda 函數等資源以開始指定無伺服器應用程式。

檢視函數的 AWS SAM 範本

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇您剛建立的函數 (LambdaIaCDemo)。
3. 在函數概觀窗格中，選擇範本。

   代替表示函數配置的圖表，您將看到一個函數的 AWS SAM 模板。範本看起來應該如下所示。

```
# This AWS SAM template has been generated from your function's
# configuration. If your function has one or more triggers, note
# that the AWS resources associated with these triggers aren't fully
# specified in this template and include placeholder values.Open this template
# in AWS Application Composer or your favorite IDE and modify
# it to specify a serverless application with other AWS resources.
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Specification template describing your function.
Resources:
  LambdaIaCDemo:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Description: ''
      MemorySize: 128
      Timeout: 3
      Handler: lambda_function.lambda_handler
      Runtime: python3.11
      Architectures:
        - x86_64
```

```
      EventInvokeConfig:
        MaximumEventAgeInSeconds: 21600
        MaximumRetryAttempts: 2
      EphemeralStorage:
        Size: 512
      RuntimeManagementConfig:
        UpdateRuntimeOn: Auto
      SnapStart:
        ApplyOn: None
      PackageType: Zip
      Policies:
        Statement:
          - Effect: Allow
            Action:
              - logs:CreateLogGroup
            Resource: arn:aws:logs:us-east-1:123456789012:*
          - Effect: Allow
            Action:
              - logs:CreateLogStream
              - logs:PutLogEvents
            Resource:
              - >-
                arn:aws:logs:us-east-1:123456789012:log-group:/aws/lambda/
LambdaIaCDemo:*
```

讓我們花一點時間看看函數的 YAML 範本，並了解一些重要概念。

範本以宣告 `Transform: AWS::Serverless-2016-10-31` 開始。此聲明是必需的，因為在幕後，AWS SAM 模板是透過 AWS CloudFormation 而部署。使用 `Transform` 陳述式將範本識別為 AWS SAM 範本檔案。

在 `Transform` 聲明之後伴隨的 `Resources` 部分。這是您要使用 AWS SAM 範本部署的 AWS 資源的定義位置。AWS SAM 範本可以包含 AWS SAM 資源和 AWS CloudFormation 資源的組合。這是因為在部署期間，AWS SAM 範本會展開為 AWS CloudFormation 範本，因此任何有效的 AWS CloudFormation 語法都可以新增至 AWS SAM 範本。

目前，範本 `Resources` 區段中只定義了一個資源，即您的 Lambda 函數 `LambdaIaCDemo`。若要將 Lambda 函數新增至 AWS SAM 範本，請使用 `AWS::Serverless::Function` 資源類型。Lambda 函數資源的 `Properties` 定義函數的執行期、函數處理常式和其他組態選項。此處也定義了函數原始碼的路徑，AWS SAM 應該用於部署該函數。若要進一步了解中的 Lambda 函數資源AWS SAM，請參閱AWS SAM開發人員指南AWS::Serverless::Function中的。

除了函數屬性和組態外，範本還會為您的函數指定 AWS Identity and Access Management (IAM) 政策。此政策授予您將日誌寫入 Amazon CloudWatch 日誌的函數權限。當您在 Lambda 主控台中建立函數時，Lambda 會自動將此政策附加至您的函數。若要進一步了解如何為AWS SAM範本中的函數指定 IAM 政策，請參閱AWS SAM開發人員指南AWS::Serverless::Function頁面上的`policies`屬性。

若要進一步瞭解 AWS SAM 範本的結構，請參閱AWS SAM 範本剖析。

使用 AWS 應用程式編寫器 來設計無伺服器應用程式

若要開始使用函數的 AWS SAM 範本做為起點來建立簡單的無伺服器應用程式，請將函數組態匯出至應用程式編寫器，然後啟動應用程式編寫器的本機同步處理模式。本機同步會自動將函數的程式碼和 AWS SAM 範本保存到本機構置機器，並在您在應用程式編寫器中添加其他 AWS 資源時保持已儲存的範本保持同步。

若要將函數匯出至應用程式編寫器

1. 在函數概觀窗格中，選擇匯出至應用程式編寫器。

   若要將函數的組態和程式碼匯出至應用程式編寫器，Lambda 會在您的帳戶中建立 Amazon S3 儲存貯體來暫時存放此資料。
2. 在對話方塊中，選擇確認並建立專案以接受此儲存貯體的預設名稱，並將函數的設定和程式碼匯出至應用程式編寫器。
3. (選擇性) 若要為 Lambda 建立的 Amazon S3 儲存貯體選擇其他名稱，請輸入新名稱，然後選擇確認並建立專案。Amazon S3 儲存貯體的名稱必須是全域唯一的，並遵循儲存貯體命名規則。

   選取確認並建立專案會開啟應用程式編寫器主控台。在畫布上，您將看到您 Lambda 函數。
4. 從選單下拉式清單中選擇啟用本機同步。
5. 在開啟的對話方塊中，選擇選取資料夾，然後選取本機建置機器上的資料夾。
6. 選擇啟用以啟用本機同步。

若要將您的函數匯出至應用程式編寫器，您需要有使用某些 API 動作的許可。如果您無法匯出函數，請見 the section called “所需的許可” 並確認您有所需的許可。

> **Note**
>
> 標準 Amazon S3 定價適用於 Lambda 在您將函數匯出至應用程式編寫器時所建立的儲存貯體。Lambda 放入儲存貯體的物件會在 10 天後自動刪除，但 Lambda 不會刪除儲存貯體本身。

若要避免額外費用新增至您的 AWS 帳戶，請在將函數匯出至應用程式編寫器之後，依照刪除儲存貯體中的指示執行。如需 Lambda 所建立 Amazon S3 儲存貯體的詳細資訊，請參閱 the section called “應用程式編寫器”。

在應用程式編寫器中設計無伺服器應用程式

啟用本機同步之後，您在應用程式編寫器中所做的變更會反映在儲存於本機建置機器上的 AWS SAM 範本中。您現在可以將其他 AWS 資源拖放到應用程式編寫器畫布上，以建置您的應用程式。在此範例中，您將 Amazon SQS 簡單佇列新增為 Lambda 函數的觸發程序，以及新增 DynamoDB 資料表供函數寫入資料。

1. 執行下列動作，將 Amazon SQS 觸發條件新增至您的 Lambda 函數：
   a. 在資源面板的搜尋欄位中，輸入 **SQS**。
   b. 將 SQS 佇列資源拖曳到畫布上，並將其放置在 Lambda 函數的左側。
   c. 選擇詳細資訊，然後為邏輯 ID 輸入 **LambdaIaCQueue**。
   d. 選擇儲存。
   e. 按一下 SQS 佇列卡上的訂閱連接埠，然後將它拖曳至 Lambda 函數卡上的左側連接埠，即可連接您的 Amazon SQS 和 Lambda 資源。兩個資源之間出現一條線表示連線成功。應用程式編寫器也會在畫布底部顯示訊息，指示出兩個資源已成功連線。
2. 執行下列動作，為您的 Lambda 函數新增 Amazon DynamoDB 資料表，以便將資料寫入：
   a. 在資源面板的搜尋欄位中，輸入 **DynamoDB**。
   b. 將 DynamoDB 資料表資源拖曳到畫布上，並將其放置在 Lambda 函數的右側。
   c. 選擇詳細資訊，然後為邏輯 ID 輸入 **LambdaIaCTable**。
   d. 選擇儲存。
   e. 按一下 Lambda 函數卡的右側連接埠，然後將其拖曳至 DynamoDB 卡上的左側連接埠，藉此將 DynamoDB 資料表連接至 Lambda 函數。

現在您已新增這些額外資源，讓我們來看看應用程式編寫器已建立的更新 AWS SAM 範本。

若要檢視更新的 AWS SAM 範本

- 在應用程式編寫器畫布上，選擇範本以從畫布檢視切換至範本檢視。

您的 AWS SAM 範本現在應該包含下列其他資源和屬性：

- 識別碼為 LambdaIaCQueue 的 Amazon SQS 佇列

```
LambdaIaCQueue:
    Type: AWS::SQS::Queue
    Properties:
      MessageRetentionPeriod: 345600
```

當您使用應用程式編寫器新增 Amazon SQS 佇列時，應用程式編寫器會設定 MessageRetentionPeriod 屬性。您也可以選取 SQS 佇列卡上的詳細資訊，然後核取或取消核取 Fifo 佇列來設定 FifoQueue 屬性。

若要設定佇列的其他屬性，您可以手動編輯範本以新增它們。若要進一步瞭解 AWS::SQS::Queue 資源及其可用的屬性，請參閱《AWS CloudFormation 使用者指南》中的AWS::SQS::Queue。

- Lambda 函數定義中的 Events 屬性，可將 Amazon SQS 佇列指定為函數的觸發程序

```
Events:
  LambdaIaCQueue:
    Type: SQS
    Properties:
      Queue: !GetAtt LambdaIaCQueue.Arn
      BatchSize: 1
```

此 Events 屬性由事件類型和一組依賴於類型的屬性組成。若要了解不同項目，AWS 服務您可以設定以觸發 Lambda 函數和可設定的屬性，請參閱開發AWS SAM人員指南EventSource中的。

- 具有識別碼 LambdaIaCTable 的 DynamoDB 資料表

```
LambdaIaCTable:
    Type: AWS::DynamoDB::Table
    Properties:
      AttributeDefinitions:
        - AttributeName: id
          AttributeType: S
      BillingMode: PAY_PER_REQUEST
      KeySchema:
        - AttributeName: id
          KeyType: HASH
      StreamSpecification:
        StreamViewType: NEW_AND_OLD_IMAGES
```

使用應用程式編寫器新增 DynamoDB 資料表時，您可以選擇 DynamoDB 資料表卡片上的「詳細資料」並編輯索引鍵值來設定資料表的索引鍵。應用程式編寫器也會設定許多其他屬性的預設值，包括 `BillingMode` 和 `StreamViewType`。

若要進一步瞭解這些屬性和您可以新增至 AWS SAM 範本的其他屬性，請參閱《AWS CloudFormation 使用者指南》中的 AWS::DynamoDB::Table。

- 一項新的 IAM 政策，可讓您在新增的 DynamoDB 資料表上執行 CRUD 操作的函數許可。

```
Policies:
...
  - DynamoDBCrudPolicy:
    TableName: !Ref LambdaIaCTable
```

完整的最終 AWS SAM 範本看起來應該如下所示。

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Specification template describing your function.
Resources:
  LambdaIaCDemo:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Description: ''
      MemorySize: 128
      Timeout: 3
      Handler: lambda_function.lambda_handler
      Runtime: python3.11
      Architectures:
        - x86_64
      EventInvokeConfig:
        MaximumEventAgeInSeconds: 21600
        MaximumRetryAttempts: 2
      EphemeralStorage:
        Size: 512
      RuntimeManagementConfig:
        UpdateRuntimeOn: Auto
      SnapStart:
        ApplyOn: None
      PackageType: Zip
```

```
      Policies:
        - Statement:
            - Effect: Allow
              Action:
                - logs:CreateLogGroup
              Resource: arn:aws:logs:us-east-1:594035263019:*
            - Effect: Allow
              Action:
                - logs:CreateLogStream
                - logs:PutLogEvents
              Resource:
                - arn:aws:logs:us-east-1:594035263019:log-group:/aws/lambda/
LambdaIaCDemo:*
        - DynamoDBCrudPolicy:
            TableName: !Ref LambdaIaCTable
      Events:
        LambdaIaCQueue:
          Type: SQS
          Properties:
            Queue: !GetAtt LambdaIaCQueue.Arn
            BatchSize: 1
      Environment:
        Variables:
          LAMBDAIACTABLE_TABLE_NAME: !Ref LambdaIaCTable
          LAMBDAIACTABLE_TABLE_ARN: !GetAtt LambdaIaCTable.Arn
  LambdaIaCQueue:
    Type: AWS::SQS::Queue
    Properties:
      MessageRetentionPeriod: 345600
  LambdaIaCTable:
    Type: AWS::DynamoDB::Table
    Properties:
      AttributeDefinitions:
        - AttributeName: id
          AttributeType: S
      BillingMode: PAY_PER_REQUEST
      KeySchema:
        - AttributeName: id
          KeyType: HASH
      StreamSpecification:
        StreamViewType: NEW_AND_OLD_IMAGES
```

使用 AWS SAM (選擇性) 部署您的無伺服器應用程式

如果您使用您剛在應用程式編寫器中建立的範本，想要用 AWS SAM 來部署無伺服器應用程式，您必須先安裝 AWS SAM CLI。若要執行此操作，請遵循安裝 AWS SAM CLI 中的說明。

在部署應用程式之前，您也需要更新應用程式編寫器與範本一起儲存的函數程式碼。目前，應用程式編寫器儲存的 `lambda_function.py` 檔案只包含 Lambda 在建立函數時提供的基本 'Hello world' 程式碼。

若要更新您的函數程式碼，請複製下列程式碼，並將其貼到儲存至本機建置機器的應用程式編寫器 `lambda_function.py` 檔案中。當您啟動本機同步模式時，您已指定應用程式編寫器要儲存此檔案的目錄。

此程式碼接受來自您在應用程式編寫器中建立之 Amazon SQS 佇列的訊息中的索引鍵值組。如果索引鍵和值都是字串，則程式碼會使用它們將項目寫入範本中定義的 DynamoDB 資料表。

更新 Python 函數程式碼

```
import boto3
import os
import json

# define the DynamoDB table that Lambda will connect to
tablename = os.environ['LAMBDAIACTABLE_TABLE_NAME']

# create the DynamoDB resource
dynamo = boto3.client('dynamodb')

def lambda_handler(event, context):
    # get the message out of the SQS event
    message = event['Records'][0]['body']
    data = json.loads(message)
    # write event data to DDB table
    if check_message_format(data):
        key = next(iter(data))
        value = data[key]
        dynamo.put_item(
            TableName=tablename,
            Item={
                'id': {'S': key},
                'Value': {'S': value}
            }
```

```
        )
    else:
        raise ValueError("Input data not in the correct format")

# check that the event object contains a single key value
# pair that can be written to the database
def check_message_format(message):
    if len(message) != 1:
        return False

    key, value = next(iter(message.items()))

    if not (isinstance(key, str) and isinstance(value, str)):
        return False

    else:
        return True
```

若要部署您的無伺服器應用程式

若要使用 AWS SAM CLI 部署您的應用程式，請執行下列步驟。若要讓您的函數正確建置和部署，Python 第 3.11 版本必須安裝在您的建置機器和 PATH。

1. 從應用程式編寫器儲存 `template.yaml` 和 `lambda_function.py` 檔案的目錄執行下列命令。

   ```
   sam build
   ```

   此命令會收集應用程式的建置成品，並將它們放置在適當的格式和位置以進行部署。
2. 若要部署應用程式並建立 AWS SAM 範本中指定的 Lambda、Amazon SQS 和 DynamoDB 資源，請執行下列命令。

   ```
   sam deploy --guided
   ```

   使用此 `--guided` 標記意味著 AWS SAM 將顯現提示，以引導您完成部署過程。對於此部署，請按 Enter 接受預設選項。

在部署程序期間，AWS SAM 會在您的 AWS 帳戶 中建立下列資源：

- 一個名為 `sam-app` 的 AWS CloudFormation 堆疊
- 名稱格式為 `sam-app-LambdaIaCDemo-`*`99VXPpYQVv1M`* 的 Lambda 函數

- 名稱格式為 `sam-app-LambdaIaCQueue-xL87VeKsGiIo` 的 Amazon SQS 佇列
- 名稱格式為 `sam-app-LambdaIaCTable-CN0S66C0VLNV` 的 DynamoDB 資料表

AWS SAM 還會建立必要的 IAM 角色和政策，讓 Lambda 函數可以讀取來自 Amazon SQS 佇列的訊息，並在 DynamoDB 資料表上執行 CRUD 操作。

若要進一步瞭解如何使用 AWS SAM 部署無伺服器應用程式，請參閱 the section called “後續步驟” 章節中的資源。

測試已部署的應用程式 (選擇性)

若要確認您的無伺服器應用程式是否正確部署，請將訊息傳送至包含索引鍵值組的 Amazon SQS 佇列，並檢查 Lambda 是否使用這些值將項目寫入 DynamoDB 資料表。

測試您的無伺服器應用程式

1. 開啟 Amazon SQS 主控台的佇列頁面，然後選取從範本所建立 AWS SAM 的佇列。名稱具有格式 `sam-app-LambdaIaCQueue-xL87VeKsGiIo`。
2. 選擇傳送和接收訊息，然後將以下 JSON 貼到傳送訊息區段裡的訊息內文中。

   ```
   {
       "myKey": "myValue"
   }
   ```

3. 選擇 傳送訊息 。

   將訊息傳送至佇列會導致 Lambda 透過範本 AWS SAM 中定義的事件來源映射調用您的函數。若要確認 Lambda 已如預期調用您的函數，請確認項目已新增至 DynamoDB 資料表中。

4. 開啟 DynamoDB 主控台的 資料表 頁面，然後選擇資料表。名稱具有格式 `sam-app-LambdaIaCTable-CN0S66C0VLNV`。
5. 選擇 探索資料表項目 。在 Items returned 窗格中，應該會看到一個包含 id `myKey` 和 數值 `myValue` 的項目。

後續步驟

若要進一步瞭解如何搭配 AWS SAM 和 AWS CloudFormation 使用應用程式編寫器，請先從搭配 AWS CloudFormation 和 AWS SAM 使用應用程式編寫開始。

對於使用 AWS SAM 部署在應用程式編寫器中設計的無伺服器應用程式的引導式教學課程，我們也建議您在 AWS 無伺服器模式研討會中執行 AWS 應用程式編寫器 教學課程。

AWS SAM 提供命令列介面 (CLI)，您可以將其與 AWS SAM 範本和支援的第三方整合搭配使用，以建置和執行無伺服器應用程式。使用 AWS SAM CLI，您可以建置和部署應用程式、執行本機測試和偵錯、設定 CI/CD 管道等。若要進一步瞭解如何使用 AWS SAM CLI，請參閱AWS Serverless Application Model開發人員指南中的 AWS SAM 入門。

若要瞭解如何使用 AWS CloudFormation 主控台以 AWS SAM 範本部署無伺服器應用程式，請從《AWS CloudFormation 使用者指南》中的使用 AWS CloudFormation 主控台開始。

Lambda 與應用程式編寫器整合的支援區域

下列 AWS 區域 項目支援與應用程式編寫器整合的 Lambda：

- 美國東部 (維吉尼亞北部)
- 美國東部 (俄亥俄)
- 美國西部 (加利佛尼亞北部)
- 美國西部 (奧勒岡)
- 非洲 (開普敦)
- 亞太區域 (香港)
- 亞太區域 (海德拉巴)
- 亞太區域 (雅加達)
- 亞太區域 (墨爾本)
- 亞太區域 (孟買)
- 亞太區域 (大阪)
- 亞太區域 (首爾)
- 亞太區域 (新加坡)
- 亞太區域 (雪梨)
- 亞太區域 (東京)
- 加拿大 (中部)
- 歐洲 (法蘭克福)
- 歐洲 (蘇黎世)
- 歐洲 (愛爾蘭)

- 歐洲 (倫敦)
- 歐洲 (斯德哥爾摩)
- 中東 (阿拉伯聯合大公國)

具有 VPC 的私有網路

Amazon Virtual Private Cloud（Amazon VPC）是 AWS 雲中的一個虛擬網絡，專用於您的 AWS 帳戶。您可以使用 Amazon VPC 來為資料庫、快取執行個體或內部服務等資源建立私有網路。如需 Amazon VPC; 的詳細資訊，請參閱《什麼是 Amazon VPC？》

Lambda 函數一律會在 Lambda 服務擁有的 VPC 內執行。Lambda 會將網路存取和安全性規則套用至此 VPC，Lambda 即會自動維護和監控 VPC。如果您的 Lambda 函數需要存取帳戶 VPC 中的資源，請設定函數來存取 VPC。Lambda 提供名為 Hyperplane ENI 的受管資源，您的 Lambda 函數可使用此資源來從 Lambda VPC 連接到您帳戶 VPC 中的 ENI (彈性網路介面)。

使用 VPC 或 Hyperplane ENI 均無需負擔額外費用。部分 VPC 元件需付費，例如 NAT 閘道。如需詳細資訊，請參閱 Amazon VPC 定價。

主題

VPC 網路元素

Amazon VPC 網路包含以下網路元素：

- 彈性網路介面 - 彈性網路介面是代表虛擬網路卡之 VPC 中的邏輯網路元件。
- 子網路 - 您 VPC 中的 IP 地址範圍。您可以將 AWS 資源新增至指定的子網路。針對必須連線至網際網路的資源使用公有子網路，並針對不會連線至網際網路的資源使用私有子網路。
- 安全群組 — 使用安全群組來控制對每個子網路中 AWS 資源的存取。
- 存取控制清單 (ACL - 使用網路 ACL 在子網路中提供額外的安全性。預設的子網路 ACL 會允許所有傳出和傳入流量。

- 路由表 — 包含一組 AWS 用於引導 VPC 網路流量的路由。您可以明確地將子網與特定路由表建立關聯。根據預設，子網路會與主路由表關聯。
- 路由 - 路由表中每個路由都會指定 IP 地址範圍，以及 Lambda 傳送該範圍之流量的目的地。路由也會指定目標，即傳送流量所經的閘道、網路介面或連線。
- NAT 閘道 — 一種 AWS 網路位址轉譯 (NAT) 服務，可控制從私有 VPC 私有子網路到網際網路的存取。
- VPC 端點 — 您可以使用 Amazon VPC 端點建立與託管於其中的服務的私有連線 AWS，而無需透過網際網路或透過 NAT 裝置、VPN 連線或 AWS Direct Connect 連線存取。如需詳細資訊，請參閱AWS PrivateLink 和 VPC 端點。

> **Tip**
>
> 若要將 Lambda 函數設定為存取 VPC 和子網路，可以使用 Lambda 主控台或 API。
> 請參閱中的〈〉一`VpcConfig`節CreateFunction以配置您的功能。將 Lambda 函數連接到您的 Amazon VPC AWS 帳戶如需詳細步驟，請參閱。

如需 Amazon VPC 聯網定義的詳細資訊，請參閱《Amazon VPC 開發人員指南》中的Amazon VPC 運作方式與Amazon VPC 常見問答集。

將 Lambda 函數連接到您的 VPC

Lambda 函數一律會在 Lambda 服務擁有的 VPC 內執行。根據預設，Lambda 函數不會連接到您帳戶中的 VPC。當您將函數連線至您帳戶的 VPC 時，除非您的 VPC 提供存取權，否則該函數無法存取網際網路。

Lambda 會使用 Hyperplane ENI 存取 VPC 中的資源。Hyperplane ENI 會使用 VPC 對 VPC NAT (V2N) 來提供從 Lambda VPC 到您的帳戶 VPC 之 NAT 功能。V2N 會提供從 Lambda VPC 到您的帳戶 VPC 的連線，但不提供反向連線。

當您建立 Lambda 函數 (或更新其 VPC 設定) 時，Lambda 會為函數 VPC 組態中的每個子網路配置 Hyperplane ENI。如果函數共用相同的子網路和安全群組，則多個 Lambda 函數可以共用網路介面。

若要連線到其他 AWS 服務，您可以使用 VPC 端點在 VPC 和支援 AWS 的服務之間進行私人通訊。另一種方法是使用 NAT 閘道將輸出流量路由到另一個 AWS 服務。

若要讓函數存取網際網路，請將傳出流量路由到公有子網路的 NAT 閘道。NAT 閘道具有公有 IP 地址，可透過 VPC 的網際網路閘道連線至網際網路。如需相關資訊，請參閱為虛擬私人電腦連線的 Lambda 函數啟用網際網路。

共用子網路

VPC 共用可讓多個 AWS 帳戶在共用、集中管理的虛擬私有雲 (VPC) 中建立其應用程式資源，例如 Amazon EC2 執行個體和 Lambda 函數。在此模型中，擁有 VPC (擁有者) 的帳戶與屬於同一個組織的其他帳戶 (參與者) 共用一或多個子網路。 AWS

若要存取私有資源，請將您的函數連線到 VPC 中的私有共用子網路。子網路擁有者必須與您共用子網路，才能將函數連線到該子網路。子網路擁有者也可以稍後取消共用子網路，從而移除連線。如需有關如何在共用子網路中共用、取消共用和管理 VPC 資源的詳細資訊，請參閱《Amazon VPC 指南》中的如何與其他帳戶共用 VPC。

Lambda Hyperplane ENI

Hyperplane ENI 是 Lambda 服務建立和管理的受管網路資源。Lambda VPC 中的多個執行環境可以使用 Hyperplane ENI 安全地存取您帳戶中 VPC 內的資源。Hyperplane ENI 會提供從 Lambda VPC 到您的帳戶 VPC 之 NAT 功能。

對於每個子網路，Lambda 會為每個唯一的安全群組集建立網路介面。帳戶中共用相同子網路和安全群組組合的函數會使用相同的網路介面。即使安全群組組態未以其他方式要求追蹤，也會自動追蹤透過 Hyperplane layer 所建立的連線。來自 VPC 的輸入封包 (與已建立的連線不對應) 會丟棄在 Hyperplane layer。如需詳細資訊，請參閱 Amazon EC2 使用者指南中的安全群組連線追蹤。

由於帳戶中的函數會共用 ENI 資源，因此 ENI 生命週期比其他 Lambda 資源更複雜。以下部分說明 ENI 生命週期。

ENI 生命週期

- 建立 ENI
- 管理 ENI
- 刪除 ENI

建立 ENI

Lambda 可能會為新建立的已啟用 VPC 的函數建立 Hyperplane ENI 資源，或為現有函數的 VPC 組態變更建立 Hyperplane ENI 資源。當 Lambda 建立所需資源時，函數仍處於擱置狀態。當 Hyperplane

ENI 準備就緒時，函數會轉換為作用中狀態，且 ENI 會變成可供使用。Lambda 可能需要數分鐘來建立 Hyperplane ENI。

對於新建立的已啟用 VPC 的函數，在函數狀態轉換為作用中之前，在函數上操作的任何調用或其他 API 動作都會失敗。

對於現有函數的 VPC 組態變更，任何函數調用都會繼續使用與舊子網路和安全群組組態關聯的 Hyperplane ENI，直到函數狀態轉換為作用中為止。

如果 Lambda 函數閒置 30 天，Lambda 會回收未使用的超平面 ENI，並將函數狀態設定為閒置。下一次調用會導致 Lambda 重新啟用閒置函數。調用失敗，而且函數會進入擱置狀態，直到 Lambda 完成 Hyperplane ENI 的建立或配置為止。

如需函數狀態的詳細資訊，請參閱Lambda 函數狀態。

管理 ENI

Lambda 會使用您函數的執行角色中的許可來建立並管理網路介面。在帳戶中為已啟用 VPC 的函數定義唯一子網路加上安全群組組合時，Lambda 會建立 Hyperplane ENI。Lambda 會為使用相同子網路和安全群組組合的帳戶中其他已啟用 VPC 的函數重複使用 Hyperplane ENI。

目前沒有任何關於可使用相同 Hyperplane ENI 的 Lambda 函數的數量配額。不過，每個 Hyperplane ENI 均支援多達 65,000 個連線/連接埠。如果連線數量超過 65,000，Lambda 會建立新的Hyperplane ENI 來提供額外的連線。

當您更新函數組態以存取不同的 VPC 時，Lambda 會終止與先前 VPC 中 Hyperplane ENI 的連線。更新連至新 VPC 之連線的程序，可能需要幾分鐘的時間。在此期間，函數的調用會繼續使用先前的 VPC。更新完成後，新的調用會開始使用新 VPC 中的 Hyperplane ENI。此時，Lambda 函數不再連接到先前的 VPC。

刪除 ENI

當您更新函數以移除其 VPC 組態時，Lambda 需要長達 20 分鐘才能刪除連接的 Hyperplane ENI。Lambda 只會在沒有其他函數 (或已發佈的函數版本) 使用該 Hyperplane ENI 時刪除 ENI。

Lambda 需要函數執行角色中的許可才能刪除 Hyperplane ENI。如果您在 Lambda 刪除 Hyperplane ENI 之前刪除執行角色，Lambda 將無法刪除 Hyperplane ENI。您可以手動執行刪除作業。

Lambda 不會刪除帳戶中的函數或函數版本正在使用的網路介面。您可以使用 Lambda ENI 搜尋工具來識別正在使用 Hyperplane ENI 的函數或函數版本。對於您不再需要的任何函數或函數版本，您可以移除 VPC 組態來讓 Lambda 刪除 Hyperplane ENI。

連線

Lambda 支援兩種連線類型：TCP (傳輸控制通訊協定) 和 UDP (使用者資料包通訊協定)。

當您建立 VPC 時，Lambda 會自動建立 DHCP 選項集，並將其與 VPC 建立關聯。您可以為 VPC 設定自己的 DHCP 選項。如需更多詳細資訊，請參閱《Amazon VPC DHCP 選項》。

Amazon 為您的 VPC 提供 DNS 伺服器 (Amazon Route 53 Resolver)。如需詳細資訊，請參閱《VPC 的 DNS 支援》。

IPv6 支援

Lambda 支援 Lambda 公有雙堆疊端點的傳入連線，以及透過 IPv6 連至雙堆疊 VPC 子網路的傳出連線。

傳入

若要透過 IPv6 調用您的函數，請使用 Lambda 的公有雙堆疊端點。雙堆疊端點可同時支援 IPv4 和 IPv6。Lambda 雙堆疊端點使用下列語法：

```
protocol://lambda.us-east-1.api.aws
```

您也可以使用 Lambda 函數 URL 透過 IPv6 來調用函數。函數 URL 端點的格式如下：

```
https://url-id.lambda-url.us-east-1.on.aws
```

傳出

您的函數可透過 IPv6 連線至雙堆疊 VPC 子網路中的資源。此選項預設為關閉。若要允許傳出 IPv6 流量，請使用主控台或具有 create-function 或 update-function-configuration 命令的 `--vpc-config Ipv6AllowedForDualStack=true` 選項。

> **Note**
>
> 若要允許 VPC 中的傳出 IPv6 流量，連線到該函數的所有子網路都必須是雙堆疊子網路。Lambda 不支援 VPC 中僅限 IPv6 子網路的輸出 IPv6 連線、未連線至 VPC 之函數的輸出 IPv6 連線，或使用 VPC 端點的輸入 IPv6 連線 (AWS PrivateLink)。

您可以更新函數程式碼來透過 IPv6 明確連線至子網路資源。下列 Python 範例開啟了通訊端，並連接到 IPv6 伺服器。

Example — 連線至 IPv6 伺服器

```
def connect_to_server(event, context):
    server_address = event['host']
    server_port = event['port']
    message = event['message']
    run_connect_to_server(server_address, server_port, message)

def run_connect_to_server(server_address, server_port, message):
    sock = socket.socket(socket.AF_INET6, socket.SOCK_STREAM, 0)
    try:
        # Send data
        sock.connect((server_address, int(server_port), 0, 0))
        sock.sendall(message.encode())
        BUFF_SIZE = 4096
        data = b''
        while True:
            segment = sock.recv(BUFF_SIZE)
            data += segment
            # Either 0 or end of data
            if len(segment) < BUFF_SIZE:
                break
        return data
    finally:
        sock.close()
```

安全

AWS 提供安全群組和網路 ACL，以提高 VPC 的安全性。安全群組控制資源的傳入與傳出流量，網路 ACL 則是控制子網的傳入與傳出流量。安全群組可為大多數子網路提供足夠的存取控制。如果您想讓 VPC 多一層安全，可以使用網路 ACL。如需詳細資訊，請參閱《Amazon VPC 中的網路間流量隱私權》。每個您建立的子網路都會自動與 VPC 的預設網路 ACL 建立關聯。您可以變更關聯，也可以變更預設網路 ACL 的內容。

如需一般安全性最佳實務，請參閱《VPC 安全最佳實務》。如需如何使用 IAM 來管理 Lambda API 和資源存取的詳細資訊，請參閱 AWS Lambda 許可。

您可以針對 VPC 設定使用 Lambda 特定條件金鑰，為您的 Lambda 函數提供額外的許可控制。如需 VPC 條件索引鍵的詳細資訊，請參閱使用 IAM 條件索引鍵進行 VPC 設定。

> **Note**
>
> 可以從共有網際網路或 AWS PrivateLink 端點調用 Lambda 函數。只能透過公有網際網路存取函數 URL。雖然 Lambda 函數確實支援 AWS PrivateLink，但函式網址則不支援。

可觀測性

您可以使用 VPC 流量日誌來擷取 VPC 中傳入和傳出網路介面之 IP 流量資訊。您可以將流程日誌資料發佈到 Amazon CloudWatch 日誌或 Amazon S3。建立流量日誌之後，您可以在選擇的目的地中擷取及檢視其資料。

附註：將函數附加到 VPC 時， CloudWatch 記錄訊息不會使用 VPC 路由。Lambda 會使用日誌的一般路由來傳送。

設定 Lambda 函數的指令集架構

Lambda 函數的指令集架構會決定 Lambda 用來執行函數的電腦處理器類型。Lambda 提供了指令集架構的選擇：

- arm64 — 64 位元 ARM 架構，適用於 AWS 重力 2 處理器。
- x86_64 - 64 位元 x86 架構，適用於 x86 處理器。

> **Note**
>
> arm64 架構在大 AWS 區域多數情況下都可以使用。如需詳細資訊，請參閱 AWS Lambda 定價。在記憶體價格表中，選擇「機臂價格」標籤，然後開啟「地區」下拉式清單，查看哪些 AWS 區域 支援使用 Lambda 的 arm64。
> 如需如何使用 arm64 架構建立函數的範例，請參閱由AWS 重力 on2 處理器提供支援的AWS Lambda 函數。

主題

使用 arm64 架構的優點

與在 x86_64 架構上執行的同等函數相比，使用 arm64 架構 (AWS Graviton2 處理器) 的 Lambda 函數可以實現更好的價格和效能。考慮將 arm64 用於運算密集的應用程式，例如高效能運算、視訊編碼和模擬工作負載。

Graviton2 CPU 使用 Neoverse N1 核心，並支援 Armv8.2 (包括 CRC 和加密延伸模組)，加上數個其他架構延伸模組。

Graviton2 透過每個 vCPU 提供更大的 L2 快取來減少記憶體讀取時間，進而改善 Web 和行動後端、微服務和資料處理系統的延遲效能。Graviton2 也提供改善的加密效能，並支援指令集，其可改善 CPU 型機器學習推論的延遲。

如需有關 AWS 重力 on2 的詳細資訊，請參閱AWS 重力子處理器。

遷移到 arm64 架構的要求

當您選取要遷移至 arm64 架構的 Lambda 函數時，為了確保順利遷移，請確定您的函數符合下列需求：

- 該函數目前使用 Lambda Amazon Linux 2 執行期。
- 部署套件只包含開放原始碼元件和您控制的原始碼，以便您可以針對遷移進行任何必要的更新。
- 如果函數程式碼包含第三方相依性，則每個程式庫或套件都會提供 arm64 版本。

函數程式碼與 arm64 架構的相容性

您的 Lambda 函數程式碼必須與函數的指令集架構相容。將函數遷移到 arm64 架構之前，請注意下列有關目前函數程式碼的幾點：

- 如果您使用內嵌的程式碼編輯器新增函數程式碼，則您的程式碼可能無需修改即可在任一架構上執行。
- 如果上傳了您的函數程式碼，則您必須上傳與目標架構相容的新程式碼。
- 如果您的函數使用層，則必須檢查每一層以確定其與新架構相容。如果層不相容，請編輯函數，將目前的層版本取代為相容的層版本。
- 如果您的函數使用 Lambda 延伸模組，則必須檢查每個延伸模組，以確定其與新架構相容。
- 如果您的函數使用容器映像部署套件類型，則必須建立與函數架構相容的新容器映像。

如何遷移到 arm64 架構

若要將 Lambda 函數遷移至 arm64 架構，建議您遵循下列步驟：

1. 為您的應用程式或工作負載建置相依性清單。常見相依性包括：
   - 函數使用的所有程式庫和套件。
   - 用來建置、部署和測試函數的工具，例如編譯器、測試套件、持續交付和持續整合 (CI/CD) 管道、佈建工具，以及指令碼。
   - Lambda 延伸模組和第三方工具，用來在生產環境中監控函數。
2. 對於每個相依性，請檢查版本，然後檢查 arm64 版本是否可用。
3. 建置環境來遷移應用程式。

4. 引導應用程式。
5. 測試和除錯應用程式。
6. 測試 arm64 函數的效能。將效能與 x86_64 版本進行比較。
7. 更新您的基礎設施管道以支援 arm64 Lambda 函數。
8. 將部署暫存至生產環境。

   例如，使用別名路由組態來分割函數 x86 和 arm64 版本之間的流量，並比較效能和延遲。

如需有 AWS 關如何為 arm64 架構建立程式碼環境的詳細資訊，包括 Java、Go、.NET 和 Python 的語言特定資訊，請參閱重力光儲存庫入門。 GitHub

設定指令集架構

您可以使用 Lambda 主控台、 AWS 開發套件 AWS Command Line Interface (AWS CLI) 或 AWS CloudFormation，為新的和現有的 Lambda 函數設定指令集架構。依照這些步驟操作，在主控台中為現有的 Lambda 函數變更指令集架構。

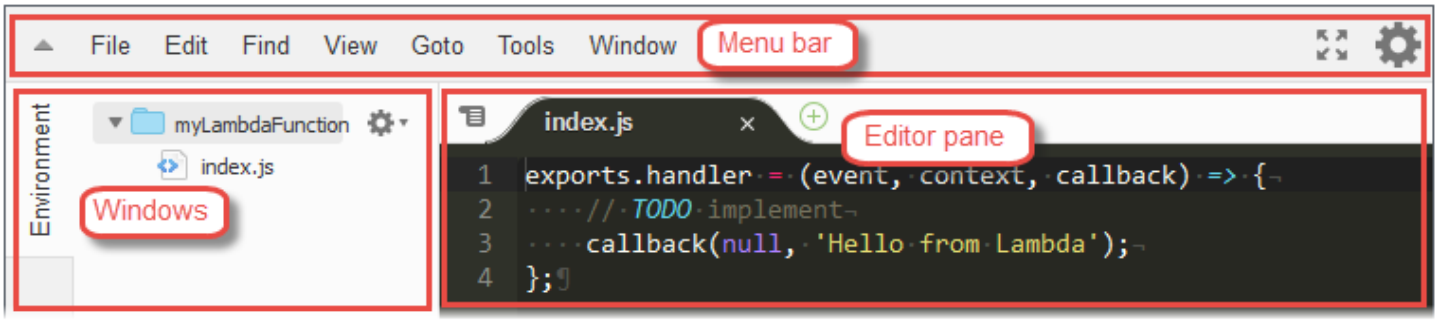
1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇函數名稱以為其設定指令集架構。
3. 在程式碼主索引標籤上的執行期設定區段選擇編輯。
4. 在 架構 選擇要用於函數的指令集架構。
5. 選擇 儲存 。

> **Note**
>
> 所有 Amazon Linux 2 執行期皆 支援 x86_64 和 ARM CPU 架構。
> 未使用 Amazon Linux 2 (例如 Go 1.x) 的執行期不支援 arm64 架構。若要搭配 Go 1.x 使用 arm64 架構，您可以在 provided.al2 執行期中執行您的函數。如需詳細資訊，請參閱 .zip 套件 和 容器映像 的部署指示。

使用 Lambda 主控台編輯器編輯代碼

可使用 Lambda 主控台中的程式碼編輯器來撰寫、測試 Lambda 函數程式碼並檢視其執行結果。程式碼編輯器支援不需要編譯的語言，例如 Node.js 和 Python。程式碼編輯器僅支援 .zip 封存檔部署套件，且部署套件的大小必須小於 3 MB。

程式碼編輯器包含選單列、視窗和編輯器窗格。



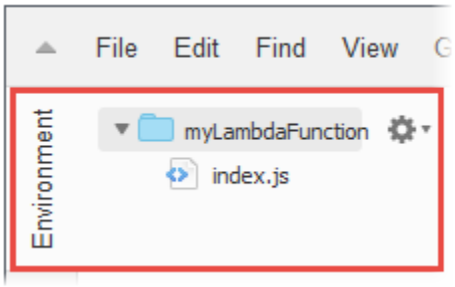
如需命令用途清單，請參閱《AWS Cloud9 使用者指南》中的選單列命令參考。請注意，該參考中所列的一些命令，無法在程式碼編輯器中使用。

主題

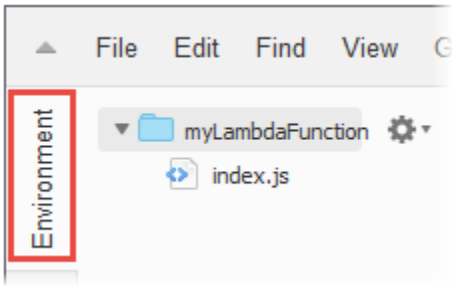
- 處理檔案和資料夾
- 使用程式碼
- 以全螢幕模式運作
- 使用偏好設定

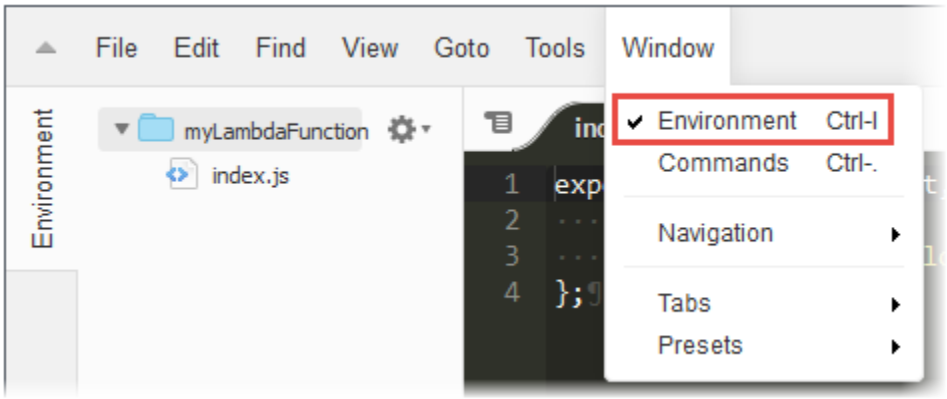
處理檔案和資料夾

您可以使用程式碼編輯器內的 Environment (環境) 視窗來建立、開啟和管理您的函式適用的檔案。



若要顯示或隱藏「環境」視窗，請選擇 Environment (環境) 按鈕。如果 Environment (環境) 按鈕未顯現，請選擇選單列上的 Window, Environment (視窗，環境)。



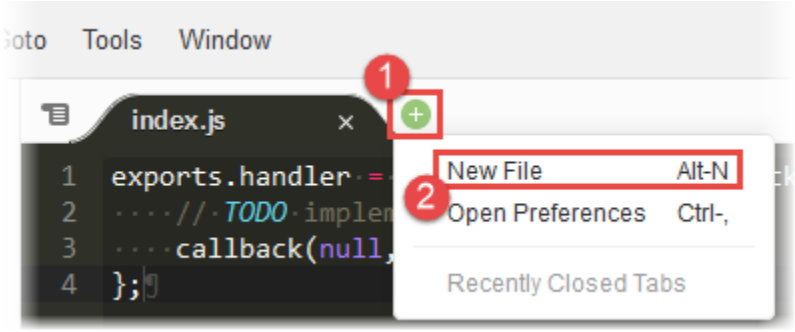


若要開啟單一檔案，並在編輯器窗格中顯示其內容，請在 Environment (環境) 視窗中按兩下檔案。

若要開啟多支檔案，並在編輯器窗格中顯示其內容，請在 Environment (環境) 視窗中選擇檔案。以滑鼠右鍵按一下選取項目，然後選擇 Open (開啟)。

若要建立新檔案，請執行以下其中一項：

- 在 Environment (環境) 視窗中，以右鍵按一下您要新檔案移往的資料夾，然後選擇 New File (開新檔案)。輸入檔案名稱和副檔名，然後按下 Enter 鍵。
- 選擇選單列上的 File, New File (檔案，開新檔案)。準備要儲存檔案時，從選單列選擇 File, Save (檔案，儲存) 或 File, Save As (檔案，另存新檔)。然後，利用出現的 Save As (另存新檔) 對話方塊為檔案命名，然後選擇檔案的儲存位置。
- 在編輯器窗格的索引標籤按鈕列中，選擇 + 按鈕，然後選擇 New File (開新檔案)。準備要儲存檔案時，從選單列選擇 File, Save (檔案，儲存) 或 File, Save As (檔案，另存新檔)。然後，利用出現的 Save As (另存新檔) 對話方塊為檔案命名，然後選擇檔案的儲存位置。



若要建立新資料夾，以右鍵按一下 Environment (環境) 視窗中您要新資料夾前往的資料夾，然後選擇 New Folder (新資料夾)。輸入資料夾的名稱，然後按下 Enter 鍵。

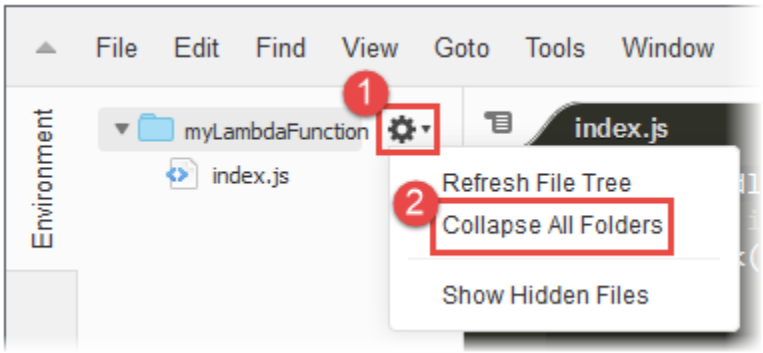
若要儲存檔案，讓檔案在編輯器窗格中保持開啟狀態，其內容仍然顯示著，此時從選單列選擇 File, Save (檔案，儲存)。

若要為檔案或資料夾重新命名，以右鍵按一下 Environment (環境) 視窗中的檔案或資料夾。輸入取代名稱，然後按下 Enter。

若要刪除檔案或資料夾，在 Environment (環境) 視窗中選擇檔案或資料夾。以滑鼠右鍵按一下選取項目，然後選擇 Delete (刪除)。然後，選擇 Yes (是) (用於單一選取項目) 或 Yes to All (全部皆是) 確認刪除。

若要剪下、複製、貼上或重複檔案或資料夾，請在 Environment (環境) 視窗中選擇檔案或資料夾。以滑鼠右鍵按一下選取項目，然後分別選擇 Cut (剪下)、Copy (複製)、Paste (貼上) 或 Duplicate (重複)。

若要摺疊資料夾，請在 Environment (環境) 視窗中選擇齒輪圖示，然後選擇 Collapse All Folders (摺疊全部資料夾)。



若要顯示或隱藏檔案，請在 Environment (環境) 視窗中選擇齒輪圖示，然後選擇 Show Hidden Files (顯示隱藏的檔案)。

若要查看為函數設定的環境變數，請執行下列動作：

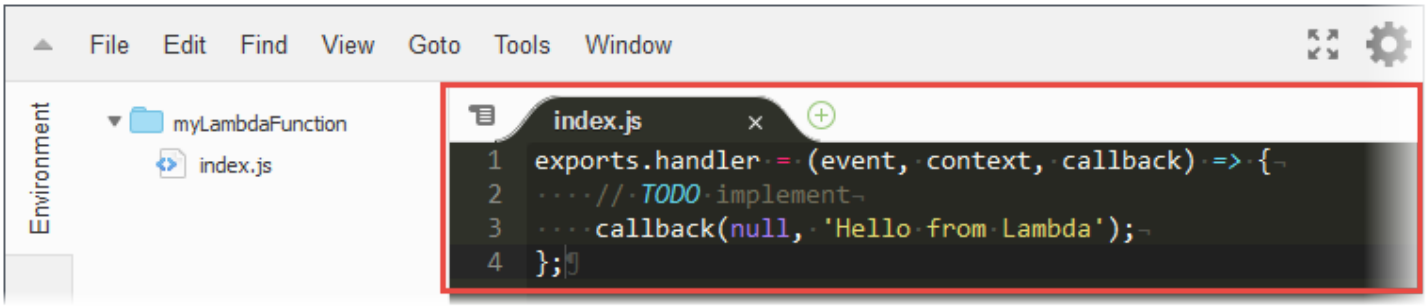
1. 選擇 程式碼 標籤。
2. 選擇環境變數索引標籤。
3. 選擇工具、顯示環境變數。

在主控台程式碼編輯器中列出環境變數時，會保持加密狀態。如果在傳輸過程中啟用加密協助程式進行加密，則這些設定會維持不變。如需詳細資訊，請參閱 保護 Lambda 環境變數。

環境變數清單為唯讀狀態，且只能在 Lambda 主控台中使用。下載函數的 .zip 封存檔時，不會包含此檔案，且您無法透過上傳此檔案來新增環境變數。

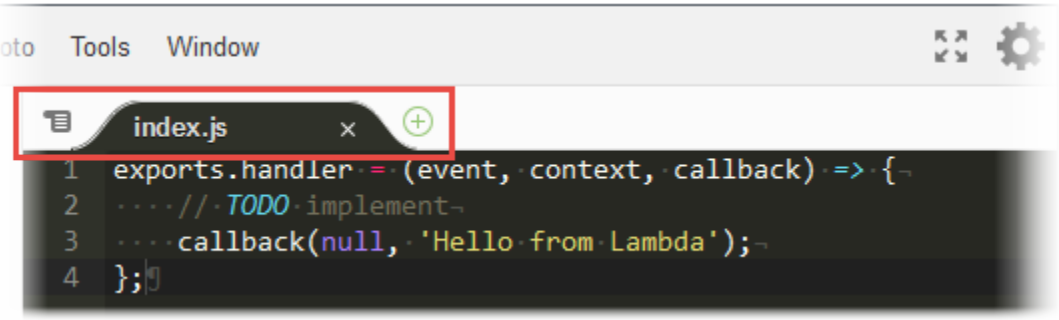
使用程式碼

使用程式碼編輯器中的編輯器窗格來視窗和撰寫程式碼。



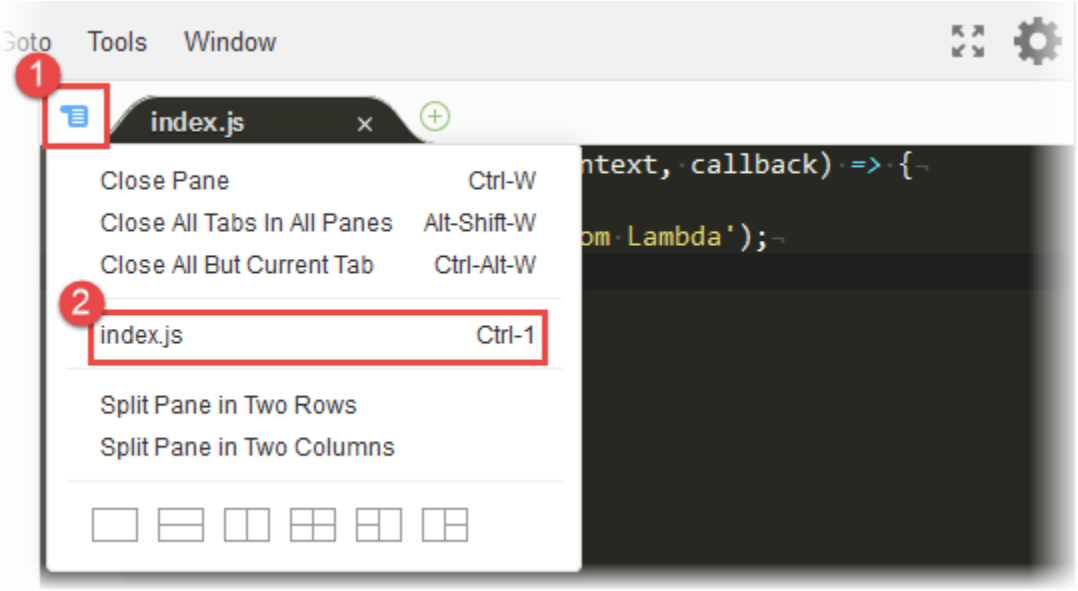
使用索引標籤按鈕

使用索引標籤按鈕列來選取、檢視和建立檔案。



若要顯示開啟的檔案內容，請執行以下其中一項：

- 選擇檔案的索引標籤。
- 在索引標籤按鈕列中選擇下拉式選單按鈕，然後選擇檔案的名稱。



若要關閉開啟的檔案，請執行以下其中一項：

- 選擇檔案索引標籤中的 X 圖示。
- 選擇檔案的索引標籤。在索引標籤按鈕列中選擇下拉式選單按鈕，然後選擇 Close Pane (關閉窗格)。

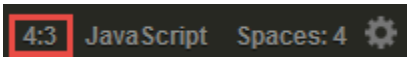
若要關閉多支開啟的檔案，請在索引標籤按鈕列選擇下拉式選單，然後依需要選擇 Close All Tabs in All Panes (關閉所有窗格中的所有索引標籤) 或 Close All But Current Tab (關閉目前索引標籤以外的所有索引標籤)。

若要建立新檔案，請在索引標籤按鈕列中選擇 + 按鈕，然後選擇 New File (開新檔案)。準備要儲存檔案時，從選單列選擇 File, Save (檔案，儲存) 或 File, Save As (檔案，另存新檔)。然後，利用出現的 Save As (另存新檔) 對話方塊為檔案命名，然後選擇檔案的儲存位置。

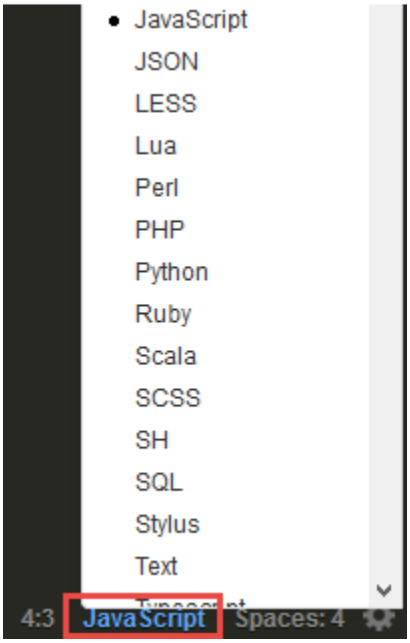
使用狀態列

使用狀態列以快速移至作用中檔案的某行，並且變更程式碼的顯示方式。

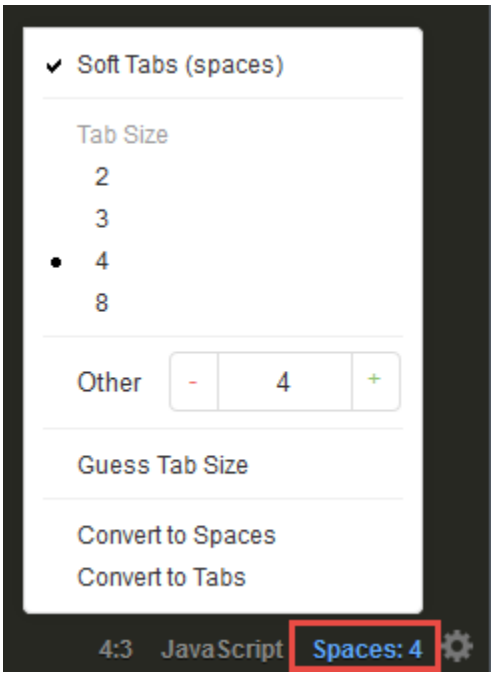
若要快速移至作用中檔案的某行，請選擇行選取器，輸入要前往的行號，然後按下 Enter 鍵。



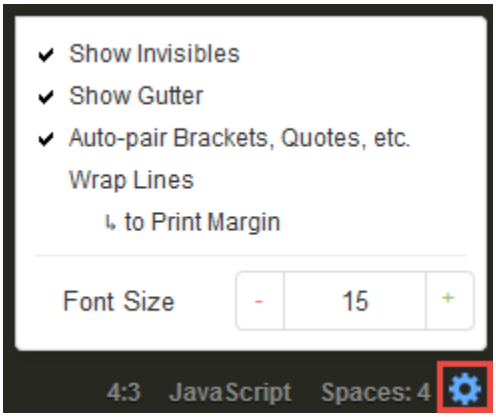
若要變更作用中檔案的程式碼色彩配置，請選擇程式碼色彩配置選取器，然後選擇新的程式碼色彩配置。



若要變更作用中檔案是否使用軟索引標籤或空格、索引標籤大小，或是否轉換成空格或索引標籤，請選擇空格及索引標籤選取器，然後選擇新的設定值。



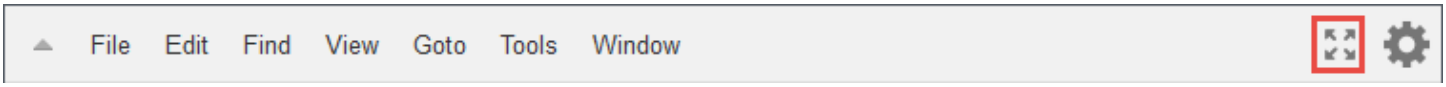
若要變更所有的檔案是否顯示或隱藏不可見的字元或裝訂邊、自動配對括弧或引號、自動換行或字型大小，請選擇齒輪圖示，然後選擇新的設定值。



以全螢幕模式運作

您可以展開程式碼編輯器以取得更多空間來處理您的程式碼。

若要展開程式碼編輯器至網頁瀏覽器邊緣，請選擇選單列上的 Toggle fullscreen (切換全螢幕) 按鈕。



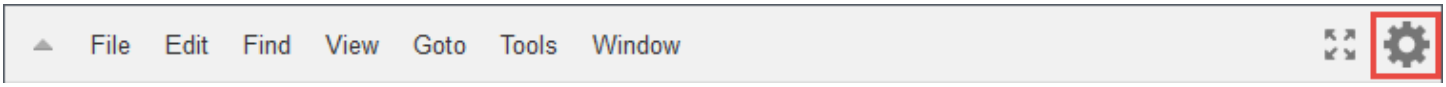
若要縮小程式碼編輯器至其原始大小，請再次選擇 Toggle fullscreen (切換全螢幕) 按鈕。

在全螢幕模式下，選單列上會顯示額外的選項：Save (儲存) 和 Test (測試)。選擇 Save (儲存) 以儲存函式程式碼。選擇 Test (測試) 或 Configure Events (設定事件) 可讓您建立或編輯函式的測試事件。

使用偏好設定

您可以變更各種程式碼編輯器設定，例如顯示哪些編碼提示和警告、程式碼摺疊行為、程式碼自動完成行為，以及其他更多功能。

若要變更程式碼編輯器設定，請選擇選單列上的 Preferences (偏好設定) 齒輪圖示。



如需各項設定的用途清單，請參閱《AWS Cloud9 使用者指南》中的下列參考資料。

- 您可以變更的專案設定
- 您所做的使用者設定變更

請注意，那些參考資料中所列的一些設定值，無法在程式碼編輯器中使用。

其他 Lambda 功能

Lambda 提供管理主控台和 API，用於管理和叫用函數。它提供支援標準功能集的執行時間，讓您可以根據自己的需求輕鬆切換語言和架構。除了函數之外，您也可以建立版本、別名、Layer 和自訂執行時間。

進階功能

- 擴展
- 並行控制項
- 函數 URL
- 非同步調用
- 事件來源映射
- 目的地
- 函數藍圖
- 測試和部署工具
- 應用程式範本

擴展

Lambda 管理執行程式碼的基礎設施，並自動擴展以回應傳入的請求。當函數被叫用的速度比函數單一執行個體處理事件的速度還要快時，Lambda 會透過執行其他執行個體進行擴展。當流量減小時，非作用中的執行個體會予以凍結或停止。您只需要依照函數初始化或處理事件所使用的時間來支付費用。

如需詳細資訊，請參閱 了解 Lambda 展函數。

並行控制項

使用並行設定以確保您的生產應用程式具有高可用性和高回應性。

若要防止函數使用太多並行，並為函數保留一部分帳戶的可用並行，請使用「預留並行」。保留的並行會將可用的並行集區分割成子集。具有預留並行的函數僅使用來自其專用子集的並行。

若要讓函數能在不受到延遲影響的情況下擴展，請使用「佈建並行」。對於初始化耗時很長時間的函數，或所有叫用都要求極低延遲的函數，並行可讓您預先初始化函數執行個體，並使其隨時保持執行狀態。Lambda 與 Application Auto Scaling 整合，以根據使用率支援佈建並行的自動擴展。

如需詳細資訊，請參閱 為函數配置保留並發。

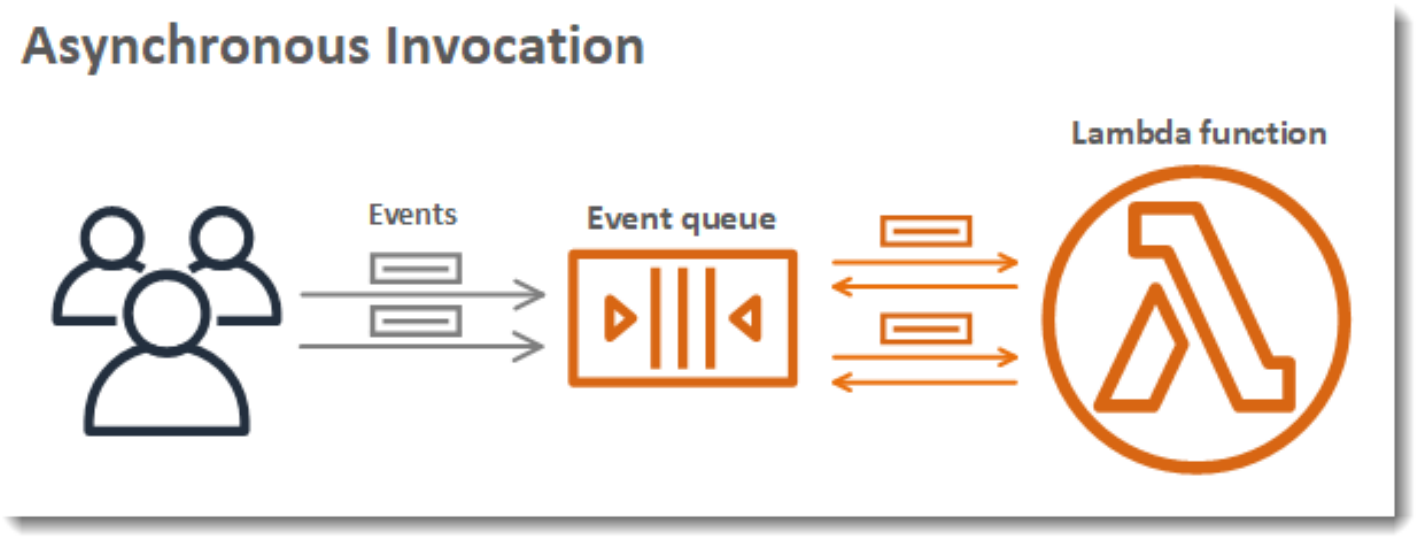
函數 URL

Lambda 透過函數 URL 提供內建的 HTTP(S) 端點支援。藉由函數 URL，您可以為 Lambda 函數指派專用的 HTTP 端點。設定函數 URL 後，您就可以透過 Web 瀏覽器、curl、Postman 或任何 HTTP 用戶端，使用該 URL 呼叫您的函數。

您可以為現有的函數新增函數 URL，或使用函數 URL 建立新函數。如需詳細資訊，請參閱 呼叫 Lambda 函數 URL。

非同步調用

當您調用函式時，您可以選擇以同步或非同步方式進行調用。使用同步調用，您會等待函式處理事件並傳回回應。使用非同步叫用，Lambda 會將事件排入佇列以進行處理，並立即傳回回應。

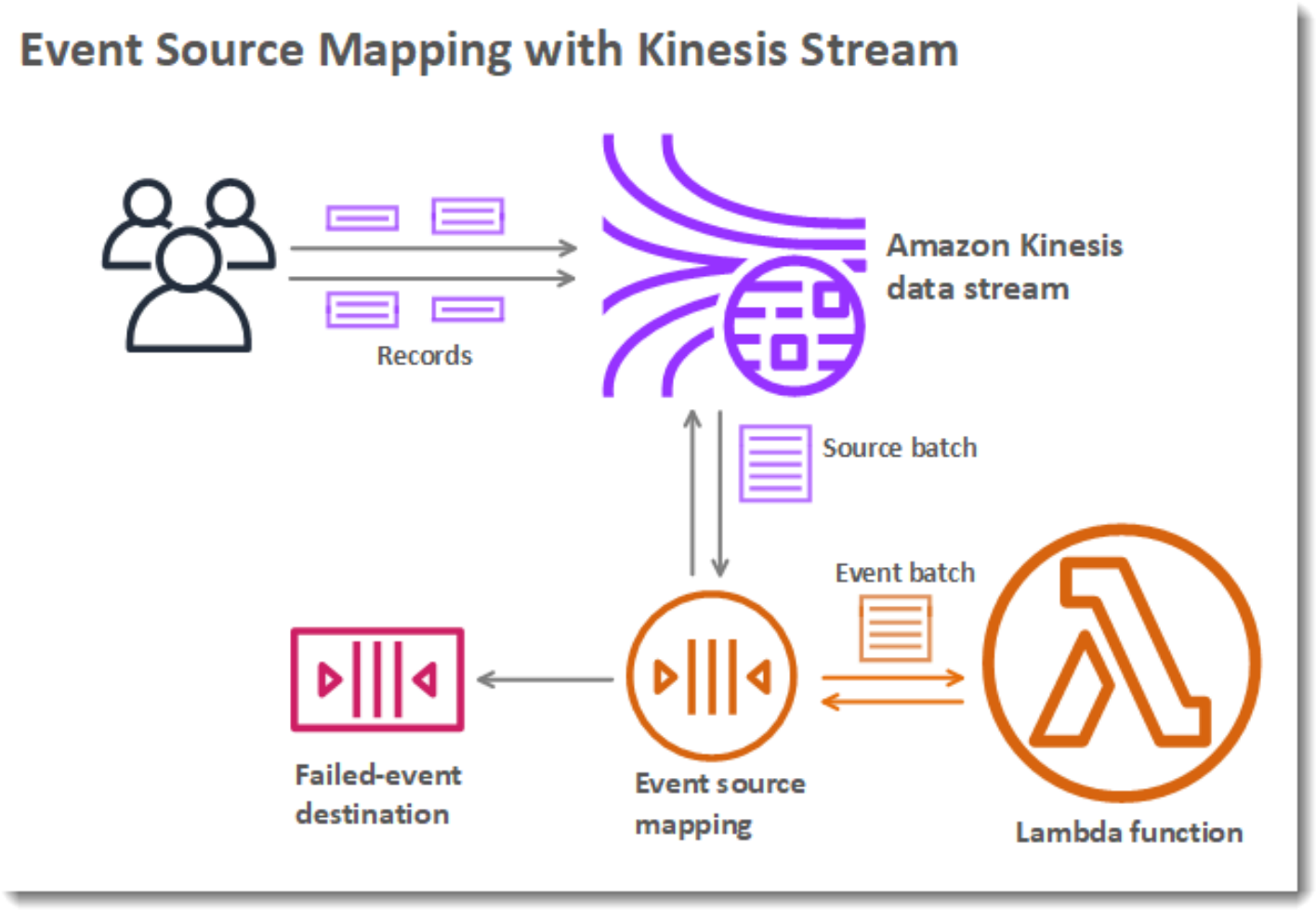


對於非同步叫用，如果函數傳回錯誤或被調節，則 Lambda 會處理重試。若要自訂此行為，您可以在函數、版本或別名上設定錯誤處理設定。您也可以設定 Lambda，將處理失敗的事件傳送至無效字母佇列，或將任何叫用的記錄傳送至目的地。

如需詳細資訊，請參閱 非同步調用。

事件來源映射

若要處理串流或佇列中的項目，您可以建立事件來源映射。事件來源映射是 Lambda 中的一種資源，它可讀取 Amazon Simple Queue Service (Amazon SQS) 佇列、Amazon Kinesis 串流或 Amazon DynamoDB 串流中的項目，並將它們分批傳送到您的函數。您的函式處理的每個事件可以包含數百個或數千個項目。

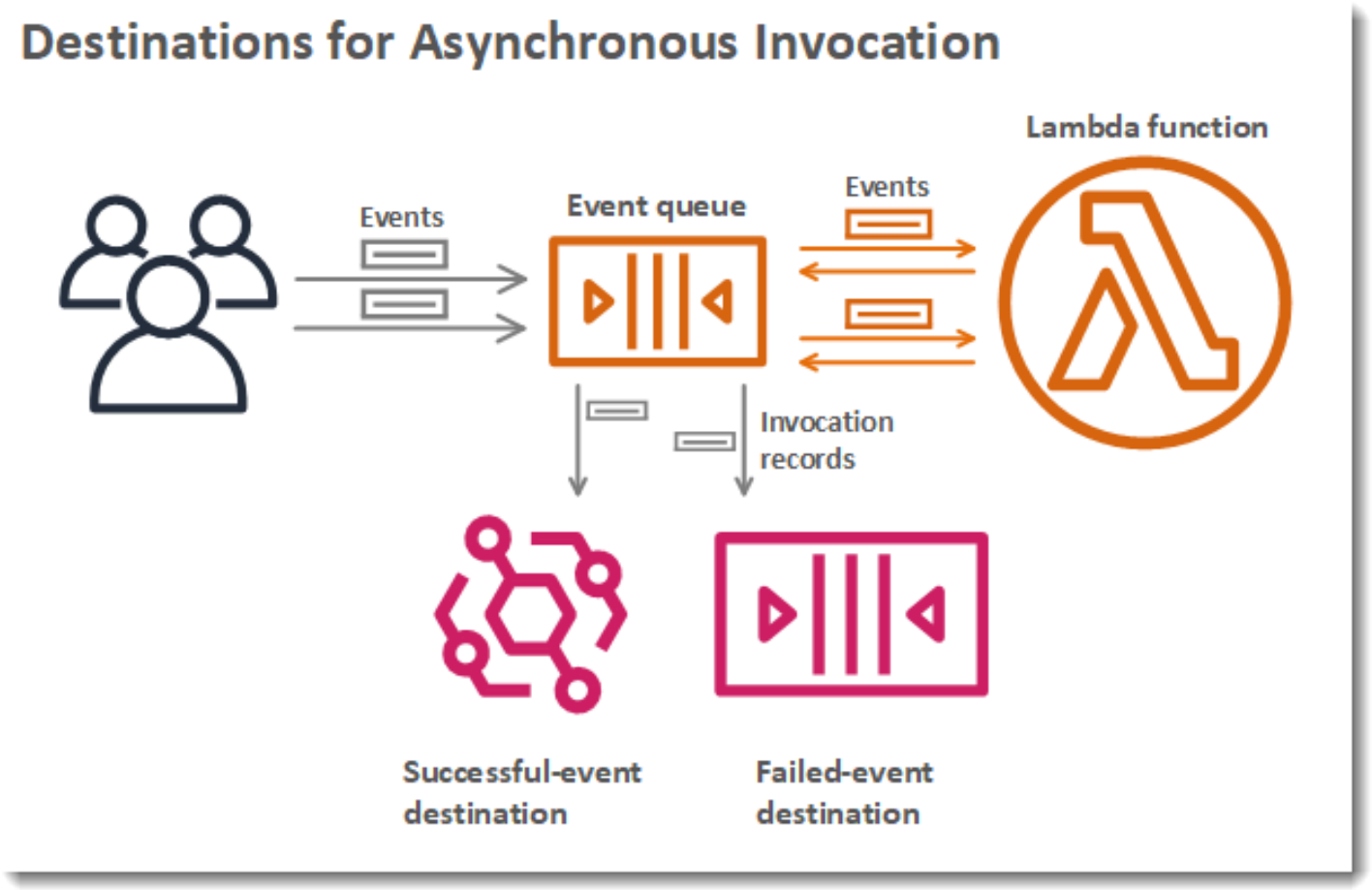


事件來源映射會維護未處理項目的本機佇列，並在函數傳回錯誤或進行節流處理時，處理重試。您可以設定事件來源映射，以自訂批次行為和錯誤處理，或將處理失敗的項目記錄傳送至目的地。

如需詳細資訊，請參閱 Lambda 如何處理串流和以佇列為基礎的事件來源的記錄。

目的地

目的地是接收函數調用記錄的 AWS 資源。對於非同步叫用，您可以設定 Lambda 以將呼叫記錄傳送到佇列、主題、函數或事件匯流排。您可以為成功叫用和處理失敗的事件設定不同的目的地。叫用記錄包含事件、函數回應以及記錄傳送原因的詳細資訊。



對於從串流中讀取的事件來源映射，您可以設定 Lambda，將處理失敗的批次記錄傳送至佇列或主題。事件來源映射的失敗記錄包含關於批次的中繼資料，並指向串流中的項目。

如需詳細資訊，請參閱 設定非同步調用的目的地，以及 AWS Lambda 搭配 Amazon DynamoDB 使用和 Lambda 如何處理來自 Amazon Kinesis Data Streams 的記錄 的錯誤處理章節。

函數藍圖

在 Lambda 主控台中建立函數時，可以選擇從頭開始、使用藍圖，或是使用容器映像。藍圖提供範例程式碼，顯示如何將 Lambda 與 AWS 服務或熱門第三方應用程式搭配使用。藍圖包含適用於 Node.js 和 Python 執行時間的範本程式碼和函數組態集。

依據 Amazon 軟體授權提供藍圖以供使用。它們只能在 Lambda 主控台中使用。

測試和部署工具

Lambda 支援按原樣部署程式碼，或作為容器映像部署。您可以使用 AWS 服務和常用的社群工具，例如 Docker 命令列介面 (CLI) 來編寫、建置和部署 Lambda 函數。若要設定 Docker CLI，請參閱

Docker Docs 網站上的取得 Docker。如需搭配使用 Docker 的簡介 AWS，請參閱 Amazon 彈性容器登錄使用者指南 AWS CLI中的使用開始使用 Amazon ECR。

AWS CLI 與 AWS SAM CLI 為命令列工具，可管理 Lambda 應用程式堆疊。除了使用 AWS CloudFormation API 管理應用程式堆疊的命令外，還 AWS CLI 支援更高層級的命令，可簡化工作，例如上傳部署套件和更新範本。 AWS SAM CLI 提供其他功能，包括驗證範本、在本機測試，以及與 CI/CD 系統整合。

- 安裝 AWS SAM CLI 的安裝
- 測試和偵錯無伺服器應用程式 AWS SAM
- 使用 CI/CD 系統部署無伺服器應用程式 AWS SAM

應用程式範本

您可以使用 Lambda 主控台建立具有持續交付管道的應用程式。Lambda 主控台中的應用程式範本包括一或多個函數的程式碼、定義函數和支援 AWS 資源的應用程式範本，以及定義 AWS CodePipeline 管道的基礎設施範本。管道具有建置和部署階段，其會在您每次推送變更至包含的 Git 儲存庫時執行。

應用程式範本會依據 MIT No Attribution 授權提供。它們只能在 Lambda 主控台中使用。

如需更多詳細資訊，請參閱 在 AWS Lambda 主控台中管理應用程式。

了解如何建置無伺服器解決方案

> **Tip**
>
> 若要了解如何建置無伺服器解決方案，請參閱無伺服器開發人員指南。

Lambda 執行期

Lambda 透過使用執行期支援多種語言。執行期會提供特定於語言的環境，其可轉傳調用事件、內容資訊以及 Lambda 與函數之間的回應。您可以使用由 Lambda 提供的執行期或是自行建置。

每個主要的程式設計語言版本都有獨立的執行階段，且具有唯一的執行階段識別符，例如 `nodejs20.x` 或 `python3.12`。若要設定函數以使用新的主要語言版本，您需變更執行期識別符。由於 AWS Lambda 無法保證主要版本之間的向後兼容性，因此這是客戶驅動的操作。

針對 定義為容器映像的函數，您可以在建立容器映像時選擇執行期和 Linux 發行版。若要變更執行期，請建立新的容器映像。

若要為部署套件使用 .zip 封存檔，您可以在建立函數時選擇執行期。若要變更執行期，您可以更新函數的組態。執行期與其中一個 Amazon Linux 發行版配對。基礎執行環境會提供其他程式庫以及環境變數，讓您可以從函數程式碼中存取。

Lambda 在執行環境中調用您的函數。執行環境提供安全且隔離的執行期環境，它會管理執行您的函數所需的資源。Lambda 會從之前的調用 (若有) 中重新使用執行環境，或者它會建立一個新的執行環境。

若要在 Lambda 中使用其他語言，例如 Go 或 Rust，請使用僅限作業系統的執行期。Lambda 執行環境提供了執行期介面，用於取得調用事件及傳送回應。您可以透過實作自訂執行期和您的函數程式碼，或在一個層中部署其他語言。

支援的執行期

下表列出受支援的 Lambda 執行期和預計的棄用日期。在執行期被棄用後，您依然能在限定時間內建立與更新函數。如需詳細資訊，請參閱 the section called “棄用後的運行時使用”。該表格提供目前預測的執行期棄用日期。這些日期提供用於規劃目的，且可能發生變更。

支援的執行期

名稱	識別符	作業系統	取代日期	封鎖函數建立	封鎖函數更新
Node.js 20	`nodejs20.x`	Amazon Linux 2023			
Node.js 18	`nodejs18.x`	Amazon Linux 2			

名稱	識別符	作業系統	取代日期	封鎖函數建立	封鎖函數更新
Node.js 16	nodejs16.x	Amazon Linux 2	2024 年 6 月 12 日	2025年2月28日	2025年3月31日
Python 3.12	python3.12	Amazon Linux 2023			
Python 3.11	python3.11	Amazon Linux 2			
Python 3.10	python3.10	Amazon Linux 2			
Python 3.9	python3.9	Amazon Linux 2			
Python 3.8	python3.8	Amazon Linux 2	2024 年 10 月 14 日	2025年2月28日	2025年3月31日
Java 21	java21	Amazon Linux 2023			
Java 17	java17	Amazon Linux 2			
Java 11	java11	Amazon Linux 2			
Java 8	java8.al2	Amazon Linux 2			
。淨值 8	dotnet8	Amazon Linux 2023			
.NET 6	dotnet6	Amazon Linux 2	2024 年 11 月 12 日	2025年2月28日	2025年3月31日
紅寶石	ruby3.3	Amazon Linux 2023			

名稱	識別符	作業系統	取代日期	封鎖函數建立	封鎖函數更新
Ruby 3.2	`ruby3.2`	Amazon Linux 2			
僅限作業系統的執行期	`provided.al2023`	Amazon Linux 2023			
僅限作業系統的執行期	`provided.al2`	Amazon Linux 2			

> **Note**
>
> 對於新區域，Lambda 將不支援設定為在未來 6 個月內棄用的執行階段。

Lambda 透過修補程式和次要版本發布的支援，讓受管執行期和其對應的容器基礎映像檔保持在最新狀態。如需詳細資訊，請參閱 Lambda 執行期更新。

棄用 Go 1.x 執行期後，Lambda 將繼續支援 Go 程式設計語言。如需詳細資訊，請參閱AWS 運算部落格上的 Amazon Linux 2 將 AWS Lambda 函數從 Go1.x 執行階段移轉至自訂執行階段。

所有支援的 Lambda 執行階段都支援 x86_64 和 arm64 架構。

新執行期版本

Lambda 僅在發行版本到達語言發布週期的長期支援 (LTS) 階段時提供新語言版本的受管理執行期。例如，對於 Node.js 發布週期，即當發行版本達到 Active LTS 階段時。

在發行版本達到長期支援階段以前，它將停留於開發中，並且仍可能有重大變更。Lambda 依預設會自動套用執行期更新，因此對執行期版本的重大變更可能使您的函數無法如預期執行。

Lambda 不會為未排定 LTS 發行版本的語言版本提供受管理執行期。

下列清單顯示即將到來的 Lambda 執行期目標啟動月份。這些日期僅為象徵性，並可能有所變更。

- Python 3.13 - 2024 年 11 月
- Node.js 22 - 2024 年 11 月

執行期淘汰政策

Lambda 執行期 的 .zip 封存檔是以需要維護和安全更新的作業系統、程式設計語言和軟體程式庫組合為基礎建置。Lambda 的標準棄用政策是在執行階段的任何主要元件到達社群長期支援 (LTS) 結束，且不再提供安全性更新時，淘汰執行階段。大多數情況下，這是語言執行期，雖然在某些情況下，執行期可以被棄用，因為操作系統 (OS) 到達 LTS 的結束。

在執行階段遭到淘汰之後， AWS 可能不再將安全性修補程式或更新套用至該執行階段，而且使用該執行階段的功能也不再符合技術支援的資格。此類已淘汰的執行階段會以「原樣」提供，不提供任何保證，且可能包含錯誤、錯誤、瑕疵或其他弱點。

若要進一步了解管理執行階段升級和淘汰，請參閱 AWS Compute 部落格上的下列章節和管理 AWS Lambda 執行階段升級。

> **Important**
>
> Lambda 偶爾會在執行期支援的語言版本的支援日期結束後，在有限期間內延遲棄用 Lambda 執行期。在此期間，Lambda 僅將安全性修補程式套用至執行期作業系統。Lambda 不會在到達支援結束日期之後，將安全性修補程式套用到程式設計語言執行期。

Node.js 16 的執行期棄用

為了回應客戶的意見 AWS ，將 Node.js 16 執行階段的淘汰延遲到社群 LTS 結束後的 9 個月。Node.js 16 執行期將在「支援的執行期」資料表中提供的日期棄用。如先前注意事項所述，在 2023 年 9 月 11 日 LTS 結束與棄用日期之間，Lambda 只會將作業系統修補程式套用至執行期。在此期間，將不會套用語言執行期的安全性修補程式。

延遲棄用 Node.js 16 可讓使用此執行期的客戶有機會將其函數直接遷移至 Node.js 20，跳過 Node.js 18。

共同責任模式

Lambda 負責策劃和發佈所有受支援的受管執行階段和容器基礎映像的安全性更新。根據預設，Lambda 會自動將這些更新套用至使用受管理執行階段的函數。若預設的自動執行時間更新設定已變更，請參閱執行時期管理控制共用職責模型。對於使用容器映像部署的功能，您負責從最新的基本映像重建函數的容器映像，並重新部署容器映像。

取代執行階段時，Lambda 對更新受管理執行階段和容器基礎映像的責任會停止。您必須負責升級函數，以使用受支援的執行階段或基本影像。

在所有情況下，您都必須負責將更新套用至函數程式碼，包括其相依性。下表彙總了您在共同責任模型下的職責。

运行时生命周期	拉姆達的責任	您的責任
受支援的受管理	提供包含安全性修補程式和其他更新的定期執行階段 依預設，自動套用執行階段更新 (the section called “執行階段管理控制”如需非預設行為，請參閱)。	更新您的函數程式碼，包括相依性，以解決任何安全性弱點。
支持的容器映像	透過安全性修補程式和其他更新，為容器基礎映像提供定期更新。	更新您的函數程式碼，包括相依性，以解決任何安全性弱點。 使用最新的基本映像定期重新建置和重新部署容器映像檔。
託管運行時接近棄用	在執行階段淘汰之前，透過文件 AWS Health Dashboard、電子郵件和 Trusted Advisor. 執行階段更新的責任在棄用時結束。	監控 Lambda 文件 AWS Health Dashboard、電子郵件或執 Trusted Advisor 行階段淘汰資訊。 在上一個執行階段被淘汰之前，將函數升級到支援的執行階段
容器映像接近棄用	棄用通知不適用於使用容器映像的函數。 容器基礎映像更新的責任在棄用時結束。	在先前的映像被棄用之前，請注意棄用排程和將函數升級到支持的基本映像。

棄用後的運行時使用

在執行階段遭到淘汰之後， AWS 可能不再將安全性修補程式或更新套用至該執行階段，而且使用該執行階段的功能也不再符合技術支援的資格。此類已淘汰的執行階段會以「原樣」提供，不提供任何保證，且可能包含錯誤、錯誤、瑕疵或其他弱點。使用已取代執行階段的函數也可能會遇到效能降低或其他問題 (例如憑證到期)，這些問題可能會導致它們無法正常運作。

在執行期棄用後至少 30 天，您仍可使用該執行期建立新的 Lambda 函數。從棄用後的 30 天開始，Lambda 開始封鎖建立新的函數。

在執行階段被淘汰後至少 60 天內，您仍然可以更新現有函數的函數代碼和配置。從淘汰後的 60 天開始，Lambda 開始封鎖現有函數程式碼和設定的更新。

> **Note**
>
> 對於某些運行時， AWS 在棄用後將 block-function-create 和 block-function-update 日期延遲到通常的 30 和 60 天以上。 AWS 因應客戶的意見而作出此變更，讓您有更多時間升級功能。請參閱中的表格the section called “支援的執行期”，the section called “已取代的執行階段”以查看執行階段的日期。

在執行階段被淘汰之後，您可以無限期地更新函數以使用較新的受支援執行階段。在生產環境中套用執行階段變更之前，您應該先測試您的函數是否可以與新的執行階段搭配使用，因為一旦 60 天的期間過去，您將無法還原至已停用的執行階段。我們建議您使用函數版本和別名來啟用復原的安全部署。

請注意，您可以繼續建立與更新函數的確切時長並非固定。每次棄用和不同的期限可能會有所不同 AWS 區域。此頁面第一區段中的「受支援執行期」表格提供封鎖函數建立和更新的名義日期。在此表格提供的日期前，Lambda 不會開始封鎖函數的建立或更新。

執行期被棄用後，您可以繼續無限期調用您的函數。不過， AWS 強烈建議您將函數移轉至支援的執行階段，以便您的功能繼續收到安全性修補程式，並符合獲得技術支援的資格。

接收執行期棄用通知

當執行階段接近淘汰日期時，如果您 AWS 帳戶 使用該執行階段中有任何函數，Lambda 會傳送電子郵件警示給您。通知也會顯示在 AWS Health Dashboard 和中 AWS Trusted Advisor。

- 接收電子郵件通知：

執行期被棄用前至少 180 天，Lambda 會向您傳送電子郵件提醒。此電子郵件將列出所有使用執行期之函數的 $LATEST 版本。若要查看受影響函數版本的完整清單，請使用 Trusted Advisor 或參閱the section called “列出使用已取代執行階段的函數”。

Lambda 會傳送電子郵件通知給您 AWS 帳戶的主要帳戶聯絡人。如需有關檢視或更新帳戶中電子郵件地址的資訊，請參閱《AWS 一般參考》中的更新聯絡資訊。

- 透過以下方式接收通知 AWS Health Dashboard：

  AWS Health Dashboard 會在執行階段停用前至少 180 天顯示通知。通知顯示在其他通知下方的您的帳戶運作狀態頁面上。通知的受影響資源索引標籤會列出所有使用執行期之函數的 $LATEST 版本。

  > **Note**
  >
  > 若要查看受影響函數版本的完整 up-to-date 清單，請使用 Trusted Advisor 或參閱the section called “列出使用已取代執行階段的函數”。

  AWS Health Dashboard 通知會在受影響的執行階段停用後 90 天過期。

- 使用 AWS Trusted Advisor

  Trusted Advisor 在執行階段停用前 180 天顯示通知。通知顯示在安全性頁面上。受影響函數的清單會顯示在使用已棄用執行期的AWS Lambda 函數的下方。此函數清單同時顯示 $LATEST 和已發佈版本，並且會自動更新以反映您的函數的最新狀態。

  您可以從 Trusted Advisor Trusted Advisor 主控台的 「偏好設定」 頁面開啟每週電子郵件通知。

列出使用已取代執行階段的函數

除了使用 Trusted Advisor 查看受計劃運行時棄用影響的實時函數列表之外，您還可以使用 AWS Command Line Interface （ AWS CLI ）或其中一個 AWS SDK 列出使用特定運行時的所有函數版本。

若要使用產生此清單 AWS CLI，請執行下列命令。取代`RUNTIME_IDENTIFIER`為已取代的執行階段名稱，然後選擇您自己的執行階段名稱 AWS 區域。如僅需列出 $LATEST 函數版本，請省略命令中的`--function-version ALL`。

```
aws lambda list-functions --function-version ALL --region us-east-1 --output text --
query "Functions[?Runtime=='RUNTIME_IDENTIFIER'].FunctionArn"
```

> **Tip**
>
> 範例命令會列出特定us-east-1區域中的功能。 AWS 帳戶 您必須針對您的帳戶具有功能的每個地區重複此指令，以及您的每個功能 AWS 帳戶。

若要進一步了解如何使用 AWS SDK 列出使用ListFunctions動作的函數，請參閱 SDK 說明文件，瞭解您偏好的程式設計語言。您也可以使用其中一個 AWS SDK，使用「DescribeLog串流」和「統計資料API」動作來收集有關最常呼叫和 most-recently-invoked 函數的GetMetric統計資料。

您也可以使用 [進 AWS Config 階查詢] 功能，列出使用受影響執行階段的所有函數。此查詢僅返回函數 $LATEST 版本，但您可以 AWS 帳戶 使用單個命令將查詢彙總到所有區域和多個區域的列表函數。要了解更多信息，請參閱AWS Config 開發人員指南中的查詢 AWS Auto Scaling 資源的當前配置狀態。

已取代的執行階段

下列執行階段已達到終止支援：

已取代的執行階段

名稱	識別符	作業系統	取代日期	封鎖函數建立	封鎖函數更新
.NET 7 (僅限容器)	dotnet7	Amazon Linux 2	2024 年 5 月 14 日		
Java 8	java8	Amazon Linux	2024 年 1 月 8 日	2024 年 2 月 8 日	2025年2月28日
Go 1.x	go1.x	Amazon Linux	2024 年 1 月 8 日	2024 年 2 月 8 日	2025年2月28日
僅限作業系統的執行期	provided	Amazon Linux	2024 年 1 月 8 日	2024 年 2 月 8 日	2025年2月28日

名稱	識別符	作業系統	取代日期	封鎖函數建立	封鎖函數更新
Ruby 2.7	ruby2.7	Amazon Linux 2	2023 年 12 月 7 日	2024 年 1 月 9 日	2025年2月28日
Node.js 14	nodejs14.x	Amazon Linux 2	2023 年 12 月 4 日	2024 年 1 月 9 日	2025年2月28日
Python 3.7	python3.7	Amazon Linux	2023 年 12 月 4 日	2024 年 1 月 9 日	2025年2月28日
.NET Core 3.1	dotnetcore3.1	Amazon Linux 2	2023 年 4 月 3 日	2023 年 4 月 3 日	2023 年 5 月 3 日
Node.js 12	nodejs12.x	Amazon Linux 2	2023 年 3 月 31 日	2023 年 3 月 31 日	2023 年 4 月 30 日
Python 3.6	python3.6	Amazon Linux	2022 年 7 月 18 日	2022 年 7 月 18 日	2022 年 8 月 29 日
.NET 5 (僅限容器)	dotnet5.0	Amazon Linux 2	2022 年 5 月 10 日		
.NET Core 2.1	dotnetcore2.1	Amazon Linux	2022 年 1 月 5 日	2022 年 1 月 5 日	2022 年 4 月 13 日
Node.js 10	nodejs10.x	Amazon Linux 2	2021 年 7 月 30 日	2021 年 7 月 30 日	2022 年 2 月 14 日
Ruby 2.5	ruby2.5	Amazon Linux	2021 年 7 月 30 日	2021 年 7 月 30 日	2022 年 3 月 31 日
Python 2.7	python2.7	Amazon Linux	2021 年 7 月 15 日	2021 年 7 月 15 日	2022 年 5 月 30 日
Node.js 8.10	nodejs8.10	Amazon Linux	2020 年 3 月 6 日		2020 年 3 月 6 日
Node.js 4.3	nodejs4.3	Amazon Linux	2020 年 3 月 5 日		2020 年 3 月 5 日

名稱	識別符	作業系統	取代日期	封鎖函數建立	封鎖函數更新
Node.js 4.3 邊緣	`nodejs4.3-edge`	Amazon Linux	2020 年 3 月 5 日		2019 年 4 月 30 日
Node.js 6.10	`nodejs6.10`	Amazon Linux	2019 年 8 月 12 日	2019 年 8 月 12 日	
.NET Core 1.0	`dotnetcore1.0`	Amazon Linux	2019 年 6 月 27 日		2019 年 7 月 30 日
.NET Core 2.0	`dotnetcore2.0`	Amazon Linux	2019 年 5 月 30 日		2019 年 5 月 30 日
Node.js 0.10	`nodejs`	Amazon Linux			2016 年 10 月 31 日

在幾乎所有情況下，語言版本或操作系統的 end-of-life 日期都是事先知道的。下列連結提供 Lambda 作為受管執行階段支援的每種語言的 end-of-life 排程。

語言和框架支援政策

- Node.js – github.com
- Python – devguide.python.org
- Ruby – www.ruby-lang.org
- Java – www.oracle.com 和 Corretto 常見問答集
- Go – golang.org
- .NET – dotnet.microsoft.com

Lambda 執行階段更新

Lambda 提供安全性更新、錯誤修正、新功能、效能增強功能，以及次要版本支援，讓每個受管理執行階段都保持最新狀態。這些執行階段更新會發佈為執行階段版本。Lambda 會將函數從舊的執行階段版本遷移至新的執行階段版本，藉此將執行階段更新套用至函數。

預設情況下，對於使用受管執行階段的函數，Lambda 會自動套用執行階段更新。若使用自動執行階段更新，Lambda 需承受修補執行階段版本的運作負擔。對大多數客戶而言，自動更新是正確的選擇。如需詳細資訊，請參閱 執行階段管理控制。

Lambda 也會將每個新的執行階段版本發佈為容器映像。若要更新容器型函數的執行階段版本，您必須從更新的基礎映像建立新的容器映像，然後重新部署函數。

每個執行階段版本都與一個版本編號和一個 ARN (Amazon Resource Name) 相關聯。執行階段版本編號使用 Lambda 定義的編號結構描述，與程式設計語言使用的版本編號無關。執行階段版本 ARN 是每個執行階段版本的唯一識別符。

您可以在函數日誌的 `INIT_START` 行和 Lambda 主控台中，檢視函數目前執行階段版本的 ARN。

執行階段版本不應與執行階段識別符混淆。每個執行階段都有唯一的執行階段識別符，例如 `python3.9` 或 `nodejs18.x`。這些會對應到每個主要的程式設計語言版本。執行階段版本會描述個別執行階段的修補程式版本。

> **Note**
>
> 在 AWS 區域 和 CPU 架構之間，相同執行階段版本編號的 ARN 可能會有所不同。

主題

執行階段管理控制

Lambda 致力於提供與現有函數回溯相容的執行階段更新。但是與軟體修補一樣，在極少數情況下，執行階段更新可能會對現有函數產生負面影響。例如，安全性修補程式可能會暴露現有函數的潛在問題，取決於先前的不安全行為。在罕見的執行階段版本不相容情況下，Lambda 執行階段管理控制有助於降低對工作負載造成影響的風險。針對每個函數版本 ($LATEST 或已發佈版本)，您可以選擇下列其中一種執行階段更新模式：

- Auto (default) (自動 (預設)) - 使用 兩階段執行階段版本推展 自動更新為最新且安全的執行階段版本。我們建議大多數客戶使用此模式，便能隨時受益於執行階段更新。
- 函數更新：當您更新函數時，更新為最新且安全的執行階段版本。當您更新函數時，Lambda 會將函數的執行階段更新為最新且安全的執行階段版本。此做法可同步執行階段更新與函數部署，讓您控制 Lambda 何時套用執行階段更新。使用此模式，您可以及早偵測並減輕罕見的執行階段更新不相容情況。使用此模式時，您必須定期更新函數，讓執行階段保持最新狀態。
- 手動：手動更新您的執行階段版本。您可以在函數組態中指定執行階段版本。該函數將無限期使用此執行階段版本。在罕見情況下，新的執行階段版本會與現有函數不相容，您可以使用此模式，讓函數復原至較舊的執行階段版本。建議您不要使用 Manual (手動) 模式來嘗試達到跨部署的執行階段一致性。如需詳細資訊，請參閱 復原執行階段版本。

將執行階段更新套用至函數的責任分配，會根據您選擇的執行階段更新模式而有所不同。如需詳細資訊，請參閱 共同責任模式。

兩階段執行階段版本推展

Lambda 會依照以下順序引入新的執行階段版本：

1. 在第一階段，每當您建立或更新函數，Lambda 都會套用新的執行階段版本。當您呼叫UpdateFunctionCode或 UpdateFunctionConfigurationAPI 作業時，會更新函數。
2. 在第二階段，Lambda 會更新任何使用 Auto (自動) 執行階段更新模式且尚未更新為新執行階段版本的函數。

推展程序的整體持續時間會因許多因素而有所不同，包括執行階段更新中所包含任何安全性修補程式的嚴重性。

如果您正在開發和部署函數，很可能會在第一階段選擇新的執行階段版本。這會將執行階段更新與函數更新同步。在極少數情況下，最新的執行階段版本會對您的應用程式造成負面影響，而此做法可讓您立即採取更正動作。非開發中的函數仍然會在第二階段獲得自動執行階段更新的運作效益。

此方法不會影響設為 Function update (函數更新) 或 Manual (手動) 模式的函數。使用 Function update (函數更新) 模式的函數，只會在您建立或更新函數時收到最新的執行階段更新。使用 Manual (手動) 模式的函數不會收到執行階段更新。

Lambda 以漸進、滾動方式於 AWS 區域發佈新的執行階段版本。如果您的函數設為 Auto (自動) 或 Function update (函數更新) 模式，那麼同時部署至不同區域或同一地區在不同時間部署的函數，可能會採用不同的執行階段版本。需要在不同環境中保證執行期版本一致性的客戶，應 使用容器映像來部署 Lambda 函數。手動 模式設計為暫時緩解措施，可在發生執行階段版本不相容函數的罕見情況下啟用執行階段版本復原。

復原執行階段版本

在罕見情況下，新的執行階段版本會與現有函數不相容，您可以將執行階段版本復原為較舊的版本。這可讓您的應用程式保持正常運作並將中斷時間縮到最短，這樣便有時間在改回最新執行階段版本之前修正不相容情況。

Lambda 不會對您可以使用任何特定執行階段版本的時間長度施加時間限制。不過，我們強烈建議您盡快更新至最新的執行階段版本，以便享有最新的安全性修補程式、效能改善項目和功能。Lambda 提供復原為較舊執行階段版本的選項，只能在發生執行階段更新相容性問題的罕見情況中當成暫時緩解的手段。長時間使用舊版執行階段版本的函數，最終可能會遇到效能降低或發生問題 (例如憑證到期)，這可能會導致它們無法正常運作。

您可以透過下列方式復原執行階段版本：

- 使用 Manual (手動) 執行階段更新模式
- 使用已發佈的函數版本

如需詳細資訊，請參閱AWS Compute 部落格上的 AWS Lambda 執行階段管理控制介紹。

使用 Manual (手動) 執行階段更新模式復原執行階段版本

如果您使用的是 Auto (自動) 執行階段版本更新模式，或是使用 $LATEST 執行階段版本，您可以使用 Manual (手動) 模式復原執行階段版本。針對您要復原的函數版本，將執行階段版本更新模式變更為 Manual (手動)，並指定上一個執行階段版本的 ARN。如需查找舊版執行階段 ARN 的詳細資訊，請參閱：識別執行階段版本變更。

> **Note**
>
> 如果您的函數版本 $LATEST 設定為使用 Manual (手動) 模式，則無法變更函數使用的 CPU 架構或執行階段版本。若要進行這些變更，您必須變更為 Auto (自動) 或 Function update (函數更新) 模式。

使用已發布的函數版本復原執行階段版本

已發佈的函數版本是您建立 $LATEST 函數程式碼和組態時的不可變快照。在 Auto (自動) 模式中，Lambda 會在執行階段版本推展的第二階段期間，自動更新已發佈函數版本的執行階段版本。在 Function update (函數更新) 模式中，Lambda 不會更新已發佈函數版本的執行階段版本。

因此，使用 Function update (函數更新) 模式發佈的函數版本，會建立函數程式碼、組態和執行階段版本的靜態快照。透過將 Function update (函數更新) 模式與函數版本搭配使用，您可以將執行階段更新與部署同步。您也可以將流量重新導向至先前發佈的函數版本，協調程式碼、組態和執行階段版本的復原。您可以將此做法整合到持續整合和持續交付 (CI/CD) 中，以便在發生執行階段更新不相容的罕見情況下進行全自動復原。使用此做法時，您必須定期更新函數，並發佈新的函數版本以取得最新的執行階段更新。如需詳細資訊，請參閱 共同責任模式。

識別執行階段版本變更

執行階段版本號碼和 ARN 會記錄在記INIT_START錄行中，Lambda 會在每次建立新的執行環境時發出至 CloudWatch 記錄檔。由於執行環境對所有函數調用都是使用相同的執行期版本，因此 Lambda 只會在執行初始化階段時發送 INIT_START 日誌行。Lambda 不會為每個函數調用發出此日誌行。Lambda 會將記錄行發送至 CloudWatch 記錄，但在主控台中看不到。

Example Example INIT_START 日誌行

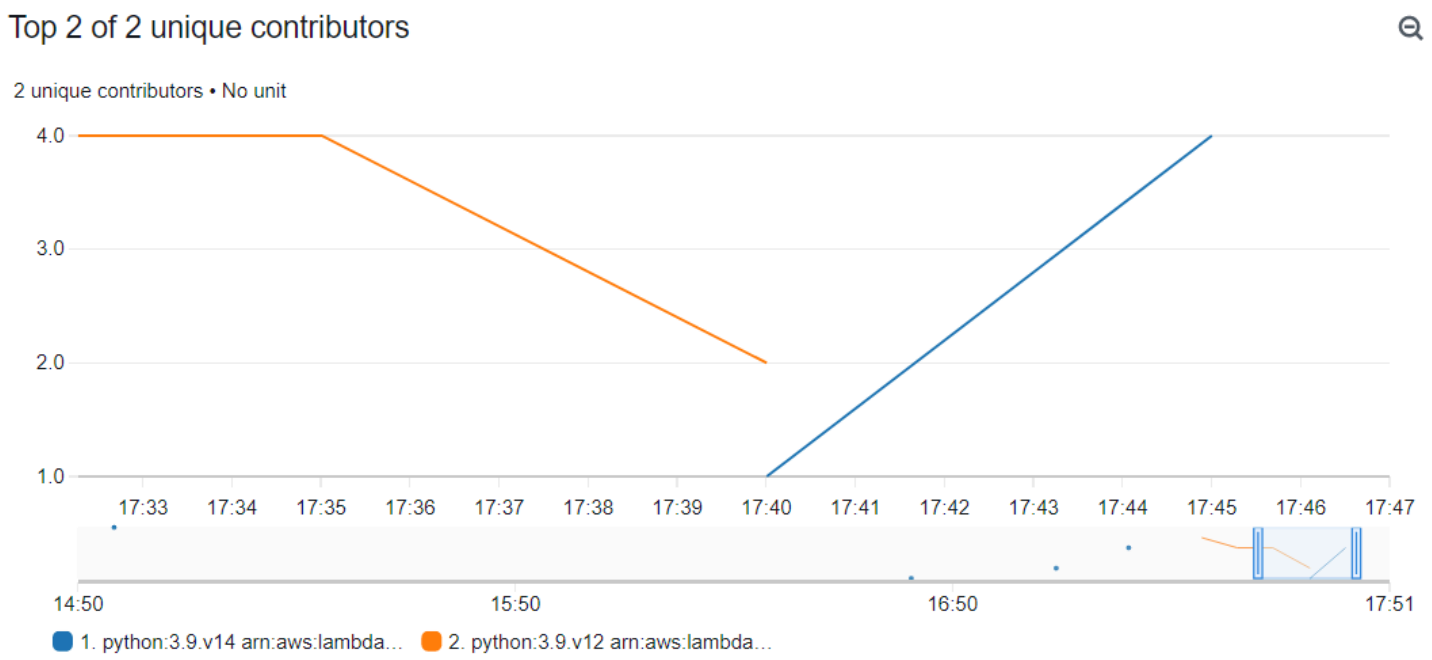
```
INIT_START Runtime Version: python:3.9.v14    Runtime Version ARN: arn:aws:lambda:eu-
south-1::runtime:7b620fc2e66107a1046b140b9d320295811af3ad5d4c6a011fad1fa65127e9e6I
```

您可以使用 Amazon CloudWatch 貢獻者深入解析來識別執行階段版本之間的轉換，而不是直接使用日誌。下列規則會計算每個 INIT_START 日誌行中不同的執行階段版本。若要使用此規則，請將範例日誌群組名稱 /aws/lambda/* 取代為函數或函數群組的適當前置詞。

```
{
  "Schema": {
```

```
    "Name": "CloudWatchLogRule",
    "Version": 1
  },
  "AggregateOn": "Count",
  "Contribution": {
    "Filters": [
      {
        "Match": "eventType",
        "In": [
          "INIT_START"
        ]
      }
    ],
    "Keys": [
      "runtimeVersion",
      "runtimeVersionArn"
    ]
  },
  "LogFormat": "CLF",
  "LogGroupNames": [
    "/aws/lambda/*"
  ],
  "Fields": {
    "1": "eventType",
    "4": "runtimeVersion",
    "8": "runtimeVersionArn"
  }
}
```

下列 CloudWatch 參與者見解報表會顯示前述規則所擷取之執行階段版本轉換的範例。橘線顯示舊執行階段版本 (python:3.9.v12) 的執行環境初始化，藍線顯示新執行階段版本 (python:3.9.v14) 的執行環境初始化。



配置執行階段管理設定

您可以使用 Lambda 主控台或 AWS Command Line Interface (AWS CLI) 來配置執行階段管理設定。

> **Note**
>
> 您可以針對每個函數版本分別配置執行階段管理設定。

設定 Lambda 如何更新執行階段版本 (主控台)

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇函數的名稱。
3. 在 Code (程式碼) 索引標籤的 Runtime settings (執行階段設定) 底下，選擇 Edit runtime management configuration (編輯執行階段組態)。
4. 在 Runtime management configuration (執行階段管理組態) 底下，選擇下列其中一項：
   - 若要讓函數自動更新為最新的執行階段版本，請選擇 Auto (自動)。
   - 若要在變更函數時將函數更新為最新的執行階段版本，請選擇 Function update (函數更新)。
   - 若要讓您的函數只在變更執行階段版本 ARN 時才更新為最新的執行階段版本，請選擇 Manual (手動)。

> **Note**
>
> 您可以在 Runtime management configuration (執行階段管理組態) 下找到執行階段版本 ARN。您也可以在函數日誌的 `INIT_START` 行中找到 ARN。

5. 選擇儲存。

設定 Lambda 如何更新執行階段版本 (AWS CLI)

若要設定函數的執行階段管理，您可以使用 `put-runtime-management-config` AWS CLI 命令搭配執行階段更新模式。使用 `Manual` 模式時，您還必須提供執行階段版本 ARN。

```
aws lambda put-runtime-management-config --function-name arn:aws:lambda:eu-west-1:069549076217:function:myfunction --update-runtime-on Manual --runtime-version-arn arn:aws:lambda:eu-west-1::runtime:8eeff65f6809a3ce81507fe733fe09b835899b99481ba22fd75b5a7338290ec1
```

您應該會看到類似下列的輸出：

```
{
  "UpdateRuntimeOn": "Manual",
  "FunctionArn": "arn:aws:lambda:eu-west-1:069549076217:function:myfunction",
  "RuntimeVersionArn": "arn:aws:lambda:eu-west-1::runtime:8eeff65f6809a3ce81507fe733fe09b835899b99481ba22fd75b5a7338290ec1"
}
```

共同責任模式

Lambda 負責為所有受支援的受管執行階段與容器映像策劃和發佈安全性更新。更新現有函數以使用最新執行階段版本的責任分配，視您使用的執行階段更新模式而有所不同。

Lambda 負責將執行階段更新套用至設定為使用 Auto (自動) 執行階段更新模式的所有函數。

對於使用 Function update (函數更新) 執行階段更新模式設定的函數，您必須負責定期更新函數。當您進行這些更新，Lambda 會負責套用執行階段更新。如果您沒有更新函數，Lambda 就不會更新執行階段。如果您沒有定期更新函數，強烈建議您設定為自動執行階段更新，以便繼續收到安全性更新。

對於設定為使用 Manual (手動) 執行階段更新模式的函數，您必須負責將函數更新為使用最新的執行階段版本。強烈建議您使用此模式的目的，只是在發生執行階段更新不相容的罕見情況下，將執行階段版

本復原當成暫時緩和措施。我們也建議您盡快變更為 Auto (自動) 模式，以縮短函數處於未修補狀態的時間。

如果您使用容器映像來部署函數，則 Lambda 負責發佈更新的基礎映像。在這種情況下，您需負責從最新的基礎映像重建函數的容器映像，並重新部署容器映像。

以下表進行總結：

部署模式	Lambda 的責任	客户的責任
受管執行階段，Auto (自動) 模式	發佈包含最新修補程式的新執行階段版本。 將執行階段修補程式套用到現有函數	在發生執行階段更新相容性問題的罕見情況下，還原至先前的執行階段版本。
受管執行階段，Function update (函數更新) 模式	發佈包含最新修補程式的新執行階段版本。	定期更新函數以取得最新的執行階段版本。 若您沒有定期更新函數，請將函數切換至Auto (自動) 模式。 在發生執行階段更新相容性問題的罕見情況下，還原至先前的執行階段版本。
受管執行階段，Manual (手動) 模式	發佈包含最新修補程式的新執行階段版本。	此模式僅適用於發生執行階段更新相容性問題的罕見情況時，暫時復原執行階段。 請盡快將函數切換至 Auto (自動) 或 Function update (函數更新) 模式及最新的執行階段版本。
容器映像	發佈包含最新修補程式的新容器映像。	使用最新的容器基礎映像定期重新部署函數，以取得最新的修補程式。

如需與 AWS 共同責任的詳細資訊，請參閱「AWS 雲端 安全性」網站上的共同責任模式。

高相容性應用程式

為了滿足修補需求，Lambda 客戶通常仰賴自動執行階段更新。如果您的應用程式對修補更新狀態有嚴格要求，您可能需要限制使用舊版執行階段。您可以使用 AWS Identity and Access Management

(IAM) 拒絕AWS帳戶中的使用者存取 PutRuntimeManagementConfigAPI 作業，以限制 Lambda 的執行階段管理控制。此操作用於選擇函數的執行階段更新模式。拒絕存取此操作會導致所有函數均預設為 Auto (自動) 模式。您可以使用服務控制政策 (SCP) 在對整個組織套用此限制。如果您必須將函數復原至較早的執行階段版本，則可以依據授與原則例外狀 case-by-case 況。

修改執行階段環境

您可以使用內部延伸項目來修改執行階段程序。內部延伸項目不是單獨程序，它們會作為執行時程序的一部分執行。

Lambda 提供特定語言環境變數，您可以進行設定，以便將選項和工具新增至執行時間。Lambda 還提供包裝函數指令碼，它允許 Lambda 將執行時間啟動委派給您的指令碼。您可以建立包裝程式指令碼來自訂執行階段啟動行為。

特定語言的環境變數

Lambda 支援僅限組態的方式，可透過下列特定語言的環境變數，在函數初始化期間預先載入程式碼：

- `JAVA_TOOL_OPTIONS` - 在 Java 中，Lambda 支援此環境變數，以在 Lambda 中設定其他命令列變數。此環境變數可讓您指定工具的初始化，特別是在您使用 `agentlib` 或 `javaagent` 選項啟動原生或 Java 程式設計語言代理程式時。如需詳細資訊，請參閱 `JAVA_TOOL_OPTIONS` 環境變數。
- `NODE_OPTIONS` - 可在 Node.js 執行期中使用。
- `DOTNET_STARTUP_HOOKS` - 在 .NET Core 3.1 及更高版本中，此環境變數指定了 Lambda 可以使用的組件 (dll) 的路徑。

使用特定語言的環境變數是設定啟動屬性的慣用方式。

包裝函式指令碼

您可以建立包裝函數指令碼來自訂 Lambda 函數的執行時間啟動行為。包裝函式指令碼可讓您設定無法透過特定語言環境變數來設定的組態參數。

> **Note**
>
> 如果包裝函式指令碼未成功啟動執行階段程序，調用可能會失敗。

所有原生 Lambda 執行期均支援包裝程式指令碼。僅限作業系統的執行期 (`provided` 執行期系列) 不支援包裝程式指令碼。

當您為函數使用包裝函數指令碼時，Lambda 會使用您的指令碼啟動執行時間。Lambda 會向您的指令碼傳送解譯器的路徑，以及標準執行時間啟動的所有原始引數。您的指令碼可以延伸或轉換程式的啟動行為。例如，指令碼可以插入和更改引數、設定環境變數，或擷取指標、錯誤和其他診斷資訊。

您可以透過將 AWS_LAMBDA_EXEC_WRAPPER 環境變數的值設定為可執行二進位檔案或指令碼的檔案系統路徑來指定指令碼。

範例：使用 Python 3.8 建立並使用包裝函式指令碼

在下面的例子中，您建立一個包裝函式指令碼來啟動與 `-X importtime` 選項的 Python 解譯器。當您執行函數時，Lambda 會產生日誌項目，以顯示每個匯入的匯入持續時間。

使用 Python 3.8 建立和使用包裝函式指令碼

1. 若要建立包裝函式指令碼，請將下列程式碼貼到名為 `importtime_wrapper` 的檔案中：

```
#!/bin/bash

# the path to the interpreter and all of the originally intended arguments
args=("$@")

# the extra options to pass to the interpreter
extra_args=("-X" "importtime")

# insert the extra options
args=("${args[@]:0:$#-1}" "${extra_args[@]}" "${args[@]: -1}")

# start the runtime with the extra options
exec "${args[@]}"
```

2. 若要授予指令碼可執行的許可，請從指令行輸入 `chmod +x importtime_wrapper`。
3. 將指令碼部署為 Lambda 層。
4. 使用 Lambda 主控台建立函數。
   a. 開啟 Lambda 主控台。
   b. 選擇建立函數。
   c. 在 Basic information (基本資訊) 下，為 Function name (函數名稱) 輸入 **`wrapper-test-function`**。
   d. 針對執行階段，選擇 Python 3.8。
   e. 選擇建立函數。
5. 將圖層新增到您的函式中。

a. 選擇您的函數，然後選擇 Code (程式碼) (如果尚未選取)。
b. 選擇 Add a layer (新增 layer)。
c. 在選擇圖層下，選擇您先前建立之相容圖層的名稱和版本。
d. 選擇新增。

6. 將代碼和環境變數新增到您的函式中。

a. 在函式程式碼編輯器中，貼上下列函式程式碼：

```
import json

  def lambda_handler(event, context):
      # TODO implement
      return {
          'statusCode': 200,
          'body': json.dumps('Hello from Lambda!')
      }
```

b. 選擇儲存。
c. 在 Environment variables (環境變數) 下，選擇 Edit (編輯)。
d. 選擇 Add environment variable (新增環境變數)。
e. 在 Key (索引鍵) 欄位，輸入 `AWS_LAMBDA_EXEC_WRAPPER`。
f. 針對數值，輸入 `/opt/importtime_wrapper`。
g. 選擇儲存。

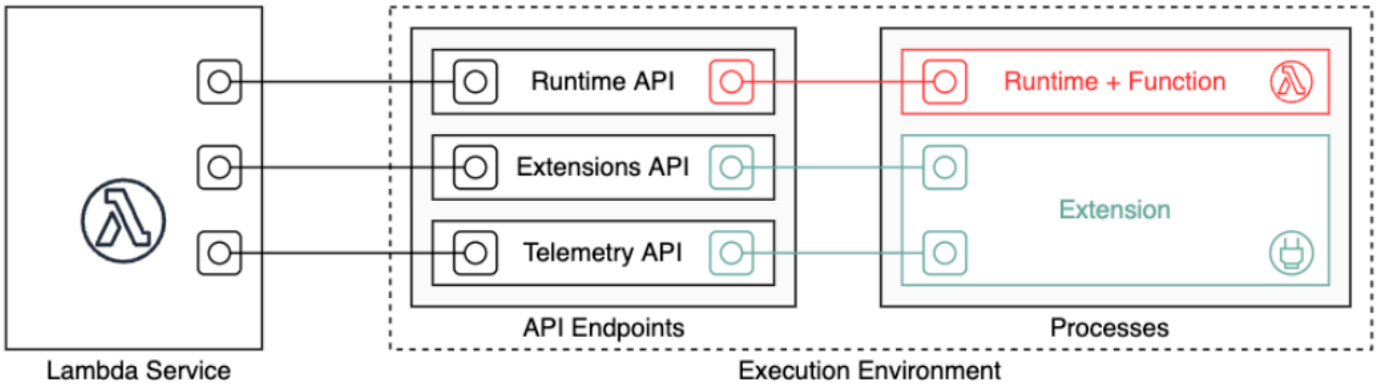
7. 若要執行函式，請選擇測試。

因為您的包裝函式指令碼使用 `-X importtime` 選項啟動 Python 解譯器，所以日誌會顯示每次導入所需的時間。例如：

```
...
2020-06-30T18:48:46.780+01:00 import time: 213 | 213 | simplejson
2020-06-30T18:48:46.780+01:00 import time: 50 | 263 | simplejson.raw_json
...
```

Lambda 執行階段 API

AWS Lambda 提供了用於自訂執行時間的 HTTP API 以接收來自 Lambda 的調用事件，並將回應資料傳回至 Lambda 執行環境。



執行時間 API 版本 2018-06-01 的 OpenAPI 規格可從 runtime-api.zip 中獲得

若要建立 API 請求 URL，執行時間會從 AWS_LAMBDA_RUNTIME_API 環境變數中取得 API 端點，新增 API 版本，以及新增所需的資源路徑。

Example 請求

```
curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/next"
```

API 方法

- 下次調用
- 調用回應
- 初始化錯誤
- 調用錯誤

下次調用

路徑 – `/runtime/invocation/next`

方法 – GET

執行時間會傳送此訊息至 Lambda 來請求調用事件。回應內文包含該次調用的承載，此 JSON 文件含有取自函式觸發條件的事件資料。回應標頭包含有關該次調用的額外資料。

回應標頭

- `Lambda-Runtime-Aws-Request-Id` - 請求 ID，它將識別觸發函數調用的請求。

  例如 `8476a536-e9f4-11e8-9739-2dfe598c3fcd`。
- `Lambda-Runtime-Deadline-Ms` - 函數逾時的日期，以 Unix 時間毫秒為單位。

  例如 `1542409706888`。
- `Lambda-Runtime-Invoked-Function-Arn` - 在調用中指定的 Lambda 函數、版本或別名的 ARN。

  例如 `arn:aws:lambda:us-east-2:123456789012:function:custom-runtime`。
- `Lambda-Runtime-Trace-Id` - AWS X-Ray 追蹤標頭。

  例如 `Root=1-5bef4de7-ad49b0e87f6ef6c87fc2e700;Parent=9a9197af755a6419;Sampled=1`。
- `Lambda-Runtime-Client-Context` - 由 AWS Mobile SDK 調用時，此為有關用戶端應用程式和裝置的資料。
- `Lambda-Runtime-Cognito-Identity` - 由 AWS Mobile SDK 調用時，此為有關 Amazon Cognito 身分提供者的資料。

請勿在 `GET` 請求上設定逾時，因為回應可能會延遲。在 Lambda 引導執行時間與執行時間有可傳回的事件之間，執行時間處理可能會凍結幾秒鐘。

請求 ID 將追蹤 Lambda 內的調用。您可以在傳送回應時使用此值指定調用。

追蹤標頭包含追蹤 ID、父系 ID 和抽樣決策。如果對請求進行抽樣，請求將由 Lambda 或上游服務進行抽樣。執行時間應設定 `_X_AMZN_TRACE_ID` 為標頭的值。X-Ray 開發套件將讀取此項以取得 ID 並決定是否要追蹤請求。

調用回應

路徑 – `/runtime/invocation/`*`AwsRequestId`*`/response`

方法 – POST

在函數執行到完成之後，執行時間會傳送調用回應至 Lambda。若為同步調用，Lambda 會將回應傳送給用戶端。

Example 成功請求

```
REQUEST_ID=156cb537-e2d4-11e8-9b34-d36013741fb9
curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/$REQUEST_ID/
response"  -d "SUCCESS"
```

初始化錯誤

如果函數傳回錯誤或執行時間在初始化期間遇到錯誤，執行時間會使用此方法將錯誤報告給 Lambda。

路徑 – `/runtime/init/error`

方法 – POST

標頭

`Lambda-Runtime-Function-Error-Type` - 執行時間遇到的錯誤類型。必要：否。

此標頭包含一個字串值。Lambda 可接受任何字串，但我們建議使用格式 <category.reason>。例如：

- 執行階段。 NoSuchHandler
- 運行時 KeyNotFound
- 執行階段。 ConfigInvalid
- 執行階段。 UnknownReason

主體參數

`ErrorRequest` - 關於錯誤的資訊。必要：否。

此欄位是具有下列結構的 JSON 物件：

```
{
      errorMessage: string (text description of the error),
      errorType: string,
      stackTrace: array of strings
}
```

請注意，Lambda 接受 `errorType` 的任何值。

下列範例顯示 Lambda 函數錯誤訊息，其中函數無法剖析調用中提供的事件資料。

Example 函數錯誤

```
{
      "errorMessage" : "Error parsing event data.",
      "errorType" : "InvalidEventDataException",
      "stackTrace": [ ]
}
```

回應內文參數

- StatusResponse – 字串. 狀態信息，隨 202 回應代碼一起傳送。
- ErrorResponse - 其他錯誤資訊，與錯誤回應代碼一起傳送。 ErrorResponse 包含錯誤類型和錯誤訊息。

回應代碼

- 202 - 已接受
- 403 - 禁止
- 500 - 容器錯誤。不可復原的狀態。執行時間應立即退出。

Example 初始化錯誤請求

```
ERROR="{\"errorMessage\" : \"Failed to load function.\", \"errorType\" :
 \"InvalidFunctionException\"}"
curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/init/error" -d "$ERROR" --
header "Lambda-Runtime-Function-Error-Type: Unhandled"
```

調用錯誤

如果函數傳回錯誤或執行時間遇到錯誤，執行時間會使用此方法將錯誤報告給 Lambda。

路徑 – /runtime/invocation/*AwsRequestId*/error

方法 – POST

標頭

Lambda-Runtime-Function-Error-Type - 執行時間遇到的錯誤類型。必要：否。

此標頭包含一個字串值。Lambda 可接受任何字串，但我們建議使用格式 <category.reason>。例如：

- 執行階段。 NoSuchHandler
- 運行時 KeyNotFound
- 執行階段。 ConfigInvalid
- 執行階段。 UnknownReason

主體參數

ErrorRequest - 關於錯誤的資訊。必要：否。

此欄位是具有下列結構的 JSON 物件：

```
{
      errorMessage: string (text description of the error),
      errorType: string,
      stackTrace: array of strings
}
```

請注意，Lambda 接受 errorType 的任何值。

下列範例顯示 Lambda 函數錯誤訊息，其中函數無法剖析調用中提供的事件資料。

Example 函數錯誤

```
{
      "errorMessage" : "Error parsing event data.",
      "errorType" : "InvalidEventDataException",
      "stackTrace": [ ]
}
```

回應內文參數

- StatusResponse – 字串. 狀態信息，隨 202 回應代碼一起傳送。
- ErrorResponse - 其他錯誤資訊，與錯誤回應代碼一起傳送。 ErrorResponse 包含錯誤類型和錯誤訊息。

回應代碼

- 202 - 已接受
- 400 - 錯誤請求

- 403 - 禁止
- 500 - 容器錯誤。不可復原的狀態。執行時間應立即退出。

Example 錯誤請求

```
REQUEST_ID=156cb537-e2d4-11e8-9b34-d36013741fb9
ERROR="{\"errorMessage\" : \"Error parsing event data.\", \"errorType\" :
 \"InvalidEventDataException\"}"
curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/$REQUEST_ID/error"
 -d "$ERROR" --header "Lambda-Runtime-Function-Error-Type: Unhandled"
```

何時使用 Lambda 的僅限作業系統執行階段

Lambda 為 Java、Python、Node.js、.NET 和 Ruby 提供受管理執行期。若要使用未提供受管理執行期的程式設計語言建立 Lambda 函數，請使用僅限作業系統的執行期 (provided 執行期系列)。僅限作業系統的執行期有三種主要的使用案例：

- 本機 ahead-of-time （AOT）編譯：Go，Rust 和 C ++ 之類的語言本地編譯為可執行二進製文件，該二進製文件不需要專用的語言運行時。這些語言僅需要可在其中執行編譯二進位檔的作業系統環境。您還可以使用 Lambda 僅限作業系統的執行期來部署以 .NET 原生 AOT 和 Java GraalVM Native 編譯的二進位檔。

  您必須在二進位中包含執行期介面用戶端。執行期介面用戶端會呼叫 Lambda 執行階段 API 來擷取函數調用，然後呼叫您的函數處理常式。Lambda 為 Go、.NET 原生 AOT、C++ 和 Rust (實驗性) 提供執行期介面用戶端。

  您必須將二進位檔編譯為適用於 Linux 環境，以及您計劃用於函數的同一指令集架構 (x86_64 或 arm64)。
- 第三方執行階段：您可以使用 off-the-shelf 執行階段來執行 Lambda 函數，例如 PHP 版 B ref 或 Swift 的 Swift 執行 AWS Lambda 階段。
- 自訂執行期：您可以為 Lambda 不提供受管理執行期的語言或語言版本建置自己的執行期，例如 Node.js 19。如需詳細資訊，請參閱 構建自定義運行時 AWS Lambda。這是僅限作業系統的執行期最不常見的使用案例。

Lambda 支援以下僅限作業系統的執行期：

僅限作業系統

名稱	識別符	作業系統	取代日期	封鎖函數建立	封鎖函數更新
僅限作業系統的執行期	`provided.al2023`	Amazon Linux 2023			
僅限作業系統的執行期	`provided.al2`	Amazon Linux 2			

與 Amazon Linux 2 相比，Amazon Linux 2023 (`provided.al2023`) 執行期具有多項優點，包括更小的部署足跡和更新版本的程式庫，如 `glibc`。

provided.al2023 執行期使用 dnf 而非 yum 做為套件管理工具，後者是 Amazon Linux 2 中的預設套件管理工具。如需provided.al2023和之間差異的詳細資訊provided.al2，請參閱 AWS 運算部落格 AWS Lambda上的介紹 Amazon Linux 2023 執行階段。

構建自定義運行時 AWS Lambda

您可以在任何程式設計語言中實作 AWS Lambda 執行階段。執行時間是一款程式，它會在調用 Lambda 函數時執行該函數的處理常式方法。您可將執行期納入函數的部署套件或是在層中分發。建立 Lambda 函數時，選擇僅限作業系統的執行期 (provided 執行期系列)。

> **Note**
>
> 建立自訂執行期是一種進階使用案例。如果您正在尋找有關編譯為本機二進製文件或使用第三方 off-the-shelf 運行時的信息，請參閱何時使用 Lambda 的僅限作業系統執行階段。

如需了解自訂執行期部署過程的演練，請參閱 教學課程：建置自訂執行期。您還可以在 awslab aws-lambda-cpp s/ 上探索在 C ++ 中實現的自定義運行時。 GitHub

主題

- 要求
- 在自訂執行階段中實作回應串流

要求

自訂執行階段必須完成特定的初始化和處理工作。執行期會執行函數的設定程式碼、從環境變數中讀取處理常式名稱，並從 Lambda 執行期 API 中讀取叫用事件。執行時間會將事件資料傳遞至函數處理常式，並將處理常式的回應發佈回 Lambda。

初始化任務

初始化任務是按照函式的每一執行個體各執行一次，以備妥用於處理調用的環境。

- 擷取設定 - 讀取環境變數以取得關於函數和環境的詳細資訊。
  - _HANDLER - 處理常式所在位置，取自函數的組態。標準格式為 *file*.*method*，其中 file 是不含副檔名的檔案名稱，而 method 則是定義於該檔案內的方法或函式的名稱。
  - LAMBDA_TASK_ROOT - 包含函數程式碼的目錄。

- AWS_LAMBDA_RUNTIME_API - 執行時間 API 的主機和連接埠。

  如需可用變數的完整清單，請參閱 定義執行時間環境變數。

- 初始化函數 - 載入處理常式檔案並執行其所包含的任何全域或靜態程式碼。函式應只建立一次靜態資源 (如開發套件用戶端和資料庫連線)，以供多次調用時重複使用。
- 處理錯誤 - 如果發生錯誤，則呼叫初始化錯誤 API 並立即退出。

初始化會計入計費執行時間和逾時。當執行觸發新函數執行個體的初始化時，您可以在記錄和 AWS X-Ray 追蹤中查看初始化時間。

Example log

```
REPORT RequestId: f8ac1208... Init Duration: 48.26 ms   Duration: 237.17 ms   Billed
 Duration: 300 ms   Memory Size: 128 MB   Max Memory Used: 26 MB
```

處理任務

經執行後，執行時間會使用 Lambda 執行時間介面來管理傳入的事件並報告錯誤。初始化任務完成後，執行時間將以迴圈處理傳入的事件。在您的執行時間程式碼中，依序執行下列步驟。

- 取得事件 - 呼叫下次調用 API 以取得下一個事件。回應內文包含事件資料。回應標頭包含請求 ID 及其他資訊。
- 傳播追蹤標頭 - 從 API 回應中的 Lambda-Runtime-Trace-Id 標頭取得 X-Ray 追蹤標頭。使用相同的值在本機設定 _X_AMZN_TRACE_ID 環境變數。X-Ray 開發套件使用這個值來連接服務之間的追蹤資料。
- 建立內容物件 - 建立含有內容資訊的物件，其資訊取自 API 回應中的環境變數和各標頭。
- 調用函數處理常式 - 將事件與內容物件傳遞至處理常式。
- 處理回應 - 呼叫調用回應 API 以發佈處理常式的回應。
- 處理錯誤 - 如果發生錯誤，則呼叫調用錯誤 API。
- 清理 - 釋放未使用的資源、傳送資料至其他服務，或是執行額外的任務後再取得下一個事件。

進入點

自訂執行時間的進入點是名為 bootstrap 的可執行檔。引導檔案可做為執行時間，也可以調用另一個建立執行時間的檔案。如果部署套件的根目錄不包含名為 bootstrap 的檔案，Lambda

會在函數的層中尋找該檔案。若 bootstrap 檔案不存在或不是可執行檔，函數會在調用時傳回 Runtime.InvalidEntrypoint 錯誤。

以下是使用 Node.js 的捆綁版本的示例bootstrap文件，在名為的單獨文件中 JavaScript 運行運行時runtime.js。

Example 引導

```
#!/bin/sh
    cd $LAMBDA_TASK_ROOT
    ./node-v11.1.0-linux-x64/bin/node runtime.js
```

在自訂執行階段中實作回應串流

針對回應串流函數，response 和 error 端點稍微修改了行為，允許執行期將部分回應串流至用戶端，並以區塊形式傳回承載。如需特定行為的詳細資訊，請參閱以下內容：

- /runtime/invocation/AwsRequestId/response - 傳播執行期的 Content-Type 標頭以傳送至用戶端。Lambda 透過 HTTP/1.1 區塊傳輸編碼，以區塊為單位傳回回應承載。回應串流的大小上限為 20 MiB。若要將回應串流至 Lambda，執行期必須：
  - 將Lambda-Runtime-Function-Response-Mode HTTP 標頭設為 streaming。
  - 將Transfer-Encoding 標頭設為 chunked。
  - 編寫符合 HTTP/1.1 區塊傳輸編碼規範的回應。
  - 成功編寫回應後會關閉基礎連線。
- /runtime/invocation/AwsRequestId/error - 執行期可以使用此端點向 Lambda 報告函數或執行期錯誤，Lambda 也接受 Transfer-Encoding 標頭。只能在執行階段開始傳送叫用回應之前呼叫此端點。
- 在 /runtime/invocation/AwsRequestId/response 中使用錯誤尾端報告串流中間錯誤 - 若要報告在執行期開始編寫叫用回應後出現的錯誤，執行期可選擇連接名為 Lambda-Runtime-Function-Error-Type 和 Lambda-Runtime-Function-Error-Body 的 HTTP 尾端標頭。Lambda 會將此視為成功回應，並將執行期提供的錯誤中繼資料轉送給用戶端。

> **Note**
>
> 若要附加尾端標頭，執行階段必須在 HTTP 要求的開頭設定標頭值 Trailer。這是 HTTP/1.1 區塊傳輸編碼規範的要求。

- `Lambda-Runtime-Function-Error-Type` - 執行期遇到的錯誤類型。此標頭包含一個字串值。Lambda 可接受任何字串，但我們建議使用格式 *<category.reason>*。例如 `Runtime.APIKeyNotFound`。
- `Lambda-Runtime-Function-Error-Body` - 有關錯誤的 Base64 編碼資訊。

教學課程：建置自訂執行期

在本教學課程中，您將建立具有自訂執行時間的 Lambda 函數。首先，您要將執行時間納入函式的部署套件中。接著再將其遷移到與函式分開而單獨管理的 Layer。最後，您要透過更新該執行時間 Layer 以資源為基礎的許可政策，將其與世界各地的人共享。

必要條件

此教學課程假設您具備基本的 Lambda 操作知識並了解 Lambda 主控台。若您尚未了解，請遵循 使用主控台建立一個 Lambda 函數 中的指示，建立您的第一個 Lambda 函數。

若要完成下列步驟，您需要 AWS Command Line Interface (AWS CLI) 版本 2。命令和預期的輸出會列在不同的區塊中：

```
aws --version
```

您應該會看到下列輸出：

```
aws-cli/2.13.27 Python/3.11.6 Linux/4.14.328-248.540.amzn2.x86_64 exe/x86_64.amzn.2
```

對於長命令，逸出字元 (\) 用於將命令分割為多行。

在 Linux 和 macOS 上，使用您偏好的 shell 和套件軟體管理工具。

> **Note**
>
> 在 Windows 中，作業系統的內建終端不支援您常與 Lambda 搭配使用的某些 Bash CLI 命令 (例如 `zip`)。若要取得 Ubuntu 和 Bash 的 Windows 整合版本，請安裝適用於 Linux 的 Windows 子系統。本指南中的 CLI 命令範例使用 Linux 格式。如果您使用的是 Windows CLI，必須重新格式化包含內嵌 JSON 文件的命令。

您需要 IAM 角色來建立 Lambda 函數。角色需要權限才能將日誌發送到 CloudWatch 日誌並訪問您的功能使用的AWS服務。如果您還沒有函數開發角色，請立即建立一個。

若要建立執行角色

1. 在 IAM 主控台中開啟 角色頁面。
2. 選擇 建立角色。
3. 建立具備下列屬性的角色。
   - 信任實體 - Lambda。
   - 權限 — AWSLambdaBasicExecutionRole.
   - 角色名稱 - **lambda-role**。

   該AWSLambdaBasicExecutionRole策略具有函數將日誌寫入日誌所需的 CloudWatch 權限。

建立函數

建立具有自訂執行時間的 Lambda 函數。本範例包括兩個檔案：執行期 `bootstrap` 檔案以及函數處理常式。兩個檔案都是以 Bash 實作。

1. 建立專案的目錄，然後切換至該目錄。

```
mkdir runtime-tutorial
cd runtime-tutorial
```

2. 建立稱為 `bootstrap` 的新檔案。這是自訂執行期。

   Example 引導

```
#!/bin/sh

set -euo pipefail

# Initialization - load function handler
source $LAMBDA_TASK_ROOT/"$(echo $_HANDLER | cut -d. -f1).sh"

# Processing
while true
do
```

```
  HEADERS="$(mktemp)"
  # Get an event. The HTTP request will block until one is received
  EVENT_DATA=$(curl -sS -LD "$HEADERS" "http://
${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/next")

  # Extract request ID by scraping response headers received above
  REQUEST_ID=$(grep -Fi Lambda-Runtime-Aws-Request-Id "$HEADERS" | tr -d
 '[:space:]' | cut -d: -f2)

  # Run the handler function from the script
  RESPONSE=$($(echo "$_HANDLER" | cut -d. -f2) "$EVENT_DATA")

  # Send the response
  curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/$REQUEST_ID/
response"  -d "$RESPONSE"
done
```

執行時間將從部署套件載入函式指令碼。其使用了兩個變數以找出指令碼。LAMBDA_TASK_ROOT 將告知套件解壓縮的位置，而 _HANDLER 則包含指令碼的名稱。

在執行期載入函數指令碼之後，它將使用執行期 API 從 Lambda 中擷取調用事件、傳遞事件至處理常式，並將回應發布回 Lambda。為了取得請求 ID，執行時間將 API 回應中的各標頭儲存至暫時檔案，然後從該檔案讀取 Lambda-Runtime-Aws-Request-Id 標頭。

> **Note**
>
> 執行時間另還負責其他任務，包括錯誤處理以及向處理常式提供內容資訊。如需詳細資訊，請參閱 要求。

3. 為函數建立指令碼。以下範例指令碼定義了一個接受事件資料的處理常式函數，會將該資料記錄到 stderr 並予以傳回。

   Example function.sh

```
function handler () {
  EVENT_DATA=$1
  echo "$EVENT_DATA" 1>&2;
  RESPONSE="Echoing request: '$EVENT_DATA'"

  echo $RESPONSE
}
```

runtime-tutorial 目錄現在應該看起來像這樣：

```
runtime-tutorial
# bootstrap
# function.sh
```

4. 將檔案轉成可執行檔並加入至 .zip 封存檔。這是部署套件。

```
chmod 755 function.sh bootstrap
zip function.zip function.sh bootstrap
```

5. 建立名為的函數 bash-runtime。對於 --role，輸入 Lambda 執行角色的 ARN。

```
aws lambda create-function --function-name bash-runtime \
--zip-file fileb://function.zip --handler function.handler --runtime
 provided.al2023 \
--role arn:aws:iam::123456789012:role/lambda-role
```

6. 調用函數。

```
aws lambda invoke --function-name bash-runtime --payload '{"text":"Hello"}'
 response.txt --cli-binary-format raw-in-base64-out
```

如果您使用 AWS CLI 第 2 版，則需要 cli-binary-format 選項。若要讓此成為預設的設定，請執行 aws configure set cli-binary-format raw-in-base64-out。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

您應該看到如下回應：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
```

7. 確認回應。

```
cat response.txt
```

您應該看到如下回應：

```
Echoing request: '{"text":"Hello"}'
```

建立 Layer

為了將執行時間程式碼與函式程式碼分開，您要建立一個僅包含執行時間的 Layer。Layer 讓您能夠單獨開發函式的依存項目，且若搭配多個函式使用同一 Layer 還可減少儲存空間用量。如需詳細資訊，請參閱 使用層管理 Lambda 相依性。

1. 建立包含 `bootstrap` 檔案的 .zip 檔案。

   ```
   zip runtime.zip bootstrap
   ```

2. 使用 publish-layer-version 命令建立 Layer。

   ```
   aws lambda publish-layer-version --layer-name bash-runtime --zip-file fileb://
   runtime.zip
   ```

   如此即建立了 Layer 的第一個版本。

更新函數

若要搭配函式使用執行期 Layer，請將函式設定成使用 Layer，並且移除函式中的執行期程式碼。

1. 更新函式組態以提取 Layer。

   ```
   aws lambda update-function-configuration --function-name bash-runtime \
   --layers arn:aws:lambda:us-east-1:123456789012:layer:bash-runtime:1
   ```

   這會將執行時間新增至 `/opt` 目錄中的函數。若要確保 Lambda 使用層中的執行期，您必須將 `boostrap` 從函數的部署套件中移除，如接下來兩個步驟所示。

2. 建立包含函數程式碼的 .zip 檔案。

   ```
   zip function-only.zip function.sh
   ```

3. 更新函式程式碼，使之僅包含處理常式指令碼。

```
aws lambda update-function-code --function-name bash-runtime --zip-file fileb://
function-only.zip
```

4. 調用函數以確認其是否可搭配執行期 layer 運作。

```
aws lambda invoke --function-name bash-runtime --payload '{"text":"Hello"}'
 response.txt --cli-binary-format raw-in-base64-out
```

如果您使用 AWS CLI 第 2 版，則需要 cli-binary-format 選項。若要讓此成為預設的設定，請執行 `aws configure set cli-binary-format raw-in-base64-out`。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

您應該看到如下回應：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
```

5. 確認回應。

```
cat response.txt
```

您應該看到如下回應：

```
Echoing request: '{"text":"Hello"}'
```

更新執行時間

1. 若要記錄執行環境的相關資訊，請更新執行時間指令碼使其輸出環境變數。

Example 引導

```
#!/bin/sh

set -euo pipefail

# Configure runtime to output environment variables
echo "##  Environment variables:"
```

```
env

# Load function handler
source $LAMBDA_TASK_ROOT/"$(echo $_HANDLER | cut -d. -f1).sh"

# Processing
while true
do
  HEADERS="$(mktemp)"
  # Get an event. The HTTP request will block until one is received
  EVENT_DATA=$(curl -sS -LD "$HEADERS" "http://
${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/next")

  # Extract request ID by scraping response headers received above
  REQUEST_ID=$(grep -Fi Lambda-Runtime-Aws-Request-Id "$HEADERS" | tr -d
 '[:space:]' | cut -d: -f2)

  # Run the handler function from the script
  RESPONSE=$($(echo "$_HANDLER" | cut -d. -f2) "$EVENT_DATA")

  # Send the response
  curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/$REQUEST_ID/
response"  -d "$RESPONSE"
done
```

2. 建立包含新版本 bootstrap 檔案的 .zip 檔案。

```
zip runtime.zip bootstrap
```

3. 建立 bash-runtime 層的新版本。

```
aws lambda publish-layer-version --layer-name bash-runtime --zip-file fileb://
runtime.zip
```

4. 設定函式以使用新版本的 Layer。

```
aws lambda update-function-configuration --function-name bash-runtime \
--layers arn:aws:lambda:us-east-1:123456789012:layer:bash-runtime:2
```

共享 Layer

如欲將層使用許可授予其他帳戶，請透過 add-layer-version-permission 命令將陳述式新增至層版本的許可政策。在各陳述式中，您可將許可授予單一帳戶、所有帳戶或某個組織。

以下列範例會授予帳戶 111122223333 存取 `bash-runtime` layer 第 2 版的許可。

```
aws lambda add-layer-version-permission --layer-name bash-runtime --statement-id
 xaccount \
--action lambda:GetLayerVersion --principal 111122223333 --version-number 2 --output
 text
```

您應該會看到類似下列的輸出：

```
e210ffdc-e901-43b0-824b-5fcd0dd26d16    {"Sid":"xaccount","Effect":"Allow","Principal":
{"AWS":"arn:aws:iam::111122223333:root"},"Action":"lambda:GetLayerVersion","Resource":"arn:aws:
east-1:123456789012:layer:bash-runtime:2"}
```

許可僅適用於單一層版本。每次建立新的層版本時均需重複此程序。

清除

刪除各個版本的 Layer。

```
aws lambda delete-layer-version --layer-name bash-runtime --version-number 1
aws lambda delete-layer-version --layer-name bash-runtime --version-number 2
```

由於函數持有對該層的第 2 版的參考，所以它仍存在於 Lambda 中。本函式將繼續運作，但無法再設定各函式使用已刪除的版本。若您修改了函數的 layer 清單，則必須指定新的版本或略去已刪除的 layer。

使用 delete-function 命令刪除函數。

```
aws lambda delete-function --function-name bash-runtime
```

在 Lambda 中使用 AVX2 向量化

進階向量延伸項目 2 (AVX2) 是 Intel x86 指令集的向量化延伸項目，可透過 256 位元的向量來執行單一指令多重資料 (SIMD) 指令。針對具有高度可平行化操作的向量演算法，使用 AVX2 可以增強 CPU 效能，從而降低延遲並提高輸送量。將 AVX2 指令集用於運算密集型工作負載，例如機器學習推論、多媒體處理、科學模擬和財務模型應用程式。

> **Note**
>
> Lambda arm64 使用 NEON SIMD 架構，且不支援 x86 AVX2 延伸模組。

若要將 AVX2 與您的 Lambda 函數搭配使用，請確保您的函數程式碼將存取 AVX2 最佳化程式碼。針對某些語言，您可以安裝 AVX2 支援的程式庫和套件版本。針對其他語言，您可以使用適當的編譯器標記集 (如果編譯器支援自動向量化)，來重新編譯程式碼和相依項。您還可以使用 AVX2 來最佳化數學運算的第三方程式庫，來編譯程式碼。例如，Intel Math Kernel Library (Intel MKL)、OpenBLAS (Basic Linear Algebra Subprograms) 和 AMD BLAS-like Library Instantiation Software (BLIS)。自動向量化的語言 (如 Java) 會自動使用 AVX2 進行運算。

您可以建立新的 Lambda 工作負載，或免費將啟用 AVX2 的現有工作負載移至 Lambda。

如需有關 AVX2 的詳細資訊，請參閱 Wikipedia 中的進階向量延伸項目 2。

從來源編譯

如果您的 Lambda 函數使用 C 或 C ++ 程式庫來執行運算密集型向量化操作，則可以設定適當的編譯器標記並重新編譯函數程式碼。然後，編譯器會自動向量化您的程式碼。

針對 `gcc` 或 `clang` 編譯器，請將 `-march=haswell` 新增至指令或將 `-mavx2` 設定為命令選項。

```
~ gcc -march=haswell main.c
or
~ gcc -mavx2 main.c

~ clang -march=haswell main.c
or
~ clang -mavx2 main.c
```

若要使用特定的程式庫，請遵循程式庫文件中的指令來編譯和建置程式庫。例如，要 TensorFlow 從源代碼構建，您可以按照 TensorFlow 網站上的安裝說明進行操作。確保使用 `-march=haswell` 編譯選項。

針對 Intel MKL 啟用 AVX2

Intel MKL 是最佳化數學運算的程式庫，其在運算平台提供支援時隱式使用 AVX2 指令。框架，例如默認情況下使用英特爾 MKL 構 PyTorch 建，因此您無需啟用 AVX2。

某些程式庫 (例如) 會在建置程序中提供選項 TensorFlow，以指定 Intel MKL 最佳化。例如，使用 TensorFlow，使用選`--config=mkl`項。

您還可以使用英特爾 MKL 構建流行的科學 Python 庫 NumPy，例如 SciPy 和。如需有關使用 Intel MKL 建置這些程式庫的說明，請參閱 Intel 網站上 Intel MKL 和 Intel 編譯器的 Numpy/Scipy。

有關英特爾 MKL 和類似庫的更多信息，請參閱維基百科中的數學內核庫，OpenBLAS 網站和 AMD BLIS 存儲庫。 GitHub

AVX2 的其他語言支援

如果您不使用 C 或 C ++ 程式庫，而且不使用 Intel MKL 進行建置，您仍然可以針對您的應用程式取得一些 AVX2 效能改善。請注意，實際改善取決於編譯器或解譯器在您的程式碼上利用 AVX2 功能的能力。

Python

Python 使用者通常使用 SciPy 和程 NumPy 式庫來處理運算密集型工作負載。您可以編譯這些程式庫以啟用 AVX2，或者使用支援 Intel MKL 的程式庫版本。

節點

針對運算密集型工作負載，請使用您需要的支援 AVX2 或 Intel MKL 的程式庫版本。

Java

Java 的 JIT 編譯器可以自動向量化您的程式碼，以使用 AVX2 指令來執行。如需有關偵測向量化程式碼的資訊，請參閱 OpenJDK 網站上的 JVM 中的程式碼向量化簡報。

Go

標準 Go 編譯器目前不支援自動向量化，但您可以使用 Go 的 GCC 編譯器 gccgo。設定 `-mavx2` 選項：

```
gcc -o avx2 -mavx2 -Wall main.c
```

內部函數

可以使用多種語言的內部函數，手動向量化您的程式碼以使用 AVX2。不過，我們不建議使用這種方法。手動編寫向量化程式碼需要大量工作。此外，偵錯和維護此類程式碼比使用依賴於自動向量化的程式碼更為困難。

配置 AWS Lambda 函數

了解如何使用 Lambda API 或主控台來設定 Lambda 函數的核心功能和選項。

記憶體

了解如何以及何時增加功能記憶。

暫時性儲存

了解如何以及何時增加功能的臨時存儲容量。

Timeout (逾時)

瞭解如何以及何時增加函數的逾時值。

環境變數

您可以讓函數程式碼具備可攜性，並使用環境變量來將它們儲存在函數的組態中，以便保存在程式碼之外。

傳出網路

您可以將 Lambda 函數與 Amazon VPC 中的 AWS 資源搭配使用。將函數連線到 VPC 可讓您在關聯式資料庫和快取等私有子網路中存取資源。

傳入網路

您可以使用界面 VPC 端點來叫用您的 Lambda 函數，而不需要透過公有網際網路。

檔案系統

您可以使用 Lambda 函數來將 Amazon EFS 掛載至本機目錄。檔案系統可讓您的函數程式碼安全存取和修改共用資源，並發揮高度並行效能。

別名

您可以將用戶端設定為使用別名來調用特定的 Lambda 函數版本，而非更新用戶端。

版本

發佈您的函數版本，即可將程式碼和組態儲存為無法變更的獨立資源。

回應串流

您可以設定 Lambda 函數 URL，將回應承載串流回用戶端。透過提高第一個位元組時間 (TTFB) 效能，回應串流有益於延遲敏感應用程式。這是因為您可以在部分回應可用時將其傳回給用戶端。此外，您可以使用回應串流來建置可傳回更大承載的函數。

設定函 Lambda 記憶體

Lambda 會按設定的記憶體數量比例來配置 CPU 功率。記憶體是可供 Lambda 函數在執行時間使用的記憶體數量。您可以使用 [記憶體] 設定增加或減少分配給功能的記憶體和 CPU 電源。您可以設定介於 128 MB 到 10,240 MB 之間的記憶體，以 1 MB 為增量。在 1,769 MB，函數等於一個完整 vCPU (每秒一個 vCPU 秒的額度)。

本頁說明如何以及何時更新 Lambda 函數的記憶體設定。

章節

決定 Lambda 函數的適當記憶體設定

記憶體是控制功能效能的主要控制桿。預設設定 128 MB 是可能的最低設定。建議您只針對簡單的 Lambda 函數使用 128 MB，例如將事件轉換並路由至其他 AWS 服務的函數。較高的記憶體分配可以改善使用匯入程式庫、Lambda 層、亞馬遜簡單儲存服務 (Amazon S3) 或亞馬遜彈性檔案系統 (Amazon EFS) 的函數的效能。按比例增加更多記憶體會增加 CPU 的數量，從而提高可用的整體計算能力。如果某個功能是 CPU，網絡或內存限制，則增加內存設置可以顯著提高其性能。

若要尋找適合您函數的記憶體配置，我們建議您使用開放原始碼AWS Lambda 電源調整工具。此工具用 AWS Step Functions 於在不同的記憶體配置下執行多個並行版本的 Lambda 函數，並測量效能。輸入函數會在您的 AWS 帳戶中執行，執行即時 HTTP 呼叫和 SDK 互動，以測量即時生產案例中可能的效能。您也可以實作 CI/CD 程序，以使用此工具自動測量您部署的新功能的效能。

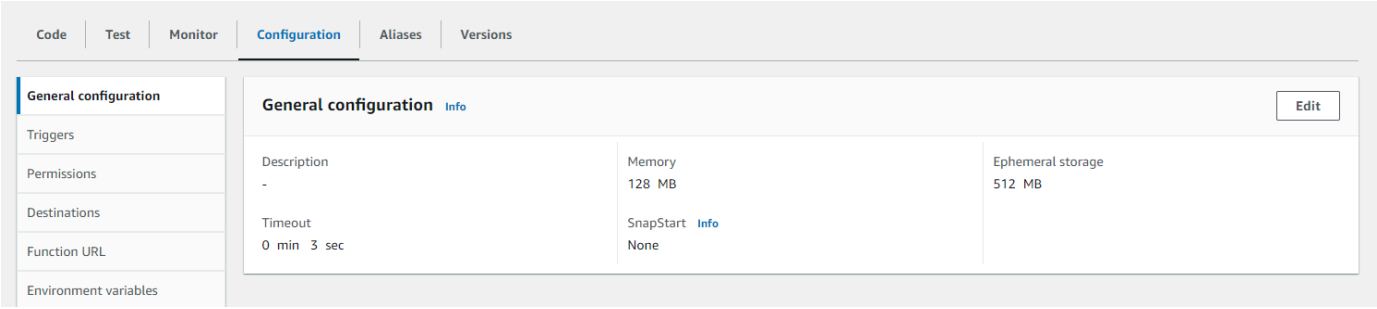
設定函數記憶體 (主控台)

您可以在 Lambda 主控台中設定函數的記憶體。

更新函數記憶體

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。

3. 選擇組態索引標籤，然後選擇一般組態。



4. 在一般組態下，選擇編輯。
5. 在記憶體中，請設定介於 128 MB 到 10,240 MB 之間的值。
6. 選擇儲存。

配置函數記憶體 (AWS CLI)

您可以使用update-function-configuration指令來設定函數的記憶體。

Example

```
aws lambda update-function-configuration \
  --function-name my-function \
  --memory-size 1024
```

配置函數記憶體 (AWS SAM)

您可以使用AWS Serverless Application Model來配置功能的記憶體。更新文`template.yaml`件中的MemorySize屬性，然後運行 sam 部署。

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Application Model template describing your function.
Resources:
  my-function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Description: ''
      MemorySize: 1024
```

```
    # Other function properties...
```

接受函數記憶體建議 (主控台)

如果您在 AWS Identity and Access Management (IAM) 中具有管理員許可，則可以選擇從中接收 Lambda 函數記憶體設定建議 AWS Compute Optimizer。如需有關針對您的帳戶或組織選擇加入記憶體建議的指示，請參閱 AWS Compute Optimizer 使用者指南中的選擇加入您的帳戶。

> **Note**
>
> Compute Optimizer 只支援使用 x86_64 架構的函數。

當您已選擇加入且 Lambda 函數符合 Compute Optimizer 需求時，您可以在一般組態中檢視並接受來自 Lambda 主控台 Compute Optimizer 的函數記憶體建議。

設定 Lambda 函數的暫時儲存

Lambda 為目錄中的函數提供暫時性的儲存空間。/tmp每個執行環境都是暫時的，而且是唯一的儲存體。您可以使用暫時儲存設定來控制分配給功能的暫時儲存空間量。您可以設定介於 512 MB 到 10,240 MB 之間的暫時儲存空間，以 1 MB 為單位。儲存在中/tmp的所有資料都會使用由管理的金鑰進行靜態加密 AWS。

本頁說明常見使用案例，以及如何更新 Lambda 函數的暫時儲存體。

章節

增加暫時儲存空間的常見使用案例

以下是幾個受益於臨時儲存空間增加的常見使用案例：

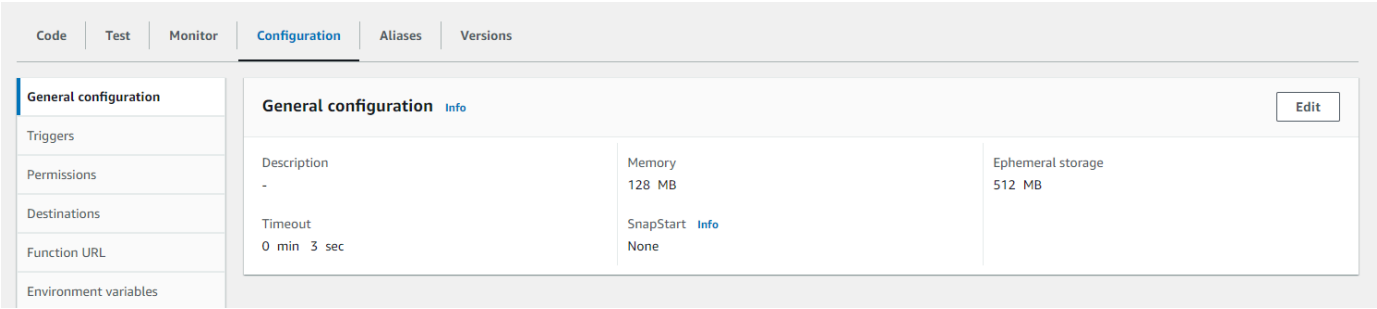
- E xtract-transform-load (ETL) 工作：當程式碼執行中間運算或下載其他資源以完成處理時，增加暫時儲存空間。更多暫存空間可讓更複雜的 ETL 任務在 Lambda 函數中執行。
- 機器學習 (ML) 推論：許多推論工作都仰賴大型參考資料檔案，包括程式庫和模型。有了更多暫時儲存，您可以從 Amazon Simple Storage Service (Amazon S3) 下載較大的模型，/tmp並在處理中使用它們。
- 資料處理：對於從 Amazon S3 下載物件以回應 S3 事件的工作負載，更多/tmp空間可讓您在不使用記憶體內處理的情況下處理較大的物件。建立 PDF 或處理媒體的工作負載也會受益於更短暫的儲存空間。
- 圖形處理：圖像處理是基於 Lambda 的應用程序的常見用例。對於處理大型 TIFF 檔案或衛星影像的工作負載，更多暫時儲存可讓您更輕鬆地在 Lambda 中使用程式庫和執行運算。

配置暫時性儲存（主控台）

您可以在 Lambda 主控台中設定暫時儲存。

若要修改函數的暫時儲存

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇組態索引標籤，然後選擇一般組態。



4. 在一般組態下，選擇編輯。
5. 對於暫時性儲存，請設定介於 512 MB 到 10,240 MB 之間的值，以 1 MB 為增量。
6. 選擇儲存。

設定暫時儲存 ()AWS CLI

您可以使用該update-function-configuration命令來配置臨時存儲。

Example

```
aws lambda update-function-configuration \
  --function-name my-function \
  --ephemeral-storage '{"Size": 1024}'
```

設定暫時儲存 ()AWS SAM

您可以使用AWS Serverless Application Model來為您的功能配置臨時儲存。更新文template.yaml件中的EphemeralStorage屬性，然後運行 sam 部署。

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Application Model template describing your function.
Resources:
  my-function:
```

```
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Description: ''
      MemorySize: 128
      Timeout: 120
      Handler: index.handler
      Runtime: nodejs20.x
      Architectures:
        - x86_64
      EphemeralStorage:
        Size: 10240
      # Other function properties...
```

設定 Lambda 函數逾時

Lambda 會在逾時之前執行程式碼一段設定的時間。逾時為 Lambda 函數可以執行的最長時間，以秒為單位。此設定的預設值為 3 秒，但您可以將此值調整為 1 秒的增量，最大值為 900 秒 (15 分鐘)。

本頁說明如何以及何時更新 Lambda 函數的逾時設定。

章節

判斷 Lambda 函數的適當逾時值

如果逾時值接近函數的平均持續時間，則函數會意外逾時的風險較高。函數的持續時間可能會根據數據傳輸和處理的量以及與該函數交互的任何服務的延遲而有所不同。逾時的一些常見原因包括：

- 從亞馬遜簡單儲存服務 (Amazon S3) 的下載量比平均值更大或需要更長的時間。
- 函數向另一個服務發出請求，這需要更長的時間來響應。
- 提供給函數的參數在函數中需要更多的計算複雜性，這會導致調用花費更長的時間。

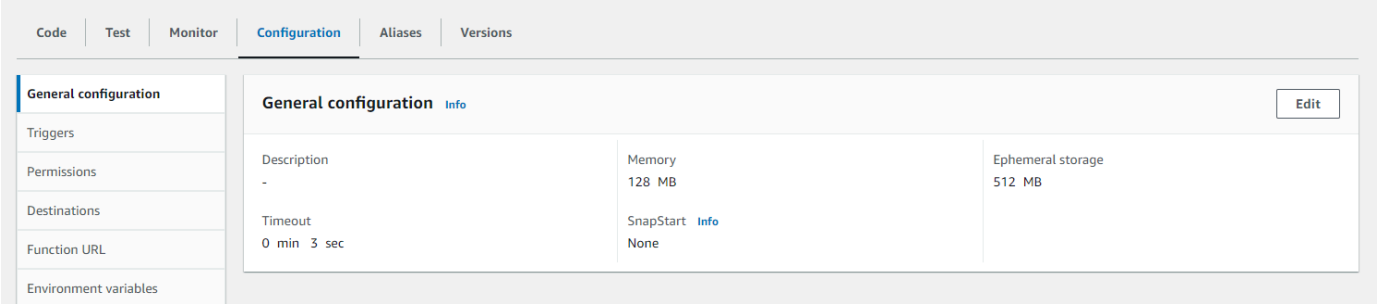
測試應用程式時，請確保您的測試能準確反映資料的大小和數量，以及實際參數值。為了方便起見，測試通常使用小範例，但是您應該在工作負載合理預期的上限使用資料集。

設定逾時 (主控台)

您可以在 Lambda 主控台中設定函數逾時。

若要修改函數的逾時

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇組態索引標籤，然後選擇一般組態。



4. 在一般組態下，選擇編輯。
5. 對於「逾時」，請設定介於 1 到 900 秒 (15 分鐘) 之間的值。
6. 選擇儲存。

配置超時（AWS CLI）

您可以使用指update-function-configuration令來設定逾時值 (以秒為單位)。下列範例指令會將函數逾時增加至 120 秒 (2 分鐘)。

Example

```
aws lambda update-function-configuration \
  --function-name my-function \
  --timeout 120
```

配置超時（AWS SAM）

您可以使用AWS Serverless Application Model來設定函數的逾時值。更新文`template.yaml`件中的超時屬性，然後運行 sam 部署。

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Application Model template describing your function.
Resources:
  my-function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Description: ''
      MemorySize: 128
```

```
      Timeout: 120
      # Other function properties...
```

使用 Lambda 環境變數來設定程式碼中的值

您可以使用環境變數來調整函數的行為，而無需更新程式碼。環境變數是存放在函數特定版本組態中的一對字串。Lambda 執行時間可讓程式碼使用環境變數，並設定其他環境變數，這些變數包含函數和調用請求的相關資訊。

> **Note**
>
> 為了提高安全性，我們建議您使 AWS Secrets Manager 用環境變數來儲存資料庫認證和其他敏感資訊，例如 API 金鑰或授權 Token。如需詳細資訊，請參閱使用建立和管理密碼 AWS Secrets Manager。

在函數調用之前，不會評估環境變數。您定義的任何值都會視為文字字串，而且不會展開。在函數程式碼中執行變數評估。

您可以使用 Lambda 主控台 ()、 AWS Command Line Interface (AWS CLI) 或使用 AWS 開發套件， AWS Serverless Application Model 在 Lambda 中設定環境變數。AWS SAM

Console

您可以在函數的未發佈版本上定義環境變數。當您發佈版本時，會鎖定該版本的環境變數以及其他版本特定組態設定。

您可以透過定義索引鍵和值，為函數建立環境變數。函數使用索引鍵的名稱來擷取環境變數的值。

在 Lambda 主控台中設定環境變數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇 Configuration (組態)，然後選擇 Environment variables (環境變數)。
4. 在 Environment variables (環境變數) 下，選擇 Edit (編輯)。
5. 選擇 Add environment variable (新增環境變數)。
6. 輸入索引鍵和值。

   請求

   - 索引鍵需以英文字母為開頭，且至少有兩個字元。
   - 索引鍵僅包含字母、數字和底線字元 (_)。

- 索引鍵不是由 Lambda 預留。
- 所有環境變數的大小總計不超過 4 KB。

7. 選擇 Save (儲存)。

在主控台程式碼編輯器中產生環境變數清單

可在 Lambda 程式碼編輯器中產生環境變數清單。這是在編碼時參考環境變數的快速方法。

1. 選擇 程式碼 標籤。
2. 選擇環境變數索引標籤。
3. 選擇工具、顯示環境變數。

在主控台程式碼編輯器中列出環境變數時，會保持加密狀態。如果在傳輸過程中啟用加密協助程式進行加密，則這些設定會維持不變。如需詳細資訊，請參閱 保護 Lambda 環境變數。

環境變數清單為唯讀狀態，且只能在 Lambda 主控台中使用。下載函數的 .zip 封存檔時，不會包含此檔案，且您無法透過上傳此檔案來新增環境變數。

AWS CLI

下列範例會在名為 `my-function` 的函數上設定兩個環境變數。

```
aws lambda update-function-configuration \
  --function-name my-function \
  --environment "Variables={BUCKET=DOC-EXAMPLE-BUCKET,KEY=file.txt}"
```

當您使用 `update-function-configuration` 命令套用環境變數時，會取代 `Variables` 結構的整個內容。若要在新增環境變數時保留現有的環境變數，請在請求中包含所有現有值。

若要取得目前的組態，請使用 `get-function-configuration` 命令。

```
aws lambda get-function-configuration \
  --function-name my-function
```

您應該會看到下列輸出：

```
{
    "FunctionName": "my-function",
    "FunctionArn": "arn:aws:lambda:us-east-2:111122223333:function:my-function",
```

```
    "Runtime": "nodejs20.x",
    "Role": "arn:aws:iam::111122223333:role/lambda-role",
    "Environment": {
        "Variables": {
            "BUCKET": "DOC-EXAMPLE-BUCKET",
            "KEY": "file.txt"
        }
    },
    "RevisionId": "0894d3c1-2a3d-4d48-bf7f-abade99f3c15",
    ...
}
```

可以從 `get-function-configuration` 的輸出中將修訂 ID 作為參數傳遞給 `update-function-configuration`。這樣可確保在讀取組態和更新組態期間，值不會變更。

若要設定函數的加密金鑰，請設定 KMSKeyARN 選項。

```
aws lambda update-function-configuration \
  --function-name my-function \
  --kms-key-arn arn:aws:kms:us-east-2:111122223333:key/055efbb4-xmpl-4336-ba9c-538c7d31f599
```

AWS SAM

您可以使用AWS Serverless Application Model來設定函數的環境變數。更新文`template.yaml`件中的環境和變量屬性，然後運行 sam 部署。

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Application Model template describing your function.
Resources:
  my-function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Description: ''
      MemorySize: 128
      Timeout: 120
      Handler: index.handler
      Runtime: nodejs18.x
      Architectures:
```

```
      - x86_64
    EphemeralStorage:
      Size: 10240
    Environment:
      Variables:
        BUCKET: DOC-EXAMPLE-BUCKET
        KEY: file.txt
    # Other function properties...
```

AWS SDKs

若要使用 AWS SDK 管理環境變數，請使用下列 API 作業。

- UpdateFunction配置
- GetFunction配置
- CreateFunction

要了解更多信息，請參閱 AWS SDK 文檔以獲取您首選的編程語言。

定義執行時間環境變數

Lambda 執行時間會在初始化期間設定數個環境變數。大多數的環境變數都會提供函數或執行時間的資訊。這些環境變數的索引鍵都已進行預留，無法在您的函數組態中設定。

預留環境變數

- _HANDLER - 在函數中設定的處理常式位置。
- _X_AMZN_TRACE_ID - X-Ray 追蹤標頭。此環境變數會隨著每次調用而變更。
  - 未針對僅限作業系統的執行期 (provided 執行期系列) 定義此環境變數。您可以使用 下次調用 中的 Lambda-Runtime-Trace-Id 回應標頭為自訂執行階段設定 _X_AMZN_TRACE_ID。
  - 對於 Java 執行期版本 17 及更高版本，不使用此環境變數。相反地，Lambda 會將追蹤資訊儲存在 com.amazonaws.xray.traceHeader 系統屬性中。
- AWS_DEFAULT_REGION— 執行 Lambda 函數的預設值 AWS 區域 。
- AWS_REGION— 執行 Lambda 函數的 AWS 區域 位置。若已完成定義，此值便會覆寫 AWS_DEFAULT_REGION。
  - 如需有關將 AWS 區域 環境變數與 AWS SDK 搭配使用的詳細資訊，請參閱 AWS SDK 和工具參考指南中的AWS 區域。

- AWS_EXECUTION_ENV – 執行時間識別符，字首為 AWS_Lambda_ (例如，AWS_Lambda_java8)。未針對僅限作業系統的執行期 (provided 執行期系列) 定義此環境變數。
- AWS_LAMBDA_FUNCTION_NAME - 函數的名稱。
- AWS_LAMBDA_FUNCTION_MEMORY_SIZE - 可供函數使用的記憶體量 (MB)。
- AWS_LAMBDA_FUNCTION_VERSION - 正在執行的函數版本。
- AWS_LAMBDA_INITIALIZATION_TYPE - 函數的初始化類型，即 on-demand、provisioned-concurrency 或 snap-start。如需資訊，請參閱設定佈建並行或 使用 Lambda 改善啟動效能 SnapStart。
- AWS_LAMBDA_LOG_GROUP_NAME、AWS_LAMBDA_LOG_STREAM_NAME — 功能的 Amazon CloudWatch 日誌群組和資料流的名稱。AWS_LAMBDA_LOG_GROUP_NAME和AWS_LAMBDA_LOG_STREAM_NAME環境變數在 Lambda SnapStart 函數中不可用。
- AWS_ACCESS_KEY、AWS_ACCESS_KEY_ID、AWS_SECRET_ACCESS_KEY、AWS_SESSION_TOKEN - 從函數的執行角色中取得的存取金鑰。
- AWS_LAMBDA_RUNTIME_API - (自訂執行時間) 執行時間 API 的主機和連接埠。
- LAMBDA_TASK_ROOT - 指向您 Lambda 函數程式碼的路徑。
- LAMBDA_RUNTIME_DIR - 指向執行時間程式庫的路徑。

下列其他環境變數並未進行預留，可以在您的函數組態中進行擴充。

未預留的環境變數

- LANG – 執行時間的地區設定 (en_US.UTF-8)。
- PATH – 執行路徑 (/usr/local/bin:/usr/bin/:/bin:/opt/bin)。
- LD_LIBRARY_PATH – 系統程式庫路徑 (/var/lang/lib:/lib64:/usr/lib64:$LAMBDA_RUNTIME_DIR:$LAMBDA_RUNTIME_DIR/lib:$LAMBDA_TASK_ROOT:$LAMBDA_TASK_ROOT/lib:/opt/lib)。
- NODE_PATH - (Node.js) Node.js 程式庫路徑 (/opt/nodejs/node12/node_modules/:/opt/nodejs/node_modules:$LAMBDA_RUNTIME_DIR/node_modules)。
- PYTHONPATH - (Python 2.7、3.6、3.8) Python 程式庫路徑 ($LAMBDA_RUNTIME_DIR)。
- GEM_PATH - (Ruby) Ruby 程式庫路徑 ($LAMBDA_TASK_ROOT/vendor/bundle/ruby/2.5.0:/opt/ruby/gems/2.5.0)。
- AWS_XRAY_CONTEXT_MISSING - 對於 X-Ray 追蹤，Lambda 將它設定為 LOG_ERROR，以避免從 X-Ray 開發套件中擲回執行時間錯誤。

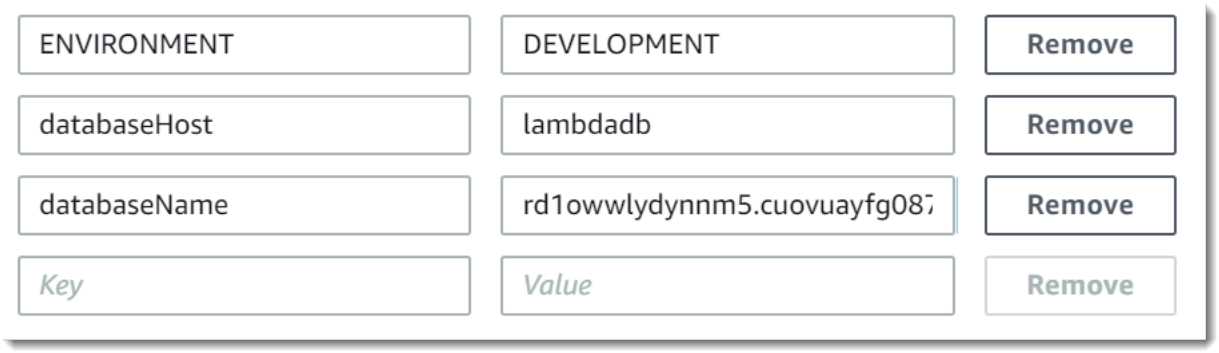
- AWS_XRAY_DAEMON_ADDRESS - 對於 X-Ray 追蹤，為 X-Ray 常駐程式的 IP 地址和連接埠。
- AWS_LAMBDA_DOTNET_PREJIT - 若是 .NET 6 和 .NET 7 執行期，請將此變數設為啟用或停用 .NET 特定執行期最佳化。值包含 always、never 和 provisioned-concurrency。如需詳細資訊，請參閱 設定函數的佈建並行。
- TZ – 環境的時區 (UTC)。執行環境使用 NTP 來同步系統時鐘。

顯示的範例值會反映最新的執行時間。特定變數或其值是否存在，可能會因先前的執行時間而有所不同。

環境變數的範例案例

您可以使用環境變數，自訂測試環境和生產環境中的函數行為。例如，您可以建立兩個具備相同程式碼，但不同組態的函數。一個函數連接到測試資料庫，另一個函數連接到生產資料庫。在此情況下，可以使用環境變數將資料庫的主機名稱和其他連線詳細資訊傳遞給函數。

以下範例顯示如何將資料庫主機和資料庫名稱定義為環境變數。



如果您希望測試環境產生比實際執行環境更多的偵錯資訊，您可以設定環境變數，以將測試環境設定為使用更詳細的記錄或更詳細的追蹤。

保護 Lambda 環境變數

若要保護您的環境變數，可以使用伺服器端加密來保護您的靜態資料，並使用用戶端加密來保護傳輸中的資料。

> **Note**
>
> 若要提高資料庫安全性，建議您使用來儲存資料庫認證， AWS Secrets Manager 而不要使用環境變數。如需詳細資訊，請參閱 AWS Lambda 與 Amazon RDS 一起使用。

靜態安全

Lambda 永遠使用 AWS KMS key提供靜態伺服器端加密。預設情況下，Lambda 使用 AWS 受管金鑰。如果此預設行為符合您的工作流程，您便不需要設定其他項目。Lambda 會 AWS 受管金鑰 在您的帳戶中建立，並為您管理該帳戶的許可。 AWS 不會向您收取使用此密鑰的費用。

如果您願意，您可以改為提供 AWS KMS 客戶管理的金鑰。您可以這樣做以控制 KMS 金鑰的輪換或滿足您的組織對管理 KMS 金鑰的請求。當您使用客戶受管的金鑰時，只有您帳戶中具有 KMS 金鑰存取權的使用者才能檢視或管理函數上的環境變數。

客戶受管金鑰會產生標準 AWS KMS 費用。如需詳細資訊，請參閱 AWS Key Management Service 定價。

傳輸中安全

為了提高額外的安全性，您可以為傳輸中的加密啟用協助程式，這可以確保您的環境變數在傳輸過程中在用戶端得到加密保護。

若要為環境變數配置加密

1. 使用 AWS Key Management Service (AWS KMS) 為 Lambda 建立任何客戶受管金鑰，以用於伺服器端和用戶端加密。如需詳細資訊，請參閱AWS Key Management Service 開發人員指南中的建立金鑰。
2. 使用 Lambda 主控台，導覽至 Edit environment variables (編輯環境變數) 頁面。
   a. 開啟 Lambda 主控台中的 函數頁面。
   b. 選擇一個函數。
   c. 選擇 Configuration (組態)，然後從左側導覽列選擇 Environment variables (環境變數)。
   d. 在 Environment variables (環境變數) 區段中，選擇 Edit (編輯)。
   e. 展開 Encryption configuration (加密組態)。
3. (選用) 啟用主控台加密協助程式，以使用用戶端加密來保護傳輸中的資料。
   a. 在 Encryption in transit (傳輸中加密)下，選擇 Enable helpers for encryption in transit (啟用傳輸中加密的協助程式)。
   b. 對於要為其啟用主控台加密協助程式的每個環境變數，請選擇環境變數旁邊的 Encrypt (加密)。
   c. 在傳輸過程中加密下 AWS KMS key ，選擇您在此程序開始時建立的客戶管理金鑰。

d. 選擇 Execution role policy (執行角色政策)，然後複製政策。此政策授予函數的執行角色解密環境變數的許可。

   儲存此政策，以便在此程序的最後一個步驟中使用。

e. 向函數中新增解密環境變數的程式碼。若要查看範例，請選擇解密秘密程式碼片段。

4. (選用) 指定用於靜態加密的客戶受管金鑰。

   a. 選擇 Use a customer master key (使用客戶主金鑰)。

   b. 選擇在此程序開始時建立的客戶受管金鑰。

5. 選擇 Save (儲存)。

6. 設定許可。

   如果搭配使用客戶受管金鑰和伺服器端加密，請向您希望其能檢視或管理該函數環境變數的任何使用者或角色授予許可。如需詳細資訊，請參閱 管理伺服器端加密 KMS 金鑰的許可。

   如果您要啟用用戶端加密來增強傳輸中安全，您的函數需要許可來呼叫 `kms:Decrypt` API 操作。將您先前在此程序中儲存的政策新增至函數的執行角色。

管理伺服器端加密 KMS 金鑰的許可

您的使用者或函數的執行角色不需要任何 AWS KMS 權限即可使用預設加密金鑰。若要使用客戶受管金鑰，您需要使用金鑰的許可。Lambda 會使用您的許可，在金鑰上建立授權。這麼做可讓 Lambda 使用它來進行加密。

- `kms:ListAliases` – 在 Lambda 主控台中檢視金鑰。
- `kms:CreateGrant`、`kms:Encrypt` - 在函數上設定客戶受管金鑰。
- `kms:Decrypt` - 檢視及管理使用客戶受管金鑰加密的環境變數。

您可以從您的 AWS 帳戶 或從金鑰的資源型權限原則取得這些權限。 `ListAliases`由 Lambda 的受管政策提供。金鑰政策會將其餘許可授予 Key users (金鑰使用者) 群組中的使用者。

沒有 `Decrypt` 許可的使用者仍然可以管理函數，但是無法在 Lambda 主控台中檢視或管理環境變數。如要防止使用者檢視環境變數，請將拒絕存取預設金鑰、客戶受管金鑰，或是所有金鑰的陳述式新增到使用者的許可。

Example IAM 政策 - 透過金鑰 ARN 拒絕存取

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "VisualEditor0",
            "Effect": "Deny",
            "Action": [
                "kms:Decrypt"
            ],
            "Resource": "arn:aws:kms:us-east-2:111122223333:key/3be10e2d-xmpl-4be4-
bc9d-0405a71945cc"
        }
    ]
}
```

如需有關管理金鑰許可的詳細資訊，請參閱《AWS Key Management Service 開發人員指南》中的AWS KMS中的金鑰政策。

擷取 Lambda 環境變數

若要在函數程式碼中擷取環境變數，請使用程式設計語言的標準方法。

Node.js

```
let region = process.env.AWS_REGION
```

Python

```
import os
  region = os.environ['AWS_REGION']
```

> **Note**
>
> 在某些情況下，您可能需要使用下列格式：
>
> ```
> region = os.environ.get('AWS_REGION')
> ```

Ruby

```
region = ENV["AWS_REGION"]
```

Java

```
String region = System.getenv("AWS_REGION");
```

Go

```
var region = os.Getenv("AWS_REGION")
```

C#

```
string region = Environment.GetEnvironmentVariable("AWS_REGION");
```

PowerShell

```
$region = $env:AWS_REGION
```

Lambda 透過靜態加密環境變數來安全地存放它們。您可以將 Lambda 設定為使用不同的加密金鑰、在用戶端加密環境變數值，或使用設定 AWS CloudFormation 範本中的環境變數 AWS Secrets Manager。

讓 Lambda 函數存取 Amazon VPC 中的資源

使用 Amazon Virtual Private Cloud (Amazon VPC)，您可以在自己的中建立私有網路 AWS 帳戶來託管資源，例如 Amazon Elastic Compute Cloud (Amazon EC2) 執行個體、Amazon Relational Database Service 服務 (Amazon RDS) 執行個體和 Amazon ElastiCache 執行個體。您可以透過包含資源的私有子網將函數附加到 VPC，讓 Lambda 函數存取 Amazon VPC 中託管的資源。遵循以下各節中的指示，使用 Lambda 主控台、 AWS Command Line Interface (AWS CLI) 或 AWS SAM將 Lambda 函數附加至 Amazon VPC。

> **Note**
>
> 每個 Lambda 函數都在由 Lambda 服務擁有和管理的 VPC 內執行。這些 VPC 由 Lambda 自動維護，客戶看不到。設定您的函數以存取 Amazon VPC 中的其他 AWS 資源並不會影響您函數在其中執行的 Lambda 管理 VPC。

章節

所需的 IAM 許可

若要將 Lambda 函數連接到您的 Amazon VPC AWS 帳戶，Lambda 需要建立和管理網路界面的許可，以便讓您的函數存取 VPC 中的資源。

Lambda 建立的網路介面稱為「超平面彈性網路介面」或「超平面 ENI」。若要進一步瞭解這些網路介面，請參閱the section called “瞭解超平面彈性網路介面 (ENI)”。

您可以通過將 AWS 託管策略附加到函數的執行角色AWSLambdaVPCAccessExecutionRole來為您的函數提供所需的權限。當您在 Lambda 主控台中建立新函數並將其連接至 VPC 時，Lambda 會自動為您新增此許可政策。

如果您想要建立自己的 IAM 許可政策，請確保新增以下所有許可：

- ec2：CreateNetwork接口
- ec2：DescribeNetwork接口-此操作僅適用於所有資源（`"Resource": "*"`）。
- ec2：DescribeSubnets
- ec2：DeleteNetwork接口 — 如果您沒有在執行角色中為DeleteNetwork接口指定資源 ID，則您的函數可能無法訪問 VPC。您可指定不重複的資源 ID，或納入所有資源 ID，例如 `"Resource": "arn:aws:ec2:us-west-2:123456789012:*/*"`。
- ec2：AssignPrivateIpAddresses
- ec2：UnassignPrivateIpAddresses

請注意，您函數的角色只需要這些權限即可創建網絡接口，而不是調用您的函數。當函數連接到 Amazon VPC 時，您仍然可以成功叫用函數，即使您從函數的執行角色中移除這些許可也是如此。

若要將您的函數連接到 VPC，Lambda 還需要使用您的 IAM 使用者角色驗證網路資源。確保您的使用者角色具有下列 IAM 許可：

- ec2：DescribeSecurity群組
- ec2：DescribeSubnets
- ec2：DescribeVpcs

> **Note**
>
> Lambda 服務會使用您授予函數執行角色的 Amazon EC2 許可，將您的函數連接到 VPC。但是，您也隱式地將這些權限授予函數的代碼。這表示您的函數程式碼能夠進行這些 Amazon EC2 API 呼叫。如需下列安全性最佳做法的建議，請參閱the section called “安全最佳實務”。

將 Lambda 函數連接到您的 Amazon VPC AWS 帳戶

使用 Lambda 主控台 AWS CLI 或 AWS SAM將您 AWS 帳戶 的函數附加到您的中的 Amazon VPC。如果您使用 AWS CLI 或 AWS SAM，或使用 Lambda 主控台將現有函數附加至 VPC，請確定函數的執行角色具有上一節所列的必要權限。

Lambda 函數無法透過專用執行個體租用直接連線到 VPC。若要連線到專用 VPC 中的資源，請透過預設租用將它對等連線到第二個 VPC。

Lambda console

在創建 Amazon VPC 時將函數附加到該功能

1. 開啟 Lambda 主控台的函數頁面，然後選擇 建立函數。
2. 在 Basic information (基本資訊) 下，對於 Function name (函數名稱)，為您的函數輸入名稱。
3. 透過執行下列動作來設定功能的 VPC 設定：

   a. 展開 Advanced settings (進階設定)。

   b. 選取 [啟用 VPC]，然後選取要附加功能的 VPC。

   c. (選擇性) 若要允許傳出 IPv6 流量，請選取允許雙堆疊子網路的 IPv6 流量。

   d. 選擇要為其建立網路介面的子網路和安全性群組。如果您選取允許雙堆疊子網路的 IPv6 流量，則所有選取的子網路均必須具有 IPv4 CIDR 區塊和 IPv6 CIDR 區塊。

   > **Note**
   >
   > 若要存取私有資源，請將您的函數連線到私有子網路。如果您的功能需要網際網路存取，請參閱the section called “VPC 功能的網際網路存取”。將函數連接到公有子網路並不會提供網際網路存取或公有 IP 位址給該函數。

4. 選擇建立函數。

將現有功能附加到 Amazon VPC

1. 開啟 Lambda 主控台的函數頁面，然後選取您的函數。
2. 選擇配置選項卡，然後選擇 VPC。
3. 選擇編輯。
4. 在 VPC 下，選取您要附加功能的 Amazon VPC。
5. (選擇性) 若要允許傳出 IPv6 流量，請選取允許雙堆疊子網路的 IPv6 流量。
6. 選擇要為其建立網路介面的子網路和安全性群組。如果您選取允許雙堆疊子網路的 IPv6 流量，則所有選取的子網路均必須具有 IPv4 CIDR 區塊和 IPv6 CIDR 區塊。

> **Note**
>
> 若要存取私有資源，請將您的函數連線到私有子網路。如果您的功能需要網際網路存取，請參閱the section called “VPC 功能的網際網路存取”。將函數連接到公有子網路並不會提供網際網路存取或公有 IP 位址給該函數。

7. 選擇儲存。

AWS CLI

在創建 Amazon VPC 時將函數附加到該功能

- 若要建立 Lambda 函數並將其附加至 VPC，請執行下列 CLI `create-function` 命令。

```
aws lambda create-function --function-name my-function \
--runtime nodejs20.x --handler index.js --zip-file fileb://function.zip \
--role arn:aws:iam::123456789012:role/lambda-role \
--vpc-config
 Ipv6AllowedForDualStack=true,SubnetIds=subnet-071f712345678e7c8,subnet-07fd123456788a03
```

指定您自己的子網路和安全群組，並`false`根據您的使用案例設`Ipv6AllowedForDualStack`定為`true`或。

將現有功能附加到 Amazon VPC

- 若要將現有函數附加至 VPC，請執行下列 CLI `update-function-configuration` 命令。

```
aws lambda update-function-configuration --function-name my-function \
--vpc-config Ipv6AllowedForDualStack=true,
 SubnetIds=subnet-071f712345678e7c8,subnet-07fd123456788a036,SecurityGroupIds=sg-0859123
```

從 VPC 取消附加功能

- 若要從 VPC 取消附加功能，請使用 VPC 子網路和安全群組的空白清單執行下列 `update-function-configuration` CLI 命令。

```
aws lambda update-function-configuration --function-name my-function \
```

```
--vpc-config SubnetIds=[],SecurityGroupIds=[]
```

AWS SAM

將您的功能附加到 VPC

- 若要將 Lambda 函數附加至 Amazon VPC，請將VpcConfig屬性新增至函數定義，如下列範例範本所示。如需有關此屬性的詳細資訊，請參閱AWSAWS CloudFormation 使用指南 VpcConfig中的:: Lambda:: Function (AWS SAM VpcConfig屬性會直接傳遞至 AWS CloudFormation AWS::Lambda::Function資源的VpcConfig屬性)。

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31

Resources:
  MyFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: ./lambda_function/
      Handler: lambda_function.handler
      Runtime: python3.12
      VpcConfig:
        SecurityGroupIds:
          - !Ref MySecurityGroup
        SubnetIds:
          - !Ref MySubnet1
          - !Ref MySubnet2
      Policies:
        - AWSLambdaVPCAccessExecutionRole

  MySecurityGroup:
    Type: AWS::EC2::SecurityGroup
    Properties:
      GroupDescription: Security group for Lambda function
      VpcId: !Ref MyVPC

  MySubnet1:
    Type: AWS::EC2::Subnet
    Properties:
      VpcId: !Ref MyVPC
      CidrBlock: 10.0.1.0/24
```

```
  MySubnet2:
    Type: AWS::EC2::Subnet
    Properties:
      VpcId: !Ref MyVPC
      CidrBlock: 10.0.2.0/24

  MyVPC:
    Type: AWS::EC2::VPC
    Properties:
      CidrBlock: 10.0.0.0/16
```

如需有關在中設定 VPC 的詳細資訊 AWS SAM，請參閱AWS《使用者指南》中的:: EC2:: VPC。AWS CloudFormation

連接至 VPC 時的網際網路存取

根據預設，Lambda 函數可以存取公用網際網路。將函數連接到 VPC 時，它只能存取該 VPC 中可用的資源。若要讓您的功能存取網際網路，您還需要將 VPC 設定為可存取網際網路。如需進一步了解，請參閱the section called “VPC 功能的網際網路存取”。

將 Lambda 與 Amazon VPC 搭配使用的最佳實務

若要確保您的 Lambda VPC 組態符合最佳實務準則，請遵循以下各節中的建議。

安全最佳實務

若要將 Lambda 函數連接到 VPC，您需要為函數的執行角色提供許多 Amazon EC2 許可。需要這些權限才能建立函數用來存取 VPC 中資源的網路介面。但是，這些權限也被隱含授予函數的代碼。這表示您的函數程式碼具有進行這些 Amazon EC2 API 呼叫的權限。

若要遵循最低權限存取的原則，請將拒絕原則 (如下列範例) 新增至函數的執行角色。此政策可防止您的函數呼叫 Lambda 服務用來將函數連接到 VPC 的 Amazon EC2 API。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": [
                "ec2:CreateNetworkInterface",
```

```
                "ec2:DeleteNetworkInterface",
                "ec2:DescribeNetworkInterfaces",
                "ec2:DetachNetworkInterface",
                "ec2:AssignPrivateIpAddresses",
                "ec2:UnassignPrivateIpAddresses",
            ],
            "Resource": [ "*" ],
            "Condition": {
                "ArnEquals": {
                    "lambda:SourceFunctionArn": [
                        "arn:aws:lambda:us-west-2:123456789012:function:my_function"
                    ]
                }
            }
        }
    ]
}
```

AWS 提供安全群組和網路存取控制清單 (ACL)，以提高 VPC 的安全性。安全群組控制資源的傳入與傳出流量，網路 ACL 則是控制子網的傳入與傳出流量。安全群組可為大多數子網路提供足夠的存取控制。如果您想讓 VPC 多一層安全，可以使用網路 ACL。如需使用 Amazon VPC 時安全最佳實務的一般指導方針，請參閱 Amazon V irtual Private Cloud 使用者指南中的 VPC 安全最佳實務。

效能最佳實務

將函數連接到 VPC 時，Lambda 會檢查是否有可用的網路資源 (超平面 ENI) 可用於連線。超平面 ENI 與安全群組和 VPC 子網路的特定組合相關聯。如果您已經將一個函數附加到 VPC，則在連接另一個函數時指定相同的子網路和安全群組，這表示 Lambda 可以共用網路資源，而不需要建立新的 Hyperplane ENI。如需有關超平面 ENI 及其生命週期的詳細資訊，請參閱。the section called “瞭解超平面彈性網路介面 (ENI)”

瞭解超平面彈性網路介面 (ENI)

超平面 ENI 是一種受管資源，可作為 Lambda 函數與您希望函數連接的資源之間的網路介面。當您將函數連接到 VPC 時，Lambda 服務會自動建立和管理這些 ENI。

您無法直接看到超平面 ENI，而且您不需要設定或管理它們。但是，了解它們的工作方式可以幫助您在將其附加到 VPC 時了解函數的行為。

當您第一次使用特定子網路和安全群組組合將函數附加至 VPC 時，Lambda 會建立超平面 ENI。您帳戶中使用相同子網路和安全群組組合的其他功能也可以使用此 ENI。Lambda 會盡可能重複使用現有的

ENI 來最佳化資源利用率，並將新 ENI 的建立降到最低。每個超平面 ENI 最多支援 65,000 個連線/連接埠。如果連線數目超過此限制，Lambda 會根據網路流量和並行需求自動調整 ENI 的數量。

對於新函數，當 Lambda 正在建立超平面 ENI 時，您的函數仍處於擱置中狀態，而且您無法呼叫它。只有在超平面 ENI 準備就緒時，您的函數才會轉換到作用中狀態，這可能需要幾分鐘的時間。對於現有函數，您無法執行以該函數為目標的其他操作，例如創建版本或更新函數的代碼，但是您可以繼續調用該函數的先前版本。

> **Note**
>
> 如果 Lambda 函數閒置 30 天，Lambda 會回收任何未使用的超平面 ENI，並將函數狀態設定為閒置。下一次叫用嘗試會失敗，且函數會重新進入擱置中狀態，直到 Lambda 完成超平面 ENI 的建立或配置為止。如需 Lambda 函數狀態的詳細資訊，請參閱the section called “函數狀態”。

如需超平面 ENI 生命週期的詳細資訊，請參閱。the section called “Lambda Hyperplane ENI”

使用 IAM 條件金鑰進行 VPC 設定

您可以針對 VPC 設定使用 Lambda 特定條件金鑰，為您的 Lambda 函數提供額外的許可控制。例如，您可以請求組織中的所有功能都連線到 VPC。您也可以指定函數的使用者可以和不可以使用的子網路和安全性群組。

Lambda 支援 IAM 政策中的以下條件金鑰：

- lambda：VpcIds— 允許或拒絕一個或多個 VPC。
- lambda: SubnetIds — 允許或拒絕一個或多個子網路。
- lambda：SecurityGroupID — 允許或拒絕一個或多個安全群組。

Lambda API 作業CreateFunction和UpdateFunction組態支援這些條件金鑰。如需有關在 IAM 政策中使用條件金鑰的詳細資訊，請參閱 IAM 使用者指南中的 IAM JSON 政策元素：條件。

> **Tip**
>
> 如果您的函數已經包含來自先前 API 請求的 VPC 組態，則可以在沒有 VPC 組態的情況下傳送 `UpdateFunctionConfiguration` 請求。

具有 VPC 設定條件金鑰的範例政策

下列範例示範如何使用條件金鑰進行 VPC 設定。建立具有所需限制的政策陳述式之後，請附加目標 使用者或角色的政策陳述式。

確保使用者僅部署與 VPC 連線的函數

若要確保所有使用者僅部署與 VPC 連線的函數，您可以拒絕不含有效 VPC ID 的函數建立和更新操作。

請注意，VPC ID 不是 `CreateFunction` 或 `UpdateFunctionConfiguration` 請求的輸入參數。Lambda 會根據子網路和安全群組參數擷取 VPC ID 值。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "EnforceVPCFunction",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Deny",
      "Resource": "*",
      "Condition": {
        "Null": {
           "lambda:VpcIds": "true"
        }
      }
    }
  ]
}
```

拒絕使用者存取特定 VPC、子網路或安全群組

若要拒絕使用者存取特定 VPC，請使用 `StringEquals` 來檢查 `lambda:VpcIds` 條件的值。下列範例會拒絕使用者存取 `vpc-1` 和 `vpc-2`。

```
{
  "Version": "2012-10-17",
  "Statement": [
```

```
    {
      "Sid": "EnforceOutOfVPC",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Deny",
      "Resource": "*",
      "Condition": {
        "StringEquals": {
            "lambda:VpcIds": ["vpc-1", "vpc-2"]
        }
      }
    }
```

若要拒絕使用者存取特定子網路，請使用 StringEquals 來檢查 lambda:SubnetIds 條件的值。下列範例會拒絕使用者存取 subnet-1 和 subnet-2。

```
{
      "Sid": "EnforceOutOfSubnet",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Deny",
      "Resource": "*",
      "Condition": {
        "ForAnyValue:StringEquals": {
            "lambda:SubnetIds": ["subnet-1", "subnet-2"]
        }
      }
    }
```

若要拒絕使用者存取特定安全性群組，請使用 StringEquals 來檢查 lambda:SecurityGroupIds 條件的值。下列範例會拒絕使用者存取 sg-1 和 sg-2。

```
{
      "Sid": "EnforceOutOfSecurityGroups",
      "Action": [
          "lambda:CreateFunction",
```

```
            "lambda:UpdateFunctionConfiguration"
        ],
        "Effect": "Deny",
        "Resource": "*",
        "Condition": {
          "ForAnyValue:StringEquals": {
              "lambda:SecurityGroupIds": ["sg-1", "sg-2"]
          }
        }
      }
  ]
}
```

允許使用者使用特定 VPC 設定建立和更新功能

若要允許使用者存取特定 VPC，請使用 StringEquals 來檢查 lambda:VpcIds 條件的值。下列範例可讓使用者存取 vpc-1 和 vpc-2。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "EnforceStayInSpecificVpc",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Allow",
      "Resource": "*",
      "Condition": {
        "StringEquals": {
            "lambda:VpcIds": ["vpc-1", "vpc-2"]
        }
      }
    }
```

若要允許使用者存取特定子網路，請使用 StringEquals 來檢查 lambda:SubnetIds 條件的值。下列範例可讓使用者存取 subnet-1 和 subnet-2。

```
{
```

```
      "Sid": "EnforceStayInSpecificSubnets",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Allow",
      "Resource": "*",
      "Condition": {
        "ForAllValues:StringEquals": {
            "lambda:SubnetIds": ["subnet-1", "subnet-2"]
        }
      }
    }
```

若要允許使用者存取特定安全性群組，請使用 `StringEquals` 來檢查 `lambda:SecurityGroupIds` 條件的值。下列範例可讓使用者存取 `sg-1` 和 `sg-2`。

```
{
      "Sid": "EnforceStayInSpecificSecurityGroup",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Allow",
      "Resource": "*",
      "Condition": {
        "ForAllValues:StringEquals": {
            "lambda:SecurityGroupIds": ["sg-1", "sg-2"]
        }
      }
    }
  ]
}
```

VPC 教學課程

在以下教學中，您會將 Lambda 函數連線至 VPC 中的資源。

- 教學課程：使用 Lambda 函數來存取 Amazon VPC 中的 Amazon RDS
- 教學課程：設定 Lambda 函數以 ElastiCache 在 Amazon VPC 中存取 Amazon

為虛擬私人電腦連線的 Lambda 函數啟用網際網路

根據預設，Lambda 函數會在可存取網際網路的 Lambda 管理 VPC 中執行。若要存取帳戶中 VPC 中的資源，您可以將 VPC 組態新增至函數。這會將功能限制在該 VPC 內的資源，除非 VPC 可以存取網際網路。本頁說明如何提供與 VPC 連線之 Lambda 函數的網際網路存取權。

我還沒有 VPC

建立 VPC

「建立 VPC」工作流程會建立 Lambda 函數從私有子網路存取公用網際網路所需的所有 VPC 資源，包括子網路、NAT 閘道、網際網路閘道和路由表項目。

若要建立 VPC

1. 前往 https://console.aws.amazon.com/vpc/ 開啟 Amazon VPC 主控台。
2. 在儀表板上，選擇建立 VPC。
3. 針對 Resources to create (建立資源)，選擇 VPC and more (VPC 等)。
4. 設定 VPC
   a. 針對自動產生名稱標籤，輸入 VPC 的名稱。
   b. 對於 IPv4 CIDR 區塊，您可以保留預設建議，或者您也可以輸入應用程式或網路所需的 CIDR 區塊。
   c. 如果您的應用程式使用 IPv6 地址進行通訊，請選擇 IPv6 CIDR 區塊 > Amazon 提供的 IPv6 CIDR 區塊。
5. 設定子網路
   a. 針對可用區域數量，選擇 2。我們建議至少兩個 AZ 以獲得高可用性。
   b. 針對公用子網路數量，選擇 2。
   c. 在 Number of private subnet (私有子網路數量) 中，選擇 2。
   d. 您可以保留公有子網路的預設 CIDR 區塊，或者您也可以展開自訂子網路 CIDR 區塊並輸入 CIDR 區塊。如需詳細資訊，請參閱子網路 CIDR 區塊。
6. 若為 NAT 閘道，請選擇每個 AZ 1 個以提高彈性。
7. 如果您選擇包含 IPv6 CIDR 區塊，請針對僅輸出網際網路閘道，選擇是。
8. 對於 VPC 端點，請保留預設值 (S3 閘道)。此選項無需支付任何費用。如需詳細資訊，請參閱 Amazon S3 適用的 VPC 人雲端節點類型。

9. 對於 DNS 選項，請保留預設設定。
10. 選擇建立 VPC。

設定 Lambda 函數

若要在建立函數時設定 VPC

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇 建立函數。
3. 在 Basic information (基本資訊) 下，對於 Function name (函數名稱)，為您的函數輸入名稱。
4. 展開 Advanced settings (進階設定)。
5. 選取 [啟用 VPC]，然後選擇一個 VPC。
6. (選擇性) 若要允許傳出 IPv6 流量，請選取允許雙堆疊子網路的 IPv6 流量。
7. 對於子網路，請選取所有私人子網路。私有子網路可透過 NAT 閘道存取網際網路。將函數連接到公共子網不會給它訪問互聯網。

> **Note**
>
> 如果您選取允許雙堆疊子網路的 IPv6 流量，則所有選取的子網路均必須具有 IPv4 CIDR 區塊和 IPv6 CIDR 區塊。

8. 對於安全性群組，請選取允許輸出流量的安全性群組。
9. 選擇建立函數。

Lambda 會透過AWSLambdaVPCAccessExecutionRole AWS 受管政策自動建立執行角色。只有為 VPC 組態建立彈性網路介面時，才需要此原則中的權限，而不是呼叫您的函數。若要套用最低權限權限，您可以在建立函數和 VPC 組態之後，從執行角色移除AWSLambdaVPCAccessExecutionRole原則。如需詳細資訊，請參閱 所需的 IAM 許可。

若要為現有函數設定 VPC

若要將 VPC 組態新增至現有功能，函數的執行角色必須具有建立和管理彈性網路介面的權限。受AWSLambdaVPCAccessExecutionRole AWS 管理的策略包括必要的權限。若要套用最低權限權限，您可以在建立 VPC 組態之後從執行角色移除AWSLambdaVPCAccessExecutionRole原則。

1. 開啟 Lambda 主控台中的 函數頁面。

2. 選擇一個函數。
3. 選擇配置選項卡，然後選擇 VPC。
4. 在 VPC 下，選擇 Edit (編輯)。
5. 選取 VPC。
6. (選擇性) 若要允許傳出 IPv6 流量，請選取允許雙堆疊子網路的 IPv6 流量。
7. 對於子網路，請選取所有私人子網路。私有子網路可透過 NAT 閘道存取網際網路。將函數連接到公共子網不會給它訪問互聯網。

> **Note**
>
> 如果您選取允許雙堆疊子網路的 IPv6 流量，則所有選取的子網路均必須具有 IPv4 CIDR 區塊和 IPv6 CIDR 區塊。

8. 對於安全性群組，請選取允許輸出流量的安全性群組。
9. 選擇儲存。

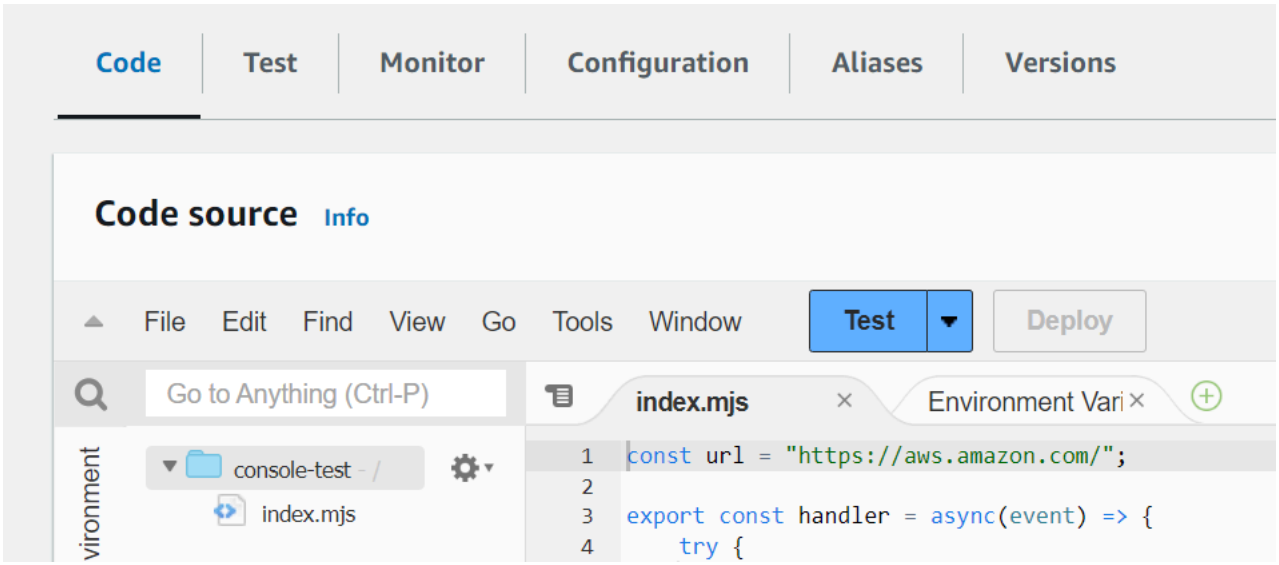
測試函數

使用下列範例程式碼，確認連接 VPC 的功能可以連線到公用網際網路。如果成功，則代碼返回一個`200`狀態碼。如果不成功，則函數逾時。

Node.js

此範例使用`fetch`，在以後的執行階段中`nodejs18.x`均可使用。

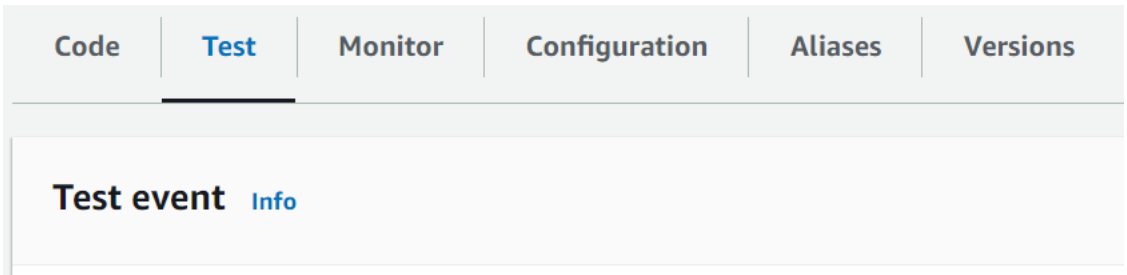
1. 在 Lambda 主控台的「程式碼原始碼」窗格中，將下列程式碼貼到 index.m js 檔案中。該函數向公共端點發出 HTTP GET 請求，並返回 HTTP 響應代碼以測試該函數是否可以訪問公共互聯網。



Example -使用異步/等待的HTTP請求

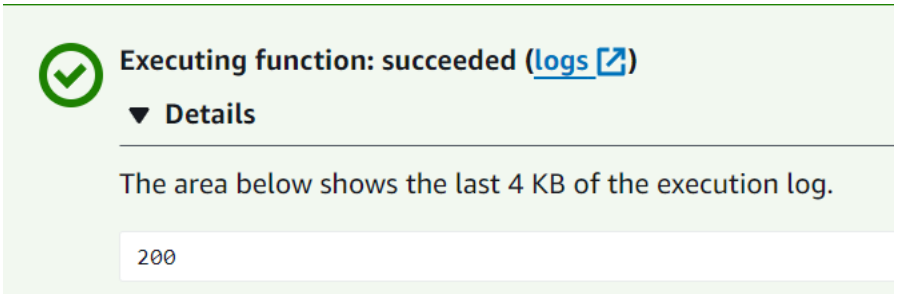
```
const url = "https://aws.amazon.com/";

export const handler = async(event) => {
    try {
        // fetch is available with Node.js 18 and later runtimes
        const res = await fetch(url);
        console.info("status", res.status);
        return res.status;
    }
    catch (e) {
        console.error(e);
        return 500;
    }
};
```

2. 選擇部署。
3. 選擇測試標籤。



4. 選擇 測試。

5. 該函數返回一個200狀態碼。這意味著該功能具有出站互聯網訪問。

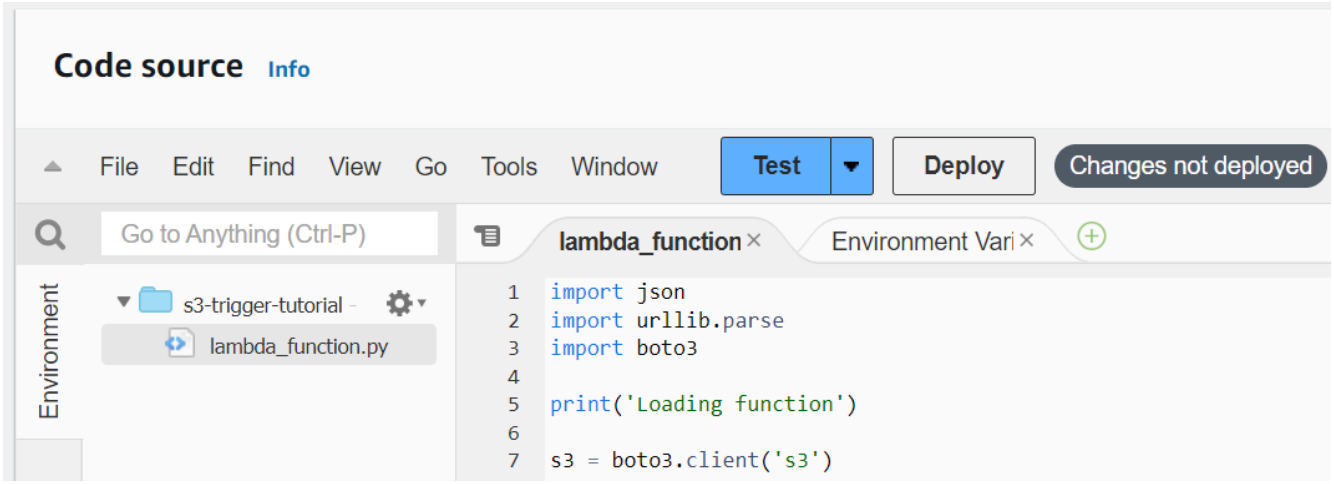


如果該功能無法訪問公共互聯網，則會收到如下錯誤消息：

```
{
  "errorMessage": "2024-04-11T17:22:20.857Z abe12jlc-640a-8157-0249-9be825c2y110
 Task timed out after 3.01 seconds"
}
```

Python

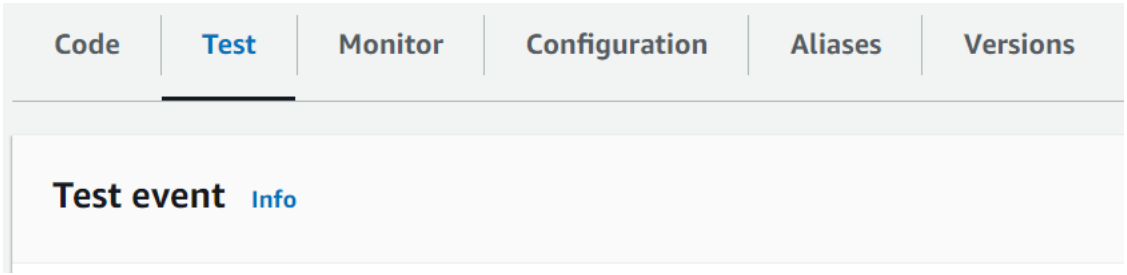
1. 在 Lambda 主控台的「程式碼原始程式碼」窗格中，將下列程式碼貼到 lambda_function.py 檔案中。該函數向公共端點發出 HTTP GET 請求，並返回 HTTP 響應代碼以測試該函數是否可以訪問公共互聯網。



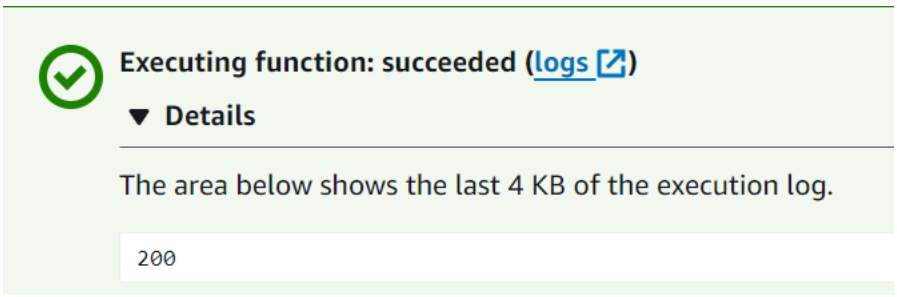

```
import urllib.request

def lambda_handler(event, context):
    try:
        response = urllib.request.urlopen('https://aws.amazon.com')
        status_code = response.getcode()
```

```
        print('Response Code:', status_code)
        return status_code
    except Exception as e:
        print('Error:', e)
        raise e
```

2. 選擇部署。
3. 選擇測試標籤。



4. 選擇 測試。
5. 該函數返回一個200狀態碼。這意味著該功能具有出站互聯網訪問。



如果該功能無法訪問公共互聯網，則會收到如下錯誤消息：

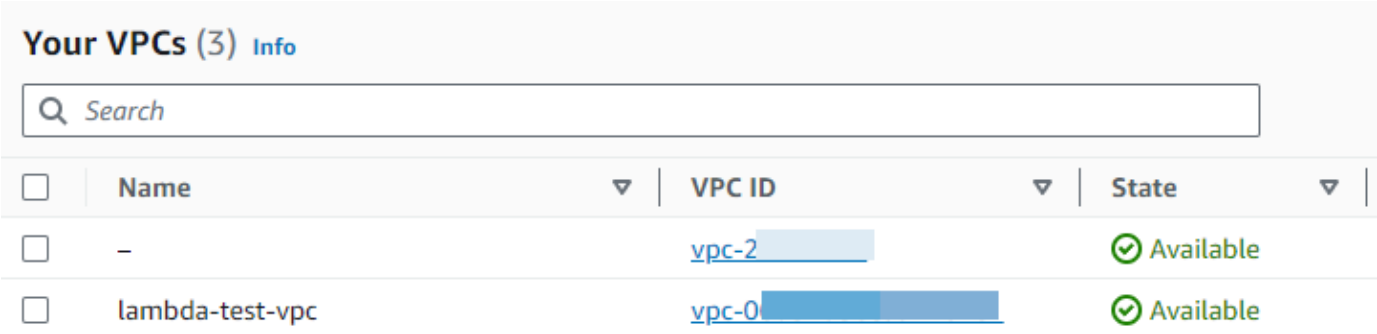
```
{
  "errorMessage": "2024-04-11T17:22:20.857Z abe12jlc-640a-8157-0249-9be825c2y110
 Task timed out after 3.01 seconds"
}
```

我已經有一個 VPC

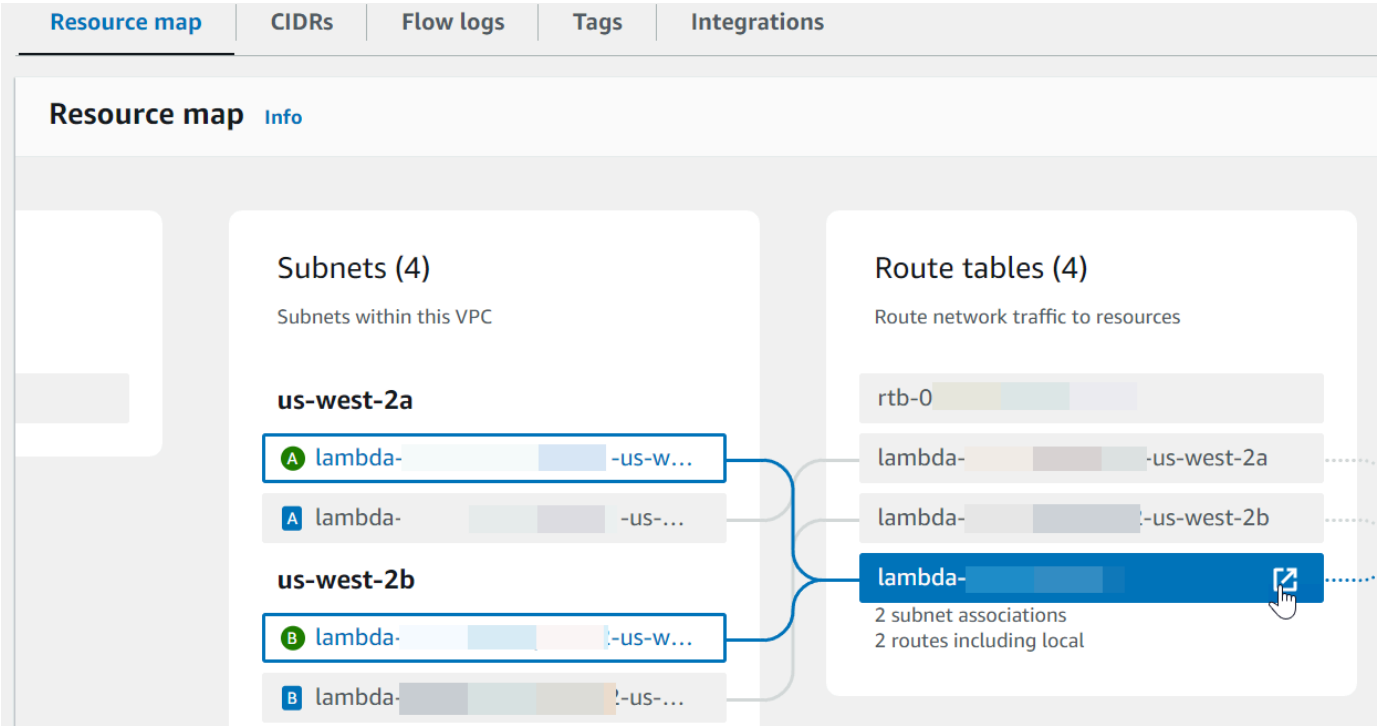
如果您已經擁有 VPC，但需要為 Lambda 函數設定公用網際網路存取權限，請按照下列步驟操作。此程序假設您的 VPC 至少有兩個子網路。如果您沒有兩個子網路，請參閱 Amazon VPC 使用者指南中的建立子網路。

驗證路由表配置

1. 前往 https://console.aws.amazon.com/vpc/ 開啟 Amazon VPC 主控台。
2. 選擇 VPC 識別碼。



3. 向下捲動至「資源對應」區段。請注意路由表對映。開啟對應至子網路的每個路由表。

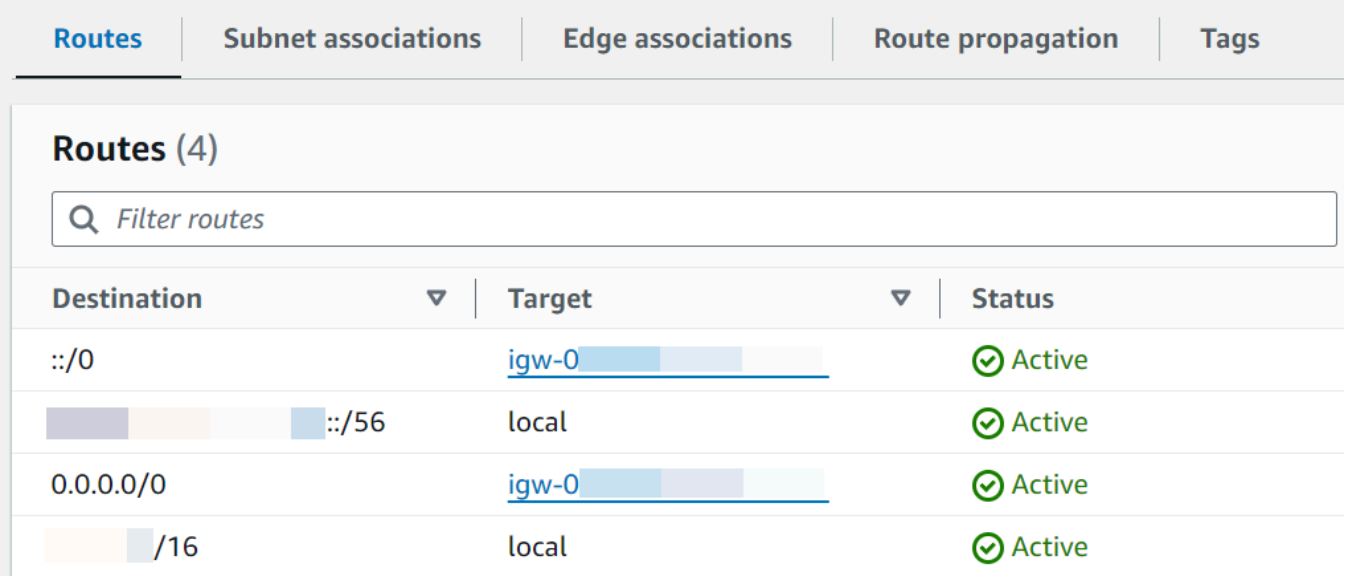


4. 向下捲動至「佈線」頁籤。檢閱路由以確定下列其中一項是否成立。這些需求中的每一個都必須由單獨的路由表來滿足。

   - 網際網路繫結流量 (`0.0.0.0/0`適用於 IPv4，`::/0`適用於 IPv6) 會路由至網際網路閘道 ()。`igw-xxxxxxxxxx`這表示與路由表關聯的子網路是公用子網路。

> **Note**
>
> 如果您的子網路沒有 IPv6 CIDR 區塊，您只會看到 IPv4 路由 () `0.0.0.0/0`。

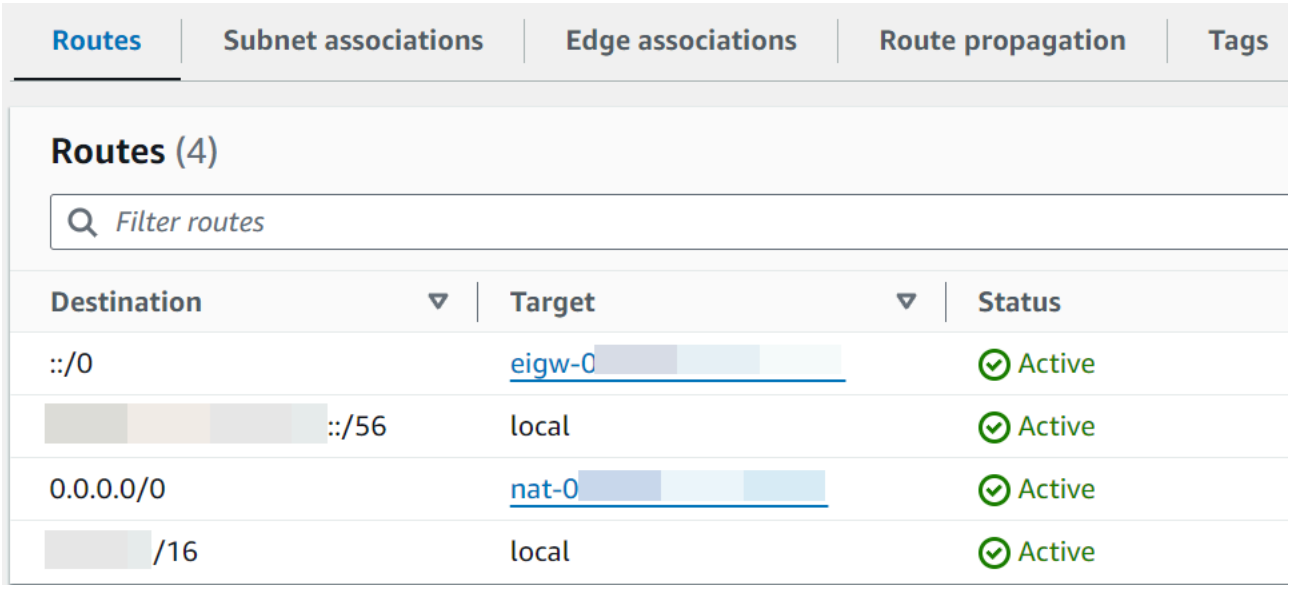
Example 公共子網路路由表



- IPv4 (`0.0.0.0/0`) 的網際網路繫結流量會路由至與公用子網路相關聯的 NAT 閘道 (`nat-xxxxxxxxxx`)。這表示子網路是可透過 NAT 閘道存取網際網路的私有子網路。

> **Note**
>
> 如果您的子網路具有 IPv6 CIDR 區塊，路由表也必須將網際網路繫結的 IPv6 流量 (`::/0`) 路由至僅限輸出的網際網路閘道 ()。`eigw-xxxxxxxxxx`如果您的子網路沒有 IPv6 CIDR 區塊，您只會看到 IPv4 路由 () `0.0.0.0/0`。

Example 私有子網路路由表



5. 重複上一個步驟，直到您檢閱了與 VPC 中子網路相關聯的每個路由表，並確認您有一個包含網際網路閘道的路由表以及具有 NAT 閘道的路由表。

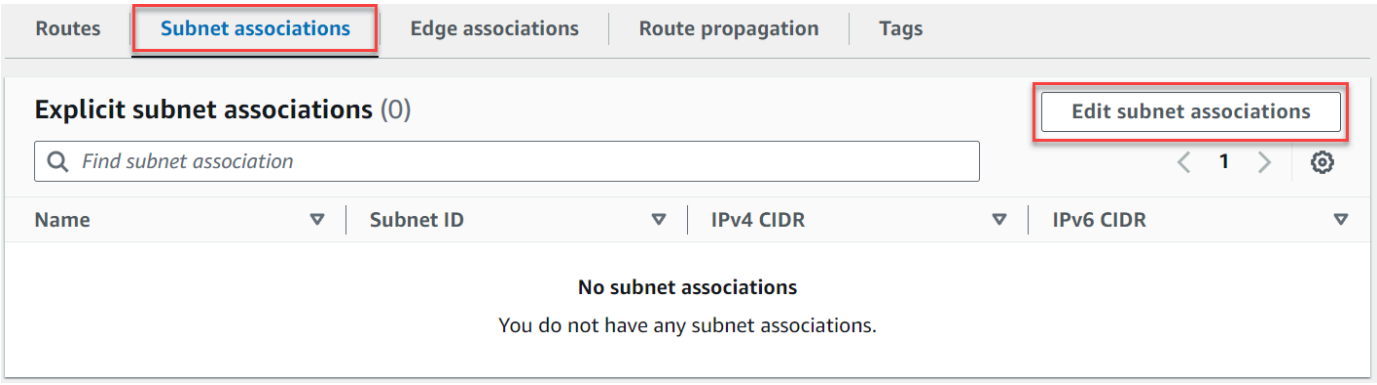
   如果您沒有兩個路由表，一個是通往網際網路閘道的路由，另一個路由到 NAT 閘道，請依照下列步驟建立遺失的資源和路由表項目。

建立路由表

請依照下列步驟建立路由表，並將其與子網路產生關聯。

使用 Amazon VPC 主控台建立自訂路由表

1. 在 https://console.aws.amazon.com/vpc/ 開啟 Amazon VPC 主控台。
2. 在導覽窗格中，選擇 Route tables (路由表)。
3. 選擇 Create route table (建立路由表)。
4. (選用) 針對 Name (名稱)，輸入路由表的名稱。
5. 在 VPC 中，選擇您的 VPC。
6. (選用) 若要新增標籤，請選擇 Add new tag (新增標籤)，然後輸入標籤鍵和標籤值。
7. 選擇 Create route table (建立路由表)。
8. 在 Subnet associations (子網關聯) 標籤上，選擇 Edit subnet associations (編輯子網關聯)。



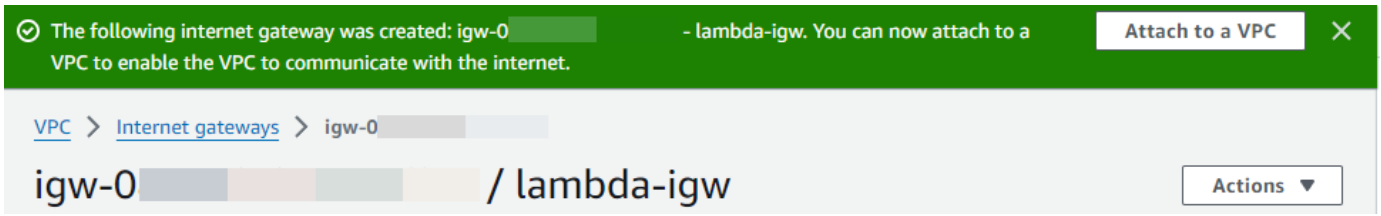
9. 選取子網路的核取方塊以和路由表建立關聯。
10. 選擇 Save associations (儲存關聯)。

建立網際網路閘道

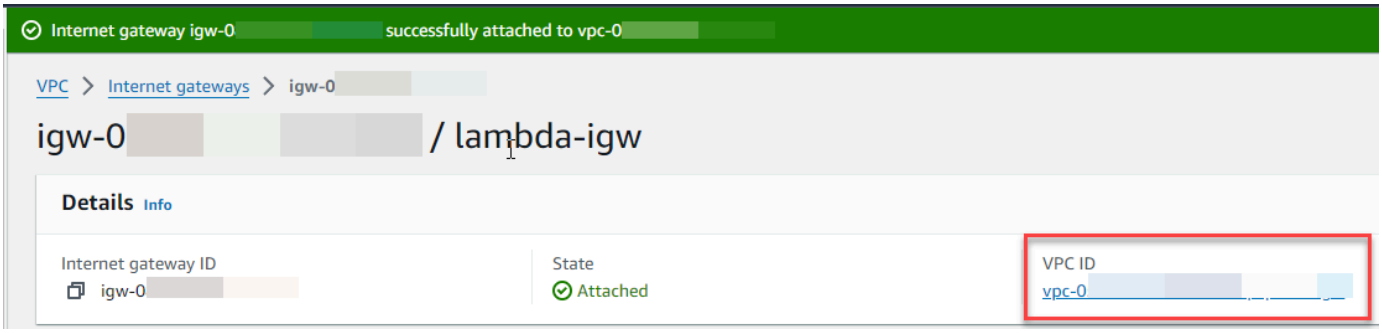
請依照下列步驟建立網際網路閘道、將其附加至您的 VPC，並將其新增至公有子網路的路由表。

建立網際網路閘道

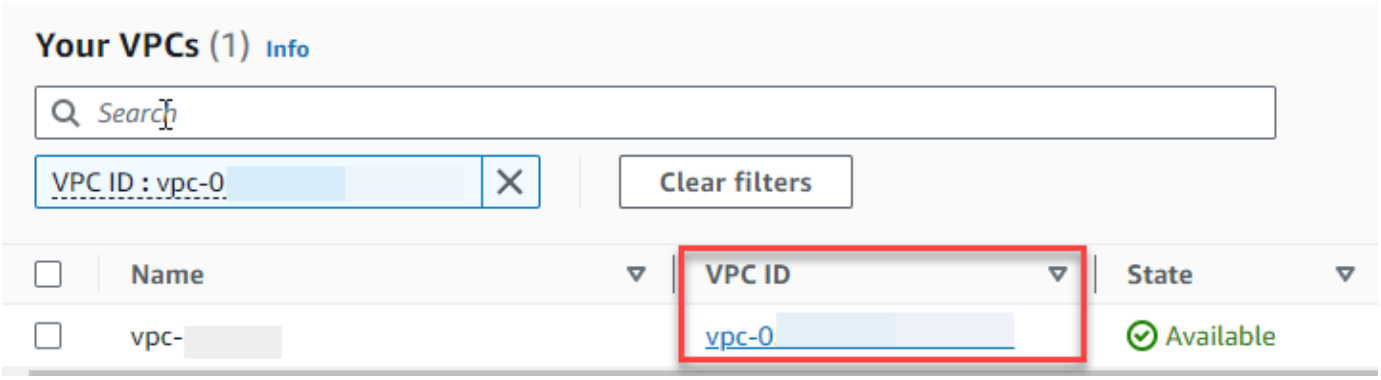
1. 在 https://console.aws.amazon.com/vpc/ 開啟 Amazon VPC 主控台。
2. 在導覽窗格中，選擇 Internet gateways (網際網路閘道)。
3. 選擇建立網際網路閘道。
4. (可選) 輸入網際網路閘道的名稱。
5. (選用) 若要新增標籤，請選擇 Add new tag (新增標籤)，然後輸入標籤金鑰和值。
6. 選擇建立網際網路閘道。
7. 從畫面頂端的橫幅中選擇 [連接至 VPC]，選取可用的 VPC，然後選擇 [連接網際網路閘道]。



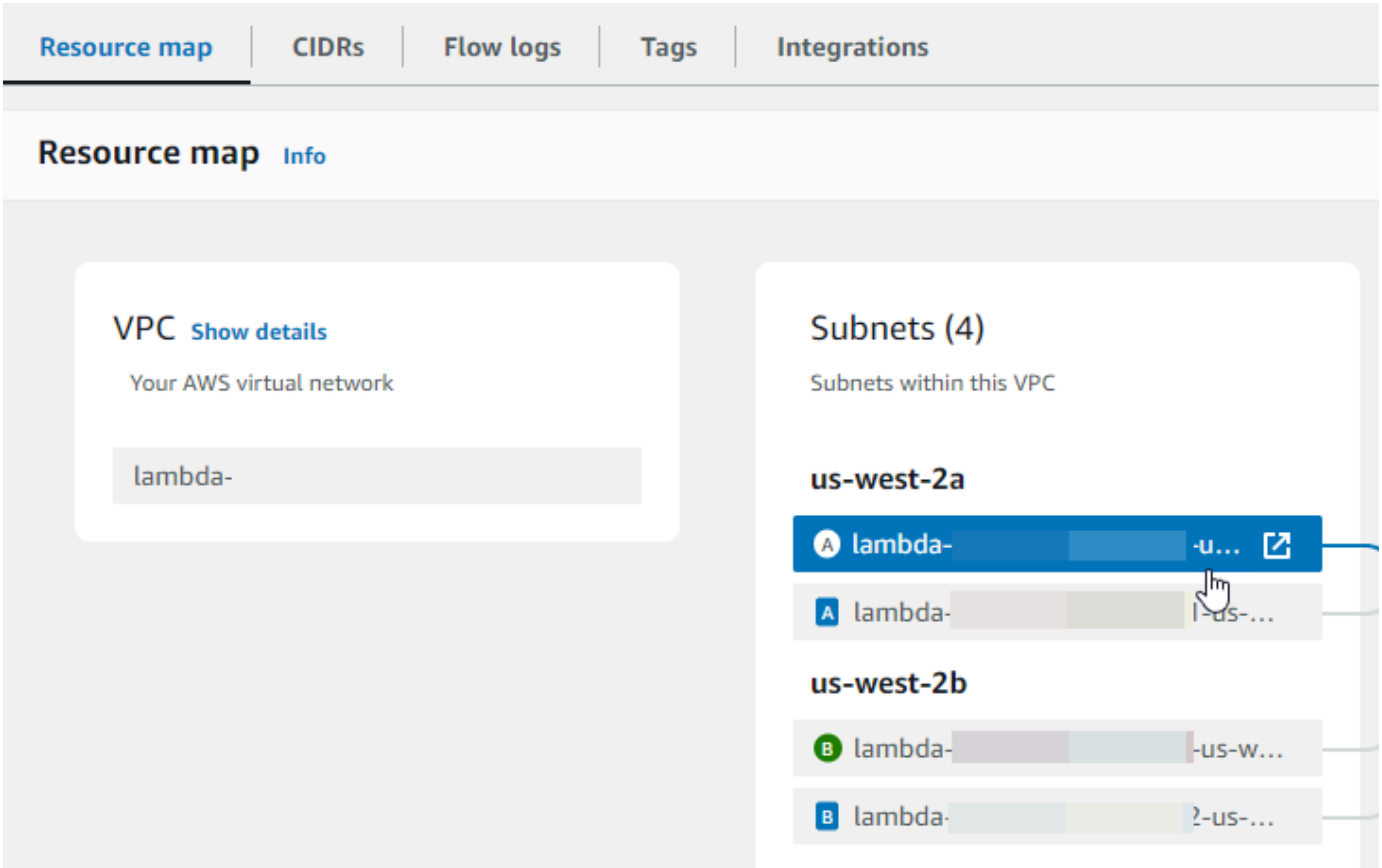
8. 選擇 VPC 識別碼。



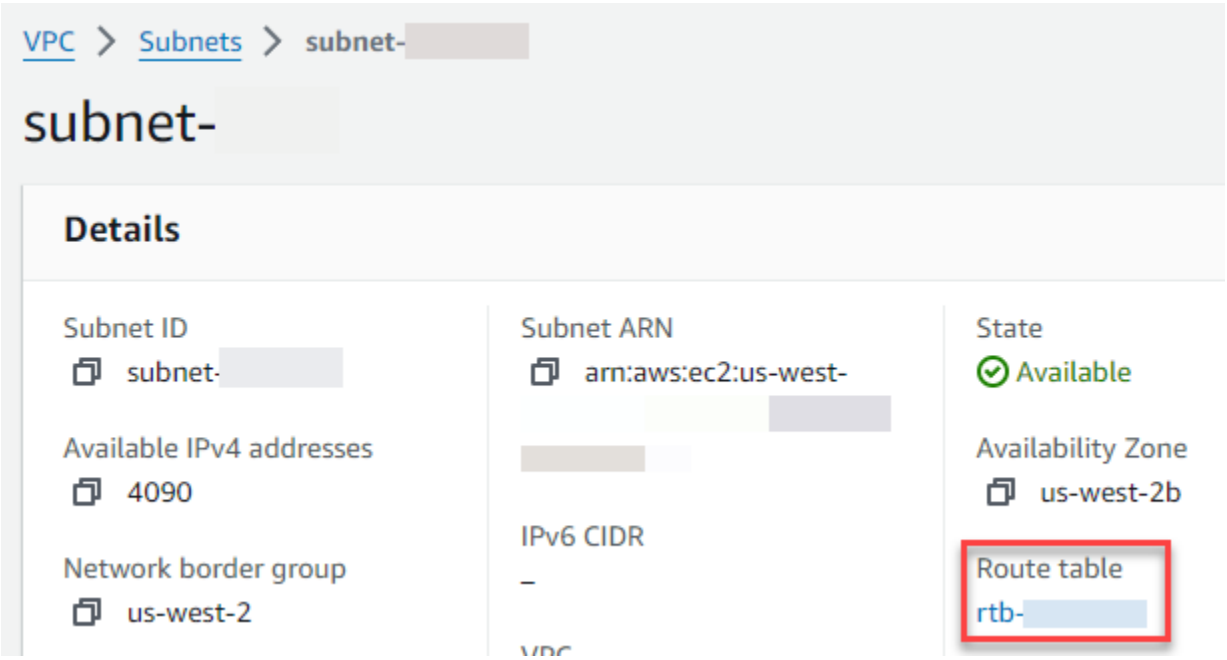
9. 再次選擇 VPC ID 以開啟 VPC 詳細資訊頁面。



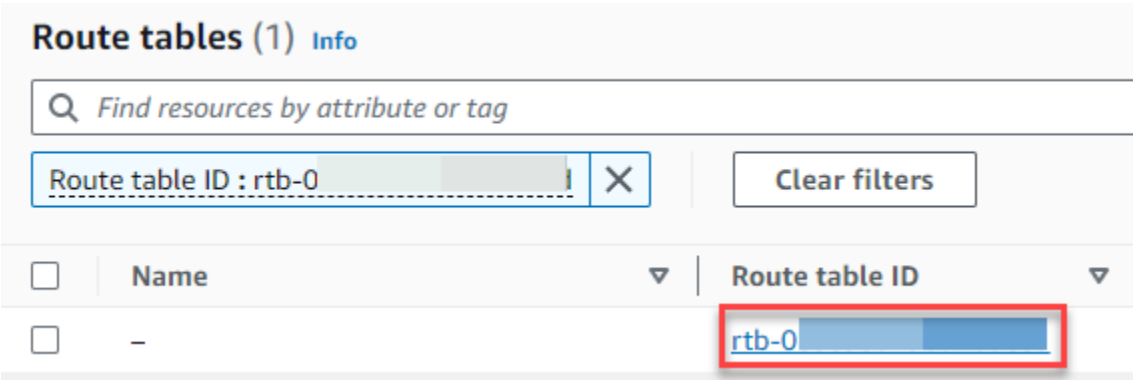
10. 向下捲動至 [資源對應] 區段，然後選擇子網路。子網路詳細資訊會顯示在新索引標籤中。



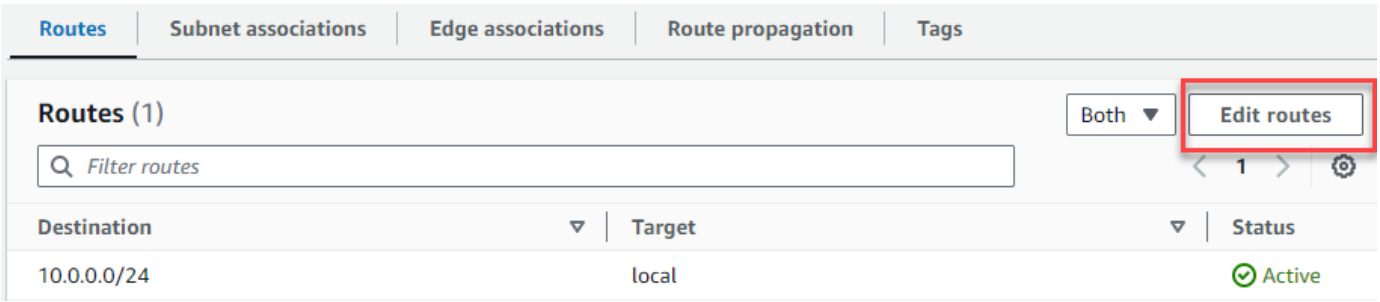
11. 選擇路由表下的鏈接。



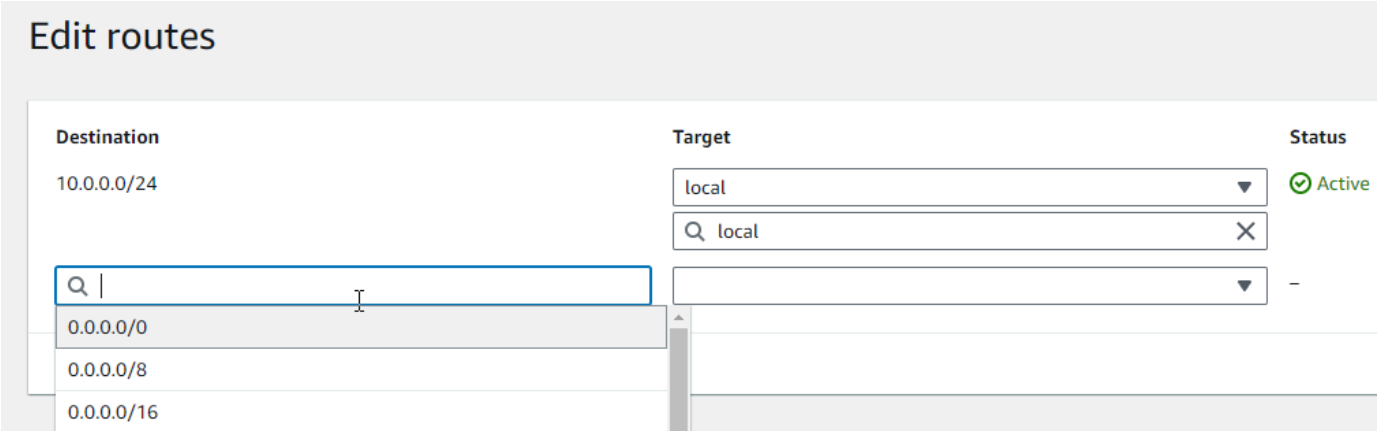
12. 選擇「路由表格 ID」以開啟路由表格詳細資訊頁面。



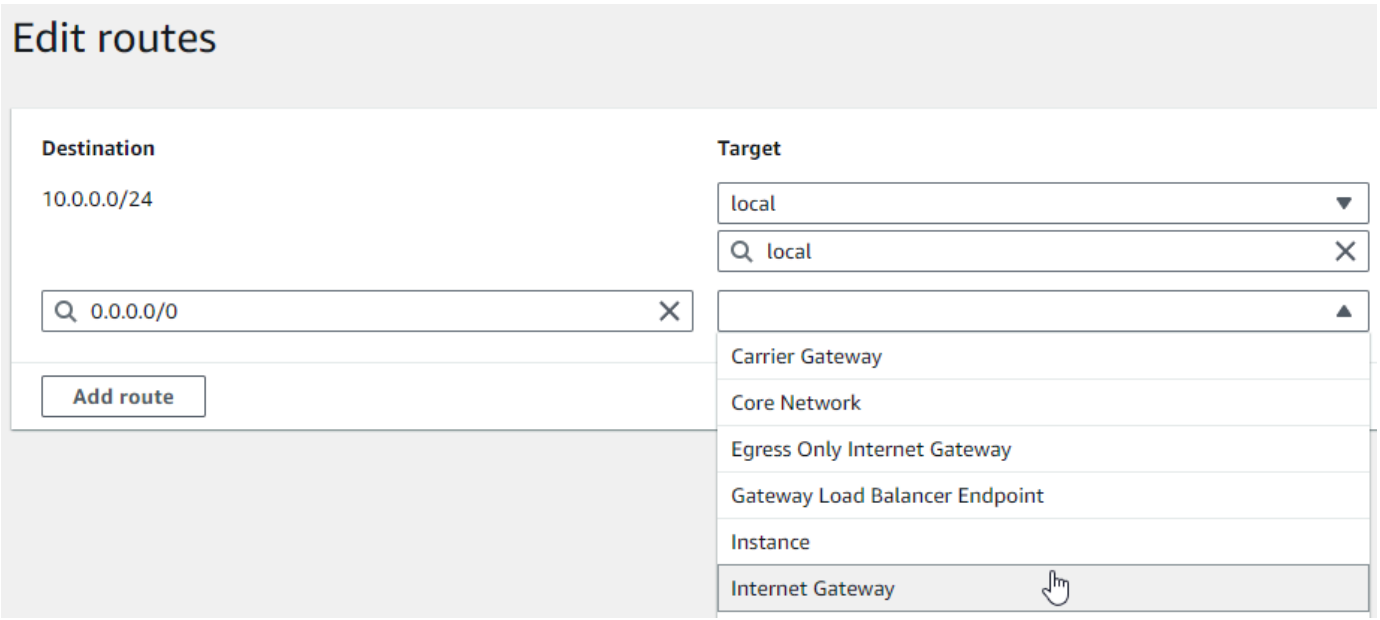
13. 在路線下，選擇編輯路線。



14. 選擇 [新增路線]，然後`0.0.0.0/0`在 [目的地] 方塊中輸入。



15. 針對 Target，選取網際網路閘道，然後選擇您先前建立的網際網路閘道。如果您的子網路具有 IPv6 CIDR 區塊，您也必須為`::/0`相同的網際網路閘道新增路由。



16. 選擇儲存變更。

建立 NAT 閘道

請依照下列步驟建立 NAT 閘道、將其與公用子網路產生關聯，然後將其新增至私有子網路的路由表。

建立 NAT 閘道並將其與公用子網路產生關聯

1. 在導覽窗格中，選擇 NAT 閘道。
2. 選擇建立 NAT 閘道。
3. (選擇性) 輸入 NAT 閘道的名稱。

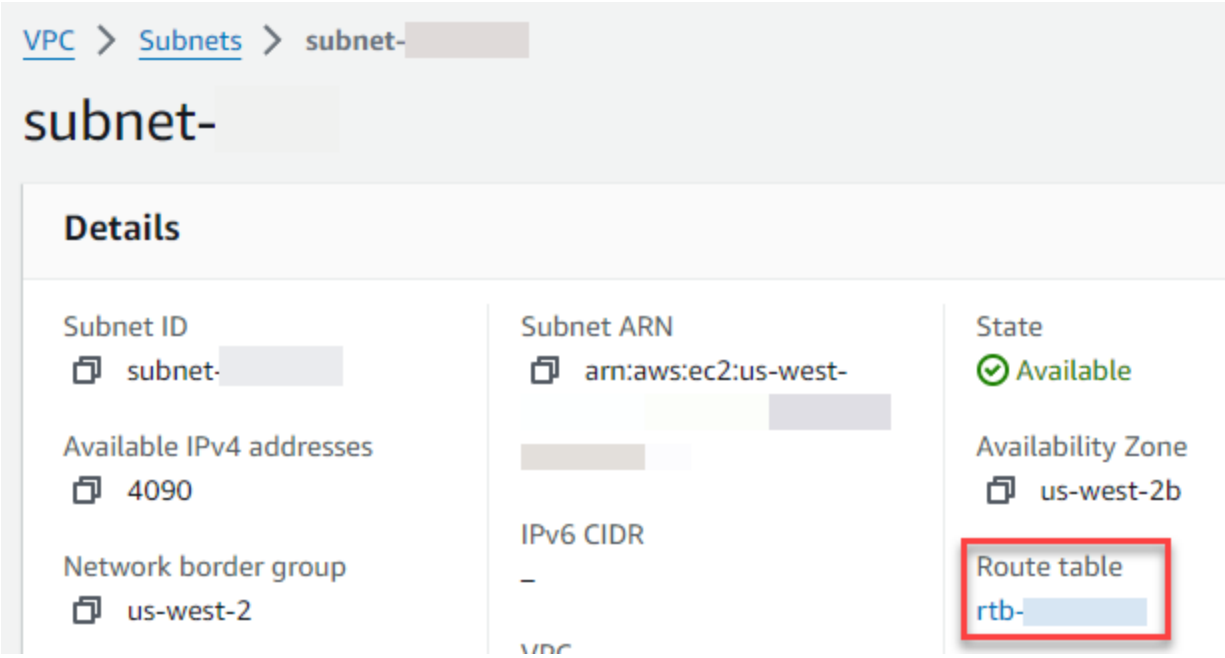
4. 對於子網路，請在 VPC 中選取公用子網路。(公用子網路是在其路由表中具有網際網路閘道的直接路由路由的子網路。)

> **Note**
>
> NAT 閘道與公用子網路相關聯，但路由表項目位於私有子網路中。

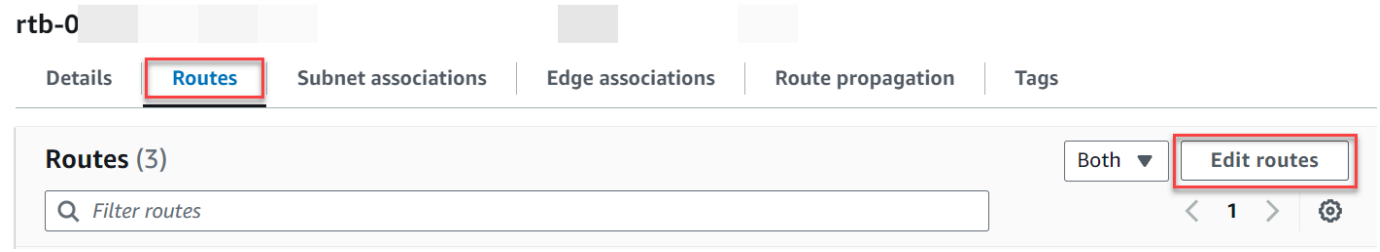
5. 對於彈性 IP 配置 ID，請選取彈性 IP 位址或選擇配置彈性 IP。
6. 選擇建立 NAT 閘道。

在私有子網路的路由表中新增路由至 NAT 閘道

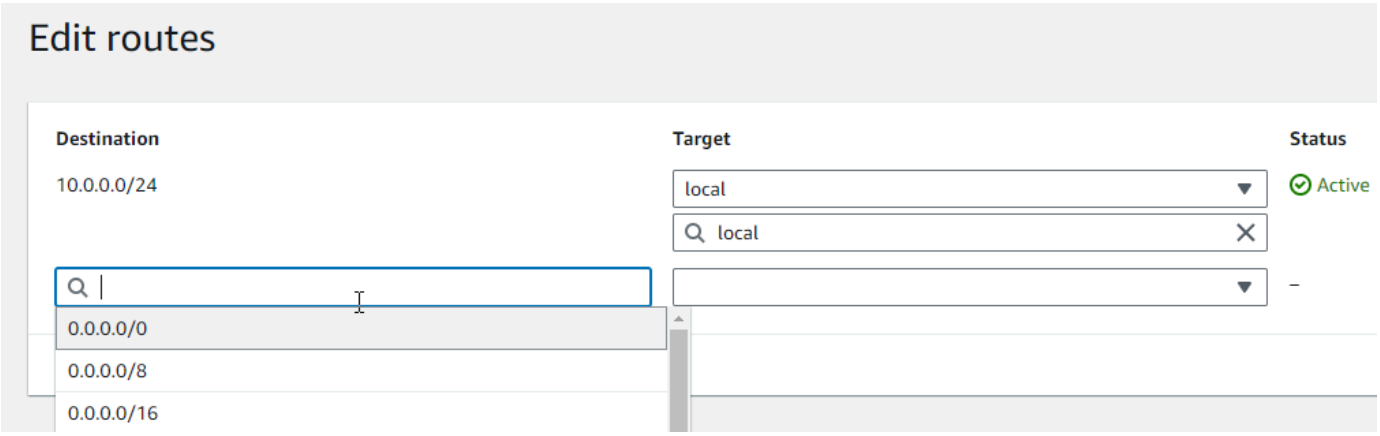
1. 在導覽窗格中，選擇 Subnets (子網)。
2. 選取 VPC 中的私有子網路。私有子網路是一個子網路，在其路由表中沒有通往網際網路閘道的路由。)
3. 選擇路由表下的鏈接。



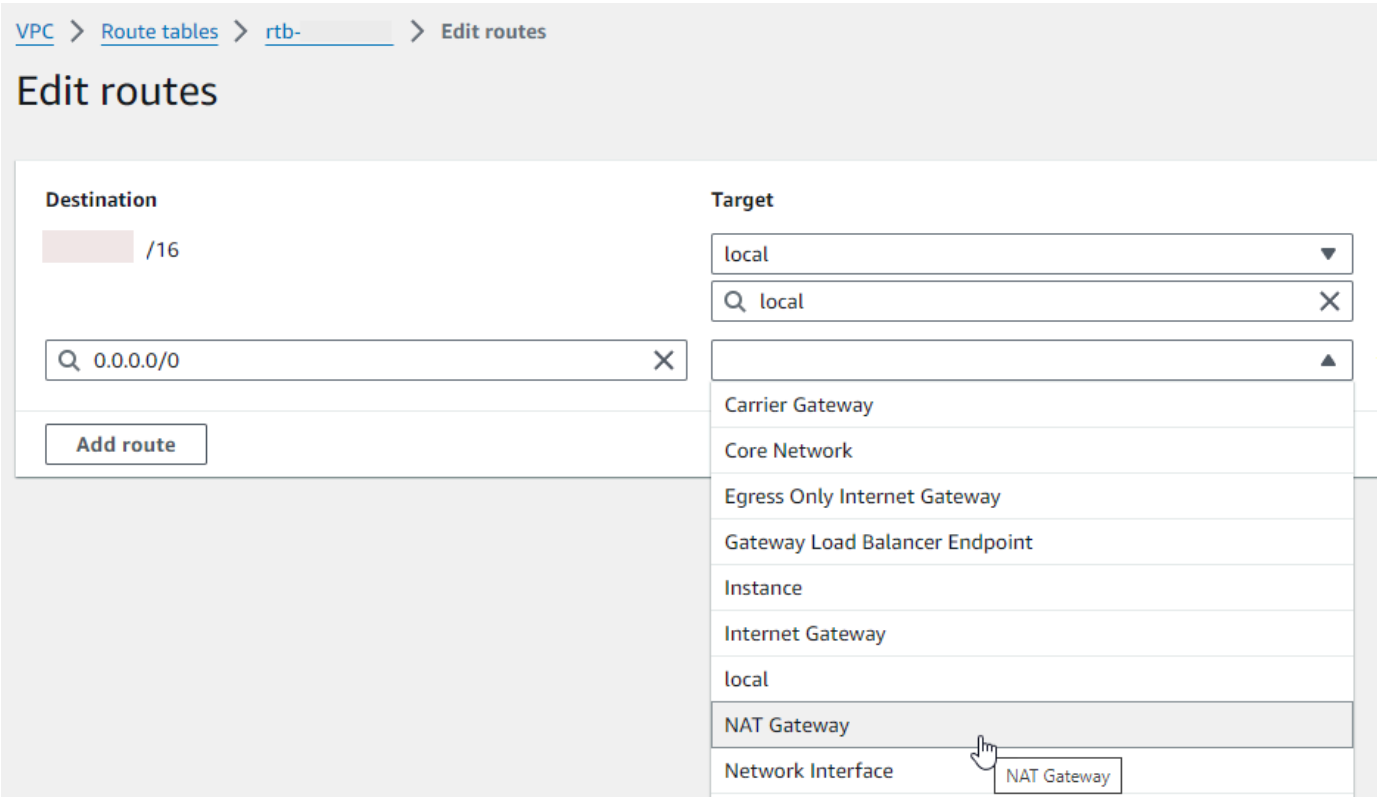
4. 向下捲動並選擇「路線」標籤頁，然後選擇「編輯路線」



5. 選擇 [新增路線]，然後`0.0.0.0/0`在 [目的地] 方塊中輸入。



6. 針對目標，選取 NAT 閘道，然後選擇您先前建立的 NAT 閘道。



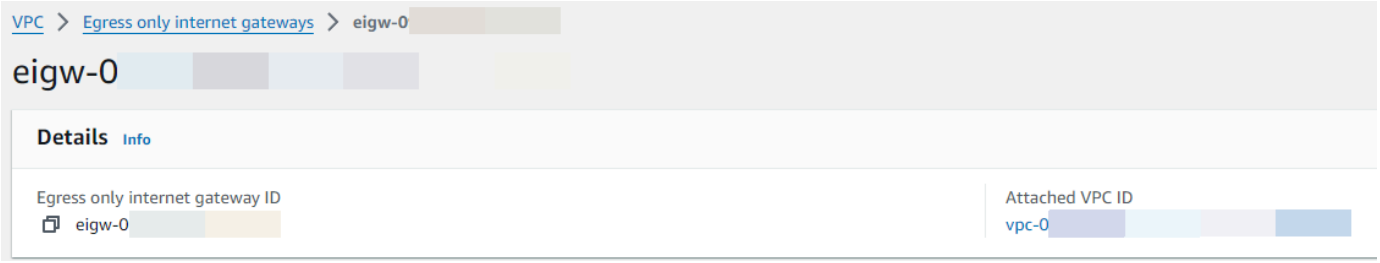
7. 選擇儲存變更。

建立僅限輸出的網際網路閘道 (僅限 IPv6)

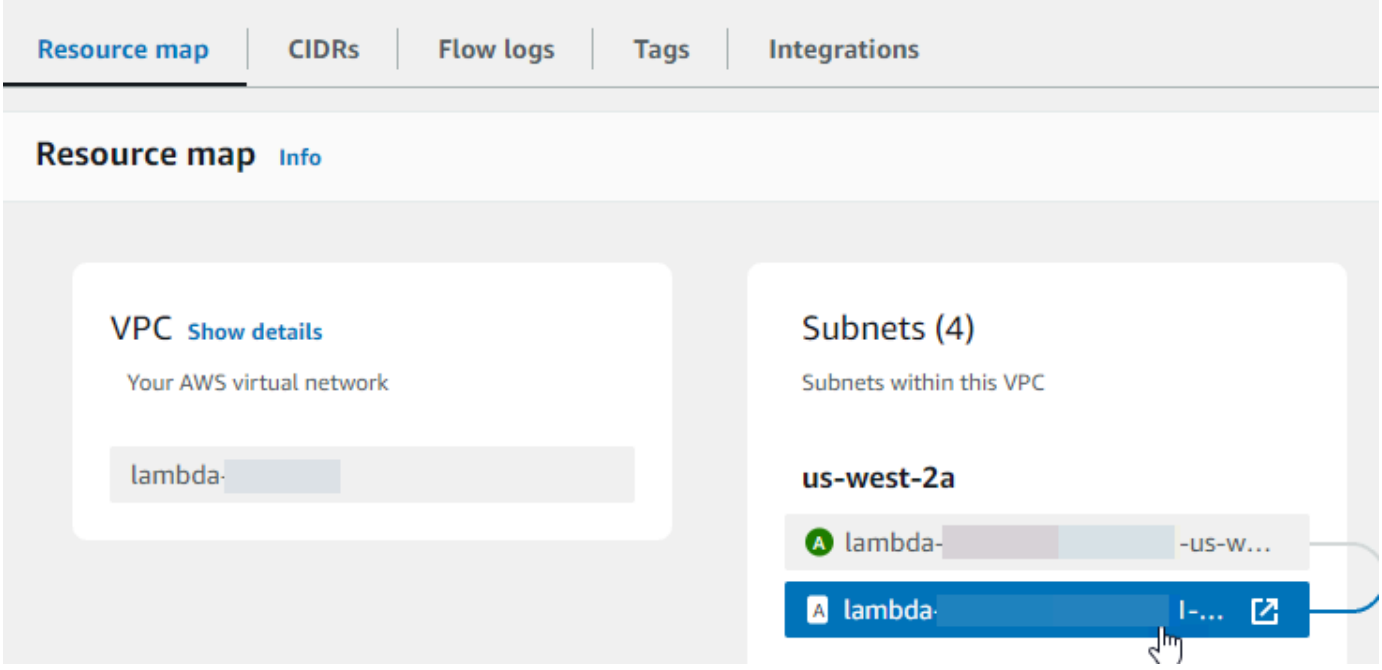
請依照下列步驟建立僅限輸出的網際網路閘道，並將其新增至您的私有子網路的路由表。

建立輸出限定網際網路閘道

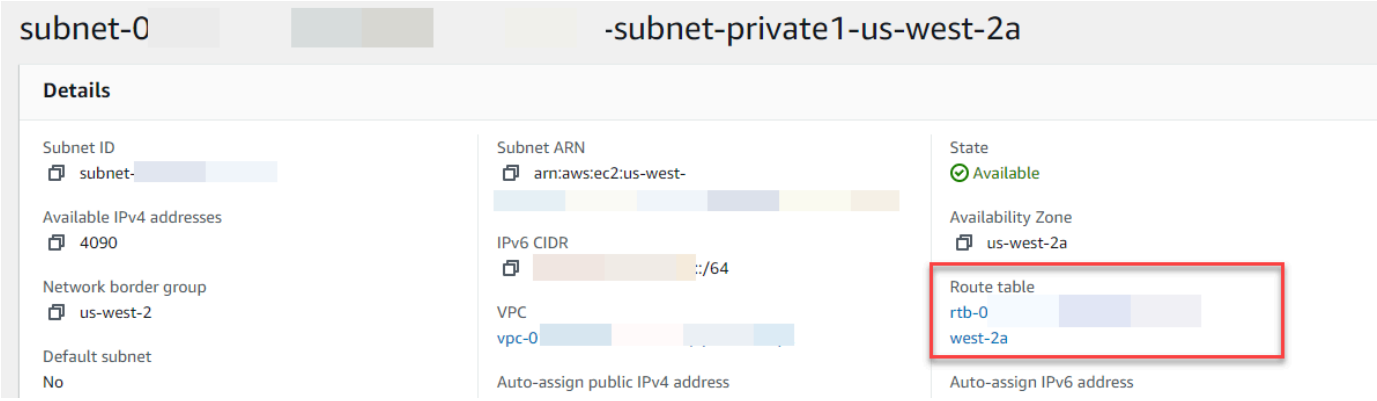
1. 在導覽窗格中，選擇輸出限定網際網路閘道。
2. 選擇建立輸出限定網際網路閘道。
3. (選擇性) 輸入名稱。
4. 選取要在其中建立輸出限定網際網路閘道的 VPC。
5. 選擇建立輸出限定網際網路閘道。
6. 在「附加的 VPC ID」下選擇連結。



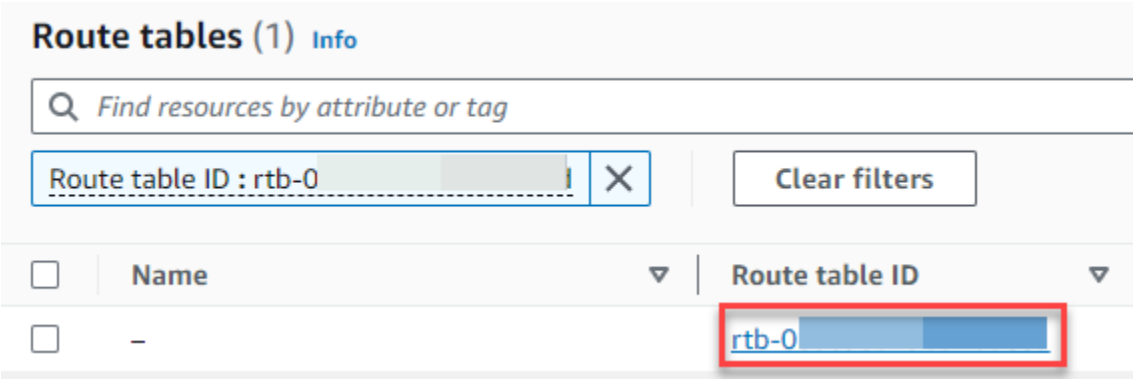
7. 選擇 VPC ID 下方的連結以開啟 VPC 詳細資訊頁面。
8. 向下捲動至 [資源對應] 區段，然後選擇私有子網路。子網路詳細資訊會顯示在新索引標籤中。



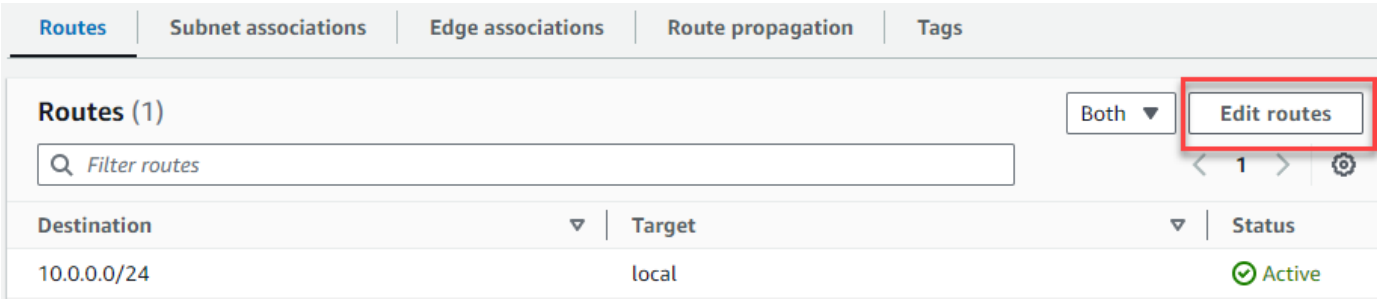
9. 選擇路由表下的鏈接。



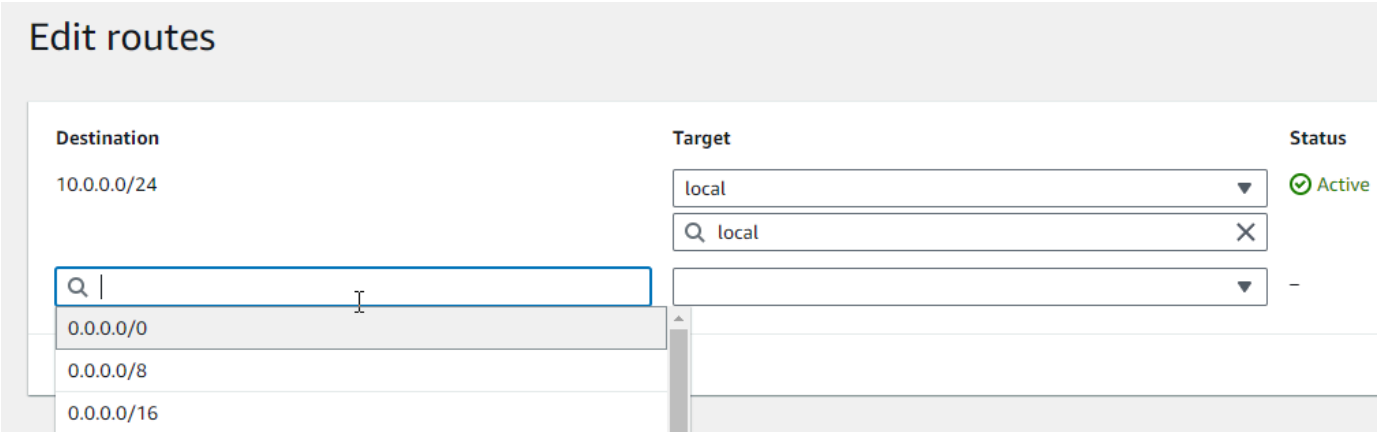
10. 選擇「路由表格 ID」以開啟路由表格詳細資訊頁面。



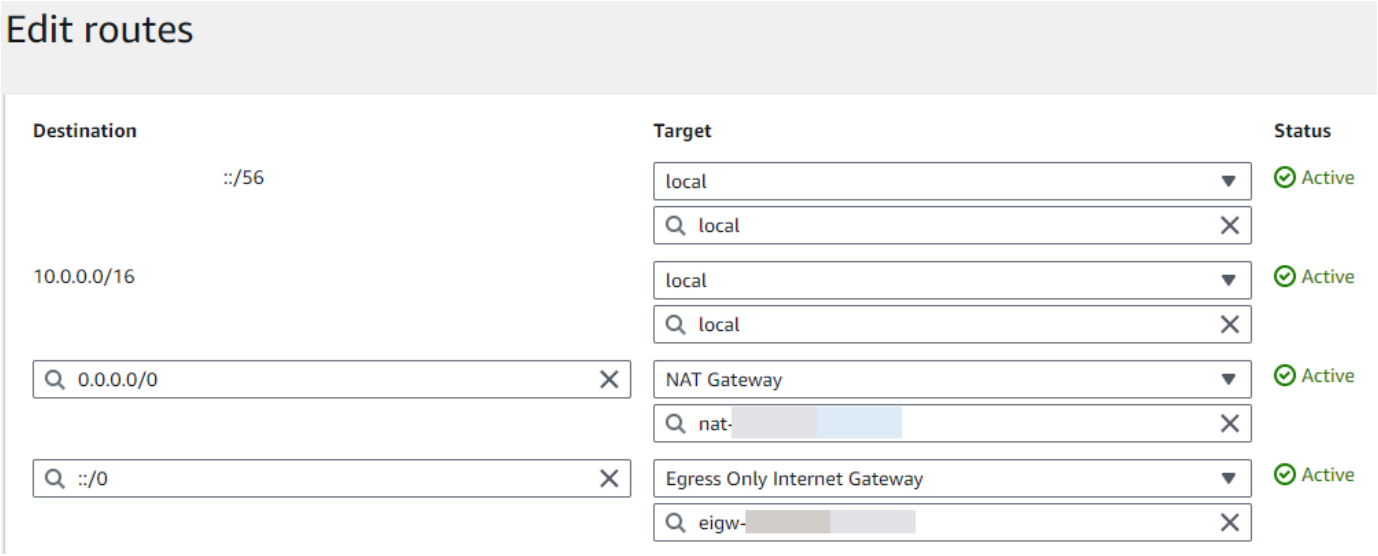
11. 在路線下，選擇編輯路線。



12. 選擇 [新增路線]，然後`::/0`在 [目的地] 方塊中輸入。



13. 針對「目標」，選取「僅出口 Internet Gateway」，然後選擇您先前建立的閘道。



14. 選擇儲存變更。

設定 Lambda 函數

若要在建立函數時設定 VPC

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇 建立函數。
3. 在 Basic information (基本資訊) 下，對於 Function name (函數名稱)，為您的函數輸入名稱。
4. 展開 Advanced settings (進階設定)。
5. 選取 [啟用 VPC]，然後選擇一個 VPC。
6. (選擇性) 若要允許傳出 IPv6 流量，請選取允許雙堆疊子網路的 IPv6 流量。
7. 對於子網路，請選取所有私人子網路。私有子網路可透過 NAT 閘道存取網際網路。將函數連接到公共子網不會給它訪問互聯網。

> **Note**
>
> 如果您選取允許雙堆疊子網路的 IPv6 流量，則所有選取的子網路均必須具有 IPv4 CIDR 區塊和 IPv6 CIDR 區塊。

8. 對於安全性群組，請選取允許輸出流量的安全性群組。
9. 選擇建立函數。

Lambda 會透過AWSLambdaVPCAccessExecutionRole AWS 受管政策自動建立執行角色。只有為 VPC 組態建立彈性網路介面時，才需要此原則中的權限，而不是呼叫您的函數。若要套用最低權限權限，您可以在建立函數和 VPC 組態之後，從執行角色移除AWSLambdaVPCAccessExecutionRole原則。如需詳細資訊，請參閱 所需的 IAM 許可。

若要為現有函數設定 VPC

若要將 VPC 組態新增至現有功能，函數的執行角色必須具有建立和管理彈性網路介面的權限。受AWSLambdaVPCAccessExecutionRole AWS 管理的策略包括必要的權限。若要套用最低權限權限，您可以在建立 VPC 組態之後從執行角色移除AWSLambdaVPCAccessExecutionRole原則。

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇配置選項卡，然後選擇 VPC。
4. 在 VPC 下，選擇 Edit (編輯)。
5. 選取 VPC。
6. (選擇性) 若要允許傳出 IPv6 流量，請選取允許雙堆疊子網路的 IPv6 流量。
7. 對於子網路，請選取所有私人子網路。私有子網路可透過 NAT 閘道存取網際網路。將函數連接到公共子網不會給它訪問互聯網。

> **Note**
>
> 如果您選取允許雙堆疊子網路的 IPv6 流量，則所有選取的子網路均必須具有 IPv4 CIDR 區塊和 IPv6 CIDR 區塊。

8. 對於安全性群組，請選取允許輸出流量的安全性群組。
9. 選擇儲存。

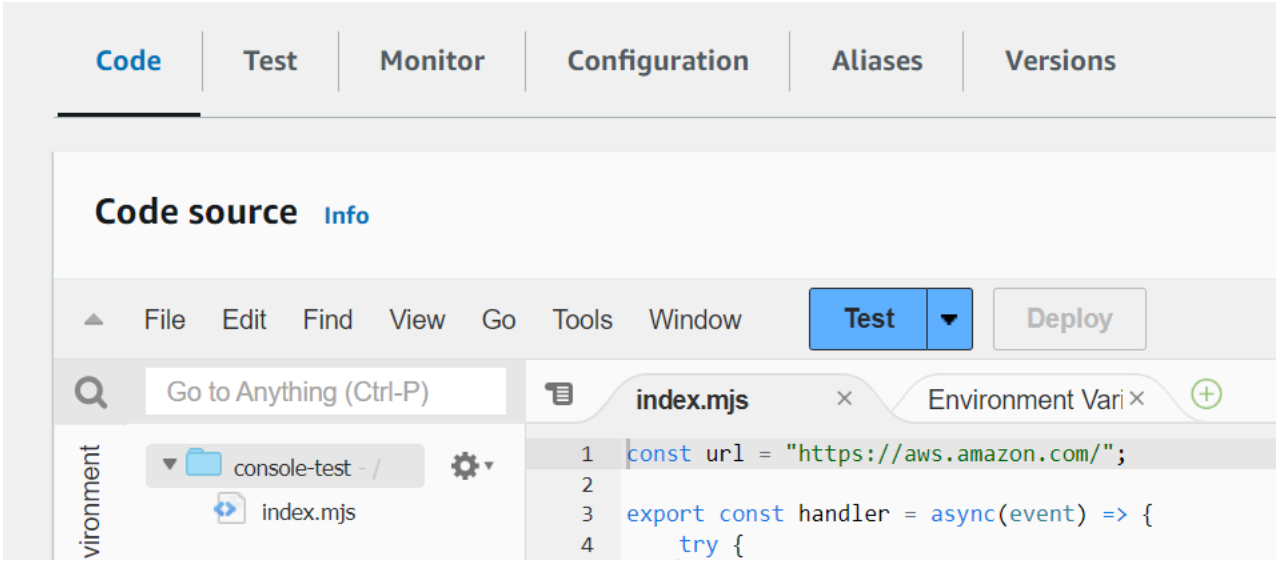
測試函數

使用下列範例程式碼，確認連接 VPC 的功能可以連線到公用網際網路。如果成功，則代碼返回一個200狀態碼。如果不成功，則函數逾時。

Node.js

此範例使用fetch，在以後的執行階段中nodejs18.x均可使用。

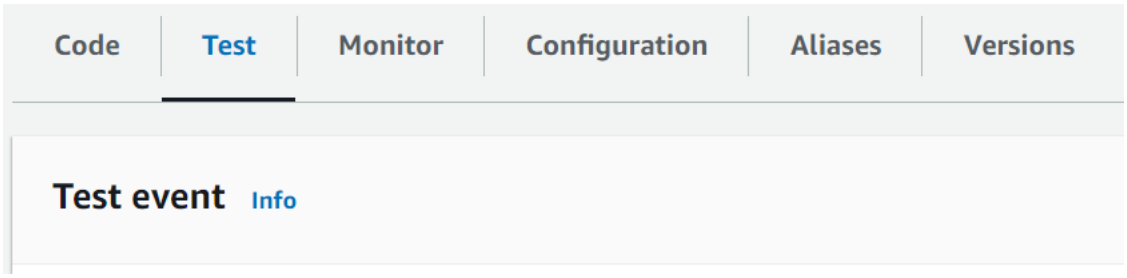
1. 在 Lambda 主控台的「程式碼原始碼」窗格中，將下列程式碼貼到 index.m js 檔案中。該函數向公共端點發出 HTTP GET 請求，並返回 HTTP 響應代碼以測試該函數是否可以訪問公共互聯網。



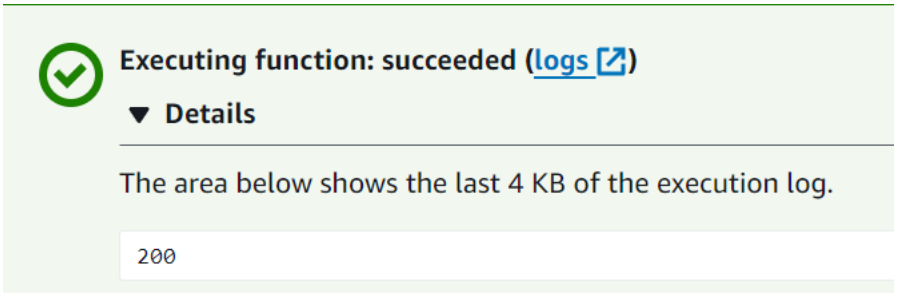
Example -使用異步/等待的HTTP請求

```
const url = "https://aws.amazon.com/";

export const handler = async(event) => {
    try {
        // fetch is available with Node.js 18 and later runtimes
        const res = await fetch(url);
        console.info("status", res.status);
        return res.status;
    }
    catch (e) {
        console.error(e);
        return 500;
    }
};
```

2. 選擇部署。
3. 選擇測試標籤。



4. 選擇 測試。
5. 該函數返回一個200狀態碼。這意味著該功能具有出站互聯網訪問。

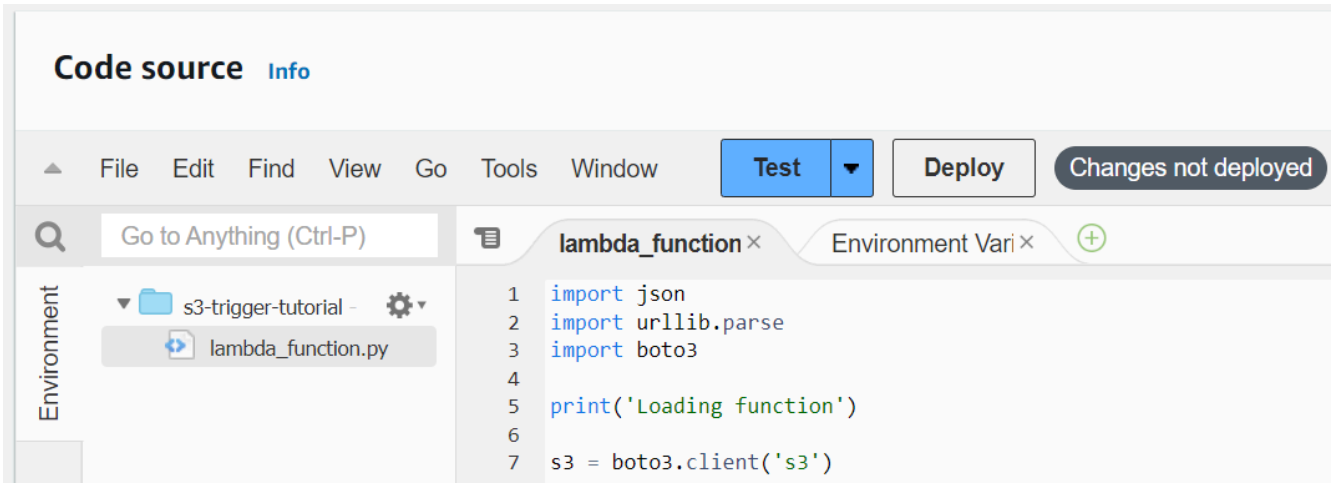


如果該功能無法訪問公共互聯網，則會收到如下錯誤消息：

```
{
  "errorMessage": "2024-04-11T17:22:20.857Z abe12jlc-640a-8157-0249-9be825c2y110
 Task timed out after 3.01 seconds"
}
```

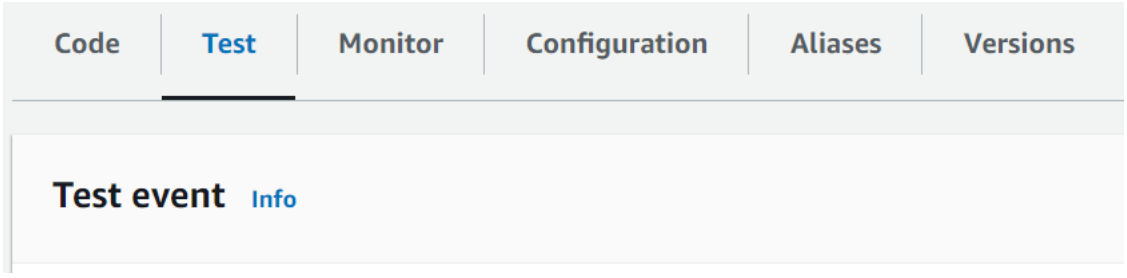
Python

1. 在 Lambda 主控台的「程式碼原始程式碼」窗格中，將下列程式碼貼到 lambda_function.py 檔案中。該函數向公共端點發出 HTTP GET 請求，並返回 HTTP 響應代碼以測試該函數是否可以訪問公共互聯網。

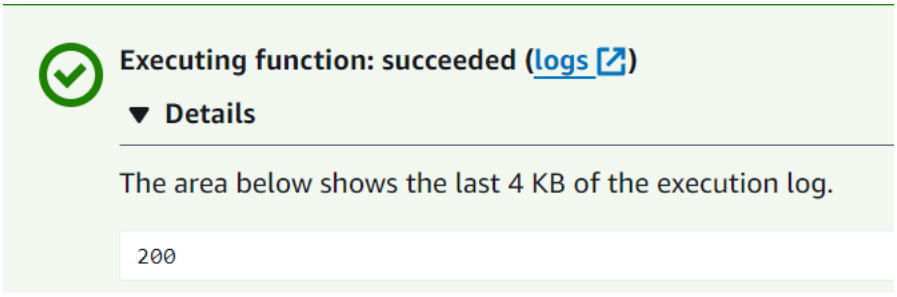



```
import urllib.request

def lambda_handler(event, context):
    try:
        response = urllib.request.urlopen('https://aws.amazon.com')
        status_code = response.getcode()
        print('Response Code:', status_code)
        return status_code
    except Exception as e:
        print('Error:', e)
        raise e
```

2. 選擇部署。
3. 選擇測試標籤。



4. 選擇 測試。
5. 該函數返回一個200狀態碼。這意味著該功能具有出站互聯網訪問。



如果該功能無法訪問公共互聯網，則會收到如下錯誤消息：

```
{
  "errorMessage": "2024-04-11T17:22:20.857Z abe12jlc-640a-8157-0249-9be825c2y110
 Task timed out after 3.01 seconds"
}
```

連線 Lambda 的傳入介面 VPC 端點

如果您使用 Amazon Virtual Private Cloud (Amazon VPC) 託管資 AWS 源，則可以在 VPC 和 Lambda 之間建立連接。您可以使用此連線來叫用您的 Lambda 函數，而不需要透過公有網際網路。

若要在 VPC 與 Lambda 之間建立私有連線，請建立介面 VPC 端點。介面端點由其提供支援 AWS PrivateLink，可讓您在沒有網際網路閘道、NAT 裝置、VPN 連線或 AWS Direct Connect 連線的情況下私有存取 Lambda API。VPC 中的執行個體不需要公有 IP 地址，即能與 Lambda API 通訊。您的 VPC 與 Lambda 之間的網路流量都不會離開 AWS 網路範圍。

每個介面端點都由子網路中的一個或多個彈性網路介面表示。網路介面會提供一個私有 IP 地址，以作為連至 Lambda 的流量進入點。

章節

- Lambda 介面端點的考量
- 為 Lambda 建立介面端點
- 為 Lambda 建立介面端點政策

Lambda 介面端點的考量

在設定 Lambda 的介面端點之前，請務必檢閱 Amazon VPC 使用者指南中的介面端點屬性和限制。

您可以從 VPC 呼叫任何 Lambda API 操作。例如，您可以從 VPC 內呼叫 `Invoke` API 來叫用 Lambda 函數。如需完整的 Lambda API 清單，請參閱 Lambda API 參考中的動作。

`use1-az3` 是 Lambda VPC 函數的有限容量區域。您不應將此可用區域中的子網路與 Lambda 函數搭配使用，因為這可能會在出現中斷狀況時導致區域備援減少。

用於持續連線的 keep-alive (保持啟用)

Lambda 一段時間後會清除閒置連線，因此您必須使用實時指示來維持持續連線。叫用函式時，嘗試重複使用閒置連線會導致連線錯誤。若要維護持續連線，請使用與執行階段相關聯的 keep-alive (保持啟用) 指令。如需範例，請參閱《AWS SDK for JavaScript 開發人員指南》中的在 Node.js 中重複使用 Keep-Alive 的連線。

計費考量

透過介面端點來存取 Lambda 函數無需額外成本。如需 Lambda 定價的詳細資訊，請參閱 AWS Lambda 定價。

的標準定價 AWS PrivateLink 適用於 Lambda 的介面端點。您的 AWS 帳戶每小時會在每個可用區域中佈建一個介面端點，以及透過介面端點處理的資料計費。如需介面端點定價的詳細資訊，請參閱 AWS PrivateLink 定價。

VPC 對等互連考量

您可以使用 VPC 對等互連，將其他 VPC 連線到具有介面端點的 VPC。VPC 對等互連是兩個 VPC 之間的網路連線。您可以在自己的 VPC 之間建立 VPC 對等互連的連線，或與其他 AWS 帳戶的 VPC 建立對等互連的連線。VPC 也可以位於兩個不同的 AWS 區域。

對等 VPC 之間的流量會保留在 AWS 網路上，且不會穿越公用網際網路。VPC 對等互連後，兩個 VPC 中的 Amazon Elastic Compute Cloud (Amazon EC2) 執行個體、Amazon Relational Database Service (Amazon RDS) 執行個體或啟用 VPC 功能的 Lambda 函數等這類資源就可以透過在其中一個 VPC 中建立的介面端點來存取 Lambda API。

為 Lambda 建立介面端點

您可以使用 Amazon VPC 主控台或 AWS Command Line Interface (AWS CLI) 為 Lambda 建立介面端點。如需詳細資訊，請參閱《Amazon VPC 使用者指南》中的建立介面端點。

若要為 Lambda 建立介面端點 (主控台)

1. 開啟 Amazon VPC 主控台的 Endpoints (端點) 頁面。
2. 選擇建立端點。
3. 對於服務類別，請確定您已選擇 AWS 服務。
4. 在服務名稱中，選擇 com.amazonaws.***region***.lambda。確認類型為介面。
5. 建立 VPC 和子網路
6. 若要啟用介面端點的私有 DNS，請選取 Enable DNS Name (啟用 DNS 名稱) 核取方塊。建議您為 VPC 端點啟用私人 DNS 名稱。 AWS 服務如此可確保使用公用服務端點的要求 (例如透過 AWS SDK 發出的要求) 會解析至您的 VPC 端點。
7. 對於安全群組，選擇一或多個安全群組。
8. 選擇建立端點。

若要使用私有 DNS 選項，您必須設定 VPC 的 `enableDnsHostnames` 和 `enableDnsSupportattributes`。如需詳細資訊，請參閱 Amazon VPC 使用者指南中的檢視並更新 VPC 的 DNS 支援。如果您為該介面端點啟用私有 DNS，您可以使用其區域的預設 DNS 名稱 (例

如 lambda.us-east-1.amazonaws.com)，向 Lambda 發出 API 請求。如需更多服務端點，請參閱《AWS 一般參考》中的服務端點和配額。

如需詳細資訊，請參閱《Amazon VPC 使用者指南》中的透過介面端點存取服務。

如需使用建立和設定端點的相關資訊 AWS CloudFormation，請參閱使用者指南中的 AWS:: EC2:: vpcendPoint 資源。AWS CloudFormation

若要為 Lambda 建立介面端點 (AWS CLI)

使用 create-vpc-endpoint 命令，並指定 VPC ID、VPC 端點 (介面) 的類型、服務名稱、要使用端點的子網路，以及要與端點網路介面建立關聯的安全群組。例如：

```
aws ec2 create-vpc-endpoint --vpc-id vpc-ec43eb89 --vpc-endpoint-type Interface --
service-name \
   com.amazonaws.us-east-1.lambda --subnet-id subnet-abababab --security-group-id
 sg-1a2b3c4d
```

為 Lambda 建立介面端點政策

若要控制誰可以使用您的介面端點，以及使用者可以存取哪些 Lambda 函數，您可以將端點政策連接至您的端點。此政策會指定下列資訊：

- 可執行動作的主體。
- 委託人可以執行的動作。
- 委託人可以對其執行動作的資源。

如需詳細資訊，請參閱《Amazon VPC 使用者指南》中的使用 VPC 端點控制對服務的存取。

範例：Lambda 動作的介面端點政策

以下是 Lambda 端點政策的範例。當連接到端點時，此政策允許使用者 MyUser 叫用函式 my-function。

> **Note**
>
> 您需要在資源中同時包含合格和不合格的函數 ARN。

```
{
   "Statement":[
      {
         "Principal":
         {
             "AWS": "arn:aws:iam::111122223333:user/MyUser"
         },
         "Effect":"Allow",
         "Action":[
            "lambda:InvokeFunction"
         ],
         "Resource": [
               "arn:aws:lambda:us-east-2:123456789012:function:my-function",
               "arn:aws:lambda:us-east-2:123456789012:function:my-function:*"
            ]
      }
   ]
}
```

設定 Lambda 函數的檔案系統存取權

您可以設定函數，將 Amazon Elastic File System (Amazon EFS) 檔案系統掛載到本機目錄。使用 Amazon EFS，您的函數程式碼可安全地存取和修改共用資源，並發揮高度並行效能。

章節

執行角色和使用者許可

如果檔案系統沒有使用者設定 AWS Identity and Access Management (IAM) 政策，EFS 會使用預設政策，授與任何可使用檔案系統掛載目標連線至檔案系統的用戶端的完整存取權。如果檔案系統具有使用者設定的 IAM 政策，則函數的執行角色必須具有正確的 `elasticfilesystem` 許可。

執行角色許可

- 彈性文件系統：ClientMount
- 彈性文件系統：ClientWrite （只讀連接不需要）

這些權限包含在AmazonElasticFileSystemClientReadWriteAccess受管理的策略中。此外，您的執行角色必須具有連線至檔案系統的 VPC 所需的許可。

設定檔案系統時，Lambda 會使用您的許可來驗證掛載目標。若要設定函數以連線至檔案系統，您的使用者需要下列許可：

使用者權限

- 彈性文件系統：目標 DescribeMount

設定檔案系統和存取點

在函數所連線的每個可用區域中，在具有掛載目標的 Amazon EFS 中建立檔案系統。為了提高效能和彈性，請至少使用兩個可用區域。例如，在簡單的組態中，您可以在不同的可用區域中擁有包含兩個私

有子網路的 VPC。此函數連線到兩個子網路，每個子網路都有可用的掛載目標。確定函數和掛載目標所使用的安全群組允許 NFS 流量 (連接埠 2049)。

> **Note**
>
> 建立檔案系統時，您選擇稍後無法變更的效能模式。General purpose (一般用途) 模式具有較低的延遲，而 Max I/O (IO 上限) 模式支援較高的輸送量上限和 IOPS。如需協助選擇，請參閱 Amazon Elastic File System 使用者指南中的 Amazon EFS 效能。

存取點會將函數的每個執行個體連線至可用區域連線到的正確掛載目標。為了獲得最佳效能，請使用非根路徑建立存取點，並限制您在每個目錄中建立的檔案數目。下列範例會在檔案系統上建立名為 `my-function` 的目錄，並將擁有者 ID 設為 1001，具有標準目錄許可 (755)。

Example 存取點組態

- 名稱 – `files`
- 使用者 ID – `1001`
- 群組 ID – `1001`
- 路徑 – `/my-function`
- 許可 – `755`
- 擁有者使用者 ID – `1001`
- 群組使用者 ID – `1001`

當函數使用存取點時，會提供使用者 ID 1001，並具有目錄的完整存取權。

如需詳細資訊，請參閱 Amazon Elastic File System 使用者指南中的下列主題。

- 為 Amazon EFS 建立資源
- 處理使用者、群組和許可

連線至檔案系統 (主控台)

此函數會透過 VPC 中的本機網路連線至檔案系統。您的函數連線到的子網路可能是包含檔案系統掛載點的子網路，或同一個可用區域中的子網路，其可將 NFS 流量 (連接埠 2049) 路由至檔案系統。

> **Note**
>
> 如果您的函數尚未連線至 VPC，請參閱讓 Lambda 函數存取 Amazon VPC 中的資源。

設定檔案系統存取權

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇 組態 ，然後選擇 檔案系統 。
4. 在 檔案系統 中，選擇 新增檔案系統。
5. 設定下列屬性：
   - EFS file system (EFS 檔案系統) - 相同 VPC 中的檔案系統存取點。
   - Local mount path (本機掛載路徑) - 檔案系統掛載於 Lambda 函數 (以 `/mnt/` 開頭) 的位置。

> **定價**
>
> Amazon EFS 會針對儲存和輸送量收取費用，費率依儲存類別而異。如需詳細資訊，請參閱 Amazon EFS 定價。
> Lambda 會針對 VPC 之間的資料傳輸收取費用。這僅適用於您的函數的 VPC 對等連線到帶有檔案系統的另一個 VPC 的情況。費率與相同區域內 VPC 之間的 Amazon EC2 資料傳輸費用相同。如需詳細資訊，請參閱 Lambda 定價。

如需 Lambda 與 Amazon EFS 整合的詳細資訊，請參閱 搭配使用 Amazon EFS 與 Lambda。

在另一個使用 Amazon EFS 檔案系統來執 AWS 帳戶 行 Lambda 函數

您可以設定函數，在另一個系統中掛載 Amazon EFS 檔案系統 AWS 帳戶。掛載檔案系統之前，您必須確認下列事項：

- 必須設定 VPC 對等互連，且必須將適當的路由新增到每個 VPC 的路由表中。
- 您要掛載之 Amazon EFS 檔案系統的安全群組，必須設定為允許從與 Lambda 函數相關聯的安全群組傳入存取。
- 必須使用相符的可用區域 (AZ) ID，在每個 VPC 中建立子網路。

- VPC 都必須啟用 DNS 主機名稱。

若要讓 Lambda 函數存取另一個檔案系統中的 Amazon EFS 檔案系統 AWS 帳戶，該檔案系統還必須具有可授予功能許可的檔案系統政策。若要了解如何建立檔案系統政策，請參閱《Amazon Elastic File System 使用者指南》中的建立檔案系統政策。

下列範例政策提供指定帳戶中的 Lambda 函數，在檔案系統上執行所有 API 動作的許可。

```
{
    "Version": "2012-10-17",
    "Id": "efs-lambda-policy",
    "Statement": [
        {
            "Sid": "efs-lambda-statement",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::{LAMBDA-ACCOUNT-ID}:root"
            },
            "Action": "*",
            "Resource": "arn:aws:elasticfilesystem:{REGION}:{ACCOUNT-ID}:file-
system/{FILE SYSTEM ID}"
        }
    ]
}
```

> **Note**
>
> 顯示的範例政策使用萬用字元字元 (「*」) 來授與指定的 Lambda 函數的權限，以 AWS 帳戶便在檔案系統上執行任何 API 作業。這包括刪除檔案系統。若要限制其他人 AWS 帳戶 可以在您的檔案系統上執行的作業，請明確指定您要允許的動作。如需可能的 API 操作清單，請參閱 Amazon Elastic File System 的動作、資源和條件索引鍵。

若要設定跨帳戶檔案系統掛載，請使用 AWS Command Line Interface (AWS CLI) `update-function-configuration` 作業。

若要在另一個檔案系統中掛載檔案系統 AWS 帳戶，請執行下列命令。使用您自己的函數名稱，並以要掛載之檔案系統的 Amazon EFS 存取點 ARN 取代 Amazon Resource Name (ARN)。`LocalMountPath` 是函數可以存取檔案系統的路徑，以 `/mnt/` 開頭。確保 Lambda 掛載路

徑與檔案系統的存取點路徑相符。舉例來說，如果存取點為 /efs，Lambda 掛載路徑必須為 /mnt/efs。

```
aws lambda update-function-configuration --function-name MyFunction \
--file-system-configs Arn=arn:aws:elasticfilesystem:us-east-1:222222222222:access-point/fsap-01234567,LocalMountPath=/mnt/test
```

為 Lambda 函數建立別名

您可以為您的 Lambda 函數建立別名。Lambda 別名是您可以更新的函數版本的指標。函數的使用者可以使用別名 Amazon Resource Name (ARN) 來存取函數版本。部署新版本時，您可以更新別名以使用新版本，或分割兩個版本之間的流量。

章節

建立函數別名 (主控台)

您可以使用 Lambda 主控台建立函數別名。

建立別名

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇 Aliases (別名)，然後選擇 Create alias (建立別名)。
4. 在 Create alias (建立別名) 頁面，執行下列動作：
    a. 輸入別名的 Name (名稱)。
    b. (選用) 輸入別名的 Description (描述)。
    c. 在 Version (版本)中，選擇要別名指向的函數版本。
    d. (選用) 若要在別名上設定路由，請展開 Weighted alias (加權別名)。如需詳細資訊，請參閱 別名路由組態。
    e. 選擇 Save (儲存)。

使用 Lambda API 管理別名

若要使用 AWS Command Line Interface (AWS CLI) 建立別名，請使用`create-alias`指令。

```
aws lambda create-alias --function-name my-function --name alias-name --function-
version version-number --description " "
```

若要變更別名以指向新版本的函數，請使用 update-alias 命令。

```
aws lambda update-alias --function-name my-function --name alias-name --function-
version version-number
```

若要刪除別名，請使用 delete-alias 命令。

```
aws lambda delete-alias --function-name my-function --name alias-name
```

上述步驟中的 AWS CLI 命令對應於下列 Lambda API 作業：

- CreateAlias
- UpdateAlias
- DeleteAlias

使用 AWS SAM 和管理別名 AWS CloudFormation

您可以使用 AWS Serverless Application Model (AWS SAM) 和來建立和管理函數別名 AWS CloudFormation。

若要瞭解如何在 AWS SAM 範本中宣告函數別名，請參閱開發人員指南中的AWS:: 無伺服器:: Function 頁面。 AWS SAM 如需使用建立和設定別名的相關資訊 AWS CloudFormation，請參閱AWS 使用指南中的:: Lambda:: Alias。 AWS CloudFormation

使用別名

每個別名都有唯一的 ARN。別名只能指向函數版本，而非另一個別名。您可以更新別名以指向新版本的函數。

諸如 Amazon Simple Storage Service (Amazon S3) 等事件來源會叫用您的 Lambda 函數。當事件發生時，這些事件來源會維護一個映射，以識別發生事件時要叫用的函數。如果您在映射組態中指定了 Lambda 函數別名，則不需要在函數版本變更時更新映射。如需詳細資訊，請參閱 Lambda 如何處理串流和以佇列為基礎的事件來源的記錄。

在資源政策中，您可以授予事件來源使用 Lambda 函數的許可。如果您在政策中指定了別名 ARN，則不需要在函數版本變更時更新政策。

資源政策

您可以使用以資源為基礎的政策，將服務、資源或帳號存取權授予您的函數。該許可的範圍取決於您是將其套用至別名、版本或整個函數。例如，如果您使用別名名稱 (例如 `helloworld:PROD`)，許可可讓您使用別名 ARN (`helloworld:PROD`) 來叫用 `helloworld` 函數。

如果您嘗試在沒有別名或特定版本的情況下叫用該函數，則會出現許可錯誤。即使您嘗試直接叫用與別名相關聯的函數版本，仍會發生此許可錯誤。

例如，當 Amazon S3 代表執行動作時，下列 AWS CLI 命令會授與 Amazon S3 許可以叫用`helloworld`函數的 `DOC-EXAMPLE-BUCKET` `PROD` 別名。

```
aws lambda add-permission --function-name helloworld \
--qualifier PROD --statement-id 1 --principal s3.amazonaws.com --action
 lambda:InvokeFunction \
--source-arn arn:aws:s3:::DOC-EXAMPLE-BUCKET --source-account 123456789012
```

如需在政策中使用資源名稱的詳細資訊，請參閱微調政策的「資源」和「條件」部分。

別名路由組態

在別名上使用路由組態，將部分流量傳送至第二個函數版本。例如，您可以將別名設定為將大部分流量傳送至現有版本，而只將一小部分的流量傳送至新版本，以降低部署新版本的風險。

請注意，Lambda 會使用簡單的概率模型來分配兩個函數版本之間的流量。在流量較低時，您可能會看到每個版本已設定流量百分比與實際流量百分比之間，存在很大差異。如果您的函數使用佈建並行，透過在別名路由作用期間設定較高數目的已佈建並行執行個體，則可以避免溢出叫用。

一個別名可以指向最多兩個 Lambda 函數版本。版本必須符合下列條件：

- 兩個版本必須擁有相同的執行角色。
- 這兩個版本都必須具有相同無效字元佇列組態，或是沒有無效字元佇列組態。
- 兩個版本都必須發佈。別名不能指向 `$LATEST`。

設定別名上的路由

> **Note**
>
> 確認函數至少有兩個已發佈的版本。若要建立其他版本，請遵循Lambda 函數版本中的指示。

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇 Aliases (別名)，然後選擇 Create alias (建立別名)。
4. 在 Create alias (建立別名) 頁面，執行下列動作：
   a. 輸入別名的 Name (名稱)。
   b. (選用) 輸入別名的 Description (描述)。
   c. 在 Version (版本)中，選擇要別名指向的第一個函數版本。
   d. 展開 Weighted alias (加權別名)。
   e. 在 Additional version (其他版本)中，選擇要別名指向的第二個函數版本。
   f. 為 Weight (%) (權重 (%))函數輸入一個權重值。權數是當別名被叫用時，指派至該版本的流量百分比。第一版收到剩餘權數。例如，若指定 10% 至 Additional version (其他版本)，第一版會自動指派 90%。
   g. 選擇儲存。

使用 CLI 設定別名路由

使用 `create-alias` 和 `update-alias` AWS CLI 命令，設定兩個函數版本之間的流量權重。當您建立或更新別名時，請在 `routing-config` 參數中指定流量權重。

以下範例會建立一個名為 routing-alias 的 Lambda 函數別名，此別名指向函數的版本 1。函數的版本 2 會接收 3% 的流量。剩餘 97% 的流量會路由至版本 1。

```
aws lambda create-alias --name routing-alias --function-name my-function --function-
version 1  \
--routing-config AdditionalVersionWeights={"2"=0.03}
```

使用 `update-alias` 命令可增加連入流量至版本 2 的百分比。在下列範例中，您會將流量增加到 5%。

```
aws lambda update-alias --name routing-alias --function-name my-function  \
--routing-config AdditionalVersionWeights={"2"=0.05}
```

若要將所有流量路由傳送至版本 2，請使用 `update-alias` 命令，將 `function-version` 屬性變更為將別名指向版本 2。此命令也會重設路由組態。

```
aws lambda update-alias --name routing-alias --function-name my-function  \
```

```
--function-version 2 --routing-config AdditionalVersionWeights={}
```

上述步驟中的 AWS CLI 命令對應於下列 Lambda API 作業：

- CreateAlias
- UpdateAlias

判斷已叫用哪個版本

當您設定兩個函數版本之間的流量權重時，有兩種方法可判斷已叫用的 Lambda 函數版本：

- CloudWatch 日誌 — Lambda 會在每次函數叫用時，自動將包含叫用版本 ID 的 CloudWatch 日誌項目發送至 Amazon 日誌。START以下是範例：

  ```
  19:44:37 START RequestId: request id Version: $version
  ```

  對於別名叫用，Lambda 使用 Executed Version 維度，依照已叫用的版本來篩選指標資料。如需詳細資訊，請參閱 使用 Lambda 函數指標。
- 回應承載 (同步呼叫) – 回應至同步函式呼叫包含 x-amz-executed-version 標題，以顯示已呼叫哪個函式版本。

Lambda 函數版本

您可以使用版本來管理函數的部署。例如，您可以發佈新版本的函數來測試 Beta 版，而不會影響穩定生產版本的使用者。每次發佈函數時，Lambda 都會建立新版本的函數。新版本是此函數未發佈版本的副本。未發佈的版本名稱為 $LATEST。

> **Note**
>
> 若要建立函數的新版本，您必須先變更未發佈的版本 ($LATEST)。這些變更可能包括更新程式碼或修改組態設定。如果 $LATEST 與先前發佈的版本相同，則在將變更部署到 $LATEST 之前，您將無法建立新版本。

發佈函數版本之後，其程式碼、執行階段、架構、記憶體、層和大多數其他組態設定都是不可變的。這表示您無法在未從 $LATEST 發佈新版本的情況下變更這些設定。您可以為已發佈的函數版本設定下列項目：

- 觸發
- 目的地
- 佈建並行
- 非同步叫用
- 資料庫連線和代理

> **Note**
>
> 搭配 Auto 模式使用執行階段管理控制項時，函數版本所使用的執行階段版本會自動更新。使用 Function update (函數更新) 或 Manual (手動) 模式時，不會更新執行階段版本。如需詳細資訊，請參閱 the section called “執行階段更新”。

章節

- 建立函數版本
- 使用版本
- 授予許可

建立函數版本

您只能在未發佈的函數版本上變更函數代碼和設定。當您發佈版本時，Lambda 會鎖定程式碼和大多數設定以為該版本的使用者保持一致的使用體驗。

您可以使用 Lambda 主控台建立函數版本。

新建函數版本

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇一個函數，然後選擇 Versions (版本)。
3. 在版本組態頁面上，選擇 Publish new version (發佈新版本)。
4. (選用) 輸入版本描述。
5. 選擇 Publish (發佈)。

或者，您可以使用 PublishVersionAPI 操作發布函數的版本。

下列 AWS CLI 指令會發佈函數的新版本。回應會傳回關於新版本的組態資訊，包含版本號碼，以及含有版本尾碼的函式 ARN。

```
aws lambda publish-version --function-name my-function
```

您應該會看到下列輸出：

```
{
  "FunctionName": "my-function",
  "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function:1",
  "Version": "1",
  "Role": "arn:aws:iam::123456789012:role/lambda-role",
  "Handler": "function.handler",
  "Runtime": "nodejs20.x",
  ...
}
```

> **Note**
>
> Lambda 會為版本控制指派依序遞增的序號。即使刪除並重新建立函數，Lambda 也不會重複使用版本號碼。

使用版本

您可以使用合格的 ARN 或不合格的 ARN 來參考您的 Lambda 函數。

- 合格的 ARN - 帶有版本尾碼的函數 ARN。下列範例指的是 `helloworld` 函數的版本 42。

```
arn:aws:lambda:aws-region:acct-id:function:helloworld:42
```

- 不合格的 ARN - 沒有版本尾碼的函數 ARN。

```
arn:aws:lambda:aws-region:acct-id:function:helloworld
```

您可以在所有相關 API 操作中使用合格或不合格的 ARN。但是，您無法使用不合格的 ARN 來建立別名。

如果您決定不發佈函數版本，則可以在事件來源映射中使用合格或不合格的 ARN 來調用函數。當您使用不合格的 ARN 調用函數時，Lambda 會隱含調用 $LESTEST。

只有在程式碼從未發佈過，或程式碼已從上次發佈的版本變更時，Lambda 才會發佈新的函數版本。如果沒有變更，函數版本會保持在最新發佈的版本。

每個 Lambda 函數版本的合格 ARN 是唯一的。發佈版本之後，您就無法變更 ARN 或函數程式碼。

授予許可

您可以使用以資源為基礎的政策或以身分為基礎的政策來授予您函數的存取權。許可的範圍取決於您是將政策套用至函數或函數的一個版本。如需政策中函數資源名稱的詳細資訊，請參閱微調政策的「資源」和「條件」部分。

您可以使用函數別名簡化事件來源和 AWS Identity and Access Management (IAM) 政策的管理。如需更多詳細資訊，請參閱 為 Lambda 函數建立別名。

設定 Lambda 函數以串流回應

您可以設定 Lambda 函數 URL，將回應承載串流回用戶端。透過提高第一個位元組時間 (TTFB) 效能，回應串流有益於延遲敏感應用程式。這是因為您可以在部分回應可用時將其傳回給用戶端。此外，您可以使用回應串流來建置可傳回更大承載的函數。與緩衝回應的 6 MB 限制相比，回應串流承載的軟性限制為 20 MB。串流回應也意味著您的函數不需要將整個回應放在記憶體裡。若是非常大的回應，這有助於減少您需要為函數設定的記憶體容量。

Lambda 串流回應的速度取決於回應的大小。函數回應的串流速度在前 6 MB 不受限制。若回應大於 6 MB，則其餘的回應會受到頻寬上限的限制。如需串流頻寬的詳細資訊，請參閱回應串流的頻寬限制。

串流回應會產生成本。如需詳細資訊，請參閱 AWS Lambda 定價。

Lambda 支援 Node.js 受管執行期的回應串流。若為其他語言，您可以使用具有自訂執行期 API 整合的自訂執行期來串流回應，或使用 Lambda Web Adapter。您可以透過 Lambda 函數網址、AWS開發套件或使用 Lambda InvokeWithResponseStreamAPI 來串流回應。

> **Note**
>
> 透過 Lambda 主控台測試函數時，您一律會看到緩衝的回應。

撰寫啟用回應串流的函數

撰寫回應串流函數的處理常式不同於典型的處理常式模式。撰寫串流函數時，請務必執行下列動作：

- 使用本機 Node.js 執行期提供的 `awslambda.streamifyResponse()` 裝飾項目包裝您的函數。
- 從容地結束串流，以確保所有資料處理都完成。

設定處理常式函數以串流回應

為了向執行期指示 Lambda 應該串流函數的回應，您必須使用 `streamifyResponse()` 裝飾項目包裝函數。這會通知執行期使用適當的邏輯路徑來串流回應，並讓函數串流回應。

`streamifyResponse()` 裝飾項目接受的函數可接受以下參數：

- `event` – 提供函數 URL 調用事件的相關資訊，例如 HTTP 方法、查詢參數和請求內文。
- `responseStream` – 提供可寫入的串流。
- `context` – 提供方法和屬性，以及有關調用、函數以及執行環境的資訊。

responseStream 物件是 Node.js writableStream。與任何此類串流一樣，您應該使用 pipeline() 方法。

Example 啟用回應串流的處理常式

```
const pipeline = require("util").promisify(require("stream").pipeline);
const { Readable } = require('stream');

exports.echo = awslambda.streamifyResponse(async (event, responseStream, _context) => {
  // As an example, convert event to a readable stream.
  const requestStream = Readable.from(Buffer.from(JSON.stringify(event)));

  await pipeline(requestStream, responseStream);
});
```

雖然 responseStream 提供寫入串流的 write() 方法，但仍建議您盡可能使用 pipeline()。使用 pipeline() 可確保可寫入的串流不會被更快的可讀串流所淹沒。

結束串流

請確保在處理常式傳回之前正確結束串流。pipeline() 方法會自動處理這種情形。

對於其他使用案例，請呼叫 responseStream.end() 方法以正確結束串流。此方法發出訊號，表示不應向串流寫入更多資料。如果您使用 pipeline() 或 pipe() 寫入串流，則不需要此方法。

Example 使用 pipeline() 結束串流的範例

```
const pipeline = require("util").promisify(require("stream").pipeline);

exports.handler = awslambda.streamifyResponse(async (event, responseStream, _context)
 => {
  await pipeline(requestStream, responseStream);
});
```

Example 未使用 pipeline() 結束串流的範例

```
exports.handler = awslambda.streamifyResponse(async (event, responseStream, _context)
 => {
  responseStream.write("Hello ");
  responseStream.write("world ");
  responseStream.write("from ");
  responseStream.write("Lambda!");
```

```
  responseStream.end();
});
```

使用 Lambda 函數 URL 調用啟用回應串流的函數

> **Note**
>
> 您必須使用函數 URL 來調用函數，以串流回應。

可以透過變更函數 URL 的調用模式來調用已啟用回應串流的函數。調用模式決定了 Lambda 用來調用函數的 API 操作。可用的調用模式如下：

- BUFFERED – 此為預設選項。Lambda 會使用 Invoke API 操作調用您的函數。承載完成時，即可使用調用結果。承載大小上限為 6 MB。
- RESPONSE_STREAM – 啟用您的函數，當承載結果變得可用時串流它們。Lambda 會使用 InvokeWithResponseStream API 操作調用您的函數。回應承載大小上限為 20 MB。但是，您可以請求增加配額。

您仍然可以透過直接呼叫 Invoke API 操作來調用函數而無需回應串流。不過，Lambda 會串流透過函數 URL 調用的所有回應承載，直到您將調用模式變更為 BUFFERED。

設定函數 URL 的調用模式 (主控台)

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇您要為其設定調用模式的函數名稱。
3. 選擇 Configuration (組態) 標籤，然後選擇 Function URL (函數 URL)。
4. 選擇編輯，然後選擇其他設定。
5. 在調用模式下，選擇所需的調用模式。
6. 選擇儲存。

設定函數 URL 的調用模式 (AWS CLI)

```
aws lambda update-function-url-config --function-name my-function --invoke-mode
 RESPONSE_STREAM
```

設定函數 URL 的調用模式 (AWS CloudFormation)

```
MyFunctionUrl:
  Type: AWS::Lambda::Url
  Properties:
    AuthType: AWS_IAM
    InvokeMode: RESPONSE_STREAM
```

如需設定函數 URL 的詳細資訊，請參閱 Lambda 函數 URL。

回應串流的頻寬限制

函數回應有效負載的前 6 MB 不受頻寬限制。在最初的高頻寬流量後，Lambda 會以最高 2 MBps 的速率串流您的回應。如果您的函數回應一直都沒超過 6 MB，則永遠不會適用此頻寬限制。

> **Note**
>
> 頻寬限制僅適用於函數的回應有效負載，不適用於函數的網路存取。

無頻寬上限的速率取決於諸多因素 (包括函數的處理速度)。一般來說，您可以預期函數回應前 6 MB 的速率會高於 2 Mbps。如果您的函數正在將回應串流至 AWS 以外的目的地，則串流速率也會取決於外部網際網路連線的速度。

教學課程：建立具有函數 URL 的回應串流 Lambda 函數

在本教學課程中，您會建立格式為 .zip 封存檔的 Lambda 函數，其包含的函數 URL 端點會傳回回應串流。如需設定函數 URL 的詳細資訊，請參閱 建立及管理函數 URL。

必要條件

此教學課程假設您具備基本的 Lambda 操作知識並了解 Lambda 主控台。若您尚未了解，請遵循 使用主控台建立一個 Lambda 函數 中的指示，建立您的第一個 Lambda 函數。

若要完成下列步驟，您需要 AWS Command Line Interface (AWS CLI) 版本 2。命令和預期的輸出會列在不同的區塊中：

```
aws --version
```

您應該會看到下列輸出：

```
aws-cli/2.13.27 Python/3.11.6 Linux/4.14.328-248.540.amzn2.x86_64 exe/x86_64.amzn.2
```

對於長命令，逸出字元 (\) 用於將命令分割為多行。

在 Linux 和 macOS 上，使用您偏好的 shell 和套件軟體管理工具。

> **Note**
>
> 在 Windows 中，作業系統的內建終端不支援您常與 Lambda 搭配使用的某些 Bash CLI 命令 (例如 `zip`)。若要取得 Ubuntu 和 Bash 的 Windows 整合版本，請安裝適用於 Linux 的 Windows 子系統。本指南中的 CLI 命令範例使用 Linux 格式。如果您使用的是 Windows CLI，必須重新格式化包含內嵌 JSON 文件的命令。

建立執行角色

建立執行角色，授予您的 Lambda 函數存取 AWS 資源的許可。

若要建立執行角色

1. 開啟 AWS Identity and Access Management (IAM) 主控台的 角色 頁面。
2. 選擇建立角色。
3. 建立具備下列屬性的角色：
   - 信任的實體類型：AWS 服務
   - 使用案例：Lambda
   - 權限 — AWSLambdaBasicExecutionRole
   - 角色名稱 - **`response-streaming-role`**。

該AWSLambdaBasicExecutionRole政策具有函數將日誌寫入 Amazon CloudWatch 日誌所需的許可。建立角色後，請記下其 Amazon Resource Name (ARN)。下一個步驟將需要此值。

建立回應串流函數 (AWS CLI)

使用 AWS Command Line Interface (AWS CLI) 建立具有函數 URL 端點的回應串流 Lambda 函數。

建立可串流回應的函數

1. 將下列程式碼範例複製至名為 `index.mjs` 的檔案中。

```
import util from 'util';
import stream from 'stream';
const { Readable } = stream;
const pipeline = util.promisify(stream.pipeline);

/* global awslambda */
export const handler = awslambda.streamifyResponse(async (event, responseStream,
 _context) => {
  const requestStream = Readable.from(Buffer.from(JSON.stringify(event)));
  await pipeline(requestStream, responseStream);
});
```

2. 建立部署套件。

```
zip function.zip index.mjs
```

3. 使用 `create-function` 命令建立一個 Lambda 函數。使用上一個步驟的角色 ARN 取代 `--role` 的值。

```
aws lambda create-function \
  --function-name my-streaming-function \
  --runtime nodejs16.x \
  --zip-file fileb://function.zip \
  --handler index.handler \
  --role arn:aws:iam::123456789012:role/response-streaming-role
```

建立函數 URL

1. 將資源型政策新增至函數，以允許存取您的函數 URL。將的值取代為您`--principal`的 AWS 帳戶 ID。

```
aws lambda add-permission \
  --function-name my-streaming-function \
  --action lambda:InvokeFunctionUrl \
  --statement-id 12345 \
  --principal 123456789012 \
  --function-url-auth-type AWS_IAM \
  --statement-id url
```

2. 使用 `create-function-url-config` 命令為函數建立 URL 端點。

```
aws lambda create-function-url-config \
  --function-name my-streaming-function \
  --auth-type AWS_IAM \
  --invoke-mode RESPONSE_STREAM
```

測試函數 URL 端點

透過調用函數來測試整合。可以在瀏覽器中開啟函數 URL，也可以使用 curl。

```
curl --request GET "<function_url>" --user "<key:token>" --aws-sigv4 "aws:amz:us-
east-1:lambda" --no-buffer
```

我們的函數 URL 使用 IAM_AUTH 驗證類型。這意味著您需要使用 AWS 訪問密鑰和密鑰簽署請求。在上一個命令中，<key:token>以 AWS 存取金鑰 ID 取代。出現提示時輸入您的 AWS 密鑰。如果您沒有 AWS 密鑰，則可以改用臨時 AWS 憑據。

清除您的資源

除非您想要保留為此教學課程建立的資源，否則您現在便可刪除。刪除您不再使用的 AWS 資源，您可以避免不必要的 AWS 帳戶費用

刪除執行角色

1. 開啟 IAM 主控台中的 角色頁面 。
2. 選取您建立的執行角色。
3. 選擇 刪除 。
4. 在文字輸入欄位中輸入角色的名稱，然後選擇 刪除 。

若要刪除 Lambda 函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇您建立的函數。
3. 選擇 Actions (動作)、Delete (刪除)。
4. 在文字輸入欄位中輸入 **delete**，然後選擇 刪除 。

部署 Lambda 函數

您可以上傳 .zip 封存檔或建立並上傳容器映像，將程式碼部署到 Lambda 函數。

主題

.zip 封存檔

.zip 封存檔包含您的應用程式的程式碼及其相依項。當您使用 Lambda 主控台或工具組撰寫函數時，Lambda 會自動建立程式碼的 .zip 封存檔。

使用 Lambda API、命令列工具或 AWS SDK 建立函數時，您必須建立部署套件。如果您的函數使用編譯的語言，或將相依項新增至函數，您還必須建立部署套件。若要部署函數的程式碼，您可以從 Amazon Simple Storage Service (Amazon S3) 或本機電腦上傳部署套件。

您可以使用 Lambda 主控台 AWS Command Line Interface (AWS CLI) 或將 .zip 檔案做為部署套件上傳到亞馬遜簡單儲存服務 (Amazon S3) 儲存貯體。

部署套件檔案權限

Lambda 執行時間需有許可才能讀取部署套裝服務中的檔案。在 Linux 許可八進位標記法中，Lambda 需要 644 個許可 (rw-r--r--) 用於非可執行檔，以及 755 個許可 (rwxr-x) 用於目錄和可執行檔。

在 Linux 和 MacOS 中，使用 chmod 命令變更部署套件中檔案和目錄的檔案許可。例如，若要提供可執行檔正確的許可，請執行下列命令。

```
chmod 755 <filepath>
```

若要在 Windows 中變更檔案許可，請參閱 Microsoft Windows 文件的 Set, View, Change, or Remove Permissions on an Object。

容器映像

您可以使用 Docker 命令列界面 (CLI) 等工具，將程式碼及相依性封裝為容器映像。然後，您可以將映像上傳至在 Amazon Elastic Container Registry (Amazon ECR) 上託管的容器登錄檔。

當您叫用函數時，Lambda 會將容器映像部署到執行環境。Lambda 會初始化任何擴展功能，然後執行該函數的初始化程式碼 (主處理常式之外的程式碼)。請注意，函數初始化持續時間包含在計費的執行時間內。

然後，Lambda 會呼叫函數設定中指定的程式碼進入點 (入口點和 CMD 容器映像設定) 來執行函數。

AWS 提供一組開放原始碼基礎映像檔，可用來為函數程式碼建立容器映像檔。您也可以使用其他容器登錄中的替代基本映像檔。 AWS 也提供開放原始碼執行階段用戶端，您可以將其新增至替代基礎映像，以使其與 Lambda 服務相容。

此外，還 AWS 提供了一個運行時界面模擬器，供您使用 Docker CLI 之類的工具在本地測試函數。

> **Note**
>
> 您可以建立每個要與 Lambda 支援的其中一個指令集架構相容的容器映像。Lambda 為每個指令集架構提供基本映像，而且 Lambda 也提供支援這兩種架構的基本映像。
> 您為函數建置的映像必須只以其中一個架構為目標。

封裝為容器映像及部署函數無需額外費用。若調用部署為容器映像的函數，您需要支付調用請求和執行持續時間的費用。您需要支付將容器映像儲存在 Amazon ECR 中產生的相關費用。如需詳細資訊，請參閱 Amazon ECR 定價。

映像安全性

當 Lambda 第一次從其原始來源 (Amazon ECR) 下載容器映像時，會使用已驗證的融合加密方法對容器映像進行最佳化、加密和存放。解密客戶資料所需的所有金鑰均使用客 AWS KMS 戶管理的金鑰進行保護。若要追蹤和稽核客戶受管金鑰的 Lambda 用量，您可以檢視 AWS CloudTrail 日誌。

以 .zip 封存檔形式部署 Lambda 函數

當您建立 Lambda 函數時，可以將函數程式碼封裝到部署套件中。Lambda 支援兩種類型的部署套件：容器映像和 .zip 封存檔。建立函數的工作流程取決於部署套件類型。若要設定一個定義為容器映像的函數，請參閱the section called “容器映像”。

可使用 Lambda 主控台和 Lambda API 來建立以 .zip 封存檔定義的函數。也可以上傳更新的 .zip 檔案來變更函數程式碼。

> **Note**
>
> 您無法變更現有函數的部署套件類型 (.zip 或容器映像檔)。例如，您無法將容器映像函數轉換為使用 .zip 檔案封存。您必須建立新的函數。

主題

建立函數

當您建立以 .zip 封存檔定義的函數時，您可以選擇函數的程式碼範本、語言版本和執行角色。Lambda 建立函數後，您可以新增函數程式碼。

建立函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇 建立函數。
3. 選擇 Author from scratch (從頭開始編寫) 或 Use a blueprint (使用藍圖) 來建立函數。
4. 在 基本資訊 下，請執行下列動作：

a. 針對 函數名稱，輸入函數名稱。函數名稱的長度限制為 64 個字元。

b. 對於執行時間，選擇函數要使用的語言版本。

c. (選用) 對於 Architecture (架構)，選擇要用於函數的指令集架構。預設架構值為 x86_64。為您的函數建置部署套件時，確定它與此指令集架構相容。

5. (選用) 在 許可 下，展開 變更預設執行角色 。您可建立新的執行角色，或使用現有的角色。

6. (選用) 展開 Advanced settings (進階設定)。您可對函數選擇程式碼簽署組態。您也可以為函數設定 (Amazon VPC) 進行存取。

7. 選擇建立函數。

Lambda 建立新函數。您現在可以使用主控台來新增函數程式碼，並設定其他函數參數與功能。如需程式碼部署指示，請參閱函數所用執行時間的處理常式頁面。

Node.js

使用 .zip 封存檔部署 Node.js Lambda 函數

Python

使用 .zip 封存檔部署 Python Lambda 函數

Ruby

使用 Ruby Lambda 函數的 .zip 封存檔

Java

使用 .zip 或 JAR 封存檔部署 Java Lambda 函數

Go

使用 .zip 封存檔部署 Go Lambda 函數

C#

使用 .zip 封存檔建置和部署 C# Lambda 函數

PowerShell

使用 .zip 檔案封存部署 PowerShell Lambda 函數

使用主控台程式碼編輯器

主控台將建立具有單一來源檔案的 Lambda 函數。對於指令碼語言，您可以使用內建的程式碼編輯器編輯該檔案並新增更多檔案。選擇 Save (儲存) 以儲存變更。然後，若要執行程式碼，請選擇 Test (測試)。

> **Note**
>
> Lambda 主控台用 AWS Cloud9 來在瀏覽器中提供整合式開發環境。您也可以使用 AWS Cloud9 在自己的環境中開發 Lambda 函數。若要取得更多資訊，請參閱使用指南 AWS 工具組中的〈使用 AWS Lambda 函數〉。 AWS Cloud9

當您儲存函數程式碼時，Lambda 主控台會建立 .zip 封存檔部署套件。當您在主控台之外開發函數程式碼 (使用 IDE) 時，您需要建立部署套件將您的程式碼上傳到 Lambda 函數。

更新函數程式碼

對於指令碼語言 (Node.js、Python 和 Ruby)，您可以在內嵌的程式碼編輯器中編輯函數程式碼。如果程式碼大於 3MB，或如果您需要新增程式庫，或對於編輯器不支援的語言 (Java、Go、C#)，必須將函數程式碼上傳為 .zip 封存。如果 .zip 封存檔小於 50 MB，您可以從本機電腦上傳 .zip 封存檔。如果此檔案大於 50 MB，請將該檔案從 Amazon S3 儲存貯體上傳至函數。

若要將函數程式碼上傳為 .zip 封存

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇要更新的函數並選擇 Code (程式碼) 索引標籤。
3. 在程式碼來源下，選擇上傳自。
4. 選擇 .zip file (.zip 檔案)，然後選擇 Upload (上傳)。
   - 在檔案選擇器中，選取新的映像版本、選擇 Open (開啟)，然後選擇 Save (儲存)。
5. (替代步驟 4) 選擇 Amazon S3 location (Amazon S3 位置)。
   - 在文字方塊中，輸入 .zip 封存檔的 S3 連結 URL，然後選擇 Save (儲存)。

變更執行階段

如果您將函數組態更新為使用新的執行階段，則可能需要更新函數程式碼，才能與新執行階段相容。如果您將函數組態更新為使用不同的執行時間，則必須提供與執行時間和架構相容的新函數程式碼。如需如何為函數程式碼建立部署套件的指示，請參閱函數所使用之執行時間的處理常式頁面。

變更執行階段

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇要更新的函數並選擇 Code (程式碼) 索引標籤。
3. 向下捲動到 Runtime settings (執行時間設定) 區段 (在程式碼編輯器下)。
4. 選擇編輯。
    a. 請在 Runtime (執行階段) 選取執行階段識別符。
    b. 對於 Handler (處理常式)，指定函數的處理常式。
    c. 對於 Architecture (架構)，選擇要用於函數的指令集架構。
5. 選擇儲存。

變更架構

在可以變更指令集架構之前，您需要確保函數的程式碼與目標架構相容。

如果您使用 Node.js、Python 或 Ruby，並在內嵌的編輯器中編輯您的函數程式碼，則現有的程式碼可能無需修改即可執行。

不過，如果您使用 .zip 封存檔部署套件來提供函數程式碼，則必須準備新的 .zip 封存檔，該封存檔會針對目標執行時間和指令集架構正確地進行編譯和建置。如需指示，請參閱函數執行時間的處理常式頁面。

變更指令集架構

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇要更新的函數並選擇 Code (程式碼) 索引標籤。
3. 在 Runtime settings (執行時間設定) 中，選擇 Edit (編輯)。
4. 對於 Architecture (架構)，選擇要用於函數的指令集架構。
5. 選擇儲存。

使用 Lambda API

若要建立及設定使用 .zip 封存檔的函數，請使用下列 API 操作：

- CreateFunction
- UpdateFunctionCode
- UpdateFunctionConfiguration

AWS CloudFormation

您可以使 AWS CloudFormation 用建立使用 .zip 檔案封存的 Lambda 函數。在 AWS CloudFormation 範本中，`AWS::Lambda::Function`資源會指定 Lambda 函數。若要取得`AWS::Lambda::Function`資源中屬性的描述，請參閱《AWS CloudFormation 使用指南》AWS::Lambda::Function中的〈〉。

在 `AWS::Lambda::Function` 資源中，設定下列屬性，以建立定義為 .zip 封存檔的函數：

- AWS::Lambda::Function
  - PackageType — 設定為`Zip`。
  - 程式碼 — 在 `S3Bucket` 和 `S3Key` 欄位中輸入 Amazon S3 儲存貯體名稱和 .zip 檔案名稱。對於 Node.js 或 Python，您可以提供 Lambda 函數的內嵌原始碼。
  - 執行時間 — 設定執行時間值。
  - 架構 — 將架構值設定為使`arm64`用 AWS 重力 2 處理器。依預設，架構值為 `x86_64`。

使用容器映像檔建立 Lambda 函數

你的 AWS Lambda 函數的代碼由腳本或編譯的程序及其依賴關係組成。使用部署套件將函數程式碼部署到 Lambda。Lambda 支援兩種類型的部署套件：容器映像和 .zip 封存檔。

您可以透過三種方式為 Lambda 函數建置容器映像：

- 使用 Lambda 的 AWS 基本映像

  AWS 基礎映像會預先載入語言執行期、用來管理 Lambda 與函數程式碼之間互動的執行期界面用戶端，以及用於本機測試的執行期界面模擬器。

- 使用 AWS 僅限作業系統的基本影像

  AWS 僅限作業系統的基本映像檔包含 Amazon Linux 散發和執行階段介面模擬器。這些映像常用於為編譯語言 (如 Go 和 Rust) 和 Lambda 不提供基礎映像的語言或語言版本 (如 Node.js 19) 建置容器映像。您還可以使用僅限作業系統的基礎映像來實作自訂執行期。若要使映像檔與 Lambda 相容，您必須在映像中加入您語言的 執行期介面用戶端。

- 使用非AWS 基本圖像

  您可以使用其他容器登錄檔中的替代基礎映像 (例如 Alpine Linux 或 Debian)。您也可以使用組織建立的自訂映像。若要使映像檔與 Lambda 相容，您必須在映像中加入您語言的 執行期介面用戶端。

> **Tip**
>
> 若要縮短 Lambda 容器函數變成作用中狀態所需的時間，請參閱 Docker 文件中的使用多階段建置。若要建置有效率的容器映像，請遵循撰寫 Dockerfiles 的最佳實務。

若要使用容器映像建立 Lambda 函數，請在本機建置映像，然後將映像上傳到 Amazon Elastic Container Registry (Amazon ECR) 儲存庫。接著，請在建立函數時指定儲存庫 URI。Amazon ECR 儲存庫必須與 Lambda 函數位於 AWS 區域 相同。只要映像與 Lambda 函數位於相同的區域，您就可以使用不同 AWS 帳戶中的映像檔建立函數。如需詳細資訊，請參閱 Amazon ECR 跨帳戶許可。

本頁面會說明建立 Lambda 相容容器映像檔的基礎映像類型和要求。

> **Note**
>
> 您無法變更現有函數的部署套件類型 (.zip 或容器映像檔)。例如，您無法將容器映像函數轉換為使用 .zip 檔案封存。您必須建立新的函數。

主題

要求

安裝 AWS Command Line Interface (AWS CLI) 版本 2 和 Docker CLI。此外也請注意下列請求：

- 該容器映像必須實作 Lambda 執行階段 API。 AWS 開放原始碼 執行期界面用戶端會實作 API。您可以將執行期介面用戶端新增至您的偏好基礎映像中，以使其與 Lambda 相容。
- 容器映像必須能夠在僅唯讀檔案系統上執行。您的函數程式碼可以存取具有 512 MB 和 10,240 MB 儲存空間的可寫入 `/tmp` 目錄，增量為 1 MB。
- 預設 Lambda 使用者必須能夠讀取執行函數程式碼所需的所有檔案。Lambda 透過定義具有最低權限許可的預設 Linux 使用者來遵守安全最佳實務。確認您的應用程式的程式碼不依賴其他 Linux 使用者無法執行的檔案。
- Lambda 僅支援 Linux 容器映像。
- Lambda 會提供多架構基礎映像。不過，您為函數建置的映像必須只以其中一個架構為目標。Lambda 不支援使用多架構容器映像的函數。

使用 Lambda 的 AWS 基本映像

您可以使用 Lambda 的其中一個 AWS 基礎映像 來建置函數程式碼的容器映像。基礎映像會預先載入語言執行期，以及在 Lambda 上執行容器映像所需的其他元件。您可以將函數程式碼和相依項新增至基礎映像，然後將其封裝為容器映像。

AWS 定期提供 Lambda 基 AWS 礎映像檔的更新。如果您的 Dockerfile 包含 FROM 屬性的映像名稱，則您的 Docker 用戶端會從 Amazon ECR 儲存庫 提取最新版本的映像。若要使用更新的基礎映像，必須重建容器映像並更新函數程式碼。

該 Node.js 20, Python 3.12, Java 21, AL2023, 和更高版本的基礎映像基於 Amazon Linux 2023 最小容器映像. 早期的基本映像使用 Amazon Linux 2。與 Amazon Linux 2 相比，AL2023 具有多項優點，包括更小的部署足跡和更新版本的程式庫，如 `glibc`。

基於 Al2023 的圖像使用`microdnf`（符號鏈接為`dnf`）作為軟件包管理器而不是`yum`，這是 Amazon Linux 2 中的默認軟件包管理器。 `microdnf`是的獨立實作`dnf`。如需以 AL2023 為基礎之映像檔中包含的套件清單，請參閱比較 Amazon Linux 2023 容器映像上安裝的套件中的最小容器欄。如需有關 AL2023 和 Amazon Linux 2 之間差異的詳細資訊，請參閱 AWS 運算部落格 AWS Lambda上的介紹 Amazon Linux 2023 執行階段。

> **Note**
>
> 要在本地運行基於 AL2023 的映像，包括使用 AWS Serverless Application Model （ AWS SAM ），您必須使用碼頭版本 20.10.10 或更高版本。

若要使用 AWS 基本映像檔建立容器映像檔，請選擇您偏好語言的指示：

- Node.js
- TypeScript（使用一個 Node.js 的基本圖像）
- Python
- Java
- Go
- .NET
- Ruby

使用 AWS 僅限作業系統的基本影像

AWS 僅限作業系統的基本映像檔包含 Amazon Linux 散發和執行階段介面模擬器。這些映像常用於為編譯語言 (如 Go 和 Rust) 和 Lambda 不提供基礎映像的語言或語言版本 (如 Node.js 19) 建置容器映像。您還可以使用僅限作業系統的基礎映像來實作自訂執行期。若要使映像檔與 Lambda 相容，您必須在映像中加入您語言的 執行期介面用戶端。

標籤	執行期	作業系統	Dockerfile	棄用
al2023	僅限作業系統的執行期	Amazon Linux 2023	僅適用於 OS 的運行時的碼頭文件 GitHub	
al2	僅限作業系統的執行期	Amazon Linux 2	僅適用於 OS 的運行時的碼頭文件 GitHub	

Amazon Elastic Container Registry 公有資源庫：gallery.ecr.aws/lambda/provided

使用非AWS 基本圖像

Lambda 支援符合下列其中一種映像資訊清單格式的任何映像：

- Docker 映像資訊清單 V2，結構描述 2 (需搭配 1.10 或更新版本的 Docker 使用)
- 開放容器計劃 (OCI) 規範 (v1.0.0 及以上版本)

Lambda 支援的未壓縮影像大小上限為 10 GB，包括所有圖層。

> **Note**
>
> 若要使映像檔與 Lambda 相容，您必須在映像中加入您語言的 執行期介面用戶端。

執行期介面用戶端

如果您使用僅限作業系統的基礎映像或替代的基礎映像，則必須在映像中加入執行期介面用戶端。執行階段介面用戶端必須擴充Lambda 執行階段 API，以管理 Lambda 與函數程式碼之間的互動。 AWS 提供下列語言的開放原始碼執行階段介面用戶端

- Node.js

- Python
- Java
- .NET
- Go
- Ruby
- Rust – Rust 執行期用戶端是實驗性套件。它可能會發生變更，僅用於評估目的。

如果您使用的語言沒有 AWS提供的運行時界面客戶端，則必須創建自己的語言。

Amazon ECR 許可

使用容器映像建立 Lambda 函數之前，您必須先在本機建置映像，並將其上傳至 Amazon ECR 儲存庫。建立函數時，請指定 Amazon ECR 儲存庫 URI。

請確定建立函數之使用者或角色的權限包含`GetRepositoryPolicy`和`SetRepositoryPolicy`。

例如，使用 IAM 主控台建立具有下列政策的角色：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "VisualEditor0",
      "Effect": "Allow",
      "Action": [
        "ecr:SetRepositoryPolicy",
        "ecr:GetRepositoryPolicy"
      ],
      "Resource": "arn:aws:ecr:us-east-1:111122223333:repository/hello-world"
    }
  ]
}
```

Amazon ECR 儲存庫政策

若為與 Amazon ECR 中的容器映像位於相同帳戶中的函數，您可以將 `ecr:BatchGetImage` 和 `ecr:GetDownloadUrlForLayer` 許可新增至 Amazon ECR 儲存庫政策。以下範例顯示最低政策：

```
{
```

```
        "Sid": "LambdaECRImageRetrievalPolicy",
        "Effect": "Allow",
        "Principal": {
          "Service": "lambda.amazonaws.com"
        },
        "Action": [
          "ecr:BatchGetImage",
          "ecr:GetDownloadUrlForLayer"
        ]
    }
```

如需 Amazon ECR 儲存庫許可的詳細資訊，請參閱《Amazon Elastic Container Registry 使用者指南》中的私有儲存庫政策。

如果 Amazon ECR 儲存庫不包含這類許可，Lambda 會將 `ecr:BatchGetImage` 和 `ecr:GetDownloadUrlForLayer` 新增至容器映像儲存庫許可。只有當呼叫 Lambda 的委託人擁有 `ecr:getRepositoryPolicy` 和 `ecr:setRepositoryPolicy` 許可時，Lambda 才會新增這些許可。

若要檢視或編輯 Amazon ECR 儲存庫許可，請遵循《Amazon Elastic Container Registry 使用指南》中設定私有儲存庫政策聲明中的指示。

Amazon ECR 跨帳戶許可

相同區域中的不同帳戶可以建立使用您帳戶擁有的容器映像之函數。在下列範例中，您的 Amazon ECR 儲存庫許可政策需要下列陳述式才能為帳戶編號 123456789012 授予存取權。

- CrossAccount權限 — 允許帳戶 123456789012 建立和更新使用此 ECR 儲存庫映像的 Lambda 函數。
- L@@ ambdaecr ImageCross AccountRetrieval 政策 — 如果長時間未叫用函數，Lambda 最終會將函數的狀態設定為非作用中。必須提供此陳述式，讓 Lambda 可以擷取容器映像來最佳化，並代表 123456789012 所擁有的函數進行快取。

Example - 將跨帳戶許可新增至儲存庫

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "CrossAccountPermission",
      "Effect": "Allow",
```

```
      "Action": [
        "ecr:BatchGetImage",
        "ecr:GetDownloadUrlForLayer"
      ],
      "Principal": {
        "AWS": "arn:aws:iam::123456789012:root"
      }
    },
    {
      "Sid": "LambdaECRImageCrossAccountRetrievalPolicy",
      "Effect": "Allow",
      "Action": [
        "ecr:BatchGetImage",
        "ecr:GetDownloadUrlForLayer"
      ],
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Condition": {
        "StringLike": {
          "aws:sourceARN": "arn:aws:lambda:us-east-1:123456789012:function:*"
        }
      }
    }
  ]
}
```

若要授與多個帳戶的存取權，您需將帳戶 ID 新增至 CrossAccountPermission 政策中的主體清單，也要新增至 LambdaECRImageCrossAccountRetrievalPolicy 中的條件評估清單。

如果您正在使用 AWS 組織中的多個帳戶，建議您在 ECR 權限原則中列舉每個帳戶 ID。此方法符合在 IAM 政策中設定狹窄許可的 AWS 安全性最佳做法。

除了 Lambda 權限之外，建立函數的使用者或角色也必須具有BatchGetImage和GetDownloadUrlForLayer權限。

函數生命週期

上傳新增或更新的容器映像之後，Lambda 會先最佳化該映像，函數才能處理呼叫。最佳化程序可能需要幾秒鐘。該函數會保持 Pending 狀態，直至程序完成。函數隨後會轉換為 Active 狀態。狀態為 Pending 時，您可以叫用該函數，但該函數的其他操作會失敗。映像更新進行中時若發生叫用，則會執行上一個映像的程式碼。

如果函數在多個星期未被叫用，Lambda 會回收其最佳化版本，並將函數轉換為 `Inactive` 狀態。若要重新啟用函數，您必須叫用它。Lambda 拒絕第一次叫用，並且該函數進入 `Pending` 狀態，直到 Lambda 重新最佳化映像。函數隨後會傳回 `Active` 狀態。

Lambda 會定期從 Amazon ECR 儲存庫擷取關聯的容器映像檔。如果對應的容器映像不再存在於 Amazon ECR 或是已撤銷許可，該函數會進入 `Failed` 狀態，並且針對任何函數叫用 Lambda 都會傳回失敗。

您可以使用 Lambda API 來取得函數狀態的相關資訊。如需更多詳細資訊，請參閱 Lambda 函數狀態。

了解 Lambda 函數叫用方法

部署 Lambda 函數之後，您可以透過數種方式呼叫它：

- Lambda 主控台 — 使用 Lambda 主控台快速建立測試事件以叫用您的函數。
- AWS SDK — 使用 SDK 以程式設計方式叫用您的函數。 AWS
- 叫用 API — 使用 Lambda 叫用 API 直接叫用您的函數。
- 的 AWS Command Line Interface （ AWS CLI ）-使用命`aws lambda invoke` AWS CLI 令直接從命令行調用您的函數。
- 函數 URL HTTP (S) 端點 — 使用函數 URL 建立可用來叫用函數的專用 HTTP (S) 端點。

所有這些方法都是調用函數的直接方法。在 Lambda 中，常見的使用案例是根據應用程式中其他位置發生的事件叫用函數。某些服務可以在每個新事件中叫用 Lambda 函數。這就是所謂的觸發器。對於以串流和佇列為基礎的服務，Lambda 會使用批次記錄叫用函數。這稱為事件來源對應。

當您調用函式時，您可以選擇以同步或非同步方式進行調用。使用同步調用，您會等待函式處理事件並傳回回應。使用非同步調用，Lambda 會將事件排入佇列以進行處理，並立即傳回回應。叫用 API 中的`InvocationType`要求參數會決定 Lambda 呼叫函數的方式。的值`RequestResponse`表示同步叫用，值`Event`表示非同步叫用。

如果函數調用導致錯誤，則對於同步調用，請在響應中查看錯誤消息，然後手動重試調用。對於非同步叫用，Lambda 會自動處理重試，並可將叫用記錄傳送至目的地。

同步調用

當您以同步方式調用函數時，Lambda 會執行函數並等候回應。當函數執行完成時，Lambda 會從函數的程式碼傳回回應，其中包含調用的函數版本等額外資料。若要透過 AWS CLI 以同步方式調用函式，請使用 `invoke` 命令。

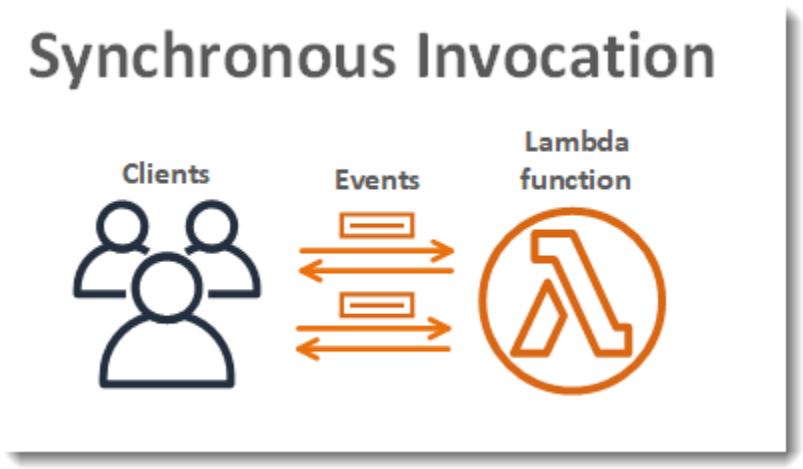
```
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{ "key": "value" }' response.json
```

如果您使用 AWS CLI 第 2 版，則需要 cli-binary-format 選項。若要讓此成為預設的設定，請執行 `aws configure set cli-binary-format raw-in-base64-out`。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

您應該會看到下列輸出：

```
{
    "ExecutedVersion": "$LATEST",
    "StatusCode": 200
}
```

下圖顯示以同步方式調用 Lambda 函數的用戶端。Lambda 會將事件直接傳送到函數，並將函數的回應傳送回調用者。



`payload` 是包含 JSON 格式事件的字串。AWS CLI 從該函數寫入回覆的檔案名稱是 `response.json`。如果函數返回一個對象或錯誤，響應主體是 JSON 格式的對象或錯誤。如果函數沒有錯誤地退出，則響應主體是`null`。

> **Note**
>
> 在傳送回應之前，Lambda 不會等待外部擴充功能完成。外部延伸項目會在執行環境中做為獨立的處理序執行，並在函數調用完成之後繼續執行。如需詳細資訊，請參閱 使用 Lambda 擴充功能擴充功能擴充。

命令的輸出會顯示於終端機，包括來自 Lambda 的回應中標頭中的資訊。這包括處理事件的版本 (當您使用別名時很實用)，以及 Lambda 傳回的狀態碼。如果 Lambda 能夠執行函數，則狀態碼為 200，即使函數傳回了錯誤。

> **Note**
>
> 對於逾時很久的函式，您的用戶端可能在等待回應的同時，在同步調用期間中斷連線。設定您的 HTTP 用戶端、SDK、防火牆、Proxy 或作業系統，以透過逾時或持續作用設定允許長時間連線。

如果 Lambda 無法執行函數，錯誤則會顯示在輸出中。

```
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload value response.json
```

您應該會看到下列輸出：

```
An error occurred (InvalidRequestContentException) when calling the Invoke operation:
 Could not parse request body into json: Unrecognized token 'value': was expecting
 ('true', 'false' or 'null')
 at [Source: (byte[])"value"; line: 1, column: 11]
```

AWS CLI 是開放原始碼工具，可讓您在命令列 shell 中使用命令來與 AWS 服務互動。若要完成本節中的步驟，您必須執行下列各項：

- AWS Command Line Interface (AWS CLI) 版本 2
- AWS CLI – 使用 `aws configure` 進行快速組態設定

您可以透過 AWS CLI，使用 `--log-type` 命令選項來擷取要調用的日誌。其回應將包含 `LogResult` 欄位，內含該次調用的 base64 編碼日誌 (最大達 4 KB)。

Example 擷取日誌 ID

下列範例顯示如何從名稱為 my-function 的函數的 LogResult 欄位來擷取日誌 ID。

```
aws lambda invoke --function-name my-function out --log-type Tail
```

您應該會看到下列輸出：

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example 解碼日誌

在相同的命令提示中，使用 base64 公用程式來解碼日誌。下列範例顯示如何擷取 my-function 的 base64 編碼日誌。

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

如果您使用 AWS CLI 第 2 版，則需要 cli-binary-format 選項。若要讓此成為預設的設定，請執行 aws configure set cli-binary-format raw-in-base64-out。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

您應該會看到下列輸出：

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

該 base64 公用程式可在 Linux、macOS 和 Ubuntu on Windows 上使用。macOS 使用者可能需要使用 base64 -D。

如需有關 Invoke API 的詳細資訊 (包括參數、標頭和錯誤的完整清單)，請參閱調用。

當您直接調用函式時，您可以檢查回應中是否有錯誤並重試。AWS CLI 和 AWS SDK 也會在用戶端逾時、調節和服務錯誤時自動重試。如需更多詳細資訊，請參閱 了解 Lambda 中的重試行為。

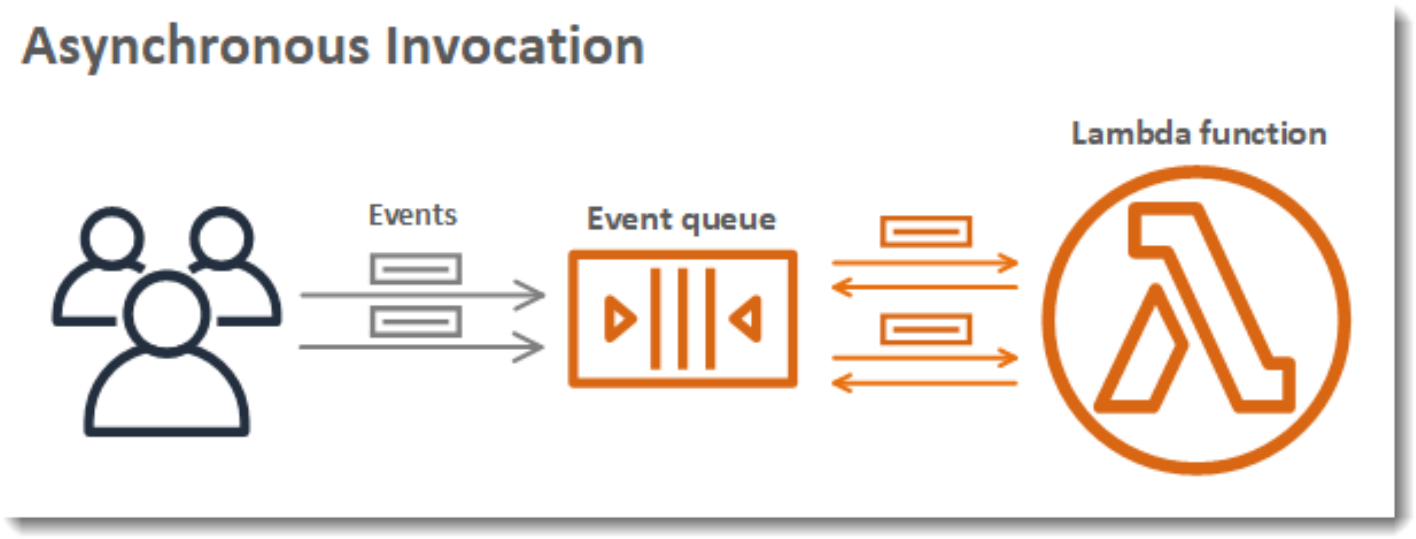
非同步調用

幾個 AWS 服務例如亞馬遜簡單儲存服務 (Amazon S3) 和亞馬遜簡單通知服務 (Amazon SNS)) 會以非同步方式叫用函數來處理事件。當您以非同步方式呼叫函數時，您不需要等待來自函數程式碼的回應。您可以將事件傳遞給 Lambda，而 Lambda 會處理其餘的工作。您可以設定 Lambda 處理錯誤的方式，並可以將叫用記錄傳送到下游資源，例如 Amazon Simple Queue Service (Amazon SQS) 或 Amazon EventBridge (EventBridge)，以將應用程式的元件鏈結在一起。

章節

Lambda 如何處理非同步調用

下圖顯示以非同步方式來調用 Lambda 函數的用戶端。Lambda 會先將事件排入佇列，再將事件傳送到函數。



針對非同步調用，Lambda 會將事件置放在佇列中，並傳回成功回應，其中不包含其他資訊。單獨的程序會從佇列讀取事件，並將事件傳送到您的函數。若要以非同步方式調用函式，請將調用類型參數設定為 `Event`。

```
aws lambda invoke \
  --function-name my-function  \
  --invocation-type Event \
  --cli-binary-format raw-in-base64-out \
  --payload '{ "key": "value" }' response.json
```

如果您使用的是 AWS CLI 版本 2，則需要此cli-binary-format選項。若要讓此成為預設的設定，請執行 `aws configure set cli-binary-format raw-in-base64-out`。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

```
{
    "StatusCode": 202
}
```

輸出檔 (`response.json`) 不包含任何資訊，但仍會在您執行此命令時建立。如果 Lambda 無法將事件新增到佇列，錯誤訊息就會顯示在命令輸出中。

Lambda 會管理函數的非同步事件佇列，並在發生錯誤時嘗試重試。如果函數傳回錯誤，則 Lambda 會嘗試多執行函數兩次，且兩次嘗試之間等候一分鐘，而第二次和第三次嘗試之間等候兩分鐘。函式錯誤包含函式程式碼所傳回的錯誤，以及函式執行時間所傳回的錯誤，例如逾時。

如果函式沒有足夠的並行可用來處理所有事件，則額外請求會遭到調節。對於調節錯誤 (429) 和系統錯誤 (500 序列)，Lambda 會將事件傳回到佇列，並嘗試再次執行函數長達 6 小時。重試間隔從第一次嘗試後的 1 秒呈指數增加到最多 5 分鐘。如果佇列包含許多項目，Lambda 會增加重試間隔，並降低從佇列讀取事件的速率。

即使您的函數並未傳回錯誤，它也可以從 Lambda 收到相同的事件很多次，因為佇列本身最終一致。如果函式來不及處理傳入事件，也可以從佇列中刪除事件，而不需傳送到函式。確保您的函式程式碼可從容地處理重複的事件，而且您有足夠的並行可用來處理所有調用。

當佇列很長時，新的事件可能會在 Lambda 有機會將其傳送到函數前就已過期。當事件過期時或所有處理嘗試皆失敗時，Lambda 便會捨棄該事件。您可以設定函數的錯誤處理，以減少 Lambda 執行的重試次數，或更快地捨棄未處理的事件。

您也可以設定 Lambda，將調用記錄傳送到另一個服務。Lambda 支持以下目的地以進行非同步調用。請注意，不支援 SQS FIFO 佇列和 SNS FIFO 主題。

- Amazon SQS - 標準 SQS 佇列。
- Amazon SNS – 標準 SNS 主題。

- AWS Lambda - Lambda 函數。
- Amazon EventBridge- EventBridge 事件總線。

呼叫記錄包含請求和回應 (JSON 格式) 的詳細資訊。您可以為成功處理的事件及所有處理嘗試皆失敗的事件設定個別目標。或者，您可以將標準 Amazon SQS 佇列或標準 Amazon SNS 主題設為捨棄事件的無效字母佇列。針對無效字母佇列，Lambda 只會傳送事件的內容，而不包含回應的詳細資訊。

如果 Lambda 無法將記錄傳送到您已設定的目的地，則會將`DestinationDeliveryFailures`指標傳送至 Amazon CloudWatch。如果您的組態中包含不受支援的目的地類型 (例如 Amazon SQS FIFO 佇列或 Amazon SNS FIFO 主題)，就可能發生這種情形。傳遞錯誤也可能因許可錯誤和大小限制而發生。如需 Lambda 調用指標的詳細資訊，請參閱：呼叫指標

> **Note**
>
> 為了防止函數觸發，您可以將函數的預留並行設為零。當您將非同步調用函數的預留並行設定為零時，Lambda 會開始將新事件傳送至設定的無效字母佇列或失敗時的事件目的地，不會進行任何重試。若要處理在預留並行設定為零時傳送的事件，您必須使用來自無效字母佇列或失敗時的事件目的地之事件。

設定非同步調用的錯誤處理

使用 Lambda 主控台設定函數、版本或別名的錯誤處理設定。

設定錯誤處理

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇 組態，然後選擇 非同步調用。
4. 在 非同步調用 下方，選擇 編輯。
5. 進行下列設定。
   - Maximum age of event (事件存留期上限) - Lambda 在非同步事件佇列中保留事件的時間上限，最多 6 小時。
   - Retry attempts (重試嘗試) - 當函數傳回錯誤時，Lambda 重試的次數上限 (介於 0 到 2)。
6. 選擇 儲存 。

當調用事件超過最大存留期，或所有重試嘗試都失敗時，Lambda 會將其捨棄。若要保留已捨棄事件的副本，請設定失敗事件的目的地。

設定非同步調用的目的地

若要保留非同步調用的記錄，請將目的地新增至您的函數。您可以選擇將成功或失敗的調用傳送至目的地。每個函數都可以有多個目的地，因此您可以為成功和失敗的事件配置單獨的目的地。傳送至目的地的每筆記錄都是包含調用之詳細資訊的 JSON 文件。與錯誤處理設定相似，您可以設定函數、函數版本或是別名的目標。

> **Note**
>
> 您也可以保留下列事件來源映射類型失敗叫用的記錄：Amazon Kinesis、AmazonDynamoDB、自我管理的 Apache 卡夫卡和 Amazon MSK。

以下資料表列出針對非同步調用記錄支援的目的地。若要讓 Lambda 成功將記錄傳送至您選擇的目的地，請確定函數的執行角色也包含相關許可。此資料表也說明每個目的地類型如何接收 JSON 調用記錄。

目的地類型	所需的許可	目的地特定 JSON 格式
Amazon SQS 佇列	平方：SendMessage	Lambda 會將調用記錄做為 `Message` 傳遞至目的地。
Amazon SNS 主題	sns:Publish	Lambda 會將調用記錄做為 `Message` 傳遞至目的地。
Lambda 函數	InvokeFunction	Lambda 傳遞調用記錄會以承載形式傳遞給函數。
EventBridge	事件：PutEvents	• Lambda 通過調用記錄作為`detail`在 PutEvents 調用。 • `source` 事件欄位的值是 `lambda`。 • `detail-type` 事件欄位的值是 Lambda Function

目的地類型	所需的許可	目的地特定 JSON 格式
		Invocation Result - Success (Lambda 函數調用結果 - 成功) 或 Lambda Function Invocation Result - Failure (Lambda 函數調用結果 - 失敗)。 • `resource` 事件欄位包含函數和目的地 Amazon Resource Names (ARN)。 • 如需其他事件欄位，請參閱 Amazon EventBridge 事件。

下列範例顯示因函式錯誤而導致處理失敗三次事件的呼叫記錄。呼叫記錄包含事件、回應以及記錄傳送原因的詳細資訊。

```
{
    "version": "1.0",
    "timestamp": "2019-11-14T18:16:05.568Z",
    "requestContext": {
        "requestId": "e4b46cbf-b738-xmpl-8880-a18cdf61200e",
        "functionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function:
$LATEST",
        "condition": "RetriesExhausted",
        "approximateInvokeCount": 3
    },
    "requestPayload": {
        "ORDER_IDS": [
            "9e07af03-ce31-4ff3-xmpl-36dce652cb4f",
            "637de236-e7b2-464e-xmpl-baf57f86bb53",
            "a81ddca6-2c35-45c7-xmpl-c3a03a31ed15"
        ]
    },
    "responseContext": {
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "responsePayload": {
```

```
        "errorMessage": "RequestId: e4b46cbf-b738-xmpl-8880-a18cdf61200e Process exited
 before completing request"
    }
}
```

下列步驟說明如何使用 Lambda 主控台設定函數的目的地。

設定非同步調用記錄的目的地

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 在 函數概觀 下，選擇 新增目的地。
4. 針對 來源，選擇 非同步調用。
5. 如為條件，請從下列選項中選擇：
   - On failure (失敗時) - 當事件的所有處理嘗試都失敗，或超過存留期上限時，傳送記錄。
   - 成功時 - 當函數成功處理非同步調用時傳送記錄。
6. 對於 目的地類型，請選擇接收調用記錄的資源類型。
7. 對於 目的地，請選擇一個資源。
8. 選擇 儲存 。

當調用與條件相符時，Lambda 會將包含調用之詳細資訊的 JSON 文件傳送到目的地。

目的地特定 JSON 格式

- 對於 Amazon SQS 和 Amazon SNS (`SnsDestination` 和 `SqsDestination`)，調用記錄會以 `Message` 形式傳遞到目的地。
- 對於 Lambda (`LambdaDestination`)，調用記錄會以承載形式傳遞給函數。
- For EventBridge (`EventBridgeDestination`)，PutEvents呼叫記錄會以呼叫`detail`中的方式傳遞。`source` 事件欄位的值是 `lambda`。`detail-type` 事件欄位的值是 Lambda Function Invocation Result – Success (Lambda 函數調用結果 – 成功) 或 Lambda Function Invocation Result – Failure (Lambda 函數調用結果 – 失敗)。`resource` 事件欄位包含函數和目的地 Amazon Resource Names (ARN)。如需其他事件欄位，請參閱 Amazon EventBridge 事件。

下列範例顯示因函式錯誤而導致處理失敗三次事件的呼叫記錄。

Example 呼叫記錄

```
{
    "version": "1.0",
    "timestamp": "2019-11-14T18:16:05.568Z",
    "requestContext": {
        "requestId": "e4b46cbf-b738-xmpl-8880-a18cdf61200e",
        "functionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function:
$LATEST",
        "condition": "RetriesExhausted",
        "approximateInvokeCount": 3
    },
    "requestPayload": {
        "ORDER_IDS": [
            "9e07af03-ce31-4ff3-xmpl-36dce652cb4f",
            "637de236-e7b2-464e-xmpl-baf57f86bb53",
            "a81ddca6-2c35-45c7-xmpl-c3a03a31ed15"
        ]
    },
    "responseContext": {
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "responsePayload": {
        "errorMessage": "RequestId: e4b46cbf-b738-xmpl-8880-a18cdf61200e Process exited
 before completing request"
    }
}
```

呼叫記錄包含事件、回應以及記錄傳送原因的詳細資訊。

追蹤傳至目的地的請求

您可以使用 AWS X-Ray 查看每個請求排入佇列、由 Lambda 函數處理並傳遞至目的地服務的連接檢視。為調用函數的函數或服務啟用 X-Ray 追蹤時，Lambda 會將 X-Ray 標頭新增至請求，並將標頭傳送至目的地服務。來自上游服務的追蹤會自動連結至來自下游 Lambda 函數和目標服務的追蹤，以建立整個應用程式的 end-to-end 檢視。如需追蹤的詳細資訊，請參閱：使用視覺化 Lambda 函數叫用 AWS X-Ray。

非同步調用組態 API

若要使用 AWS CLI 或 AWS SDK 管理非同步叫用設定，請使用下列 API 作業。

- PutFunctionEventInvokeConfig
- GetFunctionEventInvokeConfig
- UpdateFunctionEventInvokeConfig
- ListFunctionEventInvoke配置
- DeleteFunctionEventInvokeConfig

若要使用配置非同步呼叫 AWS CLI，請使用指put-function-event-invoke-config令。下列範例會設定事件存留期為 1 小時且不重試的函數。

```
aws lambda put-function-event-invoke-config --function-name error \
--maximum-event-age-in-seconds 3600 --maximum-retry-attempts 0
```

您應該會看到下列輸出：

```
{
    "LastModified": 1573686021.479,
    "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:error:$LATEST",
    "MaximumRetryAttempts": 0,
    "MaximumEventAgeInSeconds": 3600,
    "DestinationConfig": {
        "OnSuccess": {},
        "OnFailure": {}
    }
}
```

此 put-function-event-invoke-config 命令會覆寫任何在函數、版本或別名上的現有組態。若要設定選項而不重設其他項目，請使用 update-function-event-invoke-config。下列範例會將 Lambda 設定為在無法處理事件時，將記錄傳送至名為 destination 的標準 SQS 佇列。

```
aws lambda update-function-event-invoke-config --function-name error \
--destination-config '{"OnFailure":{"Destination": "arn:aws:sqs:us-
east-2:123456789012:destination"}}'
```

您應該會看到下列輸出：

```
{
    "LastModified": 1573687896.493,
    "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:error:$LATEST",
    "MaximumRetryAttempts": 0,
    "MaximumEventAgeInSeconds": 3600,
    "DestinationConfig": {
        "OnSuccess": {},
        "OnFailure": {
            "Destination": "arn:aws:sqs:us-east-2:123456789012:destination"
        }
    }
}
```

無效字母佇列

做為失敗目標的替代項目，您可以設定您的函數，使其具備一個無效字母佇列來儲存捨棄的事件，以供後續處理。無效字母佇列的運作方式與失敗目標相同，會在事件的所有處理嘗試失敗，或是在沒有處理的情況下過期時使用。但是，無效字母佇列是函數版本特定組態的一部分，因此會在您發佈版本時鎖定。失敗目標也支援其他目標，並會在呼叫記錄中包含函數回應的詳細資訊。

若要重新處理無效字母佇列中的事件，請將它設定為 Lambda 函數的事件來源。或者，您可以手動擷取事件。

您可以為無效字母佇列選擇 Amazon SQS 標準佇列或 Amazon SNS 標準主題。不支援 SQS FIFO 佇列和 Amazon SNS FIFO 主題。如果您沒有佇列或主題，請加以建立。選擇符合您的使用案例的目標類型。

- Amazon SQS 佇列 - 佇列會保留失敗的事件，直到其遭到擷取為止。如果您希望單一實體 (例如 Lambda 函數或 CloudWatch警示) 處理失敗的事件，請選擇 Amazon SQS 標準佇列。如需詳細資訊，請參閱 搭配 Amazon SQS 使用 Lambda。

  在 Amazon SQS 主控台中建立佇列。
- Amazon SNS 主題 - 主題會將失敗的事件轉送到一個或多個目的地。如果您希望多個實體對失敗的事件採取動作，請選擇 Amazon SNS 標準主題。例如，您可以設定一個主題，以將事件傳送到電子郵件地址、Lambda 函數及/或 HTTP 端點。如需詳細資訊，請參閱 使用 Amazon SNS 通知叫用 Lambda 函數。

  在 Amazon SNS 主控台中建立主題。

若要將事件傳送到佇列或主題，您的函式需要額外許可。將具有所需許可的政策新增到您函式的執行角色。

- Amazon SQS-平方米：SendMessage
- Amazon SNS–sns:Publish

如果目標佇列或主題使用客戶受管金鑰加密，則執行角色也必須是金鑰以資源為基礎政策中的使用者。

在建立目標及更新函式的執行角色之後，將無效字母佇列新增到您的函式。您可以設定多個函式來將事件傳送到相同的目標。

設定無效字母佇列

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇 組態，然後選擇 非同步調用。
4. 在 非同步調用 下方，選擇 編輯。
5. 將 DLQ 資源設定為 Amazon SQS 或 Amazon SNS。
6. 選擇目標佇列或主題。
7. 選擇 儲存 。

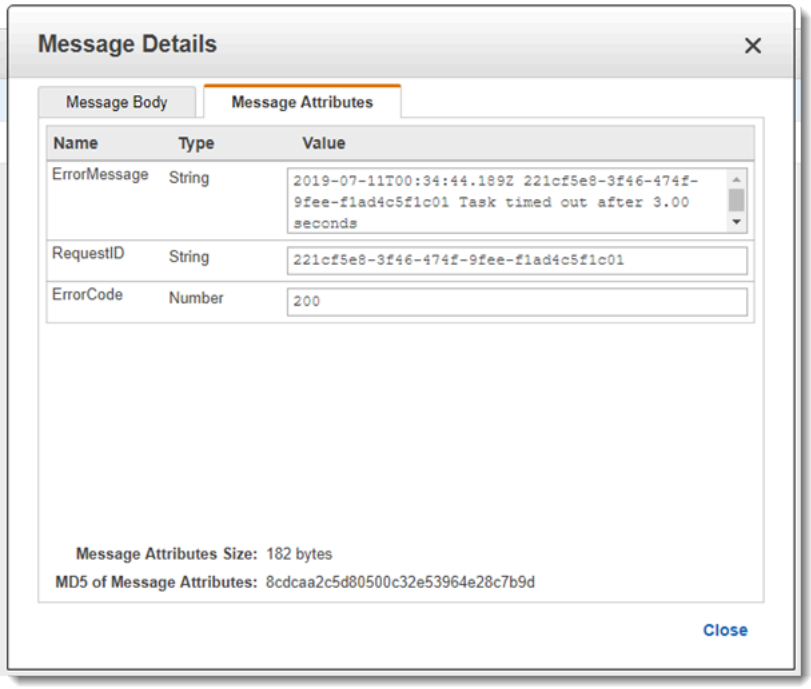
若要使用配置無效字母佇列 AWS CLI，請使用指update-function-configuration令。

```
aws lambda update-function-configuration --function-name my-function \
--dead-letter-config TargetArn=arn:aws:sns:us-east-2:123456789012:my-topic
```

Lambda 會依現狀將事件傳送到無效字母佇列，其屬性中有額外資訊。您可以使用此資訊來識別該函式所傳回的錯誤，或建立事件與日誌或 AWS X-Ray 追蹤的關聯性。

無效字母佇列訊息屬性

- RequestID (字串) - 調用請求的 ID。請求 ID 會出現於函式日誌中。您也可以使用 X-Ray SDK 在追蹤的屬性中記錄請求 ID。然後，您可以在 X-Ray 主控台中依請求 ID 搜尋追蹤。
- ErrorCode(數字) — HTTP 狀態碼。
- ErrorMessage(字串) — 錯誤訊息的前 1 KB。



如果 Lambda 無法將訊息傳送至無效字母佇列，就會刪除事件並發出錯誤量度。DeadLetter這可能由於缺乏許可，或訊息總大小超過目標佇列或主題的限制，而發生此狀況。例如，假設本文大小接近 256 KB 的 Amazon SNS 通知會觸發導致錯誤的函數。在這種情況下，Amazon SNS 新增的事件資料與 Lambda 新增的屬性結合後，可能導致訊息超過無效字母佇列中允許的大小上限。

如果您使用 Amazon SQS 作為事件來源，請對 Amazon SQS 佇列本身 (而非 Lambda 函數) 設定無效字母佇列。如需更多詳細資訊，請參閱 搭配 Amazon SQS 使用 Lambda。

Lambda 如何處理串流和以佇列為基礎的事件來源的記錄

事件來源對應是 Lambda 資源，可從串流和以佇列為基礎的服務讀取項目，並叫用含有大量記錄的函數。下列服務會使用事件來源對應來叫用 Lambda 函數：

- Amazon DynamoDB
- Amazon Kinesis
- Amazon MQ
- Amazon Managed Streaming for Apache Kafka (Amazon MSK)
- 自我管理的 Apache Kafka
- Amazon Simple Queue Service (Amazon SQS)
- Amazon DocumentDB (with MongoDB compatibility) (Amazon DocumentDB)

> **Warning**
>
> Lambda 事件來源對應至少處理每個事件一次，並且可能會重複處理記錄。為了避免與重複事件相關的潛在問題，我們強烈建議您將函數代碼設為冪等。若要深入了解，請參閱 AWS 知識中心如何讓 Lambda 函數具有冪等性。

事件來源對應與直接觸發程式有何不同

某些 AWS 服務可以使用觸發器直接叫用 Lambda 函數。這些服務會將事件推送至 Lambda，並在指定的事件發生時立即叫用函數。觸發程序適用於離散事件和即時處理。當您使用 Lambda 主控台建立觸發器時，主控台會與對應的 AWS 服務互動，以便在該服務上設定事件通知。觸發程序實際上是由產生事件的服務儲存和管理，而不是由 Lambda 進行管理。以下是一些使用觸發程序來叫用 Lambda 函數的服務範例：

- Amazon Simple Storage Service (Amazon S3)：在儲存貯體中建立、刪除或修改物件時叫用函數。如需詳細資訊，請參閱 教學課程：使用 Amazon S3 觸發條件調用 Lambda 函數。
- 亞馬遜簡單通知服務 (Amazon SNS)：在將訊息發佈到 SNS 主題時叫用函數。如需詳細資訊，請參閱 教學課程：搭 AWS Lambda 配 Amazon 簡易通知服務使用。
- Amazon API Gateway：在向特定端點發出 API 請求時叫用函數。如需詳細資訊，請參閱 使用 Amazon API Gateway 端點叫用 Lambda 函數。

事件來源對應是在 Lambda 服務中建立和管理的 Lambda 資源。事件來源對應是專為處理大量串流資料或佇列中的訊息而設計。以批次方式處理串流或佇列中的記錄比個別處理記錄更有效率。

批次處理行為

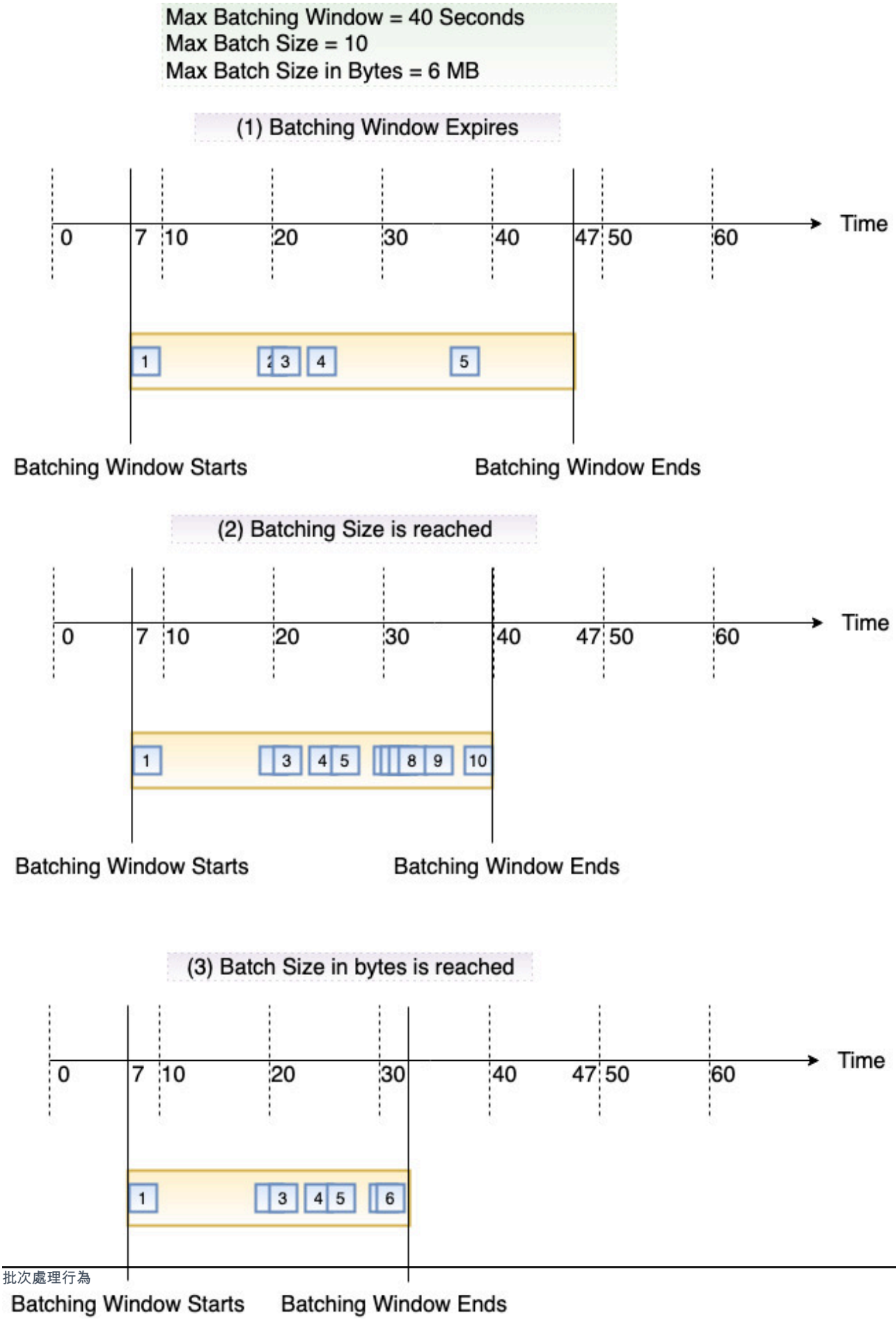
根據預設，事件來源映射會將記錄批次處理到 Lambda 傳送給函數的單個承載中。若要微調批次處理行為，您可以設定批次處理視窗 (MaximumBatchingWindowIn秒) 和批次大小 ()。BatchSize批次間隔是將記錄收集到單一承載的最長時間。批次大小是單一批次中記錄的最大數目。當下列三個條件之一成立時，Lambda 會調用您的函數：

- 批次間隔達到其最大值。預設的批次處理視窗行為會根據特定的事件來源而有所不同。
  - 對於 Kinesis、DynamoDB 和 Amazon SQS 事件來源：預設批次間隔為 0 秒。這意味著 Lambda 只有在滿足批次大小或達到有效負載大小限制時，才會將批次傳送至您的函數。若要設定批次化視窗，請進行設定`MaximumBatchingWindowInSeconds`。您可以將此參數設定為 0 到 300 秒之間的任何值，以 1 秒為增量。如果您設定批次處理視窗，則下一個視窗會在前一個函數呼叫完成後立即開始。
  - 如果事件來源是 Amazon MSK、自主管理 Apache Kafka 、Amazon MQ 以及 Amazon DocumentDB：預設批次處理時段為 500 毫秒。您可以將 `MaximumBatchingWindowInSeconds` 設定為從 0 秒到 300 秒之間的任意值，增量為秒。一旦第一條記錄到達，批次間隔就會開始。

    > **Note**
    >
    > 因為您只能以秒為單位進行變更，因此`MaximumBatchingWindowInSeconds`在變更之後，您無法還原為 500 毫秒預設批次處理視窗。要恢復預設批次間隔，必須建立新的事件來源映射。

- 已滿足批次大小。批次大小下限為 1。預設和最大批次大小取決於事件來源。如需有關這些值的詳細資訊，請參閱 `CreateEventSourceMapping` API 作業的BatchSize規格。
- 承載大小達到 6 MB。您無法修改此限制。

下圖說明了這三種情況。假設批次間隔從第 `t = 7` 秒開始。在第一種情況下，批次間隔在累積 5 條記錄後在第 `t = 47` 秒達到其最大值 40 秒。在第二種情況下，批次大小在批次間隔到期之前達到 10，因此批次間隔提前結束。在第三種情況下，承載大小在批次間隔到期之前達到最大值，因此批次間隔提前結束。


Max Batching Window = 40 Seconds
Max Batch Size = 10
Max Batch Size in Bytes = 6 MB
(1) Batching Window Expires
0
7
10
20
30
40
47
50
60
Time
1
3
4
5
Batching Window Starts
Batching Window Ends
(2) Batching Size is reached
0
7
10
20
30
40
47
50
60
Time
1
3
4
5
8
9
10
Batching Window Starts
Batching Window Ends
(3) Batch Size in bytes is reached
0
7
10
20
30
40
47
50
60
Time
1
3
4
5
6
Batching Window Starts
Batching Window Ends

事件來源映射 API

若要使用 AWS Command Line Interface (AWS CLI) 或 AWS SDK 來管理事件來源，可以使用下列 API 操作：

- CreateEventSourceMapping
- ListEventSourceMappings
- GetEventSourceMapping
- UpdateEventSourceMapping
- DeleteEventSourceMapping

AWS Lambda 搭配 Amazon DynamoDB 使用

> **Note**
>
> 如果您想要將資料傳送到 Lambda 函數以外的目標，或在傳送資料之前豐富資料，請參閱 Amazon EventBridge 管道。

您可以使用 AWS Lambda 函數來處理 Amazon DynamoDB 串流中的記錄。您可以透過 DynamoDB Streams，以在每次更新 DynamoDB 資料表時，觸發 Lambda 函數來執行額外的工作。

Lambda 會從串流讀取記錄，並透過包含串流記錄的事件同步調用函數。Lambda 會讀取批次中的記錄並調用函數，以處理來自該批次的記錄。

章節

- 範例事件
- 輪詢和批次處理串流
- 輪詢和串流開始位置
- DynamoDB Streams 中的碎片同時讀取
- 執行角色許可
- 新增權限並建立事件來源對應
- 錯誤處理
- Amazon CloudWatch 指標
- 時段

- 報告批次項目失敗
- Amazon DynamoDB Streams 組態參數
- 教學課程： AWS Lambda 與 Amazon DynamoDB 串流搭配使用
- 範例函數程式碼
- DynamoDB 應用程式的 AWS SAM 範本

範例事件

Example

```
{
  "Records": [
    {
      "eventID": "1",
      "eventVersion": "1.0",
      "dynamodb": {
        "Keys": {
          "Id": {
            "N": "101"
          }
        },
        "NewImage": {
          "Message": {
            "S": "New item!"
          },
          "Id": {
            "N": "101"
          }
        },
        "StreamViewType": "NEW_AND_OLD_IMAGES",
        "SequenceNumber": "111",
        "SizeBytes": 26
      },
      "awsRegion": "us-west-2",
      "eventName": "INSERT",
      "eventSourceARN": "arn:aws:dynamodb:us-east-2:123456789012:table/my-table/
stream/2024-06-10T19:26:16.525",
      "eventSource": "aws:dynamodb"
    },
    {
      "eventID": "2",
```

```
      "eventVersion": "1.0",
      "dynamodb": {
        "OldImage": {
          "Message": {
            "S": "New item!"
          },
          "Id": {
            "N": "101"
          }
        },
        "SequenceNumber": "222",
        "Keys": {
          "Id": {
            "N": "101"
          }
        },
        "SizeBytes": 59,
        "NewImage": {
          "Message": {
            "S": "This item has changed"
          },
          "Id": {
            "N": "101"
          }
        },
        "StreamViewType": "NEW_AND_OLD_IMAGES"
      },
      "awsRegion": "us-west-2",
      "eventName": "MODIFY",
      "eventSourceARN": "arn:aws:dynamodb:us-east-2:123456789012:table/my-table/
stream/2024-06-10T19:26:16.525",
      "eventSource": "aws:dynamodb"
    }
  ]}
```

輪詢和批次處理串流

Lambda 會輪詢您 DynamoDB 串流中的碎片，其記錄的基本速率為每秒 4 次。當記錄可用時，Lambda 會調用您的函數，並等待結果。如果處理成功，Lambda 會恢復輪詢，直到收到多筆記錄。

Lambda 預設會在記錄可用時立即調用函數。如果 Lambda 從事件來源中讀取的批次只有一筆記錄，Lambda 只會傳送一筆記錄至函數。為避免調用具有少量記錄的函數，您可設定批次間隔，請求事

件來源緩衝記錄最長達五分鐘。調用函數之前，Lambda 會繼續從事件來源中讀取記錄，直到收集到完整批次、批次間隔到期或者批次達到 6 MB 的承載限制。如需詳細資訊，請參閱 批次處理行為。

> **Warning**
>
> Lambda 事件來源對應至少處理每個事件一次，並且可能會重複處理記錄。為了避免與重複事件相關的潛在問題，我們強烈建議您將函數代碼設為冪等。若要深入了解，請參閱 AWS 知識中心如何讓 Lambda 函數具有冪等性。

ParallelizationFactor設定此設定，以同時處理具有多個 Lambda 叫用的 DynamoDB 串流碎片。您可以透過從 1 (預設) 到 10 的並行化因子指定 Lambda 從碎片輪詢的並行批次數。當您增加每個碎片的並行批次數時，Lambda 仍可確保在項目 (分區和排序索引鍵) 層級進行順序處理。

輪詢和串流開始位置

請注意，建立和更新事件來源映射期間的串流輪詢最終會一致。

- 在建立事件來源映射期間，從串流開始輪詢事件可能需要幾分鐘時間。
- 在更新事件來源映射期間，從串流停止並重新開始輪詢事件可能需要幾分鐘時間。

這種行為表示如果您指定 `LATEST` 當作串流的開始位置，事件來源映射可能會在建立或更新期間遺漏事件。若要確保沒有遺漏任何事件，請將串流開始位置指定為 `TRIM_HORIZON`。

DynamoDB Streams 中的碎片同時讀取

對於不是全域資料表的單一區域資料表，您最多可以設計兩個 Lambda 函數，以便同時讀取同一個 DynamoDB Streams 碎片。超過此限制會導致請求調節。對於全域資料表，建議您將同時函數的數量限制為一個，以避免請求限流。

執行角色許可

受AWSLambdaDynamoDBExecutionRole AWS 管政策包括 Lambda 需要從您的 DynamoDB 串流讀取的許可。將此受管原則新增至函數的執行角色。

若要將失敗批次的記錄傳送到標準 SQS 佇列或標準 SNS 主題，您的函數需要額外許可。每個目的地服務都需要不同的許可，如下所示：

- Amazon SQS-平方米：SendMessage
- Amazon SNS–sns:Publish

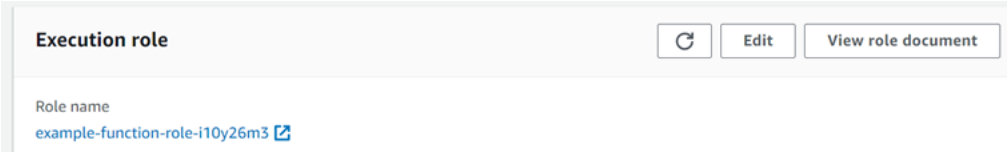
新增權限並建立事件來源對應

建立事件來源映射，指示 Lambda 從您的串流傳送記錄至 Lambda 函數。您可以建立多個事件來源映射，來使用多個 Lambda 函數處理相同資料，或使用單一函數處理來自多個串流的項目。

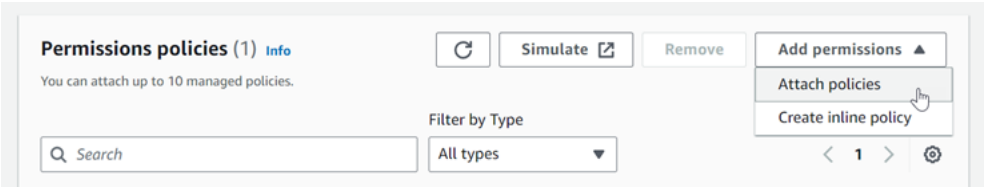
若要將您的函數設定為從 DynamoDB Streams 讀取，請將AWSLambdaDynamoDBExecutionRole AWS 受管理的原則附加至您的執行角色，然後建立 DynamoDB 觸發器。

若要新增權限並建立觸發器

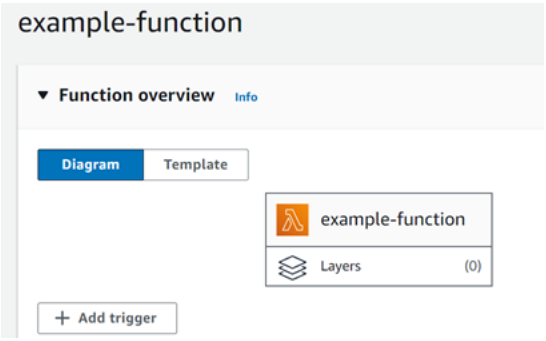
1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇函數的名稱。
3. 依序選擇 Configuration (組態) 索引標籤和 Permissions (許可)。
4. 在 [角色名稱] 下，選擇指向您的執行角色的連結。此連結會在 IAM 主控台中開啟角色。



5. 選擇新增許可，然後選擇連接政策。



6. 在搜尋欄位中輸入 AWSLambdaDynamoDBExecutionRole。將此原則新增至您的執行角色。這是一項 AWS 受管政策，其中包含您的函數需要從 DynamoDB 串流讀取的權限。如需有關此原則的詳細資訊，請參閱AWSLambdaDynamoDBExecutionRole受AWS 管理的原則參考中的。
7. 返回 Lambda 主控台中的函數。在 函式概觀 下，選擇 新增觸發條件 。



8. 選擇觸發條件類型。
9. 設定需要的選項，然後選擇 新增 。

Lambda 針對事件來源支援下列選項：

事件來源選項

- DynamoDB 資料表 - 從中讀取記錄的 DynamoDB 資料表。
- 批次大小 – 每個批次中要傳送至函數的記錄數量，最高為 10,000。Lambda 會將批次中所有記錄以單一呼叫傳送至函數，前提是事件的總大小不超過同步調用的酬載限制 (6 MB)。
- 批次間隔 - 指定調用函數前收集記錄的最長時間 (秒)。
- 開始位置 - 只處理新記錄，或所有現有的記錄。
  - 最新 - 處理已新增到串流的記錄。
  - 水平修剪 - 處理所有在串流中的記錄。

  處理任何現有的記錄後，該函式已跟上進度並持續處理新的記錄。
- 故障目的地 - 用於無法處理之記錄的標準 SQS 佇列或標準 SNS 主題。當 Lambda 捨棄太舊或已耗盡所有重試的一批記錄時，Lambda 會將該批次的詳細資料傳送至此佇列或主題。
- 重試嘗試 - 當函數傳回錯誤時，Lambda 重試的次數上限。這不適用於服務錯誤或調節，其中批次並沒有到達函數。
- 記錄最大存留期 - Lambda 傳送至函數之記錄的最大存留期。
- 在錯誤時分割批次 - 當函數傳回錯誤時，先將批次分割為兩個，再進行重試。您原始的批次大小設定仍會維持不變。
- 每個碎片的並行批次 - 同時處理來自同一個碎片的多個批次。
- 已啟用 - 設定為 true 可啟用事件來源映射。設定為 false 以停止處理記錄。Lambda 會追蹤上次處理的進度，並在重新啟用映射時從該時間點恢復處理。

> **Note**
>
> 作為 DynamoDB 觸發程序的一部分，Lambda 呼叫的 GetRecords API 呼叫不需支付費用。

若要稍後管理事件來源的組態，請選擇設計工具中的觸發。

錯誤處理

DynamoDB 事件來源對映的錯誤處理取決於在呼叫函數之前還是在函數叫用期間發生錯誤：

- 呼叫之前：如果 Lambda 事件來源對應由於節流或其他問題而無法叫用函數，則會重試直到記錄過期或超過事件來源對應上設定的保留天數上限 (秒)。MaximumRecord AgeIn
- 叫用期間：如果呼叫函數但傳回錯誤，Lambda 會重試直到記錄到期、超過最長保留天數 (MaximumRecordAgeIn秒) 或達到設定的重試配額 (MaximumRetry嘗試次數)。對於函數錯誤，您也可以設定 BisectBatchOnFunctionError，將失敗的批次分割成兩個較小的批次，隔離不良記錄並避免逾時。拆分批次不會消耗重試配額。

如果錯誤處理措施失敗，Lambda 會捨棄相應記錄，並繼續處理串流中的批次。使用預設設定時，這表示不良的記錄可能會封鎖受影響碎片上的處理長達一天。若要避免此情況，在設定函數的事件來源映射時，請使用合理的重試次數和符合您使用案例的記錄最大保留期。

設定失敗呼叫的目的地

若要保留失敗的事件來源映射調用記錄，請將目標地新增到函數的事件來源映射中。每個傳送至目的地的記錄都是 JSON 文件，其中包含有關失敗叫用的中繼資料。您可以將任何 Amazon SNS 主題或 Amazon SQS 佇列設定為目的地。您的執行角色必須具有目標的權限：

- 對於 SQS 目的地：sq s：SendMessage
- 對於 SNS 目的地：SNS: 發佈

若要使用主控台設定失敗時的目的地，請依照下列步驟執行：

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 在 函數概觀 下，選擇 新增目的地。
4. 針對來源，請選擇事件來源映射調用。
5. 對於事件來源映射，請選擇針對此函數設定的事件來源。
6. 對於條件，選取失敗時。對於事件來源映射調用，這是唯一可接受的條件。
7. 對於目標類型，請選擇 Lambda 將調用記錄傳送至的目標類型。
8. 對於 目的地，請選擇一個資源。
9. 選擇 儲存 。

您也可以使用 AWS Command Line Interface (AWS CLI) 來設定失敗時的目的地。例如，下列建立事件來源對映指令會將具有失敗時之 SQS 目的地的事件來源對應新增至：`MyFunction`

```
aws lambda create-event-source-mapping \
--function-name "MyFunction" \
--event-source-arn arn:aws:dynamodb:us-east-2:123456789012:table/my-table/
stream/2024-06-10T19:26:16.525 \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sqs:us-
east-1:123456789012:dest-queue"}}'
```

下列更新-事件來源映射命令會更新事件來源對應，以便在兩次重試嘗試後或記錄超過一個小時，將失敗的叫用記錄傳送至 SNS 目的地。

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--maximum-retry-attempts 2 \
--maximum-record-age-in-seconds 3600 \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sns:us-
east-1:123456789012:dest-topic"}}'
```

系統會以非同步的方式套用更新的設定，在處理完成之前不會反映在輸出中。使用取得事件來源對映指令來檢視目前的狀態。

若要移除目的地，請提供空白字串作為 destination-config 參數的引數：

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": ""}}'
```

下列範例顯示 DynamoDB 串流的調用記錄。

Example 調用記錄

```
{
    "requestContext": {
        "requestId": "316aa6d0-8154-xmpl-9af7-85d5f4a6bc81",
        "functionArn": "arn:aws:lambda:us-east-2:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted",
        "approximateInvokeCount": 1
    },
    "responseContext": {
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
```

```
    "version": "1.0",
    "timestamp": "2019-11-14T00:13:49.717Z",
    "DDBStreamBatchInfo": {
        "shardId": "shardId-00000001573689847184-864758bb",
        "startSequenceNumber": "80000000003126276362",
        "endSequenceNumber": "80000000003126276362",
        "approximateArrivalOfFirstRecord": "2019-11-14T00:13:19Z",
        "approximateArrivalOfLastRecord": "2019-11-14T00:13:19Z",
        "batchSize": 1,
        "streamArn": "arn:aws:dynamodb:us-east-2:123456789012:table/mytable/
stream/2019-11-14T00:04:06.388"
    }
}
```

您可以使用此資訊來從串流擷取受影響的記錄，以進行疑難排解。實際的記錄不包含在內，因此您必須處理此記錄，並在因過期而遺失之前從資料串流中擷取它們。

Amazon CloudWatch 指標

當您的函數處理完一個批次的記錄時，Lambda 會發出 `IteratorAge` 指標。指標指出處理完成時批次中最後一個記錄的存在時間。如果您的函數正在處理新的事件，可使用迭代器存留期來預估記錄新增與函數實際處理之間的延遲。

從迭代器存留期的增加趨勢可看出您函式的問題。如需詳細資訊，請參閱 使用 Lambda 函數指標。

時段

Lambda 函數可執行持續串流處理應用程式。串流表示持續在應用程式中流動的無限制資料。若要分析此持續更新輸入中的資訊，您可以使用定義的時段來限制包含的記錄。

輪轉時段是定期開啟和關閉的不同時段。依預設，Lambda 調用是無狀態的，您無法在沒有外部資料庫的情況下，將其用於處理多個持續調用的資料。然而，使用輪轉時段，您可以在不同的調用間維護狀態。此狀態包含之前為目前時段處理之訊息的彙總結果。狀態可以是每個分區最多 1 MB。如果超過該大小，則 Lambda 會提前終止時段。

串流中的每個記錄都屬於一個特定時段。Lambda 至少會處理一次每筆記錄，但不保證每筆記錄只會處理一次。在極少數情況下，例如錯誤處理，某些記錄可能會處理多次。第一次時一律會依序處理記錄。如果多次處理記錄，則可能不會按順序處理。

彙總與處理

調用您的使用者管理函數進行彙總，以及處理該彙總的最終結果。Lambda 會彙總時段中接收的所有記錄。您可以在多個批次中接收這些記錄，各自作為單獨的調用。每次調用會收到一個狀態。因此，當使

用輪轉時段時，您的 Lambda 函數回應必須包含 `state` 屬性。如果回應不包含 `state` 屬性，Lambda 會將此視為失敗的調用。為了滿足此條件，您的函數可以返回一個 `TimeWindowEventResponse` 物件，它具有下列 JSON 形狀：

Example **`TimeWindowEventResponse`** 值

```
{
    "state": {
        "1": 282,
        "2": 715
    },
    "batchItemFailures": []
}
```

> **Note**
>
> 對於 Java 函數，我們建議使用 `Map<String, String>` 來表示狀態。

在時段結束時，標記 `isFinalInvokeForWindow` 會設定為 `true` 以指示這是最終狀態，並且可隨時進行處理。處理完成後，時段結束並完成最終調用，然後丟棄該狀態。

在時段結束時，Lambda 會針對彙總結果上的動作使用最終處理。您的最終處理將同步調用。成功調用後，您的函數檢查點序號和串流處理將會繼續。如果調用失敗，則您的 Lambda 函數會暫停進一步處理，直至成功調用。

Example DynamodbTimeWindowEvent

```
{
   "Records":[
      {
         "eventID":"1",
         "eventName":"INSERT",
         "eventVersion":"1.0",
         "eventSource":"aws:dynamodb",
         "awsRegion":"us-east-1",
         "dynamodb":{
            "Keys":{
               "Id":{
                  "N":"101"
               }
```

```
            },
            "NewImage":{
               "Message":{
                  "S":"New item!"
               },
               "Id":{
                  "N":"101"
               }
            },
            "SequenceNumber":"111",
            "SizeBytes":26,
            "StreamViewType":"NEW_AND_OLD_IMAGES"
         },
         "eventSourceARN":"stream-ARN"
      },
      {
         "eventID":"2",
         "eventName":"MODIFY",
         "eventVersion":"1.0",
         "eventSource":"aws:dynamodb",
         "awsRegion":"us-east-1",
         "dynamodb":{
            "Keys":{
               "Id":{
                  "N":"101"
               }
            },
            "NewImage":{
               "Message":{
                  "S":"This item has changed"
               },
               "Id":{
                  "N":"101"
               }
            },
            "OldImage":{
               "Message":{
                  "S":"New item!"
               },
               "Id":{
                  "N":"101"
               }
            },
            "SequenceNumber":"222",
```

```
            "SizeBytes":59,
            "StreamViewType":"NEW_AND_OLD_IMAGES"
         },
         "eventSourceARN":"stream-ARN"
      },
      {
         "eventID":"3",
         "eventName":"REMOVE",
         "eventVersion":"1.0",
         "eventSource":"aws:dynamodb",
         "awsRegion":"us-east-1",
         "dynamodb":{
            "Keys":{
               "Id":{
                  "N":"101"
               }
            },
            "OldImage":{
               "Message":{
                  "S":"This item has changed"
               },
               "Id":{
                  "N":"101"
               }
            },
            "SequenceNumber":"333",
            "SizeBytes":38,
            "StreamViewType":"NEW_AND_OLD_IMAGES"
         },
         "eventSourceARN":"stream-ARN"
      }
   ],
    "window": {
        "start": "2020-07-30T17:00:00Z",
        "end": "2020-07-30T17:05:00Z"
    },
    "state": {
        "1": "state1"
    },
    "shardId": "shard123456789",
    "eventSourceARN": "stream-ARN",
    "isFinalInvokeForWindow": false,
    "isWindowTerminatedEarly": false
```

```
}
```

組態

您可以在建立或更新事件來源對映時設定輪轉時段。若要設定暫停視窗，請以秒為單位指定視窗 (TumblingWindowInSeconds)。下列範例 AWS Command Line Interface (AWS CLI) 命令會建立具有 120 秒暫停視窗的串流事件來源對應。針對彙總與處理定義的 Lambda 函數命名為 `tumbling-window-example-function`。

```
aws lambda create-event-source-mapping \
--event-source-arn arn:aws:dynamodb:us-east-2:123456789012:table/my-table/
stream/2024-06-10T19:26:16.525 \
--function-name tumbling-window-example-function \
--starting-position TRIM_HORIZON \
--tumbling-window-in-seconds 120
```

Lambda 根據記錄插入串流的時間，確定輪轉時段邊界。所有記錄都有 Lambda 在邊界確定中使用的近似時間戳記。

輪轉時段彙總不支援重新分區。分區結束後，Lambda 會考慮關閉時段，並且子分區會以新的狀態開始自己的時段。

輪轉時段完全支援現有的重試政策 `maxRetryAttempts` 和 `maxRecordAge`。

Example Handler.py - 彙總與處理

下列 Python 函數示範了如何彙總，然後處理您的最終狀態：

```
def lambda_handler(event, context):
    print('Incoming event: ', event)
    print('Incoming state: ', event['state'])

#Check if this is the end of the window to either aggregate or process.
    if event['isFinalInvokeForWindow']:
        # logic to handle final state of the window
        print('Destination invoke')
    else:
        print('Aggregate invoke')

#Check for early terminations
    if event['isWindowTerminatedEarly']:
```

```
        print('Window terminated early')

    #Aggregation logic
    state = event['state']
    for record in event['Records']:
        state[record['dynamodb']['NewImage']['Id']] = state.get(record['dynamodb']
['NewImage']['Id'], 0) + 1

    print('Returning state: ', state)
    return {'state': state}
```

報告批次項目失敗

取用和處理事件來源的串流資料時，依預設，只有在批次成功完成時，Lambda 檢查點才會到批次的最高序號。Lambda 會將所有其他結果視為完全失敗，並重試處理批次，直至達到重試限制。若要在處理串流的批次時允許部分成功，請開啟 ReportBatchItemFailures。允許部分成功有助於減少記錄的重試次數，但其不會完全消除在成功記錄中重試的可能性。

若要開啟ReportBatchItemFailures，請在 [FunctionResponse類型] 清單**ReportBatchItemFailures**中包含列舉值。此清單指示已為您的函數啟用哪些回應類型。您可以在建立或更新事件來源對應時設定此清單。

報告語法

設定批次項目失敗的報告時，會傳回 StreamsEventResponse 類別，其中包含批次項目失敗的清單。您可以使用 StreamsEventResponse 物件，來傳回批次中第一個失敗記錄的序號。您還可以使用正確的回應語法，建立自己的自訂類別。下列 JSON 結構顯示所需的回應語法：

```
{
  "batchItemFailures": [
        {
            "itemIdentifier": "<SequenceNumber>"
        }
    ]
}
```

> **Note**
>
> 如果 batchItemFailures 陣列包含多個項目，則 Lambda 會使用具有最低序列號的記錄作為檢查點。然後，Lambda 會重試從該檢查點開始的所有記錄。

成功與失敗條件

如果您傳回下列任一項目，Lambda 會將批次視為完全成功：

- 空白 `batchItemFailure` 清單
- Null `batchItemFailure` 清單
- 空白 `EventResponse`
- Null `EventResponse`

如果您傳回下列任一項目，Lambda 會將批次視為完全失敗：

- 空白字串 `itemIdentifier`
- Null `itemIdentifier`
- 具有錯誤金鑰名稱的 `itemIdentifier`

Lambda 會根據您的重試政策來重試失敗。

將批次平分

如果您的調用失敗且 `BisectBatchOnFunctionError` 已開啟，則無論您的 `ReportBatchItemFailures` 設定如何，批次都會被平分。

收到部分批次成功回應且 `BisectBatchOnFunctionError` 和 `ReportBatchItemFailures` 均開啟時，批次會依傳回的序號進行平分，並且 Lambda 僅會重試剩餘的記錄。

以下範例函數程式碼會傳回批次中失敗訊息 ID 的清單：

.NET

AWS SDK for .NET

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 .NET 報告使用 Lambda 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text.Json;
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace AWSLambda_DDB;

public class Function
{
    public StreamsEventResponse FunctionHandler(DynamoDBEvent dynamoEvent,
 ILambdaContext context)

    {
        context.Logger.LogInformation($"Beginning to process
 {dynamoEvent.Records.Count} records...");
        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 List<StreamsEventResponse.BatchItemFailure>();
        StreamsEventResponse streamsEventResponse = new StreamsEventResponse();

        foreach (var record in dynamoEvent.Records)
        {
            try
            {
                var sequenceNumber = record.Dynamodb.SequenceNumber;
                context.Logger.LogInformation(sequenceNumber);
            }
            catch (Exception ex)
            {
                context.Logger.LogError(ex.Message);
                batchItemFailures.Add(new StreamsEventResponse.BatchItemFailure()
 { ItemIdentifier = record.Dynamodb.SequenceNumber });
            }
        }

        if (batchItemFailures.Count > 0)
        {
```

```
            streamsEventResponse.BatchItemFailures = batchItemFailures;
        }

        context.Logger.LogInformation("Stream processing complete.");
        return streamsEventResponse;
    }
}
```

Go

SDK for Go V2

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 使用 Lambda 報告批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

type BatchItemFailure struct {
 ItemIdentifier string `json:"ItemIdentifier"`
}

type BatchResult struct {
 BatchItemFailures []BatchItemFailure `json:"BatchItemFailures"`
}

func HandleRequest(ctx context.Context, event events.DynamoDBEvent)
 (*BatchResult, error) {
 var batchItemFailures []BatchItemFailure
 curRecordSequenceNumber := ""
```

```
 for _, record := range event.Records {
  // Process your record
  curRecordSequenceNumber = record.Change.SequenceNumber
 }

 if curRecordSequenceNumber != "" {
  batchItemFailures = append(batchItemFailures, BatchItemFailure{ItemIdentifier:
 curRecordSequenceNumber})
 }

 batchResult := BatchResult{
  BatchItemFailures: batchItemFailures,
 }

 return &batchResult, nil
}

func main() {
 lambda.Start(HandleRequest)
}
```

Java

適用於 Java 2.x 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Java 使用 Lambda 報告批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;
import com.amazonaws.services.lambda.runtime.events.models.dynamodb.StreamRecord;
```

```
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

public class ProcessDynamodbRecords implements RequestHandler<DynamodbEvent,
 Serializable> {

    @Override
    public StreamsEventResponse handleRequest(DynamodbEvent input, Context
 context) {

        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<>();
        String curRecordSequenceNumber = "";

        for (DynamodbEvent.DynamodbStreamRecord dynamodbStreamRecord :
 input.getRecords()) {
          try {
                //Process your record
                StreamRecord dynamodbRecord = dynamodbStreamRecord.getDynamodb();
                curRecordSequenceNumber = dynamodbRecord.getSequenceNumber();

            } catch (Exception e) {
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                batchItemFailures.add(new
 StreamsEventResponse.BatchItemFailure(curRecordSequenceNumber));
                return new StreamsEventResponse(batchItemFailures);
            }
        }

       return new StreamsEventResponse();
    }
}
```

JavaScript

適用於 JavaScript (v3) 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Lambda 使用報告批次項目失敗 JavaScript

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
export const handler = async (event) => {
  const records = event.Records;
  let curRecordSequenceNumber = "";

  for (const record of records) {
    try {
      // Process your record
      curRecordSequenceNumber = record.dynamodb.SequenceNumber;
    } catch (e) {
      // Return failed record's sequence number
      return { batchItemFailures: [{ itemIdentifier:
 curRecordSequenceNumber }] };
    }
  }

  return { batchItemFailures: [] };
};
```

使用 Lambda 使用報告批次項目失敗 TypeScript

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { DynamoDBBatchItemFailure, DynamoDBStreamEvent } from "aws-lambda";

export const handler = async (event: DynamoDBStreamEvent):
 Promise<DynamoDBBatchItemFailure[]> => {
```

```
    const batchItemsFailures: DynamoDBBatchItemFailure[] = []
    let curRecordSequenceNumber

    for(const record of event.Records) {
        curRecordSequenceNumber = record.dynamodb?.SequenceNumber

        if(curRecordSequenceNumber) {
            batchItemsFailures.push({
                itemIdentifier: curRecordSequenceNumber
            })
        }
    }

    return batchItemsFailures
}
```

PHP

適用於 PHP 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 PHP 使用 Lambda 報告批次項目失敗。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\DynamoDb\DynamoDbEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';
```

```
class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): array
    {
        $dynamoDbEvent = new DynamoDbEvent($event);
        $this->logger->info("Processing records");

        $records = $dynamoDbEvent->getRecords();
        $failedRecords = [];
        foreach ($records as $record) {
            try {
                $data = $record->getData();
                $this->logger->info(json_encode($data));
                // TODO: Do interesting work based on the new data
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $failedRecords[] = $record->getSequenceNumber();
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");

        // change format for the response
        $failures = array_map(
            fn(string $sequenceNumber) => ['itemIdentifier' => $sequenceNumber],
            $failedRecords
        );

        return [
            'batchItemFailures' => $failures
        ];
    }
}
```

```
$logger = new StderrLogger();
return new Handler($logger);
```

Python

適用於 Python (Boto3) 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Python 使用 Lambda 報告批次項目失敗。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def handler(event, context):
    records = event.get("Records")
    curRecordSequenceNumber = ""

    for record in records:
        try:
            # Process your record
            curRecordSequenceNumber = record["dynamodb"]["SequenceNumber"]
        except Exception as e:
            # Return failed record's sequence number
            return {"batchItemFailures":[{"itemIdentifier":
 curRecordSequenceNumber}]}

    return {"batchItemFailures":[]}
```

Ruby

適用於 Ruby 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用紅寶石使用 Lambda 報告批次項目失敗。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event:, context:)
    records = event["Records"]
    cur_record_sequence_number = ""

    records.each do |record|
      begin
        # Process your record
        cur_record_sequence_number = record["dynamodb"]["SequenceNumber"]
      rescue StandardError => e
        # Return failed record's sequence number
        return {"batchItemFailures" => [{"itemIdentifier" =>
 cur_record_sequence_number}]}
      end
    end

    {"batchItemFailures" => []}
  end
```

Rust

適用於 Rust 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Rust 使用 Lambda 報告批次項目故障。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::{
    event::dynamodb::{Event, EventRecord, StreamRecord},
    streams::{DynamoDbBatchItemFailure, DynamoDbEventResponse},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

/// Process the stream record
fn process_record(record: &EventRecord) -> Result<(), Error> {
    let stream_record: &StreamRecord = &record.change;

    // process your stream record here...
    tracing::info!("Data: {:?}", stream_record);

    Ok(())
}

/// Main Lambda handler here...
async fn function_handler(event: LambdaEvent<Event>) ->
 Result<DynamoDbEventResponse, Error> {
    let mut response = DynamoDbEventResponse {
        batch_item_failures: vec![],
    };

    let records = &event.payload.records;

    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(response);
    }

    for record in records {
        tracing::info!("EventId: {}", record.event_id);

        // Couldn't find a sequence number
        if record.change.sequence_number.is_none() {
            response.batch_item_failures.push(DynamoDbBatchItemFailure {
                item_identifier: Some("".to_string()),
            });
            return Ok(response);
        }
```

```
        // Process your record here...
        if process_record(record).is_err() {
            response.batch_item_failures.push(DynamoDbBatchItemFailure {
                item_identifier: record.change.sequence_number.clone(),
            });
            /* Since we are working with streams, we can return the failed item
 immediately.
            Lambda will immediately begin to retry processing from this failed
 item onwards. */
            return Ok(response);
        }
    }

    tracing::info!("Successfully processed {} record(s)", records.len());

    Ok(response)
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

Amazon DynamoDB Streams 組態參數

所有 Lambda 事件來源類型都共用相同CreateEventSourceMapping的UpdateEventSourceMappingAPI 作業。但是，只有一些參數適用於 DynamoDB Streams。

適用於 DynamoDB Streams 的事件來源參數

參數	必要	預設	備註
BatchSize	否	100	上限：10,000
BisectBatchOnFunction錯誤	N	false	
DestinationConfig	N		捨棄記錄的標準 Amazon SQS 佇列或標準 Amazon SNS 主題目的地
已啟用	N	true	
EventSource阿恩	Y		資料串流或串流消費者的 ARN
FilterCriteria	N		Lambda 事件篩選
FunctionName	Y		
FunctionResponse类型	N		若要讓函數報告批次中的特定失敗，請將值 `ReportBatchItemFailures` 包含在 `FunctionResponseTypes` 中。如需詳細資訊，請參閱 報告批次項目失敗。
MaximumBatchingWindowIn秒	N	0	
MaximumRecordAgeIn秒	N	-1	-1 表示 infinite：失敗的記錄會重試，直到記錄過期為止。DynamoDB Streams

參數	必要	預設	備註
			的資料保留限制為 24 小時。 下限：-1 上限：604,800
MaximumRetry嘗試	N	-1	-1 代表無限：系統會重試失敗的記錄，直到記錄過期為止 下限：0 上限：10,000
ParallelizationFactor	N	1	上限：10
StartingPosition	Y		TRIM_HORIZON 或 LATEST
TumblingWindowInSeconds	N		下限：0 上限：900

教學課程： AWS Lambda 與 Amazon DynamoDB 串流搭配使用

在此教學課程中，您建立 Lambda 函數以從 Amazon DynamoDB Streams 中取用事件。

必要條件

此教學課程假設您具備基本的 Lambda 操作知識並了解 Lambda 主控台。若您尚未了解，請遵循 使用主控台建立一個 Lambda 函數 中的指示，建立您的第一個 Lambda 函數。

若要完成下列步驟，您需要 AWS Command Line Interface (AWS CLI) 版本 2。命令和預期的輸出會列在不同的區塊中：

```
aws --version
```

您應該會看到下列輸出：

```
aws-cli/2.13.27 Python/3.11.6 Linux/4.14.328-248.540.amzn2.x86_64 exe/x86_64.amzn.2
```

對於長命令，逸出字元 (\) 用於將命令分割為多行。

在 Linux 和 macOS 上，使用您偏好的 shell 和套件軟體管理工具。

> **Note**
>
> 在 Windows 中，作業系統的內建終端不支援您常與 Lambda 搭配使用的某些 Bash CLI 命令 (例如 `zip`)。若要取得 Ubuntu 和 Bash 的 Windows 整合版本，請安裝適用於 Linux 的 Windows 子系統。本指南中的 CLI 命令範例使用 Linux 格式。如果您使用的是 Windows CLI，必須重新格式化包含內嵌 JSON 文件的命令。

建立執行角色

建立可授與函數存取 AWS 資源之權限的執行角色。

若要建立執行角色

1. 在 IAM 主控台中開啟 角色頁面。
2. 選擇 建立角色。
3. 建立具備下列屬性的角色。
   - 信任實體 - Lambda。
   - 權限 — AWSLambdaDynamoDBExecutionRole.
   - 角色名稱 - **`lambda-dynamodb-role`**。

AWSLambdaDynamoDBExecutionRole具有函數從 DynamoDB 讀取項目並將記錄寫入記錄所需的權限。 CloudWatch

建立函數

建立可處理您事件的 Lambda 函數。函數程式碼會將部分傳入事件資料寫入 CloudWatch 記錄檔。

.NET

AWS SDK for .NET

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 .NET 與 Lambda 一起使用 DynamoDB 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text.Json;
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace AWSLambda_DDB;

public class Function
{
    public void FunctionHandler(DynamoDBEvent dynamoEvent, ILambdaContext
 context)
    {
        context.Logger.LogInformation($"Beginning to process
 {dynamoEvent.Records.Count} records...");

        foreach (var record in dynamoEvent.Records)
        {
            context.Logger.LogInformation($"Event ID: {record.EventID}");
            context.Logger.LogInformation($"Event Name: {record.EventName}");

            context.Logger.LogInformation(JsonSerializer.Serialize(record));
        }

        context.Logger.LogInformation("Stream processing complete.");
```

```
    }
}
```

Go

SDK for Go V2

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 使用與 Lambda 一起使用 DynamoDB 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-lambda-go/events"
 "fmt"
)

func HandleRequest(ctx context.Context, event events.DynamoDBEvent) (*string,
 error) {
 if len(event.Records) == 0 {
  return nil, fmt.Errorf("received empty event")
 }

 for _, record := range event.Records {
   LogDynamoDBRecord(record)
 }

 message := fmt.Sprintf("Records processed: %d", len(event.Records))
 return &message, nil
}

func main() {
```

```
 lambda.Start(HandleRequest)
}

func LogDynamoDBRecord(record events.DynamoDBEventRecord){
 fmt.Println(record.EventID)
 fmt.Println(record.EventName)
 fmt.Printf("%+v\n", record.Change)
}
```

Java

適用於 Java 2.x 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Java 與 Lambda 一起使用 DynamoDB 事件。

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import
 com.amazonaws.services.lambda.runtime.events.DynamodbEvent.DynamodbStreamRecord;
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class example implements RequestHandler<DynamodbEvent, Void> {

    private static final Gson GSON = new
 GsonBuilder().setPrettyPrinting().create();

    @Override
    public Void handleRequest(DynamodbEvent event, Context context) {
        System.out.println(GSON.toJson(event));
        event.getRecords().forEach(this::logDynamoDBRecord);
        return null;
    }

    private void logDynamoDBRecord(DynamodbStreamRecord record) {
```

```
        System.out.println(record.getEventID());
        System.out.println(record.getEventName());
        System.out.println("DynamoDB Record: " +
 GSON.toJson(record.getDynamodb()));
    }
}
```

JavaScript

適用於 JavaScript (v3) 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用使 Lambda. JavaScript

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(record => {
        logDynamoDBRecord(record);
    });
};

const logDynamoDBRecord = (record) => {
    console.log(record.eventID);
    console.log(record.eventName);
    console.log(`DynamoDB Record: ${JSON.stringify(record.dynamodb)}`);
};
```

使用使 Lambda. TypeScript

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
export const handler = async (event, context) => {
    console.log(JSON.stringify(event, null, 2));
```

```
    event.Records.forEach(record => {
        logDynamoDBRecord(record);
    });
}
const logDynamoDBRecord = (record) => {
    console.log(record.eventID);
    console.log(record.eventName);
    console.log(`DynamoDB Record: ${JSON.stringify(record.dynamodb)}`);
};
```

PHP

適用於 PHP 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 PHP 與 Lambda 一起使用 DynamoDB 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\DynamoDb\DynamoDbEvent;
use Bref\Event\DynamoDb\DynamoDbHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends DynamoDbHandler
{
    private StderrLogger $logger;

    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
```

```
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleDynamoDb(DynamoDbEvent $event, Context $context): void
    {
        $this->logger->info("Processing DynamoDb table items");
        $records = $event->getRecords();

        foreach ($records as $record) {
            $eventName = $record->getEventName();
            $keys = $record->getKeys();
            $old = $record->getOldImage();
            $new = $record->getNewImage();

            $this->logger->info("Event Name:".$eventName."\n");
            $this->logger->info("Keys:". json_encode($keys)."\n");
            $this->logger->info("Old Image:". json_encode($old)."\n");
            $this->logger->info("New Image:". json_encode($new));

            // TODO: Do interesting work based on the new data

            // Any exception thrown will be logged and the invocation will be
 marked as failed
        }

        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords items");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

適用於 Python (Boto3) 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用使用 Python 與 Lambda 一起使用動 DynamoDB 事件。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0

import json

def lambda_handler(event, context):
    print(json.dumps(event, indent=2))

    for record in event['Records']:
        log_dynamodb_record(record)

def log_dynamodb_record(record):
    print(record['eventID'])
    print(record['eventName'])
    print(f"DynamoDB Record: {json.dumps(record['dynamodb'])}")
```

Ruby

適用於 Ruby 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用紅寶石與 Lambda 一起使用 DynamoDB 事件。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0

def lambda_handler(event:, context:)
    return 'received empty event' if event['Records'].empty?

    event['Records'].each do |record|
      log_dynamodb_record(record)
    end

    "Records processed: #{event['Records'].length}"
  end

  def log_dynamodb_record(record)
    puts record['eventID']
    puts record['eventName']
    puts "DynamoDB Record: #{JSON.generate(record['dynamodb'])}"
  end
```

Rust

適用於 Rust 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Rust 與 Lambda 一起使用 DynamoDB 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

use lambda_runtime::{service_fn, tracing, Error, LambdaEvent};
use aws_lambda_events::{
    event::dynamodb::{Event, EventRecord},
   };
```

```
// Built with the following dependencies:
//lambda_runtime = "0.11.1"
//serde_json = "1.0"
//tokio = { version = "1", features = ["macros"] }
//tracing = { version = "0.1", features = ["log"] }
//tracing-subscriber = { version = "0.3", default-features = false, features =
 ["fmt"] }
//aws_lambda_events = "0.15.0"

async fn function_handler(event: LambdaEvent<Event>) ->Result<(), Error> {

    let records = &event.payload.records;
    tracing::info!("event payload: {:?}",records);
    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }

    for record in records{
        log_dynamo_dbrecord(record);
    }

    tracing::info!("Dynamo db records processed");

    // Prepare the response
    Ok(())

}

fn log_dynamo_dbrecord(record: &EventRecord)-> Result<(), Error>{
    tracing::info!("EventId: {}", record.event_id);
    tracing::info!("EventName: {}", record.event_name);
    tracing::info!("DynamoDB Record: {:?}", record.change );
    Ok(())

}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
    .with_max_level(tracing::Level::INFO)
    .with_target(false)
    .without_time()
    .init();
```

```
    let func = service_fn(function_handler);
    lambda_runtime::run(func).await?;
    Ok(())

}
```

建立函數

1. 將範本程式碼複製到名為 example.js 的檔案。
2. 建立部署套件。

```
zip function.zip example.js
```

3. 使用 create-function 命令建立一個 Lambda 函數。

```
aws lambda create-function --function-name ProcessDynamoDBRecords \
    --zip-file fileb://function.zip --handler example.handler --runtime nodejs18.x
 \
    --role arn:aws:iam::111122223333:role/lambda-dynamodb-role
```

測試 Lambda 函數

在此步驟中，您可以使用 invoke AWS Lambda CLI 命令和下列 DynamoDB 事件範例手動叫用 Lambda 函數。將以下內容複製到名為的檔案中input.txt。

Example input.txt

```
{
   "Records":[
      {
         "eventID":"1",
         "eventName":"INSERT",
         "eventVersion":"1.0",
         "eventSource":"aws:dynamodb",
         "awsRegion":"us-east-1",
         "dynamodb":{
            "Keys":{
               "Id":{
```

```
                "N":"101"
             }
          },
          "NewImage":{
             "Message":{
                "S":"New item!"
             },
             "Id":{
                "N":"101"
             }
          },
          "SequenceNumber":"111",
          "SizeBytes":26,
          "StreamViewType":"NEW_AND_OLD_IMAGES"
       },
       "eventSourceARN":"stream-ARN"
    },
    {
       "eventID":"2",
       "eventName":"MODIFY",
       "eventVersion":"1.0",
       "eventSource":"aws:dynamodb",
       "awsRegion":"us-east-1",
       "dynamodb":{
          "Keys":{
             "Id":{
                "N":"101"
             }
          },
          "NewImage":{
             "Message":{
                "S":"This item has changed"
             },
             "Id":{
                "N":"101"
             }
          },
          "OldImage":{
             "Message":{
                "S":"New item!"
             },
             "Id":{
                "N":"101"
             }
```

```
            },
            "SequenceNumber":"222",
            "SizeBytes":59,
            "StreamViewType":"NEW_AND_OLD_IMAGES"
         },
         "eventSourceARN":"stream-ARN"
      },
      {
         "eventID":"3",
         "eventName":"REMOVE",
         "eventVersion":"1.0",
         "eventSource":"aws:dynamodb",
         "awsRegion":"us-east-1",
         "dynamodb":{
            "Keys":{
               "Id":{
                  "N":"101"
               }
            },
            "OldImage":{
               "Message":{
                  "S":"This item has changed"
               },
               "Id":{
                  "N":"101"
               }
            },
            "SequenceNumber":"333",
            "SizeBytes":38,
            "StreamViewType":"NEW_AND_OLD_IMAGES"
         },
         "eventSourceARN":"stream-ARN"
      }
   ]
}
```

執行以下 invoke 命令。

```
aws lambda invoke --function-name ProcessDynamoDBRecords \
    --cli-binary-format raw-in-base64-out \
    --payload file://input.txt outputfile.txt
```

如果您使用的是 AWS CLI 版本 2，則需要此cli-binary-format選項。若要讓此成為預設的設定，請執行 `aws configure set cli-binary-format raw-in-base64-out`。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

該函數會在回應本文中傳回字串 `message` 訊息。

在 `outputfile.txt` 檔案中確認輸出。

建立啟用串流的 DynamoDB 資料表

建立啟用串流的 Amazon DynamoDB 資料表。

建立 DynamoDB 資料表

1. 開啟 DynamoDB 主控台。
2. 選擇 建立資料表 。
3. 根據下列設定建立資料表。
   - Table name (資料表名稱) – **`lambda-dynamodb-stream`**
   - Primary key (主要金鑰) - **`id`** (字串)
4. 選擇建立。

啟用串流

1. 開啟 DynamoDB 主控台。
2. 選擇 資料表。
3. 選擇 lambda-dynamodb-stream 資料表。
4. 在 匯出與串流 下，選擇 DynamoDB 串流詳細資訊 。
5. 選擇 Turn on (開啟)。
6. 針對「視景類型」，選擇「僅關鍵屬性」。
7. 選擇 [開啟串流]。

寫下串流 ARN。在下個步驟將串流與您的 Lambda 函數相關聯時會需要用到它。如需啟用串流的詳細資訊，請參閱使用 DynamoDB Streams 擷取資料表活動。

在中新增事件來源 AWS Lambda

在 AWS Lambda中建立事件來源映射。事件來源映射可將 DynamoDB 串流與您的 Lambda 函數相關聯。建立此事件來源對應之後， AWS Lambda 開始輪詢串流。

執行下列 AWS CLI `create-event-source-mapping` 命令。命令執行後，請記下 UUID。使用任何命令時，您會需要此 UUID 以參考到事件來源映射，例如當刪除事件來源映射的時候。

```
aws lambda create-event-source-mapping --function-name ProcessDynamoDBRecords \
    --batch-size 100 --starting-position LATEST --event-source DynamoDB-stream-arn
```

這會在指定的 DynamoDB 串流和 Lambda 函數之間建立映射。您可以將 DynamoDB 串流與多個 Lambda 函數相關聯，也可以將相同的 Lambda 函數與多個串流相關聯。但是 Lambda 函數會共用其所共有之串流的讀取傳送量。

您可以執行下列命令來取得事件來源映射的清單。

```
aws lambda list-event-source-mappings
```

該清單傳回所有您建立的事件來源映射，並針對每個映射顯示 `LastProcessingResult` 和其他內容。此欄位可用在發生問題時，提供資訊豐富的訊息。諸如 `No records processed` (表示尚 AWS Lambda 未開始輪詢或串流中沒有記錄) 和 `OK` (表示 AWS Lambda 成功從串流讀取記錄並叫用 Lambda 函數) 等值表示沒有問題。如果發生問題，您會收到錯誤訊息。

如果您有許多事件來源映射，請使用函數式稱參數來縮小結果。

```
aws lambda list-event-source-mappings --function-name ProcessDynamoDBRecords
```

測試設定

測試體 end-to-end 驗。當您更新資料表時，DynamoDB 會將事件記錄寫入串流。當 AWS Lambda 輪詢串流時，若偵測到串流上的新記錄，便會透過傳送事件到函數來代表您調用 Lambda 函數。

1. 在 DynamoDB 主控台上針對資料表新增、更新、刪除項目。DynamoDB 會將這些動作的記錄寫入串流。
2. AWS Lambda 輪詢串流，當它偵測到串流的更新時，它會傳入它在串流中找到的事件資料來叫用 Lambda 函數。
3. 您的函數運行並在 Amazon 創建日誌 CloudWatch。您可以驗證 Amazon CloudWatch 主控台中報告的日誌。

清除您的資源

除非您想要保留為此教學課程建立的資源，否則您現在便可刪除。刪除您不再使用的 AWS 資源，您可以避免不必要的費用 AWS 帳戶。

若要刪除 Lambda 函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇您建立的函數。
3. 選擇 Actions (動作)、Delete (刪除)。
4. 在文字輸入欄位中輸入 **delete**，然後選擇 刪除 。

刪除執行角色

1. 開啟 IAM 主控台中的 角色頁面 。
2. 選取您建立的執行角色。
3. 選擇 刪除 。
4. 在文字輸入欄位中輸入角色的名稱，然後選擇 刪除 。

若要刪除 DynamoDB 資料表

1. 開啟 DynamoDB 主控台的 資料表頁面 。
2. 選取您建立的資料表。
3. 選擇 刪除 。
4. 在文字方塊中輸入 **delete**。
5. 選擇 刪除資料表 。

範例函數程式碼

範本程式碼可用於以下語言。

主題

- Node.js
- Java 11
- C#

- Python 3
- Go

Node.js

下列範例處理來自 DynamoDB 的訊息，並記錄其中內容。

Example ProcessDynamoDBStream.js

```
console.log('Loading function');

exports.lambda_handler = function(event, context, callback) {
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(function(record) {
        console.log(record.eventID);
        console.log(record.eventName);
        console.log('DynamoDB Record: %j', record.dynamodb);
    });
    callback(null, "message");
};
```

壓縮範本程式碼以建立部署套件。如需說明，請參閱 使用 .zip 封存檔部署 Node.js Lambda 函數。

Java 11

以下範例處理來自 DynamoDB 的訊息，並記錄其內容。`handleRequest` 為 AWS Lambda 調用並提供事件資料的處理常式。處理常式使用預先定義的 `DynamodbEvent` 類別 (在 `aws-lambda-java-events` 程式庫中定義)。

Example 爪哇 EventProcessor

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler2;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent.DynamodbStreamRecord;

public class DDBEventProcessor implements
        RequestHandler2<DynamodbEvent, String> {
```

```
    public String handleRequest(DynamodbEvent ddbEvent, Context context) {
        for (DynamodbStreamRecord record : ddbEvent.getRecords()){
           System.out.println(record.getEventID());
           System.out.println(record.getEventName());
           System.out.println(record.getDynamodb().toString());

        }
        return "Successfully processed " + ddbEvent.getRecords().size() + " records.";
    }
}
```

如果處理常式均正常傳回而無例外情況，Lambda 將認為記錄的輸入批次已成功處理，並開始在串流上讀取新記錄。如果處理常式發生例外情況，Lambda 將認為記錄的輸入批次沒有成功處理，並重新調用具有相同記錄批次的函數。

相依性

- `aws-lambda-java-core`
- `aws-lambda-java-events`

建置具有 Lambda 程式庫相依性的程式碼，以建立部署套件。如需說明，請參閱使用 .zip 或 JAR 封存檔部署 Java Lambda 函數。

C#

以下範例處理來自 DynamoDB 的訊息，並記錄其內容。`ProcessDynamoEvent` 為 AWS Lambda 調用並提供事件資料的處理常式。處理常式使用預先定義的 `DynamoDbEvent` 類別 (在 `Amazon.Lambda.DynamoDBEvents` 程式庫中定義)。

Example ProcessingDynamoDBStreams.cs

```
using System;
using System.IO;
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;

using Amazon.Lambda.Serialization.Json;

namespace DynamoDBStreams
{
    public class DdbSample
```

```
    {
        private static readonly JsonSerializer _jsonSerializer = new JsonSerializer();

        public void ProcessDynamoEvent(DynamoDBEvent dynamoEvent)
        {
            Console.WriteLine($"Beginning to process {dynamoEvent.Records.Count}
 records...");

            foreach (var record in dynamoEvent.Records)
            {
                Console.WriteLine($"Event ID: {record.EventID}");
                Console.WriteLine($"Event Name: {record.EventName}");

                string streamRecordJson = SerializeObject(record.Dynamodb);
                Console.WriteLine($"DynamoDB Record:");
                Console.WriteLine(streamRecordJson);
            }

            Console.WriteLine("Stream processing complete.");
        }

        private string SerializeObject(object streamRecord)
        {
            using (var ms = new MemoryStream())
            {
                _jsonSerializer.Serialize(streamRecord, ms);
                return Encoding.UTF8.GetString(ms.ToArray());
            }
        }
    }
}
```

使用上述範例取代 .NET Core 專案中的 `Program.cs`。如需說明，請參閱 使用 .zip 封存檔建置和部署 C# Lambda 函數。

Python 3

下列範例處理來自 DynamoDB 的訊息，並記錄其中內容。

Example ProcessDynamoDBStream.py

```
from __future__ import print_function

def lambda_handler(event, context):
```

```
    for record in event['Records']:
        print(record['eventID'])
        print(record['eventName'])
    print('Successfully processed %s records.' % str(len(event['Records'])))
```

壓縮範本程式碼以建立部署套件。如需說明，請參閱 使用 .zip 封存檔部署 Python Lambda 函數。

Go

下列範例處理來自 DynamoDB 的訊息，並記錄其中內容。

Example

```
import (
    "strings"

    "github.com/aws/aws-lambda-go/events"
)

func handleRequest(ctx context.Context, e events.DynamoDBEvent) {

    for _, record := range e.Records {
        fmt.Printf("Processing request data for event ID %s, type %s.\n",
 record.EventID, record.EventName)

        // Print new values for attributes of type String
        for name, value := range record.Change.NewImage {
            if value.DataType() == events.DataTypeString {
                fmt.Printf("Attribute name: %s, value: %s\n", name, value.String())
            }
        }
    }
}
```

使用 go build 建置可執行文件並建立部署套件。如需說明，請參閱 使用 .zip 封存檔部署 Go Lambda 函數。

DynamoDB 應用程式的 AWS SAM 範本

您可以使用 AWS SAM 建置這個應用程式。若要進一步了解如何建立 AWS SAM 範本，請參閱《AWS Serverless Application Model 開發人員指南》中的 AWS SAM 範本基本概念。

以下為來自教學課程的應用程式的 AWS SAM 範本。複製以下文字至 .yaml 檔案，並儲存到您先前建立的 ZIP 套件旁。請注意，Handler 與 Runtime 參數值需應與您在前一節中建立函數時所使用的參數值相符。

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Resources:
  ProcessDynamoDBStream:
    Type: AWS::Serverless::Function
    Properties:
      Handler: handler
      Runtime: runtime
      Policies: AWSLambdaDynamoDBExecutionRole
      Events:
        Stream:
          Type: DynamoDB
          Properties:
            Stream: !GetAtt DynamoDBTable.StreamArn
            BatchSize: 100
            StartingPosition: TRIM_HORIZON

  DynamoDBTable:
    Type: AWS::DynamoDB::Table
    Properties:
      AttributeDefinitions:
        - AttributeName: id
          AttributeType: S
      KeySchema:
        - AttributeName: id
          KeyType: HASH
      ProvisionedThroughput:
        ReadCapacityUnits: 5
        WriteCapacityUnits: 5
      StreamSpecification:
        StreamViewType: NEW_IMAGE
```

如需進一步了解如何使用套件與部署命令來封裝與部署無伺服器應用程式，請參閱《AWS Serverless Application Model 開發人員指南》中的部署無伺服器應用程式。

Lambda 如何處理來自 Amazon Kinesis Data Streams 的記錄

您可以使用 Lambda 函數來處理 Amazon Kinesis 資料串流中的記錄。您可以將 Lambda 函數對應至 Kinesis Data Streams 共用輸送量取用者 (標準迭代器)，或對應至具有增強型散發功能的專用輸送量取用者。對於標準迭代程式，Lambda 會使用 HTTP 協定輪詢 Kinesis 串流中的每個碎片以尋找記錄。事件來源映射會與碎片的其他取用者共用讀取傳輸量。

如需 Kinesis 資料串流的詳細資訊，請參閱從 Amazon Kinesis Data Streams 讀取資料。

> **Note**
>
> Kinesis 會收取每個碎片的費用，以及從串流讀取資料的增強型散發。如需定價的詳細資訊，請參閱 Amazon Kinesis 定價。

輪詢和批次處理串流

Lambda 會從資料串流讀取記錄並透過包含串流記錄的事件同步調用函數。Lambda 會讀取批次中的記錄並調用函數，以處理來自該批次的記錄。每個批次包含來自單一碎片/資料串流的記錄。

Lambda 預設會在記錄可用時立即調用函數。如果 Lambda 從事件來源中讀取的批次只有一筆記錄，Lambda 只會傳送一筆記錄至函數。為避免調用具有少量記錄的函數，您可設定批次間隔，請求事件來源緩衝記錄最長達五分鐘。調用函數之前，Lambda 會繼續從事件來源中讀取記錄，直到收集到完整批次、批次間隔到期或者批次達到 6 MB 的承載限制。如需詳細資訊，請參閱 批次處理行為。

> **Warning**
>
> Lambda 事件來源對應至少處理每個事件一次，並且可能會重複處理記錄。為了避免與重複事件相關的潛在問題，我們強烈建議您將函數代碼設為冪等。若要深入了解，請參閱 AWS 知識中心如何讓 Lambda 函數具有冪等性。

ParallelizationFactor設定此設定以同時處理具有多個 Lambda 叫用的 Kinesis 資料串流碎片。您可以透過從 1 (預設) 到 10 的並行化因子指定 Lambda 從碎片輪詢的並行批次數。例如，當 `ParallelizationFactor` 設定為 2 時，您最多可以有 200 個並行 Lambda 調用，來處理 100 個 Kinesis 資料碎片 (不過在實務中，`ConcurrentExecutions` 指標可能有不同值)。當資料量急劇波動並且 `IteratorAge` 高時，這有助於縱向擴展處理輸送量。當您增加每個碎片的並行批次數時，Lambda 仍可確保在磁碟分割索引鍵層級進行順序處理。

您也可以ParallelizationFactor搭配 Kinesis 彙總使用。事件來源對應的行為取決於您是否使用增強型散發：

- 沒有增強型散發：聚合事件中的所有事件都必須具有相同的分區索引鍵。分割索引鍵也必須與彙總事件的索引鍵相符。如果彙總事件內的事件具有不同的分區索引鍵，Lambda 無法保證依分區索引鍵按順序處理事件。
- 透過增強型散發：首先，Lambda 會將彙總的事件解碼為其個別事件。聚集的事件可以具有與其包含的事件不同的分區索引鍵。不過，與磁碟分割索引鍵不對應的事件會遭到捨棄並遺失。Lambda 不會處理這些事件，也不會將它們傳送到設定的故障目的地。

範例事件

Example

```
{
    "Records": [
        {
            "kinesis": {
                "kinesisSchemaVersion": "1.0",
                "partitionKey": "1",
                "sequenceNumber":
 "49590338271490256608559692538361571095921575989136588898",
                "data": "SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
                "approximateArrivalTimestamp": 1545084650.987
            },
            "eventSource": "aws:kinesis",
            "eventVersion": "1.0",
            "eventID":
 "shardId-000000000006:49590338271490256608559692538361571095921575989136588898",
            "eventName": "aws:kinesis:record",
            "invokeIdentityArn": "arn:aws:iam::123456789012:role/lambda-role",
            "awsRegion": "us-east-2",
            "eventSourceARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-
stream"
        },
        {
            "kinesis": {
                "kinesisSchemaVersion": "1.0",
                "partitionKey": "1",
                "sequenceNumber":
 "49590338271490256608559692540925702759324208523137515618",
```

```
                "data": "VGhpcyBpcyBvbmx5IGEgdGVzdC4=",
                "approximateArrivalTimestamp": 1545084711.166
            },
            "eventSource": "aws:kinesis",
            "eventVersion": "1.0",
            "eventID":
 "shardId-000000000006:49590338271490256608559692540925702759324208523137515618",
            "eventName": "aws:kinesis:record",
            "invokeIdentityArn": "arn:aws:iam::123456789012:role/lambda-role",
            "awsRegion": "us-east-2",
            "eventSourceARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-
stream"
        }
    ]
}
```

使用 Lambda 處理 Amazon Kinesis Data Streams 記錄

若要使用 Lambda 處理 Amazon Kinesis Data Streams 記錄，請為您的串流建立一個取用者，然後建立 Lambda 事件來源對應。

設定您的資料串流及函數

您的 Lambda 函數是適用於資料串流的取用者應用程式。其會從每個碎片中一次處理一個批次的記錄。您可以將 Lambda 函數對應至共用輸送量取用程式 (標準迭代程式)，或對應至具有增強散發功能的專用輸送量取用程式。

- 標準迭代器：Lambda 會以每秒一次的基本速率輪詢 Kinesis 串流中的每個碎片以取得記錄。有更多記錄可用時，Lambda 會持續處理批次，直到函數掌握串流資訊。事件來源映射會與碎片的其他取用者共用讀取傳輸量。
- 增強型散發：若要將延遲降至最低並最大化讀取輸送量，請建立具有增強散發功能的資料串流取用者。增強散發取用者會取得每個碎片的專用連線，其不會影響其他從串流讀取的應用程式。串流取用者會使用 HTTP/2 來減少延遲，方法為透過長期連線將記錄推送至 Lambda，以及透過壓縮請求標頭。您可以使用 Kinesis 消費者 API 建立串流取用RegisterStream者。

```
aws kinesis register-stream-consumer \
--consumer-name con1 \
--stream-arn arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream
```

您應該會看到下列輸出：

```
{
    "Consumer": {
        "ConsumerName": "con1",
        "ConsumerARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream/
consumer/con1:1540591608",
        "ConsumerStatus": "CREATING",
        "ConsumerCreationTimestamp": 1540591608.0
    }
}
```

若要提高函數處理記錄的速度，請將碎片新增至資料串流。Lambda 會依序在每個碎片中處理記錄。如果您的函數傳回錯誤，則會停止處理碎片中的其他記錄。碎片越多，要一次處理的批次越多，這會降低錯誤對並行的影響。

如果您的函數無法擴展至可處理並行批次的總數，您可以請求增加配額，或為您的函數保留並行。

建立事件來源對應以叫用 Lambda 函數

若要使用資料串流中的記錄叫用 Lambda 函數，請建立事件來源對應。您可以建立多個事件來源映射，來使用多個 Lambda 函數處理相同資料，或使用單一函數處理來自多個資料串流的項目。處理來自多個串流的項目時，每個批次僅包含來自單一碎片或串流的記錄。

您可以設定事件來源對應，以處理來自不同串流的記錄 AWS 帳戶。如需進一步了解，請參閱the section called “跨帳戶對應”。

在建立事件來源對應之前，您需要授予 Lambda 函數從 Kinesis 資料串流讀取的權限。Lambda 需要下列權限來管理與 Kinesis 資料串流相關的資源：

- 室壁運動:DescribeStream
- 室壁運動:總DescribeStream結
- 室壁運動:GetRecords
- 室壁運動:GetShard迭代器
- 室壁運動:ListShards
- 室壁運動:ListStreams
- 室壁運動:SubscribeTo碎片

受 AWS 管理的策略AWSLambdaKinesisExecutionRole包括這些權限。將此受管理的原則新增至您的函數，如下列程序所述。

AWS Management Console

若要將 Kinesis 權限新增至您的函數

1. 開啟 Lambda 主控台的函數頁面，然後選取您的函數。
2. 在組態索引標籤中，選取權限。
3. 在 [執行角色] 窗格的 [角色名稱] 下，選擇函數執行角色的連結。此連結會在 IAM 主控台中開啟該角色的頁面。
4. 在「權限原則」窗格中，選擇「新增權限」，然後選取「附加原則」。
5. 在搜尋欄位中輸入 **AWSLambdaKinesisExecutionRole**。
6. 選取原則旁邊的核取方塊，然後選擇 [新增權限]。

AWS CLI

若要將 Kinesis 權限新增至您的函數

- 執行下列 CLI 命令，將AWSLambdaKinesisExecutionRole原則新增至函數的執行角色：

```
aws iam attach-role-policy \
--role-name MyFunctionRole \
--policy-arn arn:aws:iam::aws:policy/service-role/AWSLambdaKinesisExecutionRole
```

AWS SAM

若要將 Kinesis 權限新增至您的函數

- 在函數的定義中，新增Policies屬性，如下列範例所示：

```
Resources:
  MyFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: ./my-function/
      Handler: index.handler
      Runtime: nodejs20.x
      Policies:
        - AWSLambdaKinesisExecutionRole
```

設定所需權限後，請建立事件來源對應。

AWS Management Console

若要建立 Kinesis 事件來源對應

1. 開啟 Lambda 主控台的函數頁面，然後選取您的函數。
2. 在函數概觀窗格中，選擇新增觸發條件。
3. 在觸發器組態下，為來源選取 Kinesis。
4. 選取您要為其建立事件來源對應的 Kinesis 串流，並選擇性地選取串流的用戶。
5. (選擇性) 編輯事件來源對應的「Batch 大小」、「起始位置」和「Batch」視窗。
6. 選擇新增。

從主控台建立事件來源對應時，您的 IAM 角色必須具有 kinesis: ListStreams 和 kinesis：取用 ListStream者權限。

AWS CLI

若要建立 Kinesis 事件來源對應

- 執行下列 CLI 命令以建立 Kinesis 事件來源對應。根據您的使用案例選擇您自己的批次大小和起始位置。

```
aws lambda create-event-source-mapping \
--function-name MyFunction \
--event-source-arn arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream \
--starting-position LATEST \
--batch-size 100
```

若要指定批次化視窗，請加入選`--maximum-batching-window-in-seconds`項。若要取得有關使用此參數和其他參數的更多資訊，請參閱《指令參考》中的〈建立事件來源對映〉。AWS CLI

AWS SAM

若要建立 Kinesis 事件來源對應

- 在函數的定義中，新增`KinesisEvent`屬性，如下列範例所示：

```
Resources:
```

```
MyFunction:
  Type: AWS::Serverless::Function
  Properties:
    CodeUri: ./my-function/
    Handler: index.handler
    Runtime: nodejs20.x
    Policies:
      - AWSLambdaKinesisExecutionRole
    Events:
      KinesisEvent:
        Type: Kinesis
        Properties:
          Stream: !GetAtt MyKinesisStream.Arn
          StartingPosition: LATEST
          BatchSize: 100

MyKinesisStream:
  Type: AWS::Kinesis::Stream
  Properties:
    ShardCount: 1
```

若要進一步了解如何建立 Kinesis Data Streams 的事件來源對應 AWS SAM，請參閱AWS Serverless Application Model 開發人員指南中的 Kinesis。

輪詢和串流開始位置

請注意，建立和更新事件來源映射期間的串流輪詢最終會一致。

- 在建立事件來源映射期間，從串流開始輪詢事件可能需要幾分鐘時間。
- 在更新事件來源映射期間，從串流停止並重新開始輪詢事件可能需要幾分鐘時間。

這種行為表示如果您指定 LATEST 當作串流的開始位置，事件來源映射可能會在建立或更新期間遺漏事件。若要確保沒有遺漏任何事件，請將串流開始位置指定為 TRIM_HORIZON 或 AT_TIMESTAMP。

建立跨帳戶事件來源映射

Amazon Kinesis Data Streams 支援以資源為基礎的政策。因此，您可以 AWS 帳戶 使用另一個帳戶中的 Lambda 函數來處理擷取到串流中的資料。

若要使用不同的 Kinesis 串流為 Lambda 函數建立事件來源對應 AWS 帳戶，您必須使用以資源為基礎的政策來設定串流，以授與 Lambda 函數讀取項目的權限。若要了解如何設定串流以允許跨帳戶存取，請參閱 Amazon Kinesis Streams 開發人員指南中的與跨帳戶 AWS Lambda 功能共用存取權。

使用資源型政策設定串流後，可為 Lambda 函數提供所需權限，請使用上一節所述的任何方法建立事件來源對應。

如果您選擇使用 Lambda 主控台建立事件來源對應，請將串流的 ARN 直接貼到輸入欄位。如果您想要指定串流的取用者，貼上取用者的 ARN 會自動填入串流欄位。

使用 Kinesis Data Streams 和 Lambda 設定部分批次回應

取用和處理事件來源的串流資料時，依預設，只有在批次成功完成時，Lambda 檢查點才會到批次的最高序號。Lambda 會將所有其他結果視為完全失敗，並重試處理批次，直至達到重試限制。若要在處理串流的批次時允許部分成功，請開啟 `ReportBatchItemFailures`。允許部分成功有助於減少記錄的重試次數，但其不會完全消除在成功記錄中重試的可能性。

若要開啟`ReportBatchItemFailures`，請在 [FunctionResponse類型] 清單**`ReportBatchItemFailures`**中包含列舉值。此清單指示已為您的函數啟用哪些回應類型。您可以在建立或更新事件來源對應時設定此清單。

報告語法

設定批次項目失敗的報告時，會傳回 `StreamsEventResponse` 類別，其中包含批次項目失敗的清單。您可以使用 `StreamsEventResponse` 物件，來傳回批次中第一個失敗記錄的序號。您還可以使用正確的回應語法，建立自己的自訂類別。下列 JSON 結構顯示所需的回應語法：

```
{
  "batchItemFailures": [
        {
            "itemIdentifier": "<SequenceNumber>"
        }
    ]
}
```

> **Note**
>
> 如果 `batchItemFailures` 陣列包含多個項目，則 Lambda 會使用具有最低序列號的記錄作為檢查點。然後，Lambda 會重試從該檢查點開始的所有記錄。

成功與失敗條件

如果您傳回下列任一項目，Lambda 會將批次視為完全成功：

- 空白 `batchItemFailure` 清單
- Null `batchItemFailure` 清單
- 空白 `EventResponse`
- Null `EventResponse`

如果您傳回下列任一項目，Lambda 會將批次視為完全失敗：

- 空白字串 `itemIdentifier`
- Null `itemIdentifier`
- 具有錯誤金鑰名稱的 `itemIdentifier`

Lambda 會根據您的重試政策來重試失敗。

將批次平分

如果您的調用失敗且 `BisectBatchOnFunctionError` 已開啟，則無論您的 `ReportBatchItemFailures` 設定如何，批次都會被平分。

收到部分批次成功回應且 `BisectBatchOnFunctionError` 和 `ReportBatchItemFailures` 均開啟時，批次會依傳回的序號進行平分，並且 Lambda 僅會重試剩餘的記錄。

以下範例函數程式碼會傳回批次中失敗訊息 ID 的清單：

.NET

AWS SDK for .NET

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 .NET 搭配 Lambda 報告 Kinesis 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text;
using System.Text.Json.Serialization;
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
using AWS.Lambda.Powertools.Logging;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace KinesisIntegration;

public class Function
{
    // Powertools Logger requires an environment variables against your function
    // POWERTOOLS_SERVICE_NAME
    [Logging(LogEvent = true)]
    public async Task<StreamsEventResponse> FunctionHandler(KinesisEvent evnt,
 ILambdaContext context)
    {
        if (evnt.Records.Count == 0)
        {
            Logger.LogInformation("Empty Kinesis Event received");
            return new StreamsEventResponse();
        }

        foreach (var record in evnt.Records)
        {
            try
            {
                Logger.LogInformation($"Processed Event with EventId:
 {record.EventId}");
                string data = await GetRecordDataAsync(record.Kinesis, context);
                Logger.LogInformation($"Data: {data}");
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex)
            {
                Logger.LogError($"An error occurred {ex.Message}");
```

```
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                return new StreamsEventResponse
                {
                    BatchItemFailures = new
 List<StreamsEventResponse.BatchItemFailure>
                    {
                        new StreamsEventResponse.BatchItemFailure
 { ItemIdentifier = record.Kinesis.SequenceNumber }
                    }
                };
            }
        }
        Logger.LogInformation($"Successfully processed {evnt.Records.Count}
 records.");
        return new StreamsEventResponse();
    }

    private async Task<string> GetRecordDataAsync(KinesisEvent.Record record,
 ILambdaContext context)
    {
        byte[] bytes = record.Data.ToArray();
        string data = Encoding.UTF8.GetString(bytes);
        await Task.CompletedTask; //Placeholder for actual async work
        return data;
    }
}

public class StreamsEventResponse
{
    [JsonPropertyName("batchItemFailures")]
    public IList<BatchItemFailure> BatchItemFailures { get; set; }
    public class BatchItemFailure
    {
        [JsonPropertyName("itemIdentifier")]
        public string ItemIdentifier { get; set; }
    }
}
```

Go

SDK for Go V2

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 使用 Lambda 報告 Kinesis 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, kinesisEvent events.KinesisEvent)
 (map[string]interface{}, error) {
 batchItemFailures := []map[string]interface{}{}

 for _, record := range kinesisEvent.Records {
  curRecordSequenceNumber := ""

  // Process your record
  if /* Your record processing condition here */ {
   curRecordSequenceNumber = record.Kinesis.SequenceNumber
  }

  // Add a condition to check if the record processing failed
  if curRecordSequenceNumber != "" {
   batchItemFailures = append(batchItemFailures, map[string]interface{}
{"itemIdentifier": curRecordSequenceNumber})
  }
 }

 kinesisBatchResponse := map[string]interface{}{
```

```
  "batchItemFailures": batchItemFailures,
 }
 return kinesisBatchResponse, nil
}

func main() {
 lambda.Start(handler)
}
```

Java

適用於 Java 2.x 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

透過使用 Java 的 Lambda 報告 Kinesis 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

public class ProcessKinesisRecords implements RequestHandler<KinesisEvent,
 StreamsEventResponse> {

    @Override
    public StreamsEventResponse handleRequest(KinesisEvent input, Context
 context) {

        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<>();
```

```
        String curRecordSequenceNumber = "";

        for (KinesisEvent.KinesisEventRecord kinesisEventRecord :
 input.getRecords()) {
            try {
                //Process your record
                KinesisEvent.Record kinesisRecord =
 kinesisEventRecord.getKinesis();
                curRecordSequenceNumber = kinesisRecord.getSequenceNumber();

            } catch (Exception e) {
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                batchItemFailures.add(new
 StreamsEventResponse.BatchItemFailure(curRecordSequenceNumber));
                return new StreamsEventResponse(batchItemFailures);
            }
        }

       return new StreamsEventResponse(batchItemFailures);
    }
}
```

JavaScript

適用於 JavaScript (v3) 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Javascript 搭配 Lambda 報告 Kinesis 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
```

```
    try {
      console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      console.log(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      console.error(`An error occurred ${err}`);
      /* Since we are working with streams, we can return the failed item
 immediately.
           Lambda will immediately begin to retry processing from this failed
 item onwards. */
      return {
        batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
      };
    }
  }
  console.log(`Successfully processed ${event.Records.length} records.`);
  return { batchItemFailures: [] };
};

async function getRecordDataAsync(payload) {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

使用 Lambda 使用 TypeScript報告 Kinesis 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import {
  KinesisStreamEvent,
  Context,
  KinesisStreamHandler,
  KinesisStreamRecordPayload,
  KinesisStreamBatchResponse,
} from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
```

```
  serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (
  event: KinesisStreamEvent,
  context: Context
): Promise<KinesisStreamBatchResponse> => {
  for (const record of event.Records) {
    try {
      logger.info(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      logger.info(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      logger.error(`An error occurred ${err}`);
      /* Since we are working with streams, we can return the failed item
 immediately.
           Lambda will immediately begin to retry processing from this failed
 item onwards. */
      return {
        batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
      };
    }
  }
  logger.info(`Successfully processed ${event.Records.length} records.`);
  return { batchItemFailures: [] };
};

async function getRecordDataAsync(
  payload: KinesisStreamRecordPayload
): Promise<string> {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

PHP

適用於 PHP 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 PHP 使用 Lambda 報告 Kinesis 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kinesis\KinesisEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): array
    {
        $kinesisEvent = new KinesisEvent($event);
        $this->logger->info("Processing records");
        $records = $kinesisEvent->getRecords();
```

```
        $failedRecords = [];
        foreach ($records as $record) {
            try {
                $data = $record->getData();
                $this->logger->info(json_encode($data));
                // TODO: Do interesting work based on the new data
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $failedRecords[] = $record->getSequenceNumber();
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");

        // change format for the response
        $failures = array_map(
            fn(string $sequenceNumber) => ['itemIdentifier' => $sequenceNumber],
            $failedRecords
        );

        return [
            'batchItemFailures' => $failures
        ];
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

適用於 Python (Boto3) 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

透過使用 Python 的 Lambda 報告 Kinesis 批次項目失敗。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def handler(event, context):
    records = event.get("Records")
    curRecordSequenceNumber = ""

    for record in records:
        try:
            # Process your record
            curRecordSequenceNumber = record["kinesis"]["sequenceNumber"]
        except Exception as e:
            # Return failed record's sequence number
            return {"batchItemFailures":[{"itemIdentifier":
 curRecordSequenceNumber}]}

    return {"batchItemFailures":[]}
```

Ruby

適用於 Ruby 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Ruby 使用 Lambda 報告 Kinesis 批次項目失敗。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'aws-sdk'

def lambda_handler(event:, context:)
  batch_item_failures = []

  event['Records'].each do |record|
    begin
      puts "Processed Kinesis Event - EventID: #{record['eventID']}"
      record_data = get_record_data_async(record['kinesis'])
```

```
      puts "Record Data: #{record_data}"
      # TODO: Do interesting work based on the new data
    rescue StandardError => err
      puts "An error occurred #{err}"
      # Since we are working with streams, we can return the failed item
 immediately.
      # Lambda will immediately begin to retry processing from this failed item
 onwards.
      return { batchItemFailures: [{ itemIdentifier: record['kinesis']
['sequenceNumber'] }] }
    end
  end

  puts "Successfully processed #{event['Records'].length} records."
  { batchItemFailures: batch_item_failures }
end

def get_record_data_async(payload)
  data = Base64.decode64(payload['data']).force_encoding('utf-8')
  # Placeholder for actual async work
  sleep(1)
  data
end
```

Rust

適用於 Rust 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Rust 使用 Lambda 報告 Kinesis 批次項目故障。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::{
    event::kinesis::KinesisEvent,
    kinesis::KinesisEventRecord,
    streams::{KinesisBatchItemFailure, KinesisEventResponse},
```

```
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<KinesisEvent>) ->
 Result<KinesisEventResponse, Error> {
    let mut response = KinesisEventResponse {
        batch_item_failures: vec![],
    };

    if event.payload.records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(response);
    }

    for record in &event.payload.records {
        tracing::info!(
            "EventId: {}",
            record.event_id.as_deref().unwrap_or_default()
        );

        let record_processing_result = process_record(record);

        if record_processing_result.is_err() {
            response.batch_item_failures.push(KinesisBatchItemFailure {
                item_identifier: record.kinesis.sequence_number.clone(),
            });
            /* Since we are working with streams, we can return the failed item
 immediately.
            Lambda will immediately begin to retry processing from this failed
 item onwards. */
            return Ok(response);
        }
    }

    tracing::info!(
        "Successfully processed {} records",
        event.payload.records.len()
    );

    Ok(response)
}

fn process_record(record: &KinesisEventRecord) -> Result<(), Error> {
    let record_data = std::str::from_utf8(record.kinesis.data.as_slice());
```

```
    if let Some(err) = record_data.err() {
        tracing::error!("Error: {}", err);
        return Err(Error::from(err));
    }

    let record_data = record_data.unwrap_or_default();

    // do something interesting with the data
    tracing::info!("Data: {}", record_data);

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

在 Lambda 中保留 Kinesis Data Streams 事件來源的捨棄批次記錄

Kinesis 事件來源對映的錯誤處理取決於在叫用函數之前還是在函數叫用期間發生錯誤：

- 呼叫之前：如果 Lambda 事件來源對應由於節流或其他問題而無法叫用函數，則會重試直到記錄過期或超過事件來源對應上設定的保留天數上限 (秒)。MaximumRecord AgeIn
- 叫用期間：如果呼叫函數但傳回錯誤，Lambda 會重試直到記錄到期、超過最大保留天數 (MaximumRecordAgeIn秒) 或達到設定的重試配額 (MaximumRetry嘗試次數)。對於函數錯誤，您也可以設定 BisectBatchOnFunctionError，將失敗的批次分割成兩個較小的批次，隔離不良記錄並避免逾時。拆分批次不會消耗重試配額。

如果錯誤處理措施失敗，Lambda 會捨棄相應記錄，並繼續處理串流中的批次。使用預設設定時，這表示不良的記錄可能會封鎖受影響碎片上的處理長達一週。若要避免此情況，在設定函數的事件來源映射時，請使用合理的重試次數和符合您使用案例的記錄最大保留期。

設定失敗呼叫的目的地

若要保留失敗的事件來源映射調用記錄，請將目標地新增到函數的事件來源映射中。每個傳送至目的地的記錄都是 JSON 文件，其中包含有關失敗叫用的中繼資料。您可以將任何 Amazon SNS 主題或 Amazon SQS 佇列設定為目的地。您的執行角色必須具有目標的權限：

- 對於 SQS 目的地：sq s：SendMessage
- 對於 SNS 目的地：SNS: 發佈

若要使用主控台設定失敗時的目的地，請依照下列步驟執行：

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 在 函數概觀 下，選擇 新增目的地。
4. 針對來源，請選擇事件來源映射調用。
5. 對於事件來源映射，請選擇針對此函數設定的事件來源。
6. 對於條件，選取失敗時。對於事件來源映射調用，這是唯一可接受的條件。
7. 對於目標類型，請選擇 Lambda 將調用記錄傳送至的目標類型。
8. 對於 目的地，請選擇一個資源。
9. 選擇 儲存 。

您也可以使用 AWS Command Line Interface (AWS CLI) 來設定故障時的目的地。例如，下列建立事件來源對映指令會將具有失敗時之 SQS 目的地的事件來源對應新增至：MyFunction

```
aws lambda create-event-source-mapping \
--function-name "MyFunction" \
--event-source-arn arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sqs:us-
east-1:123456789012:dest-queue"}}'
```

下列更新-事件來源映射命令會更新事件來源對應，以便在兩次重試嘗試後或記錄超過一個小時，將失敗的叫用記錄傳送至 SNS 目的地。

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--maximum-retry-attempts 2 \
--maximum-record-age-in-seconds 3600 \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sns:us-
east-1:123456789012:dest-topic"}}'
```

系統會以非同步的方式套用更新的設定，在處理完成之前不會反映在輸出中。使用取得事件來源對映指令來檢視目前的狀態。

若要移除目的地，請提供空白字串作為 destination-config 參數的引數：

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": ""}}'
```

下列範例顯示 Lambda 針對失敗的 Kinesis 事件來源調用而傳送至 SQS 佇列或 SNS 主題。由於 Lambda 只會傳送這些目標類型的中繼資料，因此請使用shardIdstartSequenceNumber、、和endSequenceNumber欄位來取得完整的原始記錄。streamArn

```
{
    "requestContext": {
        "requestId": "c9b8fa9f-5a7f-xmpl-af9c-0c604cde93a5",
        "functionArn": "arn:aws:lambda:us-east-2:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted",
        "approximateInvokeCount": 1
    },
    "responseContext": {
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:38:06.021Z",
    "KinesisBatchInfo": {
        "shardId": "shardId-000000000001",
        "startSequenceNumber":
 "49601189658422359378836298521827638475320189012309704722",
        "endSequenceNumber":
 "49601189658422359378836298522902373528957594348623495186",
        "approximateArrivalOfFirstRecord": "2019-11-14T00:38:04.835Z",
```

```
        "approximateArrivalOfLastRecord": "2019-11-14T00:38:05.580Z",
        "batchSize": 500,
        "streamArn": "arn:aws:kinesis:us-east-2:123456789012:stream/mystream"
    }
}
```

您可以使用此資訊來從串流擷取受影響的記錄，以進行疑難排解。實際的記錄不包含在內，因此您必須處理此記錄，並在因過期而遺失之前從資料串流中擷取它們。

在 Lambda 中實作可設定狀態的 Kinesis Data Streams 處理

Lambda 函數可執行持續串流處理應用程式。串流表示持續在應用程式中流動的無限制資料。若要分析此持續更新輸入中的資訊，您可以使用定義的時段來限制包含的記錄。

輪轉時段是定期開啟和關閉的不同時段。依預設，Lambda 調用是無狀態的，您無法在沒有外部資料庫的情況下，將其用於處理多個持續調用的資料。然而，使用輪轉時段，您可以在不同的調用間維護狀態。此狀態包含之前為目前時段處理之訊息的彙總結果。狀態可以是每個分區最多 1 MB。如果超過該大小，則 Lambda 會提前終止時段。

串流中的每個記錄都屬於一個特定時段。Lambda 至少會處理一次每筆記錄，但不保證每筆記錄只會處理一次。在極少數情況下，例如錯誤處理，某些記錄可能會處理多次。第一次時一律會依序處理記錄。如果多次處理記錄，則可能不會按順序處理。

彙總與處理

調用您的使用者管理函數進行彙總，以及處理該彙總的最終結果。Lambda 會彙總時段中接收的所有記錄。您可以在多個批次中接收這些記錄，各自作為單獨的調用。每次調用會收到一個狀態。因此，當使用輪轉時段時，您的 Lambda 函數回應必須包含 `state` 屬性。如果回應不包含 `state` 屬性，Lambda 會將此視為失敗的調用。為了滿足此條件，您的函數可以返回一個 `TimeWindowEventResponse` 物件，它具有下列 JSON 形狀：

Example **`TimeWindowEventResponse`** 值

```
{
    "state": {
        "1": 282,
        "2": 715
    },
    "batchItemFailures": []
}
```

> **Note**
>
> 對於 Java 函數，我們建議使用 `Map<String, String>` 來表示狀態。

在時段結束時，標記 `isFinalInvokeForWindow` 會設定為 `true` 以指示這是最終狀態，並且可隨時進行處理。處理完成後，時段結束並完成最終調用，然後丟棄該狀態。

在時段結束時，Lambda 會針對彙總結果上的動作使用最終處理。您的最終處理將同步調用。成功調用後，您的函數檢查點序號和串流處理將會繼續。如果調用失敗，則您的 Lambda 函數會暫停進一步處理，直至成功調用。

Example KinesisTimeWindowEvent

```
{
    "Records": [
        {
            "kinesis": {
                "kinesisSchemaVersion": "1.0",
                "partitionKey": "1",
                "sequenceNumber":
 "49590338271490256608559692538361571095921575989136588898",
                "data": "SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
                "approximateArrivalTimestamp": 1607497475.000
            },
            "eventSource": "aws:kinesis",
            "eventVersion": "1.0",
            "eventID":
 "shardId-000000000006:49590338271490256608559692538361571095921575989136588898",
            "eventName": "aws:kinesis:record",
            "invokeIdentityArn": "arn:aws:iam::123456789012:role/lambda-kinesis-role",
            "awsRegion": "us-east-1",
            "eventSourceARN": "arn:aws:kinesis:us-east-1:123456789012:stream/lambda-
stream"
        }
    ],
    "window": {
        "start": "2020-12-09T07:04:00Z",
        "end": "2020-12-09T07:06:00Z"
    },
    "state": {
```

```
        "1": 282,
        "2": 715
    },
    "shardId": "shardId-000000000006",
    "eventSourceARN": "arn:aws:kinesis:us-east-1:123456789012:stream/lambda-stream",
    "isFinalInvokeForWindow": false,
    "isWindowTerminatedEarly": false
}
```

組態

您可以在建立或更新事件來源對映時設定輪轉時段。若要設定暫停視窗，請以秒為單位指定視窗 (TumblingWindowInSeconds)。下列範例 AWS Command Line Interface (AWS CLI) 命令會建立具有 120 秒暫停視窗的串流事件來源對應。針對彙總與處理定義的 Lambda 函數命名為 `tumbling-window-example-function`。

```
aws lambda create-event-source-mapping \
--event-source-arn arn:aws:kinesis:us-east-1:123456789012:stream/lambda-stream \
--function-name tumbling-window-example-function \
--starting-position TRIM_HORIZON \
--tumbling-window-in-seconds 120
```

Lambda 根據記錄插入串流的時間，確定輪轉時段邊界。所有記錄都有 Lambda 在邊界確定中使用的近似時間戳記。

輪轉時段彙總不支援重新分區。當碎片結束時，Lambda 認為當前窗口被關閉，並且任何子碎片都將以新的狀態啟動自己的窗口。當沒有新記錄新增至目前視窗時，Lambda 會等待最多 2 分鐘，然後再假設視窗結束。這有助於確保函數讀取當前視窗中的所有記錄，即使記錄間歇性地添加也是如此。

輪轉時段完全支援現有的重試政策 `maxRetryAttempts` 和 `maxRecordAge`。

Example Handler.py - 彙總與處理

下列 Python 函數示範了如何彙總，然後處理您的最終狀態：

```
def lambda_handler(event, context):
    print('Incoming event: ', event)
    print('Incoming state: ', event['state'])

#Check if this is the end of the window to either aggregate or process.
    if event['isFinalInvokeForWindow']:
```

```
        # logic to handle final state of the window
        print('Destination invoke')
    else:
        print('Aggregate invoke')

#Check for early terminations
    if event['isWindowTerminatedEarly']:
        print('Window terminated early')

    #Aggregation logic
    state = event['state']
    for record in event['Records']:
        state[record['kinesis']['partitionKey']] = state.get(record['kinesis']
['partitionKey'], 0) + 1

    print('Returning state: ', state)
    return {'state': state}
```

Amazon Kinesis Data Streams 事件來源對應的 Lambda 參數

所有 Lambda 事件來源對應共用相同CreateEventSourceMapping的 UpdateEventSourceMappingAPI 作業。但是，只有一些參數適用於 Kinesis。

適用於 Kinesis 的事件來源參數

參數	必要	預設	備註
BatchSize	否	100	上限：10,000
BisectBatchOnFunction錯誤	N	false	
DestinationConfig	N		用於放棄記錄的 Amazon SQS 佇列或 Amazon SNS 主題目的地。如需詳細資訊，請參閱 設定失敗呼叫的目的地。
已啟用	N	true	

參數	必要	預設	備註
EventSource阿恩	Y		資料串流或串流消費者的 ARN
FunctionName	Y		
FunctionResponse类型	N		若要讓函數報告批次中的特定失敗，請將值 `ReportBatchItemFailures` 包含在 `FunctionResponseTypes` 中。如需詳細資訊，請參閱 使用 Kinesis Data Streams 和 Lambda 設定部分批次回應。
MaximumBatchingWindowIn秒	N	0	
MaximumRecordAgeIn秒	N	-1	-1 表示無限：Lambda 不會捨棄記錄 (Kinesis Data Streams 資料保留設定仍適用) 下限：-1 上限：604,800
MaximumRetry嘗試	N	-1	-1 代表無限：系統會重試失敗的記錄，直到記錄過期為止 下限：-1 上限：10,000

參數	必要	預設	備註
ParallelizationFactor	N	1	上限：10
StartingPosition	Y		AT_TIMEST AMP、TRIM_ HORIZON 或 LATEST
StartingPosition時間戳	N		只有在設定 StartingP osition 為 AT_TIMEST AMP 時才有效。這是開始讀取的時間 (以 Unix 時間秒為單位)
TumblingWindowInSeconds	N		下限：0 上限：900

教學課程：搭配 Kinesis Data Streams 使用 Lambda

在本教學中，您會建立 Lambda 函數來使用 Amazon Kinesis 資料串流中的事件。

1. 自訂應用程式將記錄寫入串流。
2. AWS Lambda 輪詢串流，並在偵測到串流中的新記錄時叫用 Lambda 函數。
3. AWS Lambda 假設您在建立 Lambda 函數時指定的執行角色，以執行 Lambda 函數。

必要條件

此教學課程假設您具備基本的 Lambda 操作知識並了解 Lambda 主控台。若您尚未了解，請遵循 使用主控台建立一個 Lambda 函數 中的指示，建立您的第一個 Lambda 函數。

若要完成下列步驟，您需要 AWS Command Line Interface (AWS CLI) 版本 2。命令和預期的輸出會列在不同的區塊中：

```
aws --version
```

您應該會看到下列輸出：

```
aws-cli/2.13.27 Python/3.11.6 Linux/4.14.328-248.540.amzn2.x86_64 exe/x86_64.amzn.2
```

對於長命令，逸出字元 (\) 用於將命令分割為多行。

在 Linux 和 macOS 上，使用您偏好的 shell 和套件軟體管理工具。

> **Note**
>
> 在 Windows 中，作業系統的內建終端不支援您常與 Lambda 搭配使用的某些 Bash CLI 命令 (例如 `zip`)。若要取得 Ubuntu 和 Bash 的 Windows 整合版本，請安裝適用於 Linux 的 Windows 子系統。本指南中的 CLI 命令範例使用 Linux 格式。如果您使用的是 Windows CLI，必須重新格式化包含內嵌 JSON 文件的命令。

建立執行角色

建立可授與函數存取 AWS 資源之權限的執行角色。

若要建立執行角色

1. 在 IAM 主控台中開啟 角色頁面。
2. 選擇 建立角色。
3. 建立具備下列屬性的角色。
   - 信任實體 - AWS Lambda。
   - 權限 — AWSLambdaKinesisExecutionRole.
   - 角色名稱 - **`lambda-kinesis-role`**。

此AWSLambdaKinesisExecutionRole原則具有函數從 Kinesis 讀取項目並將記錄寫入記錄所需的 CloudWatch 權限。

建立函數

建立可處理 Kinesis 訊息的 Lambda 函數。函數程式碼會將 Kinesis 記錄的事件識別碼和事件資料記錄到記錄 CloudWatch 檔。

本教學課程使用 Node.js 18.x 執行期，但我們也有提供其他執行期語言的範例程式碼。您可以在下列方塊中選取索引標籤，查看您感興趣的執行期程式碼。您將在此步驟中使用的 JavaScript 程式碼位於JavaScript索引標籤中顯示的第一個範例中。

.NET

AWS SDK for .NET

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 .NET 搭配 Lambda 來使用 Kinesis 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
using AWS.Lambda.Powertools.Logging;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace KinesisIntegrationSampleCode;

public class Function
{
    // Powertools Logger requires an environment variables against your function
    // POWERTOOLS_SERVICE_NAME
    [Logging(LogEvent = true)]
    public async Task FunctionHandler(KinesisEvent evnt, ILambdaContext context)
    {
        if (evnt.Records.Count == 0)
        {
            Logger.LogInformation("Empty Kinesis Event received");
            return;
        }

        foreach (var record in evnt.Records)
        {
            try
            {
```

```
                Logger.LogInformation($"Processed Event with EventId:
 {record.EventId}");
                string data = await GetRecordDataAsync(record.Kinesis, context);
                Logger.LogInformation($"Data: {data}");
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex)
            {
                Logger.LogError($"An error occurred {ex.Message}");
                throw;
            }
        }
        Logger.LogInformation($"Successfully processed {evnt.Records.Count}
 records.");
    }

    private async Task<string> GetRecordDataAsync(KinesisEvent.Record record,
 ILambdaContext context)
    {
        byte[] bytes = record.Data.ToArray();
        string data = Encoding.UTF8.GetString(bytes);
        await Task.CompletedTask; //Placeholder for actual async work
        return data;
    }
}
```

Go

SDK for Go V2

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 搭配 Lambda 來使用 Kinesis 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main
```

```
import (
 "context"
 "log"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, kinesisEvent events.KinesisEvent) error {
 if len(kinesisEvent.Records) == 0 {
  log.Printf("empty Kinesis event received")
  return nil
 }

 for _, record := range kinesisEvent.Records {
  log.Printf("processed Kinesis event with EventId: %v", record.EventID)
  recordDataBytes := record.Kinesis.Data
  recordDataText := string(recordDataBytes)
  log.Printf("record data: %v", recordDataText)
  // TODO: Do interesting work based on the new data
 }
 log.Printf("successfully processed %v records", len(kinesisEvent.Records))
 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

適用於 Java 2.x 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Java 搭配 Lambda 來使用 Kinesis 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;

public class Handler implements RequestHandler<KinesisEvent, Void> {
    @Override
    public Void handleRequest(final KinesisEvent event, final Context context) {
        LambdaLogger logger = context.getLogger();
        if (event.getRecords().isEmpty()) {
            logger.log("Empty Kinesis Event received");
            return null;
        }
        for (KinesisEvent.KinesisEventRecord record : event.getRecords()) {
            try {
                logger.log("Processed Event with EventId: "+record.getEventID());
                String data = new String(record.getKinesis().getData().array());
                logger.log("Data:"+ data);
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex) {
                logger.log("An error occurred:"+ex.getMessage());
                throw ex;
            }
        }
        logger.log("Successfully processed:"+event.getRecords().size()+"
 records");
        return null;
    }

}
```

JavaScript

適用於 JavaScript (v3) 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Lambda 使用 JavaScript的 Kinesis 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    try {
      console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      console.log(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      console.error(`An error occurred ${err}`);
      throw err;
    }
  }
  console.log(`Successfully processed ${event.Records.length} records.`);
};

async function getRecordDataAsync(payload) {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

使用 Lambda 使用 TypeScript的 Kinesis 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import {
  KinesisStreamEvent,
```

```
  Context,
  KinesisStreamHandler,
  KinesisStreamRecordPayload,
} from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
  serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (
  event: KinesisStreamEvent,
  context: Context
): Promise<void> => {
  for (const record of event.Records) {
    try {
      logger.info(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      logger.info(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      logger.error(`An error occurred ${err}`);
      throw err;
    }
    logger.info(`Successfully processed ${event.Records.length} records.`);
  }
};

async function getRecordDataAsync(
  payload: KinesisStreamRecordPayload
): Promise<string> {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

PHP

適用於 PHP 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 PHP 使用 Lambda 消耗 Kinesis 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kinesis\KinesisEvent;
use Bref\Event\Kinesis\KinesisHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends KinesisHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleKinesis(KinesisEvent $event, Context $context): void
    {
        $this->logger->info("Processing records");
        $records = $event->getRecords();
        foreach ($records as $record) {
            $data = $record->getData();
```

```
            $this->logger->info(json_encode($data));
            // TODO: Do interesting work based on the new data

            // Any exception thrown will be logged and the invocation will be
 marked as failed
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

適用於 Python (Boto3) 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Python 搭配 Lambda 來使用 Kinesis 事件。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
import base64
def lambda_handler(event, context):

    for record in event['Records']:
        try:
            print(f"Processed Kinesis Event - EventID: {record['eventID']}")
            record_data = base64.b64decode(record['kinesis']
['data']).decode('utf-8')
            print(f"Record Data: {record_data}")
            # TODO: Do interesting work based on the new data
        except Exception as e:
            print(f"An error occurred {e}")
            raise e
```

```
    print(f"Successfully processed {len(event['Records'])} records.")
```

Ruby

適用於 Ruby 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用紅寶石使用 Lambda 消耗 Kinesis 事件。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'aws-sdk'

def lambda_handler(event:, context:)
  event['Records'].each do |record|
    begin
      puts "Processed Kinesis Event - EventID: #{record['eventID']}"
      record_data = get_record_data_async(record['kinesis'])
      puts "Record Data: #{record_data}"
      # TODO: Do interesting work based on the new data
    rescue => err
      $stderr.puts "An error occurred #{err}"
      raise err
    end
  end
  puts "Successfully processed #{event['Records'].length} records."
end

def get_record_data_async(payload)
  data = Base64.decode64(payload['data']).force_encoding('UTF-8')
  # Placeholder for actual async work
  # You can use Ruby's asynchronous programming tools like async/await or fibers
 here.
  return data
end
```

Rust

適用於 Rust 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Rust 使用 Lambda 消耗 Kinesis 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::kinesis::KinesisEvent;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<KinesisEvent>) -> Result<(), Error>
 {
    if event.payload.records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }

    event.payload.records.iter().for_each(|record| {
        tracing::info!("EventId:
 {}",record.event_id.as_deref().unwrap_or_default());

        let record_data = std::str::from_utf8(&record.kinesis.data);

        match record_data {
            Ok(data) => {
                // log the record data
                tracing::info!("Data: {}", data);
            }
            Err(e) => {
                tracing::error!("Error: {}", e);
            }
        }
    });

    tracing::info!(
        "Successfully processed {} records",
```

```
        event.payload.records.len()
    );

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

建立函數

1. 建立專案的目錄，然後切換至該目錄。

```
mkdir kinesis-tutorial
cd kinesis-tutorial
```

2. 將範例程 JavaScript 式碼複製到名為的新檔案中`index.js`。
3. 建立部署套件。

```
zip function.zip index.js
```

4. 使用 `create-function` 命令建立一個 Lambda 函數。

```
aws lambda create-function --function-name ProcessKinesisRecords \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs18.x \
--role arn:aws:iam::111122223333:role/lambda-kinesis-role
```

測試 Lambda 函數

使用 invoke AWS Lambda CLI 命令和 Kinesis 事件範例手動叫用 Lambda 函數。

測試 Lambda 函數

1. 將以下 JSON 複製到一個檔案中並儲存為 input.txt。

```
{
    "Records": [
        {
            "kinesis": {
                "kinesisSchemaVersion": "1.0",
                "partitionKey": "1",
                "sequenceNumber":
 "49590338271490256608559692538361571095921575989136588898",
                "data": "SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
                "approximateArrivalTimestamp": 1545084650.987
            },
            "eventSource": "aws:kinesis",
            "eventVersion": "1.0",
            "eventID":
 "shardId-000000000006:49590338271490256608559692538361571095921575989136588898",
            "eventName": "aws:kinesis:record",
            "invokeIdentityArn": "arn:aws:iam::111122223333:role/lambda-kinesis-
role",
            "awsRegion": "us-east-2",
            "eventSourceARN": "arn:aws:kinesis:us-east-2:111122223333:stream/
lambda-stream"
        }
    ]
}
```

2. 使用 invoke 命令來傳送事件到函數。

```
aws lambda invoke --function-name ProcessKinesisRecords \
--cli-binary-format raw-in-base64-out \
--payload file://input.txt outputfile.txt
```

如果您使用的是 AWS CLI 版本 2，則需要此cli-binary-format選項。若要讓此成為預設的設定，請執行 aws configure set cli-binary-format raw-in-base64-out。若要取得更多資

訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

回應已儲存至 out.txt。

建立 Kinesis 串流

使用 create-stream 命令來建立串流。

```
aws kinesis create-stream --stream-name lambda-stream --shard-count 1
```

執行下列 describe-stream 命令以取得串流 ARN。

```
aws kinesis describe-stream --stream-name lambda-stream
```

您應該會看到下列輸出：

```
{
    "StreamDescription": {
        "Shards": [
            {
                "ShardId": "shardId-000000000000",
                "HashKeyRange": {
                    "StartingHashKey": "0",
                    "EndingHashKey": "340282366920746074317682119384634633455"
                },
                "SequenceNumberRange": {
                    "StartingSequenceNumber":
 "49591073947768692513481539594623130411957558361251844610"
                }
            }
        ],
        "StreamARN": "arn:aws:kinesis:us-east-1:111122223333:stream/lambda-stream",
        "StreamName": "lambda-stream",
        "StreamStatus": "ACTIVE",
        "RetentionPeriodHours": 24,
        "EnhancedMonitoring": [
            {
                "ShardLevelMetrics": []
            }
        ],
        "EncryptionType": "NONE",
```

```
        "KeyId": null,
        "StreamCreationTimestamp": 1544828156.0
    }
}
```

在下一個步驟中，您會使用串流 ARN 與您的 Lambda 函數的串流建立關聯。

在 AWS Lambda中新增事件來源

執行下列 AWS CLI `add-event-source` 命令。

```
aws lambda create-event-source-mapping --function-name ProcessKinesisRecords \
--event-source  arn:aws:kinesis:us-east-1:111122223333:stream/lambda-stream \
--batch-size 100 --starting-position LATEST
```

記下映射 ID 以供後續使用。您可以執行 `list-event-source-mappings` 命令來取得事件來源映射的清單。

```
aws lambda list-event-source-mappings --function-name ProcessKinesisRecords \
--event-source arn:aws:kinesis:us-east-1:111122223333:stream/lambda-stream
```

在回應中，您可以確認狀態值為 `enabled`。您可以停用事件來源映射來暫時暫停輪詢，而不會遺失任何記錄。

測試設定

若要測試事件來源映射，請新增事件記錄到您的 Kinesis 串流。`--data` 值是一個字串，CLI 會先將該字串編碼為 base64，再將它傳送到 Kinesis。您可以執行相同命令一次以上，以新增多筆記錄到串流中。

```
aws kinesis put-record --stream-name lambda-stream --partition-key 1 \
--data "Hello, this is a test."
```

Lambda 使用執行角色自串流讀取記錄。接著，它調用您的 Lambda 函數，以記錄批次來傳遞。該功能對每個記錄中的數據進行解碼並對其進行記錄，並將輸出發送到 CloudWatch 日誌。在 CloudWatch 主控台中檢視日誌。

清除您的資源

除非您想要保留為此教學課程建立的資源，否則您現在便可刪除。刪除您不再使用的 AWS 資源，您可以避免不必要的費用 AWS 帳戶。

刪除執行角色

1. 開啟 IAM 主控台中的 角色頁面 。
2. 選取您建立的執行角色。
3. 選擇 刪除 。
4. 在文字輸入欄位中輸入角色的名稱，然後選擇 刪除 。

若要刪除 Lambda 函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇您建立的函數。
3. 選擇 Actions (動作)、Delete (刪除)。
4. 在文字輸入欄位中輸入 **delete**，然後選擇 刪除 。

刪除 Kinesis 串流

1. 登入 AWS Management Console 並開啟運動主控台，網址為 https://console.aws.amazon.com/kinesis。
2. 選取您建立的串流。
3. 選擇 動作、刪除。
4. 在文字輸入欄位中輸入 **delete**。
5. 選擇 刪除 。

搭配使用 Lambda 與 Amazon MQ

> **Note**
>
> 如果您想要將資料傳送到 Lambda 函數以外的目標，或在傳送資料之前豐富資料，請參閱 Amazon EventBridge 管道。

Amazon MQ 是一項受管訊息代理程式服務，適用於 Apache ActiveMQ 和 RabbitMQ。訊息代理程式透過主題或佇列事件目的地，允許軟體應用程式和元件使用各種程式設計語言、作業系統和正式簡訊協定來進行通訊。

Amazon MQ 還可以透過安裝 ActiveMQ 或 RabbitMQ 代理程式，並提供不同的網路拓撲和其他基礎架構需求，來代表您管理 Amazon Elastic Compute Cloud (Amazon EC2) 執行個體。

您可以使用 Lambda 函數來處理您的 Amazon MQ 訊息代理程式中的記錄。Lambda 透過事件來源映射叫用函數，這是從您的代理程式讀取訊息並同步叫用函數的 Lambda 資源。

> **Warning**
>
> Lambda 事件來源對應至少處理每個事件一次，並且可能會重複處理記錄。為了避免與重複事件相關的潛在問題，我們強烈建議您將函數代碼設為冪等。若要深入了解，請參閱 AWS 知識中心如何讓 Lambda 函數具有冪等性。

Amazon MQ 事件來源映射具有下列組態限制：

- 並行：使用 Amazon MQ 事件來源映射的 Lambda 函數具有預設並行上限設定。若使用 ActiveMQ，Lambda 服務會將並行執行環境的數量上限設為 5 個。若使用 RabbitMQ，並行執行環境的數量上限為 1 個。即使您變更函數的保留或佈建並行設定，Lambda 服務還是無法提供更多可用的執行環境。如要請求提升預設的並行上限，請聯絡 AWS Support。
- 跨帳戶 - Lambda 不支援跨帳戶處理。您無法用 Lambda 處理來自不同 AWS 帳戶中 Amazon MQ 訊息代理程式的記錄。
- 驗證 — 對於 ActiveMQ，僅支援 ActiveMQ SimpleAuthentication 外掛程式。對於 RabbitMQ，僅支援 PLAIN 身分驗證機制。使用者必須使用 AWS Secrets Manager 來管理其認證。如需 ActiveMQ 身分驗證的詳細資訊，請參閱 Amazon MQ 開發人員指南中的將 ActiveMQ 代理程式與 LDAP 整合。
- 連線配額 - 代理程式每個線路層級協定允許的連線數量上限。此配額以代理程式執行個體類型為基礎。如需詳細資訊，請參閱 Amazon MQ 開發人員指南中的 Amazon MQ 中的配額的代理程式。
- 連線 - 您可以在公有或私有 Virtual Private Cloud (VPC) 中建立代理程式。若是私有 VPC，您的 Lambda 函數需要存取 VPC 才能接收訊息。如需詳細資訊，請參閱本主題後面部分的 the section called “網路組態”。
- 事件目的地 - 僅支援佇列目的地。然而，您可以使用虛擬主題，當作為佇列與 Lambda 互動時，其行為在內部可作為主題。如需詳細資訊，請參閱 Apache ActiveMQ 網站上的虛擬目的地，以及 RabbitMQ 網站上的虛擬主機。
- 網路拓撲 - 對於 ActiveMQ，每個事件來源映射僅支援單一執行個體或待命代理程式。對於 RabbitMQ，每個事件來源映射僅支援單一執行個體代理程式或叢集部署。單一執行個體代理程式需要容錯移轉端點。如需這些代理程式部署模式的詳細資訊，請參閱 Amazon MQ 開發人員指南中的 ActiveMQ 代理程式架構和 Rabbit MQ 代理程式架構。

- 協定 - 支援的協定取決於 Amazon MQ 整合的類型。
  - 若是 ActiveMQ 整合，Lambda 會使用 OpenWire /Java 訊息服務 (JMS) 通訊協定取用訊息。不支援透過其他協定來取用訊息。在 JMS 協定中，僅支援 `TextMessage` 和 `BytesMessage`。Lambda 也支援 JMS 自訂屬性。如需有關通訊 OpenWire 協定的詳細資訊，請參閱 Apache OpenWire的 ActiveMQ 網站上。
  - 對於 RabbitMQ 整合，Lambda 透過 AMQP 0-9-1 協定來取用訊息。不支援透過其他協定來取用訊息。如需 RabbitMQ 實作 AMQP 0-9-1 協定的詳細資訊，請參閱 RabbitMQ 網站上的 AMQP 0-9-1 完整參考指南。

Lambda 會自動支援 Amazon MQ 支援的最新版 ActiveMQ 和 RabbitMQ。如需支援的最新版，請參閱 Amazon MQ 開發人員指南中的 Amazon MQ 版本備註。

> **Note**
>
> 根據預設，Amazon MQ 具有每週代理程式維護時段。代理程式在該時段不可用。若是無待命狀態的代理程式，則 Lambda 無法在該時段處理任何訊息。

章節

Lambda 取用者群組

若要與 Amazon MQ 互動，Lambda 會建立可以從您的 Amazon MQ 代理程式讀取的取用者群體。建立取用者群組時，會使用與事件來源映射 UUID 相同的 ID。

對於 Amazon MQ 事件來源，Lambda 會批次處理記錄，並在單個承載中將它們傳送到您的函數。要控制行為，您可以設定批次間隔和批次大小。Lambda 會提取訊息，直到它處理最大為 6 MB 的承載大小、批次間隔過期或記錄數達到完整批次大小。如需詳細資訊，請參閱 批次處理行為。

取用者群組會將訊息擷取為位元組 BLOB，並透過 base64 將其編碼成單一 JSON 承載，然後調用您的函數。如果函數針對批次中的任何訊息傳回錯誤，Lambda 會重試整個批次的訊息，直至處理成功或訊息過期。

> **Note**
>
> 雖然 Lambda 函數的逾時上限通常為 15 分鐘，但 Amazon MSK、自我管理的 Apache Kafka、Amazon DocumentDB 以及 Amazon MQ for ActiveMQ 和 Amazon MQ for RabbitMQ 的事件來源映射只支援 14 分鐘逾時限制上限的函數。此限制條件可確保事件來源映射能夠正確處理函數錯誤和重試。

您可以使用 Amazon CloudWatch 中的指`ConcurrentExecutions`標監視給定函數的並行使用情況。如需並行的詳細資訊，請參閱the section called “設定預留並行”。

Example Amazon MQ 記錄事件

ActiveMQ

```
{
   "eventSource": "aws:mq",
   "eventSourceArn": "arn:aws:mq:us-
west-2:111122223333:broker:test:b-9bcfa592-423a-4942-879d-eb284b418fc8",
   "messages": [
      {
        "messageID": "ID:b-9bcfa592-423a-4942-879d-eb284b418fc8-1.mq.us-
west-2.amazonaws.com-37557-1234520418293-4:1:1:1:1",
        "messageType": "jms/text-message",
        "deliveryMode": 1,
        "replyTo": null,
        "type": null,
        "expiration": "60000",
        "priority": 1,
        "correlationId": "myJMSCoID",
        "redelivered": false,
        "destination": {
          "physicalName": "testQueue"
        },
        "data":"QUJDOkFBQUE=",
        "timestamp": 1598827811958,
        "brokerInTime": 1598827811958,
        "brokerOutTime": 1598827811959,
```

```
        "properties": {
          "index": "1",
          "doAlarm": "false",
          "myCustomProperty": "value"
        }
      },
      {
        "messageID": "ID:b-9bcfa592-423a-4942-879d-eb284b418fc8-1.mq.us-
west-2.amazonaws.com-37557-1234520418293-4:1:1:1:1",
        "messageType": "jms/bytes-message",
        "deliveryMode": 1,
        "replyTo": null,
        "type": null,
        "expiration": "60000",
        "priority": 2,
        "correlationId": "myJMSCoID1",
        "redelivered": false,
        "destination": {
          "physicalName": "testQueue"
        },
        "data":"LQaGQ82S48k=",
        "timestamp": 1598827811958,
        "brokerInTime": 1598827811958,
        "brokerOutTime": 1598827811959,
        "properties": {
          "index": "1",
          "doAlarm": "false",
          "myCustomProperty": "value"
        }
      }
   ]
}
```

RabbitMQ

```
{
  "eventSource": "aws:rmq",
  "eventSourceArn": "arn:aws:mq:us-
west-2:111122223333:broker:pizzaBroker:b-9bcfa592-423a-4942-879d-eb284b418fc8",
  "rmqMessagesByQueue": {
    "pizzaQueue::/": [
      {
```

```
        "basicProperties": {
          "contentType": "text/plain",
          "contentEncoding": null,
          "headers": {
            "header1": {
              "bytes": [
                118,
                97,
                108,
                117,
                101,
                49
              ]
            },
            "header2": {
              "bytes": [
                118,
                97,
                108,
                117,
                101,
                50
              ]
            },
            "numberInHeader": 10
          },
          "deliveryMode": 1,
          "priority": 34,
          "correlationId": null,
          "replyTo": null,
          "expiration": "60000",
          "messageId": null,
          "timestamp": "Jan 1, 1970, 12:33:41 AM",
          "type": null,
          "userId": "AIDACKCEVSQ6C2EXAMPLE",
          "appId": null,
          "clusterId": null,
          "bodySize": 80
        },
        "redelivered": false,
        "data": "eyJ0aW1lb3V0IjowLCJkYXRhIjoiQ1pybWYwR3c4T3Y0YnFMUXhENEUifQ=="
      }
    ]
  }
```

```
}
```

> **Note**
>
> 在 RabbitMQ 範例中，`pizzaQueue` 是 RabbitMQ 佇列的名稱，`/` 是虛擬主機的名稱。接收訊息時，事件來源會在 `pizzaQueue::/` 列出訊息。

執行角色許可

若要從 Amazon MQ 代理程式讀取記錄，您的 Lambda 函數需要將下列許可新增至其執行角色：

- mq：DescribeBroker
- 秘密經理:價值 GetSecret
- ec2：CreateNetwork接口
- ec2：DeleteNetwork接口
- ec2：DescribeNetwork接口
- ec2：DescribeSecurity群組
- ec2：DescribeSubnets
- ec2：DescribeVpcs
- 記錄檔:CreateLog群組
- 記錄檔:CreateLog串流
- 記錄檔:PutLog事件

> **Note**
>
> 使用加密的客戶管理金鑰時，也請新增 `kms:Decrypt` 許可。

網路組態

若要透過事件來源映射授予 Lambda 代理程式的完整存取權，您的代理程式必須使用公有端點 (公有 IP 地址)，或者您必須提供建立代理程式之 Amazon VPC 的存取權。

預設情況下，當您建立 Amazon MQ 代理程式時，`PubliclyAccessible` 旗標會設為 false。為了讓您的代理程式能夠接收到公有 IP 地址，您必須將 `PubliclyAccessible` 旗標設為 true。

將 Amazon MQ 與 Lambda 搭配使用的最佳做法是使用 AWS PrivateLink 虛擬私人雲端端點，並將 Lambda 函數存取權授與代理程式的 VPC。部署 Lambda 的端點，以及 () 的端點 (僅適用於 ActiveMQ AWS Security Token Service)AWS STS的端點。如果您的代理程式使用驗證，請同時為 AWS Secrets Manager. 如需進一步了解，請參閱the section called “使用 VPC 端點”。

或者，在包含 Amazon MQ 代理程式的 VPC 中的每個公有子網路上設定 NAT 閘道。如需詳細資訊，請參閱 the section called “VPC 功能的網際網路存取”。

當您為 Amazon MQ 代理程式建立事件來源對應時，Lambda 會檢查代理程式 VPC 的子網路和安全群組是否已存在彈性網路界面 (ENI)。如果 Lambda 找到現有的 ENI，它會嘗試重複使用它們。否則，Lambda 會建立新的 ENI 以連接至事件來源並叫用您的函數。

> **Note**
>
> Lambda 函數一律在 Lambda 服務擁有的 VPC 內執行。這些 VPC 由服務自動維護，客戶看不到。您也可以將您的功能連接到 Amazon VPC。在任何一種情況下，函數的 VPC 配置都不會影響事件源映射。只有事件來源 VPC 的組態才會決定 Lambda 連線至事件來源的方式。

VPC 安全群組規則

使用下列規則 (至少) 為包含叢集的 Amazon VPC 設定安全群組：

- 傳入規則：允許為事件來源指定之安全群組的代理程式連接埠上的所有流量來自其安全群組。預設情況下，ActiveMQ 會使用連接埠 61617，RabbitMQ 則會使用連接埠 5671。
- 傳出規則：針對所有目的地，允許連接埠 443 上的所有流量。允許代理程式連接埠上的所有流量位於其安全群組。預設情況下，ActiveMQ 會使用連接埠 61617，RabbitMQ 則會使用連接埠 5671。
- 如果您使用的是 VPC 端點而不是 NAT 閘道，則與 VPC 端點相關聯的安全群組必須允許連接埠 443 上所有來自事件來源安全群組的傳入流量。

使用 VPC 端點

當您使用 VPC 端點時，會使用 ENI 透過這些端點路由呼叫函數的 API 呼叫。Lambda 服務主體需要呼叫`lambda:InvokeFunction`使用這些 ENI 的任何函數。此外，對於 ActiveMQ，Lambda 服務主體需要呼叫`sts:AssumeRole`使用 ENI 的角色。

根據預設，VPC 端點具有開放的 IAM 政策。最佳做法是將這些原則限制為僅允許特定主參與者使用該端點執行所需的動作。為了確保您的事件來源對應能夠叫用 Lambda 函數，虛擬私人雲端端點政策必須允許 Lambda 服務原則呼叫，`lambda:InvokeFunction`而且如果是 ActiveMQ，則必須允許 Lambda 服務原則呼叫。`sts:AssumeRole`將 VPC 端點原則限制為僅允許來自組織內的 API 呼叫，可防止事件來源對應正常運作。

下列範例 VPC 端點原則示範如何授與 AWS STS 和 Lambda 端點所需的存取權。

Example VPC 私人雲端端點策略- AWS STS 端點（僅適用於 ActiveMQ）

```
{
    "Statement": [
        {
            "Action": "sts:AssumeRole",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
}
```

Example VPC 私人雲端端點政策

```
{
    "Statement": [
        {
            "Action": "lambda:InvokeFunction",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
}
```

如果您的 Amazon MQ 代理程式使用身份驗證，您也可以限制 Secrets Manager 端點的 VPC 端點政策。若要呼叫機 Secrets Manager API，Lambda 會使用您的函數角色，而不是使用 Lambda 服務主體。下列範例顯示 Secrets Manager 端點策略。

Example VPC 端點原則-Secrets Manager 端點

```
{
    "Statement": [
        {
            "Action": "secretsmanager:GetSecretValue",
            "Effect": "Allow",
            "Principal": {
                "AWS": [
                    "customer_function_execution_role_arn"
                ]
            },
            "Resource": "customer_secret_arn"
        }
    ]
}
```

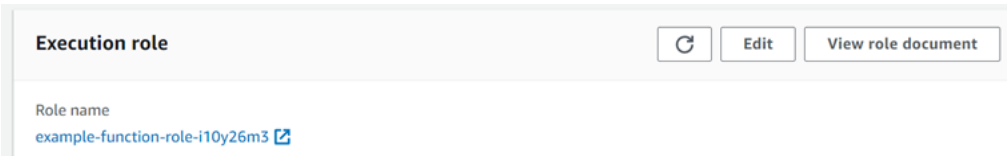
新增權限並建立事件來源對應

建立事件來源映射，指示 Lambda 將記錄從 Amazon MQ 代理程式傳送至 Lambda 函數。您可以建立多個事件來源映射，來使用多個函數處理相同資料，或使用單一函數處理來自多個來源的項目。

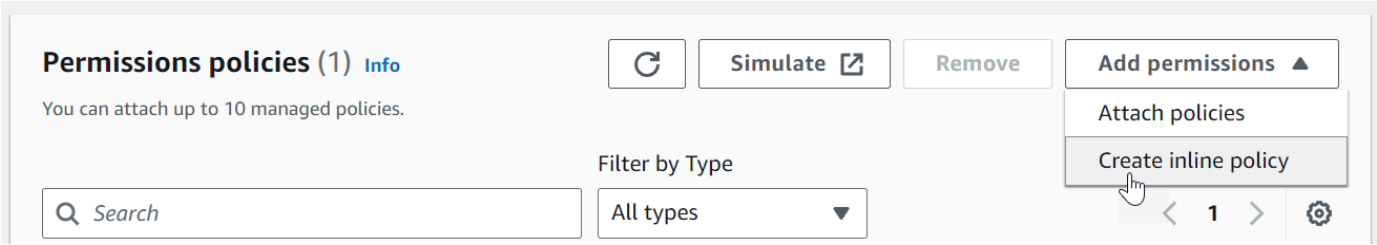
若要將函數設定為從 Amazon MQ 讀取，請新增必要的許可，並在 Lambda 主控台中建立 MQ 觸發器。

若要新增權限並建立觸發器

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇函數的名稱。
3. 依序選擇 Configuration (組態) 索引標籤和 Permissions (許可)。
4. 在 [角色名稱] 下，選擇指向您的執行角色的連結。此連結會在 IAM 主控台中開啟角色。



5. 選擇 [新增權限]，然後選擇 [建立內嵌原則]。

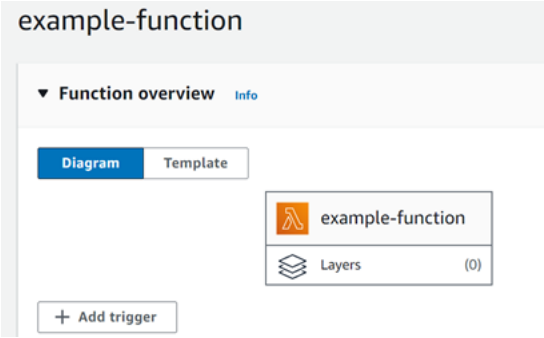


6. 在 [原則編輯器] 中，選擇 [JSON]。輸入下列政策。您的函數需要這些許可才能從 Amazon MQ 代理程式讀取。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "mq:DescribeBroker",
        "secretsmanager:GetSecretValue",
        "ec2:CreateNetworkInterface",
        "ec2:DeleteNetworkInterface",
        "ec2:DescribeNetworkInterfaces",
        "ec2:DescribeSecurityGroups",
        "ec2:DescribeSubnets",
        "ec2:DescribeVpcs",
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents"
      ],
      "Resource": "*"
    }
  ]
}
```

> **Note**
>
> 使用加密的客戶管理金鑰時，您還必須新增`kms:Decrypt`權限。

7. 選擇下一步。輸入策略名稱，然後選擇 [建立策略]。
8. 返回 Lambda 主控台中的函數。在 函式概觀 下，選擇 新增觸發條件 。



9. 選擇 MQ 觸發器類型。
10. 設定需要的選項，然後選擇 新增 。

Lambda 支援 Amazon MQ 事件來源的下列選項：

- MQ broker (MQ 代理程式) – 選取 Amazon MQ 代理程式。
- Batch size (批次大小) - 設定單一批次中要擷取的訊息數目上限。
- Queue name (佇列名稱) - 輸入要取用的 Amazon MQ 佇列。
- Source access configuration (來源存取組態) - 輸入儲存您代理程式憑證的虛擬主機資訊和 Secrets Manager 機密。
- Enable trigger (啟用觸發條件) - 停用觸發條件以停止處理記錄。

若要啟用或停用觸發條件 (或將其刪除)，請在設計工具中選擇 MQ 觸發條件。若要重新設定觸發條件，請使用事件來源映射 API 操作。

更新事件來源對應

使用指[update-event-source-mapping](#)令更新事件來源對應。下列範例命令會將事件來源映射更新為具有批次大小 2。

```
aws lambda update-event-source-mapping \
--uuid 91eaeb7e-c976-1234-9451-8709db01f137 \
--batch-size 2
```

您應該會看到下列輸出：

```
{
    "UUID": "91eaeb7e-c976-1234-9451-8709db01f137",
```

```
    "BatchSize": 2,
    "EventSourceArn": "arn:aws:mq:us-east-1:123456789012:broker:ExampleMQBroker:b-
b4d492ef-bdc3-45e3-a781-cd1a3102ecca",
    "FunctionArn": "arn:aws:lambda:us-east-1:123456789012:function:MQ-Example-
Function",
    "LastModified": 1601928393.531,
    "LastProcessingResult": "No records processed",
    "State": "Updating",
    "StateTransitionReason": "USER_INITIATED"
}
```

Lambda 會以非同步方式更新這些設定。在此程序完成之前，輸出不會反映變更。使用 get-event-source-mapping 命令可檢視資源的目前狀態。

```
aws lambda get-event-source-mapping \
--uuid 91eaeb7e-c976-4939-9451-8709db01f137
```

您應該會看到下列輸出：

```
{
    "UUID": "91eaeb7e-c976-4939-9451-8709db01f137",
    "BatchSize": 2,
    "EventSourceArn": "arn:aws:mq:us-east-1:123456789012:broker:ExampleMQBroker:b-
b4d492ef-bdc3-45e3-a781-cd1a3102ecca",
    "FunctionArn": "arn:aws:lambda:us-east-1:123456789012:function:MQ-Example-
Function",
    "LastModified": 1601928393.531,
    "LastProcessingResult": "No records processed",
    "State": "Enabled",
    "StateTransitionReason": "USER_INITIATED"
}
```

事件來源映射錯誤

當 Lambda 函數遇到無法復原的錯誤時，您的 Amazon MQ 取用者將停止處理記錄。任何其他取用者可能會繼續處理，只要他們沒有遇到相同的錯誤。若要判斷停止 EventSourceMapping 取用者的潛在原因，請檢查傳回詳細資料中的 StateTransitionReason 欄位，以取得下列其中一個程式碼：

ESM_CONFIG_NOT_VALID

事件來源對應組態無效。

EVENT_SOURCE_AUTHN_ERROR

Lambda 驗證事件來源失敗。

EVENT_SOURCE_AUTHZ_ERROR

Lambda 沒有存取事件來源所需的許可。

FUNCTION_CONFIG_NOT_VALID

該函數的配置無效。

如果記錄因大小而遭 Lambda 棄置，則也將不會得到處理。Lambda 記錄的大小限制為 6MB。若要在函數錯誤時重新傳遞訊息，您可以使用無效字母佇列 (DLQ)。如需詳細資訊，請參閱 Apache ActiveMQ 網站上的訊息重新傳遞和 DLQ 處理，以及 RabbitMQ 網站上的可靠性指南。

> **Note**
>
> Lambda 不支援自訂重新傳遞政策，相反地，Lambda 會使用 Apache ActiveMQ 網站上「重新傳遞政策」頁面中預設值的原則，且`maximumRedeliveries`設定為 6。

Amazon MQ 和 RabbitMQ 組態參數

所有 Lambda 事件來源類型都共用相同CreateEventSourceMapping的UpdateEventSourceMappingAPI 作業。但是，只有一些參數適用於 Amazon MQ 和 RabbitMQ。

適用於 Amazon MQ 和 RabbitMQ 的事件來源參數

參數	必要	預設	備註
BatchSize	否	100	上限：10,000
已啟用	N	true	
FunctionName	Y		
FilterCriteria	N		Lambda 事件篩選
MaximumBatchingWindowIn秒	N	500 毫秒	批次處理行為

參數	必要	預設	備註
佇列	N		要使用的 Amazon MQ 代理程式目的地佇列的名稱。
SourceAccess配置	N		若為 ActiveMQ，可使用 BASIC_AUTH 憑證。若為 RabbitMQ，可同時包含 BASIC_AUTH 憑證和 VIRTUAL_HOST 資訊。

搭配使用 Lambda 與 Amazon MSK

> **Note**
>
> 如果您想要將資料傳送到 Lambda 函數以外的目標，或在傳送資料之前豐富資料，請參閱 Amazon EventBridge 管道。

Amazon Managed Streaming for Apache Kafka (Amazon MSK) 是一項全受管服務，可讓您建立和執行使用 Apache Kafka 處理串流資料的應用程式。Amazon MSK 可簡化執行 Kafka 叢集的設定、擴展和管理。Amazon MSK 還可讓您更輕鬆地為多個可用區域設定應用程式，以及使用 AWS Identity and Access Management (IAM) 進行安全性。Amazon MSK 支援 Kafka 的多個開放原始碼版本。

Amazon MSK 作為事件來源時，其運作方式類似於使用 Amazon Simple Queue Service (Amazon SQS) 或 Amazon Kinesis。Lambda 會在內部輪詢事件來源中的新訊息，然後同步調用目標 Lambda 函數。Lambda 會批次讀取訊息，並將這些訊息作為事件酬載提供給函數。最大批次大小可進行設定 (預設為 100 則訊息)。如需詳細資訊，請參閱 批次處理行為。

> **Note**
>
> 雖然 Lambda 函數的逾時上限通常為 15 分鐘，但 Amazon MSK、自我管理的 Apache Kafka、Amazon DocumentDB 以及 Amazon MQ for ActiveMQ 和 Amazon MQ for RabbitMQ

的事件來源映射只支援 14 分鐘逾時限制上限的函數。此限制條件可確保事件來源映射能夠正確處理函數錯誤和重試。

Lambda 會依序讀取每個分割區的訊息。單一 Lambda 承載可以包含來自多個分割區的訊息。Lambda 處理每個批次後，會遞交該批次中訊息的偏移量。如果函數針對批次中的任何訊息傳回錯誤，Lambda 會重試整個批次的訊息，直至處理成功或訊息過期。

> **Warning**
>
> Lambda 事件來源對應至少處理每個事件一次，並且可能會重複處理記錄。為了避免與重複事件相關的潛在問題，我們強烈建議您將函數代碼設為冪等。若要深入了解，請參閱 AWS 知識中心如何讓 Lambda 函數具有冪等性。

如需如何將 Amazon MSK 設定為事件來源的範例，請參閱 AWS 運算部落格 AWS Lambda上的使用 Amazon MSK 做為事件來源。請參閱 Amazon MSK Labs 中的 Amazon MSK Lambda 整合以取得完整的教學課程。

主題

- 教學課程：使用 Amazon MSK 事件來源對應來叫用 Lambda 函數
- 範例事件
- MSK 叢集身分驗證
- 管理 API 存取和許可
- 身分驗證和授權錯誤
- 網路組態
- 將 Amazon MSK 新增為事件來源
- 建立跨帳戶事件來源映射
- 失敗時的目的地
- Amazon MSK 事件來源的自動調整規模
- 輪詢和串流開始位置
- Amazon CloudWatch 指標
- Amazon MSK 組態參數

教學課程：使用 Amazon MSK 事件來源對應來叫用 Lambda 函數

在本教學課程中，您將執行下列操作：

- 在與現有 Amazon MSK 叢集相同的 AWS 帳戶中建立 Lambda 函數。
- 設定 Lambda 的聯網和身份驗證，以便與 Amazon MSK 進行通訊。
- 設定 Lambda Amazon MSK 事件來源對應，當事件出現在主題中時，會執行 Lambda 函數。

完成這些步驟後，當事件傳送至 Amazon MSK 時，您將能夠設定 Lambda 函數，使用您自己的自訂 Lambda 程式碼自動處理這些事件。

您可以使用此功能做什麼？

範例解決方案：使用 MSK 事件來源對應，為您的客戶提供即時分數。

請考慮下列案例：您的公司主控一個 Web 應用程式，您的客戶可以在其中檢視即時賽事的相關資訊，例如體育比賽。遊戲的資訊更新會透過 Amazon MSK 上的 Kafka 主題提供給您的團隊。您想要設計一個解決方案，使用 MSK 主題的更新，以便為您開發的應用程式內的客戶提供現場活動的更新檢視。您已決定採用下列設計方法：您的用戶端應用程式將與託管於 AWS. 用戶端將使用 Amazon API Gateway WebSocket API 透過網路通訊端工作階段進行連線。

在此解決方案中，您需要一個可讀取 MSK 事件的元件，執行一些自訂邏輯，為應用程式層準備這些事件，然後將該資訊轉送至 API Gateway API。您可以透過在 Lambda 函數中提供自訂邏輯，然後使用 AWS Lambda Amazon MSK 事件來源對應來呼叫它來實作此元件。 AWS Lambda

如需使用 Amazon API Gateway WebSocket API 實作解決方案的詳細資訊，請參閱 WebSocket API 閘道文件中的 API 教學課程。

必要條件

具有下列預先設定資源的 AWS 帳號：

為了滿足這些先決條件，我們建議您遵循 Amazon MSK 文件中的「開始使用 Amazon MSK」。

- Amazon MSK 叢集。請參閱開始使用 Amazon MSK 中的建立 Amazon MS K 叢集。
- 以下配置：
  - 確保叢集安全性設定中已啟用 IAM 角色型身份驗證。如此可將 Lambda 函數限制為僅存取所需的 Amazon MSK 資源，藉此提升您的安全性。根據預設，新的 Amazon MSK 叢集會啟用此功能。

  - 確定叢集網路設定中的公用存取已關閉。限制 Amazon MSK 叢集對網際網路的存取權限，可限制處理資料的中介機構數目，從而提高安全性。根據預設，新的 Amazon MSK 叢集會啟用此功能。
- Amazon MSK 叢集中用於此解決方案的卡夫卡主題。請參閱開始使用 Amazon MSK 中的建立主題。
- Kafka 管理員主機設定用於從您的卡夫卡叢集擷取資訊，並將卡夫卡事件傳送到您的主題進行測試，例如安裝了 Kafka 管理 CLI 和 Amazon MSK IAM 程式庫的 Amazon EC2 執行個體。請參閱開始使用 Amazon MSK 中的建立用戶端電腦。

設定完這些資源後，請從您的 AWS 帳戶收集下列資訊，以確認您已準備好繼續。

- 您的 Amazon MSK 叢集的名稱。您可以在 Amazon MSK 主控台中找到此資訊。
- 叢集 UUID 是您 Amazon MSK 叢集 ARN 的一部分，您可以在 Amazon MSK 主控台中找到。請遵循 Amazon MSK 文件中列出叢集中的程序，以尋找此資訊。
- 與您的 Amazon MSK 叢集相關聯的安全群組。您可以在 Amazon MSK 主控台中找到此資訊。在下列步驟中，請將這些步驟稱為您的*##SecurityGroups*。
- 包含您的 Amazon MSK 叢集的 Amazon VPC 識別碼。您可以在 Amazon MSK 主控台中識別與 Amazon MSK 叢集相關聯的子網路，然後在 Amazon VPC 主控台中識別與子網路相關聯的 Amazon VPC，以找到此資訊。
- 解決方案中使用的 Kafka 主題名稱。您可以從 Kafka 管理主機使用 Kafka `topics` CLI 呼叫 Amazon MSK 叢集，以找到此資訊。如需 CLI 主題的詳細資訊，請參閱 Kafka 文件集中的新增和移除主題。
- 您卡夫卡主題的用戶群組名稱，適合由 Lambda 函數使用。這個群組可以由 Lambda 自動建立，因此您不需要使用 Kafka CLI 建立群組。如果您確實需要管理用戶群組，若要深入了解使用者群組 CLI，請參閱 Kafka 說明文件中的管理用戶群組。

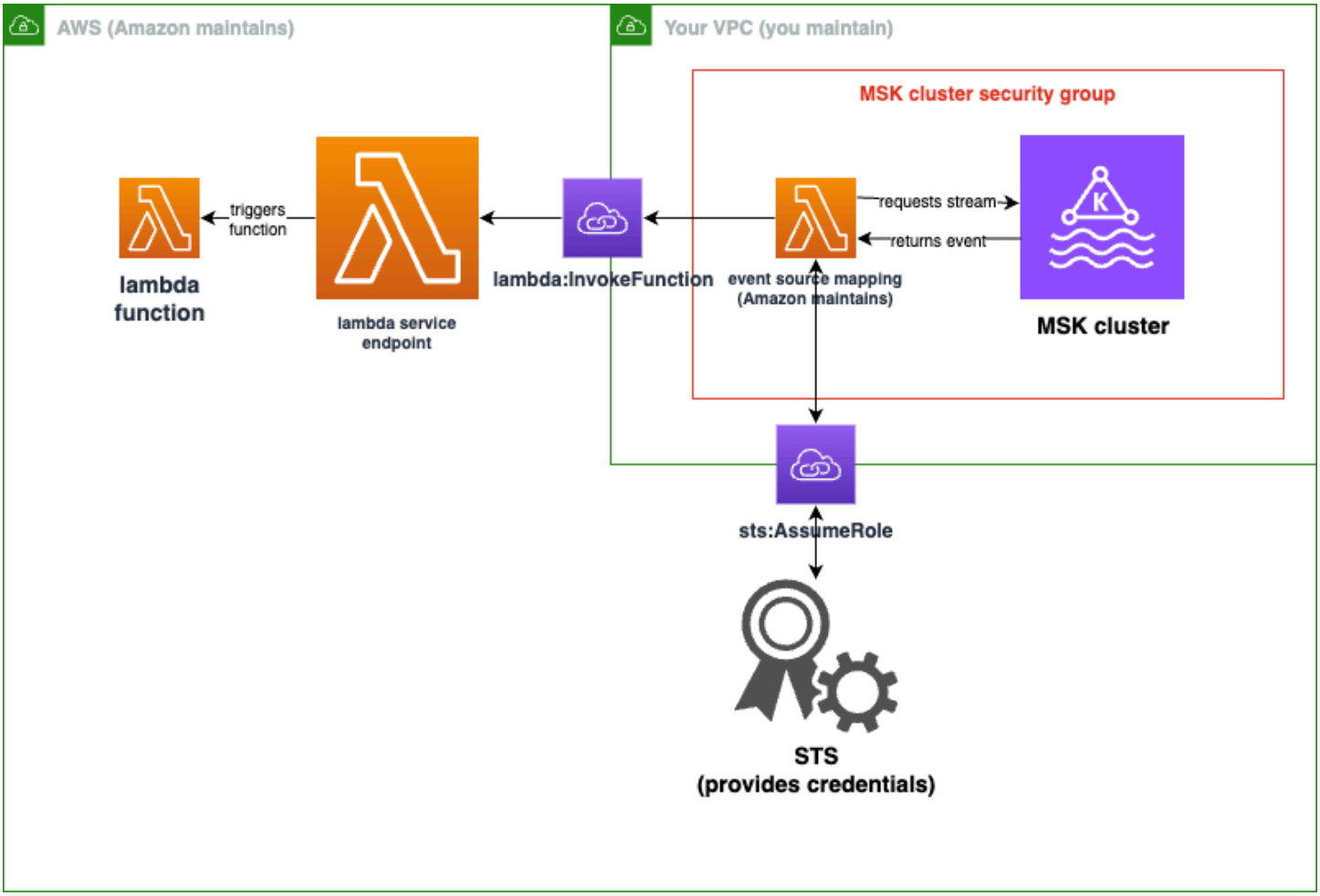
您 AWS 帳戶中的下列權限：

- 建立和管理 Lambda 函數的權限。
- 建立 IAM 政策並將其與 Lambda 函數建立關聯的權限。
- 允許在託管您的 Amazon MSK 叢集的 Amazon VPC 中建立 Amazon VPC 端點和變更聯網組態。

設定 Lambda 的網路連線能力，以便與 Amazon MSK 通訊

用 AWS PrivateLink 於連接 Lambda 和 Amazon MSK。您可以透過在 Amazon VPC 主控台中建立界面 Amazon VPC 端點來執行此操作。如需網路組態的詳細資訊，請參閱the section called “網路組態”。

當 Amazon MSK 事件來源對應代表 Lambda 函數執行時，它會擔任 Lambda 函數的執行角色。此 IAM 角色授權對應存取 IAM 保護的資源，例如 Amazon MSK 叢集。雖然這些元件共用執行角色，但 Amazon MSK 對應和 Lambda 函數對其各自的任務有不同的連線需求，如下圖所示。



您的事件來源對應屬於您的 Amazon MSK 叢集安全群組。在此聯網步驟中，從 Amazon MSK 叢集 VPC 建立 Amazon VPC 端點，以將事件來源對應連接至 Lambda 和 STS 服務。保護這些端點以接受來自 Amazon MSK 叢集安全群組的流量。然後，調整 Amazon MSK 叢集安全群組，以允許事件來源對應與 Amazon MSK 叢集進行通訊。

您可以使用配置下列步驟 AWS Management Console。

設定介面 Amazon VPC 端點以連接 Lambda 和 Amazon MSK

1. *##### Amazon VPC ## (##) ############### 443 ### TCP ###SecurityGroup SecurityGroups*遵循 Amazon EC2 文件中建立安全群組中的程序來建立安全群組。然後，按照 Amazon EC2 文件中將規則新增至安全群組中的程序來新增適當的規則。

   使用下列資訊建立安全性群組：

新增輸入規則時，請為#集中的每個安全性群組建立規則SecurityGroups。對於每個規則：

- 選取 HTTPS 做為「類型」。
- 針對來源，選取其中一個*##SecurityGroups*。

2. 建立一個端點，將 Lambda 服務連接到包含您的 Amazon MSK 叢集的 Amazon VPC。遵循建立介面端點中的程序。

   使用下列資訊建立介面端點：

   - 對於服務名稱com.amazonaws.*regionName*.lambda，請選取####託管 Lambda 函數的位置。
   - 對於虛擬私人雲端，請選取包含您的 Amazon MSK 叢集的 Amazon VPC。
   - 針對「安全性群組」SecurityGroup，選取您先前建立的##。
   - 對於子網路，請選取託管 Amazon MSK 叢集的子網路。
   - 針對原則，請提供下列原則文件，以保護端點，以供 Lambda 服務主體使用lambda:InvokeFunction動作。

   ```
   {
       "Statement": [
           {
               "Action": "lambda:InvokeFunction",
               "Effect": "Allow",
               "Principal": {
                   "Service": [
                       "lambda.amazonaws.com"
                   ]
               },
               "Resource": "*"
           }
       ]
   }
   ```

   - 確保啟用 DNS 名稱保持設置狀態。

3. 建立一個端點，將 AWS STS 服務連接到包含您的 Amazon MSK 叢集的 Amazon VPC。遵循建立介面端點中的程序。

   使用下列資訊建立介面端點：

   - 對於服務名稱，選取 AWS STS。

- 對於虛擬私人雲端，請選取包含您的 Amazon MSK 叢集的 Amazon VPC。
- 對於安全群組，選取*##SecurityGroup*。
- 對於子網路，請選取託管 Amazon MSK 叢集的子網路。
- 針對原則，請提供下列原則文件，以保護端點，以供 Lambda 服務主體使用`sts:AssumeRole`動作。

```
{
    "Statement": [
        {
            "Action": "sts:AssumeRole",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
}
```

- 確保啟用 DNS 名稱保持設置狀態。

4. 對於與 Amazon MSK 叢集相關聯的每個安全群組 (即叢*#*中)SecurityGroups，允許以下事項：

- 允許 9098 上的所有入站和出站 TCP 流量傳輸到所有*## SecurityGroups*，包括在內部。
- 允許 443 上的所有輸出 TCP 流量。

預設安全性群組規則允許其中一些流量，因此，如果叢集附加至單一安全性群組，而該群組具有預設規則，則不需要額外的規則。若要調整安全群組規則，請遵循 Amazon EC2 文件中將規則新增至安全群組中的程序。

使用下列資訊將規則新增至您的安全群組：

- 針對連接埠 9098 的每個輸入規則或輸出規則，請提供
  - 針對 Type (類型)，選擇 Custom TCP (自訂 TCP)。
  - 對於連接埠範圍，請提供 9098。
  - 針對來源，提供其中一個*##SecurityGroups*。
- 對於連接埠 443 的每個輸入規則，對於類型，選取 HTTPS。

為 Lambda 建立 IAM 角色，以便讀取您的 Amazon MSK 主題

確定 Lambda 要從您的 Amazon MSK 主題讀取的身份驗證要求，然後在政策中定義它們。建立角色 *Lambda AuthRole*，以授權 Lambda 使用這些權限。使用 `kafka-cluster` IAM 動作在您的 Amazon MSK 叢集上授權動作。然後，授權 Lambda 執行探索kafka並連接到 Amazon MSK 叢集所需的 Amazon MSK 和 Amazon EC2 動作，以及執行 CloudWatch 動作，以便 Lambda 可以記錄其完成的工作。

描述 Lambda 從 Amazon MSK 讀取的身份驗證要求

1. 撰寫 IAM 政策文件 (JSON 文件) *## AuthPolicy*，允許 Lambda 使用您的卡夫卡客戶群組讀取 Amazon MSK 叢集中的卡夫卡主題。Lambda 需要在讀取時設定卡夫卡消費群組。

   變更下列範本以符合您的先決條件：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "kafka-cluster:Connect",
                "kafka-cluster:DescribeGroup",
                "kafka-cluster:AlterGroup",
                "kafka-cluster:DescribeTopic",
                "kafka-cluster:ReadData",
                "kafka-cluster:DescribeClusterDynamicConfiguration"
            ],
            "Resource": [
                "arn:aws:kafka:region:account-id:cluster/mskClusterName/cluster-
uuid",
                "arn:aws:kafka:region:account-id:topic/mskClusterName/cluster-
uuid/mskTopicName",
                "arn:aws:kafka:region:account-id:group/mskClusterName/cluster-
uuid/mskGroupName"
            ]
        }
    ]
}
```

   欲了解更多信息，請諮詢the section called “IAM 角色型身分驗證”。撰寫您的政策時：

- 對於*##*和*## ID*，請提供託管 Amazon MSK 叢集的區域和帳戶識別碼。
- 對於 *msk ClusterName*，請提供您的 Amazon MSK 叢集的名稱。
- 對於*## UUID*，請在 ARN 中為您的 Amazon MSK 叢集提供 UUID。
- 對於 *msk TopicName*，請提供您的卡夫卡主題的名稱。
- 若為 *msk GroupName*，請提供您的卡夫卡消費群組名稱。

2. 識別 Lambda 探索和連接 Amazon MSK 叢集所需的 Amazon MSK、Amazon EC2 和 CloudWatch 許可，並記錄這些事件。

   受AWSLambdaMSKExecutionRole管理的原則會寬鬆定義必要的權限。請按照以下步驟使用它。

   在生產環境中，請根據最低權限原則評估AWSLambdaMSKExecutionRole以限制執行角色原則，然後為您的角色撰寫取代此受管理原則的原則。

如需 IAM 政策語言的詳細資訊，請參閱 IAM 文件。

現在您已經撰寫了政策文件，請建立 IAM 政策，以便將其附加到您的角色。您可以按照以下步驟使用控制台執行此操作。

從政策文件建立 IAM 政策

1. 登入 AWS Management Console 並開啟身分與存取權管理主控台，網址為 https://console.aws.amazon.com/iam/。
2. 在左側的導覽窗格中，選擇 Policies (政策)。
3. 選擇 Create policy (建立政策)。
4. 在政策編輯器中，選擇 JSON 選項。
5. 貼上*##AuthPolicy*。
6. 將許可新增至政策後，請選擇下一步。
7. 在檢視與建立頁面上，為您正在建立的政策輸入政策名稱與描述 (選用)。檢視此政策中定義的許可，來查看您的政策所授予的許可。
8. 選擇 Create policy (建立政策) 儲存您的新政策。

如需詳細資訊，請參閱 IAM 文件中的建立 IAM 政策。

現在您已擁有適當的 IAM 政策，請建立角色並將其附加至該角色。您可以按照以下步驟使用控制台執行此操作。

若要在 IAM 主控台中建立執行角色

1. 在 IAM 主控台中開啟 角色頁面。
2. 選擇 建立角色。
3. 在受信任的實體類型下，選擇 AWS 服務。
4. 在 使用案例 下，選擇 Lambda。
5. 選擇下一步。
6. 選取以下政策：
   - *## AuthPolicy*
   - AWSLambdaMSKExecutionRole
7. 選擇下一步。
8. 在角色名稱中，輸入 *lambda*，AuthRole然後選擇建立角色。

如需詳細資訊，請參閱 the section called “執行角色 (函數存取其他資源的權限)”。

建立 Lambda 函數以讀取您的 Amazon MSK 主題

建立設定為使用 IAM 角色的 Lambda 函數。您可以使用主控台建立 Lambda 函數。

若要使用驗證組態建立 Lambda 函數

1. 開啟 Lambda 主控台，然後從標頭中選取建立函數。
2. 選取從頭開始撰寫。
3. 對於函數名稱，請提供您選擇的適當名稱。
4. 對於 Runtime，請選擇的最新支援版本，Node.js以使用本教學課程中提供的程式碼。
5. 選擇 [變更預設執行角色]。
6. 選取 [使用現有角色]。
7. 對於現有角色，選取 *lambda AuthRole*。

在生產環境中，您通常需要為 Lambda 函數的執行角色新增其他政策，以便有意義地處理 Amazon MSK 事件。如需將政策新增至角色的詳細資訊，請參閱 IAM 文件中的新增或移除身分許可。

建立對應至 Lambda 函數的事件來源

您的 Amazon MSK 事件來源對應提供 Lambda 服務所需的資訊，以便在發生適當的 Amazon MSK 事件時叫用 Lambda。您可以使用主控台建立 Amazon MSK 映射。建立 Lambda 觸發器，然後自動設定事件來源對應。

若要建立 Lambda 觸發器 (以及事件來源對應)

1. 瀏覽至 Lambda 函數的概觀頁面。
2. 在「功能概覽」區段中，選擇左下角的「新增觸發器」。
3. 在 [選取來源] 下拉式清單中，選取 Amazon MSK。
4. 請勿設定驗證。
5. 對於 MSK 叢集，請選取叢集的名稱。
6. 對於 Batch 大小，輸入 1。此步驟使得此功能更容易測試，並且在生產中不是理想的價值。
7. 對於「主題名稱」，請提供您的 Kafka 主題名稱。
8. 對於消費者群組 ID，請提供您的卡夫卡用戶群組的 ID。

更新您的 Lambda 函數以讀取串流資料

Lambda 通過事件方法參數提供有關卡夫卡事件的信息。如需 Amazon MSK 事件的範例結構，請參閱the section called “ 範例事件”。瞭解如何解譯 Lambda 轉送的 Amazon MSK 事件之後，您可以變更 Lambda 函數程式碼以使用其提供的資訊。

將下列程式碼提供給您的 Lambda 函數，以記錄 Lambda Amazon MSK 事件的內容，以供測試之用：

Node.js

```
exports.handler = async (event) => {
    // Iterate through keys
    for (let key in event.records) {
      console.log('Key: ', key)
      // Iterate through records
      event.records[key].map((record) => {
        console.log('Record: ', record)
        // Decode base64
        const msg = Buffer.from(record.value, 'base64').toString()
        console.log('Message:', msg)
      })
    }
```

```
}
```

您可以使用主控台將函數程式碼提供給 Lambda。

若要更新 Lambda 函數程式碼

1. 瀏覽至 Lambda 函數的概觀頁面。
2. 選擇 程式碼 標籤。
3. 在程式碼來源 IDE 中輸入提供的程式碼。
4. 在 [程式碼原始碼] 導覽列中，選擇 [部署]。

測試您的 Lambda 函數以確認其已連接到您的 Amazon MSK 主題

您現在可以透過檢查 CloudWatch 事件記錄來確認事件來源是否叫用 Lambda。

驗證是否正在叫用 Lambda 函數

1. 使用您的卡夫卡管理主機使用 CLI 生成卡夫卡事件。`kafka-console-producer`如需詳細資訊，請參閱 Kafka 文件中的將一些事件寫入主題。針對先前步驟中定義的事件來源對應，傳送足夠的事件以填滿批次大小定義的批次，否則 Lambda 將等待更多資訊呼叫。
2. 如果您的函數執行，Lambda 會寫入發生的事情 CloudWatch。在主控台中，導覽至 Lambda 函數的詳細資料頁面。
3. 選取 Configuration (組態) 索引標籤。
4. 在側邊欄中，選取監控和作業工具。
5. 識別記CloudWatch 錄組態下的記錄群組。記錄群組的開頭應為`/aws/lambda`。選擇日誌群組的連結。
6. 在 CloudWatch 主控台中，檢查 Lambda 傳送至記錄串流的記錄事件記錄事件。識別是否存在包含來自 Kafka 事件之訊息的記錄事件，如下圖所示。如果有的話，您已經使用 Lambda 事件來源對應成功將 Lambda 函數連接到 Amazon MSK。

```
2020-08-06T15:06:18.861-04:00	START RequestId: 88ebae59-be0c-4e22-9db7-4154b437e43a Version: $LATEST
2020-08-06T15:06:18.866-04:00	2020-08-06T19:06:18.866Z 88ebae59-be0c-4e22-9db7-4154b437e43a INFO Key: mytopic-0
2020-08-06T15:06:18.866-04:00	2020-08-06T19:06:18.866Z	88ebae59-be0c-4e22-9db7-4154b437e43a	INFO	Record: {
  topic: 'mytopic',
  partition: 0,
  offset: 38,
  timestamp: 1596740777633,
  timestampType: 'CREATE_TIME',
  value: 'TWVzc2FnZSAjMQ=='
}
2020-08-06T15:06:18.866-04:00	2020-08-06T19:06:18.866Z 88ebae59-be0c-4e22-9db7-4154b437e43a INFO Message: Message #1
2020-08-06T15:06:18.890-04:00	END RequestId: 88ebae59-be0c-4e22-9db7-4154b437e43a
```

範例事件

Lambda 會在調用函數時，在事件參數中傳送訊息批次。事件酬載包含訊息陣列。陣列中的每個項目包含 Amazon MSK 主題和分割區識別符的詳細資訊，以及時間戳記和 base64 編碼的訊息。

```
{
   "eventSource":"aws:kafka",
   "eventSourceArn":"arn:aws:kafka:sa-east-1:123456789012:cluster/
vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2",
   "bootstrapServers":"b-2.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092,b-1.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092",
   "records":{
      "mytopic-0":[
         {
            "topic":"mytopic",
            "partition":0,
            "offset":15,
            "timestamp":1545084650987,
            "timestampType":"CREATE_TIME",
            "key":"abcDEFghiJKLmnoPQRstuVWXyz1234==",
            "value":"SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
            "headers":[
               {
                  "headerKey":[
                     104,
                     101,
                     97,
                     100,
                     101,
                     114,
                     86,
                     97,
```

```
                    108,
                    117,
                    101
                ]
            }
        ]
    }
  ]
 }
}
```

MSK 叢集身分驗證

Lambda 需要許可才能存取 Amazon MSK 叢集、擷取記錄和執行其他任務。Amazon MSK 支援數個選項，用於控制用戶端對 MSK 叢集的存取。

叢集存取選項

- 未驗證的存取
- SASL/SCRAM 身分驗證
- IAM 角色型身分驗證
- 交互 TLS 驗證
- 設定 mTLS 機密
- Lambda 選擇引導代理程式的方法

未驗證的存取

如果沒有用戶端透過網際網路存取叢集，您可以使用未驗證存取。

SASL/SCRAM 身分驗證

Amazon MSK 支援 Simple Authentication and Security Layer/Salted Challenge Response Authentication Mechanism (SASL/SCRAM) 身分驗證與 Transport Layer Security (TLS) 加密。若要讓 Lambda 連線到叢集，您可以將驗證認證 (使用者名稱和密碼) 儲存在 AWS Secrets Manager 密碼中。

如需使用 Secrets Manager 詳細資訊，請參閱《Amazon Managed Streaming for Apache Kafka 開發人員指南》中的AWS Secrets Manager的使用者名稱和密碼身分驗證。

Amazon MSK 不支援 SASL/PLAIN 身分驗證。

IAM 角色型身分驗證

您可以使用 IAM 來驗證連至 MSK 叢集之用戶端的身分。若您 MSK 叢集上的 IAM 身分驗證為作用中，且您未提供身分驗證密碼，則 Lambda 會自動預設使用 IAM 身分驗證。若要建立和部署使用者或角色型政策，請使用 IAM 主控台或 API。如需詳細資訊，請參閱《Amazon Managed Streaming for Apache Kafka 開發人員指南》中的「IAM 存取控制」。

若要允許 Lambda 連線至 MSK 叢集、讀取記錄，以及執行其他必要動作，請將下列許可新增至函數的執行角色。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "kafka-cluster:Connect",
                "kafka-cluster:DescribeGroup",
                "kafka-cluster:AlterGroup",
                "kafka-cluster:DescribeTopic",
                "kafka-cluster:ReadData",
                "kafka-cluster:DescribeClusterDynamicConfiguration"
            ],
            "Resource": [
                "arn:aws:kafka:region:account-id:cluster/cluster-name/cluster-uuid",
                "arn:aws:kafka:region:account-id:topic/cluster-name/cluster-uuid/topic-
name",
                "arn:aws:kafka:region:account-id:group/cluster-name/cluster-
uuid/consumer-group-id"
            ]
        }
    ]
}
```

您可以將這些許可的範圍設定為特定叢集、主題和群組。如需詳細資訊，請參閱《Amazon Managed Streaming for Apache Kafka 開發人員指南》中的 Amazon MSK Kafka 動作。

交互 TLS 驗證

相互 TLS (mTLS) 可提供用戶端與伺服器之間的雙向身分驗證。用戶端會將憑證傳送至伺服器以供伺服器驗證用戶端，而伺服器會將憑證傳送至用戶端以供用戶端驗證伺服器。

若為 Amazon MSK，Lambda 會以用戶端的身分運作。您可以設定用戶端憑證 (做為 Secrets Manager 中的機密) 來驗證 Lambda 與 MSK 叢集中的代理程式。用戶端憑證必須由伺服器信任存放區中的憑證授權機構簽署。MSK 叢集會傳送伺服器憑證到 Lambda 來驗證代理程式與 Lambda。伺服器憑證必須由 AWS 信任存放區中的憑證授權單位 (CA) 簽署。

如需如何產生用戶端憑證的說明，請參閱將 Amazon MSK 的相互 TLS 身分驗證作為事件來源。

Amazon MSK 不支援自行簽署的伺服器憑證，因為 Amazon MSK 中的所有代理程式都使用由 Amazon Trust Services CAs (根據預設，Lambda 信任此機構) 簽署的公有憑證。

如需有關適用於 Amazon MSK 的 mTLS 之詳細資訊，請參閱《Amazon Managed Streaming for Apache Kafka 開發人員指南》中的「相互 TLS 身分驗證」。

設定 mTLS 機密

CLIENT_CERTIFICATE_TLS_AUTH 機密必須有憑證欄位和私有金鑰欄位。若為加密的私有金鑰，機密需要私有金鑰密碼。憑證與私有金鑰均必須為 PEM 格式。

> **Note**
>
> Lambda 支援 PBES1 (但不支援 PBES2) 私有金鑰加密演算法。

憑證欄位必須包含憑證清單，以用戶端憑證開頭，隨後則是任何中繼憑證，並以根憑證結尾。每個憑證均必須以新的一行開始，結構如下：

```
-----BEGIN CERTIFICATE-----
        <certificate contents>
-----END CERTIFICATE-----
```

Secrets Manager 支援高達 65,536 個位元組的機密，此空間足以容納長憑證鏈。

私有金鑰必須為 PKCS #8 格式，結構如下：

```
-----BEGIN PRIVATE KEY-----
         <private key contents>
-----END PRIVATE KEY-----
```

對於已加密的私有金鑰，請使用下列結構：

```
-----BEGIN ENCRYPTED PRIVATE KEY-----
```

```
          <private key contents>
-----END ENCRYPTED PRIVATE KEY-----
```

下列範例顯示的是使用了已加密私有金鑰之 mTLS 身分驗證的機密內容。若為加密的私有金鑰，您可以在機密中包含私有金鑰密碼。

```
{
 "privateKeyPassword": "testpassword",
 "certificate": "-----BEGIN CERTIFICATE-----
MIIE5DCCAsygAwIBAgIRAPJdwaFaNRrytHBto0j5BA0wDQYJKoZIhvcNAQELBQAw
...
j0Lh4/+1HfgyE2KlmII36dg4IMzNjAFEBZiCRoPimO40s1cRqtFHXoal0QQbIlxk
cmUuiAii9R0=
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
MIIFgjCCA2qgAwIBAgIQdjNZd6uFf9hbNC5RdfmHrzANBgkqhkiG9w0BAQsFADBb
...
rQoiowbbk5wXCheYSANQIfTZ6weQTgiCHCCbuuMKNVS95FkXm0vqVD/YpXKwA/no
c8PH3PSoAaRwMMgOSA2ALJvbRz8mpg==
-----END CERTIFICATE-----",
 "privateKey": "-----BEGIN ENCRYPTED PRIVATE KEY-----
MIIFKzBVBgkqhkiG9w0BBQ0wSDAnBgkqhkiG9w0BBQwwGgQUiAFcK5hT/X7Kjmgp
...
QrSekqF+kWzmB6nAfSzgO9IaoAaytLvNgGTckWeUkWn/V0Ck+LdGUXzAC4RxZnoQ
zp2mwJn2NYB7AZ7+imp0azDZb+8YG2aUCiyqb6PnnA==
-----END ENCRYPTED PRIVATE KEY-----"
}
```

Lambda 選擇引導代理程式的方法

Lambda 會依據叢集上可用的身分驗證方法，以及您是否提供了身分驗證密碼，以此選擇引導代理程式。若您提供了 MTL 或 SASL/SCRAM 的密碼，則 Lambda 會自動選擇該身分驗證方法。若您未提供密碼，Lambda 會選取叢集上作用中的安全強度最高的身分驗證方法。以下是 Lambda 選擇代理程式的優先順序，身分驗證安全強度依次遞減：

- mTLS (已提供 mTLS 密碼）
- SASL/SCRAM (已提供 SASL /SCROM 密碼)
- SASL IAM (未提供任何密碼，且 IAM 身分驗證在作用中）
- 未驗證的 TLS (未提供任何密碼，且 IAM 身分驗證未在作用中)
- 純文字 (未提供任何密碼，且 IAM 身分驗證和未經身分驗證的 TLS 皆未在作用中)

> **Note**
>
> 若 Lambda 無法連線至最安全的代理程式類型，Lambda 便不會嘗試連線至其他 (安全強度較弱) 的代理程式類型。若您要讓 Lambda 選擇安全強度較弱較弱的代理程式類型，請停用叢集上所有安全強度較弱更高的身分驗證方法。

管理 API 存取和許可

除了存取 Amazon MSK 叢集之外，您的函數還需要執行各種 Amazon MSK API 動作的許可。您可以將這些許可新增到函數的執行角色。如果您的使用者需要存取任何 Amazon MSK API 動作，請將必要的許可新增至使用者或角色的身分政策。

您可以將下列各個許可手動新增到執行角色。或者，您可以將 AWS 受管政策附加AWSLambdaMSKExecutionRole到您的執行角色。`AWSLambdaMSKExecutionRole` 政策包含下列所有必要的 API 動作和 VPC 許可。

必要的 Lambda 函數執行角色許可

若要在 Amazon 日誌中建立日誌並將日誌存放在 CloudWatch 日誌群組中，Lambda 函數的執行角色必須具有下列許可：

- 記錄檔:CreateLog群組
- 記錄檔:CreateLog串流
- 記錄檔:PutLog事件

Lambda 函數的執行角色必須具有下列許可，Lambda 才能代您存取 Amazon MSK 叢集。

- 卡夫卡:DescribeCluster
- 卡夫卡:V2 DescribeCluster
- 卡夫卡:經紀人 GetBootstrap
- kafka：DescribeVpc連線：只有跨帳戶事件來源對應才需要。
- kafka：ListVpc連接：在執行角色中不需要，但對於正在創建跨帳戶事件源映射的 IAM 主體是必需的。

您只需要新增 `kafka:DescribeCluster` 或 `kafka:DescribeClusterV2` 即可。針對佈建的 MSK 叢集來說，任一許可皆有效。針對無伺服器 MSK 叢集，您必須使用 `kafka:DescribeClusterV2`。

> **Note**
>
> Lambda 最終計劃從相關聯的 `AWSLambdaMSKExecutionRole` 受管政策中移除 `kafka:DescribeCluster` 許可。若有使用此政策，您應遷移任何使用 `kafka:DescribeCluster` 的應用程式，並改用 `kafka:DescribeClusterV2`。

VPC 許可

如果只有某個 VPC 內的使用者可以存取 Amazon MSK 叢集，則您的 Lambda 函數必須具有存取 Amazon VPC 資源的許可。這些資源包括您的 VPC、子網路、安全群組和網路界面。若要存取這些資源，函數的執行角色必須具有下列權限。這些權限包含在AWSLambdaMSKExecutionRole AWS 受管理的策略中。

- ec2：CreateNetwork接口
- ec2：DescribeNetwork接口
- ec2：DescribeVpcs
- ec2：DeleteNetwork接口
- ec2：DescribeSubnets
- ec2：DescribeSecurity群組

選用 Lambda 函數許可

您的 Lambda 函數可能需要許可，才能：

- 若是使用 SASL/SCRAM 身分驗證，請存取您的 SCRAM 機密。
- 描述您 Secrets Manager 機密。
- 存取您的 AWS Key Management Service (AWS KMS) 客戶管理金鑰。
- 將失敗調用的記錄傳送到目的地。

Secrets Manager 和 AWS KMS 權限

視您為 Amazon MSK 代理程式設定的存取控制類型而定，您的 Lambda 函數可能需要存取您的 SCRAM 密碼的權限 (如果使用 SASL/SCRAM 驗證) 或秘密來解密客戶受管金鑰。 AWS KMS 若要連線至這些資源，函數的執行角色必須具有下列許可：

- 卡夫卡：秘密 ListScram

- 秘密經理:價值 GetSecret
- kms:Decrypt

將許可新增至您的執行角色

請依照下列步驟操作，使用 IAM 主控台將 AWS 受管政策新增AWSLambdaMSKExecutionRole至您的執行角色。

新增 AWS 受管理的策略

1. 開啟 IAM 主控台中的 Policies (政策) 頁面。
2. 在搜尋方塊中，輸入政策名稱 (AWSLambdaMSKExecutionRole)。
3. 從清單中選取政策，然後選擇 政策動作、附加。
4. 在 連接政策 頁面，從清單中選取您的執行角色，然後選擇 連接政策。

使用 IAM 政策授予使用者存取權

根據預設，使用者和角色沒有執行 Amazon MSK API 操作的許可。若要將存取權授予給組織或帳戶中的使用者，您可以新增或更新身分型政策。如需詳細資訊，請參閱《Amazon Managed Streaming for Apache Kafka 開發人員指南》中的 Amazon MSK 以身分為基礎的政策範例。

身分驗證和授權錯誤

如果缺少從 Amazon MSK 叢集取用資料所需的任何許可，Lambda 會在「LastProcessing結果」下的事件來源對應中顯示下列其中一個錯誤訊息。

錯誤訊息

- 叢集無法授權 Lambda
- SASL 身分驗證失敗
- 伺服器無法驗證 Lambda
- 提供的憑證或私有金鑰無效

叢集無法授權 Lambda

對於 SASL/SCRAM 或 mTLS，此錯誤表示提供的使用者不具備下列 Kafka 存取控制清單 (ACL) 的所有許可：

- DescribeConfigs 叢集
- 描述群組
- 讀取群組
- 描述主題
- 讀取主題

對於 IAM 存取控制，您的函數執行角色缺少存取群組或主題所需的一個或多個許可。檢閱 the section called “IAM 角色型身分驗證” 中的必要許可清單。

當您建立具有必要的 Kafka 叢集許可之 Kafka ACL 或 IAM 政策時，請將主題和群組指定為資源。主題名稱必須與事件來源映射中的主題相符。群組名稱必須與事件來源映射的 UUID 相符。

將必要的許可新增至執行角色後，可能需要數分鐘變更才會生效。

SASL 身分驗證失敗

對於 SASL/SCRAM，此錯誤表示所提供的使用者名稱和密碼是無效的。

對於 IAM 存取控制，執行角色缺少 MSK 叢集的 `kafka-cluster:Connect` 許可。將此許可新增至角色，並將叢集的 Amazon Resource Name (ARN) 指定為資源。

您可能會間歇性地看到此錯誤發生。TCP 連線數量超過 Amazon MSK 服務配額後，此叢集就會拒絕連線。Lambda 會關閉並重試，直到連線成功為止。Lambda 連接到叢集並輪詢記錄之後，上次處理結果會變更為 0K。

伺服器無法驗證 Lambda

此錯誤表示 Amazon MSK Kafka 代理程式無法使用 Lambda 來驗證。此狀況的發生原因如下：

- 您並未提供 mTLS 身分驗證的用戶端憑證。
- 您已提供用戶端憑證，但代理程式並未設定為使用 mTLS。
- 用戶端憑證不受代理程式信任。

提供的憑證或私有金鑰無效

此錯誤表示 Amazon MSK 取用者無法使用提供的憑證或私有金鑰。請確定憑證和金鑰使用 PEM 格式，且私有金鑰加密使用 PBES1 演算法。

網路組態

若要讓 Lambda 使用您的 Kafka 叢集做為事件來源，需要存取叢集所在的 Amazon VPC。我們建議您為 Lambda 部署 AWS PrivateLink VPC 人雲端端點，以存取您的 VPC。部署 Lambda 和 AWS Security Token Service (AWS STS) 的端點。如果代理程式使用身分驗證，也請為 Secrets Manager 部署 VPC 端點。如果您設定了失敗時的目的地，請同時為目的地服務部署 VPC 端點。

或者，確保與您 Kafka 叢集關聯的 VPC 每個公有子網路包含一個 NAT 閘道。如需詳細資訊，請參閱 the section called “VPC 功能的網際網路存取”。

若使用 VPC 端點，您還必須將它們設定為啟用私有 DNS 名稱。

當您為 MSK 叢集建立事件來源對應時，Lambda 會檢查叢集 VPC 的子網路和安全群組是否已存在彈性網路介面 (ENI)。如果 Lambda 找到現有的 ENI，它會嘗試重複使用它們。否則，Lambda 會建立新的 ENI 以連接至事件來源並叫用您的函數。

> **Note**
>
> Lambda 函數一律在 Lambda 服務擁有的 VPC 內執行。這些 VPC 由服務自動維護，客戶看不到。您也可以將您的功能連接到 Amazon VPC。在任何一種情況下，函數的 VPC 配置都不會影響事件源映射。只有事件來源 VPC 的組態才會決定 Lambda 連線至事件來源的方式。

您的 Amazon VPC 組態可透過 Amazon MSK API 來探索。您不需要在設定期間使用 create-event-source-mapping 命令來設定它。

如需AWS Lambda 有關設定網路的詳細資訊，請參閱運算部落格上的 VPC 中使用 Apache Kafka 叢集進行設定 AWS 。

VPC 安全群組規則

使用下列規則 (至少) 為包含叢集的 Amazon VPC 設定安全群組：

- 傳入規則 - 針對事件來源指定的安全群組，允許 Amazon MSK 代理程式連接埠上的所有流量 (9092 代表純文字、9094 代表 TLS、9096 代表 SASL、9098 代表 IAM)。
- 傳出規則：針對所有目的地，允許連接埠 443 上的所有流量。針對事件來源指定的安全群組，允許 Amazon MSK 代理程式連接埠上的所有流量 (9092 代表純文字、9094 代表 TLS、9096 代表 SASL、9098 代表 IAM)。
- 如果您使用的是 VPC 端點而不是 NAT 閘道，則與 VPC 端點相關聯的安全群組必須允許連接埠 443 上所有來自事件來源安全群組的輸入流量。

使用 VPC 端點

當您使用 VPC 端點時，會使用 ENI 透過這些端點路由呼叫函數的 API 呼叫。Lambda 服務主體需要呼叫`sts:AssumeRole`並`lambda:InvokeFunction`呼叫使用這些 ENI 的任何角色和函數。

根據預設，VPC 端點具有開放的 IAM 政策。最佳做法是將這些原則限制為只允許特定主參與者使用該端點執行所需的動作。為了確保您的事件來源對應能夠叫用 Lambda 函數，VPC 端點政策必須允許 Lambda 服務原則呼叫`sts:AssumeRole`和`lambda:InvokeFunction`。將 VPC 端點原則限制為僅允許來自組織內的 API 呼叫，可防止事件來源對應正常運作。

下列範例 VPC 端點原則示範如何授與 AWS STS 和 Lambda 端點所需的 Lambda 服務主體存取權。

Example VPC 端點策略- AWS STS 端點

```
{
    "Statement": [
        {
            "Action": "sts:AssumeRole",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
}
```

Example VPC 私人雲端端點政策

```
{
    "Statement": [
        {
            "Action": "lambda:InvokeFunction",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
```

```
        }
    ]
}
```

如果您的 Kafka 代理程式使用驗證，您也可以限制 Secrets Manager 端點的 VPC 端點原則。若要呼叫機 Secrets Manager API，Lambda 會使用您的函數角色，而不是使用 Lambda 服務主體。下列範例顯示 Secrets Manager 端點策略。

Example VPC 端點原則-Secrets Manager 端點

```
{
    "Statement": [
        {
            "Action": "secretsmanager:GetSecretValue",
            "Effect": "Allow",
            "Principal": {
                "AWS": [
                    "customer_function_execution_role_arn"
                ]
            },
            "Resource": "customer_secret_arn"
        }
    ]
}
```

如果您設定了故障時的目的地，Lambda 也會使用函數的角色來呼叫
或`s3:PutObject`使用 Lambda 管理的 ENI。`sns:Publish`

將 Amazon MSK 新增為事件來源

若要建立事件來源映射，使用 Lambda 主控台、AWS 開發套件或 AWS Command Line Interface (AWS CLI) 將 Amazon MSK 叢集新增為 Lambda 函數觸發條件。請注意，將 Amazon MSK 新增為觸發條件後，Lambda 會套用 Amazon MSK 叢集的 VPC 設定，而非 Lambda 函數的 VPC 設定。

本節說明如何使用 Lambda 主控台和 AWS CLI建立事件來源映射。

必要條件

- Amazon MSK 叢集和 Kafka 主題。如需詳細資訊，請參閱《Amazon Managed Streaming for Apache Kafka 開發人員指南》中的 Amazon MSK 入門。
- 具有存取 MSK 叢集使用之 AWS 資源之權限的執行角色。

可自訂的取用者群組 ID

將 Kafka 設為事件來源時，您可以指定取用者群組 ID。此取用者群組 ID 是您希望 Lambda 函數加入之 Kafka 取用者群組的現有識別符。您可以使用此功能將任何進行中的 Kafka 記錄處理設定從其他取用者無縫遷移至 Lambda。

如果您指定取用者群組 ID，且該取用者群組內還有其他作用中的輪詢者，則 Kafka 會將訊息分配給所有取用者。換句話說，Lambda 不會收到有關 Kafka 主題的所有訊息。如果您希望 Lambda 處理主題中的所有訊息，請關閉該取用者群組中的任何其他輪詢者。

此外，如果您指定取用者群組 ID，且 Kafka 找到具有相同 ID 的有效現有取用者群組，則 Lambda 會忽略用於事件來源映射的 `StartingPosition` 參數。相反的，Lambda 會根據取用者群組的承諾偏移量開始處理記錄。如果您指定取用者群組 ID，但 Kafka 找不到現有的取用者群組，則 Lambda 會使用指定的 `StartingPosition` 來設定事件來源。

您指定的取用者群組 ID 在所有 Kafka 事件來源中必須是唯一的。使用指定的取用者群組 ID 建立 Kafka 事件來源映射之後，您就無法更新此值。

新增 Amazon MSK 觸發條件 (主控台)

按照下列步驟將您的 Amazon MSK 叢集和 Kafka 主題新增為 Lambda 函數的觸發條件。

將 Amazon MSK 觸發條件新增至您的 Lambda 函數 (主控台)

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇 Lambda 函數的名稱。
3. 在 函式概觀 下，選擇 新增觸發條件 。
4. 在 Trigger configuration (觸發條件) 下，執行下列動作：
    a. 選擇 MSK 觸發條件類型。
    b. 對於 MSK cluster (MSK 叢集)，請選取您的叢集。
    c. 對於 批次大小，輸入單一批次中要擷取的訊息數量上限。
    d. 對於 Batch window (批次時段)，輸入 Lambda 調用函數之前收集記錄所花費的最長秒數。
    e. 對於 Topic name (主題名稱)，輸入 Kafka 主題名稱。
    f. (選用) 對於取用者群組 ID，輸入要加入的 Kafka 取用者群組 ID。
    g. (選用) 對於開始位置，請選擇最新以從最新記錄開始讀取串流、選擇水平修剪以從最早的可用記錄開始，或選擇在時間戳記為以指定開始讀取的時間戳記。
    h. (選用) 若為身分驗證，請選擇機密金鑰以供 MSK 叢集中的代理程式進行身分驗證。

i. 若要建立處於停用狀態的觸發條件以進行測試 (建議做法)，請取消勾選 啟用觸發條件 。或者，若要立即啟用觸發條件，請選取 啟用觸發條件。

5. 若要建立觸發條件，請選擇 新增 。

新增 Amazon MSK 觸發條件 (AWS CLI)

使用下列範例 AWS CLI 命令建立和檢視 Lambda 函數的 Amazon MSK 觸發器。

使用建立觸發程式 AWS CLI

Example — 為使用 IAM 身分驗證的叢集建立事件來源映射

下列範例會使用命create-event-source-mapping AWS CLI 令，將名為的 Lambda 函數對應my-kafka-function至名為的卡夫卡主題。AWSKafkaTopic主題的開始位置設定為 LATEST。當叢集使用 IAM 角色型身份驗證時，您不需要SourceAccess配置物件。範例：

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:cluster/my-cluster/fc2f5bdf-fd1b-45ad-85dd-15b4a5a6247e-2 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function
```

Example — 為使用 SASL/SCRAM 身分驗證的叢集建立事件來源映射

如果叢集使用 SASL/SCRAM 驗證，您必須包含指定的SourceAccess組態物件SASL_SCRAM_512_AUTH和秘 Secrets Manager ARN。

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:cluster/my-cluster/fc2f5bdf-fd1b-45ad-85dd-15b4a5a6247e-2 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function
  --source-access-configurations '[{"Type": "SASL_SCRAM_512_AUTH","URI":
 "arn:aws:secretsmanager:us-east-1:111122223333:secret:my-secret"}]'
```

Example — 為使用 mTLS 身分驗證的叢集建立事件來源映射

如果叢集使用 MTL 驗證，您必須包含指定的SourceAccess組態物件CLIENT_CERTIFICATE_TLS_AUTH和秘 Secrets Manager ARN。

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:cluster/my-cluster/fc2f5bdf-fd1b-45ad-85dd-15b4a5a6247e-2 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function
  --source-access-configurations '[{"Type": "CLIENT_CERTIFICATE_TLS_AUTH","URI": "arn:aws:secretsmanager:us-east-1:111122223333:secret:my-secret"}]'
```

如需詳細資訊，請參閱 CreateEventSourceMappingAPI 參考文件。

使用檢視狀態 AWS CLI

下列範例使用命get-event-source-mapping AWS CLI 令來描述您所建立之事件來源對應的狀態。

```
aws lambda get-event-source-mapping \
  --uuid 6d9bce8e-836b-442c-8070-74e77903c815
```

建立跨帳戶事件來源映射

您可以使用多 VPC 私有連線，將 Lambda 函數連接到不同 AWS 帳戶中的佈建 MSK 叢集。多 VPC 連線使用 AWS PrivateLink，可保留網路內的 AWS 所有流量。

> **Note**
>
> 您無法為無伺服器 MSK 叢集建立跨帳戶事件來源映射。

若要建立跨帳戶事件來源映射，您必須先為 MSK 叢集設定多 VPC 連線。建立事件來源映射時，請使用受管理的 VPC 連線 ARN 而非叢集 ARN，如下列範例所示。CreateEventSourceMapping作業也會根據 MSK 叢集使用的驗證類型而有所不同。

Example — 為使用 IAM 身分驗證的叢集建立跨帳戶的事件來源映射

當叢集使用 IAM 角色型身份驗證時，您不需要SourceAccess配置物件。範例：

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:vpc-connection/444455556666/my-cluster-name/51jn98b4-0a61-46cc-b0a6-61g9a3d797d5-7 \
```

```
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function
```

Example — 為使用 SASL/SCRAM 身分驗證的叢集建立跨帳戶的事件來源映射

如果叢集使用 SASL/SCRAM 驗證，您必須包含指定的SourceAccess組態物件SASL_SCRAM_512_AUTH和秘 Secrets Manager ARN。

透過 SASL/SCRAM 身分驗證，有兩種方式可將機密用於跨帳戶 Amazon MSK 事件來源映射：

- 在 Lambda 函數帳戶中建立機密，並將其與叢集機密同步。建立一個輪換以使兩個機密保持同步。此選項允許您從函數帳戶控制機密。
- 使用機密必須與 Amazon MSK 叢集相關聯。此機密必須允許 Lambda 函數帳戶的跨帳戶存取權。如需詳細資訊，請參閱不同帳戶中使用者 AWS Secrets Manager 密碼的權限。

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:vpc-connection/444455556666/
my-cluster-name/51jn98b4-0a61-46cc-b0a6-61g9a3d797d5-7 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function \
  --source-access-configurations '[{"Type": "SASL_SCRAM_512_AUTH","URI":
 "arn:aws:secretsmanager:us-east-1:444455556666:secret:my-secret"}]'
```

Example — 為使用 mTLS 身分驗證的叢集建立跨帳戶的事件來源映射

如果叢集使用 MTL 驗證，您必須包含指定的SourceAccess組態物件CLIENT_CERTIFICATE_TLS_AUTH和秘 Secrets Manager ARN。機密可以儲存在叢集帳戶或 Lambda 函數帳戶中。

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:vpc-connection/444455556666/
my-cluster-name/51jn98b4-0a61-46cc-b0a6-61g9a3d797d5-7 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function \
  --source-access-configurations '[{"Type": "CLIENT_CERTIFICATE_TLS_AUTH","URI":
 "arn:aws:secretsmanager:us-east-1:444455556666:secret:my-secret"}]'
```

失敗時的目的地

若要保留失敗的事件來源映射調用記錄，請將目標地新增到函數的事件來源映射中。每個傳送至目的地的記錄都是 JSON 文件，其中包含有關失敗叫用的中繼資料。您可以將任何 Amazon SNS 主題、Amazon SQS 佇列或 S3 儲存貯體設定為目的地。您的執行角色必須具有目標的權限：

- 對於 SQS 目的地：sq s：SendMessage
- 對於 SNS 目的地：SNS: 發佈
- 對於 S3 儲存貯體目的地：s3：PutObject和 s3：ListBuckets

此外，如果您在目的地上設定 KMS 金鑰，則視目的地類型而定，Lambda 需要下列許可：

- 如果您已針對 S3 目的地使用自己的 KMS 金鑰啟用加密，則需要 kms：金GenerateData鑰。如果 KMS 金鑰和 S3 儲存貯體目的地與 Lambda 函數和執行角色位於不同的帳戶中，請將 KMS 金鑰設定為信任執行角色以允許 KMS：GenerateDataKey。
- 如果您已針對 SQS 目的地使用自己的 KMS 金鑰啟用加密，則需要 KMS: 解密和公里：GenerateData金鑰。如果 KMS 金鑰和 SQS 佇列目的地與 Lambda 函數和執行角色位於不同的帳戶中，請將 KMS 金鑰設定為信任執行角色，以允許 KMS：解密、公里：金GenerateData鑰、公里：和公里：DescribeKey。ReEncrypt
- 如果您已針對 SNS 目的地使用自己的 KMS 金鑰啟用加密，則需要 KMS: 解密和公里：金GenerateData鑰。如果 KMS 金鑰和 SNS 主題目的地與 Lambda 函數和執行角色位於不同的帳戶中，請將 KMS 金鑰設定為信任執行角色，以允許 KMS：解密、公里：金GenerateData鑰、公里：DescribeKey和公里：。ReEncrypt

若要使用主控台設定失敗時的目的地，請依照下列步驟執行：

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 在 函數概觀 下，選擇 新增目的地。
4. 針對來源，請選擇事件來源映射調用。
5. 對於事件來源映射，請選擇針對此函數設定的事件來源。
6. 對於條件，選取失敗時。對於事件來源映射調用，這是唯一可接受的條件。
7. 對於目標類型，請選擇 Lambda 將調用記錄傳送至的目標類型。
8. 對於 目的地，請選擇一個資源。
9. 選擇 儲存 。

您也可以使用設定失敗時的目的 AWS CLI地。例如，下列建立事件來源對映指令會將具有失敗時之 SQS 目的地的事件來源對應新增至：MyFunction

```
aws lambda create-event-source-mapping \
--function-name "MyFunction" \
--event-source-arn arn:aws:kafka:us-east-1:123456789012:cluster/
vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2 \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sqs:us-
east-1:123456789012:dest-queue"}}'
```

下列更新事件來源映射命令會將 S3 故障時的目標新增至與輸入相關聯的事件來源：uuid

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": "arn:aws:s3:::dest-bucket"}}'
```

若要移除目的地，請提供空白字串作為 destination-config 參數的引數：

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": ""}}'
```

SNS 和 SQS 範例呼叫記錄

下列範例顯示 Lambda 針對失敗的 Kafka 事件來源調用而傳送至 SNS 主題或 SQS 佇列目的地。recordsInfo 之下的每個索引鍵都包含 Kafka 主題和分割區，以連字號分隔。例如，對於金鑰 "Topic-0"，Topic 是 Kafka 主題，並且 0 是分區。對於每個主題和分區，您可以使用偏移和時間戳記資料來查找原始調用記錄。

```
{
    "requestContext": {
        "requestId": "316aa6d0-8154-xmpl-9af7-85d5f4a6bc81",
        "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted" | "MaximumPayloadSizeExceeded",
        "approximateInvokeCount": 1
    },
    "responseContext": { // null if record is MaximumPayloadSizeExceeded
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
```

```
    "version": "1.0",
    "timestamp": "2019-11-14T00:38:06.021Z",
    "KafkaBatchInfo": {
        "batchSize": 500,
        "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/
vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2",
        "bootstrapServers": "...",
        "payloadSize": 2039086, // In bytes
        "recordsInfo": {
            "Topic-0": {
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            },
            "Topic-1": {
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            }
        }
    }
}
```

S3 目的地範例叫用記錄

對於 S3 的目的地，Lambda 會將整個調用記錄與中繼資料一起傳送至目的地。下列範例顯示 Lambda 針對失敗的 Kafka 事件來源調用而傳送至 S3 儲存貯體目的地。除了上一個 SQS 和 SNS 目的地範例中的所有欄位之外，此 payload 欄位還包含原始調用記錄做為逸出 JSON 字串。

```
{
    "requestContext": {
        "requestId": "316aa6d0-8154-xmpl-9af7-85d5f4a6bc81",
        "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted" | "MaximumPayloadSizeExceeded",
        "approximateInvokeCount": 1
    },
    "responseContext": { // null if record is MaximumPayloadSizeExceeded
```

```
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:38:06.021Z",
    "KafkaBatchInfo": {
        "batchSize": 500,
        "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/
vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2",
        "bootstrapServers": "...",
        "payloadSize": 2039086, // In bytes
        "recordsInfo": {
            "Topic-0": {
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            },
            "Topic-1": {
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            }
        }
    },
    "payload": "<Whole Event>" // Only available in S3
}
```

> **Tip**
>
> 建議您在目的地儲存貯體上啟用 S3 版本控制。

Amazon MSK 事件來源的自動調整規模

當您最初建立 Amazon MSK 事件來源 時，Lambda 會分配一個取用者來處理 Kafka 主題的所有分割區。每個取用者都有多個並行運行的處理器以處理增加的工作負載。此外，Lambda 會根據工作負載自動增加或減少取用者數量。為了保留每個分割區中的訊息順序，主題中每個分割區的取用者數上限是一個取用者。

每 1 分鐘，Lambda 會評估主題中所有分割區的取用者偏移延遲。如果延遲太高，則表示分割區接收訊息的速度比 Lambda 處理訊息的速度更快。如有必要，Lambda 會新增或移除主題取用者。新增或刪除取用者的擴展過程，將在三分鐘的評估期間內完成。

如果您的目標 Lambda 函數遭限流，Lambda 會減少取用者的數量。此動作可透過減少取用者可擷取和傳送至函數的訊息數量，減少函數的工作負載。

要監控 Kafka 主題的輸送量，當您的函數處理記錄時，檢視 Lambda 發出的偏移延遲指標。

若要檢查並行發生的函數調用次數，您也可以監控函數的並行指標。

輪詢和串流開始位置

請注意，建立和更新事件來源映射期間的串流輪詢最終會一致。

- 在建立事件來源映射期間，從串流開始輪詢事件可能需要幾分鐘時間。
- 在更新事件來源映射期間，從串流停止並重新開始輪詢事件可能需要幾分鐘時間。

這種行為表示如果您指定 `LATEST` 當作串流的開始位置，事件來源映射可能會在建立或更新期間遺漏事件。若要確保沒有遺漏任何事件，請將串流開始位置指定為 `TRIM_HORIZON` 或 `AT_TIMESTAMP`。

Amazon CloudWatch 指標

當您的函數處理記錄時，Lambda 會發出 `OffsetLag` 指標。此指標的值是寫入 Kafka 事件來源主題的最後一筆記錄與函數取用者群組處理的最後一筆記錄之間的偏移量的差值。您可以使用 `OffsetLag` 來預估新增記錄時與取用者群組處理記錄時之間的延遲。

`OffsetLag` 的增加趨勢可能表示函數取用者群組中的輪詢者問題。如需詳細資訊，請參閱 使用 Lambda 函數指標。

Amazon MSK 組態參數

所有 Lambda 事件來源類型都共用相同CreateEventSourceMapping的UpdateEventSourceMappingAPI 作業。但是，只有一些參數適用於 Amazon MSK。

適用於 Amazon MSK 的事件來源參數

參數	必要	預設	備註
AmazonManagedKafkaEventSourceConfig	N	包含預設為唯一值的 ConsumerGroupId 欄位。	只能在建立時進行設定
BatchSize	否	100	上限：10,000
已啟用	N	已啟用	
EventSource阿恩	Y		只能在建立時進行設定
FunctionName	Y		
FilterCriteria	N		Lambda 事件篩選
MaximumBatchingWindowIn秒	N	500 毫秒	批次處理行為
SourceAccess配置	N	沒有憑證	您事件來源的 SASL/SCRAM 或 CLIENT_CERTIFICATE_TLS_AUTH (MutualTLS) 身分驗證憑證。
StartingPosition	Y		AT_TIMESTAMP、TRIM_HORIZON 或 LATEST 只能在建立時進行設定
StartingPosition時間戳	N		如果設定為 [時間 StartingPosition 戳記]，則為必要

參數	必要	預設	備註
主題	Y		Kafka 主題名稱 只能在建立時進行設定

搭配使用 Lambda 與自我管理 Apache Kafka

> **Note**
>
> 如果您想要將資料傳送到 Lambda 函數以外的目標，或在傳送資料之前豐富資料，請參閱 Amazon EventBridge 管道。

Lambda 支援 Apache Kafka 作為事件來源。Apache Kafka 是開源事件串流平台，可支援資料管道和串流分析等工作負載。

您可以使用 AWS 受管卡夫卡服務 Amazon Managed Streaming for Apache Kafka（Amazon MSK），或自我管理的卡夫卡集群。如需搭配使用 Lambda 與 Amazon MSK 的詳細資訊，請參與 搭配使用 Lambda 與 Amazon MSK。

本主題說明如何搭配使用 Lambda 與自我管理 Kafka 叢集。在 AWS 術語中，自我管理的叢集包含非 AWS 託管的 Kafka 叢集。例如，您可以使用雲端供應商 (例如 Confluent Cloud) 託管您的 Kafka 叢集。

Apache Kafka 作為事件來源時，其運作方式類似於使用 Amazon Simple Queue Service (Amazon SQS) 或 Amazon Kinesis。Lambda 會在內部輪詢事件來源中的新訊息，然後同步調用目標 Lambda 函數。Lambda 會批次讀取訊息，並將這些訊息作為事件酬載提供給函數。批次大小上限可設定。(預設值為 100 則訊息。)

> **Warning**
>
> Lambda 事件來源對應至少處理每個事件一次，並且可能會重複處理記錄。為了避免與重複事件相關的潛在問題，我們強烈建議您將函數代碼設為冪等。若要深入了解，請參閱 AWS 知識中心如何讓 Lambda 函數具有冪等性。

對於基於 Kafka 的事件來源，Lambda 支援處理控制參數，例如批次間隔和批次大小。如需詳細資訊，請參閱 批次處理行為。

如需如何使用自我管理的 Kafka 做為事件來源的範例，請參閱在計算部落格上使用自我託管的 Apache Kafka 做為事件來源。 AWS Lambda AWS

主題

- 範例事件
- Kafka 叢集身分驗證
- 管理 API 存取和許可
- 身分驗證和授權錯誤
- 網路組態
- 將 Kafka 叢集新增為事件來源
- 失敗時的目的地
- 使用 Kafka 叢集作為事件來源
- 輪詢和串流開始位置
- Kafka 事件來源的自動調整規模
- 事件來源映射錯誤
- Amazon CloudWatch 指標
- 自我管理的 Apache Kafka 組態參數

範例事件

Lambda 會在調用 Lambda 函數時，在事件參數中傳送訊息批次。事件酬載包含訊息陣列。陣列中的每個項目包含 Kafka 主題和 Kafka 分割區識別符的詳細資訊，以及時間戳記和 base64 編碼的訊息。

```
{
   "eventSource": "SelfManagedKafka",
   "bootstrapServers":"b-2.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092,b-1.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092",
   "records":{
      "mytopic-0":[
         {
            "topic":"mytopic",
            "partition":0,
            "offset":15,
```

```
            "timestamp":1545084650987,
            "timestampType":"CREATE_TIME",
            "key":"abcDEFghiJKLmnoPQRstuVWXyz1234==",
            "value":"SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
            "headers":[
               {
                  "headerKey":[
                     104,
                     101,
                     97,
                     100,
                     101,
                     114,
                     86,
                     97,
                     108,
                     117,
                     101
                  ]
               }
            ]
         }
      ]
   }
}
```

Kafka 叢集身分驗證

Lambda 支持多種方法來進行您自我管理 Apache Kafka 叢集的身分驗證。請確保您已設定 Kafka 叢集使用其中一種支援的身分驗證方法。如需有關 Kafka 安全性的詳細資訊，請參閱 Kafka 文件的「安全性」一節。

VPC 存取

如果只有您 VPC 內的 Kafka 使用者會存取您的 Kafka 代理程式，就必須為 Amazon Virtual Private Cloud (Amazon VPC) 存取設定 Kafka 事件來源。

SASL/SCRAM 身分驗證

Lambda 支援 Simple Authentication and Security Layer/Salted Challenge Response Authentication Mechanism (SASL/SCRAM) 身分驗證與 Transport Layer Security (TLS) 加密 (`SASL_SSL`)。Lambda 會傳送加密的憑證，以便向叢集進行身分驗證。Lambda 不支援含純文字的 SASL/SCRAM (`SASL_PLAINTEXT`)。如需 SASL/SCRAM 身分驗證的詳細資訊，請參閱 RFC 5802。

Lambda 也支援 SASL/PLAIN 身分驗證。由於這套機制使用純文字憑證，與伺服器的連線必須使用 TLS 加密，以確保憑證受到保護。

若使用 SASL 身分驗證，您需將登入憑證儲存為 AWS Secrets Manager中的秘密。如需有關使用 Secrets Manager 的詳細資訊，請參閱《AWS Secrets Manager 使用者指南》中的「教學課程：建立和擷取機密」。

> **Important**
>
> 若要使用 Secrets Manager 進行驗證，密碼必須儲存在與 Lambda 函數相同的 AWS 區域中。

交互 TLS 驗證

相互 TLS (mTLS) 可提供用戶端與伺服器之間的雙向身分驗證。用戶端會將憑證傳送至伺服器以供伺服器驗證用戶端，而伺服器會將憑證傳送至用戶端以供用戶端驗證伺服器。

在自我管理 Apache Kafka 中，Lambda 會以用戶端的身分運作。您可以設定用戶端憑證 (做為 Secrets Manager 中的機密) 來驗證 Lambda 與 Kafka 代理程式。用戶端憑證必須由伺服器信任存放區中的憑證授權機構簽署。

Kafka 叢集會傳送伺服器憑證到 Lambda 來驗證 Kafka 代理程式與 Lambda。伺服器憑證可以是公有憑證授權機構憑證或私有憑證授權機構/自行簽署的憑證。公有憑證授權機構憑證必須由 Lambda 信任存放區中的憑證授權機構 (CA) 簽署。若為私有憑證授權機構/自行簽署的憑證，您可以設定伺服器根憑證授權機構憑證 (做為 Secrets Manager 中的機密)。Lambda 使用根憑證來驗證 Kafka 代理程式。

如需有關 mTLS 的詳細資訊，請參閱將 Amazon MSK 的相互 TLS 身分驗證作為事件來源。

設定用戶端憑證機密

CLIENT_CERTIFICATE_TLS_AUTH 機密必須有憑證欄位和私有金鑰欄位。若為加密的私有金鑰，機密需要私有金鑰密碼。憑證與私有金鑰均必須為 PEM 格式。

> **Note**
>
> Lambda 支援 PBES1 (但不支援 PBES2) 私有金鑰加密演算法。

憑證欄位必須包含憑證清單，以用戶端憑證開頭，隨後則是任何中繼憑證，並以根憑證結尾。每個憑證均必須以新的一行開始，結構如下：

```
-----BEGIN CERTIFICATE-----
        <certificate contents>
-----END CERTIFICATE-----
```

Secrets Manager 支援高達 65,536 個位元組的機密，此空間足以容納長憑證鏈。

私有金鑰必須為 PKCS #8 格式，結構如下：

```
-----BEGIN PRIVATE KEY-----
         <private key contents>
-----END PRIVATE KEY-----
```

對於已加密的私有金鑰，請使用下列結構：

```
-----BEGIN ENCRYPTED PRIVATE KEY-----
          <private key contents>
-----END ENCRYPTED PRIVATE KEY-----
```

下列範例顯示的是使用了已加密私有金鑰之 mTLS 身分驗證的機密內容。若為加密的私有金鑰，請在機密中包含私有金鑰密碼。

```
{"privateKeyPassword":"testpassword",
"certificate":"-----BEGIN CERTIFICATE-----
MIIE5DCCAsygAwIBAgIRAPJdwaFaNRrytHBto0j5BA0wDQYJKoZIhvcNAQELBQAw
...
j0Lh4/+1HfgyE2KlmII36dg4IMzNjAFEBZiCRoPimO40s1cRqtFHXoal0QQbIlxk
cmUuiAii9R0=
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
MIIFgjCCA2qgAwIBAgIQdjNZd6uFf9hbNC5RdfmHrzANBgkqhkiG9w0BAQsFADBb
...
rQoiowbbk5wXCheYSANQIfTZ6weQTgiCHCCbuuMKNVS95FkXm0vqVD/YpXKwA/no
c8PH3PSoAaRwMMgOSA2ALJvbRz8mpg==
-----END CERTIFICATE-----",
"privateKey":"-----BEGIN ENCRYPTED PRIVATE KEY-----
MIIFKzBVBgkqhkiG9w0BBQ0wSDAnBgkqhkiG9w0BBQwwGgQUiAFcK5hT/X7Kjmgp
...
QrSekqF+kWzmB6nAfSzgO9IaoAaytLvNgGTckWeUkWn/V0Ck+LdGUXzAC4RxZnoQ
zp2mwJn2NYB7AZ7+imp0azDZb+8YG2aUCiyqb6PnnA==
-----END ENCRYPTED PRIVATE KEY-----"
}
```

設定伺服器根憑證授權機構憑證機密

如果您的 Kafka 代理程式使用 TLS 加密與私有憑證授權機構簽署的憑證，則建立此機密。您可以使用 TLS 加密以供 VPC、SASL/SCRAM、SASL/PLAIN 或 mTLS 身分驗證之用。

伺服器根憑證授權機構憑證機密必須有包含 Kafka 代理程式根憑證授權機構憑證 (格式為 PEM) 的欄位。下列範例說明機密的結構。

```
{"certificate":"-----BEGIN CERTIFICATE-----
MIID7zCCAtegAwIBAgIBADANBgkqhkiG9w0BAQsFADCBmDELMAkGA1UEBhMCVVMx
EDAOBgNVBAgTB0FyaXpvbmExEzARBgNVBAcTClNjb3R0c2RhbGUxJTAjBgNVBAoT
HFN0YXJmaWVsZCBUZWNobm9sb2dpZXMsIEluYy4xOzA5BgNVBAMTMlN0YXJmaWVs
ZCBTZXJ2aWNlcyBSb290IENlcnRpZmljYXRlIEF1dG...
-----END CERTIFICATE-----"
}
```

管理 API 存取和許可

除了存取自我管理的 Kafka 叢集之外，您的 Lambda 函數還需要執行各種 API 動作的許可。您可以將這些許可新增到函數的執行角色。如果您的使用者需要存取任何 API 動作，請將必要的許可新增至 AWS Identity and Access Management (IAM) 使用者或角色的身分政策。

必要的 Lambda 函數許可

若要在 Amazon 日誌中建立日誌並將日誌存放在 CloudWatch 日誌群組中，Lambda 函數的執行角色必須具有下列許可：

- 記錄檔:CreateLog群組
- 記錄檔:CreateLog串流
- 記錄檔:PutLog事件

選用 Lambda 函數許可

您的 Lambda 函數可能需要許可，才能：

- 描述您 Secrets Manager 機密。
- 存取您的 AWS Key Management Service (AWS KMS) 客戶管理金鑰。
- 存取 Amazon VPC。
- 將失敗調用的記錄傳送到目的地。

Secrets Manager 和 AWS KMS 權限

視您為 Kafka 代 Secrets Manager 程式設定的存取控制類型而定，Lambda 函數可能需要存取您的秘密密碼或解密 AWS KMS 客戶受管金鑰的權限。若要連線至這些資源，函數的執行角色必須具有下列許可：

- 秘密經理:價值 GetSecret
- kms:Decrypt

VPC 許可

如果只有某個 VPC 內的使用者可以存取自我管理 Apache Kafka 叢集，則您的 Lambda 函數必須具有存取 Amazon VPC 資源的許可。這些資源包括您的 VPC、子網路、安全群組和網路界面。若要連線至這些資源，函數的執行角色必須具有下列許可：

- ec2：CreateNetwork接口
- ec2：DescribeNetwork接口
- ec2：DescribeVpcs
- ec2：DeleteNetwork接口
- ec2：DescribeSubnets
- ec2：DescribeSecurity群組

將許可新增至您的執行角色

若要存取自我管理的 Apache Kafka 叢集使用的其他 AWS 服務，Lambda 會使用您在 Lambda 函數執行角色中定義的許可政策。

根據預設，Lambda 不允許針對自我管理 Apache Kafka 叢集執行必要或選用的動作。您必須在 IAM 信任政策中建立並定義這些動作，然後將政策附加至您的執行角色。此範例會示範如何建立允許 Lambda 存取 Amazon VPC 資源的政策。

```
{
        "Version":"2012-10-17",
        "Statement":[
           {
              "Effect":"Allow",
              "Action":[
                 "ec2:CreateNetworkInterface",
                 "ec2:DescribeNetworkInterfaces",
```

```
                "ec2:DescribeVpcs",
                "ec2:DeleteNetworkInterface",
                "ec2:DescribeSubnets",
                "ec2:DescribeSecurityGroups"
            ],
            "Resource":"*"
        }
    ]
}
```

如需有關在 IAM 主控台建立 JSON 政策文件的詳細資訊，請參閱 IAM 使用者指南中的在 JSON 標籤上建立政策。

使用 IAM 政策授予使用者存取權

根據預設，使用者和角色沒有執行事件來源 API 操作的許可。若要將存取權授予組織或帳戶中的使用者，您可能需要建立或更新身分型政策。如需詳細資訊，請參閱 IAM 使用者指南中的政策控制 AWS 資源的存取。

身分驗證和授權錯誤

如果缺少從 Kafka 叢集取用資料所需的任何權限，Lambda 會在「LastProcessing結果」下的事件來源對應中顯示下列其中一個錯誤訊息。

錯誤訊息

- 叢集無法授權 Lambda
- SASL 身分驗證失敗
- 伺服器無法驗證 Lambda
- Lambda 無法驗證伺服器
- 提供的憑證或私有金鑰無效

叢集無法授權 Lambda

對於 SASL/SCRAM 或 mTLS，此錯誤表示提供的使用者不具備下列 Kafka 存取控制清單 (ACL) 的所有許可：

- DescribeConfigs 叢集
- 描述群組
- 讀取群組

- 描述主題
- 讀取主題

當您建立具有必要的 `kafka-cluster` 許可之 Kafka ACL 時，請將主題和群組指定為資源。主題名稱必須與事件來源映射中的主題相符。群組名稱必須與事件來源映射的 UUID 相符。

將必要的許可新增至執行角色後，可能需要數分鐘變更才會生效。

SASL 身分驗證失敗

若使用 SASL/SCRAM 或 SASL/PLAIN，此錯誤表示所提供的登入憑證無效。

伺服器無法驗證 Lambda

此錯誤表示 Kafka 代理程式無法驗證 Lambda。此狀況的發生原因如下：

- 您並未提供 mTLS 身分驗證的用戶端憑證。
- 您已提供用戶端憑證，但並未將 Kafka 代理程式設定為使用 mTLS 身分驗證。
- 用戶端憑證不受 Kafka 代理程式信任。

Lambda 無法驗證伺服器

此錯誤表示 Lambda 無法驗證 Kafka 代理程式。此狀況的發生原因如下：

- Kafka 代理程式會使用自行簽署的憑證或私有憑證授權機構，但不會提供伺服器根憑證授權機構憑證。
- 伺服器根憑證授權機構憑證與簽署代理程式憑證的根憑證授權機構不相符。
- 主機名稱驗證失敗，因為代理程式的憑證不包含代理程式做為主體替代名稱的 DNS 名稱或 IP 地址。

提供的憑證或私有金鑰無效

此錯誤表示 Kafka 取用者無法使用提供的憑證或私有金鑰。請確定憑證和金鑰使用 PEM 格式，且私有金鑰加密使用 PBES1 演算法。

網路組態

若要讓 Lambda 使用您的 Kafka 叢集做為事件來源，需要存取叢集所在的 Amazon VPC。我們建議您為 Lambda 部署 AWS PrivateLink VPC 人雲端端點，以存取您的 VPC。部署 Lambda 和 AWS

Security Token Service (AWS STS) 的端點。如果代理程式使用身分驗證，也請為 Secrets Manager 部署 VPC 端點。如果您設定了失敗時的目的地，請同時為目的地服務部署 VPC 端點。

或者，確保與您 Kafka 叢集關聯的 VPC 每個公有子網路包含一個 NAT 閘道。如需詳細資訊，請參閱 the section called “VPC 功能的網際網路存取”。

若使用 VPC 端點，您還必須將它們設定為啟用私有 DNS 名稱。

當您為自我管理的 Apache Kafka 叢集建立事件來源對應時，Lambda 會檢查叢集 VPC 的子網路和安全性群組是否已存在彈性網路介面 (ENI)。如果 Lambda 找到現有的 ENI，它會嘗試重複使用它們。否則，Lambda 會建立新的 ENI 以連接至事件來源並叫用您的函數。

> **Note**
>
> Lambda 函數一律在 Lambda 服務擁有的 VPC 內執行。這些 VPC 由服務自動維護，客戶看不到。您也可以將您的功能連接到 Amazon VPC。在任何一種情況下，函數的 VPC 配置都不會影響事件源映射。只有事件來源 VPC 的組態才會決定 Lambda 連線至事件來源的方式。

如需AWS Lambda 有關設定網路的詳細資訊，請參閱運算部落格上的 VPC 中使用 Apache Kafka 叢集進行設定 AWS 。

VPC 安全群組規則

使用下列規則 (至少) 為包含叢集的 Amazon VPC 設定安全群組：

- 傳入規則 - 允許為事件來源指定之安全群組的所有 Kafka 代理程式連接埠上的所有流量。Kafka 預設使用連接埠 9092。
- 傳出規則：針對所有目的地，允許連接埠 443 上的所有流量。允許為事件來源指定之安全群組的所有 Kafka 代理程式連接埠上的所有流量。Kafka 預設使用連接埠 9092。
- 如果您使用的是 VPC 端點而不是 NAT 閘道，則與 VPC 端點相關聯的安全群組必須允許連接埠 443 上所有來自事件來源安全群組的輸入流量。

使用 VPC 端點

當您使用 VPC 端點時，會使用 ENI 透過這些端點路由呼叫函數的 API 呼叫。Lambda 服務主體需要呼叫`sts:AssumeRole`並`lambda:InvokeFunction`呼叫使用這些 ENI 的任何角色和函數。

根據預設，VPC 端點具有開放的 IAM 政策。最佳做法是將這些原則限制為只允許特定主參與者使用該端點執行所需的動作。為了確保您的事件來源對應能夠叫用 Lambda 函數，VPC 端點政策必須允許

Lambda 服務原則呼叫`sts:AssumeRole`和`lambda:InvokeFunction`。將 VPC 端點原則限制為僅允許來自組織內的 API 呼叫，可防止事件來源對應正常運作。

下列範例 VPC 端點原則示範如何授與 AWS STS 和 Lambda 端點所需的 Lambda 服務主體存取權。

Example VPC 端點策略- AWS STS 端點

```
{
    "Statement": [
        {
            "Action": "sts:AssumeRole",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
}
```

Example VPC 私人雲端端點政策

```
{
    "Statement": [
        {
            "Action": "lambda:InvokeFunction",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
}
```

如果您的 Kafka 代理程式使用驗證，您也可以限制 Secrets Manager 端點的 VPC 端點原則。若要呼叫機 Secrets Manager API，Lambda 會使用您的函數角色，而不是使用 Lambda 服務主體。下列範例顯示 Secrets Manager 端點策略。

Example VPC 端點原則-Secrets Manager 端點

```
{
    "Statement": [
        {
            "Action": "secretsmanager:GetSecretValue",
            "Effect": "Allow",
            "Principal": {
                "AWS": [
                    "customer_function_execution_role_arn"
                ]
            },
            "Resource": "customer_secret_arn"
        }
    ]
}
```

如果您設定了故障時的目的地，Lambda 也會使用函數的角色來呼叫
或`s3:PutObject`使用 Lambda 管理的 ENI。`sns:Publish``sqs:sendMessage`

將 Kafka 叢集新增為事件來源

若要建立事件來源映射，使用 Lambda 主控台、AWS 開發套件或 AWS Command Line Interface (AWS CLI) 將您的 Kafka 叢集新增為 Lambda 函數觸發條件。

本節說明如何使用 Lambda 主控台和 AWS CLI建立事件來源映射。

必要條件

- 自我管理 Apache Kafka 叢集。Lambda 支援 Apache Kafka 版本 0.10.1.0 及更高版本。
- 具有存取自我管理 Kafka 叢集使用之 AWS 資源之權限的執行角色。

可自訂的取用者群組 ID

將 Kafka 設為事件來源時，您可以指定取用者群組 ID。此取用者群組 ID 是您希望 Lambda 函數加入之 Kafka 取用者群組的現有識別符。您可以使用此功能將任何進行中的 Kafka 記錄處理設定從其他取用者無縫遷移至 Lambda。

如果您指定取用者群組 ID，且該取用者群組內還有其他作用中的輪詢者，則 Kafka 會將訊息分配給所有取用者。換句話說，Lambda 不會收到有關 Kafka 主題的所有訊息。如果您希望 Lambda 處理主題中的所有訊息，請關閉該取用者群組中的任何其他輪詢者。

此外，如果您指定取用者群組 ID，且 Kafka 找到具有相同 ID 的有效現有取用者群組，則 Lambda 會忽略用於事件來源映射的 `StartingPosition` 參數。相反的，Lambda 會根據取用者群組的承諾偏移量開始處理記錄。如果您指定取用者群組 ID，但 Kafka 找不到現有的取用者群組，則 Lambda 會使用指定的 `StartingPosition` 來設定事件來源。

您指定的取用者群組 ID 在所有 Kafka 事件來源中必須是唯一的。使用指定的取用者群組 ID 建立 Kafka 事件來源映射之後，您就無法更新此值。

新增自我管理 Kafka 叢集 (主控台)

按照下列步驟，將您的 Apache Kafka 叢集和 Kafka 主題新增為 Lambda 函數的觸發條件。

將 Apache Kafka 觸發條件新增至您的 Lambda 函數 (主控台)

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇 Lambda 函數的名稱。
3. 在 函式概觀 下，選擇 新增觸發條件 。
4. 在 Trigger configuration (觸發條件) 下，執行下列動作：

    a. 選擇 Apache Kafka 觸發條件類型。

    b. 對於 Bootstrap 伺服器，輸入叢集中 Kafka 代理程式的主機和連接埠對地址，然後選擇 新增。針對叢集中的每個 Kafka 代理程式重複此操作。

    c. 對於 Topic name (主題名稱)，輸入用於在叢集中存儲記錄之 Kafka 主題的名稱。

    d. (選用) 對於 批次大小，輸入單一批次中接收的最大記錄數。

    e. 對於 Batch window (批次時段)，輸入 Lambda 調用函數之前收集記錄所花費的最長秒數。

    f. (選用) 對於取用者群組 ID ，輸入要加入的 Kafka 取用者群組 ID。

    g. (選用) 對於開始位置，請選擇最新以從最新記錄開始讀取串流、選擇水平修剪以從最早的可用記錄開始，或選擇在時間戳記為以指定開始讀取的時間戳記。

    h. (選用) 若為 VPC，請為您的 Kafka 叢集選擇 Amazon VPC。然後，選擇 VPC 子網路 和 VPC 安全群組 。

        如果只有您的 VPC 內的使用者會存取代理程式，則必須要有此設定。

    i. (選用) 對於 身分驗證 ，選擇 新增 ，然後執行下列動作：

        i. 選擇您叢集中 Kafka 代理程式的存取或身分驗證協定。

            - 如果您的 Kafka 代理程式使用 SASL/PLAIN 身分驗證，請選擇 BASIC_AUTH。

- 如果您的代理程式使用 SASL/SCRAM 身分驗證，請選擇其中一種 SASL_SCRAM 通訊協定。
- 如果您要設定 mTLS 身分驗證，請選擇 CLIENT_CERTIFICATE_TLS_AUTH 通訊協定。

ii. 若為 SASL/SCRAM 或 mTLS 身分驗證，請選擇包含 Kafka 叢集憑證的 Secrets Manager 機密金鑰。

j. (選用) 若為 加密 ，如果您的 Kafka 代理程式使用私有憑證授權機構簽署的憑證，請選擇包含 Kafka 代理程式用於 TLS 加密的根憑證授權機構憑證的 Secrets Manager 機密。

此設定適用於 SASL/SCRAM 或 SASL/PLAIN 的 TLS 加密，也適用於 mTLS 身分驗證。

k. 若要建立處於停用狀態的觸發條件以進行測試 (建議做法)，請取消勾選 啟用觸發條件 。或者，若要立即啟用觸發條件，請選取 啟用觸發條件。

5. 若要建立觸發條件，請選擇 新增 。

新增自我管理 Kafka 叢集 (AWS CLI)

使用下列範例 AWS CLI 命令，為您的 Lambda 函數建立及檢視自我管理的 Apache 卡夫卡觸發程序。

使用 SASL/SCRAM

如果 Kafka 使用者透過網際網路存取您的 Kafka 代理程式，請指定針對 SASL/SCRAM 身分驗證建立的 Secrets Manager 機密。下列範例會使用命create-event-source-mapping AWS CLI 令，將名為的 Lambda 函數對應my-kafka-function至名為的卡夫卡主題。AWSKafkaTopic

```
aws lambda create-event-source-mapping \
  --topics AWSKafkaTopic \
  --source-access-configuration Type=SASL_SCRAM_512_AUTH,URI=arn:aws:secretsmanager:us-
east-1:111122223333:secret:MyBrokerSecretName \
  --function-name arn:aws:lambda:us-east-1:111122223333:function:my-kafka-function \
  --self-managed-event-source '{"Endpoints":{"KAFKA_BOOTSTRAP_SERVERS":
["abc3.xyz.com:9092", "abc2.xyz.com:9092"]}}'
```

使用 VPC

如果只有您 VPC 內的 Kafka 使用者可存取您的 Kafka 代理程式，則必須指定您的 VPC、子網路和 VPC 安全群組。下列範例會使用命create-event-source-mapping AWS CLI 令，將名為的 Lambda 函數對應my-kafka-function至名為的卡夫卡主題。AWSKafkaTopic

```
aws lambda create-event-source-mapping \
```

```
  --topics AWSKafkaTopic \
  --source-access-configuration '[{"Type": "VPC_SUBNET", "URI":
 "subnet:subnet-0011001100"}, {"Type": "VPC_SUBNET", "URI":
 "subnet:subnet-0022002200"}, {"Type": "VPC_SECURITY_GROUP", "URI":
 "security_group:sg-0123456789"}]' \
  --function-name arn:aws:lambda:us-east-1:111122223333:function:my-kafka-function \
  --self-managed-event-source '{"Endpoints":{"KAFKA_BOOTSTRAP_SERVERS":
["abc3.xyz.com:9092", "abc2.xyz.com:9092"]}}'
```

使用檢視狀態 AWS CLI

下列範例會使用get-event-source-mapping AWS CLI 指令來描述您所建立之事件來源對應的狀態。

```
aws lambda get-event-source-mapping
          --uuid dh38738e-992b-343a-1077-3478934hjkfd7
```

失敗時的目的地

若要保留失敗的事件來源映射調用記錄，請將目標地新增到函數的事件來源映射中。每個傳送至目的地的記錄都是 JSON 文件，其中包含有關失敗叫用的中繼資料。您可以將任何 Amazon SNS 主題、Amazon SQS 佇列或 S3 儲存貯體設定為目的地。您的執行角色必須具有目標的權限：

- 對於 SQS 目的地：sq s：SendMessage
- 對於 SNS 目的地：SNS: 發佈
- 對於 S3 儲存貯體目的地：s3：PutObject和 s3：ListBuckets

此外，如果您在目的地上設定 KMS 金鑰，則視目的地類型而定，Lambda 需要下列許可：

- 如果您已針對 S3 目的地使用自己的 KMS 金鑰啟用加密，則需要 kms：金GenerateData鑰。如果 KMS 金鑰和 S3 儲存貯體目的地與 Lambda 函數和執行角色位於不同的帳戶中，請將 KMS 金鑰設定為信任執行角色以允許 KMS：GenerateDataKey。
- 如果您已針對 SQS 目的地使用自己的 KMS 金鑰啟用加密，則需要 KMS: 解密和公里：GenerateData金鑰。如果 KMS 金鑰和 SQS 佇列目的地與 Lambda 函數和執行角色位於不同的帳戶中，請將 KMS 金鑰設定為信任執行角色，以允許 KMS：解密、公里：金GenerateData鑰、公里：和公里：DescribeKey。ReEncrypt
- 如果您已針對 SNS 目的地使用自己的 KMS 金鑰啟用加密，則需要 KMS: 解密和公里：金GenerateData鑰。如果 KMS 金鑰和 SNS 主題目的地與 Lambda 函數和執行角色位於不同的帳戶

中，請將 KMS 金鑰設定為信任執行角色，以允許 KMS：解密、公里：金GenerateData鑰、公里：DescribeKey和公里：。ReEncrypt

若要使用主控台設定失敗時的目的地，請依照下列步驟執行：

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 在 函數概觀 下，選擇 新增目的地。
4. 針對來源，請選擇事件來源映射調用。
5. 對於事件來源映射，請選擇針對此函數設定的事件來源。
6. 對於條件，選取失敗時。對於事件來源映射調用，這是唯一可接受的條件。
7. 對於目標類型，請選擇 Lambda 將調用記錄傳送至的目標類型。
8. 對於 目的地，請選擇一個資源。
9. 選擇 儲存 。

您也可以使用設定失敗時的目的 AWS CLI地。例如，下列建立事件來源對映指令會將具有失敗時之 SQS 目的地的事件來源對應新增至：MyFunction

```
aws lambda create-event-source-mapping \
--function-name "MyFunction" \
--event-source-arn arn:aws:kafka:us-east-1:123456789012:cluster/
vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2 \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sqs:us-
east-1:123456789012:dest-queue"}}'
```

下列更新事件來源映射命令會將 S3 故障時的目標新增至與輸入相關聯的事件來源：uuid

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": "arn:aws:s3:::dest-bucket"}}'
```

若要移除目的地，請提供空白字串作為 destination-config 參數的引數：

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
```

```
--destination-config '{"OnFailure": {"Destination": ""}}'
```

SNS 和 SQS 範例呼叫記錄

下列範例顯示 Lambda 針對失敗的 Kafka 事件來源調用而傳送至 SNS 主題或 SQS 佇列目的地。`recordsInfo` 之下的每個索引鍵都包含 Kafka 主題和分割區，以連字號分隔。例如，對於金鑰 `"Topic-0"`，`Topic` 是 Kafka 主題，並且 `0` 是分區。對於每個主題和分區，您可以使用偏移和時間戳記資料來查找原始調用記錄。

```
{
    "requestContext": {
        "requestId": "316aa6d0-8154-xmpl-9af7-85d5f4a6bc81",
        "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted" | "MaximumPayloadSizeExceeded",
        "approximateInvokeCount": 1
    },
    "responseContext": { // null if record is MaximumPayloadSizeExceeded
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:38:06.021Z",
    "KafkaBatchInfo": {
        "batchSize": 500,
        "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/
vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2",
        "bootstrapServers": "...",
        "payloadSize": 2039086, // In bytes
        "recordsInfo": {
            "Topic-0": {
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            },
            "Topic-1": {
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
```

```
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            }
        }
    }
}
```

S3 目的地範例叫用記錄

對於 S3 的目的地，Lambda 會將整個調用記錄與中繼資料一起傳送至目的地。下列範例顯示 Lambda 針對失敗的 Kafka 事件來源調用而傳送至 S3 儲存貯體目的地。除了上一個 SQS 和 SNS 目的地範例中的所有欄位之外，此 payload 欄位還包含原始調用記錄做為逸出 JSON 字串。

```
{
    "requestContext": {
        "requestId": "316aa6d0-8154-xmpl-9af7-85d5f4a6bc81",
        "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted" | "MaximumPayloadSizeExceeded",
        "approximateInvokeCount": 1
    },
    "responseContext": { // null if record is MaximumPayloadSizeExceeded
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:38:06.021Z",
    "KafkaBatchInfo": {
        "batchSize": 500,
        "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/
vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2",
        "bootstrapServers": "...",
        "payloadSize": 2039086, // In bytes
        "recordsInfo": {
            "Topic-0": {
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            },
            "Topic-1": {
```

```
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            }
        }
    },
    "payload": "<Whole Event>" // Only available in S3
}
```

> **Tip**
>
> 建議您在目的地儲存貯體上啟用 S3 版本控制。

使用 Kafka 叢集作為事件來源

當您將 Apache Kafka 叢集新增為 Lambda 函數的觸發條件時，該叢集會用作事件來源。

Lambda 會根據您指定的，讀取您在CreateEventSourceMapping請求`Topics`中指定的 Kafka 主題中的事件`StartingPosition`資料。處理成功後，您的 Kafka 主題將遞交給 Kafka 叢集。

如果您指定 `StartingPosition` 作為 `LATEST`，Lambda 會開始讀取屬於該主題的每個分割區中的最新訊息。由於 Lambda 開始讀取訊息之前，觸發條件組態後可能存在一些延遲，所以 Lambda 不會讀取此時段產生的任何訊息。

Lambda 會處理您指定的一個或多個 Kafka 主題分割區中的記錄，並將 JSON 承載傳送至您的函數。當有更多記錄可用時，Lambda 會繼續根據您在CreateEventSourceMapping請求中指定的`BatchSize`值分批處理記錄，直到函數趕上主題為止。

如果函數針對批次中的任何訊息傳回錯誤，Lambda 會重試整個批次的訊息，直至處理成功或訊息過期。您可以將所有重試嘗試失敗的記錄傳送至失敗時的目的地，以便稍後處理。

> **Note**
>
> 雖然 Lambda 函數的逾時上限通常為 15 分鐘，但 Amazon MSK、自我管理的 Apache Kafka、Amazon DocumentDB 以及 Amazon MQ for ActiveMQ 和 Amazon MQ for RabbitMQ

的事件來源映射只支援 14 分鐘逾時限制上限的函數。此限制條件可確保事件來源映射能夠正確處理函數錯誤和重試。

輪詢和串流開始位置

請注意，建立和更新事件來源映射期間的串流輪詢最終會一致。

- 在建立事件來源映射期間，從串流開始輪詢事件可能需要幾分鐘時間。
- 在更新事件來源映射期間，從串流停止並重新開始輪詢事件可能需要幾分鐘時間。

這種行為表示如果您指定 LATEST 當作串流的開始位置，事件來源映射可能會在建立或更新期間遺漏事件。若要確保沒有遺漏任何事件，請將串流開始位置指定為 TRIM_HORIZON 或 AT_TIMESTAMP。

Kafka 事件來源的自動調整規模

當您最初建立 Apache Kafka 事件來源時，Lambda 會分配一個取用者來處理 Kafka 主題的所有分割區。每個取用者都有多個並行運行的處理器以處理增加的工作負載。此外，Lambda 會根據工作負載自動增加或減少取用者數量。為了保留每個分割區中的訊息順序，主題中每個分割區的取用者數上限是一個取用者。

每 1 分鐘，Lambda 會評估主題中所有分割區的取用者偏移延遲。如果延遲太高，則表示分割區接收訊息的速度比 Lambda 處理訊息的速度更快。如有必要，Lambda 會新增或移除主題取用者。新增或刪除取用者的擴展過程，將在三分鐘的評估期間內完成。

如果您的目標 Lambda 函數過載，則 Lambda 會減少取用者的數量。此動作可透過減少取用者可擷取和傳送至函數的訊息數量，減少函數的工作負載。

若要監控 Kafka 主題的輸送量，您可以檢視 Apache Kafka 取用者指標，例如 consumer_lag 和 consumer_offset。若要檢查並行發生的函數調用次數，您也可以監控函數的並行指標。

事件來源映射錯誤

當您將 Apache Kafka 叢集新增為 Lambda 函數的事件來源時，如果您的函數遇到錯誤，則您的 Kafka 取用者會停止處理記錄。主題分割區的取用者是訂閱、讀取和處理記錄者。您的其他 Kafka 取用者可以繼續處理記錄，只要他們沒有遇到相同的錯誤。

若要確定停止取用者的原因，請檢查 EventSourceMapping 的回應中的 StateTransitionReason 欄位。下列清單說明了您可能收到的事件來源錯誤：

ESM_CONFIG_NOT_VALID

事件來源映射組態無效。

EVENT_SOURCE_AUTHN_ERROR

Lambda 無法驗證事件來源。

EVENT_SOURCE_AUTHZ_ERROR

Lambda 沒有存取事件來源所需的許可。

FUNCTION_CONFIG_NOT_VALID

函數組態無效。

> **Note**
>
> 如果您的 Lambda 事件記錄超過允許的 6 MB 大小限制，則可能不會進行處理。

Amazon CloudWatch 指標

當您的函數處理記錄時，Lambda 會發出 `OffsetLag` 指標。此指標的值是寫入 Kafka 事件來源主題的最後一筆記錄與函數取用者群組處理的最後一筆記錄之間的偏移量的差值。您可以使用 `OffsetLag` 來預估新增記錄時與取用者群組處理記錄時之間的延遲。

`OffsetLag` 的增加趨勢可能表示函數取用者群組中的輪詢者問題。如需詳細資訊，請參閱 使用 Lambda 函數指標。

自我管理的 Apache Kafka 組態參數

所有 Lambda 事件來源類型都共用相同CreateEventSourceMapping的UpdateEventSourceMappingAPI 作業。但是，只有一些參數適用於 Apache Kafka。

適用於自我管理 Apache Kafka 的事件來源參數

參數	必要	預設	備註
BatchSize	否	100	上限：10,000
已啟用	N	已啟用	

參數	必要	預設	備註
FunctionName	Y		
FilterCriteria	N		Lambda 事件篩選
MaximumBatchingWindowIn秒	N	500 毫秒	批次處理行為
SelfManagedEventSource	Y		Kafka 代理程式清單。只能在建立時進行設定
SelfManagedKafkaEventSourceConfig	N	包含預設為唯一值的 ConsumerGroupId 欄位。	只能在建立時進行設定
SourceAccess配置	N	沒有憑證	叢集的 VPC 資訊或身分驗證憑證 對於 SASL_PLAIN，設定為 BASIC_AUTH
StartingPosition	Y		AT_TIMESTAMP、TRIM_HORIZON 或 LATEST 只能在建立時進行設定
StartingPosition時間戳	N		如果設定為 [時間 StartingPosition 戳記]，則為必要
主題	Y		主題名稱 只能在建立時進行設定

搭配 Amazon SQS 使用 Lambda

> **Note**
>
> 如果您想要將資料傳送到 Lambda 函數以外的目標，或在傳送資料之前豐富資料，請參閱 Amazon EventBridge 管道。

您可以使用 Lambda 函數處理 Amazon Simple Queue Service (Amazon SQS) 佇列中的訊息。Lambda 支援事件來源對應的標準佇列和先進先出 (FIFO) 佇列。

了解 Amazon SQS 事件來源對應的輪詢和批次處理行為

透過 Amazon SQS 事件來源對應，Lambda 會輪詢佇列，並透過事件同步叫用您的函數。每個事件都可以包含佇列中多個訊息的批次。Lambda 一次接收一個批次這些事件，並針對每個批次叫用函數一次。當您的函數成功處理批次時，Lambda 會從佇列中刪除其訊息。

當 Lambda 收到批次時，訊息會保留在佇列中，但會在佇列可見性逾時的時間長度內隱藏。如果您的函數成功處理批次中的所有訊息，Lambda 會從佇列中刪除訊息。根據預設，如果您的函數在處理批次時遇到錯誤，則該批次中的所有訊息會在可見性逾時到期後再次顯示在佇列中。因此，您的函數程式碼必須能夠多次處理相同的訊息，而不會產生副作用。

> **Warning**
>
> Lambda 事件來源對應至少處理每個事件一次，並且可能會重複處理記錄。為了避免與重複事件相關的潛在問題，我們強烈建議您將函數代碼設為冪等。若要深入了解，請參閱 AWS 知識中心如何讓 Lambda 函數具有冪等性。

若要防止 Lambda 多次處理訊息，您可以將事件來源對應設定為在函數回應中包含批次項目失敗，或者在 Lambda 函數成功處理訊息時，使用 DeleteMessageAPI 從佇列中移除訊息。

如需 Lambda 支援 SQS 事件來源對應之組態參數的詳細資訊，請參閱the section called “建立 SQS 事件來源對映”。

標準佇列訊息事件範例

Example Amazon SQS 訊息事件 (標準佇列)

```
{
```

```
    "Records": [
        {
            "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
            "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",
            "body": "Test message.",
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1545082649183",
                "SenderId": "AIDAIENQZJOLO23YVJ4VO",
                "ApproximateFirstReceiveTimestamp": "1545082649185"
            },
            "messageAttributes": {},
            "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
            "eventSource": "aws:sqs",
            "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:my-queue",
            "awsRegion": "us-east-2"
        },
        {
            "messageId": "2e1424d4-f796-459a-8184-9c92662be6da",
            "receiptHandle": "AQEBzWwaftRI0KuVm4tP+/7q1rGgNqicHq...",
            "body": "Test message.",
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1545082650636",
                "SenderId": "AIDAIENQZJOLO23YVJ4VO",
                "ApproximateFirstReceiveTimestamp": "1545082650649"
            },
            "messageAttributes": {},
            "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
            "eventSource": "aws:sqs",
            "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:my-queue",
            "awsRegion": "us-east-2"
        }
    ]
}
```

依預設，Lambda 會一次輪詢佇列中最多 10 則訊息，並將該批次傳送給函數。若要避免叫用少量記錄的函數，您可以設定批次視窗，將事件來源設定為緩衝記錄長達 5 分鐘。在叫用函數之前，Lambda 會繼續輪詢來自標準佇列的訊息，直到批次視窗過期、達到叫用承載大小配額或達到設定的最大批次大小為止。

如果您使用的是批次時間範圍，並且您的 SQS 佇列包含非常低的流量，Lambda 可能會等到 20 秒，然後再調用您的函數。即使您將批次時間範圍設定為低於 20 秒也是如此。

> **Note**
>
> 在 Java 中，還原序列化 JSON 時可能會遇到 null 指標錯誤。這可能是由於 JSON 物件映射器對「Records」和「eventSourceARN」案例轉換的方式所致。

FIFO 佇列訊息事件範例

對於 FIFO 佇列，記錄包含與重複資料刪除和定序相關的其他屬性。

Example Amazon SQS 訊息事件 (FIFO 佇列)

```
{
    "Records": [
        {
            "messageId": "11d6ee51-4cc7-4302-9e22-7cd8afdaadf5",
            "receiptHandle": "AQEBBX8nesZEXmkhsmZeyIE8iQAMig7qw...",
            "body": "Test message.",
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1573251510774",
                "SequenceNumber": "18849496460467696128",
                "MessageGroupId": "1",
                "SenderId": "AIDAIO23YVJENQZJOL4VO",
                "MessageDeduplicationId": "1",
                "ApproximateFirstReceiveTimestamp": "1573251510774"
            },
            "messageAttributes": {},
            "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
            "eventSource": "aws:sqs",
            "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:fifo.fifo",
            "awsRegion": "us-east-2"
        }
    ]
}
```

建立和設定 Amazon SQS 事件來源映射

若要使用 Lambda 處理 Amazon SQS 訊息，請使用適當的設定來設定佇列，然後建立 Lambda 事件來源對應。

配置要搭配 Lambda 使用的佇列

如果您還沒有現有的 Amazon SQS 佇列，請建立一個佇列作為 Lambda 函數的事件來源。然後設定佇列，讓 Lambda 函數有足夠的時間來處理每個批次的事件。

若要讓您的功能有時間處理每批記錄，請將來源佇列的可見性逾時設定至少為函數組態逾時的六倍。如果您的函數在處理上一個批次時被限制，則 Lambda 可以使用額外的時間來重試。

根據預設，如果 Lambda 在處理批次時發生錯誤，則該批次中的所有訊息都會返回佇列。在可見性逾時之後，Lambda 會再次看到訊息。您可以將事件來源對應設定為使用部分批次回應，僅將失敗的訊息傳回佇列。此外，如果您的函數無法多次處理訊息，Amazon SQS 可以將其傳送到無效字母佇列。我們建議您將`maxReceiveCount`來源佇列的重新磁碟原則設定為至少 5。這可讓 Lambda 在將失敗訊息直接傳送至無效字母佇列之前，有幾次重試的機會。

設定 Lambda 執行角色權限

受 AWSLambdaSQSQueueExecutionRole AWS 管政策包括 Lambda 需要從您的 Amazon SQS 佇列讀取的許可。您可以將此受管理的原則新增至函數的執行角色。

或者，如果您使用的是加密佇列，也需要將下列許可新增至您的執行角色：

- kms:Decrypt

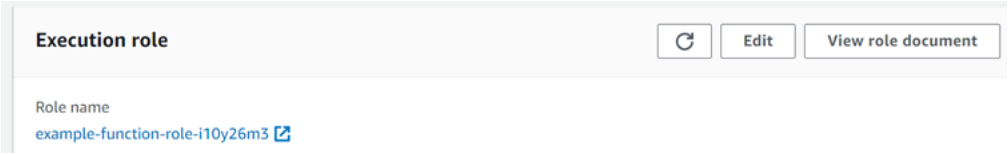
建立 SQS 事件來源對映

建立事件來源映射，指示 Lambda 從您的佇列傳送項目至 Lambda 函數。您可以建立多個事件來源映射，使用單一函數處理來自多個佇列的項目。當 Lambda 調用目標函數時，事件可能包含多個項目，最多為可設定的最大批次大小。

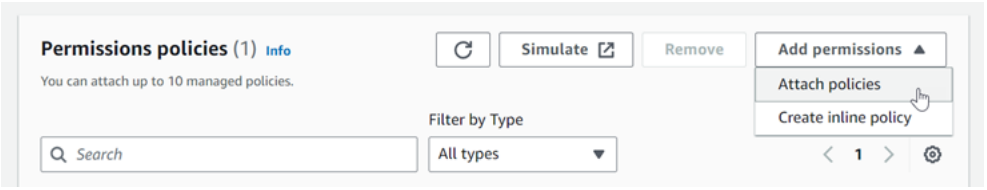
若要將函數設定為從 Amazon SQS 讀取，請將 AWSLambdaSQSQueueExecutionRole AWS 受管政策附加到執行角色。然後，使用下列步驟從主控台建立 SQS 事件來源對應。

若要新增權限並建立觸發器

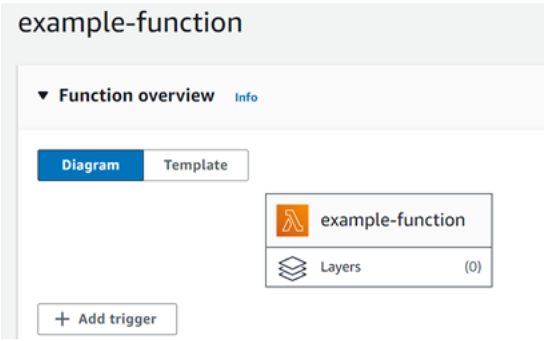
1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇函數的名稱。
3. 依序選擇 Configuration (組態) 索引標籤和 Permissions (許可)。
4. 在 [角色名稱] 下，選擇指向您的執行角色的連結。此連結會在 IAM 主控台中開啟角色。



5. 選擇新增許可，然後選擇連接政策。



6. 在搜尋欄位中輸入 AWSLambdaSQSQueueExecutionRole。將此原則新增至您的執行角色。這是一項 AWS 受管政策，其中包含函數需要從 Amazon SQS 佇列讀取的許可。如需有關此原則的詳細資訊，請參閱 AWSLambdaSQSQueueExecutionRole受AWS 管理的原則參考中的。
7. 返回 Lambda 主控台中的函數。在 函式概觀 下，選擇 新增觸發條件 。



8. 選擇觸發條件類型。
9. 設定需要的選項，然後選擇 新增 。

Lambda 支援 Amazon SQS 事件來源的下列組態選項：

SQS 佇列

要從中讀取記錄的 Amazon SQS 佇列。

啟用觸發條件

事件來源映射的狀態。系統會預設選取 啟用觸發條件 。

批次大小

每個批次中要傳送至函數的記錄數量上限。若是標準佇列，這最高可達 10,000 個記錄。若是 FIFO 佇列，最大值為 10。批次大小若超過 10，您還必須將批次間隔 (MaximumBatchingWindowInSeconds) 設定為至少 1 秒。

設定您的函數逾時，以便有足夠的時間來處理整批項目。如果項目需要長時間處理，請選擇較小的批次大小。大型批次大小可提升效率，適用非常快速或開銷龐大的工作負載。如果您在函數上設定保留並行，應最少設定五個並行執行，藉此降低 Lambda 調用函數時的調節錯誤機率。

只要事件的總大小不超過同步調用的調用有效負載大小配額（6 MB），Lambda 會在單次調用中將批次中的所有記錄傳遞給函數。Lambda 和 Amazon SQS 兩者均會產生每筆記錄的中繼資料。這個額外的中繼資料會計入總酬載大小，並且可能會導致批次中傳送的記錄總數低於您設定的批次大小。Amazon SQS 傳送的中繼資料欄位長度可能有所不同。如需 Amazon SQS 中繼資料欄位的詳細資訊，請參閱 Amazon 簡單佇列服務 ReceiveMessageAPI 參考中的 API 操作文件。

批次視窗

調用函式前收集記錄的最長時間 (單位為秒)。這僅適用於標準佇列。

如果您使用的批次視窗超過 0 秒，則必須考慮佇列可見性逾時中增加的處理時間。我們建議將您的佇列可見性逾時設定為函數逾時的六倍，再加上 `MaximumBatchingWindowInSeconds` 的值。這可讓您的 Lambda 函數有時間處理每個事件批次，並在發生節流錯誤時重試。

當訊息可供使用，Lambda 會開始批次處理訊息。Lambda 會開始一次處理五個批次，而函數有五個並行調用。如果訊息仍可用，則 Lambda 每分鐘會再為函數增加最多 300 個執行個體，上限是 1,000 個函數執行個體。若要深入了解函數擴展與並行，請參閱 Lambda 函數擴展。

若要處理更多訊息，您可以最佳化 Lambda 函數以達到更高的輸送量。如需詳細資訊，請參閱了解如何透過 Amazon SQS 標準佇列 AWS Lambda 擴展規模。

最大並行數量

事件來源可以調用的並行函數上限。如需詳細資訊，請參閱 設定 Amazon SQS 事件來源的並行上限。

篩選條件

新增篩選條件來控制 Lambda 要傳送哪些事件給函數進行處理。如需更多詳細資訊，請參閱 Lambda 事件篩選。

設定 SQS 事件來源對映的縮放行為

對於標準佇列，Lambda 會使用長輪詢來輪詢佇列，直到佇列變為作用中為止。當訊息可用時，Lambda 會開始一次處理五個批次，而函數有五個並行調用。如果訊息仍然可用，則 Lambda 會將讀取批次的處理數量增加為每分鐘最多 300 個執行個體。事件來源映射可同時處理的批次數量上限為 1,000。

針對 FIFO 佇列，Lambda 會按照其接收的順序傳送訊息到您的函數。當您傳送訊息到 FIFO 佇列時，您可以指定訊息群組 ID。Amazon SQS 可確保相同群組中的訊息依序傳遞至 Lambda。當 Lambda 將訊息分批讀取時，每個批次可能包含來自多個訊息群組的訊息，但訊息的順序會維持不變。如果您的函數傳回錯誤，函數會先在受影響的訊息上嘗試所有重試，之後 Lambda 才會從相同群組接收到額外訊息。

設定 Amazon SQS 事件來源的並行上限

您可以使用最大並行設定來控制 SQS 事件來源的縮放行為。並行上限設定限制了 Amazon SQS 事件來源可以調用的函數並行執行個體數。並行上限是事件來源層級的設定。如果您將多個 Amazon SQS 事件來源映射到一個函數，那麼每個事件來源都可以有個別的並行上限設定。您可以使用並行上限來防止一個佇列用完函數的所有預留並行配額，或其餘的帳戶並行配額。對 Amazon SQS 事件來源設定並行無需付費。

重要的是，並行上限和預留並行是兩項獨立的設定。請勿將並行上限設為超過函數的預留並行。若您設定了並行上限，請確定函數的預留並行大於或等於函數上所有 Amazon SQS 事件來源的總並行上限。若小於此上限，Lambda 可能會限流您的訊息。

如果沒有設定並行上限，Lambda 可能會將您的 Amazon SQS 事件來源擴展至帳戶的總並行配額 (預設為 1,000)。

> **Note**
>
> 對於 FIFO 佇列，並行調用的上限是訊息群組 ID (`messageGroupId`) 的數量或最大並行設定 (以較低者為準)。例如，如果您有六個訊息群組 ID，而最大並行設定為 10，則函數最多可以有六個並行調用。

您可以對新的和現有的 Amazon SQS 事件來源映射設定並行上限。

使用 Lambda 主控台設定並行上限

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇函數的名稱。
3. 在 函數概觀 下，選擇 SQS。這會開啟 Configuration (組態) 索引標籤。
4. 選取 Amazon SQS 觸發條件，然後選擇 編輯 。
5. Maximum concurrency (並行上限) 請輸入介於 2 到 1,000 之間的數字。若要關閉並行上限，請將方塊保留空白。
6. 選擇 儲存 。

使用 AWS Command Line Interface （ AWS CLI ）配置最大並發

使用 update-event-source-mapping 命令和 --scaling-config 選項。範例：

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --scaling-config '{"MaximumConcurrency":5}'
```

若要關閉並行上限，請為 --scaling-config 輸入空值：

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --scaling-config "{}"
```

使用 Lambda API 設定並行上限

ScalingConfig對物件使用CreateEventSourceMapping或UpdateEventSourceMapping動作。

處理 Lambda 中 SQS 事件來源的錯誤

為了處理與 SQS 事件來源相關的錯誤，Lambda 會自動使用具有輪詢策略的重試策略。您也可以將 SQS 事件來源對應設定為傳回部分批次回應，以自訂錯誤處理行為。

失敗調用的輪詢策略

當調用失敗，Lambda 會在實作輪詢策略時嘗試重試調用。根據 Lambda 是否因函數程式碼錯誤或限流而發生失敗，輪詢策略會略有不同。

- 如果您的函數代碼導致錯誤，Lambda 將停止處理並重試調用。與此同時，Lambda 會逐漸退回，減少分配給 Amazon SQS 事件來源映射的並行數量。佇列的可見性逾時用完之後，訊息會再次出現在佇列中。
- 如果是限流導致調用失敗，Lambda 會減少分配給 Amazon SQS 事件來源映射的並行數量，逐漸重試輪詢。Lambda 會持續重試訊息，直到訊息的時間戳記超過佇列的可見性逾時為止，此時 Lambda 會捨棄訊息。

實作部分批次回應

當 Lambda 函數在處理批次時遇到錯誤，根據預設，該批次中的所有訊息會再次顯示在佇列中，包含 Lambda 已順利處理的訊息。因此，您的函數可能最後會處理數次相同的訊息。

若要避免重新處理失敗批次中成功處理過的訊息，您可以設定事件來源映射，僅讓失敗的訊息再次可見。我們將其稱為部分批次回應。若要開啟部分批次回應，請在配置事件來源對應時ReportBatchItemFailures為「FunctionResponse類型」動作指定。這可以讓您的函數傳回部分成功，有助於減少記錄上不必要的重試次數。

啟動 ReportBatchItemFailures 時，Lambda 不會在函數調用失敗時 縮減訊息輪詢的規模。如果您預期部分訊息會失敗，且不希望這些失敗影響到訊息的處理速度，則請使用 ReportBatchItemFailures。

> **Note**
>
> 使用部分批次回應時，請注意下列事項：
>
> - 如果函數擲出例外情況，便會將整個批次視為完全失敗。
> - 如果您將此功能與 FIFO 佇列一起使用，您的函數應該在第一次失敗後停止處理訊息，並傳回 batchItemFailures 中所有失敗與尚未處理的訊息。這有助於保留佇列中訊息的順序。

若要啟動部分批次報告

1. 檢閱 實作部分批次回應的最佳實務。
2. 執行以下命令以便為函數啟用 ReportBatchItemFailures。要檢索事件源映射的 UUID，請運行列 AWS CLI 表事件源映射命令。

```
aws lambda update-event-source-mapping \
--uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
--function-response-types "ReportBatchItemFailures"
```

3. 更新函數程式碼以擷取所有例外狀況，並傳回 batchItemFailures JSON 回應中的失敗訊息。batchItemFailures 回應必須含有以 itemIdentifier JSON 值表示的訊息 ID 清單。

   例如，假設您有五則訊息的批次，其中，訊息 ID 分別為 id1、id2、id3、id4，以及 id5。您的函數已成功處理 id1、id3，以及 id5。若要讓訊息 id2 和 id4 再次於佇列中可見，您的函數應傳回以下回應：

```
{
  "batchItemFailures": [
        {
            "itemIdentifier": "id2"
```

```
        },
        {
            "itemIdentifier": "id4"
        }
    ]
}
```

以下範例函數程式碼會傳回批次中失敗訊息 ID 的清單：

.NET

AWS SDK for .NET

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 .NET 搭配 Lambda 報告 SQS 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SQSEvents;

// Assembly attribute to enable the Lambda function's JSON input to be
 converted into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJson
namespace sqsSample;

public class Function
{
    public async Task<SQSBatchResponse> FunctionHandler(SQSEvent evnt,
 ILambdaContext context)
    {
        List<SQSBatchResponse.BatchItemFailure> batchItemFailures = new
 List<SQSBatchResponse.BatchItemFailure>();
        foreach(var message in evnt.Records)
        {
            try
```

```
            {
                //process your message
                await ProcessMessageAsync(message, context);
            }
            catch (System.Exception)
            {
                //Add failed message identifier to the batchItemFailures list
                batchItemFailures.Add(new
 SQSBatchResponse.BatchItemFailure{ItemIdentifier=message.MessageId});
            }
        }
        return new SQSBatchResponse(batchItemFailures);
    }

    private async Task ProcessMessageAsync(SQSEvent.SQSMessage message,
 ILambdaContext context)
    {
        if (String.IsNullOrEmpty(message.Body))
        {
            throw new Exception("No Body in SQS Message.");
        }
        context.Logger.LogInformation($"Processed message {message.Body}");
        // TODO: Do interesting work based on the new message
        await Task.CompletedTask;
    }
}
```

Go

SDK for Go V2

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 使用 Lambda 報告 SQS 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
```

```
package main

import (
 "context"
 "encoding/json"
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, sqsEvent events.SQSEvent)
 (map[string]interface{}, error) {
 batchItemFailures := []map[string]interface{}{}

 for _, message := range sqsEvent.Records {

  if /* Your message processing condition here */ {
   batchItemFailures = append(batchItemFailures, map[string]interface{}
{"itemIdentifier": message.MessageId})
  }
 }

 sqsBatchResponse := map[string]interface{}{
  "batchItemFailures": batchItemFailures,
 }
 return sqsBatchResponse, nil
}

func main() {
 lambda.Start(handler)
}
```

Java

適用於 Java 2.x 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Java 搭配 Lambda 報告 SQS 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SQSEvent;
import com.amazonaws.services.lambda.runtime.events.SQSBatchResponse;

import java.util.ArrayList;
import java.util.List;

public class ProcessSQSMessageBatch implements RequestHandler<SQSEvent,
 SQSBatchResponse> {
    @Override
    public SQSBatchResponse handleRequest(SQSEvent sqsEvent, Context context)
 {

        List<SQSBatchResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<SQSBatchResponse.BatchItemFailure>();
        String messageId = "";
        for (SQSEvent.SQSMessage message : sqsEvent.getRecords()) {
            try {
                //process your message
                messageId = message.getMessageId();
            } catch (Exception e) {
                //Add failed message identifier to the batchItemFailures
 list
                batchItemFailures.add(new
 SQSBatchResponse.BatchItemFailure(messageId));
            }
        }
        return new SQSBatchResponse(batchItemFailures);
    }
}
```

JavaScript

適用於 JavaScript (v3) 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 JavaScript Lambda 報告 SQS 批次項目失敗

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
export const handler = async (event, context) => {
    const batchItemFailures = [];

    for (const record of event.Records) {
        try {
            await processMessageAsync(record, context);
        } catch (error) {
            batchItemFailures.push({ itemIdentifier: record.messageId });
        }
    }

    return { batchItemFailures };
};

async function processMessageAsync(record, context) {
    if (record.body && record.body.includes("error")) {
        throw new Error("There is an error in the SQS Message.");
    }
    console.log(`Processed message: ${record.body}`);
}
```

使用 TypeScript Lambda 報告 SQS 批次項目失敗

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
```

```
import { SQSEvent, SQSBatchResponse, Context, SQSBatchItemFailure,
 SQSRecord } from 'aws-lambda';

export const handler = async (event: SQSEvent, context: Context):
 Promise<SQSBatchResponse> => {
    const batchItemFailures: SQSBatchItemFailure[] = [];

    for (const record of event.Records) {
        try {
            await processMessageAsync(record);
        } catch (error) {
            batchItemFailures.push({ itemIdentifier: record.messageId });
        }
    }

    return {batchItemFailures: batchItemFailures};
};

async function processMessageAsync(record: SQSRecord): Promise<void> {
    if (record.body && record.body.includes("error")) {
        throw new Error('There is an error in the SQS Message.');
    }
    console.log(`Processed message ${record.body}`);
}
```

PHP

適用於 PHP 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 PHP 使用 Lambda 回報 SQS 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php
```

```
use Bref\Context\Context;
use Bref\Event\Sqs\SqsEvent;
use Bref\Event\Sqs\SqsHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends SqsHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleSqs(SqsEvent $event, Context $context): void
    {
        $this->logger->info("Processing SQS records");
        $records = $event->getRecords();

        foreach ($records as $record) {
            try {
                // Assuming the SQS message is in JSON format
                $message = json_decode($record->getBody(), true);
                $this->logger->info(json_encode($message));
                // TODO: Implement your custom processing logic here
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $this->markAsFailed($record);
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords SQS
 records");
    }
}

$logger = new StderrLogger();
```

```
return new Handler($logger);
```

Python

適用於 Python (Boto3) 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Python 搭配 Lambda 報告 SQS 批次項目失敗。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0

def lambda_handler(event, context):
    if event:
        batch_item_failures = []
        sqs_batch_response = {}

        for record in event["Records"]:
            try:
                # process message
            except Exception as e:
                batch_item_failures.append({"itemIdentifier":
 record['messageId']})

        sqs_batch_response["batchItemFailures"] = batch_item_failures
        return sqs_batch_response
```

Ruby

適用於 Ruby 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Ruby 搭配 Lambda 報告 SQS 批次項目失敗。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'json'

def lambda_handler(event:, context:)
  if event
    batch_item_failures = []
    sqs_batch_response = {}

    event["Records"].each do |record|
      begin
        # process message
      rescue StandardError => e
        batch_item_failures << {"itemIdentifier" => record['messageId']}
      end
    end

    sqs_batch_response["batchItemFailures"] = batch_item_failures
    return sqs_batch_response
  end
end
```

Rust

適用於 Rust 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Rust 搭配 Lambda 報告 SQS 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::{
    event::sqs::{SqsBatchResponse, SqsEvent},
    sqs::{BatchItemFailure, SqsMessage},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn process_record(_: &SqsMessage) -> Result<(), Error> {
    Err(Error::from("Error processing message"))
}

async fn function_handler(event: LambdaEvent<SqsEvent>) ->
 Result<SqsBatchResponse, Error> {
    let mut batch_item_failures = Vec::new();
    for record in event.payload.records {
        match process_record(&record).await {
            Ok(_) => (),
            Err(_) => batch_item_failures.push(BatchItemFailure {
                item_identifier: record.message_id.unwrap(),
            }),
        }
    }

    Ok(SqsBatchResponse {
        batch_item_failures,
    })
}
```

```
#[tokio::main]
async fn main() -> Result<(), Error> {
    run(service_fn(function_handler)).await
}
```

如果失敗的事件沒有返回佇列，請參閱如何疑難排解 Lambda 函數 SQS ReportBatchItemFailures？在 AWS 知識中心。

成功與失敗條件

如果您的函數傳回下列任一項目，Lambda 會將批次視為完全成功：

- 空白 `batchItemFailures` 清單
- Null `batchItemFailures` 清單
- 空白 `EventResponse`
- Null `EventResponse`

如果您的函數傳回下列任一項目，Lambda 會將批次視為完全失敗：

- 無效的 JSON 回應
- 空白字串 `itemIdentifier`
- Null `itemIdentifier`
- 具有錯誤金鑰名稱的 `itemIdentifier`
- 具有不存在之訊息 ID 的 `itemIdentifier` 值

CloudWatch 度量

若要判斷您的函數是否正確報告批次項目失敗，您可以在 `ApproximateAgeOfOldestMessage` Amazon 中監控`NumberOfMessagesDeleted`和 Amazon SQS 指標。 CloudWatch

- `NumberOfMessagesDeleted` 會追蹤從佇列移除的訊息數目。如果下降到 0，表示您的函數回應並未正確傳回失敗訊息。
- `ApproximateAgeOfOldestMessage` 會追蹤最舊訊息停留在佇列中的時間長度。此指標的急劇增加可能表示您的函數並未正確傳回失敗訊息。

Amazon SQS 事件來源對應的 Lambda 參數

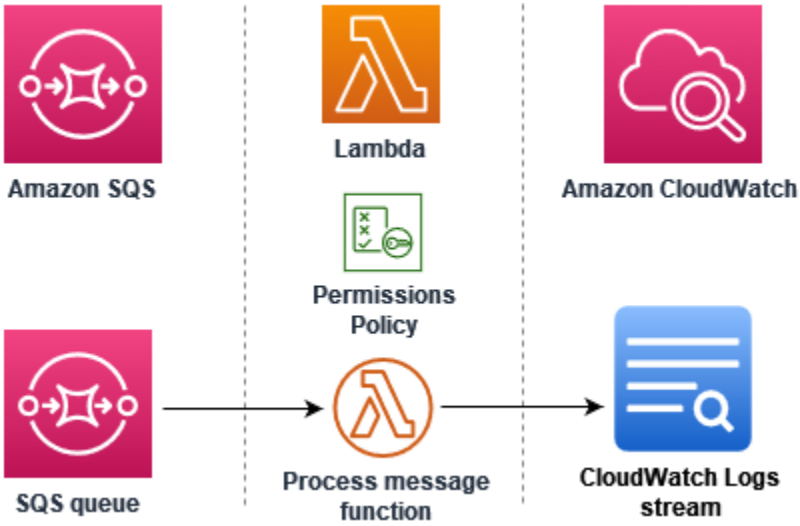
所有 Lambda 事件來源類型都共用相同CreateEventSourceMapping的UpdateEventSourceMappingAPI 作業。但是，只有一些參數適用於 Amazon SQS。

適用於 Amazon SQS 的事件來源參數

參數	必要	預設	備註
BatchSize	N	10	對於標準佇列，最大值為 10,000。對於 FIFO 隊列，最大值為 10。
已啟用	N	true	
EventSource阿恩	Y		資料串流或串流消費者的 ARN
FunctionName	Y		
FilterCriteria	N		Lambda 事件篩選
FunctionResponse類型	N		若要讓函數報告批次中的特定失敗，請將值 `ReportBatchItemFailures` 包含在 `FunctionResponseTypes` 中。如需詳細資訊，請參閱 實作部分批次回應。
MaximumBatchingWindowIn秒	N	0	
ScalingConfig	N		設定 Amazon SQS 事件來源的並行上限

教學課程：搭配 Amazon SQS 使用 Lambda

在本教學課程中，您將建立一個 Lambda 函數，該函數取用 Amazon Simple Queue Service (Amazon SQS) 佇列中的訊息。只要有新訊息加入佇列，Lambda 函數就會執行。該函數將消息寫入 Amazon CloudWatch 日誌流。下圖顯示可用來完成教學課程的 AWS 資源。



請執行下列步驟以完成本教學課程：

1. 建立將訊息寫入 CloudWatch 日誌的 Lambda 函數。
2. 建立 Amazon SQS 佇列。
3. 建立 Lambda 事件來源映射。事件來源映射會讀取 Amazon SQS 佇列，並在新增訊息時調用 Lambda 函數。
4. 將訊息新增至佇列並監視 CloudWatch 記錄檔中的結果，以測試設定。

必要條件

註冊一個 AWS 帳戶

如果您沒有 AWS 帳戶，請完成以下步驟來建立一個。

若要註冊成為 AWS 帳戶

1. 開啟 https://portal.aws.amazon.com/billing/signup。
2. 請遵循線上指示進行。

   部分註冊程序需接收來電，並在電話鍵盤輸入驗證碼。

當您註冊一個時 AWS 帳戶，將創建AWS 帳戶根使用者一個。根使用者有權存取該帳戶中的所有 AWS 服務 和資源。安全性最佳做法是將管理存取權指派給使用者，並僅使用 root 使用者來執行需要 root 使用者存取權的工作。

AWS 註冊過程完成後，會向您發送確認電子郵件。您可以隨時登錄 https://aws.amazon.com/ 並選擇我的帳戶，以檢視您目前的帳戶活動並管理帳戶。

建立具有管理權限的使用者

註冊後，請保護您的 AWS 帳戶 AWS 帳戶根使用者 AWS IAM Identity Center、啟用和建立系統管理使用者，這樣您就不會將 root 使用者用於日常工作。

保護您的 AWS 帳戶根使用者

1. 選擇 Root 使用者並輸入您的 AWS 帳戶 電子郵件地址，以帳戶擁有者身分登入。AWS Management Console在下一頁中，輸入您的密碼。

   如需使用根使用者登入的說明，請參閱 AWS 登入 使用者指南中的以根使用者身分登入。
2. 若要在您的根使用者帳戶上啟用多重要素驗證 (MFA)。

   如需指示，請參閱《IAM 使用者指南》中的為 AWS 帳戶 根使用者啟用虛擬 MFA 裝置 (主控台)。

建立具有管理權限的使用者

1. 啟用 IAM Identity Center。

   如需指示，請參閱 AWS IAM Identity Center 使用者指南中的啟用 AWS IAM Identity Center。
2. 在 IAM 身分中心中，將管理存取權授予使用者。

   若要取得有關使用 IAM Identity Center 目錄 做為身分識別來源的自學課程，請參閱《使用指南》IAM Identity Center 目錄中的「以預設值設定使用AWS IAM Identity Center 者存取」。

以具有管理權限的使用者身分登入

- 若要使用您的 IAM Identity Center 使用者簽署，請使用建立 IAM Identity Center 使用者時傳送至您電子郵件地址的簽署 URL。

如需使用 IAM 身分中心使用者登入的說明，請參閱使用AWS 登入 者指南中的登入 AWS 存取入口網站。

指派存取權給其他使用者

1. 在 IAM 身分中心中，建立遵循套用最低權限許可的最佳做法的權限集。

   如需指示，請參閱《AWS IAM Identity Center 使用指南》中的「建立權限集」。
2. 將使用者指派給群組，然後將單一登入存取權指派給群組。

   如需指示，請參閱《AWS IAM Identity Center 使用指南》中的「新增群組」。

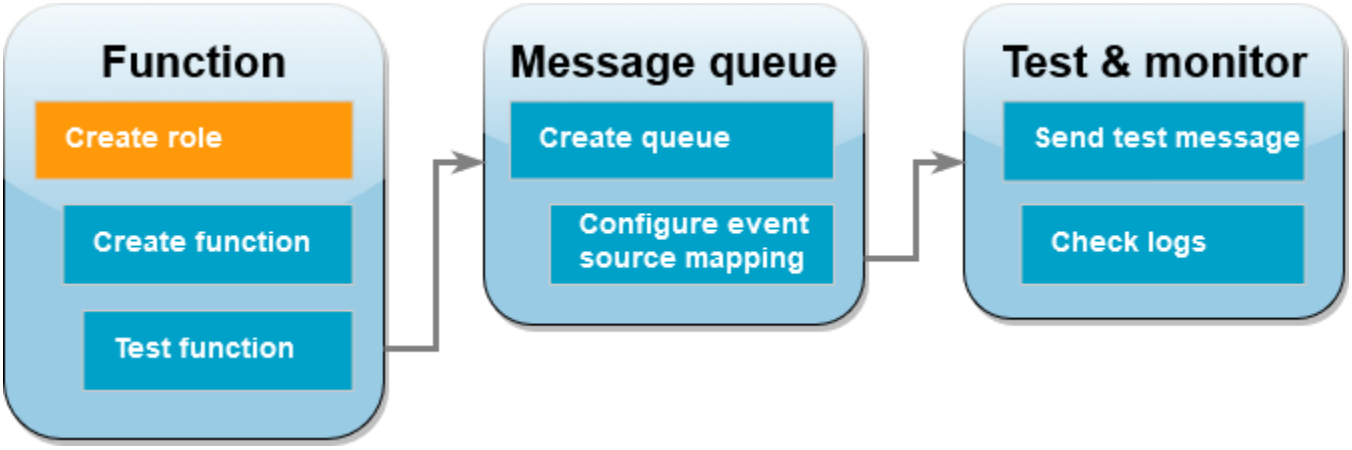
安裝 AWS Command Line Interface

如果您尚未安裝 AWS Command Line Interface，請按照安裝或更新最新版本的步驟進 AWS CLI行安裝。

本教學課程需使用命令列終端機或 Shell 來執行命令。在 Linux 和 macOS 中，使用您偏好的 Shell 和套件管理工具。

> **Note**
>
> 在 Windows 中，作業系統的內建終端不支援您常與 Lambda 搭配使用的某些 Bash CLI 命令 (例如 `zip`)。若要取得 Ubuntu 和 Bash 的 Windows 整合版本，請安裝適用於 Linux 的 Windows 子系統。

建立執行角色



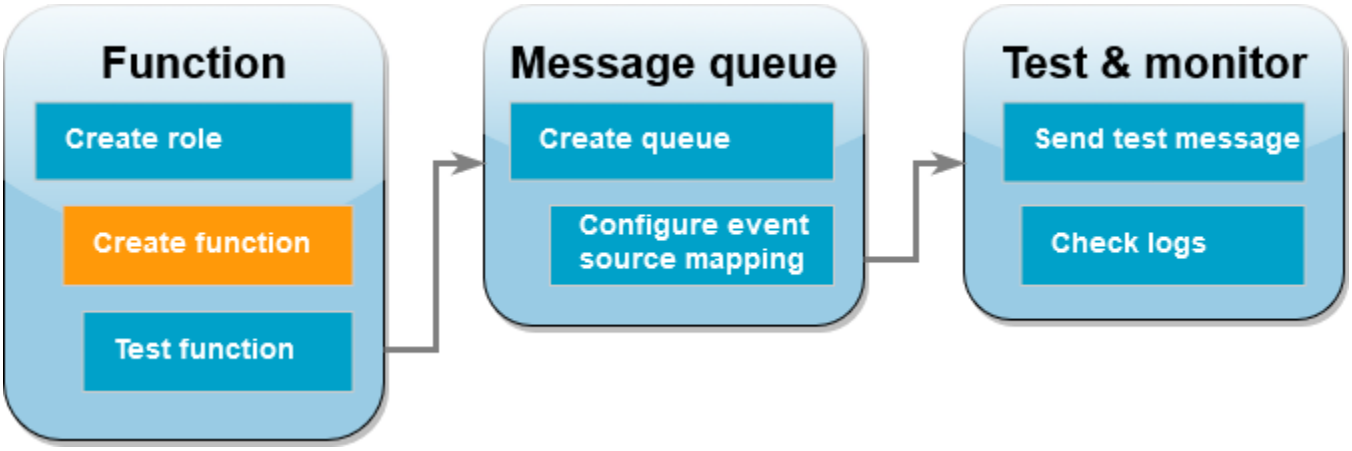
執行角色是一種 AWS Identity and Access Management (IAM) 角色，可授與 Lambda 函數存取 AWS 服務和資源的權限。若要允許您的函數從 Amazon SQS 讀取項目，請附加AWSLambdaSQSQueueExecutionRole許可政策。

建立執行角色並連接 Amazon SQS 許可政策

1. 開啟 IAM 主控台中的 角色頁面 。
2. 選擇 建立角色。
3. 針對信任的實體類型，請選擇 AWS 服務。
4. 針對使用案例，請選擇 Lambda。
5. 選擇下一步。
6. 在許可政策搜尋方塊中，輸入 **`AWSLambdaSQSQueueExecutionRole`**。
7. 選取AWSLambdaSQSQueueExecutionRole策略，然後選擇 [下一步]。
8. 針對角色詳細資訊下的角色名稱，請輸入 **`lambda-sqs-role`**，然後選擇建立角色。

角色建立後，請記下執行角色的 Amazon Resource Name (ARN)。在後續步驟中會需要用到它。

建立函數



建立 Lambda 函數，它處理 Amazon SQS 訊息。函數程式碼會將 Amazon SQS 訊息的主體 CloudWatch 記錄到日誌中。

本教學課程使用 Node.js 18.x 執行期，但我們也有提供其他執行期語言的範例程式碼。您可以在下列方塊中選取索引標籤，查看您感興趣的執行期程式碼。您將在此步驟中使用的 JavaScript 程式碼位於JavaScript索引標籤中顯示的第一個範例中。

.NET

AWS SDK for .NET

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 .NET 搭配 Lambda 來使用 SQS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SQSEvents;


// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace SqsIntegrationSampleCode
{
    public async Task FunctionHandler(SQSEvent evnt, ILambdaContext context)
    {
        foreach (var message in evnt.Records)
        {
            await ProcessMessageAsync(message, context);
        }

        context.Logger.LogInformation("done");
    }

    private async Task ProcessMessageAsync(SQSEvent.SQSMessage message,
 ILambdaContext context)
    {
        try
        {
            context.Logger.LogInformation($"Processed message {message.Body}");

            // TODO: Do interesting work based on the new message
```

```
            await Task.CompletedTask;
        }
        catch (Exception e)
        {
            //You can use Dead Letter Queue to handle failures. By configuring a
 Lambda DLQ.
            context.Logger.LogError($"An error occurred");
            throw;
        }

    }
}
```

Go

SDK for Go V2

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 搭配 Lambda 來使用 SQS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package integration_sqs_to_lambda

import (
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(event events.SQSEvent) error {
 for _, record := range event.Records {
  err := processMessage(record)
  if err != nil {
   return err
  }
```

```
 }
 fmt.Println("done")
 return nil
}

func processMessage(record events.SQSMessage) error {
 fmt.Printf("Processed message %s\n", record.Body)
 // TODO: Do interesting work based on the new message
 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

適用於 Java 2.x 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Java 搭配 Lambda 來使用 SQS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SQSEvent;
import com.amazonaws.services.lambda.runtime.events.SQSEvent.SQSMessage;

public class Function implements RequestHandler<SQSEvent, Void> {
    @Override
    public Void handleRequest(SQSEvent sqsEvent, Context context) {
        for (SQSMessage msg : sqsEvent.getRecords()) {
            processMessage(msg, context);
        }
        context.getLogger().log("done");
```

```
        return null;
    }

    private void processMessage(SQSMessage msg, Context context) {
        try {
            context.getLogger().log("Processed message " + msg.getBody());

            // TODO: Do interesting work based on the new message

        } catch (Exception e) {
            context.getLogger().log("An error occurred");
            throw e;
        }

    }
}
```

JavaScript

適用於 JavaScript (v3) 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 SQS 事件與 Lambda 用 JavaScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const message of event.Records) {
    await processMessageAsync(message);
  }
  console.info("done");
};

async function processMessageAsync(message) {
  try {
    console.log(`Processed message ${message.body}`);
    // TODO: Do interesting work based on the new message
```

```
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

使用 SQS 事件與 Lambda 用 TypeScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SQSEvent, Context, SQSHandler, SQSRecord } from "aws-lambda";

export const functionHandler: SQSHandler = async (
  event: SQSEvent,
  context: Context
): Promise<void> => {
  for (const message of event.Records) {
    await processMessageAsync(message);
  }
  console.info("done");
};

async function processMessageAsync(message: SQSRecord): Promise<any> {
  try {
    console.log(`Processed message ${message.body}`);
    // TODO: Do interesting work based on the new message
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

PHP

適用於 PHP 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 PHP 與 Lambda 一起使用 SQS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\InvalidLambdaEvent;
use Bref\Event\Sqs\SqsEvent;
use Bref\Event\Sqs\SqsHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends SqsHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws InvalidLambdaEvent
     */
    public function handleSqs(SqsEvent $event, Context $context): void
    {
        foreach ($event->getRecords() as $record) {
            $body = $record->getBody();
            // TODO: Do interesting work based on the new message
        }
```

```
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

適用於 Python (Boto3) 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Python 搭配 Lambda 來使用 SQS 事件。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event, context):
    for message in event['Records']:
        process_message(message)
    print("done")

def process_message(message):
    try:
        print(f"Processed message {message['body']}")
        # TODO: Do interesting work based on the new message
    except Exception as err:
        print("An error occurred")
        raise err
```

Ruby

適用於 Ruby 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Ruby 搭配 Lambda 來使用 SQS 事件。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event:, context:)
  event['Records'].each do |message|
    process_message(message)
  end
  puts "done"
end

def process_message(message)
  begin
    puts "Processed message #{message['body']}"
    # TODO: Do interesting work based on the new message
  rescue StandardError => err
    puts "An error occurred"
    raise err
  end
end
```

Rust

適用於 Rust 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Rust 與 Lambda 一起使用 SQS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::sqs::SqsEvent;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<SqsEvent>) -> Result<(), Error> {
    event.payload.records.iter().for_each(|record| {
        // process the record
        tracing::info!("Message body: {}",
 record.body.as_deref().unwrap_or_default())
    });

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

若要建立 Node.js Lambda 函數

1. 建立專案的目錄，然後切換至該目錄。

```
mkdir sqs-tutorial
cd sqs-tutorial
```

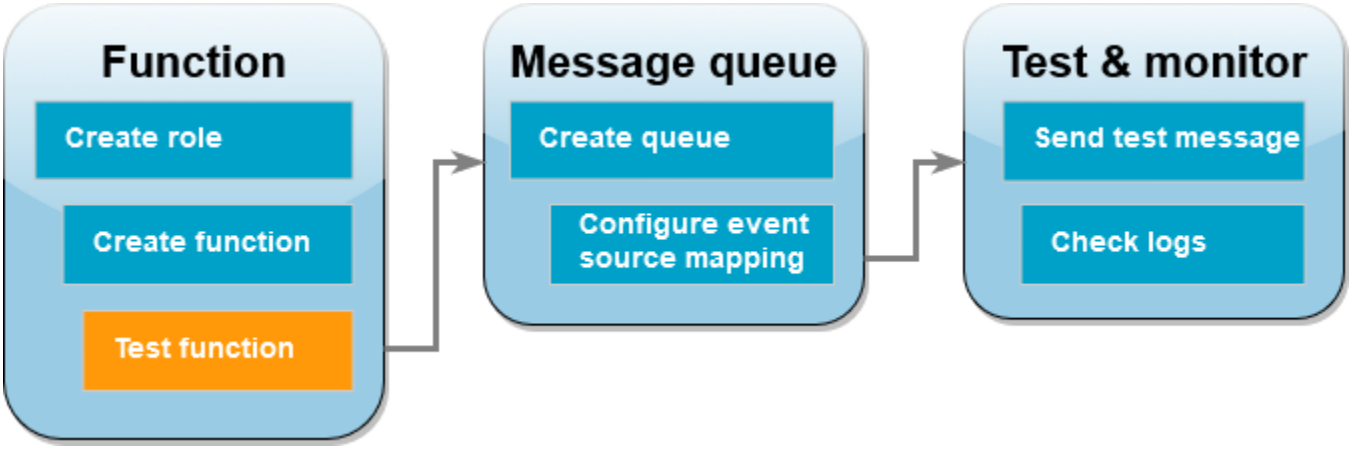
2. 將範例程 JavaScript 式碼複製到名為的新檔案中`index.js`。
3. 使用以下 `zip` 命令建立部署套件。

```
zip function.zip index.js
```

4. 使用 create-function AWS CLI 命令建立 Lambda 函數。針對 `role` 參數，輸入您之前建立的執行角色 ARN。

```
aws lambda create-function --function-name ProcessSQSRecord \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs18.x \
--role arn:aws:iam::111122223333:role/lambda-sqs-role
```

測試函數



使用`invoke` AWS CLI 命令和 Amazon SQS 事件範例，手動叫用 Lambda 函數。

使用範例事件調用 Lambda 函數

1. 將下面的 JSON 儲存為名為 `input.json` 的檔案。此 JSON 會模擬 Amazon SQS 可能傳送至 Lambda 函數的事件，其中 `"body"` 包含佇列中的實際訊息。在此範例中，訊息為 `"test"`。

   Example Amazon SQS 事件

   這是測試事件，您不需要變更訊息或帳號。

```
{
    "Records": [
        {
            "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
            "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",
            "body": "test",
            "attributes": {
```

```
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1545082649183",
                "SenderId": "AIDAIENQZJOLO23YVJ4VO",
                "ApproximateFirstReceiveTimestamp": "1545082649185"
            },
            "messageAttributes": {},
            "md5OfBody": "098f6bcd4621d373cade4e832627b4f6",
            "eventSource": "aws:sqs",
            "eventSourceARN": "arn:aws:sqs:us-east-1:111122223333:my-queue",
            "awsRegion": "us-east-1"
        }
    ]
}
```

2. 運行以下調用 AWS CLI 命令。此命令會在回應中傳回 CloudWatch 記錄。如需擷取日誌的詳細資訊，請參閱使用存取記錄 AWS CLI。

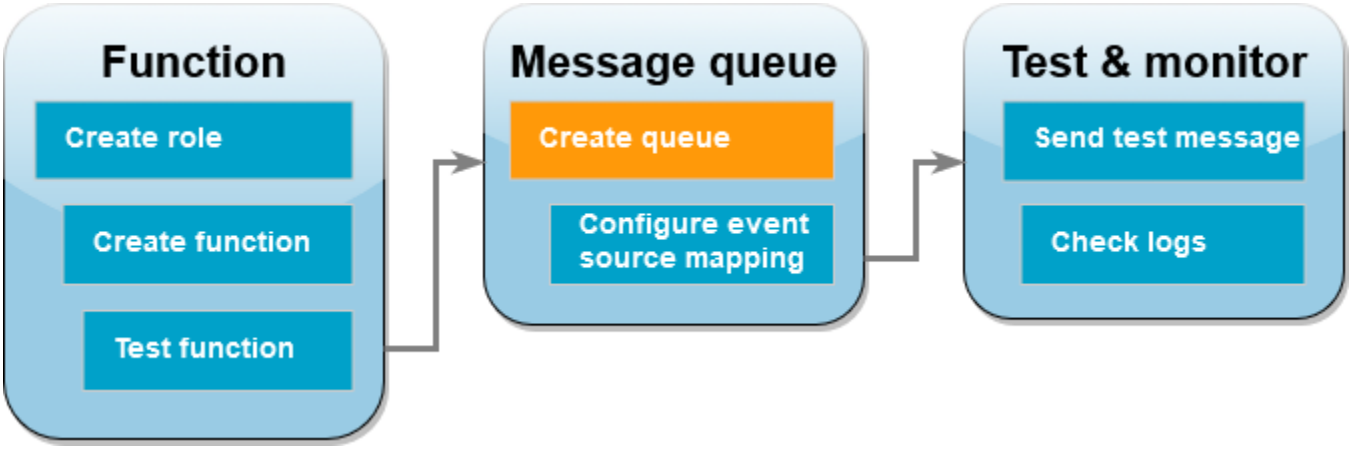
```
aws lambda invoke --function-name ProcessSQSRecord --payload file://input.json out
 --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

如果您使用的是 AWS CLI 版本 2，則需要此cli-binary-format選項。若要讓此成為預設的設定，請執行 `aws configure set cli-binary-format raw-in-base64-out`。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

3. 在回應中查找 `INFO` 日誌。這是 Lambda 函數記錄訊息內文的位置。您應該會看到類似以下內容的輸出：

```
2023-09-11T22:45:04.271Z 348529ce-2211-4222-9099-59d07d837b60 INFO Processed
 message test
2023-09-11T22:45:04.288Z 348529ce-2211-4222-9099-59d07d837b60 INFO done
```

建立 Amazon SQS 佇列



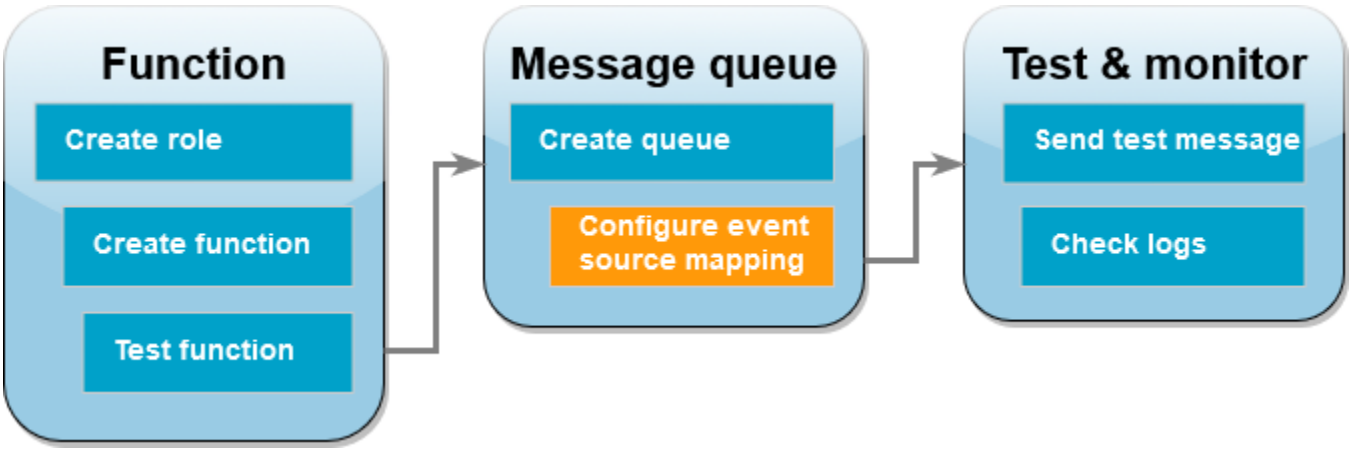
建立 Lambda 函數可用作事件來源的 Amazon SQS 佇列。

建立佇列

1. 開啟 Amazon SQS 主控台。
2. 選擇 建立佇列 。
3. 輸入佇列的名稱。將所有其他選項保留為預設值。
4. 選擇建立佇列。

建立佇列後，請記下其 ARN。在下個步驟中，將佇列與您的 Lambda 函數建立關聯時會需要用到它。

設定事件來源



透過建立事件來源映射，將 Amazon SQS 佇列連線至 Lambda 函數。事件來源映射會讀取 Amazon SQS 佇列，並在新增訊息時調用 Lambda 函數。

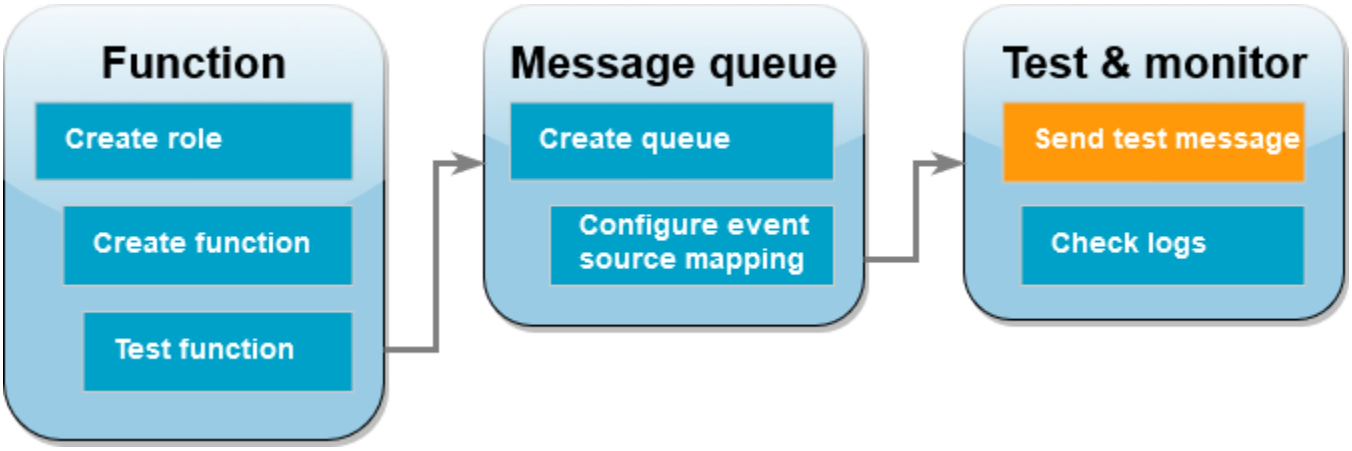
若要在 Amazon SQS 佇列和 Lambda 函數之間建立對應，請使用以下create-event-source-mapping AWS CLI 命令。範例：

```
aws lambda create-event-source-mapping --function-name ProcessSQSRecord  --batch-size
 10 \
--event-source-arn arn:aws:sqs:us-east-1:111122223333:my-queue
```

若要取得事件來源對映的清單，請使用list-event-source-mappings指令。範例：

```
aws lambda list-event-source-mappings --function-name ProcessSQSRecord
```

傳送測試訊息

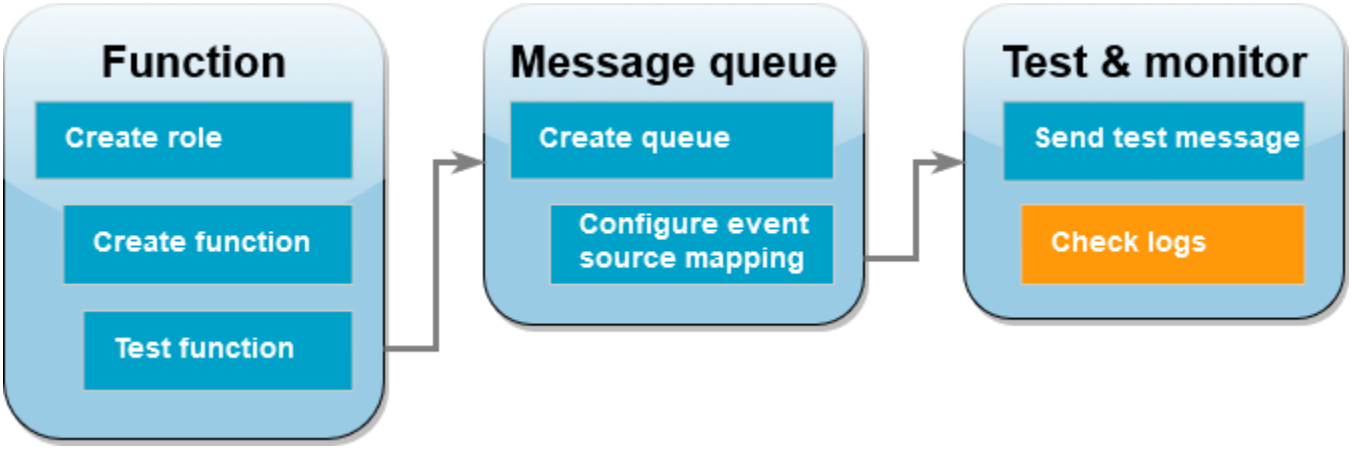


將 Amazon SQS 訊息傳送至 Lambda 函數

1. 開啟 Amazon SQS 主控台。
2. 選擇您稍早建立的佇列。
3. 選擇傳送及接收訊息。
4. 在訊息內文下，輸入測試訊息，例如「這是測試訊息」。
5. 選擇 傳送訊息 。

Lambda 輪詢佇列以查看是否有更新。當有新訊息時，Lambda 會使用佇列中的此新事件資料來叫用您的函數。如果函數處理常式傳回而無例外情況，則 Lambda 會認為訊息已成功處理，並開始讀取佇列中的新訊息。成功處理訊息之後，Lambda 從佇列中自動刪除它。如果處理常式擲出例外情況，Lambda 會認為訊息批次未成功處理，並且 Lambda 會調用具有相同訊息批次的函數。

檢查日 CloudWatch 誌



確認函數已處理訊息

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇 ProcessSQSRecord 函數。
3. 選擇 監控 。
4. 選擇 [檢視 CloudWatch 記錄]。
5. 在主 CloudWatch 控台中，選擇功能的 Log 串流。
6. 查找 INFO 日誌。這是 Lambda 函數記錄訊息內文的位置。應能看到您從 Amazon SQS 佇列傳送的訊息。範例：

```
2023-09-11T22:49:12.730Z b0c41e9c-0556-5a8b-af83-43e59efeec71 INFO Processed
 message this is a test message.
```

清除您的資源

除非您想要保留為此教學課程建立的資源，否則您現在便可刪除。刪除您不再使用的 AWS 資源，您可以避免不必要的 AWS 帳戶費用

刪除執行角色

1. 開啟 IAM 主控台中的 角色頁面 。
2. 選取您建立的執行角色。
3. 選擇 刪除 。
4. 在文字輸入欄位中輸入角色的名稱，然後選擇 刪除 。

若要刪除 Lambda 函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇您建立的函數。
3. 選擇 Actions (動作)、Delete (刪除)。
4. 在文字輸入欄位中輸入 **delete**，然後選擇 刪除 。

刪除 Amazon SQS 佇列

1. 登入 AWS Management Console 並開啟 Amazon SQS 主控台，網址為 https://console.aws.amazon.com/sqs/。
2. 選取您建立的佇列。
3. 選擇 刪除 。
4. 在文字輸入欄位中輸入 **confirm**。
5. 選擇 刪除 。

教學課程：使用跨帳戶 Amazon SQS 佇列做為事件來源

在本教學中，您會建立 Lambda 函數，以使用來自不同 AWS 帳戶之 Amazon Simple Queue Service (Amazon SQS) 佇列的訊息。本教學課程涉及兩個 AWS 帳戶：帳戶 A 是指包含 Lambda 函數的帳戶，而帳戶 B 指的是包含 Amazon SQS 佇列的帳戶。

必要條件

此教學課程假設您具備基本的 Lambda 操作知識並了解 Lambda 主控台。若您尚未了解，請遵循 使用主控台建立一個 Lambda 函數 中的指示，建立您的第一個 Lambda 函數。

若要完成下列步驟，您需要 AWS Command Line Interface (AWS CLI) 版本 2。命令和預期的輸出會列在不同的區塊中：

```
aws --version
```

您應該會看到下列輸出：

```
aws-cli/2.13.27 Python/3.11.6 Linux/4.14.328-248.540.amzn2.x86_64 exe/x86_64.amzn.2
```

對於長命令，逸出字元 (\) 用於將命令分割為多行。

在 Linux 和 macOS 上，使用您偏好的 shell 和套件軟體管理工具。

> **Note**
>
> 在 Windows 中，作業系統的內建終端不支援您常與 Lambda 搭配使用的某些 Bash CLI 命令 (例如 `zip`)。若要取得 Ubuntu 和 Bash 的 Windows 整合版本，請安裝適用於 Linux 的 Windows 子系統。本指南中的 CLI 命令範例使用 Linux 格式。如果您使用的是 Windows CLI，必須重新格式化包含內嵌 JSON 文件的命令。

建立執行角色 (帳戶 A)

在帳戶 A 中，建立一個執行角色，以授與您的函數存取所需資 AWS 源的權限。

若要建立執行角色

1. 在 AWS Identity and Access Management (IAM) 主控台中開啟「角色」頁面。
2. 選擇 建立角色。
3. 建立具備下列屬性的角色。
   - 信任實體 – AWS Lambda。
   - 權限 — AWSLambdaSQSQueueExecutionRole
   - 角色名稱 - **`cross-account-lambda-sqs-role`**。

該AWSLambdaSQSQueueExecutionRole政策具有函數從 Amazon SQS 讀取項目以及將日誌寫入 Amazon 日誌所需的許可。 CloudWatch

建立函數 (帳戶 A)

在 帳戶 A 中建立 Lambda 函數，它會處理 Amazon SQS 訊息。下列 Node.js 18 程式碼範例會將每個訊息寫入 CloudWatch 記錄檔中的記錄檔。

Example index.mjs

```
export const handler = async function(event, context) {
  event.Records.forEach(record => {
```

```
    const { body } = record;
    console.log(body);
  });
  return {};
}
```

建立函數

> **Note**
>
> 遵循這些步驟，在 Node.js 18 中建立函數。對於其他語言，步驟類似，但有些細節不同。

1. 將程式碼範例 儲存為名為 `index.mjs` 的檔案。
2. 建立部署套件。

```
zip function.zip index.mjs
```

3. 使用 `create-function` AWS Command Line Interface (AWS CLI) 指令建立函數。

```
aws lambda create-function --function-name CrossAccountSQSExample \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs18.x \
--role arn:aws:iam::<AccountA_ID>:role/cross-account-lambda-sqs-role
```

測試函數 (帳戶 A)

在帳戶 A 中，使用`invoke` AWS CLI 命令和 Amazon SQS 事件範例手動測試 Lambda 函數。

如果處理常式均正常傳回而無例外情況，Lambda 會認為訊息已成功處理，並開始讀取佇列中的新訊息。成功處理訊息之後，Lambda 從佇列中自動刪除它。如果處理常式擲出例外情況，Lambda 會認為訊息批次未成功處理，並且 Lambda 會調用具有相同訊息批次的函數。

1. 將下面的 JSON 儲存為名為 `input.txt` 的檔案。

```
{
    "Records": [
        {
            "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
            "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",
```

```
            "body": "test",
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1545082649183",
                "SenderId": "AIDAIENQZJOLO23YVJ4VO",
                "ApproximateFirstReceiveTimestamp": "1545082649185"
            },
            "messageAttributes": {},
            "md5OfBody": "098f6bcd4621d373cade4e832627b4f6",
            "eventSource": "aws:sqs",
            "eventSourceARN": "arn:aws:sqs:us-east-1:111122223333:example-queue",
            "awsRegion": "us-east-1"
        }
    ]
}
```

上述 JSON 會模擬 Amazon SQS 可能傳送至 Lambda 函數的事件，其中 "body" 包含佇列中的實際訊息。

2. 執行下列 invoke AWS CLI 命令。

```
aws lambda invoke --function-name CrossAccountSQSExample \
--cli-binary-format raw-in-base64-out \
--payload file://input.txt outputfile.txt
```

如果您使用的是 AWS CLI 版本 2，則需要此cli-binary-format選項。若要讓此成為預設的設定，請執行 aws configure set cli-binary-format raw-in-base64-out。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

3. 在檔案 outputfile.txt 中確認輸出。

建立 Amazon SQS 佇列 (帳戶 B)

在 帳戶 B 中，建立 帳戶 A 中 Lambda 函數可用作事件來源的 Amazon SQS 佇列。

建立佇列

1. 開啟 Amazon SQS 主控台。
2. 選擇 建立佇列 。
3. 建立具備下列屬性的佇列。

- Type (類型) - Standard (標準)
- 名稱 — LambdaCrossAccountQueue
- Configuration (組態) - 保留預設設定。
- 存取政策 - 選擇 進階 。貼入下列 JSON 政策中：

```
{
   "Version": "2012-10-17",
   "Id": "Queue1_Policy_UUID",
   "Statement": [{
      "Sid":"Queue1_AllActions",
      "Effect": "Allow",
      "Principal": {
         "AWS": [
            "arn:aws:iam::<AccountA_ID>:role/cross-account-lambda-sqs-role"
         ]
      },
      "Action": "sqs:*",
      "Resource": "arn:aws:sqs:us-east-1:<AccountB_ID>:LambdaCrossAccountQueue"
    }
  ]
}
```

此政策許可 帳戶 A 中的 Lambda 執行角色取用此 Amazon SQS 佇列中的訊息。

4. 建立佇列後，記錄其 Amazon Resource Name (ARN)。在下個步驟中，將佇列與您的 Lambda 函數建立關聯時會需要用到它。

設定事件來源 (帳戶 A)

在帳戶 A 中，執行下列`create-event-source-mapping` AWS CLI 命令，在帳戶 B 中的 Amazon SQS 佇列和 Lambda 函數之間建立事件來源對應。

```
aws lambda create-event-source-mapping --function-name CrossAccountSQSExample --batch-
size 10 \
--event-source-arn arn:aws:sqs:us-east-1:<AccountB_ID>:LambdaCrossAccountQueue
```

若要取得事件來源映射的清單，請執行下列命令。

```
aws lambda list-event-source-mappings --function-name CrossAccountSQSExample \
```

```
--event-source-arn arn:aws:sqs:us-east-1:<AccountB_ID>:LambdaCrossAccountQueue
```

測試設定

現在您可以測試設定，如下所示：

1. 在 帳戶 B 中，開啟 Amazon SQS 主控台。
2. 選擇 LambdaCrossAccountQueue，您之前創建的。
3. 選擇傳送及接收訊息。
4. 在 Message body (訊息主體) 中，輸入測試訊息。
5. 選擇 傳送訊息 。

帳戶 A 中您的 Lambda 函數應該會收到訊息。Lambda 會繼續輪詢佇列是否有更新。當有新訊息時，Lambda 會使用佇列中的此新事件資料來調用您的函數。您的函數運行並在 Amazon 創建日誌 CloudWatch。您可以在CloudWatch 控制台中查看日誌。

清除您的資源

除非您想要保留為此教學課程建立的資源，否則您現在便可刪除。刪除您不再使用的 AWS 資源，您可以避免不必要的費用 AWS 帳戶。

在 帳戶 A 中，清除您的執行角色和 Lambda 函數。

刪除執行角色

1. 開啟 IAM 主控台中的 角色頁面 。
2. 選取您建立的執行角色。
3. 選擇 刪除 。
4. 在文字輸入欄位中輸入角色的名稱，然後選擇 刪除 。

若要刪除 Lambda 函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇您建立的函數。
3. 選擇 Actions (動作)、Delete (刪除)。
4. 在文字輸入欄位中輸入 **`delete`**，然後選擇 刪除 。

在 帳戶 B 中，清除 Amazon SQS 佇列。

刪除 Amazon SQS 佇列

1. 登入 AWS Management Console 並開啟 Amazon SQS 主控台，網址為 https://console.aws.amazon.com/sqs/。
2. 選取您建立的佇列。
3. 選擇 刪除 。
4. 在文字輸入欄位中輸入 **confirm**。
5. 選擇 刪除 。

使用 Lambda 處理 Amazon DocumentDB 事件

您可以透過將 Amazon DocumentDB 叢集設定為事件來源，使用 Lambda 函數處理 Amazon DocumentDB (with MongoDB compatibility) 變更串流中的事件。接著，您可以在每次使用 Amazon DocumentDB 叢集變更資料時，調用 Lambda 函數來自動執行事件驅動的工作負載。

> **Note**
>
> Lambda 僅支援 Amazon DocumentDB 4.0 和 5.0 版。Lambda 不支援 3.6 版。
> 此外，針對事件來源映射，Lambda 僅支援執行個體型叢集和區域叢集。Lambda 不支援 彈性叢集 或 全域叢集。使用 Lambda 做為用戶端連線至 Amazon DocumentDB 時不適用此限制。Lambda 可以連線至所有叢集類型來執行 CRUD 操作。

Lambda 會依照抵達的順序處理來自 Amazon DocumentDB 變更串流的事件。因此，函數一次只能處理來自 DocumentDB 的一個並行調用。若要監控函數，您可以追蹤其並行指標。

> **Warning**
>
> Lambda 事件來源對應至少處理每個事件一次，並且可能會重複處理記錄。為了避免與重複事件相關的潛在問題，我們強烈建議您將函數代碼設為冪等。若要深入了解，請參閱 AWS 知識中心如何讓 Lambda 函數具有冪等性。

主題

- Amazon DocumentDB 事件範例
- 先決條件和許可
- 網路組態
- 建立 Amazon DocumentDB 事件來源映射 (主控台)
- 建立 Amazon DocumentDB 事件來源映射 (SDK 或 CLI)
- 輪詢和串流開始位置
- 監控 Amazon DocumentDB 事件來源
- 教學課程：使 AWS Lambda 用 Amazon DocumentDB 串流

Amazon DocumentDB 事件範例


```
{
    "eventSourceArn": "arn:aws:rds:us-
east-1:123456789012:cluster:canaryclusterb2a659a2-qo5tcmqkcl03",
    "events": [
        {
            "event": {
                "_id": {
                    "_data": "0163eeb6e7000000090100000009000041e1"
                },
                "clusterTime": {
                    "$timestamp": {
                        "t": 1676588775,
                        "i": 9
                    }
                },
                "documentKey": {
                    "_id": {
                        "$oid": "63eeb6e7d418cd98afb1c1d7"
                    }
                },
                "fullDocument": {
                    "_id": {
                        "$oid": "63eeb6e7d418cd98afb1c1d7"
                    },
                    "anyField": "sampleValue"
                },
                "ns": {
                    "db": "test_database",
```

```
                    "coll": "test_collection"
                },
                "operationType": "insert"
            }
        }
    ],
    "eventSource": "aws:docdb"
}
```

如需有關此範例中事件及其形狀的詳細資訊，請參閱 MongoDB 文件網站上的變更事件。

先決條件和許可

將 Amazon DocumentDB 作為 Lambda 函數的事件來源使用前，請留意以下先決條件。您必須：

- 讓現有的 Amazon DocumentDB 叢集 AWS 帳戶 與 AWS 區域 您的函數相同。如果您沒有現有叢集，則可以按照《Amazon DocumentDB 開發人員指南》中開始使用 Amazon DocumentDB所述步驟建立叢集。或者，教學課程：使 AWS Lambda 用 Amazon DocumentDB 串流 中的第一組步驟會引導您建立包含所有必要先決條件的 DocumentDB 叢集。
- 允許 Lambda 存取與您 Amazon DocumentDB 叢集相關聯的 Amazon Virtual Private Cloud (Amazon VPC) 資源。如需詳細資訊，請參閱 網路組態。
- 在您的 Amazon DocumentDB 叢集上啟用 TLS。這是預設設定。若停用 TLS，Lambda 將無法與您的叢集進行通訊。
- 啟用 Amazon DocumentDB 叢集上的變更串流。如需詳細資訊，請參閱《Amazon DocumentDB 開發人員指南》中的透過 Amazon DocumentDB 使用變更串流 。
- 為 Lambda 提供憑證以存取您的 Amazon DocumentDB 叢集。設定事件來源時，請提供包含存取叢集所需驗證詳細資料 (使用者名稱和密碼) 的 AWS Secrets Manager 金鑰。若要在設定期間提供此金鑰，請執行下列其中一項操作：
  - 如果您要使用 Lambda 主控台進行設定，請在 Secrets Manager 金鑰欄位中提供此金鑰。
  - 如果您使用 AWS Command Line Interface （ AWS CLI ）進行設置，請在`source-access-configurations`選項中提供此鍵。您可以連同 `create-event-source-mapping` 或 `update-event-source-mapping` 命令包括此選項。例如：

    ```
    aws lambda create-event-source-mapping \
        ...
        --source-access-configurations
     '[{"Type":"BASIC_AUTH","URI":"arn:aws:secretsmanager:us-
    west-2:123456789012:secret:DocDBSecret-AbC4E6"}]' \
    ```

```
    ...
```

- 您必須向 Lambda 授與許可，才能管理與 Amazon DocumentDB 串流相關的資源。將下列許可手動新增到函數的 執行角色：
  - rds:DescribeDBClusters
  - RDS: 描述 ClusterParameters
  - RDS: 描述 SubnetGroups
  - ec2：CreateNetwork接口
  - ec2：DescribeNetwork接口
  - ec2：DescribeVpcs
  - ec2：DeleteNetwork接口
  - ec2：DescribeSubnets
  - ec2：DescribeSecurity群組
  - kms:Decrypt
  - 秘密經理:價值 GetSecret
- 傳送至 Lambda 的 Amazon DocumentDB 變更串流事件大小請勿超過 6 MB。Lambda 支援的承載大小上限為 6MB。如果變更串流嘗試向 Lambda 傳送大於 6MB 的事件，Lambda 會捨棄訊息並發出 `OversizedRecordCount` 指標。Lambda 會盡力發送所有指標。

> **Note**
>
> 雖然 Lambda 函數的逾時上限通常為 15 分鐘，但 Amazon MSK、自我管理的 Apache Kafka、Amazon DocumentDB 以及 Amazon MQ for ActiveMQ 和 Amazon MQ for RabbitMQ 的事件來源映射只支援 14 分鐘逾時限制上限的函數。此限制條件可確保事件來源映射能夠正確處理函數錯誤和重試。

網路組態

若要讓 Lambda 使用您的 Amazon DocumentDB 叢集做為事件來源，需要存取叢集所在的 Amazon VPC。我們建議您為 Lambda 部署 AWS PrivateLink VPC 人雲端端點，以存取您的 VPC。為 Lambda 部署 VPC 人雲端端點，如果叢集使用驗證，也會為 Secrets Manager 部署 VPC 人雲端端點。

或者，確保與您 Amazon DocumentDB 叢集相關聯的 VPC 的每個公有子網路都含有一個 NAT 閘道。如需詳細資訊，請參閱 the section called “VPC 功能的網際網路存取”。

若使用 VPC 端點，您還必須將它們設定為啟用私有 DNS 名稱。

當您為 Amazon DocumentDB 叢集建立事件來源對應時，Lambda 會檢查叢集 VPC 的子網路和安全群組是否已存在彈性網路界面 (ENI)。如果 Lambda 找到現有的 ENI，它會嘗試重複使用它們。否則，Lambda 會建立新的 ENI 以連接至事件來源並叫用您的函數。

> **Note**
>
> Lambda 函數一律在 Lambda 服務擁有的 VPC 內執行。這些 VPC 由服務自動維護，客戶看不到。您也可以將您的功能連接到 Amazon VPC。在任何一種情況下，函數的 VPC 配置都不會影響事件源映射。只有事件來源 VPC 的組態才會決定 Lambda 連線至事件來源的方式。

VPC 安全群組規則

使用下列規則 (至少) 為包含叢集的 Amazon VPC 設定安全群組：

- 輸入規則 — 允許針對為您的事件來源指定的安全群組的 Amazon DocumentDB 叢集連接埠上的所有流量。Amazon DocumentDB 默認使用端口 27017。
- 傳出規則：針對所有目的地，允許連接埠 443 上的所有流量。允許 Amazon 文件資料庫叢集連接埠上的所有流量。Amazon DocumentDB 默認使用端口 27017。
- 如果您使用的是 VPC 端點而不是 NAT 閘道，則與 VPC 端點相關聯的安全群組必須允許連接埠 443 上所有來自事件來源安全群組的輸入流量。

使用 VPC 端點

當您使用 VPC 端點時，會使用 ENI 透過這些端點路由呼叫函數的 API 呼叫。Lambda 服務主體需要呼叫`lambda:InvokeFunction`使用這些 ENI 的任何函數。

根據預設，VPC 端點具有開放的 IAM 政策。最佳做法是將這些原則限制為僅允許特定主參與者使用該端點執行所需的動作。為了確保您的事件來源對應能夠叫用 Lambda 函數，VPC 端點政策必須允許 Lambda 服務原則呼叫`lambda:InvokeFunction`。將 VPC 端點原則限制為僅允許來自組織內的 API 呼叫，可防止事件來源對應正常運作。

下列範例 VPC 端點原則示範如何授與 Lambda 端點所需的存取權。

Example VPC 私人雲端端點政策

```
{
```

```
    "Statement": [
        {
            "Action": "lambda:InvokeFunction",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
}
```

如果您的 Amazon DocumentDB 叢集使用身份驗證，您也可以限制 Secrets Manager 端點的虛擬私人雲端端點政策。若要呼叫機 Secrets Manager API，Lambda 會使用您的函數角色，而不是使用 Lambda 服務主體。下列範例顯示 Secrets Manager 端點策略。

Example VPC 端點原則-Secrets Manager 端點

```
{
    "Statement": [
        {
            "Action": "secretsmanager:GetSecretValue",
            "Effect": "Allow",
            "Principal": {
                "AWS": [
                    "customer_function_execution_role_arn"
                ]
            },
            "Resource": "customer_secret_arn"
        }
    ]
}
```

建立 Amazon DocumentDB 事件來源映射 (主控台)

若要讓 Lambda 函數從 Amazon DocumentDB 叢集的變更串流中讀取，請建立 DocumentDB 事件來源映射。本節會說明如何透過 Lambda 主控台執行這項操作。如需 AWS SDK 和 AWS CLI 指示，請參閱the section called “建立 Amazon DocumentDB 事件來源映射 (SDK 或 CLI)”。

建立 Amazon DocumentDB 事件來源映射 (主控台)

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇函數的名稱。
3. 在 函式概觀 下，選擇 新增觸發條件 。
4. 在觸發條件組態底下的下拉式清單中，選擇 DocumentDB。
5. 設定需要的選項，然後選擇 新增 。

Lambda 支援以下 Amazon DocumentDB 事件來源的選項：

- DocumentDB 叢集：選取 Amazon DocumentDB 叢集。
- 啟用觸發條件：選擇您是否要立即啟用觸發條件。若勾選此核取方塊，您的函數會在建立事件來源映射時立即開始接收來自指定 Amazon DocumentDB 變更串流的流量。建議您取消勾選此核取方塊，以便在停用狀態下建立事件來源映射來進行測試。完成建立後，您隨時可以啟動事件來源映射。
- 資料庫名稱： 輸入叢集內要使用的資料庫名稱。
- (選用) 集合名稱：輸入資料庫內要使用的集合名稱。如果您未指定集合，Lambda 會偵聽資料庫中每個集合中的所有事件。
- 批次大小： 設定單一批次中要擷取的訊息數目上限 (最高 10,000)。預設批次大小為 100。
- 開始位置： 選擇串流中要從中開始讀取記錄的位置。
  - 最新： 僅處理已新增到串流的新記錄。Lambda 建立完事件來源後，您的函數才會開始處理記錄。這表示系統可能會捨棄部分記錄，直到您的事件來源成功建立為止。
  - 水平修剪 - 處理所有在串流中的記錄。Lambda 會透過叢集的日誌保留持續時間來決定開始讀取事件的時間點。具體來說，Lambda 會從 `current_time - log_retention_duration` 開始進行讀取。變更串流必須在此時間戳記前處於作用中狀態，Lambda 才能正確讀取所有事件。
  - At timestamp (在時間戳記為) - 從特定時間開始處理記錄。變更串流必須在指定時間戳記前處於作用中狀態，Lambda 才能正確讀取所有事件。
- 身分驗證： 選擇在叢集中存取中介裝置的身分驗證方式。
  - BASIC_AUTH： 使用基本身分驗證下，您必須提供含有憑證的 Secrets Manager 金鑰，才能存取叢集。
- Secrets Manager 金鑰：選擇含有存取 Amazon DocumentDB 叢集所需身分驗證詳細資料 (使用者名稱和密碼) 的 Secrets Manager 金鑰。
- (選用) 批次間隔：在調用函數前收集記錄的最長時間 (秒)，上限為 300 秒。

- (選用) 完整文件組態：文件更新作業方面，請選擇要傳送至串流的內容。預設值為 `Default`，這表示 Amazon DocumentDB 針對每個變更串流事件僅會傳送說明所做變更的差異。如需有關此欄位的詳細資訊，請參閱FullDocument在 MongoDB API 文件中。
  - 預設： Lambda 僅會傳送說明所做變更的部分文件。
  - UpdateLookup-Lambda 發送描述更改的增量值以及整個文檔的副本。

建立 Amazon DocumentDB 事件來源映射 (SDK 或 CLI)

若要使用 AWS SDK 來建立或管理 Amazon DocumentDB 事件來源映射，您可以使用下列 API 操作：

- CreateEventSourceMapping
- ListEventSourceMappings
- GetEventSourceMapping
- UpdateEventSourceMapping
- DeleteEventSourceMapping

若要使用建立事件來源對映 AWS CLI，請使用`create-event-source-mapping`指令。以下範例使用此命令，將名為 `my-function` 的函數映射至 Amazon DocumentDB 變更串流。事件來源是由 Amazon Resource Name (ARN) 指定，批次大小為 500，且開始時間為 Unix 時間的時間戳記。此命令也會指定 Lambda 用來連線至 Amazon DocumentDB 的 Secrets Manager 金鑰。此外也包括 `document-db-event-source-config` 參數，這個參數指定了要從哪個資料庫和集合讀取。

```
aws lambda create-event-source-mapping --function-name my-function \
    --event-source-arn arn:aws:rds:us-west-2:123456789012:cluster:privatecluster7de2-
epzcyvu4pjoy
    --batch-size 500 \
    --starting-position AT_TIMESTAMP \
    --starting-position-timestamp 1541139109 \
    --source-access-configurations
 '[{"Type":"BASIC_AUTH","URI":"arn:aws:secretsmanager:us-
east-1:123456789012:secret:DocDBSecret-BAtjxi"}]' \
    --document-db-event-source-config '{"DatabaseName":"test_database",
 "CollectionName": "test_collection"}' \
```

您應該會看到輸出，如下所示：

```
{
```

```
    "UUID": "2b733gdc-8ac3-cdf5-af3a-1827b3b11284",
    "BatchSize": 500,
    "DocumentDBEventSourceConfig": {
        "CollectionName": "test_collection",
        "DatabaseName": "test_database",
        "FullDocument": "Default"
    },
    "MaximumBatchingWindowInSeconds": 0,
    "EventSourceArn": "arn:aws:rds:us-west-2:123456789012:cluster:privatecluster7de2-
epzcyvu4pjoy",
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "LastModified": 1541348195.412,
    "LastProcessingResult": "No records processed",
    "State": "Creating",
    "StateTransitionReason": "User action"
}
```

完成建立後，您可以使用 update-event-source-mapping 命令來更新 Amazon DocumentDB 事件來源的設定。以下命令將批次大小更新為 1,000，並將批次間格更新為 10 秒。對於此命令，您必須擁有事件來源映射的 UUID (可使用 list-event-source-mapping 命令或 Lambda 主控台擷取)。

```
aws lambda update-event-source-mapping --function-name my-function \
    --uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
    --batch-size 1000 \
    --batch-window 10
```

您應該會看到類似以下內容的輸出：

```
{
    "UUID": "2b733gdc-8ac3-cdf5-af3a-1827b3b11284",
    "BatchSize": 500,
    "DocumentDBEventSourceConfig": {
        "CollectionName": "test_collection",
        "DatabaseName": "test_database",
        "FullDocument": "Default"
    },
    "MaximumBatchingWindowInSeconds": 0,
    "EventSourceArn": "arn:aws:rds:us-west-2:123456789012:cluster:privatecluster7de2-
epzcyvu4pjoy",
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "LastModified": 1541359182.919,
    "LastProcessingResult": "OK",
```

```
    "State": "Updating",
    "StateTransitionReason": "User action"
}
```

Lambda 會以非同步方式更新設定，因此處理完成之前您可能無法在輸出中看到這些變更。若要檢視事件來源映射的目前設定，請使用 get-event-source-mapping 命令。

```
aws lambda get-event-source-mapping --uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b
```

您應該會看到類似以下內容的輸出：

```
{
    "UUID": "2b733gdc-8ac3-cdf5-af3a-1827b3b11284",
    "DocumentDBEventSourceConfig": {
        "CollectionName": "test_collection",
        "DatabaseName": "test_database",
        "FullDocument": "Default"
    },
    "BatchSize": 1000,
    "MaximumBatchingWindowInSeconds": 10,
    "EventSourceArn": "arn:aws:rds:us-west-2:123456789012:cluster:privatecluster7de2-
epzcyvu4pjoy",
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "LastModified": 1541359182.919,
    "LastProcessingResult": "OK",
    "State": "Enabled",
    "StateTransitionReason": "User action"
}
```

若要刪除 Amazon DocumentDB 事件來源映射，請使用 delete-event-source-mapping 命令。

```
aws lambda delete-event-source-mapping \
    --uuid 2b733gdc-8ac3-cdf5-af3a-1827b3b11284
```

輪詢和串流開始位置

請注意，建立和更新事件來源映射期間的串流輪詢最終會一致。

- 在建立事件來源映射期間，從串流開始輪詢事件可能需要幾分鐘時間。
- 在更新事件來源映射期間，從串流停止並重新開始輪詢事件可能需要幾分鐘時間。

這種行為表示如果您指定 LATEST 當作串流的開始位置，事件來源映射可能會在建立或更新期間遺漏事件。若要確保沒有遺漏任何事件，請將串流開始位置指定為 TRIM_HORIZON 或 AT_TIMESTAMP。

監控 Amazon DocumentDB 事件來源

為協助您監控 Amazon DocumentDB 事件來源，Lambda 會在您的函數處理完一批記錄後發出 IteratorAge 指標。迭代器存留期是最近事件和目前時間戳記之間的差距。基本上，IteratorAge 指標會指出批次中最後處理的記錄經過了多久的時間。如果函數目前正在處理新的事件，可使用迭代器存留期來預估記錄新增與函數實際處理之間的延遲。從 IteratorAge 的增加趨勢可看出您函數的問題。如需詳細資訊，請參閱 使用 Lambda 函數指標。

Amazon DocumentDB 變更串流並未最佳化，無法處理事件之間的大幅時間差距。如果您的 Amazon DocumentDB 事件來源在一段時間內沒有收到任何事件，Lambda 可能會停用事件來源對應。根據叢集大小和其他工作負載，此時段的長度可能從數週到幾個月不等。

Lambda 支援的承載上限為 6MB。不過，Amazon DocumentDB 變更串流事件的大小可達 16MB。如果變更串流嘗試向 Lambda 傳送大於 6MB 的變更串流事件，Lambda 會捨棄訊息並發出 OversizedRecordCount 指標。Lambda 會盡力發送所有指標。

教學課程：使 AWS Lambda 用 Amazon DocumentDB 串流

在本教學課程中，您將建立一個基礎 Lambda 函數，它會從 Amazon DocumentDB (with MongoDB compatibility) 變更串流中取用事件。完成本教學課程需逐一進行以下階段：

- 設定您的 Amazon DocumentDB 叢集、連線到叢集，然後在叢集上啟用變更串流。
- 建立 Lambda 函數，並將 Amazon DocumentDB 叢集設定為函數的事件來源。
- 透過將項目插入您的亞馬遜資料庫來測試 end-to-end 設定。

主題

- 必要條件
- 創建環 AWS Cloud9 境
- 建立 EC2 安全群組
- 建立 DocumentDB 叢集
- 在 Secrets Manager 中建立密碼
- 安裝 mongo Shell
- 連線至 DocumentDB 叢集

- 啟用變更串流
- 建立介面 VPC 端點
- 建立執行角色
- 建立 Lambda 函式
- 建立 Lambda 事件來源映射
- 測試函數 - 手動調用
- 測試函數 - 插入記錄
- 測試函數 - 更新記錄
- 測試函數 - 刪除記錄
- 清除您的資源

必要條件

註冊一個 AWS 帳戶

如果您沒有 AWS 帳戶，請完成以下步驟來建立一個。

若要註冊成為 AWS 帳戶

1. 開啟 https://portal.aws.amazon.com/billing/signup。
2. 請遵循線上指示進行。

   部分註冊程序需接收來電，並在電話鍵盤輸入驗證碼。

   當您註冊一個時 AWS 帳戶，將創建AWS 帳戶根使用者一個。根使用者有權存取該帳戶中的所有 AWS 服務 和資源。安全性最佳做法是將管理存取權指派給使用者，並僅使用 root 使用者來執行需要 root 使用者存取權的工作。

AWS 註冊過程完成後，會向您發送確認電子郵件。您可以隨時登錄 https://aws.amazon.com/ 並選擇我的帳戶，以檢視您目前的帳戶活動並管理帳戶。

建立具有管理權限的使用者

註冊後，請保護您的 AWS 帳戶 AWS 帳戶根使用者 AWS IAM Identity Center、啟用和建立系統管理使用者，這樣您就不會將 root 使用者用於日常工作。

保護您的 AWS 帳戶根使用者

1. 選擇 Root 使用者並輸入您的 AWS 帳戶 電子郵件地址，以帳戶擁有者身分登入。AWS Management Console在下一頁中，輸入您的密碼。

   如需使用根使用者登入的說明，請參閱 AWS 登入 使用者指南中的以根使用者身分登入。

2. 若要在您的根使用者帳戶上啟用多重要素驗證 (MFA)。

   如需指示，請參閱《IAM 使用者指南》中的為 AWS 帳戶 根使用者啟用虛擬 MFA 裝置 (主控台)。

建立具有管理權限的使用者

1. 啟用 IAM Identity Center。

   如需指示，請參閱 AWS IAM Identity Center 使用者指南中的啟用 AWS IAM Identity Center。

2. 在 IAM 身分中心中，將管理存取權授予使用者。

   若要取得有關使用 IAM Identity Center 目錄 做為身分識別來源的自學課程，請參閱《使用指南》IAM Identity Center 目錄中的「以預設值設定使用AWS IAM Identity Center 者存取」。

以具有管理權限的使用者身分登入

- 若要使用您的 IAM Identity Center 使用者簽署，請使用建立 IAM Identity Center 使用者時傳送至您電子郵件地址的簽署 URL。

  如需使用 IAM 身分中心使用者登入的說明，請參閱使用AWS 登入 者指南中的登入 AWS 存取入口網站。

指派存取權給其他使用者

1. 在 IAM 身分中心中，建立遵循套用最低權限許可的最佳做法的權限集。

   如需指示，請參閱《AWS IAM Identity Center 使用指南》中的「建立權限集」。

2. 將使用者指派給群組，然後將單一登入存取權指派給群組。

   如需指示，請參閱《AWS IAM Identity Center 使用指南》中的「新增群組」。

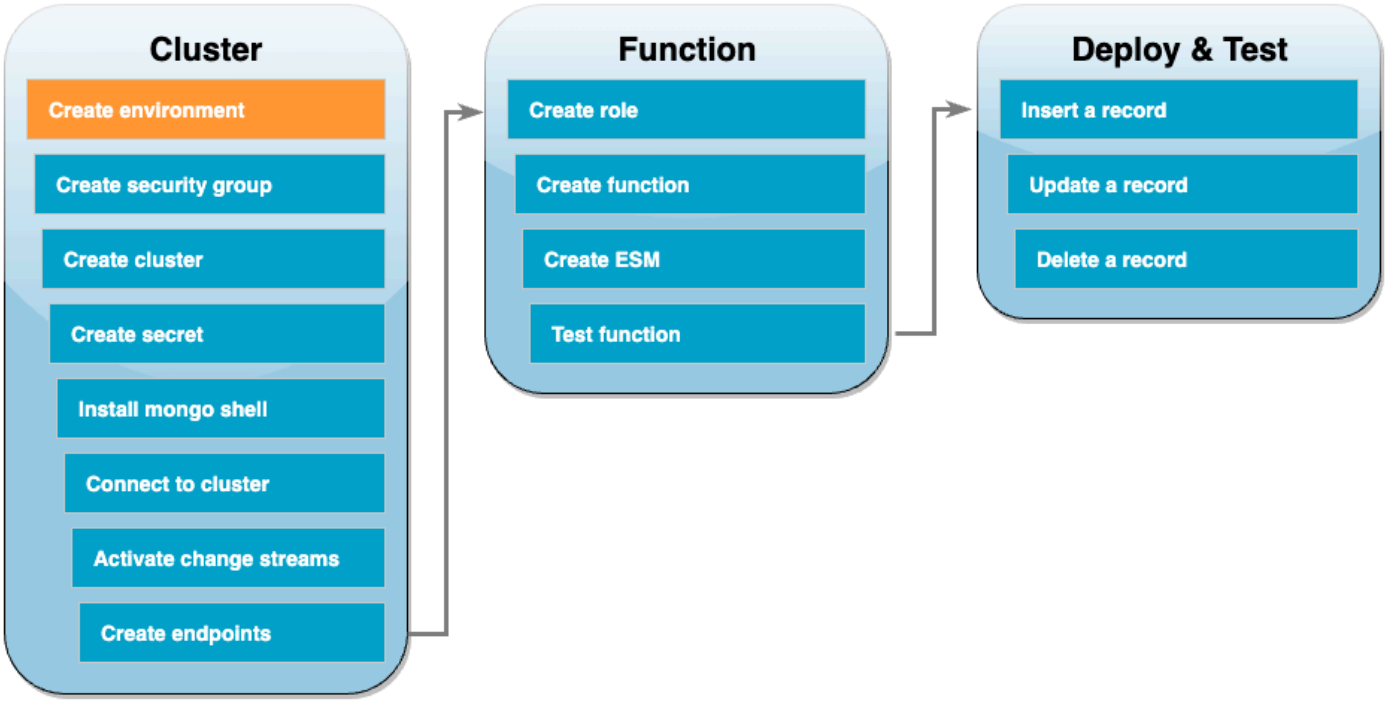
安裝 AWS Command Line Interface

如果您尚未安裝 AWS Command Line Interface，請按照安裝或更新最新版本的步驟進 AWS CLI行安裝。

本教學課程需使用命令列終端機或 Shell 來執行命令。在 Linux 和 macOS 中，使用您偏好的 Shell 和套件管理工具。

> **Note**
>
> 在 Windows 中，作業系統的內建終端不支援您常與 Lambda 搭配使用的某些 Bash CLI 命令 (例如 `zip`)。若要取得 Ubuntu 和 Bash 的 Windows 整合版本，請安裝適用於 Linux 的 Windows 子系統。

創建環 AWS Cloud9 境



在建立 Lambda 函數之前，您需要建立並設定 Amazon DocumentDB 叢集。本教學課程中設定叢集的步驟是以 Amazon DocumentDB 入門中的程序為基礎。

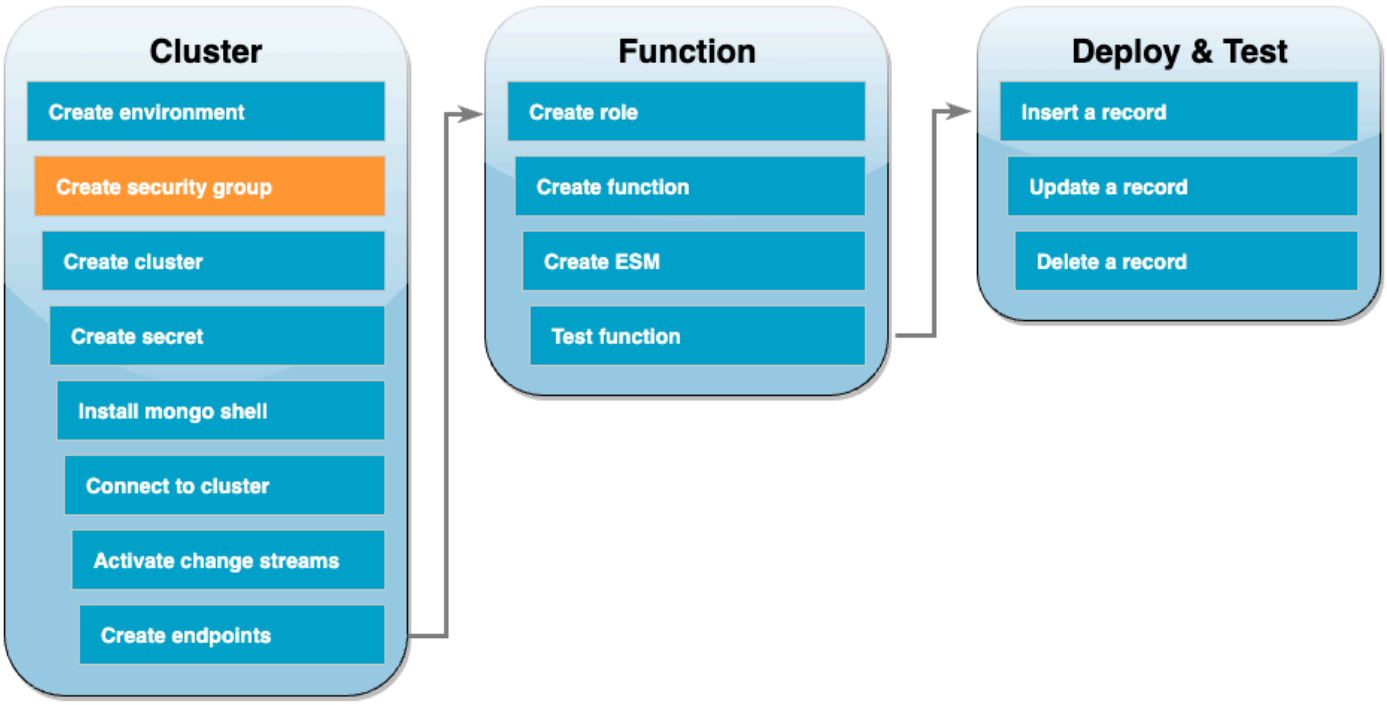
> **Note**
>
> 如果您已經設定 Amazon DocumentDB 叢集，則請務必啟用變更串流並建立必要的介面 VPC 端點。然後，可以直接跳到函數建立步驟。

首先，創建一個 AWS Cloud9 環境。您將在本教學課程中使用此環境來連接和查詢 DocumentDB 叢集。

建立 AWS Cloud9 環境的步驟

1. 開啟 Cloud9 主控台，並選擇建立環境。
2. 使用下列組態建立環境：
   - 在詳細資訊下：
     - 名稱 – `DocumentDBCloud9Environment`
     - 環境類型 – 新 EC2 執行個體
   - 在新 EC2 執行個體下：
     - 執行個體類型 – `t2.micro` (1 GiB RAM + 1 vCPU)
     - 平台 – Amazon Linux 2
     - 逾時 – 30 分鐘
   - 在網路設定下：
     - 連線 — AWS Systems Manager (超音波馬達)
     - 展開 VPC 設定下拉式選單。
     - Amazon 虛擬私有雲端 (VPC) – 選擇您的預設 VPC。
     - 子網路 – 無偏好設定
   - 請保留所有其他預設設定。
3. 選擇建立。佈建新 AWS Cloud9 環境可能需要幾分鐘的時間。

建立 EC2 安全群組



接下來，使用允許 DocumentDB 叢集和 Cloud9 環境之間的流量之規則建立 EC2 安全群組。

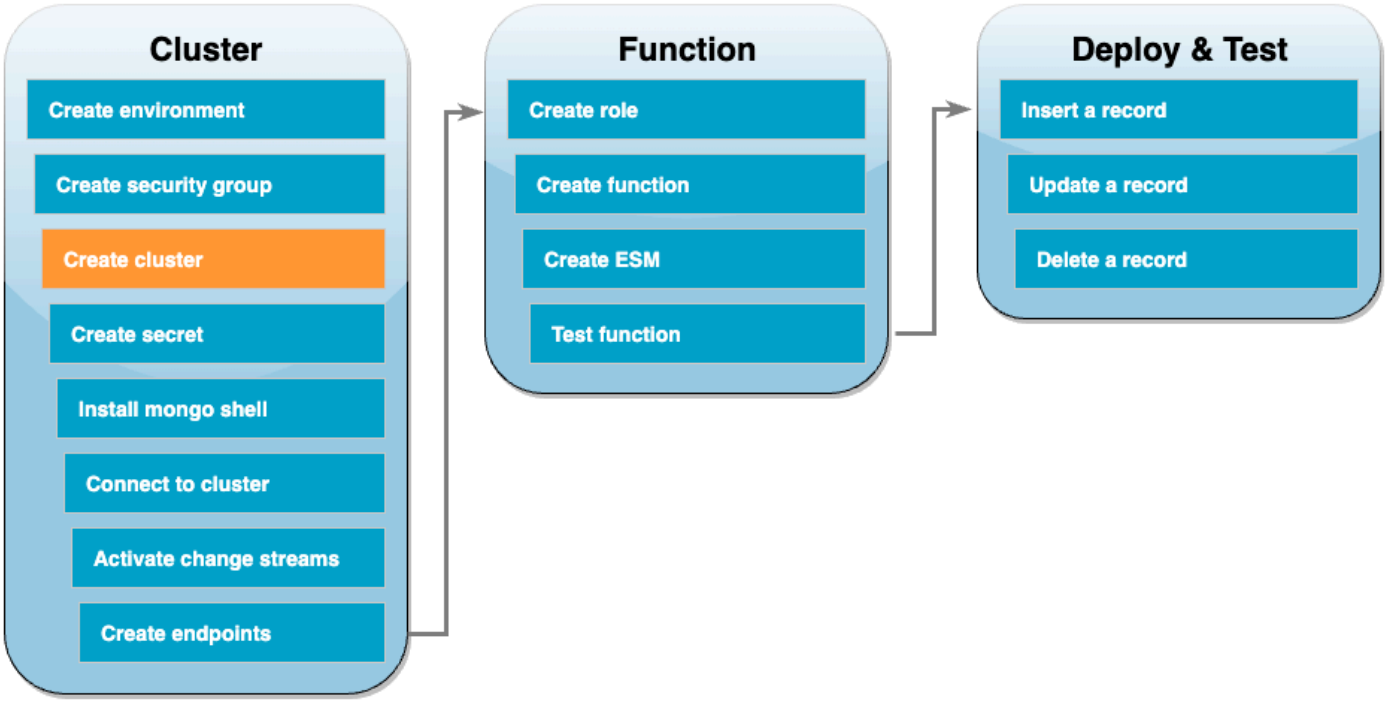
建立 EC2 安全群組

1. 開啟 EC2 主控台。在網路與安全性下，選擇安全群組。
2. 選擇建立安全群組。
3. 使用下列組態建立安全群組：
   - 在基本詳細資訊下：
     - 安全群組名稱 – `DocDBTutorial`
     - 描述 – Cloud9 和 DocumentDB 之間流量的安全群組。
     - VPC – 選擇預設 VPC。
   - 在 Inbound rules (入站規則) 下，選擇 Add rule (新增規則)。使用下列組態建立規則：
     - 類型 – 自訂 TCP
     - 連接埠範圍 – 27017
     - 來源 – 自訂

- 在「來源」旁邊的搜尋方塊中，為您在上一個步驟中建立的 AWS Cloud9 環境選擇安全性群組。若要查看可用安全群組清單，請在搜尋方塊中輸入 `cloud9`。選擇名稱為 `aws-cloud9-<environment_name>` 的安全群組。
- 請保留所有其他預設設定。

4. 選擇建立安全群組。

建立 DocumentDB 叢集



在此步驟中，您將使用上一個步驟中的安全群組建立 DocumentDB 叢集。

建立 DocumentDB 叢集

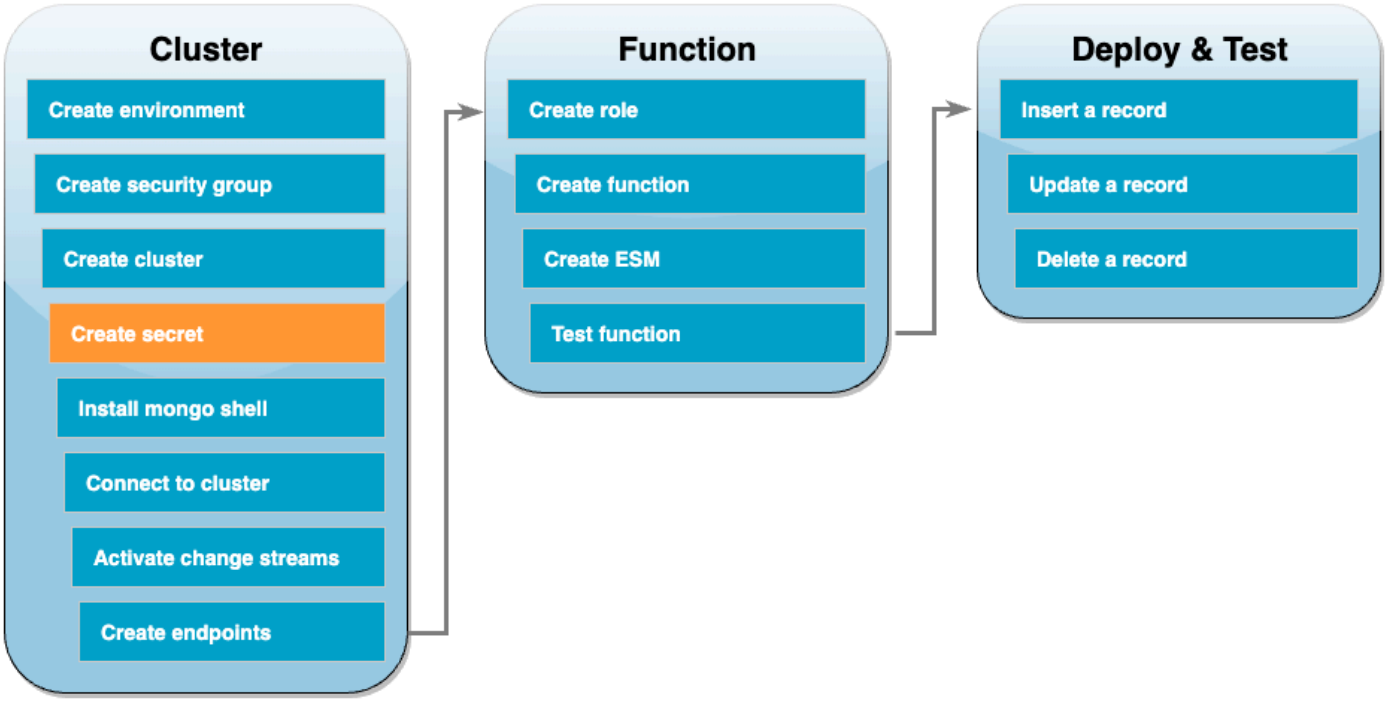
1. 開啟 DocumentDB 主控台。在叢集下，選擇建立。
2. 使用下列組態建立叢集：

   - 針對叢集類型，選擇執行個體型叢集。
   - 在組態下：
     - 發動機版本-5.0.0
     - 執行個體類別 — db.t3. 中型 (符合免費試用資格)
     - 執行個體數目 – 1。

- 在身分驗證下：
  - 輸入連線到叢集所需的使用者名稱和密碼 (與您在上一個步驟中建立機密時所用的憑證相同)。在確認密碼中，確認您的密碼。
- 開啟顯示進階設定。
- 在網路設定下：
  - 虛擬私有雲端 (VPC) – 選擇預設 VPC。
  - 子網路群組 – 預設
  - VPC 安全群組 – 除 `default (VPC)` 之外，請選擇您在上一個步驟中建立的 `DocDBTutorial (VPC)` 安全群組。
- 請保留所有其他預設設定。

3. 選擇建立叢集。佈建 DocumentDB 叢集可能需要幾分鐘的時間。

在 Secrets Manager 中建立密碼



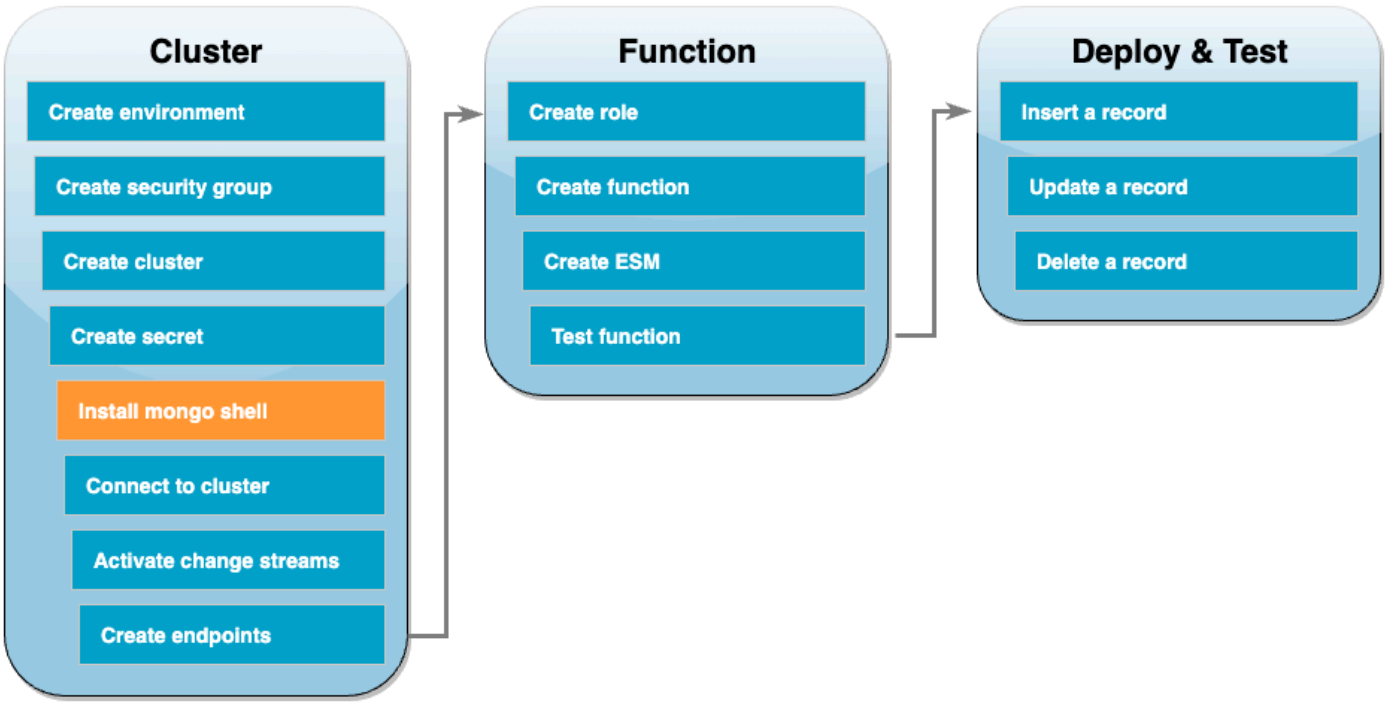
若要手動存取 DocumentDB 叢集，您必須提供使用者名稱和密碼憑證。若要讓 Lambda 存取您的叢集，您必須提供一個 Secrets Manager 機密，其中包含設定事件來源映射時的相同存取憑證。在此步驟中，您將建立此密碼。

在 Secrets Manager 中建立密碼

1. 開啟 Secrets Manager 主控台，並選擇儲存新密碼。
2. 針對選擇密碼類型，選擇以下選項：
   - 在基本詳細資訊下：
     - 密碼類型 – Amazon DocumentDB 資料庫的憑證
     - 在憑證下，輸入將用來存取 DocumentDB 叢集的使用者名稱和密碼。
     - 資料庫 – 選擇您的 DocumentDB 叢集。
     - 選擇下一步。
3. 針對設定密碼，選擇下列選項：
   - 密碼名稱 – `DocumentDBSecret`
   - 選擇下一步。
4. 選擇下一步。
5. 選擇儲存。
6. 重新整理主控台以確認您已成功儲存 `DocumentDBSecret` 密碼。

記下密碼的密碼 ARN。在後續步驟中需要它。

安裝 mongo Shell



在此步驟中，您將在 Cloud9 環境中安裝 mongo Shell。mongo Shell 是一個命令行公用程式，可以使用它來連接和查詢 DocumentDB 叢集。

在 Cloud9 環境中安裝 mongo Shell

1. 開啟 Cloud9 主控台。在先前建立的 `DocumentDBCloud9Environment` 環境旁邊，按一下 Cloud9 IDE 資料欄下的開啟連結。
2. 在終端視窗中，使用下列命令建立 MongoDB 儲存庫檔案：

```
echo -e "[mongodb-org-5.0] \nname=MongoDB Repository\nbaseurl=https://
repo.mongodb.org/yum/amazon/2/mongodb-org/5.0/x86_64/\ngpgcheck=1 \nenabled=1
 \ngpgkey=https://www.mongodb.org/static/pgp/server-5.0.asc" | sudo tee /etc/
yum.repos.d/mongodb-org-5.0.repo
```

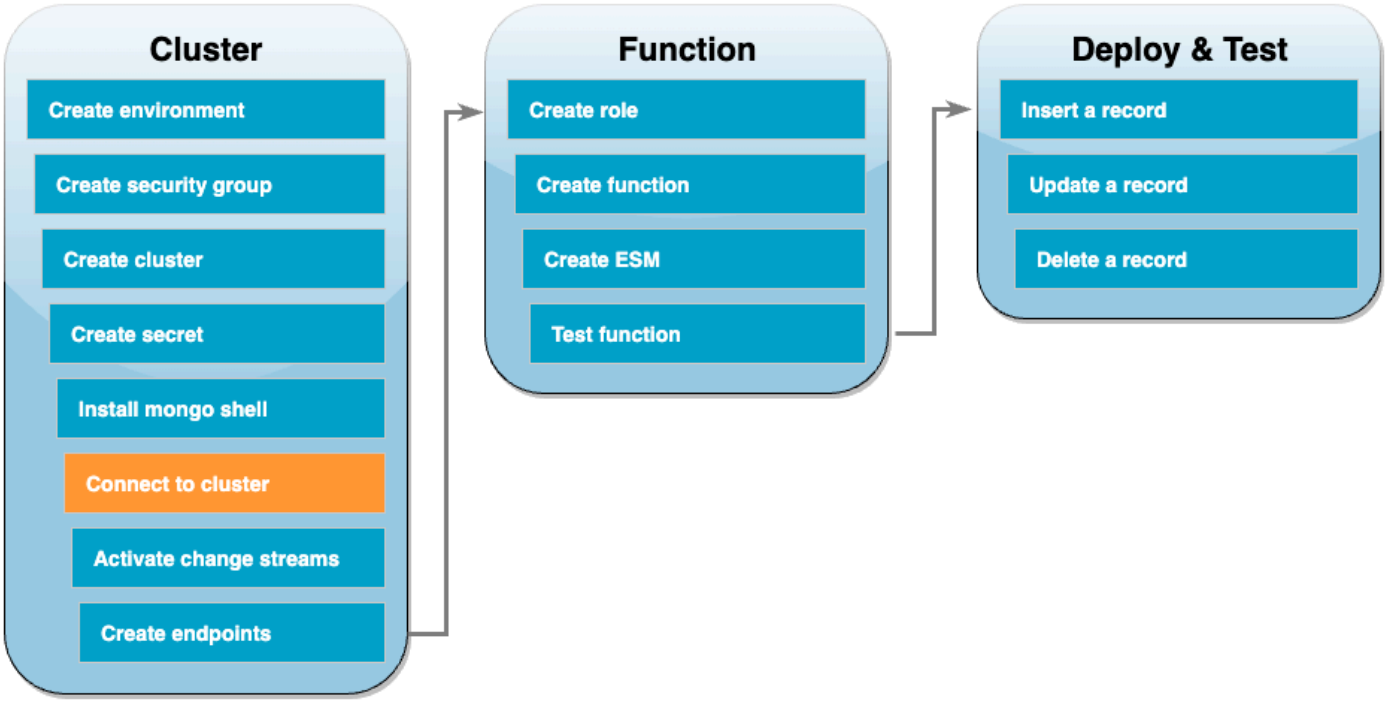
3. 然後，使用以下命令安裝 mongo Shell：

```
sudo yum install -y mongodb-org-shell
```

4. 若要加密傳輸中的資料，請下載 Amazon DocumentDB 的公有金鑰。下列命令會下載名為 `global-bundle.pem` 的檔案：

```
wget https://truststore.pki.rds.amazonaws.com/global/global-bundle.pem
```

連線至 DocumentDB 叢集



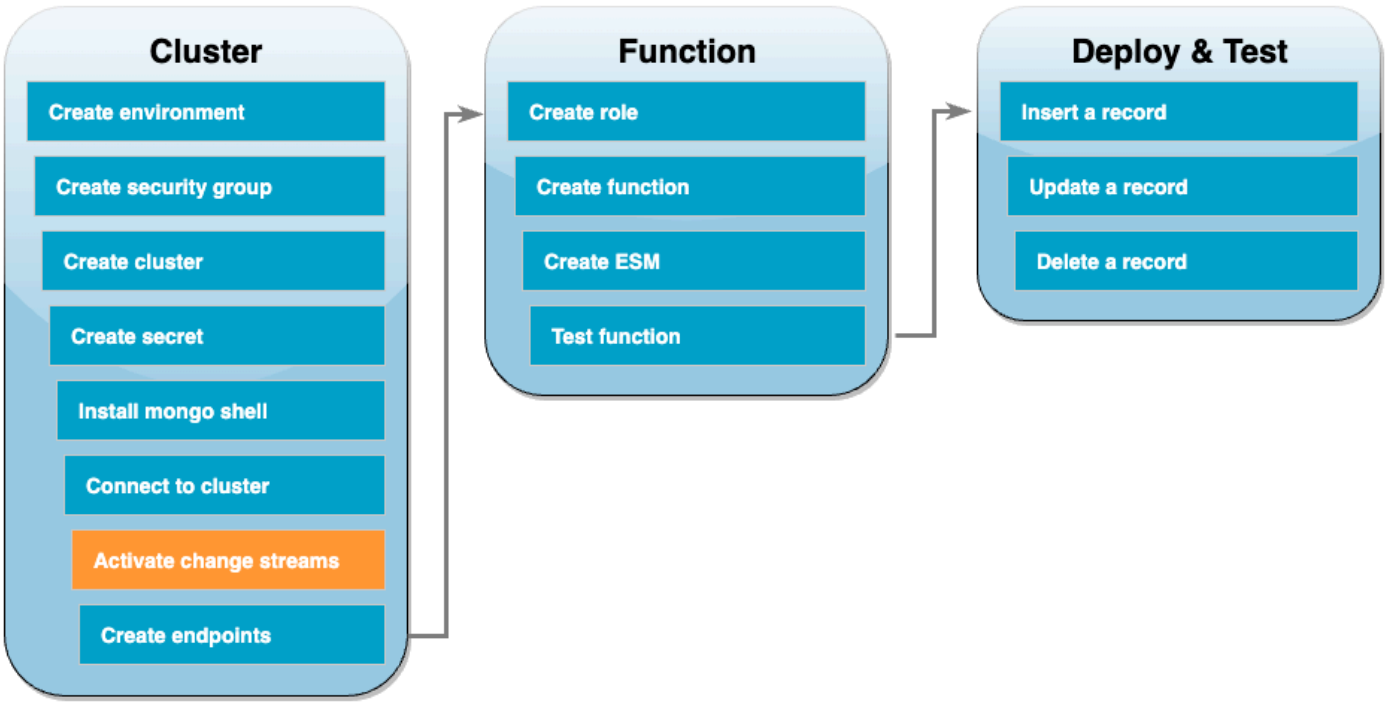
現在可以使用 mongo Shell 連線至 DocumentDB 叢集。

連線至 DocumentDB 叢集

1. 開啟 DocumentDB 主控台。在叢集下，透過選擇叢集識別碼來選擇叢集。
2. 在連線和安全索引標籤的使用 mongo Shell 連線至此叢集下，選擇複製。
3. 在 Cloud9 環境中，將此命令貼到終端中。將 `<insertYourPassword>` 取代為正確密碼。

輸入此命令之後，如果命令提示變為 `rs0:PRIMARY>`，表示您已連線至 Amazon DocumentDB 叢集。

啟用變更串流



在本教學課程中，您將追蹤 DocumentDB 叢集中 docdbdemo 資料庫 products 集合的變更。可以透過啟用變更串流來完成此操作。首先，建立 docdbdemo 資料庫，並插入記錄進行測試。

在叢集內建立新資料庫

1. 在 Cloud9 環境中，請確保仍然連線到 DocumentDB 叢集。
2. 在終端視窗中，使用下列命令建立名為 docdbdemo 的新資料庫：

```
use docdbdemo
```

3. 然後，使用下列命令將記錄插入 docdbdemo：

```
db.products.insert({"hello":"world"})
```

您應該會看到輸出，如下所示：

```
WriteResult({ "nInserted" : 1 })
```

4. 使用下列命令列出所有資料庫：

```
show dbs
```

確保您的輸出包含 docdbdemo 資料庫：

```
docdbdemo  0.000GB
```

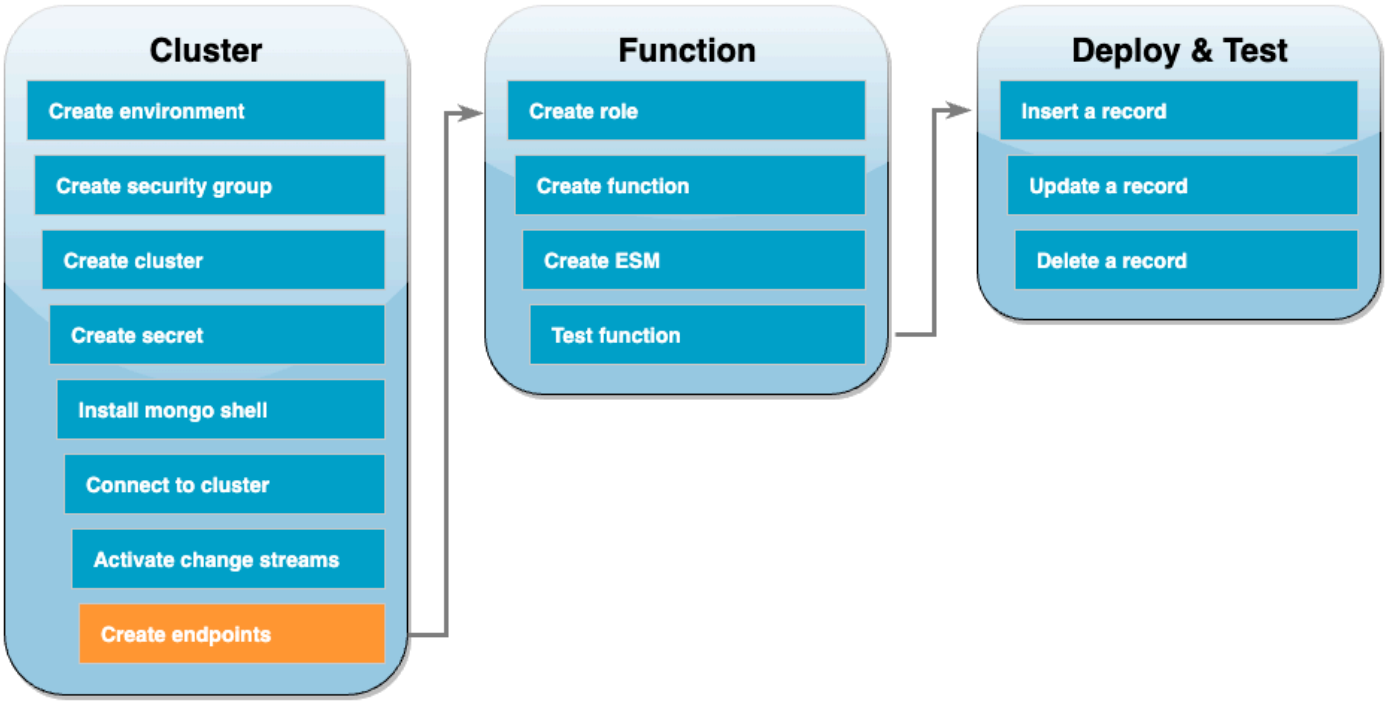
接下來，使用以下命令在 docdbdemo 資料庫的 products 集合上啟用變更串流：

```
db.adminCommand({modifyChangeStreams: 1,
    database: "docdbdemo",
    collection: "products",
    enable: true});
```

您應該會看到輸出，如下所示：

```
{ "ok" : 1, "operationTime" : Timestamp(1680126165, 1) }
```

建立介面 VPC 端點



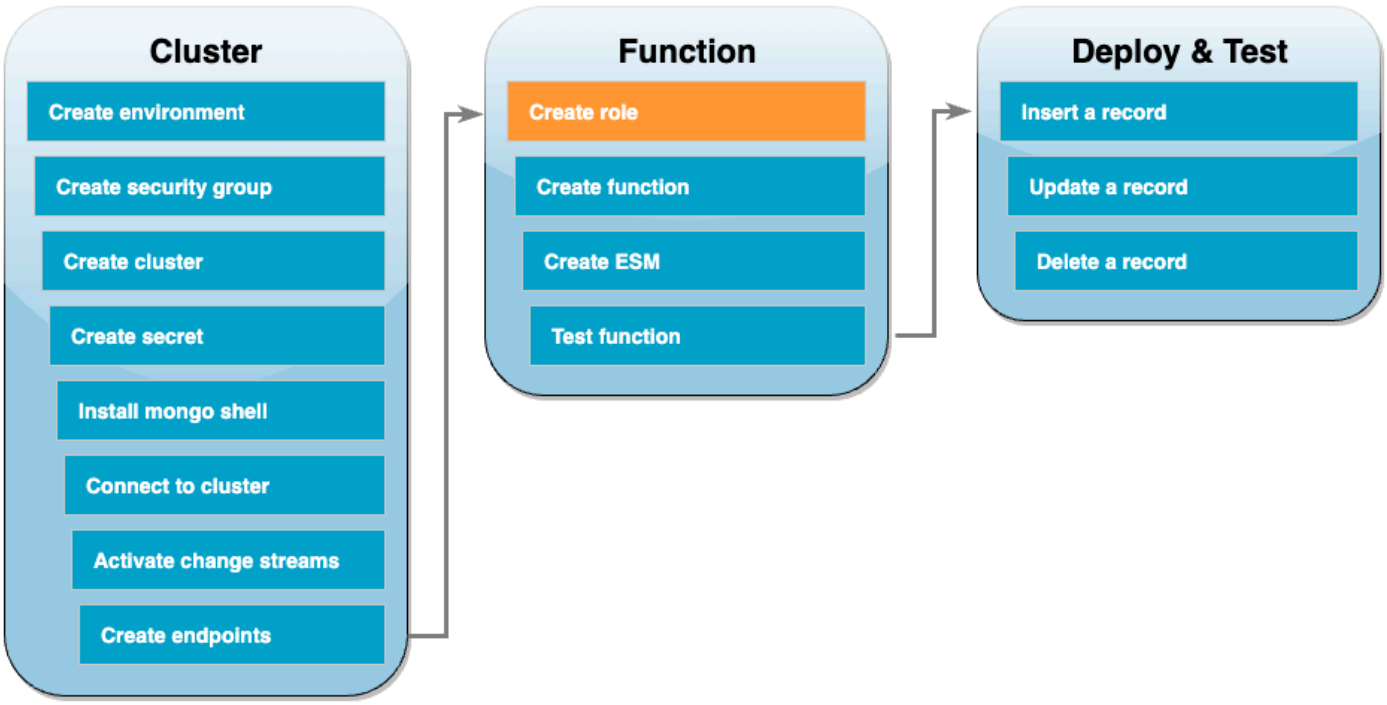
接下來，建立介面 VPC 端點，以確保 Lambda 和 Secrets Manager (稍後用來儲存我們的叢集存取憑證) 可以連線到您的預設 VPC。

建立介面 VPC 端點

1. 開啟 VPC 主控台。在左側選單的虛擬私有雲端下，選擇端點。
2. 選擇建立端點。使用下列組態建立端點：
   - 針對名稱標籤，輸入 `lambda-default-vpc`。
   - 在「服務」類別中，選擇「 AWS 服務」
   - 針對服務，在搜尋方塊中輸入 `lambda`。選擇格式為 `com.amazonaws.<region>.lambda` 的服務。
   - 針對 VPC，請選擇預設 VPC。
   - 針對子網路，請核取每個可用區域旁邊的方塊。請選擇每個可用區域的正確子網路。
   - 針對 IP 地址類型，請選擇 IPv4。
   - 針對安全群組，請選擇預設 VPC 安全群組 (群組名稱為 `default`)，以及您先前建立的安全群組 (群組名稱為 `DocDBTutorial`)。
   - 請保留所有其他預設設定。
   - 選擇建立端點。
3. 再次選擇建立端點。使用下列組態建立端點：
   - 針對名稱標籤，輸入 `secretsmanager-default-vpc`。
   - 在「服務」類別中，選擇「 AWS 服務」
   - 針對服務，在搜尋方塊中輸入 `secretsmanager`。選擇格式為 `com.amazonaws.<region>.secretsmanager` 的服務。
   - 針對 VPC，請選擇預設 VPC。
   - 針對子網路，請核取每個可用區域旁邊的方塊。請選擇每個可用區域的正確子網路。
   - 針對 IP 地址類型，請選擇 IPv4。
   - 針對安全群組，請選擇預設 VPC 安全群組 (群組名稱為 `default`)，以及您先前建立的安全群組 (群組名稱為 `DocDBTutorial`)。
   - 請保留所有其他預設設定。
   - 選擇建立端點。

這就完成了本教學課程的叢集設定部分。

建立執行角色



在接下來的一組步驟中，您將建立 Lambda 函數。首先，您需要建立授予函數存取叢集許可的執行角色。您可先建立 IAM 政策，然後將此政策連接到 IAM 角色。

建立 IAM 政策

1. 開啟 IAM 主控台的政策頁面，並選擇建立政策。
2. 選擇 JSON 標籤。在下列政策中，將陳述式最後一行中的 Secrets Manager 資源 ARN 取代為先前的密碼 ARN，並將政策複製到編輯器中。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "LambdaESMNetworkingAccess",
            "Effect": "Allow",
            "Action": [
                "ec2:CreateNetworkInterface",
                "ec2:DescribeNetworkInterfaces",
                "ec2:DescribeVpcs",
                "ec2:DeleteNetworkInterface",
                "ec2:DescribeSubnets",
```

```
                "ec2:DescribeSecurityGroups",
                "kms:Decrypt"
            ],
            "Resource": "*"
        },
        {
            "Sid": "LambdaDocDBESMAccess",
            "Effect": "Allow",
            "Action": [
                "rds:DescribeDBClusters",
                "rds:DescribeDBClusterParameters",
                "rds:DescribeDBSubnetGroups"
            ],
            "Resource": "*"
        },
        {
            "Sid": "LambdaDocDBESMGetSecretValueAccess",
            "Effect": "Allow",
            "Action": [
                "secretsmanager:GetSecretValue"
            ],
            "Resource": "arn:aws:secretsmanager:us-
east-1:123456789012:secret:DocumentDBSecret"
        }
    ]
}
```

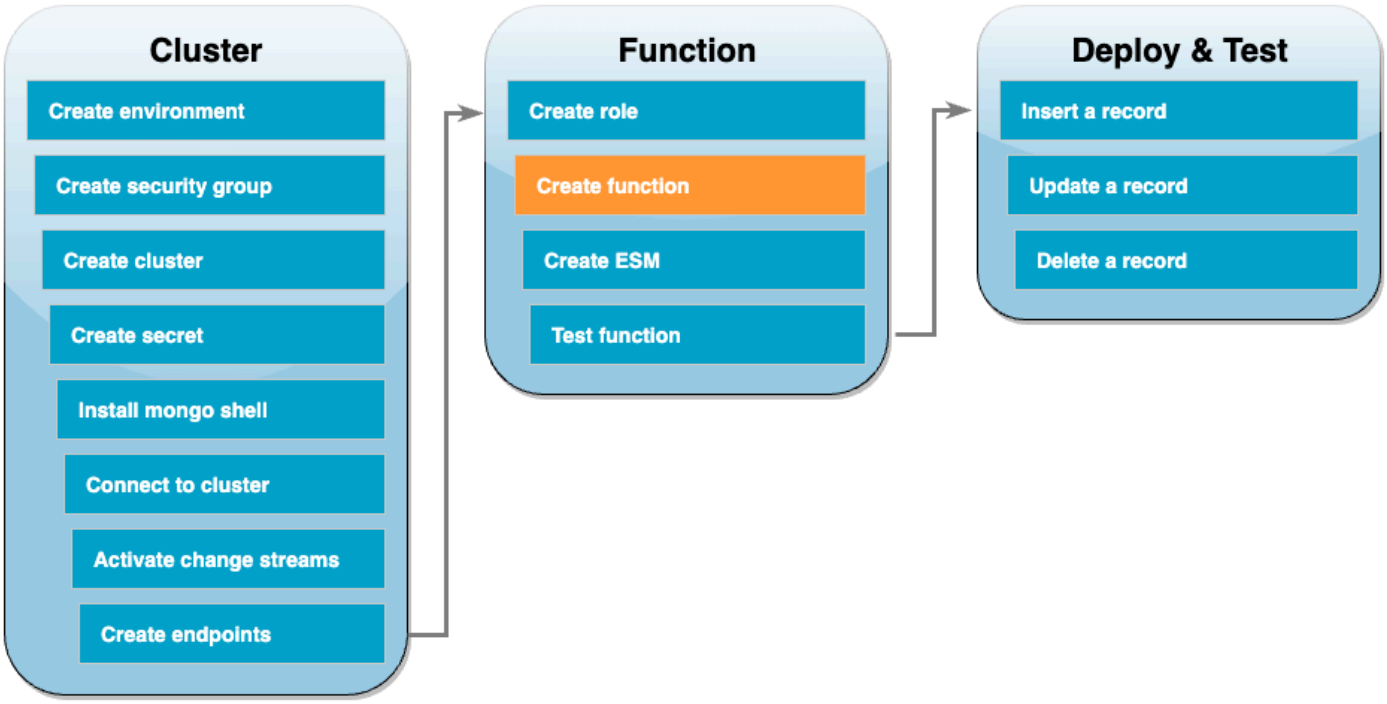
3. 選擇下一步：標籤，然後選擇下一步：檢閱。
4. 對於 Name (名稱)，輸入 AWSDocumentDBLambdaPolicy。
5. 選擇建立政策。

建立 IAM 角色

1. 開啟 IAM 主控台的角色頁面，然後選擇建立角色。
2. 針對選取信任的實體，請選擇以下選項：
   - 受信任的實體類型 — AWS 服務
   - 使用案例 – Lambda
   - 選擇下一步。

3. 對於新增許可，請選擇您剛建立的AWSDocumentDBLambdaPolicy政策，AWSLambdaBasicExecutionRole以及授予函數寫入 Amazon CloudWatch Logs 的權限。
4. 選擇下一步。
5. 在角色名稱中，輸入 AWSDocumentDBLambdaExecutionRole。
6. 選擇建立角色。

建立 Lambda 函式



下列範本程式碼會接收 DocumentDB 事件輸入，並處理其包含的訊息。

Go

SDK for Go V2

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用圍棋使用 Lambda 使用 Amazon DocumentDB 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

package main

import (
 "context"
 "encoding/json"
 "fmt"

 "github.com/aws/aws-lambda-go/lambda"
)

type Event struct {
 Events []Record `json:"events"`
}

type Record struct {
 Event struct {
  OperationType string `json:"operationType"`
  NS            struct {
   DB   string `json:"db"`
   Coll string `json:"coll"`
  } `json:"ns"`
  FullDocument interface{} `json:"fullDocument"`
 } `json:"event"`
}

func main() {
 lambda.Start(handler)
}

func handler(ctx context.Context, event Event) (string, error) {
 fmt.Println("Loading function")
 for _, record := range event.Events {
  logDocumentDBEvent(record)
 }

 return "OK", nil
}

func logDocumentDBEvent(record Record) {
 fmt.Printf("Operation type: %s\n", record.Event.OperationType)
```

```
 fmt.Printf("db: %s\n", record.Event.NS.DB)
 fmt.Printf("collection: %s\n", record.Event.NS.Coll)
 docBytes, _ := json.MarshalIndent(record.Event.FullDocument, "", "  ")
 fmt.Printf("Full document: %s\n", string(docBytes))
}
```

JavaScript

適用於 JavaScript (v3) 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Amazon DocumentDB 事件 Lambda 使用. JavaScript

```
console.log('Loading function');
exports.handler = async (event, context) => {
    event.events.forEach(record => {
        logDocumentDBEvent(record);
    });
    return 'OK';
};

const logDocumentDBEvent = (record) => {
    console.log('Operation type: ' + record.event.operationType);
    console.log('db: ' + record.event.ns.db);
    console.log('collection: ' + record.event.ns.coll);
    console.log('Full document:', JSON.stringify(record.event.fullDocument, null,
 2));
};
```

Python

適用於 Python (Boto3) 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Python 使用 Lambda 消費 Amazon DocumentDB 事件。

```
import json

def lambda_handler(event, context):
    for record in event.get('events', []):
        log_document_db_event(record)
    return 'OK'

def log_document_db_event(record):
    event_data = record.get('event', {})
    operation_type = event_data.get('operationType', 'Unknown')
    db = event_data.get('ns', {}).get('db', 'Unknown')
    collection = event_data.get('ns', {}).get('coll', 'Unknown')
    full_document = event_data.get('fullDocument', {})

    print(f"Operation type: {operation_type}")
    print(f"db: {db}")
    print(f"collection: {collection}")
    print("Full document:", json.dumps(full_document, indent=2))
```

Ruby

適用於 Ruby 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用紅寶石使用 Lambda 消費 Amazon DocumentDB 事件。

```
require 'json'

def lambda_handler(event:, context:)
  event['events'].each do |record|
    log_document_db_event(record)
  end
  'OK'
end

def log_document_db_event(record)
  event_data = record['event'] || {}
  operation_type = event_data['operationType'] || 'Unknown'
  db = event_data.dig('ns', 'db') || 'Unknown'
  collection = event_data.dig('ns', 'coll') || 'Unknown'
  full_document = event_data['fullDocument'] || {}

  puts "Operation type: #{operation_type}"
  puts "db: #{db}"
  puts "collection: #{collection}"
  puts "Full document: #{JSON.pretty_generate(full_document)}"
end
```

建立 Lambda 函數

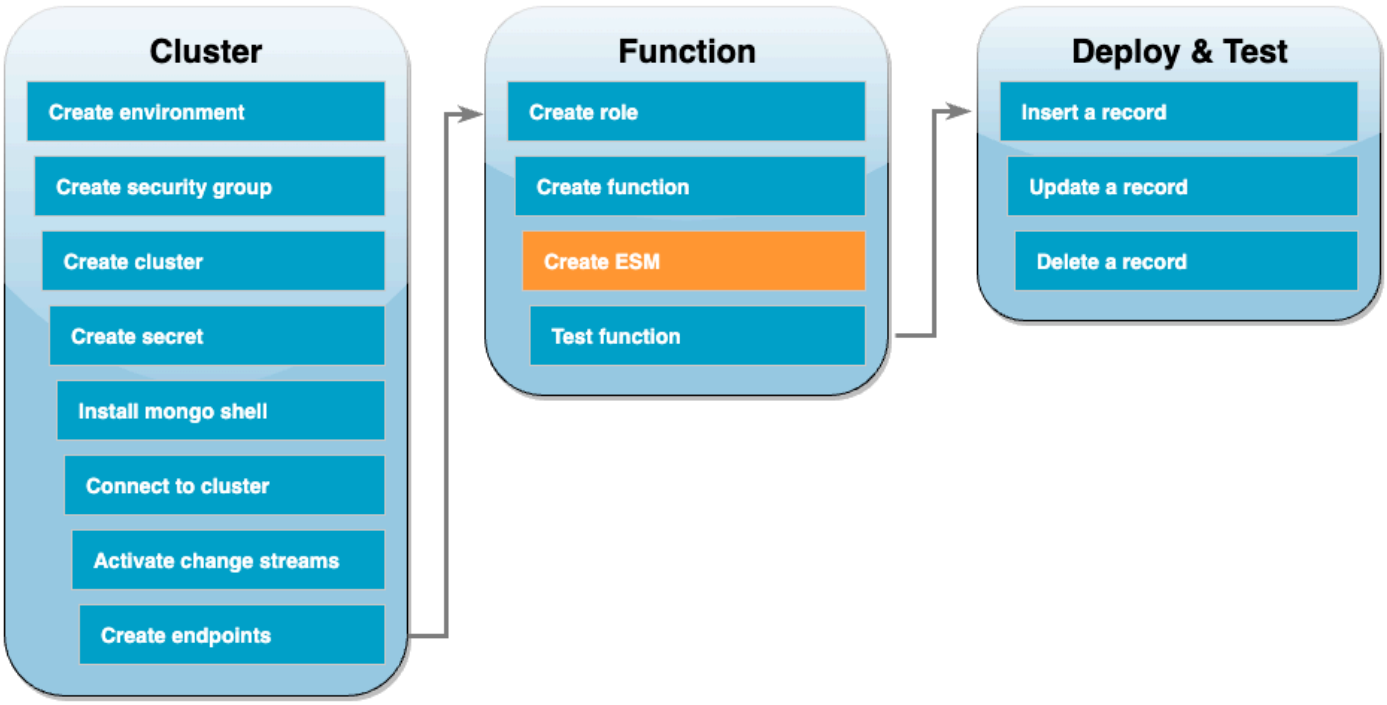
1. 將範本程式碼複製到名為 `index.js` 的檔案。
2. 使用下列命令建立部署套件。

```
zip function.zip index.js
```

3. 使用下列 CLI 命令建立函數。將 `us-east-1` 取代為區域，將 `123456789012` 取代為帳戶 ID。

```
aws lambda create-function --function-name ProcessDocumentDBRecords \
    --zip-file fileb://function.zip --handler index.handler --runtime nodejs20.x \
    --region us-east-1 \
    --role arn:aws:iam::123456789012:role/AWSDocumentDBLambdaExecutionRole
```

建立 Lambda 事件來源映射



建立可將 DocumentDB 變更串流與 Lambda 函數建立關聯的事件來源映射。建立此事件來源對應之後，AWS Lambda 立即開始輪詢串流。

建立事件來源映射

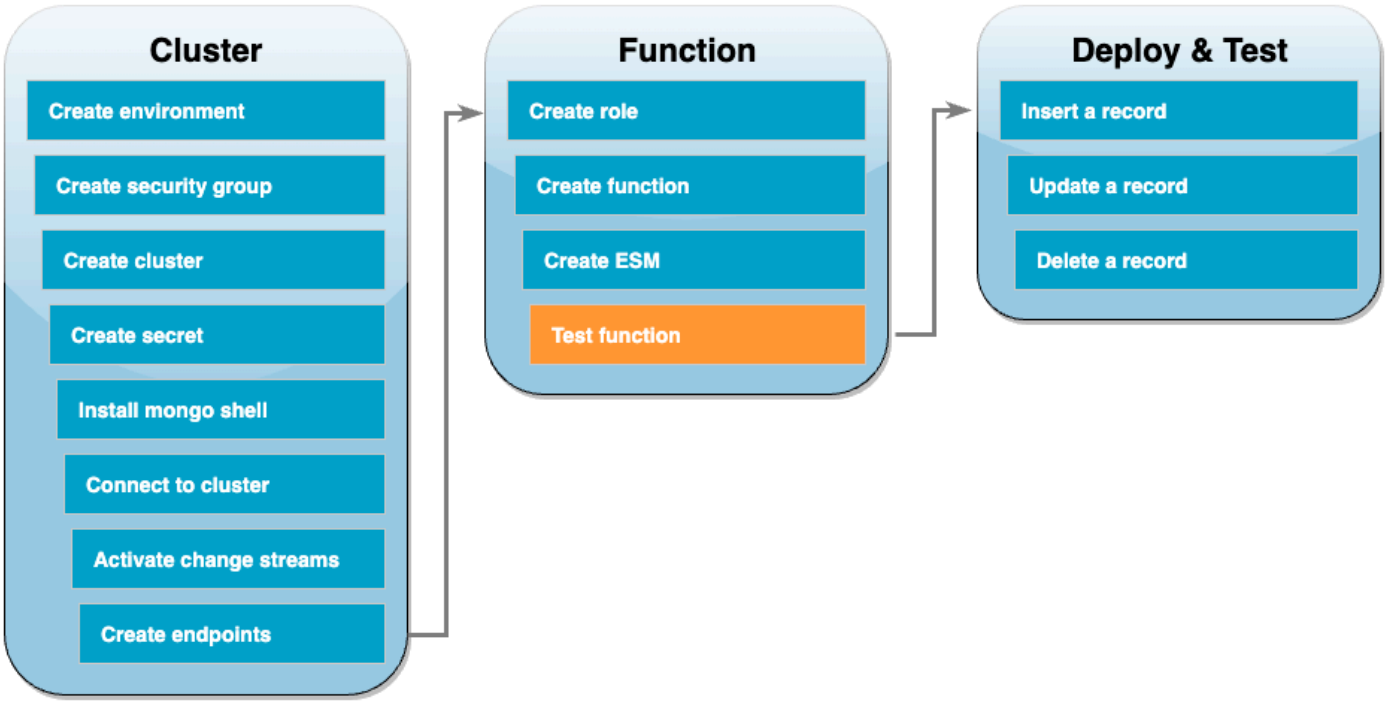
1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇您之前建立的 `ProcessDocumentDBRecords` 函數。
3. 選擇組態索引標籤，然後選擇左側選單中的觸發條件。
4. 選擇 Add trigger (新增觸發條件)。
5. 在觸發條件組態下，針對來源選取 DocumentDB。
6. 使用下列組態建立事件來源映射：

   - DocumentDB 叢集 – 選擇您先前建立的叢集。
   - 資料庫名稱 – `docdbdemo`
   - 集合名稱 – 產品
   - 批次大小 – 1
   - 起始位置 – 最新
   - 身分驗證 – BASIC_AUTH

- Secrets Manager 金鑰 – 選擇剛剛建立的 `DocumentDBSecret`。
- 批次視窗 – 1
- 完整文件配置 — UpdateLookup

7. 選擇新增。建立事件來源映射可能需要幾分鐘的時間。

測試函數 - 手動調用



若要測試您是否正確建立了函數和事件來源映射，請使用 `invoke` 命令調用函數。因此，請先將下列事件 JSON 複製到名為 `input.txt` 的檔案中：

```
{
  "eventSourceArn": "arn:aws:rds:us-east-1:123456789012:cluster:canaryclusterb2a659a2-
qo5tcmqkcl03",
  "events": [
    {
      "event": {
        "_id": {
          "_data": "0163eeb6e700000009010000009000041e1"
        },
        "clusterTime": {
          "$timestamp": {
            "t": 1676588775,
```

```
            "i": 9
          }
        },
        "documentKey": {
          "_id": {
            "$oid": "63eeb6e7d418cd98afb1c1d7"
          }
        },
        "fullDocument": {
          "_id": {
            "$oid": "63eeb6e7d418cd98afb1c1d7"
          },
          "anyField": "sampleValue"
        },
        "ns": {
          "db": "docdbdemo",
          "coll": "products"
        },
        "operationType": "insert"
      }
    }
  ],
  "eventSource": "aws:docdb"
}
```

然後，使用下列命令透過此事件調用函數：

```
aws lambda invoke --function-name ProcessDocumentDBRecords \
    --cli-binary-format raw-in-base64-out \
    --region us-east-1 \
    --payload file://input.txt out.txt
```

應看到如下回應：

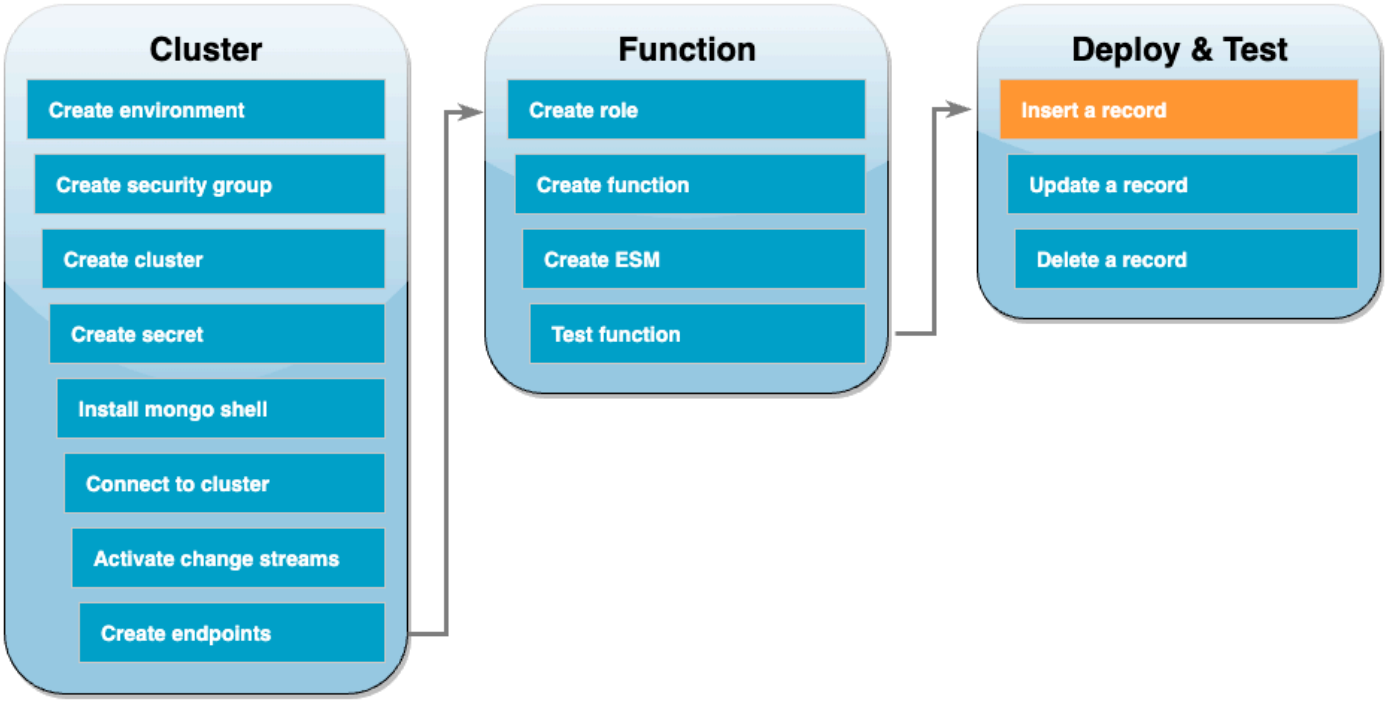
```
{
   "StatusCode": 200,
   "ExecutedVersion": "$LATEST"
}
```

您可以檢查 CloudWatch 記錄來驗證您的函數是否已成功處理事件。

透 CloudWatch 過記錄驗證手動呼叫

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇「監控」標籤頁，然後選擇「檢視 CloudWatch 記錄」。這會將您帶到與 CloudWatch 控制台中的功能相關聯的特定日誌組。
3. 選擇最新的日誌串流。在日誌訊息中，應能看到事件 JSON。

測試函數 - 插入記錄



通過直接與您的 DocumentDB 數據庫進行交互來測試您的 end-to-end 設置。在接下來的一組步驟中，您將插入記錄、更新記錄，然後將其刪除。

插入記錄

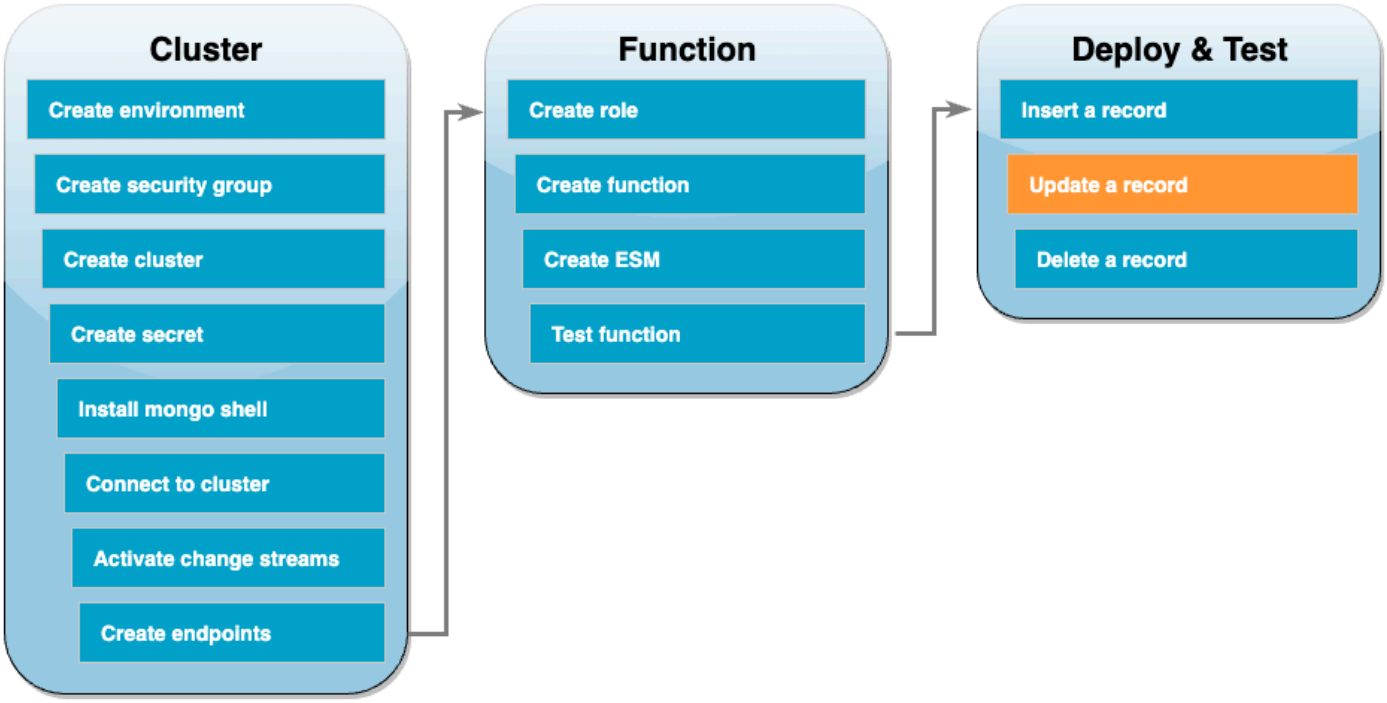
1. 在 Cloud9 環境中重新連線至 DocumentDB 叢集。
2. 使用此命令可確保您目前正在使用 docdbdemo 資料庫：

```
use docdbdemo
```

3. 將記錄插入到 docdbdemo 資料庫的 products 集合中：

```
db.products.insert({"name":"Pencil", "price": 1.00})
```

測試函數 - 更新記錄

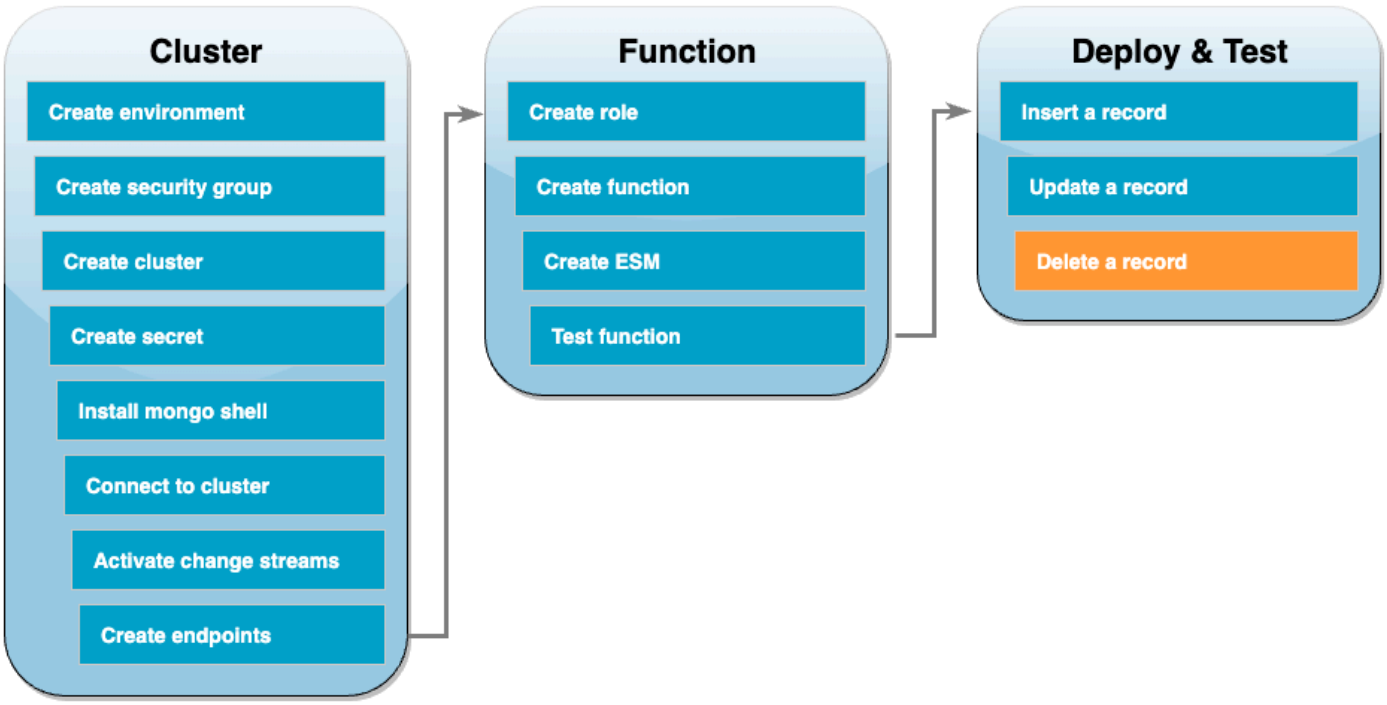


接下來，使用以下命令更新剛剛插入的記錄：

```
db.products.update(
    { "name": "Pencil" },
    { $set: { "price": 0.50 }}
)
```

檢查 CloudWatch 記錄檔，確認您的函數是否已成功處理此事件。

測試函數 - 刪除記錄



最後，使用以下命令刪除剛剛更新的記錄：

```
db.products.remove( { "name": "Pencil" } )
```

檢查 CloudWatch 記錄檔，確認您的函數是否已成功處理此事件。

清除您的資源

除非您想要保留為此教學課程建立的資源，否則您現在便可刪除。透過刪除您不再使用的 AWS 資源，可為 AWS 帳戶避免不必要的費用。

若要刪除 Lambda 函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇您建立的函數。
3. 選擇 Actions (動作)、Delete (刪除)。
4. 在文字輸入欄位中輸入 **delete**，然後選擇 刪除 。

刪除執行角色

1. 開啟 IAM 主控台中的 角色頁面 。
2. 選取您建立的執行角色。
3. 選擇 刪除 。
4. 在文字輸入欄位中輸入角色的名稱，然後選擇 刪除 。

刪除 VPC 端點。

1. 開啟 VPC 主控台。在左側選單的虛擬私有雲端下，選擇端點。
2. 選擇您建立的端點。
3. 選擇 Actions (動作)、Delete VPC endpoints (刪除 VPC 端點)。
4. 在文字輸入欄位中輸入 **`delete`**。
5. 選擇 刪除 。

刪除 Amazon DocumentDB 叢集

1. 開啟 DocumentDB 主控台。
2. 選擇您為本教學課程建立的 DocumentDB 叢集，並停用刪除保護。
3. 在主叢集頁面中，再次選擇您的 DocumentDB 叢集。
4. 選擇 動作、刪除。
5. 針對建立最終叢集快照，請選取否。
6. 在文字輸入欄位中輸入 **`delete`**。
7. 選擇 刪除 。

在 Secrets Manager 中刪除密碼

1. 開啟 Secrets Manager 主控台。
2. 選擇您為此教學課程建立的密碼。
3. 選擇動作、刪除機密。
4. 選擇 Schedule deletion (排定刪除)。

刪除 Amazon EC2 安全群組

1. 開啟 EC2 主控台。在網路與安全性下，選擇安全群組。
2. 選擇您為此教學課程建立的安全群組。
3. 選擇動作、刪除安全群組。
4. 選擇刪除。

刪除 Cloud9 環境

1. 開啟 Cloud9 主控台。
2. 選取您為本教學課程建立的環境。
3. 選擇 刪除 。
4. 在文字輸入欄位中輸入 **delete**。
5. 選擇 刪除 。

Lambda 事件篩選

您可以使用事件篩選來控制 Lambda 將哪些記錄從串流或佇列中傳送至函數。例如，您可以新增篩選條件，讓函數僅處理包含特定資料參數的 Amazon SQS 訊息。時間篩選可與事件來源映射搭配運作。您可以將篩選器新增至下列 AWS 服務的事件來源對應：

- Amazon DynamoDB
- Amazon Kinesis Data Streams
- Amazon MQ
- Amazon Managed Streaming for Apache Kafka (Amazon MSK)
- 自我管理的 Apache Kafka
- Amazon Simple Queue Service (Amazon SQS)

Lambda 不支援 Amazon DocumentDB 的事件篩選。

依預設，最多可以為單一事件來源映射定義五種不同的篩選條件。篩選條件透過 OR 運算邏輯關聯。如果事件來源的記錄滿足一個或多個篩選條件，則 Lambda 會在它傳送至函數的下一個事件中包含該記錄。如果全部篩選條件都不滿足，則 Lambda 會捨棄該記錄。

> **Note**
>
> 如果需要為一個事件來源定義五個以上的篩選條件，則可以為每個事件來源請求增加最多 10 個篩選條件的配額。如果您嘗試新增超過目前配額許可的篩選條件，Lambda 將在您嘗試並建立事件來源時傳回錯誤。

主題

事件篩選基本資訊

篩選條件標準 (`FilterCriteria`) 物件是由篩選條件清單 (`Filters`) 組成的結構。每個篩選條件均是定義事件篩選模式 (`Pattern`) 的結構。模式是 JSON 篩選條件規則的字串表示法。`FilterCriteria` 物件的結構如下所示。

```
{
   "Filters": [
        {
            "Pattern": "{ \"Metadata1\": [ rule1 ], \"data\": { \"Data1\":
 [ rule2 ] }}"
        }
    ]
}
```

補充說明，此處是篩選條件的 Pattern 在純文字 JSON 中擴展的值。

```
{
    "Metadata1": [ rule1 ],
    "data": {
        "Data1": [ rule2 ]
    }
}
```

篩選條件模式可以包含中繼資料屬性、資料屬性或兩者。可用的中繼資料參數和資料參數的格式會根據做為事件來源的不同而有所不同。 AWS 服務 例如，假設您的事件來源映射從 Amazon SQS 佇列中收到以下記錄：

```
{
    "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
    "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",
    "body": "{\n "City": "Seattle",\n "State": "WA",\n "Temperature": "46"\n}",
    "attributes": {
        "ApproximateReceiveCount": "1",
        "SentTimestamp": "1545082649183",
        "SenderId": "AIDAIENQZJOLO23YVJ4VO",
        "ApproximateFirstReceiveTimestamp": "1545082649185"
    },
    "messageAttributes": {},
    "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
    "eventSource": "aws:sqs",
    "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:my-queue",
    "awsRegion": "us-east-2"
}
```

- 中繼資料屬性是包含有關建立該記錄之事件資訊的欄位。在 Amazon SQS 記錄範例中，中繼資料屬性包括 messageID、eventSourceArn 和 awsRegion 等欄位。
- 資料屬性是包含串流或佇列資料的記錄欄位。在 Amazon SQS 事件範例中，資料欄位的索引鍵為 body，而資料屬性為 City、State 和 Temperature 欄位。

不同類型的事件來源對其資料欄位使用不同的索引鍵值。若要篩選資料屬性，請務必在篩選條件模式中使用正確的索引鍵。如需資料篩選索引鍵的清單，以及若要查看每個支援的篩選模式範例 AWS 服務，請參閱使用不同的過濾器 AWS 服務。

事件篩選可處理多級 JSON 篩選。例如，考慮 DynamoDB 串流中的下列記錄片段：

```
"dynamodb": {
    "Keys": {
        "ID": {
            "S": "ABCD"
        }
        "Number": {
            "N": "1234"
    },
    ...
}
```

假設您只想處理排序索引鍵 Number 值為 4567 的記錄。在這種情況下，您的 FilterCriteria 物件看起來像這樣：

```
{
    "Filters": [
        {
            "Pattern": "{ \"dynamodb\": { \"Keys\": { \"Number\": { \"N\":
 [ "4567" ] } } } }"
        }
    ]
}
```

補充說明，此處是篩選條件的 Pattern 在純文字 JSON 中擴展的值。

```
{
    "dynamodb": {
        "Keys": {
            "Number": {
                "N": [ "4567" ]
                }
            }
        }
}
```

處理不符合篩選條件標準的記錄

處理不符合篩選條件之記錄的方法取決於事件來源。

- 針對 Amazon SQS，如果訊息不符合您的篩選條件標準，Lambda 會自動從佇列中刪除該訊息。您不需要在 Amazon SQS 中手動刪除這些訊息。

- 針對 Kinesis 和 DynamoDB，您的篩選條件標準處理記錄後，串流迭代器將向前移動超過此記錄。如果記錄不滿足篩選條件標準，則無需從事件來源中手動刪除記錄。保留期之後，Kinesis 和 DynamoDB 會自動刪除這些舊記錄。如果希望更快地刪除記錄，請參閱變更資料保留期間。
- 對於 Amazon MSK、自我管理的 Apache Kafka 和 Amazon MQ 訊息，Lambda 會捨棄不符合篩選條件中包含的所有欄位的訊息。針對自我管理 Apache Kafka，Lambda 會在成功調用函數之後，提交相符和不相符訊息的偏移量。若為 Amazon MQ，Lambda 會在成功調用函數後確認相符的訊息，並在篩選時確認不相符的訊息。

篩選條件規則語法

對於篩選規則，Lambda 支援 Amazon EventBridge 規則，並使用與 EventBridge. 如需詳細資訊，請參閱 Amazon EventBridge 使用者指南中的 Amazon EventBridge 事件模式。

以下是可用於 Lambda 事件篩選的所有比較運算子摘要。

比較運算子	範例	Rule syntax (規則語法)
Null	UserID 為 Null (空值)	"UserID": [null]
空白	LastName 是空的	"LastName": [""]
等於	名稱為「Alice」	"Name": ["Alice"]
等於 (忽略大小寫)	名稱為「Alice」	「名稱」：[{"equals-ignore-case「：「愛麗絲」}]
及	位置為「紐約」，日期是「週一」	"Location": ["New York"], "Day": ["Monday"]
或	PaymentType 是「信用卡」或「借記卡」	「PaymentType「: [「信用卡」,「借記卡」]
或 (多個欄位)	位置為「紐約」，或日期是「週一」。	"$or": [{ "Location": ["New York"] }, { "Day": ["Monday"] }]
Not	天氣是「下雨」以外的任何天氣	"Weather": [{ "anything-but": ["Raining"] }]

比較運算子	範例	Rule syntax (規則語法)
數字 (等於)	價格為 100	"Price": [{ "numeric": ["=", 100] }]
數字 (範圍)	價格大於 10，且小於或等於 20	"Price": [{ "numeric": [">", 10, "<=", 20] }]
存在	ProductName 存在	「ProductName「：[{「存在」：真}]
不存在	ProductName 不存在	「ProductName「：[{「存在」：假}]
開頭為	區域位於美國	"Region": [{"prefix": "us-" }]
結尾為	FileName 以 .png 副檔名結束。	「FileName「: [{「後綴」：「.png」}]

> **Note**
>
> 就像 EventBridge，對於字符串，Lambda 使用精確 character-by-character 匹配而不會折疊大小寫或任何其他字符串規範化。針對數字，Lambda 也會使用字串表示法。例如，300、300.0 和 3.0e2 不會被視為相等。

請注意，Exissions 運算子僅適用於事件來源 JSON 中的葉節點。它不匹配中間節點。例如，使用下面的 JSON，過濾器模式`{ "person": { "address": [ { "exists": true } ] } }`"不會找到匹配，因為`"address"`是一個中間節點。

```
{
  "person": {
    "name": "John Doe",
    "age": 30,
    "address": {
      "street": "123 Main St",
      "city": "Anytown",
      "country": "USA"
    }
```

```
  }
}
```

將篩選條件標準連接至事件來源映射 (主控台)

依照下列步驟使用 Lambda 主控台來建立具有篩選條件標準的新事件來源映射。

若要使用篩選條件標準 (主控台) 建立新事件來源映射

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇要建立事件來源映射的函數名稱。
3. 在 函式概觀 下，選擇 新增觸發條件 。
4. 若為 Trigger configuration (觸發程序組態)，選擇支援事件篩選的觸發程序類型。如需支援的服務清單，請參閱本頁開頭的清單。
5. 展開 Additional settings (其他設定)。
6. 在 Filter criteria (篩選條件標準) 下，選擇 Add (新增)，然後定義並輸入您的篩選條件。例如，您可以輸入下列指令。

   ```
   { "Metadata" : [ 1, 2 ] }
   ```

   這會指示 Lambda 僅處理欄位 `Metadata` 等於 1 或 2 的記錄。您可以繼續選取新增，以新增更多篩選條件，直到達到允許的數目上限為止。
7. 完成新增篩選條件時，請選擇儲存。

使用主控台輸入篩選條件標準時，僅需輸入篩選模式，而不需要提供 `Pattern` 索引鍵或逸出引號。在上述指示的步驟 6 中，`{ "Metadata" : [ 1, 2 ] }` 會對應下列 `FilterCriteria`。

```
{
   "Filters": [
      {
          "Pattern": "{ \"Metadata\" : [ 1, 2 ] }"
      }
   ]
}
```

在主控台中建立事件來源映射之後，您可以在觸發程序詳細資訊中看見格式化的 `FilterCriteria`。如需有關使用主控台建立事件篩選條件的更多範例，請參閱 使用不同的過濾器 AWS 服務。

將篩選條件標準連接至事件來源映射 (AWS CLI)

假設您想要事件來源映射具有下列 `FilterCriteria`：

```
{
   "Filters": [
      {
          "Pattern": "{ \"Metadata\" : [ 1, 2 ] }"
      }
   ]
}
```

若要使用 AWS Command Line Interface (AWS CLI) 建立具有這些篩選條件的新事件來源對應，請執行下列命令。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:sqs:us-east-2:123456789012:my-queue \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"Metadata\" : [ 1, 2 ]}"}]}'
```

此create-event-source-mapping命令會為`my-function`具有指`FilterCriteria`定的函數建立新的 Amazon SQS 事件來源對應。

若要將這些篩選條件標準新增到現有事件來源映射，請執行下列命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"Metadata\" : [ 1, 2 ]}"}]}'
```

請注意，若要更新事件來源映射，您需要其 UUID。您可以從呼叫中取得 UUID。list-event-source-mappingsLambda 也會在 create-event-source-mappingCLI 回應中傳回 UUID。

若要從事件來源移除篩選準則，您可以使用空白`FilterCriteria`物件執行下列update-event-source-mapping命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria "{}"
```

如需使用建立事件篩選器的更多範例 AWS CLI，請參閱使用不同的過濾器 AWS 服務。

將篩選條件標準連接至事件來源映射 (AWS SAM)

假設您想要在中配置事件來源 AWS SAM 以使用下列篩選條件：

```
{
   "Filters": [
      {
          "Pattern": "{ \"Metadata\" : [ 1, 2 ] }"
      }
   ]
}
```

若要將這些篩選條件標準新增到事件來源映射，請將下列程式碼片段插入事件來源的 YAML 範本中。

```
FilterCriteria:
  Filters:
    - Pattern: '{"Metadata": [1, 2]}'
```

如需建立和設定事件來源對應 AWS SAM 範本的詳細資訊，請參閱《 AWS SAM 開發人員指南》一 EventSource節。如需使用範 AWS SAM 本建立事件篩選器的更多範例，請參閱使用不同的過濾器 AWS 服務。

使用不同的過濾器 AWS 服務

不同類型的事件來源對其資料欄位使用不同的索引鍵值。若要篩選資料屬性，請務必在篩選條件模式中使用正確的索引鍵。下表提供了每個支持的過濾鍵 AWS 服務。

AWS 服務	篩選索引鍵
DynamoDB	`dynamodb`
Kinesis	`data`
Amazon MQ	`data`
Amazon MSK	`value`
自我管理的 Apache Kafka	`value`
Amazon SQS	`body`

下列各節提供不同類型事件來源的篩選條件模式範例。其還為每個支援之服務提供支援的傳入資料格式和篩選條件模式內文格式的定義。

使用 DynamoDB 進行篩選

假設您有一個包含主索引鍵 CustomerName 和屬性 AccountManager 與 PaymentTerms 的 DynamoDB 表格。以下顯示 DynamoDB 表格串流中的範例記錄。

```
{
      "eventID": "1",
      "eventVersion": "1.0",
      "dynamodb": {
          "ApproximateCreationDateTime": "1678831218.0",
          "Keys": {
              "CustomerName": {
                  "S": "AnyCompany Industries"
              },
              "NewImage": {
                  "AccountManager": {
                      "S": "Pat Candella"
                  },
                  "PaymentTerms": {
                      "S": "60 days"
                  },
                  "CustomerName": {
                      "S": "AnyCompany Industries"
                  }
              },
              "SequenceNumber": "111",
              "SizeBytes": 26,
              "StreamViewType": "NEW_IMAGE"
          }
      }
  }
```

若要根據 DynamoDB 表格中的索引鍵值和屬性值進行篩選，請使用記錄中的 dynamodb 索引鍵。下列各節提供不同篩選器類型的範例。

使用表格索引鍵篩選

假設你希望你的函數只處理那些記錄，其中主鍵CustomerName是「AnyCompany 產業」。FilterCriteria 物件如下所示。

```
{
    "Filters": [
        {
            "Pattern": "{ \"dynamodb\" : { \"Keys\" : { \"CustomerName\" : { \"S\" :
 [ \"AnyCompany Industries\" ] } } } }"
        }
    ]
}
```

補充說明，此處是篩選條件的 Pattern 在純文字 JSON 中擴展的值。

```
{
    "dynamodb": {
        "Keys": {
            "CustomerName": {
                "S": [ "AnyCompany Industries" ]
                }
            }
        }
}
```

您可以使用控制台 AWS CLI 或 AWS SAM 模板添加過濾器。

Console

若要使用主控台新增此篩選條件，請遵循 將篩選條件標準連接至事件來源映射 (主控台) 中的指示，並針對篩選條件標準輸入下列字串。

```
{ "dynamodb" : { "Keys" : { "CustomerName" : { "S" : [ "AnyCompany
 Industries" ] } } } }
```

AWS CLI

若要使用 AWS Command Line Interface (AWS CLI) 建立具有這些篩選條件的新事件來源對應，請執行下列命令。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:dynamodb:us-east-2:123456789012:table/my-table \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"Keys\" :
 { \"CustomerName\" : { \"S\" : [ \"AnyCompany Industries\" ] } } } }"}]}'
```

若要將這些篩選條件標準新增到現有事件來源映射，請執行下列命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"Keys\" :
 { \"CustomerName\" : { \"S\" : [ \"AnyCompany Industries\" ] } } } }"}]}'
```

AWS SAM

若要使用新增此篩選器 AWS SAM，請將下列程式碼片段新增至事件來源的 YAML 範本。

```
FilterCriteria:
   Filters:
     - Pattern: '{ "dynamodb" : { "Keys" : { "CustomerName" : { "S" : [ "AnyCompany
 Industries" ] } } } }'
```

使用表格屬性篩選

使用 DynamoDB 時，您也可以使用 `NewImage` 和 `OldImage` 索引鍵來篩選屬性值。假設您想篩選最新表格映像中的 `AccountManager` 屬性為 "Pat Candella" 或 "Shirley Rodriguez" 的記錄。`FilterCriteria` 物件如下所示。

```
{
    "Filters": [
        {
            "Pattern": "{ \"dynamodb\" : { \"NewImage\" : { \"AccountManager\" : { \"S
\" : [ \"Pat Candella\", \"Shirley Rodriguez\" ] } } } }"
        }
    ]
}
```

補充說明，此處是篩選條件的 `Pattern` 在純文字 JSON 中擴展的值。

```
{
    "dynamodb": {
        "NewImage": {
            "AccountManager": {
                "S": [ "Pat Candella", "Shirley Rodriguez" ]
            }
        }
    }
```

```
}
```

您可以使用控制台 AWS CLI 或 AWS SAM 模板添加過濾器。

Console

若要使用主控台新增此篩選條件，請遵循 將篩選條件標準連接至事件來源映射 (主控台) 中的指示，並針對篩選條件標準輸入下列字串。

```
{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat Candella",
 "Shirley Rodriguez" ] } } } }
```

AWS CLI

若要使用 AWS Command Line Interface (AWS CLI) 建立具有這些篩選條件的新事件來源對應，請執行下列命令。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:dynamodb:us-east-2:123456789012:table/my-table \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"NewImage
\" : { \"AccountManager\" : { \"S\" : [ \"Pat Candella\", \"Shirley Rodriguez
\" ] } } } }"}]}'
```

若要將這些篩選條件標準新增到現有事件來源映射，請執行下列命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"NewImage
\" : { \"AccountManager\" : { \"S\" : [ \"Pat Candella\", \"Shirley Rodriguez
\" ] } } } }"}]}'
```

AWS SAM

若要使用新增此篩選器 AWS SAM，請將下列程式碼片段新增至事件來源的 YAML 範本。

```
FilterCriteria:
  Filters:
    - Pattern: '{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat
 Candella", "Shirley Rodriguez" ] } } } }'
```

使用布林運算式篩選

您也可以使用布林值 AND 運算式建立篩選條件。這些運算式可以同時包含資料表的索引鍵和屬性參數。假設您想篩選記錄，其中 `AccountManager` 的 `NewImage` 值是「Pat Candella」，`OldImage` 值是「Terry Whitlock」。`FilterCriteria` 物件如下所示。

```
{
    "Filters": [
        {
            "Pattern": "{ \"dynamodb\" : { \"NewImage\" : { \"AccountManager\" : { \"S
\" : [ \"Pat Candella\" ] } } } , \"dynamodb\" : { \"OldImage\" : { \"AccountManager
\" : { \"S\" : [ \"Terry Whitlock\" ] } } } }"
        }
    ]
}
```

補充說明，此處是篩選條件的 `Pattern` 在純文字 JSON 中擴展的值。

```
{
    "dynamodb" : {
        "NewImage" : {
            "AccountManager" : {
                "S" : [
                    "Pat Candella"
                ]
            }
        }
    },
    "dynamodb": {
        "OldImage": {
            "AccountManager": {
                "S": [
                    "Terry Whitlock"
                ]
            }
        }
    }
}
```

您可以使用控制台 AWS CLI 或 AWS SAM 模板添加過濾器。

Console

若要使用主控台新增此篩選條件，請遵循 將篩選條件標準連接至事件來源映射 (主控台) 中的指示，並針對篩選條件標準輸入下列字串。

```
{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat
 Candella" ] } } } , "dynamodb" : { "OldImage" : { "AccountManager" : { "S" :
 [ "Terry Whitlock" ] } } } }
```

AWS CLI

若要使用 AWS Command Line Interface (AWS CLI) 建立具有這些篩選條件的新事件來源對應，請執行下列命令。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:dynamodb:us-east-2:123456789012:table/my-table \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"NewImage
\" : { \"AccountManager\" : { \"S\" : [ \"Pat Candella\" ] } } } , \"dynamodb\" :
 { \"OldImage\" : { \"AccountManager\" : { \"S\" : [ \"Terry Whitlock\" ] } } } }
 "}]}'
```

若要將這些篩選條件標準新增到現有事件來源映射，請執行下列命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"NewImage
\" : { \"AccountManager\" : { \"S\" : [ \"Pat Candella\" ] } } } , \"dynamodb\" :
 { \"OldImage\" : { \"AccountManager\" : { \"S\" : [ \"Terry Whitlock\" ] } } } }
 "}]}'
```

AWS SAM

若要使用新增此篩選器 AWS SAM，請將下列程式碼片段新增至事件來源的 YAML 範本。

```
FilterCriteria:
  Filters:
    - Pattern: '{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat
 Candella" ] } } } , "dynamodb" : { "OldImage" : { "AccountManager" : { "S" :
 [ "Terry Whitlock" ] } } } }'
```

> **Note**
>
> DynamoDB 事件篩選不支援使用數值運算子 (數值等於和數值範圍)。即使資料表中的項目儲存為數字，這些參數也會轉換為 JSON 記錄物件中的字串。

將存在運算子與 DynamoDB 搭配使用

由於 DynamoDB 中 JSON 事件物件的結構化方式，因此使用存在運算子需要特別小心。Exexists 運算符僅適用於事件 JSON 中的葉節點，因此，如果您的過濾器模式使用 Exexists 來測試中間節點，它將無法正常工作。請考慮下列 DynamoDB 資料表項目：

```
{
  "UserID": {"S": "12345"},
  "Name": {"S": "John Doe"},
  "Organizations": {"L": [
      {"S":"Sales"},
      {"S":"Marketing"},
      {"S":"Support"}
    ]
  }
}
```

您可能想要建立如下所示的篩選器模式，以測試包含以下事件`"Organizations"`：

```
{ "dynamodb" : { "NewImage" : { "Organizations" : [ { "exists": true } ] } } }
```

但是，此過濾器模式永遠不會返回匹配項，因為不`"Organizations"`是葉節點。下列範例會示範如何正確使用 Exissions 運算子來建構所需的濾鏡模式：

```
{ "dynamodb" : { "NewImage" : {"Organizations": {"L": {"S": [ {"exists":
 true } ] } } } } }
```

JSON 格式適用於 DynamoDB 篩選

若要正確篩選來自 DynamoDB 來源的事件，資料欄位和資料欄位 (dynamodb) 的篩選條件標準都必須是有效的 JSON 格式。如果其中一個欄位不是有效的 JSON 格式，則 Lambda 會捨棄訊息或擲回例外狀況。下表摘要說明特定行為：

傳入資料格式	資料屬性的篩選條件模式格式	產生的動作
有效的 JSON	有效的 JSON	根據您的篩選條件標準之 Lambda 篩選條件。
有效的 JSON	資料屬性沒有篩選條件模式	Lambda 篩選條件 (僅限其他中繼資料屬性) 會根據您的篩選條件標準而定。
有效的 JSON	非 JSON	Lambda 會在事件來源映射建立或更新時擲回例外狀況。資料屬性的篩選條件模式必須是有效的 JSON 格式。
非 JSON	有效的 JSON	Lambda 捨棄記錄。
非 JSON	資料屬性沒有篩選條件模式	Lambda 篩選條件 (僅限其他中繼資料屬性) 會根據您的篩選條件標準而定。
非 JSON	非 JSON	Lambda 會在事件來源映射建立或更新時擲回例外狀況。資料屬性的篩選條件模式必須是有效的 JSON 格式。

使用 Kinesis 進行篩選

假設生產者將 JSON 格式的資料放入您的 Kinesis 資料串流中。範例記錄如下所示，JSON 資料在 `data` 欄位中會轉換為 Base64 編碼字串。

```
{
    "kinesis": {
        "kinesisSchemaVersion": "1.0",
        "partitionKey": "1",
        "sequenceNumber": "49590338271490256608559692538361571095921575989136588898",
        "data":
 "eyJSZWNvcmROdW1iZXIiOiAiMDAwMSIsICJUaW1lU3RhbXAiOiAieXl5eS1tbS1kZFRoaDptbTpzcyIsICJSZXF1ZXN0Q
        "approximateArrivalTimestamp": 1545084650.987
        },
```

```
    "eventSource": "aws:kinesis",
    "eventVersion": "1.0",
    "eventID":
 "shardId-000000000006:49590338271490256608559692538361571095921575989136588898",
    "eventName": "aws:kinesis:record",
    "invokeIdentityArn": "arn:aws:iam::123456789012:role/lambda-role",
    "awsRegion": "us-east-2",
    "eventSourceARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream"
}
```

只要生產者放入串流中的資料是有效的 JSON，您就可以使用事件篩選透過 `data` 索引鍵來篩選記錄。假設生產者以下列 JSON 格式將記錄放入 Kinesis 串流中。

```
{
    "record": 12345,
    "order": {
        "type": "buy",
        "stock": "ANYCO",
        "quantity": 1000
        }
}
```

若僅篩選訂單類型為「購買」的記錄，`FilterCriteria` 物件如下所示。

```
{
    "Filters": [
        {
            "Pattern": "{ \"data\" : { \"order\" : { \"type\" : [ \"buy\" ] } } }"
        }
    ]
}
```

補充說明，此處是篩選條件的 `Pattern` 在純文字 JSON 中擴展的值。

```
{
    "data": {
        "order": {
            "type": [ "buy" ]
            }
      }
}
```

您可以使用控制台 AWS CLI 或 AWS SAM 模板添加過濾器。

Console

若要使用主控台新增此篩選條件，請遵循 將篩選條件標準連接至事件來源映射 (主控台) 中的指示，並針對篩選條件標準輸入下列字串。

```
{ "data" : { "order" : { "type" : [ "buy" ] } } }
```

AWS CLI

若要使用 AWS Command Line Interface (AWS CLI) 建立具有這些篩選條件的新事件來源對應，請執行下列命令。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:kinesis:us-east-2:123456789012:stream/my-stream \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : { \"order\" : { \"type
\" : [ \"buy\" ] } } }"}]}'
```

若要將這些篩選條件標準新增到現有事件來源映射，請執行下列命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : { \"order\" : { \"type
\" : [ \"buy\" ] } } }"}]}'
```

AWS SAM

若要使用新增此篩選器 AWS SAM，請將下列程式碼片段新增至事件來源的 YAML 範本。

```
FilterCriteria:
  Filters:
    - Pattern: '{ "data" : { "order" : { "type" : [ "buy" ] } } }'
```

若要正確篩選來自 Kinesis 來源的事件，資料欄位和資料欄位的篩選條件標準都必須是有效的 JSON 格式。如果其中一個欄位不是有效的 JSON 格式，則 Lambda 會捨棄訊息或擲回例外狀況。下表摘要說明特定行為：

傳入資料格式	資料屬性的篩選條件模式格式	產生的動作
有效的 JSON	有效的 JSON	根據您的篩選條件標準之 Lambda 篩選條件。
有效的 JSON	資料屬性沒有篩選條件模式	Lambda 篩選條件 (僅限其他中繼資料屬性) 會根據您的篩選條件標準而定。
有效的 JSON	非 JSON	Lambda 會在事件來源映射建立或更新時擲回例外狀況。資料屬性的篩選條件模式必須是有效的 JSON 格式。
非 JSON	有效的 JSON	Lambda 捨棄記錄。
非 JSON	資料屬性沒有篩選條件模式	Lambda 篩選條件 (僅限其他中繼資料屬性) 會根據您的篩選條件標準而定。
非 JSON	非 JSON	Lambda 會在事件來源映射建立或更新時擲回例外狀況。資料屬性的篩選條件模式必須是有效的 JSON 格式。

篩選 Kinesis 彙總記錄

使用 Kinesis，可以將多個記錄彙總到單一 Kinesis 資料串流記錄中，以提高資料輸送量。Lambda 只會在您使用 Kinesis 增強型展開傳送時，將篩選條件標準套用至彙總記錄。不支援使用標準 Kinesis 篩選彙總記錄。使用增強型展開傳送時，您可以設定 Kinesis 專用輸送量取用程式，以充當 Lambda 函數的觸發條件。Lambda 接著會篩選彙總記錄，並僅傳遞符合篩選條件標準的記錄。

若要進一步了解 Kinesis 記錄彙總，請參閱「Kinesis Producer Library (KPL) 主要概念」頁面上的彙總部分。若要進一步了解如何將 Lambda 與 Kinesis 增強型散發搭配使用，請參閱運算部落格上的 Amazon Kinesis Data Streams 增強型散發和 AWS Lambda 來提高即時串流處理效能。 AWS

使用 Amazon MQ 進行篩選

假設您的 Amazon MQ 訊息佇列包含有效 JSON 格式或純字串的訊息。範例記錄如下所示，資料在 `data` 欄位中會轉換為 Base64 編碼字串。

ActiveMQ

```
{
    "messageID": "ID:b-9bcfa592-423a-4942-879d-eb284b418fc8-1.mq.us-
west-2.amazonaws.com-37557-1234520418293-4:1:1:1:1",
    "messageType": "jms/text-message",
    "deliveryMode": 1,
    "replyTo": null,
    "type": null,
    "expiration": "60000",
    "priority": 1,
    "correlationId": "myJMSCoID",
    "redelivered": false,
    "destination": {
      "physicalName": "testQueue"
    },
    "data":"QUJDOkFBQUE=",
    "timestamp": 1598827811958,
    "brokerInTime": 1598827811958,
    "brokerOutTime": 1598827811959,
    "properties": {
      "index": "1",
      "doAlarm": "false",
      "myCustomProperty": "value"
    }
}
```

RabbitMQ

```
{
    "basicProperties": {
        "contentType": "text/plain",
        "contentEncoding": null,
        "headers": {
            "header1": {
                "bytes": [
                  118,
```

```
                    97,
                    108,
                    117,
                    101,
                    49
                ]
            },
            "header2": {
                "bytes": [
                    118,
                    97,
                    108,
                    117,
                    101,
                    50
                ]
            },
            "numberInHeader": 10
        },
        "deliveryMode": 1,
        "priority": 34,
        "correlationId": null,
        "replyTo": null,
        "expiration": "60000",
        "messageId": null,
        "timestamp": "Jan 1, 1970, 12:33:41 AM",
        "type": null,
        "userId": "AIDACKCEVSQ6C2EXAMPLE",
        "appId": null,
        "clusterId": null,
        "bodySize": 80
        },
    "redelivered": false,
    "data": "eyJ0aW1lb3V0IjowLCJkYXRhIjoiQ1pybWYwR3c4T3Y0YnFMUXhENEUifQ=="
}
```

對於 Active MQ 和 Rabbit MQ 代理程式，您可以使用事件篩選透過 `data` 索引鍵來篩選記錄。假設您的 Amazon MQ 佇列包含以下 JSON 格式的訊息。

```
{
    "timeout": 0,
```

```
    "IPAddress": "203.0.113.254"
}
```

若要僅篩選 timeout 欄位大於 0 的記錄，FilterCriteria 物件將如下所示。

```
{
    "Filters": [
        {
            "Pattern": "{ \"data\" : { \"timeout\" : [ { \"numeric\": [ \">\",
 0] } } ] } }"
        }
    ]
}
```

補充說明，此處是篩選條件的 Pattern 在純文字 JSON 中擴展的值。

```
{
    "data": {
        "timeout": [ { "numeric": [ ">", 0 ] } ]
        }
}
```

您可以使用控制台 AWS CLI 或 AWS SAM 模板添加過濾器。

Console

若要使用主控台新增此篩選條件，請遵循 將篩選條件標準連接至事件來源映射 (主控台) 中的指示，並針對篩選條件標準輸入下列字串。

```
{ "data" : { "timeout" : [ { "numeric": [ ">", 0 ] } ] } }
```

AWS CLI

若要使用 AWS Command Line Interface (AWS CLI) 建立具有這些篩選條件的新事件來源對應，請執行下列命令。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:mq:us-east-2:123456789012:broker:my-
broker:b-8ac7cc01-5898-482d-be2f-a6b596050ea8 \
```

```
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : { \"timeout\" :
 [ { \"numeric\": [ \">\", 0 ] } ] } }"}]}'
```

若要將這些篩選條件標準新增到現有事件來源映射，請執行下列命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : { \"timeout\" :
 [ { \"numeric\": [ \">\", 0 ] } ] } }"}]}'
```

AWS SAM

若要使用新增此篩選器 AWS SAM，請將下列程式碼片段新增至事件來源的 YAML 範本。

```
FilterCriteria:
  Filters:
    - Pattern: '{ "data" : { "timeout" : [ { "numeric": [ ">", 0 ] } ] } }'
```

使用 Amazon MQ，您也可以篩選訊息為純字串的記錄。假設您只想處理訊息以「結果：」開頭的記錄。`FilterCriteria` 物件如下所示。

```
{
    "Filters": [
        {
            "Pattern": "{ \"data\" : [ { \"prefix\": \"Result: \" } ] }"
        }
    ]
}
```

補充說明，此處是篩選條件的 `Pattern` 在純文字 JSON 中擴展的值。

```
{
    "data": [
        {
        "prefix": "Result: "
        }
    ]
}
```

您可以使用控制台 AWS CLI 或 AWS SAM 模板添加過濾器。

Console

若要使用主控台新增此篩選條件，請遵循 將篩選條件標準連接至事件來源映射 (主控台) 中的指示，並針對篩選條件標準輸入下列字串。

```
{ "data" : [ { "prefix": "Result: " } ] }
```

AWS CLI

若要使用 AWS Command Line Interface (AWS CLI) 建立具有這些篩選條件的新事件來源對應，請執行下列命令。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:mq:us-east-2:123456789012:broker:my-
broker:b-8ac7cc01-5898-482d-be2f-a6b596050ea8 \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : [ { \"prefix\":
 \"Result: \" } ] }"}]}'
```

若要將這些篩選條件標準新增到現有事件來源映射，請執行下列命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : [ { \"prefix\":
 \"Result: \" } ] }"}]}'
```

AWS SAM

若要使用新增此篩選器 AWS SAM，請將下列程式碼片段新增至事件來源的 YAML 範本。

```
FilterCriteria:
  Filters:
    - Pattern: '{ "data" : [ { "prefix": "Result " } ] }'
```

Amazon MQ 訊息必須是 UTF-8 編碼的字串，可以是純字串或 JSON 格式。這是因為 Lambda 會在套用篩選條件之前，將 Amazon MQ 位元組陣列解碼成 UTF-8。如果您的訊息使用其他編碼方式 (例如 UTF-16 或 ASCII)，或者訊息格式與 `FilterCriteria` 格式不相符，則 Lambda 只會處理中繼資料篩選條件。下表摘要說明特定行為：

傳入訊息 格式	訊息屬性的篩選條件模式格式	產生的動作
純文字的字串	純文字的字串	根據您的篩選條件標準之 Lambda 篩選條件。
純文字的字串	資料屬性沒有篩選條件模式	Lambda 篩選條件 (僅限其他中繼資料屬性) 會根據您的篩選條件標準而定。
純文字的字串	有效的 JSON	Lambda 篩選條件 (僅限其他中繼資料屬性) 會根據您的篩選條件標準而定。
有效的 JSON	純文字的字串	Lambda 篩選條件 (僅限其他中繼資料屬性) 會根據您的篩選條件標準而定。
有效的 JSON	資料屬性沒有篩選條件模式	Lambda 篩選條件 (僅限其他中繼資料屬性) 會根據您的篩選條件標準而定。
有效的 JSON	有效的 JSON	根據您的篩選條件標準之 Lambda 篩選條件。
非 UTF-8 編碼字串	JSON、純字串或沒有模式	Lambda 篩選條件 (僅限其他中繼資料屬性) 會根據您的篩選條件標準而定。

使用 Amazon MSK 和自我管理的 Apache Kafka 進行篩選

假設生產者正在將訊息寫入 Amazon MSK 或自我管理的 Apache Kafka 叢集中的某個主題，可以是有效的 JSON 格式或純字串。範例記錄如下所示，訊息在 `value` 欄位中會轉換為 Base64 編碼字串。

```
{
    "mytopic-0":[
        {
            "topic":"mytopic",
            "partition":0,
            "offset":15,
```

```
            "timestamp":1545084650987,
            "timestampType":"CREATE_TIME",
            "value":"SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
            "headers":[]
        }
    ]
}
```

假設 Apache Kafka 生產者正在以下列 JSON 格式將訊息寫入到您的主題。

```
{
    "device_ID": "AB1234",
    "session":{
        "start_time": "yyyy-mm-ddThh:mm:ss",
        "duration": 162
    }
}
```

您可以使用 `value` 索引鍵來篩選記錄。假設您只想篩選 `device_ID` 以字母 AB 開頭的記錄。`FilterCriteria` 物件如下所示。

```
{
    "Filters": [
        {
            "Pattern": "{ \"value\" : { \"device_ID\" : [ { \"prefix\": \"AB\" } ] } }"
        }
    ]
}
```

補充說明，此處是篩選條件的 `Pattern` 在純文字 JSON 中擴展的值。

```
{
    "value": {
        "device_ID": [ { "prefix": "AB" } ]
      }
}
```

您可以使用控制台 AWS CLI 或 AWS SAM 模板添加過濾器。

Console

若要使用主控台新增此篩選條件，請遵循 將篩選條件標準連接至事件來源映射 (主控台) 中的指示，並針對篩選條件標準輸入下列字串。

```
{ "value" : { "device_ID" : [ { "prefix":  "AB" } ] } }
```

AWS CLI

若要使用 AWS Command Line Interface (AWS CLI) 建立具有這些篩選條件的新事件來源對應，請執行下列命令。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:kafka:us-east-2:123456789012:cluster/my-cluster/
b-8ac7cc01-5898-482d-be2f-a6b596050ea8 \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"value\" : { \"device_ID\" :
 [ { \"prefix\":  \"AB\" } ] } }"}]}'
```

若要將這些篩選條件標準新增到現有事件來源映射，請執行下列命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"value\" : { \"device_ID\" :
 [ { \"prefix\":  \"AB\" } ] } }"}]}'
```

AWS SAM

若要使用新增此篩選器 AWS SAM，請將下列程式碼片段新增至事件來源的 YAML 範本。

```
FilterCriteria:
  Filters:
    - Pattern: '{ "value" : { "device_ID" : [ { "prefix":  "AB" } ] } }'
```

使用 Amazon MSK 和自我管理的 Apache Kafka，您也可以篩選訊息為純字串的記錄。假設您想忽略字串為「錯誤」的訊息。`FilterCriteria` 物件如下所示。

```
{
    "Filters": [
```

```
        {
            "Pattern": "{ \"value\" : [ { \"anything-but\": [ \"error\" ] } ] }"
        }
    ]
}
```

補充說明，此處是篩選條件的 Pattern 在純文字 JSON 中擴展的值。

```
{
    "value": [
        {
        "anything-but": [ "error" ]
        }
    ]
}
```

您可以使用控制台 AWS CLI 或 AWS SAM 模板添加過濾器。

Console

若要使用主控台新增此篩選條件，請遵循 將篩選條件標準連接至事件來源映射 (主控台) 中的指示，並針對篩選條件標準輸入下列字串。

```
{ "value" : [ { "anything-but": [ "error" ] } ] }
```

AWS CLI

若要使用 AWS Command Line Interface (AWS CLI) 建立具有這些篩選條件的新事件來源對應，請執行下列命令。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:kafka:us-east-2:123456789012:cluster/my-cluster/
b-8ac7cc01-5898-482d-be2f-a6b596050ea8 \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"value\" : [ { \"anything-but\":
 [ \"error\" ] } ] }"}]}'
```

若要將這些篩選條件標準新增到現有事件來源映射，請執行下列命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
```

```
    --filter-criteria '{"Filters": [{"Pattern": "{ \"value\" : [ { \"anything-but\":
 [ \"error\" ] } ] }"}]}'
```

AWS SAM

若要使用新增此篩選器 AWS SAM，請將下列程式碼片段新增至事件來源的 YAML 範本。

```
FilterCriteria:
  Filters:
    - Pattern: '{ "value" : [ { "anything-but": [ "error" ] } ] }'
```

Amazon MSK 和自我管理的 Apache Kafka 訊息必須是 UTF-8 編碼的字串，可以是純字串或 JSON 格式。這是因為 Lambda 會在套用篩選條件之前，將 Amazon MSK 位元組陣列解碼成 UTF-8。如果您的訊息使用其他編碼方式 (例如 UTF-16 或 ASCII)，或者訊息格式與 `FilterCriteria` 格式不相符，則 Lambda 只會處理中繼資料篩選條件。下表摘要說明特定行為：

傳入訊息 格式	訊息屬性的篩選條件模式格式	產生的動作
純文字的字串	純文字的字串	根據您的篩選條件標準之 Lambda 篩選條件。
純文字的字串	資料屬性沒有篩選條件模式	Lambda 篩選條件 (僅限其他中繼資料屬性) 會根據您的篩選條件標準而定。
純文字的字串	有效的 JSON	Lambda 篩選條件 (僅限其他中繼資料屬性) 會根據您的篩選條件標準而定。
有效的 JSON	純文字的字串	Lambda 篩選條件 (僅限其他中繼資料屬性) 會根據您的篩選條件標準而定。
有效的 JSON	資料屬性沒有篩選條件模式	Lambda 篩選條件 (僅限其他中繼資料屬性) 會根據您的篩選條件標準而定。
有效的 JSON	有效的 JSON	根據您的篩選條件標準之 Lambda 篩選條件。

傳入訊息 格式	訊息屬性的篩選條件模式格式	產生的動作
非 UTF-8 編碼字串	JSON、純字串或沒有模式	Lambda 篩選條件 (僅限其他中繼資料屬性) 會根據您的篩選條件標準而定。

使用 Amazon SQS 進行篩選

假設您的 Amazon SQS 佇列包含以下 JSON 格式的訊息。

```
{
    "RecordNumber": 0000,
    "TimeStamp": "yyyy-mm-ddThh:mm:ss",
    "RequestCode": "AAAA"
}
```

此佇列的範例記錄如下所示。

```
{
    "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
    "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",
    "body": "{\n "RecordNumber": 0000,\n "TimeStamp": "yyyy-mm-ddThh:mm:ss",\n
 "RequestCode": "AAAA"\n}",
    "attributes": {
        "ApproximateReceiveCount": "1",
        "SentTimestamp": "1545082649183",
        "SenderId": "AIDAIENQZJOLO23YVJ4VO",
        "ApproximateFirstReceiveTimestamp": "1545082649185"
        },
    "messageAttributes": {},
    "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
    "eventSource": "aws:sqs",
    "eventSourceARN": "arn:aws:sqs:us-west-2:123456789012:my-queue",
    "awsRegion": "us-west-2"
}
```

若要根據 Amazon SQS 訊息的內容進行篩選，請使用 Amazon SQS 訊息記錄中的 body 金鑰。假設您只想處理 Amazon SQS 訊息中的 RequestCode 為 "BBBB" 的記錄。FilterCriteria 物件如下所示。

```
{
    "Filters": [
        {
            "Pattern": "{ \"body\" : { \"RequestCode\" : [ \"BBBB\" ] } }"
        }
    ]
}
```

補充說明，此處是篩選條件的 Pattern 在純文字 JSON 中擴展的值。

```
{
    "body": {
        "RequestCode": [ "BBBB" ]
        }
}
```

您可以使用控制台 AWS CLI 或 AWS SAM 模板添加過濾器。

Console

若要使用主控台新增此篩選條件，請遵循 將篩選條件標準連接至事件來源映射 (主控台) 中的指示，並針對篩選條件標準輸入下列字串。

```
{ "body" : { "RequestCode" : [ "BBBB" ] } }
```

AWS CLI

若要使用 AWS Command Line Interface (AWS CLI) 建立具有這些篩選條件的新事件來源對應，請執行下列命令。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:sqs:us-east-2:123456789012:my-queue \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"body\" : { \"RequestCode\" :
 [ \"BBBB\" ] } }"}]}'
```

若要將這些篩選條件標準新增到現有事件來源映射，請執行下列命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
```

```
    --filter-criteria '{"Filters": [{"Pattern": "{ \"body\" : { \"RequestCode\" :
 [ \"BBBB\" ] } }"}]}'
```

AWS SAM

若要使用新增此篩選器 AWS SAM，請將下列程式碼片段新增至事件來源的 YAML 範本。

```
FilterCriteria:
  Filters:
    - Pattern: '{ "body" : { "RequestCode" : [ "BBBB" ] } }'
```

假設您希望函數僅處理 `RecordNumber` 大於 9999 的記錄。`FilterCriteria` 物件如下所示。

```
{
    "Filters": [
        {
            "Pattern": "{ \"body\" : { \"RecordNumber\" : [ { \"numeric\": [ \">\",
 9999 ] } ] } }"
        }
    ]
}
```

補充說明，此處是篩選條件的 `Pattern` 在純文字 JSON 中擴展的值。

```
{
    "body": {
        "RecordNumber": [
            {
                "numeric": [ ">", 9999 ]
            }
        ]
    }
}
```

您可以使用控制台 AWS CLI 或 AWS SAM 模板添加過濾器。

Console

若要使用主控台新增此篩選條件，請遵循 將篩選條件標準連接至事件來源映射 (主控台) 中的指示，並針對篩選條件標準輸入下列字串。

```
{ "body" : { "RecordNumber" : [ { "numeric": [ ">", 9999 ] } ] } }
```

AWS CLI

若要使用 AWS Command Line Interface (AWS CLI) 建立具有這些篩選條件的新事件來源對應，請執行下列命令。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:sqs:us-east-2:123456789012:my-queue \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"body\" : { \"RecordNumber\" :
 [ { \"numeric\": [ \">\", 9999 ] } ] } }"}]}'
```

若要將這些篩選條件標準新增到現有事件來源映射，請執行下列命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"body\" : { \"RecordNumber\" :
 [ { \"numeric\": [ \">\", 9999 ] } ] } }"}]}'
```

AWS SAM

若要使用新增此篩選器 AWS SAM，請將下列程式碼片段新增至事件來源的 YAML 範本。

```
FilterCriteria:
  Filters:
    - Pattern: '{ "body" : { "RecordNumber" : [ { "numeric": [ ">", 9999 ] } ] } }'
```

針對 Amazon SQS，訊息內文可以是任何字串。但是，如果您的 `FilterCriteria` 預期 body 為有效的 JSON 格式，這就會造成問題。反之亦然 — 如果傳入的訊息內文是有效的 JSON 格式，但您的選條件標準預期 body 應為純字串，則此可能會導致意外的行為。

為了避免此問題，請確定 `FilterCriteria` 中的內文格式與從佇列收到的訊息中的 body 之預期格式相符。篩選訊息之前，Lambda 會自動評估傳入訊息內文之格式和 body 的篩選條件模式之格式。如果有不相符的情形，Lambda 就會捨棄訊息。下表摘要說明此評估：

傳入訊息 **body** 格式	篩選條件模式 **body** 格式	產生的動作
純文字的字串	純文字的字串	根據您的篩選條件標準之 Lambda 篩選條件。
純文字的字串	資料屬性沒有篩選條件模式	Lambda 篩選條件 (僅限其他中繼資料屬性) 會根據您的篩選條件標準而定。
純文字的字串	有效的 JSON	Lambda 捨棄訊息。
有效的 JSON	純文字的字串	Lambda 捨棄訊息。
有效的 JSON	資料屬性沒有篩選條件模式	Lambda 篩選條件 (僅限其他中繼資料屬性) 會根據您的篩選條件標準而定。
有效的 JSON	有效的 JSON	根據您的篩選條件標準之 Lambda 篩選條件。

在主控台中，測試 Lambda 函數。

您可以藉由使用測試事件調用函數，在主控台中測試您的 Lambda 函數。一個測試事件是函數的 JSON 輸入。如果您的函數不需要輸入，則該事件可以是空白的文件（{}）。

當您在主控台執行測試時，Lambda 會與測試事件同步調用函數。函數執行期會將 JSON 事件轉換為物件，並將其傳遞給程式碼的處理常式方法以進行處理。

> **建立測試事件**
>
> 您必須先建立私有或可共用的測試事件，才能在主控台中測試。

使用測試事件調用函數

若要測試函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇要測試的函數名稱。
3. 選擇測試標籤。
4. 在測試事件下，選擇建立新事件或編輯已儲存事件，然後選擇您要使用的已儲存事件。
5. (選擇性) - 選擇事件 JSON 的範本。
6. 選擇 測試。
7. 若要檢閱測試結果，在 Execution result (執行結果) 下，展開 Details (詳細資訊)。

若要在不儲存測試事件的情況下調用您的函數，選擇測試，然後再儲存。這會建立一個未儲存的測試事件，而 Lambda 僅會在工作階段期間保留該事件。

您也可以從程式碼索引標籤存取已儲存和未儲存的測試事件。在該處，選擇測試，然後選擇測試事件。

建立私有測試事件

只有事件建立者才能使用私有測試事件，且其不需要額外的許可即可使用。每個函數您可以建立並儲存最多 10 個私有測試事件。

若要建立私有測試事件

1. 開啟 Lambda 主控台中的 函數頁面。

2. 選擇要測試的函數名稱。
3. 選擇測試標籤。
4. 在 Test event (測試事件) 下，執行以下動作：
    a. 選擇 Template (範本)。
    b. 輸入該測試的 Name (名稱)。
    c. 在文字輸入方塊中，輸入 JSON 測試事件。
    d. 在 Event sharing settings (事件共用設定) 下，選擇 Private (私有)。
5. 選擇儲存變更。

您也可以在 Code (程式碼) 索引標籤上建立新的測試事件。在該處，選擇 Test (測試)、Configure test event (配置測試事件)。

建立可共用測試事件

可共用測試事件是您可以與相同 AWS 帳戶中其他使用者共用的測試事件。您可以編輯其他使用者的可共用測試事件，並使用它們來調用您的函數。

Lambda 會將可共用的測試事件儲存為結構描述，儲存在名`lambda-testevent-schemas`為的 Amazon EventBridge (CloudWatch 事件) 架構登錄中 由於 Lambda 利用此登錄檔來存放和呼叫您建立的可共用測試事件，我們建議您不要編輯此登錄檔或使用 `lambda-testevent-schemas` 名稱建立登錄檔。

若要查看、共用和編輯可共用的測試事件，您必須擁有下列所有 EventBridge (E CloudWatch vents) 結構描述登錄 API 作業的權限：

- `schemas.CreateRegistry`
- `schemas.CreateSchema`
- `schemas.DeleteSchema`
- `schemas.DeleteSchemaVersion`
- `schemas.DescribeRegistry`
- `schemas.DescribeSchema`
- `schemas.GetDiscoveredSchema`
- `schemas.ListSchemaVersions`
- `schemas.UpdateSchema`

請注意，儲存對可共用測試事件所做的編輯將覆蓋該事件。

如果您無法建立、編輯或查看可共用測試事件，請檢查您的帳戶是否具有執行這些操作所需要的許可。如果您擁有必要的權限，但仍無法存取可共用的測試事件，請檢查是否有任何可能限制存取 EventBridge (E CloudWatch vents) 登錄的資源型政策。

若要建立可共用測試事件

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇要測試的函數名稱。
3. 選擇測試標籤。
4. 在 Test event (測試事件) 下，執行以下動作：
   a. 選擇 Template (範本)。
   b. 輸入該測試的 Name (名稱)。
   c. 在文字輸入方塊中，輸入 JSON 測試事件。
   d. 在 Event sharing settings (事件共用設定) 下，選擇 Shareable (可共用)。
5. 選擇儲存變更。

> 搭配 AWS Serverless Application Model 使用可共用的測試事件。
>
> 您可以使用 AWS SAM 來調用可共享測試事件。請參閱《AWS Serverless Application Model 開發人員指南》中的 `sam remote test-event`。

刪除可共用測試事件結構描述

當您刪除可共用測試事件時，Lambda 會將其從 `lambda-testevent-schemas` 登錄檔中移除。如果您是從登錄檔移除最後一個可共用測試事件，則 Lambda 會刪除登錄檔。

如果刪除函數，Lambda 不會刪除任何關聯的可共用測試事件結構描述。您必須從 EventBridge (CloudWatch 事件) 主控台手動清除這些資源。

Lambda 函數狀態

Lambda 會在所有函數的函數組態中包含狀態欄位，以指出函數何時可以叫用。State 提供函數目前狀態的相關資訊，包括是否可以成功叫用函數。函數狀態不會變更函數叫用的行為或函數執行程式碼的方式。函數狀態包括：

- Pending – Lambda 建立函數之後，會將狀態設定為待定中。處於待定狀態時，Lambda 會嘗試建立或設定函數的資源，例如 VPC 或 EFS 資源。Lambda 不會叫用待定狀態期間的函數。在函數上操作的任何叫用或其他 API 操作都會失敗。
- Active – Lambda 完成資源組態和佈建之後，您的函數會轉換為啟用中狀態。只有啟用中的函數才能成功叫用。
- Failed – 表示資源配置或佈建發生錯誤。
- Inactive – 當函數空閒足夠長的時間，以至於 Lambda 回收為其配置的外部資源時，該函數將變為非啟用狀態。當您嘗試叫用非啟用中的函數，叫用會失敗而 Lambda 會將函數設定為待定狀態，直到重新建立函數資源為止。如果 Lambda 無法重新建立資源，則函數會返回非作用中狀態。如果您的函數處於非活動狀態，請參閱函數StatusCode和StatusCodeReason屬性以進一步解決疑難排解。您可能需要解決任何錯誤，並重新部署您的函數以將其恢復到活動狀態。

如果您是使用 SDK 型自動化工作流程或是直接呼叫 Lambda 的服務 API，請務必在調用之前檢查函數的狀態，以驗證其是否處於作用中狀態。您可以使用 Lambda API 動作來執行此操作 GetFunction，或使用適用於 Java 2.0 的AWS SDK 來設定服務員。

```
aws lambda get-function --function-name my-function --query 'Configuration.[State,
 LastUpdateStatus]'
```

您應該會看到下列輸出：

```
[
 "Active",
 "Successful"
]
```

當函數的建立處於待定狀態時，下列作業會失敗：

- Invoke
- UpdateFunctionCode
- UpdateFunctionConfiguration

- PublishVersion

更新時的函數狀態

Lambda 使用 LastUpdateStatus 屬性為正在進行更新的函數提供其他上下文，此屬性可以具有以下狀態：

- InProgress – 現有函數正在進行更新。正在進行函數更新時，叫用會移至函數先前的程式碼和組態。
- Successful – 更新已完成。Lambda 完成更新後，將保持此狀態，直到進一步更新為止。
- Failed – 函數更新失敗。Lambda 會中止更新，並且函數的先前程式碼和配置仍可供使用。

Example

以下是對正在進行更新的函數執行 get-function-configuration 的結果。

```
{
    "FunctionName": "my-function",
    "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
    "Runtime": "nodejs20.x",
    "VpcConfig": {
        "SubnetIds": [
            "subnet-071f712345678e7c8",
            "subnet-07fd123456788a036",
            "subnet-0804f77612345cacf"
        ],
        "SecurityGroupIds": [
            "sg-085912345678492fb"
        ],
        "VpcId": "vpc-08e1234569e011e83"
    },
    "State": "Active",
    "LastUpdateStatus": "InProgress",
    ...
}
```

FunctionConfiguration 有 LastUpdateStatusReason 和 LastUpdateStatusReasonCode 兩個其他屬性，以協助進行更新方面的故障診斷。

當非同步更新正在進行時，下列作業會失敗：

- UpdateFunctionCode
- UpdateFunctionConfiguration
- PublishVersion
- TagResource

了解 Lambda 中的重試行為

直接叫用函數時，您可以決定處理函數程式碼相關錯誤的策略。Lambda 不會自動代您重試這類錯誤。若要進行重試，您可以手動重新叫用函數、將失敗的事件傳送到佇列進行偵錯，或忽略錯誤。您函式的程式碼可能已完全執行、部分執行或完全未執行。如果您重試，請確保您函式的程式碼可處理相同的事件很多次，而不會導致重複的交易或其他不想要的副作用。

當您間接叫用函式時，您需要留意叫用端的重試行為及任何遭遇請求的服務。這包括以下案例。

- Asynchronous invocation (非同步叫用) - Lambda 會重試函數錯誤兩次。如果函式沒有足夠的容量來處理所有傳入的請求，事件可能在佇列中等待數小時或數天才會傳送到函式。您可以在函式上設定無效信件佇列，以擷取未成功處理的事件。如需詳細資訊，請參閱 非同步調用。
- Event source mappings (事件來源映射) - 從串流中讀取的事件來源映射會重試整批項目。在錯誤解決或項目過期前，重複的錯誤會阻礙受影響碎片的處理。若要偵測已停滯的碎片，您可以監控反覆運算器年齡指標。

  對於從佇列讀取的事件來源映射，您可藉由設定來源佇列的可見性逾時和再驅動政策，決定重試之間的時間長度和失敗事件的目的地。如需詳細資訊，請參閱 Lambda 如何處理串流和以佇列為基礎的事件來源的記錄 和 使用來自其 AWS 他服務的事件叫用 Lambda 下的服務特定主題。
- AWS 服務- AWS 服務可以同步或異步調用您的功能。對於同步叫用，服務會決定是否要重試。例如，如果 Lambda 函數傳回 `TemporaryFailure` 回應代碼，則 Amazon S3 批次操作會重試該操作。來自上游使用者或用戶端的 Proxy 要求的服務可能有重試策略，或可能會將錯誤回應轉送回要求者。例如，API Gateway 一律會將錯誤回應轉送回要求者。

  對於非同步叫用，行為與您以同步方式叫用函式時相同。如需詳細資訊，請參閱 使用來自其 AWS 他服務的事件叫用 Lambda 之下的服務特定主題和叫用服務的文件。
- Other accounts and clients (其他帳戶和用戶端) - 當您授與其他帳戶的存取權時，您可使用以資源為基礎的政策來限制其可設定的服務或資源，以叫用您的函數。為了保護您的函數以免過載，請考慮透過 Amazon API Gateway 在您的函數前面放置 API 層。

為了協助您處理 Lambda 應用程式中的錯誤，Lambda 會與 Amazon CloudWatch 和 AWS X-Ray. 您可以使用日誌、指標、警示和追蹤的組合，快速偵測及識別您的函式程式碼、API 或其他支援您應用程式的資源中的問題。如需更多詳細資訊，請參閱 監控與疑難排解 Lambda 函數。

使用 Lambda 遞迴迴路偵測來防止無限迴圈

當您將 Lambda 函數設定為輸出至調用該函數的相同服務或資源時，就可以建立無限遞迴迴圈。例如，Lambda 函數可能會將訊息寫入 Amazon Simple Queue Service (Amazon SQS) 佇列，然後調用相同的函數。此調用會導致函數將另一則訊息寫入佇列，進而再次調用函數。

無意的遞迴迴圈可能會導致您的. AWS 帳戶迴圈也可能導致 Lambda 擴展和使用您帳戶的所有可用並行處理。為了減少意外迴圈的影響，Lambda 可以在特定類型的遞迴迴圈發生後不久偵測到它們。當 Lambda 偵測到遞迴迴圈時，會停止調用函數並通知您。

如果您的設計故意使用遞迴模式，則可以請求關閉 Lambda 遞迴迴圈偵測。若要請求此變更，請聯絡 AWS Support。

> **Important**
>
> 如果您的設計故意使用 Lambda 函數將資料寫回呼叫函數的相同 AWS 資源，請小心並實作適當的防護欄，以防止向您 AWS 帳戶計費。若要進一步了解使用遞迴調用模式的最佳實務，請參閱無伺服器園地中的 導致 Lambda 函數失控的遞迴模式。

章節

了解遞迴迴圈偵測

Lambda 中的遞迴迴圈偵測透過追蹤事件來運作。Lambda 是事件驅動型運算服務，可在特定事件發生時執行函數程式碼。例如，將項目新增至 Amazon SQS 佇列或 Amazon Simple Notification Service (Amazon SNS) 主題時。Lambda 會將事件以 JSON 物件的形式傳遞至函數，其中包含系統狀態中的變更資訊。當事件導致函數執行時，這稱為調用。

若要偵測遞迴迴圈，Lambda 會使用 AWS X-Ray 追蹤標頭。當支援遞迴迴圈偵測的AWS 服務 將事件傳送至 Lambda 時，這些事件會自動使用中繼資料進行註解。當您的 Lambda 函數 AWS 服務 使用支援的 AWS SDK 版本將這些事件之一寫入另一個支援的事件時，它會更新此中繼資料。更新後的中繼資料包含事件調用函數的次數計數。

> **Note**
>
> 您不需要啟用 X-Ray 作用中追蹤，即可使用此功能。依預設，所有 AWS 客戶都會開啟遞迴迴路偵測。使用此功能無需支付任何費用。

請求鏈是由相同觸發事件引起的一系列 Lambda 調用。例如，假設 Amazon SQS 佇列調用 Lambda 函數。然後，Lambda 函數會將已處理的事件傳回相同的 Amazon SQS 佇列，然後再次調用函數。在此範例中，函數的每次調用都屬於相同的請求鏈。

如果您的函數在相同的請求鏈中被調用超過 16 次，則 Lambda 會自動停止該請求鏈中的下一個函數調用並通知您。如果函數設有多個觸發條件，來自其他觸發條件的調用不會受影響。

> **Note**
>
> 當來源佇列再驅動政策的 `maxReceiveCount` 設定高於 16 時，Lambda 遞迴保護不會阻止 Amazon SQS 在偵測到遞迴迴圈並終止後重試訊息。當 Lambda 偵測到遞迴迴圈並捨棄後續調用時，會將 `RecursiveInvocationException` 傳回至事件來源映射。這會增加訊息上的`receiveCount`值。Lambda 會繼續重試訊息，並繼續封鎖函數叫用，直到 Amazon SQS 判斷超過，並將訊息傳送到設定的無效字母佇列為止。`maxReceiveCount`

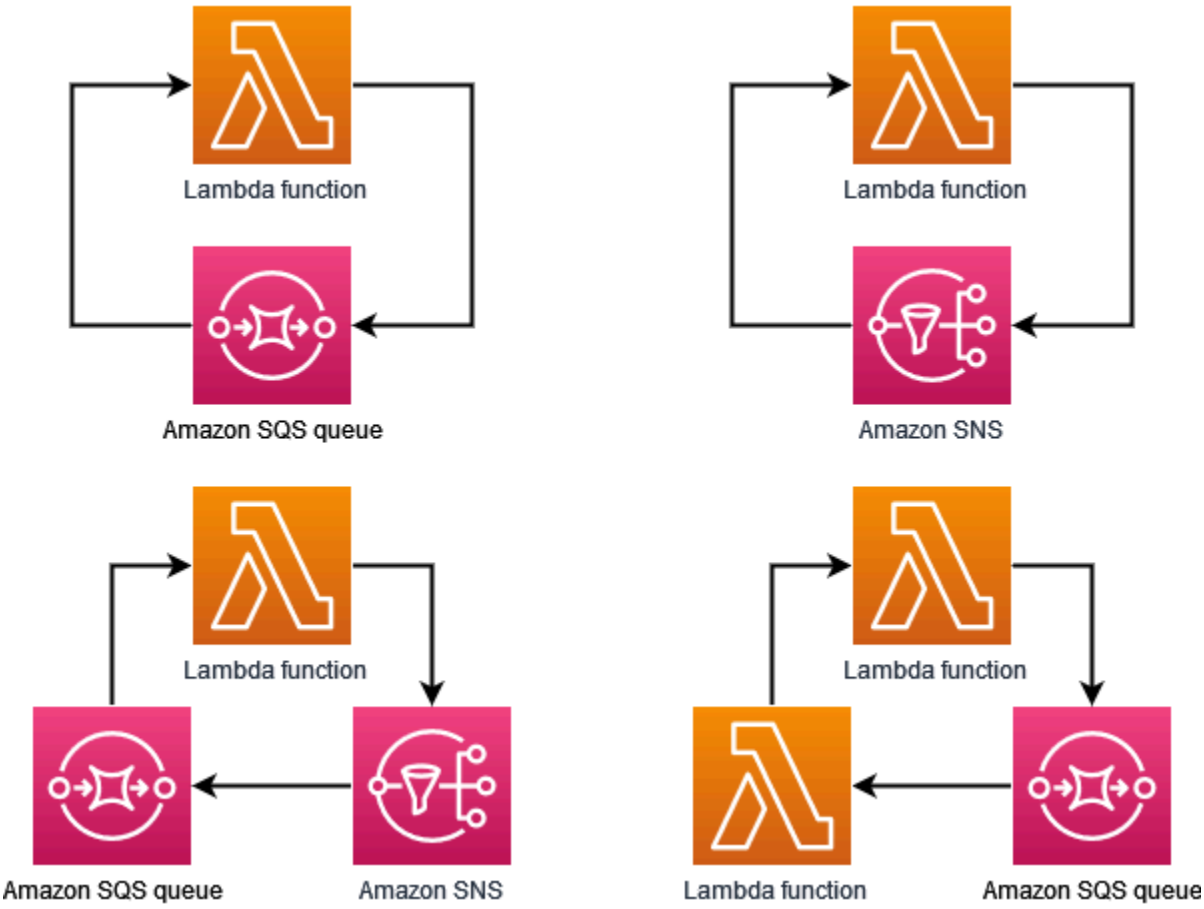
如果您為函數設定了故障時的目的地或無效字母佇列，則 Lambda 也會將事件從已停止的調用中傳送至目的地或無效字母佇列。為函數設定目的地或無效字母佇列時，請勿使用 Amazon SNS 主題或 Amazon SQS 佇列，因為函數也會將這兩項當成事件觸發條件或事件來源映射使用。如果將事件傳送到調用函數的相同資源，則可以建立另一個遞迴迴圈。

支援 AWS 服務 的軟體開發套件

Lambda 只能偵測包含某些受支援的遞迴迴圈 AWS 服務。若要偵測到遞迴迴圈，您的函式也必須使用其中一個支援的 AWS SDK。

支援 AWS 服務

Lambda 目前可偵測函數、Amazon SQS 和 Amazon SNS 之間的遞迴迴圈。Lambda 也會偵測僅由 Lambda 函數組成的迴圈，這些函數可以同步或非同步相互調用。下圖顯示 Lambda 可以偵測的一些迴圈範例：



當另一個 AWS 服務 例如 Amazon DynamoDB 或亞馬遜簡單儲存服務 (Amazon S3) 形成迴圈的一部分時，Lambda 目前無法偵測並停止它。

由於 Lambda 目前只偵測涉及 Amazon SQS 和 Amazon SNS 的遞迴迴圈，因此涉及其他項目的迴圈仍有 AWS 服務 可能導致您的 Lambda 函數意外使用。

為了防止向您收取意外費用 AWS 帳戶，我們建議您設定 Amazon CloudWatch 警示以提醒您不尋常的使用模式。例如，您可以設定 CloudWatch 為通知您 Lambda 函數並行或叫用中的尖峰情況。也可以設定帳單警示，當帳戶中的支出超過指定的閾值時通知您。或者，可以使用 AWS Cost Anomaly Detection 來提醒您不尋常的帳單模式。

支援的 AWS 開發套件

若要讓 Lambda 偵測遞迴迴圈，函數必須使用下列其中一個 SDK 版本或更高版本：

執行期	所需的最低 AWS SDK 版本
Node.js	2.1147.0 (SDK 版本 2)

執行期	所需的最低 AWS SDK 版本
	3.105.0 (SDK 版本 3)
Python	1.24.46 (boto3) 1.27.46 (botocore)
Java 8 和 Java 11	1.12.200 (SDK 版本 1) 2.17.135 (SDK 版本 2)
Java 17	2.20.81
Java 21	2.21.24
.NET	3.7.293.0
Ruby	3.134.0
PHP	3.232.0

某些 Lambda 執行階段 (例如 Python 和 Node.js) 包含 AWS 開發套件的版本。如果函數執行期中包含的 SDK 版本低於所需的最低版本，則可以將支援的 SDK 版本新增到函數的部署套件。您也可以使用 Lambda 層將支援的 SDK 版本新增至函數。如需每個 Lambda 執行期包含的 SDK 清單，請參閱 Lambda 執行期。

Lambda Go 執行期不支援 Lambda 遞迴偵測。

遞迴迴圈通知

當 Lambda 停止遞迴迴圈時，您會透過 AWS Health Dashboard 和電子郵件接收通知。您也可以使用 CloudWatch 指標來監控 Lambda 已停止的遞迴叫用次數。

AWS Health Dashboard 通知

當 Lambda 停止遞迴叫用時，會在 [您的帳戶健全狀況] 頁面上的 [開啟和最近的問題] 下方 AWS Health Dashboard 顯示通知。請注意，在 Lambda 停止遞迴調用後，可能需要三個小時才會顯示此通知。如需有關在「」中檢視帳戶事件的詳細資訊 AWS Health Dashboard，請參閱「 AWS Health 使用者指南」中的「開始使用 Health 儀表板 — 您的帳戶AWS健康狀況」。

電子郵件提醒

當 Lambda 第一次停止函數的遞迴調用時，會傳送電子郵件提醒給您。對於 AWS 帳戶中的每個函數，Lambda 每 24 小時最多傳送一封電子郵件。Lambda 傳送電子郵件通知後，即使 Lambda 停止函數的進一步遞迴調用，在接下來的 24 小時也不會再收到該函數的任何電子郵件。請注意，在 Lambda 停止遞迴調用後，可能需要三個小時才會收到此電子郵件提醒。

Lambda 會傳送遞迴圈電子郵件警示給您 AWS 帳戶的主要客戶聯絡人和備用營運聯絡人。如需有關檢視或更新帳戶中電子郵件地址的資訊，請參閱《AWS 一般參考》中的更新聯絡資訊。

Amazon CloudWatch 指標

此指CloudWatch 標會`RecursiveInvocationsDropped`記錄 Lambda 已停止的函數叫用次數，因為您的函數已在單一請求鏈中叫用超過 16 次。Lambda 會在停止遞迴調用時立即發出此指標。若要檢視此測量結果，請遵循在 CloudWatch主控台上檢視測量結果的指示，然後選擇測量結果`RecursiveInvocationsDropped`。

回應遞迴迴圈偵測通知

當函數被相同觸發事件調用超過 16 次時，Lambda 會停止該事件的下一個函數調用，以中斷遞迴迴圈。若要防止 Lambda 已中斷的遞迴迴圈再次發生，請執行下列動作：

- 將函數的可用並行處理降為零，這會限制所有將來的調用。
- 移除或停用正在調用函數的觸發條件或事件來源映射。
- 識別並修正將事件寫回呼叫函式之 AWS 資源的程式碼瑕疵。當您使用變數來定義函數的事件來源和目標時，就會出現常見的缺陷來源。檢查兩個變數沒有使用相同的值。

此外，如果 Lambda 函數的事件來源是 Amazon SQS 佇列，則請考慮在來源佇列上設定無效字母佇列。

> **Note**
>
> 請確定在來源佇列上設定無效字母佇列，而不是在 Lambda 函數上。您在函數上設定的無效字母佇列用於佇列的非同步調用函數，而不是事件來源佇列。

如果事件來源是 Amazon SNS 主題，則請考慮為函數新增故障時的目的地。

將函數的可用並行處理降為零 (主控台)

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇 函數的名稱。
3. 選擇調節。
4. 在調節函數對話方塊中，選擇確認。

若要移除函數的觸發條件或事件來源映射 (主控台)

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇 函數的名稱。
3. 選擇組態索引標籤，然後觸發條件。
4. 在觸發條件下，選取要刪除的觸發條件或事件來源映射，然後選擇刪除。
5. 在刪除觸發條件對話方塊中，選擇刪除。

若要停用函數的事件來源映射 (AWS CLI)

1. 若要尋找您要停用之事件來源對應的 UUID，請執行 AWS Command Line Interface (AWS CLI) 清單-事件來源對應指令。

   ```
   aws lambda list-event-source-mappings
   ```

2. 若要停用事件來源對應，請執行下列 AWS CLI 更新-事件來源對應命令。

   ```
   aws lambda update-event-source-mapping --function-name MyFunction \
   --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 --no-enabled
   ```

Lambda 函數 URL

函數 URL 是 Lambda 函數專用的 HTTP(S) 端點。您可以透過 Lambda 主控台或 Lambda API 建立及設定函數 URL。當您建立函數 URL 時，Lambda 會自動為您產生不重複的 URL 端點。函數 URL 一旦建立，其 URL 端點便永遠不會變更。函數 URL 端點的格式如下：

```
https://<url-id>.lambda-url.<region>.on.aws
```

> **Note**
>
> 下列地區不支援函數 URL：亞太區域 (海德拉巴) (ap-south-2)、亞太區域 (墨爾本) (ap-southeast-4)、加拿大西部 (卡加利ca-west-1) ()、歐洲 (西班牙eu-south-2) ()、歐洲 (蘇黎世eu-central-2) ()、以色列 (特拉維夫il-central-1) () 和中東 (阿聯酋) (me-central-1)。

函數 URL 可支援雙堆疊，能同時支援 IPv4 和 IPv6。為函數設定函數 URL 後，您可以利用 Web 瀏覽器、curl、Postman 或任何 HTTP 用戶端，透過 HTTP(S) 端點呼叫函數。

> **Note**
>
> 您只能透過公有網際網路存取您的函數 URL。雖然 Lambda 函數確實支援 AWS PrivateLink，但函數 URL 不支援。

Lambda 函數 URL 使用以資源為基礎的政策，妥善控管安全性和存取權。函數 URL 也支援跨來源資源共用 (CORS) 組態選項。

您可以將函數 URL 套用至任何函數別名或未發佈的$LATEST函數版本。函數 URL 無法新增至其他任何函數版本。

主題

- 建立及管理 Lambda 函數 URL
- 控制對 Lambda 函數網址的存取
- 呼叫 Lambda 函數 URL
- 監控 Lambda 函數 URL
- 教學課程：建立具有函數 URL 的 Lambda 函數

建立及管理 Lambda 函數 URL

函數 URL 是 Lambda 函數專用的 HTTP(S) 端點。您可以透過 Lambda 主控台或 Lambda API 建立及設定函數 URL。當您建立函數 URL 時，Lambda 會自動為您產生不重複的 URL 端點。函數 URL 一旦建立，其 URL 端點便永遠不會變更。函數 URL 端點的格式如下：

```
https://<url-id>.lambda-url.<region>.on.aws
```

> **Note**
>
> 下列地區不支援函數 URL：亞太區域 (海德拉巴) (ap-south-2)、亞太區域 (墨爾本) (ap-southeast-4)、加拿大西部 (卡加利ca-west-1) ()、歐洲 (西班牙eu-south-2) ()、歐洲 (蘇黎世eu-central-2) ()、以色列 (特拉維夫il-central-1) () 和中東 (阿聯酋) (me-central-1)。

以下部分說明如何使用 Lambda 主控台和 AWS CloudFormation 範本建立和管理函數 URL AWS CLI

主題

建立函數 URL (主控台)

請遵循下列步驟，使用主控台建立函數 URL。

為現有函數建立函數 URL (主控台)

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇您要為其建立函數 URL 的函數名稱。
3. 選擇 Configuration (組態) 標籤，然後選擇 Function URL (函數 URL)。

4. 選擇 Create function URL (建立函數 URL)。
5. 為 Auth type (驗證類型) 選擇 AWS_IAM 或 NONE (無)。如需函數 URL 身分驗證的詳細資訊，請參閱存取控制。
6. (選用) 選取 Configure cross-origin resource sharing (CORS) (設定跨來源資源共享 (CORS))，然後完成函數 URL 的 CORS 設定。如需 CORS 的詳細資訊，請參閱「跨來源資源共享 (CORS)」。
7. 選擇儲存。

這樣即可為 $LATEST 未發佈版本的函數建立函數 URL。函數 URL 會顯示於主控台的 Function overview (函數概觀) 區段。

為現有別名建立函數 URL (主控台)

1. 開啟 Lambda 主控台中的函數頁面。
2. 找到您要為其建立函數 URL 的函數別名，選擇其函數名稱。
3. 選擇 Aliases (別名) 標籤，接著找到您要為其建立函數 URL 的函數別名，選擇該別名的名稱。
4. 選擇 Configuration (組態) 標籤，然後選擇 Function URL (函數 URL)。
5. 選擇 Create function URL (建立函數 URL)。
6. 為 Auth type (驗證類型) 選擇 AWS_IAM 或 NONE (無)。如需函數 URL 身分驗證的詳細資訊，請參閱存取控制。
7. (選用) 選取 Configure cross-origin resource sharing (CORS) (設定跨來源資源共享 (CORS))，然後完成函數 URL 的 CORS 設定。如需 CORS 的詳細資訊，請參閱「跨來源資源共享 (CORS)」。
8. 選擇儲存。

這樣即可為函數別名建立函數 URL。函數 URL 會顯示於主控台中別名的 Function overview (函數概觀) 區段。

使用函數 URL 建立新函數 (主控台)

建立具有函數 URL 的新函數 (主控台)

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇 建立函數。
3. 在基本資訊下，請執行下列動作：

a. 為 Function name (函數名稱) 輸入您函數的名稱，例如 **my-function**。

b. 為 Runtime (執行階段) 選擇您偏好的語言執行階段，例如 Node.js 18.x。

c. 為 Architecture (架構) 選擇 x86_64 或 arm64。

d. 展開 Permissions (許可)，接著選擇建立新的執行角色或使用現有角色。

4. 展開 Advanced settings (進階設定)，然後選取 Function URL (函數 URL)。

5. 為 Auth type (驗證類型) 選擇 AWS_IAM 或 NONE (無)。如需函數 URL 身分驗證的詳細資訊，請參閱存取控制。

6. (選用) 選取 Configure cross-origin resource sharing (CORS) (設定跨來源資源共享 (CORS))。只要您在建立函數的過程中選取此選項，在預設情形下，您的函數 URL 就能允許所有來源的請求。建立函數後，您可以編輯函數 URL 的 CORS 設定。如需 CORS 的詳細資訊，請參閱「跨來源資源共享 (CORS)」。

7. 選擇建立函數。

這樣即可為 $LATEST 未發佈版本的函數建立具有函數 URL 的新函數。函數 URL 會顯示於主控台的 Function overview (函數概觀) 區段。

建立函數 URL (AWS CLI)

若要使用 AWS Command Line Interface (AWS CLI) 為現有 Lambda 函數建立函數 URL，請執行下列命令：

```
aws lambda create-function-url-config \
    --function-name my-function \
    --qualifier prod \ // optional
    --auth-type AWS_IAM
    --cors-config {AllowOrigins="https://example.com"} // optional
```

這會將函數 URL 新增至函數 **my-function** 的 **prod** 限定詞。如需有關這些組態參數的詳細資訊，請參閱 API 參考資料CreateFunctionUrlConfig中的。

> **Note**
>
> 若要透過建立函數 URL AWS CLI，函數必須已經存在。

將函數 URL 新增至 CloudFormation 範本

若要將資`AWS::Lambda::Url`源新增至 AWS CloudFormation 範本，請使用下列語法：

JSON

```
{
  "Type" : "AWS::Lambda::Url",
  "Properties" : {
      "AuthType" : String,
      "Cors" : Cors,
      "Qualifier" : String,
      "TargetFunctionArn" : String
    }
}
```

YAML

```
Type: AWS::Lambda::Url
Properties:
  AuthType: String
  Cors:
    Cors
  Qualifier: String
  TargetFunctionArn: String
```

參數

- (必要) `AuthType` – 定義函數 URL 的身分驗證類型。可能的值為 `AWS_IAM` 或 `NONE`。如果您希望只讓完成身分驗證的使用者存取，請設為 `AWS_IAM`。如要繞過 IAM 身分驗證，並允許任何使用者向您的函數提出請求，請設為 `NONE`。
- (選用) `Cors` – 定義函數 URL 的 CORS 設定。若要新增`Cors`至中的`AWS::Lambda::Url`資源 CloudFormation，請使用下列語法。

  Example AWS::Lambda::Url.Cors (JSON)

  ```
  {
    "AllowCredentials" : Boolean,
    "AllowHeaders" : [ String, ... ],
    "AllowMethods" : [ String, ... ],
  ```

```
  "AllowOrigins" : [ String, ... ],
  "ExposeHeaders" : [ String, ... ],
  "MaxAge" : Integer
}
```

Example AWS::Lambda::Url.Cors (YAML)

```
  AllowCredentials: Boolean
  AllowHeaders:
    - String
  AllowMethods:
    - String
  AllowOrigins:
    - String
  ExposeHeaders:
    - String
  MaxAge: Integer
```

- (選用) `Qualifier` – 別名名稱。
- (必要) `TargetFunctionArn` - Lambda 函數的名稱或 Amazon Resource Name (ARN)。有效名稱的格式包括：
  - 函數名稱 – `my-function`
  - 函數 ARN – `arn:aws:lambda:us-west-2:123456789012:function:my-function`
  - 部分 ARN – `123456789012:function:my-function`

跨來源資源共享 (CORS)

如要定義不同來源存取您函數 URL 的方式，請使用跨來源資源共享 (CORS)。如果您打算從其他網域呼叫函數 URL，建議您設定 CORS。Lambda 為函數 URL 支援以下 CORS 標頭。

CORS 標頭	CORS 組態屬性	範例值

Access-Control-Allow-Origin	`AllowOrigins`	* (允許所有來源)
		`https://www.example.com`
		`http://localhost:60905`

CORS 標頭	CORS 組態屬性	範例值
Access-Control-Allow-Methods	AllowMethods	GET, POST, DELETE, *
Access-Control-Allow-Headers	AllowHeaders	Date, Keep-Alive , X-Custom-Header
Access-Control-Expose-Headers	ExposeHeaders	Date, Keep-Alive , X-Custom-Header
Access-Control-Allow-Credentials	AllowCredentials	TRUE
Access-Control-Max-Age	MaxAge	5 (預設)、300

當您使用 Lambda 主控台或設定函數 URL 的 CORS 時 AWS CLI，Lambda 會透過函數 URL 自動將 CORS 標頭新增至所有回應。或者，您也可以手動將 CORS 標頭新增至函數回應中。如果標頭之間有所衝突，系統會優先使用您在函數 URL 上設定的 CORS 標頭。

調節函數 URL

調節作業會限制函數處理請求的速度。這在許多情況下都相當實用，例如防止函數多載下游資源，或處理突然激增的請求數。

您可以設定預留並行，調節 Lambda 函數透過函數 URL 處理請求的速度。預留並行可限制函數的並行呼叫次數上限。函數的每秒請求率 (RPS) 上限相當於所設定預留並行數的 10 倍。例如，如果您將函數的預留並行值設為 100，則 RPS 最高為 1,000。

每當函數並行數量超過預留的並行數，函數 URL 就會傳回 HTTP 429 狀態碼。如果函數收到的請求超過所設定預留並行數 10 倍的 RPS 最大值，您也會收到 HTTP 429 錯誤。如需預留並行的詳細資訊，請參閱「為函數配置保留並發」。

停用函數 URL

在緊急情況下，您可能會希望拒絕傳入函數 URL 的所有流量。若要停用函數 URL，請將預留並行設為零。這樣就能調節函數 URL 收到的所有請求，進而使 URL 傳回 HTTP 429 狀態回應。如要重新啟動函數 URL，請刪除預留並行的組態，或將組態設為大於零的數量。

刪除函數 URL

當您刪除函數 URL，您就無法復原。建立新函數 URL 會產生不同的 URL 地址。

> **Note**
>
> 如果您刪除具有驗證類型 NONE 的函數 URL，Lambda 不會自動刪除關聯的資源型政策。如果您想要刪除此政策，則必須手動執行。

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇函數的名稱。
3. 選擇 Configuration (組態) 標籤，然後選擇 Function URL (函數 URL)。
4. 選擇刪除。
5. 將 delete 一詞輸入欄位以確認刪除。
6. 選擇刪除。

> **Note**
>
> 當您刪除具有函數 URL 的函數時，Lambda 會以非同步方式刪除函數 URL。如果您立即在同一帳戶中創建具有相同名稱的新函數，則原始函數 URL 可能會映射到新函數而不是刪除。

控制對 Lambda 函數網址的存取

您可使用 AuthType 參數和連接至特定函數的資源型政策，控制對 Lambda 函數 URL 的存取權。這兩個元件的組態能決定誰可以對函數 URL 呼叫或執行其他管理動作。

AuthType 參數決定 Lambda 如何對函數 URL 的請求執行身分驗證或授權。設定函數 URL 時，您必須指定以下任一 AuthType 選項：

- AWS_IAM— Lambda 使用 AWS Identity and Access Management (IAM) 根據 IAM 主體的身分識別政策和函數的資源型政策來驗證和授權請求。如果您希望只讓完成身分驗證的 使用者和角色透過函數 URL 呼叫您的函數，請選擇此選項。

- NONE – Lambda 不會在呼叫函數前執行任何身分驗證。然而，函數的資源型政策永遠有效，而且您必須授予公有存取權，您的函數 URL 才能接收請求。選擇此選項，即可允許使用者公開存取函數 URL，而且不必完成身分驗證。

除了 AuthType 之外，您也可以使用資源型政策授予其他 AWS 帳戶 呼叫函數的許可。如需詳細資訊，請參閱 使用 Lambda 中以資源為基礎的政策。

有關安全性的其他見解，您可 AWS Identity and Access Management Access Analyzer 以使用對函數 URL 的外部訪問進行全面分析。IAM Access Analyzer 也能監控您 Lambda 函數新增或更新的許可，以協助您識別授予公有和跨帳戶存取權的許可。IAM 存取分析器可供任何 AWS 客戶免費使用。若要開始使用 IAM 存取分析器，請參閱使用 AWS IAM 存取分析器。

此頁面包含兩種驗證類型的資源型政策範例，以及如何使用 AddPermissionAPI 作業或 Lambda 主控台建立這些政策。如需了解在設定許可後呼叫函數 URL 的相關資訊，請參閱呼叫 Lambda 函數 URL。

主題

- 使用 AWS_IAM 驗證類型
- 使用 NONE 驗證類型
- 控管和存取權控制

使用 **AWS_IAM** 驗證類型

如果您選擇採用 AWS_IAM 驗證類型，使用者必須擁有 lambda:InvokeFunctionUrl 許可才能呼叫您的 Lambda 函數 URL。視提出呼叫請求的使用者而定，您可能必須使用資源型政策授予此許可。

如果提出要求的主體與函數 URL 位於 AWS 帳戶 相同的函數 URL 中，則主體必須在其身分識別型原則中具有**lambda:InvokeFunctionUrl**權限，或是在函數的資源型原則中具有授與權限。換句話說，如果使用者已經由身分型政策擁有 lambda:InvokeFunctionUrl 許可，即可自行決定是否使用資源型政策。政策評估作業需遵循決定是否允許或拒絕帳戶中的請求一文所述的規則。

如果提出請求的委託人位於不同帳戶，則委託人必須同時經由身分型政策取得 lambda:InvokeFunctionUrl 許可，並且針對其嘗試呼叫的函數，透過資源型政策取得許可。在這些跨帳戶的情況中，政策評估作業需遵循決定是否允許跨帳戶請求一文所述的規則。

對於跨帳戶互動的範例，下列以資源為基礎的策略允許中的example角色叫 AWS 帳戶 444455556666用與函數相關聯的函數 URL：my-function

Example 函數 URL 跨帳戶呼叫政策

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::444455556666:role/example"
            },
            "Action": "lambda:InvokeFunctionUrl",
            "Resource": "arn:aws:lambda:us-east-1:123456789012:function:my-function",
            "Condition": {
                "StringEquals": {
                    "lambda:FunctionUrlAuthType": "AWS_IAM"
                }
            }
        }
    ]
}
```

您可以依照以下步驟，透過主控台建立此政策陳述式：

將 URL 呼叫許可授予其他帳戶 (主控台)

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇您要授予 URL 呼叫許可的函數名稱。
3. 依序選擇 Configuration (組態) 索引標籤和 Permissions (許可)。
4. 在 Resource-based policy (資源型政策) 底下，選擇 Add permissions (新增許可)。

5. 選擇 Function URL (函數 URL)。
6. 針對 Auth type (驗證類型) 選擇 AW_IAM。
7. (選用) 針對 Statement ID (陳述式 ID) 輸入政策陳述式的陳述式 ID。
8. 依據您要授予許可的使用者或角色，在主體輸入其 Amazon Resource Name (ARN)。例如：**444455556666**。
9. 選擇儲存。

或者，您也可以使用下列 add 權限 AWS Command Line Interface (AWS CLI) 命令來建立這個原則陳述式：

```
aws lambda add-permission --function-name my-function \
    --statement-id example0-cross-account-statement \
    --action lambda:InvokeFunctionUrl \
    --principal 444455556666 \
    --function-url-auth-type AWS_IAM
```

在先前的範例中，`lambda:FunctionUrlAuthType` 條件索引鍵值為 `AWS_IAM`。唯有當函數 URL 的驗證類型也是 `AWS_IAM` 時，此政策才會允許您存取函數 URL。

使用 **NONE** 驗證類型

> **Important**
>
> 當您的函數 URL 驗證類型為 `NONE`，而且資源型政策授予公有存取權時，任何未經身分驗證的使用者只要取得您的函數 URL，都可以呼叫您的函數。

部分情況下，您可能需要將函數 URL 設為公有狀態。例如，您可能希望為直接透過 Web 瀏覽器提出的請求提供服務。如要允許使用者公開存取您的函數 URL，請選擇 `NONE` 驗證類型。

如果您選擇 `NONE` 驗證類型，Lambda 就不會使用 IAM 對存取函數 URL 的請求執行身分驗證。不過，使用者仍必須擁有 `lambda:InvokeFunctionUrl` 許可，才能成功呼叫您的函數 URL。您可以使用以下資源型政策來授予 `lambda:InvokeFunctionUrl` 許可：

Example 函數 URL 呼叫政策 (適用於所有未經身分驗證的委託人)

```
{
    "Version": "2012-10-17",
```

```
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": "*",
            "Action": "lambda:InvokeFunctionUrl",
            "Resource": "arn:aws:lambda:us-east-1:123456789012:function:my-function",
            "Condition": {
                "StringEquals": {
                    "lambda:FunctionUrlAuthType": "NONE"
                }
            }
        }
    ]
}
```

> **Note**
>
> 當您NONE透過主控台或 AWS Serverless Application Model (AWS SAM) 建立具有驗證類型的函數 URL 時，Lambda 會自動為您建立上述以資源為基礎的政策陳述式。如果政策早已建立，或建立應用程式的使用者或角色沒有適當的許可，Lambda 就不會為您建立該政策。如果您直接使用 AWS CLI AWS CloudFormation、或 Lambda API，則必須自行新增`lambda:InvokeFunctionUrl`權限。如此一來，您的函數就會設為公有狀態。
> 此外，如果您刪除具有驗證類型 NONE 的函數 URL，則 Lambda 不會自動刪除關聯的資源型政策。如果您想要刪除此政策，則必須手動執行。

此陳述式的 `lambda:FunctionUrlAuthType` 條件索引鍵值為 NONE。唯有當函數 URL 的驗證類型也是 NONE 時，此政策陳述式才會允許您存取函數 URL。

如果函數的資源型政策並未授予 `lambda:invokeFunctionUrl` 許可，當使用者嘗試呼叫您的函數 URL，畫面會顯示 403 禁止錯誤代碼，即使函數 URL 使用 NONE 驗證類型，依然會發生此錯誤。

控管和存取權控制

除了函數 URL 呼叫許可之外，您也可以控制函數 URL 設定動作的存取權。Lambda 支援以下適用於函數 URL 的 IAM 政策動作：

- `lambda:InvokeFunctionUrl` – 使用函數 URL 呼叫 Lambda 函數。
- `lambda:CreateFunctionUrlConfig` – 建立函數 URL 並設定其 AuthType。
- `lambda:UpdateFunctionUrlConfig` – 更新函數 URL 組態及其 AuthType。

- `lambda:GetFunctionUrlConfig` – 檢視函數 URL 的詳細資訊。
- `lambda:ListFunctionUrlConfigs` – 列出函數 URL 組態。
- `lambda:DeleteFunctionUrlConfig` – 刪除函數 URL。

> **Note**
>
> Lambda 主控台僅支援為 `lambda:InvokeFunctionUrl` 新增許可。如需執行其他所有動作，您必須使用 Lambda API 或 AWS CLI新增許可。

若要允許或拒絕其他 AWS 實體的功能 URL 存取，請在 IAM 政策中包含這些動作。例如，以下策略授予example角色 AWS 帳戶 444455556666權限，以更新帳戶**my-function**中函數的函數 URL 123456789012。

Example 跨帳戶函數 URL 政策

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::444455556666:role/example"
            },
            "Action": "lambda:UpdateFunctionUrlConfig",
            "Resource": "arn:aws:lambda:us-east-2:123456789012:function:my-function"
        }
    ]
}
```

條件索引鍵

如要精細控制函數 URL 的存取權，請使用條件索引鍵。Lambda 為函數 URL 額外支援一種條件索引鍵：FunctionUrlAuthType。FunctionUrlAuthType 索引鍵可定義描述函數 URL 所用驗證類型的列舉值。此值可以是 AWS_IAM 或 NONE。

您可以在與函數相關聯的政策中使用此條件索引鍵。例如，您可能希望限制哪些人可以變更函數 URL 的組態。若要針對 URL 驗證類型為 NONE 的所有函數拒絕所有 UpdateFunctionUrlConfig 請求，您可以定義以下政策：

Example 明確拒絕請求的函數 URL 政策

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Principal": "*",
            "Action":[
                "lambda:UpdateFunctionUrlConfig"
            ],
            "Resource": "arn:aws:lambda:us-east-1:123456789012:function:*",
            "Condition": {
                "StringEquals": {
                    "lambda:FunctionUrlAuthType": "NONE"
                }
            }
        }
    ]
}
```

要授予example角色對具有 URL 身份驗證類型的函數發出CreateFunctionUrlConfig和UpdateFunctionUrlConfig請求的 AWS 帳戶444455556666權限AWS_IAM，您可以定義以下策略：

Example 明確允許請求的函數 URL 政策

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::444455556666:role/example"
            },
            "Action":[
                "lambda:CreateFunctionUrlConfig",
                "lambda:UpdateFunctionUrlConfig"
            ],
            "Resource": "arn:aws:lambda:us-east-1:123456789012:function:*",
            "Condition": {
                "StringEquals": {
                    "lambda:FunctionUrlAuthType": "AWS_IAM"
```

```
                }
            }
        }
    ]
}
```

您也可以在 服務控制政策(SCP) 中使用此條件索引鍵。在 AWS Organizations中使用 SCP 管理整個組織的許可。例如，若要拒絕使用者建立或更新使用 AWS_IAM 以外驗證類型的函數 URL，請使用以下服務控制政策：

Example 明確拒絕請求的函數 URL SCP

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action":[
                "lambda:CreateFunctionUrlConfig",
                "lambda:UpdateFunctionUrlConfig"
            ],
            "Resource": "arn:aws:lambda:*:123456789012:function:*",
            "Condition": {
                "StringNotEquals": {
                    "lambda:FunctionUrlAuthType": "AWS_IAM"
                }
            }
        }
    ]
}
```

呼叫 Lambda 函數 URL

函數 URL 是 Lambda 函數專用的 HTTP(S) 端點。您可以透過 Lambda 主控台或 Lambda API 建立及設定函數 URL。當您建立函數 URL 時，Lambda 會自動為您產生不重複的 URL 端點。函數 URL 一旦建立，其 URL 端點便永遠不會變更。函數 URL 端點的格式如下：

```
https://<url-id>.lambda-url.<region>.on.aws
```

> **Note**
>
> 下列地區不支援函數 URL：亞太區域 (海德拉巴) (ap-south-2)、亞太區域 (墨爾本) (ap-southeast-4)、加拿大西部 (卡加利ca-west-1) ()、歐洲 (西班牙eu-south-2) ()、歐洲 (蘇黎世eu-central-2) ()、以色列 (特拉維夫il-central-1) () 和中東 (阿聯酋) (me-central-1)。

函數 URL 可支援雙堆疊，能同時支援 IPv4 和 IPv6。設定函數 URL 後，您可以利用 Web 瀏覽器、curl、Postman 或任何 HTTP 用戶端，透過 HTTP(S) 端點呼叫函數。如要呼叫函數 URL，您必須擁有 `lambda:InvokeFunctionUrl` 許可。如需詳細資訊，請參閱 存取控制。

主題

函數 URL 呼叫基礎知識

如果您的函數 URL 使用 `AWS_IAM` 驗證類型，您必須使用 AWS Signature 第 4 版 (SigV4) 簽署各個 HTTP 請求。awscurl、Postman 和 AWS SigV4 Proxy 等工具均提供使用 SigV4 簽署請求的內建功能。

如果您不使用工具簽署函數 URL 的 HTTP 請求，則必須使用 SigV4 手動簽署各個請求。您的函數 URL 收到請求時，Lambda 也會計算 SigV4 簽章。唯有簽章相符，Lambda 才會處理請求。如需使用 SigV4 手動簽署請求的相關說明，請參閱《Amazon Web Services 一般參考 指南》中的使用 Signature 第 4 版簽署 AWS 請求。

如果您的函數 URL 使用 `NONE` 驗證類型，您不必使用 SigV4 簽署請求。您可以使用 Web 瀏覽器、curl、Postman 或任何 HTTP 用戶端來呼叫您的函數。

如要測試函數的簡易 GET 請求，請使用 Web 瀏覽器。例如，如果函數 URL 為 `https://abcdefg.lambda-url.us-east-1.on.aws`，而且接受字串參數 `message`，則請求 URL 可能如下所示：

```
https://abcdefg.lambda-url.us-east-1.on.aws/?message=HelloWorld
```

如要測試其他 HTTP 請求 (例如 POST 請求)，您可以使用 curl 等工具。例如，如果您希望在函數 URL 的 POST 請求中納入某些 JSON 資料，您可以使用下列 curl 命令：

```
curl -v 'https://abcdefg.lambda-url.us-east-1.on.aws/?message=HelloWorld' \
-H 'content-type: application/json' \
-d '{ "example": "test" }'
```

請求和回應承載

用戶端呼叫您的函數 URL 時，Lambda 會先將請求映射至事件物件，再將請求傳遞給函數。接著，函數的回應會映射至 Lambda 透過函數 URL 回傳給用戶端的 HTTP 回應。

請求和回應事件格式會採取與 Amazon API Gateway 承載格式 2.0 版一樣的結構描述。

請求承載格式

請求承載的結構如下：

```
{
  "version": "2.0",
  "routeKey": "$default",
  "rawPath": "/my/path",
  "rawQueryString": "parameter1=value1&parameter1=value2&parameter2=value",
  "cookies": [
    "cookie1",
    "cookie2"
  ],
  "headers": {
    "header1": "value1",
    "header2": "value1,value2"
  },
  "queryStringParameters": {
    "parameter1": "value1,value2",
    "parameter2": "value"
  },
```

```
  "requestContext": {
    "accountId": "123456789012",
    "apiId": "<urlid>",
    "authentication": null,
    "authorizer": {
        "iam": {
                "accessKey": "AKIA...",
                "accountId": "111122223333",
                "callerId": "AIDA...",
                "cognitoIdentity": null,
                "principalOrgId": null,
                "userArn": "arn:aws:iam::111122223333:user/example-user",
                "userId": "AIDA..."
        }
    },
    "domainName": "<url-id>.lambda-url.us-west-2.on.aws",
    "domainPrefix": "<url-id>",
    "http": {
      "method": "POST",
      "path": "/my/path",
      "protocol": "HTTP/1.1",
      "sourceIp": "123.123.123.123",
      "userAgent": "agent"
    },
    "requestId": "id",
    "routeKey": "$default",
    "stage": "$default",
    "time": "12/Mar/2020:19:03:58 +0000",
    "timeEpoch": 1583348638390
  },
  "body": "Hello from client!",
  "pathParameters": null,
  "isBase64Encoded": false,
  "stageVariables": null
}
```

參數	描述	範例
version	此事件的承載格式版本。Lambda 函數 URL 目前支援承載格式 2.0 版。	2.0

參數	描述	範例
routeKey	函數 URL 不使用此參數。Lambda 將此設為 $default 作為預留位置使用。	$default
rawPath	請求路徑。例如，如果請求 URL 為 https://{url-id}.lambda-url.{region}.on.aws/example/test/demo ，則原始路徑值為 /example/test/demo 。	/example/test/demo
rawQueryString	內含請求查詢字串參數的原始字串。支援的字元包含 a-z、A-Z、0-9、.、_、-、%、&、=，以及 +。	"?parameter1=value1&parameter2=value2"
cookies	內含隨請求一併傳送之所有 Cookie 的陣列。	["Cookie_1=Value_1", "Cookie_2=Value_2"]
headers	以鍵值對形式呈現的請求標頭清單。	{"header1": "value1", "header2": "value2"}
queryStringParameters	請求的查詢參數。例如，如果請求 URL 為 https://{url-id}.lambda-url.{region}.on.aws/example?name=Jane ，則 queryStringParameters 值為具備索引鍵 name 和值 Jane 的 JSON 物件。	{"name": "Jane"}

參數	描述	範例
requestContext	內含請求其他資訊的物件，例如 requestId 、請求的時間，以及呼叫者的身分 (如果透過 AWS Identity and Access Management (IAM) 取得授權的話)。	
requestContext.accountId	函數擁有者的 AWS 帳戶 ID。	"123456789012"
requestContext.apiId	函數 URL 的 ID。	"33anwqw8fj"
requestContext.authentication	函數 URL 不使用此參數。Lambda 將此設為 null。	null
requestContext.authorizer	內含呼叫者身分相關資訊的物件 (如果函數 URL 使用 AWS_IAM 驗證類型的話)，否則 Lambda 會將此設為 null。	
requestContext.authorizer.iam.accessKey	呼叫者身分的存取金鑰。	"AKIAIOSFODNN7EXAMPLE"
requestContext.authorizer.iam.accountId	呼叫者身分的 AWS 帳戶 ID。	"111122223333"
requestContext.authorizer.iam.callerId	呼叫者的 ID (使用者 ID)。	"AIDACKCEVSQ6C2EXAMPLE"
requestContext.authorizer.iam.cognitoIdentity	函數 URL 不使用此參數。Lambda 將此設為 null，或將此從 JSON 排除。	null

參數	描述	範例
requestContext.authorizer.iam.principalOrgId	與呼叫者身分相關聯的委託人組織 ID。	"AIDACKCEVSQORGEXAMPLE"
requestContext.authorizer.iam.userArn	呼叫者身分的使用者 Amazon Resource Name (ARN)。	"arn:aws:iam::111122223333:user/example-user"
requestContext.authorizer.iam.userId	呼叫者身分的使用者 ID。	"AIDACOSFODNN7EXAMPLE2"
requestContext.domainName	函數 URL 的網域名稱。	"<url-id>.lambda-url.us-west-2.on.aws"
requestContext.domainPrefix	函數 URL 的網域前綴。	"<url-id>"
requestContext.http	內含 HTTP 請求詳細資訊的物件。	
requestContext.http.method	此請求所採用的 HTTP 方法。有效值包括 GET、POST、PUT、HEAD、OPT 和 DELETE。	GET
requestContext.http.path	請求路徑。例如，如果請求 URL 為 https://{url-id}.lambda-url.{region}.on.aws/example/test/demo ，則路徑值為 /example/test/demo 。	/example/test/demo
requestContext.http.protocol	請求的通訊協定。	HTTP/1.1

參數	描述	範例
requestContext.http.sourceIp	提出請求之即時 TCP 連線的來源 IP 地址。	123.123.123.123
requestContext.http.userAgent	使用者代理程式請求標頭的值。	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) Gecko/20100101 Firefox/42.0
requestContext.requestId	呼叫請求的 ID。您可以使用此 ID 追蹤與函數相關的呼叫日誌。	e1506fd5-9e7b-434f-bd42-4f8fa224b599
requestContext.routeKey	函數 URL 不使用此參數。Lambda 將此設為 $default 作為預留位置使用。	$default
requestContext.stage	函數 URL 不使用此參數。Lambda 將此設為 $default 作為預留位置使用。	$default
requestContext.time	請求的時間戳記。	"07/Sep/2021:22:50:22 +0000"
requestContext.timeEpoch	Unix epoch 時間格式的請求時間戳記。	"1631055022677"
body	請求的本文。如果請求的內容屬於二進位類型，則本文會採用 base64 編碼。	{"key1": "value1", "key2": "value2"}
pathParameters	函數 URL 不使用此參數。Lambda 將此設為 null，或將此從 JSON 排除。	null
isBase64Encoded	如果本文為二進位承載並採用 base64 編碼，此值為 TRUE，否則為 FALSE。	FALSE

參數	描述	範例
stageVariables	函數 URL 不使用此參數。Lambda 將此設為 null，或將此從 JSON 排除。	null

回應承載格式

函數傳回回應時，Lambda 會剖析回應內容，並將其轉換為 HTTP 回應。函數回應承載的格式如下：

```
{
   "statusCode": 201,
    "headers": {
        "Content-Type": "application/json",
        "My-Custom-Header": "Custom Value"
    },
    "body": "{ \"message\": \"Hello, world!\" }",
    "cookies": [
        "Cookie_1=Value1; Expires=21 Oct 2021 07:48 GMT",
        "Cookie_2=Value2; Max-Age=78000"
    ],
    "isBase64Encoded": false
}
```

Lambda 會為您推斷回應格式。如果您的函數傳回有效的 JSON，但未傳回 statusCode，則 Lambda 會採取下列假設：

- statusCode 是 200。
- content-type 是 application/json。
- body 是函數的回應。
- isBase64Encoded 是 false。

以下範例顯示 Lambda 函數輸出與回應承載之間的映射情形，以及回應承載與最終 HTTP 回應之間的映射情形。用戶端呼叫您的函數 URL 時，會看見 HTTP 回應。

字串回應的輸出範例

Lambda 函數輸出	轉譯後的回應輸出	HTTP 回應 (用戶端看見的內容)

`"Hello, world!"`	`{ "statusCode": 200, "body": "Hello, world!", "headers": { "content-type": "application/json" }, "isBase64Encoded": false }`	`HTTP/2 200`
		`date: Wed, 08 Sep 2021 18:02:24 GMT`
		`content-type: application/json`
		`content-length: 15`
		
		`"Hello, world!"`

JSON 回應的輸出範例

Lambda 函數輸出	轉譯後的回應輸出	HTTP 回應 (用戶端看見的內容)

`{ "message": "Hello, world!" }`	`{ "statusCode": 200, "body": { "message": "Hello, world!" }, "headers": { "content-type": "application/json" }, "isBase64Encoded": false }`	`HTTP/2 200`
		`date: Wed, 08 Sep 2021 18:02:24 GMT`
		`content-type: application/json`
		`content-length: 34`
		
		`{ "message": "Hello, world!" }`

自訂回應的輸出範例

Lambda 函數輸出	轉譯後的回應輸出	HTTP 回應 (用戶端看見的內容)

<pre>{
  "statusCode": 201,
  "headers": {
  "Content-Type": "application/json",
  "My-Custom-Header": "Custom Value"
  },
  "body": JSON.stringify({
  "message": "Hello, world!"
  }),
  "isBase64Encoded": false
}</pre>
	<pre>{
	  "statusCode": 201,
	  "headers": {
	  "Content-Type": "application/json",
	  "My-Custom-Header": "Custom Value"
	  },
	  "body": JSON.stringify({
	  "message": "Hello, world!"
	  }),
	  "isBase64Encoded": false
	}</pre>
		<pre>HTTP/2 201
		date: Wed, 08 Sep 2021 18:02:24 GMT
		content-type: application/json
		content-length: 27
		my-custom-header: Custom Value
		
		{
		  "message": "Hello, world!"
		}</pre>

Cookie

如要從函數傳回 Cookie，請勿手動新增 `set-cookie` 標頭。您應將 Cookie 加入回應承載物件中。Lambda 會自動轉譯 Cookie，並以 `set-cookie` 標頭形式新增至 HTTP 回應，如下所示。

回應傳回 Cookie 的輸出範例

Lambda 函數輸出	HTTP 回應 (用戶端看見的內容)

<pre>{
  "statusCode": 201,
  "headers": {
  "Content-Type": "application/json",
  "My-Custom-Header": "Custom Value"
  },
  "body": JSON.stringify({
  "message": "Hello, world!"</pre>
	<pre>HTTP/2 201
	date: Wed, 08 Sep 2021 18:02:24 GMT
	content-type: application/json
	content-length: 27
	my-custom-header: Custom Value
	set-cookie: Cookie_1=Value2; Expires=21 Oct 2021 07:48 GMT
	set-cookie: Cookie_2=Value2; Max-Age=78000</pre>

Lambda 函數輸出	HTTP 回應 (用戶端看見的內容)

<pre>   }),
    "cookies": [
        "Cookie_1=Value1; Expires=21
 Oct 2021 07:48 GMT",
        "Cookie_2=Value2; Max-Age=7
8000"
    ],
    "isBase64Encoded": false
}</pre>
	<pre>{
	  "message": "Hello, world!"
	}</pre>

監控 Lambda 函數 URL

您可以使用 AWS CloudTrail 和 Amazon CloudWatch 來監控您的功能 URL。

主題

使用監控功能的 URL CloudTrail

對於函數 URL，Lambda 會自動支援將下列 API 作業記錄為記 CloudTrail 錄檔中的事件：

- CreateFunctionUrlConfig
- UpdateFunctionUrlConfig
- DeleteFunctionUrlConfig
- GetFunctionUrlConfig
- ListFunctionUrlConfigs

每個日誌項目都包含呼叫者身分、提出請求的時間，以及其他詳細資訊等相關資訊。您可以檢視您的活動紀錄，查看過去 90 天內的所有 CloudTrail 活動。如要保留 90 天前的記錄，您可以建立線索。

根據預設， CloudTrail 不會記錄被視為資料事件的`InvokeFunctionUrl`要求。不過，您可以開啟資料事件登入 CloudTrail。如需詳細資訊，請參閱《AWS CloudTrail 使用者指南》中的記錄資料事件。

CloudWatch 函數 URL 的度量

Lambda 會將有關函數 URL 請求的彙總指標傳送至 CloudWatch。使用這些指標，您可以在 CloudWatch 控制台中監視功能 URL，構建儀表板並配置警報。

函數 URL 支援以下呼叫指標。建議您搭配 Sum 統計數字一併檢視這些指標。

- `UrlRequestCount` – 對此函數提出的請求數量。
- `Url4xxCount` – 傳回 4XX HTTP 狀態碼的請求數量。收到 4XX 系列代碼表示用戶端發生錯誤，例如請求錯誤。
- `Url5xxCount` – 傳回 5XX HTTP 狀態碼的請求數量。收到 5XX 系列代碼表示伺服器端發生錯誤，例如函數錯誤和逾時。

函數 URL 也支援以下效能指標。建議您搭配 Average 或 Max 統計數字一併檢視這些指標。

- UrlRequestLatency – 函數 URL 從收到請求到傳回回應所經過的時間。

這些呼叫和效能指標均支援以下維度：

- FunctionName – 針對指派給函數$LATEST未發佈版本或任何函數別名的函數 URL，查看其彙總指標，例如 hello-world-function。
- Resource – 檢視特定函數 URL 的指標。您可使用函數名稱，並搭配函數未發佈的$LATEST版本或任一函數別名來加以定義，例如 hello-world-function:$LATEST。
- ExecutedVersion – 根據所執行的版本檢視特定函數 URL 的指標。此維度的主要功能是追蹤指派給$LATEST未發佈版本的函數 URL。

教學課程：建立具有函數 URL 的 Lambda 函數

在本教學課程中，您會建立格式為 .zip 封存檔的 Lambda 函數，其包含的公有函數 URL 端點會傳回兩個數字的乘積。如需設定函數 URL 的詳細資訊，請參閱 建立及管理函數 URL。

必要條件

此教學課程假設您具備基本的 Lambda 操作知識並了解 Lambda 主控台。若您尚未了解，請遵循 使用主控台建立一個 Lambda 函數 中的指示，建立您的第一個 Lambda 函數。

若要完成下列步驟，您需要 AWS Command Line Interface (AWS CLI) 版本 2。命令和預期的輸出會列在不同的區塊中：

```
aws --version
```

您應該會看到下列輸出：

```
aws-cli/2.13.27 Python/3.11.6 Linux/4.14.328-248.540.amzn2.x86_64 exe/x86_64.amzn.2
```

對於長命令，逸出字元 (\) 用於將命令分割為多行。

在 Linux 和 macOS 上，使用您偏好的 shell 和套件軟體管理工具。

> **Note**
>
> 在 Windows 中，作業系統的內建終端不支援您常與 Lambda 搭配使用的某些 Bash CLI 命令 (例如 `zip`)。若要取得 Ubuntu 和 Bash 的 Windows 整合版本，請安裝適用於 Linux 的 Windows 子系統。本指南中的 CLI 命令範例使用 Linux 格式。如果您使用的是 Windows CLI，必須重新格式化包含內嵌 JSON 文件的命令。

建立執行角色

建立執行角色，授予您的 Lambda 函數存取 AWS 資源的許可。

若要建立執行角色

1. 開啟 AWS Identity and Access Management (IAM) 主控台的「角色」頁面。
2. 選擇建立角色。
3. 對於受信任的實體類型，請選取AWS 服務，然後針對使用案例選取 Lambda。

4. 選擇下一步。
5. 在 [權限原則] 窗格**AWSLambdaBasicExecutionRole**中，於搜尋方塊中輸入。
6. 選取AWSLambdaBasicExecutionRole AWS 受管理策略旁邊的核取方塊，然後選擇下一步。
7. 輸入 **lambda-url-role**「角色」名稱，然後選擇「建立角色」。

該AWSLambdaBasicExecutionRole政策具有函數將日誌寫入 Amazon CloudWatch 日誌所需的許可。稍後在教學課程中，您需要角色的 Amazon 資源名稱 (ARN) 來建立您的 Lambda 函數。

若要尋找執行角色的 ARN

1. 開啟 AWS Identity and Access Management (IAM) 主控台的「角色」頁面。
2. 選取您剛建立的角色 (lambda-url-role)。
3. 在 [摘要] 窗格中，複製 ARN。

建立具有函數 URL 的 Lambda 函數 (.zip 封存檔)

使用 .zip 封存檔建立具有函數 URL 端點的 Lambda 函數。

建立函數

1. 將下列程式碼範例複製至名為 index.js 的檔案中。

   Example index.js

   ```
   exports.handler = async (event) => {
       let body = JSON.parse(event.body);
       const product = body.num1 * body.num2;
       const response = {
           statusCode: 200,
           body: "The product of " + body.num1 + " and " + body.num2 + " is " +
    product,
       };
       return response;
   };
   ```

2. 建立部署套件。

   ```
   zip function.zip index.js
   ```

3. 使用 `create-function` 命令建立一個 Lambda 函數。請務必使用您先前在教學課程中複製的執行角色的 ARN 來取代角色 ARN。

```
aws lambda create-function \
    --function-name my-url-function \
    --runtime nodejs18.x \
    --zip-file fileb://function.zip \
    --handler index.handler \
    --role arn:aws:iam::123456789012:role/lambda-url-role
```

4. 將資源型政策新增至授予許可的函數，以允許公開存取您的函數 URL。

```
aws lambda add-permission \
    --function-name my-url-function \
    --action lambda:InvokeFunctionUrl \
    --principal "*" \
    --function-url-auth-type "NONE" \
    --statement-id url
```

5. 使用 `create-function-url-config` 命令為函數建立 URL 端點。

```
aws lambda create-function-url-config \
    --function-name my-url-function \
    --auth-type NONE
```

測試函數 URL 端點

使用 HTTP 用戶端 (例如 curl 或 Postman) 呼叫函數 URL 端點，藉以呼叫 Lambda 函數。

```
curl 'https://abcdefg.lambda-url.us-east-1.on.aws/' \
-H 'Content-Type: application/json' \
-d '{"num1": "10", "num2": "10"}'
```

您應該會看到下列輸出：

```
The product of 10 and 10 is 100
```

使用函數 URL（CloudFormation）創建一個 Lambda 函數

您也可以使用該類型建立具有函數 URL 端點的 Lambda 函 AWS CloudFormation 數`AWS::Lambda::Url`。

```
Resources:
  MyUrlFunction:
    Type: AWS::Lambda::Function
    Properties:
      Handler: index.handler
      Runtime: nodejs18.x
      Role: arn:aws:iam::123456789012:role/lambda-url-role
      Code:
        ZipFile: |
          exports.handler = async (event) => {
              let body = JSON.parse(event.body);
              const product = body.num1 * body.num2;
              const response = {
                  statusCode: 200,
                  body: "The product of " + body.num1 + " and " + body.num2 + " is " +
 product,
              };
              return response;
          };
      Description: Create a function with a URL.
  MyUrlFunctionPermissions:
    Type: AWS::Lambda::Permission
    Properties:
      FunctionName: !Ref MyUrlFunction
      Action: lambda:InvokeFunctionUrl
      Principal: "*"
      FunctionUrlAuthType: NONE
  MyFunctionUrl:
    Type: AWS::Lambda::Url
    Properties:
      TargetFunctionArn: !Ref MyUrlFunction
      AuthType: NONE
```

建立具有函數 URL 的 Lambda 函數 (AWS SAM)

您也可以使用 AWS Serverless Application Model () 建立使用函數 URL 設定的 Lambda 函AWS SAM 數。

```
ProductFunction:
  Type: AWS::Serverless::Function
  Properties:
    CodeUri: function/.
    Handler: index.handler
    Runtime: nodejs18.x
    AutoPublishAlias: live
    FunctionUrlConfig:
      AuthType: NONE
```

清除您的資源

除非您想要保留為此教學課程建立的資源，否則您現在便可刪除。刪除您不再使用的 AWS 資源，您可以避免不必要的 AWS 帳戶費用

刪除執行角色

1. 開啟 IAM 主控台中的 角色頁面 。
2. 選取您建立的執行角色。
3. 選擇 刪除 。
4. 在文字輸入欄位中輸入角色的名稱，然後選擇 刪除 。

若要刪除 Lambda 函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇您建立的函數。
3. 選擇 Actions (動作)、Delete (刪除)。
4. 在文字輸入欄位中輸入 **delete**，然後選擇 刪除 。

管理 AWS Lambda 功能

了解如何使用 Lambda API 或主控台調整及保護與 Lambda 函數相關聯的資源。

搭配使用 Lambda AWS CLI

您可以使用 AWS Command Line Interface 來管理功能和其他 AWS Lambda 資源。會 AWS CLI 使用 AWS SDK for Python (Boto) 與 Lambda API 進行互動。在此教學課程中，您會使用 AWS CLI來管理並調用 Lambda 函數。

函數擴展

您可以設定兩個函數層級的並行控制項：預留並行和佈建並行。並行是作用中函數的執行個體數量，可進行設定以確保關鍵功能避免限流。

程式碼簽署

適用於 Lambda 的程式碼簽署提供信任和完整性控制，可讓您確認只有核准開發人員已發佈的未修改程式碼才會部署在您的 Lambda 函數中。

使用標籤進行整理

您可以標記 Lambda 函數來啟動屬性型存取控制 (ABAC)，並依擁有者、專案或部門分門別類整理。

使用分層

您可以套用先前建立的分層來減少部署套件大小，並促進程式碼共用和責任分離，以便在撰寫商業邏輯時更快速重複執行。

搭配使用 Lambda 與 AWS CLI

您可以使用 AWS Command Line Interface 來管理函式和其他 AWS Lambda 資源。AWS CLI 使用 AWS SDK for Python (Boto) 與 Lambda API 互動。您可使用它來了解 API，並將該知識套用在搭配使用 Lambda 與 AWS 開發套件的建置應用程式中。

在此教學課程中，您會使用 AWS CLI 來管理並調用 Lambda 函數。如需詳細資訊，請參閱 AWS Command Line Interface 使用者指南中的什麼是 AWS CLI？。

必要條件

此教學課程假設您具備基本的 Lambda 操作知識並了解 Lambda 主控台。如果您尚未執行，請依照the section called “使用主控台建立一個 Lambda 函數”中的說明。

若要完成下列步驟，您需要 AWS Command Line Interface (AWS CLI) 版本 2。命令和預期的輸出會列在不同的區塊中：

```
aws --version
```

您應該會看到下列輸出：

```
aws-cli/2.13.27 Python/3.11.6 Linux/4.14.328-248.540.amzn2.x86_64 exe/x86_64.amzn.2
```

對於長命令，逸出字元 (\) 用於將命令分割為多行。

在 Linux 和 macOS 上，使用您偏好的 shell 和套件軟體管理工具。

> **Note**
>
> 在 Windows 中，作業系統的內建終端不支援您常與 Lambda 搭配使用的某些 Bash CLI 命令 (例如 `zip`)。若要取得 Ubuntu 和 Bash 的 Windows 整合版本，請安裝適用於 Linux 的 Windows 子系統。本指南中的 CLI 命令範例使用 Linux 格式。如果您使用的是 Windows CLI，必須重新格式化包含內嵌 JSON 文件的命令。

建立執行角色

建立執行角色，授予您的函式存取 AWS 資源的許可。如要使用AWS CLI 建立執行角色，請使用 `create-role` 命令。

在下列範例中，您可以指定內嵌信任政策。JSON 字串中轉義引號的請求有所不同，這取決於您的 shell。

```
aws iam create-role --role-name lambda-ex --assume-role-policy-document '{"Version":
 "2012-10-17","Statement": [{ "Effect": "Allow", "Principal": {"Service":
 "lambda.amazonaws.com"}, "Action": "sts:AssumeRole"}]}'
```

您也可使用 JSON 檔案來定義角色的信任政策。在下列範例中，`trust-policy.json` 為當前目錄中的檔案。此信任政策會授予服務主體 `lambda.amazonaws.com` 呼叫 AWS Security Token Service (AWS STS) `AssumeRole` 動作的許可，以便 Lambda 能使用角色的許可。

Example trust-policy.json

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

```
aws iam create-role --role-name lambda-ex --assume-role-policy-document file://trust-
policy.json
```

您應該會看到下列輸出：

```
{
    "Role": {
        "Path": "/",
        "RoleName": "lambda-ex",
        "RoleId": "AROAQFOXMPL6TZ6ITKWND",
        "Arn": "arn:aws:iam::123456789012:role/lambda-ex",
        "CreateDate": "2020-01-17T23:19:12Z",
        "AssumeRolePolicyDocument": {
            "Version": "2012-10-17",
            "Statement": [
```

```
                {
                    "Effect": "Allow",
                    "Principal": {
                        "Service": "lambda.amazonaws.com"
                    },
                    "Action": "sts:AssumeRole"
                }
            ]
        }
    }
}
```

使用 attach-policy-to-role 命令將許可新增至角色。透過新增 AWSLambdaBasicExecutionRole 受管政策開始。

```
aws iam attach-role-policy --role-name lambda-ex --policy-arn arn:aws:iam::aws:policy/
service-role/AWSLambdaBasicExecutionRole
```

該AWSLambdaBasicExecutionRole策略具有函數將日誌寫入日誌所需的 CloudWatch 權限。

建立函數

下列範例會記錄環境變數和事件物件的值。

Example index.js

```
exports.handler = async function(event, context) {
  console.log("ENVIRONMENT VARIABLES\n" + JSON.stringify(process.env, null, 2))
  console.log("EVENT\n" + JSON.stringify(event, null, 2))
  return context.logStreamName
}
```

建立函數

1. 將範本程式碼複製到名為 index.js 的檔案。
2. 建立部署套件。

   ```
   zip function.zip index.js
   ```

3. 使用 create-function 命令建立一個 Lambda 函數。使用您的帳戶 ID，取代角色 ARN 中的反白文字。

```
aws lambda create-function --function-name my-function \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs20.x \
--role arn:aws:iam::123456789012:role/lambda-ex
```

您應該會看到下列輸出：

```
{
    "FunctionName": "my-function",
    "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
    "Runtime": "nodejs20.x",
    "Role": "arn:aws:iam::123456789012:role/lambda-ex",
    "Handler": "index.handler",
    "CodeSha256": "FpFMvUhayLkOoVBpNuNiIVML/tuGv2iJQ7t0yWVTU8c=",
    "Version": "$LATEST",
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "RevisionId": "88ebe1e1-bfdf-4dc3-84de-3017268fa1ff",
    ...
}
```

若要從命令列取得某次調用的日誌，請使用 `--log-type` 選項。其回應將包括 `LogResult` 欄位，內含該次調用的 base64 編碼日誌 (最大達 4 KB)。

```
aws lambda invoke --function-name my-function out --log-type Tail
```

您應該會看到下列輸出：

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

您可以使用 `base64` 公用程式將日誌解碼。

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text |  base64 -d
```

您應該會看到下列輸出：

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
  "AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

base64 公用程式適用於 Linux、macOS 以及 Ubuntu on Windows。macOS 請使用 base64 -D 命令。

如需從命令列取得完整日誌事件，您可以在您函式的輸出中納入日誌串流名稱，如上述範例中所示。以下範例指令碼會呼叫名為 my-function 的函式，並下載最後 5 個日誌事件。

Example get-logs.sh 指令碼

此範例需要 my-function 傳回日誌串流 ID。

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-name
 $(cat out) --limit 5
```

該指令碼使用 sed 以從輸出檔案移除引述，並休眠 15 秒以使日誌可供使用。輸出包括來自 Lambda 的回應以及來自 get-log-events 命令的輸出。

```
./get-logs.sh
```

您應該會看到下列輸出：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
```

```
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```

更新函數

建立函數之後，您可以設定函數的其他功能，例如觸發程序、網路存取和檔案系統存取。您也可以調整與函數相關聯的資源，例如記憶體和並行處理。這些組態適用於定義為 .zip 封存檔的函數，以及定義為容器映像的函數。

使用指update-function-configuration令來設定函數。以下範例將函數記憶體設定為 256 MB。

Example update-function-configuration 命令

```
aws lambda update-function-configuration \
--function-name my-function \
--memory-size 256
```

列出您帳戶中的 Lambda 函數

執行下列 AWS CLI `list-functions` 命令，來擷取您已建立的函數清單。

```
aws lambda list-functions --max-items 10
```

您應該會看到下列輸出：

```
{
    "Functions": [
        {
            "FunctionName": "cli",
            "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-
function",
            "Runtime": "nodejs20.x",
            "Role": "arn:aws:iam::123456789012:role/lambda-ex",
            "Handler": "index.handler",
            ...
        },
        {
            "FunctionName": "random-error",
            "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:random-
error",
            "Runtime": "nodejs20.x",
            "Role": "arn:aws:iam::123456789012:role/lambda-role",
            "Handler": "index.handler",
            ...
        },
        ...
      ],
      "NextToken": "eyJNYXJrZXIiOiBudWxsLCAiYm90b190cnVuY2F0ZV9hbW91bnQiOiAxMH0="
}
```

在回應中，Lambda 傳回一份多達 10 個函數的清單。如果有更多的函式可以取回，`NextToken` 會提供可在下一個 `list-functions` 請求中使用的標記。下列 `list-functions` AWS CLI 命令範例清楚說明 `--starting-token` 參數。

```
aws lambda list-functions --max-items 10 --starting-
token eyJNYXJrZXIiOiBudWxsLCAiYm90b190cnVuY2F0ZV9hbW91bnQiOiAxMH0=
```

擷取 Lambda 函數

Lambda CLI `get-function` 命令會傳回 Lambda 函數中繼資料和預先簽署的 URL，供您用來下載函數的部署套件。

```
aws lambda get-function --function-name my-function
```

您應該會看到下列輸出：

```
{
    "Configuration": {
        "FunctionName": "my-function",
        "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
        "Runtime": "nodejs20.x",
        "Role": "arn:aws:iam::123456789012:role/lambda-ex",
        "CodeSha256": "FpFMvUhayLkOoVBpNuNiIVML/tuGv2iJQ7t0yWVTU8c=",
        "Version": "$LATEST",
        "TracingConfig": {
            "Mode": "PassThrough"
        },
        "RevisionId": "88ebe1e1-bfdf-4dc3-84de-3017268fa1ff",
        ...
    },
    "Code": {
        "RepositoryType": "S3",
        "Location": "https://awslambda-us-east-2-tasks.s3.us-east-2.amazonaws.com/
snapshots/123456789012/my-function-4203078a-b7c9-4f35-..."
    }
}
```

如需詳細資訊，請參閱GetFunction。

清除

執行下列 `delete-function` 命令以刪除 `my-function` 函數。

```
aws lambda delete-function --function-name my-function
```

刪除您在 IAM 主控台中建立的 IAM 角色。如需關於刪除角色的詳細資訊，請參閱 IAM 使用者指南中的刪除角色或執行個體設定檔。

了解 Lambda 展函數

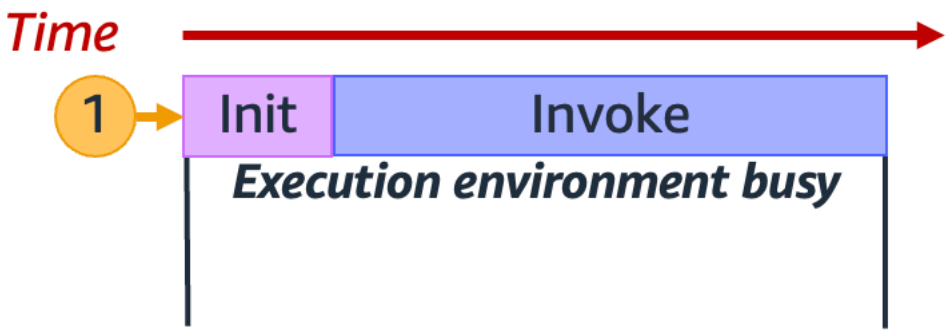
並發性是您的 AWS Lambda 函數同時處理的飛行中請求的數量。Lambda 會針對每個並行請求佈建個別的執行環境執行個體。函數收到更多請求時，Lambda 會自動處理執行環境的擴展數量，直到達到帳戶的並行上限為止。根據預設，Lambda 會為您的帳戶提供一個 AWS 區域中所有函數的總並行上限 1,000 個並行執行數。為了支援您的特定帳戶需求，您可以請求提高配額，並設定函數層級並行控制項，讓您的重要函數不會發生限流。

本主題說明 Lambda 中的並行概念和函數調整。本主題結束時，您將能了解如何計算並行、視覺化兩個主要並行控制選項 (預留和佈建)、預估適當的並行控制設定，以及檢視指標以進一步最佳化。

章節

了解和視覺化並行

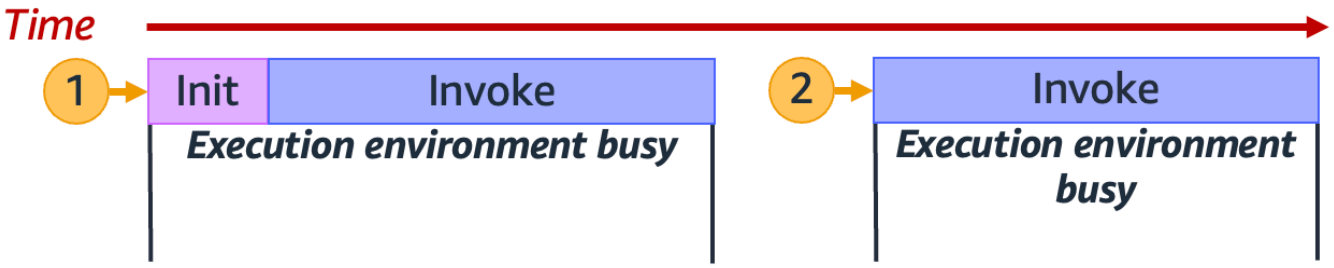
Lambda 會在安全且隔離的執行環境中調用函數。為了處理請求，Lambda 必須先初始化執行環境 (初始化階段)，然後才能用它來調用函數 (調用階段)：



> **Note**
>
> 實際初始化和調用持續時間可能因許多因素而異，例如您選擇的執行階段和 Lambda 函數程式碼。上圖表的用意並非表示初始化和調用階段持續時間的確切比例。

上圖使用矩形來代表單一執行環境。當函數收到第一個請求 (以帶有 1 標籤的黃色圓圈表示)，Lambda 會建立一個新的執行環境，並在初始化階段於主處理常式之外執行程式碼。接著 Lambda 會在調用階段執行函數的主要處理常式程式碼。在整個過程中，此執行環境會處於忙碌狀態，無法處理其他請求。

Lambda 完成處理第一個請求後，此執行環境就可以處理同一個函數的其他請求。Lambda 不需要為後續請求重新初始化環境。

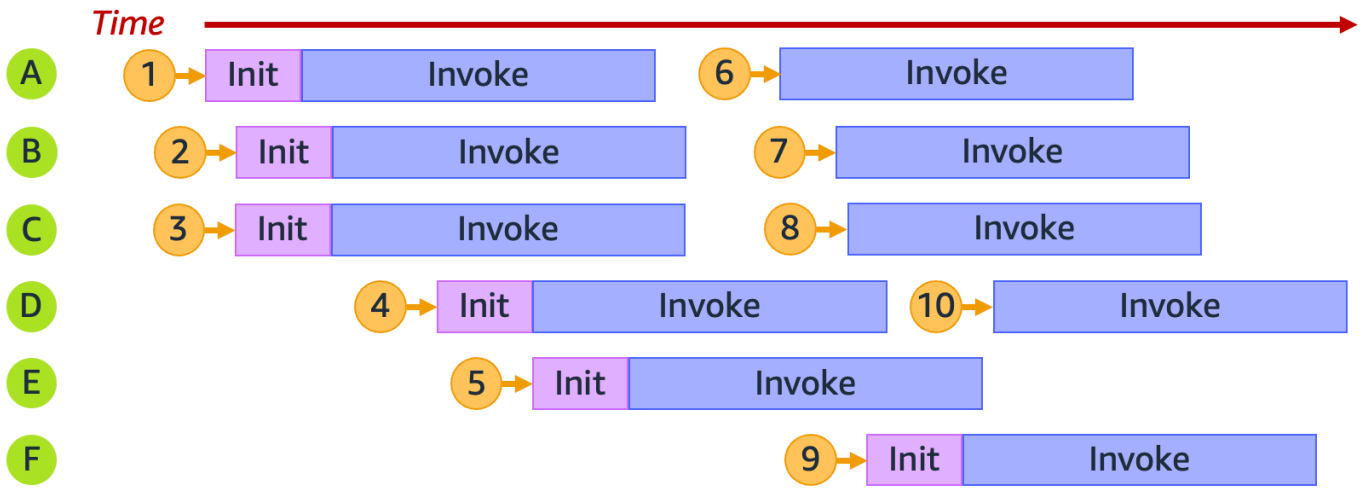


在上圖中，Lambda 重複使用執行環境來處理第二個請求 (以帶 2 標籤的黃色圓圈表示)。

目前為止，我們只將注意力放在執行環境的單一執行個體 (即並行 1)。實際上，Lambda 可能需要平行佈建多個執行環境執行個體，以便處理所有傳入的請求。當函數收到新請求，可能會發生的情況有以下兩種：

- 如果預先初始化的執行環境執行個體可用，Lambda 會用它來處理請求。
- 若不可用，Lambda 便會建立新的執行環境執行個體來處理請求。

例如，我們來看看當函數收到 10 個請求時會發生什麼情形：



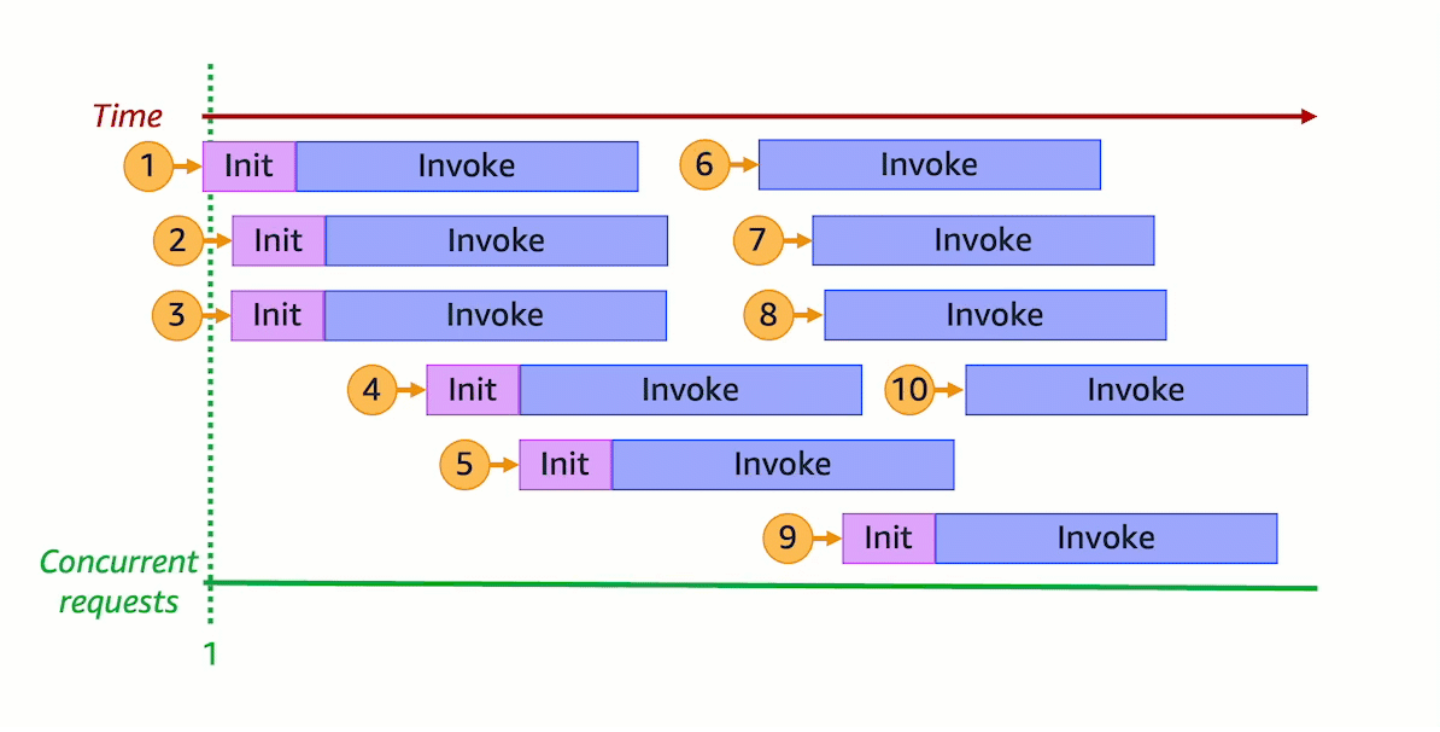
在上圖中，每個水平平面都代表單一執行環境執行個體 (以 A 至 F 標記)。以下是 Lambda 處理每個請求的方式：

Lambda 對請求 1 到 10 的行為

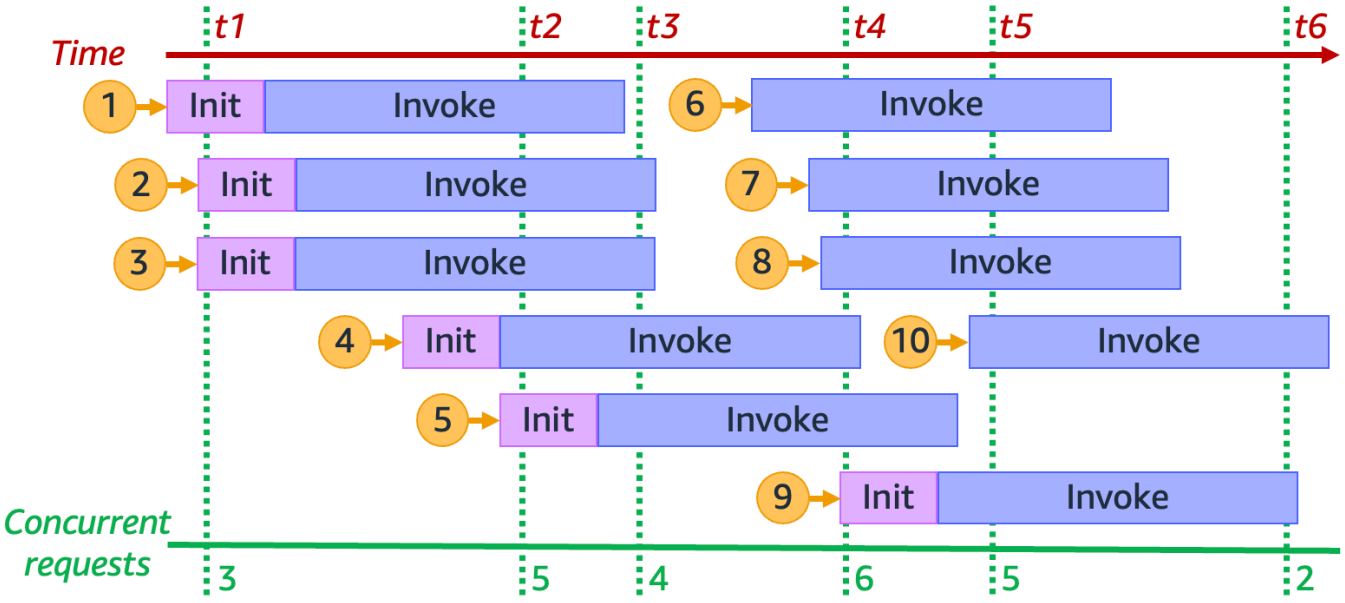
請求	Lambda 行為	推理
1	佈建新環境 A	這是第一個請求；沒有可用的執行環境執行個體。
2	佈建新環境 B	現有的執行環境執行個體 A 忙碌中。
3	佈建新環境 C	現有的執行環境執行個體 A 和 B 都忙碌中。
4	佈建新環境 D	現有的執行環境執行個體 A、B 和 C 都忙碌中。
5	佈建新環境 E	現有的執行環境執行個體 A、B、C 和 D 都忙碌中。
6	重複使用環境 A	執行環境執行個體 A 已完成處理請求 1，現在可供使用。
7	重複使用環境 B	執行環境執行個體 B 已完成處理請求 2，現在可供使用。

請求	Lambda 行為	推理
8	重複使用環境 C	執行環境執行個體 C 已完成處理請求 3，現在可供使用。
9	佈建新環境 F	現有的執行環境執行個體 A、B、C、D 和 E 都忙碌中。
10	重複使用環境 D	執行環境執行個體 D 已完成處理請求 4，現在可供使用。

當函數收到更多並行請求時，Lambda 會擴展執行環境執行個體的數量做為回應。以下動畫追蹤一段時間內並行請求的數目：



將上方動畫凍結在六個不同的時間點，我們可以得到下圖：



在上圖中，我們可以在任何時間點繪製一條垂直線，並計算與此線相交的環境數量。這可以得出該時間點並行請求的數量。例如，在 `t1` 時間點，有三個作用中環境可以處理三個並行請求。在 `t4` 時間點有六個作用中環境可處理六個並行請求，達到此模擬中的並行請求數上限。

總之，函數的並行就是代表函數同時處理的請求數目。為了回應函數並行的增加，Lambda 會佈建更多執行環境執行個體以滿足請求的需求。

計算函數的並發

在一般情況下，一個系統的並行是同時處理多個任務的能力。在 Lambda 中，並行是函數可同時處理的傳輸中請求數目。測量 Lambda 函數並行的快速實用方法是使用以下公式：

```
Concurrency = (average requests per second) * (average request duration in seconds)
```

並行與每秒請求數不同。例如，假設函數平均每秒收到 100個請求。如果平均請求持續時間為一秒，那麼並行也會是 100：

```
Concurrency = (100 requests/second) * (1 second/request) = 100
```

但是，如果平均請求持續時間為 500 毫秒，則並行為 50：

```
Concurrency = (100 requests/second) * (0.5 second/request) = 50
```

在實務中並行 50 代表什麼意思？ 如果平均請求持續時間為 500 毫秒，則可以將函數的一個執行個體視為每秒能處理兩個請求。如此一來，函數需要 50 個執行個體來處理每秒 100 個請求的負載。並行 50 表示 Lambda 必須佈建 50 個執行環境執行個體，才能有效率地處理此工作負載，而不需要進行限流。底下以方程式的形式表示：

```
Concurrency = (100 requests/second) / (2 requests/second) = 50
```

如果函數收到的請求數量是兩倍 (每秒 200 個請求)，但處理每個請求只需要一半的時間 (250 毫秒)，則並行仍然是 50：

```
Concurrency = (200 requests/second) * (0.25 second/request) = 50
```

測驗您對並行的理解

假設您有一個函數，執行時間平均需要 200 毫秒。在峰值負載期間，您觀察到每秒有 5,000 個請求。峰值負載期間函數的並行為何？

答案

平均函數持續時間為 200 毫秒或 0.2 秒。使用並行公式，您可以插入數字來得出並行為 1,000：

```
Concurrency = (5,000 requests/second) * (0.2 seconds/request) = 1,000
```

或者，平均函數持續時間為 200 毫秒表示函數每秒可以處理 5 個請求。若要處理每秒 5,000 個請求的工作負載，您需要 1,000 個執行環境執行個體。因此並行為 1,000：

```
Concurrency = (5,000 requests/second) / (5 requests/second) = 1,000
```

區分並行性和每秒要求

如前一節所述，並行與每秒請求數不同。這是使用平均請求持續時間小於 100 毫秒的函數時特別重要的區別。

通常，執行環境的每個執行個體每秒最多可處理 10 個請求 此限制適用於同步隨需函數，以及使用佈建並行的函數。如果您不熟悉此限制，您可能會不知道為什麼這類函數在某些情況下會遇到限流。

例如，假設有一個平均請求持續時間為 50 毫秒的函數。在每秒 200 個請求時，此函數的並行如下：

```
Concurrency = (200 requests/second) * (0.05 second/request) = 10
```

根據此結果，您可能預期只需要 10 個執行環境執行個體來處理此負載。不過，每個執行環境每秒只能處理 10 個執行。也就是說，在 10 個執行環境中，您的函數每秒只能處理 200 個請求總數中的 100 個請求。此函數遇到了限流。

因此，在為函數配置並行設定時，您必須同時考慮並行和每秒請求數。在這種情況下，您的函數需要 20 個執行環境，即使它只有 10 個並行。

測試您對並行的理解 (低於 100 毫秒函數)

假設您有一個函數，執行時間平均需要 20 毫秒。在峰值負載期間，您觀察到每秒有 3,000 個請求。峰值負載期間函數的並行為何？

答案

平均函數持續時間為 20 毫秒或 0.02 秒。使用並行公式，您可以插入數字來得出並行為 60：

```
Concurrency = (3,000 requests/second) * (0.02 seconds/request) = 60
```

不過，每個執行環境每秒只能處理 10 個請求。使用 60 個執行環境，您的函數每秒最多可以處理 600 個請求。若要完全消化 3,000 個請求，函數至少需要 300 個執行環境的執行個體。

瞭解保留並行與佈建的並行

根據預設，您帳戶設有一個區域中所有函數的並行執行數上限 1,000。您的函數會隨需共用此 1,000 並行的集區。如果用完可用並行，函數便會發生限流 (即開始捨棄請求)。

某些函數的重要性可能高於其他函數。因此，您可能會想配置並行設定，確保重要函數可獲得所需的並行。並行控制項有兩種：預留並行和佈建並行。

- 使用預留並行將帳戶的一部分並行預留給某個函數。如果您不希望其他函數占用所有可用的未預留並行，此功能非常有用。
- 使用佈建並行為函數預先初始化多個環境執行個體。這對於縮短冷啟動延遲很有幫助。

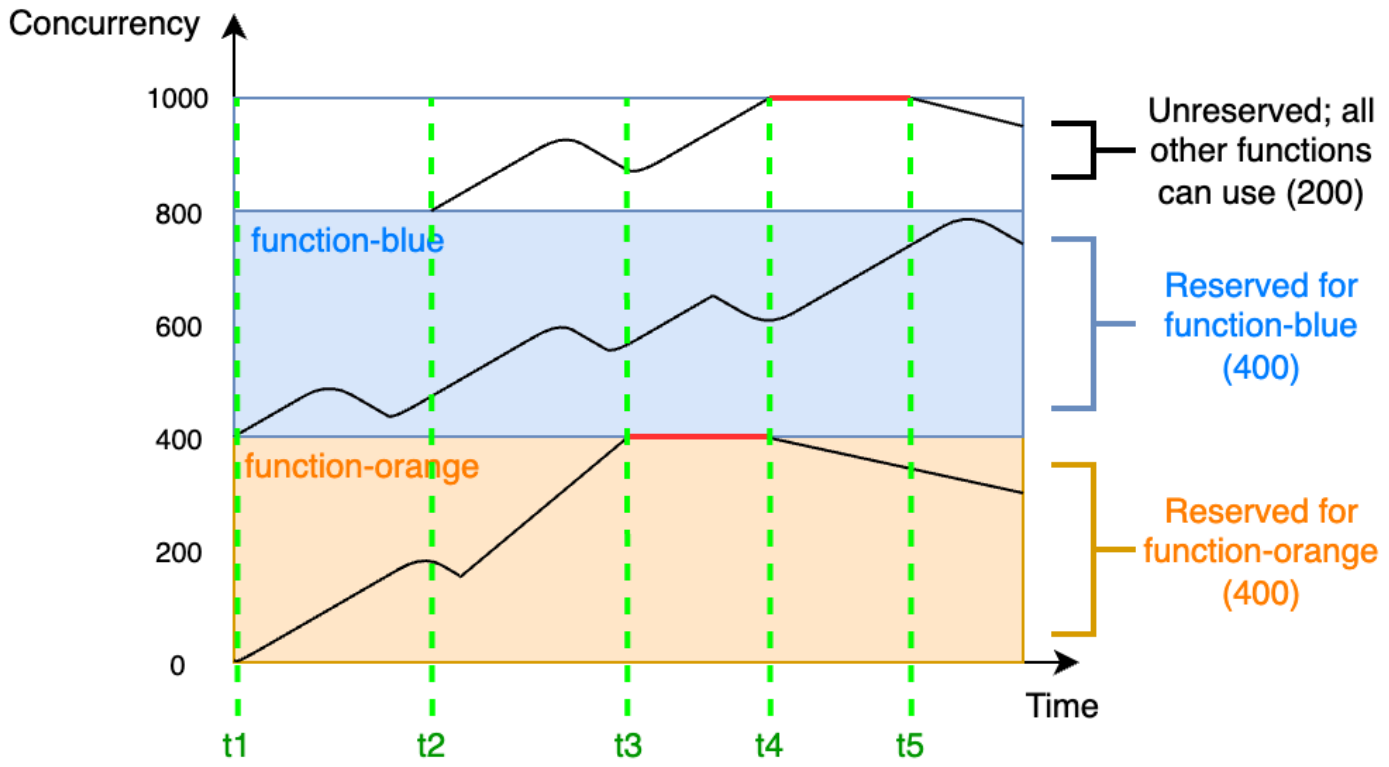
預留並行

如果您想要保證函數隨時可以使用一定數量的並行，請使用預留並行。

預留並行是要分配給函數的並行執行個體數量上限。將預留並行專門留給某個函數時，其他函數都無法使用該並行。換句話說，設定預留並行可能會影響其他函數可用的並行集區。沒有預留並行的函數會共用未預留剩餘的並行集區。

設定預留並行會計入您的整體帳戶並行上限。設定函數的預留並行不收費。

為了更清楚理解預留並行，請參考下圖：



在本圖中，您的帳戶在此區域中所有函數的並行上限為預設值 1,000。假設您有兩個重要函數 `function-blue` 和 `function-orange`，且經常預期出現高調用量。您決定將 400個單位的預留並行提供給 `function-blue`，並為 `function-orange` 提供 400 單位的預留並行。在此範例中，您帳戶中的所有其他函數必須共用剩餘 200 單位的未預留並行。

本圖有五個特點：

- 在 `t1`，`function-orange` 和 `function-blue` 都開始接收請求。每個函數開始使用自己分配到的預留並行單元。
- 在 `t2`，`function-orange` 和 `function-blue` 穩定接收更多的請求。同時，您部署了一些其他 Lambda 函數，且這些函數開始接收請求。您沒有將預留並行分配給這些函數。這些函數開始使用剩餘 200 單位的未預留並行。

- 在 t3，function-orange 達到並行上限 400。雖然您帳戶中的其他地方存在未使用的並行，但 function-orange 無法存取。紅線表示 function-orange 正在進行限流，且 Lambda 可能會捨棄請求。
- 在 t4，function-orange 接收的請求開始變少，而且不再限流。但是，其他函數出現流量尖峰並開始限流。雖然您帳戶中的其他地方存在未使用的並行，但這些其他函數無法存取。紅線表示其他函數正在進行限流。
- 在 t5，其他函數接收的請求開始變少，而且不再限流。

在此範例中，請注意預留並行產生了下列影響：

- 函數可以獨立於帳戶中的其他函數進行擴展。您的帳戶下相同區域中沒有預留並行的所有函數，都會共用未預留的並行集區。如果沒有預留並行，其他函數可能會用盡所有可用的並行。這可以視需要防止您的重要數擴展。
- 您的函數無法無止盡擴展。預留並行會對函數的最大並行設定上限。這表示函數不能使用為其他函數預留的並行，也不能使用未預留集區中的並行。您可以預留並行以防止函數用完帳戶中的所有可用並行，或過載下游資源。
- 您可能無法使用帳戶可用的所有並行。預留並行會計入您帳戶的並行上限，但這也表示其他函數無法使用該部分預留並行。如果函數沒有用盡您為它預留的所有並行，那麼實際上就浪費了這些並行。除非浪費的並行可讓您帳戶中的其他函數獲益，這才不會成為問題。

若要管理函數的預留並行設定，請參閱：為函數配置保留並發。

佈建並行

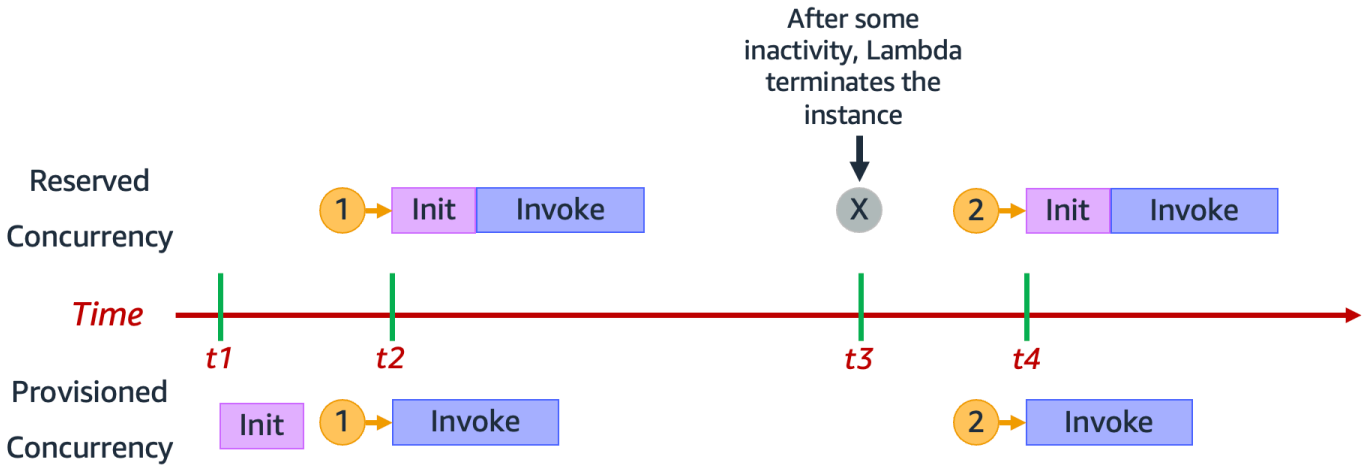
您可以使用預留並行來定義為 Lambda 函數預留的執行環境數目上限。不過，這些環境都不會預先初始化。因此，函數調用可能需要花更長的時間，因為 Lambda 必須先初始化新環境，才能用來調用函數。當 Lambda 必須初始化新環境才能執行調用，就稱為冷啟動。若要緩解冷啟動情形，您可以使用佈建並行。

佈建並行是您要分配給函數的預先初始化執行環境數。如果您為函數設定了佈建並行，Lambda 便會初始化該數量的執行環境，以便立即回應函數請求。

> **Note**
>
> 設定佈建並行時，您的帳戶會產生額外費用。如果您正在使用 Java 11 或 Java 17 執行階段，您也可以使用 Lambda SnapStart 來緩解冷啟動問題，而無需額外付費。 SnapStart 使用執行環境的快取快照，大幅改善啟動效能。您不能在相同的函數版本上同時使用 SnapStart 和佈建

的並行。如需 SnapStart 功能、限制和支援區域的詳細資訊，請參閱使用 Lambda 改善啟動效能 SnapStart。

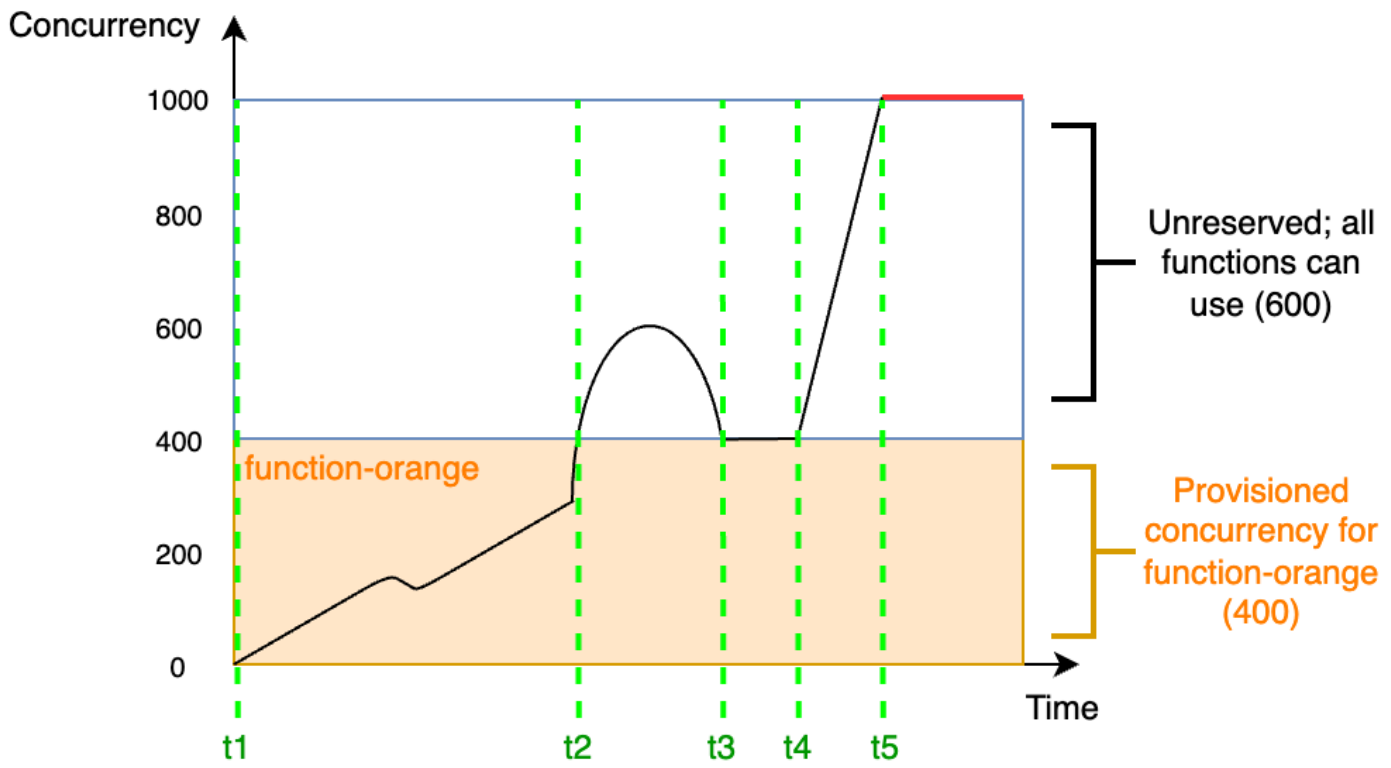
使用佈建並行時，Lambda 仍會在背景回收執行環境。但是在任何給定時間，Lambda 總是會確保預先初始化的環境數量等於函數佈建並行設定的值。此行為與預留並行不同，Lambda 可能會在閒置一段時間後完全終止環境。下圖說明這一點，比較為函數設定預留並行和佈建並行的情況下，單一執行環境的生命週期有何差異。



本圖有四個特點：

時間	預留並行	佈建並行
t1	什麼都沒發生。	Lambda 會預先初始化執行環境執行個體。
t2	請求 1 傳入。Lambda 必須初始化新的執行環境執行個體。	請求 1 傳入。Lambda 使用預先初始化的環境執行個體。
t3	閒置一段時間後，Lambda 會終止作用中環境執行個體。	什麼都沒發生。
t4	請求 2 傳入。Lambda 必須初始化新的執行環境執行個體。	請求 2 傳入。Lambda 使用預先初始化的環境執行個體。

為了更清楚理解佈建並行，請參考下圖：



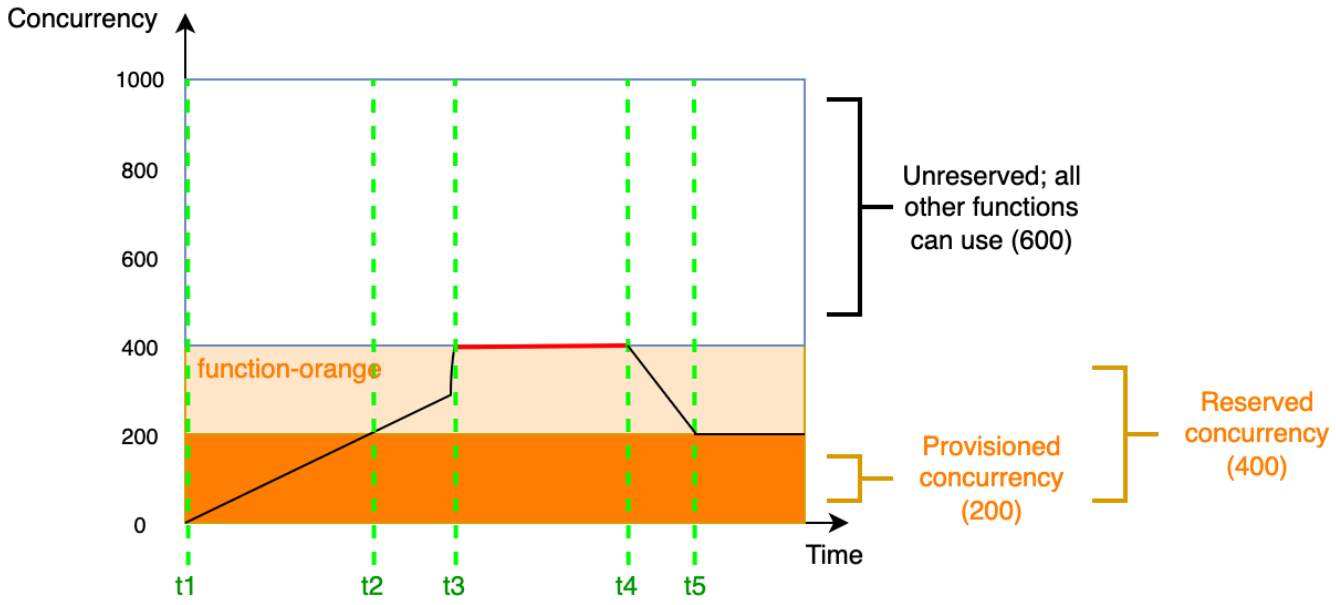
在此圖中，您的帳戶並行上限為 1,000。您決定將 400 單位的佈建並行提供給 `function-orange`。您帳戶中包括 `function-orange` 在內的所有函數，都可以使用剩餘 600 單位的未預留並行。

本圖有五個特點：

- 在 `t1`，`function-orange` 開始接收請求。由於 Lambda 已預先初始化 400 個執行環境執行個體，因此 `function-orange` 已準備好立即進行調用。
- 在 `t2`，`function-orange` 達到 400 個並行請求。因此，`function-orange` 用盡了佈建並行。但是由於仍有未預留並行可用，Lambda 還是可以用來處理 `function-orange` 的其他請求 (沒有限流)。Lambda 必須建立新的執行個體來處理這些請求，而且函數可能會發生冷啟動延遲。
- 在 `t3`，流量短暫達到峰值後 `function-orange` 回到 400 個並行請求。Lambda 再度能在沒有冷啟動延遲的情況下處理所有請求。
- 在 `t4`,帳戶中的函數發生流量暴增情形。此暴增可能來自 `function-orange` 或帳戶中的任何其他函數。Lambda 使用未預留並行處理這些請求。
- 在 `t5`，帳戶中的函數達到最大並行上限 1,000，並且發生限流。

上一個範例只考慮了佈建並行。實際上，您可以對函數同時設定佈建並行和預留並行。如果您有一個函數在工作日期間負責處理調用的一致性負載，但週末期間經常會遇到流量峰值，便可以這麼做。在這種

情況下，您可以使用佈建並行設定環境的基準數量來處理平日的請求，並使用預留並行處理週末峰值流量。請思考下圖情形：



在此圖中，假設您為 `function-orange` 設定了 200 單位的佈建並行，以及 400 單位的預留並行。因為您已設定預留並行，因此 `function-orange` 完全無法使用 600 單位的未預留並行。

此圖有 5 個特點：

- 在 `t1`，`function-orange` 開始接收請求。由於 Lambda 已預先初始化 200 個執行環境執行個體，因此 `function-orange` 已準備好立即進行調用。
- 在 `t2`，`function-orange` 用盡了所有佈建並行。`function-orange` 可以使用預留並行繼續處理請求，但這些請求可能會遇到冷啟動延遲。
- 在 `t3`，`function-orange` 達到 400 個並行請求。因此，`function-orange` 用盡了所有預留並行。由於 `function-orange` 不能使用未預留並行，因此請求開始限流。
- 在 `t4`，`function-orange` 接收的請求開始變少，而且不再限流。
- 在 `t5`，`function-orange` 下降到 200 個並行請求，因此所有請求都能再次使用佈建並行 (即無冷啟動延遲)。

預留並行和佈建並行都會計入您的帳戶並行上限和區域配額中。換句話說，分配預留和佈建並行可能會影響其他函數可用的並行集區。設定已佈建的並行會產生您的. AWS 帳戶

> **Note**
>
> 如果在函數版本與別名功能上佈建的並行數量加到函數的預留並行，則所有呼叫都會在佈建的並行上執行。此組態也有對未發佈版本函數 ($LATEST) 進行節流的效果，以防止其執行。您無法配置多於函數預留並行的佈建並行。

若要管理函數的佈建並行設定，請參閱 設定函數的佈建並行。若要根據排程或應用程式使用率自動佈建並行擴展，請參閱 使用應用程式 Auto Scaling 自動化佈建的並行管理。

Lambda 如何配置佈建並行

佈建並行不會在設定後立即上線。Lambda 會在準備一兩分鐘後，開始配置佈建的並行。對於每個函數，Lambda 每分鐘最多可佈建 6,000 個執行環境，無論是什麼 AWS 區域。這與函數的並行縮放速率完全相同。

當您提交配置佈建並行的請求時，在 Lambda 完全配置完成之前，您無法存取任何這些環境。例如，如果您請求 5,000 個佈建並行，則在 Lambda 完全完成 5,000 個執行環境的配置之前，您的任何請求都不能使用佈建的並行處理。

比較預留並行和佈建並行

下表總結並比較預留和佈建並行。

主題	預留並行	佈建並行
定義	函數的執行環境執行個體數目上限。	為函數設定預先佈建的執行環境執行個體數目。
佈建行為	Lambda 會隨需佈建新的執行個體。	Lambda 會預先佈建執行個體 (即在函數開始接收請求之前)。
冷啟動行為	由於 Lambda 必須隨需建立新執行個體，因此可能會發生冷啟動延遲。	由於 Lambda 不需要隨需建立執行個體，因此不可能會冷啟動延遲。
限流行為	達到預留並行上限時，會對函數限流。	如果沒有設定預留並行：達到佈建並行上限時，函數會使用未預留的並行。

主題	預留並行	佈建並行
		如果有設定預留並行：達到預留並行上限時，會對函數限流。
未設定情況下的預設行為	函數會使用帳戶中可用的未預留並行。	Lambda 不會預先佈建任何執行個體。反之，如果沒有設定預留並行：函數會使用帳戶中可用的未預留並行。 如果有設定預留並行：函數會使用預留並行。
定價	不收取額外費用。	會產生額外費用。

並行配額

Lambda 會針對區域中所有函數可用的並行總量設定配額。這些配額存在於兩個層級：

- 帳戶層級，預設情況下，函數最多可有 1,000 單位的並行。若要提升配額，請參閱《Service Quotas 使用者指南》中的請求提升配額。
- 函數層級，預設情況下，您可在所有函數間保留最多 900 單位的並行。無論您的總帳戶並行限制為何，Lambda 一律會為未明確保留並行保留的函數保留 100 個並行單位。例如，如果您將帳戶並行上限增加到 2,000，就可以在函數層級預留最多 1,900 單位的並行。

若要檢查您目前的帳戶層級並行配額，請使用 AWS Command Line Interface (AWS CLI) 執行下列命令：

```
aws lambda get-account-settings
```

您應該會看到類似以下的輸出：

```
{
    "AccountLimit": {
        "TotalCodeSize": 80530636800,
        "CodeSizeUnzipped": 262144000,
        "CodeSizeZipped": 52428800,
```

```
        "ConcurrentExecutions": 1000,
        "UnreservedConcurrentExecutions": 900
    },
    "AccountUsage": {
        "TotalCodeSize": 410759889,
        "FunctionCount": 8
    }
}
```

`ConcurrentExecutions` 是您的帳戶層級並行配額總計。`UnreservedConcurrentExecutions` 是您仍可配置給函數的保留並行數量。

函數收到更多請求時，Lambda 會自動提高執行環境的數量來處理這些請求，直到您的帳戶達到並行配額為止。但是，為了防止因應突然爆發的流量而過度擴展，Lambda 限制了函數擴展的速度。這種並行擴展率是您帳戶中的函數可以根據不斷增加的請求而擴展的最大速率。(也就是說，Lambda 建立新執行環境的速度可以有多快。) 並行調整比率與帳戶層級並行限制不同，也就是函數可用的並行總金額。

在每個 AWS 區域函數和每個函數中，您的並行擴展速率為每 10 秒 1,000 個執行環境執行個體。換句話說，Lambda 每隔 10 秒就可以為每個函數配置最多 1,000 個額外執行環境的執行個體。

通常情況下，您不需要擔心此限制。對於大多數使用案例，Lambda 的擴展速率已足夠。

重要的是，並行縮放速率是函數層級限制。這意味著帳戶中的每個函數都可以獨立於其他函數進行擴展。

如需擴展行為的詳細資訊，請參閱 Lambda 擴展行為。

為函數配置保留並發

在 Lambda 中，並行是函數目前正在處理的傳輸中請求數目。有兩種類型的並行控制項可用：

- 預留並行 – 它是分配給函數的並行執行個體數量上限。當某個函數具有預留並行時，其他函數都無法使用該並行。保留並發對於確保您最關鍵的函數始終具有足夠的並發性來處理傳入的請求非常有用。設定函數的預留並行不會產生額外費用。
- 佈建並行 – 它是您分配給函數的預先初始化執行環境數目。這些執行環境已準備好立即回應傳入的函數請求。佈建的並行對於減少功能的冷啟動延遲非常有用。設定已佈建的並行會產生額外費用給您的. AWS 帳戶

本主題詳細說明如何管理及設定預留並行。如需這兩種並行控制項的概念性概觀，請參閱預留並行與佈建並行。如需有關設定佈建並行的資訊，請參閱 the section called “設定佈建並行”。

> **Note**
>
> 連結至 Amazon MQ 事件來源映射的 Lambda 函數具有預設並行上限。若使用 Apache Active MQ，並行執行個體數量上限為 5 個。若使用 ARabbit MQ，並行執行個體數量上限為 1 個。為函數設定保留或佈建並行不會改變這些上限。若要在使用 Amazon MQ 時要求增加預設的並行上限，請聯絡 AWS Support。

章節

- 設定預留並行
- 準確估算函數所需的保留並發

設定預留並行

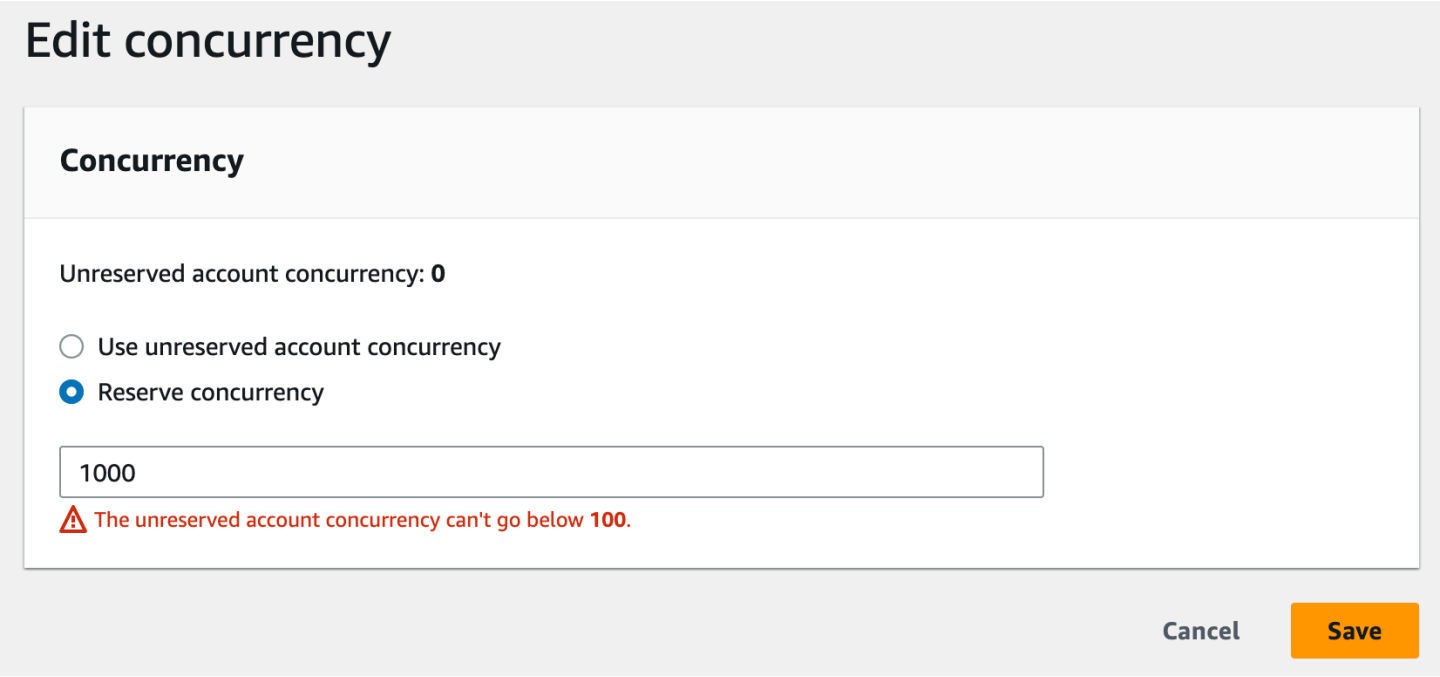
您可以使用 Lambda 主控台或 Lambda API 設定函數的預留並行設定。

預留函數並行的方式 (主控台)

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇要預留並行的函數。
3. 選擇 Configuration (組態)，然後選擇 Concurrency (並行)。
4. 在 Concurrency (並行) 下，選擇 Edit (編輯)。

5. 選擇 Reserve concurrency (預留並行)。輸入要為函式預留的並行數量。
6. 選擇儲存。

您最多可以預留的數量為未預留帳戶並行數量減去 100 的值。剩餘的 100 個並行單位會用於未使用預留並行的函數。例如，如果您帳戶的並行上限為 1,000，則您無法為單一函數預留全部 1,000 個並行單位。



為函數預留並行會影響其他函數可用的並行集區。例如，如果您為 `function-a` 預留 100 個並行單位，則帳戶中的其他函數必須共用剩餘的 900 個並行單位 (即使 `function-a` 未使用全部 100 個預留並行單位也一樣)。

若要刻意節流函數，請將其預留並行設為 0。此動作會防止函數處理任何事件，直到您移除限制為止。

若要透過 Lambda API 設定預留並行，請使用下列 API 操作。

- PutFunction并发性
- GetFunction并发性
- DeleteFunction并发性

例如，若要使用 AWS Command Line Interface (CLI) 設定保留並行，請使用命`put-function-concurrency`令。下列命令會為名為 `my-function` 的函數預留 100 個並行單位：

```
aws lambda put-function-concurrency --function-name my-function \
```

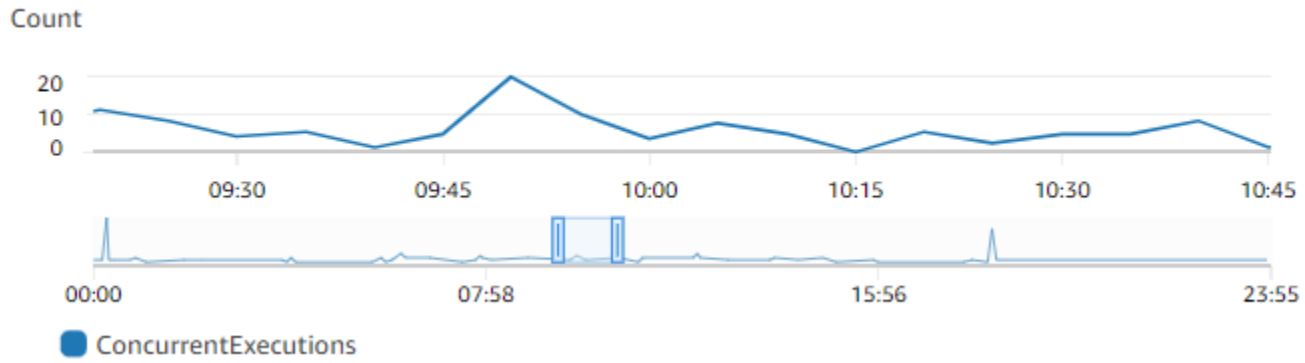
```
    --reserved-concurrent-executions 100
```

您應該會看到類似以下的輸出：

```
{
    "ReservedConcurrentExecutions": 100
}
```

準確估算函數所需的保留並發

如果您的函數目前正在為流量提供服務，則可以使用指標輕鬆檢視其並行指CloudWatch 標。具體而言，ConcurrentExecutions 指標會顯示帳戶中每個函數的並行調用次數。



前一張圖表顯示此函數一天平均可處理 5 到 10 個並行請求，峰值時段則為 20 個請求。假設帳戶中還有很多其他函數。如果此函數對您的應用程式很重要，且您不想要捨棄任何請求，則請將預留並行設為 20 (或以上)。

或者，回想一下您也可以使用以下公式計算並行：

```
Concurrency = (average requests per second) * (average request duration in seconds)
```

將每秒平均請求乘以平均請求持續時間 (以秒為單位)，可讓您粗略估計需要預留多少並行。您可以使用 Invocation 指標來預估每秒平均請求數，並使用 Duration 指標預估平均請求持續時間 (以秒為單位)。如需詳細資訊，請參閱使用 Lambda 函數指標。

設定函數的佈建並行

在 Lambda 中，並行是函數目前正在處理的傳輸中請求數目。有兩種類型的並行控制項可用：

- 預留並行 – 它是分配給函數的並行執行個體數量上限。當某個函數具有預留並行時，其他函數都無法使用該並行。保留並發對於確保您最關鍵的函數始終具有足夠的並發性來處理傳入的請求非常有用。設定函數的預留並行不會產生額外費用。
- 佈建並行 – 它是您分配給函數的預先初始化執行環境數目。這些執行環境已準備好立即回應傳入的函數請求。佈建的並行對於減少功能的冷啟動延遲非常有用。設定已佈建的並行會產生額外費用給您的. AWS 帳戶

本主題詳細說明如何管理及設定佈建並行。如需這兩種並行控制項的概念性概觀，請參閱預留並行與佈建並行。如需有關設定預留並行的詳細資訊，請參閱the section called “設定預留並行”。

> **Note**
>
> 連結至 Amazon MQ 事件來源映射的 Lambda 函數具有預設並行上限。若使用 Apache Active MQ，並行執行個體數量上限為 5 個。若使用 ARabbit MQ，並行執行個體數量上限為 1 個。為函數設定保留或佈建並行不會改變這些上限。若要在使用 Amazon MQ 時要求增加預設的並行上限，請聯絡 AWS Support。

章節

設定佈建並行

您可以使用 Lambda 主控台或 Lambda API 設定函數的佈建並行設定。

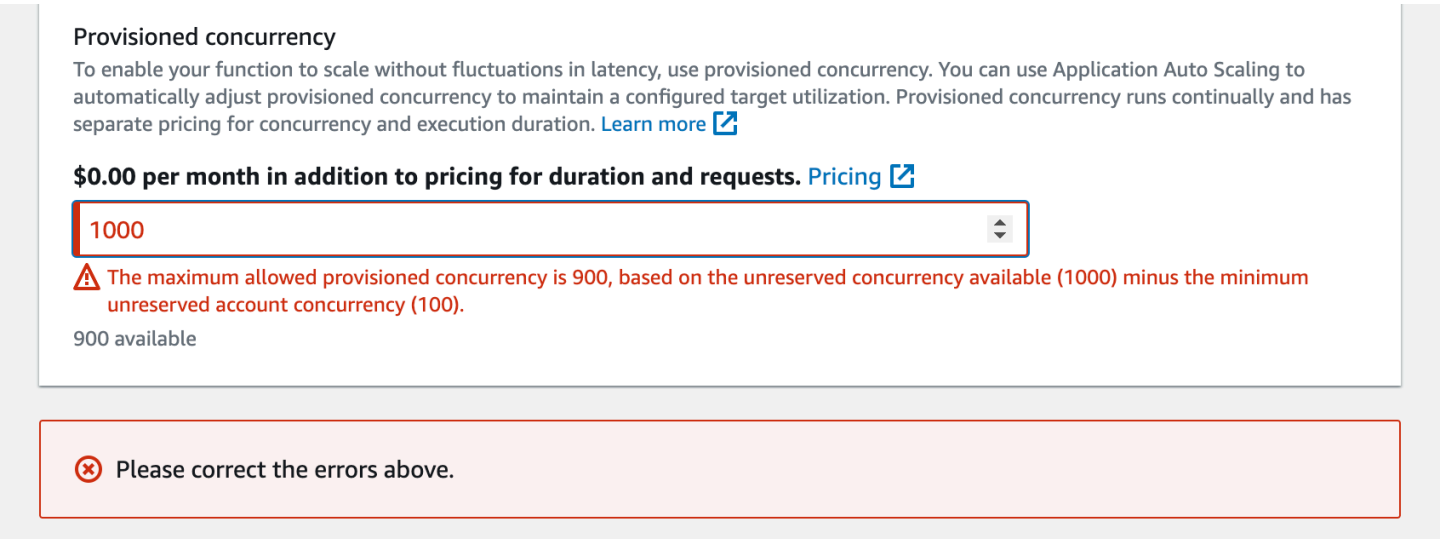
若要為函數配置佈建並行 (主控台)

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇您要配置佈建並行的函數。
3. 選擇 Configuration (組態)，然後選擇 Concurrency (並行)。
4. 在 Provisioned concurrency configurations (佈建並行組態) 下方，選擇 Add configuration (新增組態)。
5. 選擇限定詞類型，以及別名或版本。

> **Note**
>
> 您無法將佈建並行與任何函數的 $LATEST 版本搭配使用。
> 如果您的函數有事件來源，請確定事件來源指向正確的函數別名或版本。否則，您的函數將不會使用佈建並行環境。

6. 在佈建並行下輸入一個數字。Lambda 會提供每月預估費用。
7. 選擇儲存。

您最多可以在帳戶中設定的數量為未預留帳戶並行數量減去 100 的值。剩餘的 100 個並行單位會用於未使用預留並行的函數。例如，如果帳戶的並行限制為 1,000，而且您尚未將任何預留或佈建並行指派給任何其他函數，則單一函數最多可以設定 900 個佈建並行單位。



為函數設定佈建並行會影響其他函數可用的並行集區。例如，如果您為 `function-a` 設定 100 個佈建並行單位，則帳戶中的其他函數必須共用剩餘的 900 個並行單位。即使 `function-a` 沒有使用所有 100 個單位也是如此。

您可以為同一函數同時配置預留並行和佈建並行。在這種情況下，佈建並行不能超過預留並行。

此限制也適用於不同函數版本。您可以指派給特定函數版本的佈建並行上限等於函數的預留並行減去其他函數版本上的佈建並行。

若要透過 Lambda API 設定佈建並行，請使用下列 API 操作。

- PutProvisionedConcurrencyConfig
- GetProvisionedConcurrencyConfig
- ListProvisionedConcurrencyConfigs
- DeleteProvisionedConcurrencyConfig

例如，若要使用 AWS Command Line Interface (CLI) 設定已佈建並行，請使用命`put-provisioned-concurrency-config`令。下列命令會為名為 `my-function` 之函數的 `BLUE` 別名配置 100 個佈建並行單位：

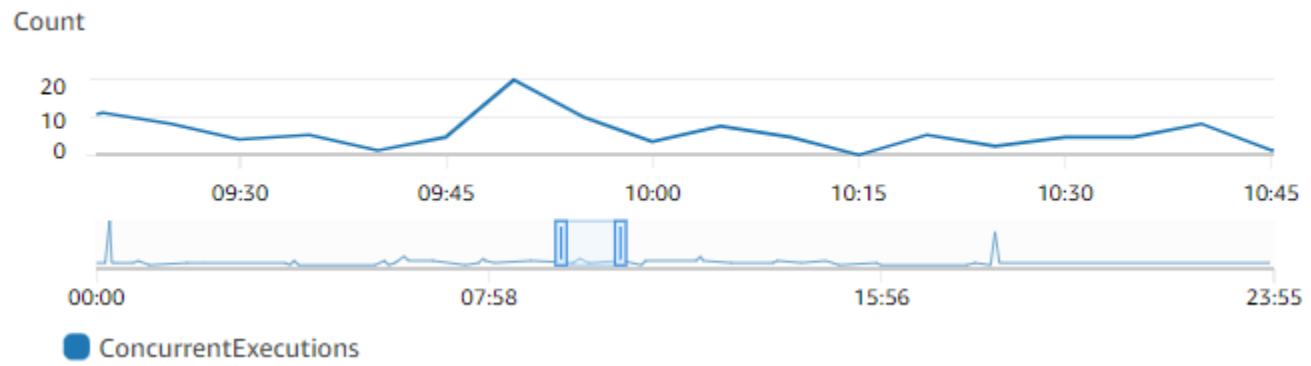
```
aws lambda put-provisioned-concurrency-config --function-name my-function \
  --qualifier BLUE \
  --provisioned-concurrent-executions 100
```

您應該會看到類似以下的輸出：

```
{
  "Requested ProvisionedConcurrentExecutions": 100,
  "Allocated ProvisionedConcurrentExecutions": 0,
  "Status": "IN_PROGRESS",
  "LastModified": "2023-01-21T11:30:00+0000"
}
```

準確估算函數所需的佈建並行

您可以使用量度檢視任何作用中函數的並行量CloudWatch 度。具體而言，`ConcurrentExecutions` 指標會顯示帳戶中函數的並行調用次數。



前一張圖表顯示此函數在任何特定時間平均可處理 5 到 10 個並行請求，峰值時段則為 20 個請求。假設帳戶中還有很多其他函數。如果此函數對您的應用程式很重要，而且每次調用都需要低延遲回應，請將佈建並行設為至少 20 個單位。

回想一下，您也可以使用以下公式計算並行：

```
Concurrency = (average requests per second) * (average request duration in seconds)
```

若要預估您需要多少並行，將每秒平均請求乘以平均請求持續時間 (以秒為單位)。您可以使用 `Invocation` 指標來預估每秒平均請求數，並使用 `Duration` 指標預估平均請求持續時間 (以秒為單位)。

設定佈建並行時，Lambda 建議在函數通常需要的並行量上額外加上 10% 的緩衝。例如，如果函數通常在 200 個並行請求達到峰值，可將佈建並行設定為 220 (200 個並行請求 + 10% = 220 佈建並行)。

使用佈建並行時最佳化函數程式碼

如果您使用的是佈建並行，請考慮重新建構函數程式碼以最佳化低延遲。對於使用佈建並行的函數，Lambda 會在配置期間執行任何初始化程式碼，例如載入程式庫和實體化用戶端。因此，建議將盡可能多的初始化移至主要函數處理常式以外，以避免在實際調用函數時影響延遲。相比之下，在主處理常式程式碼中初始化程式庫或實例化用戶端表示您的函數必須在每次叫用時執行此函數 (無論您是否使用已佈建的並行，都會發生這種情況)。

對於隨需調用，每次函數冷啟動時，Lambda 可能需要重新執行初始化程式碼。針對此類函數，您可以選擇延遲特定功能的初始化，直至您的函數需要該功能。例如，請參閱下列 Lambda 處理常式的控制流程：

```
def handler(event, context):
    ...
```

```
    if ( some_condition ):
        // Initialize CLIENT_A to perform a task
    else:
        // Do nothing
```

在前面的範例中，開發人員在 `if` 陳述式中初始化 `CLIENT_A`，而不是在主要處理常式之外進行初始化。如此一來，Lambda 只會在滿足 `some_condition` 的情況下執行此程式碼。如果您在主要處理常式之外初始化 `CLIENT_A`，Lambda 會在每次冷啟動時執行該程式碼。這可能會增加整體延遲。

使用環境變數來檢視及控制佈建的並行行為

您的函數可能會耗盡本身的所有佈建並行。Lambda 使用隨需執行個體來處理任何超出流量。為了判斷 Lambda 用於特定環境的初始化類型，請檢查 `AWS_LAMBDA_INITIALIZATION_TYPE` 環境變數的值。此變數有兩個可能的值：`provisioned-concurrency` 或 `on-demand`。`AWS_LAMBDA_INITIALIZATION_TYPE` 的值不可變，並且在整個環境的生命週期中保持不變。若要檢查函數程式碼中環境變數的值，請參閱???。

如果您使用 .NET 6 或 .NET 7 執行期，您可以設定 `AWS_LAMBDA_DOTNET_PREJIT` 環境變數，以改善函數的延遲 (即使函數未使用佈建並行)。.NET 執行期會對第一次呼叫的每個程式庫採用延遲編譯和初始化。因此，Lambda 函數的第一次調用可能需要比後續調用更長的時間。若要減輕此問題，您可以針對 `AWS_LAMBDA_DOTNET_PREJIT` 選擇以下三個值之一：

- `ProvisionedConcurrency`：Lambda 使用佈建的 ahead-of-time 並行功能，針對所有環境執行 JIT 編譯。這是預設值。
- `Always`：即使函數不使用佈建的並行，Lambda 也會針對每個環境執行 ahead-of-time JIT 編譯。
- `Never`：Lambda 會停用所有環境的 ahead-of-time JIT 編譯功能。

透過佈建並行了解記錄和計費行為

針對佈建並行環境，函數的初始化程式碼會在配置期間執行，且定期執行一次，因為 Lambda 會回收環境的執行個體。在環境執行個體處理請求之後，您可以在記錄和追蹤中查看初始化時間。請特別注意，即使環境執行個體從未處理請求，Lambda 也會針對初始化計費。佈建並行會持續執行，並與初始化和調用成本分開計費。如需詳細資訊，請參閱 AWS Lambda 定價。

此外，當您使用佈建並行設定 Lambda 函數時，Lambda 會預先初始化該執行環境，以便在函數叫用請求之前提供該函數。但是，您的函數僅在實際調用函數時 CloudWatch 才發布調用日誌。因此，即使初始化事先發生，初始化持續時間欄位仍會出現在第一個函式叫用的`REPORT`記錄行中。這並不意味著該功能經歷了冷啟動。

使用應用程式 Auto Scaling 自動化佈建的並行管理

Application Auto Scaling 可讓您依據排程或根據使用率來管理佈建並行。如果您的函數接收到可預測的流量模式，請使用排程擴展。如果您想要讓函數維持特定的使用率百分比，請使用目標追蹤擴展政策。

排程擴展

Application Auto Scaling 可讓您根據可預測的負載變化來設定自己的擴展排程。如需詳細資訊和範例，請參閱應用程式 Auto Scaling 使用指南中的應用程式自動調整排程調整，以及在 AWS Compute Blog 上針對週期尖峰使用量排定 AWS Lambda佈建的並行。

目標追蹤

透過目標追蹤，Application Auto Scaling 會根據您定義擴展政策的方式，建立和管理一組 CloudWatch 警示。這些警示啟動時，Application Auto Scaling 會自動調整使用佈建並行配置的環境數量。對沒有可預測流量模式的應用程式使用目標追蹤。

若要使用目標追蹤擴展佈建並行，請使用 `RegisterScalableTarget` 和 `PutScalingPolicy` Application Auto Scaling API 作業。例如，如果您使用的是 AWS Command Line Interface (CLI)，請依照下列步驟執行：

1. 請將函數的別名註冊為擴展目標。以下範例會為函數 `my-function` 註冊別名 BLUE：

```
aws application-autoscaling register-scalable-target --service-namespace lambda \
    --resource-id function:my-function:BLUE --min-capacity 1 --max-capacity 100 \
    --scalable-dimension lambda:function:ProvisionedConcurrency
```

2. 將擴展政策套用至目標。下列範例會設定「Application Auto Scaling」，以調整別名的佈建並行組態，使使用率保持接近 70%，但您可以套用 10% 到 90% 之間的任何值。

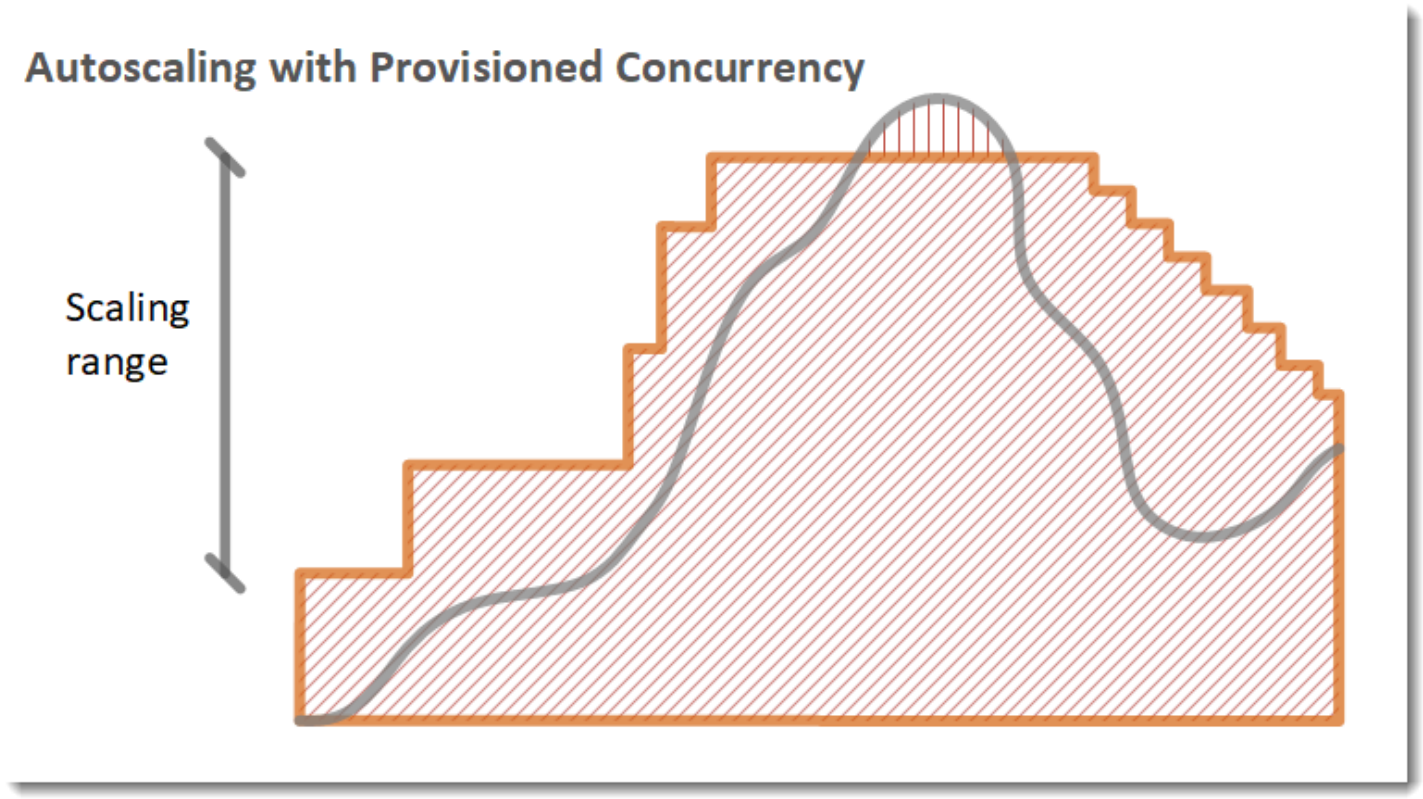
```
aws application-autoscaling put-scaling-policy \
    --service-namespace lambda \
    --scalable-dimension lambda:function:ProvisionedConcurrency \
    --resource-id function:my-function:BLUE \
    --policy-name my-policy \
    --policy-type TargetTrackingScaling \
    --target-tracking-scaling-policy-configuration '{ "TargetValue":
 0.7, "PredefinedMetricSpecification": { "PredefinedMetricType":
 "LambdaProvisionedConcurrencyUtilization" }}'
```

您應該會看到輸出，如下所示：

```
{
    "PolicyARN": "arn:aws:autoscaling:us-
east-2:123456789012:scalingPolicy:12266dbb-1524-xmpl-a64e-9a0a34b996fa:resource/lambda/
function:my-function:BLUE:policyName/my-policy",
    "Alarms": [
        {
            "AlarmName": "TargetTracking-function:my-function:BLUE-AlarmHigh-aed0e274-
xmpl-40fe-8cba-2e78f000c0a7",
            "AlarmARN": "arn:aws:cloudwatch:us-
east-2:123456789012:alarm:TargetTracking-function:my-function:BLUE-AlarmHigh-aed0e274-
xmpl-40fe-8cba-2e78f000c0a7"
        },
        {
            "AlarmName": "TargetTracking-function:my-function:BLUE-AlarmLow-7e1a928e-
xmpl-4d2b-8c01-782321bc6f66",
            "AlarmARN": "arn:aws:cloudwatch:us-
east-2:123456789012:alarm:TargetTracking-function:my-function:BLUE-AlarmLow-7e1a928e-
xmpl-4d2b-8c01-782321bc6f66"
        }
    ]
}
```

「Application Auto Scaling CloudWatch」會在中建立兩個 第一個警示會在佈建並行的使用率不斷超過 70% 時觸發。發生這種情況時，Application Auto Scaling 會配置更多佈建並行來減少使用率。第二個警示會在使用率不斷低於 63% (70% 目標的 90%) 時觸發。發生這種情況時，Application Auto Scaling 會減少別名的佈建並行。

在下列範例中，函數會根據使用率在佈建並行的最小和最大數量之間進行擴展。



圖例

- 函數執行個體
- 開啟請求
- 佈建並行
- 標準並行

當開啟的請求數量增加時，Application Auto Scaling 會大幅增加佈建並行，直到達到設定的最大值為止。在此之後，如果您尚未達到帳戶並行限制，則函數可以繼續按標準、未預留並行的方式擴展。當利用率下降且保持偏低時，Application Auto Scaling 會定期小幅減少佈建並行。

依預設，兩個 Application Auto Scaling 警示都會使用平均統計資料。經歷快速暴增流量的函數可能不會觸發這些警示。例如，假設您的 Lambda 函數執行速度很快 (即 20-100 毫秒)，而您的流量會快速暴增。在此情況下，請求數量將在暴增期間超過配置的佈建並行。不過，Application Auto Scaling 需要

暴增負載維持至少 3 分鐘，才能佈建其他環境。此外，這兩個 CloudWatch 警報都需要 3 個達到目標平均值的資料點，才能啟動 auto 擴展政策。如果您的函數經歷快速爆發的流量，則使用最大統計資料而非平均統計資料可以更有效地擴展佈建並行，以將冷啟動降至最低。

如需目標追蹤擴展政策的詳細資訊，請參閱 Application Auto Scaling 的目標追蹤擴展政策。

Lambda 擴展行為

函數收到更多請求時，Lambda 會自動提高執行環境的數量來處理這些請求，直到您的帳戶達到並行配額為止。但是，為了防止因應突然爆發的流量而過度擴展，Lambda 限制了函數擴展的速度。這種並行擴展率是您帳戶中的函數可以根據要求增加而擴展的最大速率。(也就是說，Lambda 建立新執行環境的速度可以有多快。) 並行調整比率與帳戶層級並行限制不同，也就是函數可用的並行總金額。

並行擴展率

在每個 AWS 區域 函數和每個函數中，您的並行擴展速率為每 10 秒 1,000 個執行環境執行個體。換句話說，Lambda 每隔 10 秒就可以為每個函數配置最多 1,000 個額外執行環境的執行個體。

通常情況下，您不需要擔心此限制。對於大多數使用案例，Lambda 的擴展速率已足夠。

重要的是，並行縮放速率是函數層級限制。這意味著帳戶中的每個函數都可以獨立於其他函數進行擴展。

> **Note**
>
> 實際上，Lambda 會進行最佳嘗試在一段時間內持續重新填滿並行擴展速率，而不是每 10 秒重新填滿 1,000 個單位。

Lambda 不會累積並行擴展比率的未使用部分。這意味著在任何時刻，您的擴展速率最大始終為 1,000 個並行單位。例如，如果您沒有在 10 秒的間隔內使用任何可用的 1,000 個並行單位，則不會在接下來的 10 秒間隔內額外累積 1,000 個單位。在接下來的 10 秒間隔內，您的並行擴展速率仍然是 1,000。

只要您的函數持續接收到越來越多的請求，Lambda 就會以您可用的最快速率進行擴展，最高可達您帳戶的並行限制。您可以透過設定保留並行來限制個別函數可以使用的並行數量。當請求傳入的速度超過您函數可擴展的速度，或當您的函數達到最大並行時，額外請求會因為限流錯誤而請求失敗 (狀態碼為 429)。

監控並行

Lambda 會發出 Amazon CloudWatch 指標，以協助您監控函數的並行性。本主題將說明這些指標及其解譯方式。

章節

一般並行指標

使用下列指標來監控 Lambda 函數的並行。每個指標的精細程度為 1 分鐘。

- `ConcurrentExecutions` – 在指定時間點的作用中並行調用數。Lambda 會針對所有函數、版本和別名發出此指標。對於 Lambda 主控台中的任何函數，Lambda 會在監控索引標籤的指標下以原生方式顯示 `ConcurrentExecutions` 的圖表。使用 MAX 檢視此指標。
- `UnreservedConcurrentExecutions` – 使用未預留並行的作用中並行調用數。Lambda 會針對區域中的所有函數發出此指標。使用 MAX 檢視此指標。
- `ClaimedAccountConcurrency` – 無法用於隨需調用的並行數量。`ClaimedAccountConcurrency` 等於 `UnreservedConcurrentExecutions` 加上配置並行的數量 (亦即預留並行總數加上佈建並行總數)。如果 `ClaimedAccountConcurrency` 超過帳戶並行限制，您可以請求提高帳戶並行上限。使用 MAX 檢視此指標。如需詳細資訊，請參閱 使用 `ClaimedAccountConcurrency` 指標。

佈建並行指標

使用下列指標來監控使用佈建並行的 Lambda 函數。每個指標的精細程度為 1 分鐘。

- `ProvisionedConcurrentExecutions` – 目前使用佈建並行處理調用的執行環境執行個體數。Lambda 會針對已設定佈建並行的每個函數版本和別名發出此指標。使用 MAX 檢視此指標。

`ProvisionedConcurrentExecutions` 與您配置的佈建並行總數不同。例如，假設您將 100 個佈建並行單位配置給函數版本。在任何指定的分鐘內，如果在這 100 個執行環境中最多 50 個同時處理調用，則 MAX(`ProvisionedConcurrentExecutions`) 的值為 50。

- ProvisionedConcurrentInvocations - Lambda 使用佈建並行調用函數程式碼的次數。Lambda 會針對已設定佈建並行的每個函數版本和別名發出此指標。使用 SUM 檢視此指標。

ProvisionedConcurrentInvocations 與 ProvisionedConcurrentExecutions 不同，ProvisionedConcurrentInvocations 計算調用總數，ProvisionedConcurrentExecutions 則計算作用中環境數。若要了解這種區別，請考慮以下案例：

在此範例中，假設您每分鐘收到 1 次調用，而且每次調用都需要 2 分鐘才能完成。每個橘色水平長條代表一個請求。假設您將 10 個佈建並行單位配置給此函數，這樣每個請求都會在佈建並行上執行。

在分鐘 0 和 1 之間，收到 Request 1。在分鐘 1 時，MAX(ProvisionedConcurrentExecutions) 的值為 1，因為在過去一分鐘內最多有 1 個執行環境處於作用中狀態。SUM(ProvisionedConcurrentInvocations) 的值也是 1，因為在過去一分鐘內收到 1 個新請求。

在分鐘 1 和 2 之間，收到 Request 2，且 Request 1 繼續執行。在分鐘 2 時，MAX(ProvisionedConcurrentExecutions) 的值為 2，因為在過去一分鐘內最多有 2 個執行環境處於作用中狀態。不過，SUM(ProvisionedConcurrentInvocations) 的值為 1，因為在過去一分鐘內只收到 1 個新請求。此指標行為會一直持續到範例結束為止。

- ProvisionedConcurrencySpilloverInvocations – 當所有佈建並行都在使用中時，Lambda 使用標準 (預留或未預留) 並行調用函數的次數。Lambda 會針對已設定佈建並行的每個函數版本和別名發出此指標。使用 SUM 檢視此指標。ProvisionedConcurrentInvocations + ProvisionedConcurrencySpilloverInvocation 值應等於函數調用總數 (即 Invocations 指標)。

ProvisionedConcurrencyUtilization – 使用中佈建並行百分比 (即 ProvisionedConcurrentExecutions 的值除以配置的佈建並行總數)。Lambda 會針對已設定佈建並行的每個函數版本和別名發出此指標。使用 MAX 檢視此指標。

例如，假設您將 100 個佈建並行單位佈建給函數版本。在任何指定的分鐘內，如果在這 100 個執行環境中最多 60 個同時處理調用，則 MAX(ProvisionedConcurrentExecutions) 的值為 60，MAX(ProvisionedConcurrentUtilization) 的值為 0.6。

ProvisionedConcurrencySpilloverInvocations 的值偏高可能表示您需要為函數配置額外的佈建並行。或者，您可以設定 Application Auto Scaling，以根據預先定義的閾值來處理佈建並行的自動擴展。

相反地，ProvisionedConcurrencyUtilization 的值持續偏低可能表示您為函數配置過多佈建並行。

使用 ClaimedAccountConcurrency 指標

Lambda 會使用 ClaimedAccountConcurrency 指標來判斷您的帳戶可用於隨需調用的並行數量。Lambda 會使用以下公式計算 ClaimedAccountConcurrency：

```
ClaimedAccountConcurrency = UnreservedConcurrentExecutions + (allocated concurrency)
```

UnreservedConcurrentExecutions 是使用未預留並行的作用中並行調用數目。配置的並行是下列兩個部分的總和 (以「預留並行」取代 RC，以「佈建並行」取代 PC)：

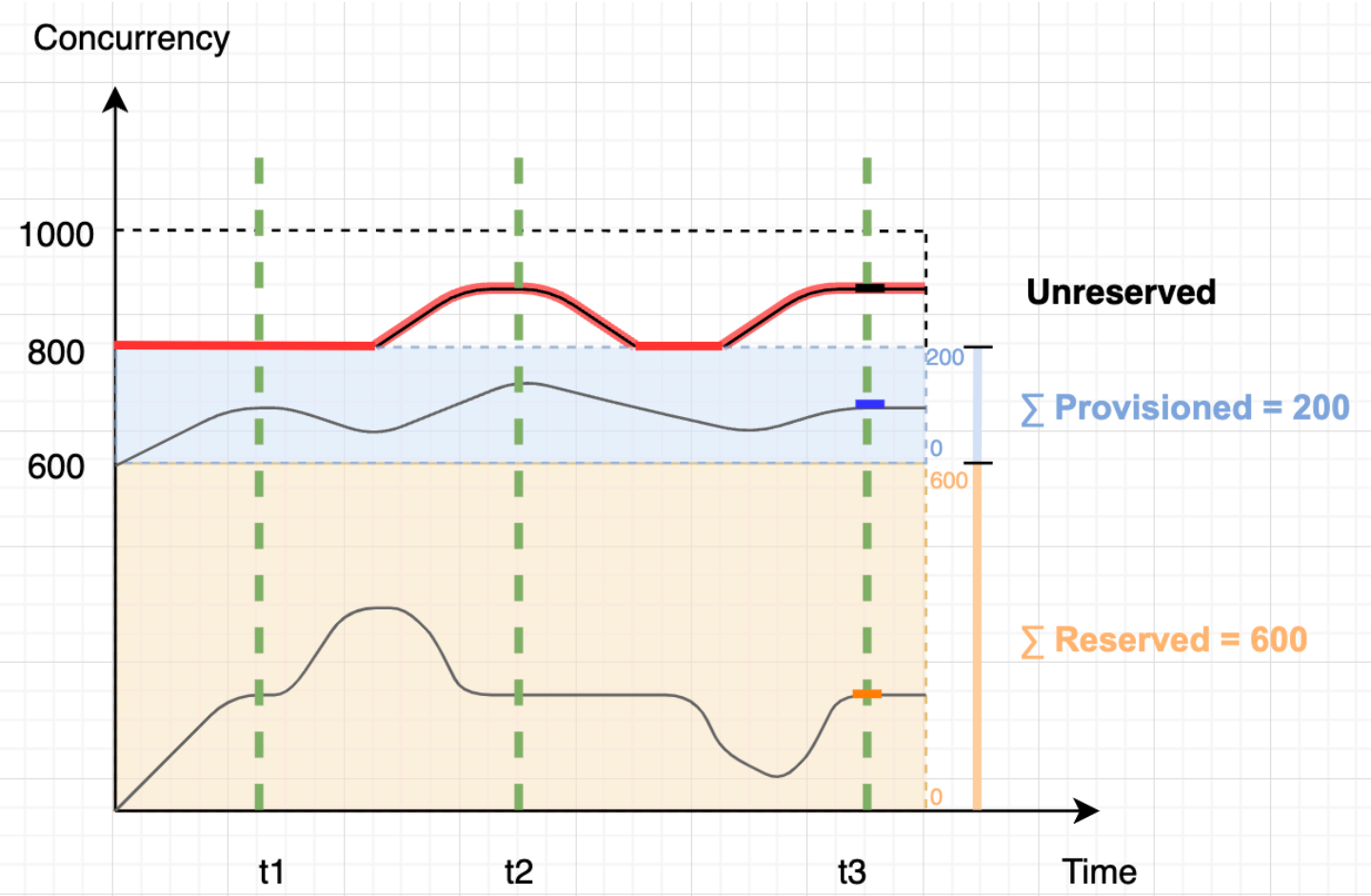
- 區域中所有函數的 RC 總數。
- 區域中使用 PC 的所有函數的 PC 總數，使用 RC 的函數除外。

> **Note**
>
> 您不能為一個函數配置多於 RC 的 PC。因此，一個函數的 RC 總是大於或等於它的 PC。對於同時具有 PC 和 RC 的函數，Lambda 在計算這些函數的配置並行比重時僅考慮 RC，即兩者當中的較大值。

Lambda 會使用 ClaimedAccountConcurrency 指標來判斷可用於隨需調用的並行數量，而不使用 ConcurrentExecutions。雖然 ConcurrentExecutions 指標對於追蹤作用中並行調用的數量很

有用，但並不總是反映您真正的並行可用性。這是因為 Lambda 也會考慮預留並行和佈建並行來確定可用性。

為了說明 ClaimedAccountConcurrency，請考慮一種情況，即您可以跨函數設定大量預留並行和佈建並行，但其中有很多未被使用。在下列範例中，假設您的帳戶並行限制為 1,000，而您的帳戶中有兩種主要的函數：function-orange 和 function-blue。您為 function-orange 配置 600 個單位的預留並行。您為 function-blue 配置 200 個單位的佈建並行。假設隨著時間推移，您要部署額外的函數並觀測以下流量模式：



在上一張圖表中，黑線表示隨時間推移的實際並行使用，紅線表示隨時間推移的 ClaimedAccountConcurrency 值。在這種情況下，儘管函數的實際並行利用率較低，但 ClaimedAccountConcurrency 始終至少為 800。這是因為您為 function-orange 和 function-blue 配置了 800 個單位總數的並行。從 Lambda 的角度來看，您已經「聲明」這些並行供使用，因此對於其他函數，您實際上只剩下 200 個單位的並行。

針對這個情況，在 ClaimedAccountConcurrency 公式中配置的並行是 800。然後，我們可以在圖表的不同點衍生 ClaimedAccountConcurrency 值。

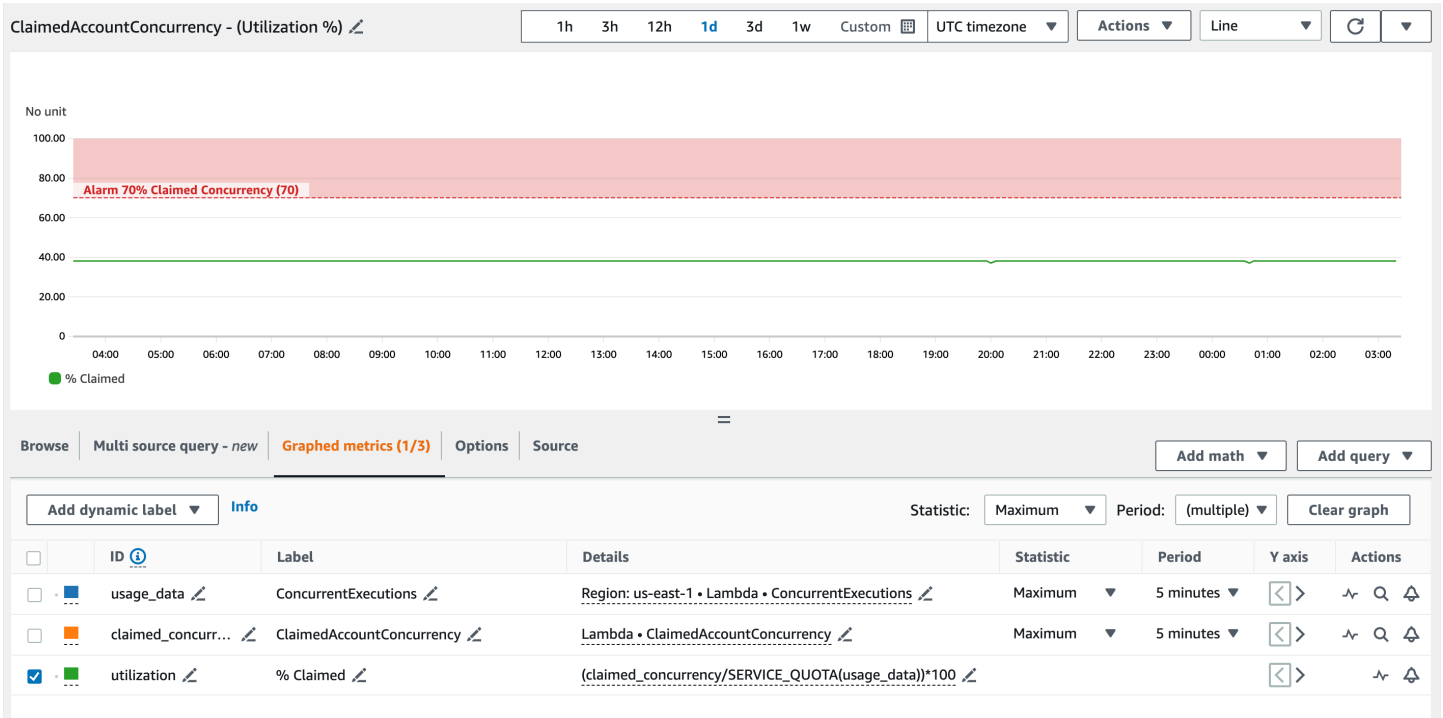
- 在 t1，ClaimedAccountConcurrency 為 800 (800 + 0 UnreservedConcurrentExecutions)。
- 在 t2，ClaimedAccountConcurrency 為 900 (800 + 100 UnreservedConcurrentExecutions)。
- 在 t3，ClaimedAccountConcurrency 還是 900 (800 + 100 UnreservedConcurrentExecutions)。

設定**ClaimedAccountConcurrency**量度 CloudWatch

Lambda 會在中發出ClaimedAccountConcurrency量度 CloudWatch。使用此指標和 SERVICE_QUOTA(ConcurrentExecutions) 的值取得有關您的帳戶的並行利用率百分比，如下方公式所示：

```
Utilization = (ClaimedAccountConcurrency/SERVICE_QUOTA(ConcurrentExecutions)) * 100%
```

下列螢幕擷取畫面說明如何在中繪製此公式的圖形 CloudWatch。綠色 claim_utilization 線代表此帳戶中的並行利用率，約為 40%：



上一個螢幕擷取畫面還包含在並行使用率超過 70% 時進入ALARM狀態的 CloudWatch 警示。您可以使用 ClaimedAccountConcurrency 指標和類似的警示來主動確定您何時可能需要請求更高的帳戶並行上限。

設定程式碼簽章 AWS Lambda

的程式碼簽章 AWS Lambda 有助於確保只有受信任的程式碼才能在 Lambda 函數中執行。啟用函數的程式碼簽署時，Lambda 會檢查每個程式碼部署，並確認程式碼套件是由受信任的來源簽署。

> **Note**
>
> 定義為容器映像的函數不支援程式碼簽署。

若要驗證程式碼完整性，請使用 AWS Signer 為函數和 Layer 建立數位簽署的程式碼套件。若使用者嘗試部署程式碼套件時，Lambda 會對程式碼套件執行驗證檢查，然後再接受部署。因為程式碼簽署驗證檢查會在部署時執行，因此不會影響函數執行的效能。

您也可以使用 AWS 簽署者建立簽署設定檔。您可以使用簽署描述檔來建立簽署的程式碼套件。使用 AWS Identity and Access Management (IAM) 控制誰可以簽署程式碼套件並建立簽署設定檔。如需詳細資訊，請參閱《AWS Signer 開發人員指南》中的驗證與存取控制。

若要啟用函數的程式碼簽署，您可以建立程式碼簽署組態並將其附加至函數。程式碼簽署組態會定義允許簽署描述檔的清單，以及在任何驗證檢查失敗時要採取的政策動作。

Lambda 層會遵循與函數程式碼套件相同的已簽署程式碼套件格式。將層新增至已啟用程式碼簽署的函數時，Lambda 會檢查該層是否由允許的簽署描述檔簽署。若您啟用函數的程式碼簽署，新增至函數的所有 Layer 也必須由其中一個允許的簽署描述檔來簽署。

使用 IAM 來控制誰可以建立程式碼簽署組態。通常，僅允許特定的管理使用者擁有此功能。此外，您可以設定 IAM 政策，來強制只有開發人員可建立已啟用程式碼簽署的函數。

您可以設定程式碼簽署以記錄 AWS CloudTrail的變更。對功能的成功部署和封鎖部署會記錄到，並提 CloudTrail 供有關簽章和驗證檢查的資訊。

您可以使用 Lambda 主控台 () 和 AWS Command Line Interface (AWS CLI) 設定函數的程式碼簽章。AWS CloudFormation AWS Serverless Application Model AWS SAM

使用簽署 AWS 者或程式碼簽章無須額外付費。 AWS Lambda

章節

- 建立程式碼簽署組態
- 更新程式碼簽署組態
- 刪除程式碼簽署組態
- 啟用函數的程式碼簽署
- 設定 IAM 政策
- 使用 Lambda API 來設定程式碼簽署

簽署驗證

在您將已簽署的程式碼套件部署至函數時，Lambda 會執行下列驗證檢查：

1. 完整性 - 驗證程式碼套件自簽署後尚未修改。Lambda 將套件的雜湊與簽章中的雜湊作比較。
2. 過期 - 驗證程式碼套件的簽章尚未過期。
3. 不相符 - 驗證程式碼套件是否使用其中一個允許的 Lambda 函數的簽署描述檔來進行簽署。如果簽署不存在，也會發生不相符情況。
4. 撤銷 - 驗證程式碼套件的簽章尚未被叫用。

如果任何驗證檢查失敗，程式碼簽署組態中定義的簽章驗證政策將確定 Lambda 會採取下列哪些動作：

- 警告 — Lambda 允許部署程式碼套件，但會發出警告。Lambda 會發出新的 Amazon CloudWatch 指標，並將警告儲存在 CloudTrail 日誌中。
- 強制 - Lambda 發出警告 (與「警告」動作相同) 並阻止程式碼套件的部署。

您可以設定到期、不相符和撤銷驗證檢查的政策。請注意，您無法設定完整性檢查的政策。如果完整性檢查失敗，Lambda 會阻止部署。

組態先決條件

在您可以設定 Lambda 函數的程式碼簽章之前，請先使用 AWS 簽署者執行下列動作：

- 建立一個或多個簽署描述檔。
- 使用簽署描述檔為您的函數建立簽署的程式碼套件。

如需詳細資訊，請參閱《AWS Signer 開發人員指南 》中的建立簽署描述檔 (主控台)。

建立程式碼簽署組態

程式碼簽署組態會定義允許的簽署描述檔和簽署驗證政策。

建立程式碼簽署組態 (主控台)

1. 開啟 Lambda 主控台中的 Code signing configurations (程式碼簽署組態) 頁面。
2. 選擇建立組態。
3. 針對 Description (描述)，輸入組態的描述性名稱。
4. 在 Signing profiles (簽署描述檔) 下，可將多達 20 個簽署描述檔新增至組態。
   a. 針對 Signing profile version ARN (簽署描述檔版本 ARN)，選擇描述檔版本的 Amazon Resource Name (ARN)，或輸入 ARN。
   b. 若要新增額外的簽署描述檔，選擇 Add signing profiles (新增簽署描述檔)。
5. 在 Signature validation policy (簽署驗證政策) 下，選擇 Warn (警告) 或 Enforce (強制)。
6. 選擇建立組態。

更新程式碼簽署組態

若更新程式碼簽署組態，變更會影響已附加程式碼簽署組態的函數的未來部署。

更新程式碼簽署組態 (主控台)

1. 開啟 Lambda 主控台中的 Code signing configurations (程式碼簽署組態) 頁面。
2. 選取要更新的程式碼簽署組態，然後選擇 Edit (編輯)。
3. 針對 Description (描述)，輸入組態的描述性名稱。
4. 在 Signing profiles (簽署描述檔) 下，可將多達 20 個簽署描述檔新增至組態。
   a. 針對 Signing profile version ARN (簽署描述檔版本 ARN)，選擇描述檔版本的 Amazon Resource Name (ARN)，或輸入 ARN。
   b. 若要新增額外的簽署描述檔，選擇 Add signing profiles (新增簽署描述檔)。
5. 在 Signature validation policy (簽署驗證政策) 下，選擇 Warn (警告) 或 Enforce (強制)。
6. 選擇儲存變更。

刪除程式碼簽署組態

只有在沒有任何函數使用的情況下，才能刪除程式碼簽署組態。

刪除程式碼簽署組態 (主控台)

1. 開啟 Lambda 主控台中的 Code signing configurations (程式碼簽署組態) 頁面。
2. 選取要刪除的程式碼簽署組態，然後選擇 Delete (刪除)。
3. 若要確認，再次選擇 Delete (刪除)。

啟用函數的程式碼簽署

若要啟用函數的程式碼簽署，請將程式碼簽署組態與函數建立關聯。

將程式碼簽署組態與函數 (主控台) 關聯

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇您要啟用程式碼簽署的函數。
3. 開啟 Configuration (組態) 索引標籤。
4. 向下捲動並選擇 [程式碼簽章]。
5. 選擇編輯。
6. 在 Edit code signing (編輯程式碼簽署) 中，選擇此函數的程式碼簽署組態。
7. 選擇儲存。

設定 IAM 政策

若要授予使用者存取程式碼簽署 API 操作的許可，請將一個或多個政策陳述式附加至使用者政策。如需使用者政策的詳細資訊，請參閱在 Lambda 中使用以身分識別為基礎的 IAM 政策。

下列範例政策陳述式會授予建立、更新及擷取程式碼簽署組態的許可。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
```

```
            "lambda:CreateCodeSigningConfig",
            "lambda:UpdateCodeSigningConfig",
            "lambda:GetCodeSigningConfig"
          ],
        "Resource": "*"
      }
    ]
}
```

管理員可以使用 CodeSigningConfigArn 條件金鑰，來指定開發人員必須用於建立或更新函數的程式碼簽署組態。

下列範例政策陳述式會授予建立函數的許可。政策聲明包括 lambda:CodeSigningConfigArn 條件以指定允許的程式碼簽署組態。Lambda 會阻止任何 CreateFunction API 請求，前提是如果其 CodeSigningConfigArn 參數遺失或不符合條件中的值。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowReferencingCodeSigningConfig",
      "Effect": "Allow",
      "Action": [
          "lambda:CreateFunction",
        ],
      "Resource": "*",
      "Condition": {
          "StringEquals": {
              "lambda:CodeSigningConfigArn":
                  “arn:aws:lambda:us-west-2:123456789012:code-signing-
config:csc-0d4518bd353a0a7c6”
          }
      }
    }
  ]
}
```

使用 Lambda API 來設定程式碼簽署

若要使用 AWS CLI 或 AWS SDK 管理程式碼簽章設定，請使用下列 API 作業：

- ListCodeSigningConfigs

- CreateCodeSigningConfig
- GetCodeSigningConfig
- UpdateCodeSigningConfig
- DeleteCodeSigningConfig

若要管理函數的程式碼簽署組態，請使用下列 API 操作：

- CreateFunction
- GetFunctionCodeSigningConfig
- PutFunctionCodeSigningConfig
- DeleteFunctionCodeSigningConfig
- ListFunctionsByCodeSigningConfig

在 Lambda 函數上使用標籤

您可以標記 AWS Lambda 函數來啟動屬性型存取控制 (ABAC)，並依擁有者、專案或部門進行整理。標籤是各種 AWS 服務之間統一支援的自由格式索引鍵/值組，可用於 ABAC、篩選資源以及新增詳細資訊至帳單報告。

標籤適用於函數層級，而不適用於版本或別名。Lambda 在您發佈版本時建立快照的版本特定組態中並不包含標籤。

章節

使用標籤所需的許可

為使用函數的人員授予對 AWS Identity and Access Management IAM 身分 (使用者、群組或角色) 適當許可：

- lambda：ListTags — 當函數具有標籤時，將此權限授予需要調用`GetFunction`或對其進行調用`ListTags`的任何人。
- lambda: TagResource — 將此權限授予需要致電`CreateFunction`或的任何人`TagResource`。

如需詳細資訊，請參閱 在 Lambda 中使用以身分識別為基礎的 IAM 政策。

搭配使用標籤與 Lambda 主控台

您可以使用 Lambda 主控台建立具有標籤的函數、將標籤新增至現有函數，以及依您新增的標籤篩選函數。

在建立函數時新增標籤

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇 建立函數。
3. 選擇 Author from scratch (從頭開始撰寫) 或 Container image (容器映像)。

4. 在 基本資訊 下，請執行下列動作：

   a. 針對 函數名稱，輸入函數名稱。函數名稱的長度限制為 64 個字元。

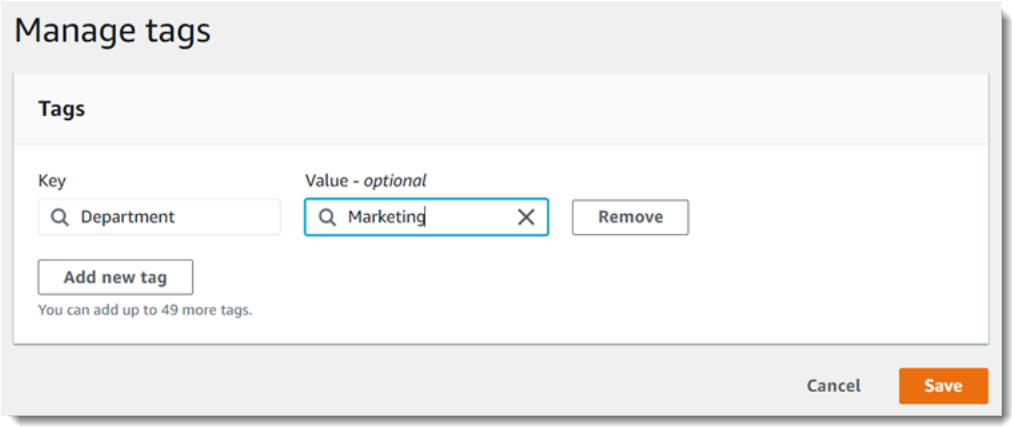
   b. 對於執行時間，選擇函數要使用的語言版本。

   c. (選用) 在 Architecture (架構) 欄位，選擇要用於函數的指令集架構。預設架構值為 x86_64。為您的函數建置部署套件時，請確定其與您選擇的指令集架構相容。

5. 展開 Advanced settings (進階設定)，然後選取 Enable tags (啟用標籤)。
6. 選擇 Add new tag (新增標籤)，然後輸入 Key (索引鍵) 和選用的 Value (值)。若要新增更多標籤，請重複此步驟。
7. 選擇 建立函式。

為現有函數新增標籤

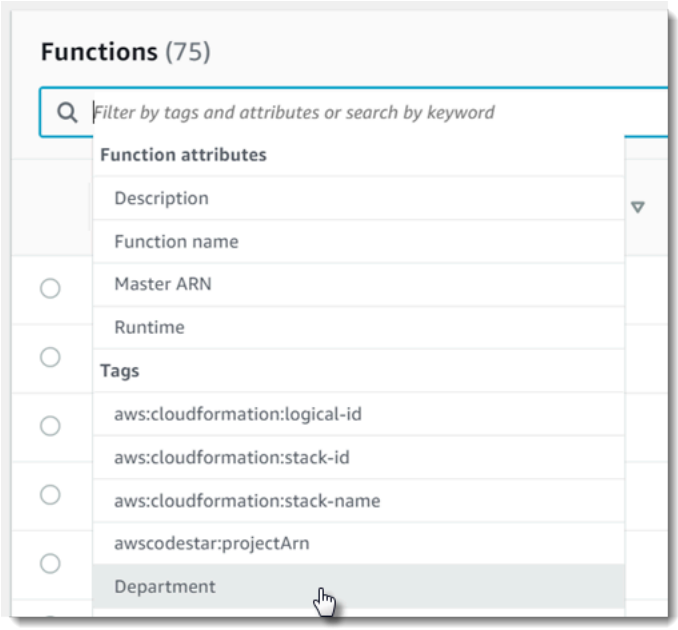
1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇函數的名稱。
3. 選擇 Configuration (組態)，然後選擇 Tags (標籤)。
4. 在 Tags (標籤) 下，選擇 Manage tags (管理標籤)。
5. 選擇 Add new tag (新增標籤)，然後輸入 Key (索引鍵) 和選用的 Value (值)。若要新增更多標籤，請重複此步驟。



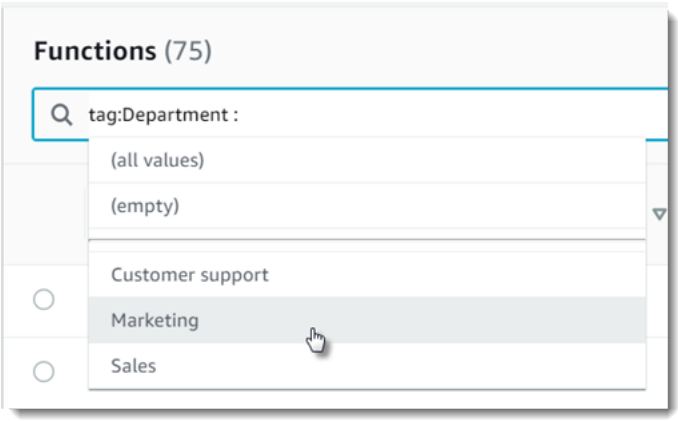
6. 選擇儲存。

使用標籤篩選函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇搜尋列以查看函數屬性和標籤索引鍵的清單。



3. 選擇標籤索引鍵以查看目前 AWS 區域內使用中的值清單。
4. 選擇一個數值以查看具有該數值的函數，或選擇 (all values) (所有值) 查看具有該索引鍵標籤的所有函數。



搜尋列也支援搜尋標籤鍵。輸入 `tag` 以僅查看標籤索引鍵清單，或輸入索引鍵名稱以在清單中尋找。

搭配使用標籤與 AWS CLI

新增和移除標籤

若要建立具有標籤的新 Lambda 函數，請使用具有 --tags 選項的 create-function 命令。

```
aws lambda create-function --function-name my-function
```

```
--handler index.js --runtime nodejs20.x \
--role arn:aws:iam::123456789012:role/lambda-role \
--tags Department=Marketing,CostCenter=1234ABCD
```

若要將標籤新增至現有函數，請使用 tag-resource 命令。

```
aws lambda tag-resource \
--resource arn:aws:lambda:us-east-2:123456789012:function:my-function \
--tags Department=Marketing,CostCenter=1234ABCD
```

若要移除標籤，請使用 untag-resource 命令。

```
aws lambda untag-resource --resource arn:aws:lambda:us-east-1:123456789012:function:my-function \
--tag-keys Department
```

檢視函數上的標籤

若您要檢視已套用至特定 Lambda 函數的標籤，可使用以下其中一個 AWS CLI 命令：

- ListTags— 若要檢視與此函數相關聯的標籤清單，請包含您的 Lambda 函數 ARN (Amazon 資源名稱)：

  ```
  aws lambda list-tags --resource arn:aws:lambda:us-east-1:123456789012:function:my-function
  ```

- GetFunction— 若要檢視與此函數相關聯的標籤清單，請包含您的 Lambda 函數名稱：

  ```
  aws lambda get-function --function-name my-function
  ```

依標籤篩選函數

您可以使用 AWS Resource Groups Tagging API GetResourcesAPI 操作按標籤過濾資源。`GetResources` 操作可接收最多 10 個篩選條件，每個篩選條件皆包含標籤索引鍵與最多 10 個標籤值。為 `GetResources` 提供一個 `ResourceType`，即可依特定資源類型進行篩選。

如需有關 AWS Resource Groups 的詳細資訊，請參閱《AWS Resource Groups 和標籤使用者指南》中的什麼是資源群組？。

標籤的需求

下列需求適用於標籤：

- 每個資源的標籤數上限：50
- 索引鍵長度上限：128 個 UTF-8 Unicode 字元
- 值長度上限：256 個 UTF-8 Unicode 字元
- 標籤鍵與值皆區分大小寫。
- 標籤名稱或值不可使用 `aws:` 字首，因為它只保留給 AWS 使用。您不可編輯或刪除具此字首的標籤名稱或值。具此字首的標籤，不算在受資源限制的標籤計數內。
- 若您規劃跨多項服務和資源使用標記結構描述，請謹記，其他服務可能設有字元數量使用限制。通常，允許使用的字元為：可用 UTF-8 表示的英文字母、空格和數字，加上以下特殊字元：+ - = . _ : / @。

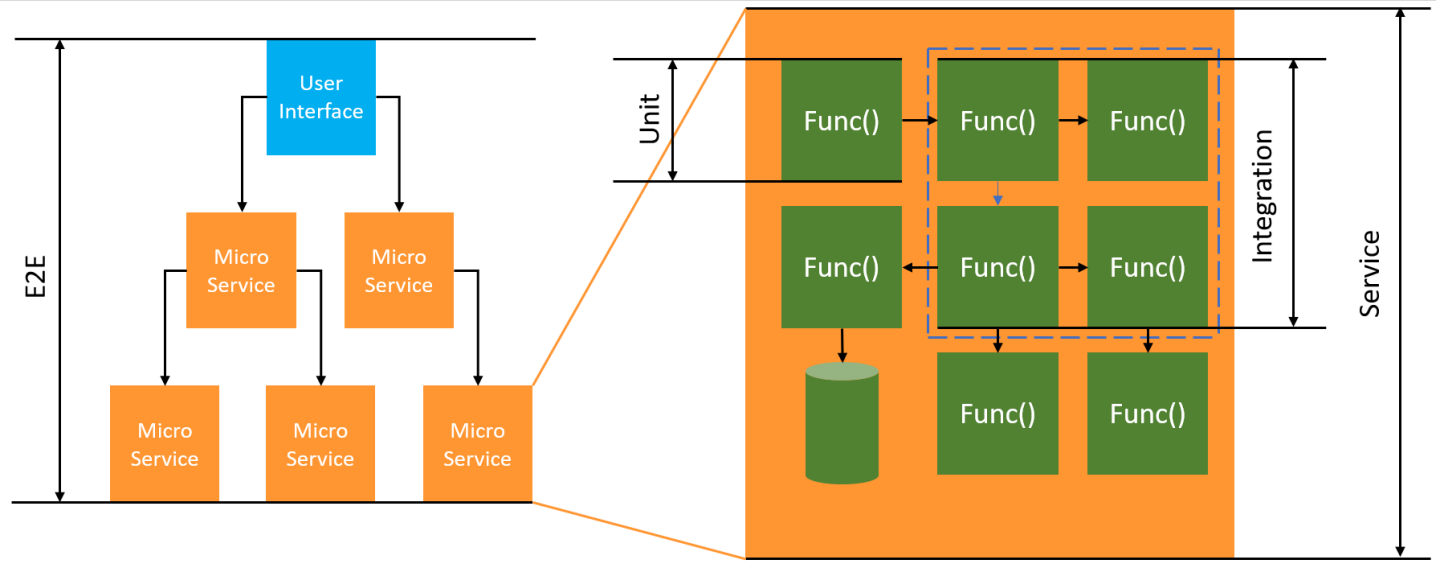
如何測試無伺服器功能和應用程式

測試無伺服器函數會使用傳統的測試類型和技術，但您也必須考慮測試整個無伺服器應用程式。以雲端為基礎的測試會為您的函數和無伺服器應用程式提供最準確的品質測量標準。

無伺服器應用程式架構包括透過 API 呼叫提供關鍵應用程式功能的受管服務。因此，您的開發週期應包括自動化測試，以便在函數和服務互動時驗證功能。

如果您未建立以雲端為基礎的測試，則可能會因本機環境與部署環境之間的差異而遇到問題。您的持續整合程序應先針對雲端佈建的一組資源進行測試，然後再將程式碼升級至下一個部署環境 (例如 QA、暫存或生產環境)。

繼續閱讀這份簡短指南，了解無伺服器應用程式的測試策略，或造訪無伺服器測試範例儲存庫，深入了解所選語言和執行期的特定實際範例。



若為無伺服器測試，您仍需要寫入單元、整合及端對端測試。

- 單元測試：針對一組隔離的程式碼區塊進行的測試。例如，驗證商業邏輯以計算指定的特定項目與目的地的運費。
- 整合測試：涉及到兩個以上元件或服務進行互動的測試 (通常在雲端環境)。例如，驗證函數是否有處理佇列中的事件。
- E nd-to-end 測試-測試，驗證整個應用程序的行為。例如，確保基礎設施的設定正確無誤，以及事件如預期在服務之間流動，以記錄客戶的訂單。

目標業務成果

測試無伺服器解決方案時可能需要更多時間來設定測試，以驗證服務之間的事件驅動互動。閱讀本指南時，請謹記以下實際的商業原因：

- 提高應用程式的品質
- 降低建構功能和修復錯誤的時間

應用程式的品質取決於測試各種情境來驗證功能。仔細思考業務情境並自動化這些針對雲端服務進行的測試，將有助於提高應用程式的品質。

在反覆進行的開發週期中發現軟體錯誤和組態問題對成本和排程的影響最小。如果在開發過程中仍未發現問題，則在生產過程中尋找和修復問題時，將需要耗費更多人力。

經過妥善規劃的無伺服器測試策略可驗證 Lambda 函數和應用程式是否在雲端環境中如預期般運行，藉此提高軟體品質並縮短反覆運算的時間。

測試項目

建議您採用能測試受管服務 行為、雲端組態、安全政策以及程式碼整合的測試策略，以提升軟體品質。行為測試 (又稱為黑箱測試) 會在不了解所有內部情況的狀況下驗證系統是否如預期般正常運作。

- 執行單元測試以檢查 Lambda 函數內的商業邏輯。
- 確認整合服務實際上是否有調用，且輸入參數是否正確無誤。
- 檢查事件是否通過工作流程 end-to-end 中的所有預期服務。

在傳統的伺服器型架構中，團隊通常會定義測試的範圍，且只納入在應用程式伺服器上執行的程式碼。其他元件、服務或相依項目通常會被認定屬於外部且超出測試範圍。

無伺服器應用程式通常由小型工作單位組成，例如從資料庫擷取產品、處理佇列項目或是調整儲存空間映像大小的 Lambda 函數。每個元件都會在各自的環境中執行。團隊可能會在單一應用程式中負起這類眾多小型單位的責任。

部分應用程式功能可完全委派給受管服務 (例如 Amazon S3)，也可以在不使用任何內部開發程式碼的情況下建立。您不需要測試這類受管服務，不過您確實需要測試與這些服務的整合。

如何測試無伺服器

您可能對測試在本機部署的應用程式的方式不陌生：您撰寫的測試會針對完全在桌面作業系統或容器內執行的程式碼進行測試。例如，您可以透過請求調用本機 Web 服務元件，然後對回應做出聲明。

無伺服器解決方案是根據您的函數程式碼和雲端受管服務 (例如佇列、資料庫、事件匯流排和簡訊傳送系統) 建置而成。這些元件皆透過 事件驅動的架構 連接，其中訊息 (稱為 事件) 會從一項資源流向另一項資源。這些互動可以是同步的，例如當 Web 服務立即傳回結果時，或是稍後完成的非同步動作，例如將項目放置在佇列中或啟動工作流程步驟。您的測試策略必須納入這兩種情境，並測試服務之間的互動。對於非同步交互，您可能需要檢測下游組件中可能無法立即觀察到的副作用。

複寫整個雲端環境 (包括佇列、資料庫資料表、事件匯流排、安全性政策等) 的作法並不實際。由於本機環境與雲端中部署的環境之間存在差異，因此您必定會遇到問題。環境之間的變化將增加重現和修復錯誤所需的時間。

在無伺服器應用程式中，架構元件通常完全位於雲端之中，因此您必須對雲端中的程式碼和服務進行測試，才能夠開發功能和修正錯誤。

測試技術

在實際情況下，您的測試策略可能會包括各種技術，以提升解決方案的品質。您將會採用快速互動測試來偵錯主控台中的函數、使用自動化單元測試來檢查隔離的商業邏輯、透過模擬來驗證對外部服務的呼叫，以及偶爾對模擬服務的模擬器進行測試。

- 在雲端測試：您可以部署基礎設施和程式碼，透過實際的服務、安全政策、組態和基礎設施的具體參數進行測試。以雲端為基礎的測試可為您的程式碼提供最精準的品質測量方法。

  您可以透過在主控台中偵錯函數，進而在雲端中迅速進行測試。您可以從範例測試事件資源庫中進行選擇，也可以建立自訂事件來單獨測試函數。您也可以透過主控台與團隊分享測試事件。

  若要在開發和建置生命週期中 自動化 測試，則必須在主控台之外進行測試。如需自動化策略和資源，請參閱本指南中特定語言的測試章節。
- 透過模擬 (又稱為 假物件) 進行測試：模擬為程式碼中的物件，可模擬和替代外部服務。模擬物件提供預先定義的行為來驗證服務呼叫和參數。假物件 是一種模擬實作，它會透過捷徑來簡化或提升效能。例如，假物件的資料存取物件可能會從記憶體內的資料儲存傳回資料。模擬物件可以模仿和簡化複雜的相依項目，但也可能產生更多的模擬來代替巢狀的相依項目。
- 使用模擬器進行測試：您可以設定應用程式 (有時透過第三方) 來模擬本機環境中的雲端服務。速度是模擬器的優勢，但設定和與生產服務的一致性是它的劣勢。請謹慎使用模擬器。

在雲端進行測試

雲端測試對於測試的所有階段都很有價值，包括單元測試、整合測試和 end-to-end 測試。當您針對同時與雲端服務互動的雲端程式碼進行測試時，便可以獲得最精準的程式碼品質測量方法。

您可以透過在 AWS Management Console 加入測試事件輕鬆在雲端中執行 Lambda 函數。一個測試事件是函數的 JSON 輸入。如果您的函數不需要輸入，該事件可以是空白的 JSON 文件 (`{}`)。主控台提供各種服務整合的範例事件。在主控台中建立事件後，您也可以與團隊分享事件，讓測試變得更容易，結果更一致。

了解如何 在主控台中偵錯範例函數。

> **Note**
>
> 雖然在主控台中執行函數是一種快速偵錯的作法，但 自動化 測試週期是提高應用程式品質和開發速度的關鍵。

您可以在 無伺服器測試範例儲存庫 中取得測試自動化範例。下列命令列會執行自動化的 Python 整合測試範例：

```
python -m pytest -s tests/integration -v
```

雖然測試在本機執行，但它會與雲端資源互動。這些資源已使用 AWS Serverless Application Model 和 AWS SAM 命令列工具進行部署。測試程式碼會先擷取已部署的堆疊輸出，其中包括 API 端點、函數 ARN 和安全角色。接下來，測試會將請求傳送到 API 端點，該端點會以 Amazon S3 儲存貯體清單進行回應。此測試的執行對象完全是雲端資源，以驗證這些資源是否已完成部署、獲得保護並如預期般正常運作。

```
========================== test session starts ==========================
      platform darwin -- Python 3.10.10, pytest-7.3.1, pluggy-1.0.0
      -- /Users/t/code/aws/serverless-test-samples/python-test-samples/apigw-lambda/
venv/bin/python
      cachedir: .pytest_cache
      rootdir: /Users/t/code/aws/serverless-test-samples/python-test-samples/apigw-
lambda
      plugins: mock-3.10.0
      collected 1 item
```

```
      tests/integration/test_api_gateway.py::TestApiGateway::test_api_gateway

      --> Stack outputs:

        HelloWorldApi
        = https://p7teqs3162.execute-api.us-west-2.amazonaws.com/Prod/hello/
        > API Gateway endpoint URL for Prod stage for Hello World function

        PythonTestDemo
        = arn:aws:lambda:us-west-2:1234567890:function:testing-apigw-lambda-
PythonTestDemo-iSij8evaTdxl
        > Hello World Lambda Function ARN

        PythonTestDemoIamRole
        = arn:aws:iam::1234567890:role/testing-apigw-lambda-PythonTestDemoRole-
IZELQQ9MG4HQ
        > Implicit IAM Role created for Hello World function

      --> Found API endpoint for "testing-apigw-lambda" stack...
      --> https://p7teqs3162.execute-api.us-west-2.amazonaws.com/Prod/hello/
      API Gateway response:
      amplify-dev-123456789-deployment|myapp-prod-p-loggingbucket-123456|s3-java-
bucket-123456789
      PASSED

      ========================= 1 passed in 1.53s =========================
```

若是開發雲端原生應用程式，則在雲端進行測試可獲得以下優勢：

- 您可以測試 每項 可用的服務。
- 您一律會使用最新的服務 API 和傳回值。
- 雲端測試環境與您的生產環境非常類似。
- 測試可以涵蓋安全策略、服務配額、組態和基礎設施的具體參數。
- 每位開發人員都可以在雲端中迅速建立一或多個測試環境。
- 雲端測試可提高程式碼在生產環境中順利執行的把握度。

在雲中進行測試確實有一些缺點。部署到雲端環境的時間通常比部署到本機桌面環境的時間更長，是在雲端進行測試時最明顯的缺點。

幸運的是，AWS 無伺服器應用程式模型 (S AWS AM) 加速、AWS Cloud Development Kit (AWS CDK) 監視模式和 SST (第三方) 等工具可減少與雲端部署反覆運算相關的延遲。這些工具可以監控您的基礎設施和程式碼，並自動將增量更新部署到您的雲端環境。

> **Note**
>
> 請參閱無伺服器開發人員指南，了解如何建立基礎結構即程式碼，以進一步了解 AWS Serverless Application Model AWS CloudFormation、和 AWS Cloud Development Kit (AWS CDK)。

不同於本機測試，在雲端進行測試需使用額外的資源，而這可能會產生服務費用。建立隔離的測試環境可能會增加 DevOps 團隊的負擔，尤其是在對帳戶和基礎架構有嚴格控制的組織中。即便如此，在面對複雜的基礎設施情境時，開發人員設定和維護複雜本機環境的時間成本，可能會與使用基礎設施即程式碼自動化工具建立的一次性測試環境相似 (或更高)。

即使有這些考量，在雲端進行測試仍然是保證無伺服器解決方案品質的 最佳作法。

透過模擬物件進行測試

使用模擬物件進行測試是一種技術，您可以在程式碼中建立替代物件，來模擬雲端服務的行為。

例如，您可以撰寫使用 Amazon S3 服務模擬的測試，該測試會在呼叫CreateObject方法時傳回特定回應。執行測試時，模擬物件會傳回該項已設計程式的回應，而不呼叫 Amazon S3 或任何其他服務端點。

模擬物件通常由模擬架構產生，以減少開發工作量。一些模擬框架是通用的，其他模擬框架是專門為 AWS SDK 設計的，例如 Moto，一個用於嘲笑 AWS 服務和資源的 Python 庫。

請注意，模擬物件與模擬器不同，模擬通常由開發人員建立或設定為測試程式碼的一部分，而模擬器是獨立的應用程式，它會以與模擬的系統相同的方式公開功能。

以下是使用模擬物件的優勢：

- 模擬物件可以模擬超出應用程式控制範圍的第三方服務，例如 API 和軟體即服務 (SaaS)供應商，無需直接存取這類服務。
- 模擬物件在測試失敗條件非常實用 (尤其當這種情況很難模擬時，例如服務中斷)。
- 模擬物件可以在設定後讓您迅速進行本機測試。
- 模擬物件可以為幾乎任何類型的物件提供替代行為，因此模擬策略可以為各種比模擬器更廣泛的服務建立涵蓋範圍。

- 當新功能或行為開放使用時，模擬物件測試便可以更迅速地做出回應。通過使用通用的模擬框架，您可以在更新的 AWS SDK 可用時立即模擬新功能。

以下是模擬物件測試的缺點：

- 模擬物件通常需要進行相當複雜的設定和組態工作，尤其是在嘗試確定不同服務的傳回值以正確模擬回應的情況。
- 模擬物件是由開發人員撰寫、設定以及進行必要維護，而這會加重他們的責任。
- 您可能需要存取雲端，才能了解服務的 API 和傳回值。
- 模擬物件維護起來可能並不容易。當模擬的雲端 API 簽章發生變更，或傳回值結構描述發生變化時，便必須更新模擬。若要為了呼叫新的 API 而擴展應用程式邏輯，則也必須更新模擬物件。
- 使用模擬物件的測試可能會在桌面環境中順利通過，但在雲端中卡住。結果可能會與目前的 API 不相符。您無法對服務組態和配額進行測試。
- 模擬框架在測試或檢測 AWS Identity and Access Management (IAM) 政策或配額限制方面受到限制。雖然模擬物件在授權失敗或超過配額時的模擬效果更佳，但您無法透過測試確定生產環境中實際將會發生的結果。

透過模擬進行測試

仿真器通常是模仿生 AWS 產服務的本地運行應用程序。

模擬器具有類似於其雲端對應項目的 API，且會提供類似的傳回值。模擬器也可以模擬由 API 呼叫啟動的狀態變更。例如，您可以使 AWS SAM 用 AWS SAM 本機執行函數來模擬 Lambda 服務，以便快速叫用函數。如需詳細資訊，請參閱 《AWS Serverless Application Model 開發人員指南》 中的 AWS SAM 本機。

以下是使用模擬器進行測試的優點：

- 模擬器可讓您快速進行本機開發迭代和測試。
- 模擬器為習慣在本機環境中開發程式碼的開發人員提供了一個熟悉的環境。例如，如果您熟悉 N 層應用程式的開發，則可能會具有類似於在生產環境、本機電腦中執行的資料庫引擎和 Web 伺服器，以提供快速、本機且隔離的測試功能。
- 模擬器不需要對雲端基礎設施 (例如開發人員的雲端帳戶) 進行任何變更，因此可以輕鬆透過現有的測試模式進行實作。

以下是使用模擬器進行測試的缺點：

- 模擬器可能不易進行設定和複製，尤其是用於 CI/CD 管道時。這可能會導致管理個人軟體的 IT 人員或開發人員的工作量有所提升。
- 模擬功能和 API 通常會跟不上服務更新。這可能會導致出現錯誤，因為測試的程式碼與實際的 API 不相符，且阻礙了新功能的採用。
- 模擬器需要在支援、更新、錯誤修復和功能平等方面有所提升。這些工作是模擬器開發者 (可能為第三方公司) 的責任。
- 使用模擬器的測試可能會在本機產生成功的結果，但可能會因為生產安全政策、服務間組態或超過 Lambda 配額而在雲端中產生失敗。
- 許多 AWS 服務沒有可用的仿真器。如果您仰賴模擬，則可能會無法對應用程式的某些部分以令人滿意方式進行測試。

最佳實務

下列各節提供成功測試無伺服器應用程式的建議。

您可以在 無伺服器測試範例儲存庫 中找到測試和測試自動化的實際範例。

排定雲端測試的優先順序

在雲端進行測試可提供最可靠、精準且完整的測試涵蓋範圍。在雲端環境中執行測試不僅會全面測試商業邏輯，還會測試安全政策、服務組態、配額，以及最新的 API 簽章和傳回值。

構建程式碼以實現可測試性

將 Lambda 專屬程式碼與核心商業邏輯分隔開來，以簡化您的測試和 Lambda 函數。

您的 Lambda 函數 處理常式 應為一個小型的轉接器，它會接收事件資料，且只將重要的詳細資料傳遞給您的商業邏輯方法。您可以透過這項策略以商業邏輯為核心進行全面測試，無需擔心特定 Lambda 詳細資訊。您的 AWS Lambda 函數不需要設定複雜的環境或大量相依性，即可建立和初始化待測元件。

一般而言，您應撰寫一個處理常式，從傳入的 事件 和 內容 物件擷取和驗證資料，然後將該輸入傳送至執行商業邏輯的方法。

加快回饋迴圈的開發速度

您可以透過一些工具和技術加快回饋迴圈的開發速度。例如，AWS SAM Accelerate 和 AWS CDK 監看模式都可以縮短更新雲端環境所需的時間。

GitHub 無伺服器測試範例儲存庫中的範例會探索其中一些技術。

我們也建議您在開發期間儘早建立和測試雲端資源，而不只是在登記至原始碼控制後才進行。這種作法可在制定解決方案時加快探索和實驗的速度。此外，從開發電腦自動化部署作業可協助您更快發現雲端組態問題，並減少更新和程式碼審查程序所浪費的人力。

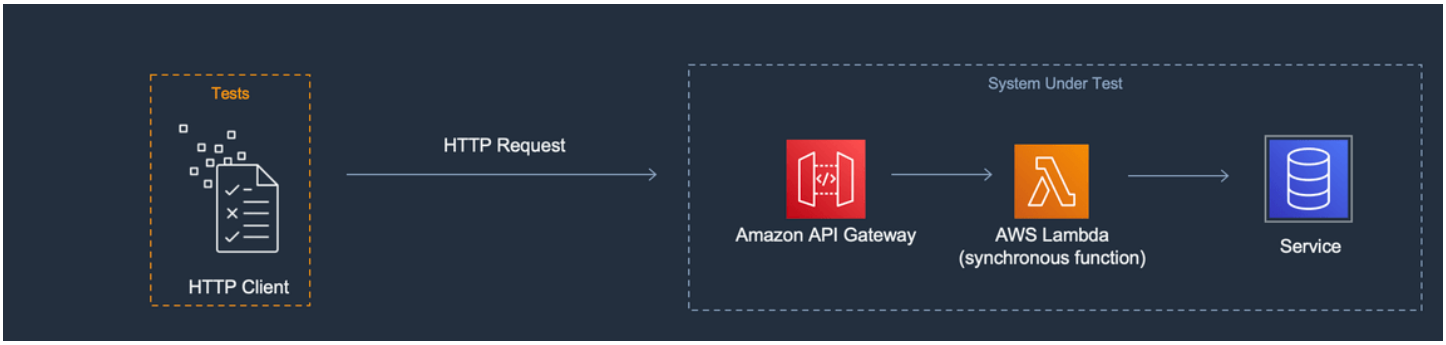
全力進行整合測試

使用 Lambda 建置應用程式時，最佳作法是一起測試元件。

針對兩個 (或以上) 架構元件進行的測試稱為 整合測試。整合測試的目標不僅是了解程式碼將如何在各個元件中執行，還要了解託管程式碼的環境將會如何運作。E nd-to-end 測試是特殊類型的集成測試，用於驗證整個應用程序的行為。

若要建置整合測試，請將應用程式部署至雲端環境。您可以透過本機環境或 CI/CD 管道完成這項動作。接著，請撰寫測試以訓練待測試的系統 (SUT)，並驗證預期的行為。

例如，待測試的系統可能是使用 API Gateway、Lambda 和 DynamoDB 的應用程式。測試可以對 API Gateway 端點進行合成 HTTP 呼叫，並驗證回應是否包含預期的有效負載。此測試會驗證 AWS Lambda 程式碼是否正確，而且每個服務都已正確設定為處理請求，包括它們之間的 IAM 許可。此外，您可以將測試設計為寫入各種大小的紀錄，以驗證服務配額 (例如 DynamoDB 中的最大紀錄大小) 是否設定正確。



建立獨立的測試環境

在雲端中進行測試通常需要獨立的開發人員環境，好讓測試、資料和事件不會出現重疊的狀況。

一種方法是為每個開發人員提供專用 AWS 帳戶。這種作法可避免在共享程式碼基底中工作的多位開發人員在嘗試部署資源或調用 API 時可能出現的資源命名衝突。

自動化測試程序應為每個堆疊建立名稱獨一無二的資源。例如，您可以設置腳本或 TOML 配置文件，以便 AWS SAM CLI sam 部署或 sam sync 命令將自動指定具有唯一前綴的堆棧。

在某些情況下，開發人員共享一個 AWS 帳戶。這可能是因為堆疊中有運作、佈建及設定時成本非常高的資源。例如，您可以共用資料庫，以便更容易正確設定和植入資料

如果開發人員共用帳戶，則應設定界限，以識別擁有權並排除重疊的狀況。其中一種作法是在堆疊名稱前面加上開發人員的使用者 ID。另一種流行的作法是根據 程式碼分支 設定堆疊。使用分支界限時，環境會獨立出來，但開發人員仍可以共用資源 (例如關聯式資料庫)。當開發人員一次處理多個分支時，這種作法便是最佳實務。

雲端測試對於測試的所有階段都很有價值，包括單元測試、整合測試和 end-to-end 測試。保持適當的隔離是必要的作法，但您仍會希望 QA 環境能盡可能與您的生產環境相似。因此，團隊會為 QA 環境加入變更控制程序。

生產前環境和生產環境方面，您通常會在帳戶層級設定界限，以便將工作負載隔離出來，使其不受擾鄰問題的影響，並實作最低權限安全控制以保護敏感資料的安全。工作負載具有配額。您不會希望測試消耗分配給生產環境 (擾鄰) 的配額，或可以存取客戶的資料。負載測試是應從生產堆疊中隔離出來的另一項活動。

在所有情況下，環境都應設定警示和控制項，以避免出現不必要的支出。例如，您可以限制可建立的資源類型、層級或大小，並在預估成本超過指定閾值時設定電子郵件提醒。

將模擬物件用於隔離的商業邏輯

模擬架構是撰寫快速單元測試的實用工具。當測試涵蓋複雜的內部商業邏輯 (例如數學或財務計算或模擬) 時，它們的優勢就會特別明顯。尋找具有大量測試案例或輸入變化的單元測試，其中這些輸入不會改變對其他雲端服務的模式或呼叫內容。

模擬物件單元測試所涵蓋的程式碼也應涵蓋在雲端的測試之中。此為建議作法，因為開發人員的筆記型電腦或建置機器環境的設定方式可能會不同於雲端中的生產環境。例如，使用特定輸入參數執行時，Lambda 函數使用的記憶體或時間可能比分配到的還多。或者，您的程式碼可能會含有未以相同方式 (或完全不同的方式) 設定的環境變數，且這些差異可能會導致程式碼出現不同的行為，或執行失敗。

模擬物件在整合測試中較不具優勢，因為實作必要模擬物件的人力需求會隨著連接點的數量而上升。End-to-end 測試不應該使用模擬，因為這些測試通常處理無法使用模擬框架輕鬆模擬的狀態和複雜邏輯。

最後，請避免使用模擬雲端服務來驗證服務呼叫是否有正確實作。請改為在雲端中進行雲端服務呼叫，以驗證行為、組態和功能實作。

請謹慎使用模擬器

模擬器對於部分使用案例而言可能會很方便 (例如網際網路存取受限、不可靠或緩慢的開發團隊)。不過，在大多數情況下，請謹慎使用模擬器。

在不使用模擬器的情況下，您將能夠使用最新的服務功能和最新的 API 進行建置和創新。您無需等待供應商發布新版本，即可實現功能平等。您將可以減少購買和設定多個開發系統和建置機器的前期和持續支出。此外，您也可以避開許多雲端服務根本就不提供模擬器的問題。仰賴模擬的測試策略將使您無法使用這些服務 (進而衍生出成本可能更高的解決方法)，或產生未經過良好測試的程式碼和組態。

當您使用模擬進行測試時，您仍然必須在雲端中進行測試，以驗證組態，並測試與雲端服務的互動 (只能在模擬環境中進行模擬)。

在本機進行測試的難題

當您使用模擬器和模擬呼叫在本機電腦上進行測試時，當程式碼在 CI/CD 管道中從一個環境進入到另一個環境時，您可能會遇到測試不一致的情況。在電腦上驗證應用程式商業邏輯的單元測試可能無法準確測試雲端服務的關鍵面向。

以下範例提供了使用模擬物件和模擬器進行本機測試時應留意的情況：

範例：Lambda 函數建立 S3 儲存貯體

如果 Lambda 函數的邏輯仰賴於建立 S3 儲存貯體，則完整測試應確認已呼叫 Amazon S3，且儲存貯體已成功建立。

- 在模擬物件測試設定中，您可能會模擬成功回應，並可能加入測試案例來應對回應失敗的狀況。
- 在模擬測試案例中，可能會呼叫 CreateBucketAPI，但您需要注意，進行本機呼叫的身分並非來自 Lambda 服務。呼叫身分不會像在雲端中一樣擔任安全角色，因此會改用預留位置驗證 (可能會有更寬鬆的角色或使用者身分，在雲端中執行時會有所不同)。

模擬物件和模擬設定將測試 Lambda 函數在呼叫 Amazon S3 時會執行的動作；不過，這些測試將不會驗證設定的 Lambda 函數 是否能成功建立 Amazon S3 儲存貯體。您必須確定指派給函數的角色具有允許函數執行 `s3:CreateBucket` 動作的附加安全政策。若沒有的話，該函數在部署到雲端環境時可能會出現失敗的狀況。

範例：Lambda 函數處理來自 Amazon SQS 佇列的訊息

如果 Amazon SQS 佇列是 Lambda 函數的來源，則完整測試應驗證訊息放入佇列時是否已成功調用 Lambda 函數。

模擬測試和模擬物件測試通常設定為直接執行 Lambda 函數程式碼，並透過傳遞 JSON 事件承載 (或還原序列化物件) 作為函數處理常式的輸入來模擬 Amazon SQS 整合。

模擬 Amazon SQS 整合的本機測試將測試 Amazon SQS 使用指定承載呼叫 Lambda 函數時，Lambda 函數會執行的動作，但測試不會驗證 Amazon SQS 在部署到雲端環境時是否會成功調用 Lambda 函數。

以下是您在使用 Amazon SQS 和 Lambda 時可能會遇到的一些組態問題範例：

- Amazon SQS 可見性逾時過低，導致原本只要調用一次，變成調用多次。
- Lambda 函數的執行角色不允許從佇列 (透過 `sqs:ReceiveMessage`、`sqs:DeleteMessage` 或 `sqs:GetQueueAttributes`) 讀取訊息。
- 傳遞至 Lambda 函數的範例事件超出 Amazon SQS 訊息大小配額。因此測試無效，因為 Amazon SQS 永遠無法傳送那麼大的訊息。

如上述範例所示，涵蓋商業邏輯但不涵蓋雲端服務之間組態的測試可能會產生不可靠的結果。

常見問答集

我有一個 Lambda 函數，它可以執行計算並傳回結果，無需呼叫任何其他服務。我真的需要在雲端進行測試嗎？

是。Lambda 函數具有可能改變測試結果的組態參數。所有 Lambda 函數程式碼都依賴於逾時和記憶體設定，如果未正確設定這些設定，可能會導致函數失敗。Lambda 政策也啟用 Amazon 的標準輸出記錄功能 CloudWatch。即使您的代碼沒有 CloudWatch 直接調用，也需要許可才能啟用日誌記錄。此必要許可無法精確模擬。

雲端中的測試如何協助進行單元測試？ 如果測試在雲中進行且連接到其他資源，那這樣不就是整合測試嗎？

我們將 單元測試 定義為獨立在架構組件上運行的測試，但這並不會阻止測試加入可能會呼叫其他服務或使用某些網路通訊的元件。

許多無伺服器應用程式都具有可隔離進行測試的架構元件 (即使在雲端也是如此)。其中一個例子是接受輸入、處理資料並將訊息傳送至 Amazon SQS 佇列的 Lambda 函數。此函數的單元測試可能會測試輸入值是否會導致某些值出現在佇列訊息之中。

考慮使用「準備、執行、驗證」模式撰寫的測試：

- 準備：分配資源 (用於接收訊息的佇列和待測函數)。
- 執行：呼叫待測函數。
- 驗證：擷取函數發送的訊息，並驗證輸出。

模擬物件測試方法包括使用正在進行的模擬物件來模擬佇列，以及建立含有 Lambda 函數程式碼的類別或模組的進行中執行個體。驗證階段期間，佇列的訊息會從模擬物件進行擷取。

在雲端方法中，測試會針對測試目的建立 Amazon SQS 佇列，並透過設定為將隔離的 Amazon SQS 佇列作為輸出目的地使用的環境變數來部署 Lambda 函數。執行 Lambda 函數後，測試會從 Amazon SQS 佇列擷取訊息。

雲端測試會執行相同的程式碼、驗證相同的行為，並驗證應用程式功能是否正確。不過，它具有能夠驗證 Lambda 函數設定的額外優勢：IAM 角色、IAM 政策以及函數的逾時和記憶體設定。

後續步驟和資源

使用以下資源以進一步了解並探索測試的實際範例。

實作範例

的無伺服器測試範例儲存庫 GitHub 包含遵循本指南中描述的模式和最佳作法的具體測試範例。儲存庫包含前幾節所述之模擬物件、模擬和雲端測試程序的範例程式碼和引導式逐步解說。使用此儲存庫可以快速獲得最新的無伺服器測試指南。 AWS

深入閱讀

造訪無伺服器領域網站，存取最新的部落格、影片和 AWS 無伺服器技術訓練。

還建議閱讀以下 AWS 博客文章：

- 使用加速加速加速無伺服器開 AWS SAM 發 (AWS 部落格文章)
- 使用 CDK 手錶提高開發速度（AWS 博客文章）
- 嘲笑服務集成與 AWS Step Functions 本地（AWS 博客文章）
- 開始測試無伺服器應用程式 (AWS 部落格文章)

工具

- AWS SAM — 測試和偵錯無伺服器應用程式
- AWS SAM — 與自動化測試集成
- Lambda：在 Lambda 主控台中測試 Lambda 函數

使用 Node.js 建置 Lambda 函數

您可以在中使用 Node.js 執行 JavaScript 程式碼 AWS Lambda。Lambda 提供用於執行程式碼來處理事件的 Node.js 執行期。您的程式碼會在包含您管理之 AWS SDK for JavaScript AWS Identity and Access Management (IAM) 角色的登入資料的環境中執行。若要進一步瞭解 Node.js 執行階段隨附的 SDK 版本，請參閱the section called “包含執行階段的 SDK 版本”。

Lambda 支援以下 Node.js 執行期。

Node.js

名稱	識別符	作業系統	取代日期	封鎖函數建立	封鎖函數更新
Node.js 20	`nodejs20.x`	Amazon Linux 2023			
Node.js 18	`nodejs18.x`	Amazon Linux 2			
Node.js 16	`nodejs16.x`	Amazon Linux 2	2024 年 6 月 12 日	2025年2月28日	2025年3月31日

> **Note**
>
> Node.js 18 及更新版本的執行階段會針對 JavaScript v3 使用 AWS SDK。若要從較早的執行階段移轉函數，請遵循上的移轉研討會 GitHub。如需有關第 3 JavaScript 版 AWS SDK 的詳細資訊，請參閱模組化 AWS SDK 的部落格文章。 JavaScript

若要建立 Node.js 函數

1. 開啟 Lambda 主控台。
2. 選擇建立函數。
3. 進行下列設定：
   - 函數名稱：輸入函數名稱。
   - 執行期：選擇 Node.js 20.x。

4. 選擇建立函數。
5. 若要設定測試事件，請選擇 Test (測試)。
6. 事件名稱輸入 **`test`**。
7. 選擇儲存變更。
8. 若要調用函數，請選擇 Test (測試)。

主控台會建立一個 Lambda 函數，其具有名為 `index.js` 或 `index.mjs` 的單一來源檔案。您可以使用內建的程式碼編輯器編輯該檔案並加入更多檔案。選擇 Save (儲存) 以儲存變更。然後，若要執行程式碼，請選擇 Test (測試)。

> **Note**
>
> Lambda 主控台用 AWS Cloud9 來在瀏覽器中提供整合式開發環境。您也可以使用 AWS Cloud9 在自己的環境中開發 Lambda 函數。若要取得更多資訊，請參閱使用指南 AWS 工具組中的〈使用 AWS Lambda 函數〉。 AWS Cloud9

`index.js` 或 `index.mjs` 檔案會匯出名為 `handler` 的函數，用於接受事件物件與內容物件。這就是在調用函數時，Lambda 呼叫的處理常式函數。Node.js 函數執行期會從 Lambda 中取得調用事件並將它們傳遞至處理常式。在函式組態中，處理常式值為 `index.handler`。

當您儲存函數程式碼時，Lambda 主控台會建立 .zip 封存檔部署套件。當您在主控台之外開發函數程式碼 (使用 IDE) 時，您需要建立部署套件將您的程式碼上傳到 Lambda 函數。

> **Note**
>
> 若要在您的本機環境中開始進行應用程式開發，請部署本指南 GitHub 儲存庫中提供的其中一個範例應用程式。
>
> 以 Node.js 編寫的範例 Lambda 應用程式
>
> - 空白 nodejs-一個 Node.js 函數，顯示日誌記錄，環境變量， AWS X-Ray 跟踪，圖層，單元測試和 SDK 的使用。 AWS
> - nodejs-apig - 具有公有 API 端點的函數，它會處理來自 API Gateway 的事件並傳回 HTTP 回應。

- efs-nodejs - 在 Amazon VPC 中使用 Amazon EFS 檔案系統的函數。此範例包含設為與 Lambda 搭配使用的 VPC、檔案系統、掛載目標以及存取點。

除了傳遞調用事件外，函式執行期還會傳遞內容物件至處理常式。內容物件包含了有關調用、函式以及執行環境的額外資訊。更多詳細資訊將另由環境變數提供。

您的 Lambda 函數隨附一個 CloudWatch 日誌記錄群組。函數運行時將有關每次調用的詳細信息發送到 CloudWatch 日誌。它在調用期間會轉送您的函數輸出的任何記錄。如果您的函數傳回錯誤，Lambda 會對該錯誤進行格式化之後傳回給調用端。

主題

Node.js 初始化

Node.js 有一個唯一的事件迴圈模型，導致其初始化行為與其他執行期不同。具體而言，Node.js 使用支援異步操作的非阻塞 I/O 模型。此模型允許 Node.js 高效地執行大多數工作負載。例如，如果 Node.js 函數進行網路呼叫，則該請求可能被指定為異步操作並放入回呼佇列中。函數可能會繼續處理主呼叫堆疊中的其他操作，而不會透過等待網路呼叫返回而被阻止。網路呼叫完成後，系統會執行其回呼，然後從回呼佇列中移除。

某些初始化任務可異步執行。不能保證這些異步任務在調用之前完成執行。例如，在 Lambda 執行處理常式函數時，進行網路呼叫以從 AWS 參數存放區擷取參數的程式碼可能無法完成。因此，在調用期間，變數可能為空。為避免這種情況，請務必在繼續執行其餘的函數核心商業邏輯之前，將變數和其他非同步程式碼完全初始化。

或者，您可以將函數程式碼指定為 ES 模組，這樣便能使用位於檔案最上層的 `await`，超出了函數處理常式的範圍。當您對每個 `Promise` 執行 `await`，非同步初始化程式碼會在處理常式調用之前完成，從而在降低冷啟動延遲方面最大限度地提高佈建並行的效率。如需詳細資訊和範例，請參閱在 AWS Lambda中使用 Node.js ES 模組和頂層 await。

將函數處理常式指定為 ES 模組

預設情況下，Lambda 會將帶有 `.js` 尾碼的檔案視為 CommonJS 模組。您可以選擇將程式碼指定為 ES 模組。您可利用兩種方式進行：在函數的 `package.json` 檔案中將 `type` 指定為 `module`，或使用 `.mjs` 副檔名。若使用第一種方式下，函數程式碼會將所有 `.js` 檔案視為 ES 模組，而在第二種情況下，只有您使用 `.mjs` 指定的檔案為 ES 模組。您可以透過分別將它們命名為 `.mjs` 和 `.cjs` 來混合 ES 模組和 CommonJS 模組，因為 `.mjs` 檔案一律為 ES 模組，而 `.cjs` 檔案一律為 CommonJS 模組。

載入 ES 模組時，Lambda 會在`NODE_PATH`環境變數中搜尋資料夾。您可以使用 ES 模組`import`陳述式載入執行階段中包含的 AWS SDK。您也可以從分層載入 ES 模組。

包含執行階段的 SDK 版本

Node.js 執行階段中包含的 AWS SDK 版本取決於執行階段版本和您的 AWS 區域. 若要尋找您正在使用的執行階段中包含的 SDK 版本，請使用下列程式碼建立 Lambda 函數。

> **Note**
>
> 下面顯示的 Node.js 版本 18 及以上的示例代碼使用 CommonJS 格式。如果您在 Lambda 主控台中建立函數，請務必將包含程式碼的檔案重新命名為`index.js`。

Example Node.js 18 及以上版本

```
const { version } = require("@aws-sdk/client-s3/package.json");

exports.handler = async () => ({ version });
```

這會傳回下列格式的回應：

```
{
  "version": "3.462.0"
```

```
}
```

使用保持連線保持 TCP 連線

預設 Node.js HTTP/HTTPS 代理程式會為每個新的請求建立新的 TCP 連線。為了避免建立新連接的成本，您可以使`keepAlive: true`用重複使用函數使用 AWS SDK 進行的連接 JavaScript。保持連線可以減少使用 SDK 進行多次 API 呼叫的 Lambda 函數的請求次數。

在包含在 Lambda 執行階段`nodejs18.x`及更新版本的 JavaScript 3.x 版 AWS SDK 中，依預設會啟用保持活動狀態。若要停用保持活動狀態，請參閱 3.x AWS 版 SDK 開發人員指南中的 Node.js 中以保持活動狀態重複使用連線。 JavaScript 有關使用保持活動的更多信息，請參閱開發人員工具博客上的模塊化 AWS SDK 默認打開 HTTP 保持活動狀態。 JavaScript AWS

CA 憑證載入

對於 Node.js 18 以下的 Node.js 執行階段版本，Lambda 會自動載入亞馬遜特定的 CA (憑證授權單位) 憑證，讓您更輕鬆地建立與其他 AWS 服務互動的函數。例如，Lambda 包含在 Amazon RDS 資料庫上安裝的驗證伺服器身分憑證所需的 Amazon RDS 憑證。此行為可能會在冷啟動期間產生效能影響。

從 Node.js 20 開始，在預設情況下，Lambda 不再載入額外的 CA 憑證。Node.js 20 執行期包含一個憑證檔案，其中包含所有 Amazon CA 憑證位於 `/var/runtime/ca-cert.pem`。若要從 Node.js 18 及更早版本的執行期還原相同的行為，請將 `NODE_EXTRA_CA_CERTS`環境變數設定為 `/var/runtime/ca-cert.pem`。

為了獲得最佳效能，我們建議您只將需要的憑證與部署套件搭配，並透過 `NODE_EXTRA_CA_CERTS` 環境變數載入憑證。憑證檔案應包含一或多個 PEM 格式的受信任根憑證或中繼 CA 憑證。例如，對於 RDS，請在程式碼旁包含所需的憑證做為 `certificates/rds.pem`。然後，藉由將 `NODE_EXTRA_CA_CERTS` 設定為 `/var/task/certificates/rds.pem` 載入憑證。

在 Node.js 中 Lambda 義函數處理常式

Lambda 函數處理常式是您的函數程式碼中處理事件的方法。當有人呼叫您的函數時，Lambda 會執行處理常式方法。函數會執行，直到處理常式傳回回應、結束或逾時為止。

下列範例函數會記錄事件物件的內容，並傳回記錄檔的位置。

> **Note**
>
> 此頁面會顯示 CommonJS 和 ES 模組處理常式的範例。若要了解這兩個處理常式類型之間的差異，請參閱 將函數處理常式指定為 ES 模組。

ES module handler

Example

```
export const handler = async (event, context) => {
  console.log("EVENT: \n" + JSON.stringify(event, null, 2));
  return context.logStreamName;
};
```

CommonJS module handler

Example

```
exports.handler = async function (event, context) {
  console.log("EVENT: \n" + JSON.stringify(event, null, 2));
  return context.logStreamName;
};
```

當您設定函數時，處理常式的設定值就是檔案名稱，以及已匯出之處理常式方法的名稱，並且以點分隔。主控台中的預設值，例如在此指南中為 `index.handler`。這表示 `handler` 方法是透過 `index.js` 檔案匯出的。

執行程序會將引數傳送至處理常式方法。第一個引數是 `event` 物件，其中包含來自叫用端的資訊。呼叫者會在呼叫 Invoke 時，以 JSON 格式的字串傳遞此資訊，然後執行時間將它轉換為物件。當 AWS 服務叫用您的函數時，事件結構會因服務而異。

第二個引數為 內容物件，其中包含有關呼叫、函式和執行環境的資訊。在上述範例中，該函式從內容物件取得日誌串流的名稱並傳回給叫用端。

您也可以使用回呼引數 (此引數是您在非同步處理常式中呼叫來傳送回應的函數)。建議您使用非同步/等待 (而不是回呼)。非同步/等待改善了可讀性、錯誤處理及效率。如需有關非同步/等待和回呼之間差異的詳細資訊，請參閱 使用回呼。

命名

當您設定函數時，處理常式的設定值就是檔案名稱，以及已匯出之處理常式方法的名稱，並且以點分隔。在控制台中創建的功能和本指南中的示例的默認值是`index.handler`。這表示從`index.js`或`index.mjs`檔案匯出的`handler`方法。

如果要在主控台中使用不同檔案名稱或函數處理常式名稱建立函數，您必須編輯預設處理常式名稱。

變更函數處理常式名稱的方式 (主控台)

1. 開啟 Lambda 主控台的函數頁面，然後選擇您的函數。
2. 選擇 程式碼 索引標籤。
3. 向下捲動至執行時間設定窗格，並選擇編輯。
4. 在處理常式中，輸入函數處理常式的新名稱。
5. 選擇 儲存 。

使用 async/await

如果您的程式碼執行非同步任務，請使用非同步/等待模式，以確保處理常式能順利完成執行。非同步/等待是一種在 Node.js 中撰寫非同步程式碼的簡潔可讀模式，無需巢狀回呼或鏈結承諾。您可以透過非同步/等待模式撰寫讀起來像同步程式碼的程式碼，同時仍維持非同步和非封鎖的特性。

`async` 關鍵字會將函數標記為非同步，且 `await` 關鍵字會暫停函數的執行，直到 `Promise` 獲得解決為止。

> **Note**
>
> 請務必等待非同步事件完成。如果函數在非同步事件完成之前傳回，則函數可能會失敗或導致應用程式中的非預期行為。當 `forEach` 迴圈包含非同步事件時，可能會發生這種情況。`forEach` 迴圈期望一個同步呼叫。如需更多資訊，請參閱 Mozilla 文件中的 Array.prototype.forEach()。

ES module handler

Example - 具有 async/await 的 HTTP 請求

```
const url = "https://aws.amazon.com/";

export const handler = async(event) => {
    try {
        // fetch is available in Node.js 18 and later runtimes
        const res = await fetch(url);
        console.info("status", res.status);
        return res.status;
    }
    catch (e) {
        console.error(e);
        return 500;
    }
};
```

CommonJS module handler

Example - 具有 async/await 的 HTTP 請求

```
const https = require("https");
let url = "https://aws.amazon.com/";

exports.handler = async function (event) {
  let statusCode;
  await new Promise(function (resolve, reject) {
    https.get(url, (res) => {
        statusCode = res.statusCode;
        resolve(statusCode);
      }).on("error", (e) => {
        reject(Error(e));
      });
  });
  console.log(statusCode);
  return statusCode;
};
```

下一個範例會使用非同步/等待來列出您的 Amazon Simple Storage Service 儲存貯體。

> **Note**
>
> 使用此範例前，請確定函數的執行角色具有 Amazon S3 讀取許可。

ES module handler

Example — 具有異步/等待的 AWS SDK v3

此範例使用 AWS SDK for JavaScript v3，它可在以後的執行階段中nodejs18.x使用。

```
import {S3Client, ListBucketsCommand} from '@aws-sdk/client-s3';
const s3 = new S3Client({region: 'us-east-1'});

export const handler = async(event) => {
  const data = await s3.send(new ListBucketsCommand({}));
  return data.Buckets;

};
```

CommonJS module handler

Example — 具有異步/等待的 AWS SDK v3

此範例使用 AWS SDK for JavaScript v3，它可在以後的執行階段中nodejs18.x使用。

```
const { S3Client, ListBucketsCommand } = require('@aws-sdk/client-s3');
const s3 = new S3Client({ region: 'us-east-1' });

exports.handler = async (event) => {
  const data = await s3.send(new ListBucketsCommand({}));
  return data.Buckets;
};
```

使用回呼

建議您使用 非同步/等待 來宣告函數處理常式，而不是使用回呼。非同步/等待是更好的選擇，以下列出幾項原因：

- 可讀性： 非同步/等待程式碼比回呼程式碼更容易閱讀和理解，回呼程式碼可能很快就會變得難以理解，並引發回呼地獄。

- 偵錯和錯誤處理： 回呼型程式碼的偵錯工作難度可能不低。呼叫堆疊可能會變得難以理解，且可能會很容易接受錯誤。您可以透過非同步/等待使用 try/catch 區塊來處理錯誤。
- 效率： 回呼通常需要在程式碼的不同部分之間進行切換。非同步/等待可以減少切換環境的次數，進而產生更有效率的程式碼。

在處理常式中使用回呼時，函數將繼續執行，直到 事件迴圈 清空或函數逾時為止。直到完成所有的事件迴圈任務，回應才會傳送到叫用端。如果函式逾時，便會傳回錯誤。您可以通過將上下文 .callbackWaitsFor EmptyEvent 循環設置為 false 來配置運行時立即發送響應。

回乎函式需要兩個引數，一個 `Error` 和回應。回應物件必須與 `JSON.stringify` 相容。

以下範例函式檢查 URL 及傳回狀態碼給叫用端。

ES module handler

Example - 具有 callback 的 HTTP 請求

```
import https from "https";
let url = "https://aws.amazon.com/";

export function handler(event, context, callback) {
  https.get(url, (res) => {
      callback(null, res.statusCode);
    }).on("error", (e) => {
      callback(Error(e));
    });
}
```

CommonJS module handler

Example - 具有 callback 的 HTTP 請求

```
const https = require("https");
let url = "https://aws.amazon.com/";

exports.handler = function (event, context, callback) {
  https.get(url, (res) => {
      callback(null, res.statusCode);
    }).on("error", (e) => {
      callback(Error(e));
    });
```

```
};
```

在下一個範例中，Amazon S3 的回應會在可用時立即傳回給啟動程式。逾時的事件迴圈已凍結，並繼續執行下一個時間叫用的函式。

> **Note**
>
> 使用此範例前，請確定函數的執行角色具有 Amazon S3 讀取許可。

ES module handler

Example -帶 callbackWaitsForEmptyEvent循環的 AWS SDK V3

此範例使用 AWS SDK for JavaScript v3，它可在以後的執行階段中nodejs18.x使用。

```
import AWS from "@aws-sdk/client-s3";
const s3 = new AWS.S3({});

export const handler = function (event, context, callback) {
  context.callbackWaitsForEmptyEventLoop = false;
  s3.listBuckets({}, callback);
  setTimeout(function () {
    console.log("Timeout complete.");
  }, 5000);
};
```

CommonJS module handler

Example -帶 callbackWaitsForEmptyEvent循環的 AWS SDK V3

此範例使用 AWS SDK for JavaScript v3，它可在以後的執行階段中nodejs18.x使用。

```
const AWS = require("@aws-sdk/client-s3");
const s3 = new AWS.S3({});

exports.handler = function (event, context, callback) {
  context.callbackWaitsForEmptyEventLoop = false;
  s3.listBuckets({}, callback);
  setTimeout(function () {
    console.log("Timeout complete.");
```

```
  }, 5000);
};
```

使用 .zip 封存檔部署 Node.js Lambda 函數

AWS Lambda 函數的程式碼包含包含函式處理常式程式碼的 .js 或 .mjs 檔案，以及您的程式碼所依賴的任何其他套件和模組。若要將此函數程式碼部署到 Lambda，您可以使用部署套件。此套件可以是 .zip 封存檔或容器映像。如需有關搭配使用容器映像與 Node.js 的詳細資訊，請參閱使用容器映像部署 Node.js Lambda 函數。

若要建立 .zip 封存檔的部署套件，您可以使用命令列工具的內建 .zip 封存檔公用程式，或任何其他 .zip 檔案公用程式 (例如 7zip)。以下各節顯示的範例假設您在 Linux 或 MacOS 環境中使用命令列 `zip` 工具。若要在 Windows 中使用相同命令，您可以安裝適用於 Linux 的 Windows 子系統，以取得 Ubuntu 和 Bash 的 Windows 整合版本。

請注意，Lambda 使用 POSIX 檔案許可，因此在建立 .zip 封存檔之前，您可能需要設定部署套件資料夾的許可。

主題

Node.js 中的執行期相依項

對於使用 Node.js 執行期的 Lambda 函數，相依項可以是任何 Node.js 模組。Node.js 執行期包含許多常見程式庫，以及 AWS SDK for JavaScript的某個版本。`nodejs16.x` Lambda 執行期包含 SDK 的第 2.x 版。執行期版本 `nodejs18.x` 及更新版本包含 SDK 的第 3 版本。若要搭配執行期 `nodejs18.x` 及更新版本使用 SDK 的第 2 版，請將 SDK 新增至您的 .zip 檔部署套件。如果選擇的執行期包含正在使用的 SDK 版本，則不需要在 .zip 檔案中包含 SDK 程式庫。若要瞭解您正在使用的執行階段中包含哪個版本的 SDK，請參閱the section called “包含執行階段的 SDK 版本”。

Lambda 會定期更新 Node.js 執行期中的 SDK 程式庫，以納入最新功能和安全升級。Lambda 也會將安全修補程式和更新套用至執行期中包含的其他程式庫。若要完全控制套件中的相依項，可以將任何包含執行期之相依項的偏好版本新增至部署套件。例如，如果您想要使用特定版本的 SDK JavaScript，您可以將其作為相依性包含在 .zip 檔案中。如需有關將包含執行期的相依項新增至 .zip 檔案的詳細資訊，請參閱 相依項搜尋路徑和含執行期程式庫。

在 AWS 共同責任模式下，您負責管理函數部署套件中的任何相依項。這包括套用更新和安全性修補程式。若要更新函數部署套件中的相依項，請先建立新的 .zip 檔案，然後將其上傳至 Lambda。如需詳細資訊，請參閱 建立含相依項的 .zip 部署套件 和 使用 .zip 檔案建立及更新 Node.js Lambda 函數。

建立不含相依項的 .zip 部署套件

如果除了 Lambda 執行期中包含的程式庫之外，函數程式碼沒有其他相依項，則 .zip 檔案只包含具有函數處理常式程式碼的 `index.js` 或 `index.mjs` 檔案。使用您慣用的 zip 公用程式建立 .zip 檔案，並將 `index.js` 或 `index.mjs` 檔案放在根目錄下。如果包含處理常式程式碼的檔案不在 .zip 檔案的根目錄下，則 Lambda 將無法執行程式碼。

若要了解如何部署 .zip 檔案以建立新的 Lambda 函數或更新現有函數，請參閱使用 .zip 檔案建立及更新 Node.js Lambda 函數。

建立含相依項的 .zip 部署套件

如果函數程式碼相依於 Lambda Node.js 執行期中不包含的套件或模組，則可以使用函數程式碼將這些相依項新增至 .zip 檔案，或使用 Lambda 層。本節中的指示說明如何在 .zip 部署套件中包含相依項。如需如何在層中包含相依項的指示，請參閱the section called “為相依項建立 Node.js 層”。

下列範例 CLI 命令會建立名為 `my_deployment_package.zip` 的 .zip 檔案，其中包含帶有函數處理常式程式碼及其相依項的 `index.js` 或 `index.mjs` 檔案。在此範例中，可以使用 npm 套件管理工具來安裝相依項。

建立部署套件

1. 導覽到包含 `index.js` 或 `index.mjs` 原始程式碼檔案的專案目錄。在此範例中，目錄名為 `my_function`。

   ```
   cd my_function
   ```

2. 使用 `npm install` 命令將函數的所需程式庫安裝在 `node_modules` 目錄中。在此範例中，將安裝 適用於 Node.js 的 AWS X-Ray SDK。

   ```
   npm install aws-xray-sdk
   ```

   這會建立如下資料夾結構：

   ```
   ~/my_function
   ```

```
### index.mjs
### node_modules
    ### async
    ### async-listener
    ### atomic-batcher
    ### aws-sdk
    ### aws-xray-sdk
    ### aws-xray-sdk-core
```

您也可以將自己建立的自訂模組新增至部署套件。在 node_modules 下建立一個帶有模組名稱的目錄，並將自訂編寫的套件儲存在此處。

3. 在根目錄建立包含專案資料夾內容的 .zip 檔案。使用 r (遞迴) 選項，以確保 zip 會壓縮子資料夾。

```
zip -r my_deployment_package.zip .
```

為相依項建立 Node.js 層

本節中的指示說明如何在層中包含相依項。如需如何在部署套件中包含相依項的指示，請參閱the section called “建立含相依項的 .zip 部署套件”。

將層新增至函數時，Lambda 會將層內容載入該執行環境的 /opt 目錄。在每一次 Lambda 執行期中，PATH 變數已包含 /opt 目錄中的特定資料夾路徑。若要確保PATH變數會擷取圖層內容，您的圖層 .zip 檔案應該在下列資料夾路徑中具有其相依性：

- nodejs/node_modules
- nodejs/node16/node_modules (NODE_PATH)
- nodejs/node18/node_modules (NODE_PATH)
- nodejs/node20/node_modules (NODE_PATH)

例如，您的層 .zip 檔案結構可能如下所示：

```
xray-sdk.zip
# nodejs/node_modules/aws-xray-sdk
```

此外，Lambda 會自動偵測 /opt/lib 目錄中的程式庫，以及 /opt/bin 目錄中的二進位檔案。若要確保 Lambda 正確找到您的層內容，您也可以建立結構如下的層：

```
custom-layer.zip
# lib
    | lib_1
    | lib_2
# bin
    | bin_1
    | bin_2
```

封裝層之後，請參閱the section called “建立和刪除層”及the section called “新增層”，完成層設定。

相依項搜尋路徑和含執行期程式庫

Node.js 執行期包含許多常見程式庫，以及 AWS SDK for JavaScript的某個版本。如果想要使用不同版本之包含執行期的程式庫，則可以透過將其與函數綁定在一起，或將其新增為部署套件中的相依項來執行此操作。例如，可以透過將其新增至 .zip 部署套件，來使用不同版本的 SDK。也可以將其包含在函數的 Lambda 層中。

當您在程式碼中使用 `import` 或 `require` 陳述式時，Node.js 執行期會在 `NODE_PATH` 路徑中搜尋目錄，直到找到模組為止。依預設，執行期搜尋的第一個位置是 .zip 部署套件解壓縮並掛載的目錄 (`/var/task`)。如果您在部署套件中納入含執行期程式庫的版本，則此版本的優先順序會高於執行期中包含的版本。部署套件中的相依項也優先於圖層中的相依項。

當您將相依項新增至圖層時，Lambda 會將其擷取到 `/opt/nodejs/nodexx/node_modules`，其中 `nodexx` 表示您所使用的執行期版本。在搜尋路徑中，此目錄的優先順序高於包含含執行期程式庫的目錄 (`/var/lang/lib/node_modules`)。因此，函數層中程式庫的優先順序高於執行期中包含的版本。

可以新增下列程式碼行，以查看 Lambda 函數的完整搜尋路徑。

```
console.log(process.env.NODE_PATH)
```

您也可以在 .zip 套件內的個別資料夾中新增相依項。例如，可以將自訂模組新增至名為 `common` 的 .zip 套件中的資料夾。解壓縮並掛載您的 .zip 套件時，此資料夾會放在 `/var/task` 目錄中。若要在程式碼中使用 .zip 部署套件中資料夾的相依性，請使用 `import { } from` 或 `const { } = require()` 陳述式，具體取決於您使用的是 CJS 還是 ESM 模組解析度。例如：

```
import { myModule } from './common'
```

如果將程式碼與 `esbuild`、`rollup` 或類似屬性綁定在一起，則函數使用的相依項會一起綁定在一個或多個檔案中。建議盡可能使用此方法來確定相依項。與將相依項新增至部署套件相比，綁定程式碼可提高效能，因為減少了 I/O 操作。

使用 .zip 檔案建立及更新 Node.js Lambda 函數

建立 .zip 部署套件後，您可以使用該套件建立新的 Lambda 函數或更新現有函數。您可以使用 Lambda 主控台、和 Lambda API 來部署您的 .zip 套件。 AWS Command Line Interface您也可以使用 AWS Serverless Application Model (AWS SAM) 和 AWS CloudFormation 建立並更新 Lambda 函數。

Lambda 的 .zip 部署套件大小上限為 250 MB (解壓縮)。請注意，此限制適用於您上傳的所有檔案 (包括任何 Lambda 層) 的大小總和。

Lambda 執行時間需有許可才能讀取部署套裝服務中的檔案。在 Linux 權限八進制標記法中，Lambda 需要 644 個權限才能用於不可執行的檔案 (rw-r--r--) 和 755 個權限 () 用於目錄和可執行檔。rwxr-xr-x

在 Linux 和 MacOS 中，使用 `chmod` 命令變更部署套件中檔案和目錄的檔案許可。例如，若要提供可執行檔正確的許可，請執行下列命令。

```
chmod 755 <filepath>
```

若要在 Windows 中變更檔案許可，請參閱 Microsoft Windows 文件的 Set, View, Change, or Remove Permissions on an Object。

透過主控台使用 .zip 檔案建立及更新函數

若要建立新函數，您必須先在主控台中建立函數，然後上傳您的 .zip 封存檔。若要更新現有函數，請開啟函數的頁面，然後按照同樣的程序新增更新後的 .zip 檔案。

如果您的 .zip 檔案小於 50 MB，您可以透過直接從本機電腦上傳檔案來建立或更新函數。若 .zip 檔案大於 50 MB，您必須先將套件上傳至 Amazon S3 儲存貯體。如需有關如何使用將檔案上傳到 Amazon S3 儲存貯體的指示 AWS Management Console，請參閱開始使用 Amazon S3。若要使用上載檔案 AWS CLI，請參閱《使用指南》中的AWS CLI 〈移動物件〉。

> **Note**
>
> 您無法變更現有函數的部署套件類型 (.zip 或容器映像檔)。例如，您無法將容器映像函數轉換為使用 .zip 檔案封存。您必須建立新的函數。

若要建立新的函數 (主控台)

1. 開啟 Lambda 主控台的函數頁面，然後選擇建立函數。
2. 選擇 Author from scratch (從頭開始撰寫)。
3. 在基本資訊下，請執行下列動作：
   a. 在函數名稱中輸入函數名稱。
   b. 在執行期中選取要使用的執行期。
   c. (選用) 在架構中選擇要用於函數的指令集架構。預設架構值為 x86_64。請確定函數的 .zip 部署套件與您選取的指令集架構相容。
4. (選用) 在 許可 下，展開 變更預設執行角色 。您可建立新的執行角色，或使用現有的角色。
5. 選擇建立函數。Lambda 會使用您選擇的執行期建立一個基本的「Hello world」函數。

若要從本機電腦上傳 .zip 封存檔 (主控台)

1. 在 Lambda 主控台的函數頁面中選擇要上傳 .zip 檔案的函數。
2. 選取程式碼索引標籤。
3. 在程式碼來源窗格中選擇上傳來源。
4. 選擇 .zip 檔案。
5. 若要上傳 .zip 檔案，請執行下列操作：
   a. 選擇上傳，然後在檔案選擇器中選取您的 .zip 檔案。
   b. 選擇 Open (開啟)。
   c. 選擇儲存。

若要從 Amazon S3 儲存貯體上傳 .zip 封存檔 (控制台)

1. 在 Lambda 主控台的函數頁面中選擇要上傳新 .zip 檔案的函數。
2. 選取程式碼索引標籤。
3. 在程式碼來源窗格中選擇上傳來源。
4. 選擇 Amazon S3 位置。
5. 貼上 .zip 檔案的 Amazon S3 連結 URL，然後選擇儲存。

使用主控台程式碼編輯器更新 .zip 檔案函數

對於某些具有 .zip 部署套件的函數，您可以使用 Lambda 主控台的內建程式碼編輯器直接更新函數程式碼。若要使用此功能，您的函數必須符合下列條件：

- 您的函數必須使用其中一種轉譯語言執行期 (Python、Node.js 或 Ruby)
- 函數的部署套件必須小於 3MB。

具有容器映像部署套件之函數的函數程式碼無法直接在主控台中編輯。

若要使用主控台程式碼編輯器更新函數程式碼

1. 開啟 Lambda 主控台的函數頁面，然後選取您的函數。
2. 選取程式碼索引標籤。
3. 在程式碼來源窗格中，選取您的原始程式碼檔案，然後在整合式程式碼編輯器中加以編輯。
4. 完成編輯程式碼後，請選擇部署，以儲存變更並更新函數。

使用 .zip 檔案建立和更新函數 AWS CLI

您可以使用 AWS CLI 建立新函數，或使用 .zip 檔案更新現有函數。使用建立函數和update-function-code命令來部署您的 .zip 套件。如果您的 .zip 檔案小於 50 MB，則可以從本機建置電腦的檔案位置上傳 .zip 套件。若檔案較大，則必須先從 Amazon S3 儲存貯體上傳 .zip 套件。如需有關如何使用將檔案上傳到 Amazon S3 儲存貯體的指示 AWS CLI，請參閱使用AWS CLI 者指南中的移動物件。

> **Note**
>
> 如果您使用從 Amazon S3 儲存貯體上傳 .zip 檔案 AWS CLI，則該儲存貯體必須與您的函數位於 AWS 區域 相同的位置。

若要使用 .zip 檔案與建立新函數 AWS CLI，您必須指定下列項目：

- 函數名稱 (`--function-name`)
- 函數的執行期 (`--runtime`)
- 函數執行角色的 Amazon Resource Name (ARN) (`--role`)
- 函數程式碼中處理常式方法的名稱 (`--handler`)

您也必須指定 .zip 檔案的位置。如果您的 .zip 檔案位於本機建置電腦上的資料夾中，請使用 `--zip-file` 選項來指定檔案路徑，如下列範例命令所示。

```
aws lambda create-function --function-name myFunction \
--runtime nodejs20.x --handler index.handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--zip-file fileb://myFunction.zip
```

若要在 Amazon S3 儲存貯體中指定 .zip 檔案的位置，請使用如下列範例命令所示的 `--code` 選項。您只需針對版本控制的物件使用 `S3ObjectVersion` 參數。

```
aws lambda create-function --function-name myFunction \
--runtime nodejs20.x --handler index.handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--code S3Bucket=DOC-EXAMPLE-BUCKET,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion
```

若要使用 CLI 更新現有函數，您可以使用 `--function-name` 參數指定函數的名稱。您也必須指定要用來更新函數程式碼的 .zip 檔案的位置。如果您的 .zip 檔案位於本機建置電腦上的資料夾中，請使用 `--zip-file` 選項來指定檔案路徑，如下列範例命令所示。

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

若要在 Amazon S3 儲存貯體中指定 .zip 檔案的位置，請使用如下列範例命令所示的 `--s3-bucket` 和 `--s3-key` 選項。您只需針對版本控制的物件使用 `--s3-object-version` 參數。

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket DOC-EXAMPLE-BUCKET --s3-key myFileName.zip --s3-object-version myObject
 Version
```

透過 Lambda API 使用 .zip 檔案建立及更新函數

若要使用 .zip 封存檔建立及更新函數，請使用下列 API 操作：

- CreateFunction
- UpdateFunctionCode

使用 .zip 文件創建和更新函數 AWS SAM

AWS Serverless Application Model (AWS SAM) 是一個工具組，可協助簡化在 AWS上建置和執行無伺服器應用程式的程序。您可以在 YAML 或 JSON 範本中定義應用程式的資源，並使用 AWS SAM 命令列介面 (AWS SAM CLI) 來建置、封裝及部署應用程式。當您從 AWS SAM 範本建立 Lambda 函數時， AWS SAM 會使用函數程式碼和您指定的任何相依性，自動建立 .zip 部署套件或容器映像檔。若要進一步了解如 AWS SAM 何使用建置和部署 Lambda 函數，請參閱開AWS Serverless Application Model發人員指南 AWS SAM中的入門使用。

您也可以使用現有的 .zip 檔案封存 AWS SAM 來建立 Lambda 函數。若要使用建立 Lambda 函數 AWS SAM，您可以將 .zip 檔案儲存在 Amazon S3 儲存貯體或建置機器的本機資料夾中。如需有關如何使用將檔案上傳到 Amazon S3 儲存貯體的指示 AWS CLI，請參閱使用AWS CLI 者指南中的移動物件。

在 AWS SAM 範本中，`AWS::Serverless::Function`資源會指定您的 Lambda 函數。在本資源中設定下列屬性，以使用 .zip 封存檔建立函數：

- `PackageType`：設定為 `Zip`
- `CodeUri`-設定為函數程式碼的 Amazon S3 URI、本機資料夾的路徑或FunctionCode物件
- `Runtime`：設定為所選執行期

使用時 AWS SAM，如果您的 .zip 檔案大於 50MB，則不需要先將其上傳到 Amazon S3 儲存貯體。AWS SAM 可以從本地構建機器上的位置上傳 .zip 軟件包，最大允許大小為 250MB（解壓縮）。

若要進一步瞭解如何使用 .zip 檔案部署函數 AWS SAM，請參閱AWS SAM 開發人員指南AWS::Serverless::Function中的。

使用 .zip 文件創建和更新函數 AWS CloudFormation

您可以使 AWS CloudFormation 用 .zip 檔案封存來建立 Lambda 函數。若要使用 .zip 檔案建立 Lambda 函數，您必須先將檔案上傳至 Amazon S3 儲存貯體。如需有關如何使用將檔案上傳到 Amazon S3 儲存貯體的指示 AWS CLI，請參閱使用AWS CLI 者指南中的移動物件。

在 AWS CloudFormation 範本中，`AWS::Lambda::Function`資源會指定您的 Lambda 函數。在本資源中設定下列屬性，以使用 .zip 封存檔建立函數：

- `PackageType`：設定為 `Zip`
- `Code`：在 `S3Bucket` 和 `S3Key` 欄位中輸入 Amazon S3 儲存貯體名稱和 .zip 檔案名稱。
- `Runtime`：設定為所選執行期

AWS CloudFormation 產生的 .zip 檔案不能超過 4MB。若要進一步瞭解有關使用 .zip 檔案部署函數的更多資訊 AWS CloudFormation，請參閱使用AWS CloudFormation 者指南AWS::Lambda::Function中的。

使用容器映像部署 Node.js Lambda 函數

您可以透過三種方式為 Node.js Lambda 函數建置容器映像：

- 使用 Node.js 的 AWS 基本映像檔

  AWS 基礎映像會預先載入語言執行期、用來管理 Lambda 與函數程式碼之間互動的執行期界面用戶端，以及用於本機測試的執行期界面模擬器。
- 使用 AWS 僅限作業系統的基本影像

  AWS 僅限作業系統的基本映像檔包含 Amazon Linux 散發和執行階段介面模擬器。這些映像常用於為編譯語言 (如 Go 和 Rust) 和 Lambda 不提供基礎映像的語言或語言版本 (如 Node.js 19) 建置容器映像。您還可以使用僅限作業系統的基礎映像來實作自訂執行期。若要使映像與 Lambda 相容，您必須在映像中加入適用於 Node.js 的執行期介面用戶端。
- 使用非AWS 基本圖像

  您可以使用其他容器登錄檔中的替代基礎映像 (例如 Alpine Linux 或 Debian)。您也可以使用組織建立的自訂映像。若要使映像與 Lambda 相容，您必須在映像中加入適用於 Node.js 的執行期介面用戶端。

> **Tip**
>
> 若要縮短 Lambda 容器函數變成作用中狀態所需的時間，請參閱 Docker 文件中的使用多階段建置。若要建置有效率的容器映像，請遵循撰寫 Dockerfiles 的最佳實務。

本頁面會說明如何為 Lambda 建置、測試和部署容器映像。

主題

AWS Node.js 的基本映像檔

AWS 為 Node.js 提供下列基本影像：

標籤	執行期	作業系統	Dockerfile	棄用
20	Node.js 20	Amazon Linux 2023	用於 Node.js 20 的碼頭文件 GitHub	
18	Node.js 18	Amazon Linux 2	用於 Node.js 18 的碼頭文件 GitHub	
16	Node.js 16	Amazon Linux 2	用於 Node.js 16 的碼頭文件 GitHub	2024 年 6 月 12 日

Amazon ECR 儲存庫：gallery.ecr.aws/lambda/nodejs

Node.js 20 及更新版本的基本映像檔是以 Amazon Linux 2023 最小容器映像為基礎。早期的基本映像使用 Amazon Linux 2。與 Amazon Linux 2 相比，AL2023 具有多項優點，包括更小的部署足跡和更新版本的程式庫，如 `glibc`。

基於 Al2023 的圖像使用`microdnf`（符號鏈接為`dnf`）作為軟件包管理器而不是`yum`，這是 Amazon Linux 2 中的默認軟件包管理器。 `microdnf`是的獨立實作`dnf`。如需以 AL2023 為基礎之映像檔中包含的套件清單，請參閱比較 Amazon Linux 2023 容器映像上安裝的套件中的最小容器欄。如需有關 AL2023 和 Amazon Linux 2 之間差異的詳細資訊，請參閱 AWS 運算部落格 AWS Lambda上的介紹 Amazon Linux 2023 執行階段。

> **Note**
>
> 要在本地運行基於 AL2023 的映像，包括使用 AWS Serverless Application Model（AWS SAM），您必須使用碼頭版本 20.10.10 或更高版本。

使用 Node.js 的 AWS 基本映像檔

必要條件

若要完成本節中的步驟，您必須執行下列各項：

- AWS Command Line Interface (AWS CLI) 第二版
- 泊塢視窗 (Node.js 20 及更高版本的基本映像檔的最低版本 20.10.10)
- Node.js

從基礎映像建立映像

若要從 Node.js 的 AWS 基本映像檔建立容器映像

1. 建立專案的目錄，然後切換至該目錄。

```
mkdir example
cd example
```

2. 使用 npm 建立新 Node.js 專案。若要接受互動體驗中提供的預設選項，請按下 Enter。

```
npm init
```

3. 建立稱為 index.js 的新檔案。您可以將下列範例函數程式碼新增至檔案進行測試，或使用您自己的函數程式碼。

   Example CommonJS 處理常式

```
exports.handler = async (event) => {
    const response = {
        statusCode: 200,
        body: JSON.stringify('Hello from Lambda!'),
    };
    return response;
};
```

4. 如果您的函數依賴於除了以外的庫 AWS SDK for JavaScript，請使用 npm 將它們添加到您的包中。
5. 建立包含下列組態的新 Dockerfile。

   - 將 FROM 屬性設定為基礎映像的 URI。
   - 使用 COPY 指令{LAMBDA_TASK_ROOT}，將函數程式碼和執行階段相依性複製到 Lambda 定義的環境變數。
   - 將 CMD 引數設定為 Lambda 函數處理常式。

   Example Dockerfile

```
FROM public.ecr.aws/lambda/nodejs:20

# Copy function code
```

```
COPY index.js ${LAMBDA_TASK_ROOT}

# Set the CMD to your handler (could also be done as a parameter override outside
 of the Dockerfile)
CMD [ "index.handler" ]
```

6. 使用 docker build 命令建立 Docker 映像檔。以下範例將映像命名為 docker-image 並為其提供 test 標籤。

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> 此命令會指定 --platform linux/amd64 選項，確保無論建置機器的架構為何，您的容器都與 Lambda 執行環境相容。如果您打算使用 ARM64 指令集架構建立 Lambda 函數，務必將命令變更為改用 --platform linux/arm64 選項。

(選用) 在本機測試映像

1. 使用 docker run 命令啟動 Docker 影像。在此範例中，docker-image 為映像名稱，test 為標籤。

```
docker run --platform linux/amd64 -p 9000:8080 docker-image:test
```

此命令將映像作為容器執行，並在 localhost:9000/2015-03-31/functions/function/invocations 建立本機端點。

> **Note**
>
> 如果您為 ARM64 指令集架構建立 Docker 映像檔，請務必將--platform linux/*arm64*選項改用選項。--platform linux/*amd64*

2. 從新的終端機視窗，將事件張貼至本機端點。

Linux/macOS

在 Linux 或 macOS 中，執行下列 curl 命令：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

在中 PowerShell，執行下列`Invoke-WebRequest`命令：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

3. 取得容器 ID。

```
docker ps
```

4. 使用 docker kill 命令停止容器。在此命令中，將 3766c4ab331c 替換為上一步驟中的容器 ID。

```
docker kill 3766c4ab331c
```

部署映像

若要將映像上傳至 Amazon ECR 並建立 Lambda 函數

1. 使用 get-login-password 命令，向 Amazon ECR 登錄檔驗證 Docker CLI。
   - 將`--region`值設定為您 AWS 區域 要建立 Amazon ECR 儲存庫的位置。
   - `111122223333`用您的 AWS 帳戶 身份證替換。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 使用 create-repository 命令在 Amazon ECR 中建立儲存庫。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 儲存庫必須與 Lambda 函數位於 AWS 區域 相同。

如果成功，您將會看到以下回應：

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. 從上一步驟的輸出中複製 `repositoryUri`。
4. 執行 docker tag 命令，將 Amazon ECR 儲存庫中的本機映像標記為最新版本。在此命令中：
   - 將 `docker-image:test` 替換為 Docker 映像檔的名稱和標籤。

- 將 <ECRrepositoryUri> 替換為複製的 repositoryUri。確保在 URI 的末尾包含 :latest。

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

範例：

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. 執行 docker push 命令，將本機映像部署至 Amazon ECR 儲存庫。確保在儲存庫 URI 的末尾包含 :latest。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 建立函數的執行角色 (若您還沒有的話)。在下一個步驟中您需要角色的 Amazon Resource Name (ARN)。
7. 建立 Lambda 函數。對於 ImageUri，從之前的設定中指定儲存庫 URI。確保在 URI 的末尾包含 :latest。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> 只要映像與 Lambda 函數位於相同的區域，您就可以使用不同 AWS 帳戶中的映像檔建立函數。如需詳細資訊，請參閱 Amazon ECR 跨帳戶許可。

8. 調用函數。

```
aws lambda invoke --function-name hello-world response.json
```

您應該看到如下回應：

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 若要查看函數的輸出，請檢查 `response.json` 檔案。

若要更新函數程式碼，您必須再次建置映像、將新映像上傳到 Amazon ECR 存放庫，然後使用 update-function-code 命令將映像部署到 Lambda 函數。

Lambda 將圖像標記解析為特定的圖像摘要。這表示如果您將用於部署函數的映像標籤指向 Amazon ECR 中的新映像，Lambda 不會自動更新函數以使用新映像。若要將新映像部署到相同的 Lambda 函數，您必須使用`update-function-code`命令，即使 Amazon ECR 中的影像標籤保持不變。

透過執行期介面用戶端使用替代基礎映像

如果您使用僅限作業系統的基礎映像或替代的基礎映像，則必須在映像中加入執行期介面用戶端。執行期介面用戶端會讓您擴充 Lambda 執行階段 API，管理 Lambda 與函數程式碼之間的互動。

使用 npm 套件管理員安裝 Node.js 執行期界面用戶端：

```
npm install aws-lambda-ric
```

您也可以從下載 Node.js 執行階段介面用戶端 GitHub。執行期介面用戶端支援下列 NodeJS 版本：

- 14.x
- 16.x
- 18.x
- 20.x

下列範例會示範如何使用非AWS 基底影像建置 Node.js 的容器映像檔。範例 Dockerfile 使用 `buster` 基礎映像。Dockerfile 包含執行期界面用戶端。

必要條件

若要完成本節中的步驟，您必須執行下列各項：

- AWS Command Line Interface (AWS CLI) 第二版

- Docker
- Node.js

使用替代基礎映像建立映像

若要從非AWS 基本映像檔建立容器映像

1. 建立專案的目錄，然後切換至該目錄。

```
mkdir example
cd example
```

2. 使用 npm 建立新 Node.js 專案。若要接受互動體驗中提供的預設選項，請按下 Enter。

```
npm init
```

3. 建立稱為 index.js 的新檔案。您可以將下列範例函數程式碼新增至檔案進行測試，或使用您自己的函數程式碼。

   Example CommonJS 處理常式

```
exports.handler = async (event) => {
    const response = {
        statusCode: 200,
        body: JSON.stringify('Hello from Lambda!'),
    };
    return response;
};
```

4. 建立新的 Dockerfile。下列 Dockerfile 使用 buster 基礎映像，而非 AWS 基礎映像。Dockerfile 包含執行期介面用戶端，可讓映像與 Lambda 相容。Dockerfile 使用多階段建置。第一階段會建立組建映像，這是安裝函數相依項的標準 Node.js 環境。第二階段會建立更細微的映像，包含函數程式碼及其相依項。這會減少最終映像的大小。

   - 將 FROM 屬性設為基礎映像識別符。
   - 使用 COPY 命令複製函數程式碼和執行期相依項。
   - 將 ENTRYPOINT 設為您希望 Docker 容器在啟動時執行的模組。在此案例中，模組是執行期界面用戶端。
   - 將 CMD 引數設定為 Lambda 函數處理常式。

Example Dockerfile

```
# Define custom function directory
ARG FUNCTION_DIR="/function"

FROM node:20-buster as build-image

# Include global arg in this stage of the build
ARG FUNCTION_DIR

# Install build dependencies
RUN apt-get update && \
    apt-get install -y \
    g++ \
    make \
    cmake \
    unzip \
    libcurl4-openssl-dev

# Copy function code
RUN mkdir -p ${FUNCTION_DIR}
COPY . ${FUNCTION_DIR}

WORKDIR ${FUNCTION_DIR}

# Install Node.js dependencies
RUN npm install

# Install the runtime interface client
RUN npm install aws-lambda-ric

# Grab a fresh slim copy of the image to reduce the final size
FROM node:20-buster-slim

# Required for Node runtimes which use npm@8.6.0+ because
# by default npm writes logs under /home/.npm and Lambda fs is read-only
ENV NPM_CONFIG_CACHE=/tmp/.npm

# Include global arg in this stage of the build
ARG FUNCTION_DIR

# Set working directory to function root directory
```

```
WORKDIR ${FUNCTION_DIR}

# Copy in the built dependencies
COPY --from=build-image ${FUNCTION_DIR} ${FUNCTION_DIR}

# Set runtime interface client as default command for the container runtime
ENTRYPOINT ["/usr/local/bin/npx", "aws-lambda-ric"]
# Pass the name of the function handler as an argument to the runtime
CMD ["index.handler"]
```

5. 使用 docker build 命令建立 Docker 映像檔。以下範例將映像命名為 `docker-image` 並為其提供 `test` 標籤。

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> 此命令會指定 `--platform linux/amd64` 選項，確保無論建置機器的架構為何，您的容器都與 Lambda 執行環境相容。如果您打算使用 ARM64 指令集架構建立 Lambda 函數，務必將命令變更為改用 `--platform linux/arm64` 選項。

(選用) 在本機測試映像

使用 執行期界面模擬器 以在本機測試映像。您可以將模擬器構建到映像中，也可以使用以下步驟將其安裝在本地計算機上。

若要在本機電腦上安裝並執行執行期介面模擬器

1. 從您的項目目錄中運行以下命令以下載運行時接口仿真器（x86-64 架構）GitHub 並將其安裝在本地計算機上。

   Linux/macOS

```
mkdir -p ~/.aws-lambda-rie && \
    curl -Lo ~/.aws-lambda-rie/aws-lambda-rie https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie && \
    chmod +x ~/.aws-lambda-rie/aws-lambda-rie
```

   要安裝 arm64 模擬器，請使用以下命令替換上一個命令中的 GitHub 存儲庫 URL：

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

PowerShell

```
$dirPath = "$HOME\.aws-lambda-rie"
if (-not (Test-Path $dirPath)) {
    New-Item -Path $dirPath -ItemType Directory
}

$downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/
releases/latest/download/aws-lambda-rie"
$destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
```

若要安裝 arm64 模擬器，請將 $downloadLink 更換為下列項目：

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

2. 使用 docker run 命令啟動 Docker 影像。注意下列事項：

   - docker-image 是映像名稱，而 test 是標籤。
   - /usr/local/bin/npx aws-lambda-ric index.handler 是 Dockerfile 中的 ENTRYPOINT，後面接著 CMD。

   Linux/macOS

```
docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p
 9000:8080 \
    --entrypoint /aws-lambda/aws-lambda-rie \
    docker-image:test \
        /usr/local/bin/npx aws-lambda-ric index.handler
```

   PowerShell

```
docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p
 9000:8080 `
--entrypoint /aws-lambda/aws-lambda-rie `
```

```
docker-image:test `
    /usr/local/bin/npx aws-lambda-ric index.handler
```

此命令將映像作為容器執行，並在 localhost:9000/2015-03-31/functions/function/invocations 建立本機端點。

> **Note**
>
> 如果您為 ARM64 指令集架構建立 Docker 映像檔，請務必將--platform linux/*arm64*選項改用選項。--platform linux/*amd64*

3. 將事件張貼至本機端點。

Linux/macOS

在 Linux 或 macOS 中，執行下列 curl 命令：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

在中 PowerShell，執行下列Invoke-WebRequest命令：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

4. 取得容器 ID。

```
docker ps
```

5. 使用 docker kill 命令停止容器。在此命令中，將 3766c4ab331c 替換為上一步驟中的容器 ID。

```
docker kill 3766c4ab331c
```

部署映像

若要將映像上傳至 Amazon ECR 並建立 Lambda 函數

1. 使用 get-login-password 命令，向 Amazon ECR 登錄檔驗證 Docker CLI。
   - 將--region值設定為您 AWS 區域 要建立 Amazon ECR 儲存庫的位置。
   - 111122223333用您的 AWS 帳戶 身份證替換。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 使用 create-repository 命令在 Amazon ECR 中建立儲存庫。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 儲存庫必須與 Lambda 函數位於 AWS 區域 相同。

如果成功，您將會看到以下回應：

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
```

```
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. 從上一步驟的輸出中複製 `repositoryUri`。
4. 執行 docker tag 命令，將 Amazon ECR 儲存庫中的本機映像標記為最新版本。在此命令中：
   - 將 `docker-image:test` 替換為 Docker 映像檔的名稱和標籤。
   - 將 <ECRrepositoryUri> 替換為複製的 `repositoryUri`。確保在 URI 的末尾包含 `:latest`。

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

範例：

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world:latest
```

5. 執行 docker push 命令，將本機映像部署至 Amazon ECR 儲存庫。確保在儲存庫 URI 的末尾包含 `:latest`。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 建立函數的執行角色 (若您還沒有的話)。在下一個步驟中您需要角色的 Amazon Resource Name (ARN)。
7. 建立 Lambda 函數。對於 `ImageUri`，從之前的設定中指定儲存庫 URI。確保在 URI 的末尾包含 `:latest`。

```
aws lambda create-function \
  --function-name hello-world \
```

```
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> 只要映像與 Lambda 函數位於相同的區域，您就可以使用不同 AWS 帳戶中的映像檔建立函數。如需詳細資訊，請參閱 Amazon ECR 跨帳戶許可。

8. 調用函數。

```
aws lambda invoke --function-name hello-world response.json
```

您應該看到如下回應：

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 若要查看函數的輸出，請檢查 `response.json` 檔案。

若要更新函數程式碼，您必須再次建置映像、將新映像上傳到 Amazon ECR 存放庫，然後使用 update-function-code 命令將映像部署到 Lambda 函數。

Lambda 將圖像標記解析為特定的圖像摘要。這表示如果您將用於部署函數的映像標籤指向 Amazon ECR 中的新映像，Lambda 不會自動更新函數以使用新映像。若要將新映像部署到相同的 Lambda 函數，您必須使用`update-function-code`命令，即使 Amazon ECR 中的影像標籤保持不變。

Node.js 中的 AWS Lambda 內容物件

當 Lambda 執行您的函數時，它會將內容物件傳遞至處理常式。此物件提供的方法和各項屬性提供了有關調用、函式以及執行環境的資訊。

內容方法

- `getRemainingTimeInMillis()` - 傳回執行逾時前剩餘的毫秒數。

內容屬性

- `functionName` – Lambda 函數的名稱。
- `functionVersion` – 函數的版本。
- `invokedFunctionArn` - 用於調用此函數的 Amazon Resource Name (ARN)。指出調用者是否指定版本號或別名。
- `memoryLimitInMB` - 分配給函數的記憶體數量。
- `awsRequestId` - 調用請求的識別符。
- `logGroupName` - 函數的日誌群組。
- `logStreamName` - 函數執行個體的記錄串流。
- `identity` - (行動應用程式) 已授權請求的 Amazon Cognito 身分的相關資訊。
  - `cognitoIdentityId` - 已驗證的 Amazon Cognito 身分。
  - `cognitoIdentityPoolId` - 授權調用的 Amazon Cognito 身分集區。
- `clientContext` - (行動應用程式) 用戶端應用程式提供給 Lambda 的用戶端內容。
  - `client.installation_id`
  - `client.app_title`
  - `client.app_version_name`
  - `client.app_version_code`
  - `client.app_package_name`
  - `env.platform_version`
  - `env.platform`
  - `env.make`
  - `env.model`
  - `env.locale`

  - `Custom` - 用戶端應用程式所設定的自訂值。
- `callbackWaitsForEmptyEventLoop` - 設為 false 將會在回呼執行時立即傳送回應，而不會等待 Node.js 事件迴圈成空白。如果此為 false，任何未完成的事件都會於下次調用時繼續執行。

以下範例函式紀錄內文資訊和傳回日誌的位置。

Example index.js 檔案

```
exports.handler = async function(event, context) {
  console.log('Remaining time: ', context.getRemainingTimeInMillis())
  console.log('Function name: ', context.functionName)
  return context.logStreamName
}
```

AWS Lambda Node.js 中的函數日誌記錄

AWS Lambda 代表您自動監控 Lambda 函數，並將日誌傳送到 Amazon CloudWatch。您的 Lambda 函數隨附一個 CloudWatch 日誌記錄群組和函數每個執行個體的日誌串流。Lambda 執行期環境會將每次調用的詳細資訊傳送至日誌串流，並且轉傳來自函數程式碼的日誌及其他輸出。如需詳細資訊，請參閱 使用 Amazon CloudWatch 日誌 AWS Lambda。

本頁說明如何從 Lambda 函數的程式碼產生記錄輸出，或使用 Lambda 主控台或主控台存取 CloudWatch 日誌。 AWS Command Line Interface

章節

建立傳回日誌的函數

若要由您的函式程式碼輸出日誌，您可以在主控台物件上使用方法，或任何能寫入 `stdout` 或 `stderr` 的記錄程式庫。下列範例會記錄環境變數和事件物件的值。

Example index.js 檔案 - 記錄

```
exports.handler = async function(event, context) {
  console.log("ENVIRONMENT VARIABLES\n" + JSON.stringify(process.env, null, 2))
  console.info("EVENT\n" + JSON.stringify(event, null, 2))
  console.warn("Event not processed.")
  return context.logStreamName
}
```

Example 記錄格式

```
START RequestId: c793869b-ee49-115b-a5b6-4fd21e8dedac Version: $LATEST
2019-06-07T19:11:20.562Z c793869b-ee49-115b-a5b6-4fd21e8dedac INFO ENVIRONMENT
 VARIABLES
```

```
{
  "AWS_LAMBDA_FUNCTION_VERSION": "$LATEST",
  "AWS_LAMBDA_LOG_GROUP_NAME": "/aws/lambda/my-function",
  "AWS_LAMBDA_LOG_STREAM_NAME": "2019/06/07/[$LATEST]e6f4a0c4241adcd70c262d34c0bbc85c",
  "AWS_EXECUTION_ENV": "AWS_Lambda_nodejs12.x",
  "AWS_LAMBDA_FUNCTION_NAME": "my-function",
  "PATH": "/var/lang/bin:/usr/local/bin:/usr/bin/:/bin:/opt/bin",
  "NODE_PATH": "/opt/nodejs/node10/node_modules:/opt/nodejs/node_modules:/var/runtime/
node_modules",
  ...
}
2019-06-07T19:11:20.563Z c793869b-ee49-115b-a5b6-4fd21e8dedac INFO EVENT
{
  "key": "value"
}
2019-06-07T19:11:20.564Z c793869b-ee49-115b-a5b6-4fd21e8dedac WARN Event not processed.
END RequestId: c793869b-ee49-115b-a5b6-4fd21e8dedac
REPORT RequestId: c793869b-ee49-115b-a5b6-4fd21e8dedac Duration: 128.83 ms Billed
 Duration: 200 ms Memory Size: 128 MB Max Memory Used: 74 MB Init Duration: 166.62 ms
 XRAY TraceId: 1-5d9d007f-0a8c7fd02xmpl480aed55ef0 SegmentId: 3d752xmpl1bbe37e Sampled:
 true
```

Node.js 執行時間會記錄每次呼叫的 START、END 和 REPORT 行。它會將時間戳記、請求 ID 和日誌層級新增到函式所記錄的每個項目。報告明細行提供下列詳細資訊。

REPORT 行資料欄位

- RequestId— 呼叫的唯一要求 ID。
- 持續時間 - 函數的處理常式方法處理事件所花費的時間量。
- 計費持續時間 - 調用的計費時間量。
- 記憶體大小 - 分配給函數的記憶體數量。
- 使用的記憶體上限 - 函數所使用的記憶體數量。
- 初始化持續時間 - 對於第一個提供的請求，這是執行期載入函數並在處理常式方法之外執行程式碼所花費的時間量。
- XRAY TraceId — 針對追蹤的要求，則為AWS X-Ray 追蹤識別碼。
- SegmentId— 針對追蹤的請求，X-Ray 區段 ID。
- 已取樣 - 對於追蹤的請求，這是取樣結果。

您可以在 Lambda 主控台、 CloudWatch 記錄主控台或從命令列檢視記錄。

搭配 Node.js 使用 Lambda 進階日誌控制項

為了讓您更妥善地控制擷取、處理和使用函數日誌的方式，您可以針對支援的 Node.js 執行期設定下列記錄選項：

- 日誌格式 - 在純文字和結構化 JSON 格式之間為您的日誌進行選擇
- 日誌級別-對於 JSON 格式的日誌，選擇 Lambda 發送到 Amazon 的日誌詳細級別 CloudWatch，例如錯誤，調試或信息
- 日誌組-選擇您的功能發送日誌的日誌組 CloudWatch

如需這些日誌選項的詳細資訊，以及如何設定函數以使用這些選項的說明，請參閱 the section called “設定 Lambda 函數的進階日誌控制項”。

若要使用日誌格式和日誌層級選項與 Node.js Lambda 函數搭配使用，請參閱以下各章節中的指引。

搭配 Node.js 使用結構化的 JSON 日誌

如果您為函數的記錄格式選取 JSON，Lambda 會使用、、`console.trace`、`console.debug`、`console.log`、`console.info`和的主控台方法將記錄輸出傳送`console.warn`至 CloudWatch 結構化 JSON。`console.error`每個 JSON 日誌物件都包含至少四個鍵值對，其中包含下列索引鍵：

- `"timestamp"` - 產生日誌訊息的時間
- `"level"` - 指派給訊息的日誌層級
- `"message"` - 日誌訊息的內容
- `"requestId"` - 進行調用的唯一請求 ID。

依據您的函數使用的記錄方法，此 JSON 物件還可能包含其他鍵值對。例如，如果您的函數使用 `console` 方法來記錄使用多個引數的錯誤物件，JSON 物件將包含具有索引鍵 `errorMessage`、`errorType` 和 `stackTrace` 的額外鍵值對。

如果您的代碼已經使用另一個日誌庫（例如 Powertools 的）來生成 JSON 結構化日誌，則不需要進行任何更改。 AWS Lambda Lambda 不會對任何已經進行 JSON 編碼的日誌進行雙重編碼，因此您函數的應用程式日誌將繼續像以前一樣被擷取。

如需有關使用 Powertools AWS Lambda 記錄封裝以在 Node.js 執行階段中建立 JSON 結構化記錄檔的詳細資訊，請參閱the section called “日誌”。

JSON 格式日誌輸出範例

下列範例顯示當您將函數的記錄格式設定為 JSON 時，如何在 CloudWatch 記錄檔中擷取使用具有單引數和多個引數的console方法產生的各種記錄輸出。

第一個範例使用 console.error 方法輸出簡單字串。

Example Node.js 日誌程式碼

```
export const handler = async (event) => {
  console.error("This is a warning message");
  ...
}
```

Example JSON 日誌記錄

```
{
    "timestamp":"2023-11-01T00:21:51.358Z",
    "level":"ERROR",
    "message":"This is a warning message",
    "requestId":"93f25699-2cbf-4976-8f94-336a0aa98c6f"
}
```

您還可以透過 console 方法使用單個或多個引數輸出結構更複雜的日誌訊息。在下一個範例中，您將使用 console.log 與單個引數輸出兩個鍵值對。請注意，Lambda 傳送至 CloudWatch 記錄的 JSON 物件中的"message"欄位並未設定字串化。

Example Node.js 日誌程式碼

```
export const handler = async (event) => {
  console.log({data: 12.3, flag: false});
  ...
}
```

Example JSON 日誌記錄

```
{
    "timestamp": "2023-12-08T23:21:04.664Z",
    "level": "INFO",
    "requestId": "405a4537-9226-4216-ac59-64381ec8654a",
    "message": {
        "data": 12.3,
```

```
        "flag": false
    }
}
```

在下一個範例中，您要再次使用 console.log 方法建立日誌輸出。這一次，該方法使用兩個引數、一個包含兩個鍵值對的映射和一個識別字串。請注意，在這種情況下，由於您提供了兩個引數，Lambda 會對 "message" 欄位進行字串化。

Example Node.js 日誌程式碼

```
export const handler = async (event) => {
  console.log('Some object - ', {data: 12.3, flag: false});
  ...
}
```

Example JSON 日誌記錄

```
{
    "timestamp": "2023-12-08T23:21:04.664Z",
    "level": "INFO",
    "requestId": "405a4537-9226-4216-ac59-64381ec8654a",
    "message": "Some object -  { data: 12.3, flag: false }"
}
```

Lambda 會指派使用 console.log 日誌層級 INFO 產生的輸出。

最後一個範例顯示如何使用這些console方法將錯誤物件輸出至 CloudWatch 記錄檔。請注意，當您使用多個引數記錄錯誤物件時，Lambda 會新增 errorMessage、errorType 和 stackTrace 欄位至日誌輸出。

Example Node.js 日誌程式碼

```
export const handler = async (event) => {
  let e1 = new ReferenceError("some reference error");
  let e2 = new SyntaxError("some syntax error");
  console.log(e1);
  console.log("errors logged - ", e1, e2);
};
```

Example JSON 日誌記錄

```
{
```

```
    "timestamp": "2023-12-08T23:21:04.632Z",
    "level": "INFO",
    "requestId": "405a4537-9226-4216-ac59-64381ec8654a",
    "message": {
        "errorType": "ReferenceError",
        "errorMessage": "some reference error",
        "stackTrace": [
            "ReferenceError: some reference error",
            "    at Runtime.handler (file:///var/task/index.mjs:3:12)",
            "    at Runtime.handleOnceNonStreaming (file:///var/runtime/
index.mjs:1173:29)"
        ]
    }
}

{
    "timestamp": "2023-12-08T23:21:04.646Z",
    "level": "INFO",
    "requestId": "405a4537-9226-4216-ac59-64381ec8654a",
    "message": "errors logged -  ReferenceError: some reference error
\n    at Runtime.handler (file:///var/task/index.mjs:3:12)\n    at
 Runtime.handleOnceNonStreaming
    (file:///var/runtime/index.mjs:1173:29) SyntaxError: some syntax
 error\n    at Runtime.handler (file:///var/task/index.mjs:4:12)\n    at
 Runtime.handleOnceNonStreaming
    (file:///var/runtime/index.mjs:1173:29)",
    "errorType": "ReferenceError",
    "errorMessage": "some reference error",
    "stackTrace": [
        "ReferenceError: some reference error",
        "    at Runtime.handler (file:///var/task/index.mjs:3:12)",
        "    at Runtime.handleOnceNonStreaming (file:///var/runtime/index.mjs:1173:29)"
    ]
}
```

記錄多個錯誤類型時，會從提供給 `console` 方法的第一個錯誤類型中擷取額外欄位 `errorMessage`、`errorType` 和 `stackTrace`。

搭配結構化 JSON 日誌使用內嵌指標格式 (EMF) 用戶端程式庫

AWS 提供 Node.js 的開放原始碼用戶端程式庫，您可以使用這些程式庫建立內嵌度量格式 (EMF) 記錄。如果您有使用這些庫的現有函數，並且將函數的日誌格式更改為 JSON，則 CloudWatch 可能不再識別代碼發出的指標。

如果您的代碼當前直接使用`console.log`或使用 Powertools to AWS Lambda (TypeScript) 發出 EMF 日誌，如果 CloudWatch 將函數的日誌格式更改為 JSON，也將無法解析這些日誌。

> **Important**
>
> 為了確保您的功能的 EMF 日誌繼續被正確解析 CloudWatch，請將您的 EMF 和 Powertools 的 AWS Lambda庫更新為最新版本。如果切換到 JSON 日誌格式，我們也建議您進行測試，以確保與函數的內嵌指標相容。如果您的程式碼直接使用 `console.log` 發出 EMF 記錄，請變更您的程式碼以將這些指標直接輸出至 `stdout`，如下列程式碼範例所示。

Example 發出內嵌指標至 **`stdout`** 的程式碼

```
process.stdout.write(JSON.stringify(
    {
        "_aws": {
            "Timestamp": Date.now(),
            "CloudWatchMetrics": [{
                "Namespace": "lambda-function-metrics",
                "Dimensions": [["functionVersion"]],
                "Metrics": [{
                    "Name": "time",
                    "Unit": "Milliseconds",
                    "StorageResolution": 60
                }]
            }]
        },
        "functionVersion": "$LATEST",
        "time": 100,
        "requestId": context.awsRequestId
    }
) + "\n")
```

搭配 Node.js 使用日誌層級篩選

AWS Lambda 為了根據日誌級別過濾應用程序日誌，您的函數必須使用 JSON 格式的日誌。您可以透過兩種方式達成此操作：

- 使用標準主控台方法建立記錄輸出，並將函數設定為使用 JSON 記錄格式。 AWS Lambda 然後使用中描述的 JSON 對象中的「級別」鍵值對過濾日誌輸出the section called “搭配 Node.js 使用結構化

的 JSON 日誌”。若要瞭解如何設定函數的日誌格式，請參閱 the section called “設定 Lambda 函數的進階日誌控制項”。

- 使用其他日誌程式庫或方法，在您的程式碼中建立 JSON 結構化日誌，其中包含定義日誌輸出層級的「層級」索引鍵值組。例如，您可以使用 Powertools 從您的 AWS Lambda 代碼生成 JSON 結構化日誌輸出。請參閱 the section called “日誌” 以瞭解有關在 Node.js 執行期使用 Powertools 的更多資訊。

  若要讓 Lambda 篩選函數的日誌，您還必須在 JSON 日誌輸出中包含 `"timestamp"` 索引鍵值組。必須以有效的 RFC 3339 時間戳記格式指定時間。如果您沒有提供有效的時間戳記，Lambda 會為日誌指派層級 INFO，並為您新增時間戳記。

當您將函數設定為使用記錄層級篩選時，您可以從下列選項中選取要傳送 AWS Lambda 至 CloudWatch 記錄檔的記錄層級：

日誌層級	標準用量
TRACE (大多數詳細資訊)	用於追蹤程式碼執行路徑的最精細資訊
DEBUG	系統偵錯的詳細資訊
INFO	記錄函數正常操作的訊息
WARN	有關可能導致未解決意外行為的潛在錯誤的消息
ERROR	有關阻止程式碼按預期執行的問題的訊息
FATAL (最少詳細資訊)	有關導致應用程式停止運作的嚴重錯誤訊息

Lambda 會將所選層級且較低層級的記錄傳送至 CloudWatch。例如，如果您設定 WARN 的日誌層級，Lambda 會傳送相對應於 WARN、ERROR 和 FATAL 層級的日誌檔。

使用 Lambda 主控台

您可以在調用 Lambda 函數之後，使用 Lambda 主控台來檢視日誌輸出。

如果可以從內嵌程式碼編輯器測試您的程式碼，您會在執行結果中找到日誌。使用主控台測試功能以調用函數時，您會在詳細資訊區段找到日誌輸出。

使用控 CloudWatch 制台

您可以使用 Amazon 主 CloudWatch 控台來檢視所有 Lambda 函數叫用的日誌。

在 CloudWatch 主控台上檢視記錄檔

1. 在主控台上開啟 [記錄群組] 頁 CloudWatch 面。
2. 選擇您的函數的日誌群組 (/aws/lambda/***your-function-name***)。
3. 選擇日誌串流

每個日誌串流都會對應至函式的執行個體。當您更新 Lambda 函數，以及建立額外執行個體以處理多個並行調用時，便會出現日誌串流。要查找特定調用的日誌，我們建議使用檢測您的函數。 AWS X-Ray X-Ray 會在追蹤內記錄有關請求和日誌串流的詳細資訊。

使用 AWS Command Line Interface （ AWS CLI ）

這 AWS CLI 是一種開放原始碼工具，可讓您使用命令列殼層中的命令與 AWS 服務互動。若要完成本節中的步驟，您必須執行下列各項：

- AWS Command Line Interface (AWS CLI) 第二版
- AWS CLI -快速配置 `aws configure`

您可以透過 AWS CLI，使用 `--log-type` 命令選項來擷取要調用的日誌。其回應將包含 `LogResult` 欄位，內含該次調用的 base64 編碼日誌 (最大達 4 KB)。

Example 擷取日誌 ID

下列範例顯示如何從名稱為 `my-function` 的函數的 `LogResult` 欄位來擷取日誌 ID。

```
aws lambda invoke --function-name my-function out --log-type Tail
```

您應該會看到下列輸出：

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
```

```
}
```

Example 解碼日誌

在相同的命令提示中，使用 base64 公用程式來解碼日誌。下列範例顯示如何擷取 my-function 的 base64 編碼日誌。

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

如果您使用的是 AWS CLI 版本 2，則需要此cli-binary-format選項。若要讓此成為預設的設定，請執行 aws configure set cli-binary-format raw-in-base64-out。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

您應該會看到下列輸出：

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

該 base64 公用程式可在 Linux、macOS 和 Ubuntu on Windows 上使用。macOS 使用者可能需要使用 base64 -D。

Example get-logs.sh 指令碼

在相同的命令提示中，使用下列指令碼下載最後五個日誌事件。該指令碼使用 sed 以從輸出檔案移除引述，並休眠 15 秒以使日誌可供使用。輸出包括來自 Lambda 的回應以及來自 get-log-events 命令的輸出。

複製下列程式碼範例的內容，並將您的 Lambda 專案目錄儲存為 get-logs.sh。

如果您使用的是 AWS CLI 版本 2，則需要此cli-binary-format選項。若要讓此成為預設的設定，請執行 aws configure set cli-binary-format raw-in-base64-out。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

```
#!/bin/bash
```

```
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS 和 Linux (僅限)

在相同的命令提示中，macOS 和 Linux 使用者可能需要執行下列命令，以確保指令碼可執行。

```
chmod -R 755 get-logs.sh
```

Example 擷取最後五個記錄事件

在相同的命令提示中，執行下列指令碼以取得最後五個日誌事件。

```
./get-logs.sh
```

您應該會看到下列輸出：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
```

```
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```

刪除日誌

當您刪除函數時，不會自動刪除日誌群組。若要避免無限期地儲存日誌，請刪除日誌群組，或設定保留期間，系統會在該時間之後自動刪除日誌。

在中檢測 Node.js 程式碼 AWS Lambda

Lambda 與 AWS X-Ray 整合，可協助您追蹤、偵錯和最佳化 Lambda 應用程式。您可以使用 X-Ray 來追蹤請求，因為它會周遊您應用程式中的資源，其中可能包含 Lambda 函數和其他 AWS 服務。

若要將追蹤資料傳送至 X-Ray，您可以使用以下兩個 SDK 庫之一：

- AWS 適用於 OpenTelemetry (ADOT) 的發行版 — 安全、可生產就緒且 AWS支援的 (OTel) SDK 發行 OpenTelemetry 版。
- 適用於 Node.js 的 AWS X-Ray SDK – 用於生成追蹤資料並將其傳送至 X-Ray 的 SDK。

每個 SDK 均提供將遙測資料傳送至 X-Ray 服務的方法。然後，您可以使用 X-Ray 來檢視、篩選應用程式的效能指標並獲得洞察，從而識別問題和進行最佳化的機會。

> **Important**
>
> 用於 AWS Lambda SDK 的 X-Ray 和 Powertools 是由提供的緊密集成的儀表解決方案的一部分。 AWS ADOT Lambda Layers 是用於追蹤檢測之業界通用標準的一部分，這類檢測一般會收集更多資料，但可能不適用於所有使用案例。您可以使用任一解決方案在 X-Ray 中實作 end-to-end 追蹤。若要深入了解如何在兩者之間做選擇，請參閱在 AWS Distro for OpenTelemetry 和 X-Ray SDK 之間進行選擇。

章節

使用 ADOT 來檢測您的 Node.js 函數

ADOT 提供全受管 Lambda 層，包含使用 OTel SDK 收集遙測資料所需的一切內容。透過取用此層，您可以檢測 Lambda 函數，而無需修改任何函數程式碼。您還可以將層設定為對 OTel 進行自訂初始化。如需詳細資訊，請參閱 ADOT 文件中的針對 Lambda 上的 ADOT 收集器進行自訂組態設定。

針對 Python 執行階段，您可以新增適用於 ADOT Javascript 的AWS 受管 Lambda 層來自動檢測您的函數。有關如何添加此層的詳細說明，請參閱 AWS ADOT 文檔 JavaScript中的 OpenTelemetry Lambda Support 發行版。

使用 X-Ray SDK 來檢測 Node.js 函數

若要記錄 Lambda 函數對應用程式中其他資源所進行之呼叫的詳細資料，您也可以使用 適用於 Node.js 的 AWS X-Ray SDK。若要取得開發套件，請將 `aws-xray-sdk-core` 套件新增至應用程式的相依性。

Example blank-nodejs/package.json

```
{
  "name": "blank-nodejs",
  "version": "1.0.0",
  "private": true,
  "devDependencies": {
    "jest": "29.7.0"
  },
  "dependencies": {
    "@aws-sdk/client-lambda": "3.345.0",
    "aws-xray-sdk-core": "3.5.3"
  },
  "scripts": {
    "test": "jest"
  }
}
```

要在 AWS SDK for JavaScript v3 中檢測 AWS SDK 客戶端，請使用該`captureAWSv3Client`方法包裝客戶端實例。

Example blank-nodejs/function/index.js — 追蹤 AWS SDK 用戶端

```
const AWSXRay = require('aws-xray-sdk-core');
```

```
const { LambdaClient, GetAccountSettingsCommand } = require('@aws-sdk/client-lambda');

// Create client outside of handler to reuse
const lambda = AWSXRay.captureAWSv3Client(new LambdaClient());

// Handler
exports.handler = async function(event, context) {
    event.Records.forEach(record => {
  ...
```

Lambda 執行階段會設定一些環境變數，以配置 X-Ray SDK。例如，Lambda 會將 AWS_XRAY_CONTEXT_MISSING 設定為 LOG_ERROR，以避免從 X-Ray SDK 中擲回執行階段錯誤。若要設置自訂內容遺失策略，請覆寫函式組態中的環境變數以使其不要有值，然後您可以透過程式設計方式設置內容遺失策略。

Example 初始化程式碼範例

```
const AWSXRay = require('aws-xray-sdk-core');

// Configure the context missing strategy to do nothing
AWSXRay.setContextMissingStrategy(() => {});
```

如需詳細資訊，請參閱 the section called “設定環境變數”。

新增正確的依賴項並進行必要的程式碼變更後，請透過 Lambda 主控台或 API 在函數的組態中啟用追蹤。

透過 Lambda 主控台來啟用追蹤

若要使用控制台在 Lambda 函數上切換主動追蹤，請按照下列步驟操作：

開啟主動追蹤

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇 組態 ，然後選擇 監控和操作工具 。
4. 選擇 編輯 。
5. 在 X-Ray 下，打開 主動追蹤 。
6. 選擇 儲存 。

透過 Lambda API 啟用追蹤

使用 AWS CLI 或 AWS SDK 在 Lambda 函數上設定追蹤，並使用下列 API 作業：

- UpdateFunction配置
- GetFunction配置
- CreateFunction

下列範例 AWS CLI 命令可在名為 my- function 的函式上啟用主動追蹤。

```
aws lambda update-function-configuration \
--function-name my-function \
--tracing-config Mode=Active
```

追蹤模式是您發布函數版本時版本特定組態的一部分。您無法變更已發佈版本上的追蹤模式。

使用啟動追蹤 AWS CloudFormation

若要啟動 AWS CloudFormation 範本中的`AWS::Lambda::Function`資源追蹤，請使用`TracingConfig`屬性。

Example function-inline.yml - 追蹤組態

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

對於 AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` 資源，請使用`Tracing`屬性。

Example template.yml - 追蹤組態

```
Resources:
  function:
```

```
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

解讀 X-Ray 追蹤

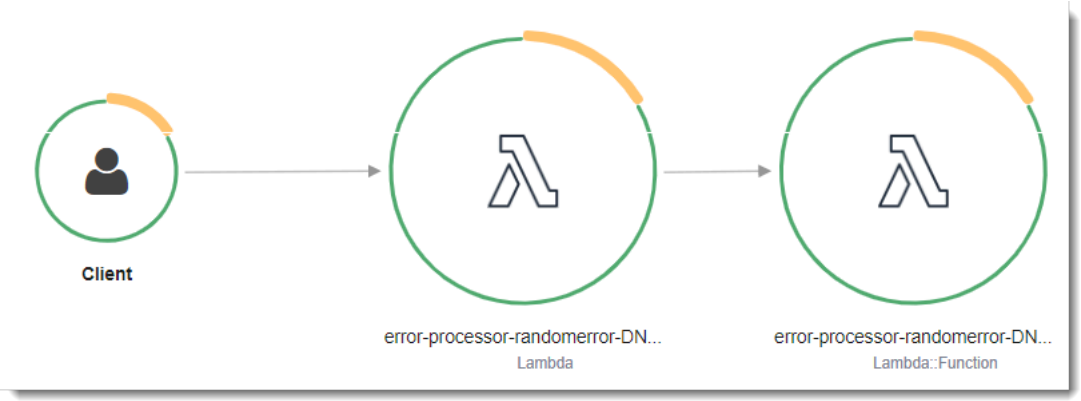
您的函數需要將追蹤資料上傳至 X-Ray 的許可。當您在 Lambda 主控台中啟用追蹤時，Lambda 會將必要的許可新增至函數的執行角色。否則，請將AWSXRayDaemonWriteAccess原則新增至執行角色。

設定主動追蹤之後，您可以透過應用程式來觀察特定請求。X-Ray 服務圖顯示了有關應用程式及其所有元件的資訊。下圖演示了具有兩個功能的應用程序。主要函式會處理事件，有時會傳回錯誤。頂部的第二個函數處理出現在第一個日誌組中的錯誤，並使用 AWS SDK 調用 X-Ray，Amazon 簡單存儲服務（Amazon S3）和亞馬遜 CloudWatch 日誌。

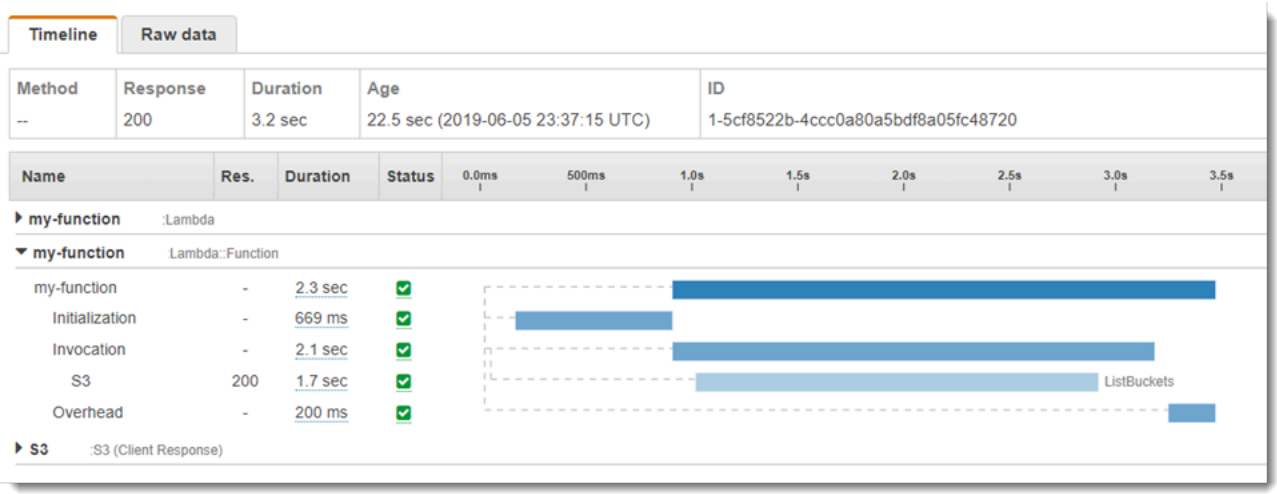
X-Ray 無法追蹤應用程式的所有請求。X-Ray 會套用取樣演算法以確保追蹤的效率，同時仍提供所有請求的代表範本。取樣率為每秒 1 次請求和 5% 的額外請求。

> **Note**
>
> 您無法針對函數設定 X-Ray 取樣率。

在 X-Ray 中，追蹤會記錄一或多個服務所處理之要求的相關資訊。Lambda 會記錄每個追蹤 2 個區段，在服務圖表上建立兩個節點。下列影像會強調顯示這兩個節點：



左側第一個節點代表接收調用請求的 Lambda 服務。第二個節點代表您特定的 Lambda 函數。下列範例顯示了具有這 2 個區段的追蹤。兩者都被命名為我的函數，但一個具有的起源`AWS::Lambda`，另一個具有的`AWS::Lambda::Function`起源。如果`AWS::Lambda`區段顯示錯誤，表示 Lambda 服務發生問題。如果`AWS::Lambda::Function`區段顯示錯誤，表示您的函數發生問題。



此範例會展開區`AWS::Lambda::Function`段，以顯示其三個子區段：

- 初始化 - 表示載入函數和執行初始化程式碼所花費的時間。只有函數的每個執行個體所處理的第一個事件會顯示此子區段。
- 調用 – 表示執行處理常式程式碼所花費的時間。
- 額外負荷 - 表示 Lambda 執行期為做好準備以處理下一個事件所花費的時間。

您也可以檢測 HTTP 用戶端、記錄 SQL 查詢，以及建立具有註釋和中繼資料的自訂子區段。如需詳細資訊，請參閱《AWS X-Ray 開發人員指南》中的 適用於 Node.js 的 AWS X-Ray SDK 許可。

> **定價**
>
> 作為免費方案的一部分，您可以每月免費使用 X-Ray 追蹤，最多達到一定限制。 AWS 達到閾值後，X-Ray 會收取追蹤儲存及擷取的費用。如需詳細資訊，請參閱 AWS X-Ray 定價。

將執行時間相依項存放存在層中 (X-Ray SDK)

如果您使用 X-Ray SDK 來檢測 AWS SDK 用戶端您的函數程式碼，您的部署套件可能會變得相當大。為了避免每次更新函數程式碼時上傳執行時間相依性，請將 X-Ray SDK 封裝在一個 Lambda 層中。

以下範例會顯示存放 適用於 Node.js 的 AWS X-Ray SDK的 `AWS::Serverless::LayerVersion` 資源。

Example template.yml - 相依性層

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: function/.
      Tracing: Active
      Layers:
        - !Ref libs
      ...
  libs:
    Type: AWS::Serverless::LayerVersion
    Properties:
      LayerName: blank-nodejs-lib
      Description: Dependencies for the blank sample app.
      ContentUri: lib/.
      CompatibleRuntimes:
        - nodejs16.x
```

透過此組態，您只有在變更執行時間相依性時才會更新程式庫層。由於函數部署套件僅含有您的程式碼，因此有助於減少上傳時間。

為相依性建立圖層需要建置變更，才能在部署之前產生圖層封存。如需工作範例，請參閱 blank-nodejs 範例應用程式。

使用建置 Lambda 函數 TypeScript

您可以使用 Node.js 執行階段在中執行 TypeScript 程式碼 AWS Lambda。因為 Node.js 本機不會執行 TypeScript 程式碼，所以您必須先將 TypeScript 程式碼轉譯成. JavaScript 然後，使用這些 JavaScript 檔案將函數程式碼部署到 Lambda。您的程式碼在包含 AWS SDK 的環境中執行 JavaScript，其中包含您管理的 AWS Identity and Access Management (IAM) 角色的登入資料。若要進一步瞭解 Node.js 執行階段隨附的 SDK 版本，請參閱the section called “包含執行階段的 SDK 版本”。

Lambda 支援以下 Node.js 執行期。

Node.js

名稱	識別符	作業系統	取代日期	封鎖函數建立	封鎖函數更新
Node.js 20	`nodejs20.x`	Amazon Linux 2023			
Node.js 18	`nodejs18.x`	Amazon Linux 2			
Node.js 16	`nodejs16.x`	Amazon Linux 2	2024 年 6 月 12 日	2025年2月28日	2025年3月31日

主題

建立 TypeScript 開發環境

使用本機整合式開發環境 (IDE)、文字編輯器，或AWS Cloud9撰寫 TypeScript 函數程式碼。您無法在 Lambda 主控台上建立 TypeScript 程式碼。

為了轉換你的 TypeScript 代碼，設置一個編譯器，如 esbuild 或微軟的 TypeScript 編譯器（`tsc`），它與發行版捆綁在TypeScript 一起。您可以使用 AWS Serverless Application Model (AWS SAM) 或簡化程式碼的AWS Cloud Development Kit (AWS CDK)建置和部署 TypeScript 程式碼。這兩種工具都使用 esbuild 將 TypeScript 代碼轉換為. JavaScript

使用 esbuild 時，請考慮下列事項：

- 有幾個TypeScript 警告。
- 您必須設定 TypeScript 轉置設定，以符合您計劃使用的 Node.js 執行階段。如需詳細資訊，請參閱 esbuild 文件中的目標。如需示範如何鎖定 Lambda 支援的特定 Node.js 版本的 tsconfig.json 檔案範例，請參閱存放庫。TypeScript GitHub
- esbuild 不執行類型檢查。若要檢查類型，請使用 `tsc` 編譯器。執行 `tsc -noEmit` 或將 `"noEmit"` 參數新增至 tsconfig.json 檔案，如下列範例所示。這將配置`tsc`為不發出 JavaScript 文件。檢查類型後，請使用 esbuild 將 TypeScript 文件轉換為。 JavaScript

Example tsconfig.json

```
{
 "compilerOptions": {
   "target": "es2020",
   "strict": true,
   "preserveConstEnums": true,
   "noEmit": true,
   "sourceMap": false,
   "module":"commonjs",
   "moduleResolution":"node",
   "esModuleInterop": true,
   "skipLibCheck": true,
   "forceConsistentCasingInFileNames": true,
   "isolatedModules": true,
 },
 "exclude": ["node_modules", "**/*.test.ts"]
}
```

定義 Lambda 函數處理常式 TypeScript

Lambda 函數處理常式是您的函數程式碼中處理事件的方法。當有人呼叫您的函數時，Lambda 會執行處理常式方法。函數會執行，直到處理常式傳回回應、結束或逾時為止。

Example TypeScript 處理器

此範例函式記錄事件物件的內容並傳回日誌的位置。注意下列事項：

- 在 Lambda 函數中使用這段程式碼之前，您必須先新增 @types/aws-lambda 套件當作開發相依項。此套件包含 Lambda 的類型定義。安裝 `@types/aws-lambda` 後，`import` 陳述式 (`import ... from 'aws-lambda'`) 會匯入類型定義。不會匯入 `aws-lambda` NPM 套件，這是不相關的第三方工具。如需詳細資訊，請參閱儲存庫中的 DefinitelyTyped GitHub aws-lambda。
- 此範例中的處理常式是 ES 模組，必須在 `package.json` 檔案或透過使用 `.mjs` 檔案副檔名來進行指定。如需詳細資訊，請參閱將函數處理常式指定為 ES 模組。

```
import { Handler } from 'aws-lambda';

export const handler: Handler = async (event, context) => {
    console.log('EVENT: \n' + JSON.stringify(event, null, 2));
    return context.logStreamName;
};
```

執行程序會將引數傳送至處理常式方法。第一個引數是 `event` 物件，其中包含來自叫用端的資訊。呼叫者會在呼叫 Invoke 時，以 JSON 格式的字串傳遞此資訊，然後執行時間將它轉換為物件。當 AWS 服務叫用您的函數時，事件結構會因服務而異。使用時 TypeScript，我們建議對事件物件使用類型註釋。如需詳細資訊，請參閱 使用事件物件的類型。

第二個引數為 內容物件，其中包含有關呼叫、函式和執行環境的資訊。在上述範例中，該函式從內容物件取得日誌串流的名稱並傳回給叫用端。

您也可以使用回呼引數 (此引數是您在非同步處理常式中呼叫來傳送回應的函數)。建議您使用非同步/等待 (而不是回呼)。非同步/等待改善了可讀性、錯誤處理及效率。如需有關非同步/等待和回呼之間差異的詳細資訊，請參閱 使用回呼。

使用 async/await

如果您的程式碼執行非同步任務，請使用非同步/等待模式，以確保處理常式能順利完成執行。非同步/等待是一種在 Node.js 中撰寫非同步程式碼的簡潔可讀模式，無需巢狀回呼或鏈結承諾。您可以透過非同步/等待模式撰寫讀起來像同步程式碼的程式碼，同時仍維持非同步和非封鎖的特性。

`async` 關鍵字會將函數標記為非同步，且 `await` 關鍵字會暫停函數的執行，直到 `Promise` 獲得解決為止。

Example TypeScript 功能-異步

此範例會使用 `fetch` (可在 `nodejs18.x` 執行階段使用)。注意下列事項：

- 在 Lambda 函數中使用這段程式碼之前，您必須先新增 @types/aws-lambda 套件當作開發相依項。此套件包含 Lambda 的類型定義。安裝 `@types/aws-lambda` 後，`import` 陳述式 (`import ... from 'aws-lambda'`) 會匯入類型定義。不會匯入 `aws-lambda` NPM 套件，這是不相關的第三方工具。如需詳細資訊，請參閱儲存庫中的 DefinitelyTyped GitHub aws-lambda。
- 此範例中的處理常式是 ES 模組，必須在 `package.json` 檔案或透過使用 `.mjs` 檔案副檔名來進行指定。如需詳細資訊，請參閱將函數處理常式指定為 ES 模組。

```
import { APIGatewayProxyEvent, APIGatewayProxyResult } from 'aws-lambda';
const url = 'https://aws.amazon.com/';
export const lambdaHandler = async (event: APIGatewayProxyEvent):
 Promise<APIGatewayProxyResult> => {
    try {
        // fetch is available with Node.js 18
        const res = await fetch(url);
        return {
            statusCode: res.status,
            body: JSON.stringify({
                message: await res.text(),
            }),
        };
    } catch (err) {
        console.log(err);
        return {
            statusCode: 500,
            body: JSON.stringify({
                message: 'some error happened',
            }),
        };
```

```
    }
};
```

使用回呼

建議您使用 非同步/等待 來宣告函數處理常式，而不是使用回呼。非同步/等待是更好的選擇，以下列出幾項原因：

- 可讀性： 非同步/等待程式碼比回呼程式碼更容易閱讀和理解，回呼程式碼可能很快就會變得難以理解，並引發回呼地獄。
- 偵錯和錯誤處理： 回呼型程式碼的偵錯工作難度可能不低。呼叫堆疊可能會變得難以理解，且可能會很容易接受錯誤。您可以透過非同步/等待使用 try/catch 區塊來處理錯誤。
- 效率： 回呼通常需要在程式碼的不同部分之間進行切換。非同步/等待可以減少切換環境的次數，進而產生更有效率的程式碼。

在處理常式中使用回呼時，函數將繼續執行，直到 事件迴圈 清空或函數逾時為止。直到完成所有的事件迴圈任務，回應才會傳送到叫用端。如果函式逾時，便會傳回錯誤。您可以通過將上下文 .callback WaitsFor EmptyEvent 循環設置為 false 來配置運行時立即發送響應。

回乎函式需要兩個引數，一個 `Error` 和回應。回應物件必須與 `JSON.stringify` 相容。

Example TypeScript 函數與回調

此範例函數會從 Amazon API Gateway 接收事件、記錄事件和內容物件，然後將回應傳回 API Gateway。注意下列事項：

- 在 Lambda 函數中使用這段程式碼之前，您必須先新增 @types/aws-lambda 套件當作開發相依項。此套件包含 Lambda 的類型定義。安裝 `@types/aws-lambda` 後，`import` 陳述式 (`import ... from 'aws-lambda'`) 會匯入類型定義。不會匯入 `aws-lambda` NPM 套件，這是不相關的第三方工具。如需詳細資訊，請參閱儲存庫中的 DefinitelyTyped GitHub aws-lambda。
- 此範例中的處理常式是 ES 模組，必須在 `package.json` 檔案或透過使用 `.mjs` 檔案副檔名來進行指定。如需詳細資訊，請參閱將函數處理常式指定為 ES 模組。

```
import { Context, APIGatewayProxyCallback, APIGatewayEvent } from 'aws-lambda';

export const lambdaHandler = (event: APIGatewayEvent, context: Context, callback:
 APIGatewayProxyCallback): void => {
    console.log(`Event: ${JSON.stringify(event, null, 2)}`);
```

```
    console.log(`Context: ${JSON.stringify(context, null, 2)}`);
    callback(null, {
        statusCode: 200,
        body: JSON.stringify({
            message: 'hello world',
        }),
    });
};
```

使用事件物件的類型

建議您不要使用任何類型的處理常式引數和傳回類型，因為您無法檢查類型。而是使用 sam 本機產生事件 AWS Serverless Application Model CLI 命令來產生事件，或使用 @types/aws-lambda 套件中的開放原始碼定義。

使用 sam local generate-event 命令來產生事件

1. 產生 Amazon Simple Storage Service (Amazon S3) 代理事件。

   ```
   sam local generate-event s3 put >> S3PutEvent.json
   ```

2. 使用快速類型公用程式，從 S3 PutEvent .json 檔案產生類型定義。

   ```
   npm install -g quicktype
   quicktype S3PutEvent.json -o S3PutEvent.ts
   ```

3. 在程式碼中使用產生的類型。

   ```
   import { S3PutEvent } from './S3PutEvent';

   export const lambdaHandler = async (event: S3PutEvent): Promise<void> => {
     event.Records.map((record) => console.log(record.s3.object.key));
   };
   ```

使用 @types/aws-lambda 套件中的開放原始碼定義來產生事件

1. 新增 @types/aws-lambda 套件作為開發相依項。

   ```
   npm install -D @types/aws-lambda
   ```

2. 在程式碼中使用類型

```
import { S3Event } from "aws-lambda";

export const lambdaHandler = async (event: S3Event): Promise<void> => {
  event.Records.map((record) => console.log(record.s3.object.key));
};
```

使用 .zip 檔案封存在 Lambda 中部署轉譯的程 TypeScript 式碼

在您可以部署 TypeScript 程式碼之前AWS Lambda，您需要將 JavaScript其轉譯至. 本頁說明使用 .zip 檔案封存檔建置程式碼並將程式 TypeScript 碼部署至 Lambda 的三種方式：

- 使用 AWS Serverless Application Model (AWS SAM)
- 使用 AWS Cloud Development Kit (AWS CDK)
- 使用 AWS Command Line Interface (AWS CLI) 和 esbuild

AWS SAM並AWS CDK簡化建置和部署 TypeScript 功能。AWS SAM 範本規範提供了一個簡單而清晰的語法，來描述構成無伺服器應用程式的 Lambda 函數、API、許可、組態和事件。藉助 AWS CDK，您可以在雲端建置可靠、可擴展且經濟高效的應用程式，並具有程式設計語言的強大表達能力。AWS CDK 適用於有一定經驗到經驗豐富的 AWS 使用者。AWS CDK和使AWS SAM用 esbuild 將 TypeScript 代碼轉換為. JavaScript

使用AWS SAM將程 TypeScript 式碼部署至 Lambda

請依照下列步驟使用下載、建置和部署範例 Hello World TypeScript 應用程式AWS SAM。此應用程式實作一個基本的 API 後端。其包含 Amazon API Gateway 端點和 Lambda 函數。當您將 GET 請求傳送至 API Gateway 端點時，會叫用 Lambda 函數。該函數會傳回 `hello world` 訊息。

> **Note**
>
> AWS SAM使用電子構建從 TypeScript 代碼創建 Node.js Lambda 函數。電子構建支持目前處於公開預覽中。在公開預覽期間，esbuild 支援可能面臨向後相容變更。

必要條件

若要完成本節中的步驟，您必須執行下列各項：

- AWS CLI 第 2 版
- AWS SAM CLI 1.75 版或更新版本
- Node.js 18.x

部署範例 AWS SAM 應用程式

1. 使用 Hello World TypeScript 範本初始化應用程式。

```
sam init --app-template hello-world-typescript --name sam-app --package-type Zip --
runtime nodejs18.x
```

2. (選用) 範例應用程式包含常用工具的組態，如用於程式碼檢查的 ESLint 和用於單元測試的 Jest。執行檢查和測試命令：

```
cd sam-app/hello-world
npm install
npm run lint
npm run test
```

3. 建置應用程式。

```
cd sam-app
sam build
```

4. 部署應用程式。

```
sam deploy --guided
```

5. 依照螢幕上的提示操作。若要接受互動體驗中提供的預設選項，請以 Enter 回應。
6. 輸出會顯示 REST API 的端點。在瀏覽器中開啟端點以測試函數。您應看到此回應：

```
{"message":"hello world"}
```

7. 這是可透過網際網路存取的公有 API 端點。建議您在測試後刪除端點。

```
sam delete
```

使用AWS CDK將 TypeScript 程式碼部署至 Lambda

請遵循下列步驟，使 TypeScript 用AWS CDK. 此應用程式實作一個基本的 API 後端。其包含 API Gateway 端點和 Lambda 函數。當您將 GET 請求傳送至 API Gateway 端點時，會叫用 Lambda 函數。該函數會傳回 `hello world` 訊息。

必要條件

若要完成本節中的步驟，您必須執行下列各項：

- AWS CLI 第 2 版
- AWS CDK 第 2 版
- Node.js 18.x
- Docker 或 esbuild

部署範例 AWS CDK 應用程式

1. 為您的新應用程式建立專案目錄。

```
mkdir hello-world
cd hello-world
```

2. 初始化應用程式。

```
cdk init app --language typescript
```

3. 新增 @types/aws-lambda 套件作為開發相依項。此套件包含 Lambda 的類型定義。

```
npm install -D @types/aws-lambda
```

4. 開啟 lib 目錄。您應該會看到一個名為 hello-world-stack.ts 的檔案。在此目錄中建立兩個新檔案：hello-world.function.ts 和 hello-world.ts。
5. 開啟 hello-world.function.ts，並將下列程式碼新增至檔案。這是 Lambda 函數的程式碼。

> **Note**
>
> `import` 陳述式會從 @types/aws-lambda 中匯入類型定義。不會匯入 `aws-lambda` NPM 套件，這是不相關的第三方工具。如需詳細資訊，請參閱儲存庫中的 DefinitelyTyped GitHub aws-lambda。

```
import { Context, APIGatewayProxyResult, APIGatewayEvent } from 'aws-lambda';

export const handler = async (event: APIGatewayEvent, context: Context):
 Promise<APIGatewayProxyResult> => {
    console.log(`Event: ${JSON.stringify(event, null, 2)}`);
    console.log(`Context: ${JSON.stringify(context, null, 2)}`);
    return {
        statusCode: 200,
```

```
        body: JSON.stringify({
            message: 'hello world',
        }),
    };
};
```

6. 開啟 hello-world.ts，然後將下列程式碼新增至檔案。這包含NodejsFunction 建立 Lambda 函數的建構，以及LambdaRestApi 建立 REST API 的建構。

```
import { Construct } from 'constructs';
import { NodejsFunction } from 'aws-cdk-lib/aws-lambda-nodejs';
import { LambdaRestApi } from 'aws-cdk-lib/aws-apigateway';

export class HelloWorld extends Construct {
  constructor(scope: Construct, id: string) {
    super(scope, id);
    const helloFunction = new NodejsFunction(this, 'function');
    new LambdaRestApi(this, 'apigw', {
      handler: helloFunction,
    });
  }
}
```

`NodejsFunction` 建構預設會假設以下內容：

- 您的函數處理常式被稱為 `handler`。
- 包含函數程式碼 (hello-world.function.ts) 的 .ts 檔案，與包含建構 (hello-world.ts) 的 .ts 檔案位於同一目錄中。該建構使用建構 ID ("hello-world") 和 Lambda 處理常式檔案的名稱 ("function") 來尋找函數程式碼。例如，如果您的函數程式碼位於名稱為 hello-world.my-function.ts 檔案，則 hello-world.ts 檔案必須引用如下所示函數程式碼：

```
const helloFunction = new NodejsFunction(this, 'my-function');
```

您可以變更此行為並設定其他 esbuild 參數。如需詳細資訊，請參閱 AWS CDK API 參考中的設定 esbuild。

7. 開啟 hello-world-stack.ts。這是定義 AWS CDK 堆疊的程式碼。將程式碼取代為以下內容：

```
import { Stack, StackProps } from 'aws-cdk-lib';
import { Construct } from 'constructs';
```

```
import { HelloWorld } from './hello-world';

export class HelloWorldStack extends Stack {
  constructor(scope: Construct, id: string, props?: StackProps) {
    super(scope, id, props);
    new HelloWorld(this, 'hello-world');
  }
}
```

8. 在包含 cdk.json 檔案的 hello-world 目錄中部署您的應用程式。

```
cdk deploy
```

9. AWS CDK 使用 esbuild 建置並封裝 Lambda 函數，然後將該函數部署至 Lambda 執行時間。輸出會顯示 REST API 的端點。在瀏覽器中開啟端點以測試函數。您應看到此回應：

```
{"message":"hello world"}
```

這是可透過網際網路存取的公有 API 端點。建議您在測試後刪除端點。

使用AWS CLI和電子建置將 TypeScript 程式碼部署到 Lambda

下列範例示範如何使用 esbuild 和. esbuild 產生一個具有所有相依性的 JavaScript 檔案，將 TypeScript 程式碼轉譯及部署至 Lambda AWS CLI。這是您需要新增至 .zip 封存的唯一檔案。

必要條件

若要完成本節中的步驟，您必須執行下列各項：

- AWS CLI 第 2 版
- Node.js 18.x
- Lambda 函數的執行角色
- 若您是 Windows 使用者，則為壓縮檔公用程式，如 7zip。

部署範例函數

1. 在您的本機電腦上，為新函數建立專案目錄。
2. 建立具有 npm 的新 Node.js 專案，或您選擇的套件管理器。

```
npm init
```

3. 新增 @types/aws-lambda 和 esbuild 套件作為開發相依項。@types/aws-lambda 套件包含 Lambda 的類型定義。

```
npm install -D @types/aws-lambda esbuild
```

4. 建立名稱為 index.ts 的新檔案。將下列程式碼新增至新檔案： 這是 Lambda 函數的程式碼。該函數會傳回 hello world 訊息。函數不會建立任何 API Gateway 資源。

> **Note**
>
> import 陳述式會從 @types/aws-lambda 中匯入類型定義。不會匯入 aws-lambda NPM 套件，這是不相關的第三方工具。如需詳細資訊，請參閱儲存庫中的 DefinitelyTyped GitHub aws-lambda。

```
import { Context, APIGatewayProxyResult, APIGatewayEvent } from 'aws-lambda';

export const handler = async (event: APIGatewayEvent, context: Context):
 Promise<APIGatewayProxyResult> => {
  console.log(`Event: ${JSON.stringify(event, null, 2)}`);
  console.log(`Context: ${JSON.stringify(context, null, 2)}`);
  return {
      statusCode: 200,
      body: JSON.stringify({
          message: 'hello world',
      }),
   };
};
```

5. 將建置指令碼新增至 package.json 檔案。這會將 esbuild 設定為自動建立 .zip 部署套件。如需詳細資訊，請參閱 esbuild 文件中的建置指令碼。

   Linux and MacOS

```
"scripts": {
  "prebuild": "rm -rf dist",
  "build": "esbuild index.ts --bundle --minify --sourcemap --platform=node --
target=es2020 --outfile=dist/index.js",
```

```
  "postbuild": "cd dist && zip -r index.zip index.js*"
},
```

Windows

在此範例中，"postbuild" 命令會使用 7zip 公用程式來建立 .zip 檔案。使用您偏好的 Windows zip 公用程式，並視需要修改命令。

```
"scripts": {
  "prebuild": "del /q dist",
  "build": "esbuild index.ts --bundle --minify --sourcemap --platform=node --
target=es2020 --outfile=dist/index.js",
  "postbuild": "cd dist && 7z a -tzip index.zip index.js*"
},
```

6. 建置套件。

```
npm run build
```

7. 使用 .zip 部署套件來建立 Lambda 函數。用執行角色的 Amazon Resource Name (ARN) 取代反白顯示的文字。

```
aws lambda create-function --function-name hello-world --runtime "nodejs18.x" --
role arn:aws:iam::123456789012:role/lambda-ex --zip-file "fileb://dist/index.zip"
 --handler index.handler
```

8. 執行測試事件，以確認函數傳回下列回應。如果您想使用 API Gateway 叫用此函數，請建立並設定 REST API。

```
{
  "statusCode": 200,
  "body": "{\"message\":\"hello world\"}"
}
```

使用容器映像在 Lambda 中部署轉譯的 TypeScript 程式碼

您可以將 TypeScript 程式碼作為 Node.js 容器映像部署至 AWS Lambda 函式。 AWS 提供 Node.js 的基本映像檔，以協助您建置容器映像檔。這些基本映像檔會預先載入語言執行階段，以及在 Lambda 上執行映像所需的其他元件。 AWS 為每個基本圖像提供了一個 Docker 文件，以幫助構建容器映像。

如果您使用社群或私有企業基礎映像，必須將 Node.js 執行時間介面用戶端 (RIC) 新增至基礎映像，以使其與 Lambda 相容。

Lambda 提供執行期介面模擬器，供您在本機測試函數。Node.js 的 AWS 基本映像檔包括執行階段介面模擬器。如果您使用替代基礎映像 (例如 Alpine Linux 或 Debian 映像)，便可 將模擬器建置到映像中 或 將其安裝在本機電腦上。

使用 Node.js 基本映像來構建和打包 TypeScript 函數代碼

必要條件

若要完成本節中的步驟，您必須執行下列各項：

- AWS Command Line Interface (AWS CLI) 第二版
- Docker
- Node.js 18.x

從基礎映像建立映像

若要從 Lambda 的 AWS 基本映像檔建立映像檔

1. 在您的本機電腦上，為新函數建立專案目錄。
2. 建立具有 npm 的新 Node.js 專案，或您選擇的套件管理員。

```
npm init
```

3. 新增 @types/aws-lambda 和 esbuild 套件作為開發相依項。`@types/aws-lambda` 套件包含 Lambda 的類型定義。

```
npm install -D @types/aws-lambda esbuild
```

4. 將建置指令碼新增至 `package.json` 檔案。

```
  "scripts": {
```

```
  "build": "esbuild index.ts --bundle --minify --sourcemap --platform=node --
target=es2020 --outfile=dist/index.js"
}
```

5. 建立稱為 index.ts 的新檔案。將下列範本程式碼新增至新檔案。這是 Lambda 函數的程式碼。該函數會傳回 hello world 訊息。

> **Note**
>
> import 陳述式會從 @types/aws-lambda 中匯入類型定義。不會匯入 aws-lambda NPM 套件，這是不相關的第三方工具。如需詳細資訊，請參閱儲存庫中的 DefinitelyTyped GitHub aws-lambda。

```
import { Context, APIGatewayProxyResult, APIGatewayEvent } from 'aws-lambda';

export const handler = async (event: APIGatewayEvent, context: Context):
 Promise<APIGatewayProxyResult> => {
    console.log(`Event: ${JSON.stringify(event, null, 2)}`);
    console.log(`Context: ${JSON.stringify(context, null, 2)}`);
    return {
        statusCode: 200,
        body: JSON.stringify({
            message: 'hello world',
        }),
    };
};
```

6. 建立包含下列組態的新 Dockerfile。

   - 將 FROM 屬性設定為基礎映像的 URI。
   - 設定 CMD 引數以指定 Lambda 函數處理常式。

   Example Dockerfile

   下列 Dockerfile 使用多階段建置。第一步將 TypeScript 代碼轉換為 JavaScript。第二個步驟會產生僅包含 JavaScript 檔案和生產相依性的容器映像。

```
FROM public.ecr.aws/lambda/nodejs:18 as builder
WORKDIR /usr/app
```

```
COPY package.json index.ts  ./
RUN npm install
RUN npm run build

FROM public.ecr.aws/lambda/nodejs:18
WORKDIR ${LAMBDA_TASK_ROOT}
COPY --from=builder /usr/app/dist/* ./
CMD ["index.handler"]
```

7. 使用 docker build 命令建立 Docker 映像檔。以下範例將映像命名為 docker-image 並為其提供 test 標籤。

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> 此命令會指定 --platform linux/amd64 選項，確保無論建置機器的架構為何，您的容器都與 Lambda 執行環境相容。如果您打算使用 ARM64 指令集架構建立 Lambda 函數，務必將命令變更為改用 --platform linux/arm64 選項。

(選用) 在本機測試映像

1. 使用 docker run 命令啟動 Docker 影像。在此範例中，docker-image 為映像名稱，test 為標籤。

```
docker run --platform linux/amd64 -p 9000:8080 docker-image:test
```

此命令將映像作為容器執行，並在 localhost:9000/2015-03-31/functions/function/invocations 建立本機端點。

> **Note**
>
> 如果您為 ARM64 指令集架構建立 Docker 映像檔，請務必將--platform linux/*arm64*選項改用選項。--platform linux/*amd64*

2. 從新的終端機視窗，將事件張貼至本機端點。

Linux/macOS

在 Linux 或 macOS 中，執行下列 curl 命令：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

在中 PowerShell，執行下列Invoke-WebRequest命令：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

3. 取得容器 ID。

```
docker ps
```

4. 使用 docker kill 命令停止容器。在此命令中，將 3766c4ab331c 替換為上一步驟中的容器 ID。

```
docker kill 3766c4ab331c
```

部署映像

若要將映像上傳至 Amazon ECR 並建立 Lambda 函數

1. 使用 get-login-password 命令，向 Amazon ECR 登錄檔驗證 Docker CLI。
   - 將`--region`值設定為您 AWS 區域 要建立 Amazon ECR 儲存庫的位置。
   - `111122223333`用您的 AWS 帳戶 身份證替換。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 使用 create-repository 命令在 Amazon ECR 中建立儲存庫。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 儲存庫必須與 Lambda 函數位於 AWS 區域 相同。

如果成功，您將會看到以下回應：

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
```

```
    }
}
```

3. 從上一步驟的輸出中複製 `repositoryUri`。
4. 執行 docker tag 命令，將 Amazon ECR 儲存庫中的本機映像標記為最新版本。在此命令中：
   - 將 `docker-image:test` 替換為 Docker 映像檔的名稱和標籤。
   - 將 `<ECRrepositoryUri>` 替換為複製的 `repositoryUri`。確保在 URI 的末尾包含 `:latest`。

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

範例：

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. 執行 docker push 命令，將本機映像部署至 Amazon ECR 儲存庫。確保在儲存庫 URI 的末尾包含 `:latest`。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 建立函數的執行角色 (若您還沒有的話)。在下一個步驟中您需要角色的 Amazon Resource Name (ARN)。
7. 建立 Lambda 函數。對於 `ImageUri`，從之前的設定中指定儲存庫 URI。確保在 URI 的末尾包含 `:latest`。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> 只要映像與 Lambda 函數位於相同的區域，您就可以使用不同 AWS 帳戶中的映像檔建立函數。如需詳細資訊，請參閱 Amazon ECR 跨帳戶許可。

8. 調用函數。

```
aws lambda invoke --function-name hello-world response.json
```

您應該看到如下回應：

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 若要查看函數的輸出，請檢查 `response.json` 檔案。

若要更新函數程式碼，您必須再次建置映像、將新映像上傳到 Amazon ECR 存放庫，然後使用 update-function-code 命令將映像部署到 Lambda 函數。

Lambda 將圖像標記解析為特定的圖像摘要。這表示如果您將用於部署函數的映像標籤指向 Amazon ECR 中的新映像，Lambda 不會自動更新函數以使用新映像。若要將新映像部署到相同的 Lambda 函數，您必須使用`update-function-code`命令，即使 Amazon ECR 中的影像標籤保持不變。

AWS Lambda上下文對象 TypeScript

當 Lambda 執行您的函數時，它會將內容物件傳遞至處理常式。此物件提供的方法和各項屬性提供了有關調用、函式以及執行環境的資訊。

內容方法

- `getRemainingTimeInMillis()` - 傳回執行逾時前剩餘的毫秒數。

內容屬性

- `functionName` – Lambda 函數的名稱。
- `functionVersion` – 函數的版本。
- `invokedFunctionArn` - 用於調用此函數的 Amazon Resource Name (ARN)。指出調用者是否指定版本號或別名。
- `memoryLimitInMB` - 分配給函數的記憶體數量。
- `awsRequestId` - 調用請求的識別符。
- `logGroupName` - 函數的日誌群組。
- `logStreamName` - 函數執行個體的記錄串流。
- `identity` - (行動應用程式) 已授權請求的 Amazon Cognito 身分的相關資訊。
  - `cognitoIdentityId` - 已驗證的 Amazon Cognito 身分。
  - `cognitoIdentityPoolId` - 授權調用的 Amazon Cognito 身分集區。
- `clientContext` - (行動應用程式) 用戶端應用程式提供給 Lambda 的用戶端內容。
  - `client.installation_id`
  - `client.app_title`
  - `client.app_version_name`
  - `client.app_version_code`
  - `client.app_package_name`
  - `env.platform_version`
  - `env.platform`
  - `env.make`
  - `env.model`
  - `env.locale`

  - `Custom` - 用戶端應用程式所設定的自訂值。
- `callbackWaitsForEmptyEventLoop` - 設為 false 將會在回呼執行時立即傳送回應，而不會等待 Node.js 事件迴圈成空白。如果此為 false，任何未完成的事件都會於下次調用時繼續執行。

您可以使用 @types/aws-lambda npm 套件處理內容物件。

Example index.ts 檔案

以下範例函式紀錄內文資訊和傳回日誌的位置。

> **Note**
>
> 在 Lambda 函數中使用這段程式碼之前，您必須先新增 @types/aws-lambda 套件當作開發相依項。此套件包含 Lambda 的類型定義。安裝 `@types/aws-lambda` 後，`import` 陳述式 (`import ... from 'aws-lambda'`) 會匯入類型定義。不會匯入 `aws-lambda` NPM 套件，這是不相關的第三方工具。如需詳細資訊，請參閱儲存庫中的 DefinitelyTyped GitHub aws-lambda。

```
import { Context } from 'aws-lambda';
export const lambdaHandler = async (event: string, context: Context): Promise<string>
 => {
  console.log('Remaining time: ', context.getRemainingTimeInMillis());
  console.log('Function name: ', context.functionName);
  return context.logStreamName;
};
```

AWS Lambda 功能登錄 TypeScript

AWS Lambda 自動監控 Lambda 函數，並將日誌項目傳送到 Amazon CloudWatch。您的 Lambda 函數隨附一個 CloudWatch 日誌記錄群組和函數每個執行個體的日誌串流。Lambda 執行期環境會將每次調用的詳細資訊和函數程式碼的其他輸出，傳送至日誌串流。如需有關 CloudWatch 記錄檔的詳細資訊，請參閱使用 Amazon CloudWatch 日誌 AWS Lambda。

若要由您的函數程式碼輸出日誌，您可以在主控台物件上使用方法。若要更詳細記錄，您可以使用任何寫入 `stdout` 或 `stderr` 的記錄程式庫。

章節

工具與程式庫

Powertools to AWS Lambda (TypeScript) 是開發人員工具組，用於實作無伺服器最佳實務並提高開發人員速度。Logger 公用程式提供 Lambda 優化記錄器，其中包含有關所有函數之函數內容的其他資訊，輸出結構為 JSON。使用此公用程式執行下列操作：

- 從 Lambda 內容、冷啟動和 JSON 形式的結構記錄輸出中擷取關鍵欄位
- 在收到指示時記錄 Lambda 調用事件 (預設為停用)
- 透過日誌採樣僅列印調用百分比的所有日誌 (預設為停用)
- 在任何時間點將其他金鑰附加至結構化日誌
- 使用自訂日誌格式化程式 (自帶格式化程式)，以與組織的日誌記錄 RFC 相容的結構輸出日誌。

使用 Powertools 進行 AWS Lambda （ TypeScript ）和結構化日 AWS SAM 誌記錄

請按照以下步驟下載，構建和部署示例 Hello World TypeScript 應用程序，其中包含集成的 Powertools 的 AWS Lambda （ TypeScript ）模塊使用 AWS SAM. 此應用程式實作了基本 API 後端，並使用

Powertools 發送日誌、指標和追蹤。其包含 Amazon API Gateway 端點和 Lambda 函數。當您將 GET 請求傳送至 API Gateway 端點時，Lambda 函數會叫用、使用內嵌指標格式將記錄和指標傳送至 CloudWatch，並將追蹤傳送至 AWS X-Ray。該函數會傳回 `hello world` 訊息。

必要條件

若要完成本節中的步驟，您必須執行下列各項：

- Node.js 18.x 或更新版本
- AWS CLI 第二版
- AWS SAM CLI 版本 1.75 或更新版本。如果您使用較舊版本的 AWS SAM CLI，請參閱升級 AWS SAM CLI。

部署範例 AWS SAM 應用程式

1. 使用 Hello World TypeScript 範本初始化應用程式。

```
sam init --app-template hello-world-powertools-typescript --name sam-app --package-type Zip --runtime nodejs18.x
```

2. 建置應用程式。

```
cd sam-app && sam build
```

3. 部署應用程式。

```
sam deploy --guided
```

4. 依照螢幕上的提示操作。若要接受互動體驗中提供的預設選項，請按下 `Enter`。

> **Note**
>
> 因為HelloWorldFunction 可能沒有定義授權，這可以嗎？ ，請務必輸入y。

5. 取得已部署應用程式的 URL：

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. 調用 API 端點：

```
curl <URL_FROM_PREVIOUS_STEP>
```

成功的話，您將會看到以下回應：

```
{"message":"hello world"}
```

7. 若要獲取該函數的日誌，請執行 sam 日誌。如需詳細資訊，請參閱《AWS Serverless Application Model 開發人員指南》 中的 使用日誌。

```
sam logs --stack-name sam-app
```

日誌輸出如下：

```
2023/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2022-08-31T09:33:10.552000
 START RequestId: 70693159-7e94-4102-a2af-98a6343fb8fb Version: $LATEST
2023/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2022-08-31T09:33:10.594000
 2022-08-31T09:33:10.557Z 70693159-7e94-4102-a2af-98a6343fb8fb
 INFO {"_aws":{"Timestamp":1661938390556,"CloudWatchMetrics":
[{"Namespace":"sam-app","Dimensions":[["service"]],"Metrics":
[{"Name":"ColdStart","Unit":"Count"}]}]},"service":"helloWorld","ColdStart":1}
2023/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2022-08-31T09:33:10.595000
 2022-08-31T09:33:10.595Z 70693159-7e94-4102-a2af-98a6343fb8fb INFO
 {"level":"INFO","message":"This is an INFO log - sending HTTP 200 - hello world
 response","service":"helloWorld","timestamp":"2022-08-31T09:33:10.594Z"}
2023/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2022-08-31T09:33:10.655000
 2022-08-31T09:33:10.655Z 70693159-7e94-4102-a2af-98a6343fb8fb INFO
 {"_aws":{"Timestamp":1661938390655,"CloudWatchMetrics":[{"Namespace":"sam-
app","Dimensions":[["service"]],"Metrics":[]}]},"service":"helloWorld"}
2023/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2022-08-31T09:33:10.754000 END
 RequestId: 70693159-7e94-4102-a2af-98a6343fb8fb
2023/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2022-08-31T09:33:10.754000
 REPORT RequestId: 70693159-7e94-4102-a2af-98a6343fb8fb Duration: 201.55 ms Billed
 Duration: 202 ms Memory Size: 128 MB Max Memory Used: 66 MB Init Duration: 252.42
 ms
XRAY TraceId: 1-630f2ad5-1de22b6d29a658a466e7ecf5 SegmentId: 567c116658fbf11a
 Sampled: true
```

8. 這是可透過網際網路存取的公有 API 端點。建議您在測試後刪除端點。

```
sam delete
```

管理日誌保留

當您刪除函數時，不會自動刪除日誌群組。若要避免無限期儲存記錄檔，請刪除記錄群組，或設定保留期間，之後 CloudWatch 會自動刪除記錄檔。若要設定記錄保留，請將下列項目新增至 AWS SAM 範本：

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
    # Omitting other properties

  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

使用動力工具 AWS Lambda （ TypeScript ）和結構化日 AWS CDK 誌記錄

請按照以下步驟下載，構建和部署示例 Hello World TypeScript 應用程序，其中包含集成的 Powertools 的 AWS Lambda （ TypeScript ）模塊使用 AWS CDK. 此應用程式實作了基本 API 後端，並使用 Powertools 發送日誌、指標和追蹤。其包含 Amazon API Gateway 端點和 Lambda 函數。當您將 GET 請求傳送至 API Gateway 端點時，Lambda 函數會叫用、使用內嵌指標格式將記錄和指標傳送至 CloudWatch，並將追蹤傳送至 AWS X-Ray。該函數會傳回 `hello world` 訊息。

必要條件

若要完成本節中的步驟，您必須執行下列各項：

- Node.js 18.x 或更新版本
- AWS CLI 第二版
- AWS CDK 第二版
- AWS SAM CLI 版本 1.75 或更新版本。如果您使用較舊版本的 AWS SAM CLI，請參閱升級 AWS SAM CLI。

部署範例 AWS CDK 應用程式

1. 為您的新應用程式建立專案目錄。

```
mkdir hello-world
cd hello-world
```

2. 初始化應用程式。

```
cdk init app --language typescript
```

3. 新增 @types/aws-lambda 套件作為開發相依項。

```
npm install -D @types/aws-lambda
```

4. 安裝 Powertools Logger 公用程式。

```
npm install @aws-lambda-powertools/logger
```

5. 開啟 lib 目錄。您應看到名稱為 hello-world-stack.ts 的檔案。在此目錄中建立兩個新檔案：hello-world.function.ts 和 hello-world.ts。
6. 開啟 hello-world.function.ts，並將下列程式碼新增至檔案。這是 Lambda 函數的程式碼。

```
import { APIGatewayEvent, APIGatewayProxyResult, Context } from 'aws-lambda';
import { Logger } from '@aws-lambda-powertools/logger';
const logger = new Logger();

export const handler = async (event: APIGatewayEvent, context: Context):
 Promise<APIGatewayProxyResult> => {
  logger.info('This is an INFO log - sending HTTP 200 - hello world response');
  return {
    statusCode: 200,
    body: JSON.stringify({
      message: 'hello world',
    }),
  };
};
```

7. 開啟 hello-world.ts，然後將下列程式碼新增至檔案。其中包含NodejsFunction 建立 Lambda 函數的建構、設定 Powertools 的環境變數，以及將記錄保留設定為一週。它也包含LambdaRestApi 建立 REST API 的建構。

```
import { Construct } from 'constructs';
import { NodejsFunction } from 'aws-cdk-lib/aws-lambda-nodejs';
import { LambdaRestApi } from 'aws-cdk-lib/aws-apigateway';
```

```
import { RetentionDays } from 'aws-cdk-lib/aws-logs';
import { CfnOutput } from 'aws-cdk-lib';

export class HelloWorld extends Construct {
  constructor(scope: Construct, id: string) {
    super(scope, id);
    const helloFunction = new NodejsFunction(this, 'function', {
      environment: {
        Powertools_SERVICE_NAME: 'helloWorld',
        LOG_LEVEL: 'INFO',
      },
      logRetention: RetentionDays.ONE_WEEK,
    });
    const api = new LambdaRestApi(this, 'apigw', {
      handler: helloFunction,
    });
    new CfnOutput(this, 'apiUrl', {
      exportName: 'apiUrl',
      value: api.url,
    });
  }
}
```

8. 開啟 hello-world-stack.ts。這是定義 AWS CDK 堆疊的程式碼。將程式碼取代為以下內容：

```
import { Stack, StackProps } from 'aws-cdk-lib';
import { Construct } from 'constructs';
import { HelloWorld } from './hello-world';

export class HelloWorldStack extends Stack {
  constructor(scope: Construct, id: string, props?: StackProps) {
    super(scope, id, props);
    new HelloWorld(this, 'hello-world');
  }
}
```

9. 返回專案目錄。

```
cd hello-world
```

10. 部署您的應用程式。

```
cdk deploy
```

11. 取得已部署應用程式的 URL：

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?ExportName==`apiUrl`].OutputValue' --output text
```

12. 調用 API 端點：

```
curl <URL_FROM_PREVIOUS_STEP>
```

成功的話，您將會看到以下回應：

```
{"message":"hello world"}
```

13. 若要獲取該函數的日誌，請執行 sam 日誌。如需詳細資訊，請參閱《AWS Serverless Application Model 開發人員指南》 中的 使用日誌。

```
sam logs --stack-name HelloWorldStack
```

日誌輸出如下：

```
2023/01/31/[$LATEST]2ca67f180dcd4d3e88b5d68576740c8e 2022-08-31T14:48:37.047000
 START RequestId: 19ad1007-ff67-40ce-9afe-0af0a9eb512c Version: $LATEST
2023/01/31/[$LATEST]2ca67f180dcd4d3e88b5d68576740c8e 2022-08-31T14:48:37.050000 {
"level": "INFO",
"message": "This is an INFO log - sending HTTP 200 - hello world response",
"service": "helloWorld",
"timestamp": "2022-08-31T14:48:37.048Z",
"xray_trace_id": "1-630f74c4-2b080cf77680a04f2362bcf2"
}
2023/01/31/[$LATEST]2ca67f180dcd4d3e88b5d68576740c8e 2022-08-31T14:48:37.082000 END
 RequestId: 19ad1007-ff67-40ce-9afe-0af0a9eb512c
2023/01/31/[$LATEST]2ca67f180dcd4d3e88b5d68576740c8e 2022-08-31T14:48:37.082000
 REPORT RequestId: 19ad1007-ff67-40ce-9afe-0af0a9eb512c Duration: 34.60 ms Billed
 Duration: 35 ms Memory Size: 128 MB Max Memory Used: 57 MB Init Duration: 173.48
 ms
```

14. 這是可透過網際網路存取的公有 API 端點。建議您在測試後刪除端點。

```
cdk destroy
```

使用 Lambda 主控台

您可以在調用 Lambda 函數之後，使用 Lambda 主控台來檢視日誌輸出。

如果可以從內嵌程式碼編輯器測試您的程式碼，您會在執行結果中找到日誌。使用主控台測試功能以調用函數時，您會在詳細資訊區段找到日誌輸出。

使用控 CloudWatch 制台

您可以使用 Amazon 主 CloudWatch 控台來檢視所有 Lambda 函數叫用的日誌。

在 CloudWatch 主控台上檢視記錄檔

1. 在主控台上開啟 [記錄群組] 頁 CloudWatch 面。
2. 選擇您的函數的日誌群組 (/aws/lambda/***your-function-name***)。
3. 選擇日誌串流

每個日誌串流都會對應至函式的執行個體。當您更新 Lambda 函數，以及建立額外執行個體以處理多個並行調用時，便會出現日誌串流。要查找特定調用的日誌，我們建議使用檢測您的函數。 AWS X-Ray X-Ray 會在追蹤內記錄有關請求和日誌串流的詳細資訊。

追蹤 TypeScript 程式碼 AWS Lambda

Lambda 會與 AWS X-Ray 整合，以協助您追蹤、偵錯和最佳化 Lambda 應用程式。您可以使用 X-Ray 來追蹤請求，因為它會周遊您應用程式中的資源，其中可能包含 Lambda 函數和其他 AWS 服務。

若要將追蹤資料傳送至 X-Ray，您可以使用以下三個 SDK 庫之一：

- AWS適用於 OpenTelemetry (ADOT) 的發行版 — (OTel) SDK 的安全、可生產就緒且AWS支援的發行版本 OpenTelemetry 。
- 適用於 Node.js 的 AWS X-Ray SDK：用於產生追蹤資料並傳送至 X-Ray 的 SDK。
- Powertools in AWS Lambda (TypeScript) — 用於實作無伺服器最佳做法並提高開發人員速度的開發人員工具組。

每個 SDK 均提供將遙測資料傳送至 X-Ray 服務的方法。然後，您可以使用 X-Ray 來檢視、篩選應用程式的效能指標並獲得洞察，從而識別問題和進行最佳化的機會。

> **Important**
>
> X-Ray 和適用於 AWS Lambda SDK 的 Powertools 包含在 AWS 提供的緊密整合檢測解決方案中。ADOT Lambda Layers 是用於追蹤檢測之業界通用標準的一部分，這類檢測一般會收集更多資料，但可能不適用於所有使用案例。您可以使用任一解決方案在 X-Ray 中實作 end-to-end 追蹤。若要深入了解如何在兩者之間做選擇，請參閱在 AWS Distro for OpenTelemetry 和 X-Ray SDK 之間進行選擇。

章節

- 使用動力工具進行AWS Lambda（TypeScript）和跟AWS SAM踪
- 使用動力工具AWS Lambda（TypeScript）和用AWS CDK於跟踪
- 解讀 X-Ray 追蹤

使用動力工具進行AWS Lambda（TypeScript）和跟AWS SAM踪

請按照以下步驟下載，構建和部署示例 Hello World TypeScript 應用程序，其中包含集成的 Powertools的AWS Lambda（TypeScript）模塊使用AWS SAM. 此應用程式實作了基本 API 後端，並使用 Powertools 發送日誌、指標和追蹤。其包含 Amazon API Gateway 端點和 Lambda 函數。當您將

GET 請求傳送至 API Gateway 端點時，Lambda 函數會叫用、使用內嵌指標格式將記錄和指標傳送至 CloudWatch，並將追蹤傳送至AWS X-Ray。該函數會傳回 `hello world` 訊息。

必要條件

若要完成本節中的步驟，您必須執行下列各項：

- Node.js 18.x 或更新版本
- AWS CLI 第 2 版
- AWS SAM CLI 1.75 版或更新版本 如果您使用舊版 AWS SAM CLI，請參閱升級 AWS SAM CLI。

部署範例 AWS SAM 應用程式

1. 使用 Hello World TypeScript 範本初始化應用程式。

```
sam init --app-template hello-world-powertools-typescript --name sam-app --package-type Zip --runtime nodejs18.x --no-tracing
```

2. 建置應用程式。

```
cd sam-app && sam build
```

3. 部署應用程式。

```
sam deploy --guided
```

4. 依照螢幕上的提示操作。若要接受互動體驗中提供的預設選項，請按下 `Enter`。

> **Note**
>
> 因為HelloWorldFunction 可能沒有定義授權，這可以嗎？ ，確保輸入y。

5. 取得已部署應用程式的 URL：

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. 調用 API 端點：

```
curl <URL_FROM_PREVIOUS_STEP>
```

成功的話，您將會看到以下回應：

```
{"message":"hello world"}
```

7. 若要取得函數的追蹤，請執行 sam 追蹤。

```
sam traces
```

追蹤輸出如下：

```
XRay Event [revision 1] at (2023-01-31T11:29:40.527000) with id
 (1-11a2222-111a222222cb33de3b95daf9) and duration (0.483s)
  - 0.425s - sam-app/Prod [HTTP: 200]
    - 0.422s - Lambda [HTTP: 200]
  - 0.406s - sam-app-HelloWorldFunction-Xyzv11a1bcde [HTTP: 200]
  - 0.172s - sam-app-HelloWorldFunction-Xyzv11a1bcde
    - 0.179s - Initialization
    - 0.112s - Invocation
      - 0.052s - ## app.lambdaHandler
        - 0.001s - ### MySubSegment
    - 0.059s - Overhead
```

8. 這是可透過網際網路存取的公有 API 端點。建議您在測試後刪除端點。

```
sam delete
```

X-Ray 無法追蹤應用程式的所有請求。X-Ray 會套用取樣演算法以確保追蹤的效率，同時仍提供所有請求的代表範本。取樣率為每秒 1 次請求和 5% 的額外請求。

> **Note**
>
> 您無法針對函數設定 X-Ray 取樣率。

使用動力工具AWS Lambda（TypeScript）和用AWS CDK於跟踪

請按照以下步驟下載，構建和部署示例 Hello World TypeScript 應用程序，其中包含集成的 Powertools 的AWS Lambda（TypeScript）模塊使用AWS CDK. 此應用程式實作了基本 API 後端，並使用 Powertools 發送日誌、指標和追蹤。其包含 Amazon API Gateway 端點和 Lambda 函數。當您將

GET 請求傳送至 API Gateway 端點時，Lambda 函數會叫用、使用內嵌指標格式將記錄和指標傳送至 CloudWatch，並將追蹤傳送至AWS X-Ray。該函數會傳回 `hello world` 訊息。

必要條件

若要完成本節中的步驟，您必須執行下列各項：

- Node.js 18.x 或更新版本
- AWS CLI 第 2 版
- AWS CDK 第 2 版
- AWS SAM CLI 1.75 版或更新版本 如果您使用舊版 AWS SAM CLI，請參閱升級 AWS SAM CLI。

部署範例 AWS Cloud Development Kit (AWS CDK) 應用程式

1. 為您的新應用程式建立專案目錄。

```
mkdir hello-world
cd hello-world
```

2. 初始化應用程式。

```
cdk init app --language typescript
```

3. 新增 @types/aws-lambda 套件作為開發相依項。

```
npm install -D @types/aws-lambda
```

4. 安裝 Powertools Tracer 公用程式。

```
npm install @aws-lambda-powertools/tracer
```

5. 開啟 lib 目錄。您應該會看到一個名為 hello-world-stack.ts 的檔案。在此目錄中建立兩個新檔案：hello-world.function.ts 和 hello-world.ts。
6. 開啟 hello-world.function.ts，並將下列程式碼新增至檔案。這是 Lambda 函數的程式碼。

```
import { APIGatewayEvent, APIGatewayProxyResult, Context } from 'aws-lambda';
import { Tracer } from '@aws-lambda-powertools/tracer';
const tracer = new Tracer();
```

```
export const handler = async (event: APIGatewayEvent, context: Context):
 Promise<APIGatewayProxyResult> => {
  // Get facade segment created by Lambda
  const segment = tracer.getSegment();

  // Create subsegment for the function and set it as active
  const handlerSegment = segment.addNewSubsegment(`## ${process.env._HANDLER}`);
  tracer.setSegment(handlerSegment);

  // Annotate the subsegment with the cold start and serviceName
  tracer.annotateColdStart();
  tracer.addServiceNameAnnotation();

  // Add annotation for the awsRequestId
  tracer.putAnnotation('awsRequestId', context.awsRequestId);
  // Create another subsegment and set it as active
  const subsegment = handlerSegment.addNewSubsegment('### MySubSegment');
  tracer.setSegment(subsegment);
  let response: APIGatewayProxyResult = {
    statusCode: 200,
    body: JSON.stringify({
      message: 'hello world',
    }),
  };
  // Close subsegments (the Lambda one is closed automatically)
  subsegment.close(); // (### MySubSegment)
  handlerSegment.close(); // (## index.handler)

  // Set the facade segment as active again (the one created by Lambda)
  tracer.setSegment(segment);
  return response;
};
```

7. 開啟 hello-world.ts，然後將下列程式碼新增至檔案。其中包含NodejsFunction 建立 Lambda 函數的建構、設定 Powertools 的環境變數，以及將記錄保留設定為一週。它也包含LambdaRestApi 建立 REST API 的建構。

```
import { Construct } from 'constructs';
import { NodejsFunction } from 'aws-cdk-lib/aws-lambda-nodejs';
import { LambdaRestApi } from 'aws-cdk-lib/aws-apigateway';
import { CfnOutput } from 'aws-cdk-lib';
import { Tracing } from 'aws-cdk-lib/aws-lambda';
```

```
export class HelloWorld extends Construct {
  constructor(scope: Construct, id: string) {
    super(scope, id);
    const helloFunction = new NodejsFunction(this, 'function', {
      environment: {
        POWERTOOLS_SERVICE_NAME: 'helloWorld',
      },
      tracing: Tracing.ACTIVE,
    });
    const api = new LambdaRestApi(this, 'apigw', {
      handler: helloFunction,
    });
    new CfnOutput(this, 'apiUrl', {
      exportName: 'apiUrl',
      value: api.url,
    });
  }
}
```

8. 開啟 hello-world-stack.ts。這是定義 AWS CDK 堆疊的程式碼。將程式碼取代為以下內容：

```
import { Stack, StackProps } from 'aws-cdk-lib';
import { Construct } from 'constructs';
import { HelloWorld } from './hello-world';

export class HelloWorldStack extends Stack {
  constructor(scope: Construct, id: string, props?: StackProps) {
    super(scope, id, props);
    new HelloWorld(this, 'hello-world');
  }
}
```

9. 部署您的應用程式。

```
cd ..
cdk deploy
```

10. 取得已部署應用程式的 URL：

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?ExportName==`apiUrl`].OutputValue' --output text
```

11. 調用 API 端點：

```
curl <URL_FROM_PREVIOUS_STEP>
```

成功的話，您將會看到以下回應：

```
{"message":"hello world"}
```

12. 若要取得函數的追蹤，請執行 sam 追蹤。

```
sam traces
```

追蹤輸出如下：

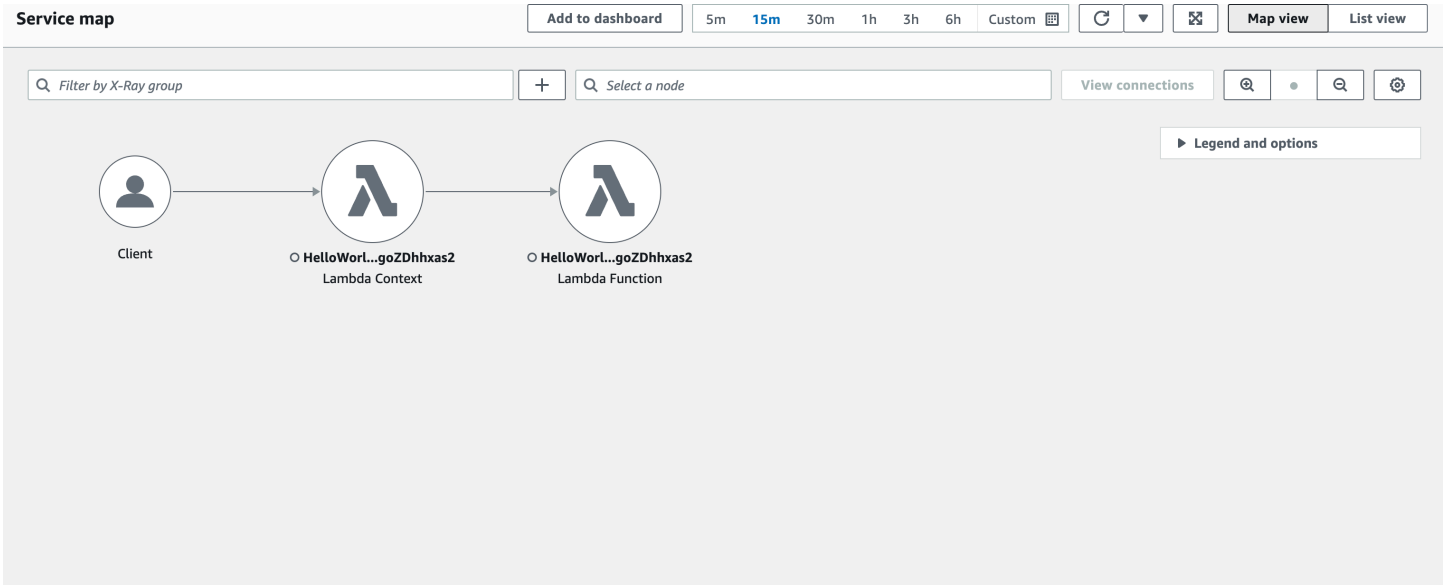
```
XRay Event [revision 1] at (2023-01-31T11:50:06.997000) with id
 (1-11a2222-111a222222cb33de3b95daf9) and duration (0.449s)
  - 0.350s - HelloWorldStack-helloworldfunction111A2BCD-Xyzv11a1bcde [HTTP: 200]
  - 0.157s - HelloWorldStack-helloworldfunction111A2BCD-Xyzv11a1bcde
    - 0.169s - Initialization
    - 0.058s - Invocation
      - 0.055s - ## index.handler
        - 0.000s - ### MySubSegment
    - 0.099s - Overhead
```

13. 這是可透過網際網路存取的公有 API 端點。建議您在測試後刪除端點。

```
cdk destroy
```

解讀 X-Ray 追蹤

設定主動追蹤之後，您可以透過應用程式來觀察特定請求。X-Ray 追蹤地圖提供了有關應用程式及其所有元件的資訊。下列範例顯示範例應用程式的追蹤：


Service map
Add to dashboard
5m
15m
30m
1h
3h
6h
Custom
Map view
List view
Filter by X-Ray group
Select a node
View connections
Legend and options
Client
HelloWorl...goZDhhxas2
Lambda Context
HelloWorl...goZDhhxas2
Lambda Function

使用 Python 建置 Lambda 函數

您可以在 AWS Lambda中執行 Python 程式碼。Lambda 提供用於執行程式碼來處理事件的 Python 執行期。您的程式碼在包含 SDK for Python (Boto3) 環境中執行，其中包含您所管理之 AWS Identity and Access Management (IAM) 角色的登入資料。若要進一步瞭解 Python 執行階段隨附的 SDK 版本，請參閱the section called “包含執行階段的 SDK 版本”。

Lambda 支援以下 Python 執行期。

Python

名稱	識別符	作業系統	取代日期	封鎖函數建立	封鎖函數更新
Python 3.12	`python3.12`	Amazon Linux 2023			
Python 3.11	`python3.11`	Amazon Linux 2			
Python 3.10	`python3.10`	Amazon Linux 2			
Python 3.9	`python3.9`	Amazon Linux 2			
Python 3.8	`python3.8`	Amazon Linux 2	2024 年 10 月 14 日	2025年2月28日	2025年3月31日

> **Note**
>
> 此資料表中的執行期資訊會不斷更新。如需在 Lambda 中使用 AWS SDK 的詳細資訊，請參閱在無伺服器陸地中管理 Lambda 函數中的 AWS SDK。

若要建立 Python 函數

1. 開啟 Lambda 主控台。
2. 選擇建立函數。

3. 進行下列設定：
   - 函數名稱：輸入函數名稱。
   - 執行期：選擇 Python 3.12。
4. 選擇建立函數。
5. 若要設定測試事件，請選擇 Test (測試)。
6. 事件名稱輸入 **`test`**。
7. 選擇儲存變更。
8. 若要調用函數，請選擇 Test (測試)。

主控台將建立一個 Lambda 函數，其具有名為 `lambda_function` 的單一來源檔案。您可以使用內建的程式碼編輯器編輯該檔案並加入更多檔案。選擇 Save (儲存) 以儲存變更。然後，若要執行程式碼，請選擇 Test (測試)。

> **Note**
>
> Lambda 主控台用 AWS Cloud9 來在瀏覽器中提供整合式開發環境。您也可以使用 AWS Cloud9 在自己的環境中開發 Lambda 函數。若要取得更多資訊，請參閱使用指南 AWS 工具組中的〈使用 AWS Lambda 函數〉。 AWS Cloud9

> **Note**
>
> 若要在本機環境中開始進行應用程式開發，請部署本指南 GitHub 儲存庫中提供的其中一個範例應用程式。
>
> 以 Python 編寫的範例 Lambda 應用程式
>
> - 空白蟒蛇- 一個 Python 函數，顯示日誌記錄，環境變量， AWS X-Ray 跟踪，圖層，單元測試和 SDK 的使用。 AWS

您的 Lambda 函數隨附 CloudWatch 日誌記錄群組。函數運行時將有關每次調用的詳細信息發送到 CloudWatch 日誌。它在調用期間會轉送您的函數輸出的任何記錄。如果您的函數傳回錯誤，Lambda 會對該錯誤進行格式化之後傳回給調用端。

主題

包含執行階段的 SDK 版本

Python 執行階段中包含的 AWS SDK 版本取決於執行階段版本和您的 AWS 區域. 若要尋找您正在使用的執行階段中包含的 SDK 版本，請使用下列程式碼建立 Lambda 函數。

```
import boto3
import botocore

def lambda_handler(event, context):
   print(f'boto3 version: {boto3.__version__}')
   print(f'botocore version: {botocore.__version__}')
```

回應格式

在 Python 3.12 及更高版本 Python 的執行期中，函數傳回的 JSON 回應包含 Unicode 字元。早期版本 Python 的執行期會在回應中傳回 Unicode 字元的逸出序列。例如，在 Python 3.11 中，如果您傳回 Unicode 字串，如 "こんにちは"，它將逸出 Unicode 字元並傳回 "\u3053\u3093\u306b\u3061\u306f"。Python 3.12 執行期會傳回原始的 "こんにちは"。

使用 Unicode 回應可使 Lambda 回應變小，因此能更容易地將較大的回應納入同步函數的 6 MB 最大承載大小。在之前的範例中，逸出版本為 32 位元組，相較之下，Unicode 字串為 17 位元組。

當您升級到 Python 3.12 時，可能需要調整您的程式碼以適應新的回應格式。若呼叫者預期得到逸出 Unicode，您必須新增程式碼至傳回的函數以便手動逸出 Unicode，或調整呼叫者以處理 Unicode 傳回。

延伸模組正常關機

Python 3.12 及更高版本 Python 的執行期為具有外部延伸模組的函數提供正常關機功能。當 Lambda 關閉執行環境時，它會傳送 SIGTERM 訊號到執行期，然後傳送 SHUTDOWN 事件到每個註冊的外部延伸模組。您可以捕獲 Lambda 函數中的 SIGTERM 訊息並清理資源，例如由函數建立的資料庫連線等。

若要詳細了解執行環境生命週期，請參閱 Lambda 執行環境。如需如何搭配擴充功能使用正常關機的範例，請參閱AWS 範例 GitHub 儲存庫。

在 Python 中 Lambda 義函數處理程序

Lambda 函數處理常式是您的函數程式碼中處理事件的方法。當有人呼叫您的函數時，Lambda 會執行處理常式方法。函數會執行，直到處理常式傳回回應、結束或逾時為止。

在 Python 中建立函數處理常式時，您可以使用下列一般語法：

```
def handler_name(event, context):
    ...
    return some_value
```

命名

建立 Lambda 函數時指定的 Lambda 函數處理常式名稱衍生自下列項目：

- Lambda 處理常式函數所在的檔案名稱。
- Python 處理常式函數的名稱。

函數處理常式可以是任何名稱；但 Lambda 主控台中的預設名稱為 `lambda_function.lambda_handler`。此函數處理常式名稱會反映函數名稱 (`lambda_handler`)，以及存放處理常式程式碼的檔案 (`lambda_function.py`)。

如果要在主控台中使用不同檔案名稱或函數處理常式名稱建立函數，您必須編輯預設處理常式名稱。

變更函數處理常式名稱的方式 (主控台)

1. 開啟 Lambda 主控台的函數頁面，然後選擇您的函數。
2. 選擇 程式碼 索引標籤。
3. 向下捲動至執行時間設定窗格，並選擇編輯。
4. 在處理常式中，輸入函數處理常式的新名稱。
5. 選擇 儲存 。

運作方式

Lambda 調用函數處理常式時，Lambda 執行時間會將兩個引數傳遞給函數處理常式：

- 第一個引數是事件物件。事件是一種 JSON 格式的文件，會包含供 Lambda 函數處理的資料。Lambda 執行時間會將事件轉換為物件，再將它傳遞到您的函數程式碼。其通常屬於 Python `dict` 類型。同時也屬於 `list`、`str`、`int`、`float` 或 `NoneType` 類型。

  事件物件包含調用服務的資訊。當您呼叫函數時，您決定事件的結構和內容。當 AWS 服務叫用您的函數時，服務會定義事件結構。如需有關來自 AWS 服務之事件的詳細資訊，請參閱使用來自其 AWS 他服務的事件叫用 Lambda。
- 第二個引數是內容物件。Lambda 在執行時間將內容物件傳遞到您的函數。此物件提供的方法和各項屬性提供了有關調用、函式以及執行時間環境的資訊。

傳回值

處理常式也可選擇傳回值。傳回值的情況取決於調用該函數的調用類型和服務。例如：

- 如果您使用RequestResponse叫用類型，例如同步調用，會將 Python 函數呼叫的結果 AWS Lambda 傳回給用戶端叫用 Lambda 函數 (在呼叫要求的 HTTP 回應中，序列化為 JSON)。例如，AWS Lambda 主控台使用 `RequestResponse` 調用類型。因此，當您在主控台上調用函數時，主控台即會顯示傳回值。
- 如果處理常式傳回 `json.dumps` 無法序列化的物件，則執行時間會傳回錯誤。
- 如果處理常式傳回 None (如沒有 `return` 陳述式的 Python 函數隱含作業)，則執行時間會傳回 `null`。
- 如果使用 `Event` 調用類型 (非同步調用)，便會捨棄該值。

> **Note**
>
> 在 Python 3.9 及更高版本中，Lambda 會在錯誤回應中包含調用的請求 ID。

範例

下述章節顯示您可以與 Lambda 搭配使用的 Python 函數範例。如果您使用 Lambda 主控台編寫您的函數，則不需要附加 .zip 封存檔即可執行本節中的函數。這些函數使用標準 Python 程式庫，其包含在您選擇的 Lambda 執行時間內。如需詳細資訊，請參閱 Lambda 部署套件。

傳回訊息

下列範例顯示稱為 lambda_handler 的函數。該函數接受使用者的名字和姓氏輸入，並傳回一則訊息，其中包含其作為輸入接收的事件資料。

```
def lambda_handler(event, context):
    message = 'Hello {} {}!'.format(event['first_name'], event['last_name'])
    return {
        'message' : message
    }
```

您可以使用下列事件資料來調用函數：

```
{
    "first_name": "John",
    "last_name": "Smith"
}
```

該回應顯示作為輸入傳遞的事件資料：

```
{
    "message": "Hello John Smith!"
}
```

剖析回應

下列範例顯示稱為 lambda_handler 的函數。該函數會使用在執行時間由 Lambda 傳遞的事件資料。其剖析 JSON 回應中 AWS_REGION 傳回的環境變數。

```
import os
import json

def lambda_handler(event, context):
    json_region = os.environ['AWS_REGION']
    return {
        "statusCode": 200,
        "headers": {
            "Content-Type": "application/json"
        },
        "body": json.dumps({
            "Region ": json_region
```

```
        })
    }
```

您可以使用任何事件資料來調用函數：

```
{
  "key1": "value1",
  "key2": "value2",
  "key3": "value3"
}
```

Lambda 執行時間會在初始化期間設定數個環境變數。如需在執行時間回應傳回的環境變數詳細資訊，請參閱使用 Lambda 環境變數來設定程式碼中的值。

此範例中的函數取決於調用 API (200 中) 的成功回應。如需有關調用 API 狀態的詳細資訊，請參閱調用回應語法。

傳回計算

下列範例顯示稱為 lambda_handler 的函數。該函數接受使用者輸入並將計算傳回給使用者。如需有關此範例的詳細資訊，請參閱aws-doc-sdk-examples GitHub 存放庫。

```
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)

def lambda_handler(event, context):
    ...
    result = None
    action = event.get('action')
    if action == 'increment':
        result = event.get('number', 0) + 1
        logger.info('Calculated result of %s', result)
    else:
        logger.error("%s is not a valid action.", action)

    response = {'result': result}
    return response
```

您可以使用下列事件資料來調用函數：

```
{
    "action": "increment",
    "number": 3
}
```

使用 .zip 封存檔部署 Python Lambda 函數

AWS Lambda 函數的程式碼包含一個包含函式處理常式程式碼的 .py 檔案，以及您的程式碼所依賴的任何其他套件和模組。若要將此函數程式碼部署到 Lambda，您可以使用部署套件。此套件可以是 .zip 封存檔或容器映像。如需搭配 Python 使用容器映像檔的詳細資訊，請參閱使用容器映像部署 Python Lambda 函數。

若要建立 .zip 封存檔的部署套件，您可以使用命令列工具的內建 .zip 封存檔公用程式，或任何其他 .zip 檔案公用程式 (例如 7zip)。以下各節顯示的範例假設您在 Linux 或 MacOS 環境中使用命令列 `zip` 工具。若要在 Windows 中使用相同命令，您可以安裝適用於 Linux 的 Windows 子系統，以取得 Ubuntu 和 Bash 的 Windows 整合版本。

請注意，Lambda 使用 POSIX 檔案許可，因此在建立 .zip 封存檔之前，您可能需要設定部署套件資料夾的許可。

主題

Python 中的執行期相依項

對於使用 Python 執行期的 Lambda 函數，相依項可以是任何 Python 套件或模組。使用 .zip 歸檔部署函數時，您可以使用函數程式碼將這些相依性新增至 .zip 檔案，或使用 Lambda 層。 圖層是單獨的 .zip 檔案，可以包含其他程式碼和內容。若要進一步了解如何在 Python 中使用 Lambda 圖層，請參閱the section called “圖層”。

Python Lambda 行階段包括 AWS SDK for Python (Boto3) 及其相依性。Lambda 會在執行期為您無法新增相依項的部署案例提供 SDK。這些案例包括使用內建程式碼編輯器在主控台中建立函數，或在 AWS Serverless Application Model (AWS SAM) 或 AWS CloudFormation 範本中使用內嵌函數。

Lambda 會定期更新 Python 執行期中的程式庫，以納入最新的更新和安全性修補程式。如果您的函數使用執行階段中包含的 Boto3 SDK 版本，但您的部署套件包含 SDK 相依項，則可能會導致版本不相

符問題。例如，您的部署套件可能包含 SDK 相依項 urllib3。Lambda 在執行期更新 SDK 時，執行期新版本與部署套件中 urllib3 版本之間的相容性問題可能會導致函數失敗。

> **Important**
>
> 為了保持對相依項的完全控制，並避免可能發生的版本不相容問題，建議您將函數的所有相依項新增至部署套件，即使這些相依項的版本包含在 Lambda 執行期中。這包括 Boto3 SDK。

若要瞭解您正在使用的執行階段中包含哪個版本的適用於 Python (Boto3) 的 SDK，請參閱。the section called “包含執行階段的 SDK 版本”

在 AWS 共同責任模式下，您負責管理函數部署套件中的任何相依項。這包括套用更新和安全性修補程式。若要更新函數部署套件中的相依項，請先建立新的 .zip 檔案，然後將其上傳至 Lambda。如需詳細資訊，請參閱 建立含相依項的 .zip 部署套件 和 使用 .zip 檔案建立及更新 Python Lambda 函數。

建立不含相依項的 .zip 部署套件

如果您的函數程式碼沒有相依項，則 .zip 檔案只會包含具有函數處理常式程式碼的 .py 檔案。使用您慣用的 zip 公用程式建立 .zip 檔案，並將 .py 檔案放在根目錄下。如果 .py 檔案不在 .zip 檔案的根目錄下，Lambda 將無法執行您的程式碼。

若要了解如何部署 .zip 檔案以建立新的 Lambda 函數或更新現有函數，請參閱使用 .zip 檔案建立及更新 Python Lambda 函數。

建立含相依項的 .zip 部署套件

如果您的函數程式碼依賴於其他套件或模組，您可以使用函數程式碼將這些相依性新增至 .zip 檔案，或使用 Lambda 層。本節中的指示說明如何在 .zip 部署套件中包含相依項。若要讓 Lambda 執行程式碼，您必須將含有處理常式程式碼和所有函數相依性的 .py 檔案安裝在 .zip 檔案的根目錄。

假設您的函數代碼儲存在名為 `lambda_function.py` 的檔案中。下列範例 CLI 命令會建立名為 `my_deployment_package.zip` 的 .zip 檔案，其中包含函數程式碼及其相依項。您可以將相依項直接安裝到專案目錄中的資料夾，也可以使用 Python 虛擬環境。

若要建立部署套件 (專案目錄)

1. 導覽至包含 `lambda_function.py` 原始程式碼檔案的專案目錄。在此範例中，目錄名為 `my_function`。

```
cd my_function
```

2. 建立名為 package 的新目錄，您將在其中安裝相依項。

```
mkdir package
```

請注意，對於 .zip 部署套件，Lambda 預期您的原始程式碼及其相依項全部位於 .zip 檔案的根目錄。不過，直接在專案目錄中安裝相依項可能會產生大量新檔案和資料夾，並使導覽 IDE 變得困難。您可以在此處建立單獨的 package 目錄，將您的相依項與原始程式碼分開。

3. 在 package 目錄中安裝您的相依項。以下範例使用 pip 從 Python Package Index 安裝 Boto3 SDK。如果您的函數程式碼使用您自己建立的 Python 套件，請將其儲存在 package 目錄中。

```
pip install --target ./package boto3
```

4. 建立 .zip 檔案，將已安裝的程式庫放在根目錄下。

```
cd package
zip -r ../my_deployment_package.zip .
```

這會在專案目錄中產生 my_deployment_package.zip 檔案。

5. 將 lambda_function.py 檔案新增至 .zip 檔案的根目錄

```
cd ..
zip my_deployment_package.zip lambda_function.py
```

您的 .zip 檔案應具有扁平的目錄結構，並將函數的處理常式程式碼和所有相依項資料夾安裝在根目錄下，如下所示。

```
my_deployment_package.zip
|- bin
|  |-jp.py
|- boto3
|  |-compat.py
|  |-data
|  |-docs
...
|- lambda_function.py
```

如果包含函數處理常式程式碼的 .py 檔案不在 .zip 檔案的根目錄下，Lambda 將無法執行您的程式碼。

若要建立部署套件 (虛擬環境)

1. 在您的專案目錄中建立並啟用虛擬環境。在此範例中，專案目錄名為 my_function。

```
~$ cd my_function
~/my_function$ python3.12 -m venv my_virtual_env
~/my_function$ source ./my_virtual_env/bin/activate
```

2. 使用 pip 安裝所需的程式庫。以下範例會安裝 Boto3 SDK

```
(my_virtual_env) ~/my_function$ pip install boto3
```

3. 使用 pip show 在在虛擬環境中找出 pip 安裝相依項的位置。

```
(my_virtual_env) ~/my_function$ pip show <package_name>
```

pip 安裝程式庫的資料夾可以命名為 site-packages 或 dist-packages。此資料夾可以位於 lib/python3.x 或 lib64/python3.x 目錄下 (其中 python3.x 代表您所使用的 Python 版本)。

4. 停用虛擬環境

```
(my_virtual_env) ~/my_function$ deactivate
```

5. 導覽至包含您使用 pip 所安裝相依項的目錄，在專案目錄中建立 .zip 檔案，並將安裝的相依項放在根目錄下。在此範例中，pip 已經在 my_virtual_env/lib/python3.12/site-packages 目錄中安裝了您的相依項。

```
~/my_function$ cd my_virtual_env/lib/python3.12/site-packages
~/my_function/my_virtual_env/lib/python3.12/site-packages$ zip -r ../../../../
my_deployment_package.zip .
```

6. 導覽至包含處理常式程式碼的 .py 檔案所在專案目錄的根目錄，並將該檔案新增至 .zip 套件的根目錄。在此範例中，您的函數程式碼檔案名稱為 lambda_function.py。

```
~/my_function/my_virtual_env/lib/python3.12/site-packages$ cd ../../../../
```

```
~/my_function$ zip my_deployment_package.zip lambda_function.py
```

相依項搜尋路徑和含執行期程式庫

當您在程式碼中使用 `import` 陳述式時，Python 執行期會在其搜尋路徑中搜尋目錄，直到找到模組或套件為止。依預設，執行期搜尋的第一個位置是 .zip 部署套件解壓縮並掛載的目錄 (`/var/task`)。如果您在部署套件中納入含執行期程式庫的版本，則您的版本的優先順序會高於執行期中包含的版本。部署套件中的相依項也優先於圖層中的相依項。

當您將相依項新增至層時，Lambda 會將其擷取到 `/opt/python/lib/python3.x/site-packages`(其中 `python3.x` 表示您所使用的執行期版本) 或 `/opt/python`。在搜尋路徑中，這些目錄的優先順序會高於包含含執行期程式庫和使用 pip 安裝的程式庫的目錄 (`/var/runtime` 和 `/var/lang/lib/python3.x/site-packages`)。因此，函數層中程式庫的優先順序高於執行期中包含的版本。

> **Note**
>
> 在 Python 3.11 託管運行時間和基本映像中， AWS SDK 及其依賴項安裝在`/var/lang/lib/python3.11/site-packages`目錄中。

您可以新增下列程式碼片段，以查看 Lambda 函數的完整搜尋路徑。

```
import sys

search_path = sys.path
print(search_path)
```

> **Note**
>
> 由於部署套件或圖層中的相依項優先於含執行期程式庫，因此如果您在套件中包含 urllib3 這類 SDK 相依項而不同時包含 SDK 的話，則可能會造成版本不相容問題。如果您部署自己的 Boto3 相依項版本，則還必須在部署套件中部署 Boto3 作為相依項。我們建議您封裝函數的所有相依項，即使其版本包含在執行期中。

您也可以在 .zip 套件內的個別資料夾中新增相依項。例如，您可以將 Boto3 SDK 的版本新增至名為 `common` 的 .zip 套件中的資料夾。解壓縮並掛載您的 .zip 套件時，此資料夾會放在 `/var/task` 目

錄中。若要在程式碼中使用來自 .zip 部署套件中資料夾的相依項，請使用 `import from` 陳述式。例如，若要使用來自 .zip 套件中名為 common 的資料夾的 Boto3 版本，請使用下列陳述式。

```
from common import boto3
```

使用 __pycache__ 資料夾

我們建議您不要在函數的部署套件中包含 `__pycache__` 資料夾。在架構或作業系統不同的建置機器上編譯的 Python 位元組程式碼可能與 Lambda 執行環境不相容。

建立含原生程式庫的 .zip 部署套件

如果您的函數只使用純 Python 套件和模組，您可以使用 `pip install` 命令在任何本機建置機器上安裝相依項，並建立 .zip 檔案。許多流行的 Python 庫（包括 NumPy 熊貓）都不是純粹的 Python，並且包含用 C 或 C ++ 編寫的代碼。將包含 C/C++ 程式碼的程式庫新增至部署套件時，必須正確建置套件，以確保套件與 Lambda 執行環境相容。

Python Package Index (PyPI)上提供的大多數套件都可以作為「wheel」(.whl 檔案)。.whl 檔案是一種 ZIP 檔案，其中的內建發佈包含針對特定作業系統和指令集架構預先編譯的二進位檔。若要讓您的部署套件與 Lambda 相容，請安裝適用於 Linux 作業系統和您函數指令集架構的 wheel。

某些套件可能只能作為原始檔發佈。對於這些套件，您需要自行編譯和建置 C/C++ 元件。

若要查看所需套件可用的發佈，請執行以下操作：

1. 在 Python Package Index 主頁上搜尋套件名稱。
2. 選擇您要使用的套件版本。
3. 選擇下載檔案。

使用內建發佈 (wheel)

若要下載與 Lambda 相容的 wheel，請使用 pip `--platform` 選項。

如果您的 Lambda 函數使用 x86_64 指令集架構，請執行下列 `pip install` 命令，在 `package` 目錄中安裝相容的 wheel。以您所使用的 Pytho 執行期版本取代 `--python 3.x`。

```
pip install \
--platform manylinux2014_x86_64 \
--target=package \
--implementation cp \
```

```
--python-version 3.x \
--only-binary=:all: --upgrade \
<package_name>
```

如果您的函數使用 arm64 指令集架構，請執行下列命令。以您所使用的 Pytho 執行期版本取代 `--python 3.x`。

```
pip install \
--platform manylinux2014_aarch64 \
--target=package \
--implementation cp \
--python-version 3.x \
--only-binary=:all: --upgrade \
<package_name>
```

使用原始檔發佈

如果您的套件只能作為原始檔發佈，則需要自行建置 C/C++ 程式庫。若要讓您的套件與 Lambda 執行環境相容，您需要在使用相同 Amazon Linux 2 作業系統的環境中建置套件。您可以在 Amazon EC2 Linux 執行個體中建置套件來執行此操作。

若要了解如何啟動和連接到 Amazon EC2 Linux 執行個體，請參閱《適用於 Linux 執行個體的 Amazon EC2 使用者指南》中的教學課程：Amazon EC2 Linux 執行個體入門。

使用 .zip 檔案建立及更新 Python Lambda 函數

建立 .zip 部署套件後，您可以使用該套件建立新的 Lambda 函數或更新現有函數。您可以使用 Lambda 主控台、和 Lambda API 來部署您的 .zip 套件。 AWS Command Line Interface您也可以使用 AWS Serverless Application Model (AWS SAM) 和 AWS CloudFormation 建立並更新 Lambda 函數。

Lambda 的 .zip 部署套件大小上限為 250 MB (解壓縮)。請注意，此限制適用於您上傳的所有檔案 (包括任何 Lambda 層) 的大小總和。

Lambda 執行時間需有許可才能讀取部署套裝服務中的檔案。在 Linux 許可八進位標記法中，Lambda 需要 644 個許可 (rw-r--r--) 用於非可執行檔，以及 755 個許可 (rwxr-x) 用於目錄和可執行檔。

在 Linux 和 MacOS 中，使用 `chmod` 命令變更部署套件中檔案和目錄的檔案許可。例如，若要提供可執行檔正確的許可，請執行下列命令。

```
chmod 755 <filepath>
```

若要在 Windows 中變更檔案許可，請參閱 Microsoft Windows 文件的 Set, View, Change, or Remove Permissions on an Object。

透過主控台使用 .zip 檔案建立及更新函數

若要建立新函數，您必須先在主控台中建立函數，然後上傳您的 .zip 封存檔。若要更新現有函數，請開啟函數的頁面，然後按照同樣的程序新增更新後的 .zip 檔案。

如果您的 .zip 檔案小於 50 MB，您可以透過直接從本機電腦上傳檔案來建立或更新函數。若 .zip 檔案大於 50 MB，您必須先將套件上傳至 Amazon S3 儲存貯體。如需有關如何使用將檔案上傳到 Amazon S3 儲存貯體的指示 AWS Management Console，請參閱開始使用 Amazon S3。若要使用上載檔案 AWS CLI，請參閱《使用指南》中的AWS CLI 〈移動物件〉。

> **Note**
>
> 您無法變更現有函數的部署套件類型 (.zip 或容器映像檔)。例如，您無法將容器映像函數轉換為使用 .zip 檔案封存。您必須建立新的函數。

若要建立新的函數 (主控台)

1. 開啟 Lambda 主控台的函數頁面，然後選擇建立函數。
2. 選擇 Author from scratch (從頭開始撰寫)。
3. 在基本資訊下，請執行下列動作：
   a. 在函數名稱中輸入函數名稱。
   b. 在執行期中選取要使用的執行期。
   c. (選用) 在架構中選擇要用於函數的指令集架構。預設架構值為 x86_64。請確定函數的 .zip 部署套件與您選取的指令集架構相容。
4. (選用) 在 許可 下，展開 變更預設執行角色 。您可建立新的執行角色，或使用現有的角色。
5. 選擇建立函數。Lambda 會使用您選擇的執行期建立一個基本的「Hello world」函數。

若要從本機電腦上傳 .zip 封存檔 (主控台)

1. 在 Lambda 主控台的函數頁面中選擇要上傳 .zip 檔案的函數。
2. 選取程式碼索引標籤。
3. 在程式碼來源窗格中選擇上傳來源。

4. 選擇 .zip 檔案。
5. 若要上傳 .zip 檔案，請執行下列操作：
   a. 選擇上傳，然後在檔案選擇器中選取您的 .zip 檔案。
   b. 選擇 Open (開啟)。
   c. 選擇儲存。

若要從 Amazon S3 儲存貯體上傳 .zip 封存檔 (控制台)

1. 在 Lambda 主控台的函數頁面中選擇要上傳新 .zip 檔案的函數。
2. 選取程式碼索引標籤。
3. 在程式碼來源窗格中選擇上傳來源。
4. 選擇 Amazon S3 位置。
5. 貼上 .zip 檔案的 Amazon S3 連結 URL，然後選擇儲存。

使用主控台程式碼編輯器更新 .zip 檔案函數

對於某些具有 .zip 部署套件的函數，您可以使用 Lambda 主控台的內建程式碼編輯器直接更新函數程式碼。若要使用此功能，您的函數必須符合下列條件：

- 您的函數必須使用其中一種轉譯語言執行期 (Python、Node.js 或 Ruby)
- 函數的部署套件必須小於 3MB。

具有容器映像部署套件之函數的函數程式碼無法直接在主控台中編輯。

若要使用主控台程式碼編輯器更新函數程式碼

1. 開啟 Lambda 主控台的函數頁面，然後選取您的函數。
2. 選取程式碼索引標籤。
3. 在程式碼來源窗格中，選取您的原始程式碼檔案，然後在整合式程式碼編輯器中加以編輯。
4. 完成編輯程式碼後，請選擇部署，以儲存變更並更新函數。

使用 .zip 檔案建立和更新函數 AWS CLI

您可以使用 AWS CLI 建立新函數，或使用 .zip 檔案更新現有函數。使用 create-function 和 update-function-code 命令來部署您的 .zip 套件。如果您的 .zip 檔案小於 50 MB，則可以從本機建置電腦的檔案位置上傳 .zip 套件。若檔案較大，則必須先從 Amazon S3 儲存貯體上傳 .zip 套件。如需有關如何使用將檔案上傳到 Amazon S3 儲存貯體的指示 AWS CLI，請參閱使用AWS CLI 者指南中的移動物件。

> **Note**
>
> 如果您使用從 Amazon S3 儲存貯體上傳 .zip 檔案 AWS CLI，則該儲存貯體必須與您的函數位於 AWS 區域 相同的位置。

若要使用 .zip 檔案與建立新函數 AWS CLI，您必須指定下列項目：

- 函數名稱 (`--function-name`)
- 函數的執行期 (`--runtime`)
- 函數執行角色的 Amazon Resource Name (ARN) (`--role`)
- 函數程式碼中處理常式方法的名稱 (`--handler`)

您也必須指定 .zip 檔案的位置。如果您的 .zip 檔案位於本機建置電腦上的資料夾中，請使用 `--zip-file` 選項來指定檔案路徑，如下列範例命令所示。

```
aws lambda create-function --function-name myFunction \
--runtime python3.12 --handler lambda_function.lambda_handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--zip-file fileb://myFunction.zip
```

若要在 Amazon S3 儲存貯體中指定 .zip 檔案的位置，請使用如下列範例命令所示的 `--code` 選項。您只需針對版本控制的物件使用 `S3ObjectVersion` 參數。

```
aws lambda create-function --function-name myFunction \
--runtime python3.12 --handler lambda_function.lambda_handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--code S3Bucket=DOC-EXAMPLE-BUCKET,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion
```

若要使用 CLI 更新現有函數，您可以使用 `--function-name` 參數指定函數的名稱。您也必須指定要用來更新函數程式碼的 .zip 檔案的位置。如果您的 .zip 檔案位於本機建置電腦上的資料夾中，請使用 `--zip-file` 選項來指定檔案路徑，如下列範例命令所示。

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

若要在 Amazon S3 儲存貯體中指定 .zip 檔案的位置，請使用如下列範例命令所示的 `--s3-bucket` 和 `--s3-key` 選項。您只需針對版本控制的物件使用 `--s3-object-version` 參數。

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket DOC-EXAMPLE-BUCKET --s3-key myFileName.zip --s3-object-version myObject
 Version
```

透過 Lambda API 使用 .zip 檔案建立及更新函數

若要使用 .zip 封存檔建立及更新函數，請使用下列 API 操作：

- CreateFunction
- UpdateFunction代碼

使用 .zip 文件創建和更新函數 AWS SAM

AWS Serverless Application Model (AWS SAM) 是一個工具組，可協助簡化在 AWS上建置和執行無伺服器應用程式的程序。您可以在 YAML 或 JSON 範本中定義應用程式的資源，並使用 AWS SAM 命令列介面 (AWS SAM CLI) 來建置、封裝及部署應用程式。當您從 AWS SAM 範本建立 Lambda 函數時， AWS SAM 會使用函數程式碼和您指定的任何相依性，自動建立 .zip 部署套件或容器映像檔。若要進一步了解如 AWS SAM 何使用建置和部署 Lambda 函數，請參閱開AWS Serverless Application Model發人員指南 AWS SAM中的入門使用。

您也可以使用現有的 .zip 檔案封存 AWS SAM 來建立 Lambda 函數。若要使用建立 Lambda 函數 AWS SAM，您可以將 .zip 檔案儲存在 Amazon S3 儲存貯體或建置機器的本機資料夾中。如需有關如何使用將檔案上傳到 Amazon S3 儲存貯體的指示 AWS CLI，請參閱使用AWS CLI 者指南中的移動物件。

在 AWS SAM 範本中，`AWS::Serverless::Function`資源會指定您的 Lambda 函數。在本資源中設定下列屬性，以使用 .zip 封存檔建立函數：

- `PackageType`：設定為 `Zip`
- `CodeUri`-設定為函數程式碼的 Amazon S3 URI、本機資料夾的路徑或FunctionCode物件
- `Runtime`：設定為所選執行期

使用時 AWS SAM，如果您的 .zip 檔案大於 50MB，則不需要先將其上傳到 Amazon S3 儲存貯體。AWS SAM 可以從本地構建機器上的位置上傳 .zip 軟件包，最大允許大小為 250MB（解壓縮）。

若要進一步瞭解如何使用 .zip 檔案部署函數 AWS SAM，請參閱AWS SAM 開發人員指南AWS::Serverless::Function中的。

使用 .zip 文件創建和更新函數 AWS CloudFormation

您可以使 AWS CloudFormation 用 .zip 檔案封存來建立 Lambda 函數。若要使用 .zip 檔案建立 Lambda 函數，您必須先將檔案上傳至 Amazon S3 儲存貯體。如需有關如何使用將檔案上傳到 Amazon S3 儲存貯體的指示 AWS CLI，請參閱使用AWS CLI 者指南中的移動物件。

對於 Node.js 和 Python 運行時，您還可以在 AWS CloudFormation 模板中提供內聯源代碼。 AWS CloudFormation 然後在構建函數時創建一個包含代碼的 .zip 文件。

使用現有的 .zip 檔案

在 AWS CloudFormation 範本中，`AWS::Lambda::Function`資源會指定您的 Lambda 函數。在本資源中設定下列屬性，以使用 .zip 封存檔建立函數：

- `PackageType`：設定為 `Zip`
- `Code`：在 `S3Bucket` 和 `S3Key` 欄位中輸入 Amazon S3 儲存貯體名稱和 .zip 檔案名稱。
- `Runtime`：設定為所選執行期

從內嵌程式碼建立 .zip 檔案

您可以在 AWS CloudFormation 模板中聲明用 Python 或 Node.js 內聯編寫的簡單函數。由於程式碼內嵌在 YAML 或 JSON 中，因此您無法將任何外部相依項新增至部署套件。這意味著您的函數必須使用運行時中包含的 AWS SDK 版本。範本的需求 (例如必須逸出某些字元) 也會讓使用 IDE 的語法檢查和程式碼完成功能變得更加困難。也就是說，您的範本可能需要進行其他測試。由於這些限制，內聯聲明函數最適合不經常更改的非常簡單的代碼。

若要從 Node.js 和 Python 執行期的內嵌程式碼建立 .zip 檔案，請在範本的 `AWS::Lambda::Function` 資源中設定下列屬性：

- `PackageType`：設定為 `Zip`
- `Code`：在 `ZipFile` 欄位中輸入您的函數程式碼
- `Runtime`：設定為所選執行期

AWS CloudFormation 產生的 .zip 檔案不能超過 4MB。若要進一步瞭解有關使用 .zip 檔案部署函數的更多資訊 AWS CloudFormation，請參閱使用AWS CloudFormation 者指南AWS::Lambda::Function中的。

使用容器映像部署 Python Lambda 函數

您可以透過三種方式為 Python Lambda 函數建置容器映像：

- 使用 Python 的 AWS 基本圖像

  AWS 基礎映像會預先載入語言執行期、用來管理 Lambda 與函數程式碼之間互動的執行期界面用戶端，以及用於本機測試的執行期界面模擬器。
- 使用 AWS 僅限作業系統的基本影像

  AWS 僅限作業系統的基本映像檔包含 Amazon Linux 散發和執行階段介面模擬器。這些映像常用於為編譯語言 (如 Go 和 Rust) 和 Lambda 不提供基礎映像的語言或語言版本 (如 Node.js 19) 建置容器映像。您還可以使用僅限作業系統的基礎映像來實作自訂執行期。若要使映像與 Lambda 相容，您必須在映像中加入適用於 Python 的執行期介面用戶端。
- 使用非AWS 基本圖像

  您可以使用其他容器登錄檔中的替代基礎映像 (例如 Alpine Linux 或 Debian)。您也可以使用組織建立的自訂映像。若要使映像與 Lambda 相容，您必須在映像中加入適用於 Python 的執行期介面用戶端。

> **Tip**
>
> 若要縮短 Lambda 容器函數變成作用中狀態所需的時間，請參閱 Docker 文件中的使用多階段建置。若要建置有效率的容器映像，請遵循撰寫 Dockerfiles 的最佳實務。

本頁面會說明如何為 Lambda 建置、測試和部署容器映像。

主題

AWS Python 的基本圖像

AWS 為 Python 提供了以下基本圖像：

標籤	執行期	作業系統	Dockerfile	棄用
3.12	Python 3.12	Amazon Linux 2023	碼頭文件對於 Python 3.12 GitHub	
3.11	Python 3.11	Amazon Linux 2	碼頭文件對於 Python 3.11 GitHub	
3.10	Python 3.10	Amazon Linux 2	碼頭文件對於 Python 3.10 GitHub	
3.9	Python 3.9	Amazon Linux 2	碼頭文件對於 Python 3.9 GitHub	
3.8	Python 3.8	Amazon Linux 2	碼頭文件 Python 3.8 上 GitHub	2024 年 10 月 14 日

Amazon ECR 儲存庫：gallery.ecr.aws/lambda/python

Python 3.12 及更高版本的基本圖像基於 Amazon Linux 2023 最小容器映像。基 Python 圖像是基於 Amazon Linux 2 圖像。與 Amazon Linux 2 相比，以 AL2023 為基礎的映像提供了許多優勢，包括較小的部署佔用空間和更新版本的程式庫，例如. `glibc`

基於 Al2023 的圖像使用`microdnf`（符號鏈接為`dnf`）作為軟件包管理器而不是`yum`，這是 Amazon Linux 2 中的默認軟件包管理器。 `microdnf`是的獨立實作`dnf`。如需 AL2023 映像檔中包含的套件清單，請參閱比較 Amazon Linux 2023 容器映像上安裝的套件中的最小容器欄。如需有關 AL2023 和 Amazon Linux 2 之間差異的詳細資訊，請參閱 AWS 運算部落格 AWS Lambda上的介紹 Amazon Linux 2023 執行階段。

> **Note**
>
> 要在本地運行基於 AL2023 的映像，包括使用 AWS Serverless Application Model （ AWS SAM ），您必須使用碼頭版本 20.10.10 或更高版本。

基礎映像中的相依性搜尋路徑

當您在程式碼中使用 `import` 陳述式時，Python 執行期會在其搜尋路徑中搜尋目錄，直到找到模組或套件為止。在預設情況下，執行期會先搜尋 `{LAMBDA_TASK_ROOT}` 目錄。如果您在映像中納入含執行期程式庫的版本，則此版本的優先順序會高於執行期中包含的版本。

搜尋路徑中包含的其他步驟取決於您使用的 Python Lambda 基礎映像版本：

- Python 3.11 及更高版本：含執行期的程式庫和使用 pip 安裝的程式庫已安裝在 `/var/lang/lib/python3.11/site-packages` 目錄。此目錄的優先順序會高於搜尋路徑中的 `/var/runtime`。您可以使用 pip 安裝更新的版本來覆寫 SDK。您可以使用 pip 來確認含執行期的 SDK 及其相依性是否與您安裝的任何套件相容。
- Python 3.8-3.10：含執行期的程式庫已安裝在 `/var/runtime` 目錄。使用 pip 安裝的程式庫已安裝在 `/var/lang/lib/python3.x/site-packages` 目錄。`/var/runtime` 目錄的優先順序會高於搜尋路徑中的 `/var/lang/lib/python3.x/site-packages`。

您可以新增下列程式碼片段，以查看 Lambda 函數的完整搜尋路徑。

```
import sys

search_path = sys.path
print(search_path)
```

使用 Python 的 AWS 基本圖像

必要條件

若要完成本節中的步驟，您必須執行下列各項：

- AWS Command Line Interface (AWS CLI) 第二版
- 碼頭工人（Python 3.12 及更高版本的基本圖像的最低版本 20.10.10）
- Python

從基礎映像建立映像

若要從 Python 的 AWS 基本映像檔建立容器映像檔

1. 建立專案的目錄，然後切換至該目錄。

```
mkdir example
cd example
```

2. 建立稱為 lambda_function.py 的新檔案。您可以將下列範例函數程式碼新增至檔案進行測試，或使用您自己的函數程式碼。

   Example Python 函數

```
import sys
def handler(event, context):
    return 'Hello from AWS Lambda using Python' + sys.version + '!'
```

3. 建立稱為 requirements.txt 的新檔案。如果您使用上一個步驟的範例函數程式碼，請將檔案保留空白，因為沒有任何相依項。否則，請列出每個所需的程式庫。例如，如果您的函數使用 AWS SDK for Python (Boto3)，您的 requirements.txt 看起來應該像這樣：

   Example requirements.txt

```
boto3
```

4. 建立包含下列組態的新 Dockerfile。

   - 將 FROM 屬性設定為基礎映像的 URI。
   - 使用 COPY 指令{LAMBDA_TASK_ROOT}，將函數程式碼和執行階段相依性複製到 Lambda 定義的環境變數。
   - 將 CMD 引數設定為 Lambda 函數處理常式。

   Example Dockerfile

```
FROM public.ecr.aws/lambda/python:3.12

# Copy requirements.txt
COPY requirements.txt ${LAMBDA_TASK_ROOT}

# Install the specified packages
RUN pip install -r requirements.txt

# Copy function code
COPY lambda_function.py ${LAMBDA_TASK_ROOT}
```

```
# Set the CMD to your handler (could also be done as a parameter override outside
 of the Dockerfile)
CMD [ "lambda_function.handler" ]
```

5. 使用 docker build 命令建立 Docker 映像檔。以下範例將映像命名為 docker-image 並為其提供 test 標籤。

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> 此命令會指定 --platform linux/amd64 選項，確保無論建置機器的架構為何，您的容器都與 Lambda 執行環境相容。如果您打算使用 ARM64 指令集架構建立 Lambda 函數，務必將命令變更為改用 --platform linux/arm64 選項。

(選用) 在本機測試映像

1. 使用 docker run 命令啟動 Docker 影像。在此範例中，docker-image 為映像名稱，test 為標籤。

```
docker run --platform linux/amd64 -p 9000:8080 docker-image:test
```

此命令將映像作為容器執行，並在 localhost:9000/2015-03-31/functions/function/invocations 建立本機端點。

> **Note**
>
> 如果您為 ARM64 指令集架構建立 Docker 映像檔，請務必將--platform linux/*arm64*選項改用選項。--platform linux/*amd64*

2. 從新的終端機視窗，將事件張貼至本機端點。

Linux/macOS

在 Linux 或 macOS 中，執行下列 curl 命令：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

在中 PowerShell，執行下列Invoke-WebRequest命令：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

3. 取得容器 ID。

```
docker ps
```

4. 使用 docker kill 命令停止容器。在此命令中，將 3766c4ab331c 替換為上一步驟中的容器 ID。

```
docker kill 3766c4ab331c
```

部署映像

若要將映像上傳至 Amazon ECR 並建立 Lambda 函數

1. 使用 get-login-password 命令，向 Amazon ECR 登錄檔驗證 Docker CLI。

   - 將--region值設定為您 AWS 區域 要建立 Amazon ECR 儲存庫的位置。
   - 111122223333用您的 AWS 帳戶 ID 替換。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 使用 create-repository 命令在 Amazon ECR 中建立儲存庫。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 儲存庫必須與 Lambda 函數位於 AWS 區域 相同。

如果成功，您將會看到以下回應：

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. 從上一步驟的輸出中複製 `repositoryUri`。
4. 執行 docker tag 命令，將 Amazon ECR 儲存庫中的本機映像標記為最新版本。在此命令中：
   - 將 `docker-image:test` 替換為 Docker 映像檔的名稱和標籤。

- 將 <ECRrepositoryUri> 替換為複製的 repositoryUri。確保在 URI 的末尾包含 :latest。

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

範例：

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. 執行 docker push 命令，將本機映像部署至 Amazon ECR 儲存庫。確保在儲存庫 URI 的末尾包含 :latest。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 建立函數的執行角色 (若您還沒有的話)。在下一個步驟中您需要角色的 Amazon Resource Name (ARN)。
7. 建立 Lambda 函數。對於 ImageUri，從之前的設定中指定儲存庫 URI。確保在 URI 的末尾包含 :latest。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> 只要映像與 Lambda 函數位於相同的區域，您就可以使用不同 AWS 帳戶中的映像檔建立函數。如需詳細資訊，請參閱 Amazon ECR 跨帳戶許可。

8. 調用函數。

```
aws lambda invoke --function-name hello-world response.json
```

您應該看到如下回應：

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 若要查看函數的輸出，請檢查 `response.json` 檔案。

若要更新函數程式碼，您必須再次建置映像、將新映像上傳到 Amazon ECR 存放庫，然後使用 update-function-code 命令將映像部署到 Lambda 函數。

Lambda 將圖像標記解析為特定的圖像摘要。這表示如果您將用於部署函數的映像標籤指向 Amazon ECR 中的新映像，Lambda 不會自動更新函數以使用新映像。若要將新映像部署到相同的 Lambda 函數，您必須使用`update-function-code`命令，即使 Amazon ECR 中的影像標籤保持不變。

透過執行期介面用戶端使用替代基礎映像

如果您使用僅限作業系統的基礎映像或替代的基礎映像，則必須在映像中加入執行期介面用戶端。執行期介面用戶端會讓您擴充 Lambda 執行階段 API，管理 Lambda 與函數程式碼之間的互動。

使用 pip 套件管理員安裝 Python 執行期介面用戶端。

```
pip install awslambdaric
```

您也可以從下載 Python 執行階段介面用戶端 GitHub。

下面的例子演示了如何使用非AWS 基本圖像構建 Python 的容器映像。範例 Dockerfile 使用官方 Python 基礎映像。Dockerfile 包含 Python 執行期界面用戶端。

必要條件

若要完成本節中的步驟，您必須執行下列各項：

- AWS Command Line Interface (AWS CLI) 第二版
- Docker
- Python

使用替代基礎映像建立映像

若要從非AWS 基本影像建立容器映像檔

1. 建立專案的目錄，然後切換至該目錄。

```
mkdir example
cd example
```

2. 建立稱為 `lambda_function.py` 的新檔案。您可以將下列範例函數程式碼新增至檔案進行測試，或使用您自己的函數程式碼。

   Example Python 函數

```
import sys
def handler(event, context):
    return 'Hello from AWS Lambda using Python' + sys.version + '!'
```

3. 建立稱為 `requirements.txt` 的新檔案。如果您使用上一個步驟的範例函數程式碼，請將檔案保留空白，因為沒有任何相依項。否則，請列出每個所需的程式庫。例如，如果您的函數使用 AWS SDK for Python (Boto3)，您的 `requirements.txt` 看起來應該像這樣：

   Example requirements.txt

```
boto3
```

4. 建立新的 Dockerfile。下列 Dockerfile 使用官方 Python 基礎映像，而非 AWS 基礎映像。Dockerfile 包含執行期介面用戶端，可讓映像與 Lambda 相容。下列範例 Dockerfile 使用多階段建置。

   - 將 `FROM` 屬性設定為基礎映像。
   - 將 `ENTRYPOINT` 設為您希望 Docker 容器在啟動時執行的模組。在此案例中，模組是執行期界面用戶端。
   - 將 `CMD` 設定為 Lambda 函數處理常式。

   Example Dockerfile

```
# Define custom function directory
ARG FUNCTION_DIR="/function"
```

```
FROM python:3.12 as build-image

# Include global arg in this stage of the build
ARG FUNCTION_DIR

# Copy function code
RUN mkdir -p ${FUNCTION_DIR}
COPY . ${FUNCTION_DIR}

# Install the function's dependencies
RUN pip install \
    --target ${FUNCTION_DIR} \
        awslambdaric

# Use a slim version of the base Python image to reduce the final image size
FROM python:3.12-slim

# Include global arg in this stage of the build
ARG FUNCTION_DIR
# Set working directory to function root directory
WORKDIR ${FUNCTION_DIR}

# Copy in the built dependencies
COPY --from=build-image ${FUNCTION_DIR} ${FUNCTION_DIR}

# Set runtime interface client as default command for the container runtime
ENTRYPOINT [ "/usr/local/bin/python", "-m", "awslambdaric" ]
# Pass the name of the function handler as an argument to the runtime
CMD [ "lambda_function.handler" ]
```

5. 使用 docker build 命令建立 Docker 映像檔。以下範例將映像命名為 `docker-image` 並為其提供 `test` 標籤。

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> 此命令會指定 `--platform linux/amd64` 選項，確保無論建置機器的架構為何，您的容器都與 Lambda 執行環境相容。如果您打算使用 ARM64 指令集架構建立 Lambda 函數，務必將命令變更為改用 `--platform linux/arm64` 選項。

(選用) 在本機測試映像

使用 執行期界面模擬器 以在本機測試映像。您可以將模擬器構建到映像中，也可以使用以下步驟將其安裝在本地計算機上。

若要在本機電腦上安裝並執行執行期介面模擬器

1. 從您的項目目錄中運行以下命令以下載運行時接口仿真器（x86-64 架構） GitHub 並將其安裝在本地計算機上。

   Linux/macOS

   ```
   mkdir -p ~/.aws-lambda-rie && \
       curl -Lo ~/.aws-lambda-rie/aws-lambda-rie https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie && \
       chmod +x ~/.aws-lambda-rie/aws-lambda-rie
   ```

   要安裝 arm64 模擬器，請使用以下命令替換上一個命令中的 GitHub 存儲庫 URL：

   ```
   https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
   ```

   PowerShell

   ```
   $dirPath = "$HOME\.aws-lambda-rie"
   if (-not (Test-Path $dirPath)) {
       New-Item -Path $dirPath -ItemType Directory
   }

   $downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie"
   $destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
   Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
   ```

   若要安裝 arm64 模擬器，請將 `$downloadLink` 更換為下列項目：

   ```
   https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
   ```

2. 使用 docker run 命令啟動 Docker 影像。注意下列事項：

- `docker-image` 是映像名稱，而 `test` 是標籤。
- `/usr/local/bin/python -m awslambdaric lambda_function.handler` 是 Dockerfile 中的 `ENTRYPOINT`，後面接著 `CMD`。

Linux/macOS

```
docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p
 9000:8080 \
    --entrypoint /aws-lambda/aws-lambda-rie \
    docker-image:test \
        /usr/local/bin/python -m awslambdaric lambda_function.handler
```

PowerShell

```
docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p
 9000:8080 `
--entrypoint /aws-lambda/aws-lambda-rie `
docker-image:test `
    /usr/local/bin/python -m awslambdaric lambda_function.handler
```

此命令將映像作為容器執行，並在 `localhost:9000/2015-03-31/functions/function/invocations` 建立本機端點。

> **Note**
>
> 如果您為 ARM64 指令集架構建立 Docker 映像檔，請務必將`--platform linux/arm64`選項改用選項。`--platform linux/amd64`

3. 將事件張貼至本機端點。

Linux/macOS

在 Linux 或 macOS 中，執行下列 `curl` 命令：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

在中 PowerShell，執行下列Invoke-WebRequest命令：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

4. 取得容器 ID。

```
docker ps
```

5. 使用 docker kill 命令停止容器。在此命令中，將 3766c4ab331c 替換為上一步驟中的容器 ID。

```
docker kill 3766c4ab331c
```

部署映像

若要將映像上傳至 Amazon ECR 並建立 Lambda 函數

1. 使用 get-login-password 命令，向 Amazon ECR 登錄檔驗證 Docker CLI。

   - 將--region值設定為您 AWS 區域 要建立 Amazon ECR 儲存庫的位置。
   - 111122223333用您的 AWS 帳戶 ID 替換。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 使用 create-repository 命令在 Amazon ECR 中建立儲存庫。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 儲存庫必須與 Lambda 函數位於 AWS 區域 相同。

如果成功，您將會看到以下回應：

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. 從上一步驟的輸出中複製 `repositoryUri`。
4. 執行 docker tag 命令，將 Amazon ECR 儲存庫中的本機映像標記為最新版本。在此命令中：
   - 將 `docker-image:test` 替換為 Docker 映像檔的名稱和標籤。

- 將 <ECRrepositoryUri> 替換為複製的 repositoryUri。確保在 URI 的末尾包含 :latest。

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

範例：

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. 執行 docker push 命令，將本機映像部署至 Amazon ECR 儲存庫。確保在儲存庫 URI 的末尾包含 :latest。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 建立函數的執行角色 (若您還沒有的話)。在下一個步驟中您需要角色的 Amazon Resource Name (ARN)。
7. 建立 Lambda 函數。對於 ImageUri，從之前的設定中指定儲存庫 URI。確保在 URI 的末尾包含 :latest。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> 只要映像與 Lambda 函數位於相同的區域，您就可以使用不同 AWS 帳戶中的映像檔建立函數。如需詳細資訊，請參閱 Amazon ECR 跨帳戶許可。

8. 調用函數。

```
aws lambda invoke --function-name hello-world response.json
```

您應該看到如下回應：

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 若要查看函數的輸出，請檢查 `response.json` 檔案。

若要更新函數程式碼，您必須再次建置映像、將新映像上傳到 Amazon ECR 存放庫，然後使用 update-function-code 命令將映像部署到 Lambda 函數。

Lambda 將圖像標記解析為特定的圖像摘要。這表示如果您將用於部署函數的映像標籤指向 Amazon ECR 中的新映像，Lambda 不會自動更新函數以使用新映像。若要將新映像部署到相同的 Lambda 函數，您必須使用`update-function-code`命令，即使 Amazon ECR 中的影像標籤保持不變。

如需如何從 Alpine 基礎映像中建立 Python 映像的範例，請參閱 AWS 部落格上的Lambda 的容器映像支援。

使用 Python 函數的圖層

Lambda 層是包含補充代碼或數據的 .zip 文件歸檔。層通常具備程式庫相依性、自訂執行期或組態檔案。建立圖層包含三個一般步驟：

1. Package 圖層內容。這意味著創建一個包含要在函數中使用的依賴關係的 .zip 文件歸檔。
2. 在 Lambda 中建立圖層。
3. 將圖層添加到您的函數中。

本主題包含如何使用外部程式庫相依性正確封裝和建立 Python Lambda 層的步驟和指導。

主題

必要條件

若要遵循本節中的步驟，您必須具備下列項目：

- Python 3.11 和點子包安裝程序
- AWS Command Line Interface (AWS CLI) 第二版

在本主題中，我們會參考 awsdocs GitHub 儲存庫上的`layer-python`範例應用程式。此應用程序包含下載依賴關係並生成圖層的腳本。該應用程序還包含使用圖層中的依賴關係的相應功能。創建圖層後，您可以部署和調用相應的功能以驗證一切正常。由於您對函數使用 Python 3.11 執行階段，因此這些圖層也必須與 Python 3.11 相容。

在範`layer-python`例應用程式中，有兩個範例：

- 第一個範例涉及將requests程式庫封裝到 Lambda 層中。該layer/目錄包含用於生成圖層的腳本。目function/錄包含範例函數，可協助測試圖層是否有效。本自學課程的大部分內容將逐步介紹如何建立和封裝此圖層。
- 第二個範例涉及將numpy程式庫封裝到 Lambda 層中。該layer-numpy/目錄包含用於生成圖層的腳本。目function-numpy/錄包含範例函數，可協助測試圖層是否有效。如需如何建立和封裝此圖層的範例，請參閱the section called “使用manylinux車輪分佈”。

與 Amazon Linux 的 Python 層兼容性

建立層的第一步是將所有層內容綁定至 .zip 封存檔。由於 Lambda 函數是在 Amazon Linux 上執行，因此您的層內容必須能夠在 Linux 環境中編譯和建置。

在 Python 中，除了源代碼分發之外，大多數軟.whl件包還可以作為輪子（文件）使用。每個輪子都是一種內置發行版，支持 Python 版本，操作系統和機器指令集的特定組合。

車輪對於確保您的圖層與 Amazon Linux 兼容非常有用。當您下載依賴關係時，請盡可能下載通用輪。(依預設，pip安裝通用輪 (如果有的話)。通用滾輪包含any作為平台標籤，表示它與包括 Amazon Linux 在內的所有平台兼容。

在以下範例中，您將程式requests庫封裝到 Lambda 層中。該requests庫是可作為通用輪子使用的軟件包的一個示例。

並非所有 Python 軟件包都作為通用輪子分發。例如，numpy具有多個輪子分佈，每個輪子分佈都支援不同的平台集。對於這類軟件包，請下載manylinux分發以確保與 Amazon Linux 的兼容性。如需如何封裝此類圖層的詳細說明，請參閱the section called “使用manylinux車輪分佈”。

在極少數情況下，Python 包可能無法作為輪子使用。如果只有來源分發 (sdist) 存在，我們建議您根據 Amazon Linux 2023 基本容器映像在 Docker 環境中安裝並封裝相依性。如果您想要包含使用其他語言 (例如 C/C++) 撰寫的自訂程式庫，我們也建議您使用這種方法。這個方法會模仿碼頭視窗中的 Lambda 執行環境，並確保您的非 Python 套件相依性與 Amazon Linux 相容。

Python 執行階段的圖層路徑

將層新增至函數時，Lambda 會將層內容載入該執行環境的 /opt 目錄。在每一次 Lambda 執行期中，PATH 變數已包含 /opt 目錄中的特定資料夾路徑。若要確保PATH變數會擷取圖層內容，您的圖層 .zip 檔案應該在下列資料夾路徑中具有其相依性：

- python

- python/lib/python3.*x*/site-packages

例如，您在此自學課程中建立的產生圖層 .zip 檔案具有以下目錄結構：

```
layer_content.zip
# python
    # lib
        # python3.11
            # site-packages
                # requests
                # <other_dependencies> (i.e. dependencies of the requests package)
                # ...
```

資requests源庫已正確定位於目python/lib/python3.11/site-packages錄中。這可確保 Lambda 可以在函數叫用期間找到程式庫。

封裝圖層內容

在此範例中，您將 Python requests 程式庫封裝在圖層 .zip 檔案中。完成下列步驟以安裝和封裝圖層內容。

安裝和封裝圖層內容的步驟

1. 克隆存aws-lambda-developer-guide GitHub 儲庫，其中包含您在sample-apps/layer-python目錄中需要的示例代碼。

   ```
   git clone https://github.com/awsdocs/aws-lambda-developer-guide.git
   ```

2. 導覽至範layer-python例應用程式的layer目錄。此目錄包含您用來建立和正確封裝圖層的程序檔。

   ```
   cd aws-lambda-developer-guide/sample-apps/layer-python/layer
   ```

3. 檢查requirements.txt檔案。此檔案定義您要包含在圖層中的相依性，也就是資requests源庫。您可以更新此檔案，以包括要包含在自己圖層中的任何相依性。

   Example requirements.txt

   ```
   requests==2.31.0
   ```

4. 請確定您擁有執行這兩個指令碼的權限。

```
chmod 744 1-install.sh && chmod 744 2-package.sh
```

5. 使用下列命令執行指令1-install.sh碼：

```
./1-install.sh
```

這個腳本用venv來創建一個名為的 Python 虛擬環境create_layer。然後，它會在create_layer/lib/python3.11/site-packages目錄中安裝所有必要的依賴關係。

Example 1-install.sh

```
python3.11 -m venv create_layer
source create_layer/bin/activate
pip install -r requirements.txt
```

6. 使用下列命令執行指令2-package.sh碼：

```
./2-package.sh
```

此指令碼會將create_layer/lib目錄中的內容複製到名為的新目錄中python。然後，它將python目錄的內容壓縮到名為layer_content.zip的文件中。這是圖層的 .zip 檔案。您可以解壓縮檔案，並確認檔案包含正確的檔案結構，如the section called “Python 執行階段的圖層路徑”本節所示。

Example 2-package.sh

```
mkdir python
cp -r create_layer/lib python/
zip -r layer_content.zip python
```

建立圖層

在本節中，您將取得上一節中產生的layer_content.zip檔案，並將其上傳為 Lambda 層。您可以透過 AWS Command Line Interface (AWS CLI) 使用 AWS Management Console 或 Lambda API 上傳圖層。上傳圖層 .zip 檔案時，請在下列指PublishLayerVersion AWS CLI 令中指定python3.11為相容的執行階段，並指定arm64為相容的架構。

```
aws lambda publish-layer-version --layer-name python-requests-layer \
```

```
    --zip-file fileb://layer_content.zip \
    --compatible-runtimes python3.11 \
    --compatible-architectures "arm64"
```

從響應中，請注意LayerVersionArn，看起來像arn:aws:lambda:us-east-1:*123456789012*:layer:python-requests-layer:1。當您將圖層添加到函數時，在本教程的下一步中，您將需要此 Amazon 資源名稱（ARN）。

將圖層添加到功能中

在本節中，您將部署在函數程式碼中使用程式requests庫的 Lambda 函數範例，然後附加該層。要部署該功能，您需要一個the section called “執行角色 (函數存取其他資源的權限)”. 如果您沒有現有的執行角色，請依照可摺疊區段中的步驟執行。否則，請跳至下一節以部署該功能。

(選擇性) 建立執行角色

若要建立執行角色

1. 在 IAM 主控台中開啟 角色頁面。
2. 選擇 建立角色。
3. 建立具備下列屬性的角色。
   - 信任實體 - Lambda。
   - 權限 — AWSLambdaBasicExecutionRole.
   - 角色名稱 - **lambda-role**。

   該AWSLambdaBasicExecutionRole策略具有函數將日誌寫入日誌所需的 CloudWatch 權限。

若要部署 Lambda 函數

1. 導覽至 function/ 目錄。如果您目前位於目layer/錄中，請執行下列命令：

   ```
   cd ../function
   ```

2. 檢閱函數程式碼。該函數導入requests庫，提出一個簡單的 HTTP GET 請求，然後返回狀態碼和正文。

   ```
   import requests
   ```

```
def lambda_handler(event, context):
    print(f"Version of requests library: {requests.__version__}")
    request = requests.get('https://api.github.com/')
    return {
        'statusCode': request.status_code,
        'body': request.text
    }
```

3. 使用下列命令建立 .zip 檔案部署套件：

```
zip my_deployment_package.zip lambda_function.py
```

4. 部署功能。在下列 AWS CLI 命令中，以您的執行角色 ARN 取代`--role`參數：

```
aws lambda create-function --function-name python_function_with_layer \
    --runtime python3.11 \
    --architectures "arm64" \
    --handler lambda_function.lambda_handler \
    --role arn:aws:iam::123456789012:role/lambda-role \
    --zip-file fileb://my_deployment_package.zip
```

（可選）調用您的函數而不附加圖層

此時，您可以選擇性地嘗試在附加圖層之前調用函數。如果你嘗試這個，那麼你應該得到一個導入錯誤，因為你的函數無法引用該`requests`包。要調用您的函數，請使用以下 AWS CLI 命令：

```
aws lambda invoke --function-name python_function_with_layer \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "key": "value" }' response.json
```

您應該會看到輸出，如下所示：

```
{
    "StatusCode": 200,
    "FunctionError": "Unhandled",
    "ExecutedVersion": "$LATEST"
}
```

若要檢視特定錯誤，請開啟輸出`response.json`檔案。您應該會看到`ImportModuleError`包含以下錯誤訊息：

```
"errorMessage": "Unable to import module 'lambda_function': No module named 'requests'"
```

接下來，將圖層附加到功能上。在下列 AWS CLI 指令中，將`--layers`參數取代為您先前提到的圖層版本 ARN：

```
aws lambda update-function-configuration --function-name python_function_with_layer \
    --cli-binary-format raw-in-base64-out \
    --layers "arn:aws:lambda:us-east-1:123456789012:layer:python-requests-layer:1"
```

最後，嘗試使用以下 AWS CLI 命令調用您的函數：

```
aws lambda invoke --function-name python_function_with_layer \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "key": "value" }' response.json
```

您應該會看到輸出，如下所示：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
```

輸出`response.json`文件包含有關響應的詳細信息。

(選擇性) 清理您的資源

除非您想要保留為此教學課程建立的資源，否則您現在便可刪除。刪除您不再使用的 AWS 資源，您可以避免不必要的費用 AWS 帳戶。

若要刪除 Lambda 圖層

1. 開啟 Lambda 主控台中的 層頁面。
2. 選取您建立的圖層。
3. 選擇刪除，然後再次選擇刪除。

若要刪除 Lambda 函數

1. 開啟 Lambda 主控台中的 函數頁面。

2. 選擇您建立的函數。
3. 選擇 Actions (動作)、Delete (刪除)。
4. 在文字輸入欄位中輸入 **delete**，然後選擇 刪除 。

使用manylinux車輪分佈

有時候，您想要作為依賴包含的軟件包不會有通用輪子（具體來說，它沒有any作為平台標籤）。在此情況下，請manylinux改為下載支援的滾輪。這樣可以確保您的圖層庫與 Amazon Linux 兼容。

numpy是一個沒有通用輪的軟件包。如果您想要在圖層中numpy包含套件，則可以完成下列範例步驟，以正確安裝和封裝圖層。

安裝和封裝圖層內容的步驟

1. 克隆存aws-lambda-developer-guide GitHub 儲庫，其中包含您在sample-apps/layer-python目錄中需要的示例代碼。

   ```
   git clone https://github.com/awsdocs/aws-lambda-developer-guide.git
   ```

2. 導覽至範layer-python例應用程式的layer-numpy目錄。此目錄包含您用來建立和正確封裝圖層的程序檔。

   ```
   cd aws-lambda-developer-guide/sample-apps/layer-python/layer-numpy
   ```

3. 檢查requirements.txt檔案。此檔案定義您要包含在圖層中的相依性，也就是資numpy源庫。在這裡，您可以指定與 Python 3.11，Amazon Linux 和指x86_64令集兼容的manylinux車輪分佈的 URL：

   Example requirements.txt

   ```
   https://files.pythonhosted.org/packages/3a/d0/
   edc009c27b406c4f9cbc79274d6e46d634d139075492ad055e3d68445925/numpy-1.26.4-cp311-
   cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl
   ```

4. 請確定您擁有執行這兩個指令碼的權限。

   ```
   chmod 744 1-install.sh && chmod 744 2-package.sh
   ```

5. 使用下列命令執行指令1-install.sh碼：

```
./1-install.sh
```

這個腳本用venv來創建一個名為的 Python 虛擬環境create_layer。然後，它會在create_layer/lib/python3.11/site-packages目錄中安裝所有必要的依賴關係。在此情況下，指pip令會有所不同，因為您必須將標--platform籤指定為manylinux2014_x86_64。即使您的本地計算機使用 macOS 或 Windows，這也會告訴pip您安裝正確的manylinux滾輪。

Example 1-install.sh

```
python3.11 -m venv create_layer
source create_layer/bin/activate
pip install -r requirements.txt --platform=manylinux2014_x86_64 --only-binary=:all:
 --target ./create_layer/lib/python3.11/site-packages
```

6. 使用下列命令執行指令2-package.sh碼：

```
./2-package.sh
```

此指令碼會將create_layer/lib目錄中的內容複製到名為的新目錄中python。然後，它將python目錄的內容壓縮到名為layer_content.zip的文件中。這是圖層的 .zip 檔案。您可以解壓縮檔案，並確認檔案包含正確的檔案結構，如the section called “Python 執行階段的圖層路徑”區段所示。

Example 2-package.sh

```
mkdir python
cp -r create_layer/lib python/
zip -r layer_content.zip python
```

若要將此層上傳至 Lambda，請使用下列PublishLayerVersion AWS CLI 指令：

```
aws lambda publish-layer-version --layer-name python-numpy-layer \
    --zip-file fileb://layer_content.zip \
    --compatible-runtimes python3.11 \
    --compatible-architectures "x86_64"
```

從響應中，請注意LayerVersionArn，看起來像arn:aws:lambda:us-east-1:*123456789012*:layer:python-numpy-layer:1。若要確認您的層是否如預期般運作，請在function-numpy目錄中部署 Lambda 函數。

若要部署 Lambda 函數

1. 導覽至 function-numpy/ 目錄。如果您目前位於目layer-numpy/錄中，請執行下列命令：

```
cd ../function-numpy
```

2. 檢閱函數程式碼。該函數導入numpy庫，創建一個簡單的numpy數組，然後返回一個虛擬的狀態碼和 body。

```
import json
import numpy as np

def lambda_handler(event, context):

    x = np.arange(15, dtype=np.int64).reshape(3, 5)
    print(x)

    return {
        'statusCode': 200,
        'body': json.dumps('Hello from Lambda!')
    }
```

3. 使用下列命令建立 .zip 檔案部署套件：

```
zip my_deployment_package.zip lambda_function.py
```

4. 部署功能。在下列 AWS CLI 命令中，以您的執行角色 ARN 取代--role參數：

```
aws lambda create-function --function-name python_function_with_numpy \
    --runtime python3.11 \
    --handler lambda_function.lambda_handler \
    --role arn:aws:iam::123456789012:role/lambda-role \
    --zip-file fileb://my_deployment_package.zip
```

（可選）調用您的函數而不附加圖層

或者，您可以嘗試在附加圖層之前調用函數。如果你嘗試這個，那麼你應該得到一個導入錯誤，因為你的函數無法引用該numpy包。要調用您的函數，請使用以下 AWS CLI 命令：

```
aws lambda invoke --function-name python_function_with_numpy \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "key": "value" }' response.json
```

您應該會看到輸出，如下所示：

```
{
    "StatusCode": 200,
    "FunctionError": "Unhandled",
    "ExecutedVersion": "$LATEST"
}
```

若要檢視特定錯誤，請開啟輸出response.json檔案。您應該會看到ImportModuleError包含以下錯誤訊息：

```
"errorMessage": "Unable to import module 'lambda_function': No module named 'numpy'"
```

接下來，將圖層附加到功能上。在下列 AWS CLI 指令中，將--layers參數取代為圖層版本 ARN：

```
aws lambda update-function-configuration --function-name python_function_with_numpy \
    --cli-binary-format raw-in-base64-out \
    --layers "arn:aws:lambda:us-east-1:123456789012:layer:python-requests-layer:1"
```

最後，嘗試使用以下 AWS CLI 命令調用您的函數：

```
aws lambda invoke --function-name python_function_with_numpy \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "key": "value" }' response.json
```

您應該會看到輸出，如下所示：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
```

```
}
```

您可以檢查函數日誌以驗證代碼是否將numpy數組打印為標準輸出。

Python 中的 AWS Lambda內容物件

當 Lambda 執行您的函數時，它會將內容物件傳遞至處理常式。此物件提供的方法和各項屬性提供了有關調用、函式以及執行環境的資訊。如需如何將內容物件傳遞至函數處理常式的詳細資訊，請參閱在 Python 中 Lambda 義函數處理程序。

內容方法

- `get_remaining_time_in_millis` - 傳回執行逾時前剩餘的毫秒數。

內容屬性

- `function_name` – Lambda 函數的名稱。
- `function_version` – 函數的版本。
- `invoked_function_arn` - 用於調用此函數的 Amazon Resource Name (ARN)。指出調用者是否指定版本號或別名。
- `memory_limit_in_mb` - 分配給函數的記憶體數量。
- `aws_request_id` - 調用請求的識別符。
- `log_group_name` - 函數的日誌群組。
- `log_stream_name` - 函數執行個體的記錄串流。
- `identity` - (行動應用程式) 已授權請求的 Amazon Cognito 身分的相關資訊。
  - `cognito_identity_id` - 已驗證的 Amazon Cognito 身分。
  - `cognito_identity_pool_id` - 授權調用的 Amazon Cognito 身分集區。
- `client_context` - (行動應用程式) 用戶端應用程式提供給 Lambda 的用戶端內容。
  - `client.installation_id`
  - `client.app_title`
  - `client.app_version_name`
  - `client.app_version_code`
  - `client.app_package_name`
  - `custom` - 行動用戶端應用程式所設定的 `dict` 自訂值。
  - `env` - AWS 開發套件所提供的 `dict` 環境資訊。

以下範例顯示記錄著內容資訊的處理常式函式。

Example handler.py

```
import time

def lambda_handler(event, context):
    print("Lambda function ARN:", context.invoked_function_arn)
    print("CloudWatch log stream name:", context.log_stream_name)
    print("CloudWatch log group name:",  context.log_group_name)
    print("Lambda Request ID:", context.aws_request_id)
    print("Lambda function memory limits in MB:", context.memory_limit_in_mb)
    # We have added a 1 second delay so you can see the time remaining in
 get_remaining_time_in_millis.
    time.sleep(1)
    print("Lambda time remaining in MS:", context.get_remaining_time_in_millis())
```

除了上面所列的選項，您也可以將 AWS X-Ray 開發套件用於 檢測 Python 代碼 AWS Lambda，識別重要的程式碼路徑，追蹤它們的效能，並且擷取要進行分析的資料。

AWS Lambda 函數日誌記 Python

AWS Lambda 自動監控 Lambda 函數，並將日誌項目傳送到 Amazon CloudWatch。您的 Lambda 函數隨附一個 CloudWatch 日誌記錄群組和函數每個執行個體的日誌串流。Lambda 執行期環境會將每次調用的詳細資訊和函數程式碼的其他輸出，傳送至日誌串流。如需有關 CloudWatch 記錄檔的詳細資訊，請參閱使用 Amazon CloudWatch 日誌 AWS Lambda。

若要從函數程式碼輸出日誌，可以使用內建 `logging` 模組。如需更詳細的項目，您可以使用任何寫入 `stdout` 或 `stderr` 的記錄程式庫。

列印至日誌

若要將基本輸出傳送至日誌，請使用您函數中的 `print` 方法。下列範例會記錄記 CloudWatch 錄檔群組和串流的值，以及事件物件。

請注意，如果您的函數使用 Python `print` 陳述式輸出記錄，Lambda 只能以純文字格式將 CloudWatch 記錄輸出傳送至記錄。若要擷取結構化 JSON 中的記錄，您必須使用支援的記錄程式庫。如需詳細資訊，請參閱the section called “搭配 Python 使用 Lambda 進階日誌控制項”。

Example lambda_function.py

```
import os
def lambda_handler(event, context):
    print('## ENVIRONMENT VARIABLES')
    print(os.environ['AWS_LAMBDA_LOG_GROUP_NAME'])
    print(os.environ['AWS_LAMBDA_LOG_STREAM_NAME'])
    print('## EVENT')
    print(event)
```

Example 記錄輸出

```
START RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95 Version: $LATEST
## ENVIRONMENT VARIABLES
/aws/lambda/my-function
2023/08/31/[$LATEST]3893xmpl7fac4485b47bb75b671a283c
## EVENT
{'key': 'value'}
END RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95
REPORT RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95  Duration: 15.74 ms  Billed
 Duration: 16 ms Memory Size: 128 MB Max Memory Used: 56 MB  Init Duration: 130.49 ms
```

```
XRAY TraceId: 1-5e34a614-10bdxmplf1fb44f07bc535a1   SegmentId: 07f5xmpl2d1f6f85
 Sampled: true
```

Python 執行時間會記錄每次調用的 START、END 和 REPORT 行。REPORT 行包含以下資料：

REPORT 行資料欄位

- RequestId— 呼叫的唯一要求 ID。
- 持續時間 - 函數的處理常式方法處理事件所花費的時間量。
- 計費持續時間 - 調用的計費時間量。
- 記憶體大小 - 分配給函數的記憶體數量。
- 使用的記憶體上限 - 函數所使用的記憶體數量。
- 初始化持續時間 - 對於第一個提供的請求，這是執行期載入函數並在處理常式方法之外執行程式碼所花費的時間量。
- XRAY TraceId — 針對追蹤的要求，則為AWS X-Ray 追蹤識別碼。
- SegmentId— 針對追蹤的請求，X-Ray 區段 ID。
- 已取樣 - 對於追蹤的請求，這是取樣結果。

使用記錄程式庫

如需更詳細的日誌，請使用標準程式庫中的 logging 模組，或任何寫入 `stdout` 或 `stderr` 的第三方記錄程式庫。

對於支援的 Python 執行期，您可以選擇是以純文字還是 JSON 擷取使用標準 `logging` 模組建立的日誌。如需進一步了解，請參閱the section called “搭配 Python 使用 Lambda 進階日誌控制項”。

目前，所有 Python 執行期的預設日誌格式都是純文字。下列範例顯示如何在 Logs 中以純文字擷取使用標準`logging`模組建立的 CloudWatch 記錄輸出。

```
import os
import logging
logger = logging.getLogger()
logger.setLevel("INFO")

def lambda_handler(event, context):
    logger.info('## ENVIRONMENT VARIABLES')
```

```
    logger.info(os.environ['AWS_LAMBDA_LOG_GROUP_NAME'])
    logger.info(os.environ['AWS_LAMBDA_LOG_STREAM_NAME'])
    logger.info('## EVENT')
    logger.info(event)
```

來自 `logger` 的輸出包含記錄等級、時間戳記和請求 ID。

```
START RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125 Version: $LATEST
[INFO]  2023-08-31T22:12:58.534Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    ##
 ENVIRONMENT VARIABLES
[INFO]  2023-08-31T22:12:58.534Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    /aws/
lambda/my-function
[INFO]  2023-08-31T22:12:58.534Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    2023/01/31/
[$LATEST]1bbe51xmplb34a2788dbaa7433b0aa4d
[INFO]  2023-08-31T22:12:58.535Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    ## EVENT
[INFO]  2023-08-31T22:12:58.535Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    {'key':
 'value'}
END RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125
REPORT RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125  Duration: 2.75 ms   Billed
 Duration: 3 ms Memory Size: 128 MB Max Memory Used: 56 MB  Init Duration: 113.51 ms
XRAY TraceId: 1-5e34a66a-474xmpl7c2534a87870b4370   SegmentId: 073cxmpl3e442861
 Sampled: true
```

> **Note**
>
> 當您的函數的日誌格式設定為純文字時，Python 執行期的預設日誌層級設定為 WARN。這表示 Lambda 只會將 WARN 層級和更低層級的記錄輸出傳送至 CloudWatch 記錄檔。若要變更預設日誌層級，請使用 Python `logging setLevel()` 方法，如此範例程式碼所示。如果將函數的日誌格式設定為 JSON，我們建議您使用 Lambda 進階記錄控制項來設定函數的日誌層級，而不是在程式碼中設定日誌層級。如需進一步了解，請參閱 the section called “搭配 Python 使用日誌層級篩選”

搭配 Python 使用 Lambda 進階日誌控制項

為了讓您更妥善地控制擷取、處理和使用函數日誌的方式，您可以針對支援的 Lambda Python 執行期設定下列記錄選項：

- 日誌格式 - 在純文字和結構化 JSON 格式之間為您的日誌進行選擇

- 日誌級別-對於 JSON 格式的日誌，選擇 Lambda 發送到 Amazon 的日誌詳細級別 CloudWatch，例如錯誤，調試或信息
- 日誌組-選擇您的功能發送日誌的日誌組 CloudWatch

如需這些日誌選項的詳細資訊，以及如何設定函數以使用這些選項的說明，請參閱 the section called "設定 Lambda 函數的進階日誌控制項"。

若要進一步瞭解如何將日誌格式和日誌層級選項與 Python Lambda 函數搭配使用，請參閱以下各節中的指引。

搭配 Python 使用結構化的 JSON 日誌

如果您為函數的記錄格式選取 JSON，Lambda 會將 Python 標準記錄程式庫的記錄輸出傳送 CloudWatch 至結構化 JSON。每個 JSON 日誌物件都包含至少四個鍵值對，其中包含下列索引鍵：

- `"timestamp"` - 產生日誌訊息的時間
- `"level"` - 指派給訊息的日誌層級
- `"message"` - 日誌訊息的內容
- `"requestId"` - 進行調用的唯一請求 ID。

Python `logging` 程式庫還可以新增額外的鍵值對 (如 `"logger"`) 到這個 JSON 物件。

以下各節中的範例說明當您將函數的記錄檔格式設定為 JSON 時，如何在 CloudWatch 記錄檔中擷取使用 Python 程式`logging`庫產生的記錄輸出。

請注意，如果您使用 `print` 方法產生基本的日誌輸出，如 the section called "列印至日誌" 中所描述，即使您將函數的日誌格式設定為 JSON，Lambda 仍會以純文字擷取這些輸出。

使用 Python 記錄程式庫進行標準 JSON 日誌輸出

下面的示例代碼片段和日誌輸出顯示了當函數的日誌格式設置為 JSON 時，如何在 CloudWatch 日誌中捕獲使用 Python `logging` 庫生成的標準日誌輸出。

Example Python 日誌記錄程式碼

```
import logging
logger = logging.getLogger()
```

```
def lambda_handler(event, context):
    logger.info("Inside the handler function")
```

Example JSON 日誌記錄

```
{
    "timestamp":"2023-10-27T19:17:45.586Z",
    "level":"INFO",
    "message":"Inside the handler function",
    "logger": "root",
    "requestId":"79b4f56e-95b1-4643-9700-2807f4e68189"
}
```

在 JSON 中記錄額外的參數

當函數的日誌格式設置為 JSON 時，您也可以使用`extra`關鍵字將 Python 字典傳遞給日誌輸出，以使用標準 Python `logging` 庫記錄其他參數。

Example Python 日誌記錄程式碼

```
import logging

def lambda_handler(event, context):
    logging.info(
        "extra parameters example",
        extra={"a":"b", "b": [3]},
    )
```

Example JSON 日誌記錄

```
{
  "timestamp": "2023-11-02T15:26:28Z",
  "level": "INFO",
  "message": "extra parameters example",
  "logger": "root",
  "requestId": "3dbd5759-65f6-45f8-8d7d-5bdc79a3bd01",
  "a": "b",
  "b": [
    3
  ]
```

```
}
```

在 JSON 中記錄例外狀況

下列程式碼片段顯示當您將日誌格式設定為 JSON 時，如何在函數的日誌輸出中擷取 Python 的例外狀況。請注意，使用 `logging.exception` 產生的日誌檔輸出會指派到日誌層級 ERROR。

Example Python 日誌記錄程式碼

```
import logging

def lambda_handler(event, context):
    try:
        raise Exception("exception")
    except:
        logging.exception("msg")
```

Example JSON 日誌記錄

```
{
  "timestamp": "2023-11-02T16:18:57Z",
  "level": "ERROR",
  "message": "msg",
  "logger": "root",
  "stackTrace": [
    "  File \"/var/task/lambda_function.py\", line 15, in lambda_handler\n    raise
 Exception(\"exception\")\n"
  ],
  "errorType": "Exception",
  "errorMessage": "exception",
  "requestId": "3f9d155c-0f09-46b7-bdf1-e91dab220855",
  "location": "/var/task/lambda_function.py:lambda_handler:17"
}
```

JSON 結構化日誌與其他記錄工具

如果您的代碼已經使用另一個日誌庫（例如 Powertools for AWS Lambda）來生成 JSON 結構化日誌，則不需要進行任何更改。 AWS Lambda 不會對任何已經進行 JSON 編碼的記錄進行雙重編碼。即使您將函數設定為使用 JSON 記錄格式，記錄輸出也會顯示 CloudWatch 在您定義的 JSON 結構中。

下面的示例演示了如何使用 Powertools 的 AWS Lambda 包生成的日誌輸出在 CloudWatch 日誌中捕獲。無論函數的日誌組態設定為 JSON 還是 TEXT，此日誌輸出的格式都相同。如需使用 Powertools

的詳細資訊 AWS Lambda，請參閱the section called “使用動力工具 AWS Lambda （Python）和結構化日 AWS SAM 誌記錄”和 the section called “使用動力工具 AWS Lambda （Python）和結構化日 AWS CDK 誌記錄”

Example Python 日誌代碼片段（使用動力工具） AWS Lambda

```
from aws_lambda_powertools import Logger

logger = Logger()

def lambda_handler(event, context):
    logger.info("Inside the handler function")
```

Example JSON 日誌記錄（使用電動工具） AWS Lambda

```
{
    "level": "INFO",
    "location": "lambda_handler:7",
    "message": "Inside the handler function",
    "timestamp": "2023-10-31 22:38:21,010+0000",
    "service": "service_undefined",
    "xray_trace_id": "1-654181dc-65c15d6b0fecbdd1531ecb30"
}
```

搭配 Python 使用日誌層級篩選

透過設定記錄層級篩選，您可以選擇只傳送特定記錄層級或更低等級的記錄 CloudWatch 檔至記錄檔。若要瞭解如何設定函數的日誌層級篩選，請參閱 the section called “日誌層級篩選”。

AWS Lambda 為了根據日誌級別過濾應用程序日誌，您的函數必須使用 JSON 格式的日誌。您可以透過兩種方式達成此操作：

- 使用標準 Python `logging` 程式庫建立日誌輸出，並配置您的函數以使用 JSON 日誌格式。然後 AWS Lambda 使用 the section called “搭配 Python 使用結構化的 JSON 日誌” 中描述的 JSON 物件中的「層級」索引鍵值組篩選日誌輸出。若要瞭解如何設定函數的日誌格式，請參閱 the section called “設定 Lambda 函數的進階日誌控制項”。
- 使用其他日誌程式庫或方法，在您的程式碼中建立 JSON 結構化日誌，其中包含定義日誌輸出層級的「層級」索引鍵值組。例如，您可以使用 Powertools 從您的 AWS Lambda 代碼生成 JSON 結構化日誌輸出。

您也可以使用列印陳述式輸出包含日誌層級識別碼的 JSON 物件。下列列印陳述式會產生 JSON 格式的輸出，其中記錄層級設定為 INFO。 AWS Lambda 如果您的函數的 CloudWatch 日誌記錄級別設置為「信息」，「調試」或「跟踪」，則將 JSON 對象發送到日誌。

```
print('{"msg":"My log message", "level":"info"}')
```

若要讓 Lambda 篩選函數的日誌，您還必須在 JSON 日誌輸出中包含 `"timestamp"` 索引鍵值組。必須以有效的 RFC 3339 時間戳記格式指定時間。如果您沒有提供有效的時間戳記，Lambda 會為日誌指派層級 INFO，並為您新增時間戳記。

在 Lambda 主控台檢視日誌

您可以在調用 Lambda 函數之後，使用 Lambda 主控台來檢視日誌輸出。

如果可以從內嵌程式碼編輯器測試您的程式碼，您會在執行結果中找到日誌。使用主控台測試功能以調用函數時，您會在詳細資訊區段找到日誌輸出。

在 CloudWatch 主控台中檢視記錄

您可以使用 Amazon 主 CloudWatch 控台來檢視所有 Lambda 函數叫用的日誌。

在 CloudWatch 主控台上檢視記錄檔

1. 在主控台上開啟 [記錄群組] 頁 CloudWatch 面。
2. 選擇您的函數的日誌群組 (/aws/lambda/***your-function-name***)。
3. 選擇日誌串流

每個日誌串流都會對應至函式的執行個體。當您更新 Lambda 函數，以及建立額外執行個體以處理多個並行調用時，便會出現日誌串流。要查找特定調用的日誌，我們建議使用檢測您的函數。 AWS X-Ray X-Ray 會在追蹤內記錄有關請求和日誌串流的詳細資訊。

檢視記錄 AWS CLI

這 AWS CLI 是一種開放原始碼工具，可讓您使用命令列殼層中的命令與 AWS 服務互動。若要完成本節中的步驟，您必須執行下列各項：

- AWS Command Line Interface (AWS CLI) 第二版

- AWS CLI -快速配置 aws configure

您可以透過 AWS CLI，使用 --log-type 命令選項來擷取要調用的日誌。其回應將包含 LogResult 欄位，內含該次調用的 base64 編碼日誌 (最大達 4 KB)。

Example 擷取日誌 ID

下列範例顯示如何從名稱為 my-function 的函數的 LogResult 欄位來擷取日誌 ID。

```
aws lambda invoke --function-name my-function out --log-type Tail
```

您應該會看到下列輸出：

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example 解碼日誌

在相同的命令提示中，使用 base64 公用程式來解碼日誌。下列範例顯示如何擷取 my-function 的 base64 編碼日誌。

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

如果您使用的是 AWS CLI 版本 2，則需要此cli-binary-format選項。若要讓此成為預設的設定，請執行 aws configure set cli-binary-format raw-in-base64-out。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

您應該會看到下列輸出：

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

該 base64 公用程式可在 Linux、macOS 和 Ubuntu on Windows 上使用。macOS 使用者可能需要使用 base64 -D。

Example get-logs.sh 指令碼

在相同的命令提示中，使用下列指令碼下載最後五個日誌事件。該指令碼使用 sed 以從輸出檔案移除引述，並休眠 15 秒以使日誌可供使用。輸出包括來自 Lambda 的回應以及來自 get-log-events 命令的輸出。

複製下列程式碼範例的內容，並將您的 Lambda 專案目錄儲存為 get-logs.sh。

如果您使用的是 AWS CLI 版本 2，則需要此cli-binary-format選項。若要讓此成為預設的設定，請執行 aws configure set cli-binary-format raw-in-base64-out。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS 和 Linux (僅限)

在相同的命令提示中，macOS 和 Linux 使用者可能需要執行下列命令，以確保指令碼可執行。

```
chmod -R 755 get-logs.sh
```

Example 擷取最後五個記錄事件

在相同的命令提示中，執行下列指令碼以取得最後五個日誌事件。

```
./get-logs.sh
```

您應該會看到下列輸出：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
```

```
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```

刪除日誌

當您刪除函數時，不會自動刪除日誌群組。若要避免無限期地儲存日誌，請刪除日誌群組，或設定保留期間，系統會在該時間之後自動刪除日誌。

工具與程式庫

Powertools for AWS Lambda (Python) 是一個開發人員工具組，用於實作無伺服器最佳實務並提高開發人員速度。Logger 公用程式提供 Lambda 優化記錄器，其中包含有關所有函數之函數內容的其他資訊，輸出結構為 JSON。使用此公用程式執行下列操作：

- 從 Lambda 內容、冷啟動和 JSON 形式的結構記錄輸出中擷取關鍵欄位
- 在收到指示時記錄 Lambda 調用事件 (預設為停用)
- 透過日誌採樣僅列印調用百分比的所有日誌 (預設為停用)
- 在任何時間點將其他金鑰附加至結構化日誌
- 使用自訂日誌格式化程式 (自帶格式化程式)，以與組織的日誌記錄 RFC 相容的結構輸出日誌。

使用動力工具 AWS Lambda （Python）和結構化日 AWS SAM 誌記錄

請依照以下步驟操作，使用 AWS SAM透過整合式 Powertools for Python 模組，下載、建置和部署範例 Hello World Python 應用程式。此應用程式實作了基本 API 後端，並使用 Powertools 發送日誌、指標和追蹤。其包含 Amazon API Gateway 端點和 Lambda 函數。當您將 GET 請求傳送至 API Gateway 端點時，Lambda 函數會叫用、使用內嵌指標格式將記錄和指標傳送至 CloudWatch，並將追蹤傳送至 AWS X-Ray。該函數會傳回 `hello world` 訊息。

必要條件

若要完成本節中的步驟，您必須執行下列各項：

- Python 3.9
- AWS CLI 第二版
- AWS SAM CLI 版本 1.75 或更新版本。如果您使用較舊版本的 AWS SAM CLI，請參閱升級 AWS SAM CLI。

部署範例 AWS SAM 應用程式

1. 使用 Hello World Python 範本來初始化應用程式。

```
sam init --app-template hello-world-powertools-python --name sam-app --package-type
 Zip --runtime python3.9 --no-tracing
```

2. 建置應用程式。

```
cd sam-app && sam build
```

3. 部署應用程式。

```
sam deploy --guided
```

4. 依照螢幕上的提示操作。若要接受互動體驗中提供的預設選項，請按下 Enter。

> **Note**
>
> 因為HelloWorldFunction 可能沒有定義授權，這可以嗎？ ，請務必輸入y。

5. 取得已部署應用程式的 URL：

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. 調用 API 端點：

```
curl GET <URL_FROM_PREVIOUS_STEP>
```

成功的話，您將會看到以下回應：

```
{"message":"hello world"}
```

7. 若要獲取該函數的日誌，請執行 sam 日誌。如需詳細資訊，請參閱《AWS Serverless Application Model 開發人員指南》 中的 使用日誌。

```
sam logs --stack-name sam-app
```

日誌輸出如下：

```
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04
 2023-02-03T14:59:50.371000 INIT_START Runtime Version:
 python:3.9.v16    Runtime Version ARN: arn:aws:lambda:us-
east-1::runtime:07a48df201798d627f2b950f03bb227aab4a655a1d019c3296406f95937e2525
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.112000
 START RequestId: d455cfc4-7704-46df-901b-2a5cce9405be Version: $LATEST
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.114000 {
```

```
  "level": "INFO",
  "location": "hello:23",
  "message": "Hello world API - HTTP 200",
  "timestamp": "2023-02-03 14:59:51,113+0000",
  "service": "PowertoolsHelloWorld",
  "cold_start": true,
  "function_name": "sam-app-HelloWorldFunction-YBg8yfYtOc9j",
  "function_memory_size": "128",
  "function_arn": "arn:aws:lambda:us-east-1:111122223333:function:sam-app-
HelloWorldFunction-YBg8yfYtOc9j",
  "function_request_id": "d455cfc4-7704-46df-901b-2a5cce9405be",
  "correlation_id": "e73f8aef-5e07-436e-a30b-63e4b23f0047",
  "xray_trace_id": "1-63dd2166-434a12c22e1307ff2114f299"
}
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.126000 {
  "_aws": {
    "Timestamp": 1675436391126,
    "CloudWatchMetrics": [
      {
        "Namespace": "Powertools",
        "Dimensions": [
          [
            "function_name",
            "service"
          ]
        ],
        "Metrics": [
          {
            "Name": "ColdStart",
            "Unit": "Count"
          }
        ]
      }
    ]
  },
  "function_name": "sam-app-HelloWorldFunction-YBg8yfYtOc9j",
  "service": "PowertoolsHelloWorld",
  "ColdStart": [
    1.0
  ]
}
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.126000 {
  "_aws": {
    "Timestamp": 1675436391126,
```

```
    "CloudWatchMetrics": [
      {
        "Namespace": "Powertools",
        "Dimensions": [
          [
            "service"
          ]
        ],
        "Metrics": [
          {
            "Name": "HelloWorldInvocations",
            "Unit": "Count"
          }
        ]
      }
    ]
  },
  "service": "PowertoolsHelloWorld",
  "HelloWorldInvocations": [
    1.0
  ]
}
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.128000 END
 RequestId: d455cfc4-7704-46df-901b-2a5cce9405be
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.128000
 REPORT RequestId: d455cfc4-7704-46df-901b-2a5cce9405be    Duration: 16.33 ms
 Billed Duration: 17 ms    Memory Size: 128 MB    Max Memory Used: 64 MB    Init
 Duration: 739.46 ms
XRAY TraceId: 1-63dd2166-434a12c22e1307ff2114f299    SegmentId: 3c5d18d735a1ced0
 Sampled: true
```

8. 這是可透過網際網路存取的公有 API 端點。建議您在測試後刪除端點。

```
sam delete
```

管理日誌保留

當您刪除函數時，不會自動刪除日誌群組。若要避免無限期儲存記錄檔，請刪除記錄群組，或設定保留期間，之後 CloudWatch 會自動刪除記錄檔。若要設定記錄保留，請將下列項目新增至 AWS SAM 範本：

```
Resources:
```

```
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      # Omitting other properties

  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

使用動力工具 AWS Lambda （Python）和結構化日 AWS CDK 誌記錄

請按照下面的步驟下載，構建和部署示例你好世界 Python 應用程序集成動力工具 AWS Lambda （Python）模塊使用 AWS CDK. 此應用程式實作了基本 API 後端，並使用 Powertools 發送日誌、指標和追蹤。其包含 Amazon API Gateway 端點和 Lambda 函數。當您將 GET 請求傳送至 API Gateway 端點時，Lambda 函數會叫用、使用內嵌指標格式將記錄和指標傳送至 CloudWatch，並將追蹤傳送至 AWS X-Ray。函數會傳回 hello world 訊息。

必要條件

若要完成本節中的步驟，您必須執行下列各項：

- Python 3.9
- AWS CLI 第二版
- AWS CDK 第二版
- AWS SAM CLI 版本 1.75 或更新版本。如果您使用較舊版本的 AWS SAM CLI，請參閱升級 AWS SAM CLI。

部署範例 AWS CDK 應用程式

1. 為您的新應用程式建立專案目錄。

   ```
   mkdir hello-world
   cd hello-world
   ```

2. 初始化應用程式。

   ```
   cdk init app --language python
   ```

3. 安裝 Python 相依項。

```
pip install -r requirements.txt
```

4. 在根資料夾下建立 lambda_function 目錄。

```
mkdir lambda_function
cd lambda_function
```

5. 建立檔案 app.py，並將下列程式碼新增至檔案。這是 Lambda 函數的程式碼。

```
from aws_lambda_powertools.event_handler import APIGatewayRestResolver
from aws_lambda_powertools.utilities.typing import LambdaContext
from aws_lambda_powertools.logging import correlation_paths
from aws_lambda_powertools import Logger
from aws_lambda_powertools import Tracer
from aws_lambda_powertools import Metrics
from aws_lambda_powertools.metrics import MetricUnit

app = APIGatewayRestResolver()
tracer = Tracer()
logger = Logger()
metrics = Metrics(namespace="PowertoolsSample")

@app.get("/hello")
@tracer.capture_method
def hello():
    # adding custom metrics
    # See: https://docs.powertools.aws.dev/lambda-python/latest/core/metrics/
    metrics.add_metric(name="HelloWorldInvocations", unit=MetricUnit.Count,
 value=1)

    # structured log
    # See: https://docs.powertools.aws.dev/lambda-python/latest/core/logger/
    logger.info("Hello world API - HTTP 200")
    return {"message": "hello world"}

# Enrich logging with contextual information from Lambda
@logger.inject_lambda_context(correlation_id_path=correlation_paths.API_GATEWAY_REST)
# Adding tracer
# See: https://docs.powertools.aws.dev/lambda-python/latest/core/tracer/
@tracer.capture_lambda_handler
```

```
# ensures metrics are flushed upon request completion/failure and capturing
 ColdStart metric
@metrics.log_metrics(capture_cold_start_metric=True)
def lambda_handler(event: dict, context: LambdaContext) -> dict:
    return app.resolve(event, context)
```

6. 開啟 hello_world 目錄。您應看到名稱為 hello_world_stack.py 的檔案。

```
cd ..
cd hello_world
```

7. 開啟 hello_world_stack.py，並將下列程式碼新增至檔案。其中包含建立 Lambda 函數的 Lambda 建構函式、設定 Powertools 的環境變數，並將記錄保留設定為一週，以及建立 REST API 的 ApiGatewayv1 個建構函式。

```
from aws_cdk import (
    Stack,
    aws_apigateway as apigwv1,
    aws_lambda as lambda_,
    CfnOutput,
    Duration
)
from constructs import Construct

class HelloWorldStack(Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        # Powertools Lambda Layer
        powertools_layer = lambda_.LayerVersion.from_layer_version_arn(
            self,
            id="lambda-powertools",
            # At the moment we wrote this example, the aws_lambda_python_alpha CDK
 constructor is in Alpha, o we use layer to make the example simpler
            # See https://docs.aws.amazon.com/cdk/api/v2/python/
aws_cdk.aws_lambda_python_alpha/README.html
            # Check all Powertools layers versions here: https://
docs.powertools.aws.dev/lambda-python/latest/#lambda-layer
            layer_version_arn=f"arn:aws:lambda:
{self.region}:017000801446:layer:AWSLambdaPowertoolsPythonV2:21"
        )
```

```
        function = lambda_.Function(self,
            'sample-app-lambda',
            runtime=lambda_.Runtime.PYTHON_3_9,
            layers=[powertools_layer],
            code = lambda_.Code.from_asset("./lambda_function/"),
            handler="app.lambda_handler",
            memory_size=128,
            timeout=Duration.seconds(3),
            architecture=lambda_.Architecture.X86_64,
            environment={
                "POWERTOOLS_SERVICE_NAME": "PowertoolsHelloWorld",
                "POWERTOOLS_METRICS_NAMESPACE": "PowertoolsSample",
                "LOG_LEVEL": "INFO"
            }
        )

        apigw = apigwv1.RestApi(self, "PowertoolsAPI",
 deploy_options=apigwv1.StageOptions(stage_name="dev"))

        hello_api = apigw.root.add_resource("hello")
        hello_api.add_method("GET", apigwv1.LambdaIntegration(function,
 proxy=True))

        CfnOutput(self, "apiUrl", value=f"{apigw.url}hello")
```

8. 部署您的應用程式。

```
cd ..
cdk deploy
```

9. 取得已部署應用程式的 URL：

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?OutputKey==`apiUrl`].OutputValue' --output text
```

10. 調用 API 端點：

```
curl GET <URL_FROM_PREVIOUS_STEP>
```

成功的話，您將會看到以下回應：

```
{"message":"hello world"}
```

11. 若要獲取該函數的日誌，請執行 sam 日誌。如需詳細資訊，請參閱《AWS Serverless Application Model 開發人員指南》 中的 使用日誌。

```
sam logs --stack-name HelloWorldStack
```

日誌輸出如下：

```
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04
 2023-02-03T14:59:50.371000 INIT_START Runtime Version:
 python:3.9.v16    Runtime Version ARN: arn:aws:lambda:us-
east-1::runtime:07a48df201798d627f2b950f03bb227aab4a655a1d019c3296406f95937e2525
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.112000
 START RequestId: d455cfc4-7704-46df-901b-2a5cce9405be Version: $LATEST
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.114000 {
  "level": "INFO",
  "location": "hello:23",
    "message": "Hello world API - HTTP 200",
  "timestamp": "2023-02-03 14:59:51,113+0000",
  "service": "PowertoolsHelloWorld",
  "cold_start": true,
  "function_name": "sam-app-HelloWorldFunction-YBg8yfYtOc9j",
  "function_memory_size": "128",
  "function_arn": "arn:aws:lambda:us-east-1:111122223333:function:sam-app-
HelloWorldFunction-YBg8yfYtOc9j",
  "function_request_id": "d455cfc4-7704-46df-901b-2a5cce9405be",
  "correlation_id": "e73f8aef-5e07-436e-a30b-63e4b23f0047",
  "xray_trace_id": "1-63dd2166-434a12c22e1307ff2114f299"
}
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.126000 {
  "_aws": {
    "Timestamp": 1675436391126,
    "CloudWatchMetrics": [
      {
        "Namespace": "Powertools",
        "Dimensions": [
          [
            "function_name",
            "service"
          ]
        ],
        "Metrics": [
          {
```

```
            "Name": "ColdStart",
            "Unit": "Count"
          }
        ]
      }
    ]
  },
  "function_name": "sam-app-HelloWorldFunction-YBg8yfYtOc9j",
  "service": "PowertoolsHelloWorld",
  "ColdStart": [
    1.0
  ]
}
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.126000 {
  "_aws": {
    "Timestamp": 1675436391126,
    "CloudWatchMetrics": [
      {
        "Namespace": "Powertools",
        "Dimensions": [
          [
            "service"
          ]
        ],
        "Metrics": [
          {
            "Name": "HelloWorldInvocations",
            "Unit": "Count"
          }
        ]
      }
    ]
  },
  "service": "PowertoolsHelloWorld",
  "HelloWorldInvocations": [
    1.0
  ]
}
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.128000 END
 RequestId: d455cfc4-7704-46df-901b-2a5cce9405be
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.128000
 REPORT RequestId: d455cfc4-7704-46df-901b-2a5cce9405be    Duration: 16.33 ms
 Billed Duration: 17 ms    Memory Size: 128 MB    Max Memory Used: 64 MB    Init
 Duration: 739.46 ms
```

```
XRAY TraceId: 1-63dd2166-434a12c22e1307ff2114f299    SegmentId: 3c5d18d735a1ced0
 Sampled: true
```

12. 這是可透過網際網路存取的公有 API 端點。建議您在測試後刪除端點。

```
cdk destroy
```

以 Python 測試 AWS Lambda 函數

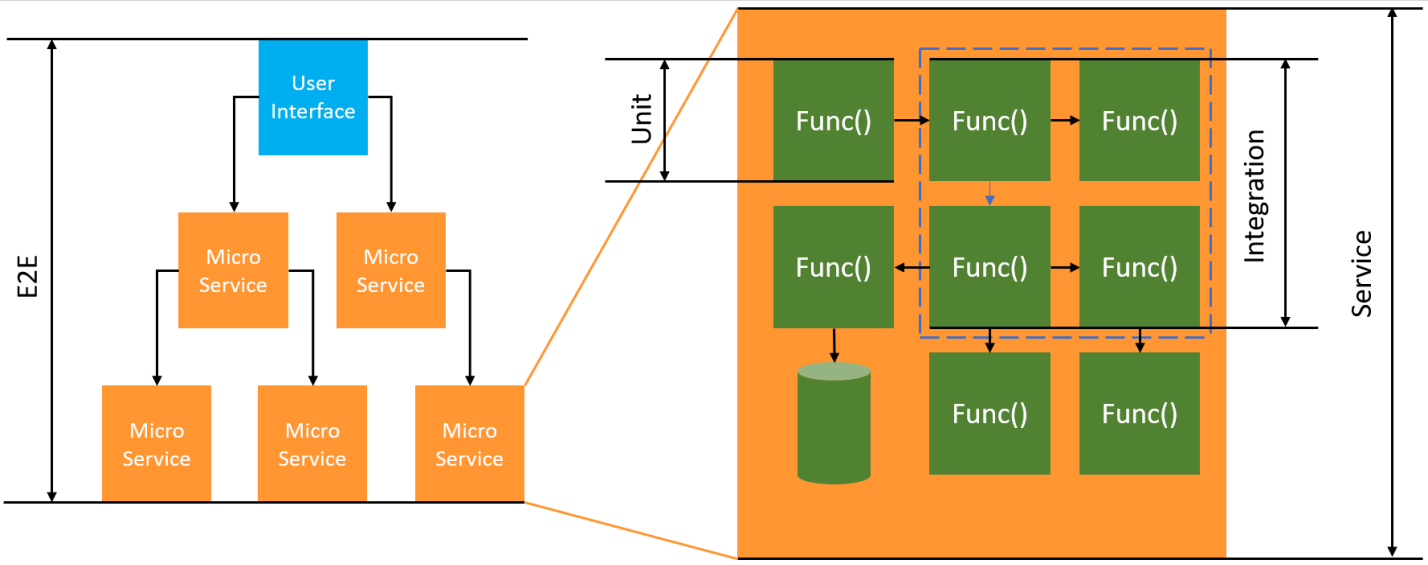
> **Note**
>
> 如需測試無伺服器解決方案之技術和最佳實務的完整介紹，請參閱測試函數章節。

測試無伺服器函數會使用傳統的測試類型和技術，但您也必須考慮測試整個無伺服器應用程式。以雲端為基礎的測試會為您的函數和無伺服器應用程式提供最準確的品質測量標準。

無伺服器應用程式架構包括透過 API 呼叫提供關鍵應用程式功能的受管服務。因此，您的開發週期應包括自動化測試，以便在函數和服務互動時驗證功能。

如果您未建立以雲端為基礎的測試，則可能會因本機環境與部署環境之間的差異而遇到問題。您的持續整合程序應先針對雲端佈建的一組資源進行測試，然後再將程式碼升級至下一個部署環境 (例如 QA、暫存或生產環境)。

繼續閱讀這份簡短指南，了解無伺服器應用程式的測試策略，或造訪無伺服器測試範例儲存庫，深入了解所選語言和執行期的特定實際範例。



對於無服務器測試，您仍然會編寫單元，集成和end-to-end測試。

- 單元測試：針對一組隔離的程式碼區塊進行的測試。例如，驗證商業邏輯以計算指定的特定項目與目的地的運費。
- 整合測試：涉及到兩個以上元件或服務進行互動的測試 (通常在雲端環境)。例如，驗證函數是否有處理佇列中的事件。

- E nd-to-end 測試-測試，驗證整個應用程序的行為。例如，確保基礎設施的設定正確無誤，以及事件如預期在服務之間流動，以記錄客戶的訂單。

測試無伺服器應用程式

通常會混合使用多種方法來測試無伺服器應用程式程式碼，包括在雲端進行測試、透過模擬物件進行測試，以及偶爾使用模擬器進行測試。

在雲端進行測試

雲端測試對於測試的所有階段都很有價值，包括單元測試、整合測試和 end-to-end 測試。您可以針對部署在雲端中的程式碼執行測試，並與雲端服務互動。這是最準確的程式碼品質測量方法。

您可以透過主控台使用測試事件，輕鬆在雲端對 Lambda 函數進行偵錯。一個測試事件是函數的 JSON 輸入。如果您的函數不需要輸入，該事件可以是空白的 JSON 文件 (`{}`)。主控台提供各種服務整合的範例事件。在主控台中建立事件後，您可以與團隊分享事件，讓測試變得更容易，結果更一致。

> **Note**
>
> 在控制台中測試函數是簡便快速的入門方式，而將測試週期自動化可確保應用程式的品質和開發速度。

測試工具

您可以透過一些工具和技術加快回饋迴圈的開發速度。例如，AWS SAM Accelerate 和 AWS CDK 監看模式都可以縮短更新雲端環境所需的時間。

Moto 是用於模擬 AWS 服務和資源的 Python 程式庫，您可以使用裝飾項目攔截和模擬回應，幾乎不用修改即可測試函數。

Powertools for AWS Lambda (Python) 的驗證功能提供裝飾項目，可用來驗證 Python 函數的輸入事件和輸出回應。

如需詳細資訊，請閱讀部落格文章：使用 Python 和 Mock AWS Services 對 Lambda 進行單元測試。

若要降低與雲端部署反覆運算相關的延遲，請參閱 AWS Serverless Application Model (AWS SAM) Accelerate、AWS Cloud Development Kit (AWS CDK) 監看模式。這些工具會監控您的基礎架構和程式碼是否變更。它們會自動建立增量更新並將其部署到您的雲端環境中，藉此回應這些變更。

如需這些工具的使用範例，請前往 Python 測試範例程式碼儲存庫。

檢測 Python 代碼 AWS Lambda

Lambda 與 AWS X-Ray 整合，可協助您追蹤、偵錯和最佳化 Lambda 應用程式。您可以使用 X-Ray 來追蹤請求，因為它會周遊您應用程式中的資源，其中可能包含 Lambda 函數和其他 AWS 服務。

若要將追蹤資料傳送至 X-Ray，您可以使用以下三個 SDK 庫之一：

- AWS 適用於 OpenTelemetry (ADOT) 的發行版 — 安全、可生產就緒且 AWS支援的 (OTel) SDK 發行版本 OpenTelemetry 。
- 適用於 Python 的 AWS X-Ray SDK – 用於生成追蹤資料並將其傳送至 X-Ray 的 SDK。
- 適用於 AWS Lambda (Python) 的 Powertools — 可實作無伺服器最佳實務並提高開發人員速度的開發人員工具組。

每個 SDK 均提供將遙測資料傳送至 X-Ray 服務的方法。然後，您可以使用 X-Ray 來檢視、篩選應用程式的效能指標並獲得洞察，從而識別問題和進行最佳化的機會。

> **Important**
>
> 用於 AWS Lambda SDK 的 X-Ray 和 Powertools 是由提供的緊密集成的儀表解決方案的一部分。 AWS ADOT Lambda Layers 是用於追蹤檢測之業界通用標準的一部分，這類檢測一般會收集更多資料，但可能不適用於所有使用案例。您可以使用任一解決方案在 X-Ray 中實作 end-to-end 追蹤。若要深入了解如何在兩者之間做選擇，請參閱在 AWS Distro for OpenTelemetry 和 X-Ray SDK 之間進行選擇。

章節

- 使用動力工具進行 AWS Lambda （ Python ）和跟 AWS SAM 踪
- 使用動力工具 AWS Lambda （ Python ）和跟 AWS CDK 踪
- 使用 ADOT 來檢測您的 Python 函數
- 使用 X-Ray SDK 來檢測 Python 功能
- 透過 Lambda 主控台來啟用追蹤
- 透過 Lambda API 啟用追蹤
- 使用啟動追蹤 AWS CloudFormation
- 解讀 X-Ray 追蹤
- 將執行時間相依項存放存在層中 (X-Ray SDK)

使用動力工具進行 AWS Lambda （Python）和跟 AWS SAM 踪

請按照下面的步驟下載，構建和部署示例你好世界 Python 應用程序集成動力工具 AWS Lambda （Python）模塊使用 AWS SAM. 此應用程式實作了基本 API 後端，並使用 Powertools 發送日誌、指標和追蹤。其包含 Amazon API Gateway 端點和 Lambda 函數。當您將 GET 請求傳送至 API Gateway 端點時，Lambda 函數會叫用、使用內嵌指標格式將記錄和指標傳送至 CloudWatch，並將追蹤傳送至 AWS X-Ray。函數會傳回 hello world 訊息。

必要條件

若要完成本節中的步驟，您必須執行下列各項：

- Python 3.9
- AWS CLI 第二版
- AWS SAM CLI 版本 1.75 或更新版本。如果您使用較舊版本的 AWS SAM CLI，請參閱升級 AWS SAM CLI。

部署範例 AWS SAM 應用程式

1. 使用 Hello World Python 範本來初始化應用程式。

```
sam init --app-template hello-world-powertools-python --name sam-app --package-type
 Zip --runtime python3.9 --no-tracing
```

2. 建置應用程式。

```
cd sam-app && sam build
```

3. 部署應用程式。

```
sam deploy --guided
```

4. 依照螢幕上的提示操作。若要接受互動體驗中提供的預設選項，請按下 Enter。

> **Note**
>
> 因為HelloWorldFunction 可能沒有定義授權，這可以嗎？ ，請務必輸入y。

5. 取得已部署應用程式的 URL：

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. 調用 API 端點：

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

成功的話，您將會看到以下回應：

```
{"message":"hello world"}
```

7. 若要取得函數的追蹤，請執行 sam 追蹤。

```
sam traces
```

追蹤輸出如下：

```
New XRay Service Graph
  Start time: 2023-02-03 14:59:50+00:00
  End time: 2023-02-03 14:59:50+00:00
  Reference Id: 0 - (Root) AWS::Lambda - sam-app-HelloWorldFunction-YBg8yfYtOc9j -
 Edges: [1]
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0.924
  Reference Id: 1 - AWS::Lambda::Function - sam-app-HelloWorldFunction-YBg8yfYtOc9j
 - Edges: []
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0.016
  Reference Id: 2 - client - sam-app-HelloWorldFunction-YBg8yfYtOc9j - Edges: [0]
   Summary_statistics:
     - total requests: 0
     - ok count(2XX): 0
     - error count(4XX): 0
```

```
      - fault count(5XX): 0
      - total response time: 0

XRay Event [revision 1] at (2023-02-03T14:59:50.204000) with id
 (1-63dd2166-434a12c22e1307ff2114f299) and duration (0.924s)
 - 0.924s - sam-app-HelloWorldFunction-YBg8yfYtOc9j [HTTP: 200]
 - 0.016s - sam-app-HelloWorldFunction-YBg8yfYtOc9j
   - 0.739s - Initialization
   - 0.016s - Invocation
     - 0.013s - ## lambda_handler
       - 0.000s - ## app.hello
   - 0.000s - Overhead
```

8. 這是可透過網際網路存取的公有 API 端點。建議您在測試後刪除端點。

```
sam delete
```

X-Ray 無法追蹤應用程式的所有請求。X-Ray 會套用取樣演算法以確保追蹤的效率，同時仍提供所有請求的代表範本。取樣率為每秒 1 次請求和 5% 的額外請求。

> **Note**
>
> 您無法針對函數設定 X-Ray 取樣率。

使用動力工具 AWS Lambda （Python）和跟 AWS CDK 踪

請按照下面的步驟下載，構建和部署示例你好世界 Python 應用程序集成動力工具 AWS Lambda （Python）模塊使用 AWS CDK. 此應用程式實作了基本 API 後端，並使用 Powertools 發送日誌、指標和追蹤。其包含 Amazon API Gateway 端點和 Lambda 函數。當您將 GET 請求傳送至 API Gateway 端點時，Lambda 函數會叫用、使用內嵌指標格式將記錄和指標傳送至 CloudWatch，並將追蹤傳送至 AWS X-Ray。函數會傳回 hello world 訊息。

必要條件

若要完成本節中的步驟，您必須執行下列各項：

- Python 3.9
- AWS CLI 第二版
- AWS CDK 第二版

- AWS SAM CLI 版本 1.75 或更新版本。如果您使用較舊版本的 AWS SAM CLI，請參閱升級 AWS SAM CLI。

部署範例 AWS CDK 應用程式

1. 為您的新應用程式建立專案目錄。

```
mkdir hello-world
cd hello-world
```

2. 初始化應用程式。

```
cdk init app --language python
```

3. 安裝 Python 相依項。

```
pip install -r requirements.txt
```

4. 在根資料夾下建立 lambda_function 目錄。

```
mkdir lambda_function
cd lambda_function
```

5. 建立檔案 app.py，並將下列程式碼新增至檔案。這是 Lambda 函數的程式碼。

```
from aws_lambda_powertools.event_handler import APIGatewayRestResolver
from aws_lambda_powertools.utilities.typing import LambdaContext
from aws_lambda_powertools.logging import correlation_paths
from aws_lambda_powertools import Logger
from aws_lambda_powertools import Tracer
from aws_lambda_powertools import Metrics
from aws_lambda_powertools.metrics import MetricUnit

app = APIGatewayRestResolver()
tracer = Tracer()
logger = Logger()
metrics = Metrics(namespace="PowertoolsSample")

@app.get("/hello")
@tracer.capture_method
def hello():
    # adding custom metrics
```

```
    # See: https://docs.powertools.aws.dev/lambda-python/latest/core/metrics/
    metrics.add_metric(name="HelloWorldInvocations", unit=MetricUnit.Count,
 value=1)

    # structured log
    # See: https://docs.powertools.aws.dev/lambda-python/latest/core/logger/
    logger.info("Hello world API - HTTP 200")
    return {"message": "hello world"}

# Enrich logging with contextual information from Lambda
@logger.inject_lambda_context(correlation_id_path=correlation_paths.API_GATEWAY_REST)
# Adding tracer
# See: https://docs.powertools.aws.dev/lambda-python/latest/core/tracer/
@tracer.capture_lambda_handler
# ensures metrics are flushed upon request completion/failure and capturing
 ColdStart metric
@metrics.log_metrics(capture_cold_start_metric=True)
def lambda_handler(event: dict, context: LambdaContext) -> dict:
    return app.resolve(event, context)
```

6. 開啟 hello_world 目錄。您應看到名稱為 hello_world_stack.py 的檔案。

```
cd ..
cd hello_world
```

7. 開啟 hello_world_stack.py，並將下列程式碼新增至檔案。其中包含建立 Lambda 函數的 Lambda 建構函式、設定 Powertools 的環境變數，並將記錄保留設定為一週，以及建立 REST API 的 ApiGatewayv1 個建構函式。

```
from aws_cdk import (
    Stack,
    aws_apigateway as apigwv1,
    aws_lambda as lambda_,
    CfnOutput,
    Duration
)
from constructs import Construct

class HelloWorldStack(Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)
```

```
        # Powertools Lambda Layer
        powertools_layer = lambda_.LayerVersion.from_layer_version_arn(
            self,
            id="lambda-powertools",
            # At the moment we wrote this example, the aws_lambda_python_alpha CDK
 constructor is in Alpha, o we use layer to make the example simpler
            # See https://docs.aws.amazon.com/cdk/api/v2/python/
aws_cdk.aws_lambda_python_alpha/README.html
            # Check all Powertools layers versions here: https://
docs.powertools.aws.dev/lambda-python/latest/#lambda-layer
            layer_version_arn=f"arn:aws:lambda:
{self.region}:017000801446:layer:AWSLambdaPowertoolsPythonV2:21"
        )

        function = lambda_.Function(self,
            'sample-app-lambda',
            runtime=lambda_.Runtime.PYTHON_3_9,
            layers=[powertools_layer],
            code = lambda_.Code.from_asset("./lambda_function/"),
            handler="app.lambda_handler",
            memory_size=128,
            timeout=Duration.seconds(3),
            architecture=lambda_.Architecture.X86_64,
            environment={
                "POWERTOOLS_SERVICE_NAME": "PowertoolsHelloWorld",
                "POWERTOOLS_METRICS_NAMESPACE": "PowertoolsSample",
                "LOG_LEVEL": "INFO"
            }
        )

        apigw = apigwv1.RestApi(self, "PowertoolsAPI",
 deploy_options=apigwv1.StageOptions(stage_name="dev"))

        hello_api = apigw.root.add_resource("hello")
        hello_api.add_method("GET", apigwv1.LambdaIntegration(function,
 proxy=True))

        CfnOutput(self, "apiUrl", value=f"{apigw.url}hello")
```

8. 部署您的應用程式。

```
cd ..
cdk deploy
```

9. 取得已部署應用程式的 URL：

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?OutputKey==`apiUrl`].OutputValue' --output text
```

10. 調用 API 端點：

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

成功的話，您將會看到以下回應：

```
{"message":"hello world"}
```

11. 若要取得函數的追蹤，請執行 sam 追蹤。

```
sam traces
```

追蹤輸出如下：

```
New XRay Service Graph
  Start time: 2023-02-03 14:59:50+00:00
  End time: 2023-02-03 14:59:50+00:00
  Reference Id: 0 - (Root) AWS::Lambda - sam-app-HelloWorldFunction-YBg8yfYtOc9j -
 Edges: [1]
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0.924
  Reference Id: 1 - AWS::Lambda::Function - sam-app-HelloWorldFunction-YBg8yfYtOc9j
 - Edges: []
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0.016
  Reference Id: 2 - client - sam-app-HelloWorldFunction-YBg8yfYtOc9j - Edges: [0]
   Summary_statistics:
     - total requests: 0
```

```
      - ok count(2XX): 0
      - error count(4XX): 0
      - fault count(5XX): 0
      - total response time: 0

XRay Event [revision 1] at (2023-02-03T14:59:50.204000) with id
 (1-63dd2166-434a12c22e1307ff2114f299) and duration (0.924s)
 - 0.924s - sam-app-HelloWorldFunction-YBg8yfYtOc9j [HTTP: 200]
 - 0.016s - sam-app-HelloWorldFunction-YBg8yfYtOc9j
   - 0.739s - Initialization
   - 0.016s - Invocation
     - 0.013s - ## lambda_handler
       - 0.000s - ## app.hello
   - 0.000s - Overhead
```

12. 這是可透過網際網路存取的公有 API 端點。建議您在測試後刪除端點。

```
cdk destroy
```

使用 ADOT 來檢測您的 Python 函數

ADOT 提供全受管 Lambda 層，包含使用 OTel SDK 收集遙測資料所需的一切內容。透過取用此層，您可以檢測 Lambda 函數，而無需修改任何函數程式碼。您還可以將層設定為對 OTel 進行自訂初始化。如需詳細資訊，請參閱 ADOT 文件中的針對 Lambda 上的 ADOT 收集器進行自訂組態設定。

對於 Python 執行時間，您可以新增適用於 ADOT Python 的AWS 受管 Lambda 層來自動檢測您的函數。此層同時適用於 arm64 和 x86_64 架構。有關如何添加此層的詳細說明，請參閱 AWS ADOT 文檔中的 Python 的 OpenTelemetry Lambda Support 發行版。

使用 X-Ray SDK 來檢測 Python 功能

若要記錄 Lambda 函數對應用程式中其他資源所進行之呼叫的詳細資料，您也可以使用 適用於 Python 的 AWS X-Ray SDK。若要取得開發套件，請將 `aws-xray-sdk` 套件新增至應用程式的相依性。

Example requirements.txt

```
jsonpickle==1.3
aws-xray-sdk==2.4.3
```

在函數代碼中，您可以通過使用`aws_xray_sdk.core`模塊修補`boto3`庫來檢測 AWS SDK 客戶端。

Example 功能-跟踪 AWS SDK 客戶端

```
import boto3
from aws_xray_sdk.core import xray_recorder
from aws_xray_sdk.core import patch_all

logger = logging.getLogger()
logger.setLevel(logging.INFO)
patch_all()

client = boto3.client('lambda')
client.get_account_settings()

def lambda_handler(event, context):
    logger.info('## ENVIRONMENT VARIABLES\r' + jsonpickle.encode(dict(**os.environ)))
  ...
```

新增正確的依賴項並進行必要的程式碼變更後，請透過 Lambda 主控台或 API 在函數的組態中啟用追蹤。

透過 Lambda 主控台來啟用追蹤

若要使用控制台在 Lambda 函數上切換主動追蹤，請按照下列步驟操作：

開啟主動追蹤

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇 組態 ，然後選擇 監控和操作工具 。
4. 選擇 編輯 。
5. 在 X-Ray 下，打開 主動追蹤 。
6. 選擇 儲存 。

透過 Lambda API 啟用追蹤

使用 AWS CLI 或 AWS SDK 在 Lambda 函數上設定追蹤功能，並使用下列 API 作業：

- UpdateFunction配置

- GetFunction配置
- CreateFunction

下列範例 AWS CLI 命令可在名為 my- function 的函式上啟用主動追蹤。

```
aws lambda update-function-configuration \
--function-name my-function \
--tracing-config Mode=Active
```

追蹤模式是您發布函數版本時版本特定組態的一部分。您無法變更已發佈版本上的追蹤模式。

使用啟動追蹤 AWS CloudFormation

若要啟動 AWS CloudFormation 範本中的`AWS::Lambda::Function`資源追蹤，請使用`TracingConfig`屬性。

Example function-inline.yml - 追蹤組態

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

對於 AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` 資源，請使用`Tracing`屬性。

Example template.yml - 追蹤組態

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

解讀 X-Ray 追蹤

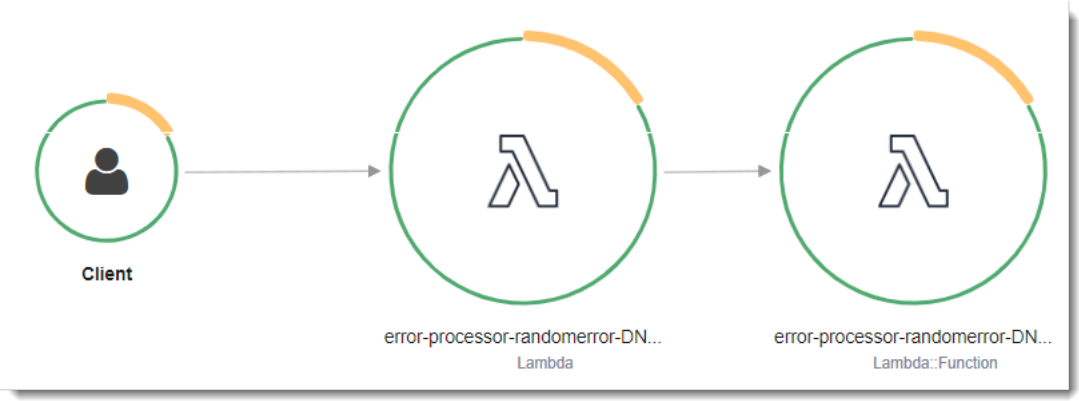
您的函數需要將追蹤資料上傳至 X-Ray 的許可。當您在 Lambda 主控台中啟用追蹤時，Lambda 會將必要的許可新增至函數的執行角色。否則，請將AWSXRayDaemonWriteAccess原則新增至執行角色。

設定主動追蹤之後，您可以透過應用程式來觀察特定請求。X-Ray 服務圖顯示了有關應用程式及其所有元件的資訊。下圖演示了具有兩個功能的應用程序。主要函式會處理事件，有時會傳回錯誤。頂部的第二個函數處理出現在第一個日誌組中的錯誤，並使用 AWS SDK 調用 X-Ray，Amazon 簡單存儲服務（Amazon S3）和亞馬遜 CloudWatch 日誌。

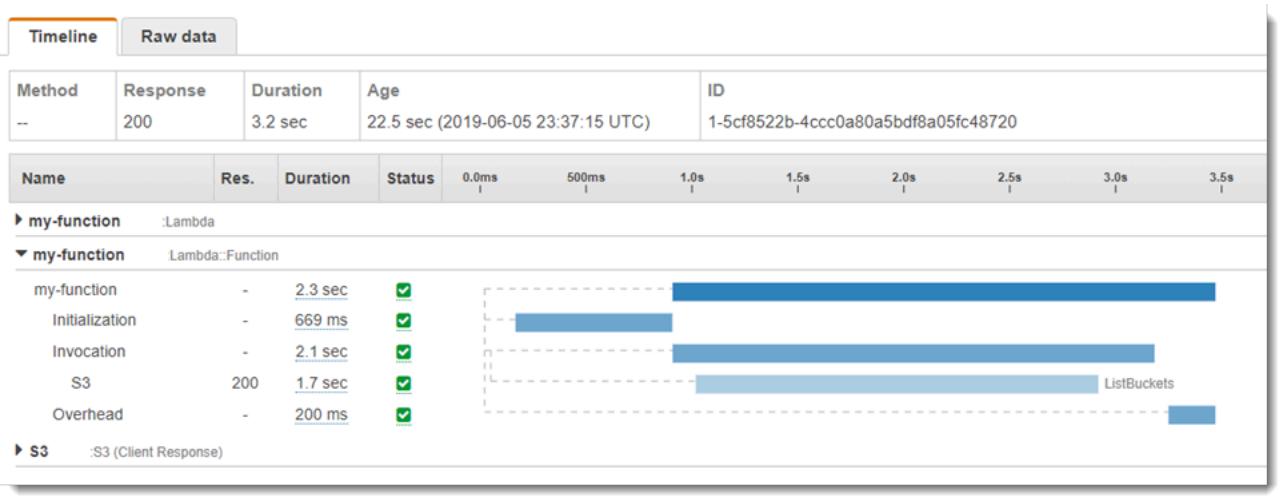
X-Ray 無法追蹤應用程式的所有請求。X-Ray 會套用取樣演算法以確保追蹤的效率，同時仍提供所有請求的代表範本。取樣率為每秒 1 次請求和 5% 的額外請求。

> **Note**
>
> 您無法針對函數設定 X-Ray 取樣率。

在 X-Ray 中，追蹤會記錄一或多個服務所處理之要求的相關資訊。Lambda 會記錄每個追蹤 2 個區段，在服務圖表上建立兩個節點。下列影像會強調顯示這兩個節點：



左側第一個節點代表接收調用請求的 Lambda 服務。第二個節點代表您特定的 Lambda 函數。下列範例顯示了具有這 2 個區段的追蹤。兩者都被命名為我的函數，但一個具有的起源`AWS::Lambda`，另一個具有的`AWS::Lambda::Function`起源。如果`AWS::Lambda`區段顯示錯誤，表示 Lambda 服務發生問題。如果`AWS::Lambda::Function`區段顯示錯誤，表示您的函數發生問題。



此範例會展開區`AWS::Lambda::Function`段，以顯示其三個子區段：

- 初始化 - 表示載入函數和執行初始化程式碼所花費的時間。只有函數的每個執行個體所處理的第一個事件會顯示此子區段。
- 調用 – 表示執行處理常式程式碼所花費的時間。
- 額外負荷 - 表示 Lambda 執行期為做好準備以處理下一個事件所花費的時間。

您也可以檢測 HTTP 用戶端、記錄 SQL 查詢，以及建立具有註釋和中繼資料的自訂子區段。如需詳細資訊，請參閱《AWS X-Ray 開發人員指南》中的 適用於 Python 的 AWS X-Ray SDK 許可。

> **定價**
>
> 作為免費方案的一部分，您可以每月免費使用 X-Ray 追蹤，最多達到一定限制。 AWS 達到閾值後，X-Ray 會收取追蹤儲存及擷取的費用。如需詳細資訊，請參閱 AWS X-Ray 定價。

將執行時間相依項存放存在層中 (X-Ray SDK)

如果您使用 X-Ray SDK 來檢測 AWS SDK 用戶端您的函數程式碼，您的部署套件可能會變得相當大。為了避免每次更新函數程式碼時上傳執行時間相依性，請將 X-Ray SDK 封裝在一個 Lambda 層中。

以下範例會顯示存放 適用於 Python 的 AWS X-Ray SDK的 `AWS::Serverless::LayerVersion` 資源。

Example template.yml - 相依性層

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: function/.
      Tracing: Active
      Layers:
        - !Ref libs
      ...
  libs:
    Type: AWS::Serverless::LayerVersion
    Properties:
      LayerName: blank-python-lib
      Description: Dependencies for the blank-python sample app.
      ContentUri: package/.
      CompatibleRuntimes:
        - python3.8
```

透過此組態，您只有在變更執行時間相依性時才會更新程式庫層。由於函數部署套件僅含有您的程式碼，因此有助於減少上傳時間。

為相依性建立圖層需要建置變更，才能在部署之前產生圖層封存。如需工作範例，請參閱 blank-python 範例應用程式。

使用 Ruby 建置 Lambda 函數

您可以在 AWS Lambda中執行 Ruby 程式碼。Lambda 提供用於執行程式碼來處理事件的 Ruby 執行期。您的程式碼會在包含您管理之 AWS SDK for Ruby AWS Identity and Access Management (IAM) 角色的登入資料的環境中執行。若要進一步瞭解 Ruby 執行階段隨附的 SDK 版本，請參閱the section called “包含執行階段的 SDK 版本”。

Lambda 支援以下 Ruby 執行期。

Ruby

名稱	識別符	作業系統	取代日期	封鎖函數建立	封鎖函數更新
紅寶石	`ruby3.3`	Amazon Linux 2023			
Ruby 3.2	`ruby3.2`	Amazon Linux 2			

建立 Ruby 函式

1. 開啟 Lambda 主控台。
2. 選擇建立函數。
3. 進行下列設定：
   - 函數名稱：輸入函數名稱。
   - 執行期：選擇 Ruby 3.2。
4. 選擇建立函數。
5. 若要設定測試事件，請選擇 Test (測試)。
6. 事件名稱輸入 **`test`**。
7. 選擇儲存變更。
8. 若要調用函數，請選擇 Test (測試)。

主控台將建立一個 Lambda 函數，其具有名為 `lambda_function.rb` 的單一來源檔案。您可以使用內建的程式碼編輯器編輯該檔案並加入更多檔案。選擇 Save (儲存) 以儲存變更。然後，若要執行程式碼，請選擇 Test (測試)。

> **Note**
>
> Lambda 主控台用 AWS Cloud9 來在瀏覽器中提供整合式開發環境。您也可以使用 AWS Cloud9 在自己的環境中開發 Lambda 函數。若要取得更多資訊，請參閱使用指南 AWS 工具組中的〈使用 AWS Lambda 函數〉。 AWS Cloud9

`lambda_function.rb` 檔案匯出名為 `lambda_handler` 的函數，它接受事件物件與內容物件。這就是在調用函數時，Lambda 呼叫的處理常式函數。Ruby 函數執行期會從 Lambda 中取得調用事件並其傳遞至處理常式。在函式組態中，處理常式值為 `lambda_function.lambda_handler`。

當您儲存函數程式碼時，Lambda 主控台會建立 .zip 封存檔部署套件。當您在主控台之外開發函數程式碼 (使用 IDE) 時，您需要建立部署套件將您的程式碼上傳到 Lambda 函數。

> **Note**
>
> 若要在您的本機環境中開始進行應用程式開發，請部署本指南 GitHub 儲存庫中提供的其中一個範例應用程式。
>
> 以 Ruby 編寫的範例 Lambda 應用程式
>
> - 空白紅寶石 — 一個 Ruby 函數，顯示日誌記錄，環境變量， AWS X-Ray 跟踪，圖層，單元測試和 SDK 的使用。 AWS
> - 適用於 AWS Lambda 的 Ruby 程式碼範例 — 以 Ruby 撰寫的程式碼範例，示範如何與 AWS Lambda 互動。

除了傳遞調用事件外，函式執行期還會傳遞內容物件至處理常式。內容物件包含了有關調用、函式以及執行環境的額外資訊。更多詳細資訊將另由環境變數提供。

您的 Lambda 函數隨附一個 CloudWatch 日誌記錄群組。函數運行時將有關每次調用的詳細信息發送到 CloudWatch 日誌。它在調用期間會轉送您的函數輸出的任何記錄。如果您的函數傳回錯誤，Lambda 會對該錯誤進行格式化之後傳回給調用端。

主題

- 包含執行階段的 SDK 版本
- 啟用 Yet Another Ruby JIT (YJIT)
- 在 Ruby 中定義 Lambda 函數處理程

- 使用 Ruby Lambda 函數的 .zip 封存檔
- 使用容器映像部署 Ruby Lambda 函數
- Ruby 中的 AWS Lambda 內容物件
- AWS Lambda 紅寶石中的函數登錄
- 檢測 Ruby 代碼 AWS Lambda

包含執行階段的 SDK 版本

Ruby 執行階段中包含的 AWS SDK 版本取決於執行階段版本和您的 AWS 區域. AWS SDK for Ruby 被設計為模塊化，並且由 AWS 服務. 若要尋找您正在使用的執行階段中包含之特定服務 gem 的版本號碼，請使用下列格式的程式碼建立 Lambda 函數。將您`Aws::S3`的程式碼使用的服務 gem 名稱取代`aws-sdk-s3`和。

```
require 'aws-sdk-s3'

def lambda_handler(event:, context:)
  puts "Service gem version: #{Aws::S3::GEM_VERSION}"
  puts "Core version: #{Aws::CORE_GEM_VERSION}"
end
```

啟用 Yet Another Ruby JIT (YJIT)

Ruby 3.2 執行期支援 YJIT，一個輕量級、簡約 Ruby JIT 編譯器。YJIT 提供了明顯更高的性能，但使用的記憶體也比 Ruby 解譯器更多。建議將 YJIT 用於 Ruby on Rails 工作負載。

依預設不會啟用 YJIT。若要為 Ruby 3.2 函數啟用 YJIT，請將 `RUBY_YJIT_ENABLE` 環境變數設定為 `1`。若要確認已啟用 YJIT，請列印 `RubyVM::YJIT.enabled?` 方法的結果。

Example – 確認已啟用 YJIT

```
puts(RubyVM::YJIT.enabled?())
# => true
```

在 Ruby 中定義 Lambda 函數處理程

Lambda 函數處理常式是您的函數程式碼中處理事件的方法。當有人呼叫您的函數時，Lambda 會執行處理常式方法。函數會執行，直到處理常式傳回回應、結束或逾時為止。

以下範例中，檔案 `function.rb` 定義了一個名為 `handler` 的處理常式方法。此處理常式函式接受兩個物件做為輸入並將傳回 JSON 文件。

Example function.rb

```
require 'json'

def handler(event:, context:)
    { event: JSON.generate(event), context: JSON.generate(context.inspect) }
end
```

在您的函數組態中，`handler` 設定會告知 Lambda 應至何處尋找處理常式。接續前述範例，此設定的正確值為 **`function.handler`**。其包括兩個以句點分隔的名稱：檔案的名稱和處理常式方法的名稱。

您也可以透過類別定義處理常式方法。以下範例在名為 `LambdaFunctions` 的模組之下透過名為 `Handler` 的類別定義 `process` 處理常式方法。

Example source.rb

```
module LambdaFunctions
  class Handler
    def self.process(event:,context:)
      "Hello!"
    end
  end
end
```

就本例而言，handler 設定是 **`source.LambdaFunctions::Handler.process`**。

處理常式接受的兩個物件分別為叫用事件和內容。事件是 Ruby 物件，其包含了由叫用端所提供的承載。如果承載是 JSON 文件，事件物件即為 Ruby 雜湊；否則將會是字串。內容物件具有方法和各項屬性，提供了有關叫用、函式以及執行環境的資訊。

每次叫用您的 Lambda 函數時都將執行函數處理常式。位於處理常式外部的靜態程式碼則是按照函式的每一執行個體各執行一次。如果您的處理常式使用了像是開發套件用戶端和資料庫連線之類的資源，您即可由處理常式方法外部建立該等資源，以供多次叫用時重複使用。

函式的每一執行個體均可處理多個叫用事件，但是一次僅處理一個事件。在任何特定時間內處理某一事件的執行個體數目稱為函式的並行數。如需 Lambda 執行環境的詳細資訊，請參閱 Lambda 執行環境。

使用 Ruby Lambda 函數的 .zip 封存檔

AWS Lambda 函數的代碼包含一個包含函數處理程序代碼的 .rb 文件，以及代碼所依賴的任何其他依賴項（gem）。若要將此函數程式碼部署到 Lambda，您可以使用部署套件。此套件可以是 .zip 封存檔或容器映像。如需搭配 Ruby 使用容器映像的詳細資訊，請參閱使用容器映像部署 Ruby Lambda 函數。

若要建立 .zip 封存檔的部署套件，您可以使用命令列工具的內建 .zip 封存檔公用程式，或任何其他 .zip 檔案公用程式 (例如 7zip)。以下各節顯示的範例假設您在 Linux 或 MacOS 環境中使用命令列 `zip` 工具。若要在 Windows 中使用相同命令，您可以安裝適用於 Linux 的 Windows 子系統，以取得 Ubuntu 和 Bash 的 Windows 整合版本。

請注意，Lambda 使用 POSIX 檔案許可，因此在建立 .zip 封存檔之前，您可能需要設定部署套件資料夾的許可。

以下各節中的範例命令使用 Bundler 公用程式，將相依項新增至部署套件。若要安裝 bundler，請執行下列命令。

```
gem install bundler
```

章節

Ruby 中的相依項

對於使用 Ruby 執行期的 Lambda 函數，相依項可以是任何 Ruby gem。使用 .zip 封存部署函數時，您可以使用函數程式碼將這些相依項新增至 .zip 檔案，或使用 Lambda 層。 圖層是單獨的 .zip 檔案，可以包含其他程式碼和內容。若要進一步了解如何使用 Lambda 層，請參閱 Lambda 層。

紅寶石執行階段包含 AWS SDK for Ruby. 如果函數使用 SDK，則不需要將其與程式碼綁定在一起。但是，若要維持相依項的完全控制或使用特定版本的 SDK，可以將其新增到函數的部署套件。您可以將 SDK 包含在 .zip 檔案中，也可以使用 Lambda 層進行新增。.zip 檔案或 Lambda 層中的相依項優先於

執行期中包含的版本。要了解您的運行時版本中包含了哪個版本的 SDK for Ruby，請參閱the section called “包含執行階段的 SDK 版本”。

在 AWS 共同責任模式下，您負責管理函數部署套件中的任何相依項。這包括套用更新和安全性修補程式。若要更新函數部署套件中的相依項，請先建立新的 .zip 檔案，然後將其上傳至 Lambda。如需詳細資訊，請參閱 建立含相依項的 .zip 部署套件 和 使用 .zip 檔案建立及更新 Ruby Lambda 函數。

建立不含相依項的 .zip 部署套件

如果您的函數程式碼沒有相依項，則 .zip 檔案只會包含具有函數處理常式程式碼的 .rb 檔案。使用您慣用的 zip 公用程式建立 .zip 檔案，並將 .rb 檔案放在根目錄下。如果 .rb 檔案不在 .zip 檔案的根目錄下，則 Lambda 將無法執行您的程式碼。

若要了解如何部署 .zip 檔案以建立新的 Lambda 函數或更新現有函數，請參閱使用 .zip 檔案建立及更新 Ruby Lambda 函數。

建立含相依項的 .zip 部署套件

如果您的函數程式碼相依於其他 Ruby gem，則您可以使用函數程式碼將這些相依項新增至 .zip 檔案，或使用 Lambda 層。本節中的指示說明如何在 .zip 部署套件中包含相依項。如需如何在層中包含相依項的指示，請參閱the section called “為相依項建立 Ruby 層”。

假設函數程式碼儲存在專案目錄中名為 `lambda_function.rb` 的檔案中。下列範例 CLI 命令會建立名為 `my_deployment_package.zip` 的 .zip 檔案，其中包含函數程式碼及其相依項。

建立部署套件

1. 在專案目錄中，建立 `Gemfile` 以在其中指定相依項。

   ```
   bundle init
   ```

2. 使用您偏好的文字編輯器，編輯 `Gemfile` 以指定函數的相依項。例如，若要使用 TZInfo gem，請編輯 `Gemfile`，如下所示。

   ```
   source "https://rubygems.org"
   gem "tzinfo"
   ```

3. 執行以下命令來安裝專案目錄中 `Gemfile` 指定的 gem。此命令將 `vendor/bundle` 設定為 gem 安裝的預設路徑。

   ```
   bundle config set --local path 'vendor/bundle' && bundle install
   ```

您應該會看到類似下列的輸出。

```
Fetching gem metadata from https://rubygems.org/..........
Resolving dependencies...
Using bundler 2.4.13
Fetching tzinfo 2.0.6
Installing tzinfo 2.0.6
...
```

> **Note**
>
> 若稍後再次在全域安裝 gem，請執行下列命令。
>
> ```
> bundle config set --local system 'true'
> ```

4. 建立 .zip 封存檔，其中包含具有函數處理常式程式碼的 `lambda_function.rb` 檔案以及在上一個步驟中安裝的相依項。

```
zip -r my_deployment_package.zip lambda_function.rb vendor
```

您應該會看到類似下列的輸出。

```
adding: lambda_function.rb (deflated 37%)
  adding: vendor/ (stored 0%)
  adding: vendor/bundle/ (stored 0%)
  adding: vendor/bundle/ruby/ (stored 0%)
  adding: vendor/bundle/ruby/3.2.0/ (stored 0%)
  adding: vendor/bundle/ruby/3.2.0/build_info/ (stored 0%)
  adding: vendor/bundle/ruby/3.2.0/cache/ (stored 0%)
  adding: vendor/bundle/ruby/3.2.0/cache/aws-eventstream-1.0.1.gem (deflated 36%)
...
```

為相依項建立 Ruby 層

本節中的指示說明如何在層中包含相依項。如需如何在部署套件中包含相依項的指示，請參閱the section called “建立含相依項的 .zip 部署套件”。

將層新增至函數時，Lambda 會將層內容載入該執行環境的 /opt 目錄。在每一次 Lambda 執行期中，PATH 變數已包含 /opt 目錄中的特定資料夾路徑。若要確保PATH變數會擷取圖層內容，您的圖層 .zip 檔案應該在下列資料夾路徑中具有其相依性：

- ruby/gems/2.7.0 (GEM_PATH)
- ruby/lib (RUBYLIB)

例如，您的層 .zip 檔案結構可能如下所示：

```
json.zip
# ruby/gems/2.7.0/
                | build_info
                | cache
                | doc
                | extensions
                | gems
                | # json-2.1.0
                # specifications
                  # json-2.1.0.gemspec
```

此外，Lambda 會自動偵測 /opt/lib 目錄中的程式庫，以及 /opt/bin 目錄中的二進位檔案。若要確保 Lambda 正確找到您的層內容，您也可以建立結構如下的層：

```
custom-layer.zip
# lib
    | lib_1
    | lib_2
# bin
    | bin_1
    | bin_2
```

封裝層之後，請參閱the section called “建立和刪除層”及the section called “新增層”，完成層設定。

建立含原生程式庫的 .zip 部署套件

諸如 nokogiri、nio4r 和 mysql 等許多常見 Ruby gem 包含用 C 語言編寫的原生延伸模組。將包含 C 程式碼的程式庫新增至部署套件時，必須正確建置套件，以確保其與 Lambda 執行環境相容。

對於生產應用程式，我們建議您使用 AWS Serverless Application Model (AWS SAM) 來建置和部署您的程式碼。在 AWS SAM 使用該sam build --use-container選項在 Lambda 類似的 Docker 容

器中構建函數。若要進一步了解如 AWS SAM 何使用部署函數程式碼，請參閱AWS SAM 開發人員指南中的建置應用程式。

若要建立包含含有原生擴充功能之 gem 的 .zip 部署套件 AWS SAM，您也可以使用容器將相依性封裝在與 Lambda Ruby 執行階段環境相同的環境中。若要完成這些步驟，必須在建置電腦上安裝 Docker。若要進一步了解如何安裝 Docker，請參閱 Install Docker Engine。

在 Docker 容器中建立 .zip 部署套件

1. 在本機建置電腦上建立一個資料夾，將容器儲存在其中。在該資料夾中，建立一個名為 `dockerfile` 的檔案並將以下程式碼貼到其中。

   ```
   FROM public.ecr.aws/sam/build-ruby3.2:latest-x86_64
   RUN gem update bundler
   CMD "/bin/bash"
   ```

2. 在建立 `dockerfile` 的資料夾中，執行以下命令以建立 Docker 容器。

   ```
   docker build -t awsruby32 .
   ```

3. 導覽到包含 `.rb` 檔案 (具有函數處理常式程式碼) 和 `Gemfile` (用於指定函數相依項) 的專案目錄。從該目錄內，執行下列命令以啟動 Lambda Ruby 容器。

   Linux/MacOS

   ```
   docker run --rm -it -v $PWD:/var/task -w /var/task awsruby32
   ```

   > **Note**
   >
   > 在 MacOS 中，您可能會看到警告，通知您所請求映像的平台與偵測到的主機平台不符。請忽略此警告。

   Windows PowerShell

   ```
   docker run --rm -it -v ${pwd}:var/task -w /var/task awsruby32
   ```

   當容器啟動時，應能看到一個 bash 提示。

```
bash-4.2#
```

4. 設定套件公用程式，以安裝本機 `vendor/bundle` 目錄中 `Gemfile` 指定的 gem，並安裝相依項。

```
bash-4.2# bundle config set --local path 'vendor/bundle' && bundle install
```

5. 使用函數程式碼和其相依項建立 .zip 部署套件。在此範例中，包含函數處理常式程式碼的檔案命名為 `lambda_function.rb`。

```
bash-4.2# zip -r my_deployment_package.zip lambda_function.rb vendor
```

6. 結束容器並回到本機專案目錄。

```
bash-4.2# exit
```

現在可以使用 .zip 檔案部署套件來建立或更新 Lambda 函數。請參閱[使用 .zip 檔案建立及更新 Ruby Lambda 函數]

使用 .zip 檔案建立及更新 Ruby Lambda 函數

建立 .zip 部署套件後，您可以使用該套件建立新的 Lambda 函數或更新現有函數。您可以使用 Lambda 主控台、和 Lambda API 來部署您的 .zip 套件。 AWS Command Line Interface您也可以使用 AWS Serverless Application Model (AWS SAM) 和 AWS CloudFormation 建立並更新 Lambda 函數。

Lambda 的 .zip 部署套件大小上限為 250 MB (解壓縮)。請注意，此限制適用於您上傳的所有檔案 (包括任何 Lambda 層) 的大小總和。

Lambda 執行時間需有許可才能讀取部署套裝服務中的檔案。在 Linux 權限八進制標記法中，Lambda 需要 644 個權限才能用於不可執行的檔案 (rw-r--r--) 和 755 個權限 () 用於目錄和可執行檔。rwxr-xr-x

在 Linux 和 MacOS 中，使用 `chmod` 命令變更部署套件中檔案和目錄的檔案許可。例如，若要提供可執行檔正確的許可，請執行下列命令。

```
chmod 755 <filepath>
```

若要在 Windows 中變更檔案許可，請參閱 Microsoft Windows 文件的 Set, View, Change, or Remove Permissions on an Object。

透過主控台使用 .zip 檔案建立及更新函數

若要建立新函數，您必須先在主控台中建立函數，然後上傳您的 .zip 封存檔。若要更新現有函數，請開啟函數的頁面，然後按照同樣的程序新增更新後的 .zip 檔案。

如果您的 .zip 檔案小於 50 MB，您可以透過直接從本機電腦上傳檔案來建立或更新函數。若 .zip 檔案大於 50 MB，您必須先將套件上傳至 Amazon S3 儲存貯體。如需有關如何使用將檔案上傳到 Amazon S3 儲存貯體的指示 AWS Management Console，請參閱開始使用 Amazon S3。若要使用上載檔案 AWS CLI，請參閱《使用指南》中的AWS CLI 〈移動物件〉。

> **Note**
>
> 您無法變更現有函數的部署套件類型 (.zip 或容器映像檔)。例如，您無法將容器映像函數轉換為使用 .zip 檔案封存。您必須建立新的函數。

若要建立新的函數 (主控台)

1. 開啟 Lambda 主控台的函數頁面，然後選擇建立函數。
2. 選擇 Author from scratch (從頭開始撰寫)。
3. 在基本資訊下，請執行下列動作：
   a. 在函數名稱中輸入函數名稱。
   b. 在執行期中選取要使用的執行期。
   c. (選用) 在架構中選擇要用於函數的指令集架構。預設架構值為 x86_64。請確定函數的 .zip 部署套件與您選取的指令集架構相容。
4. (選用) 在 許可 下，展開 變更預設執行角色 。您可建立新的執行角色，或使用現有的角色。
5. 選擇建立函數。Lambda 會使用您選擇的執行期建立一個基本的「Hello world」函數。

若要從本機電腦上傳 .zip 封存檔 (主控台)

1. 在 Lambda 主控台的函數頁面中選擇要上傳 .zip 檔案的函數。
2. 選取程式碼索引標籤。
3. 在程式碼來源窗格中選擇上傳來源。

4. 選擇 .zip 檔案。
5. 若要上傳 .zip 檔案，請執行下列操作：
   a. 選擇上傳，然後在檔案選擇器中選取您的 .zip 檔案。
   b. 選擇 Open (開啟)。
   c. 選擇儲存。

若要從 Amazon S3 儲存貯體上傳 .zip 封存檔 (控制台)

1. 在 Lambda 主控台的函數頁面中選擇要上傳新 .zip 檔案的函數。
2. 選取程式碼索引標籤。
3. 在程式碼來源窗格中選擇上傳來源。
4. 選擇 Amazon S3 位置。
5. 貼上 .zip 檔案的 Amazon S3 連結 URL，然後選擇儲存。

使用主控台程式碼編輯器更新 .zip 檔案函數

對於某些具有 .zip 部署套件的函數，您可以使用 Lambda 主控台的內建程式碼編輯器直接更新函數程式碼。若要使用此功能，您的函數必須符合下列條件：

- 您的函數必須使用其中一種轉譯語言執行期 (Python、Node.js 或 Ruby)
- 函數的部署套件必須小於 3MB。

具有容器映像部署套件之函數的函數程式碼無法直接在主控台中編輯。

若要使用主控台程式碼編輯器更新函數程式碼

1. 開啟 Lambda 主控台的函數頁面，然後選取您的函數。
2. 選取程式碼索引標籤。
3. 在程式碼來源窗格中，選取您的原始程式碼檔案，然後在整合式程式碼編輯器中加以編輯。
4. 完成編輯程式碼後，請選擇部署，以儲存變更並更新函數。

使用 .zip 檔案建立和更新函數 AWS CLI

您可以使用 AWS CLI 建立新函數，或使用 .zip 檔案更新現有函數。使用建立函數和update-function-code命令來部署您的 .zip 套件。如果您的 .zip 檔案小於 50 MB，則可以從本機建置電腦的檔案位置上傳 .zip 套件。若檔案較大，則必須先從 Amazon S3 儲存貯體上傳 .zip 套件。如需有關如何使用將檔案上傳到 Amazon S3 儲存貯體的指示 AWS CLI，請參閱使用AWS CLI 者指南中的移動物件。

> **Note**
>
> 如果您使用從 Amazon S3 儲存貯體上傳 .zip 檔案 AWS CLI，則該儲存貯體必須與您的函數位於 AWS 區域 相同的位置。

若要使用 .zip 檔案與建立新函數 AWS CLI，您必須指定下列項目：

- 函數名稱 (`--function-name`)
- 函數的執行期 (`--runtime`)
- 函數執行角色的 Amazon Resource Name (ARN) (`--role`)
- 函數程式碼中處理常式方法的名稱 (`--handler`)

您也必須指定 .zip 檔案的位置。如果您的 .zip 檔案位於本機建置電腦上的資料夾中，請使用 `--zip-file` 選項來指定檔案路徑，如下列範例命令所示。

```
aws lambda create-function --function-name myFunction \
--runtime ruby3.2 --handler lambda_function.lambda_handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--zip-file fileb://myFunction.zip
```

若要在 Amazon S3 儲存貯體中指定 .zip 檔案的位置，請使用如下列範例命令所示的 `--code` 選項。您只需針對版本控制的物件使用 `S3ObjectVersion` 參數。

```
aws lambda create-function --function-name myFunction \
--runtime ruby3.2 --handler lambda_function.lambda_handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--code S3Bucket=DOC-EXAMPLE-BUCKET,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion
```

若要使用 CLI 更新現有函數，您可以使用 `--function-name` 參數指定函數的名稱。您也必須指定要用來更新函數程式碼的 .zip 檔案的位置。如果您的 .zip 檔案位於本機建置電腦上的資料夾中，請使用 `--zip-file` 選項來指定檔案路徑，如下列範例命令所示。

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

若要在 Amazon S3 儲存貯體中指定 .zip 檔案的位置，請使用如下列範例命令所示的 `--s3-bucket` 和 `--s3-key` 選項。您只需針對版本控制的物件使用 `--s3-object-version` 參數。

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket DOC-EXAMPLE-BUCKET --s3-key myFileName.zip --s3-object-version myObject
 Version
```

透過 Lambda API 使用 .zip 檔案建立及更新函數

若要使用 .zip 封存檔建立及更新函數，請使用下列 API 操作：

- CreateFunction
- UpdateFunctionCode

使用 .zip 文件創建和更新函數 AWS SAM

AWS Serverless Application Model (AWS SAM) 是一個工具組，可協助簡化在 AWS上建置和執行無伺服器應用程式的程序。您可以在 YAML 或 JSON 範本中定義應用程式的資源，並使用 AWS SAM 命令列介面 (AWS SAM CLI) 來建置、封裝及部署應用程式。當您從 AWS SAM 範本建立 Lambda 函數時， AWS SAM 會使用函數程式碼和您指定的任何相依性，自動建立 .zip 部署套件或容器映像檔。若要進一步了解如 AWS SAM 何使用建置和部署 Lambda 函數，請參閱開AWS Serverless Application Model發人員指南 AWS SAM中的入門使用。

您也可以使用現有的 .zip 檔案封存 AWS SAM 來建立 Lambda 函數。若要使用建立 Lambda 函數 AWS SAM，您可以將 .zip 檔案儲存在 Amazon S3 儲存貯體或建置機器的本機資料夾中。如需有關如何使用將檔案上傳到 Amazon S3 儲存貯體的指示 AWS CLI，請參閱使用AWS CLI 者指南中的移動物件。

在 AWS SAM 範本中，`AWS::Serverless::Function`資源會指定您的 Lambda 函數。在本資源中設定下列屬性，以使用 .zip 封存檔建立函數：

- `PackageType`：設定為 `Zip`
- `CodeUri`-設定為函數程式碼的 Amazon S3 URI、本機資料夾的路徑或FunctionCode物件
- `Runtime`：設定為所選執行期

使用時 AWS SAM，如果您的 .zip 檔案大於 50MB，則不需要先將其上傳到 Amazon S3 儲存貯體。AWS SAM 可以從本地構建機器上的位置上傳 .zip 軟件包，最大允許大小為 250MB（解壓縮）。

若要進一步瞭解如何使用 .zip 檔案部署函數 AWS SAM，請參閱AWS SAM 開發人員指南AWS::Serverless::Function中的。

使用 .zip 文件創建和更新函數 AWS CloudFormation

您可以使 AWS CloudFormation 用 .zip 檔案封存來建立 Lambda 函數。若要使用 .zip 檔案建立 Lambda 函數，您必須先將檔案上傳至 Amazon S3 儲存貯體。如需有關如何使用將檔案上傳到 Amazon S3 儲存貯體的指示 AWS CLI，請參閱使用AWS CLI 者指南中的移動物件。

在 AWS CloudFormation 範本中，`AWS::Lambda::Function`資源會指定您的 Lambda 函數。在本資源中設定下列屬性，以使用 .zip 封存檔建立函數：

- `PackageType`：設定為 `Zip`
- `Code`：在 `S3Bucket` 和 `S3Key` 欄位中輸入 Amazon S3 儲存貯體名稱和 .zip 檔案名稱。
- `Runtime`：設定為所選執行期

AWS CloudFormation 產生的 .zip 檔案不能超過 4MB。若要進一步瞭解有關使用 .zip 檔案部署函數的更多資訊 AWS CloudFormation，請參閱使用AWS CloudFormation 者指南AWS::Lambda::Function中的。

使用容器映像部署 Ruby Lambda 函數

有三種方法可以為 Ruby Lambda 函數構建容器映像：

- 使用 Ruby 的 AWS 基本圖像

  AWS 基礎映像會預先載入語言執行期、用來管理 Lambda 與函數程式碼之間互動的執行期界面用戶端，以及用於本機測試的執行期界面模擬器。
- 使用 AWS 僅限作業系統的基本影像

  AWS 僅限作業系統的基本映像檔包含 Amazon Linux 散發和執行階段介面模擬器。這些映像常用於為編譯語言 (如 Go 和 Rust) 和 Lambda 不提供基礎映像的語言或語言版本 (如 Node.js 19) 建置容器映像。您還可以使用僅限作業系統的基礎映像來實作自訂執行期。若要使映像與 Lambda 相容，您必須在映像中加入適用於 Ruby 的執行期介面用戶端。
- 使用非AWS 基本圖像

  您可以使用其他容器登錄檔中的替代基礎映像 (例如 Alpine Linux 或 Debian)。您也可以使用組織建立的自訂映像。若要使映像與 Lambda 相容，您必須在映像中加入適用於 Ruby 的執行期介面用戶端。

> **Tip**
>
> 若要縮短 Lambda 容器函數變成作用中狀態所需的時間，請參閱 Docker 文件中的使用多階段建置。若要建置有效率的容器映像，請遵循撰寫 Dockerfiles 的最佳實務。

本頁面會說明如何為 Lambda 建置、測試和部署容器映像。

主題

AWS 紅寶石的基本圖像

AWS 提供 Ruby 的下列基本影像：

標籤	執行期	作業系統	Dockerfile	棄用
3.3	紅寶石	Amazon Linux 2023	碼頭文件對於紅寶石 3.3 GitHub	
3.2	Ruby 3.2	Amazon Linux 2	碼頭文件的紅寶石 3.2 GitHub	

Amazon ECR 儲存庫：gallery.ecr.aws/lambda/ruby

使用 Ruby 的 AWS 基本圖像

必要條件

若要完成本節中的步驟，您必須執行下列各項：

- AWS Command Line Interface (AWS CLI) 第二版
- Docker
- Ruby

從基礎映像建立映像

建立適用於 Ruby 的容器映像

1. 建立專案的目錄，然後切換至該目錄。

```
mkdir example
cd example
```

2. 建立稱為 Gemfile 的新檔案。這是您列出應用程序所需 RubyGems 軟件包的地方。可從中取 AWS SDK for Ruby 得 RubyGems。您應該選擇要安裝的特定 AWS 服務寶石。例如，若要使用 Ruby Gem for Lambda，您的 Gemfile 看起來應該會像：

```
source 'https://rubygems.org'

gem 'aws-sdk-lambda'
```

或者，aws-sdk 寶石包含所有可用的 AWS 服務寶石。這個 Gem 非常大，我們建議您僅在依賴許多 AWS 服務時才使用它。

3. 使用套件安裝作業安裝 Gemfile 中指定的相依項。

```
bundle install
```

4. 建立稱為 lambda_function.rb 的新檔案。您可以將下列範例函數程式碼新增至檔案進行測試，或使用您自己的函數程式碼。

Example Ruby 函數

```
module LambdaFunction
  class Handler
    def self.process(event:,context:)
      "Hello from Lambda!"
    end
  end
end
```

5. 建立新的 Dockerfile。下列範例 Dockerfile 使用 AWS 基礎映像。此 Dockerfile 使用下列組態：

- 將 FROM 屬性設定為基礎映像的 URI。
- 使用 COPY 指令{LAMBDA_TASK_ROOT}，將函數程式碼和執行階段相依性複製到 Lambda 定義的環境變數。
- 將 CMD 引數設定為 Lambda 函數處理常式。

Example Dockerfile

```
FROM public.ecr.aws/lambda/ruby:3.2

# Copy Gemfile and Gemfile.lock
COPY Gemfile Gemfile.lock ${LAMBDA_TASK_ROOT}/

# Install Bundler and the specified gems
RUN gem install bundler:2.4.20 && \
    bundle config set --local path 'vendor/bundle' && \
    bundle install

# Copy function code
```

```
COPY lambda_function.rb ${LAMBDA_TASK_ROOT}/

# Set the CMD to your handler (could also be done as a parameter override outside
 of the Dockerfile)
CMD [ "lambda_function.LambdaFunction::Handler.process" ]
```

6. 使用 docker build 命令建立 Docker 映像檔。以下範例將映像命名為 docker-image 並為其提供 test 標籤。

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> 此命令會指定 --platform linux/amd64 選項，確保無論建置機器的架構為何，您的容器都與 Lambda 執行環境相容。如果您打算使用 ARM64 指令集架構建立 Lambda 函數，務必將命令變更為改用 --platform linux/arm64 選項。

(選用) 在本機測試映像

1. 使用 docker run 命令啟動 Docker 影像。在此範例中，docker-image 為映像名稱，test 為標籤。

```
docker run --platform linux/amd64 -p 9000:8080 docker-image:test
```

此命令將映像作為容器執行，並在 localhost:9000/2015-03-31/functions/function/invocations 建立本機端點。

> **Note**
>
> 如果您為 ARM64 指令集架構建立 Docker 映像檔，請務必將--platform linux/*arm64*選項改用選項。--platform linux/*amd64*

2. 從新的終端機視窗，將事件張貼至本機端點。

Linux/macOS

在 Linux 或 macOS 中，執行下列 curl 命令：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

在中 PowerShell，執行下列`Invoke-WebRequest`命令：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

3. 取得容器 ID。

```
docker ps
```

4. 使用 docker kill 命令停止容器。在此命令中，將 3766c4ab331c 替換為上一步驟中的容器 ID。

```
docker kill 3766c4ab331c
```

部署映像

若要將映像上傳至 Amazon ECR 並建立 Lambda 函數

1. 使用 get-login-password 命令，向 Amazon ECR 登錄檔驗證 Docker CLI。

   - 將`--region`值設定為您 AWS 區域 要建立 Amazon ECR 儲存庫的位置。
   - `111122223333`用您的 AWS 帳戶 身份證替換。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 使用 create-repository 命令在 Amazon ECR 中建立儲存庫。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 儲存庫必須與 Lambda 函數位於 AWS 區域 相同。

如果成功，您將會看到以下回應：

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. 從上一步驟的輸出中複製 `repositoryUri`。
4. 執行 docker tag 命令，將 Amazon ECR 儲存庫中的本機映像標記為最新版本。在此命令中：
   - 將 `docker-image:test` 替換為 Docker 映像檔的名稱和標籤。

- 將 <ECRrepositoryUri> 替換為複製的 repositoryUri。確保在 URI 的末尾包含 :latest。

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

範例：

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. 執行 docker push 命令，將本機映像部署至 Amazon ECR 儲存庫。確保在儲存庫 URI 的末尾包含 :latest。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 建立函數的執行角色 (若您還沒有的話)。在下一個步驟中您需要角色的 Amazon Resource Name (ARN)。
7. 建立 Lambda 函數。對於 ImageUri，從之前的設定中指定儲存庫 URI。確保在 URI 的末尾包含 :latest。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> 只要映像與 Lambda 函數位於相同的區域，您就可以使用不同 AWS 帳戶中的映像檔建立函數。如需詳細資訊，請參閱 Amazon ECR 跨帳戶許可。

8. 調用函數。

```
aws lambda invoke --function-name hello-world response.json
```

您應該看到如下回應：

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 若要查看函數的輸出，請檢查 `response.json` 檔案。

若要更新函數程式碼，您必須再次建置映像、將新映像上傳到 Amazon ECR 存放庫，然後使用 update-function-code 命令將映像部署到 Lambda 函數。

Lambda 將圖像標記解析為特定的圖像摘要。這表示如果您將用於部署函數的映像標籤指向 Amazon ECR 中的新映像，Lambda 不會自動更新函數以使用新映像。若要將新映像部署到相同的 Lambda 函數，您必須使用`update-function-code`命令，即使 Amazon ECR 中的影像標籤保持不變。

透過執行期介面用戶端使用替代基礎映像

如果您使用僅限作業系統的基礎映像或替代的基礎映像，則必須在映像中加入執行期介面用戶端。執行期介面用戶端會讓您擴充 Lambda 執行階段 API，管理 Lambda 與函數程式碼之間的互動。

使用 RubyGems .org 套件管理員為 Ruby 安裝 Lambda 執行階段介面用戶端：

```
gem install aws_lambda_ric
```

您也可以從下載 Ruby 執行階段介面用戶端 GitHub。執行期介面用戶端支援 Ruby 2.5.x 至 2.7.x 版。

下面的例子演示了如何使用非AWS 基本圖像為 Ruby 構建容器映像。範例 Dockerfile 使用官方 Ruby 基礎映像。Dockerfile 包含執行期界面用戶端。

必要條件

若要完成本節中的步驟，您必須執行下列各項：

- AWS Command Line Interface (AWS CLI) 第二版
- Docker
- Ruby

使用替代基礎映像建立映像

使用替代基礎映像為 Ruby 建立容器映像

1. 建立專案的目錄，然後切換至該目錄。

```
mkdir example
cd example
```

2. 建立稱為 Gemfile 的新檔案。這是您列出應用程序所需 RubyGems 軟件包的地方。可從中取 AWS SDK for Ruby 得 RubyGems。您應該選擇要安裝的特定 AWS 服務寶石。例如，若要使用 Ruby Gem for Lambda，您的 Gemfile 看起來應該會像：

```
source 'https://rubygems.org'

gem 'aws-sdk-lambda'
```

或者，aws-sdk 寶石包含所有可用的 AWS 服務寶石。這個 Gem 非常大，我們建議您僅在依賴許多 AWS 服務時才使用它。

3. 使用套件安裝作業安裝 Gemfile 中指定的相依項。

```
bundle install
```

4. 建立稱為 lambda_function.rb 的新檔案。您可以將下列範例函數程式碼新增至檔案進行測試，或使用您自己的函數程式碼。

Example Ruby 函數

```
module LambdaFunction
  class Handler
    def self.process(event:,context:)
      "Hello from Lambda!"
    end
  end
end
```

5. 建立新的 Dockerfile。下列 Dockerfile 使用 Ruby 基礎映像，而非 AWS 基礎映像。Dockerfile 包含 適用於 Ruby 的執行期介面用戶端，可讓映像與 Lambda 相容。或者，您可以將執行期界面用戶端新增到應用程式的 Gemfile 中。

- 將 FROM 屬性設定為 Ruby 基礎映像。

- 建立函數程式碼的目錄，以及指向該目錄的環境變數。在此範例中，目錄是/var/task，它會反映 Lambda 執行環境。但是，您可以為函數代碼選擇任何目錄，因為 Dockerfile 不使用 AWS 基本映像。
- 將 ENTRYPOINT 設為您希望 Docker 容器在啟動時執行的模組。在此案例中，模組是執行期界面用戶端。
- 將 CMD 引數設定為 Lambda 函數處理常式。

Example Dockerfile

```
FROM ruby:2.7

# Install the runtime interface client for Ruby
RUN gem install aws_lambda_ric

# Add the runtime interface client to the PATH
ENV PATH="/usr/local/bundle/bin:${PATH}"

# Create a directory for the Lambda function
ENV LAMBDA_TASK_ROOT=/var/task
RUN mkdir -p ${LAMBDA_TASK_ROOT}
WORKDIR ${LAMBDA_TASK_ROOT}

# Copy Gemfile and Gemfile.lock
COPY Gemfile Gemfile.lock ${LAMBDA_TASK_ROOT}/

# Install Bundler and the specified gems
RUN gem install bundler:2.4.20 && \
    bundle config set --local path 'vendor/bundle' && \
    bundle install

# Copy function code
COPY lambda_function.rb ${LAMBDA_TASK_ROOT}/

# Set runtime interface client as default command for the container runtime
ENTRYPOINT [ "aws_lambda_ric" ]

# Set the CMD to your handler (could also be done as a parameter override outside
 of the Dockerfile)
CMD [ "lambda_function.LambdaFunction::Handler.process" ]
```

6. 使用 docker build 命令建立 Docker 映像檔。以下範例將映像命名為 docker-image 並為其提供 test 標籤。

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> 此命令會指定 --platform linux/amd64 選項，確保無論建置機器的架構為何，您的容器都與 Lambda 執行環境相容。如果您打算使用 ARM64 指令集架構建立 Lambda 函數，務必將命令變更為改用 --platform linux/arm64 選項。

(選用) 在本機測試映像

使用 執行期界面模擬器 以在本機測試映像。您可以將模擬器構建到映像中，也可以使用以下步驟將其安裝在本地計算機上。

若要在本機電腦上安裝並執行執行期介面模擬器

1. 從您的項目目錄中運行以下命令以下載運行時接口仿真器（x86-64 架構） GitHub 並將其安裝在本地計算機上。

   Linux/macOS

```
mkdir -p ~/.aws-lambda-rie && \
    curl -Lo ~/.aws-lambda-rie/aws-lambda-rie https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie && \
    chmod +x ~/.aws-lambda-rie/aws-lambda-rie
```

   要安裝 arm64 模擬器，請使用以下命令替換上一個命令中的 GitHub 存儲庫 URL：

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
```

   PowerShell

```
$dirPath = "$HOME\.aws-lambda-rie"
if (-not (Test-Path $dirPath)) {
    New-Item -Path $dirPath -ItemType Directory
```

```
}

$downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/
releases/latest/download/aws-lambda-rie"
$destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
```

若要安裝 arm64 模擬器，請將 $downloadLink 更換為下列項目：

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

2. 使用 docker run 命令啟動 Docker 影像。注意下列事項：

   - docker-image 是映像名稱，而 test 是標籤。
   - aws_lambda_ric lambda_function.LambdaFunction::Handler.process 是 Dockerfile 中的 ENTRYPOINT，後面接著 CMD。

   Linux/macOS

   ```
   docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p
    9000:8080 \
       --entrypoint /aws-lambda/aws-lambda-rie \
       docker-image:test \
           aws_lambda_ric lambda_function.LambdaFunction::Handler.process
   ```

   PowerShell

   ```
   docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p
    9000:8080 `
   --entrypoint /aws-lambda/aws-lambda-rie `
   docker-image:test `
       aws_lambda_ric lambda_function.LambdaFunction::Handler.process
   ```

   此命令將映像作為容器執行，並在 localhost:9000/2015-03-31/functions/function/invocations 建立本機端點。

> **Note**
>
> 如果您為 ARM64 指令集架構建立 Docker 映像檔，請務必將`--platform linux/arm64`選項改用選項。`--platform linux/amd64`

3. 將事件張貼至本機端點。

   Linux/macOS

   在 Linux 或 macOS 中，執行下列 `curl` 命令：

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
    '{"payload":"hello world!"}'
   ```

   PowerShell

   在中 PowerShell，執行下列`Invoke-WebRequest`命令：

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{}' -ContentType "application/json"
   ```

   此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
    "application/json"
   ```

4. 取得容器 ID。

   ```
   docker ps
   ```

5. 使用 docker kill 命令停止容器。在此命令中，將 3766c4ab331c 替換為上一步驟中的容器 ID。

```
docker kill 3766c4ab331c
```

部署映像

若要將映像上傳至 Amazon ECR 並建立 Lambda 函數

1. 使用 get-login-password 命令，向 Amazon ECR 登錄檔驗證 Docker CLI。
   - 將--region值設定為您 AWS 區域 要建立 Amazon ECR 儲存庫的位置。
   - 111122223333用您的 AWS 帳戶 身份證替換。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 使用 create-repository 命令在 Amazon ECR 中建立儲存庫。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 儲存庫必須與 Lambda 函數位於 AWS 區域 相同。

如果成功，您將會看到以下回應：

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
```

```
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. 從上一步驟的輸出中複製 `repositoryUri`。
4. 執行 docker tag 命令，將 Amazon ECR 儲存庫中的本機映像標記為最新版本。在此命令中：
   - 將 `docker-image:test` 替換為 Docker 映像檔的名稱和標籤。
   - 將 `<ECRrepositoryUri>` 替換為複製的 `repositoryUri`。確保在 URI 的末尾包含 `:latest`。

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

範例：

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. 執行 docker push 命令，將本機映像部署至 Amazon ECR 儲存庫。確保在儲存庫 URI 的末尾包含 `:latest`。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 建立函數的執行角色 (若您還沒有的話)。在下一個步驟中您需要角色的 Amazon Resource Name (ARN)。
7. 建立 Lambda 函數。對於 `ImageUri`，從之前的設定中指定儲存庫 URI。確保在 URI 的末尾包含 `:latest`。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> 只要映像與 Lambda 函數位於相同的區域，您就可以使用不同 AWS 帳戶中的映像檔建立函數。如需詳細資訊，請參閱 Amazon ECR 跨帳戶許可。

8. 調用函數。

```
aws lambda invoke --function-name hello-world response.json
```

您應該看到如下回應：

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 若要查看函數的輸出，請檢查 `response.json` 檔案。

若要更新函數程式碼，您必須再次建置映像、將新映像上傳到 Amazon ECR 存放庫，然後使用 update-function-code 命令將映像部署到 Lambda 函數。

Lambda 將圖像標記解析為特定的圖像摘要。這表示如果您將用於部署函數的映像標籤指向 Amazon ECR 中的新映像，Lambda 不會自動更新函數以使用新映像。若要將新映像部署到相同的 Lambda 函數，您必須使用`update-function-code`命令，即使 Amazon ECR 中的影像標籤保持不變。

Ruby 中的 AWS Lambda 內容物件

當 Lambda 執行您的函數時，它會將內容物件傳遞至處理常式。此物件提供的方法和各項屬性提供了有關調用、函式以及執行環境的資訊。

內容方法

- `get_remaining_time_in_millis` - 傳回執行逾時前剩餘的毫秒數。

內容屬性

- `function_name` – Lambda 函數的名稱。
- `function_version` – 函數的版本。
- `invoked_function_arn` - 用於調用此函數的 Amazon Resource Name (ARN)。指出調用者是否指定版本號或別名。
- `memory_limit_in_mb` - 分配給函數的記憶體數量。
- `aws_request_id` - 調用請求的識別符。
- `log_group_name` - 函數的日誌群組。
- `log_stream_name` - 函數執行個體的記錄串流。
- `deadline_ms` - 執行逾時的日期，以 Unix 時間毫秒為單位。
- `identity` - (行動應用程式) 已授權請求的 Amazon Cognito 身分的相關資訊。
- `client_context` - (行動應用程式) 用戶端應用程式提供給 Lambda 的用戶端內容。

AWS Lambda 紅寶石中的函數登錄

AWS Lambda 代表您自動監控 Lambda 函數，並將日誌傳送到 Amazon CloudWatch。您的 Lambda 函數隨附一個 CloudWatch 日誌記錄群組和函數每個執行個體的日誌串流。Lambda 執行期環境會將每次調用的詳細資訊傳送至日誌串流，並且轉傳來自函數程式碼的日誌及其他輸出。如需詳細資訊，請參閱 使用 Amazon CloudWatch 日誌 AWS Lambda。

本頁說明如何從 Lambda 函數的程式碼產生記錄輸出，或使用 Lambda 主控台或主控台存取 CloudWatch 日誌。 AWS Command Line Interface

章節

建立傳回日誌的函數

若要由您的函式程式碼輸出日誌，可使用 `puts` 陳述式或者任何能夠寫入 `stdout` 或 `stderr` 的記錄程式庫。下列範例會記錄環境變數和事件物件的值。

Example lambda_function.rb

```
# lambda_function.rb

def handler(event:, context:)
    puts "## ENVIRONMENT VARIABLES"
    puts ENV.to_a
    puts "## EVENT"
    puts event.to_a
end
```

Example 記錄格式

```
START RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95 Version: $LATEST
## ENVIRONMENT VARIABLES
```

```
environ({'AWS_LAMBDA_LOG_GROUP_NAME': '/aws/lambda/my-function',
 'AWS_LAMBDA_LOG_STREAM_NAME': '2020/01/31/[$LATEST]3893xmpl7fac4485b47bb75b671a283c',
 'AWS_LAMBDA_FUNCTION_NAME': 'my-function', ...})
## EVENT
{'key': 'value'}
END RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95
REPORT RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95  Duration: 15.74 ms  Billed
 Duration: 16 ms Memory Size: 128 MB Max Memory Used: 56 MB  Init Duration: 130.49 ms
XRAY TraceId: 1-5e34a614-10bdxmplf1fb44f07bc535a1   SegmentId: 07f5xmpl2d1f6f85
 Sampled: true
```

Ruby 執行時間會記錄每次調用的 START、END 和 REPORT 行。報告明細行提供下列詳細資訊。

REPORT 行資料欄位

- RequestId— 呼叫的唯一要求 ID。
- 持續時間 - 函數的處理常式方法處理事件所花費的時間量。
- 計費持續時間 - 調用的計費時間量。
- 記憶體大小 - 分配給函數的記憶體數量。
- 使用的記憶體上限 - 函數所使用的記憶體數量。
- 初始化持續時間 - 對於第一個提供的請求，這是執行期載入函數並在處理常式方法之外執行程式碼所花費的時間量。
- XRAY TraceId — 針對追蹤的要求，則為AWS X-Ray 追蹤識別碼。
- SegmentId— 針對追蹤的請求，X-Ray 區段 ID。
- 已取樣 - 對於追蹤的請求，這是取樣結果。

如需更詳細的日誌，請使用 the section called “記錄程式庫”。

使用 Lambda 主控台

您可以在調用 Lambda 函數之後，使用 Lambda 主控台來檢視日誌輸出。

如果可以從內嵌程式碼編輯器測試您的程式碼，您會在執行結果中找到日誌。使用主控台測試功能以調用函數時，您會在詳細資訊區段找到日誌輸出。

使用控 CloudWatch 制台

您可以使用 Amazon 主 CloudWatch 控台來檢視所有 Lambda 函數叫用的日誌。

在 CloudWatch 主控台上檢視記錄檔

1. 在主控台上開啟 [記錄群組] 頁 CloudWatch 面。
2. 選擇您的函數的日誌群組 (/aws/lambda/*your-function-name*)。
3. 選擇日誌串流

每個日誌串流都會對應至函式的執行個體。當您更新 Lambda 函數，以及建立額外執行個體以處理多個並行調用時，便會出現日誌串流。要查找特定調用的日誌，我們建議使用檢測您的函數。 AWS X-Ray X-Ray 會在追蹤內記錄有關請求和日誌串流的詳細資訊。

使用 AWS Command Line Interface （ AWS CLI ）

這 AWS CLI 是一種開放原始碼工具，可讓您使用命令列殼層中的命令與 AWS 服務互動。若要完成本節中的步驟，您必須執行下列各項：

- AWS Command Line Interface (AWS CLI) 第二版
- AWS CLI -快速配置 `aws configure`

您可以透過 AWS CLI，使用 `--log-type` 命令選項來擷取要調用的日誌。其回應將包含 `LogResult` 欄位，內含該次調用的 base64 編碼日誌 (最大達 4 KB)。

Example 擷取日誌 ID

下列範例顯示如何從名稱為 `my-function` 的函數的 `LogResult` 欄位來擷取日誌 ID。

```
aws lambda invoke --function-name my-function out --log-type Tail
```

您應該會看到下列輸出：

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example 解碼日誌

在相同的命令提示中，使用 base64 公用程式來解碼日誌。下列範例顯示如何擷取 my-function 的 base64 編碼日誌。

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

如果您使用的是 AWS CLI 版本 2，則需要此cli-binary-format選項。若要讓此成為預設的設定，請執行 aws configure set cli-binary-format raw-in-base64-out。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

您應該會看到下列輸出：

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

該 base64 公用程式可在 Linux、macOS 和 Ubuntu on Windows 上使用。macOS 使用者可能需要使用 base64 -D。

Example get-logs.sh 指令碼

在相同的命令提示中，使用下列指令碼下載最後五個日誌事件。該指令碼使用 sed 以從輸出檔案移除引述，並休眠 15 秒以使日誌可供使用。輸出包括來自 Lambda 的回應以及來自 get-log-events 命令的輸出。

複製下列程式碼範例的內容，並將您的 Lambda 專案目錄儲存為 get-logs.sh。

如果您使用的是 AWS CLI 版本 2，則需要此cli-binary-format選項。若要讓此成為預設的設定，請執行 aws configure set cli-binary-format raw-in-base64-out。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
```

```
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS 和 Linux (僅限)

在相同的命令提示中，macOS 和 Linux 使用者可能需要執行下列命令，以確保指令碼可執行。

```
chmod -R 755 get-logs.sh
```

Example 擷取最後五個記錄事件

在相同的命令提示中，執行下列指令碼以取得最後五個日誌事件。

```
./get-logs.sh
```

您應該會看到下列輸出：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
```

```
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```

刪除日誌

當您刪除函數時，不會自動刪除日誌群組。若要避免無限期地儲存日誌，請刪除日誌群組，或設定保留期間，系統會在該時間之後自動刪除日誌。

記錄程式庫

Ruby 記錄程式庫會傳回易於讀取的精簡日誌。使用記錄公用程式輸出與函數相關的詳細資訊、訊息和錯誤碼。

```
# lambda_function.rb

require 'logger'

def handler(event:, context:)
  logger = Logger.new($stdout)
  logger.info('## ENVIRONMENT VARIABLES')
  logger.info(ENV.to_a)
  logger.info('## EVENT')
  logger.info(event)
  event.to_a
end
```

來自 `logger` 的輸出包含記錄等級、時間戳記和請求 ID。

```
START RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125 Version: $LATEST
[INFO]  2020-01-31T22:12:58.534Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    ##
 ENVIRONMENT VARIABLES

[INFO]  2020-01-31T22:12:58.534Z    1c8df7d3-xmpl-46da-9778-518e6eca8125
    environ({'AWS_LAMBDA_LOG_GROUP_NAME': '/aws/lambda/my-function',
 'AWS_LAMBDA_LOG_STREAM_NAME': '2020/01/31/[$LATEST]1bbe51xmplb34a2788dbaa7433b0aa4d',
 'AWS_LAMBDA_FUNCTION_NAME': 'my-function', ...})

[INFO]  2020-01-31T22:12:58.535Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    ## EVENT

[INFO]  2020-01-31T22:12:58.535Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    {'key':
 'value'}

END RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125
REPORT RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125  Duration: 2.75 ms   Billed
 Duration: 3 ms Memory Size: 128 MB Max Memory Used: 56 MB  Init Duration: 113.51 ms
XRAY TraceId: 1-5e34a66a-474xmpl7c2534a87870b4370   SegmentId: 073cxmpl3e442861
 Sampled: true
```

檢測 Ruby 代碼 AWS Lambda

Lambda 與 AWS X-Ray 之整合，可讓您追蹤、偵錯和最佳化 Lambda 應用程式。您可以使用 X-Ray 來追踪請求，因為它會周遊應用程式中的資源，從前端 API 到後端的存儲體和資料庫。只需將 X-Ray SDK 庫添加到構建配置中，您就可以記錄函數對 AWS 服務進行的任何調用的錯誤和延遲。

設定主動追蹤之後，您可以透過應用程式來觀察特定請求。X-Ray 服務圖顯示了有關應用程式及其所有元件的資訊。下圖演示了具有兩個功能的應用程序。主要函式會處理事件，有時會傳回錯誤。頂部的第二個函數處理出現在第一個日誌組中的錯誤，並使用 AWS SDK 調用 X-Ray，Amazon 簡單存儲服務（Amazon S3）和亞馬遜 CloudWatch 日誌。

若要使用控制台在 Lambda 函數上切換主動追蹤，請按照下列步驟操作：

開啟主動追蹤

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇 組態 ，然後選擇 監控和操作工具 。
4. 選擇 編輯 。
5. 在 X-Ray 下，打開 主動追蹤 。
6. 選擇 儲存 。

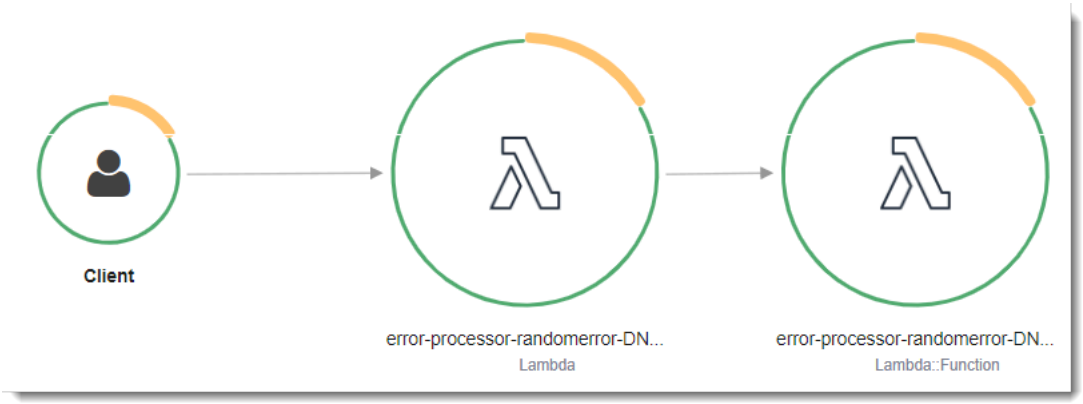
> **定價**
>
> 作為免費方案的一部分，您可以每月免費使用 X-Ray 追蹤，最多達到一定限制。 AWS 達到閾值後，X-Ray 會收取追蹤儲存及擷取的費用。如需詳細資訊，請參閱 AWS X-Ray 定價。

您的函數需要將追蹤資料上傳至 X-Ray 的許可。當您在 Lambda 主控台中啟用追蹤時，Lambda 會將必要的許可新增至函數的執行角色。否則，請將AWSXRayDaemonWriteAccess原則新增至執行角色。

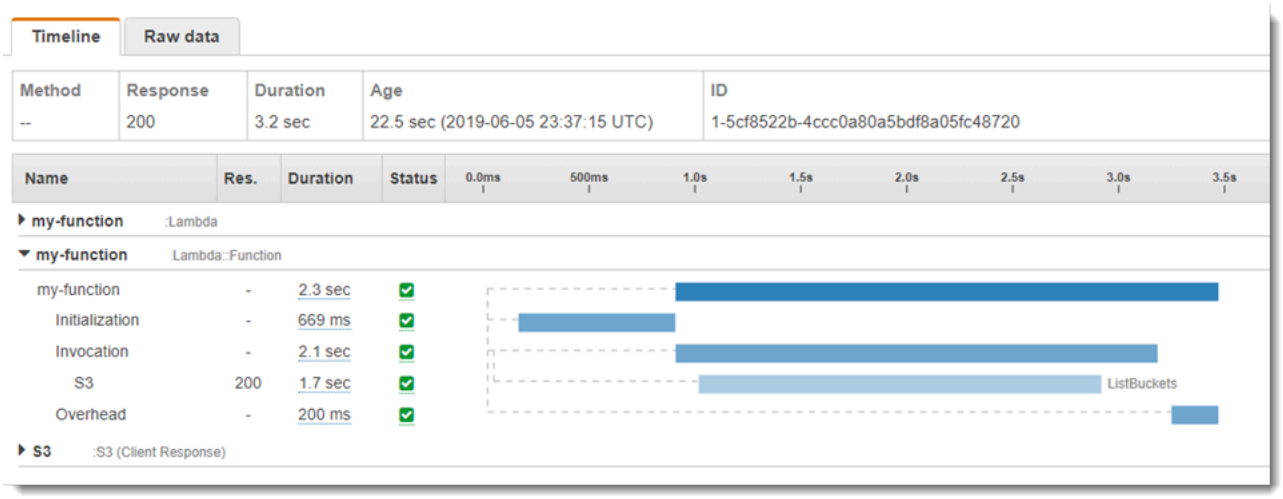
X-Ray 無法追蹤應用程式的所有請求。X-Ray 會套用取樣演算法以確保追蹤的效率，同時仍提供所有請求的代表範本。取樣率為每秒 1 次請求和 5% 的額外請求。

> **Note**
>
> 您無法針對函數設定 X-Ray 取樣率。

在 X-Ray 中，追蹤會記錄一或多個服務所處理之要求的相關資訊。Lambda 會記錄每個追蹤 2 個區段，在服務圖表上建立兩個節點。下圖反白顯示這兩個節點：



左側第一個節點代表接收調用請求的 Lambda 服務。第二個節點代表您特定的 Lambda 函數。下列範例顯示了具有這 2 個區段的追蹤。兩者都被命名為我的函數，但一個具有的起源`AWS::Lambda`，另一個具有的`AWS::Lambda::Function`起源。如果`AWS::Lambda`區段顯示錯誤，表示 Lambda 服務發生問題。如果`AWS::Lambda::Function`區段顯示錯誤，表示您的函數發生問題。



此範例會展開區AWS::Lambda::Function段，以顯示其三個子區段：

- 初始化 - 表示載入函數和執行初始化程式碼所花費的時間。只有函數的每個執行個體所處理的第一個事件會顯示此子區段。
- 調用 – 表示執行處理常式程式碼所花費的時間。
- 額外負荷 - 表示 Lambda 執行時間為做好準備以處理下一個事件所花費的時間。

您可以測試處理常式程式碼，以記錄中繼資料並追蹤下游呼叫。若要記錄處理常式對其他資源和服務所進行之呼叫的詳細資料，請使用適用於 Ruby 的 X-Ray 開發套件。若要取得開發套件，請將 aws-xray-sdk 套件新增至應用程式的相依性。

Example blank-ruby/function/Gemfile

```
# Gemfile
source 'https://rubygems.org'

gem 'aws-xray-sdk', '0.11.4'
gem 'aws-sdk-lambda', '1.39.0'
gem 'test-unit', '3.3.5'
```

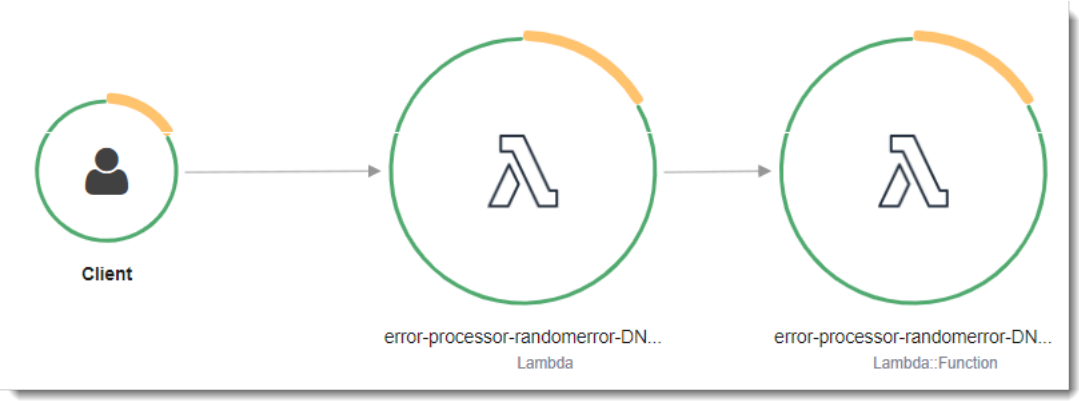
要檢測 AWS SDK 客戶端，請在初始化代碼中創建客戶端後需要該aws-xray-sdk/lambda模塊。

Example 空白紅寶石/函數/lambda 函數 .rb- 跟踪 SDK 客戶端 AWS

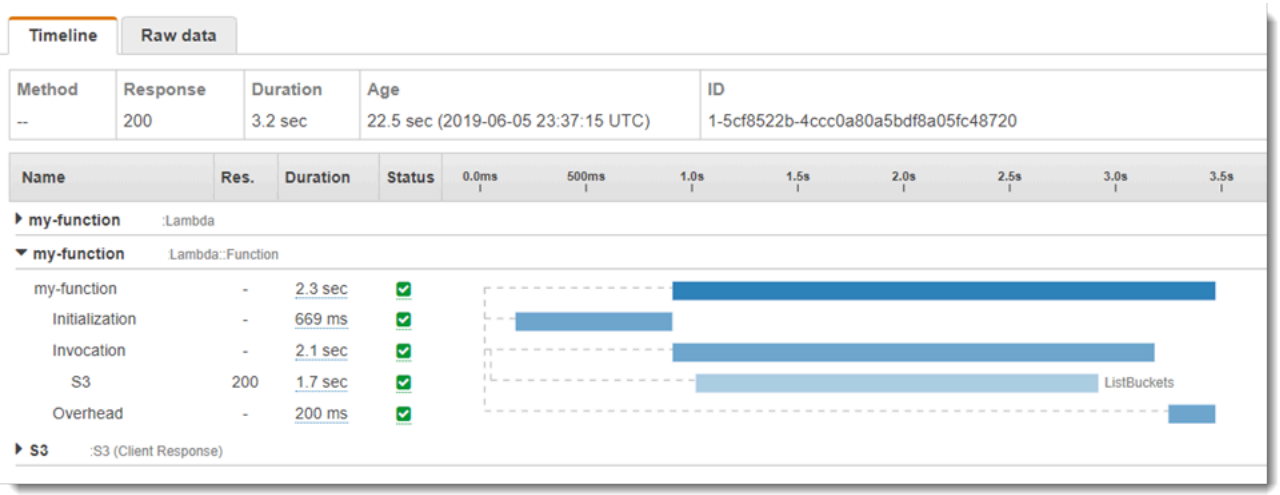
```
# lambda_function.rb
require 'logger'
require 'json'
```

```
require 'aws-sdk-lambda'
$client = Aws::Lambda::Client.new()
$client.get_account_settings()

require 'aws-xray-sdk/lambda'

def lambda_handler(event:, context:)
  logger = Logger.new($stdout)
  ...
```

在 X-Ray 中，追蹤會記錄一或多個服務所處理之要求的相關資訊。Lambda 會記錄每個追蹤 2 個區段，在服務圖表上建立兩個節點。下圖反白顯示這兩個節點：



左側第一個節點代表接收調用請求的 Lambda 服務。第二個節點代表您特定的 Lambda 函數。下列範例顯示了具有這 2 個區段的追蹤。兩者都被命名為我的函數，但一個具有的起源`AWS::Lambda`，另一個具有的`AWS::Lambda::Function`起源。如果`AWS::Lambda`區段顯示錯誤，表示 Lambda 服務發生問題。如果`AWS::Lambda::Function`區段顯示錯誤，表示您的函數發生問題。



此範例會展開區`AWS::Lambda::Function`段，以顯示其三個子區段：

- 初始化 - 表示載入函數和執行初始化程式碼所花費的時間。只有函數的每個執行個體所處理的第一個事件會顯示此子區段。
- 調用 – 表示執行處理常式程式碼所花費的時間。
- 額外負荷 - 表示 Lambda 執行期為做好準備以處理下一個事件所花費的時間。

您也可以檢測 HTTP 用戶端、記錄 SQL 查詢，以及建立具有註釋和中繼資料的自訂子區段。如需詳細資訊，請參閱開 AWS X-Ray 發人員指南中的適用於 Ruby 的 X-Ray SDK。

章節

- 透過 Lambda API 啟用主動追蹤
- 啟用作用中追蹤 AWS CloudFormation
- 將執行時間相依性儲存在圖層中

透過 Lambda API 啟用主動追蹤

若要使用 AWS CLI 或 AWS SDK 管理追蹤組態，請使用下列 API 作業：

- UpdateFunction配置
- GetFunction配置
- CreateFunction

下列範例 AWS CLI 命令可在名為 my- function 的函式上啟用主動追蹤。

```
aws lambda update-function-configuration \
--function-name my-function \
--tracing-config Mode=Active
```

追蹤模式是您發布函數版本時版本特定組態的一部分。您無法變更已發佈版本上的追蹤模式。

啟用作用中追蹤 AWS CloudFormation

若要啟動 AWS CloudFormation 範本中的`AWS::Lambda::Function`資源追蹤，請使用`TracingConfig`屬性。

Example function-inline.yml - 追蹤組態

```
Resources:
```

```
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

對於 AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` 資源，請使用`Tracing`屬性。

Example template.yml - 追蹤組態

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

將執行時間相依性儲存在圖層中

如果您使用 X-Ray SDK 來檢測 AWS SDK 用戶端您的函數程式碼，您的部署套件可能會變得相當大。為了避免每次更新函數程式碼時上傳執行時間相依性，請將 X-Ray SDK 封裝在一個 Lambda 層中。

下面的例子顯示了儲存適用於 Ruby 的 X-Ray 開發套件的 `AWS::Serverless::LayerVersion` 資源。

Example template.yml - 相依性層

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: function/.
      Tracing: Active
      Layers:
        - !Ref libs
      ...
  libs:
    Type: AWS::Serverless::LayerVersion
    Properties:
      LayerName: blank-ruby-lib
```

```
      Description: Dependencies for the blank-ruby sample app.
      ContentUri: lib/.
      CompatibleRuntimes:
        - ruby2.5
```

透過此組態，您只有在變更執行時間相依性時才會更新程式庫層。由於函數部署套件僅含有您的程式碼，因此有助於減少上傳時間。

為相依性建立圖層需要建置變更，才能在部署之前產生圖層封存。如需工作範例，請參閱 blank-ruby 範例應用程式。

使用 Java 建置 Lambda 函數

您可以在 AWS Lambda中執行 Java 程式碼。Lambda 提供用於執行程式碼來處理事件的 Java 執行期。您的程式碼在 Amazon Linux 環境中執行，其中包含來自您管理之 AWS Identity and Access Management (IAM) 角色的 AWS 登入資料。

Lambda 支援以下 Java 執行期。

Java

名稱	識別符	作業系統	取代日期	封鎖函數建立	封鎖函數更新
Java 21	`java21`	Amazon Linux 2023			
Java 17	`java17`	Amazon Linux 2			
Java 11	`java11`	Amazon Linux 2			
Java 8	`java8.al2`	Amazon Linux 2			

Lambda 提供下列適用於 Java 函數的程式庫：

- com.amazonaws:aws-lambda-java-core (必要) - 定義處理常式方法介面，以及執行期傳遞給處理常式的內容物件。如果您定義自己的輸入類型，這是您需要的唯一程式庫。
- com.amazonaws:aws-lambda-java-events - 來自調用 Lambda 函數的服務的輸入事件類型。
- com.amazonaws:aws-lambda-java-log4j2 - 一個 Apache Log4j 2 的附加程式庫，您可以使用它將當前調用的請求 ID 新增至函數日誌。
- AWS 適用於 Java 2.0 的開發套件 — 適用於 Java 程式設計語言的官方 AWS 開發套件。

> **Important**
>
> 請勿使用 JDK API 的私有元件，如私有欄位、方法或類別。非公有 API 元件可能在任何更新中發生變更或被移除，導致您的應用程式中斷。

若要建立 Lambda 函數

1. 開啟 Lambda 主控台。
2. 選擇建立函數。
3. 進行下列設定：
   - 函數名稱：輸入函數名稱。
   - 執行階段：選擇 Java 21。
4. 選擇建立函數。
5. 若要設定測試事件，請選擇 Test (測試)。
6. 事件名稱輸入 **`test`**。
7. 選擇儲存變更。
8. 若要調用函數，請選擇 Test (測試)。

主控台將建立一個 Lambda 函數，它具有名為 `Hello` 的處理常式類別。由於 Java 是一種編譯語言，因此您無法在 Lambda 主控台中檢視或編輯原始碼，但可以修改其組態，加以調用，並設定觸發條件。

> **Note**
>
> 若要在您的本機環境中開始進行應用程式開發，請部署本指南 GitHub 儲存庫中提供的其中一個範例應用程式。

`Hello` 類別有一個名為 `handleRequest` 的函式，其接受事件物件與內容物件。這就是在調用函數時，Lambda 呼叫的處理常式函數。Java 函數執行期會從 Lambda 取得調用事件並其傳遞至處理常式。在函式組態中，處理常式值為 `example.Hello::handleRequest`。

若要更新函數的程式碼，您可建立部署套件，這是包含函數程式碼的 ZIP 封存檔。隨著函式開發的進展，您需要將函式程式碼存放於原始碼控制系統、加入程式庫並進行自動化部署。首先，透過命令列建立部署套件並更新您的程式碼。

除了傳遞調用事件外，函式執行期還會傳遞內容物件至處理常式。內容物件包含了有關調用、函式以及執行環境的額外資訊。更多詳細資訊將另由環境變數提供。

您的 Lambda 函數隨附一個 CloudWatch 日誌記錄群組。函數運行時將有關每次調用的詳細信息發送到 CloudWatch 日誌。它在調用期間會轉送您的函數輸出的任何記錄。如果您的函數傳回錯誤，Lambda 會對該錯誤進行格式化之後傳回給調用端。

主題

- 在 Java 中 Lambda 義函數處理程序
- 使用 .zip 或 JAR 封存檔部署 Java Lambda 函數
- 使用容器映像部署 Java Lambda 函數
- 使用 Java Lambda 函數的圖層
- 使用 Lambda 改善啟動效能 SnapStart
- Java Lambda 函數自訂設定
- AWS Lambda Java 中的上下文對象
- AWS Lambda Java 中的函數日誌記錄
- 在中檢測 Java 程式碼 AWS Lambda
- Java 範例應用程式 AWS Lambda

在 Java 中 Lambda 義函數處理程序

Lambda 函數處理常式是您的函數程式碼中處理事件的方法。當有人呼叫您的函數時，Lambda 會執行處理常式方法。函數會執行，直到處理常式傳回回應、結束或逾時為止。

本指南的 GitHub 存放庫提供了示 easy-to-deploy 範各種處理常式類型的範例應用程式。如需詳細資訊，請參閱本主題的結尾內容。

章節

處理常式範例：Java 17 執行期

在下列 Java 17 範例中，名為 `HandlerIntegerJava17` 的類別會定義名為 `handleRequest` 的處理常式方法。處理常式方法接受以下輸入：

- `IntegerRecord`，這是表示事件資料的自訂 Java 記錄。在此範例中，如下所示定義 `IntegerRecord`：

```
record IntegerRecord(int x, int y, String message) {
}
```

- 內容物件，其提供的方法和各項屬性提供了有關調用、函數以及執行環境的資訊。

假設我們想要撰寫這樣一個函數，其記錄來自輸入 `IntegerRecord` 的 `message`，並傳回 `x` 和 `y` 的總和。以下是函數程式碼：

Example HandlerIntegerJava17.java

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
```

```
import com.amazonaws.services.lambda.runtime.RequestHandler;

// Handler value: example.HandlerInteger
public class HandlerIntegerJava17 implements RequestHandler<IntegerRecord, Integer>{

  @Override
  /*
   * Takes in an InputRecord, which contains two integers and a String.
   * Logs the String, then returns the sum of the two Integers.
   */
  public Integer handleRequest(IntegerRecord event, Context context)
  {
    LambdaLogger logger = context.getLogger();
    logger.log("String found: " + event.message());
    return event.x() + event.y();
  }
}

record IntegerRecord(int x, int y, String message) {
}
```

透過在函數的組態中設定處理常式參數，來指定您希望 Lambda 調用的方法。可以使用下列格式表示處理常式：

- *package*.*Class*::*method* - 完整格式。例如：example.Handler::handleRequest。
- *package*.*Class* - 實作處理常式介面類別的縮寫格式。例如：example.Handler。

當 Lambda 調用處理常式時，Lambda 執行期會接收 JSON 格式字串形式的事件，並將其轉換為物件。在上一個範例中，範例事件可能如下所示：

Example event.json

```
{
  "x": 1,
  "y": 20,
  "message": "Hello World!"
}
```

您可以儲存此檔案，並使用下列 AWS Command Line Interface (CLI) 指令在本機測試您的函數：

```
aws lambda invoke --function-name function_name --payload file://event.json out.json
```

處理常式範例：Java 11 執行期及更低版本

Lambda 支援在 Java 17 和更新執行期內的記錄。在所有 Java 執行期中，可以使用類別來表示事件資料。下列範例接受整數清單和內容物件作為輸入，並傳回清單中所有整數的總和。

Example Handler.java

在下列範例中，名為 Handler 的類別會定義名為 handleRequest 的處理常式方法。處理常式方法會將一個事件和內容物件作為輸入，並傳回一個字串。

Example HandlerList. 爪哇

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;

import java.util.List;

// Handler value: example.HandlerList
public class HandlerList implements RequestHandler<List<Integer>, Integer>{

  @Override
  /*
   * Takes a list of Integers and returns its sum.
   */
  public Integer handleRequest(List<Integer> event, Context context)
  {
    LambdaLogger logger = context.getLogger();
    logger.log("EVENT TYPE: " + event.getClass().toString());
    return event.stream().mapToInt(Integer::intValue).sum();
  }
}
```

如需更多範例，請參閱範例處理常式程式碼。

初始化程式碼

在首次調用函數之前，Lambda 會在初始化階段執行靜態程式碼和類別建構函數。在初始化期間建立的資源會在調用之間保留在記憶體中，並且可供處理常式重複使用數千次。因此，可以在主處理常式方法之外新增初始化程式碼，以便節省運算時間並在多個調用中重複使用資源。

在下列範例中，用戶端初始化程式碼位於主處理常式方法之外。執行期會在函數提供其第一個事件之前初始化用戶端。由於 Lambda 無需再次初始化用戶端，因此後續事件的速度會快得多。

Example Handler.java

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;

import java.util.Map;

import software.amazon.awssdk.services.lambda.LambdaClient;
import software.amazon.awssdk.services.lambda.model.GetAccountSettingsResponse;
import software.amazon.awssdk.services.lambda.model.LambdaException;

// Handler value: example.Handler
public class Handler implements RequestHandler<Map<String,String>, String> {

    private static final LambdaClient lambdaClient = LambdaClient.builder().build();

    @Override
    public String handleRequest(Map<String,String> event, Context context) {

        LambdaLogger logger = context.getLogger();
        logger.log("Handler invoked");

        GetAccountSettingsResponse response = null;
        try {
            response = lambdaClient.getAccountSettings();
        } catch(LambdaException e) {
            logger.log(e.getMessage());
        }
        return response != null ? "Total code size for your account is " +
 response.accountLimit().totalCodeSize() + " bytes" : "Error";
    }
}
```

選擇輸入和輸出類型

您可以在處理常式方法的簽章中指定事件對應的物件類型。在上述範例中，Java 執行階段會將事件還原序列化為實作 `Map<String,String>` 介面的類型。S tring-to-string 地圖適用於平面事件，如下所示：

Example Event.json - 天氣資料

```
{
  "temperatureK": 281,
  "windKmh": -3,
  "humidityPct": 0.55,
  "pressureHPa": 1020
}
```

不過，每個欄位的值必須是字串或數字。如果事件包含具有物件作為值的欄位，執行階段無法將其還原序列化並傳回錯誤。

選擇與函數處理的事件資料搭配使用的輸入類型。您可以使用基本類型或已妥善定義的類型。

輸入類型

- `Integer`、`Long`、`Double` 等等 - 事件是沒有其他格式的數字，例如 `3.5`。執行階段會將值轉換為指定類型的物件。
- `String` - 事件是 JSON 字串，包括引號，例如 `"My string."`。執行階段會將值 (不含引號) 轉換為 `String` 物件。
- *`Type`*、`Map<String,`*`Type`*`>` 等 - 該事件是一個 JSON 物件。執行階段會將它還原序列化為指定類型或介面的物件。
- `List<Integer>`、`List<String>`、`List<Object>` 等等 - 該事件是一個 JSON 陣列。執行階段會將它還原序列化為指定類型或介面的物件。
- `InputStream` - 該事件是任何 JSON 類型。執行階段會將文件的位元組串流傳遞給處理常式而不進行修改。您還原序列化輸入和並將輸出寫入輸出串流。
- 程式庫類型 — 對於由 AWS 服務傳送的事件，請使用 aws-lamb da-java 事件程式庫中的類型。

如果您定義自己的輸入類型，則此物件應該為可還原序列化且可變動的普通舊型 Java 物件 (POJO)，具備適用於事件中每個欄位的預設建構函數和屬性。事件中未對應至屬性的金鑰以及未包含在事件中的屬性都會遭捨棄且不會發生錯誤。

輸出類型可以是物件或 void。執行階段會將傳回的值序列化為文字。如果輸出是具有欄位的物件，執行階段會將其序列化為 JSON 文件。如果它是一個包裝基本值的類型，執行階段會傳回代表該值的文字。

處理程式界面

該 aws-lambda-java-core 程式庫會定義處理常式方法的兩個介面。使用提供的介面來簡化處理常式組態，並在編譯階段驗證處理常式方法簽章。

- 運行時. 服務. RequestHandler
- 運行時. 服務. RequestStream處理器

RequestHandler 介面是一個一般類型，它有兩個參數：輸入類型和輸出類型。兩種類型都必須是物件。當您使用此介面時，Java 執行階段會將事件還原序列化為具有輸入類型的物件，並將輸出序列化為文字。當內建序列化與您的輸入和輸出類型一同作業時，請使用此介面。

Example Handler.java - 處理常式介面

```
// Handler value: example.Handler
public class Handler implements RequestHandler<Map<String,String>, String>{
  @Override
  public String handleRequest(Map<String,String> event, Context context)
```

若要使用您自己的序列化，請實作 RequestStreamHandler 介面。透過此介面，Lambda 會將處理常式傳遞給一個輸入串流和輸出串流。處理常式會從輸入串流讀取位元組，寫入到輸出串流，並傳回 void 值。

下面的例子會使用緩衝的讀取器和寫入器類型以與輸入和輸出串流一起作業。

Example HandlerStream. 爪哇

```
import com.amazonaws.services.lambda.runtime.Context
import com.amazonaws.services.lambda.runtime.LambdaLogger
import com.amazonaws.services.lambda.runtime.RequestStreamHandler
...
// Handler value: example.HandlerStream
public class HandlerStream implements RequestStreamHandler {
  @Override
  /*
```

```
    * Takes an InputStream and an OutputStream. Reads from the InputStream,
    * and copies all characters to the OutputStream.
    */
   public void handleRequest(InputStream inputStream, OutputStream outputStream, Context
  context) throws IOException
   {
     LambdaLogger logger = context.getLogger();
     BufferedReader reader = new BufferedReader(new InputStreamReader(inputStream,
  Charset.forName("US-ASCII")));
     PrintWriter writer = new PrintWriter(new BufferedWriter(new
  OutputStreamWriter(outputStream, Charset.forName("US-ASCII"))));
     int nextChar;
     try {
       while ((nextChar = reader.read()) != -1) {
         outputStream.write(nextChar);
       }
     } catch (IOException e) {
       e.printStackTrace();
     } finally {
       reader.close();
       String finalString = writer.toString();
       logger.log("Final string result: " + finalString);
       writer.close();
     }
   }
}
```

範例處理常式程式碼

本指南的 GitHub 儲存庫包含範例應用程式，示範各種處理常式類型和介面的使用方式。每個範例應用程式都包含可輕鬆部署和清理的指令碼、 AWS SAM 範本和支援資源。

以 Java 編寫的範例 Lambda 應用程式

- java17-examples – 一個 Java 函數，示範如何使用 Java 記錄來表示輸入事件資料物件。
- java-basic - 具有單元測試和變數日誌組態的最小 Java 函數集合。
- java-events - Java 函數集合，其中包含如何處理來自各種服務 (例如 Amazon API Gateway、Amazon SQS 和 Amazon Kinesis) 事件的骨架程式碼。這些函數使用最新版 aws-lambda-java-events 程式庫 (3.0.0 及更新版)。這些範例不需要 AWS SDK 作為相依性。
- s3-java - 一種 Java 函數，它處理來自 Amazon S3 的通知事件，並使用 Java Class Library (JCL) 以從上傳的映像檔案建立縮圖。

- 使用 API Gateway 調用 Lambda 函數 - 一個 Java 函數，其可掃描包含員工資訊的 Amazon DynamoDB 資料表。然後，其會使用 Amazon Simple Notification Service 向員工傳送文字訊息，慶祝他們的工作週年紀念日。此範例使用 API Gateway 調用函數。

java-events和s3-java應用程式會將 AWS 服務事件做為輸入，並傳回字串。java-basic 應用程式包含數種類型的處理常式：

- Handler.java - 將 Map<String,String> 作為輸入。
- HandlerInteger.java-接受一個Integer作為輸入。
- HandlerList.java-接受一個List<Integer>作為輸入。
- HandlerStream.java — 接受InputStream和作OutputStream為輸入。
- HandlerString.java-接受一個String作為輸入。
- HandlerWeatherData.java — 採用自訂類型做為輸入。

若要測試不同的處理常式類型，只要變更 AWS SAM 範本中的處理常式值即可。如需詳細說明，請參閱範例應用程式的讀我檔案。

使用 .zip 或 JAR 封存檔部署 Java Lambda 函數

你的 AWS Lambda 函數的代碼由腳本或編譯的程序及其依賴關係組成。使用部署套件將函數程式碼部署到 Lambda。Lambda 支援兩種類型的部署套件：容器映像和 .zip 封存檔。

本頁說明如何將部署套件建立為 .zip 檔案或 Jar 檔案，然後使用部署套件將函數程式碼部署至 AWS Lambda 使用 AWS Command Line Interface (AWS CLI)。

章節

必要條件

這 AWS CLI 是一種開放原始碼工具，可讓您使用命令列殼層中的命令與 AWS 服務互動。若要完成本節中的步驟，您必須執行下列各項：

- AWS Command Line Interface (AWS CLI) 第二版
- AWS CLI -快速配置 `aws configure`

工具與程式庫

Lambda 提供下列適用於 Java 函數的程式庫：

- com.amazonaws: aws-lambda-java-core (必要) — 定義處理常式方法介面和執行階段傳遞給處理常式的上下文物件。如果您定義自己的輸入類型，這是您需要的唯一程式庫。
- 亞馬遜：aws-lambda-java-events— 來自調用 Lamb da 函數的服務的事件的輸入類型。
- 亞馬遜：aws-lambda-java-log4j2-Apache Log4j 的 2 附加程序庫，您可以使用它來添加當前調用的請求 ID 到您的函數日誌。

- AWS 適用於 Java 2.0 的開發套件 — 適用於 Java 程式設計語言的官方 AWS 開發套件。

這些程式庫可透過 Maven Central Repository 取得。請將它們新增到您的建置定義中，如下所示：

Gradle

```
dependencies {
    implementation 'com.amazonaws:aws-lambda-java-core:1.2.2'
    implementation 'com.amazonaws:aws-lambda-java-events:3.11.1'
    runtimeOnly 'com.amazonaws:aws-lambda-java-log4j2:1.5.1'
}
```

Maven

```
  <dependencies>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-core</artifactId>
      <version>1.2.2</version>
    </dependency>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-events</artifactId>
      <version>3.11.1</version>
    </dependency>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-log4j2</artifactId>
      <version>1.5.1</version>
    </dependency>
  </dependencies>
```

若要建立部署套件，請將函數程式碼和相依性編譯成單一 .zip 檔案或 Java 封存 (JAR) 檔案。針對 Gradle，請使用Zip建置類型。針對 Apache Maven，請使用 Maven Shade 外掛程式。若要上傳您的部署套件，請使用 Lambda 主控台、Lambda API 或 AWS Serverless Application Model (AWS SAM)。

> **Note**
>
> 若要將您部署套件大小保持較小，請以階層方式將您的函式相依項目進行封裝。各層可讓您獨立管理相依項目，並可供多個函數使用，也可和其他帳戶共用。如需詳細資訊，請參閱 Lambda 層。

使用 Gradle 建立部署套件

若要在 Gradle 中建立具有函數程式碼和相依項的部署套件，請使用 `Zip` 建置類型。以下是完整的範本 build.gradle 檔案的範例：

Example build.gradle - 建置任務

```
task buildZip(type: Zip) {
    into('lib') {
        from(jar)
        from(configurations.runtimeClasspath)
    }
}
```

此建置組態會在 `build/distributions` 目錄中產生部署套件。在 `into('lib')` 陳述式中，`jar` 任務會將包含主要類別的 JAR 封存檔組合至名稱為 `lib` 的資料夾中。此外，`configurations.runtimeClassPath` 任務會將相依項程式庫從建置的類別路徑複製到名稱為 `lib` 的資料夾中。

Example build.gradle - 相依性

```
dependencies {
    ...
    implementation 'com.amazonaws:aws-lambda-java-core:1.2.2'
    implementation 'com.amazonaws:aws-lambda-java-events:3.11.1'
    implementation 'org.apache.logging.log4j:log4j-api:2.17.1'
    implementation 'org.apache.logging.log4j:log4j-core:2.17.1'
    runtimeOnly 'org.apache.logging.log4j:log4j-slf4j18-impl:2.17.1'
    runtimeOnly 'com.amazonaws:aws-lambda-java-log4j2:1.5.1'
    ...
}
```

Lambda 會以 Unicode 字母順序載入 JAR 檔案。如果 `lib` 目錄的多個 JAR 檔案包含相同類別，則會使用第一個 JAR。您可以使用以下 shell 指令碼來識別重複的類別：

Example test-zip.sh

```
mkdir -p expanded
unzip path/to/my/function.zip -d expanded
find ./expanded/lib -name '*.jar' | xargs -n1 zipinfo -1 | grep '.*.class' | sort |
 uniq -c | sort
```

為相依項建立 Java 層

> **Note**
>
> 使用具 Java 等編譯語言函數的層，可能不會具有與使用 Python 等轉譯語言一樣多的好處。由於 Java 是編譯語言，因此您的函數仍須在 init 階段將任何共用組件手動載入記憶體中，這可能會增加冷啟動時間。相反地，建議您在編譯時加入任何共用程式碼，利用任何內建的編譯器最佳化功能。

本節中的指示說明如何在層中包含相依項。如需如何在部署套件中包含相依項的指示，請參閱the section called “使用 Gradle 建立部署套件”或the section called “使用 Maven 建立部署套件”。

將層新增至函數時，Lambda 會將層內容載入該執行環境的 `/opt` 目錄。在每一次 Lambda 執行期中，PATH 變數已包含 `/opt` 目錄中的特定資料夾路徑。若要確保PATH變數會擷取圖層內容，您的圖層 .zip 檔案應該在下列資料夾路徑中具有其相依性：

- `java/lib` (CLASSPATH)

例如，您的層 .zip 檔案結構可能如下所示：

```
jackson.zip
# java/lib/jackson-core-2.2.3.jar
```

此外，Lambda 會自動偵測 `/opt/lib` 目錄中的程式庫，以及 `/opt/bin` 目錄中的二進位檔案。若要確保 Lambda 正確找到您的層內容，您也可以建立結構如下的層：

```
custom-layer.zip
# lib
```

```
    | lib_1
    | lib_2
# bin
    | bin_1
    | bin_2
```

封裝層之後，請參閱the section called “建立和刪除層”及the section called “新增層”，完成層設定。

使用 Maven 建立部署套件

要使用 Maven 建置部署套件，請使用 Maven Shade 外掛程式。該外掛程式會建立一個 JAR 檔案，其中包含編譯的函數代碼及其所有相依性。

Example pom.xml - 外掛程式組態

```
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>3.2.2</version>
        <configuration>
          <createDependencyReducedPom>false</createDependencyReducedPom>
        </configuration>
        <executions>
          <execution>
            <phase>package</phase>
            <goals>
              <goal>shade</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
```

若要建置部署套件，請使用 `mvn package` 命令。

```
[INFO] Scanning for projects...
[INFO] -----------------------< com.example:java-maven >-----------------------
[INFO] Building java-maven-function 1.0-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
...
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ java-maven ---
[INFO] Building jar: target/java-maven-1.0-SNAPSHOT.jar
[INFO]
```

```
[INFO] --- maven-shade-plugin:3.2.2:shade (default) @ java-maven ---
[INFO] Including com.amazonaws:aws-lambda-java-core:jar:1.2.2 in the shaded jar.
[INFO] Including com.amazonaws:aws-lambda-java-events:jar:3.11.1 in the shaded jar.
[INFO] Including joda-time:joda-time:jar:2.6 in the shaded jar.
[INFO] Including com.google.code.gson:gson:jar:2.8.6 in the shaded jar.
[INFO] Replacing original artifact with shaded artifact.
[INFO] Replacing target/java-maven-1.0-SNAPSHOT.jar with target/java-maven-1.0-
SNAPSHOT-shaded.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  8.321 s
[INFO] Finished at: 2020-03-03T09:07:19Z
[INFO] ------------------------------------------------------------------------
```

此命令會在 `target` 目錄中產生一個 JAR 檔案。

> **Note**
>
> 如果您正在使用多版本 JAR (MRJAR)，則必須在 `lib` 目錄中包含 MRJAR (即由 Maven Shade 外掛程式產生的陰影 JAR)，並在將部署套件上傳到 Lambda 之前對其進行壓縮。否則，Lambda 可能無法正確解壓縮 JAR 檔案，導致您的 `MANIFEST.MF` 檔案被忽略。

如果您使用附加器程式庫 (`aws-lambda-java-log4j2`)，則還必須配置 Maven Shade 外掛程式的轉換器。轉換器程式庫會合併出現在附加器程式庫和 Log4j 中快取檔案的版本。

Example pom.xml - 具備 Log4j 2 附加器的外掛程式組態

```
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>3.2.2</version>
        <configuration>
          <createDependencyReducedPom>false</createDependencyReducedPom>
        </configuration>
        <executions>
          <execution>
            <phase>package</phase>
            <goals>
              <goal>shade</goal>
            </goals>
```

```
            <configuration>
              <transformers>
                <transformer
implementation="com.github.edwgiz.maven_shade_plugin.log4j2_cache_transformer.PluginsCacheFile
                </transformer>
              </transformers>
            </configuration>
          </execution>
        </executions>
        <dependencies>
          <dependency>
            <groupId>com.github.edwgiz</groupId>
            <artifactId>maven-shade-plugin.log4j2-cachefile-transformer</artifactId>
            <version>2.13.0</version>
          </dependency>
        </dependencies>
      </plugin>
```

使用 Lambda 主控台上傳部署套件

若要建立新函數，您必須先在主控台中建立函數，然後上傳您的 .zip 或 JAR 檔案。若要更新現有函數，請開啟函數的頁面，然後按照同樣的程序新增更新後的 .zip 或 JAR 檔案。

如果您的部署套件檔案小於 50 MB，您可以透過直接從本機電腦上傳檔案來建立或更新函數。若 .zip 或 JAR 檔案大於 50 MB，您必須先將套件上傳至 Amazon S3 儲存貯體。如需有關如何使用將檔案上傳到 Amazon S3 儲存貯體的指示 AWS Management Console，請參閱開始使用 Amazon S3。若要使用上載檔案 AWS CLI，請參閱《使用指南》中的AWS CLI 〈移動物件〉。

> **Note**
>
> 您無法變更現有函數的部署套件類型 (.zip 或容器映像檔)。例如，您無法將容器映像函數轉換為使用 .zip 檔案封存。您必須建立新的函數。

若要建立新的函數 (主控台)

1. 開啟 Lambda 主控台的函數頁面，然後選擇建立函數。
2. 選擇 Author from scratch (從頭開始撰寫)。
3. 在基本資訊下，請執行下列動作：

    a. 在函數名稱中輸入函數名稱。

b. 在執行期中選取要使用的執行期。

c. (選用) 在架構中選擇要用於函數的指令集架構。預設架構值為 x86_64。請確定函數的 .zip 部署套件與您選取的指令集架構相容。

4. (選用) 在 許可 下，展開 變更預設執行角色 。您可建立新的執行角色，或使用現有的角色。

5. 選擇建立函數。Lambda 會使用您選擇的執行期建立一個基本的「Hello world」函數。

若要從本機電腦上傳 .zip 或 JAR 封存檔 (主控台)

1. 在 Lambda 主控台的函數頁面中選擇要上傳 .zip 或 JAR 檔案的函數。

2. 選取程式碼索引標籤。

3. 在程式碼來源窗格中選擇上傳來源。

4. 選擇 .zip 或 .jar 檔案。

5. 若要上傳 .zip 或 JAR 檔案，請執行下列操作：

a. 選取上傳，然後在檔案選擇器中選取您的 .zip 或 JAR 檔案。

b. 選擇 Open (開啟)。

c. 選擇儲存。

若要從 Amazon S3 儲存貯體上傳 .zip 或 JAR 封存檔 (控制台)

1. 在 Lambda 主控台的函數頁面中選擇要上傳新 .zip 或 JAR 檔案的函數。

2. 選取程式碼索引標籤。

3. 在程式碼來源窗格中選擇上傳來源。

4. 選擇 Amazon S3 位置。

5. 貼上 .zip 檔案的 Amazon S3 連結 URL，然後選擇儲存。

上傳部署套件 AWS CLI

您可以使用 AWS CLI 建立新函數，或使用 .zip 或 JAR 檔案更新現有函數。使用建立函數和update-function-code命令來部署您的 .zip 或 JAR 套件。如果您的檔案小於 50 MB，則可以從本機建置電腦的檔案位置上傳套件。若檔案較大，則必須先從 Amazon S3 儲存貯體上傳 .zip 或 JAR 套件。如需有關如何使用將檔案上傳到 Amazon S3 儲存貯體的指示 AWS CLI，請參閱使用AWS CLI 者指南中的移動物件。

> **Note**
>
> 如果您使用從 Amazon S3 儲存貯體上傳 .zip 或 JAR 檔案 AWS CLI，則該儲存貯體必須與您的函數位於 AWS 區域 相同的位置。

若要使用 .zip 或 JAR 檔案與建立新函數 AWS CLI，您必須指定下列項目：

- 函數名稱 (`--function-name`)
- 函數的執行期 (`--runtime`)
- 函數執行角色的 Amazon Resource Name (ARN) (`--role`)
- 函數程式碼中處理常式方法的名稱 (`--handler`)

您也必須指定 .zip 或 JAR 檔案的位置。如果您的 .zip 或 JAR 檔案位於本機建置電腦上的資料夾中，請使用 `--zip-file` 選項來指定檔案路徑，如下列範例命令所示。

```
aws lambda create-function --function-name myFunction \
--runtime java21 --handler example.handler \
--role arn:aws:iam::123456789012:role/service-role/my-lambda-role \
--zip-file fileb://myFunction.zip
```

若要在 Amazon S3 儲存貯體中指定 .zip 檔案的位置，請使用如下列範例命令所示的 `--code` 選項。您只需針對版本控制的物件使用 `S3ObjectVersion` 參數。

```
aws lambda create-function --function-name myFunction \
--runtime java21 --handler example.handler \
--role arn:aws:iam::123456789012:role/service-role/my-lambda-role \
--code S3Bucket=DOC-EXAMPLE-BUCKET,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion
```

若要使用 CLI 更新現有函數，您可以使用 `--function-name` 參數指定函數的名稱。您也必須指定要用來更新函數程式碼的 .zip 檔案的位置。如果您的 .zip 檔案位於本機建置電腦上的資料夾中，請使用 `--zip-file` 選項來指定檔案路徑，如下列範例命令所示。

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

若要在 Amazon S3 儲存貯體中指定 .zip 檔案的位置，請使用如下列範例命令所示的 `--s3-bucket` 和 `--s3-key` 選項。您只需針對版本控制的物件使用 `--s3-object-version` 參數。

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket DOC-EXAMPLE-BUCKET --s3-key myFileName.zip --s3-object-version myObject
 Version
```

上傳部署套件 AWS SAM

您可以使 AWS SAM 用自動化函數程式碼、組態和相依性的部署。 AWS SAM 是的延伸 AWS CloudFormation ，可提供簡化語法來定義無伺服器應用程式。下列範例範本會透過在 Gradle 使用的 `build/distributions` 目錄中的部署套件定義函數：

Example template.yml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Description: An AWS Lambda application that calls the Lambda API.
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: build/distributions/java-basic.zip
      Handler: example.Handler
      Runtime: java21
      Description: Java function
      MemorySize: 512
      Timeout: 10
      # Function's execution role
      Policies:
        - AWSLambdaBasicExecutionRole
        - AWSLambda_ReadOnlyAccess
        - AWSXrayWriteOnlyAccess
        - AWSLambdaVPCAccessExecutionRole
      Tracing: Active
```

若要建立函數，請使用 `package` 和 `deploy` 指令。這些命令是對 AWS CLI的自訂命令。它們會包裝其他命令，將部署套件上傳至 Amazon S3，使用物件 URI 重寫範本，並更新函數的程式碼。

下面的範例指令碼會執行 Gradle 建置並上傳到它建立的部署套件。它在您第一次運行它時創建一個 AWS CloudFormation 堆棧。如果堆疊已存在，則指令碼會加以更新。

Example deploy.sh

```
#!/bin/bash
```

```
set -eo pipefail
aws cloudformation package --template-file template.yml --s3-bucket MY_BUCKET --output-
template-file out.yml
aws cloudformation deploy --template-file out.yml --stack-name java-basic --
capabilities CAPABILITY_NAMED_IAM
```

如需完整的工作範例，請參閱下列範例應用程式：

以 Java 編寫的範例 Lambda 應用程式

- java17-examples – 一個 Java 函數，示範如何使用 Java 記錄來表示輸入事件資料物件。
- java-basic - 具有單元測試和變數日誌組態的最小 Java 函數集合。
- java-events - Java 函數集合，其中包含如何處理來自各種服務 (例如 Amazon API Gateway、Amazon SQS 和 Amazon Kinesis) 事件的骨架程式碼。這些函數使用最新版本的aws-lambda-java-events庫（3.0.0 及更新版本）。這些範例不需要 AWS SDK 作為相依性。
- s3-java - 一種 Java 函數，它處理來自 Amazon S3 的通知事件，並使用 Java Class Library (JCL) 以從上傳的映像檔案建立縮圖。
- 使用 API Gateway 調用 Lambda 函數 - 一個 Java 函數，其可掃描包含員工資訊的 Amazon DynamoDB 資料表。然後，其會使用 Amazon Simple Notification Service 向員工傳送文字訊息，慶祝他們的工作週年紀念日。此範例使用 API Gateway 叫用函數。

使用容器映像部署 Java Lambda 函數

您可以透過三種方式為 Java Lambda 函數建置容器映像：

- 使用 Java 的 AWS 基本映像檔

  AWS 基礎映像會預先載入語言執行期、用來管理 Lambda 與函數程式碼之間互動的執行期界面用戶端，以及用於本機測試的執行期界面模擬器。
- 使用 AWS 僅限作業系統的基本影像

  AWS 僅限作業系統的基本映像檔包含 Amazon Linux 散發和執行階段介面模擬器。這些映像常用於為編譯語言 (如 Go 和 Rust) 和 Lambda 不提供基礎映像的語言或語言版本 (如 Node.js 19) 建置容器映像。您還可以使用僅限作業系統的基礎映像來實作自訂執行期。若要使映像與 Lambda 相容，您必須在映像中加入適用於 Java 的執行期介面用戶端。
- 使用非AWS 基本圖像

  您可以使用其他容器登錄檔中的替代基礎映像 (例如 Alpine Linux 或 Debian)。您也可以使用組織建立的自訂映像。若要使映像與 Lambda 相容，您必須在映像中加入適用於 Java 的執行期介面用戶端。

> **Tip**
>
> 若要縮短 Lambda 容器函數變成作用中狀態所需的時間，請參閱 Docker 文件中的使用多階段建置。若要建置有效率的容器映像，請遵循撰寫 Dockerfiles 的最佳實務。

本頁面會說明如何為 Lambda 建置、測試和部署容器映像。

主題

AWS 用於 Java 的基本映像檔

AWS 提供 Java 的下列基本影像：

標籤	執行期	作業系統	Dockerfile	棄用
21	Java 21	Amazon Linux 2023	碼頭文件上的 Java 21 GitHub	
17	Java 17	Amazon Linux 2	碼頭文件上的 Java 17 GitHub	
11	Java 11	Amazon Linux 2	碼頭文件上的 Java 11 GitHub	
8.al2	Java 8	Amazon Linux 2	用於 Java 8 的碼頭文件 GitHub	

Amazon ECR 儲存庫：gallery.ecr.aws/lambda/java

Java 21 及更高版本的基本映像基於 Amazon Linux 2023 最小容器映像。早期的基本映像使用 Amazon Linux 2。與 Amazon Linux 2 相比，AL2023 具有多項優點，包括更小的部署足跡和更新版本的程式庫，如 `glibc`。

基於 Al2023 的圖像使用`microdnf`（符號鏈接為`dnf`）作為軟件包管理器而不是`yum`，這是 Amazon Linux 2 中的默認軟件包管理器。 `microdnf`是的獨立實作`dnf`。如需 AL2023 映像檔中包含的套件清單，請參閱比較 Amazon Linux 2023 容器映像上安裝的套件中的最小容器欄。如需有關 AL2023 和 Amazon Linux 2 之間差異的詳細資訊，請參閱 AWS 運算部落格 AWS Lambda上的介紹 Amazon Linux 2023 執行階段。

> **Note**
>
> 要在本地運行基於 AL2023 的映像，包括使用 AWS Serverless Application Model （ AWS SAM ），您必須使用碼頭版本 20.10.10 或更高版本。

使用 Java 的 AWS 基本映像檔

必要條件

若要完成本節中的步驟，您必須執行下列各項：

- Java (例如，Amazon Corretto)

- 碼頭工具（Java 21 及更高版本的基本映像的最低版本 20.10.10）
- Apache Maven 或 Gradle
- AWS Command Line Interface (AWS CLI) 第二版

從基礎映像建立映像

Maven

1. 執行下列命令，使用 Lambda 的原型建立 Maven 專案。下列是必要參數：

   - 服務 — 要在 Lambda 函數中使用的用 AWS 服務 戶端。如需可用來源的清單，請參閱上的. GitHub
   - 區域 — 您想要建立 Lambda 函數的 AWS 區域 位置。
   - groupId – 應用程式的完整套件命名空間。
   - artifactId – 您的專案名稱。這會成為您專案的目錄名稱。

   在 Linux 和 macOS 中，執行此命令：

   ```
   mvn -B archetype:generate \
      -DarchetypeGroupId=software.amazon.awssdk \
      -DarchetypeArtifactId=archetype-lambda -Dservice=s3 -Dregion=US_WEST_2 \
      -DgroupId=com.example.myapp \
      -DartifactId=myapp
   ```

   在中 PowerShell，執行下列命令：

   ```
   mvn -B archetype:generate `
      "-DarchetypeGroupId=software.amazon.awssdk" `
      "-DarchetypeArtifactId=archetype-lambda" "-Dservice=s3" "-Dregion=US_WEST_2"
    `
      "-DgroupId=com.example.myapp" `
      "-DartifactId=myapp"
   ```

   Lambda 的 Maven 原型已預先設定為使用 Java SE 8 編譯，並包含對 AWS SDK for Java的相依性。如果您使用不同的原型或使用其他方法來建立專案，則必須為 Maven 設定 Java 編譯器並將 SDK 宣告為相依項。

2. 開啟 *myapp*/src/main/java/com/example/*myapp* 目錄並找到 App.java 檔案。這是 Lambda 函數的程式碼。可以使用提供的範本程式碼進行測試，也可以將其替換為您自己的程式碼。
3. 返回專案的根目錄，然後使用以下組態建立新的 Dockerfile：

   - 將 FROM 屬性設定為基礎映像的 URI。
   - 將 CMD 引數設定為 Lambda 函數處理常式。

   Example Dockerfile

   ```
   FROM public.ecr.aws/lambda/java:21

   # Copy function code and runtime dependencies from Maven layout
   COPY target/classes ${LAMBDA_TASK_ROOT}
   COPY target/dependency/* ${LAMBDA_TASK_ROOT}/lib/

   # Set the CMD to your handler (could also be done as a parameter override
    outside of the Dockerfile)
   CMD [ "com.example.myapp.App::handleRequest" ]
   ```

4. 編譯專案並收集執行期相依性。

   ```
   mvn compile dependency:copy-dependencies -DincludeScope=runtime
   ```

5. 使用 docker build 命令建立 Docker 映像檔。以下範例將映像命名為 docker-image 並為其提供 test 標籤。

   ```
   docker build --platform linux/amd64 -t docker-image:test .
   ```

   > **Note**
   >
   > 此命令會指定 --platform linux/amd64 選項，確保無論建置機器的架構為何，您的容器都與 Lambda 執行環境相容。如果您打算使用 ARM64 指令集架構建立 Lambda 函數，務必將命令變更為改用 --platform linux/arm64 選項。

Gradle

1. 建立專案的目錄，然後切換至該目錄。

```
mkdir example
cd example
```

2. 執行以下命令，讓 Gradle 在環境的 `example` 目錄中產生新的 Java 應用程式專案。對於選取建置指令碼 DSL，選擇 2：Groovy。

```
gradle init --type java-application
```

3. 開啟 */example*/app/src/main/java/*example* 目錄並找到 `App.java` 檔案。這是 Lambda 函數的程式碼。可以使用以下範本程式碼進行測試，也可以將其替換為您自己的程式碼。

   Example App.java

```
package com.example;
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
public class App implements RequestHandler<Object, String> {
    public String handleRequest(Object input, Context context) {
        return "Hello world!";
    }
}
```

4. 開啟 `build.gradle` 檔案。如果使用上一個步驟的範本函數程式碼，請將 `build.gradle` 的內容替換為下列值。如果使用自己的函數程式碼，則請根據需要修改 `build.gradle` 檔案。

   Example build.gradle (Groovy DSL)

```
plugins {
  id 'java'
}
group 'com.example'
version '1.0-SNAPSHOT'
sourceCompatibility = 1.8
repositories {
  mavenCentral()
}
dependencies {
  implementation 'com.amazonaws:aws-lambda-java-core:1.2.1'
}
```

```
jar {
  manifest {
      attributes 'Main-Class': 'com.example.App'
  }
}
```

5. 步驟 2 中的 `gradle init` 命令也在 `app/test` 目錄中產生了一個虛擬測試案例。為實現本教學課程的目的，請刪除 `/test` 目錄，以略過正在執行的測試。
6. 建置專案。

```
gradle build
```

7. 在專案的根目錄 (`/example`) 中，使用以下組態建立一個 Dockerfile：
   - 將 `FROM` 屬性設定為基礎映像的 URI。
   - 使用 `COPY` 指令`{LAMBDA_TASK_ROOT}`，將函數程式碼和執行階段相依性複製到 Lambda 定義的環境變數。
   - 將 `CMD` 引數設定為 Lambda 函數處理常式。

   Example Dockerfile

```
FROM public.ecr.aws/lambda/java:21

# Copy function code and runtime dependencies from Gradle layout
COPY app/build/classes/java/main ${LAMBDA_TASK_ROOT}

# Set the CMD to your handler (could also be done as a parameter override
 outside of the Dockerfile)
CMD [ "com.example.App::handleRequest" ]
```

8. 使用 docker build 命令建立 Docker 映像檔。以下範例將映像命名為 `docker-image` 並為其提供 `test` 標籤。

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> 此命令會指定 `--platform linux/amd64` 選項，確保無論建置機器的架構為何，您的容器都與 Lambda 執行環境相容。如果您打算使用 ARM64 指令集架構建立 Lambda 函數，務必將命令變更為改用 `--platform linux/arm64` 選項。

(選用) 在本機測試映像

1. 使用 docker run 命令啟動 Docker 影像。在此範例中，`docker-image` 為映像名稱，`test` 為標籤。

   ```
   docker run --platform linux/amd64 -p 9000:8080 docker-image:test
   ```

   此命令將映像作為容器執行，並在 `localhost:9000/2015-03-31/functions/function/invocations` 建立本機端點。

   > **Note**
   >
   > 如果您為 ARM64 指令集架構建立 Docker 映像檔，請務必將`--platform linux/arm64`選項改用選項。`--platform linux/amd64`

2. 從新的終端機視窗，將事件張貼至本機端點。

   Linux/macOS

   在 Linux 或 macOS 中，執行下列 `curl` 命令：

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
    '{"payload":"hello world!"}'
   ```

PowerShell

在中 PowerShell，執行下列Invoke-WebRequest命令：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

3. 取得容器 ID。

```
docker ps
```

4. 使用 docker kill 命令停止容器。在此命令中，將 3766c4ab331c 替換為上一步驟中的容器 ID。

```
docker kill 3766c4ab331c
```

部署映像

若要將映像上傳至 Amazon ECR 並建立 Lambda 函數

1. 使用 get-login-password 命令，向 Amazon ECR 登錄檔驗證 Docker CLI。
   - 將--region值設定為您 AWS 區域 要建立 Amazon ECR 儲存庫的位置。
   - 111122223333用您的 AWS 帳戶 身份證替換。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 使用 create-repository 命令在 Amazon ECR 中建立儲存庫。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 儲存庫必須與 Lambda 函數位於 AWS 區域 相同。

如果成功，您將會看到以下回應：

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. 從上一步驟的輸出中複製 `repositoryUri`。
4. 執行 docker tag 命令，將 Amazon ECR 儲存庫中的本機映像標記為最新版本。在此命令中：
   - 將 `docker-image:test` 替換為 Docker 映像檔的名稱和標籤。
   - 將 <ECRrepositoryUri> 替換為複製的 `repositoryUri`。確保在 URI 的末尾包含 `:latest`。

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

範例：

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. 執行 docker push 命令，將本機映像部署至 Amazon ECR 儲存庫。確保在儲存庫 URI 的末尾包含 `:latest`。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 建立函數的執行角色 (若您還沒有的話)。在下一個步驟中您需要角色的 Amazon Resource Name (ARN)。
7. 建立 Lambda 函數。對於 `ImageUri`，從之前的設定中指定儲存庫 URI。確保在 URI 的末尾包含 `:latest`。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> 只要映像與 Lambda 函數位於相同的區域，您就可以使用不同 AWS 帳戶中的映像檔建立函數。如需詳細資訊，請參閱 Amazon ECR 跨帳戶許可。

8. 調用函數。

```
aws lambda invoke --function-name hello-world response.json
```

您應該看到如下回應：

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 若要查看函數的輸出，請檢查 `response.json` 檔案。

若要更新函數程式碼，您必須再次建置映像、將新映像上傳到 Amazon ECR 存放庫，然後使用 update-function-code 命令將映像部署到 Lambda 函數。

Lambda 將圖像標記解析為特定的圖像摘要。這表示如果您將用於部署函數的映像標籤指向 Amazon ECR 中的新映像，Lambda 不會自動更新函數以使用新映像。若要將新映像部署到相同的 Lambda 函數，您必須使用`update-function-code`命令，即使 Amazon ECR 中的影像標籤保持不變。

透過執行期介面用戶端使用替代基礎映像

如果您使用僅限作業系統的基礎映像或替代的基礎映像，則必須在映像中加入執行期介面用戶端。執行期介面用戶端會讓您擴充 Lambda 執行階段 API，管理 Lambda 與函數程式碼之間的互動。

在 Dockerfile 中安裝適用於 Java 的執行期介面用戶端，或作為專案中的相依項。例如，若要使用 Maven 套件管理員安裝執行期界面用戶端，請將以下內容新增至您的 `pom.xml` 檔案中：

```
<dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-lambda-java-runtime-interface-client</artifactId>
    <version>2.3.2</version>
</dependency>
```

如需套件詳細資訊，請參閱 Maven Central Repository 中的 AWS Lambda Java 執行期介面用戶端。您也可以檢閱 AWS Lambda Java Support 程式 GitHub 庫儲存庫中的執行階段介面用戶端原始程式碼。

以下範例示範如何使用 Amazon Corretto 映像為 Java 建置容器映像。Amazon Corretto 是 Open Java Development Kit (OpenJDK) 的免費、多平台的生產就緒分佈。Maven 專案包括執行期介面用戶端作為相依項。

必要條件

若要完成本節中的步驟，您必須執行下列各項：

- Java (例如，Amazon Corretto)
- Docker
- Apache Maven
- AWS Command Line Interface (AWS CLI) 第二版

使用替代基礎映像建立映像

1. 建立 Maven 專案。下列是必要參數：

   - groupId – 應用程式的完整套件命名空間。
   - artifactId – 您的專案名稱。這會成為您專案的目錄名稱。

   Linux/macOS

   ```
   mvn -B archetype:generate \
      -DarchetypeArtifactId=maven-archetype-quickstart \
      -DgroupId=example \
      -DartifactId=myapp \
      -DinteractiveMode=false
   ```

   PowerShell

   ```
   mvn -B archetype:generate `
      -DarchetypeArtifactId=maven-archetype-quickstart `
      -DgroupId=example `
      -DartifactId=myapp `
      -DinteractiveMode=false
   ```

2. 開啟專案目錄。

   ```
   cd myapp
   ```

3. 開啟 pom.xml 檔案並將內容替換如下：此檔案包括 aws-lambda-java-runtime-interface-client 作為相依項。或者，您可以在 Dockerfile 中安裝執行期界面用戶端。不過，最簡單的方法是將程式庫包含為相依項。

   ```
   <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
   www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
   maven-v4_0_0.xsd">
     <modelVersion>4.0.0</modelVersion>
     <groupId>example</groupId>
     <artifactId>hello-lambda</artifactId>
     <packaging>jar</packaging>
     <version>1.0-SNAPSHOT</version>
     <name>hello-lambda</name>
   ```

```
  <url>http://maven.apache.org</url>
  <properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>
  <dependencies>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-runtime-interface-client</artifactId>
      <version>2.3.2</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-dependency-plugin</artifactId>
        <version>3.1.2</version>
        <executions>
          <execution>
            <id>copy-dependencies</id>
            <phase>package</phase>
            <goals>
              <goal>copy-dependencies</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

4. 開啟 *myapp*/src/main/java/com/example/*myapp* 目錄並找到 App.java 檔案。這是 Lambda 函數的程式碼。將程式碼取代為以下內容。

   Example 函數處理常式

```
package example;

public class App {
    public static String sayHello() {
        return "Hello world!";
    }
```

```
}
```

5. 步驟 1 中的 `mvn -B archetype:generate` 命令也在 `src/test` 目錄中產生了一個虛擬測試案例。為實現本教學課程的目的，請將整個產生的 `/test` 目錄刪除，以略過執行測試程序。
6. 返回專案的根目錄，然後建立新的 Dockerfile。下列 Dockerfile 範例使用 Amazon Corretto 映像。Amazon Corretto 是 OpenJDK 的免費、多平台的生產就緒分佈。
   - 將 `FROM` 屬性設定為基礎映像的 URI。
   - 將 `ENTRYPOINT` 設為您希望 Docker 容器在啟動時執行的模組。在此案例中，模組是執行期界面用戶端。
   - 將 `CMD` 引數設定為 Lambda 函數處理常式。

   Example Dockerfile

```
FROM public.ecr.aws/amazoncorretto/amazoncorretto:21 as base

# Configure the build environment
FROM base as build
RUN yum install -y maven
WORKDIR /src

# Cache and copy dependencies
ADD pom.xml .
RUN mvn dependency:go-offline dependency:copy-dependencies

# Compile the function
ADD . .
RUN mvn package

# Copy the function artifact and dependencies onto a clean base
FROM base
WORKDIR /function

COPY --from=build /src/target/dependency/*.jar ./
COPY --from=build /src/target/*.jar ./

# Set runtime interface client as default command for the container runtime
ENTRYPOINT [ "/usr/bin/java", "-cp", "./*",
 "com.amazonaws.services.lambda.runtime.api.client.AWSLambda" ]
# Pass the name of the function handler as an argument to the runtime
```

```
CMD [ "example.App::sayHello" ]
```

7. 使用 docker build 命令建立 Docker 映像檔。以下範例將映像命名為 docker-image 並為其提供 test 標籤。

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> 此命令會指定 --platform linux/amd64 選項，確保無論建置機器的架構為何，您的容器都與 Lambda 執行環境相容。如果您打算使用 ARM64 指令集架構建立 Lambda 函數，務必將命令變更為改用 --platform linux/arm64 選項。

(選用) 在本機測試映像

使用 執行期界面模擬器 以在本機測試映像。您可以將模擬器構建到映像中，也可以使用以下步驟將其安裝在本地計算機上。

若要在本機電腦上安裝並執行執行期介面模擬器

1. 從您的項目目錄中運行以下命令以下載運行時接口仿真器（x86-64 架構） GitHub 並將其安裝在本地計算機上。

   Linux/macOS

```
mkdir -p ~/.aws-lambda-rie && \
    curl -Lo ~/.aws-lambda-rie/aws-lambda-rie https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie && \
    chmod +x ~/.aws-lambda-rie/aws-lambda-rie
```

   要安裝 arm64 模擬器，請使用以下命令替換上一個命令中的 GitHub 存儲庫 URL：

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
```

   PowerShell

```
$dirPath = "$HOME\.aws-lambda-rie"
if (-not (Test-Path $dirPath)) {
```

```
    New-Item -Path $dirPath -ItemType Directory
}

$downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/
releases/latest/download/aws-lambda-rie"
$destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
```

若要安裝 arm64 模擬器，請將 $downloadLink 更換為下列項目：

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

2. 使用 docker run 命令啟動 Docker 影像。注意下列事項：

   - docker-image 是映像名稱，而 test 是標籤。
   - /usr/bin/java -cp './*' com.amazonaws.services.lambda.runtime.api.client.AWSLambda example.App::sayHello 是 Dockerfile 中的 ENTRYPOINT，後面接著 CMD。

   Linux/macOS

   ```
   docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p
    9000:8080 \
       --entrypoint /aws-lambda/aws-lambda-rie \
       docker-image:test \
           /usr/bin/java -cp './*'
    com.amazonaws.services.lambda.runtime.api.client.AWSLambda
    example.App::sayHello
   ```

   PowerShell

   ```
   docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p
    9000:8080 `
   --entrypoint /aws-lambda/aws-lambda-rie `
   docker-image:test `
       /usr/bin/java -cp './*'
    com.amazonaws.services.lambda.runtime.api.client.AWSLambda
    example.App::sayHello
   ```

此命令將映像作為容器執行，並在 `localhost:9000/2015-03-31/functions/function/invocations` 建立本機端點。

> **Note**
>
> 如果您為 ARM64 指令集架構建立 Docker 映像檔，請務必將`--platform linux/arm64`選項改用選項。`--platform linux/amd64`

3. 將事件張貼至本機端點。

   Linux/macOS

   在 Linux 或 macOS 中，執行下列 `curl` 命令：

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
    '{"payload":"hello world!"}'
   ```

   PowerShell

   在中 PowerShell，執行下列`Invoke-WebRequest`命令：

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{}' -ContentType "application/json"
   ```

   此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
    "application/json"
   ```

4. 取得容器 ID。

```
docker ps
```

5. 使用 docker kill 命令停止容器。在此命令中，將 3766c4ab331c 替換為上一步驟中的容器 ID。

```
docker kill 3766c4ab331c
```

部署映像

若要將映像上傳至 Amazon ECR 並建立 Lambda 函數

1. 使用 get-login-password 命令，向 Amazon ECR 登錄檔驗證 Docker CLI。
   - 將--region值設定為您 AWS 區域 要建立 Amazon ECR 儲存庫的位置。
   - 111122223333用您的 AWS 帳戶 身份證替換。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 使用 create-repository 命令在 Amazon ECR 中建立儲存庫。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 儲存庫必須與 Lambda 函數位於 AWS 區域 相同。

如果成功，您將會看到以下回應：

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
```

```
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. 從上一步驟的輸出中複製 `repositoryUri`。
4. 執行 docker tag 命令，將 Amazon ECR 儲存庫中的本機映像標記為最新版本。在此命令中：
   - 將 `docker-image:test` 替換為 Docker 映像檔的名稱和標籤。
   - 將 `<ECRrepositoryUri>` 替換為複製的 `repositoryUri`。確保在 URI 的末尾包含 `:latest`。

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

範例：

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world:latest
```

5. 執行 docker push 命令，將本機映像部署至 Amazon ECR 儲存庫。確保在儲存庫 URI 的末尾包含 `:latest`。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 建立函數的執行角色 (若您還沒有的話)。在下一個步驟中您需要角色的 Amazon Resource Name (ARN)。
7. 建立 Lambda 函數。對於 `ImageUri`，從之前的設定中指定儲存庫 URI。確保在 URI 的末尾包含 `:latest`。

```
aws lambda create-function \
  --function-name hello-world \
```

```
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> 只要映像與 Lambda 函數位於相同的區域，您就可以使用不同 AWS 帳戶中的映像檔建立函數。如需詳細資訊，請參閱 Amazon ECR 跨帳戶許可。

8. 調用函數。

```
aws lambda invoke --function-name hello-world response.json
```

您應該看到如下回應：

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 若要查看函數的輸出，請檢查 `response.json` 檔案。

若要更新函數程式碼，您必須再次建置映像、將新映像上傳到 Amazon ECR 存放庫，然後使用 update-function-code 命令將映像部署到 Lambda 函數。

Lambda 將圖像標記解析為特定的圖像摘要。這表示如果您將用於部署函數的映像標籤指向 Amazon ECR 中的新映像，Lambda 不會自動更新函數以使用新映像。若要將新映像部署到相同的 Lambda 函數，您必須使用`update-function-code`命令，即使 Amazon ECR 中的影像標籤保持不變。

使用 Java Lambda 函數的圖層

Lambda 層是包含補充代碼或數據的 .zip 文件歸檔。層通常具備程式庫相依性、自訂執行期或組態檔案。建立圖層包含三個一般步驟：

1. Package 圖層內容。這意味著創建一個包含要在函數中使用的依賴關係的 .zip 文件歸檔。
2. 在 Lambda 中建立圖層。
3. 將圖層添加到您的函數中。

本主題包含如何使用外部程式庫相依性正確封裝和建立 Java Lambda 層的步驟和指導。

主題

- 必要條件
- 與 Amazon Linux 的 Java 層兼容性
- Java 執行階段的圖層路徑
- 封裝圖層內容
- 建立圖層
- 將圖層添加到您的函數

必要條件

若要遵循本節中的步驟，您必須具備下列項目：

- 爪哇共和國
- 阿帕奇 Maven 3.8.6 或更高版本
- AWS Command Line Interface (AWS CLI) 第二版

> **Note**
>
> 確保 Maven 引用的 Java 版本與您打算部署的函數的 Java 版本相同。例如，對於 Java 21 函數，該`mvn -v`命令應該在輸出中列出 Java 版本 21：
>
> ```
> Apache Maven 3.8.6
> ...
> ```

```
Java version: 21.0.2, vendor: Oracle Corporation, runtime: /Library/Java/
JavaVirtualMachines/jdk-21.jdk/Contents/Home
...
```

在本主題中，我們會參考 awsdocs GitHub 儲存庫上的layer-java範例應用程式。此應用程序包含下載依賴關係並生成圖層的腳本。該應用程序還包含一個使用圖層中的依賴關係的對應函數。創建圖層後，您可以部署和調用相應的功能以驗證一切正常。因為您使用 Java 21 執行階段作為函式，所以這些圖層也必須與 Java 21 相容。

layer-java範例應用程式包含兩個子目錄中的單一範例。目layer錄包含定義圖層相依性的pom.xml檔案，以及用來產生圖層的指令碼。目function錄包含範例函數，可協助測試圖層是否有效。本自學課程將逐步介紹如何建立和封裝此圖層。

與 Amazon Linux 的 Java 層兼容性

建立層的第一步是將所有層內容綁定至 .zip 封存檔。由於 Lambda 函數是在 Amazon Linux 上執行，因此您的層內容必須能夠在 Linux 環境中編譯和建置。

Java 代碼被設計為獨立於平台，因此即使它不使用 Linux 環境，您也可以將圖層打包到本地計算機上。將 Java 層上傳到 Lambda 之後，它仍然可以與 Amazon Linux 兼容。

Java 執行階段的圖層路徑

將層新增至函數時，Lambda 會將層內容載入該執行環境的 /opt 目錄。在每一次 Lambda 執行期中，PATH 變數已包含 /opt 目錄中的特定資料夾路徑。若要確保PATH變數會擷取圖層內容，您的圖層 .zip 檔案應該在下列資料夾路徑中具有其相依性：

- java/lib

例如，您在此自學課程中建立的產生圖層 .zip 檔案具有以下目錄結構：

```
layer_content.zip
# java
    # lib
        # layer-java-layer-1.0-SNAPSHOT.jar
```

layer-java-layer-1.0-SNAPSHOT.jarJAR 文件（包含我們所有必需的依賴關係的 uber-jar）正確位於目錄中。java/lib這可確保 Lambda 可以在函數叫用期間找到程式庫。

封裝圖層內容

在此範例中，您可以將下列兩個 Java 程式庫封裝到單一 JAR 檔案中：

- aws-lambda-java-core— 一組最小的介面定義，用於在中使用 Java AWS Lambda
- 傑克遜-一套流行的數據處理工具，特別是用於使用 JSON。

完成下列步驟以安裝和封裝圖層內容。

安裝和封裝圖層內容的步驟

1. 克隆存aws-lambda-developer-guide GitHub 儲庫，其中包含您在sample-apps/layer-java目錄中需要的示例代碼。

```
git clone https://github.com/awsdocs/aws-lambda-developer-guide.git
```

2. 導覽至範layer-java例應用程式的layer目錄。此目錄包含您用來建立和正確封裝圖層的程序檔。

```
cd aws-lambda-developer-guide/sample-apps/layer-java/layer
```

3. 檢查pom.xml檔案。在<dependencies>本節中，您可以定義要包括在圖層中的相依性，也就是aws-lambda-java-core和jackson-databind資源庫。您可以更新此檔案，以包括要包含在自己圖層中的任何相依性。

   Example pom.xml

```
<dependencies>
    <dependency>
        <groupId>com.amazonaws</groupId>
        <artifactId>aws-lambda-java-core</artifactId>
        <version>1.2.3</version>
    </dependency>

    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>2.17.0</version>
    </dependency>
</dependencies>
```

> **Note**
>
> 此pom.xml檔案的<build>區段包含兩個外掛程式。會 maven-compiler-plugin編譯原始程式碼。將您的工maven-shade-plugin件打包到單個 uber-jar 中。

4. 請確定您擁有執行這兩個指令碼的權限。

```
chmod 744 1-install.sh && chmod 744 2-package.sh
```

5. 使用下列命令執行指令1-install.sh碼：

```
./1-install.sh
```

此腳本mvn clean install在當前目錄中運行。這將創建具有目錄中所有必需的依賴關係的uber-jar。target/

Example 1-install.sh

```
mvn clean install
```

6. 使用下列命令執行指令2-package.sh碼：

```
./2-package.sh
```

此指令碼會建立正確封裝圖層內容所需的java/lib目錄結構。然後，它將 uber-jar 從目錄複製到新創建的/target目錄中。java/lib最後，腳本壓縮java目錄的內容到一個名為layer_content.zip的文件。這是圖層的 .zip 檔案。您可以解壓縮檔案，並確認檔案包含正確的檔案結構，如the section called “Java 執行階段的圖層路徑”本節所示。

Example 2-package.sh

```
mkdir java
mkdir java/lib
cp -r target/layer-java-layer-1.0-SNAPSHOT.jar java/lib/
zip -r layer_content.zip java
```

建立圖層

在本節中，您將取得上一節中產生的`layer_content.zip`檔案，並將其上傳為 Lambda 層。您可以透過 AWS Command Line Interface (AWS CLI) 使用 AWS Management Console 或 Lambda API 上傳圖層。上傳圖層 .zip 檔案時，請在下列指PublishLayerVersion AWS CLI 令中指定`java21`為相容的執行階段，並指定`arm64`為相容的架構。

```
aws lambda publish-layer-version --layer-name java-jackson-layer \
    --zip-file fileb://layer_content.zip \
    --compatible-runtimes java21 \
    --compatible-architectures "arm64"
```

從響應中，請注意`LayerVersionArn`，看起來像`arn:aws:lambda:us-east-1:`*`123456789012`*`:layer:java-jackson-layer:1`。當您將圖層添加到函數時，在本教程的下一步中，您將需要此 Amazon 資源名稱（ARN）。

將圖層添加到您的函數

在本節中，您將在函數程式碼中部署使用 Jackson 程式庫的範例 Lambda 函數，然後附加圖層。要部署該功能，您需要一個the section called “執行角色 (函數存取其他資源的權限)”. 如果您沒有現有的執行角色，請依照可摺疊區段中的步驟執行。否則，請跳至下一節以部署該功能。

(選擇性) 建立執行角色

若要建立執行角色

1. 在 IAM 主控台中開啟 角色頁面。
2. 選擇 建立角色。
3. 建立具備下列屬性的角色。
   - 信任實體 - Lambda。
   - 權限 — AWSLambdaBasicExecutionRole.
   - 角色名稱 - **`lambda-role`**。

   該AWSLambdaBasicExecutionRole策略具有函數將日誌寫入日誌所需的 CloudWatch 權限。

若要部署 Lambda 函數

1. 導覽至 function/ 目錄。如果您目前位於目layer/錄中，請執行下列命令：

```
cd ../function
```

2. 檢閱函數程式碼。該函數接受Map<String, String>作為輸入，並使用傑克遜將輸入寫入為 JSON 字符串，然後再將其轉換為預定義的 F1Car Java 對象。最後，函數會使用 F1Car 物件中的欄位來建構函數傳回的字串。

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.fasterxml.jackson.databind.ObjectMapper;

import java.io.IOException;
import java.util.Map;

public class Handler {

    public String handleRequest(Map<String, String> input, Context context) throws
 IOException {
        // Parse the input JSON
        ObjectMapper objectMapper = new ObjectMapper();
        F1Car f1Car =
 objectMapper.readValue(objectMapper.writeValueAsString(input), F1Car.class);

        StringBuilder finalString = new StringBuilder();
        finalString.append(f1Car.getDriver());
        finalString.append(" is a driver for team ");
        finalString.append(f1Car.getTeam());
        return finalString.toString();
    }
}
```

3. 使用下面的 Maven 命令構建項目：

```
mvn package
```

這個命令會在名為的target/目錄中產生一個 JAR 檔案layer-java-function-1.0-SNAPSHOT.jar。

4. 部署功能。在下列 AWS CLI 命令中，以您的執行角色 ARN 取代`--role`參數：

```
aws lambda create-function --function-name java_function_with_layer \
    --runtime java21 \
    --architectures "arm64" \
    --handler example.Handler::handleRequest \
    --timeout 30 \
    --role arn:aws:iam::123456789012:role/lambda-role \
    --zip-file fileb://target/layer-java-function-1.0-SNAPSHOT.jar
```

（可選）調用您的函數而不附加圖層

此時，您可以選擇性地嘗試在附加圖層之前調用函數。如果你嘗試這個，那麼你應該得到一個，`ClassNotFoundException`因為你的函數無法引用該`requests`包。要調用您的函數，請使用以下 AWS CLI 命令：

```
aws lambda invoke --function-name java_function_with_layer \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "driver": "Max Verstappen", "team": "Red Bull" }' response.json
```

您應該會看到輸出，如下所示：

```
{
    "StatusCode": 200,
    "FunctionError": "Unhandled",
    "ExecutedVersion": "$LATEST"
}
```

若要檢視特定錯誤，請開啟輸出`response.json`檔案。您應該會看到`ClassNotFoundException`包含以下錯誤訊息：

```
"errorMessage":"com.fasterxml.jackson.databind.ObjectMapper","errorType":"java.lang.ClassNotFou
```

接下來，將圖層附加到功能上。在下列 AWS CLI 指令中，將`--layers`參數取代為您先前提到的圖層版本 ARN：

```
aws lambda update-function-configuration --function-name java_function_with_layer \
    --cli-binary-format raw-in-base64-out \
    --layers "arn:aws:lambda:us-east-1:123456789012:layer:java-jackson-layer:1"
```

最後，嘗試使用以下 AWS CLI 命令調用您的函數：

```
aws lambda invoke --function-name java_function_with_layer \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "driver": "Max Verstappen", "team": "Red Bull" }' response.json
```

您應該會看到輸出，如下所示：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
```

這表明該功能能夠使用傑克遜依賴關係來正確執行該功能。您可以檢查輸出`response.json`文件是否包含正確的返回字符串：

```
"Max Verstappen is a driver for team Red Bull"
```

(選擇性) 清理您的資源

除非您想要保留為此教學課程建立的資源，否則您現在便可刪除。刪除您不再使用的 AWS 資源，您可以避免不必要的費用 AWS 帳戶。

若要刪除 Lambda 圖層

1. 開啟 Lambda 主控台中的 層頁面。
2. 選取您建立的圖層。
3. 選擇刪除，然後再次選擇刪除。

若要刪除 Lambda 函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇您建立的函數。
3. 選擇 Actions (動作)、Delete (刪除)。
4. 在文字輸入欄位中輸入 **delete**，然後選擇 刪除 。

使用 Lambda 改善啟動效能 SnapStart

適用 SnapStart 於 Java 的 Lambda 可將延遲敏感應用程式的啟動效能提升高達 10 倍，無需額外成本，通常不需要變更函數程式碼。啟動延遲的最大歸因 (通常稱為冷啟動時間) 是 Lambda 花費在初始化函數的時間，其中包括載入函數的程式碼、啟動執行階段以及初始化函數程式碼。

Lambda 會在您發佈函數版本時初始化您的函數。 SnapStartLambda 會擷取已初始化執行環境的記憶體和磁碟狀態的 Firecracker microVM 快照、將快照加密，並快取以進行低延遲存取。第一次調用函數版本時且呼叫縱向擴展時，Lambda 會從快取的快照恢復新的執行環境，而不是從頭開始執行，進而改善啟動延遲。

> **Important**
>
> 如果您的應用程式依賴狀態的唯一性，則必須評估函數程式碼，並確認其對快照作業具有復原能力。如需詳細資訊，請參閱 以 Lambda 處理獨特性 SnapStart。

主題

支援的功能和限制

SnapStart 支援 Java 11 及更新版本的 Java 管理執行階段。不支援其他受管執行期 (例如 `nodejs20.x` 和 `python3.12`)、僅限作業系統的執行期 和容器映像。

SnapStart 不支援佈建並行、arm64 架構、Amazon Elastic File System (Amazon EFS) 或大於 512 MB 的暫時儲存。

若要使用 SnapStart，您可以使用 Lambda 主控台、 AWS Command Line Interface (AWS CLI)、Lambda API AWS SDK for Java AWS CloudFormation、、、 AWS Serverless Application Model (AWS SAM) 和 AWS Cloud Development Kit (AWS CDK). 如需詳細資訊，請參閱 啟用和管理 Lambda SnapStart。

> **Note**
>
> 您 SnapStart 只能在指向版本的已發佈函數版本和別名上使用。您不能 SnapStart 在函數的未發布版本（$LATEST）上使用。

支援地區

SnapStart 可在以下內容中使用 AWS 區域：

- 美國東部 (維吉尼亞北部)
- 美國東部 (俄亥俄)
- 美國西部 (加利佛尼亞北部)
- 美國西部 (奧勒岡)
- 非洲 (開普敦)
- 亞太區域 (香港)
- Asia Pacific (Mumbai)
- 亞太區域 (海德拉巴)
- 亞太區域 (東京)
- 亞太區域 (首爾)
- 亞太區域 (大阪)
- 亞太區域 (新加坡)
- 亞太區域 (悉尼)
- 亞太區域 (雅加達)
- 亞太區域 (墨爾本)

- 加拿大 (中部)
- 歐洲 (斯德哥爾摩)
- 歐洲 (法蘭克福)
- 歐洲 (蘇黎世)
- 歐洲 (愛爾蘭)
- 歐洲 (倫敦)
- Europe (Paris)
- 歐洲 (米蘭)
- 歐洲 (西班牙)
- 中東 (阿拉伯聯合大公國)
- Middle East (Bahrain)
- 南美洲 (聖保羅)

相容性考量

Lambda 會使用單一快照作為多個執行環境的初始狀態。 SnapStart如果您的函數在初始化階段使用以下任何一項，則可能需要在使用之前進行一些更改 SnapStart：

Uniqueness

如果您的初始化程式碼會產生包含在快照中的唯一內容，則在跨執行環境中重複使用該內容時，該內容可能不是唯一的。為了在使用時保持唯一性 SnapStart，您必須在初始化後生成唯一的內容。這包括唯一 ID、唯一密碼，以及用來產生偽隨機性的熵。若要了解如何還原唯一性，請參閱：以 Lambda 處理獨特性 SnapStart。

網路連線

Lambda 從快照恢復函數時，無法保證函數在初始化階段建立的連線狀態。驗證網路連線的狀態，並視需要重新建立。在大多數情況下， AWS SDK 建立的網路連線會自動恢復。針對其他連線，請檢閱最佳實務。

暫存資料

某些函數會在初始化階段下載或初始化暫存資料，例如臨時憑證或快取的時間戳記。在使用函數處理常式之前重新整理暫時資料，即使不使用也是如此。 SnapStart

SnapStart 定價

沒有任何額外費用 SnapStart。我們會根據函數的請求數量、程式碼執行的時間，以及為函數配置的記憶體向您收費。持續時間是從程式碼開始執行的時間開始計算，直到傳回資料或結束為止，四捨五入至最接近的 1 ms。

持續時間費用適用於在函數處理常式中執行的程式碼、在處理常式之外宣告的初始化程式碼、執行階段 (JVM) 載入所花的時間，以及在執行階段掛鉤中執行的任何程式碼。如需 Lambda 如何計算持續時間的詳細資訊，請參閱：監控 Lambda SnapStart。

對於使用設定的函數 SnapStart，Lambda 會定期回收執行環境，並重新執行初始化程式碼。為提供備援，Lambda 會在多個可用區域建立快照。每次 Lambda 在另一個可用區域重新執行初始化程式碼時，都會收取費用。如需費用的詳細資訊，請參閱 AWS Lambda 定價。

比較 Lambda SnapStart 和佈建並行

當函數擴展時，Lambda SnapStart 和佈建的並行可以減少冷啟動和異常延遲。 SnapStart 幫助您將啟動性能提高高達 10 倍，無需額外費用。佈建並行可讓函數保持初始化，並準備好在兩位數的毫秒內回應。設定已佈建的並行會產生您的. AWS 帳戶如果您的應用程式有嚴格的冷啟動延遲需求，請使用佈建並行。您不能在相同的函數版本上同時使用 SnapStart 和佈建的並發。

> **Note**
>
> SnapStart 與大規模函數調用一起使用時效果最好。不常調用的函數效能改進效果可能不會相同。

其他資源

除了閱讀本章中的其他主題之外，我們還建議您嘗試使用 AWS Lambda SnapStart研討會更快啟動，並觀看 AWS Re: Invent 2022 開始的 Java 函數會話的快速冷啟動。

啟用和管理 Lambda SnapStart

若要使用 SnapStart，請 SnapStart 在新的或現有的 Lambda 函數上啟用。然後發布並調用函數版本。

主題

啟動 SnapStart (主控台)

若要啟 SnapStart 動某個函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇函數的名稱。
3. 選擇 組態 ，然後選擇 一般組態 。
4. 在 一般組態 窗格中，選擇 編輯 。
5. 在 [編輯基本設定] 頁面上 SnapStart，選擇 [已發佈的版本]。
6. 選擇儲存。
7. 發佈函數版本。Lambda 會初始化程式碼、建立初始化執行環境的快照，然後快取快照以實現低延遲存取。
8. 調用函數版本。

啟動 SnapStart （ AWS CLI ）

SnapStart 為現有功能啟動

1. 透過使用--snap-start選項執行update-function-configuration指令來更新函數組態。

```
aws lambda update-function-configuration \
  --function-name my-function \
  --snap-start ApplyOn=PublishedVersions
```

2. 使用 publish-version 命令發佈函數版本。

```
aws lambda publish-version \
  --function-name my-function
```

3. 執行指 SnapStart get-function-configuration令並指定版本號碼，以確認已針對函數版本啟用。以下範例命令指定第 1 版。

```
aws lambda get-function-configuration \
  --function-name my-function:1
```

如果回應顯示為「OptimizationStatus是」(State) On 且「狀態」 SnapStart 為Active，則會啟動且快照可供指定的函數版本使用。

```
"SnapStart": {
    "ApplyOn": "PublishedVersions",
    "OptimizationStatus": "On"
 },
 "State": "Active",
```

4. 執行 invoke 命令並指定版本來調用函數版本。以下範例調用第 1 版。

```
aws lambda invoke \
  --cli-binary-format raw-in-base64-out \
  --function-name my-function:1 \
  --payload '{ "name": "Bob" }' \
  response.json
```

如果您使用 AWS CLI 第 2 版，則需要 cli-binary-format 選項。若要讓此成為預設的設定，請執行 `aws configure set cli-binary-format raw-in-base64-out`。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

若要 SnapStart 在建立新函數時啟動

1. 執行 create-function 命令搭配 --snap-start 選項來建立函數。針對 --role，請指定執行角色的 Amazon Resource Name (ARN)。

```
aws lambda create-function \
  --function-name my-function \
  --runtime "java21" \
  --zip-file fileb://my-function.zip \
  --handler my-function.handler \
  --role arn:aws:iam::111122223333:role/lambda-ex \
  --snap-start ApplyOn=PublishedVersions
```

2. 使用 publish-version 命令來建立版本。

```
aws lambda publish-version \
  --function-name my-function
```

3. 執行指 SnapStart get-function-configuration令並指定版本號碼，以確認已針對函數版本啟用。以下範例命令指定第 1 版。

```
aws lambda get-function-configuration \
  --function-name my-function:1
```

如果回應顯示為「OptimizationStatus是」(State) On 且「狀態」 SnapStart 為Active，則會啟動且快照可供指定的函數版本使用。

```
"SnapStart": {
     "ApplyOn": "PublishedVersions",
     "OptimizationStatus": "On"
  },
  "State": "Active",
```

4. 執行 invoke 命令並指定版本來調用函數版本。以下範例調用第 1 版。

```
aws lambda invoke \
  --cli-binary-format raw-in-base64-out \
  --function-name my-function:1 \
  --payload '{ "name": "Bob" }' \
  response.json
```

如果您使用 AWS CLI 第 2 版，則需要 cli-binary-format 選項。若要讓此成為預設的設定，請執行 `aws configure set cli-binary-format raw-in-base64-out`。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

啟動 SnapStart (API)

若要啟動 SnapStart

1. 執行以下任意一項：
   - 透過將 CreateFunctionAPI 動作與參數搭配使用，建立 SnapStart 啟動的新函SnapStart數。
   - 透過將UpdateFunctionConfiguration動作與參數搭配使用來啟 SnapStart 用既有函SnapStart數。
2. 使用PublishVersion動作發佈函數版本。Lambda 會初始化程式碼、建立初始化執行環境的快照，然後快取快照以實現低延遲存取。
3. 使用 SnapStart 動作確認功能版本已啟GetFunctionConfiguration動。指定版本號碼，以確認 SnapStart 已為該版本啟動。如果回應顯示為「OptimizationStatus是」(State) On 且「狀態」SnapStart 為`Active`，則會啟動且快照可供指定的函數版本使用。

```
"SnapStart": {
        "ApplyOn": "PublishedVersions",
        "OptimizationStatus": "On"
    },
    "State": "Active",
```

4. 使用 Invoke 動作調用函數版本。

Lambda SnapStart 和函數狀態

當您使用時，可能會發生下列函數狀態 SnapStart。當 Lambda 定期回收執行環境，並針對已設定的函數重新執行初始化程式碼時，也會發生這種情況。 SnapStart

- `Pending` - Lambda 正在初始化您的程式碼，並擷取初始化執行環境的快照。在函數上操作的任何調用或其他 API 動作都會失敗。
- `Active` - 快照建立完成，您可以調用函數。要使用 SnapStart，您必須調用已發布的函數版本，而不是未發布的版本（$LATEST）。

- `Inactive` - 函數版本已經 14 天沒有被調用。當函數版本變成 `Inactive`，Lambda 會刪除快照。如果您在 14 天後調用函數版本，Lambda 會傳回 `SnapStartNotReadyException` 回應並開始初始化新快照。等待函數版本進入 `Active` 狀態，然後再次調用它。
- `Failed` - Lambda 在執行初始化程式碼或建立快照時發生錯誤。

更新快照

Lambda 會為每個已發佈的函數版本建立快照。若要更新快照，請發佈新函數版本。Lambda 會使用最新的執行階段和安全性修補程式自動更新快照。

SnapStart 搭配使用 AWS SDK for Java

為了從函數發出 AWS SDK 呼叫，Lambda 會擔任函數的執行角色，以產生一組暫時性憑證。這些憑證在函數調用期間可當成環境變數使用。您不需要直接在程式碼中提供 SDK 的憑證。根據預設，憑證提供者鏈會依序檢查您可以設定憑證的每個位置，並選擇第一個可用的位置，通常是環境變數 (`AWS_ACCESS_KEY_ID`、`AWS_SECRET_ACCESS_KEY` 和 `AWS_SESSION_TOKEN`)。

> **Note**
>
> 啟動 SnapStart 時，Java 執行階段會自動使用容器認證 (`AWS_CONTAINER_CREDENTIALS_FULL_URI`和`AWS_CONTAINER_AUTHORIZATION_TOKEN`)，而非存取金鑰環境變數。這樣可防止憑證在函數還原之前過期。

SnapStart 與AWS CloudFormation、AWS SAM和一起使用 AWS CDK

- AWS CloudFormation：在範本中宣告SnapStart實體。
- AWS Serverless Application Model（AWS SAM）：在模板中聲明SnapStart屬性。
- AWS Cloud Development Kit (AWS CDK)：使用SnapStartProperty類型。

刪除快照

Lambda 會在發生以下情況時刪除快照：

- 您刪除了函數或函數版本。

- 您 14 天內都沒有調用函數版本。連續 14 天都沒有呼叫之後，函數版本會轉換為 Inactive (非作用中) 狀態。如果您在 14 天後調用函數版本，Lambda 會傳回 `SnapStartNotReadyException` 回應並開始初始化新快照。等待函數版本進入 Active (作用中) 狀態，然後再次調用它。

根據一般資料保護規範 (GDPR)，Lambda 會移除與已刪除快照相關聯的所有資源。

以 Lambda 處理獨特性 SnapStart

當呼叫在 SnapStart 函數上擴展時，Lambda 會使用單一初始化快照來恢復多個執行環境。如果您的初始化程式碼會產生包含在快照中的唯一內容，則在跨執行環境中重複使用該內容時，該內容可能不是唯一的。為了在使用時保持唯一性 SnapStart，您必須在初始化後生成唯一的內容。這包括唯一 ID、唯一密碼，以及用來產生偽隨機性的熵。

以下是維持程式碼唯一性的建議最佳實務。Lambda 也提供開放原始碼SnapStart 掃描工具，協助檢查是否具有唯一性的程式碼。如果您在初始化階段產生唯一資料，便可使用執行階段掛鉤來還原唯一性。使用執行階段掛鉤，您可以在 Lambda 建立快照前執行特定程式碼，或在 Lambda 從快照恢復函數後立即執行特定程式碼。

避免儲存初始化期間依賴唯一性的狀態

在函數的初始化階段，請避免快取預定為唯一的資料，例如產生用於日誌記錄的唯一 ID。相反地，建議您在函數處理常式中產生唯一的資料，或使用執行階段掛鉤。

Example - 在函數處理常式中產生唯一 ID

以下程式碼範例示範如何在函數處理常式中產生 UUID。

```
import java.util.UUID;
  public class Handler implements RequestHandler<String, String> {
    private static UUID uniqueSandboxId = null;
    @Override
    public String handleRequest(String event, Context context) {
      if (uniqueSandboxId == null)
        uniqueSandboxId = UUID.randomUUID();
      System.out.println("Unique Sandbox Id: " + uniqueSandboxId);
      return "Hello, World!";
    }
  }
```

使用密碼編譯安全的虛擬隨機數產生器 (CSPRNGs)

如果您的應用程式依賴隨機性，建議您使用密碼編譯安全隨機數產生器 (CSPRNGs)。Java 的 Lambda 受管理執行階段包含兩個內建的 CSRNG (OpenSSL 1.0.2 和`java.security.SecureRandom`)，可自動維護隨機性。 SnapStart始終從中獲取隨機數`/dev/random`或`/dev/urandom`保持隨機性的軟件。 SnapStart

Example — 爪哇. 安全性. SecureRandom

下列範例使用 `java.security.SecureRandom`，即使從快照還原函數，也會產生唯一的序號。

```
import java.security.SecureRandom;
  public class Handler implements RequestHandler<String, String> {
    private static SecureRandom rng = new SecureRandom();
    @Override
    public String handleRequest(String event, Context context) {
      for (int i = 0; i < 10; i++) {
        System.out.println(rng.next());
      }
      return "Hello, World!";
    }
  }
```

SnapStart 掃描工具

Lambda 提供掃描工具，可協助檢查確保唯一性的程式碼。 SnapStart 掃描工具是一種開放原始碼SpotBugs外掛程式，可針對一組規則執行靜態分析。掃描工具有助於識別可能會破壞有關唯一性假設的潛在程式碼實作。如需安裝指示和掃描工具執行的檢查清單，請參閱上 GitHub的 aws-lambda-snapstart-java-rules 儲存庫。

若要深入了解如何使用處理唯一性 SnapStart，請參閱 AWS Compute 部落格AWS Lambda SnapStart上的加快啟動速度。

在 Lambda 函數快照前後實作程式碼

您可以在 Lambda 建立快照之前或 Lambda 從快照恢復函數之後，使用執行階段掛鉤來實作程式碼。執行階段掛鉤可做為開放原始碼 Coordinated Restore at Checkpoint (CRaC) 專案的一部分使用。目前正在針對 Java 開發套件 (OpenJDK) 開發 CRaC。如需如何搭配參考應用程式使用 CraC 的範例，請參閱上的 CraC 儲存庫。 GitHubCRaC 使用三個主要元素：

- `Resource` - 具有兩種方法 (`beforeCheckpoint()` 和 `afterRestore()`) 的介面。使用這些方法來實作您想在快照建立之前和還原之後執行的程式碼。
- `Context <R extends Resource>` - 若要接收檢查點和還原的通知，必須透過 `Context` 註冊 `Resource`。
- `Core` - 協調服務，透過靜態方法 `Core.getGlobalContext()` 提供預設的全域 `Context`。

如需 `Context` 和 `Resource` 的詳細資訊，請參閱 CRaC 文件中的套件 org.crac。

使用下列步驟，透過 org.crac 套件實作執行階段掛鉤。Lambda 執行階段包含自訂的 CRaC 內容實作，可在檢查點之前和還原之後呼叫執行階段掛鉤。

步驟 1：更新建置組態

將 `org.crac` 相依性新增到建置組態。以下範例使用 Gradle。如需其他建置系統的範例，請參閱 Apache Maven 文件。

```
dependencies {
    compile group: 'com.amazonaws', name: 'aws-lambda-java-core', version: '1.2.1'
    # All other project dependecies go here:
    # ...
    # Then, add the org.crac dependency:
 implementation group: 'org.crac', name: 'crac', version: '1.4.0'
}
```

步驟 2：更新 Lambda 處理常式

Lambda 函數處理常式是您的函數程式碼中處理事件的方法。當有人呼叫您的函數時，Lambda 會執行處理常式方法。函數會執行，直到處理常式傳回回應、結束或逾時為止。

如需詳細資訊，請參閱 在 Java 中 Lambda 義函數處理程序。

以下處理常式範例示範如何在檢查點之前 (`beforeCheckpoint()`) 和還原之後 (`afterRestore()`) 執行程式碼。此處理常式也會向執行階段管理的全域 `Context` 註冊 `Resource`。

> **Note**
>
> Lambda 建立快照時，初始化程式碼最多可能會執行 15 分鐘。時間上限為 130 秒或設定的函數逾時 (最長 900 秒)，以較高者為準。您的 `beforeCheckpoint()` 執行時間掛鉤會計入初始化程式碼時間限制。Lambda 還原快照時，執行階段 (JVM) 必須載入，且 `afterRestore()` 執行階段掛鉤必須在逾時限制 (10 秒) 內完成。否則，你會得到一個 SnapStartTimeoutException.

```
...
  import org.crac.Resource;
  import org.crac.Core;
  ...
public class CRaCDemo implements RequestStreamHandler, Resource {
    public CRaCDemo() {
      Core.getGlobalContext().register(this);
    }
    public String handleRequest(String name, Context context) throws IOException {
      System.out.println("Handler execution");
      return "Hello " + name;
    }
    @Override
    public void beforeCheckpoint(org.crac.Context<? extends Resource> context)
        throws Exception {
      System.out.println("Before checkpoint");
    }
    @Override
    public void afterRestore(org.crac.Context<? extends Resource> context)
        throws Exception {
      System.out.println("After restore");
```

`Context` 僅會對已註冊物件維持 `WeakReference`。如果 `Resource` 是回收的垃圾，則執行階段掛鉤程式不會執行。您的程式碼必須維持對 `Resource` 的強式參考，才能保證執行階段掛鉤會執行。

以下是須避免的兩個模式範例：

Example - 沒有強式參考的物件

```
Core.getGlobalContext().register( new MyResource() );
```

Example - 匿名類別的物件

```
Core.getGlobalContext().register( new Resource() {

   @Override
   public void afterRestore(Context<? extends Resource> context) throws Exception {
    // ...
   }

   @Override
   public void beforeCheckpoint(Context<? extends Resource> context) throws Exception {
    // ...
   }

} );
```

而是維持強式參考。在下列範例中，已註冊的資源不是回收的垃圾，且執行階段掛鉤會一致地執行。

Example - 有強式參考的物件

```
Resource myResource = new MyResource(); // This reference must be maintained to prevent
 the registered resource from being garbage collected
Core.getGlobalContext().register( myResource );
```

監控 Lambda SnapStart

您可以使用 Amazon 監控您的 Lambda SnapStart 函數 CloudWatchAWS X-Ray，和Lambda 遙測 API.

> **Note**
>
> AWS_LAMBDA_LOG_GROUP_NAME和AWS_LAMBDA_LOG_STREAM_NAME環境變數在 Lambda SnapStart 函數中不可用。

CloudWatch 對於 SnapStart

與 SnapStart 函數的CloudWatch 日誌流格式有一些差異：

- 初始化日誌：建立新的執行環境時，REPORT 不會含有 Init Duration 欄位。這是因為 Lambda 會在您建立版本時初始化 SnapStart 函數，而不是在函數叫用期間。對於 SnapStart 函數，Init Duration欄位位於記INIT_REPORT錄中。此記錄會顯示 初始化階段 的持續時間詳細資訊，包括任何 beforeCheckpoint 執行階段掛鉤的持續時間。
- 調用日誌：建立新的執行環境時，REPORT 會含有 Restore Duration 和 Billed Restore Duration 欄位：
  - Restore Duration：Lambda 還原快照、載入執行時間 (JVM) 和執行任何 afterRestore 執行時間掛鉤所需的時間。還原快照的程序可能包括在 MicroVM 以外的活動上花費的時間。此時間在 Restore Duration 中報告。
  - Billed Restore Duration：Lambda 載入執行時間 (JVM) 和執行任何 afterRestore 掛鉤所需的時間。您不需為還原快照所花的時間支付費用。

> **Note**
>
> 持續時間費用適用於在函數處理常式中執行的程式碼、在處理常式之外宣告的初始化程式碼、執行階段 (JVM) 載入所花的時間，以及在執行階段掛鉤中執行的任何程式碼。如需詳細資訊，請參閱 SnapStart 定價。

冷啟動持續時間是 Restore Duration + Duration 的總和。

下列範例是傳回函數延遲百分位數的 Lambda 見解查詢。 SnapStart如需 Lambda Insights 查詢的詳細資訊，請參閱：使用查詢故障排除函式的範例工作流程。

```
filter @type = "REPORT"
  | parse @log /\d+:\/aws\/lambda\/(?<function>.*)/
  | parse @message /Restore Duration: (?<restoreDuration>.*?) ms/
  | stats
count(*) as invocations,
pct(@duration+coalesce(@initDuration,0)+coalesce(restoreDuration,0), 50) as p50,
pct(@duration+coalesce(@initDuration,0)+coalesce(restoreDuration,0), 90) as p90,
pct(@duration+coalesce(@initDuration,0)+coalesce(restoreDuration,0), 99) as p99,
pct(@duration+coalesce(@initDuration,0)+coalesce(restoreDuration,0), 99.9) as p99.9
group by function, (ispresent(@initDuration) or ispresent(restoreDuration)) as
 coldstart
  | sort by coldstart desc
```

X-Ray 主動追蹤 SnapStart

您可以使用 X-Ray 來追蹤 Lambda SnapStart 函數的請求。與 SnapStart 功能的 X-Ray 子段有一些差異：

- SnapStart 函數沒有`Initialization`子區段。
- `Restore` 子區段會顯示 Lambda 還原快照、載入執行期 (JVM) 和執行任何 `afterRestore` 執行期勾點所需的時間。還原快照的程序可能包括在 MicroVM 以外的活動上花費的時間。此時間在 `Restore` 子區段中報告。您不需要為在 MicroVM 外還原快照所花費的時間付費。

適用於的遙測 API 事件 SnapStart

Lambda 會將下列 SnapStart 事件傳送至遙測 API：

- `platform.restoreStart` - 顯示 `Restore` 階段開始的時間。
- `platform.restoreRuntimeDone` - 顯示 `Restore` 階段是否成功。執行階段傳送 `restore/next` 執行階段 API 請求時，Lambda 會產生此訊息。可能的狀態有三種：成功、失敗和逾時。
- `platform.restoreReport` - 顯示 `Restore` 階段持續的時間長度，以及您須為此階段支付的費用。

Amazon API Gateway 和函數 URL 指標

如果您使用 API Gateway 建立 Web API，則可以使用IntegrationLatency指標來測量 end-to-end 延遲 (API Gateway 將要求轉送至後端，以及從後端收到回應之間的時間)。

如果您使用 Lambda 函數 URL，則可以使用UrlRequestLatency指標來測量 end-to-end 延遲 (函數 URL 接收請求到函數 URL 傳回回應之間的時間)。

Lambda 的安全性模型 SnapStart

Lambda SnapStart 支援靜態加密。Lambda 使用 AWS KMS key 來加密快照。預設情況下，Lambda 使用 AWS 受管金鑰。如果此預設行為符合您的工作流程，您便不需要設定其他項目。否則，您可以使用創建功能或update-function-configuration命令中的`--kms-key-arn`選項來提供AWS KMS客戶管理的密鑰。這麼做可控制 KMS 金鑰的輪換或滿足貴組織對管理 KMS 金鑰的要求。客戶受管的金鑰會產生標準的 AWS KMS 費用。如需詳細資訊，請參閱 AWS Key Management Service 定價。

當您刪除 SnapStart 函數或函數版本時，對該函數或函數版本的所有`Invoke`請求都會失敗。Lambda 會自動刪除 14 天未叫用的快照。根據一般資料保護規範 (GDPR)，Lambda 會移除與已刪除快照相關聯的所有資源。

最大 Lambda SnapStart 效能

主題

效能調校

> **Note**
>
> SnapStart 與大規模函數調用一起使用時效果最好。不常調用的函數效能改進效果可能不會相同。

為了充分發揮的優點 SnapStart，我們建議您在初始化程式碼中預先載入會導致啟動延遲的類別，而不是在函數處理常式中。這會將與繁重類別載入相關聯的延遲移出呼叫路徑，從而最佳化啟動效能。SnapStart

如果您無法在初始化期間預先載入類別，建議您使用虛擬調用預先載入類別。若要這麼做，請更新函數處理常式程式程式碼，如下列範例所示，從 AWS Labs GitHub 儲存庫上的寵物存放區函式。

```
private static SpringLambdaContainerHandler<AwsProxyRequest, AwsProxyResponse> handler;
  static {
      try {
          handler =
 SpringLambdaContainerHandler.getAwsProxyHandler(PetStoreSpringAppConfig.class);

          // Use the onStartup method of the handler to register the custom filter
          handler.onStartup(servletContext -> {
              FilterRegistration.Dynamic registration =
 servletContext.addFilter("CognitoIdentityFilter", CognitoIdentityFilter.class);
              registration.addMappingForUrlPatterns(EnumSet.of(DispatcherType.REQUEST),
 false, "/*");
          });

          // Send a fake Amazon API Gateway request to the handler to load classes
 ahead of time
          ApiGatewayRequestIdentity identity = new ApiGatewayRequestIdentity();
          identity.setApiKey("foo");
          identity.setAccountId("foo");
```

```
            identity.setAccessKey("foo");

            AwsProxyRequestContext reqCtx = new AwsProxyRequestContext();
            reqCtx.setPath("/pets");
            reqCtx.setStage("default");
            reqCtx.setAuthorizer(null);
            reqCtx.setIdentity(identity);

            AwsProxyRequest req = new AwsProxyRequest();
            req.setHttpMethod("GET");
            req.setPath("/pets");
            req.setBody("");
            req.setRequestContext(reqCtx);

            Context ctx = new TestContext();
            handler.proxy(req, ctx);


        } catch (ContainerInitializationException e) {
            // if we fail here. We re-throw the exception to force another cold start
            e.printStackTrace();
            throw new RuntimeException("Could not initialize Spring framework", e);
        }
    }
```

網路最佳做法

Lambda 從快照恢復函數時，無法保證函數在初始化階段建立的連線狀態。在大多數情況下， AWS SDK 建立的網路連線會自動恢復。針對其他連線，建議遵循最佳實務操作。

重新建立網路連線

函數從快照恢復時，請務必重新建立網路連線。建議您在函數處理常式中重新建立網路連線。或者，您可以使用 `afterRestore` 執行階段掛鉤。

請勿使用主機名做為唯一的執行環境識別符

建議您不要使用 `hostname` 將執行環境識別為應用程式中唯一的節點或容器。使用時 SnapStart，會使用單一快照作為多個執行環境的初始狀態，而且所有執行環境都會傳回相同的`hostname`值`InetAddress.getLocalHost()`。如果應用程式需要唯一的執行環境識別或 `hostname` 值，建議您在函數處理常式中產生唯一的 ID。或者，使用 `afterRestore` 執行階段掛鉤來產生唯一的 ID，然後使用這個唯一 ID 做為執行環境的識別符。

避免將連線繫結至固定來源連接埠

建議您避免將網路連線繫結至固定來源連接埠。函數從快照恢復時連線會重新建立，而繫結至固定來源連接埠的網路連線可能會失敗。

避免使用 DNS 快取

Lambda 函數已快取 DNS 回應。如果您搭配使用其他 DNS 快取 SnapStart，則當功能從快照恢復時，您可能會遇到連線逾時的情況。

此`java.util.logging.Logger`類別可間接啟用 JVM DNS 快取。若要覆寫預設設定，請在初始化之前將網路位址 .cache.ttl 設定為 0。 `logger`範例：

```
public class MyHandler {
  // first set TTL property
  static{
   java.security.Security.setProperty("networkaddress.cache.ttl" , "0");
  }
 // then instantiate logger
  var logger = org.apache.logging.log4j.LogManager.getLogger(MyHandler.class);
}
```

若要避免`UnknownHostException`失敗，建議您`networkaddress.cache.negative.ttl`將設定為 0。您可以使用`AWS_LAMBDA_JAVA_NETWORKADDRESS_CACHE_NEGATIVE_TTL=0`環境變數為 Lambda 函數設定此屬性。

停用 JVM DNS 快取並不會停用 Lambda 的受管理 DNS 快取。

Java Lambda 函數自訂設定

本頁說明中 Java 函數的特定設定 AWS Lambda。您可以使用這些設定來自訂 Java 執行期啟動行為。這可以減少整體函數延遲並提高整體函數效能，而無需修改任何程式碼。

章節

JAVA_TOOL_OPTIONS 環境變數

在 Java 中，Lambda 支援 `JAVA_TOOL_OPTIONS` 環境變數，以在 Lambda 中設定其他命令列變數。您可以透過各種方式使用此環境變數，例如自訂分層編譯設定。下一個範例將示範如何針對此使用案例使用 `JAVA_TOOL_OPTIONS` 環境變數。

範例：自訂分層編譯設定

分層編譯是 Java 虛擬機器 (JVM) 的一項功能。您可以使用特定的分層編譯設置來充分利用 JVM 的 just-in-time （ JIT ）編譯器。通常，C1 編譯器針對快速啟動時間進行了最佳化。C2 編譯器針對最佳整體效能進行了最佳化，但也使用更多記憶體，並且需要更長的時間來達成目標。

有 5 個不同等級的分層編譯。在層級 0，JVM 會解譯 Java 位元組程式碼。在層級 4，JVM 會使用 C2 編譯器來分析應用程式啟動期間收集的分析資料。隨著時間的推移，它會監控程式碼使用情況以識別最佳化。

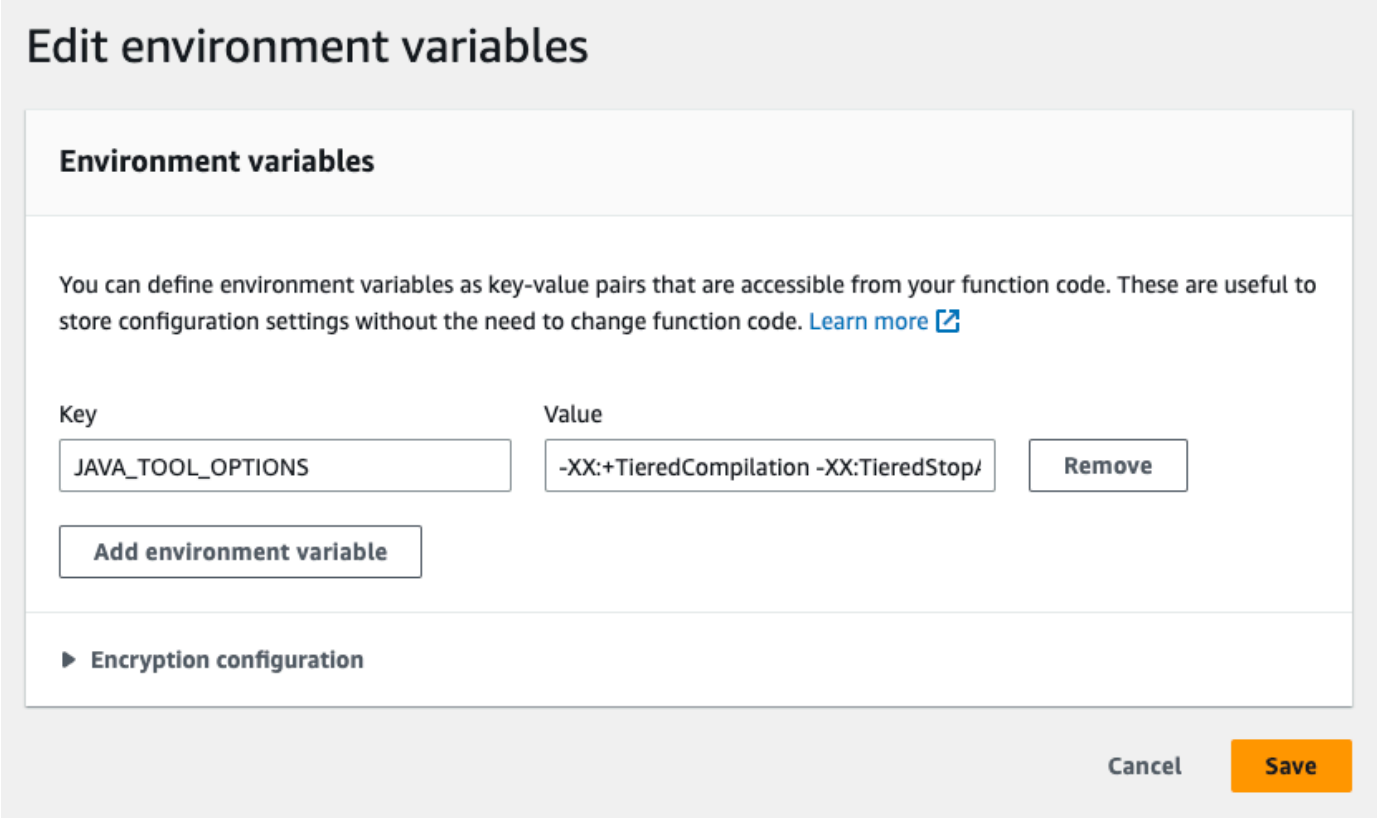
自訂分層編譯等級可協助您減少 Java 函數冷啟動延遲。例如，將分層編譯級別設置為 1，以使 JVM 使用 C1 編譯器。此編譯器可以快速產生最佳化的原生程式碼，但不會產生任何分析資料，也不會使用 C2 編譯器。

在 Java 17 執行期中，分層編譯的 JVM 標誌預設設定為在級別 1 停止。對於 Java 11 執行期及更低版本，可以透過執行以下步驟將分層編譯級別設為 1：

自訂分層編譯設定 (主控台)

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇您要自訂分層編譯的 Java 函數。
3. 選擇組態索引標籤，然後選擇左側選單中的環境變數。
4. 選擇編輯。
5. 選擇 Add environment variable (新增環境變數)。

6. 針對索引鍵，輸入 JAVA_TOOL_OPTIONS。針對值，輸入 -XX:+TieredCompilation -XX:TieredStopAtLevel=1。



7. 選擇儲存。

> **Note**
>
> 您也可以使用 Lambda SnapStart 來緩解冷啟動問題。 SnapStart使用執行環境的快取快照，大幅改善啟動效能。如需 SnapStart 功能、限制和支援區域的詳細資訊，請參閱使用 Lambda 改善啟動效能 SnapStart。

範例：使用 JAVA_TOOL_OPTIONS 自訂 GC 行為

Java 11 執行期使用串行垃圾回收器 (GC) 進行垃圾回收。依預設，Java 17 執行期也會使用串行垃圾回收器。但是，對於 Java 17，您也可以使用 JAVA_TOOL_OPTIONS 環境變數來變更預設垃圾回收器。可以在並行垃圾回收器和 Shenandoah 垃圾回收器之間進行選擇。

例如，如果工作負載使用更多記憶體和多個 CPU，則請考慮使用並行垃圾回收器以獲得更好的效能。可以透過將下列內容附加到 JAVA_TOOL_OPTIONS 環境變數的值來執行此操作：

```
-XX:+UseParallelGC
```

AWS Lambda Java 中的上下文對象

當 Lambda 執行您的函數時，它會將內容物件傳遞至處理常式。此物件提供的方法和各項屬性提供了有關調用、函式以及執行環境的資訊。

內容方法

- `getRemainingTimeInMillis()` - 傳回執行逾時前剩餘的毫秒數。
- `getFunctionName()` – 傳回 Lambda 函數的名稱。
- `getFunctionVersion()` - 傳回函數的版本。
- `getInvokedFunctionArn()` - 傳回用於叫用此函數的 Amazon Resource Name (ARN)。指出叫用者是否指定版本號或別名。
- `getMemoryLimitInMB()` - 傳回分配給函數的記憶體數量。
- `getAwsRequestId()` - 傳回叫用請求的識別符。
- `getLogGroupName()` - 傳回函數的日誌群組。
- `getLogStreamName()` - 傳回函數執行個體的記錄串流。
- `getIdentity()` - (行動應用程式) 傳回已授權請求的 Amazon Cognito 身分的相關資訊。
- `getClientContext()` - (行動應用程式) 傳回用戶端應用程式提供給 Lambda 的用戶端內容。
- `getLogger()` - 傳回函數的 Logger 物件。

下面的例子顯示使用內容物件存取 Lambda 記錄器的函數。

Example Handler.java

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;

import java.util.Map;

// Handler value: example.Handler
public class Handler implements RequestHandler<Map<String,String>, Void>{

  @Override
  public Void handleRequest(Map<String,String> event, Context context)
```

```
  {
    LambdaLogger logger = context.getLogger();
    logger.log("EVENT TYPE: " + event.getClass());
    return null;
  }
}
```

該函數在返回之前記錄傳入事件的類型null。

Example 記錄輸出

```
EVENT TYPE: class java.util.LinkedHashMap
```

內容物件的介面可在 aws-lambda-java-core 程式庫中使用。您可以實作此介面來建立測試的內容類別。下面的例子顯示針對大多數屬性和進行中測試記錄器傳回虛擬值的內容類別。

Example src /測試/爪/示例/.java TestContext

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.CognitoIdentity;
import com.amazonaws.services.lambda.runtime.ClientContext;
import com.amazonaws.services.lambda.runtime.LambdaLogger;

public class TestContext implements Context{

  public TestContext() {}
  public String getAwsRequestId(){
    return new String("495b12a8-xmpl-4eca-8168-160484189f99");
  }
  public String getLogGroupName(){
    return new String("/aws/lambda/my-function");
  }
  public String getLogStreamName(){
    return new String("2020/02/26/[$LATEST]704f8dxmpla04097b9134246b8438f1a");
  }
  public String getFunctionName(){
    return new String("my-function");
  }
  public String getFunctionVersion(){
    return new String("$LATEST");
  }
```

```
  public String getInvokedFunctionArn(){
    return new String("arn:aws:lambda:us-east-2:123456789012:function:my-function");
  }
  public CognitoIdentity getIdentity(){
    return null;
  }
  public ClientContext getClientContext(){
    return null;
  }
  public int getRemainingTimeInMillis(){
    return 300000;
  }
  public int getMemoryLimitInMB(){
    return 512;
  }
  public LambdaLogger getLogger(){
    return new TestLogger();
  }

}
```

如需記錄日誌的詳細資訊，請參閱 AWS Lambda Java 中的函數日誌記錄。

範例應用程式中的內容

本指南的 GitHub 存放庫包含示範如何使用前後關聯物件的範例應用程式。每個範例應用程式都包含可輕鬆部署和清理的指令碼、 AWS Serverless Application Model (AWS SAM) 範本以及支援資源。

以 Java 編寫的範例 Lambda 應用程式

- java17-examples – 一個 Java 函數，示範如何使用 Java 記錄來表示輸入事件資料物件。
- java-basic - 具有單元測試和變數日誌組態的最小 Java 函數集合。
- java-events - Java 函數集合，其中包含如何處理來自各種服務 (例如 Amazon API Gateway、Amazon SQS 和 Amazon Kinesis) 事件的骨架程式碼。這些函數使用最新版 aws-lambda-java-events 程式庫 (3.0.0 及更新版)。這些範例不需要 AWS SDK 作為相依性。
- s3-java - 一種 Java 函數，它處理來自 Amazon S3 的通知事件，並使用 Java Class Library (JCL) 以從上傳的映像檔案建立縮圖。
- 使用 API Gateway 調用 Lambda 函數 - 一個 Java 函數，其可掃描包含員工資訊的 Amazon DynamoDB 資料表。然後，其會使用 Amazon Simple Notification Service 向員工傳送文字訊息，慶祝他們的工作週年紀念日。此範例使用 API Gateway 叫用函數。

AWS Lambda Java 中的函數日誌記錄

AWS Lambda 自動監控 Lambda 函數，並將日誌項目傳送到 Amazon CloudWatch。您的 Lambda 函數隨附一個 CloudWatch 日誌記錄群組和函數每個執行個體的日誌串流。Lambda 執行期環境會將每次調用的詳細資訊和函數程式碼的其他輸出，傳送至日誌串流。如需有關 CloudWatch 記錄檔的詳細資訊，請參閱使用 Amazon CloudWatch 日誌 AWS Lambda。

若要由您的函數程式碼輸出日誌，可以使用 java.lang.System 的方法，或任何能寫入 stdout 或 stderr 的記錄模組。

章節

建立傳回日誌的函數

若要由您的函數程式碼輸出日誌，您可以使用 java.lang.System 的方法，或任何能寫入 `stdout` 或 `stderr` 的記錄模組。在 aws-lambda-java-core 程式庫會提供了一個名為 `LambdaLogger` 的記錄器類別，您可以從內容物件加以存取。記錄器類別支援多行日誌。

下面範例使用由內容物件提供的 `LambdaLogger` 記錄器。

Example Handler.java

```
// Handler value: example.Handler
public class Handler implements RequestHandler<Object, String>{
  Gson gson = new GsonBuilder().setPrettyPrinting().create();
  @Override
  public String handleRequest(Object event, Context context)
```

```
  {
    LambdaLogger logger = context.getLogger();
    String response = new String("SUCCESS");
    // log execution details
    logger.log("ENVIRONMENT VARIABLES: " + gson.toJson(System.getenv()));
    logger.log("CONTEXT: " + gson.toJson(context));
    // process event
    logger.log("EVENT: " + gson.toJson(event));
    return response;
  }
}
```

Example 記錄格式

```
START RequestId: 6bc28136-xmpl-4365-b021-0ce6b2e64ab0 Version: $LATEST
ENVIRONMENT VARIABLES:
{
    "_HANDLER": "example.Handler",
    "AWS_EXECUTION_ENV": "AWS_Lambda_java8",
    "AWS_LAMBDA_FUNCTION_MEMORY_SIZE": "512",
    ...
}
CONTEXT:
{
    "memoryLimit": 512,
    "awsRequestId": "6bc28136-xmpl-4365-b021-0ce6b2e64ab0",
    "functionName": "java-console",
    ...
}
EVENT:
{
  "records": [
    {
      "messageId": "19dd0b57-xmpl-4ac1-bd88-01bbb068cb78",
      "receiptHandle": "MessageReceiptHandle",
      "body": "Hello from SQS!",
       ...
    }
  ]
}
END RequestId: 6bc28136-xmpl-4365-b021-0ce6b2e64ab0
REPORT RequestId: 6bc28136-xmpl-4365-b021-0ce6b2e64ab0 Duration: 198.50 ms Billed
 Duration: 200 ms Memory Size: 512 MB Max Memory Used: 90 MB Init Duration: 524.75 ms
```

Java 執行時間會記錄每次調用的 START、END 和 REPORT 行。報告明細行提供下列詳細資訊：

REPORT 行資料欄位

- RequestId— 呼叫的唯一要求 ID。
- 持續時間 - 函數的處理常式方法處理事件所花費的時間量。
- 計費持續時間 - 調用的計費時間量。
- 記憶體大小 - 分配給函數的記憶體數量。
- 使用的記憶體上限 - 函數所使用的記憶體數量。
- 初始化持續時間 - 對於第一個提供的請求，這是執行期載入函數並在處理常式方法之外執行程式碼所花費的時間量。
- XRAY TraceId — 針對追蹤的要求，則為AWS X-Ray 追蹤識別碼。
- SegmentId— 針對追蹤的請求，X-Ray 區段 ID。
- 已取樣 - 對於追蹤的請求，這是取樣結果。

搭配 Java 使用 Lambda 進階日誌控制項

為了讓您更妥善地控制擷取、處理和使用函數日誌的方式，您可以針對支援的 Java 執行期設定下列記錄選項：

- 日誌格式 - 在純文字和結構化 JSON 格式之間為您的日誌進行選擇
- 日誌級別-對於 JSON 格式的日誌，選擇 Lambda 發送到的日誌的詳細信息級別 CloudWatch，例如錯誤，調試或信息
- 日誌組-選擇您的功能發送日誌的日誌組 CloudWatch

如需這些日誌選項的詳細資訊，以及如何設定函數以使用這些選項的說明，請參閱 the section called “設定 Lambda 函數的進階日誌控制項”。

若要使用日誌格式和日誌層級選項與 Java Lambda 函數搭配使用，請參閱以下各章節中的指引。

搭配 Java 使用結構化 JSON 日誌格式

如果您為函數的日誌格式選取 JSON，Lambda 會以結構化 JSON 形式使用 `LambdaLogger` 類別傳送日誌輸出。每個 JSON 日誌物件都包含至少四個鍵值對，其中包含下列索引鍵：

- `"timestamp"` - 產生日誌訊息的時間

- "level" - 指派給訊息的日誌層級
- "message" - 日誌訊息的內容
- "AWSrequestId" - 進行調用的唯一請求 ID。

視您使用的記錄方法而定，以 JSON 格式擷取之函數的日誌輸出也可以包含其他鍵值對。

若要將您使用 LambdaLogger 記錄器建立的日誌指派一個層級，你需要在你的日誌命令提供一個 LogLevel 引數，如以下的例子。

Example Java 日誌程式碼

```
LambdaLogger logger = context.getLogger();
logger.log("This is a debug log", LogLevel.DEBUG);
```

此範例程式碼所輸出的記錄檔會在 CloudWatch Logs 中擷取，如下所示：

Example JSON 日誌記錄

```
{
    "timestamp":"2023-11-01T00:21:51.358Z",
    "level":"DEBUG",
    "message":"This is a debug log",
    "AWSrequestId":"93f25699-2cbf-4976-8f94-336a0aa98c6f"
}
```

如果您沒有為日誌輸出指派層級，Lambda 會自動為其指派層級 INFO。

如果您的程式碼已經使用另一個記錄程式庫來生成 JSON 結構化日誌，則不需要進行任何更改。Lambda 不會對任何已經進行 JSON 編碼的記錄進行雙重編碼。即使您將函數設定為使用 JSON 記錄格式，記錄輸出也會顯示 CloudWatch 在您定義的 JSON 結構中。

搭配 Java 使用日誌層級篩選

AWS Lambda 為了根據日誌級別過濾應用程序日誌，您的函數必須使用 JSON 格式的日誌。您可以透過兩種方式達成此操作：

- 使用標準 LambdaLogger 建立日誌輸出，並將函數設定為使用 JSON 日誌格式。然後，Lambda 會使用 the section called “搭配 Java 使用結構化 JSON 日誌格式” 中所述 JSON 物件中的「層級」索

引鍵值組篩選您的日誌輸出。若要瞭解如何設定函數的日誌格式，請參閱 the section called “設定 Lambda 函數的進階日誌控制項”。

- 使用其他日誌程式庫或方法，在您的程式碼中建立 JSON 結構化日誌，其中包含定義日誌輸出層級的「層級」索引鍵值組。您可以使用可將 JSON 日誌寫入 `stdout` 或 `stderr` 的任何記錄程式庫。例如，您可以使用 AWS Lambda 或 Log4j2 軟件包從代碼生成 JSON 結構化日誌輸出。如需進一步了解，請參閱 the section called “使用動力工具 AWS Lambda （Java）和結構化日 AWS SAM 誌記錄” 和 the section called “使用 Log4j2 和 SLF4J 進行進階日誌記錄”。

當您將函數設定為使用記錄層級篩選時，您必須從下列選項中選取要 Lambda 傳送至記錄的 CloudWatch 記錄層級：

日誌層級	標準用量
TRACE (大多數詳細資訊)	用於追蹤程式碼執行路徑的最精細資訊
DEBUG	系統偵錯的詳細資訊
INFO	記錄函數正常操作的訊息
WARN	有關可能導致未解決意外行為的潛在錯誤的消息
ERROR	有關阻止程式碼按預期執行的問題的訊息
FATAL (最少詳細資訊)	有關導致應用程式停止運作的嚴重錯誤訊息

若要讓 Lambda 篩選函數的日誌，您還必須在 JSON 日誌輸出中包含 `"timestamp"` 索引鍵值組。必須以有效的 RFC 3339 時間戳記格式指定時間。如果您沒有提供有效的時間戳記，Lambda 會為日誌指派層級 INFO，並為您新增時間戳記。

Lambda 會將所選層級且較低層級的記錄傳送至 CloudWatch。例如，如果您設定 WARN 的日誌層級，Lambda 會傳送相對應於 WARN、ERROR 和 FATAL 層級的日誌檔。

使用 Log4j2 和 SLF4J 進行進階日誌記錄

> **Note**
>
> AWS Lambda 在其受管理的執行階段或基本容器映像中不包含 Log4j2。因此，這些問題不受 CVE-2021-44228、CVE-2021-45046 以及 CVE-2021-45105 中描述的問題的影響。

對於客戶函數包含受影響的 Log4j2 版本的情況，我們已將變更套用至 Lambda Java 受管執行時間和基礎容器映像，這有助於緩解 CVE-2021-44228、CVE-2021-45046 和 CVE-2021-45105 中的問題。由於此變更，使用 Log4J2 的客戶可能會看到一個額外的日誌條目，類似於「`Transforming org/apache/logging/log4j/core/lookup/JndiLookup (java.net.URLClassLoader@...)`」。在 Log4J2 輸出中參考 jndi 映射器的任何日誌字串都將替換為「`Patched JndiLookup::lookup()`」。
除此變更之外，我們強烈建議其函數包含 Log4j2 的所有客戶將 Log4j2 更新至最新版本。具體來說，在其功能中使用 aws-lambda-java-log 4j2 程式庫的客戶應該更新至 1.5.0 版 (或更新版本)，並重新部署其功能。此版本將基礎 Log4j2 公用程式相依性更新為 2.17.0 版 (或更高版本)。更新後的 aws-lambda-java-log 4j2 二進製文件可在 Maven 存儲庫中找到，其源代碼可在 Github 上找到。
最後，請注意，在任何情況下都不應使用與 aws-lambda-java-log4j (v1.0.0 or 1.0.1) 相關的任何程式庫。這些程式庫與 log4j 的 第 1.x 版本相關，該版本已於 2015 年停止使用。這些程式庫不受支援、未受維護、未經修補，且具有已知的安全性漏洞。

若要自訂記錄輸出、在單元測試期間支援記錄，以及記錄 AWS SDK 呼叫，請使用搭配 SLF4J 的 Apache Log4j2。Log4j 是適用於 Java 程式的日誌程式庫，讓您能夠配置日誌級別和使用附加器程式庫。SLF4J 是一個外觀程式庫，可讓您改變使用的程式庫，而無須您的函數程式碼。

若要將請求 ID 新增至函數的日誌中，請使用 aws-lambda-java-log4j2 程式庫中的附加器。

Example src/main/resources/log4j2.xml - 附加器組態

```
<Configuration>
  <Appenders>
    <Lambda name="Lambda" format="${env:AWS_LAMBDA_LOG_FORMAT:-TEXT}">
       <LambdaTextFormat>
         <PatternLayout>
             <pattern>%d{yyyy-MM-dd HH:mm:ss} %X{AWSRequestId} %-5p %c{1} - %m%n </
pattern>
         </PatternLayout>
       </LambdaTextFormat>
       <LambdaJSONFormat>
         <JsonTemplateLayout eventTemplateUri="classpath:LambdaLayout.json" />
       </LambdaJSONFormat>
     </Lambda>
   </Appenders>
   <Loggers>
     <Root level="${env:AWS_LAMBDA_LOG_LEVEL:-INFO}">
```

```
      <AppenderRef ref="Lambda"/>
    </Root>
    <Logger name="software.amazon.awssdk" level="WARN" />
    <Logger name="software.amazon.awssdk.request" level="DEBUG" />
  </Loggers>
</Configuration>
```

您可以透過在 <LambdaTextFormat> 和 <LambdaJSONFormat> 標籤下指定佈局來決定如何將 Log4j2 日誌配置為純文本或 JSON 輸出。

在此範例中，每一行前面都會以文字模式加上日期、時間、請求 ID、日誌層級和類別名稱。在 JSON 模式下<JsonTemplateLayout>，會與 aws-lambda-java-log4j2 程式庫一起隨附的組態搭配使用。

SLF4J 是使用 Java 程式碼進行日誌記錄的外觀程式庫。在您的函數程式碼中，您可以使用 SLF4J 記錄器工廠來擷取帶有日誌層級方法的記錄器，如 info() 和 warn()。在您的建置組態中，您可將日誌記錄程式庫和 SLF4J 轉接器包含在 classpath 中。透過在建置配置中更改程式庫，您可以在不更改函數代碼的情況下更改記錄器類型。需要 SLF4J 才能從 SDK for Java 中擷取日誌。

在下面範例程式碼中，處理常式類別會使用 SLF4J 來擷取記錄器。

Example src/main/java/example/HandlerS3.java – 透過 SLF4J 進行日誌記錄

```
package example;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.S3Event;

import static org.apache.logging.log4j.CloseableThreadContext.put;


public class HandlerS3 implements RequestHandler<S3Event, String>{
    private static final Logger logger = LoggerFactory.getLogger(HandlerS3.class);

    @Override
    public String handleRequest(S3Event event, Context context) {
        for(var record : event.getRecords()) {
            try (var loggingCtx = put("awsRegion", record.getAwsRegion())) {
                loggingCtx.put("eventName", record.getEventName());
```

```
                loggingCtx.put("bucket", record.getS3().getBucket().getName());
                loggingCtx.put("key", record.getS3().getObject().getKey());

                logger.info("Handling s3 event");
            }
        }

        return "Ok";
    }
}
```

此程式碼產生的日誌輸出如以下所示。

Example 記錄格式

```
{
    "timestamp": "2023-11-15T16:56:00.815Z",
    "level": "INFO",
    "message": "Handling s3 event",
    "logger": "example.HandlerS3",
    "AWSRequestId": "0bced576-3936-4e5a-9dcd-db9477b77f97",
    "awsRegion": "eu-south-1",
    "bucket": "java-logging-test-input-bucket",
    "eventName": "ObjectCreated:Put",
    "key": "test-folder/"
}
```

建置組態會使用 Lambda 附加器和 SLF4J 轉接器的執行期相依性，以及 Log4j2 的實作相依性。

Example build.gradle - 日誌記錄相依性

```
dependencies {
    ...
    'com.amazonaws:aws-lambda-java-log4j2:[1.6.0,)',
    'com.amazonaws:aws-lambda-java-events:[3.11.3,)',
    'org.apache.logging.log4j:log4j-layout-template-json:[2.17.1,)',
    'org.apache.logging.log4j:log4j-slf4j2-impl:[2.19.0,)',
    ...
}
```

當您在本機執行程式碼進行測試時，帶有 Lambda 記錄器的內容物件將無法使用，並且 Lambda 附加器沒有可使用的請求 ID。對於範例測試組態，請參閱下節中的範例應用程式。

其他工具與程式庫

適用於 AWS Lambda (Java) 的 Powertools 是開發人員工具組，用來實作無伺服器最佳實務並提高開發人員速度。記錄公用程式提供 Lambda 優化記錄器，其中包含有關所有函數之函數內容的其他資訊，輸出結構為 JSON。使用此公用程式執行下列操作：

- 從 Lambda 內容、冷啟動和 JSON 形式的結構記錄輸出中擷取關鍵欄位
- 在收到指示時記錄 Lambda 調用事件 (預設為停用)
- 透過日誌採樣僅列印調用百分比的所有日誌 (預設為停用)
- 在任何時間點將其他金鑰附加至結構化日誌
- 使用自訂日誌格式化程式 (自帶格式化程式)，以與組織的日誌記錄 RFC 相容的結構輸出日誌。

使用動力工具 AWS Lambda （ Java ）和結構化日 AWS SAM 誌記錄

請按照下面的步驟下載，構建和部署一個示例你好世界 Java 應用程序與集成的 Powertools AWS Lambda （ Java ）一模塊使用 AWS SAM. 此應用程式實作了基本 API 後端，並使用 Powertools 發送日誌、指標和追蹤。其包含 Amazon API Gateway 端點和 Lambda 函數。當您將 GET 請求傳送至 API Gateway 端點時，Lambda 函數會叫用、使用內嵌指標格式將記錄和指標傳送至 CloudWatch，並將追蹤傳送至 AWS X-Ray。該函數會傳回 `hello world` 訊息。

必要條件

若要完成本節中的步驟，您必須執行下列各項：

- Java 11
- AWS CLI 第二版
- AWS SAM CLI 版本 1.75 或更新版本。如果您使用較舊版本的 AWS SAM CLI，請參閱升級 AWS SAM CLI。

部署範例 AWS SAM 應用程式

1. 使用 Hello World Java 範本來初始化應用程式。

```
sam init --app-template hello-world-powertools-java --name sam-app --package-type
 Zip --runtime java11 --no-tracing
```

2. 建置應用程式。

```
cd sam-app && sam build
```

3. 部署應用程式。

```
sam deploy --guided
```

4. 依照螢幕上的提示操作。若要接受互動體驗中提供的預設選項，請按下 Enter。

> **Note**
>
> 因為HelloWorldFunction 可能沒有定義授權，這可以嗎？ ，請務必輸入y。

5. 取得已部署應用程式的 URL：

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. 調用 API 端點：

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

成功的話，您將會看到以下回應：

```
{"message":"hello world"}
```

7. 若要獲取該函數的日誌，請執行 sam 日誌。如需詳細資訊，請參閱《AWS Serverless Application Model 開發人員指南》 中的 使用日誌。

```
sam logs --stack-name sam-app
```

日誌輸出如下：

```
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:34.095000
 INIT_START Runtime Version: java:11.v15    Runtime Version ARN: arn:aws:lambda:eu-
central-1::runtime:0a25e3e7a1cc9ce404bc435eeb2ad358d8fa64338e618d0c224fe509403583ca
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:34.114000
 Picked up JAVA_TOOL_OPTIONS: -XX:+TieredCompilation -XX:TieredStopAtLevel=1
```

```
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:34.793000
 Transforming org/apache/logging/log4j/core/lookup/JndiLookup
 (lambdainternal.CustomerClassLoader@1a6c5a9e)
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:35.252000
 START RequestId: 7fcf1548-d2d4-41cd-a9a8-6ae47c51f765 Version: $LATEST
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:36.531000 {
  "_aws": {
    "Timestamp": 1675416276051,
    "CloudWatchMetrics": [
      {
        "Namespace": "sam-app-powerools-java",
        "Metrics": [
          {
            "Name": "ColdStart",
            "Unit": "Count"
          }
        ],
        "Dimensions": [
          [
            "Service",
            "FunctionName"
          ]
        ]
      }
    ]
  },
  "function_request_id": "7fcf1548-d2d4-41cd-a9a8-6ae47c51f765",
  "traceId":
 "Root=1-63dcd2d1-25f90b9d1c753a783547f4dd;Parent=e29684c1be352ce4;Sampled=1",
  "FunctionName": "sam-app-HelloWorldFunction-y9Iu1FLJJBGD",
  "functionVersion": "$LATEST",
  "ColdStart": 1.0,
  "Service": "service_undefined",
  "logStreamId": "2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81",
  "executionEnvironment": "AWS_Lambda_java11"
}
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:36.974000 Feb
 03, 2023 9:24:36 AM com.amazonaws.xray.AWSXRayRecorder <init>
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:36.993000 Feb
 03, 2023 9:24:36 AM com.amazonaws.xray.config.DaemonConfiguration <init>
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:36.993000
 INFO: Environment variable AWS_XRAY_DAEMON_ADDRESS is set. Emitting to daemon on
 address XXXX.XXXX.XXXX.XXXX:2000.
```

```
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:37.331000
 09:24:37.294 [main] INFO  helloworld.App - {"version":null,"resource":"/
hello","path":"/hello/","httpMethod":"GET","headers":{"Accept":"*/
*","CloudFront-Forwarded-Proto":"https","CloudFront-Is-Desktop-
Viewer":"true","CloudFront-Is-Mobile-Viewer":"false","CloudFront-Is-
SmartTV-Viewer":"false","CloudFront-Is-Tablet-Viewer":"false","CloudFront-
Viewer-ASN":"16509","CloudFront-Viewer-Country":"IE","Host":"XXXX.execute-
api.eu-central-1.amazonaws.com","User-Agent":"curl/7.86.0","Via":"2.0
 f0300a9921a99446a44423d996042050.cloudfront.net (CloudFront)","X-Amz-
Cf-Id":"t9W5ByT11HaY33NM8YioKECn_4eMpNsOMPfEVRczD7T1RdhbtiwV1Q==","X-
Amzn-Trace-Id":"Root=1-63dcd2d1-25f90b9d1c753a783547f4dd","X-Forwarded-
For":"XX.XXX.XXX.XX, XX.XXX.XXX.XX","X-Forwarded-Port":"443","X-
Forwarded-Proto":"https"},"multiValueHeaders":{"Accept":["*/
*"],"CloudFront-Forwarded-Proto":["https"],"CloudFront-Is-Desktop-Viewer":
["true"],"CloudFront-Is-Mobile-Viewer":["false"],"CloudFront-Is-SmartTV-
Viewer":["false"],"CloudFront-Is-Tablet-Viewer":["false"],"CloudFront-Viewer-
ASN":["16509"],"CloudFront-Viewer-Country":["IE"],"Host":["XXXX.execute-
api.eu-central-1.amazonaws.com"],"User-Agent":["curl/7.86.0"],"Via":["2.0
 f0300a9921a99446a44423d996042050.cloudfront.net (CloudFront)"],"X-Amz-
Cf-Id":["t9W5ByT11HaY33NM8YioKECn_4eMpNsOMPfEVRczD7T1RdhbtiwV1Q=="],"X-
Amzn-Trace-Id":["Root=1-63dcd2d1-25f90b9d1c753a783547f4dd"],"X-Forwarded-
For":["XXX, XXX"],"X-Forwarded-Port":["443"],"X-Forwarded-Proto":
["https"]},"queryStringParameters":null,"multiValueQueryStringParameters":null,"pathParamet
{"accountId":"XXX","stage":"Prod","resourceId":"at73a1","requestId":"ba09ecd2-
acf3-40f6-89af-fad32df67597","operationName":null,"identity":
{"cognitoIdentityPoolId":null,"accountId":null,"cognitoIdentityId":null,"caller":null,"apiK
hello","httpMethod":"GET","apiId":"XXX","path":"/Prod/
hello/","authorizer":null},"body":null,"isBase64Encoded":false}
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:37.351000
 09:24:37.351 [main] INFO  helloworld.App - Retrieving https://
checkip.amazonaws.com
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:39.313000 {
  "function_request_id": "7fcf1548-d2d4-41cd-a9a8-6ae47c51f765",
  "traceId":
 "Root=1-63dcd2d1-25f90b9d1c753a783547f4dd;Parent=e29684c1be352ce4;Sampled=1",
  "xray_trace_id": "1-63dcd2d1-25f90b9d1c753a783547f4dd",
  "functionVersion": "$LATEST",
  "Service": "service_undefined",
  "logStreamId": "2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81",
  "executionEnvironment": "AWS_Lambda_java11"
}
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:39.371000 END
 RequestId: 7fcf1548-d2d4-41cd-a9a8-6ae47c51f765
```

```
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:39.371000
 REPORT RequestId: 7fcf1548-d2d4-41cd-a9a8-6ae47c51f765    Duration: 4118.98 ms
 Billed Duration: 4119 ms    Memory Size: 512 MB    Max Memory Used: 152 MB    Init
 Duration: 1155.47 ms
XRAY TraceId: 1-63dcd2d1-25f90b9d1c753a783547f4dd    SegmentId: 3a028fee19b895cb
 Sampled: true
```

8. 這是可透過網際網路存取的公有 API 端點。建議您在測試後刪除端點。

```
sam delete
```

管理日誌保留

當您刪除函數時，不會自動刪除日誌群組。若要避免無限期儲存記錄檔，請刪除記錄群組，或設定保留期間，之後 CloudWatch 會自動刪除記錄檔。若要設定記錄保留，請將下列項目新增至 AWS SAM 範本：

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      # Omitting other properties

  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

使用 Lambda 主控台

您可以在調用 Lambda 函數之後，使用 Lambda 主控台來檢視日誌輸出。

如果可以從內嵌程式碼編輯器測試您的程式碼，您會在執行結果中找到日誌。使用主控台測試功能以調用函數時，您會在詳細資訊區段找到日誌輸出。

使用控 CloudWatch 制台

您可以使用 Amazon 主 CloudWatch 控台來檢視所有 Lambda 函數叫用的日誌。

在 CloudWatch 主控台上檢視記錄檔

1. 在主控台上開啟 [記錄群組] 頁 CloudWatch 面。
2. 選擇您的函數的日誌群組 (/aws/lambda/*your-function-name*)。
3. 選擇日誌串流

每個日誌串流都會對應至函式的執行個體。當您更新 Lambda 函數，以及建立額外執行個體以處理多個並行調用時，便會出現日誌串流。要查找特定調用的日誌，我們建議使用檢測您的函數。 AWS X-Ray X-Ray 會在追蹤內記錄有關請求和日誌串流的詳細資訊。

使用 AWS Command Line Interface （ AWS CLI ）

這 AWS CLI 是一種開放原始碼工具，可讓您使用命令列殼層中的命令與 AWS 服務互動。若要完成本節中的步驟，您必須執行下列各項：

- AWS Command Line Interface (AWS CLI) 第二版
- AWS CLI -快速配置 `aws configure`

您可以透過 AWS CLI，使用 `--log-type` 命令選項來擷取要調用的日誌。其回應將包含 `LogResult` 欄位，內含該次調用的 base64 編碼日誌 (最大達 4 KB)。

Example 擷取日誌 ID

下列範例顯示如何從名稱為 `my-function` 的函數的 `LogResult` 欄位來擷取日誌 ID。

```
aws lambda invoke --function-name my-function out --log-type Tail
```

您應該會看到下列輸出：

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example 解碼日誌

在相同的命令提示中，使用 base64 公用程式來解碼日誌。下列範例顯示如何擷取 my-function 的 base64 編碼日誌。

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

如果您使用的是 AWS CLI 版本 2，則需要此cli-binary-format選項。若要讓此成為預設的設定，請執行 aws configure set cli-binary-format raw-in-base64-out。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

您應該會看到下列輸出：

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

該 base64 公用程式可在 Linux、macOS 和 Ubuntu on Windows 上使用。macOS 使用者可能需要使用 base64 -D。

Example get-logs.sh 指令碼

在相同的命令提示中，使用下列指令碼下載最後五個日誌事件。該指令碼使用 sed 以從輸出檔案移除引述，並休眠 15 秒以使日誌可供使用。輸出包括來自 Lambda 的回應以及來自 get-log-events 命令的輸出。

複製下列程式碼範例的內容，並將您的 Lambda 專案目錄儲存為 get-logs.sh。

如果您使用的是 AWS CLI 版本 2，則需要此cli-binary-format選項。若要讓此成為預設的設定，請執行 aws configure set cli-binary-format raw-in-base64-out。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
```

```
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS 和 Linux (僅限)

在相同的命令提示中，macOS 和 Linux 使用者可能需要執行下列命令，以確保指令碼可執行。

```
chmod -R 755 get-logs.sh
```

Example 擷取最後五個記錄事件

在相同的命令提示中，執行下列指令碼以取得最後五個日誌事件。

```
./get-logs.sh
```

您應該會看到下列輸出：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
```

```
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```


刪除日誌

當您刪除函數時，不會自動刪除日誌群組。若要避免無限期地儲存日誌，請刪除日誌群組，或設定保留期間，系統會在該時間之後自動刪除日誌。

日誌記錄程式碼範例

本指南的 GitHub 儲存庫包含示範如何使用各種記錄設定的範例應用程式。每個範例應用程式都包含可輕鬆部署和清理的指令碼、 AWS SAM 範本和支援資源。

以 Java 編寫的範例 Lambda 應用程式

- java17-examples – 一個 Java 函數，示範如何使用 Java 記錄來表示輸入事件資料物件。
- java-basic - 具有單元測試和變數日誌組態的最小 Java 函數集合。
- java-events - Java 函數集合，其中包含如何處理來自各種服務 (例如 Amazon API Gateway、Amazon SQS 和 Amazon Kinesis) 事件的骨架程式碼。這些函數使用最新版 aws-lambda-java-events 程式庫 (3.0.0 及更新版)。這些範例不需要 AWS SDK 作為相依性。
- s3-java - 一種 Java 函數，它處理來自 Amazon S3 的通知事件，並使用 Java Class Library (JCL) 以從上傳的映像檔案建立縮圖。
- 使用 API Gateway 調用 Lambda 函數 - 一個 Java 函數，其可掃描包含員工資訊的 Amazon DynamoDB 資料表。然後，其會使用 Amazon Simple Notification Service 向員工傳送文字訊息，慶祝他們的工作週年紀念日。此範例使用 API Gateway 調用函數。

java-basic 範例應用程式會顯示支援日誌記錄測試的最小日誌記錄組態。處理常式程式碼會使用內容物件提供的 LambdaLogger 記錄器。對於測試，應用程式會使用實作具有 Log4j2 記錄器的 LambdaLogger 介面的自訂 TestLogger 類別。它使用 SLF4J 作為與 AWS SDK 兼容性的外觀。建置輸出中會排除記錄程式庫，使部署套件不會變太大。

在中檢測 Java 程式碼 AWS Lambda

Lambda 與 AWS X-Ray 整合，可協助您追蹤、偵錯和最佳化 Lambda 應用程式。您可以使用 X-Ray 來追蹤請求，因為它會周遊您應用程式中的資源，其中可能包含 Lambda 函數和其他 AWS 服務。

若要將追蹤資料傳送至 X-Ray，您可以使用以下兩個 SDK 庫之一：

- AWS 適用於 OpenTelemetry (ADOT) 的發行版 — 安全、可生產就緒且 AWS支援的 (OTel) SDK 發行 OpenTelemetry 版。
- 適用於 JAVA 的 AWS X-Ray SDK – 用於生成追蹤資料並將其傳送至 X-Ray 的 SDK。
- 適用於 AWS Lambda (Java) 的 Powertools — 實作無伺服器最佳做法並提高開發人員速度的開發人員工具組。

每個 SDK 均提供將遙測資料傳送至 X-Ray 服務的方法。然後，您可以使用 X-Ray 來檢視、篩選應用程式的效能指標並獲得洞察，從而識別問題和進行最佳化的機會。

> **Important**
>
> 用於 AWS Lambda SDK 的 X-Ray 和 Powertools 是由提供的緊密集成的儀表解決方案的一部分。 AWS ADOT Lambda Layers 是用於追蹤檢測之業界通用標準的一部分，這類檢測一般會收集更多資料，但可能不適用於所有使用案例。您可以使用任一解決方案在 X-Ray 中實作 end-to-end 追蹤。若要深入了解如何在兩者之間做選擇，請參閱在 AWS Distro for OpenTelemetry 和 X-Ray SDK 之間進行選擇。

章節

- 使用動力工具 AWS Lambda （ Java ）和跟 AWS SAM 踪
- 使用動力工具 AWS Lambda （ Java ）和跟 AWS CDK 踪
- 使用 ADOT 來檢測您的 Java 函數
- 使用 X-Ray SDK 來檢測 Java 功能
- 透過 Lambda 主控台來啟用追蹤
- 透過 Lambda API 啟用追蹤
- 使用啟動追蹤 AWS CloudFormation
- 解讀 X-Ray 追蹤
- 將執行時間相依項存放存在層中 (X-Ray SDK)

- 樣本應用程式中的 X-Ray 追蹤 (X-Ray SDK)

使用動力工具 AWS Lambda （Java）和跟 AWS SAM 踪

請按照下面的步驟下載，構建和部署一個示例你好世界 Java 應用程序與集成的 Powertools AWS Lambda （Java）模塊使用 AWS SAM. 此應用程式實作了基本 API 後端，並使用 Powertools 發送日誌、指標和追蹤。其包含 Amazon API Gateway 端點和 Lambda 函數。當您將 GET 請求傳送至 API Gateway 端點時，Lambda 函數會叫用、使用內嵌指標格式將記錄和指標傳送至 CloudWatch，並將追蹤傳送至 AWS X-Ray。該函數會傳回 `hello world` 訊息。

必要條件

若要完成本節中的步驟，您必須執行下列各項：

- Java 11
- AWS CLI 第二版
- AWS SAM CLI 版本 1.75 或更新版本。如果您使用較舊版本的 AWS SAM CLI，請參閱升級 AWS SAM CLI。

部署範例 AWS SAM 應用程式

1. 使用 Hello World Java 範本來初始化應用程式。

```
sam init --app-template hello-world-powertools-java --name sam-app --package-type
 Zip --runtime java11 --no-tracing
```

2. 建置應用程式。

```
cd sam-app && sam build
```

3. 部署應用程式。

```
sam deploy --guided
```

4. 依照螢幕上的提示操作。若要接受互動體驗中提供的預設選項，請按下 `Enter`。

> **Note**
>
> 因為HelloWorldFunction 可能沒有定義授權，這可以嗎？ ，請務必輸入y。

5. 取得已部署應用程式的 URL：

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. 調用 API 端點：

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

成功的話，您將會看到以下回應：

```
{"message":"hello world"}
```

7. 若要取得函數的追蹤，請執行 sam 追蹤。

```
sam traces
```

追蹤輸出如下：

```
New XRay Service Graph
  Start time: 2023-02-03 14:31:48+01:00
  End time: 2023-02-03 14:31:48+01:00
  Reference Id: 0 - (Root) AWS::Lambda - sam-app-HelloWorldFunction-y9Iu1FLJJBGD -
 Edges: []
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 5.587
  Reference Id: 1 - client - sam-app-HelloWorldFunction-y9Iu1FLJJBGD - Edges: [0]
   Summary_statistics:
     - total requests: 0
     - ok count(2XX): 0
     - error count(4XX): 0
     - fault count(5XX): 0
```

```
     - total response time: 0

XRay Event [revision 3] at (2023-02-03T14:31:48.500000) with id
 (1-63dd0cc4-3c869dec72a586875da39777) and duration (5.603s)
 - 5.587s - sam-app-HelloWorldFunction-y9Iu1FLJJBGD [HTTP: 200]
 - 4.053s - sam-app-HelloWorldFunction-y9Iu1FLJJBGD
   - 1.181s - Initialization
   - 4.037s - Invocation
     - 1.981s - ## handleRequest
       - 1.840s - ## getPageContents
   - 0.000s - Overhead
```

8. 這是可透過網際網路存取的公有 API 端點。建議您在測試後刪除端點。

```
sam delete
```

使用動力工具 AWS Lambda （Java）和跟 AWS CDK 踪

請按照下面的步驟下載，構建和部署一個示例你好世界 Java 應用程序與集成的 Powertools AWS Lambda （Java）模塊使用 AWS CDK. 此應用程式實作了基本 API 後端，並使用 Powertools 發送日誌、指標和追蹤。其包含 Amazon API Gateway 端點和 Lambda 函數。當您將 GET 請求傳送至 API Gateway 端點時，Lambda 函數會叫用、使用內嵌指標格式將記錄和指標傳送至 CloudWatch，並將追蹤傳送至 AWS X-Ray。函數會傳回 hello world 訊息。

必要條件

若要完成本節中的步驟，您必須執行下列各項：

- Java 11
- AWS CLI 第二版
- AWS CDK 第二版
- AWS SAM CLI 版本 1.75 或更新版本。如果您使用較舊版本的 AWS SAM CLI，請參閱升級 AWS SAM CLI。

部署範例 AWS CDK 應用程式

1. 為您的新應用程式建立專案目錄。

```
mkdir hello-world
```

```
cd hello-world
```

2. 初始化應用程式。

```
cdk init app --language java
```

3. 使用以下命令來建立 Maven 專案：

```
mkdir app
cd app
mvn archetype:generate -DgroupId=helloworld -DartifactId=Function -
DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

4. 在 hello-world\app\Function 目錄中開啟 pom.xml，並將現有程式碼替換為下面的程式碼，其中包括 Powertools 的相依性和 Maven 外掛程式。

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>helloworld</groupId>
  <artifactId>Function</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Function</name>
  <url>http://maven.apache.org</url>
<properties>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
    <log4j.version>2.17.2</log4j.version>
</properties>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>software.amazon.lambda</groupId>
            <artifactId>powertools-tracing</artifactId>
            <version>1.3.0</version>
```

```
        </dependency>
        <dependency>
            <groupId>software.amazon.lambda</groupId>
            <artifactId>powertools-metrics</artifactId>
            <version>1.3.0</version>
        </dependency>
        <dependency>
            <groupId>software.amazon.lambda</groupId>
            <artifactId>powertools-logging</artifactId>
            <version>1.3.0</version>
        </dependency>
        <dependency>
            <groupId>com.amazonaws</groupId>
            <artifactId>aws-lambda-java-core</artifactId>
            <version>1.2.2</version>
        </dependency>
        <dependency>
            <groupId>com.amazonaws</groupId>
            <artifactId>aws-lambda-java-events</artifactId>
            <version>3.11.1</version>
        </dependency>
  </dependencies>
<build>
    <plugins>
        <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>aspectj-maven-plugin</artifactId>
            <version>1.14.0</version>
            <configuration>
                <source>${maven.compiler.source}</source>
                <target>${maven.compiler.target}</target>
                <complianceLevel>${maven.compiler.target}</complianceLevel>
                <aspectLibraries>
                    <aspectLibrary>
                        <groupId>software.amazon.lambda</groupId>
                        <artifactId>powertools-tracing</artifactId>
                    </aspectLibrary>
                    <aspectLibrary>
                        <groupId>software.amazon.lambda</groupId>
                        <artifactId>powertools-metrics</artifactId>
                    </aspectLibrary>
                    <aspectLibrary>
                        <groupId>software.amazon.lambda</groupId>
                        <artifactId>powertools-logging</artifactId>
```

```
                    </aspectLibrary>
                </aspectLibraries>
            </configuration>
            <executions>
                <execution>
                    <goals>
                        <goal>compile</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
        <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>3.4.1</version>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>shade</goal>
                        </goals>
                        <configuration>
                            <transformers>
                                <transformer

 implementation="com.github.edwgiz.maven_shade_plugin.log4j2_cache_transformer.PluginsCache
                                </transformer>
                            </transformers>
                            <createDependencyReducedPom>false</
createDependencyReducedPom>
                            <finalName>function</finalName>

                        </configuration>
                    </execution>
                </executions>
                <dependencies>
                    <dependency>
                        <groupId>com.github.edwgiz</groupId>
                        <artifactId>maven-shade-plugin.log4j2-cachefile-
transformer</artifactId>
                        <version>2.15</version>
                    </dependency>
                </dependencies>
        </plugin>
```

```
    </plugins>
</build>
</project>
```

5. 建立 hello-world\app\src\main\resource 目錄並為日誌組態建立 log4j.xml。

```
mkdir -p src/main/resource
cd src/main/resource
touch log4j.xml
```

6. 開啟 log4j.xml 並新增以下程式碼。

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration>
    <Appenders>
        <Console name="JsonAppender" target="SYSTEM_OUT">
            <JsonTemplateLayout
 eventTemplateUri="classpath:LambdaJsonLayout.json" />
        </Console>
    </Appenders>
    <Loggers>
        <Logger name="JsonLogger" level="INFO" additivity="false">
            <AppenderRef ref="JsonAppender"/>
        </Logger>
        <Root level="info">
            <AppenderRef ref="JsonAppender"/>
        </Root>
    </Loggers>
</Configuration>
```

7. 從 hello-world\app\Function\src\main\java\helloworld 目錄中開啟 App.java，並將現有程式碼替換為下面的程式碼。這是 Lambda 函數的程式碼。

```
package helloworld;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.URL;
import java.util.HashMap;
import java.util.Map;
import java.util.stream.Collectors;
```

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyRequestEvent;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyResponseEvent;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import software.amazon.lambda.powertools.logging.Logging;
import software.amazon.lambda.powertools.metrics.Metrics;
import software.amazon.lambda.powertools.tracing.CaptureMode;
import software.amazon.lambda.powertools.tracing.Tracing;

import static software.amazon.lambda.powertools.tracing.CaptureMode.*;

/**
 * Handler for requests to Lambda function.
 */
public class App implements RequestHandler<APIGatewayProxyRequestEvent,
 APIGatewayProxyResponseEvent> {
    Logger log = LogManager.getLogger(App.class);


    @Logging(logEvent = true)
    @Tracing(captureMode = DISABLED)
    @Metrics(captureColdStart = true)
    public APIGatewayProxyResponseEvent handleRequest(final
 APIGatewayProxyRequestEvent input, final Context context) {
        Map<String, String> headers = new HashMap<>();
        headers.put("Content-Type", "application/json");
        headers.put("X-Custom-Header", "application/json");

        APIGatewayProxyResponseEvent response = new APIGatewayProxyResponseEvent()
                .withHeaders(headers);
        try {
            final String pageContents = this.getPageContents("https://
checkip.amazonaws.com");
            String output = String.format("{ \"message\": \"hello world\",
 \"location\": \"%s\" }", pageContents);

            return response
                    .withStatusCode(200)
                    .withBody(output);
        } catch (IOException e) {
            return response
                    .withBody("{}")
```

```
                    .withStatusCode(500);
        }
    }
    @Tracing(namespace = "getPageContents")
    private String getPageContents(String address) throws IOException {
        log.info("Retrieving {}", address);
        URL url = new URL(address);
        try (BufferedReader br = new BufferedReader(new
 InputStreamReader(url.openStream()))) {
            return br.lines().collect(Collectors.joining(System.lineSeparator()));
        }
    }
}
```

8. 從 hello-world\src\main\java\com\myorg 目錄中開啟 HelloWorldStack.java，並將現有程式碼替換為下面的程式碼。此程式碼使用 Lambda 建構函式和 ApiGatewayv2 個建構函式來建立 REST API 和 Lambda 函數。

```
package com.myorg;

import software.amazon.awscdk.*;
import software.amazon.awscdk.services.apigatewayv2.alpha.*;
import
 software.amazon.awscdk.services.apigatewayv2.integrations.alpha.HttpLambdaIntegration;
import
 software.amazon.awscdk.services.apigatewayv2.integrations.alpha.HttpLambdaIntegrationProps
import software.amazon.awscdk.services.lambda.Code;
import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.lambda.FunctionProps;
import software.amazon.awscdk.services.lambda.Runtime;
import software.amazon.awscdk.services.lambda.Tracing;
import software.amazon.awscdk.services.logs.RetentionDays;
import software.amazon.awscdk.services.s3.assets.AssetOptions;
import software.constructs.Construct;

import java.util.Arrays;
import java.util.List;

import static java.util.Collections.singletonList;
import static software.amazon.awscdk.BundlingOutput.ARCHIVED;

public class HelloWorldStack extends Stack {
    public HelloWorldStack(final Construct scope, final String id) {
```

```
        this(scope, id, null);
    }

    public HelloWorldStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        List<String> functionPackagingInstructions = Arrays.asList(
                "/bin/sh",
                "-c",
                "cd Function " +
                        "&& mvn clean install " +
                        "&& cp /asset-input/Function/target/function.jar /asset-
output/"
        );
        BundlingOptions.Builder builderOptions = BundlingOptions.builder()
                .command(functionPackagingInstructions)
                .image(Runtime.JAVA_11.getBundlingImage())
                .volumes(singletonList(
                        // Mount local .m2 repo to avoid download all the
 dependencies again inside the container
                        DockerVolume.builder()
                                .hostPath(System.getProperty("user.home") +
 "/.m2/")
                                .containerPath("/root/.m2/")
                                .build()
                ))
                .user("root")
                .outputType(ARCHIVED);

        Function function = new Function(this, "Function", FunctionProps.builder()
                .runtime(Runtime.JAVA_11)
                .code(Code.fromAsset("app", AssetOptions.builder()
                        .bundling(builderOptions
                                .command(functionPackagingInstructions)
                                .build())
                        .build()))
                .handler("helloworld.App::handleRequest")
                .memorySize(1024)
                .tracing(Tracing.ACTIVE)
                .timeout(Duration.seconds(10))
                .logRetention(RetentionDays.ONE_WEEK)
                .build());
```

```
        HttpApi httpApi = new HttpApi(this, "sample-api", HttpApiProps.builder()
                .apiName("sample-api")
                .build());

        httpApi.addRoutes(AddRoutesOptions.builder()
                .path("/")
                .methods(singletonList(HttpMethod.GET))
                .integration(new HttpLambdaIntegration("function", function,
 HttpLambdaIntegrationProps.builder()
                        .payloadFormatVersion(PayloadFormatVersion.VERSION_2_0)
                        .build()))
                .build());

        new CfnOutput(this, "HttpApi", CfnOutputProps.builder()
                .description("Url for Http Api")
                .value(httpApi.getApiEndpoint())
                .build());
    }
}
```

9. 從 hello-world 目錄中開啟 pom.xml，並將現有程式碼替換為下面的程式碼。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
maven.apache.org/xsd/maven-4.0.0.xsd"
         xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.myorg</groupId>
    <artifactId>hello-world</artifactId>
    <version>0.1</version>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <cdk.version>2.70.0</cdk.version>
        <constructs.version>[10.0.0,11.0.0)</constructs.version>
        <junit.version>5.7.1</junit.version>
    </properties>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
```

```
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.1</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>

            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>exec-maven-plugin</artifactId>
                <version>3.0.0</version>
                <configuration>
                    <mainClass>com.myorg.HelloWorldApp</mainClass>
                </configuration>
            </plugin>
        </plugins>
    </build>

    <dependencies>
        <!-- AWS Cloud Development Kit -->
        <dependency>
            <groupId>software.amazon.awscdk</groupId>
            <artifactId>aws-cdk-lib</artifactId>
            <version>${cdk.version}</version>
        </dependency>
        <dependency>
            <groupId>software.constructs</groupId>
            <artifactId>constructs</artifactId>
            <version>${constructs.version}</version>
        </dependency>
        <dependency>
          <groupId>org.junit.jupiter</groupId>
          <artifactId>junit-jupiter</artifactId>
          <version>${junit.version}</version>
          <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>software.amazon.awscdk</groupId>
            <artifactId>apigatewayv2-alpha</artifactId>
            <version>${cdk.version}-alpha.0</version>
        </dependency>
        <dependency>
            <groupId>software.amazon.awscdk</groupId>
```

```
            <artifactId>apigatewayv2-integrations-alpha</artifactId>
            <version>${cdk.version}-alpha.0</version>
        </dependency>
    </dependencies>
</project>
```

10. 確保您位於 hello-world 目錄中並部署您的應用程式。

```
cdk deploy
```

11. 取得已部署應用程式的 URL：

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?OutputKey==`HttpApi`].OutputValue' --output text
```

12. 調用 API 端點：

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

成功的話，您將會看到以下回應：

```
{"message":"hello world"}
```

13. 若要取得函數的追蹤，請執行 sam 追蹤。

```
sam traces
```

追蹤輸出如下：

```
New XRay Service Graph
  Start time: 2023-02-03 14:59:50+00:00
  End time: 2023-02-03 14:59:50+00:00
  Reference Id: 0 - (Root) AWS::Lambda - sam-app-HelloWorldFunction-YBg8yfYtOc9j -
 Edges: [1]
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0.924
```

```
  Reference Id: 1 - AWS::Lambda::Function - sam-app-HelloWorldFunction-YBg8yfYtOc9j
 - Edges: []
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0.016
  Reference Id: 2 - client - sam-app-HelloWorldFunction-YBg8yfYtOc9j - Edges: [0]
   Summary_statistics:
     - total requests: 0
     - ok count(2XX): 0
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0

XRay Event [revision 1] at (2023-02-03T14:59:50.204000) with id
 (1-63dd2166-434a12c22e1307ff2114f299) and duration (0.924s)
 - 0.924s - sam-app-HelloWorldFunction-YBg8yfYtOc9j [HTTP: 200]
 - 0.016s - sam-app-HelloWorldFunction-YBg8yfYtOc9j
   - 0.739s - Initialization
   - 0.016s - Invocation
     - 0.013s - ## lambda_handler
       - 0.000s - ## app.hello
   - 0.000s - Overhead
```

14. 這是可透過網際網路存取的公有 API 端點。建議您在測試後刪除端點。

```
cdk destroy
```

使用 ADOT 來檢測您的 Java 函數

ADOT 提供全受管 Lambda 層，包含使用 OTel SDK 收集遙測資料所需的一切內容。透過取用此層，您可以檢測 Lambda 函數，而無需修改任何函數程式碼。您還可以將層設定為對 OTel 進行自訂初始化。如需詳細資訊，請參閱 ADOT 文件中的針對 Lambda 上的 ADOT 收集器進行自訂組態設定。

對於 Java 執行時間，您可以選擇要取用的兩層：

- AWS 適用於 ADOT Java 的受管 Lambda 層 (自動分析代理程式) — 此層會在啟動時自動轉換您的函數程式碼，以收集追蹤資料。有關如何與 ADOT Java 代理程式一起使用此層的詳細說明，請參閱 AWS ADOT 文件中的 Java OpenTelemetry Lambda Support 發行版（自動檢測代理程式）。

- AWS 適用於 ADOT Java 的受管 Lambda 層 — 此層也提供 Lambda 函數的內建檢測，但需要進行一些手動程式碼變更才能初始化 oTel SDK。有關如何使用此層的詳細說明，請參閱 AWS ADOT 文檔中的 Java OpenTelemetry Lambda Support 發行版。

使用 X-Ray SDK 來檢測 Java 功能

若要記錄函數對應用程式中其他資源和服務呼叫的資料，請將適用於 Java 的 X-Ray 開發套件新增至您的建置組態。下面的示例顯示了一個 Gradle 構建配置，其中包括激活 AWS SDK for Java 2.x 客戶端自動檢測的庫。

Example build.gradle - 追蹤相依性

```
dependencies {
    implementation platform('software.amazon.awssdk:bom:2.16.1')
    implementation platform('com.amazonaws:aws-xray-recorder-sdk-bom:2.11.0')
    ...
    implementation 'com.amazonaws:aws-xray-recorder-sdk-core'
    implementation 'com.amazonaws:aws-xray-recorder-sdk-aws-sdk'
    implementation 'com.amazonaws:aws-xray-recorder-sdk-aws-sdk-v2-instrumentor'
    ...
}
```

新增正確的依賴項並進行必要的程式碼變更後，請透過 Lambda 主控台或 API 在函數的組態中啟用追蹤。

透過 Lambda 主控台來啟用追蹤

若要使用控制台在 Lambda 函數上切換主動追蹤，請按照下列步驟操作：

開啟主動追蹤

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇 組態 ，然後選擇 監控和操作工具 。
4. 選擇 編輯 。
5. 在 X-Ray 下，打開 主動追蹤 。
6. 選擇 儲存 。

透過 Lambda API 啟用追蹤

使用 AWS CLI 或 AWS SDK 在 Lambda 函數上設定追蹤，並使用下列 API 作業：

- UpdateFunction配置
- GetFunction配置
- CreateFunction

下列範例 AWS CLI 命令可在名為 my- function 的函式上啟用主動追蹤。

```
aws lambda update-function-configuration \
--function-name my-function \
--tracing-config Mode=Active
```

追蹤模式是您發布函數版本時版本特定組態的一部分。您無法變更已發佈版本上的追蹤模式。

使用啟動追蹤 AWS CloudFormation

若要啟動 AWS CloudFormation 範本中的`AWS::Lambda::Function`資源追蹤，請使用`TracingConfig`屬性。

Example function-inline.yml - 追蹤組態

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

對於 AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` 資源，請使用`Tracing`屬性。

Example template.yml - 追蹤組態

```
Resources:
  function:
```

```
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

解讀 X-Ray 追蹤

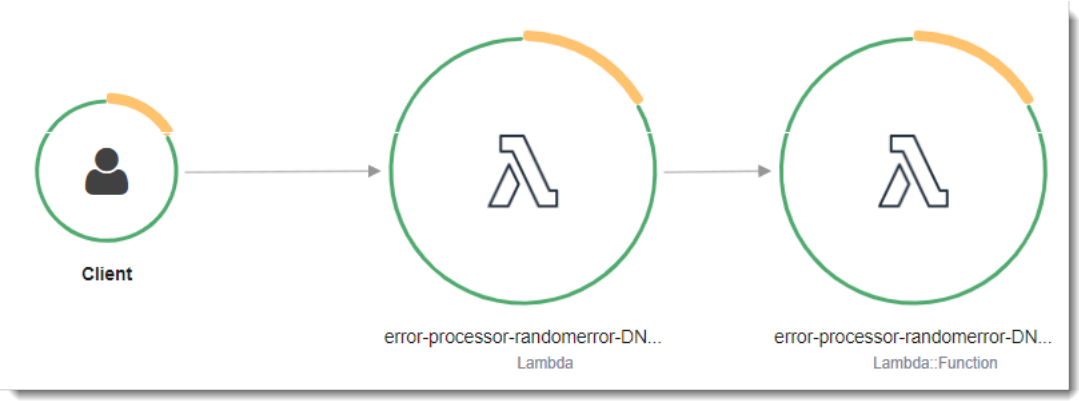
您的函數需要將追蹤資料上傳至 X-Ray 的許可。當您在 Lambda 主控台中啟用追蹤時，Lambda 會將必要的許可新增至函數的執行角色。否則，請將AWSXRayDaemonWriteAccess原則新增至執行角色。

設定主動追蹤之後，您可以透過應用程式來觀察特定請求。X-Ray 服務圖顯示了有關應用程式及其所有元件的資訊。下圖演示了具有兩個功能的應用程序。主要函式會處理事件，有時會傳回錯誤。頂部的第二個函數處理出現在第一個日誌組中的錯誤，並使用 AWS SDK 調用 X-Ray，Amazon 簡單存儲服務（Amazon S3）和亞馬遜 CloudWatch 日誌。

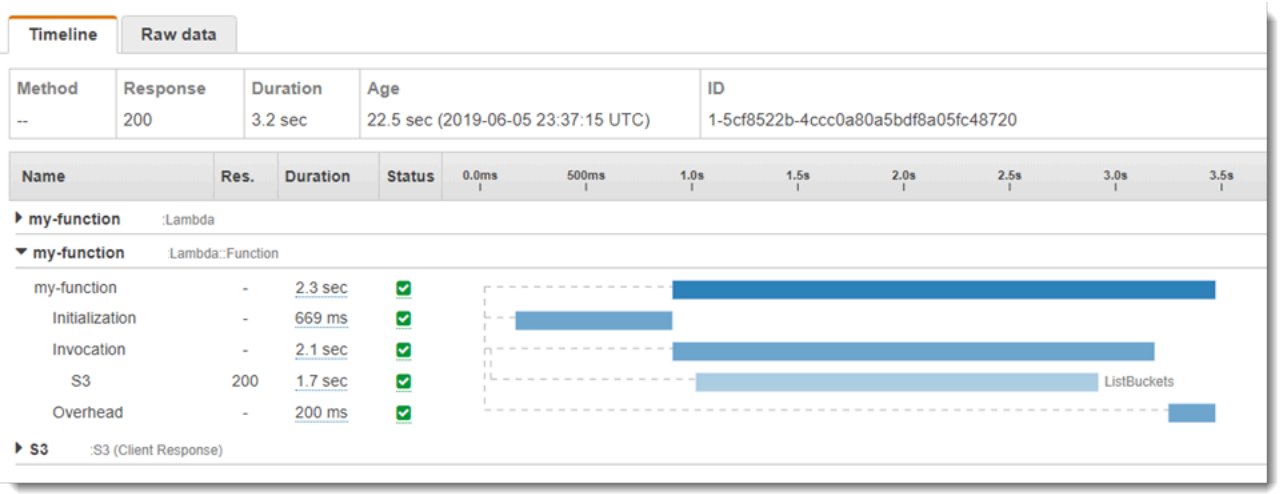
X-Ray 無法追蹤應用程式的所有請求。X-Ray 會套用取樣演算法以確保追蹤的效率，同時仍提供所有請求的代表範本。取樣率為每秒 1 次請求和 5% 的額外請求。

> **Note**
>
> 您無法針對函數設定 X-Ray 取樣率。

在 X-Ray 中，追蹤會記錄一或多個服務所處理之要求的相關資訊。Lambda 會記錄每個追蹤 2 個區段，在服務圖表上建立兩個節點。下列影像會強調顯示這兩個節點：



左側第一個節點代表接收調用請求的 Lambda 服務。第二個節點代表您特定的 Lambda 函數。下列範例顯示了具有這 2 個區段的追蹤。兩者都被命名為我的函數，但一個具有的起源AWS::Lambda，另一個具有的AWS::Lambda::Function起源。如果AWS::Lambda區段顯示錯誤，表示 Lambda 服務發生問題。如果AWS::Lambda::Function區段顯示錯誤，表示您的函數發生問題。



此範例會展開區AWS::Lambda::Function段，以顯示其三個子區段：

- 初始化 - 表示載入函數和執行初始化程式碼所花費的時間。只有函數的每個執行個體所處理的第一個事件會顯示此子區段。
- 調用 – 表示執行處理常式程式碼所花費的時間。
- 額外負荷 - 表示 Lambda 執行期為做好準備以處理下一個事件所花費的時間。

> **Note**
>
> Lambda SnapStart 函數還包括一個 Restore 子區段。Restore 子區段會顯示 Lambda 還原快照、載入執行期 (JVM) 和執行任何 afterRestore 執行期勾點所需的時間。還原快照的程

序可能包括在 MicroVM 以外的活動上花費的時間。此時間在 `Restore` 子區段中報告。您不需要為在 MicroVM 外還原快照所花費的時間付費。

您也可以檢測 HTTP 用戶端、記錄 SQL 查詢，以及建立具有註釋和中繼資料的自訂子區段。如需詳細資訊，請參閱《AWS X-Ray 開發人員指南》中的 適用於 JAVA 的 AWS X-Ray SDK。

定價

作為免費方案的一部分，您可以每月免費使用 X-Ray 追蹤，最多達到一定限制。 AWS 達到閾值後，X-Ray 會收取追蹤儲存及擷取的費用。如需詳細資訊，請參閱 AWS X-Ray 定價。

將執行時間相依項存放存在層中 (X-Ray SDK)

如果您使用 X-Ray SDK 來檢測 AWS SDK 用戶端您的函數程式碼，您的部署套件可能會變得相當大。為了避免每次更新函數程式碼時上傳執行時間相依性，請將 X-Ray SDK 封裝在一個 Lambda 層中。

下面的範例顯示了可存放適用於 Java 的 AWS SDK for Java 和 X-Ray SDK 的 `AWS::Serverless::LayerVersion` 資源。

Example template.yml - 相依性層

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: build/distributions/blank-java.zip
      Tracing: Active
      Layers:
        - !Ref libs
      ...
  libs:
    Type: AWS::Serverless::LayerVersion
    Properties:
      LayerName: blank-java-lib
      Description: Dependencies for the blank-java sample app.
      ContentUri: build/blank-java-lib.zip
      CompatibleRuntimes:
        - java21
```

透過此組態，您只有在變更執行時間相依性時才會更新程式庫層。由於函數部署套件僅含有您的程式碼，因此有助於減少上傳時間。

為相依性建立層需要建置配置變更，才能在部署之前產生層存檔。如需工作範例，請參閱上的 Java 基本範例應用程式。 GitHub

樣本應用程式中的 X-Ray 追蹤 (X-Ray SDK)

本指南的 GitHub 儲存庫包含示範如何使用 X-Ray 追蹤的範例應用程式。每個範例應用程式都包含可輕鬆部署和清理的指令碼、 AWS SAM 範本和支援資源。

以 Java 編寫的範例 Lambda 應用程式

- java17-examples – 一個 Java 函數，示範如何使用 Java 記錄來表示輸入事件資料物件。
- java-basic - 具有單元測試和變數日誌組態的最小 Java 函數集合。
- java-events - Java 函數集合，其中包含如何處理來自各種服務 (例如 Amazon API Gateway、Amazon SQS 和 Amazon Kinesis) 事件的骨架程式碼。這些函數使用最新版 aws-lambda-java-events 程式庫 (3.0.0 及更新版)。這些範例不需要 AWS SDK 作為相依性。
- s3-java - 一種 Java 函數，它處理來自 Amazon S3 的通知事件，並使用 Java Class Library (JCL) 以從上傳的映像檔案建立縮圖。
- 使用 API Gateway 調用 Lambda 函數 - 一個 Java 函數，其可掃描包含員工資訊的 Amazon DynamoDB 資料表。然後，其會使用 Amazon Simple Notification Service 向員工傳送文字訊息，慶祝他們的工作週年紀念日。此範例使用 API Gateway 調用函數。

所有範例應用程式都已啟用 Lambda 函數的主動追蹤功能。例如，`s3-java`應用程式會顯示用 AWS SDK for Java 2.x 戶端的自動檢測、測試的區段管理、自訂子區段，以及使用 Lambda 層來儲存執行階段相依性。

Java 範例應用程式 AWS Lambda

本指南的 GitHub 存放庫提供示範如何在中使用 Java 的範例應用程式 AWS Lambda。每個範例應用程式都包含可輕鬆部署和清理的指令碼、 AWS CloudFormation 範本和支援資源。

以 Java 編寫的範例 Lambda 應用程式

- java17-examples – 一個 Java 函數，示範如何使用 Java 記錄來表示輸入事件資料物件。
- java-basic - 具有單元測試和變數日誌組態的最小 Java 函數集合。
- java-events - Java 函數集合，其中包含如何處理來自各種服務 (例如 Amazon API Gateway、Amazon SQS 和 Amazon Kinesis) 事件的骨架程式碼。這些函數使用最新版 aws-lambda-java-events 程式庫 (3.0.0 及更新版)。這些範例不需要 AWS SDK 作為相依性。
- s3-java - 一種 Java 函數，它處理來自 Amazon S3 的通知事件，並使用 Java Class Library (JCL) 以從上傳的映像檔案建立縮圖。
- 使用 API Gateway 調用 Lambda 函數 - 一個 Java 函數，其可掃描包含員工資訊的 Amazon DynamoDB 資料表。然後，其會使用 Amazon Simple Notification Service 向員工傳送文字訊息，慶祝他們的工作週年紀念日。此範例使用 API Gateway 調用函數。

在 Lambda 上執行熱門 Java 框架

- 彈簧雲函數示例- 來自 Spring 的一個示例，演示瞭如何使用 S pring 雲函數框架來創建 Lambda 函數。 AWS
- 無伺服器 Spring Boot 應用程式示範 — 示範如何在受管理的 Java 執行階段中設定典型的 Spring Boot 應用程式 SnapStart，或是使用自訂執行階段作為 GraalVM 原生映像檔的範例。
- 無伺服器微型應用程式示範 — 示範如何在受管理的 Java 執行階段中使用 Micronaut 的範例 SnapStart，或是使用自訂執行階段的 GraalVM 原生映像檔。請參閱《Micronaut/Lambda 指南》以進一步瞭解。
- 無伺服器 Quarkus 應用程式示範 — 示範如何在受管理的 Java 執行階段中使用 Quarkus SnapStart，或是使用自訂執行階段作為 GraalVM 原生映像檔的範例。若要深入了解，請參閱「夸克斯/Lambda」指南和「夸克斯/指南」。SnapStart

若您不熟悉 Java 中的 Lambda 函數，請從 `java-basic` 範例開始。若要開始使用 Lambda 事件來源，請參閱 `java-events` 範例。這兩個範例集都會顯示 Lambda 的 Java 程式庫、環境變數、SDK 和 AWS SDK 的使用 AWS X-Ray 方式。這些範例需要最少的設定，不到一分鐘的時間就可以從命令列完成部署。

使用 Go 建置 Lambda 函數

Go 的實作方式與其他受管執行期不同。由於 Go 原生編譯為可執行二進製文件，因此它不需要專用的語言運行時。使用僅限作業系統的執行階段 (provided執行階段系列) 將 Go 函數部署至 Lambda。

主題

Go 執行期支援

已取代 Lambda 的 Go 1.x 受管理執行階段。如果您有使用 Go 1.x 執行階段的函數，則必須將函數移轉至provided.al2023或provided.al2。provided.al2023與provided.al2執行階段相比，執行階段具有多項優勢go1.x，包括對 arm64 架構 (AWS Graviton2 處理器) 的支援、較小的二進位檔案，以及稍快的叫用時間。

本次遷移不需要變更任何程式碼。唯一必須做出的變更與建置部署套件的方式以及用來建立函數的執行期有關。如需詳細資訊，請參閱AWS 運算部落格上的 Amazon Linux 2 將 AWS Lambda 函數從 Go1.x 執行階段移轉至自訂執行階段。

僅限作業系統

名稱	識別符	作業系統	取代日期	封鎖函數建立	封鎖函數更新
僅限作業系統的執行期	provided.al2023	Amazon Linux 2023			
僅限作業系統的執行期	provided.al2	Amazon Linux 2			

工具與程式庫

Lambda 為 Go 執行時間提供以下工具和程式庫：

- AWS 適用於圍棋的 AWS SDK：圍棋編程語言的官方 SDK。
- github.com/aws/aws-lambda-go/lambda：針對 Go 實作 Lambda 程式設計模型。此軟件包用於調 AWS Lambda 用您的處理程序。
- github.com/aws/aws-lambda-go/lambdacontext：協助程式用來存取來自內容物件的內容資訊。
- github.com/aws/aws-lambda-go/events：此程式庫提供常用事件來源整合的類型定義。
- github.com/aws/aws-lambda-go/cmd/build-lambda-zip：這個工具可以用來在 Windows 上建立 .zip 檔案封存。

如需詳細資訊，請參閱 < 繼續 > 。 GitHub

Lambda 為 Go 執行時間提供下列範例應用程式：

以 Go 編寫的範例 Lambda 應用程式

- go-al2：傳回公有 IP 地址的「hello world」函數。此應用程式使用 `provided.al2` 自訂執行期。
- 空白移動 — Go 函數，顯示 Lambda 的 Go 程式庫、記錄、環境變數和 SDK 的使用方式 AWS 。此應用程式使用 `go1.x` 執行期。

在圍棋中定義 Lambda 函數處理

Lambda 函數處理常式是您的函數程式碼中處理事件的方法。當有人呼叫您的函數時，Lambda 會執行處理常式方法。函數會執行，直到處理常式傳回回應、結束或逾時為止。

以 Go 撰寫的 Lambda 函數被製作成 Go 可執行檔。在 Lambda 函數程式碼中，需要納入 github.com/aws/aws-lambda-go/lambda 套件，它可針對 Go 實作 Lambda 程式設計模型。此外，您需要實作處理函式程式碼及 `main()` 函式。

Example 前往 Lambda 函數

```
package main

import (
 "context"
 "fmt"
 "github.com/aws/aws-lambda-go/lambda"
)

type MyEvent struct {
 Name string `json:"name"`
}

func HandleRequest(ctx context.Context, event *MyEvent) (*string, error) {
 if event == nil {
  return nil, fmt.Errorf("received nil event")
 }
 message := fmt.Sprintf("Hello %s!", event.Name)
 return &message, nil
}

func main() {
 lambda.Start(HandleRequest)
}
```

以下是此函數的範例輸入：

```
{
  "name": "Jane"
}
```

注意下列事項：

- package main：在 Go 中，包含 `func main()` 的套件一律要命名`main`。
- import：使用此項以納入您的 Lambda 函數所需的程式碼。在此執行個體中，它包含：
  - context：Go 中的 AWS Lambda 內容物件。
  - fmt: Go 格式化物件用於設定您的函式傳回值的格式。
  - github.com/aws/aws-lambda-go/lambda：如先前所述，針對 Go 實作 Lambda 程式設計模型。
- func HandleRequest（ctx 上下文，事件 *MyEvent）（* 字符串，錯誤）：這是您的 Lambda 處理程序的簽名。它是 Lambda 函數的進入點，包含調用函數時執行的邏輯。此外，內含的參數如下所示：
  - ctx context.Context：提供 Lambda 函數叫用的執行時間資訊。`ctx` 是可供宣告的變數，讓您能透過 Go 中的 AWS Lambda 內容物件 利用可用資訊。
  - event * MyEvent: 這是一個名為指向`event`的參數`MyEvent`。它代表 Lambda 函數的輸入。
  - *字串，錯誤：處理常式傳回兩個值。第一個是指向包含 Lambda 函數結果之字串的指標。第二個是錯誤類型，如果沒有錯誤，即為 `nil`，而如果出現錯誤，則包含了標準錯誤資訊。
  - return &message, nil：傳回兩個值。第一個是指向字串訊息的指標，這是使用輸入事件中的 `Name` 欄位所建構的問候語。第二個值：`nil`，表示函數未遇到任何錯誤。
- func main()：執行 Lambda 函數程式碼的進入點。這是必要的。

  將 `lambda.Start(HandleRequest)` 加入到 `func main(){}` 程式碼括弧之間，就會執行 Lambda 函數。依 Go 語言標準，左括弧 { 必須直接放在 `main` 函數簽章的結尾。

命名

provided.al2 和 provided.al2023 執行期

對於在 .zip 部署套件中使用 `provided.al2` 或 `provided.al2023` 執行期的 Go 函數，含有函數程式碼的可執行檔必須命名為 `bootstrap`。如果您使用 .zip 檔案部署函數，則 `bootstrap` 檔案必須位於 .zip 檔案的根層級。對於在容器映像中使用 `provided.al2` 或 `provided.al2023` 執行期的 Go 函數，您可以為可執行檔使用任何名稱。

您可以為處理常式使用任何名稱。若要參考程式碼中的處理常式值，您可以使用 `_HANDLER` 環境變數。

go1.x 執行期

對於使用 `go1.x` 執行期的 Go 函數，可執行檔和處理常式可共用任何名稱。例如，如果您將處理常式的值設為 `Handler`，則 Lambda 會呼叫 `Handler` 可執行檔中的 `main()` 函數。

若要變更 Lambda 主控台中的函數處理常式名稱，請在 Runtime settings (執行時間設定) 窗格中，選擇 Edit (編輯)。

Lambda 函數處理常式使用結構化類型

在上面的範例中，輸入類型是簡單字串。不過，您也可將結構化事件傳入到您的函式處理常式：

```
package main

import (
 "fmt"
 "github.com/aws/aws-lambda-go/lambda"
)

type MyEvent struct {
 Name string `json:"What is your name?"`
 Age  int    `json:"How old are you?"`
}

type MyResponse struct {
 Message string `json:"Answer"`
}

func HandleLambdaEvent(event *MyEvent) (*MyResponse, error) {
 if event == nil {
  return nil, fmt.Errorf("received nil event")
 }
 return &MyResponse{Message: fmt.Sprintf("%s is %d years old!", event.Name,
 event.Age)}, nil
}

func main() {
 lambda.Start(HandleLambdaEvent)
}
```

以下是此函數的範例輸入：

```
{
    "What is your name?": "Jim",
    "How old are you?": 33
}
```

回應如下所示：

```
{
    "Answer": "Jim is 33 years old!"
}
```

若要匯出，事件結構中的欄位名稱必須大寫。如需有關處理 AWS 事件來源事件的詳細資訊，請參閱 aws- lambda-go /事件。

有效的處理常式簽章

當您以 Go 建置 Lambda 函數處理常式時，會有數個選項，但務必堅守以下規則：

- 處理常式必須是一個函式。
- 處理常式採用 0 到 2 之間的引數。如果有兩個引數，第一個引數必須實作 `context.Context`。
- 處理常式傳回 0 到 2 之間的引數。若有單一傳回值，它必須實作 `error`。如果有兩個傳回值，第二個值必須實作 `error`。

下方將列出有效的處理常式簽章。`TIn` 和 `TOut` 皆代表與 encoding/json 標準程式庫相容的類型。如需詳細資訊，請參閱 func Unmarshal 以了解這些類型如何還原序列化。

- `func ()`
- `func () error`
- `func (TIn) error`
- `func () (TOut, error)`
- `func (context.Context) error`
- `func (context.Context, TIn) error`
- `func (context.Context) (TOut, error)`
- `func (context.Context, TIn) (TOut, error)`

使用全域狀態

您可以宣告及修改全域變數，這些變數與您的 Lambda 函數處理常式程式碼是不相關的。此外，您的處理常式可能宣告 `init` 函式，其已在您載入處理常式時執行。這與標準 Go 程序中的行 AWS Lambda 為相同。Lambda 函數的單一執行個體絕對不會同時處理多個事件。

Example Go 函數與全域變數

> **Note**
>
> 此代碼使用 AWS SDK for Go V2。如需詳細資訊，請參閱 AWS SDK for Go V2 入門。

```
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/service/s3"
 "github.com/aws/aws-sdk-go-v2/service/s3/types"
 "log"
)

var invokeCount int
var myObjects []types.Object

func init() {
 // Load the SDK configuration
 cfg, err := config.LoadDefaultConfig(context.TODO())
 if err != nil {
  log.Fatalf("Unable to load SDK config: %v", err)
 }

 // Initialize an S3 client
 svc := s3.NewFromConfig(cfg)

 // Define the bucket name as a variable so we can take its address
 bucketName := "DOC-EXAMPLE-BUCKET"
 input := &s3.ListObjectsV2Input{
  Bucket: &bucketName,
 }
```

```
 // List objects in the bucket
 result, err := svc.ListObjectsV2(context.TODO(), input)
 if err != nil {
  log.Fatalf("Failed to list objects: %v", err)
 }
 myObjects = result.Contents
}

func LambdaHandler(ctx context.Context) (int, error) {
 invokeCount++
 for i, obj := range myObjects {
  log.Printf("object[%d] size: %d key: %s", i, obj.Size, *obj.Key)
 }
 return invokeCount, nil
}

func main() {
 lambda.Start(LambdaHandler)
}
```

Go 中的 AWS Lambda 內容物件

當 Lambda 執行您的函數時，它會將內容物件傳遞至處理常式。此物件提供的方法和各項屬性包含了有關叫用、函式以及執行環境的資訊。

Lambda 內容程式庫提供下列全域變數、方法和屬性。

全域變數

- `FunctionName` – Lambda 函數的名稱。
- `FunctionVersion` – 函數的版本。
- `MemoryLimitInMB` - 分配給函數的記憶體數量。
- `LogGroupName` - 函數的日誌群組。
- `LogStreamName` - 函數執行個體的記錄串流。

內容方法

- `Deadline` - 傳回執行逾時的日期，以 Unix 時間毫秒為單位。

內容屬性

- `InvokedFunctionArn` - 用於叫用此函數的 Amazon Resource Name (ARN)。指出叫用者是否指定版本號或別名。
- `AwsRequestID` - 叫用請求的識別符。
- `Identity` - (行動應用程式) 已授權請求的 Amazon Cognito 身分的相關資訊。
- `ClientContext` - (行動應用程式) 用戶端應用程式提供給 Lambda 的用戶端內容。

存取叫用內容資訊

Lambda 函數有權存取有關其環境和叫用請求的中繼資料。這可以在套件內容進行存取。如果處理常式將 `context.Context` 納為參數，Lambda 即會將函數的相關資訊插入至內容的 `Value` 屬性。請注意，您需要匯入 `lambdacontext` 程式庫以存取 `context.Context` 物件的內容。

```
package main

import (
```

```
        "context"
        "log"
        "github.com/aws/aws-lambda-go/lambda"
        "github.com/aws/aws-lambda-go/lambdacontext"
)

func CognitoHandler(ctx context.Context) {
        lc, _ := lambdacontext.FromContext(ctx)
        log.Print(lc.Identity.CognitoIdentityPoolID)
}

func main() {
        lambda.Start(CognitoHandler)
}
```

在上面的例子中，lc是用於消耗上下文對象捕獲
並log.Print(lc.Identity.CognitoIdentityPoolID)打印該信息的信息的變量，在本例中為CognitoIdentityPool ID。

以下範例說明如何使用內容物件來監控執行 Lambda 函數的時間長度。這可讓您分析效能期望值，並在必要時據以調整您的函式程式碼。

```
package main

import (
        "context"
        "log"
        "time"
        "github.com/aws/aws-lambda-go/lambda"
)

func LongRunningHandler(ctx context.Context) (string, error) {

        deadline, _ := ctx.Deadline()
        deadline = deadline.Add(-100 * time.Millisecond)
        timeoutChannel := time.After(time.Until(deadline))

        for {

                select {

                case <- timeoutChannel:
                        return "Finished before timing out.", nil
```

```
                default:
                        log.Print("hello!")
                        time.Sleep(50 * time.Millisecond)
                }
        }
}

func main() {
        lambda.Start(LongRunningHandler)
}
```

使用 .zip 封存檔部署 Go Lambda 函數

你的 AWS Lambda 函數的代碼由腳本或編譯的程序及其依賴關係組成。使用部署套件將函數程式碼部署到 Lambda。Lambda 支援兩種類型的部署套件：容器映像和 .zip 封存檔。

本頁說明如何建立 .zip 檔案做為 Go 執行階段的部署套件，然後使用 .zip 檔案將函數程式碼部署至 AWS Lambda 使用 AWS Management Console、 AWS Command Line Interface (AWS CLI) 和 AWS Serverless Application Model (AWS SAM)。

請注意，Lambda 使用 POSIX 檔案許可，因此在建立 .zip 封存檔之前，您可能需要設定部署套件資料夾的許可。

章節

在 macOS 和 Linux 上建立 .zip 檔案

下列步驟會說明如何使用 `go build` 命令編譯可執行檔，以及如何為 Lambda 建立 .zip 檔案部署套件。在編譯程式碼之前，請確定您已從中安裝了 lambda 套件 GitHub。此模組會讓您實作執行期界面，管理 Lambda 與函數程式碼之間的互動。若要下載此程式庫，請執行下列命令。

```
go get github.com/aws/aws-lambda-go/lambda
```

如果您的函數使用 AWS SDK for Go，請下載一組標準的 SDK 模組，以及應用程式所需的任何 AWS 服務 API 用戶端。若要瞭解如何安裝適用 SDK for Go，請參閱 AWS SDK for Go V2 入門。

使用提供的運行時系列

Go 的實作方式與其他受管執行期不同。由於 Go 原生編譯為可執行二進製文件，因此它不需要專用的語言運行時。使用僅限作業系統的執行階段 (`provided`執行階段系列) 將 Go 函數部署至 Lambda。

建立 .zip 部署套件 (macOS/Linux) 的方式

1. 在含有應用程式 `main.go` 檔案的專案目錄中編譯可執行檔。注意下列事項：
   - 可執行檔必須命名為 `bootstrap`。如需詳細資訊，請參閱 命名。

- 設定您的目標 指令集架構。僅限作業系統的執行階段同時支援 arm64 和 x86_64。
- 您可以使用選用 `lambda.norpc` 標籤，來排除 lambda 程式庫的遠端程序呼叫 (RPC) 元件。只有在使用已取代的 Go 1.x 執行階段時，才需要 RPC 元件。排除 RPC 會縮減部署套件的大小。

若是 arm64 架構：

```
GOOS=linux GOARCH=arm64 go build -tags lambda.norpc -o bootstrap main.go
```

若是 x86_64 架構：

```
GOOS=linux GOARCH=amd64 go build -tags lambda.norpc -o bootstrap main.go
```

2. (可選) 您可能需要用 Linux 上的 `CGO_ENABLED=0` 設定編譯套件：

```
GOOS=linux GOARCH=arm64 CGO_ENABLED=0 go build -o bootstrap -tags lambda.norpc
 main.go
```

此命令為標準 C 程式庫 (`libc`) 版本建立了一個穩定的二進位套件，這在 Lambda 和其他設備上可能不同。

3. 透過將可執行檔封裝在 .zip 檔案中建立部署套件。

```
zip myFunction.zip bootstrap
```

> **Note**
>
> `bootstrap` 檔案必須位於 .zip 檔案的根層級。

4. 建立函數。注意下列事項：

- 二進位檔必須命名為 `bootstrap`，但處理常式名稱可以是任何名稱。如需詳細資訊，請參閱 命名。
- 如果您使用的是 arm64，才需要選擇 `--architectures` 選項。預設值為 x86_64。
- 針對 `--role`，請指定 執行角色 的 Amazon Resource Name (ARN)。

```
aws lambda create-function --function-name myFunction \
--runtime provided.al2023 --handler bootstrap \
```

```
--architectures arm64 \
--role arn:aws:iam::111122223333:role/lambda-ex \
--zip-file fileb://myFunction.zip
```

在 Windows 上建立 .zip 檔案

下列步驟顯示如何從下載 Windows build-lambda-zip工具 GitHub、編譯可執行檔，以及建立 .zip 部署套件。

> **Note**
>
> 如果尚未這樣做，您必須安裝 git，然後將 `git` 可執行檔新增到 Windows %PATH% 環境變數。

在編譯代碼之前，請確保您已從中安裝了 lambda 庫 GitHub。若要下載此程式庫，請執行下列命令。

```
go get github.com/aws/aws-lambda-go/lambda
```

如果您的函數使用 AWS SDK for Go，請下載一組標準的 SDK 模組，以及應用程式所需的任何 AWS 服務 API 用戶端。若要瞭解如何安裝適用 SDK for Go，請參閱 AWS SDK for Go V2 入門。

使用提供的運行時系列

Go 的實作方式與其他受管執行期不同。由於 Go 原生編譯為可執行二進製文件，因此它不需要專用的語言運行時。使用僅限作業系統的執行階段 (`provided`執行階段系列) 將 Go 函數部署至 Lambda。

建立 .zip 部署套件的方式 (Windows)

1. 從下載build-lambda-zip工具 GitHub。

   ```
   go install github.com/aws/aws-lambda-go/cmd/build-lambda-zip@latest
   ```

2. 使用您的 `GOPATH` 工具來建立 .zip 檔案。如果您已預設安裝 Go，則該工具通常位於 `%USERPROFILE%\Go\bin` 中。否則，導覽至您安裝 Go 執行期之處，然後執行下列其中一個動作：

   cmd.exe

   在 cmd.exe 中執行下列其中一項 (視您的目標 指令集架構 而定)。僅限作業系統的執行階段同時支援 arm64 和 x86_64。

您可以使用選用 `lambda.norpc` 標籤，來排除 lambda 程式庫的遠端程序呼叫 (RPC) 元件。只有在使用已取代的 Go 1.x 執行階段時，才需要 RPC 元件。排除 RPC 會縮減部署套件的大小。

Example - 適用 x86_64 架構

```
set GOOS=linux
set GOARCH=amd64
set CGO_ENABLED=0
go build -tags lambda.norpc -o bootstrap main.go
%USERPROFILE%\Go\bin\build-lambda-zip.exe -o myFunction.zip bootstrap
```

Example - 適用 arm64 架構

```
set GOOS=linux
set GOARCH=arm64
set CGO_ENABLED=0
go build -tags lambda.norpc -o bootstrap main.go
%USERPROFILE%\Go\bin\build-lambda-zip.exe -o myFunction.zip bootstrap
```

PowerShell

在中 PowerShell，根據您的目標指令集架構，執行下列其中一個動作。僅限作業系統的執行階段同時支援 arm64 和 x86_64。

您可以使用選用 `lambda.norpc` 標籤，來排除 lambda 程式庫的遠端程序呼叫 (RPC) 元件。只有在使用已取代的 Go 1.x 執行階段時，才需要 RPC 元件。排除 RPC 會縮減部署套件的大小。

若是 x86_64 架構：

```
$env:GOOS = "linux"
$env:GOARCH = "amd64"
$env:CGO_ENABLED = "0"
go build -tags lambda.norpc -o bootstrap main.go
~\Go\Bin\build-lambda-zip.exe -o myFunction.zip bootstrap
```

若是 arm64 架構：

```
$env:GOOS = "linux"
$env:GOARCH = "arm64"
$env:CGO_ENABLED = "0"
go build -tags lambda.norpc -o bootstrap main.go
~\Go\Bin\build-lambda-zip.exe -o myFunction.zip bootstrap
```

3. 建立函數。注意下列事項：

   - 二進位檔必須命名為 `bootstrap`，但處理常式名稱可以是任何名稱。如需詳細資訊，請參閱 命名。
   - 如果您使用的是 arm64，才需要選擇 `--architectures` 選項。預設值為 x86_64。
   - 針對 `--role`，請指定 執行角色 的 Amazon Resource Name (ARN)。

```
aws lambda create-function --function-name myFunction \
--runtime provided.al2023 --handler bootstrap \
--architectures arm64 \
--role arn:aws:iam::111122223333:role/lambda-ex \
--zip-file fileb://myFunction.zip
```

使用 .zip 檔案建立及更新 Go Lambda 函數

建立 .zip 部署套件後，您可以使用該套件建立新的 Lambda 函數或更新現有函數。您可以使用 Lambda 主控台、和 Lambda API 來部署您的 .zip 套件。 AWS Command Line Interface您也可以使用 AWS Serverless Application Model (AWS SAM) 和 AWS CloudFormation 建立並更新 Lambda 函數。

Lambda 的 .zip 部署套件大小上限為 250 MB (解壓縮)。請注意，此限制適用於您上傳的所有檔案 (包括任何 Lambda 層) 的大小總和。

Lambda 執行時間需有許可才能讀取部署套裝服務中的檔案。在 Linux 權限八進制標記法中，Lambda 需要 644 個權限才能用於不可執行的檔案 (rw-r--r--) 和 755 個權限 () 用於目錄和可執行檔。rwxr-xr-x

在 Linux 和 MacOS 中，使用 chmod 命令變更部署套件中檔案和目錄的檔案許可。例如，若要提供可執行檔正確的許可，請執行下列命令。

```
chmod 755 <filepath>
```

若要在 Windows 中變更檔案許可，請參閱 Microsoft Windows 文件的 Set, View, Change, or Remove Permissions on an Object。

透過主控台使用 .zip 檔案建立及更新函數

若要建立新函數，您必須先在主控台中建立函數，然後上傳您的 .zip 封存檔。若要更新現有函數，請開啟函數的頁面，然後按照同樣的程序新增更新後的 .zip 檔案。

如果您的 .zip 檔案小於 50 MB，您可以透過直接從本機電腦上傳檔案來建立或更新函數。若 .zip 檔案大於 50 MB，您必須先將套件上傳至 Amazon S3 儲存貯體。如需有關如何使用將檔案上傳到 Amazon S3 儲存貯體的指示 AWS Management Console，請參閱開始使用 Amazon S3。若要使用上載檔案 AWS CLI，請參閱《使用指南》中的AWS CLI 〈移動物件〉。

> **Note**
>
> 您無法轉換現有的容器映像函數以使用 .zip 封存檔。您必須建立新的函數。

若要建立新的函數 (主控台)

1. 開啟 Lambda 主控台的函數頁面，然後選擇建立函數。
2. 選擇 Author from scratch (從頭開始撰寫)。
3. 在基本資訊下，請執行下列動作：
   a. 在函數名稱中輸入函數名稱。
   b. 對於 Runtime (執行時間)，選擇 `provided.al2023`。
4. (選用) 在 許可 下，展開 變更預設執行角色 。您可建立新的執行角色，或使用現有的角色。
5. 選擇建立函數。Lambda 會使用您選擇的執行期建立一個基本的「Hello world」函數。

若要從本機電腦上傳 .zip 封存檔 (主控台)

1. 在 Lambda 主控台的函數頁面中選擇要上傳 .zip 檔案的函數。
2. 選取程式碼索引標籤。
3. 在程式碼來源窗格中選擇上傳來源。
4. 選擇 .zip 檔案。
5. 若要上傳 .zip 檔案，請執行下列操作：

a. 選擇上傳，然後在檔案選擇器中選取您的 .zip 檔案。

b. 選擇 Open (開啟)。

c. 選擇儲存。

若要從 Amazon S3 儲存貯體上傳 .zip 封存檔 (控制台)

1. 在 Lambda 主控台的函數頁面中選擇要上傳新 .zip 檔案的函數。
2. 選取程式碼索引標籤。
3. 在程式碼來源窗格中選擇上傳來源。
4. 選擇 Amazon S3 位置。
5. 貼上 .zip 檔案的 Amazon S3 連結 URL，然後選擇儲存。

使用 .zip 檔案建立和更新函數 AWS CLI

您可以使用 AWS CLI 建立新函數，或使用 .zip 檔案更新現有函數。使用建立函數和update-function-code命令來部署您的 .zip 套件。如果您的 .zip 檔案小於 50 MB，則可以從本機建置電腦的檔案位置上傳 .zip 套件。若檔案較大，則必須先從 Amazon S3 儲存貯體上傳 .zip 套件。如需有關如何使用將檔案上傳到 Amazon S3 儲存貯體的指示 AWS CLI，請參閱使用AWS CLI 者指南中的移動物件。

> **Note**
>
> 如果您使用從 Amazon S3 儲存貯體上傳 .zip 檔案 AWS CLI，則該儲存貯體必須與您的函數位於 AWS 區域 相同的位置。

若要使用 .zip 檔案與建立新函數 AWS CLI，您必須指定下列項目：

- 函數名稱 (`--function-name`)
- 函數的執行期 (`--runtime`)
- 函數執行角色的 Amazon Resource Name (ARN) (`--role`)
- 函數程式碼中處理常式方法的名稱 (`--handler`)

您也必須指定 .zip 檔案的位置。如果您的 .zip 檔案位於本機建置電腦上的資料夾中，請使用 `--zip-file` 選項來指定檔案路徑，如下列範例命令所示。

```
aws lambda create-function --function-name myFunction \
--runtime provided.al2023 --handler bootstrap \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--zip-file fileb://myFunction.zip
```

若要在 Amazon S3 儲存貯體中指定 .zip 檔案的位置，請使用如下列範例命令所示的 `--code` 選項。您只需針對版本控制的物件使用 `S3ObjectVersion` 參數。

```
aws lambda create-function --function-name myFunction \
--runtime provided.al2023 --handler bootstrap \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--code S3Bucket=DOC-EXAMPLE-BUCKET,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion
```

若要使用 CLI 更新現有函數，您可以使用 `--function-name` 參數指定函數的名稱。您也必須指定要用來更新函數程式碼的 .zip 檔案的位置。如果您的 .zip 檔案位於本機建置電腦上的資料夾中，請使用 `--zip-file` 選項來指定檔案路徑，如下列範例命令所示。

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

若要在 Amazon S3 儲存貯體中指定 .zip 檔案的位置，請使用如下列範例命令所示的 `--s3-bucket` 和 `--s3-key` 選項。您只需針對版本控制的物件使用 `--s3-object-version` 參數。

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket DOC-EXAMPLE-BUCKET --s3-key myFileName.zip --s3-object-version myObject
 Version
```

透過 Lambda API 使用 .zip 檔案建立及更新函數

若要使用 .zip 封存檔建立及更新函數，請使用下列 API 操作：

- CreateFunction
- UpdateFunctionCode

使用 .zip 文件創建和更新函數 AWS SAM

AWS Serverless Application Model (AWS SAM) 是一個工具組，可協助簡化在 AWS上建置和執行無伺服器應用程式的程序。您可以在 YAML 或 JSON 範本中定義應用程式的資源，並使用 AWS SAM 命令列介面 (AWS SAM CLI) 來建置、封裝及部署應用程式。當您從 AWS SAM 範本建立 Lambda 函數

時， AWS SAM 會使用函數程式碼和您指定的任何相依性，自動建立 .zip 部署套件或容器映像檔。若要進一步了解如 AWS SAM 何使用建置和部署 Lambda 函數，請參閱開AWS Serverless Application Model發人員指南 AWS SAM中的入門使用。

您也可以使用現有的 .zip 檔案封存 AWS SAM 來建立 Lambda 函數。若要使用建立 Lambda 函數 AWS SAM，您可以將 .zip 檔案儲存在 Amazon S3 儲存貯體或建置機器的本機資料夾中。如需有關如何使用將檔案上傳到 Amazon S3 儲存貯體的指示 AWS CLI，請參閱使用AWS CLI 者指南中的移動物件。

在 AWS SAM 範本中，`AWS::Serverless::Function`資源會指定您的 Lambda 函數。在本資源中設定下列屬性，以使用 .zip 封存檔建立函數：

- `PackageType`：設定為 `Zip`
- `CodeUri`-設定為函數程式碼的 Amazon S3 URI、本機資料夾的路徑或FunctionCode物件
- `Runtime`：設定為所選執行期

使用時 AWS SAM，如果您的 .zip 檔案大於 50MB，則不需要先將其上傳到 Amazon S3 儲存貯體。AWS SAM 可以從本地構建機器上的位置上傳 .zip 軟件包，最大允許大小為 250MB（解壓縮）。

若要進一步瞭解如何使用 .zip 檔案部署函數 AWS SAM，請參閱AWS SAM 開發人員指南AWS::Serverless::Function中的。

示例：使用 AWS SAM 提供的 .al2023 構建 Go 函數

1. 建立具有下列屬性的 AWS SAM 範本：

   - BuildMethod：指定應用程式的編譯器。請使用 `go1.x`。
   - Runtime：使用 `provided.al2023`。
   - CodeUri：輸入代碼的路徑。
   - Architectures：為 arm64 架構使用 `[arm64]`。若是 x86_64 指令集架構，請使用 `[amd64]` 或移除 `Architectures` 屬性。

   Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
```

```
    Metadata:
      BuildMethod: go1.x
    Properties:
      CodeUri: hello-world/ # folder where your main program resides
      Handler: bootstrap
      Runtime: provided.al2023
      Architectures: [arm64]
```

2. 使用 sam build 命令來編譯可執行檔。

```
sam build
```

3. 使用 sam deploy 命令將函數部署到 Lambda。

```
sam deploy --guided
```

使用 .zip 文件創建和更新函數 AWS CloudFormation

您可以使 AWS CloudFormation 用 .zip 檔案封存來建立 Lambda 函數。若要使用 .zip 檔案建立 Lambda 函數，您必須先將檔案上傳至 Amazon S3 儲存貯體。如需有關如何使用將檔案上傳到 Amazon S3 儲存貯體的指示 AWS CLI，請參閱使用AWS CLI 者指南中的移動物件。

在 AWS CloudFormation 範本中，`AWS::Lambda::Function`資源會指定您的 Lambda 函數。在本資源中設定下列屬性，以使用 .zip 封存檔建立函數：

- `PackageType`：設定為 `Zip`
- `Code`：在 `S3Bucket` 和 `S3Key` 欄位中輸入 Amazon S3 儲存貯體名稱和 .zip 檔案名稱。
- `Runtime`：設定為所選執行期

AWS CloudFormation 產生的 .zip 檔案不能超過 4MB。若要進一步瞭解有關使用 .zip 檔案部署函數的更多資訊 AWS CloudFormation，請參閱使用AWS CloudFormation 者指南AWS::Lambda::Function中的。

為相依項建立 Go 層

> **Note**
>
> 以 Go 等編譯語言函數使用層，可能不會具有與使用 Python 等轉譯語言一樣的好處。由於 Go 是編譯語言，因此您的函數仍須在 init 階段，將所有共用組件手動載入記憶體中，這可能會增

加冷啟動時間。相反地，建議您在編譯時加入任何共用程式碼，利用任何內建的編譯器最佳化功能。

本節中的指示說明如何在層中包含相依項。

Lambda 會自動偵測 `/opt/lib` 目錄中的任何程式庫，以及 `/opt/bin` 目錄中的任何二進位檔案。若要確保 Lambda 正確找到您的層內容，請建立層，結構如下：

```
custom-layer.zip
# lib
    | lib_1
    | lib_2
# bin
    | bin_1
    | bin_2
```

封裝層之後，請參閱the section called “建立和刪除層”及the section called “新增層”，完成層設定。

使用容器映像來部署 Go Lambda 函數

有兩種方法可以為 Go Lambda 函數構建容器映像：

- 使用 AWS 僅限作業系統的基本影像

  Go 的實作方式與其他受管執行期不同。由於 Go 原生編譯為可執行二進製文件，因此它不需要專用的語言運行時。使用僅限作業系統的基本映像檔，為 Lambda 建立 Go 映像檔。若要使映像檔與 Lambda 相容，您必須在映像中加入 `aws-lambda-go/lambda` 套件。

- 使用非AWS 基本圖像

  您可以使用其他容器登錄檔中的替代基礎映像 (例如 Alpine Linux 或 Debian)。您也可以使用組織建立的自訂映像。若要使映像檔與 Lambda 相容，您必須在映像中加入 `aws-lambda-go/lambda` 套件。

> **Tip**
>
> 若要縮短 Lambda 容器函數變成作用中狀態所需的時間，請參閱 Docker 文件中的使用多階段建置。若要建置有效率的容器映像，請遵循撰寫 Dockerfiles 的最佳實務。

本頁面會說明如何為 Lambda 建置、測試和部署容器映像。

AWS 用於部署 Go 功能的基本映像

Go 的實作方式與其他受管執行期不同。由於 Go 原生編譯為可執行二進製文件，因此它不需要專用的語言運行時。使用僅限作業系統的基礎映像檔，將 Go 函數部署至 Lambda。

僅限作業系統

名稱	識別符	作業系統	取代日期	封鎖函數建立	封鎖函數更新
僅限作業系統的執行期	`provided.al2023`	Amazon Linux 2023			
僅限作業系統的執行期	`provided.al2`	Amazon Linux 2			

Amazon Elastic Container Registry 公有資源庫：gallery.ecr.aws/lambda/provided

Go 執行期介面用戶端

此 `aws-lambda-go/lambda` 套件包含執行期介面實作。如需在映像中使用 `aws-lambda-go/lambda` 的範例，請參閱 使用 AWS 僅限作業系統的基本影像 或 使用非AWS 基本圖像。

使用 AWS 僅限作業系統的基本影像

Go 的實作方式與其他受管執行期不同。由於 Go 原生編譯為可執行二進製文件，因此它不需要專用的語言運行時。使用僅限 OS 的基本映像為 Go 函數構建容器映像。

標籤	執行期	作業系統	Dockerfile	棄用
al2023	僅限作業系統的執行期	Amazon Linux 2023	僅適用於 OS 的運行時的碼頭文件 GitHub	
al2	僅限作業系統的執行期	Amazon Linux 2	僅適用於 OS 的運行時的碼頭文件 GitHub	

如需有關這類基礎映像的詳細資訊，請參閱 Amazon ECR 公有庫中 提供的內容。

您必須將 aws-lambda-go/lambda 套件加入 Go 處理常式。此套件會實作 Go 的程式設計模型 (包括執行期介面)。

必要條件

若要完成本節中的步驟，您必須執行下列各項：

- Go
- Docker
- AWS Command Line Interface (AWS CLI) 第二版

從 provided.al2023 基礎映像建立映像

使用 **`provided.al2023`** 基礎映像建置和部署 Go 函數

1. 建立專案的目錄，然後切換至該目錄。

```
mkdir hello
cd hello
```

2. 初始化新的 Go 模組。

```
go mod init example.com/hello-world
```

3. 將 lambda 程式庫新增為新模組的相依項。

```
go get github.com/aws/aws-lambda-go/lambda
```

4. 建立名為 `main.go` 的檔案，然後在文字編輯器中開啟。這是 Lambda 函數的程式碼。可以使用以下範本程式碼進行測試，也可以將其替換為您自己的程式碼。

```
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, event events.APIGatewayProxyRequest)
 (events.APIGatewayProxyResponse, error) {
 response := events.APIGatewayProxyResponse{
  StatusCode: 200,
  Body:       "\"Hello from Lambda!\"",
 }
 return response, nil
}

func main() {
 lambda.Start(handler)
}
```

5. 使用文字編輯器在專案目錄中建立 Dockerfile。下列範例 Dockerfile 使用多階段建置。這可讓您在每個步驟中使用不同的基礎映像。您可以使用一個映像 (例如 Go 基礎映像檔) 來編譯程式碼並建置可執行二進位檔案。然後，可以在最終的 FROM 陳述式中使用不同的映像 (例如 `provided.al2023`) 來定義您部署到 Lambda 的映像。建置程序與最終部署映像是分開的，因此最終映像僅包含執行應用程式所需的檔案。

您可以使用選用 `lambda.norpc` 標籤，來排除 lambda 程式庫的遠端程序呼叫 (RPC) 元件。只有在使用已取代的 Go 1.x 執行階段時，才需要 RPC 元件。排除 RPC 會縮減部署套件的大小。

Example – 多階段建置 Dockerfile

> **Note**
>
> 確保您在 Dockerfile 中指定的 Go 版本 (例如，`golang:1.20`) 與用於建立應用程式的 Go 版本相同。

```
FROM golang:1.20 as build
WORKDIR /helloworld
# Copy dependencies list
COPY go.mod go.sum ./
# Build with optional lambda.norpc tag
COPY main.go .
RUN go build -tags lambda.norpc -o main main.go
# Copy artifacts to a clean image
FROM public.ecr.aws/lambda/provided:al2023
COPY --from=build /helloworld/main ./main
ENTRYPOINT [ "./main" ]
```

6. 使用 docker build 命令建立 Docker 映像檔。以下範例將映像命名為 `docker-image` 並為其提供 `test` 標籤。

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> 此命令會指定 `--platform linux/amd64` 選項，確保無論建置機器的架構為何，您的容器都與 Lambda 執行環境相容。如果您打算使用 ARM64 指令集架構建立 Lambda 函數，務必將命令變更為改用 `--platform linux/arm64` 選項。

(選用) 在本機測試映像

使用 執行期界面模擬器 以在本機測試映像。執行期界面模擬器包含在 `provided.al2023` 基礎映像中。

若要在本機電腦上執行執行期界面模擬器

1. 使用 docker run 命令啟動 Docker 影像。注意下列事項：

   - `docker-image` 是映像名稱，而 `test` 是標籤。
   - `./main` 是來自 Dockerfile 的 ENTRYPOINT。

   ```
   docker run -d -p 9000:8080 \
   --entrypoint /usr/local/bin/aws-lambda-rie \
   docker-image:test ./main
   ```

   此命令將映像作為容器執行，並在 `localhost:9000/2015-03-31/functions/function/invocations` 建立本機端點。
2. 從新的終端機視窗，使用 curl 命令將事件張貼至下列端點：

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   此命令會透過空白事件調用函數，並傳回一個回應。部分函數可能需要使用 JSON 承載。範例：

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
    '{"payload":"hello world!"}'
   ```
3. 取得容器 ID。

   ```
   docker ps
   ```
4. 使用 docker kill 命令停止容器。在此命令中，將 `3766c4ab331c` 替換為上一步驟中的容器 ID。

   ```
   docker kill 3766c4ab331c
   ```

部署映像

若要將映像上傳至 Amazon ECR 並建立 Lambda 函數

1. 使用 get-login-password 命令，向 Amazon ECR 登錄檔驗證 Docker CLI。

   - 將`--region`值設定為您 AWS 區域 要建立 Amazon ECR 儲存庫的位置。
   - `111122223333`用您的 AWS 帳戶 身份證替換。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 使用 create-repository 命令在 Amazon ECR 中建立儲存庫。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 儲存庫必須與 Lambda 函數位於 AWS 區域 相同。

如果成功，您將會看到以下回應：

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. 從上一步驟的輸出中複製 `repositoryUri`。
4. 執行 docker tag 命令，將 Amazon ECR 儲存庫中的本機映像標記為最新版本。在此命令中：
   - 將 `docker-image:test` 替換為 Docker 映像檔的名稱和標籤。

- 將 <ECRrepositoryUri> 替換為複製的 repositoryUri。確保在 URI 的末尾包含 :latest。

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

範例：

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. 執行 docker push 命令，將本機映像部署至 Amazon ECR 儲存庫。確保在儲存庫 URI 的末尾包含 :latest。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 建立函數的執行角色 (若您還沒有的話)。在下一個步驟中您需要角色的 Amazon Resource Name (ARN)。
7. 建立 Lambda 函數。對於 ImageUri，從之前的設定中指定儲存庫 URI。確保在 URI 的末尾包含 :latest。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> 只要映像與 Lambda 函數位於相同的區域，您就可以使用不同 AWS 帳戶中的映像檔建立函數。如需詳細資訊，請參閱 Amazon ECR 跨帳戶許可。

8. 調用函數。

```
aws lambda invoke --function-name hello-world response.json
```

您應該看到如下回應：

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 若要查看函數的輸出，請檢查 `response.json` 檔案。

若要更新函數程式碼，您必須再次建置映像、將新映像上傳到 Amazon ECR 存放庫，然後使用 update-function-code 命令將映像部署到 Lambda 函數。

Lambda 將圖像標記解析為特定的圖像摘要。這表示如果您將用於部署函數的映像標籤指向 Amazon ECR 中的新映像，Lambda 不會自動更新函數以使用新映像。若要將新映像部署到相同的 Lambda 函數，您必須使用`update-function-code`命令，即使 Amazon ECR 中的影像標籤保持不變。

使用非AWS 基本圖像

您可以從非AWS 基礎映像為 Go 構建容器映像。下列步驟中的 Dockerfile 範例使用 Alpine 基礎映像。

您必須將 aws-lambda-go/lambda 套件加入 Go 處理常式。此套件會實作 Go 的程式設計模型 (包括執行期介面)。

必要條件

若要完成本節中的步驟，您必須執行下列各項：

- Go
- Docker
- AWS Command Line Interface (AWS CLI) 第二版

使用替代基礎映像建立映像

使用 Alpine 基礎映像建置和部署 Go 函數

1. 建立專案的目錄，然後切換至該目錄。

```
mkdir hello
cd hello
```

2. 初始化新的 Go 模組。

```
go mod init example.com/hello-world
```

3. 將 lambda 程式庫新增為新模組的相依項。

```
go get github.com/aws/aws-lambda-go/lambda
```

4. 建立名為 `main.go` 的檔案，然後在文字編輯器中開啟。這是 Lambda 函數的程式碼。可以使用以下範本程式碼進行測試，也可以將其替換為您自己的程式碼。

```
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, event events.APIGatewayProxyRequest)
 (events.APIGatewayProxyResponse, error) {
 response := events.APIGatewayProxyResponse{
  StatusCode: 200,
  Body:       "\"Hello from Lambda!\"",
 }
 return response, nil
}

func main() {
 lambda.Start(handler)
}
```

5. 使用文字編輯器在專案目錄中建立 Dockerfile。下列 Dockerfile 範例使用 Alpine 基礎映像。

   Example Dockerfile

> **Note**
>
> 確保您在 Dockerfile 中指定的 Go 版本 (例如，`golang:1.20`) 與用於建立應用程式的 Go 版本相同。

```
FROM golang:1.20.2-alpine3.16 as build
WORKDIR /helloworld
# Copy dependencies list
COPY go.mod go.sum ./
# Build
COPY main.go .
RUN go build -o main main.go
# Copy artifacts to a clean image
FROM alpine:3.16
COPY --from=build /helloworld/main /main
ENTRYPOINT [ "/main" ]
```

6. 使用 docker build 命令建立 Docker 映像檔。以下範例將映像命名為 docker-image 並為其提供 test 標籤。

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> 此命令會指定 --platform linux/amd64 選項，確保無論建置機器的架構為何，您的容器都與 Lambda 執行環境相容。如果您打算使用 ARM64 指令集架構建立 Lambda 函數，務必將命令變更為改用 --platform linux/arm64 選項。

(選用) 在本機測試映像

使用 執行期界面模擬器 以在本機測試映像。您可以將模擬器構建到映像中，也可以使用以下步驟將其安裝在本地計算機上。

若要在本機電腦上安裝並執行執行期介面模擬器

1. 從您的項目目錄中運行以下命令以下載運行時接口仿真器（x86-64 架構） GitHub 並將其安裝在本地計算機上。

   Linux/macOS

```
mkdir -p ~/.aws-lambda-rie && \
    curl -Lo ~/.aws-lambda-rie/aws-lambda-rie https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie && \
```

```
    chmod +x ~/.aws-lambda-rie/aws-lambda-rie
```

要安裝 arm64 模擬器，請使用以下命令替換上一個命令中的 GitHub 存儲庫 URL：

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

PowerShell

```
$dirPath = "$HOME\.aws-lambda-rie"
if (-not (Test-Path $dirPath)) {
    New-Item -Path $dirPath -ItemType Directory
}

$downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/
releases/latest/download/aws-lambda-rie"
$destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
```

若要安裝 arm64 模擬器，請將 $downloadLink 更換為下列項目：

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

2. 使用 docker run 命令啟動 Docker 影像。注意下列事項：

   - `docker-image` 是映像名稱，而 `test` 是標籤。
   - `/main` 是來自 Dockerfile 的 ENTRYPOINT。

   Linux/macOS

```
docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p
 9000:8080 \
    --entrypoint /aws-lambda/aws-lambda-rie \
    docker-image:test \
        /main
```

PowerShell

```
docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p
 9000:8080 `
--entrypoint /aws-lambda/aws-lambda-rie `
docker-image:test `
    /main
```

此命令將映像作為容器執行，並在 localhost:9000/2015-03-31/functions/function/invocations 建立本機端點。

> **Note**
>
> 如果您為 ARM64 指令集架構建立 Docker 映像檔，請務必將--platform linux/*arm64*選項改用選項。--platform linux/*amd64*

3. 將事件張貼至本機端點。

Linux/macOS

在 Linux 或 macOS 中，執行下列 curl 命令：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

在中 PowerShell，執行下列Invoke-WebRequest命令：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

4. 取得容器 ID。

```
docker ps
```

5. 使用 docker kill 命令停止容器。在此命令中，將 3766c4ab331c 替換為上一步驟中的容器 ID。

```
docker kill 3766c4ab331c
```

部署映像

若要將映像上傳至 Amazon ECR 並建立 Lambda 函數

1. 使用 get-login-password 命令，向 Amazon ECR 登錄檔驗證 Docker CLI。
   - 將`--region`值設定為您 AWS 區域 要建立 Amazon ECR 儲存庫的位置。
   - `111122223333`用您的 AWS 帳戶 身份證替換。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 使用 create-repository 命令在 Amazon ECR 中建立儲存庫。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 儲存庫必須與 Lambda 函數位於 AWS 區域 相同。

如果成功，您將會看到以下回應：

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. 從上一步驟的輸出中複製 `repositoryUri`。
4. 執行 docker tag 命令，將 Amazon ECR 儲存庫中的本機映像標記為最新版本。在此命令中：
   - 將 `docker-image:test` 替換為 Docker 映像檔的名稱和標籤。
   - 將 `<ECRrepositoryUri>` 替換為複製的 `repositoryUri`。確保在 URI 的末尾包含 `:latest`。

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

範例：

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world:latest
```

5. 執行 docker push 命令，將本機映像部署至 Amazon ECR 儲存庫。確保在儲存庫 URI 的末尾包含 `:latest`。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 建立函數的執行角色 (若您還沒有的話)。在下一個步驟中您需要角色的 Amazon Resource Name (ARN)。
7. 建立 Lambda 函數。對於 `ImageUri`，從之前的設定中指定儲存庫 URI。確保在 URI 的末尾包含 `:latest`。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> 只要映像與 Lambda 函數位於相同的區域，您就可以使用不同 AWS 帳戶中的映像檔建立函數。如需詳細資訊，請參閱 Amazon ECR 跨帳戶許可。

8. 調用函數。

```
aws lambda invoke --function-name hello-world response.json
```

您應該看到如下回應：

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 若要查看函數的輸出，請檢查 `response.json` 檔案。

若要更新函數程式碼，您必須再次建置映像、將新映像上傳到 Amazon ECR 存放庫，然後使用 update-function-code 命令將映像部署到 Lambda 函數。

Lambda 將圖像標記解析為特定的圖像摘要。這表示如果您將用於部署函數的映像標籤指向 Amazon ECR 中的新映像，Lambda 不會自動更新函數以使用新映像。若要將新映像部署到相同的 Lambda 函數，您必須使用`update-function-code`命令，即使 Amazon ECR 中的影像標籤保持不變。

AWS Lambda 函數登錄 Go

AWS Lambda 代表您自動監控 Lambda 函數，並將日誌傳送到 Amazon CloudWatch。您的 Lambda 函數隨附一個 CloudWatch 日誌記錄群組和函數每個執行個體的日誌串流。Lambda 執行期環境會將每次調用的詳細資訊傳送至日誌串流，並且轉傳來自函數程式碼的日誌及其他輸出。如需詳細資訊，請參閱 使用 Amazon CloudWatch 日誌 AWS Lambda。

本頁說明如何從 Lambda 函數的程式碼產生記錄輸出，或使用 Lambda 主控台或主控台存取 CloudWatch 日誌。 AWS Command Line Interface

章節

- 建立傳回日誌的函數
- 使用 Lambda 主控台
- 使用控 CloudWatch 制台
- 使用 AWS Command Line Interface （ AWS CLI ）
- 刪除日誌

建立傳回日誌的函數

若要從您的函數程式碼輸出日誌，您可以使用 fmt 套件上的方法，或任何能寫入 `stdout` 或 `stderr` 的記錄程式庫。下列範例使用日誌套件。

Example main.go - 記錄

```
func handleRequest(ctx context.Context, event events.SQSEvent) (string, error) {
  // event
  eventJson, _ := json.MarshalIndent(event, "", "  ")
  log.Printf("EVENT: %s", eventJson)
  // environment variables
  log.Printf("REGION: %s", os.Getenv("AWS_REGION"))
  log.Println("ALL ENV VARS:")
  for _, element := range os.Environ() {
    log.Println(element)
  }
```

Example 記錄格式

```
START RequestId: dbda340c-xmpl-4031-8810-11bb609b4c71 Version: $LATEST
```

```
2020/03/27 03:40:05 EVENT: {
  "Records": [
    {
      "messageId": "19dd0b57-b21e-4ac1-bd88-01bbb068cb78",
      "receiptHandle": "MessageReceiptHandle",
      "body": "Hello from SQS!",
      "md5OfBody": "7b27xmplb47ff90a553787216d55d91d",
      "md5OfMessageAttributes": "",
      "attributes": {
        "ApproximateFirstReceiveTimestamp": "1523232000001",
        "ApproximateReceiveCount": "1",
        "SenderId": "123456789012",
        "SentTimestamp": "1523232000000"
      },
      ...
2020/03/27 03:40:05 AWS_LAMBDA_LOG_STREAM_NAME=2020/03/27/
[$LATEST]569cxmplc3c34c7489e6a97ad08b4419
2020/03/27 03:40:05 AWS_LAMBDA_FUNCTION_NAME=blank-go-function-9DV3XMPL6XBC
2020/03/27 03:40:05 AWS_LAMBDA_FUNCTION_MEMORY_SIZE=128
2020/03/27 03:40:05 AWS_LAMBDA_FUNCTION_VERSION=$LATEST
2020/03/27 03:40:05 AWS_EXECUTION_ENV=AWS_Lambda_go1.x
END RequestId: dbda340c-xmpl-4031-8810-11bb609b4c71
REPORT RequestId: dbda340c-xmpl-4031-8810-11bb609b4c71 Duration: 38.66 ms Billed
 Duration: 39 ms Memory Size: 128 MB Max Memory Used: 54 MB Init Duration: 203.69 ms
XRAY TraceId: 1-5e7d7595-212fxmpl9ee07c4884191322 SegmentId: 42ffxmpl0645f474 Sampled:
 true
```

Go 執行時間會記錄每次調用的 START、END 和 REPORT 行。報告明細行提供下列詳細資訊。

REPORT 行資料欄位

- RequestId— 呼叫的唯一要求 ID。
- 持續時間 - 函數的處理常式方法處理事件所花費的時間量。
- 計費持續時間 - 調用的計費時間量。
- 記憶體大小 - 分配給函數的記憶體數量。
- 使用的記憶體上限 - 函數所使用的記憶體數量。
- 初始化持續時間 - 對於第一個提供的請求，這是執行期載入函數並在處理常式方法之外執行程式碼所花費的時間量。
- XRAY TraceId — 針對追蹤的要求，則為AWS X-Ray 追蹤識別碼。
- SegmentId— 針對追蹤的請求，X-Ray 區段 ID。

- 已取樣 - 對於追蹤的請求，這是取樣結果。

使用 Lambda 主控台

您可以在調用 Lambda 函數之後，使用 Lambda 主控台來檢視日誌輸出。

如果可以從內嵌程式碼編輯器測試您的程式碼，您會在執行結果中找到日誌。使用主控台測試功能以調用函數時，您會在詳細資訊區段找到日誌輸出。

使用控 CloudWatch 制台

您可以使用 Amazon 主 CloudWatch 控台來檢視所有 Lambda 函數叫用的日誌。

在 CloudWatch 主控台上檢視記錄檔

1. 在主控台上開啟 [記錄群組] 頁 CloudWatch 面。
2. 選擇您的函數的日誌群組 (/aws/lambda/***your-function-name***)。
3. 選擇日誌串流

每個日誌串流都會對應至函式的執行個體。當您更新 Lambda 函數，以及建立額外執行個體以處理多個並行調用時，便會出現日誌串流。要查找特定調用的日誌，我們建議使用檢測您的函數。 AWS X-Ray X-Ray 會在追蹤內記錄有關請求和日誌串流的詳細資訊。

使用 AWS Command Line Interface （ AWS CLI ）

這 AWS CLI 是一種開放原始碼工具，可讓您使用命令列殼層中的命令與 AWS 服務互動。若要完成本節中的步驟，您必須執行下列各項：

- AWS Command Line Interface (AWS CLI) 第二版
- AWS CLI -快速配置 `aws configure`

您可以透過 AWS CLI，使用 `--log-type` 命令選項來擷取要調用的日誌。其回應將包含 `LogResult` 欄位，內含該次調用的 base64 編碼日誌 (最大達 4 KB)。

Example 擷取日誌 ID

下列範例顯示如何從名稱為 `my-function` 的函數的 `LogResult` 欄位來擷取日誌 ID。

```
aws lambda invoke --function-name my-function out --log-type Tail
```

您應該會看到下列輸出：

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example 解碼日誌

在相同的命令提示中，使用 base64 公用程式來解碼日誌。下列範例顯示如何擷取 my-function 的 base64 編碼日誌。

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

如果您使用的是 AWS CLI 版本 2，則需要此cli-binary-format選項。若要讓此成為預設的設定，請執行 aws configure set cli-binary-format raw-in-base64-out。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

您應該會看到下列輸出：

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

該 base64 公用程式可在 Linux、macOS 和 Ubuntu on Windows 上使用。macOS 使用者可能需要使用 base64 -D。

Example get-logs.sh 指令碼

在相同的命令提示中，使用下列指令碼下載最後五個日誌事件。該指令碼使用 sed 以從輸出檔案移除引述，並休眠 15 秒以使日誌可供使用。輸出包括來自 Lambda 的回應以及來自 get-log-events 命令的輸出。

複製下列程式碼範例的內容，並將您的 Lambda 專案目錄儲存為 `get-logs.sh`。

如果您使用的是 AWS CLI 版本 2，則需要此cli-binary-format選項。若要讓此成為預設的設定，請執行 `aws configure set cli-binary-format raw-in-base64-out`。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS 和 Linux (僅限)

在相同的命令提示中，macOS 和 Linux 使用者可能需要執行下列命令，以確保指令碼可執行。

```
chmod -R 755 get-logs.sh
```

Example 擷取最後五個記錄事件

在相同的命令提示中，執行下列指令碼以取得最後五個日誌事件。

```
./get-logs.sh
```

您應該會看到下列輸出：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
```

```
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```


刪除日誌

當您刪除函數時，不會自動刪除日誌群組。若要避免無限期地儲存日誌，請刪除日誌群組，或設定保留期間，系統會在該時間之後自動刪除日誌。

檢測 Go 代碼 AWS Lambda

Lambda 與 AWS X-Ray 整合，可協助您追蹤、偵錯和最佳化 Lambda 應用程式。您可以使用 X-Ray 來追蹤請求，因為它會周遊您應用程式中的資源，其中可能包含 Lambda 函數和其他 AWS 服務。

若要將追蹤資料傳送至 X-Ray，您可以使用以下兩個 SDK 庫之一：

- AWS 適用於 OpenTelemetry (ADOT) 的發行版 — 安全、可生產就緒且 AWS支援的 (OTel) SDK 發行版本 OpenTelemetry 。
- AWS 適用於 Go 的 X-Ray SDK — 用於生成跟踪數據並將其發送到 X-Ray 的 SDK。

每個 SDK 均提供將遙測資料傳送至 X-Ray 服務的方法。然後，您可以使用 X-Ray 來檢視、篩選應用程式的效能指標並獲得洞察，從而識別問題和進行最佳化的機會。

> **Important**
>
> 用於 AWS Lambda SDK 的 X-Ray 和 Powertools 是由提供的緊密集成的儀表解決方案的一部分。 AWS ADOT Lambda Layers 是用於追蹤檢測之業界通用標準的一部分，這類檢測一般會收集更多資料，但可能不適用於所有使用案例。您可以使用任一解決方案在 X-Ray 中實作 end-to-end 追蹤。若要深入了解如何在兩者之間做選擇，請參閱在 AWS Distro for OpenTelemetry 和 X-Ray SDK 之間進行選擇。

章節

使用 ADOT 來檢測您的 Go 函數

ADOT 提供全受管 Lambda 層，包含使用 OTel SDK 收集遙測資料所需的一切內容。透過取用此層，您可以檢測 Lambda 函數，而無需修改任何函數程式碼。您還可以將層設定為對 OTel 進行自訂初始化。如需詳細資訊，請參閱 ADOT 文件中的針對 Lambda 上的 ADOT 收集器進行自訂組態設定。

針對 Go 執行階段，您可以新增適用於 ADOT Go 的AWS 受管 Lambda 層來自動檢測您的函數。有關如何添加此層的詳細說明，請參閱 AWS ADOT 文檔中的 OpenTelemetry Lambda Go Support 發行版。

使用 X-Ray SDK 來檢測 Go 函數

若要記錄 Lambda 函數對應用程式中其他資源進行呼叫的詳細資訊，您也可以使 AWS X-Ray SDK for Go。要獲取 SDK，請使用以下命令從其GitHub 存儲庫下載 SDK`go get`：

```
go get github.com/aws/aws-xray-sdk-go
```

要檢測 AWS SDK 客戶端，請將客戶端傳遞給該`xray.AWS()`方法。然後，便可以使用該方法的`WithContext`版本來追蹤呼叫。

```
svc := s3.New(session.New())
xray.AWS(svc.Client)
...
svc.ListBucketsWithContext(ctx aws.Context, input *ListBucketsInput)
```

新增正確的依賴項並進行必要的程式碼變更後，請透過 Lambda 主控台或 API 在函數的組態中啟用追蹤。

透過 Lambda 主控台來啟用追蹤

若要使用控制台在 Lambda 函數上切換主動追蹤，請按照下列步驟操作：

開啟主動追蹤

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇 組態 ，然後選擇 監控和操作工具 。
4. 選擇 編輯 。
5. 在 X-Ray 下，打開 主動追蹤 。
6. 選擇 儲存 。

透過 Lambda API 啟用追蹤

使用 AWS CLI 或 AWS SDK 在 Lambda 函數上設定追蹤功能，並使用下列 API 作業：

- UpdateFunction配置
- GetFunction配置
- CreateFunction

下列範例 AWS CLI 命令可在名為 my- function 的函式上啟用主動追蹤。

```
aws lambda update-function-configuration \
--function-name my-function \
--tracing-config Mode=Active
```

追蹤模式是您發布函數版本時版本特定組態的一部分。您無法變更已發佈版本上的追蹤模式。

使用啟動追蹤 AWS CloudFormation

若要啟動 AWS CloudFormation 範本中的`AWS::Lambda::Function`資源追蹤，請使用`TracingConfig`屬性。

Example function-inline.yml - 追蹤組態

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

對於 AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` 資源，請使用`Tracing`屬性。

Example template.yml - 追蹤組態

```
Resources:
  function:
```

```
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

解讀 X-Ray 追蹤

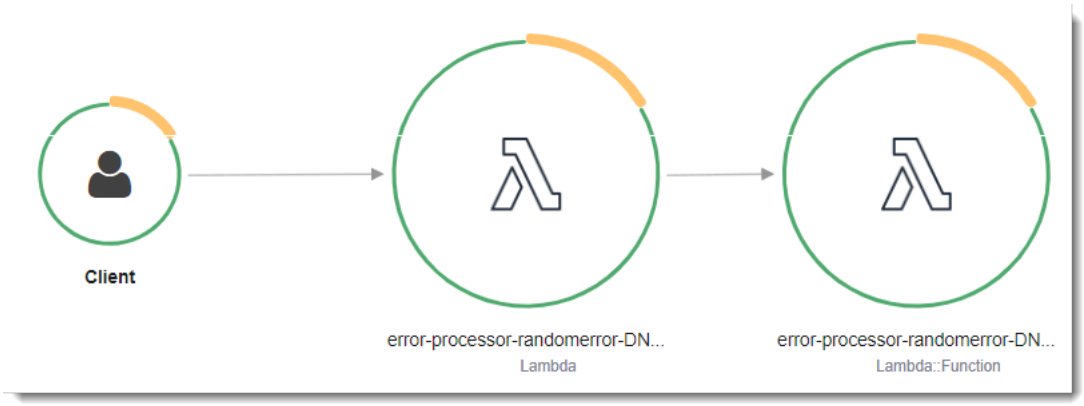
您的函數需要將追蹤資料上傳至 X-Ray 的許可。當您在 Lambda 主控台中啟用追蹤時，Lambda 會將必要的許可新增至函數的執行角色。否則，請將AWSXRayDaemonWriteAccess原則新增至執行角色。

設定主動追蹤之後，您可以透過應用程式來觀察特定請求。X-Ray 服務圖顯示了有關應用程式及其所有元件的資訊。下圖演示了具有兩個功能的應用程序。主要函式會處理事件，有時會傳回錯誤。頂部的第二個函數處理出現在第一個日誌組中的錯誤，並使用 AWS SDK 調用 X-Ray，Amazon 簡單存儲服務（Amazon S3）和亞馬遜 CloudWatch 日誌。

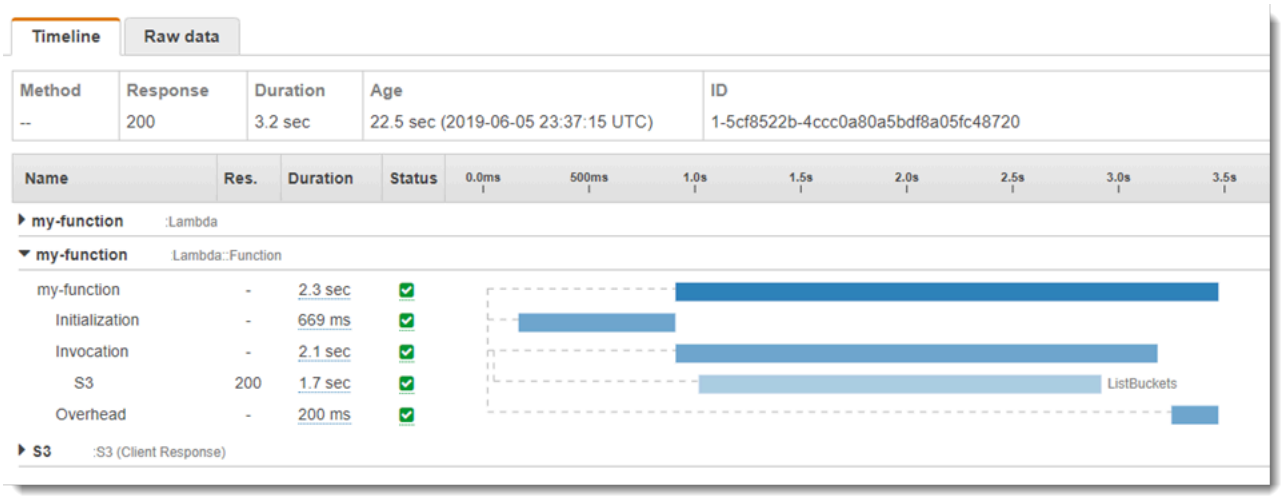
X-Ray 無法追蹤應用程式的所有請求。X-Ray 會套用取樣演算法以確保追蹤的效率，同時仍提供所有請求的代表範本。取樣率為每秒 1 次請求和 5% 的額外請求。

> **Note**
>
> 您無法針對函數設定 X-Ray 取樣率。

在 X-Ray 中，追蹤會記錄一或多個服務所處理之要求的相關資訊。Lambda 會記錄每個追蹤 2 個區段，在服務圖表上建立兩個節點。下列影像會強調顯示這兩個節點：



左側第一個節點代表接收調用請求的 Lambda 服務。第二個節點代表您特定的 Lambda 函數。下列範例顯示了具有這 2 個區段的追蹤。兩者都被命名為我的函數，但一個具有的起源`AWS::Lambda`，另一個具有的`AWS::Lambda::Function`起源。如果`AWS::Lambda`區段顯示錯誤，表示 Lambda 服務發生問題。如果`AWS::Lambda::Function`區段顯示錯誤，表示您的函數發生問題。



此範例會展開區`AWS::Lambda::Function`段，以顯示其三個子區段：

- 初始化 - 表示載入函數和執行初始化程式碼所花費的時間。只有函數的每個執行個體所處理的第一個事件會顯示此子區段。
- 調用 – 表示執行處理常式程式碼所花費的時間。
- 額外負荷 - 表示 Lambda 執行期為做好準備以處理下一個事件所花費的時間。

您也可以檢測 HTTP 用戶端、記錄 SQL 查詢，以及建立具有註釋和中繼資料的自訂子區段。如需詳細資訊，請參閱《AWS X-Ray 開發人員指南》中的適用於 Go 的AWS X-Ray 開發套件。

定價

作為免費方案的一部分，您可以每月免費使用 X-Ray 追蹤，最多達到一定限制。 AWS 達到閾值後，X-Ray 會收取追蹤儲存及擷取的費用。如需詳細資訊，請參閱 AWS X-Ray 定價。

使用 環境變數

如要在 Go 中存取環境變數，請使用 Getenv 函數。

以下說明執行方式。請注意，函式會匯入 fmt 套件，設定列印的結果格式，而 os 套件是與平台無關的系統界面，您可透過它存取環境變數。

```
package main

import (
 "fmt"
 "os"
 "github.com/aws/aws-lambda-go/lambda"
)

func main() {
 fmt.Printf("%s is %s. years old\n", os.Getenv("NAME"), os.Getenv("AGE"))

}
```

如需 Lambda 執行時間設定的環境變數清單，請參閱 定義執行時間環境變數。

使用 C# 建置 Lambda 函數

您可以使用受管 .NET 6 或 .NET 8 執行階段、自訂執行階段或容器映像檔，在 Lambda 中執行 .NET 應用程式。編譯應用程式程式碼之後，您能以 .zip 檔或容器映像檔形式部署至 Lambda。Lambda 提供適用於 .NET 語言的以下執行期：

.NET

名稱	識別符	作業系統	取代日期	封鎖函數建立	封鎖函數更新
。淨值 8	dotnet8	Amazon Linux 2023			
.NET 6	dotnet6	Amazon Linux 2	2024 年 11 月 12 日	2025年2月28日	2025年3月31日

設定您的 .NET 開發環境

若要開發和建置 Lambda 函數，您可以使用任何常用的 .NET 整合式開發環境 (IDE)，包括 Microsoft 視覺工作室、視覺工作室程式碼和 JetBrains 附加程式。為了簡化您的開發體驗，請 AWS 提供一組 .NET 專案範本，以及Amazon.Lambda.Tools命令列介面 (CLI)。

執行下列 .NET CLI 命令，即可安裝這些專案範本和命令列工具。

安裝 .NET 專案範本

要安裝項目模板（.NET 8），請執行以下操作：

```
dotnet new install Amazon.Lambda.Templates
```

若要安裝專案範本 (.NET 6)，請執行以下操作：

```
dotnet new --install Amazon.Lambda.Templates
```

> **Note**
>
> 如果您使用的是 .NET 6 受管 Lambda 執行階段，建議您升級為使用 .NET 8。若要深入了解，請參閱AWS 計算部落格 AWS Lambda上的管理 AWS Lambda 執行階段升級和介紹 .NET 8 執行階段。

安裝和更新 CLI 工具

執行下列命令以安裝、更新和解除安裝 `Amazon.Lambda.Tools` CLI。

若要安裝命令列工具，請執行以下操作：

```
dotnet tool install -g Amazon.Lambda.Tools
```

若要更新指令行工具：

```
dotnet tool update -g Amazon.Lambda.Tools
```

若要解除安裝命令列工具：

```
dotnet tool uninstall -g Amazon.Lambda.Tools
```

在 C #中定義Lambda 函數處理

Lambda 函數處理常式是您的函數程式碼中處理事件的方法。當有人呼叫您的函數時，Lambda 會執行處理常式方法。函數會執行，直到處理常式傳回回應、結束或逾時為止。

當您的函數遭調用，且 Lambda 執行了函數的處理常式方法時，系統會將兩個引數傳遞給函數。第一個引數是 `event` 物件。當另一個 AWS 服務 調用您的函數時，該event對象包含有關導致您的函數被調用的事件的數據。例如，來自 API Gateway 的 `event` 物件中，包含路徑、HTTP 方法和 HTTP 標頭的相關資訊。確切的事件結構根據 AWS 服務 調用函數而有所不同。如需個別服務事件格式的詳細資訊，請參閱：整合其他服務。

Lambda 也會傳遞 `context` 物件給函數。此物件包含有關調用、函數以及執行環境的資訊。如需詳細資訊，請參閱 the section called “Context”。

所有 Lambda 事件的原生格式，都是代表 JSON 格式事件的位元組串流。除非您的函數輸入與輸出參數為 `System.IO.Stream` 類型，否則必須將其序列化。透過設定 `LambdaSerializer` 組件屬性來指定要使用的序列化程式。如需詳細資訊，請參閱 the section called “Lambda 函數中的序列化”。

主題

- 適用於 Lambda 的 .NET 執行模型
- 類別庫處理常式
- 可執行組件處理常式
- Lambda 函數中的序列化
- 使用 Lambda Annotations 架構簡化函數程式碼
- Lambda 函數處理常式限制

適用於 Lambda 的 .NET 執行模型

在 .NET 中執行 Lambda 函數有兩種不同的執行模型：類別庫和可執行組件做法。

若使用類別庫做法，您可以為 Lambda 提供一個字串，指出要調用之函數的 `AssemblyName`、`ClassName`、和 `Method`。如需此字串格式的詳細資訊，請參閱：the section called “類別庫處理常式”。在函數的初始化階段，系統會初始化函數的類別，並執行建構函數中的任何程式碼。

若使用可執行組件做法，您可以使用 C# 9 的頂層陳述式功能。此方法會產生一個可執行組件，每當 Lambda 收到函數的調用命令時，就會執行該組件。您只需提供要執行的可執行組件名稱給 Lambda。

以下各節提供這兩種做法的範例函數程式碼。

類別庫處理常式

下列 Lambda 函數程式碼顯示使用類別庫做法之 Lambda 函數的處理常式方法 (FunctionHandler) 範例。在此範例函數中，Lambda 收到調用函數的 API Gateway 傳來的事件。函數會從資料庫讀取記錄，並在 API Gateway 回應中傳回該記錄。

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace GetProductHandler;

public class Function
{
    private readonly IDatabaseRepository _repo;

    public Function()
    {
        this._repo = new DatabaseRepository();
    }

    public async Task<APIGatewayProxyResponse> FunctionHandler(APIGatewayProxyRequest
 request)
    {
        var id = request.PathParameters["id"];

        var databaseRecord = await this._repo.GetById(id);

        return new APIGatewayProxyResponse
        {
            StatusCode = (int)HttpStatusCode.OK,
            Body = JsonSerializer.Serialize(databaseRecord)
        };
    }
}
```

建立 Lambda 函數時，您需要以處理常式字串的形式向 Lambda 提供函數處理常式的相關資訊。目的是指示 Lambda 在調用函數時要在程式碼中執行哪種方法。若使用 C#，使用類別庫做法時處理常式字串的格式如下：

ASSEMBLY::TYPE::METHOD，其中：

- ASSEMBLY 是應用程式的 .NET 組件名稱。如果您使用 Amazon.Lambda.Tools CLI 建置應用程式，而沒有使用 .csproj 檔案中的 AssemblyName 屬性來設定組件名稱，則 ASSEMBLY 單純為 .csproj 檔案的名稱。
- TYPE 是處理常式類型的完整名稱，包含 Namespace 和 ClassName。
- METHOD 是程式碼中函數處理常式方法的名稱。

在顯示的範例程式碼中，如果組件名為 GetProductHandler，則處理常式字串會是 GetProductHandler::GetProductHandler.Function::FunctionHandler。

可執行組件處理常式

在下列範例中，Lambda 函數定義為可執行的組件。此程式碼中的處理常式方法名為 Handler。使用可執行組件時，必須引導 Lambda 執行期。若要執行此作業，請使用 LambdaBootstrapBuilder.Create 方法。此方法的輸入是函數當成處理常式使用的方法，以及要使用的 Lambda 序列化程式。

如需有關使用頂層陳述式的詳細資訊，請參閱 AWS 運算部落格 AWS Lambda上的 .NET 6 執行階段簡介。

```
namespace GetProductHandler;

IDatabaseRepository repo = new DatabaseRepository();

await LambdaBootstrapBuilder.Create<APIGatewayProxyRequest>(Handler, new
 DefaultLambdaJsonSerializer())
    .Build()
    .RunAsync();

async Task<APIGatewayProxyResponse> Handler(APIGatewayProxyRequest apigProxyEvent,
 ILambdaContext context)
{
    var id = input.PathParameters["id"];

    var databaseRecord = await this.repo.GetById(id);

    return new APIGatewayProxyResponse
    {
        StatusCode = (int)HttpStatusCode.OK,
        Body = JsonSerializer.Serialize(databaseRecord)
    };
```

```
};
```

使用可執行組件時，指示 Lambda 如何執行程式碼的處理常式字串就是組件的名稱。在這個範例中，即為 GetProductHandler。

Lambda 函數中的序列化

若您的 Lambda 函數使用輸入或輸出類型而非 Stream 物件，您必須將序列化程式庫新增至您的應用程式。若要實作序列化，您可以使用 System.Text.Json 和 Newtonsoft.Json 提供的標準反射式序列化，或是使用原始碼產生的序列化。

使用原始碼產生的序列化

源代碼生成的序列化是 .NET 版本 6 及更高版本的功能，允許在編譯時生成序列化代碼。此方式不需要使用反射，且可提高函數的效能。若要在函數中使用原始碼產生的序列化，請執行以下操作：

- 建立一個繼承自 JsonSerializerContext 的新部分分類，為需要序列化或還原序列化的所有類型新增 JsonSerializable 屬性。
- 設定 LambdaSerializer 以使用 SourceGeneratorLambdaJsonSerializer<T>。
- 更新應用程式程式碼中的任何手動序列化或還原序列化，以使用新建立的類別。

以下程式碼顯示的範例函數，是使用原始碼產生的序列化。

```
[assembly:
 LambdaSerializer(typeof(SourceGeneratorLambdaJsonSerializer<CustomSerializer>))]

public class Function
{
    private readonly IDatabaseRepository _repo;

    public Function()
    {
        this._repo = new DatabaseRepository();
    }

    public async Task<APIGatewayProxyResponse> FunctionHandler(APIGatewayProxyRequest
 request)
    {
        var id = request.PathParameters["id"];

        var databaseRecord = await this._repo.GetById(id);
```

```
        return new APIGatewayProxyResponse
        {
            StatusCode = (int)HttpStatusCode.OK,
            Body = JsonSerializer.Serialize(databaseRecord,
 CustomSerializer.Default.Product)
        };
    }
}

[JsonSerializable(typeof(APIGatewayProxyRequest))]
[JsonSerializable(typeof(APIGatewayProxyResponse))]
[JsonSerializable(typeof(Product))]
public partial class CustomSerializer : JsonSerializerContext
{

}
```

> **Note**
>
> 如果您想透過 Lambda 使用原生預先編譯 (AOT)，必須使用原始碼產生的序列化。

使用反射式序列化

AWS 提供預先建置的程式庫，讓您快速將序列化新增至應用程式。您可以使用`Amazon.Lambda.Serialization.SystemTextJson`或`Amazon.Lambda.Serialization.Json` NuGet 套件來設定此項目。`Amazon.Lambda.Serialization.SystemTextJson` 會在背景使用 `System.Text.Json` 來執行序列化任務，而 `Amazon.Lambda.Serialization.Json` 會使用 `Newtonsoft.Json` 套件。

您也可以實作 `ILambdaSerializer` 界面 (隨 `Amazon.Lambda.Core` 程式庫提供) 來建立自己的序列化程式庫。此界面定義了兩種方法：

- `T Deserialize<T>(Stream requestStream);`

  若實作此方法，會將請求承載從 Invoke API 還原序列化至傳遞到 Lambda 函數處理常式的物件。
- `T Serialize<T>(T response, Stream responseStream);`

  若實作此方法，會將從 Lambda 函數處理常式傳回的結果，序列化至 Invoke API 作業傳回的回應承載中。

使用 Lambda Annotations 架構簡化函數程式碼

Lambda 註解是 .NET 6 和 .NET 8 的框架，它簡化了使用 C# 編寫 Lambda 函數的過程。透過 Annotations 架構，您可以取代使用一般程式設計模型編寫的大部分 Lambda 函數程式碼。使用此架構編寫的程式碼使用更簡單的表達式，讓您可專注於商業邏輯。

以下範例程式碼示範使用 Annotations 架構可如何簡化編寫 Lambda 函數的程序。第一個範例顯示使用一般 Lambda 程式模型編寫的程式碼，第二個範例顯示使用 Annotations 架構的對等程式碼。

```
public APIGatewayHttpApiV2ProxyResponse LambdaMathAdd(APIGatewayHttpApiV2ProxyRequest
 request, ILambdaContext context)
{
    if (!request.PathParameters.TryGetValue("x", out var xs))
    {
        return new APIGatewayHttpApiV2ProxyResponse
        {
            StatusCode = (int)HttpStatusCode.BadRequest
        };
    }
    if (!request.PathParameters.TryGetValue("y", out var ys))
    {
        return new APIGatewayHttpApiV2ProxyResponse
        {
            StatusCode = (int)HttpStatusCode.BadRequest
        };
    }
    var x = int.Parse(xs);
    var y = int.Parse(ys);
    return new APIGatewayHttpApiV2ProxyResponse
    {
        StatusCode = (int)HttpStatusCode.OK,
        Body = (x + y).ToString(),
        Headers = new Dictionary#string, string> { { "Content-Type", "text/plain" } }
    };
}
```

```
[LambdaFunction]
[HttpApi(LambdaHttpMethod.Get, "/add/{x}/{y}")]
public int Add(int x, int y)
{
    return x + y;
}
```

如需使用 Lambda 註解如何簡化程式碼的另一個範例，請參閱awsdocs/aws-doc-sdk-examples GitHub 儲存庫中的這個跨服務範例應用程式。PamApiAnnotations 資料夾在主要 function.cs 檔案中使用 Lambda Annotations。為了進行比較，PamApi 資料夾包含使用一般 Lambda 程式設計模型編寫的對等檔案。

Annotations 架構使用原始碼產生器來產生程式碼，會將 Lambda 程式設計模型轉換為第二個範例中呈現的程式碼。

如需如何使用 Lambda Annotations for .NET 的詳細資訊，請參閱下列資源：

- 存aws/aws-lambda-dotnet GitHub 放庫。
- 在 AWS 開發人員工具部落格中介紹 .NET 註解 Lambda 架構 (預覽版)。
- 包Amazon.Lambda.Annotations NuGet 裝。

使用 Lambda Annotations 架構進行相依性插入

您也可以透過 Lambda Annotations 架構，使用熟悉的語法將相依性插入新增至 Lambda 函數。將 [LambdaStartup] 屬性新增至 Startup.cs 檔案時，Lambda Annotations 架構會在編譯時產生所需的程式碼。

```
[LambdaStartup]
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddSingleton<IDatabaseRepository, DatabaseRepository>();
    }
}
```

Lambda 函數可以使用建構函數插入來插入服務，或使用 [FromServices] 屬性插入個別方法中。

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace GetProductHandler;

public class Function
{
    private readonly IDatabaseRepository _repo;
```

```
    public Function(IDatabaseRepository repo)
    {
        this._repo = repo;
    }

    [LambdaFunction]
    [HttpApi(LambdaHttpMethod.Get, "/product/{id}")]
    public async Task<Product> FunctionHandler([FromServices] IDatabaseRepository
 repository, string id)
    {
        return await this._repo.GetById(id);
    }
}
```

Lambda 函數處理常式限制

請注意，處理常式簽章有若干限制。

- 處理常式簽章不能為 unsafe，且不得使用指標類型，但可以在處理函式方法及其依存項目中使用 unsafe 內容。如需詳細資訊，請參閱 Microsoft 文檔網站上的不安全 (C# 參考)。
- 處理常式不會傳送使用 params 關鍵字的參數變數，也不得使用支援參數變數的 ArgIterator 作為輸入或傳回參數。
- 處理常式不得為通用方法，例如 IList<T> Sort<T>(IList<T> input)。
- 不支援具有 async void 簽章的非同步處理常式。

使用 .zip 封存檔建置和部署 C# Lambda 函數

.NET 部署套件 (.zip 封存檔) 包含函數的編譯組件，及其所有的組件相依項。該套件也包含 *proj*`.deps.json` 檔案。這會對 .NET 執行期發出訊號，告知函數的所有相依項和用於設定執行期的 *proj*`.runtimeconfig.json` 檔案。

若要部署個別 Lambda 函數，您可以使用 `Amazon.Lambda.Tools` .NET Lambda Global CLI。若使用 `dotnet lambda deploy-function` 命令，系統會自動建立 .zip 部署套件並部署至 Lambda。不過，我們建議您使用 AWS Serverless Application Model (AWS SAM) 或 AWS Cloud Development Kit (AWS CDK) 將 .NET 應用程式部署到 AWS.

無伺服器應用程式通常由 Lambda 函數和其他受管理共同 AWS 服務 運作以執行特定商務工作的組合。 AWS SAM 並使用其 AWS 服務 他函數大規模 AWS CDK 簡化建置和部署 Lambda 函數的作業範AWS SAM 本規格提供簡單且簡潔的語法來描述組成無伺服器應用程式的 Lambda 函數、API、權限、組態和其他 AWS 資源。您可以使用 AWS CDK 將雲端基礎架構定義為程式碼，協助您運用 .NET 等現代程式設計語言，在雲端建置可靠、可擴展且經濟實惠的應用程式。 AWS CDK 和 AWS SAM 使用 .NET Lambda 全域 CLI 來封裝您的函數。

雖然可以使用 .NET Core CLI，以 C# 函數使用 Lambda 層，但建議不要這麼做。使用層的 C# 函數會在 初始化階段 中手動載入共用組件，這可能會增加冷啟動時間。請改在編譯時加入所有共用程式碼，利用 .NET 編譯器的內建最佳化功能。

您可以在以下各節中找到使用 AWS SAM、和 .NET Lambda 全域 CLI 建置和部署 .NET Lambda 函數的相關說明。 AWS CDK

主題

- 使用 .NET Lambda Global CLI
- 使用部署 C# Lambda 函數 AWS SAM
- 使用部署 C# Lambda 函數 AWS CDK
- 部署 ASP.NET 應用程式

使用 .NET Lambda Global CLI

透過 .NET CLI 和 .NET Lambda Global Tools 延伸模組 (`Amazon.Lambda.Tools`)，可以跨平台建立 .NET 式 Lambda 應用程式、封裝這些應用程式，然後部署到 Lambda。在本節中，您將學習如何使用 .NET CLI 和 Amazon Lambda 範本建立新的 Lambda .NET 專案，以及如何使用 `Amazon.Lambda.Tools` 封裝和部署這些專案。

主題

必要條件

.NET 8 開發套件

如果您尚未這樣做，請安裝 .NET 8 SDK 和執行階段。

AWS 亞馬遜. Lambda. NET 專案範本

若要產生 Lambda 函數程式碼，請使用Amazon.Lambda.Templates NuGet 套件。若要安裝此範本套件，請執行下列命令：

```
dotnet new install Amazon.Lambda.Templates
```

AWS 亞馬遜工具. NET 全球 CLI 工具

若要建立 Lambda 函數，請使用 Amazon.Lambda.Tools .NET Core Global Tools 延伸模組。若要安裝 Amazon.Lambda.Tools，請執行下列命令：

```
dotnet tool install -g Amazon.Lambda.Tools
```

如需有關 Amazon.Lambda.Tools .NET CLI 延伸模組的詳細資訊，請參閱上的 .NET CLI 存放庫的 AWS 擴充功能 GitHub。

使用 .NET CLI 建立 .NET 專案

在 .NET CLI 中，在命令列中使用 `dotnet new` 命令建立 .NET 專案。Lambda 提供使用`Amazon.Lambda.Templates` NuGet 套件的其他範本。

安裝此套件後，執行以下命令即可查看可用範本的清單。

```
dotnet new list
```

若要檢查範本的詳細資訊，請使用 help 選項。例如，若要查看有關 lambda.EmptyFunction 範本的詳細資訊，請執行以下命令。

```
dotnet new lambda.EmptyFunction --help
```

若要建立 .NET Lambda 函數的基本範本，請使用 lambda.EmptyFunction 範本。這會建立一個簡單的函數，此函數會接受字串作為輸入，並使用 ToUpper 方法轉換為大寫。此範本支援以下選項：

- --name - 函數的名稱。
- --region— 要在其中建立函數的 AWS 區域。
- --profile— AWS SDK for .NET 認證檔案中設定檔的名稱。若要深入了解 .NET 中的認AWS 證設定檔，請參閱. NET 開發人員指南中的「AWS 設定認證」。

在此範例中，我們建立myDotnetFunction使用預設描述檔和 AWS 區域 設定名稱的新空函式：

```
dotnet new lambda.EmptyFunction --name myDotnetFunction
```

此命令會在您的專案目錄中建立以下檔案和目錄。

```
### myDotnetFunction
    ### src
    #   ### myDotnetFunction
    #       ### Function.cs
    #       ### Readme.md
    #       ### aws-lambda-tools-defaults.json
    #       ### myDotnetFunction.csproj
    ### test
        ### myDotnetFunction.Tests
            ### FunctionTest.cs
            ### myDotnetFunction.Tests.csproj
```

在 src/myDotnetFunction 目錄下，檢查下列檔案：

- aws-lambda-tools-defaults.json：您在部署 Lambda 函數時，在此檔案中指定命令列選項。例如：

```
  "profile" : "default",
  "region" : "us-east-2",
  "configuration" : "Release",
  "function-architecture": "x86_64",
```

```
  "function-runtime":"dotnet8",
  "function-memory-size" : 256,
  "function-timeout" : 30,
  "function-handler" : "myDotnetFunction::myDotnetFunction.Function::FunctionHandler"
```

- Function.cs：您的 Lambda 處理常式函數程式碼。它是 C# 範本，包含了預設的 `Amazon.Lambda.Core` 程式庫和預設的 `LambdaSerializer` 屬性。如需關於序列化需求及選項的詳細資訊，請參閱Lambda 函數中的序列化。它也包含可編輯以套用 Lambda 函數程式碼的範例函數。

```
using Amazon.Lambda.Core;

// Assembly attribute to enable the Lambda function's JSON input to be converted into
 a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializ

namespace myDotnetFunction;

public class Function
{

    /// <summary>
    /// A simple function that takes a string and does a ToUpper
    /// </summary#
    /// <param name="input"></param>
    /// <param name="context"></param>
    /// <returns></returns>
    public string FunctionHandler(string input, ILambdaContext context)
    {
        return input.ToUpper();
    }
}
```

- 我的 DotnetFunction .csproj：一個 MSBuild 文件，列出了構成您的應用程序的文件和程序集。

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <GenerateRuntimeConfigurationFiles>true</GenerateRuntimeConfigurationFiles>
    <AWSProjectType>Lambda</AWSProjectType>
```

```
    <!-- This property makes the build directory similar to a publish directory and
 helps the AWS .NET Lambda Mock Test Tool find project dependencies. -->
    <CopyLocalLockFileAssemblies>true</CopyLocalLockFileAssemblies>
    <!-- Generate ready to run images during publishing to improve cold start time.
 -->
    <PublishReadyToRun>true</PublishReadyToRun>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Amazon.Lambda.Core" Version="2.2.0" />
    <PackageReference Include="Amazon.Lambda.Serialization.SystemTextJson"
 Version="2.4.0" />
  </ItemGroup>
</Project>
```

- Readme：使用此檔案來記錄您的 Lambda 函數。

在 `myfunction/test` 目錄下，檢查下列檔案：

- 我的 DotnetFunction .tests.csproj：如前所述，這是一個 MSBuild 文件，列出了組成測試項目的文件和程序集。也請注意，它包含 `Amazon.Lambda.Core` 程式庫，以便您順利整合測試函數時所需要的任何 Lambda 範本。

```
<Project Sdk="Microsoft.NET.Sdk">
   ...

    <PackageReference Include="Amazon.Lambda.Core" Version="2.2.0 " />
   ...
```

- FunctionTest.cs：它包含在`src`目錄中的相同 C# 代碼模板文件。編輯此檔案以鏡像您的函數的生產程式碼，並在上傳 Lambda 函數至生產環境之前進行測試。

```
using Xunit;
using Amazon.Lambda.Core;
using Amazon.Lambda.TestUtilities;

using MyFunction;

namespace MyFunction.Tests
{
    public class FunctionTest
    {
        [Fact]
```

```
        public void TestToUpperFunction()
        {

            // Invoke the lambda function and confirm the string was upper cased.
            var function = new Function();
            var context = new TestLambdaContext();
            var upperCase = function.FunctionHandler("hello world", context);

            Assert.Equal("HELLO WORLD", upperCase);
        }
    }
}
```

使用 .NET CLI 部署 .NET 專案

若要建置部署套件並部署到 Lambda，請使用 `Amazon.Lambda.Tools` CLI 工具。若要從先前步驟中建立的檔案部署函數，請先瀏覽至包含函數 `.csproj` 檔案的資料夾。

```
cd myDotnetFunction/src/myDotnetFunction
```

若要將程式碼以 .zip 部署套件形式部署至 Lambda，請執行下列命令。選擇您的函數名稱。

```
dotnet lambda deploy-function myDotnetFunction
```

在部署期間，精靈會要求您選取the section called “執行角色 (函數存取其他資源的權限)”. 在此範例中，請選取 `lambda_basic_role`。

部署函數後，您可以使用 `dotnet lambda invoke-function` 命令在雲端進行測試。在 `lambda.EmptyFunction` 範本的程式碼範例中，您可以使用 `--payload` 選項傳入字串來測試函數。

```
dotnet lambda invoke-function myDotnetFunction --payload "Just checking if everything
 is OK"
```

如果函數部署成功，您應該會看到類似以下內容的輸出。

```
dotnet lambda invoke-function myDotnetFunction --payload "Just checking if everything
 is OK"
Amazon Lambda Tools for .NET Core applications (5.8.0)
```

```
Project Home: https://github.com/aws/aws-extensions-for-dotnet-cli, https://github.com/
aws/aws-lambda-dotnet

Payload:
"JUST CHECKING IF EVERYTHING IS OK"

Log Tail:
START RequestId: id Version: $LATEST
END RequestId: id
REPORT RequestId: id  Duration: 0.99 ms       Billed Duration: 1 ms         Memory
 Size: 256 MB     Max Memory Used: 12 MB
```

透過 .NET CLI 使用 Lambda 層

> **Note**
>
> 搭配 C# 等編譯語言函數使用層，可能不會具有與使用 Python 等轉譯語言一樣多的好處。由於 C# 是編譯語言，因此您的函數仍須在 init 階段，將所有共用組件手動載入記憶體中，這可能會增加冷啟動時間。相反地，建議您在編譯時加入任何共用程式碼，利用任何內建的編譯器最佳化功能。

.NET CLI 支援命令，協助您發布層及部署使用層的 C# 函數。若要將層發布至指定的 Amazon S3 儲存貯體，請在與 `.csproj` 檔案相同的目錄中執行以下命令：

```
dotnet lambda publish-layer <layer_name> --layer-type runtime-package-store --s3-
bucket <s3_bucket_name>
```

接著使用 .NET CLI 部署函數時，請在以下命令中指定要使用的層 ARN：

```
dotnet lambda deploy-function <function_name> --function-layers arn:aws:lambda:us-
east-1:123456789012:layer:layer-name:1
```

如需 Hello World 函式的完整範例，請參閱 blank-csharp-with-layer範例。

使用部署 C# Lambda 函數 AWS SAM

AWS Serverless Application Model (AWS SAM) 是一個工具組，可協助簡化在 AWS上建置和執行無伺服器應用程式的程序。您可以在 YAML 或 JSON 範本中定義應用程式的資源，並使用 AWS SAM

命令列介面 (AWS SAM CLI) 來建置、封裝及部署應用程式。當您從 AWS SAM 範本建立 Lambda 函數時，AWS SAM 會使用函數程式碼和您指定的任何相依性，自動建立 .zip 部署套件或容器映像檔。AWS SAM 然後使用AWS CloudFormation 堆棧部署您的函數。若要進一步了解如 AWS SAM 何使用建置和部署 Lambda 函數，請參閱開AWS Serverless Application Model發人員指南 AWS SAM中的入門使用。

下列步驟說明如何使用 AWS SAM來下載、建置和部署範例 .NET Hello World 應用程式。此範例應用程式使用 Lambda 函數和 Amazon API Gateway 端點來實作基本 API 後端。當您傳送 HTTP GET 請求至 API Gateway 端點時，API Gateway 端點會調用您的 Lambda 函數。此函數會傳回「hello world」訊息，以及處理您請求之 Lambda 函數執行個體的 IP 地址。

當您使用建置和部署應用程式時 AWS SAM，AWS SAM CLI 會在幕後使用`dotnet lambda package`命令封裝個別 Lambda 函數程式碼組合。

必要條件

.NET 8 開發套件

安裝 .NET 8 SDK 和執行階段。

AWS SAM CLI 版本 1.39 或更新版本

若要瞭解如何安裝最新版本的 AWS SAM CLI，請參閱安裝 AWS SAM CLI。

部署範例 AWS SAM 應用程式

1. 使用以下命令，初始化使用 Hello world .NET 範本的應用程式。

```
sam init --app-template hello-world --name sam-app \
--package-type Zip --runtime dotnet8
```

此命令會在您的專案目錄中建立以下檔案和目錄。

```
### sam-app
    ### README.md
    ### events
    #   ### event.json
    ### omnisharp.json
    ### samconfig.toml
    ### src
    #   ### HelloWorld
```

```
#       ### Function.cs
#       ### HelloWorld.csproj
#       ### aws-lambda-tools-defaults.json
### template.yaml
### test
    ### HelloWorld.Test
        ### FunctionTest.cs
        ### HelloWorld.Tests.csproj
```

2. 瀏覽至包含 `template.yaml file` 的目錄。此檔案是為應用程式定義 AWS 資源的範本，包括 Lambda 函數和 API Gateway API。

```
cd sam-app
```

3. 若要建置應用程式的原始碼，請執行下列命令。

```
sam build
```

4. 若要將應用程式部署到 AWS，請執行下列命令。

```
sam deploy --guided
```

此命令會透過下列一系列提示封裝並部署應用程式。若要接受預設選項，請按 Enter。

> **Note**
>
> 因為HelloWorldFunction 可能沒有定義授權，這可以嗎？ ，一定要輸入y。

- 堆疊名稱：要部署至 AWS CloudFormation的堆疊名稱。此名稱對您的和而言必須是唯一 AWS 帳戶 的 AWS 區域。
- AWS 區域： AWS 區域 您想要部署您的應用程式。
- 部署前確認變更：選取是，即可在 AWS SAM 部署應用程式變更之前手動檢閱任何變更集。如果您選取否， AWS SAM CLI 會自動部署應用程式變更。
- 允許建立 SAM CLI IAM 角色：許多 AWS SAM 範本 (包括本範例中的 Hello world) 建立 AWS Identity and Access Management (IAM) 角色，以授予 Lambda 函數存取其他角色的權限 AWS 服務。選取 [是] 以提供部署建立或修改 IAM 角色之 AWS CloudFormation 堆疊的權限。
- 停用復原：依預設，如果在建立或部署堆疊期間 AWS SAM 遇到錯誤，它會將堆疊復原至先前的版本。選取「否」以接受此預設設定。

- HelloWorldFunction 可能沒有定義授權，這樣可以：輸入y。
- 將引數儲存至 samconfig.toml：選取是以儲存您選擇的組態。您之後可以在沒有參數的情況下重新執行 `sam deploy`，以將變更部署到應用程式。

5. 應用程式部署完成後，CLI 會傳回 Hello World Lambda 函數的 Amazon Resource Name (ARN)，以及為應用程式建立的 IAM 角色。此外也會顯示您的 API Gateway API 端點。若要測試應用程式，請在瀏覽器中開啟端點。您應該會看到類似以下內容的回應。

```
{"message":"hello world","location":"34.244.135.203"}
```

6. 若要刪除資源，請執行下列命令。請注意，您建立的 API 端點，是可以透過網際網路存取的公有端點。建議您在測試後刪除此端點。

```
sam delete
```

後續步驟

若要進一步了解如 AWS SAM 何使用 .NET 建置和部署 Lambda 函數，請參閱下列資源：

- AWS Serverless Application Model (AWS SAM) 開發人員指南
- 使用 AWS Lambda 和 SAM CLI 建置無伺服器 .NET 應用程式

使用部署 C# Lambda 函數 AWS CDK

AWS Cloud Development Kit (AWS CDK) 這是一個開源軟件開發框架，用於將雲基礎架構定義為具有現代編程語言和 .NET 等框架的代碼。 AWS CDK 執行項目以生成 AWS CloudFormation 模板，然後用於部署代碼。

若要使用建置和部署範例 Hello world .NET 應用程式 AWS CDK，請遵循下列各節中的指示。範例應用程式會實作基本 API 後端 (包含 API Gateway 端點和 Lambda 函數)。當您傳送 HTTP GET 請求至端點時，API Gateway 會調用 Lambda 函數 此函數會傳回 Hello world 訊息，以及處理您請求之 Lambda 執行個體的 IP 地址。

必要條件

.NET 8 開發套件

安裝 .NET 8 SDK 和執行階段。

AWS CDK 第二版

若要瞭解如何安裝最新版本的， AWS CDK 請參閱 AWS Cloud Development Kit (AWS CDK) v2 開發人員指南 AWS CDK中的入門。

部署範例 AWS CDK 應用程式

1. 為範例應用程式建立專案目錄，並瀏覽至該目錄。

```
mkdir hello-world
cd hello-world
```

2. 執行下列命令來初始化新的 AWS CDK 應用程式。

```
cdk init app --language csharp
```

此命令會在您的專案目錄中建立以下檔案和目錄

```
### README.md
### cdk.json
### src
    ### HelloWorld
    #   ### GlobalSuppressions.cs
    #   ### HelloWorld.csproj
    #   ### HelloWorldStack.cs
    #   ### Program.cs
    ### HelloWorld.sln
```

3. 開啟 `src` 目錄，並使用 .NET CLI 建立新的 Lambda 函數。這是您將使用 AWS CDK部署的功能。在此範例中，您將使用 `lambda.EmptyFunction` 範本建立名為 `HelloWorldLambda` 的 Hello world 函數。

```
cd src
dotnet new lambda.EmptyFunction -n HelloWorldLambda
```

完成此步驟後，專案目錄中的目錄結構應如下所示。

```
### README.md
### cdk.json
### src
```

```
    ### HelloWorld
    #   ### GlobalSuppressions.cs
    #   ### HelloWorld.csproj
    #   ### HelloWorldStack.cs
    #   ### Program.cs
    ### HelloWorld.sln
    ### HelloWorldLambda
        ### src
        #   ### HelloWorldLambda
        #       ### Function.cs
        #       ### HelloWorldLambda.csproj
        #       ### Readme.md
        #       ### aws-lambda-tools-defaults.json
        ### test
            ### HelloWorldLambda.Tests
                ### FunctionTest.cs
                ### HelloWorldLambda.Tests.csproj
```

4. 在 src/HelloWorld 目錄中，開啟 HelloWorldStack.cs 檔案。將檔案的內容取代為下列程式碼。

```
using Amazon.CDK;
using Amazon.CDK.AWS.Lambda;
using Amazon.CDK.AWS.Logs;
using Constructs;

namespace CdkTest
{
    public class HelloWorldStack : Stack
    {
        internal HelloWorldStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
            var buildOption = new BundlingOptions()
            {
                Image = Runtime.DOTNET_8.BundlingImage,
                User = "root",
                OutputType = BundlingOutput.ARCHIVED,
                Command = new string[]{
            "/bin/sh",
                "-c",
                " dotnet tool install -g Amazon.Lambda.Tools"+
                " && dotnet build"+
```

```
                " && dotnet lambda package --output-package /asset-output/
function.zip"
                }
            };

             var helloWorldLambdaFunction = new Function(this,
 "HelloWorldFunction", new FunctionProps
            {
                Runtime = Runtime.DOTNET_8,
                MemorySize = 1024,
                LogRetention = RetentionDays.ONE_DAY,
                Handler =
 "HelloWorldLambda::HelloWorldLambda.Function::FunctionHandler",
                Code = Code.FromAsset("./src/HelloWorldLambda/src/
HelloWorldLambda", new Amazon.CDK.AWS.S3.Assets.AssetOptions
                {
                    Bundling = buildOption
                }),
            });
        }
    }
}
```

這個程式碼是用來編譯和綁定應用程式程式碼，以及 Lambda 函數本身的定義。BundlingOptions 物件可建立一個 zip 檔及一組用於產生 zip 檔內容的命令。在此情況中，dotnet lambda package 命令用於編譯和產生 zip 檔。

5. 若要部署應用程式，請執行下列命令。

```
cdk deploy
```

6. 使用 .NET Lambda CLI 調用已部署的 Lambda 函數。

```
dotnet lambda invoke-function HelloWorldFunction -p "hello world"
```

7. 完成測試後，除非您想要保留建立的資源，否則可直接刪除。若要刪除資源，請執行下列命令。

```
cdk destroy
```

後續步驟

若要進一步了解如 AWS CDK 何使用 .NET 建置和部署 Lambda 函數，請參閱下列資源：

- 在 C# 中使用 AWS CDK
- 使用 AWS CDK 建置、封裝和發佈 .NET C# Lambda 函數

部署 ASP.NET 應用程式

除了託管事件驅動型函數之外，您還可以搭配使用 .NET 與 Lambda 來託管輕量型 ASP.NET 應用程式。您可以使用套件建置和部署 ASP.NET 應用`Amazon.Lambda.AspNetCoreServer` NuGet 程式。在本節中，您將了解如何使用 .NET Lambda CLI 工具，將 ASP.NET Web API 部署至 Lambda。

主題

- 必要條件
- 將 ASP.NET Web API 部署到 Lambda
- 將 ASP.NET Minimal API 部署到 Lambda

必要條件

.NET 8 開發套件

安裝 .NET 8 SDK 和核心執行階段。

Amazon.Lambda.Tools

若要建立 Lambda 函數，請使用 Amazon.Lambda.Tools .NET Core Global Tools 延伸模組。若要安裝 Amazon.Lambda.Tools，請執行下列命令：

```
dotnet tool install -g Amazon.Lambda.Tools
```

如需有關 Amazon.Lambda.Tools .NET CLI 延伸模組的詳細資訊，請參閱上的 .NET CLI 存放庫的 AWS 擴充功能 GitHub。

Amazon.Lambda.Templates

若要產生 Lambda 函數程式碼，請使用Amazon.Lambda.Templates NuGet 套件。若要安裝此範本套件，請執行下列命令：

```
dotnet new --install Amazon.Lambda.Templates
```

將 ASP.NET Web API 部署到 Lambda

若要使用 ASP.NET 部署 Web API，您可以使用 .NET Lambda 範本建立新的 Web API 專案。使用下列命令來初始化新的 ASP.NET Web API 專案。在範例命令中，我們將專案命名為 `AspNetOnLambda`。

```
dotnet new serverless.AspNetCoreWebAPI -n AspNetOnLambda
```

此命令會在您的專案目錄中建立以下檔案和目錄。

```
.
### AspNetOnLambda
    ### src
    #   ### AspNetOnLambda
    #       ### AspNetOnLambda.csproj
    #       ### Controllers
    #       #   ### ValuesController.cs
    #       ### LambdaEntryPoint.cs
    #       ### LocalEntryPoint.cs
    #       ### Readme.md
    #       ### Startup.cs
    #       ### appsettings.Development.json
    #       ### appsettings.json
    #       ### aws-lambda-tools-defaults.json
    #       ### serverless.template
    ### test
        ### AspNetOnLambda.Tests
            ### AspNetOnLambda.Tests.csproj
            ### SampleRequests
            #   ### ValuesController-Get.json
            ### ValuesControllerTests.cs
            ### appsettings.json
```

當 Lambda 調用函數時，使用的進入點就是 `LambdaEntryPoint.cs` 檔案。.NET Lambda 範本建立的檔案包含下列程式碼。

```
namespace AspNetOnLambda;
```

```
public class LambdaEntryPoint : Amazon.Lambda.AspNetCoreServer.APIGatewayProxyFunction
{
    protected override void Init(IWebHostBuilder builder)
    {
        builder
            .UseStartup#Startup#();
    }

    protected override void Init(IHostBuilder builder)
    {
    }
}
```

Lambda 使用的進入點，必須繼承自 Amazon.Lambda.AspNetCoreServer 套件中三個基本類別的其中一個。這三個基本類別為：

- APIGatewayProxyFunction
- APIGatewayHttpApiV2ProxyFunction
- ApplicationLoadBalancerFunction

當您使用提供的 .NET Lambda 範本建立 LambdaEntryPoint.cs 檔案時，所使用的預設類別為 APIGatewayProxyFunction。您在函數中使用的基本類別，取決於位於 Lambda 函數前的 API 層。

三個基本類別都包含名為 FunctionHandlerAsync 的公有方法。Lambda 用來調用函數的處理常式字串中，會包含此方法的名稱。FunctionHandlerAsync 方法會將輸入事件承載轉換為正確的 ASP.NET 格式，並將 ASP.NET 回應轉換回 Lambda 回應承載。在顯示的範例 AspNetOnLambda 專案中，處理常式字串如下所示。

```
AspNetOnLambda::AspNetOnLambda.LambdaEntryPoint::FunctionHandlerAsync
```

若要將 API 部署至 Lambda，請執行下列命令，瀏覽至包含原始程式碼檔案的目錄，並使用 AWS CloudFormation部署函數。

```
cd AspNetOnLambda/src/AspNetOnLambda
dotnet lambda deploy-serverless
```

> **Tip**
>
> 使用**`dotnet lambda deploy-serverless`**命令部署 API 時，會根據 AWS CloudFormation 您在部署期間指定的堆疊名稱，為 Lambda 函數命名。若要為 Lambda 函數提供自訂名稱，請編輯`serverless.template`檔案以將`FunctionName`屬性新增至`AWS::Serverless::Function`資源。如需詳細資訊，請參閱使AWS CloudFormation 用指南中的「名稱類型」。

將 ASP.NET Minimal API 部署到 Lambda

若要將 ASP.NET Minimal API 部署到 Lambda，您可以使用 .NET Lambda 範本建立新的 Minimal API 專案。使用下列命令來初始化新的 Minimal API 專案。在此範例中，我們將專案命名為 `MinimalApiOnLambda`。

```
dotnet new serverless.AspNetCoreMinimalAPI -n MinimalApiOnLambda
```

此命令會在您的專案目錄中建立以下檔案和目錄。

```
### MinimalApiOnLambda
    ### src
        ### MinimalApiOnLambda
            ### Controllers
            #   ### CalculatorController.cs
            ### MinimalApiOnLambda.csproj
            ### Program.cs
            ### Readme.md
            ### appsettings.Development.json
            ### appsettings.json
            ### aws-lambda-tools-defaults.json
            ### serverless.template
```

`Program.cs` 檔案包含下列程式碼。

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllers();
```

```
// Add AWS Lambda support. When application is run in Lambda Kestrel is swapped out as
 the web server with Amazon.Lambda.AspNetCoreServer. This
// package will act as the webserver translating request and responses between the
 Lambda event source and ASP.NET Core.
builder.Services.AddAWSLambdaHosting(LambdaEventSource.RestApi);

var app = builder.Build();


app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();

app.MapGet("/", () => "Welcome to running ASP.NET Core Minimal API on AWS Lambda");

app.Run();
```

若要將 Minimal API 設為在 Lambda 上執行，您可能需要編輯此程式碼，讓 Lambda 和 ASP.NET Core 之間的請求和回應能正確轉譯。根據預設，函數會針對 REST API 事件來源加以設定。若使用 HTTP API 或 Application Load Balancer，請以下列其中一個選項取代 `(LambdaEventSource.RestApi)`：

- `(LambdaEventSource.HttpAPi)`
- `(LambdaEventSource.ApplicationLoadBalancer)`

若要將 API 部署至 Lambda，請執行下列命令，瀏覽至包含原始程式碼檔案的目錄，並使用 AWS CloudFormation部署函數。

```
cd MinimalApiOnLambda/src/MinimalApiOnLambda
dotnet lambda deploy-serverless
```

使用容器映像部署 .NET Lambda 函數

您可以透過三種方式為 .NET Lambda 函數建置容器映像：

- 使用 .NET 的 AWS 基本映像

  AWS 基礎映像會預先載入語言執行期、用來管理 Lambda 與函數程式碼之間互動的執行期界面用戶端，以及用於本機測試的執行期界面模擬器。

- 使用 AWS 僅限作業系統的基本影像

  AWS 僅限作業系統的基本映像檔包含 Amazon Linux 散發和執行階段介面模擬器。這些映像常用於為編譯語言 (如 Go 和 Rust) 和 Lambda 不提供基礎映像的語言或語言版本 (如 Node.js 19) 建置容器映像。您還可以使用僅限作業系統的基礎映像來實作自訂執行期。若要使映像檔與 Lambda 相容，您必須在映像中加入 適用於 .NET 的執行期介面用戶端。

- 使用非AWS 基本圖像

  您可以使用其他容器登錄檔中的替代基礎映像 (例如 Alpine Linux 或 Debian)。您也可以使用組織建立的自訂映像。若要使映像檔與 Lambda 相容，您必須在映像中加入 適用於 .NET 的執行期介面用戶端。

> **Tip**
>
> 若要縮短 Lambda 容器函數變成作用中狀態所需的時間，請參閱 Docker 文件中的使用多階段建置。若要建置有效率的容器映像，請遵循撰寫 Dockerfiles 的最佳實務。

本頁面會說明如何為 Lambda 建置、測試和部署容器映像。

主題

AWS 適用於 .NET 的基本映

AWS 為 .NET 提供下列基本影像：

標籤	執行期	作業系統	Dockerfile	棄用
8	。淨網 8	Amazon Linux 2023	適用於 .NET 8 的碼頭文件 GitHub	
6	.NET 6	Amazon Linux 2	適用於 .NET 6 的碼頭文件 GitHub	2024 年 11 月 12 日

Amazon ECR 儲存庫：gallery.ecr.aws/lambda/dotnet

使用 .NET 的 AWS 基本映像

必要條件

若要完成本節中的步驟，您必須執行下列各項：

- .NET SDK — 下列步驟使用 .NET 8 基本映像檔。確保您的 .NET 版本與您在 Dockerfile 中指定的基礎映像的版本相符。
- Docker

使用基礎映像建立和部署映像

在下面的步驟中，您可以使用 Amazon.Lambda.Templates 和 Amazon.Lambda.Tools 來建立 .NET 專案。然後，您可以建置 Docker 映像檔，將該映像檔上傳到 Amazon ECR，然後將其部署到 Lambda 函數。

1. 安裝亞馬遜. NuGet

```
dotnet new install Amazon.Lambda.Templates
```

2. 使用 `lambda.image.EmptyFunction` 範本建立 .NET 專案。

```
dotnet new lambda.image.EmptyFunction --name MyFunction --region us-east-1
```

3. 導覽至 *MyFunction*`/src/`*MyFunction* 目錄。這是儲存專案檔案的位置。檢查下列檔案：

   - aws-lambda-tools-defaults.json – 您在部署 Lambda 函數時，在此檔案中指定命令列選項。

- Function.cs – 您的 Lambda 處理常式函數程式碼。這是 C# 範本，包含了預設的 `Amazon.Lambda.Core` 程式庫和預設的 `LambdaSerializer` 屬性。如需有關序列化需求及選項的詳細資訊，請參閱 Lambda 函數中的序列化。可以使用提供的程式碼進行測試，也可以將其替換為您自己的程式碼。
- MyFunction.csproj — .NET 專案檔案，其中列出了構成應用程式的檔案和組件。
- Readme.md：此檔案包含有關範例 Lambda 函數的詳細資訊。

4. 檢查 `src/MyFunction` 目錄中的 Dockerfile。可以使用提供的 Dockerfile 進行測試，也可以將其替換為您自己的 Dockerfile。如果使用自己的，請確保：

   - 將 `FROM` 屬性設定為基礎映像的 URI。您的 .NET 版本必須與基礎映像的版本相符。
   - 將 `CMD` 引數設定為 Lambda 函數處理常式。這應符合 `aws-lambda-tools-defaults.json` 中的 `image-command`。

   Example Dockerfile

   ```
   # You can also pull these images from DockerHub amazon/aws-lambda-dotnet:8
   FROM public.ecr.aws/lambda/dotnet:8

   # Copy function code to Lambda-defined environment variable
   COPY publish/* ${LAMBDA_TASK_ROOT}

   # Set the CMD to your handler (could also be done as a parameter override outside
    of the Dockerfile)
   CMD [ "MyFunction::MyFunction.Function::FunctionHandler" ]
   ```

5. 安裝 Amazon.Lambda.Tools .NET Global Tool。

   ```
   dotnet tool install -g Amazon.Lambda.Tools
   ```

   如果已安裝 Amazon.Lambda.Tools，則請確保您有最新版本。

   ```
   dotnet tool update -g Amazon.Lambda.Tools
   ```

6. 如果您尚未這麼做，請將目錄變更為 `MyFunction/src/MyFunction`。

   ```
   cd src/MyFunction
   ```

7. 使用 Amazon.Lambda.Tools 建置 Docker 映像檔，將其推送至新的 Amazon ECR 儲存庫，然後部署 Lambda 函數。

   對於 `--function-role`，指定函數執行角色的角色名稱 (而非 Amazon Resource Name (ARN))。例如 `lambda-role`。

   ```
   dotnet lambda deploy-function MyFunction --function-role lambda-role
   ```

   如需有關. NET 全域工具的詳細資訊，請參閱上的 .NET CLI 存放庫的AWS 擴充功能。 GitHub

8. 調用函數。

   ```
   dotnet lambda invoke-function MyFunction --payload "Testing the function"
   ```

   如果一切順利，您會看到下列項目：

   ```
   Payload:
   "TESTING THE FUNCTION"

   Log Tail:
   START RequestId: id Version: $LATEST
   END RequestId: id
   REPORT RequestId: id  Duration: 0.99 ms       Billed Duration: 1 ms         Memory
    Size: 256 MB     Max Memory Used: 12 MB
   ```

9. 刪除 Lambda 函數。

   ```
   dotnet lambda delete-function MyFunction
   ```

透過執行期介面用戶端使用替代基礎映像

如果您使用僅限作業系統的基礎映像或替代的基礎映像，則必須在映像中加入執行期介面用戶端。執行期介面用戶端會讓您擴充 Lambda 執行階段 API，管理 Lambda 與函數程式碼之間的互動。

下列範例會示範如何使用非AWS 基底映像檔建立 .NET 的容器映像檔，以及如何新增亞馬遜。 RuntimeSupport 套件，也就是 .NET 的 Lambda 執行階段介面用戶端。該示例碼頭文件使用 Microsoft .NET 8 基本映像。

必要條件

若要完成本節中的步驟，您必須執行下列各項：

- .NET SDK — 下列步驟使用 .NET 8 基本映像檔。確保您的 .NET 版本與您在 Dockerfile 中指定的基礎映像的版本相符。
- Docker

使用替代基礎映像建立和部署映像

1. 安裝亞馬遜. NuGet

```
dotnet new install Amazon.Lambda.Templates
```

2. 使用 `lambda.CustomRuntimeFunction` 範本建立 .NET 專案。這個模板包括亞馬遜。RuntimeSupport包裝。

```
dotnet new lambda.CustomRuntimeFunction --name MyFunction --region us-east-1
```

3. 導覽至 *MyFunction*`/src/`*MyFunction* 目錄。這是儲存專案檔案的位置。檢查下列檔案：
   - aws-lambda-tools-defaults.json – 您在部署 Lambda 函數時，在此檔案中指定命令列選項。
   - Function.cs：此程式碼含有一個類別，其中包含可將 `Amazon.Lambda.RuntimeSupport` 程式庫初始化為自舉的 `Main` 方法。`Main` 方法是函數處理過程的進入點。`Main` 方法會將函數處理常式包裝在自舉可以使用的包裝函式之中。如需詳細資訊，請參閱使用亞馬遜。RuntimeSupport 作為存儲庫中的類 GitHub 庫。
   - MyFunction.csproj — .NET 專案檔案，其中列出了構成應用程式的檔案和組件。
   - Readme.md：此檔案包含有關範例 Lambda 函數的詳細資訊。
4. 開啓 `aws-lambda-tools-defaults.json` 檔案並「新增」下列幾行程式碼：

```
  "package-type": "image",
  "docker-host-build-output-dir": "./bin/Release/lambda-publish"
```

   - package-type：將部署套件定義為容器映像。
   - docker-host-build-output-dir：為建置過程設定輸出目錄。

Example aws-lambda-tools-defaults.json

```
{
  "Information": [
    "This file provides default values for the deployment wizard inside Visual
 Studio and the AWS Lambda commands added to the .NET Core CLI.",
    "To learn more about the Lambda commands with the .NET Core CLI execute the
 following command at the command line in the project root directory.",
    "dotnet lambda help",
    "All the command line options for the Lambda command can be specified in this
 file."
  ],
  "profile": "",
  "region": "us-east-1",
  "configuration": "Release",
  "function-runtime": "provided.al2023",
  "function-memory-size": 256,
  "function-timeout": 30,
  "function-handler": "bootstrap",
  "msbuild-parameters": "--self-contained true",
  "package-type": "image",
  "docker-host-build-output-dir": "./bin/Release/lambda-publish"
}
```

5. 在 *MyFunction*/src/*MyFunction* 目錄建立 Dockerfile。下列範例 Dockerfile 使用 Microsoft .NET 基礎映像，而非 AWS 基礎映像。

   - 將 FROM 屬性設為基礎映像識別符。您的 .NET 版本必須與基礎映像的版本相符。
   - 使用 COPY 命令將函數複製到 /var/task 目錄。
   - 將 ENTRYPOINT 設為您希望 Docker 容器在啟動時執行的模組。在這種情況下，該模組是會初始化 Amazon.Lambda.RuntimeSupport 程式庫的自舉。

   Example Dockerfile

   ```
   # You can also pull these images from DockerHub amazon/aws-lambda-dotnet:8
   FROM mcr.microsoft.com/dotnet/runtime:8.0

   # Set the image's internal work directory
   WORKDIR /var/task
   ```

```
# Copy function code to Lambda-defined environment variable
COPY "bin/Release/net8.0/linux-x64"  .

# Set the entrypoint to the bootstrap
ENTRYPOINT ["/usr/bin/dotnet", "exec", "/var/task/bootstrap.dll"]
```

6. 安裝 Amazon.Lambda.Tools .NET Global Tools 延伸模組。

```
dotnet tool install -g Amazon.Lambda.Tools
```

如果已安裝 Amazon.Lambda.Tools，則請確保您有最新版本。

```
dotnet tool update -g Amazon.Lambda.Tools
```

7. 使用 Amazon.Lambda.Tools 建置 Docker 映像檔，將其推送至新的 Amazon ECR 儲存庫，然後部署 Lambda 函數。

對於 `--function-role`，指定函數執行角色的角色名稱 (而非 Amazon Resource Name (ARN))。例如 `lambda-role`。

```
dotnet lambda deploy-function MyFunction --function-role lambda-role
```

如需有關. NET CLI 延AWS 伸模組的詳細資訊，請參閱上的. GitHub

8. 調用函數。

```
dotnet lambda invoke-function MyFunction --payload "Testing the function"
```

如果一切順利，您會看到下列項目：

```
Payload:
"TESTING THE FUNCTION"

Log Tail:
START RequestId: id Version: $LATEST
END RequestId: id
REPORT RequestId: id  Duration: 0.99 ms       Billed Duration: 1 ms         Memory
 Size: 256 MB     Max Memory Used: 12 MB
```

9. 刪除 Lambda 函數。

```
dotnet lambda delete-function MyFunction
```

將 .NET Lambda 函數代碼編譯為本機運行時格式

.NET 8 支持本地 ahead-of-time （ AOT ）編譯。使用原生 AOT，您可以將 Lambda 函數程式碼編譯為原生執行階段格式，這樣便不需要在執行階段編譯 .NET 程式碼。原生 AOT 編譯可縮短以 .NET 撰寫之 Lambda 函數的冷啟動時間。如需詳細資訊，請參閱 AWS 計算部落格 AWS Lambda上的 .NET 8 執行階段簡介。

章節

Lambda 執行時間

若要使用原生 AOT 編譯部署 Lambda 函數建置，請使用受管理的 .NET 8 Lambda 執行階段。這個執行階段支援使用 x86_64 和 arm64 架構。

當您在不使用 AOT 的情況下部署 .NET Lambda 函數時，您的應用程式會先編譯成中繼語言 (IL) 程式碼。在執行階段，Lambda 執行階段中的 just-in-time (JIT) 編譯器會取得 IL 程式碼，並視需要將其編譯成機器程式碼。使用使用原生 AOT 提前編譯的 Lambda 函數，您可以在部署函數時將程式碼編譯成機器程式碼，這樣您就不需要依賴 Lambda 執行階段中的 .NET 執行階段或 SDK 來在程式碼執行之前編譯程式碼。

AOT 的一個限制是，您的應用程式程式碼必須在具有與 .NET 8 執行階段使用的相同 Amazon Linux 2023 (AL2023) 作業系統的環境中進行編譯。.NET Lambda CLI 提供了使用 AL2023 映像檔在碼頭容器中編譯應用程式的功能。

為了避免潛在的跨架構相容性問題，我們強烈建議您在與您為函數設定的相同處理器架構的環境中編譯程式碼。若要深入了解跨架構編譯的限制，請參閱 Microsoft .NET 文件中的交叉編譯。

必要條件

Docker

若要使用原生 AOT，您的函數程式碼必須在與 .NET 8 執行階段具有相同 AL2023 作業系統的環境中編譯。以下各節中的 .NET CLI 命令使用泊塢視窗在 AL2023 環境中開發和建置 Lambda 函數。

.NET 8 SDK

原生 AOT 編譯是 .NET 8 的一項功能。您必須在建置電腦上安裝 .NET 8 SDK，而不僅是執行階段。

Amazon.Lambda.Tools

若要建立 Lambda 函數，請使用 Amazon.Lambda.Tools .NET Core Global Tools 延伸模組。若要安裝 Amazon.Lambda.Tools，請執行下列命令：

```
dotnet tool install -g Amazon.Lambda.Tools
```

如需有關 Amazon.Lambda.Tools .NET CLI 延伸模組的詳細資訊，請參閱上的 .NET CLI 存放庫的 AWS 擴充功能 GitHub。

Amazon.Lambda.Templates

若要產生 Lambda 函數程式碼，請使用Amazon.Lambda.Templates NuGet 套件。若要安裝此範本套件，請執行下列命令：

```
dotnet new install Amazon.Lambda.Templates
```

開始使用

.NET 全域 CLI 和 AWS Serverless Application Model (AWS SAM) 都提供使用原生 AOT 建置應用程式的入門範本。若要建置您的第一個原生 AOT Lambda 函數，請按照下列指示中的步驟操作。

初始化和部署原生 AOT 編譯的 Lambda 函數

1. 使用原生 AOT 範本初始化新專案，然後瀏覽至包含所建立 `.cs` 和 `.csproj` 檔案的目錄。在此範例中，我們將函數命名為 `NativeAotSample`。

   ```
   dotnet new lambda.NativeAOT -n NativeAotSample
   cd ./NativeAotSample/src/NativeAotSample
   ```

原生 AOT 範本建立的 Function.cs 檔案包含以下函數程式碼。

```
using Amazon.Lambda.Core;
using Amazon.Lambda.RuntimeSupport;
using Amazon.Lambda.Serialization.SystemTextJson;
using System.Text.Json.Serialization;

namespace NativeAotSample;

public class Function
{
    /// <summary>
    /// The main entry point for the Lambda function. The main function is called
 once during the Lambda init phase. It
    /// initializes the .NET Lambda runtime client passing in the function handler
 to invoke for each Lambda event and
    /// the JSON serializer to use for converting Lambda JSON format to the .NET
 types.
    /// </summary>
    private static async Task Main()
    {
        Func<string, ILambdaContext, string> handler = FunctionHandler;
        await LambdaBootstrapBuilder.Create(handler, new
 SourceGeneratorLambdaJsonSerializer<LambdaFunctionJsonSerializerContext>())
            .Build()
            .RunAsync();
    }

    /// <summary>
    /// A simple function that takes a string and does a ToUpper.
    ///
    /// To use this handler to respond to an AWS event, reference the appropriate
 package from
    /// https://github.com/aws/aws-lambda-dotnet#events
    /// and change the string input parameter to the desired event type. When the
 event type
    /// is changed, the handler type registered in the main method needs to be
 updated and the LambdaFunctionJsonSerializerContext
    /// defined below will need the JsonSerializable updated. If the return type
 and event type are different then the
    /// LambdaFunctionJsonSerializerContext must have two JsonSerializable
 attributes, one for each type.
    ///
```

```
    // When using Native AOT extra testing with the deployed Lambda functions is
 required to ensure
    // the libraries used in the Lambda function work correctly with Native AOT. If
 a runtime
    // error occurs about missing types or methods the most likely solution will be
 to remove references to trim-unsafe
    // code or configure trimming options. This sample defaults to partial TrimMode
 because currently the AWS
    // SDK for .NET does not support trimming. This will result in a larger
 executable size, and still does not
    // guarantee runtime trimming errors won't be hit.
    /// </summary>
    /// <param name="input"></param>
    /// <param name="context"></param>
    /// <returns></returns>
    public static string FunctionHandler(string input, ILambdaContext context)
    {
        return input.ToUpper();
    }
}

/// <summary>
/// This class is used to register the input event and return type for the
 FunctionHandler method with the System.Text.Json source generator.
/// There must be a JsonSerializable attribute for each type used as the input and
 return type or a runtime error will occur
/// from the JSON serializer unable to find the serialization information for
 unknown types.
/// </summary>
[JsonSerializable(typeof(string))]
public partial class LambdaFunctionJsonSerializerContext : JsonSerializerContext
{
    // By using this partial class derived from JsonSerializerContext, we can
 generate reflection free JSON Serializer code at compile time
    // which can deserialize our class and properties. However, we must attribute
 this class to tell it what types to generate serialization code for.
    // See https://docs.microsoft.com/en-us/dotnet/standard/serialization/system-
text-json-source-generation
```

原生 AOT 會將應用程式編譯成單一原生二進位檔。該二進位檔的進入點是 `static Main` 方法。在 `static Main` 中，系統會引導 Lambda 執行期並設定 `FunctionHandler` 方法。在執行期引導程序中，原始碼產生的序列化程式是使用 new

`SourceGeneratorLambdaJsonSerializer<LambdaFunctionJsonSerializerContext>()` 加以設定

2. 若要將應用程式部署到 Lambda，請確定本機環境中的 Docker 正在執行，並執行下列命令。

```
dotnet lambda deploy-function
```

在幕後，.NET 全域 CLI 會下載 AL2023 Docker 映像檔，並在執行中的容器中編譯您的應用程式程式碼。編譯後的二進位檔會輸出回本機檔案系統，然後再部署至 Lambda。

3. 執行以下命令來測試函數。以您在部署精靈中為函數選擇的名稱取代 `<FUNCTION_NAME>`。

```
dotnet lambda invoke-function <FUNCTION_NAME> --payload "hello world"
```

CLI 的回應包括冷啟動 (初始化持續時間) 的效能詳細資訊，以及函數調用的總執行時間。

4. 若要依照上述步驟刪除您建立的 AWS 資源，請執行下列命令。以您在部署精靈中為函數選擇的名稱取代 `<FUNCTION_NAME>`。刪除不再使用的 AWS 資源後，您就可以避免向您收取不必要的費用 AWS 帳戶。

```
dotnet lambda delete-function <FUNCTION_NAME>
```

序列化

若要使用原生 AOT 將函數部署至 Lambda，您的函數程式碼必須使用原始碼產生的序列化。原始碼產生器不會使用執行期反射，來收集序列化存取物件屬性所需的中繼資料，而是會產生建置應用程式時編譯的 C# 原始碼檔案。若要正確設定原始碼產生的序列化程式，請確認函數使用的所有輸入和輸出物件及自訂類型都包含在內。例如，接收來自 API Gateway REST API 的事件並傳回自訂 `Product` 類型的 Lambda 函數，會包含定義如下的序列化程式。

```
[JsonSerializable(typeof(APIGatewayProxyRequest))]
[JsonSerializable(typeof(APIGatewayProxyResponse))]
[JsonSerializable(typeof(Product))]
public partial class CustomSerializer : JsonSerializerContext
{
}
```

裁剪

本機 AOT 會在編譯過程中裁剪應用程式程式碼，以盡可能縮小二進位檔。與舊版 .NET 相比，Lambda 的 .NET 8 提供了改進的裁剪支援。Lambda 執行階段程式庫、AWS .NET 開發套件、.NET Lambda 註解和 .NET 8 本身已新增 Support 援。

這些改進提供了消除構建時間修剪警告的可能性，但 .NET 永遠不會完全修剪安全。這表示函數相依的程式庫，可能會在編譯過程中遭到部分裁剪。您可以通過定義`TrimmerRootAssemblies`為`.csproj`文件的一部分來管理這一點，如以下示例所示。

```
<ItemGroup>
    <TrimmerRootAssembly Include="AWSSDK.Core" />
    <TrimmerRootAssembly Include="AWSXRayRecorder.Core" />
    <TrimmerRootAssembly Include="AWSXRayRecorder.Handlers.AwsSdk" />
    <TrimmerRootAssembly Include="Amazon.Lambda.APIGatewayEvents" />
    <TrimmerRootAssembly Include="bootstrap" />
    <TrimmerRootAssembly Include="Shared" />
</ItemGroup>
```

請注意，當您收到 trim 警告時，新增產生警告的類別`TrimmerRootAssembly`可能無法解決問題。修剪警告表示該類正在嘗試訪問某些其他類，直到運行時才能確定。若要避免執行階段錯誤，請將此第二個類別加入`TrimmerRootAssembly`。

若要深入了解如何管理修剪警告，請參閱 Microsoft .NET 文件中的修剪警告簡介。

故障診斷

Error: Cross-OS native compilation is not supported (錯誤：不支援跨作業系統原生編譯)。

    您的 Amazon.Lambda.Tools .NET Core 全域工具版本太舊。請更新至最新版本再重試。

Docker: image operating system "linux" cannot be used on this platform (Docker：映像檔作業系統 "linux" 不能在此平台上使用)。

    系統上的 Docker 設定為使用 Windows 容器。請更換至 Linux 容器以執行原生 AOT 建置環境。

如需有關常見錯誤的詳細資訊，請參閱上的 .NET 存放庫的AWS 原生 Aot。 GitHub

C# 中的 AWS Lambda 內容物件

當 Lambda 執行您的函數時，它會將內容物件傳遞至處理常式。此物件提供的各項屬性包含了有關叫用、函式以及執行環境的資訊。

內容屬性

- `FunctionName` – Lambda 函數的名稱。
- `FunctionVersion` – 函數的版本。
- `InvokedFunctionArn` - 用於調用此函數的 Amazon Resource Name (ARN)。指出調用者是否指定版本號或別名。
- `MemoryLimitInMB` - 分配給函數的記憶體數量。
- `AwsRequestId` - 調用請求的識別符。
- `LogGroupName` - 函數的日誌群組。
- `LogStreamName` - 函數執行個體的記錄串流。
- `RemainingTime(TimeSpan)` - 執行逾時前剩餘的毫秒數。
- `Identity` - (行動應用程式) 已授權請求的 Amazon Cognito 身分的相關資訊。
- `ClientContext` - (行動應用程式) 用戶端應用程式提供給 Lambda 的用戶端內容。
- `Logger` 函式的 Logger 物件。

基於監控目的，您可以使用 `ILambdaContext` 物件中的資訊來輸出函數調用的相關資訊。下列程式碼範例說明如何將內容資訊新增至結構化日誌記錄架構。在此範例中，函數會將 `AwsRequestId` 新增至日誌輸出。如果 Lambda 函數即將逾時，函數也會使用 `RemainingTime` 屬性取消傳輸中的任務。

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace GetProductHandler;

public class Function
{
    private readonly IDatabaseRepository _repo;

    public Function()
    {
        this._repo = new DatabaseRepository();
    }
```

```
    public async Task<APIGatewayProxyResponse> FunctionHandler(APIGatewayProxyRequest
 request, ILambdaContext context)
    {
        Logger.AppendKey("AwsRequestId", context.AwsRequestId);

        var id = request.PathParameters["id"];

        using var cts = new CancellationTokenSource();

        try
        {
            cts.CancelAfter(context.RemainingTime.Add(TimeSpan.FromSeconds(-1)));

            var databaseRecord = await this._repo.GetById(id, cts.Token);

            return new APIGatewayProxyResponse
            {
                StatusCode = (int)HttpStatusCode.OK,
                Body = JsonSerializer.Serialize(databaseRecord)
            };
        }
        finally
        {
            cts.Cancel();

            return new APIGatewayProxyResponse
            {
                StatusCode = (int)HttpStatusCode.InternalServerError,
                Body = JsonSerializer.Serialize(databaseRecord)
            };
        }
    }
}
```

C# 中的 Lambda 函數日誌記錄

AWS Lambda 自動監控 Lambda 函數，並將日誌項目傳送到 Amazon CloudWatch。您的 Lambda 函數隨附一個 CloudWatch 日誌記錄群組和函數每個執行個體的日誌串流。Lambda 執行期環境會將每次調用的詳細資訊和函數程式碼的其他輸出，傳送至日誌串流。如需有關 CloudWatch 記錄檔的詳細資訊，請參閱使用 Amazon CloudWatch 日誌 AWS Lambda。

章節

建立傳回日誌的函數

若要由您的函數程式碼輸出日誌，您可以使用主控台類別的方法，或任何能寫入 `stdout` 或 `stderr` 的記錄程式庫。

.NET 執行期會記錄每次調用的 START、END 和 REPORT 行。報告明細行提供下列詳細資訊。

REPORT 行資料欄位

- RequestId— 呼叫的唯一要求 ID。
- 持續時間 - 函數的處理常式方法處理事件所花費的時間量。
- 計費持續時間 - 調用的計費時間量。
- 記憶體大小 - 分配給函數的記憶體數量。
- 使用的記憶體上限 - 函數所使用的記憶體數量。
- 初始化持續時間 - 對於第一個提供的請求，這是執行期載入函數並在處理常式方法之外執行程式碼所花費的時間量。
- XRAY TraceId — 針對追蹤的要求，則為AWS X-Ray 追蹤識別碼。

- SegmentId— 針對追蹤的請求，X-Ray 區段 ID。
- 已取樣 - 對於追蹤的請求，這是取樣結果。

工具與程式庫

Powertools to AWS Lambda (.NET) 是一套開發人員工具組，用來實作無伺服器最佳實務並提高開發人員速度。記錄公用程式提供 Lambda 優化記錄器，其中包含有關所有函數之函數內容的其他資訊，輸出結構為 JSON。使用此公用程式執行下列操作：

- 從 Lambda 內容、冷啟動和 JSON 形式的結構記錄輸出中擷取關鍵欄位
- 在收到指示時記錄 Lambda 調用事件 (預設為停用)
- 透過日誌採樣僅列印調用百分比的所有日誌 (預設為停用)
- 在任何時間點將其他金鑰附加至結構化日誌
- 使用自訂日誌格式化程式 (自帶格式化程式)，以與組織的日誌記錄 RFC 相容的結構輸出日誌。

使用動力工具進行 AWS Lambda （ .NET ）和結構化日 AWS SAM 誌記錄

請按照下面的步驟來下載，構建和部署示例你好世界 C# 應用程序與集成 Powertools 的模塊使用 AWS SAM. AWS Lambda 此應用程式實作了基本 API 後端，並使用 Powertools 發送日誌、指標和追蹤。其包含 Amazon API Gateway 端點和 Lambda 函數。當您將 GET 請求傳送至 API Gateway 端點時，Lambda 函數會叫用、使用內嵌指標格式將記錄和指標傳送至 CloudWatch，並將追蹤傳送至 AWS X-Ray。該函數會傳回 `hello world` 訊息。

必要條件

若要完成本節中的步驟，您必須執行下列各項：

- . NET 6 或. NET 8
- AWS CLI 第二版
- AWS SAM CLI 版本 1.75 或更新版本。如果您使用較舊版本的 AWS SAM CLI，請參閱升級 AWS SAM CLI。

部署範例 AWS SAM 應用程式

1. 使用 Hello World TypeScript 範本初始化應用程式。

```
sam init --app-template hello-world-powertools-dotnet --name sam-app --package-type
 Zip --runtime dotnet6 --no-tracing
```

2. 建置應用程式。

```
cd sam-app && sam build
```

3. 部署應用程式。

```
sam deploy --guided
```

4. 依照螢幕上的提示操作。若要接受互動體驗中提供的預設選項，請按下 Enter。

> **Note**
>
> 因為HelloWorldFunction 可能沒有定義授權，這可以嗎？ ，請務必輸入y。

5. 取得已部署應用程式的 URL：

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. 調用 API 端點：

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

成功的話，您將會看到以下回應：

```
{"message":"hello world"}
```

7. 若要獲取該函數的日誌，請執行 sam 日誌。如需詳細資訊，請參閱《AWS Serverless Application Model 開發人員指南》 中的 使用日誌。

```
sam logs --stack-name sam-app
```

日誌輸出如下：

```
2023/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8
 2023-02-20T14:15:27.988000 INIT_START Runtime Version:
```

```
 dotnet:6.v13        Runtime Version ARN: arn:aws:lambda:ap-
southeast-2::runtime:699f346a05dae24c58c45790bc4089f252bf17dae3997e79b17d939a288aa1ec
2023/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2023-02-20T14:15:28.229000
 START RequestId: bed25b38-d012-42e7-ba28-f272535fb80e Version: $LATEST
2023/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2023-02-20T14:15:29.259000
 2023-02-20T14:15:29.201Z        bed25b38-d012-42e7-ba28-f272535fb80e    info
   {"_aws":{"Timestamp":1676902528962,"CloudWatchMetrics":[{"Namespace":"sam-
app-logging","Metrics":[{"Name":"ColdStart","Unit":"Count"}],"Dimensions":
[["FunctionName"],["Service"]]}]},"FunctionName":"sam-app-HelloWorldFunction-
haKIoVeose2p","Service":"PowertoolsHelloWorld","ColdStart":1}
2023/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2023-02-20T14:15:30.479000
 2023-02-20T14:15:30.479Z        bed25b38-d012-42e7-ba28-f272535fb80e    info
 {"ColdStart":true,"XrayTraceId":"1-63f3807f-5dbcb9910c96f50742707542","CorrelationId":"d3d
a549-4d67b2fdc015","FunctionName":"sam-app-HelloWorldFunction-
haKIoVeose2p","FunctionVersion":"$LATEST","FunctionMemorySize":256,"FunctionArn":"arn:aws:l
southeast-2:123456789012:function:sam-app-HelloWorldFunction-
haKIoVeose2p","FunctionRequestId":"bed25b38-d012-42e7-ba28-
f272535fb80e","Timestamp":"2023-02-20T14:15:30.4602970Z","Level":"Information","Service":"P
 world API - HTTP 200"}
2023/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2023-02-20T14:15:30.599000
 2023-02-20T14:15:30.599Z        bed25b38-d012-42e7-ba28-f272535fb80e    info
   {"_aws":{"Timestamp":1676902528922,"CloudWatchMetrics":[{"Namespace":"sam-
app-logging","Metrics":[{"Name":"ApiRequestCount","Unit":"Count"}],"Dimensions":
[["Service"]]}]},"Service":"PowertoolsHelloWorld","ApiRequestCount":1}
2023/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2023-02-20T14:15:30.680000 END
 RequestId: bed25b38-d012-42e7-ba28-f272535fb80e
2023/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2023-02-20T14:15:30.680000
 REPORT RequestId: bed25b38-d012-42e7-ba28-f272535fb80e  Duration: 2450.99 ms
  Billed Duration: 2451 ms Memory Size: 256 MB     Max Memory Used: 74 MB  Init
 Duration: 240.05 ms
XRAY TraceId: 1-63f3807f-5dbcb9910c96f50742707542       SegmentId: 16b362cd5f52cba0
```

8. 這是可透過網際網路存取的公有 API 端點。建議您在測試後刪除端點。

```
sam delete
```

管理日誌保留

當您刪除函數時，不會自動刪除日誌群組。若要避免無限期儲存記錄檔，請刪除記錄群組，或設定保留期間，之後 CloudWatch 會自動刪除記錄檔。若要設定記錄保留，請將下列項目新增至 AWS SAM 範本：

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      # Omitting other properties

  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

使用 Lambda 主控台

您可以在調用 Lambda 函數之後，使用 Lambda 主控台來檢視日誌輸出。

如果可以從內嵌程式碼編輯器測試您的程式碼，您會在執行結果中找到日誌。使用主控台測試功能以調用函數時，您會在詳細資訊區段找到日誌輸出。

使用控 CloudWatch 制台

您可以使用 Amazon 主 CloudWatch 控台來檢視所有 Lambda 函數叫用的日誌。

在 CloudWatch 主控台上檢視記錄檔

1. 在主控台上開啟 [記錄群組] 頁 CloudWatch 面。
2. 選擇您的函數的日誌群組 (/aws/lambda/***your-function-name***)。
3. 選擇日誌串流

每個日誌串流都會對應至函式的執行個體。當您更新 Lambda 函數，以及建立額外執行個體以處理多個並行調用時，便會出現日誌串流。要查找特定調用的日誌，我們建議使用檢測您的函數。 AWS X-Ray X-Ray 會在追蹤內記錄有關請求和日誌串流的詳細資訊。

使用 AWS Command Line Interface （ AWS CLI ）

這 AWS CLI 是一種開放原始碼工具，可讓您使用命令列殼層中的命令與 AWS 服務互動。若要完成本節中的步驟，您必須執行下列各項：

- AWS Command Line Interface (AWS CLI) 第二版
- AWS CLI -快速配置 `aws configure`

您可以透過 AWS CLI，使用 `--log-type` 命令選項來擷取要調用的日誌。其回應將包含 `LogResult` 欄位，內含該次調用的 base64 編碼日誌 (最大達 4 KB)。

Example 擷取日誌 ID

下列範例顯示如何從名稱為 `my-function` 的函數的 `LogResult` 欄位來擷取日誌 ID。

```
aws lambda invoke --function-name my-function out --log-type Tail
```

您應該會看到下列輸出：

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example 解碼日誌

在相同的命令提示中，使用 `base64` 公用程式來解碼日誌。下列範例顯示如何擷取 `my-function` 的 base64 編碼日誌。

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

如果您使用的是 AWS CLI 版本 2，則需要此cli-binary-format選項。若要讓此成為預設的設定，請執行 `aws configure set cli-binary-format raw-in-base64-out`。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

您應該會看到下列輸出：

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

該 base64 公用程式可在 Linux、macOS 和 Ubuntu on Windows 上使用。macOS 使用者可能需要使用 base64 -D。

Example get-logs.sh 指令碼

在相同的命令提示中，使用下列指令碼下載最後五個日誌事件。該指令碼使用 sed 以從輸出檔案移除引述，並休眠 15 秒以使日誌可供使用。輸出包括來自 Lambda 的回應以及來自 get-log-events 命令的輸出。

複製下列程式碼範例的內容，並將您的 Lambda 專案目錄儲存為 get-logs.sh。

如果您使用的是 AWS CLI 版本 2，則需要此cli-binary-format選項。若要讓此成為預設的設定，請執行 aws configure set cli-binary-format raw-in-base64-out。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS 和 Linux (僅限)

在相同的命令提示中，macOS 和 Linux 使用者可能需要執行下列命令，以確保指令碼可執行。

```
chmod -R 755 get-logs.sh
```

Example 擷取最後五個記錄事件

在相同的命令提示中，執行下列指令碼以取得最後五個日誌事件。

```
./get-logs.sh
```

您應該會看到下列輸出：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
```

```
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```


刪除日誌

當您刪除函數時，不會自動刪除日誌群組。若要避免無限期地儲存日誌，請刪除日誌群組，或設定保留期間，系統會在該時間之後自動刪除日誌。

檢測 C# 代碼 AWS Lambda

Lambda 與 AWS X-Ray 整合，可協助您追蹤、偵錯和最佳化 Lambda 應用程式。您可以使用 X-Ray 來追蹤請求，因為它會周遊您應用程式中的資源，其中可能包含 Lambda 函數和其他 AWS 服務。

若要將追蹤資料傳送至 X-Ray，您可以使用以下三個 SDK 庫之一：

- AWS 適用於 OpenTelemetry (ADOT) 的發行版 — 安全、可生產就緒且 AWS支援的 (OTel) SDK 發行版本 OpenTelemetry 。
- 適用於 .NET 的 AWS X-Ray SDK – 用於生成追蹤資料並將其傳送至 X-Ray 的 SDK。
- 適用於 AWS Lambda (.NET) 的 Powertools — 實作無伺服器最佳做法並提高開發人員速度的開發人員工具組。

每個 SDK 均提供將遙測資料傳送至 X-Ray 服務的方法。然後，您可以使用 X-Ray 來檢視、篩選應用程式的效能指標並獲得洞察，從而識別問題和進行最佳化的機會。

> **Important**
>
> 用於 AWS Lambda SDK 的 X-Ray 和 Powertools 是由提供的緊密集成的儀器解決方案的一部分。 AWS ADOT Lambda Layers 是用於追蹤檢測之業界通用標準的一部分，這類檢測一般會收集更多資料，但可能不適用於所有使用案例。您可以使用任一解決方案在 X-Ray 中實作 end-to-end 追蹤。若要深入了解如何在兩者之間做選擇，請參閱在 AWS Distro for OpenTelemetry 和 X-Ray SDK 之間進行選擇。

章節

- 使用動力工具進行 AWS Lambda （ .NET ）和跟 AWS SAM 踪
- 使用 X-Ray SDK 來檢測 .NET 函數
- 透過 Lambda 主控台來啟用追蹤
- 透過 Lambda API 啟用追蹤
- 使用啟動追蹤 AWS CloudFormation
- 解讀 X-Ray 追蹤

使用動力工具進行 AWS Lambda （ .NET ）和跟 AWS SAM 踪

請按照下面的步驟來下載，構建和部署示例你好世界 C# 應用程序與集成 Powertools 的模塊使用 AWS SAM. AWS Lambda 此應用程式實作了基本 API 後端，並使用 Powertools 發送日誌、指標和追蹤。其包含 Amazon API Gateway 端點和 Lambda 函數。當您將 GET 請求傳送至 API Gateway 端點時，Lambda 函數會叫用、使用內嵌指標格式將記錄和指標傳送至 CloudWatch，並將追蹤傳送至 AWS X-Ray。函數會傳回 hello world 訊息。

必要條件

若要完成本節中的步驟，您必須執行下列各項：

- . NET 6 或. NET 8
- AWS CLI 第二版
- AWS SAM CLI 版本 1.75 或更新版本。如果您使用較舊版本的 AWS SAM CLI，請參閱升級 AWS SAM CLI。

部署範例 AWS SAM 應用程式

1. 使用 Hello World TypeScript 範本初始化應用程式。

```
sam init --app-template hello-world-powertools-dotnet --name sam-app --package-type
 Zip --runtime dotnet6 --no-tracing
```

2. 建置應用程式。

```
cd sam-app && sam build
```

3. 部署應用程式。

```
sam deploy --guided
```

4. 依照螢幕上的提示操作。若要接受互動體驗中提供的預設選項，請按下 Enter。

> **Note**
>
> 因為HelloWorldFunction 可能沒有定義授權，這可以嗎？ ，請務必輸入y。

5. 取得已部署應用程式的 URL：

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. 調用 API 端點：

```
curl <URL_FROM_PREVIOUS_STEP>
```

成功的話，您將會看到以下回應：

```
{"message":"hello world"}
```

7. 若要取得函數的追蹤，請執行 sam 追蹤。

```
sam traces
```

追蹤輸出如下：

```
New XRay Service Graph
  Start time: 2023-02-20 23:05:16+08:00
  End time: 2023-02-20 23:05:16+08:00
  Reference Id: 0 - AWS::Lambda - sam-app-HelloWorldFunction-pNjujb7mEoew - Edges:
 [1]
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 2.814
  Reference Id: 1 - AWS::Lambda::Function - sam-app-HelloWorldFunction-pNjujb7mEoew
 - Edges: []
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 2.429
  Reference Id: 2 - (Root) AWS::ApiGateway::Stage - sam-app/Prod - Edges: [0]
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
```

```
      - fault count(5XX): 0
      - total response time: 2.839
  Reference Id: 3 - client - sam-app/Prod - Edges: [2]
   Summary_statistics:
      - total requests: 0
      - ok count(2XX): 0
      - error count(4XX): 0
      - fault count(5XX): 0
      - total response time: 0

XRay Event [revision 3] at (2023-02-20T23:05:16.521000) with id
 (1-63f38c2c-270200bf1d292a442c8e8a00) and duration (2.877s)
 - 2.839s - sam-app/Prod [HTTP: 200]
   - 2.836s - Lambda [HTTP: 200]
 - 2.814s - sam-app-HelloWorldFunction-pNjujb7mEoew [HTTP: 200]
 - 2.429s - sam-app-HelloWorldFunction-pNjujb7mEoew
   - 0.230s - Initialization
   - 2.389s - Invocation
     - 0.600s - ## FunctionHandler
       - 0.517s - Get Calling IP
   - 0.039s - Overhead
```

8. 這是可透過網際網路存取的公有 API 端點。建議您在測試後刪除端點。

```
sam delete
```

X-Ray 無法追蹤應用程式的所有請求。X-Ray 會套用取樣演算法以確保追蹤的效率，同時仍提供所有請求的代表範本。取樣率為每秒 1 次請求和 5% 的額外請求。

> **Note**
>
> 您無法針對函數設定 X-Ray 取樣率。

使用 X-Ray SDK 來檢測 .NET 函數

您可以測試函數代碼以記錄中繼資料並追蹤下游呼叫。若要記錄函數對其他資源和服務所發出呼叫的詳細資訊，請使用 適用於 .NET 的 AWS X-Ray SDK。要取得開發套件，請將 `AWSXRayRecorder` 套件新增到您的專案檔案中。

```
<Project Sdk="Microsoft.NET.Sdk">
```

```
  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <GenerateRuntimeConfigurationFiles>true</GenerateRuntimeConfigurationFiles>
    <AWSProjectType>Lambda</AWSProjectType>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Amazon.Lambda.Core" Version="2.1.0" />
    <PackageReference Include="Amazon.Lambda.SQSEvents" Version="2.1.0" />
    <PackageReference Include="Amazon.Lambda.Serialization.Json" Version="2.1.0" />
    <PackageReference Include="AWSSDK.Core" Version="3.7.103.24" />
    <PackageReference Include="AWSSDK.Lambda" Version="3.7.104.3" />
    <PackageReference Include="AWSXRayRecorder.Core" Version="2.13.0" />
    <PackageReference Include="AWSXRayRecorder.Handlers.AwsSdk" Version="2.11.0" />
  </ItemGroup>
</Project>
```

有一系列 Nuget 軟件包可為 AWS SDK，實體框架和 HTTP 請求提供自動檢測。若要查看完整的組態選項集，請參閱《AWS X-Ray 開發人員指南》中的適用於 .NET 的AWS X-Ray SDK。

新增所需的 Nuget 套件後，請設定自動檢測。最佳實務是在函數的處理常式函數之外執行此設定。這麼做可讓您利用執行環境重新使用來改善函數的效能。在下列程式碼範例中，會在函式建構函式中呼叫`RegisterXRayForAllServices`方法，以便為所有 AWS SDK 呼叫新增分析。

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace GetProductHandler;

public class Function
{
    private readonly IDatabaseRepository _repo;

    public Function()
    {
        // Add auto instrumentation for all AWS SDK calls
        // It is important to call this method before initializing any SDK clients
        AWSSDKHandler.RegisterXRayForAllServices();
        this._repo = new DatabaseRepository();
    }

    public async Task<APIGatewayProxyResponse> FunctionHandler(APIGatewayProxyRequest
 request)
    {
```

```
        var id = request.PathParameters["id"];

        var databaseRecord = await this._repo.GetById(id);

        return new APIGatewayProxyResponse
        {
            StatusCode = (int)HttpStatusCode.OK,
            Body = JsonSerializer.Serialize(databaseRecord)
        };
    }
}
```

透過 Lambda 主控台來啟用追蹤

若要使用控制台在 Lambda 函數上切換主動追蹤，請按照下列步驟操作：

開啟主動追蹤

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇 組態 ，然後選擇 監控和操作工具 。
4. 選擇 編輯 。
5. 在 X-Ray 下，打開 主動追蹤 。
6. 選擇 儲存 。

透過 Lambda API 啟用追蹤

使用 AWS CLI 或 AWS SDK 在 Lambda 函數上設定追蹤功能，並使用下列 API 作業：

- UpdateFunction配置
- GetFunction配置
- CreateFunction

下列範例 AWS CLI 命令可在名為 my- function 的函式上啟用主動追蹤。

```
aws lambda update-function-configuration \
--function-name my-function \
```

```
--tracing-config Mode=Active
```

追蹤模式是您發布函數版本時版本特定組態的一部分。您無法變更已發佈版本上的追蹤模式。

使用啟動追蹤 AWS CloudFormation

若要啟動 AWS CloudFormation 範本中的`AWS::Lambda::Function`資源追蹤，請使用`TracingConfig`屬性。

Example function-inline.yml - 追蹤組態

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

對於 AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` 資源，請使用`Tracing`屬性。

Example template.yml - 追蹤組態

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

解讀 X-Ray 追蹤

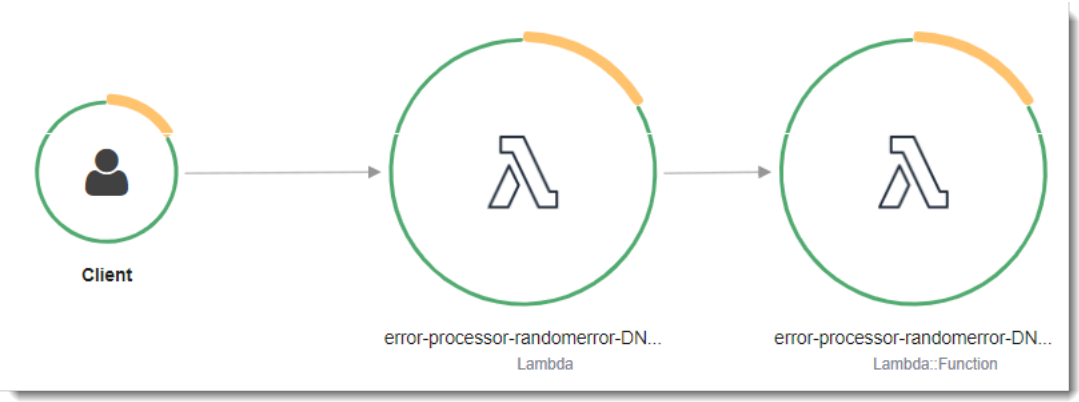
您的函數需要將追蹤資料上傳至 X-Ray 的許可。當您在 Lambda 主控台中啟用追蹤時，Lambda 會將必要的許可新增至函數的執行角色。否則，請將AWSXRayDaemonWriteAccess原則新增至執行角色。

設定主動追蹤之後，您可以透過應用程式來觀察特定請求。X-Ray 服務圖顯示了有關應用程式及其所有元件的資訊。下圖演示了具有兩個功能的應用程序。主要函式會處理事件，有時會傳回錯誤。頂部的第二個函數處理出現在第一個日誌組中的錯誤，並使用 AWS SDK 調用 X-Ray，Amazon 簡單存儲服務（Amazon S3）和亞馬遜 CloudWatch 日誌。

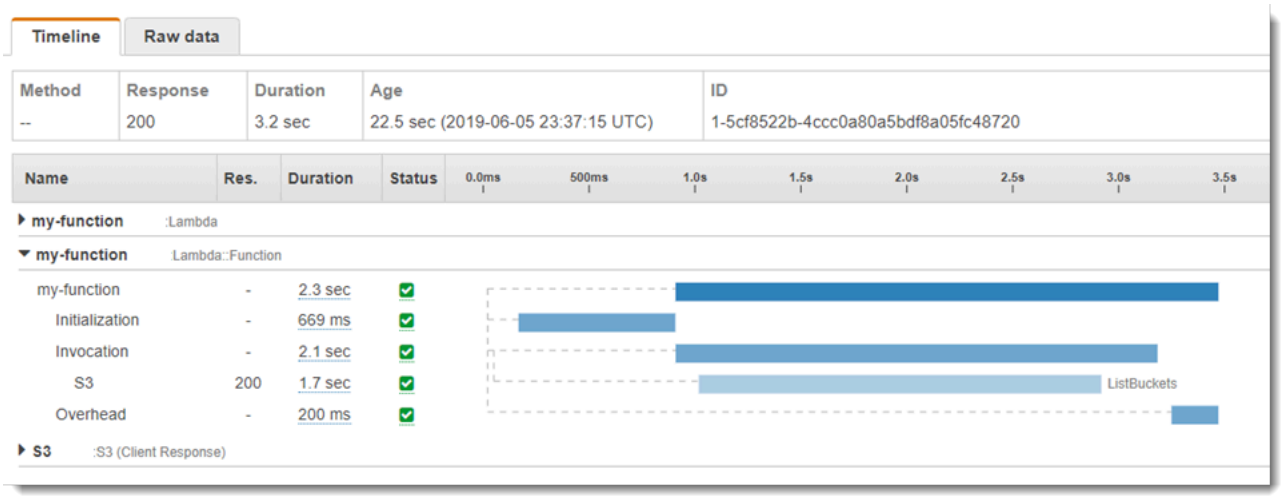
X-Ray 無法追蹤應用程式的所有請求。X-Ray 會套用取樣演算法以確保追蹤的效率，同時仍提供所有請求的代表範本。取樣率為每秒 1 次請求和 5% 的額外請求。

> **Note**
>
> 您無法針對函數設定 X-Ray 取樣率。

在 X-Ray 中，追蹤會記錄一或多個服務所處理之要求的相關資訊。Lambda 會記錄每個追蹤 2 個區段，在服務圖表上建立兩個節點。下列影像會強調顯示這兩個節點：



左側第一個節點代表接收調用請求的 Lambda 服務。第二個節點代表您特定的 Lambda 函數。下列範例顯示了具有這 2 個區段的追蹤。兩者都被命名為我的函數，但一個具有的起源`AWS::Lambda`，另一個具有的`AWS::Lambda::Function`起源。如果`AWS::Lambda`區段顯示錯誤，表示 Lambda 服務發生問題。如果`AWS::Lambda::Function`區段顯示錯誤，表示您的函數發生問題。



此範例會展開區AWS::Lambda::Function段，以顯示其三個子區段：

- 初始化 - 表示載入函數和執行初始化程式碼所花費的時間。只有函數的每個執行個體所處理的第一個事件會顯示此子區段。
- 調用 – 表示執行處理常式程式碼所花費的時間。
- 額外負荷 - 表示 Lambda 執行期為做好準備以處理下一個事件所花費的時間。

您也可以檢測 HTTP 用戶端、記錄 SQL 查詢，以及建立具有註釋和中繼資料的自訂子區段。如需詳細資訊，請參閱《AWS X-Ray 開發人員指南》中的 適用於 .NET 的 AWS X-Ray SDK 許可。

> **定價**
>
> 作為免費方案的一部分，您可以每月免費使用 X-Ray 追蹤，最多達到一定限制。 AWS 達到閾值後，X-Ray 會收取追蹤儲存及擷取的費用。如需詳細資訊，請參閱 AWS X-Ray 定價。

以 C# 測試 AWS Lambda 函數

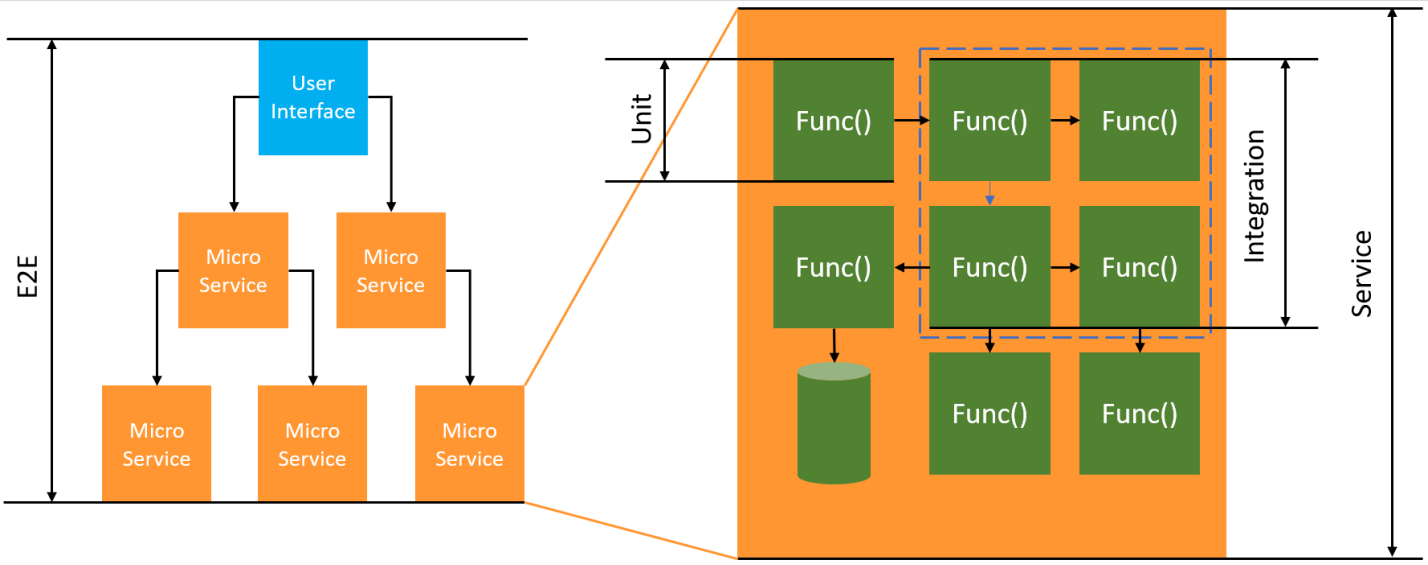
> **Note**
>
> 如需測試無伺服器解決方案之技術和最佳實務的完整介紹，請參閱測試函數章節。

測試無伺服器函數會使用傳統的測試類型和技術，但您也必須考慮測試整個無伺服器應用程式。以雲端為基礎的測試會為您的函數和無伺服器應用程式提供最準確的品質測量標準。

無伺服器應用程式架構包括透過 API 呼叫提供關鍵應用程式功能的受管服務。因此，您的開發週期應包括自動化測試，以便在函數和服務互動時驗證功能。

如果您未建立以雲端為基礎的測試，則可能會因本機環境與部署環境之間的差異而遇到問題。您的持續整合程序應先針對雲端佈建的一組資源進行測試，然後再將程式碼升級至下一個部署環境 (例如 QA、暫存或生產環境)。

繼續閱讀這份簡短指南，了解無伺服器應用程式的測試策略，或造訪無伺服器測試範例儲存庫，深入了解所選語言和執行期的特定實際範例。



對於無服務器測試，您仍然會編寫單元，集成和end-to-end測試。

- 單元測試：針對一組隔離的程式碼區塊進行的測試。例如，驗證商業邏輯以計算指定的特定項目與目的地的運費。
- 整合測試：涉及到兩個以上元件或服務進行互動的測試 (通常在雲端環境)。例如，驗證函數是否有處理佇列中的事件。

- E nd-to-end 測試-測試，驗證整個應用程序的行為。例如，確保基礎設施的設定正確無誤，以及事件如預期在服務之間流動，以記錄客戶的訂單。

測試無伺服器應用程式

通常會混合使用多種方法來測試無伺服器應用程式程式碼，包括在雲端進行測試、透過模擬物件進行測試，以及偶爾使用模擬器進行測試。

在雲端進行測試

雲端測試對於測試的所有階段都很有價值，包括單元測試、整合測試和 end-to-end 測試。您可以針對部署在雲端中的程式碼執行測試，並與雲端服務互動。這是最準確的程式碼品質測量方法。

您可以透過主控台使用測試事件，輕鬆在雲端對 Lambda 函數進行偵錯。一個測試事件是函數的 JSON 輸入。如果您的函數不需要輸入，該事件可以是空白的 JSON 文件 (`{}`)。主控台提供各種服務整合的範例事件。在主控台中建立事件後，您可以與團隊分享事件，讓測試變得更容易，結果更一致。

> **Note**
>
> 在控制台中測試函數是簡便快速的入門方式，而將測試週期自動化可確保應用程式的品質和開發速度。

測試工具

為了加速開發週期，您可以在測試函數時使用多種工具和技巧。例如，AWS SAM Accelerate 和 AWS CDK 監看模式都可以縮短更新雲端環境所需的時間。

您定義 Lambda 函數程式碼的方式，讓您可輕鬆加入單元測試。Lambda 需要使用公有無參數建構函數來初始化類別。引入第二個內部建構函數，可讓您控制應用程式使用的相依項。

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace GetProductHandler;

public class Function
{
    private readonly IDatabaseRepository _repo;
```

```
    public Function(): this(null)
    {
    }

    internal Function(IDatabaseRepository repo)
    {
        this._repo = repo ?? new DatabaseRepository();
    }

    public async Task<APIGatewayProxyResponse> FunctionHandler(APIGatewayProxyRequest
 request)
    {
        var id = request.PathParameters["id"];

        var databaseRecord = await this._repo.GetById(id);

        return new APIGatewayProxyResponse
        {
            StatusCode = (int)HttpStatusCode.OK,
            Body = JsonSerializer.Serialize(databaseRecord)
        };
    }
}
```

若要為此函數編寫測試，您可以初始化 Function 類別的新執行個體，並傳入 IDatabaseRepository 的模擬實作。以下範例使用 XUnit、Moq 和 FluentAssertions 來編寫簡單的測試，確保 FunctionHandler 會傳回 200 狀態碼。

```
using Xunit;
using Moq;
using FluentAssertions;

public class FunctionTests
{
    [Fact]
    public async Task TestLambdaHandler_WhenInputIsValid_ShouldReturn200StatusCode()
    {
        // Arrange
        var mockDatabaseRepository = new Mock<IDatabaseRepository>();

        var functionUnderTest = new Function(mockDatabaseRepository.Object);

        // Act
```

```
        var response = await functionUnderTest.FunctionHandler(new
 APIGatewayProxyRequest());

        // Assert
        response.StatusCode.Should().Be(200);
    }
}
```

如需更詳細的範例，包括非同步測試的範例，請參閱上的 .NET test 範例儲存庫 GitHub。

使用建置 Lambda 函數 PowerShell

以下各節說明在中編寫 Lambda 函數程式碼時，常見的程式設計模式和核心概念如何套用 PowerShell。

Lambda 提供下列範例應用程式 PowerShell：

- 空白電源外殼-顯示使用日誌記錄，環境變量和 SDK 的 PowerShell 函數。 AWS

在開始之前，您必須先設定開 PowerShell 發環境。如需如何執行此動作的詳細資訊，請參閱設定 PowerShell 開發環境。

若要瞭解如何使用 AWSLambdaPSCore 模組從範本下載範例 PowerShell 專案、建立 PowerShell 部署套件，以及將 PowerShell 功能部署到 AWS 雲端，請參閱使用 .zip 檔案封存部署 PowerShell Lambda 函數。

Lambda 提供適用於 .NET 語言的以下執行期：

.NET

名稱	識別符	作業系統	取代日期	封鎖函數建立	封鎖函數更新
。淨值 8	dotnet8	Amazon Linux 2023			
.NET 6	dotnet6	Amazon Linux 2	2024 年 11 月 12 日	2025年2月28日	2025年3月31日

主題

- 設定 PowerShell 開發環境
- 使用 .zip 檔案封存部署 PowerShell Lambda 函數
- 定義 Lambda 函數處理常式 PowerShell
- AWS Lambda上下文對象 PowerShell
- AWS Lambda 功能登錄 PowerShell

設定 PowerShell 開發環境

Lambda 為 PowerShell 執行階段提供了一組工具和程式庫。有關安裝說明，請參閱（詳見）的 PowerShell Lambda 工具 GitHub。

此 AWSLambdaPSCore 模組包含下列指令程式，可協助撰寫和發佈 PowerShell Lambda 函數：

- 取得 AWSPowerShellLambdaTemplate- 傳回入門範本的清單。
- 新增-AWSPowerShellLambda — 根據範本建立初始 PowerShell 指令碼。
- 發佈 AWSPowerShellLambda- 將指定的指 PowerShell 令碼發佈至 Lambda。
- 新增 AWSPowerShellLambdaPackage — 建立 Lambda 部署套件，您可以在 CI/CD 系統中使用該套件進行部署。

使用 .zip 檔案封存部署 PowerShell Lambda 函數

PowerShell 執行階段的部署套件包含您的 PowerShell 指令碼、指 PowerShell 令碼所需的 PowerShell 模組，以及裝載 PowerShell Core 所需的組件。

建立 Lambda 函數

若要開始使用 Lambda 撰寫和叫用指令 PowerShell 碼，您可以使用指New-AWSPowerShellLambda令程式根據範本建立入門指令碼。您可以使用 Publish-AWSPowerShellLambda cmdlet 將指令碼部署至 Lambda。然後，您可以透過命令列或 Lambda 主控台測試指令碼。

若要建立新 PowerShell 指令碼，請上傳並進行測試，請執行下列動作：

1. 若要檢視可用範本的清單，請執行以下命令：

```
PS C:\> Get-AWSPowerShellLambdaTemplate

Template               Description
--------               -----------
Basic                  Bare bones script
CodeCommitTrigger      Script to process AWS CodeCommit Triggers
...
```

2. 若要根據 Basic 範本建立範例指令碼，請執行下列命令：

```
New-AWSPowerShellLambda -ScriptName MyFirstPSScript -Template Basic
```

一個名為 MyFirstPSScript.ps1 的新檔案已建立在目前目錄下的新子目錄中。該目錄的名稱依據 -ScriptName 參數而定。您可以使用 -Directory 參數來選擇另一個目錄。

您可以看到新的檔案包含下列內容：

```
# PowerShell script file to run as a Lambda function
#
# When executing in Lambda the following variables are predefined.
#   $LambdaInput - A PSObject that contains the Lambda function input data.
#   $LambdaContext - An Amazon.Lambda.Core.ILambdaContext object that contains
 information about the currently running Lambda environment.
#
```

```
# The last item in the PowerShell pipeline is returned as the result of the Lambda
 function.
#
# To include PowerShell modules with your Lambda function, like the
 AWSPowerShell.NetCore module, add a "#Requires" statement
# indicating the module and version.

#Requires -Modules @{ModuleName='AWSPowerShell.NetCore';ModuleVersion='3.3.618.0'}

# Uncomment to send the input to CloudWatch Logs
# Write-Host (ConvertTo-Json -InputObject $LambdaInput -Compress -Depth 5)
```

3. 若要查看 PowerShell 指令碼中的記錄訊息傳送到 Amazon CloudWatch Logs 的方式，請取消註解範例指令碼`Write-Host`行。

   若要示範如何從您的 Lambda 函數傳回資料，請在指令碼的結尾處以 `$PSVersionTable` 新增一行。這會將「」新增`$PSVersionTable`至 PowerShell 配管。 PowerShell 指令碼完成後，PowerShell 管線中的最後一個物件就是 Lambda 函數的傳回資料。 `$PSVersionTable`是一個 PowerShell 全局變量，還提供有關運行環境的信息。

   完成這些變更之後，最後兩行的範例指令碼應類似：

   ```
   Write-Host (ConvertTo-Json -InputObject $LambdaInput -Compress -Depth 5)
   $PSVersionTable
   ```

4. 在您編輯 `MyFirstPSScript.ps1` 檔案之後，請將目錄變更為指令碼的位置。然後執行下列命令，將指令碼發佈至 Lambda：

   ```
   Publish-AWSPowerShellLambda -ScriptPath .\MyFirstPSScript.ps1 -Name
    MyFirstPSScript -Region us-east-2
   ```

   請注意，`-Name` 參數指定 Lambda 函數名稱，此名稱會出現在 Lambda 主控台。您可以使用此函式來手動叫用您的指令碼。

5. 使用 AWS Command Line Interface (AWS CLI) `invoke` 命令來叫用函數。

   ```
   > aws lambda invoke --function-name MyFirstPSScript out
   ```

定義 Lambda 函數處理常式 PowerShell

當叫用 Lambda 函數時，Lambda 處理常式會叫用指 PowerShell 令碼。

呼叫指 PowerShell 令碼時，會預先定義下列變數：

- *$ LambdaInput* — 包含處理常式輸入的 PSObject。此輸入可以是事件資料 (由事件來源發佈) 或您提供的自訂輸入，例如字串或任何自訂資料物件。
- *$ LambdaContext* — Amazon.Lambda.Core.I LambdaContext 物件，您可以使用此物件來存取目前呼叫的相關資訊，例如目前函數的名稱、記憶體限制、剩餘執行時間和記錄。

例如，請考慮下列 PowerShell 範例程式碼。

```
#Requires -Modules @{ModuleName='AWSPowerShell.NetCore';ModuleVersion='3.3.618.0'}
Write-Host 'Function Name:' $LambdaContext.FunctionName
```

此指令碼會傳回從 $ LambdaContext 變數取得的 FunctionName 屬性。

> **Note**
>
> 您必須使用指 PowerShell 令碼中的`#Requires`陳述式來指出指令碼所依賴的模組。此陳述式執行兩個重要任務。1）它與其他開發人員溝通腳本使用哪些模塊，並 2）它標識 AWS PowerShell 工具需要與腳本打包的依賴模塊，作為部署的一部分。如需有關中`#Requires`陳述式的詳細資訊 PowerShell，請參閱關於需求。如需 PowerShell 部署套件的詳細資訊，請參閱使用 .zip 檔案封存部署 PowerShell Lambda 函數。
>
> 當您的 PowerShell Lambda 函數使用 AWS PowerShell 指令程式時，請務必設定參考`AWSPowerShell.NetCore`模組的`#Requires`陳述式，該陳述式可支援 PowerShell 核心，而不是僅支援 Windows 的`AWSPowerShell`模組。 PowerShell此外，請務必使用版本 3.3.270.0 或更新版本的 `AWSPowerShell.NetCore`，它可最佳化 cmdlet 匯入程序。如果您使用舊版本，將面臨較長的冷啟動時間。如需詳細資訊，請參閱AWS 適用於 PowerShell 的工具。

傳回資料

有些 Lambda 叫用旨在將資料傳回至呼叫者。例如，如果叫用是為了回應來自 API Gateway 的 Web 請求，則我們的 Lambda 函數必須傳回回應。對於 PowerShell Lambda，新增至 PowerShell 管線的

最後一個物件是來自 Lambda 叫用的傳回資料。如果該物件為字串，將以其原樣傳回。否則，會使用 `ConvertTo-Json` cmdlet 將該物件轉換為 JSON。

例如，請考慮下列新增`$PSVersionTable`至 PowerShell 管線的 PowerShell 陳述式：

```
$PSVersionTable
```

PowerShell 指令碼完成後， PowerShell 管線中的最後一個物件就是 Lambda 函數的傳回資料。`$PSVersionTable`是一個 PowerShell 全局變量，還提供有關運行環境的信息。

AWS Lambda上下文對象 PowerShell

當 Lambda 執行您的函數時，它會傳遞內容資訊，方法是讓 $LambdaContext 變數可用於處理常式。此變數提供的方法和各項屬性包含了有關叫用、函式以及執行環境的資訊。

內容屬性

- FunctionName – Lambda 函數的名稱。
- FunctionVersion – 函數的版本。
- InvokedFunctionArn - 用於調用此函數的 Amazon Resource Name (ARN)。指出調用者是否指定版本號或別名。
- MemoryLimitInMB - 分配給函數的記憶體數量。
- AwsRequestId - 調用請求的識別符。
- LogGroupName - 函數的日誌群組。
- LogStreamName - 函數執行個體的記錄串流。
- RemainingTime - 執行逾時前剩餘的毫秒數。
- Identity - (行動應用程式) 已授權請求的 Amazon Cognito 身分的相關資訊。
- ClientContext - (行動應用程式) 用戶端應用程式提供給 Lambda 的用戶端內容。
- Logger - 函數的 Logger 物件。

下面的 PowerShell 代碼片段顯示了一個簡單的處理程序函數，打印一些上下文信息。

```
#Requires -Modules @{ModuleName='AWSPowerShell.NetCore';ModuleVersion='3.3.618.0'}
Write-Host 'Function name:' $LambdaContext.FunctionName
Write-Host 'Remaining milliseconds:' $LambdaContext.RemainingTime.TotalMilliseconds
Write-Host 'Log group name:' $LambdaContext.LogGroupName
Write-Host 'Log stream name:' $LambdaContext.LogStreamName
```

AWS Lambda 功能登錄 PowerShell

AWS Lambda 代表您自動監控 Lambda 函數，並將日誌傳送到 Amazon CloudWatch。您的 Lambda 函數隨附一個 CloudWatch 日誌記錄群組和函數每個執行個體的日誌串流。Lambda 執行期環境會將每次調用的詳細資訊傳送至日誌串流，並且轉傳來自函數程式碼的日誌及其他輸出。如需詳細資訊，請參閱 使用 Amazon CloudWatch 日誌 AWS Lambda。

本頁說明如何從 Lambda 函數的程式碼產生記錄輸出，或使用 Lambda 主控台或主控台存取 CloudWatch 日誌。 AWS Command Line Interface

章節

建立傳回日誌的函數

若要從函數程式碼輸出記錄檔，您可以在 Microsoft 上使用指令程式。 PowerShell.Utility，或任何寫入`stdout`或`stderr`的記錄模組。以下範例使用 `Write-Host`。

Example function/Handler.ps1 - 記錄

```
#Requires -Modules @{ModuleName='AWSPowerShell.NetCore';ModuleVersion='3.3.618.0'}
Write-Host `## Environment variables
Write-Host AWS_LAMBDA_FUNCTION_VERSION=$Env:AWS_LAMBDA_FUNCTION_VERSION
Write-Host AWS_LAMBDA_LOG_GROUP_NAME=$Env:AWS_LAMBDA_LOG_GROUP_NAME
Write-Host AWS_LAMBDA_LOG_STREAM_NAME=$Env:AWS_LAMBDA_LOG_STREAM_NAME
Write-Host AWS_EXECUTION_ENV=$Env:AWS_EXECUTION_ENV
Write-Host AWS_LAMBDA_FUNCTION_NAME=$Env:AWS_LAMBDA_FUNCTION_NAME
Write-Host PATH=$Env:PATH
Write-Host `## Event
Write-Host (ConvertTo-Json -InputObject $LambdaInput -Compress -Depth 3)
```

Example 記錄格式

```
START RequestId: 56639408-xmpl-435f-9041-ac47ae25ceed Version: $LATEST
```

```
Importing module ./Modules/AWSPowerShell.NetCore/3.3.618.0/AWSPowerShell.NetCore.psd1
[Information] - ## Environment variables
[Information] - AWS_LAMBDA_FUNCTION_VERSION=$LATEST
[Information] - AWS_LAMBDA_LOG_GROUP_NAME=/aws/lambda/blank-powershell-
function-18CIXMPLHFAJJ
[Information] - AWS_LAMBDA_LOG_STREAM_NAME=2020/04/01/
[$LATEST]53c5xmpl52d64ed3a744724d9c201089
[Information] - AWS_EXECUTION_ENV=AWS_Lambda_dotnet6_powershell_1.0.0
[Information] - AWS_LAMBDA_FUNCTION_NAME=blank-powershell-function-18CIXMPLHFAJJ
[Information] - PATH=/var/lang/bin:/usr/local/bin:/usr/bin/:/bin:/opt/bin
[Information] - ## Event
[Information] -
{
    "Records": [
        {
            "messageId": "19dd0b57-b21e-4ac1-bd88-01bbb068cb78",
            "receiptHandle": "MessageReceiptHandle",
            "body": "Hello from SQS!",
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1523232000000",
                "SenderId": "123456789012",
                "ApproximateFirstReceiveTimestamp": "1523232000001"
            },
            ...
END RequestId: 56639408-xmpl-435f-9041-ac47ae25ceed
REPORT RequestId: 56639408-xmpl-435f-9041-ac47ae25ceed Duration: 3906.38 ms Billed
 Duration: 4000 ms Memory Size: 512 MB Max Memory Used: 367 MB Init Duration: 5960.19
 ms
XRAY TraceId: 1-5e843da6-733cxmple7d0c3c020510040 SegmentId: 3913xmpl20999446 Sampled:
 true
```

.NET 執行期會記錄每次調用的 START、END 和 REPORT 行。報告明細行提供下列詳細資訊。

REPORT 行資料欄位

- RequestId— 呼叫的唯一要求 ID。
- 持續時間 - 函數的處理常式方法處理事件所花費的時間量。
- 計費持續時間 - 調用的計費時間量。
- 記憶體大小 - 分配給函數的記憶體數量。
- 使用的記憶體上限 - 函數所使用的記憶體數量。

- 初始化持續時間 - 對於第一個提供的請求，這是執行期載入函數並在處理常式方法之外執行程式碼所花費的時間量。
- XRAY TraceId — 針對追蹤的要求，則為AWS X-Ray 追蹤識別碼。
- SegmentId— 針對追蹤的請求，X-Ray 區段 ID。
- 已取樣 - 對於追蹤的請求，這是取樣結果。

使用 Lambda 主控台

您可以在調用 Lambda 函數之後，使用 Lambda 主控台來檢視日誌輸出。

如果可以從內嵌程式碼編輯器測試您的程式碼，您會在執行結果中找到日誌。使用主控台測試功能以調用函數時，您會在詳細資訊區段找到日誌輸出。

使用控 CloudWatch 制台

您可以使用 Amazon 主 CloudWatch 控台來檢視所有 Lambda 函數叫用的日誌。

在 CloudWatch 主控台上檢視記錄檔

1. 在主控台上開啟 [記錄群組] 頁 CloudWatch 面。
2. 選擇您的函數的日誌群組 (/aws/lambda/***your-function-name***)。
3. 選擇日誌串流

每個日誌串流都會對應至函式的執行個體。當您更新 Lambda 函數，以及建立額外執行個體以處理多個並行調用時，便會出現日誌串流。要查找特定調用的日誌，我們建議使用檢測您的函數。 AWS X-Ray X-Ray 會在追蹤內記錄有關請求和日誌串流的詳細資訊。

使用 AWS Command Line Interface （ AWS CLI ）

這 AWS CLI 是一種開放原始碼工具，可讓您使用命令列殼層中的命令與 AWS 服務互動。若要完成本節中的步驟，您必須執行下列各項：

- AWS Command Line Interface (AWS CLI) 第二版
- AWS CLI -快速配置 `aws configure`

您可以透過 AWS CLI，使用 `--log-type` 命令選項來擷取要調用的日誌。其回應將包含 `LogResult` 欄位，內含該次調用的 base64 編碼日誌 (最大達 4 KB)。

Example 擷取日誌 ID

下列範例顯示如何從名稱為 my-function 的函數的 LogResult 欄位來擷取日誌 ID。

```
aws lambda invoke --function-name my-function out --log-type Tail
```

您應該會看到下列輸出：

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example 解碼日誌

在相同的命令提示中，使用 base64 公用程式來解碼日誌。下列範例顯示如何擷取 my-function 的 base64 編碼日誌。

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

如果您使用的是 AWS CLI 版本 2，則需要此cli-binary-format選項。若要讓此成為預設的設定，請執行 aws configure set cli-binary-format raw-in-base64-out。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

您應該會看到下列輸出：

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

該 base64 公用程式可在 Linux、macOS 和 Ubuntu on Windows 上使用。macOS 使用者可能需要使用 base64 -D。

Example get-logs.sh 指令碼

在相同的命令提示中，使用下列指令碼下載最後五個日誌事件。該指令碼使用 sed 以從輸出檔案移除引述，並休眠 15 秒以使日誌可供使用。輸出包括來自 Lambda 的回應以及來自 get-log-events 命令的輸出。

複製下列程式碼範例的內容，並將您的 Lambda 專案目錄儲存為 get-logs.sh。

如果您使用的是 AWS CLI 版本 2，則需要此cli-binary-format選項。若要讓此成為預設的設定，請執行 aws configure set cli-binary-format raw-in-base64-out。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS 和 Linux (僅限)

在相同的命令提示中，macOS 和 Linux 使用者可能需要執行下列命令，以確保指令碼可執行。

```
chmod -R 755 get-logs.sh
```

Example 擷取最後五個記錄事件

在相同的命令提示中，執行下列指令碼以取得最後五個日誌事件。

```
./get-logs.sh
```

您應該會看到下列輸出：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
```

```
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```

刪除日誌

當您刪除函數時，不會自動刪除日誌群組。若要避免無限期地儲存日誌，請刪除日誌群組，或設定保留期間，系統會在該時間之後自動刪除日誌。

使用 Rust 建置 Lambda 函數

由於 Rust 會編譯成原生程式碼，因此您不需要專用的執行期即可在 Lambda 上執行 Rust 程式碼。而是使用 Rust 執行期用戶端在本機建置專案，然後使用 `provided.al2023` 或 `provided.al2` 執行期將其部署到 Lambda。當您使用 `provided.al2023` 或 `provided.al2` 時，Lambda 會自動使作業系統與最新修補程式保持最新狀態。

> **Note**
>
> Rust 執行期用戶端是實驗性套件。它可能會發生變更，僅用於評估目的。

適用於 Rust 的工具和程式庫

- 適用於 Rust 的 AWS SDK：Rust 的 AWS 開發套件提供 Rust API 來與亞馬遜網路服務基礎設施服務互動。
- Lambda 的 Rust 執行期用戶端：Rust 執行期用戶端是實驗性套件。SDK 可能會發生重大變更，不建議用於生產環境。
- Cargo Lambda：此程式庫提供命令列應用程式來處理使用 Rust 建置的 Lambda 函數。
- Lambda HTTP：此程式庫提供一個包裝程式來處理 HTTP 事件。
- Lambda 延伸：此程式庫可支援使用 Rust 撰寫的 Lambda 延伸。
- AWS Lambda 事件：此程式庫提供常見事件來源整合的類型定義。

Rust 的範本 Lambda 應用程式

- 基本 Lambda 函數：顯示如何處理基本事件的 Rust 函數。
- 具有錯誤處理功能的 Lambda 函數：示範如何在 Lambda 中處理自訂 Rust 錯誤的 Rust 函數。
- 含有共用資源的 Lambda 函數：可在建立 Lambda 函數之前初始化共用資源的 Rust 專案。
- Lambda HTTP 事件：處理 HTTP 事件的 Rust 函數。
- 帶有 CORS 標頭的 Lambda HTTP 事件：使用 Tower 插入 CORS 標頭的 Rust 函數。
- Lambda REST API：使用 Axum 和 Diesel 連線至 PostgreSQL 資料庫的 REST API。
- 無伺服器 Rust 示範：Rust 專案，顯示 Lambda Rust 程式庫、記錄、環境變數和 AWS SDK 的使用方式。
- 基本 Lambda 延伸：顯示如何處理基本延伸事件的 Rust 延伸。

- Lambda 記錄 Amazon 資料 Firehose 擴充功能：Rust 擴充功能，顯示如何將 Lambda 日誌傳送至 Firehose。

主題

- 在 Rust Lambda 定義函數處理程序
- Rust 中的 Lambda 內容物件
- 使用 Rust 處理 HTTP 事件
- 使用 .zip 封存檔部署 Rust Lambda 函數
- Rust 中的 Lambda 函數日誌記錄

在 Rust Lambda 定義函數處理程序

> **Note**
>
> Rust 執行期用戶端是實驗性套件。它可能會發生變更，僅用於評估目的。

Lambda 函數處理常式是您的函數程式碼中處理事件的方法。當有人呼叫您的函數時，Lambda 會執行處理常式方法。函數會執行，直到處理常式傳回回應、結束或逾時為止。

將您的 Lambda 函數程式碼編寫為 Rust 可執行檔。實作處理常式函數程式碼和主函數，並包含以下內容：

- 來自 crates.io 的 lambda_runtime 套件，它可實作 Rust 的 Lambda 程式設計模型。
- 將 Tokio 包含在相依項中。Lambda 的 Rust 執行期用戶端使用 Tokio 來處理非同步呼叫。

Example – 處理 JSON 事件的 Rust 處理常式

下列範例使用 serde_json 套件來處理基礎 JSON 事件：

```
use lambda_runtime::{service_fn, LambdaEvent, Error};
use serde_json::{json, Value};

async fn handler(event: LambdaEvent<Value>) -> Result<Value, Error> {
 let payload = event.payload;
 let first_name = payload["firstName"].as_str().unwrap_or("world");
 Ok(json!({ "message": format!("Hello, {first_name}!") }))
}

#[tokio::main]
async fn main() -> Result<(), Error> {
 lambda_runtime::run(service_fn(handler)).await
}
```

注意下列事項：

- `use`：匯入 Lambda 函數所需的程式庫。
- `async fn main`：執行 Lambda 函數程式碼的進入點。Rust 執行期用戶端使用 Tokio 作為異步執行期，因此您必須使用 `#[tokio::main]` 來標註主函數。

- `async fn handler(event: LambdaEvent<Value>) -> Result<Value,Error>`：這是 Lambda 處理常式簽章。它包含叫用函數時執行的程式碼。
  - `LambdaEvent<Value>`：這是一個一般類型，描述 Lambda 執行期接收的事件以及 Lambda 函數內容。
  - `Result<Value, Error>`：該函數會傳回 `Result` 類型。如果函數成功，則結果為 JSON 值。如果函數不成功，則結果為錯誤。

使用共用狀態

您可以宣告獨立於 Lambda 函數處理常式程式碼的共用變數。這些變數可協助您在函數接收任何事件之前在 初始化階段 過程中載入狀態資訊。

Example – 跨函數執行個體共用 Amazon S3 用戶端

注意下列事項：

- `use aws_sdk_s3::Client`：此範例要求您將 `aws-sdk-s3 = "0.26.0"` 新增到 `Cargo.toml` 檔案中的相依項清單。
- `aws_config::from_env`：此範例要求您將 `aws-config = "0.55.1"` 新增到 `Cargo.toml` 檔案中的相依項清單。

```
use aws_sdk_s3::Client;
use lambda_runtime::{service_fn, Error, LambdaEvent};
use serde::{Deserialize, Serialize};

#[derive(Deserialize)]
struct Request {
    bucket: String,
}

#[derive(Serialize)]
struct Response {
    keys: Vec<String>,
}

async fn handler(client: &Client, event: LambdaEvent<Request>) -> Result<Response,
 Error> {
    let bucket = event.payload.bucket;
    let objects = client.list_objects_v2().bucket(bucket).send().await?;
```

```
    let keys = objects
        .contents()
        .map(|s| s.iter().flat_map(|o| o.key().map(String::from)).collect())
        .unwrap_or_default();
    Ok(Response { keys })
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    let shared_config = aws_config::from_env().load().await;
    let client = Client::new(&shared_config);
    let shared_client = &client;
    lambda_runtime::run(service_fn(move |event: LambdaEvent<Request>| async move {
        handler(&shared_client, event).await
    }))
    .await
}
```

Rust 中的 Lambda 內容物件

> **Note**
>
> Rust 執行期用戶端是實驗性套件。它可能會發生變更，僅用於評估目的。

Lambda 執行函數時，會將內容物件新增至處理常式接收 LambdaEvent 的內容物件。此物件提供的各項屬性包含了有關叫用、函式以及執行環境的資訊。

內容屬性

- `request_id`：Lambda 服務產生的 AWS 請求 ID。
- `deadline`：目前叫用的執行期限，以毫秒為單位。
- `invoked_function_arn`：被叫用之 Lambda 函數的 Amazon Resource Name (ARN)。
- `xray_trace_id`：目前叫用的 AWS X-Ray 追蹤 ID。
- `client_content`：由 AWS Mobile SDK 傳送的用戶端內容物件。除非使用 AWS Mobile SDK 叫用函數，否則此欄位為空。
- `identity`：叫用該函數的 Amazon Cognito 身分。除非使用 Amazon Cognito 身分集區發出的 AWS 憑證對 Lambda API 進行叫用請求，否則此欄位為空。
- `env_config`：來自本機環境變數的 Lambda 函數組態。此屬性包括諸如函數名稱、記憶體分配、版本和日誌串流等資訊。

存取叫用內容資訊

Lambda 函數有權存取有關其環境和叫用請求的中繼資料。函數處理常式接收的 `LambaEvent` 物件包含 `context` 中繼資料：

```
use lambda_runtime::{service_fn, LambdaEvent, Error};
use serde_json::{json, Value};

async fn handler(event: LambdaEvent<Value>) -> Result<Value, Error> {
    let invoked_function_arn = event.context.invoked_function_arn;
    Ok(json!({ "message": format!("Hello, this is function
 {invoked_function_arn}!") }))
}
```

```
#[tokio::main]
async fn main() -> Result<(), Error> {
    lambda_runtime::run(service_fn(handler)).await
}
```

使用 Rust 處理 HTTP 事件

> **Note**
>
> Rust 執行期用戶端是實驗性套件。它可能會發生變更，僅用於評估目的。

Amazon API Gateway API、Application Load Balancer 以及 Lambda 函數 URL 可將 HTTP 事件傳送至 Lambda。您可以使用來自 crates.io 的 aws_lambda_events 套件來處理來自這些來源的事件。

Example – 處理 API Gateway 代理請求

注意下列事項：

- `use aws_lambda_events::apigw::{ApiGatewayProxyRequest, ApiGatewayProxyResponse}`：aws_lambda_events 套件包含許多 Lambda 事件。為了減少編譯時間，請使用功能標誌來激活所需的事件。範例：`aws_lambda_events = { version = "0.8.3", default-features = false, features = ["apigw"] }`。
- `use http::HeaderMap`：此匯入要求您將 http 套件新增到相依項。

```
use aws_lambda_events::apigw::{ApiGatewayProxyRequest, ApiGatewayProxyResponse};
use http::HeaderMap;
use lambda_runtime::{service_fn, Error, LambdaEvent};

async fn handler(
    _event: LambdaEvent<ApiGatewayProxyRequest>,
) -> Result<ApiGatewayProxyResponse, Error> {
    let mut headers = HeaderMap::new();
    headers.insert("content-type", "text/html".parse().unwrap());
    let resp = ApiGatewayProxyResponse {
        status_code: 200,
        multi_value_headers: headers.clone(),
        is_base64_encoded: false,
        body: Some("Hello AWS Lambda HTTP request".into()),
        headers,
    };
    Ok(resp)
}

#[tokio::main]
```

```
async fn main() -> Result<(), Error> {
    lambda_runtime::run(service_fn(handler)).await
}
```

Lambda 的 Rust 執行期用戶端也提供這些事件類型的抽象表示，讓您可以使用原生 HTTP 類型，而不論是哪個服務傳送事件。下列程式碼等同於前一個範例，而且可直接使用 Lambda 函數 URL、Application Load Balancer 和 API Gateway。

> **Note**
>
> 該 lambda_http 套件使用下面的 lambda_runtime 套件。不必單獨匯入 `lambda_runtime`。

Example – 處理 HTTP 請求

```
use lambda_http::{service_fn, Error, IntoResponse, Request, RequestExt, Response};

async fn handler(event: Request) -> Result<impl IntoResponse, Error> {
    let resp = Response::builder()
        .status(200)
        .header("content-type", "text/html")
        .body("Hello AWS Lambda HTTP request")
        .map_err(Box::new)?;
    Ok(resp)
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    lambda_http::run(service_fn(handler)).await
}
```

有關如何使用的另一個示例`lambda_http`，請參閱 Labs 存儲庫上的 http-axum 代碼示例。 AWS GitHub

Rust 的 HTTP Lambda 事件範例

- Lambda HTTP 事件：處理 HTTP 事件的 Rust 函數。
- 帶有 CORS 標頭的 Lambda HTTP 事件：使用 Tower 插入 CORS 標頭的 Rust 函數。
- 含有共用資源的 Lambda HTTP 事件：一個 Rust 函數，它使用在建立函數處理常式之前初始化的共用資源。

使用 .zip 封存檔部署 Rust Lambda 函數

> **Note**
>
> Rust 執行期用戶端是實驗性套件。它可能會發生變更，僅用於評估目的。

本頁說明如何編譯 Rust 函數，然後使用 Cargo Lambda 將編譯後的二進位檔部署到 AWS Lambda。它也會示範如何使用 AWS Command Line Interface 和 AWS Serverless Application Model CLI 來部署已編譯的二進位檔。

章節

必要條件

- Rust
- AWS Command Line Interface (AWS CLI) 版本 2

在 macOS、Windows 或 Linux 上建置 Rust 函數

下列步驟示範如何使用 Rust 為您的第一個 Lambda 函數建立專案，並使用 Cargo Lambda 進行編譯。

1. 安裝 Cargo Lambda，這是一個 Cargo 子命令，它可以在 macOS、Windows 和 Linux 上為 Lambda 編譯 Rust 函數。

   要在任何已安裝 Python 3 的系統上安裝 Cargo Lambda，請使用 pip：

   ```
   pip3 install cargo-lambda
   ```

   要在 macOS 或 Linux 上安裝 Cargo Lambda，請使用 Homebrew：

   ```
   brew tap cargo-lambda/cargo-lambda
   ```

```
brew install cargo-lambda
```

要在 Windows 上安裝 Cargo Lambda，請使用 Scoop：

```
scoop bucket add cargo-lambda
scoop install cargo-lambda/cargo-lambda
```

如需其他選項，請參閱 Cargo Lambda 文件中的安裝。

2. 建立套件結構。此命令會在 `src/main.rs` 中建立一些基礎函數程式碼。可以使用此程式碼進行測試，也可以將其替換為您自己的程式碼。

```
cargo lambda new my-function
```

3. 在套件的根目錄中，執行 build 子命令來編譯函數中的程式碼。

```
cargo lambda build --release
```

(選用) 如果您想要在 Lambda 上使用 AWS Graviton2，請新增 `--arm64` 標記以便為 ARM CPU 編譯程式碼。

```
cargo lambda build --release --arm64
```

4. 在部署 Rust 函數之前，請先在您的機器上設定 AWS 憑證。

```
aws configure
```

使用 Cargo Lambda 部署 Rust 函數二進位檔

使用 deploy 子命令將編譯後的二進位檔部署至 Lambda。此命令會建立執行角色，然後建立 Lambda 函數。若要指定現有的執行角色，請使用 --iam-role 標記。

```
cargo lambda deploy my-function
```

使用 AWS CLI 部署 Rust 函數二進位檔

您也可以使用 AWS CLI 部署二進位檔。

1. 使用 build 子命令，建置 .zip 部署套件。

```
cargo lambda build --release --output-format zip
```

2. 將 .zip 套件部署至 Lambda。針對 `--role`，指定執行角色的 ARN。

```
aws lambda create-function --function-name my-function \
     --runtime provided.al2023 \
     --role arn:aws:iam::111122223333:role/lambda-role \
     --handler rust.handler \
     --zip-file fileb://target/lambda/my-function/bootstrap.zip
```

使用 AWS SAM CLI 部署 Rust 函數二進位檔

您也可以使用 AWS SAM CLI 部署二進位檔。

1. 使用資源和屬性定義建立 AWS SAM 範本。如需詳細資訊，請參閱《AWS Serverless Application Model 開發人員指南》中的 AWS::Serverless::Function。

   Example Rust 二進位檔的 SAM 資源和屬性定義

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: SAM template for Rust binaries
Resources:
  RustFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: target/lambda/my-function/
      Handler: rust.handler
      Runtime: provided.al2023
Outputs:
  RustFunction:
    Description: "Lambda Function ARN"
    Value: !GetAtt RustFunction.Arn
```

2. 使用 build 子命令來編譯函數。

```
cargo lambda build --release
```

3. 使用 sam deploy 命令將函數部署到 Lambda。

```
sam deploy --guided
```

如需有關使用 AWS SAM CLI 建置 Rust 函數的詳細資訊，請參閱《AWS Serverless Application Model 開發人員指南》中的使用 Cargo Lambda 建置 Rust Lambda 函數。

使用 Cargo Lambda 叫用您的 Rust 函數

使用 invoke 子命令，透過承載來測試您的函數。

```
cargo lambda invoke --remote --data-ascii '{"command": "Hello world"}' my-function
```

使用 AWS CLI 叫用 Rust 函數

您也可以使用 AWS CLI 來叫用函數。

```
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"command": "Hello world"}' /tmp/out.txt
```

如果您使用 AWS CLI 第 2 版，則需要 cli-binary-format 選項。若要讓此成為預設的設定，請執行 `aws configure set cli-binary-format raw-in-base64-out`。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

Rust 中的 Lambda 函數日誌記錄

> **Note**
>
> Rust 執行期用戶端是實驗性套件。它可能會發生變更，僅用於評估目的。

AWS Lambda代表您自動監控 Lambda 函數，並將日誌傳送到 Amazon CloudWatch。您的 Lambda 函數隨附一個 CloudWatch 日誌記錄群組和函數每個執行個體的日誌串流。Lambda 執行期環境會將每次調用的詳細資訊傳送至日誌串流，並且轉傳來自函數程式碼的日誌及其他輸出。如需詳細資訊，請參閱 使用 Amazon CloudWatch 日誌 AWS Lambda。此頁面說明如何從 Lambda 函數的程式碼中產生日誌輸出。

建立編寫日誌的函數

若要從您的函式程式碼輸出日誌，可使用寫入到 `stdout` 或 `stderr` 的任何日誌記錄函數，例如 `println!` 巨集。下列範例使用 `println!` 在函數處理常式啟動時和完成之前列印訊息。

```
use lambda_runtime::{service_fn, LambdaEvent, Error};
use serde_json::{json, Value};
async fn handler(event: LambdaEvent<Value>) -> Result<Value, Error> {
    println!("Rust function invoked");
    let payload = event.payload;
    let first_name = payload["firstName"].as_str().unwrap_or("world");
    println!("Rust function responds to {}", &first_name);
    Ok(json!({ "message": format!("Hello, {first_name}!") }))
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    lambda_runtime::run(service_fn(handler)).await
}
```

帶有追蹤套件的進階日誌記錄

追蹤是檢測 Rust 程式以收集結構化、以事件為基礎的診斷資訊的架構。此架構提供公用程式來自訂日誌記錄輸出層級和格式，例如建立結構化的 JSON 日誌訊息。若要使用此架構，必須在實作函數處理常式之前初始化 `subscriber`。然後，您可以使用追蹤巨集 (例如 `debug`、`info` 和 `error`) 來指定每個案例所需的日誌記錄層級。

Example – 使用追蹤套件

注意下列事項：

- `tracing_subscriber::fmt().json()`：包含此選項時，日誌會格式化為 JSON。若要使用此選項，必須將 `json` 功能包含在 `tracing-subscriber` 相依項中 (例如，`tracing-subscriber = { version = "0.3.11", features = ["json"] }`)。
- `#[tracing::instrument(skip(event), fields(req_id = %event.context.request_id))]`：每次叫用處理常式時，此註釋都會產生一個範圍。此範圍會將請求 ID 新增至每個日誌行。
- `{ %first_name }`：此建構模組將 `first_name` 欄位新增到使用該欄位的日誌行。此欄位的值對應具有相同名稱的變數。

```
use lambda_runtime::{service_fn, Error, LambdaEvent};
use serde_json::{json, Value};
#[tracing::instrument(skip(event), fields(req_id = %event.context.request_id))]
async fn handler(event: LambdaEvent<Value>) -> Result<Value, Error> {
    tracing::info!("Rust function invoked");
    let payload = event.payload;
    let first_name = payload["firstName"].as_str().unwrap_or("world");
    tracing::info!({ %first_name }, "Rust function responds to event");
    Ok(json!({ "message": format!("Hello, {first_name}!") }))
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt().json()
        .with_max_level(tracing::Level::INFO)
        // this needs to be set to remove duplicated information in the log.
        .with_current_span(false)
        // this needs to be set to false, otherwise ANSI color codes will
        // show up in a confusing manner in CloudWatch logs.
        .with_ansi(false)
        // disabling time is handy because CloudWatch will add the ingestion time.
        .without_time()
        // remove the name of the function from every log entry
        .with_target(false)
        .init();
    lambda_runtime::run(service_fn(handler)).await
}
```

叫用此 Rust 函數時，其會列印類似於以下內容的兩條日誌行：

```
{"level":"INFO","fields":{"message":"Rust function invoked"},"spans":
[{"req_id":"45daaaa7-1a72-470c-9a62-e79860044bb5","name":"handler"}]}
{"level":"INFO","fields":{"message":"Rust function responds to
 event","first_name":"David"},"spans":[{"req_id":"45daaaa7-1a72-470c-9a62-
e79860044bb5","name":"handler"}]}
```

使用來自其 AWS 他服務的事件叫用 Lambda

某些 AWS 服務可以使用觸發器直接叫用 Lambda 函數。這些服務會將事件推送至 Lambda，並在指定的事件發生時立即叫用函數。觸發器適用於離散事件和即時處理。當您使用 Lambda 主控台建立觸發器時，主控台會與對應的 AWS 服務互動，以便在該服務上設定事件通知。觸發程序實際上是由產生事件的服務儲存和管理，而不是由 Lambda 進行管理。

事件是以 JSON 格式構建的資料。JSON 結構因產生它的服務和事件類型而異，但它們都包含函數處理事件所需的資料。

一個函數可以有多個觸發器。每個觸發條件皆做為獨立調用函數的用戶端，而 Lambda 傳遞給函數的每個事件只會包含來自一個觸發條件的資料。Lambda 將事件文件轉換為物件並將其傳遞給您的函數處理常式。

視服務而定，事件驅動的叫用可以是同步或非同步的。

- 對於同步調用，產生事件的服務會等待來自您的函數回應。該服務定義函數需要在回應中傳回的資料。服務控制項錯誤策略，例如發生錯誤時是否重試。
- 對於非同步調用，Lambda 會先將事件排入佇列，再將事件傳送至您的函數。當 Lambda 將事件排入佇列時，它會立即向產生事件的服務傳送成功回應。在函數處理事件後，Lambda 不會向事件產生服務傳回回應。

建立觸發器

建立觸發器的最簡單方法是使用 Lambda 主控台。當您使用主控台建立觸發器時，Lambda 會自動將必要的權限新增至函數的資源型政策。

若要使用 Lambda 主控台建立觸發器

1. 開啟 Lambda 主控台中的函數頁面。
2. 選取您要為其建立觸發器的函數。
3. 在函數概觀窗格中，選擇新增觸發條件。
4. 選取您要叫用函數的 AWS 服務。
5. 填寫 [觸發器] 組態窗格中的選項，然後選擇 [新增]。根據 AWS 服務 您選擇呼叫函數的不同，觸發器組態選項會有所不同。

可以叫用 Lambda 函數的服務

下表列出可以叫用 Lambda 函數的服務。

服務	調用的方法
Amazon Alexa	事件驅動；同步調用
Amazon Managed Streaming for Apache Kafka	事件來源對映
自我管理的 Apache Kafka	事件來源對映
Amazon API Gateway	事件驅動；同步調用
AWS CloudFormation	事件驅動；非同步調用
Amazon CloudFront （Lambda @Edge）	事件驅動；同步調用
Amazon CloudWatch 日誌	事件驅動；非同步調用
AWS CodeCommit	事件驅動；非同步調用
AWS CodePipeline	事件驅動；非同步調用
Amazon Cognito	事件驅動；同步調用
AWS Config	事件驅動；非同步調用
Amazon Connect	事件驅動；同步調用
Amazon DynamoDB	事件來源對映
Amazon Elastic File System	特殊整合
Elastic Load Balancing (Application Load Balancer)	事件驅動；同步調用
AWS IoT	事件驅動；非同步調用

服務	調用的方法
Amazon Kinesis	事件來源對映
Amazon 數據 Firehose	事件驅動；同步調用
Amazon Lex	事件驅動；同步調用
Amazon MQ	事件來源對映
Amazon Simple Email Service	事件驅動；非同步調用
Amazon Simple Notification Service	事件驅動；非同步調用
Amazon Simple Queue Service	事件來源對映
Amazon Simple Storage Service (Amazon S3)	事件驅動；非同步調用
Amazon Simple Storage Service 批次	事件驅動；同步調用
Secrets Manager	事件驅動；同步調用
Amazon VPC Lattice	事件驅動；同步調用
AWS X-Ray	特殊整合

常見的 Lambda 應用程式類型和使用案例

Lambda 函數和觸發程式是在 AWS Lambda 上建置應用程式時的核心元件。Lambda 函數是處理事件的程式碼和執行時間，而觸發程式是叫用函數的 AWS 服務或應用程式。為了說明，請考量下列情況：

- 檔案處理 – 假設您有相片分享應用程式。有人使用您的應用程式上傳相片，並且應用程式將這些使用者相片儲存在 Amazon S3 儲存貯體中。接著，您的應用程式會建立每個使用者的相片縮圖版本，並顯示於該使用者的檔案頁面。在這種情況下，您可以選擇建立 Lambda 函數以自動建立縮圖。Amazon S3 是受支援的 AWS 事件來源之一，它可發佈物件建立的事件並叫用 Lambda 函數。Lambda 函數程式碼可以從 S3 儲存貯體讀取相片物件、建立相片縮圖版本，然後儲存至另一個 S3 儲存貯體中。
- 資料與分析 – 假設您正在建置分析應用程式，並存放原始資料至 DynamoDB 資料表中。當撰寫、更新或是刪除資料表中的項目時，DynamoDB Streams 可將項目更新事件發佈至與資料表相關聯的串流中。在此情況下，事件資料提供項目金鑰、事件名稱 (如插入、更新與刪除) 與其他相關詳細資訊。您可以撰寫 Lambda 函數，透過彙總原始資料，產生自訂指標。
- 網站 – 假設您正在建立網站，且在 Lambda 中託管後端邏輯。您可以使用 Amazon API Gateway 作為 HTTP 端點，透過 HTTP 叫用 Lambda 函數。現在，您的 Web 用戶端可叫用 API，然後 API Gateway 可將請求路由至 Lambda。
- 行動應用程式 – 假設您有一個生產事件的自訂行動應用程式。您可以建立 Lambda 函數來處理自訂應用程式所發佈的事件。例如，您可以在自訂行動應用程式中設定 Lambda 函數來處理點擊事件。

AWS Lambda 支援許多 AWS 服務當做事件來源。如需詳細資訊，請參閱 使用來自其 AWS 他服務的事件叫用 Lambda。當您設定這些事件來源以觸發 Lambda 函數時，事件發生時將自動叫用 Lambda 函數。您定義事件來源映射，這可定義您如何識別追蹤哪些事件以及叫用哪個 Lambda 函數。

以下是事件來源及 end-to-end 體驗運作方式的簡介範例。

範例 1：Amazon S3 推送事件並叫用 Lambda 函數

Amazon S3 可在儲存貯體中發佈諸如 PUT、POST、COPY 與 DELETE 等不同類型的物件事件。您可以使用儲存貯體通知功能，設定事件來源映射，以指示 Amazon S3 在發生特定類型的事件時叫用 Lambda 函數。

下列是典型序列：

1. 使用者在儲存貯體中建立物件。
2. Amazon S3 偵測物件建立的事件。

3. Amazon S3 會使用執行角色提供的許可叫用 Lambda 函數。
4. AWS Lambda 會執行 Lambda 函數，並指定該事件為參數。

您可以設定 Amazon S3，作為儲存貯體通知動作來叫用您的函數。若要授予 Amazon S3 許可以便叫用該函數，請更新函數的以資源為基礎的政策。

範例 2：AWS Lambda 自 Kinesis 串流中提取事件並叫用 Lambda 函數

若為輪詢型事件來源，在該來源上偵測到記錄時，AWS Lambda 會輪詢該來源並叫用 Lambda 函數。

- CreateEventSourceMapping
- UpdateEventSourceMapping

下列步驟說明自訂應用程式如何將記錄寫入至 Kinesis 串流。

1. 自訂應用程式會將記錄寫入至 Kinesis 串流。
2. 當服務在串流上偵測到新記錄時，AWS Lambda 將持續輪詢串流並叫用 Lambda 函數。AWS Lambda 依據您在 Lambda 中建立的事件來源映射，知道要輪詢哪個串流，以及要叫用哪個 Lambda 函數。
3. Lambda 函數連同傳入的事件一起被叫用。

使用以串流為基礎的事件來源時，您會在 AWS Lambda 中建立事件來源映射。Lambda 會從串流中讀取項目並同步叫用該函數。您不需要授予 Lambda 叫用該函數的許可，但是它需要許可才能從串流中讀取。

使用 AWS Lambda 搭配 Alexa

您可以使用 Lambda 函數建置服務，它們會提供新技能給 Amazon Echo 上的語音助理 Alexa。Alexa Skills 套件提供建立這些新技能所需的 API、工具和說明文件，而提供此套件的是以 Lambda 函數方式執行的您自己的服務。Amazon Echo 使用者可藉由詢問 Alexa 問題或提出要求來存取這些新技能。

Alexa 技能套件可在上取得 GitHub。

- 適用於 Java 的 Alexa Skills Kit SDK
- 適用於 Node.js 的 Alexa Skills Kit SDK
- 適用於 Python 的 Alexa Skills Kit SDK

Example Alexa 智慧型家庭事件

```
{
  "header": {
    "payloadVersion": "1",
    "namespace": "Control",
    "name": "SwitchOnOffRequest"
  },
  "payload": {
    "switchControlAction": "TURN_ON",
    "appliance": {
      "additionalApplianceDetails": {
        "key2": "value2",
        "key1": "value1"
      },
      "applianceId": "sampleId"
    },
    "accessToken": "sampleAccessToken"
  }
}
```

如需詳細資訊，請參閱《使用 Alexa Skills Kit 建置技能》指南中的將自訂技能託管為 AWS Lambda 函數。

使用 Amazon API Gateway 端點叫用 Lambda 函數

您可以使用 Amazon API Gateway 為您的 Lambda 函數建立具有 HTTP 端點的 Web API。API Gateway 提供了用於建立和記錄 Web API 的工具，可將 HTTP 請求路由至 Lambda 函數。您可以使用身分驗證和授權控制來保護對 API 的存取。您的 API 可以透過網際網路提供流量，也可以只在 VPC 內存取。

API 中的資源定義一個或多個方法，例如 GET 或 POST。方法具有將請求路由到 Lambda 函數或其他整合類型的整合。您可以個別定義每個資源和方法，或使用特殊資源和方法類型來比對所有符合某模式的請求。代理資源會擷取資源下的所有路徑。ANY 方法會擷取所有 HTTP 方法。

章節

選擇 API 類型

API Gateway 支援三種調用 Lambda 函數的 API 類型：

- HTTP API：一個輕量級，低延遲的 REST API。
- 其餘 API：一個可定制的，功能豐富的 REST API。
- WebSocket API：一種 Web API，用於維護與客戶端的持續連接以進行全雙工通信。

HTTP API 和 REST API 都是處理 HTTP 請求並傳回回應的 RESTful API。HTTP API 較新，並且是使用 API Gateway 版本 2 API 建置而成。下列功能是 HTTP API 的新功能：

HTTP API 功能

- 自動部署 - 當您修改路由或整合時，變更會自動部署至已啟用自動部署的階段。

- 預設階段 - 您可以建立預設階段 (`$default`)，在 API URL 的根路徑提供請求。對於具名階段，您必須在路徑的開頭加入階段名稱。
- CORS 組態 - 您可以設定 API 將 CORS 標頭新增到傳出回應中，而不是在函數程式碼中手動新增。

REST API 是 API Gateway 自推出以來支援的典型 RESTful API。REST API 目前具有更多的自訂、整合和管理功能。

REST API 功能

- 整合類型 - REST API 支援自訂 Lambda 整合。您可以使用自訂整合，只將請求的本文傳送到函數，或者在將請求本文傳送到函數之前套用轉換範本。
- 存取控制 - REST API 支援其他身分驗證和授權選項。
- 監視和追蹤 — REST API 支援 AWS X-Ray 追蹤和其他記錄選項。

如需詳細比較，請參閱 API Gateway 開發人員指南中的在 HTTP API 和 REST API 之間選擇。

WebSocket API 也會使用 API Gateway 第 2 版 API，並支援類似的功能集。針對受益於用戶端和 WebSocket API 之間持續連線的應用程式使用 API。 WebSocket API 提供全雙工通訊，這表示用戶端和 API 都可以連續傳送訊息，而無需等待回應。

HTTP API 支援簡化的事件格式 (2.0 版)。下列範例顯示來自 HTTP API 的事件。

Example event-v2.json - API Gateway 代理事件 (HTTP API)

```
{
    "version": "2.0",
    "routeKey": "ANY /nodejs-apig-function-1G3XMPLZXVXYI",
    "rawPath": "/default/nodejs-apig-function-1G3XMPLZXVXYI",
    "rawQueryString": "",
    "cookies": [
        "s_fid=7AABXMPL1AFD9BBF-0643XMPL09956DE2",
        "regStatus=pre-register"
    ],
    "headers": {
        "accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/
webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9",
        "accept-encoding": "gzip, deflate, br",
        ...
    },
    "requestContext": {
```

```
        "accountId": "123456789012",
        "apiId": "r3pmxmplak",
        "domainName": "r3pmxmplak.execute-api.us-east-2.amazonaws.com",
        "domainPrefix": "r3pmxmplak",
        "http": {
            "method": "GET",
            "path": "/default/nodejs-apig-function-1G3XMPLZXVXYI",
            "protocol": "HTTP/1.1",
            "sourceIp": "205.255.255.176",
            "userAgent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
 (KHTML, like Gecko) Chrome/80.0.3987.132 Safari/537.36"
        },
        "requestId": "JKJaXmPLvHcESHA=",
        "routeKey": "ANY /nodejs-apig-function-1G3XMPLZXVXYI",
        "stage": "default",
        "time": "10/Mar/2020:05:16:23 +0000",
        "timeEpoch": 1583817383220
    },
    "isBase64Encoded": true
}
```

如需詳細資訊，請參閱《API Gateway 開發人員指南》中的 AWS Lambda 整合。

將端點新增至您的 Lambda 函數

若要將公有端點新增至您的 Lambda 函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 在 函式概觀 下，選擇 新增觸發條件 。
4. 選取 API Gateway (API Gateway)。
5. 選擇 Create an API (建立 API) 或 Use an existing API (使用現有 API)。
   a. 全新 API：對於 API type (API 類型)，請選擇 HTTP API。如需詳細資訊，請參閱 API 類型。
   b. 現有 API：從下拉式選單中選取 API 或輸入 API ID (例如，r3pmxmplak)。
6. 在 Security (安全性) 中，選擇 Open (開啟)。
7. 選擇 Add (新增)。

代理整合

API Gateway API 由階段、資源、方法和整合所組成。階段和資源決定端點的路徑：

API 路徑格式

- `/prod/` - `prod` 階段和根資源。
- `/prod/user` - `prod` 階段和 `user` 資源。
- `/dev/{proxy+}` - `dev` 階段中的任何路由。
- `/` - (HTTP API) 預設階段和根資源。

Lambda 整合將路徑和 HTTP 方法組合映射到一個 Lambda 函數。您可以設定 API Gateway 依現狀傳遞 HTTP 請求的主體 (自訂整合)，或將請求主體封裝在一個包含所有請求資訊 (包括標頭、資源、路徑和方法) 的文件中。

如需詳細資訊，請參閱在 API Gateway 中設定 Lambda 代理整合。

事件格式

Amazon API Gateway 使用包含 HTTP 請求之 JSON 表示的事件來同步調用您的函數。對於自訂整合，事件是請求的本文。對於代理整合，事件具有已定義的結構。下列範例顯示來自 API Gateway REST API 的代理事件。

Example event.json API Gateway 代理事件 (REST API)

```
{
      "resource": "/",
      "path": "/",
      "httpMethod": "GET",
      "requestContext": {
          "resourcePath": "/",
          "httpMethod": "GET",
          "path": "/Prod/",
          ...
      },
      "headers": {
          "accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/
webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9",
          "accept-encoding": "gzip, deflate, br",
```

```
          "Host": "70ixmpl4fl.execute-api.us-east-2.amazonaws.com",
          "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
 (KHTML, like Gecko) Chrome/80.0.3987.132 Safari/537.36",
          "X-Amzn-Trace-Id": "Root=1-5e66d96f-7491f09xmpl79d18acf3d050",
          ...
      },
      "multiValueHeaders": {
          "accept": [
              "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/
apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9"
          ],
          "accept-encoding": [
              "gzip, deflate, br"
          ],
          ...
      },
      "queryStringParameters": null,
      "multiValueQueryStringParameters": null,
      "pathParameters": null,
      "stageVariables": null,
      "body": null,
      "isBase64Encoded": false
  }
```

回應格式

API Gateway 會等待函數的回應並將結果轉達給呼叫者。對於自訂整合，您可以定義整合回應和方法回應，將函數的輸出轉換為 HTTP 回應。對於代理整合，函數必須以特定格式的回應表示作出回應。

下列範例顯示來自 Node.js 函數的回應物件。回應物件表示包含 JSON 文件的成功 HTTP 回應。

Example index.mjs - 代理整合回應物件 (Node.js)

```
var response = {
      "statusCode": 200,
      "headers": {
        "Content-Type": "application/json"
      },
      "isBase64Encoded": false,
      "multiValueHeaders": {
        "X-Custom-Header": ["My value", "My other value"],
      },
      "body": "{\n  \"TotalCodeSize\": 104330022,\n  \"FunctionCount\": 26\n}"
```

```
    }
```

Lambda 執行時間會將回應物件序列化為 JSON，並將其傳送至 API。API 會剖析該回應並用它來建立 HTTP 回應，然後將其傳送到發出原始請求的用戶端。

Example HTTP 回應

```
< HTTP/1.1 200 OK
  < Content-Type: application/json
  < Content-Length: 55
  < Connection: keep-alive
  < x-amzn-RequestId: 32998fea-xmpl-4268-8c72-16138d629356
  < X-Custom-Header: My value
  < X-Custom-Header: My other value
  < X-Amzn-Trace-Id: Root=1-5e6aa925-ccecxmplbae116148e52f036
  <
  {
    "TotalCodeSize": 104330022,
    "FunctionCount": 26
  }
```

許可

Amazon API Gateway 會取得許可，從函數的以資源為基礎的政策中調用您的函數。您可以將呼叫許可授與整個 API，或將有限存取權授與階段、資源或方法。

當您透過使用 Lambda 主控台、使用 API Gateway 主控台或在 AWS SAM 範本中，將 API 新增至函數時，會自動更新函數的以資源為基礎的政策。範例函數政策範例如下。

Example 函數政策

```
{
  "Version": "2012-10-17",
  "Id": "default",
  "Statement": [
    {
      "Sid": "nodejs-apig-functiongetEndpointPermissionProd-BWDBXMPLXE2F",
      "Effect": "Allow",
      "Principal": {
        "Service": "apigateway.amazonaws.com"
      },
```

```
      "Action": "lambda:InvokeFunction",
      "Resource": "arn:aws:lambda:us-east-2:111122223333:function:nodejs-apig-
function-1G3MXMPLXVXYI",
      "Condition": {
        "StringEquals": {
          "aws:SourceAccount": "111122223333"
        },
        "ArnLike": {
          "aws:SourceArn": "arn:aws:execute-api:us-east-2:111122223333:ktyvxmpls1/*/
GET/"
        }
      }
    }
  ]
}
```

您可以使用下列 API 操作來手動管理函數政策許可：

- AddPermission
- RemovePermission
- GetPolicy

若要將調用許可授與現有 API，請使用 add-permission 命令。

```
aws lambda add-permission --function-name my-function \
--statement-id apigateway-get --action lambda:InvokeFunction \
--principal apigateway.amazonaws.com \
--source-arn "arn:aws:execute-api:us-east-2:123456789012:mnh1xmpli7/default/GET/"
```

您應該會看到下列輸出：

```
{
    "Statement": "{\"Sid\":\"apigateway-test-2\",\"Effect\":\"Allow\",\"Principal
\":{\"Service\":\"apigateway.amazonaws.com\"},\"Action\":\"lambda:InvokeFunction
\",\"Resource\":\"arn:aws:lambda:us-east-2:123456789012:function:my-function
\",\"Condition\":{\"ArnLike\":{\"AWS:SourceArn\":\"arn:aws:execute-api:us-
east-2:123456789012:mnh1xmpli7/default/GET\"}}}"
}
```

> **Note**
>
> 如果您的函數和 API 不同 AWS 區域，則源 ARN 中的區域標識符必須與函數的「區域」相匹配，而不是 API 的「區域」。當 API Gateway 調用函數時，它會使用基於 API 的 ARN，但修改以匹配函數的區域的資源 ARN。

此範例中的來源 ARN 會將許可授與 API 的預設階段中根資源 GET 方法的整合 (ID 為 `mnh1xmpli7`)。您可以在來源 ARN 中使用星號，將許可授與多個階段、方法或資源。

資源模式

- `mnh1xmpli7/*/GET/*` - 所有階段中所有資源上的 GET 方法。
- `mnh1xmpli7/prod/ANY/user` - `prod` 階段中 `user` 資源上的 ANY 方法。
- `mnh1xmpli7/*/*/*` - 所有階段中所有資源上的任何方法。

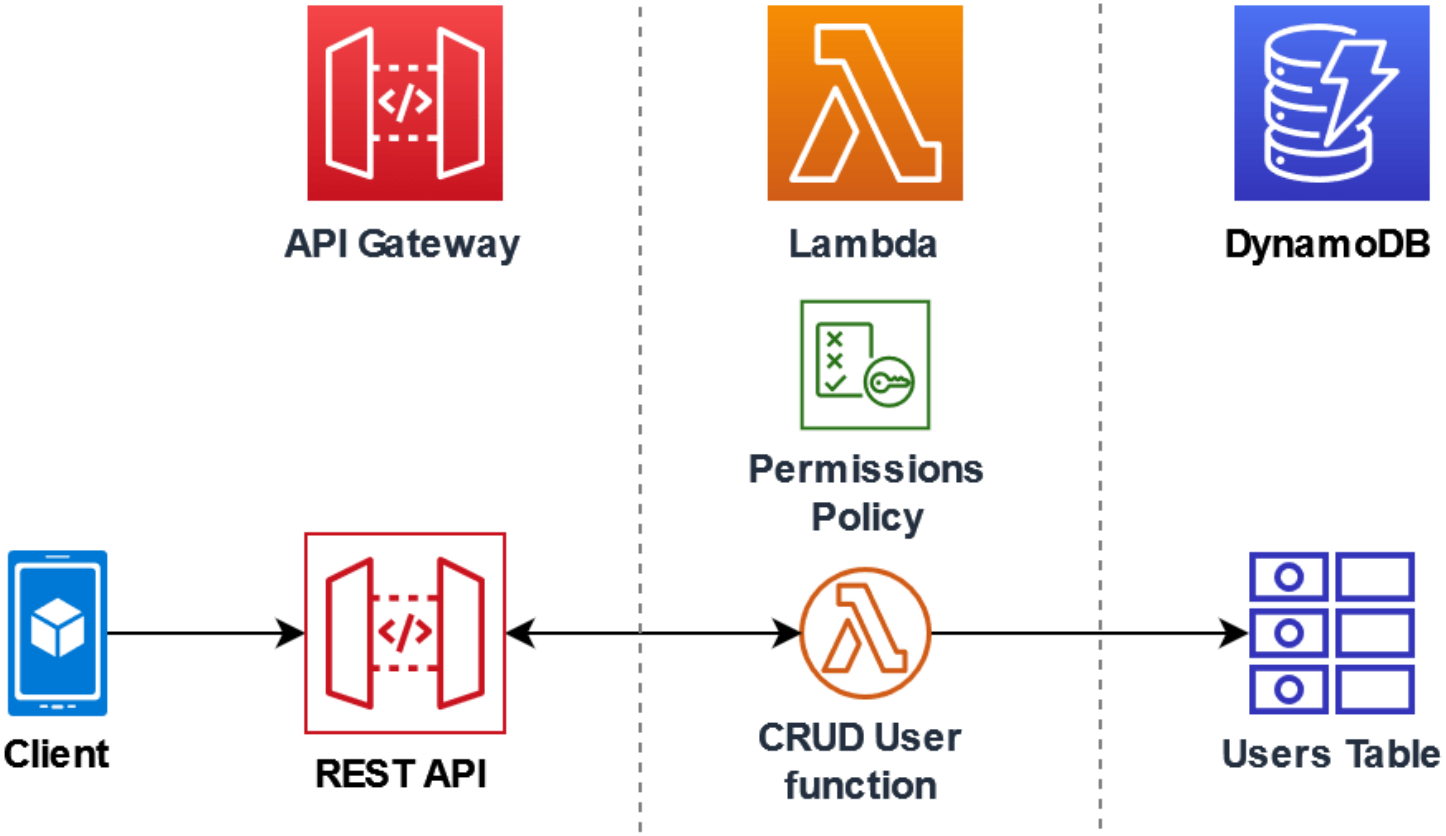
如需檢視政策和移除陳述式的詳細資訊，請參閱 清理以資源為基礎的政策。

範例應用程式

具有 Node.js 範例應用程式的 API Gateway 包含一個含有 AWS SAM 範本的函數，該函式會建立已啟用 AWS X-Ray 追蹤的 REST API。它還包括用於部署，調用函數，測試 API 和清理的腳本。

教學課程：搭配使用 Lambda 與 API Gateway

在此教學課程中，您將建立 REST API，並透過此 API 調用 Lambda 函數。Lambda 函數會對 DynamoDB 資料表執行建立、讀取、更新及刪除 (CRUD) 操作。這裡提供的函數僅供示範，您將學習如何設定可調用任何 Lambda 函數的 API Gateway REST API。



使用 API Gateway 為使用者提供安全的 HTTP 端點以調用 Lambda 函數，並透過流量限流以及自動驗證和授權 API 呼叫，協助管理函數的大量呼叫。API Gateway 還使用 AWS Identity and Access Management (IAM) 和 Amazon Cognito 提供靈活的安全控制。對於需要預先授權才能呼叫應用程式的使用案例，這非常有用。

完成本教學課程需逐一進行以下階段：

1. 以 Python 或 Node.js 建立並設定 Lambda 函數，用於對 DynamoDB 資料表執行操作。
2. 在 API Gateway 中建立 REST API 以連接 Lambda 函數。
3. 建立 DynamoDB 資料表，然後在主控台中使用您的 Lambda 函數進行測試。
4. 在終端內使用 curl 部署 API 並測試完整設定。

完成這些階段後，您將了解如何使用 API Gateway 建立 HTTP 端點，以安全地調用任何規模的 Lambda 函數。您也會學習如何部署 API，以及如何在控制台中以及使用終端傳送 HTTP 請求來進行測試。

章節

- 建立許可政策
- 建立執行角色
- 建立函數
- 使用調用函數 AWS CLI
- 使用 API Gateway 建立 REST API
- 在 REST API 上建立資源
- 建立 HTTP POST 方法
- 建立 DynamoDB 資料表
- 測試 API Gateway、Lambda 和 DynamoDB 的整合
- 部署 API
- 使用 curl 來透過 HTTP 請求調用函數
- 清除資源 (選用)

必要條件

註冊一個 AWS 帳戶

如果您沒有 AWS 帳戶，請完成以下步驟來建立一個。

若要註冊成為 AWS 帳戶

1. 開啟 https://portal.aws.amazon.com/billing/signup。
2. 請遵循線上指示進行。

   部分註冊程序需接收來電，並在電話鍵盤輸入驗證碼。

   當您註冊一個時 AWS 帳戶，將創建AWS 帳戶根使用者一個。根使用者有權存取該帳戶中的所有 AWS 服務 和資源。安全性最佳做法是將管理存取權指派給使用者，並僅使用 root 使用者來執行需要 root 使用者存取權的工作。

AWS 註冊過程完成後，會向您發送確認電子郵件。您可以隨時登錄 https://aws.amazon.com/ 並選擇我的帳戶，以檢視您目前的帳戶活動並管理帳戶。

建立具有管理權限的使用者

註冊後，請保護您的 AWS 帳戶 AWS 帳戶根使用者 AWS IAM Identity Center、啟用和建立系統管理使用者，這樣您就不會將 root 使用者用於日常工作。

保護您的 AWS 帳戶根使用者

1. 選擇 Root 使用者並輸入您的 AWS 帳戶 電子郵件地址，以帳戶擁有者身分登入。AWS Management Console在下一頁中，輸入您的密碼。

   如需使用根使用者登入的說明，請參閱 AWS 登入 使用者指南中的以根使用者身分登入。
2. 若要在您的根使用者帳戶上啟用多重要素驗證 (MFA)。

   如需指示，請參閱《IAM 使用者指南》中的為 AWS 帳戶 根使用者啟用虛擬 MFA 裝置 (主控台)。

建立具有管理權限的使用者

1. 啟用 IAM Identity Center。

   如需指示，請參閱 AWS IAM Identity Center 使用者指南中的啟用 AWS IAM Identity Center。
2. 在 IAM 身分中心中，將管理存取權授予使用者。

   若要取得有關使用 IAM Identity Center 目錄 做為身分識別來源的自學課程，請參閱《使用指南》IAM Identity Center 目錄中的「以預設值設定使用AWS IAM Identity Center 者存取」。

以具有管理權限的使用者身分登入

- 若要使用您的 IAM Identity Center 使用者簽署，請使用建立 IAM Identity Center 使用者時傳送至您電子郵件地址的簽署 URL。

  如需使用 IAM 身分中心使用者登入的說明，請參閱使用AWS 登入 者指南中的登入 AWS 存取入口網站。

指派存取權給其他使用者

1. 在 IAM 身分中心中，建立遵循套用最低權限許可的最佳做法的權限集。

   如需指示，請參閱《AWS IAM Identity Center 使用指南》中的「建立權限集」。
2. 將使用者指派給群組，然後將單一登入存取權指派給群組。

如需指示，請參閱《AWS IAM Identity Center 使用指南》中的「新增群組」。

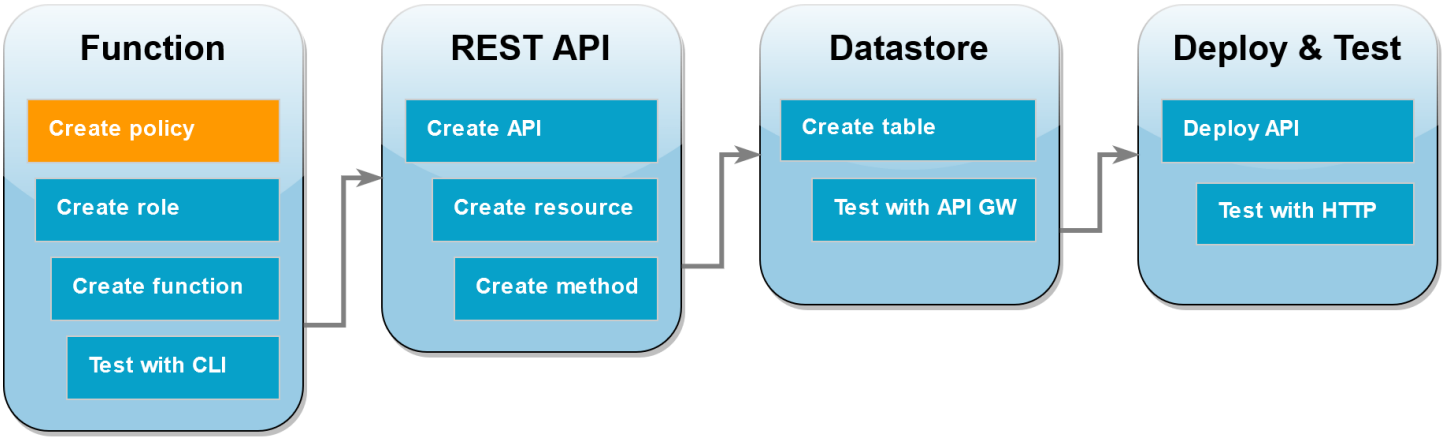
安裝 AWS Command Line Interface

如果您尚未安裝 AWS Command Line Interface，請按照安裝或更新最新版本的步驟進 AWS CLI行安裝。

本教學課程需使用命令列終端機或 Shell 來執行命令。在 Linux 和 macOS 中，使用您偏好的 Shell 和套件管理工具。

> **Note**
>
> 在 Windows 中，作業系統的內建終端不支援您常與 Lambda 搭配使用的某些 Bash CLI 命令 (例如 `zip`)。若要取得 Ubuntu 和 Bash 的 Windows 整合版本，請安裝適用於 Linux 的 Windows 子系統。

建立許可政策



在為 Lambda 函數建立執行角色之前，您必須先建立權限原則，以授與函數存取所需 AWS 資源的權限。在本教學課程中，該政策允許 Lambda 在 DynamoDB 表上執行 CRUD 操作並寫入 Amazon 日誌。 CloudWatch

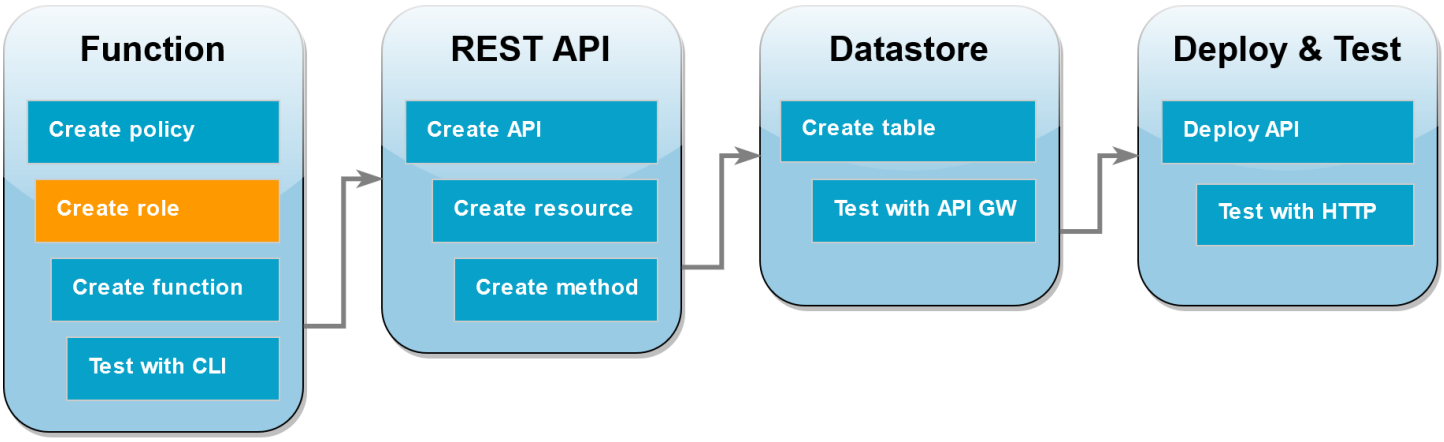
建立政策

1. 開啟 IAM 主控台中的 政策 頁面。
2. 選擇 建立政策 。
3. 選擇 JSON 索引標籤，然後將下列政策貼到 JSON 編輯器。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Stmt1428341300017",
      "Action": [
        "dynamodb:DeleteItem",
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:Query",
        "dynamodb:Scan",
        "dynamodb:UpdateItem"
      ],
      "Effect": "Allow",
      "Resource": "*"
    },
    {
      "Sid": "",
      "Resource": "*",
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents"
      ],
      "Effect": "Allow"
    }
  ]
}
```

4. 選擇 下一步：標籤 。
5. 選擇 下一步：檢閱 。
6. 在 檢閱政策 下，針對政策 名稱 ，輸入 **lambda-apigateway-policy**。
7. 選擇 建立政策 。

建立執行角色



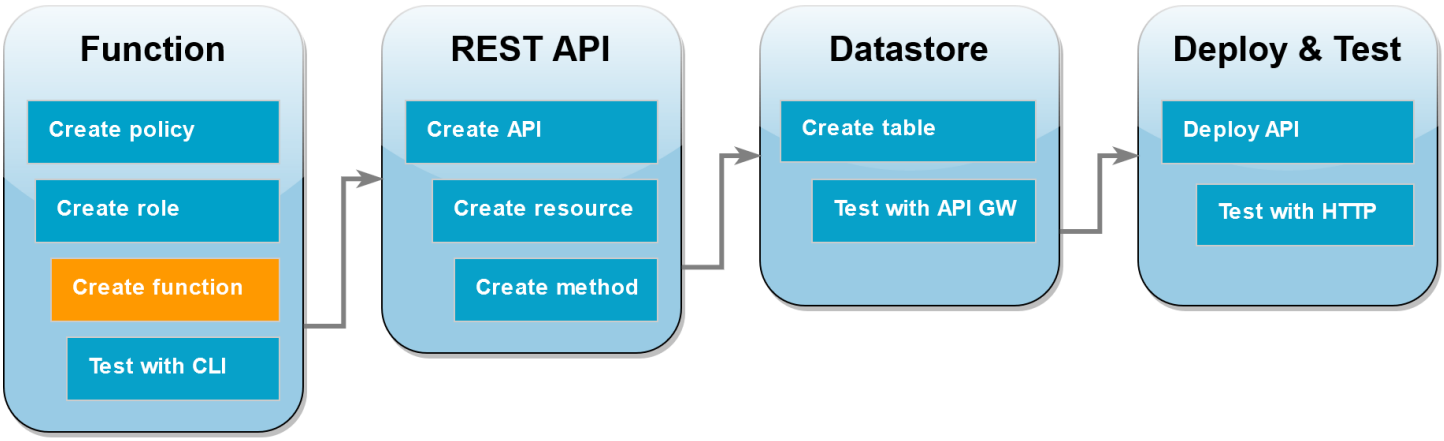
執行角色是一種 AWS Identity and Access Management (IAM) 角色，可授與 Lambda 函數存取 AWS 服務和資源的權限。若要讓函數對 DynamoDB 資料表執行操作，您需附加在上個步驟中建立的許可政策。

建立執行角色並附加自訂許可政策

1. 開啟 IAM 主控台中的 角色頁面 。
2. 選擇 建立角色。
3. 信任的實體類型請選擇 AWS 服務，使用案例則選擇 Lambda。
4. 選擇 下一步。
5. 在政策搜尋方塊中，輸入 **lambda-apigateway-policy**。
6. 在搜尋結果中，選取您建立的政策 (lambda-apigateway-policy)，然後選擇 下一步 。
7. 在 角色詳細資料 底下，角色名稱 請輸入 **lambda-apigateway-role**，然後選擇 建立角色 。

在教學課程的後續階段中，需用到您剛才建立的角色之 Amazon Resource Name (ARN)。在 IAM 主控台的 角色 頁面上，選擇角色的名稱 (lambda-apigateway-role)，然後複製 摘要 頁面上顯示的 角色 ARN 。

建立函數



下列程式碼範例會從 API Gateway 接收事件輸入，指定要對您建立的 DynamoDB 資料表執行的操作及一些承載資料。如果函數收到的參數有效，就會對資料表執行請求的操作。

Node.js

Example index.mjs

```
console.log('Loading function');

import { DynamoDBDocumentClient, PutCommand, GetCommand,
         UpdateCommand, DeleteCommand} from "@aws-sdk/lib-dynamodb";
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";

const ddbClient = new DynamoDBClient({ region: "us-west-2" });
const ddbDocClient = DynamoDBDocumentClient.from(ddbClient);

// Define the name of the DDB table to perform the CRUD operations on
const tablename = "lambda-apigateway";

/**
 * Provide an event that contains the following keys:
 *
 *   - operation: one of 'create,' 'read,' 'update,' 'delete,' or 'echo'
 *   - payload: a JSON object containing the parameters for the table item
 *              to perform the operation on
 */
export const handler = async (event, context) => {

    const operation = event.operation;
```

```
     if (operation == 'echo'){
          return(event.payload);
     }

    else {
        event.payload.TableName = tablename;

        switch (operation) {
          case 'create':
               await ddbDocClient.send(new PutCommand(event.payload));
               break;
          case 'read':
               var table_item = await ddbDocClient.send(new
 GetCommand(event.payload));
               console.log(table_item);
               break;
          case 'update':
               await ddbDocClient.send(new UpdateCommand(event.payload));
               break;
          case 'delete':
               await ddbDocClient.send(new DeleteCommand(event.payload));
               break;
          default:
            return ('Unknown operation: ${operation}');
          }
    }
};
```

> **Note**
>
> 在此範例中，DynamoDB 資料表的名稱定義為函數程式碼中的變數。在實際的應用程式中，最佳實務是將此參數做為環境變數來傳遞，並避免對資料表名稱進行硬式編碼。如需詳細資訊，請參閱使用 AWS Lambda 環境變數。

建立函數

1. 將程式碼範例儲存為名為的檔案，`index.mjs`並在必要時編輯程式碼中指定的 AWS 區域。程式碼中指定的區域，必須與您稍後在教學課程中建立的 DynamoDB 資料表區域相同。
2. 使用以下 `zip` 命令建立部署套件。

```
zip function.zip index.mjs
```

3. 使用create-function AWS CLI 指令建立 Lambda 函數。對於 role 參數，輸入您先前複製的執行角色 Amazon Resource Name (ARN)。

```
aws lambda create-function \
--function-name LambdaFunctionOverHttps \
--zip-file fileb://function.zip \
--handler index.handler \
--runtime nodejs20.x \
--role arn:aws:iam::123456789012:role/service-role/lambda-apigateway-role
```

Python 3

Example LambdaFunctionOverHttps. PY

```
import boto3
import json

# define the DynamoDB table that Lambda will connect to
tableName = "lambda-apigateway"

# create the DynamoDB resource
dynamo = boto3.resource('dynamodb').Table(tableName)

print('Loading function')

def lambda_handler(event, context):
    '''Provide an event that contains the following keys:

      - operation: one of the operations in the operations dict below
      - payload: a JSON object containing parameters to pass to the
                 operation being performed
    '''

    # define the functions used to perform the CRUD operations
    def ddb_create(x):
        dynamo.put_item(**x)

    def ddb_read(x):
        dynamo.get_item(**x)
```

```
    def ddb_update(x):
        dynamo.update_item(**x)

    def ddb_delete(x):
        dynamo.delete_item(**x)

    def echo(x):
        return x

    operation = event['operation']

    operations = {
        'create': ddb_create,
        'read': ddb_read,
        'update': ddb_update,
        'delete': ddb_delete,
        'echo': echo,
    }

    if operation in operations:
        return operations[operation](event.get('payload'))
    else:
        raise ValueError('Unrecognized operation "{}"'.format(operation))
```

> **Note**
>
> 在此範例中，DynamoDB 資料表的名稱定義為函數程式碼中的變數。在實際的應用程式中，最佳實務是將此參數做為環境變數來傳遞，並避免對資料表名稱進行硬式編碼。如需詳細資訊，請參閱使用 AWS Lambda 環境變數。

若要建立函數

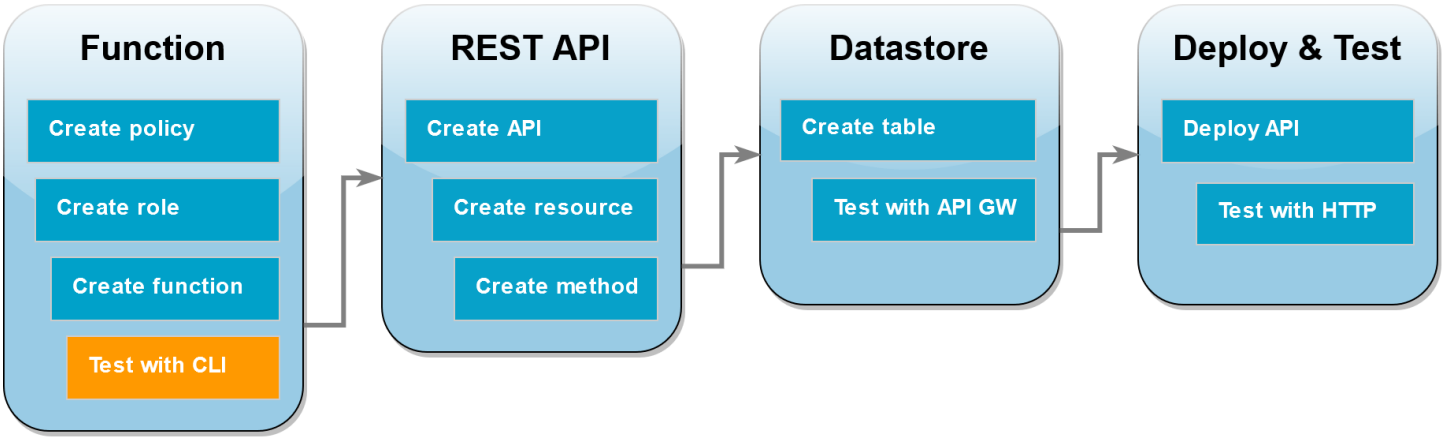
1. 將程式碼範例儲存為名為 `LambdaFunctionOverHttps.py` 的檔案。
2. 使用以下 `zip` 命令建立部署套件。

   ```
   zip function.zip LambdaFunctionOverHttps.py
   ```

3. 使用`create-function` AWS CLI 指令建立 Lambda 函數。對於 `role` 參數，輸入您先前複製的執行角色 Amazon Resource Name (ARN)。

```
aws lambda create-function \
--function-name LambdaFunctionOverHttps \
--zip-file fileb://function.zip \
--handler LambdaFunctionOverHttps.lambda_handler \
--runtime python3.12 \
--role arn:aws:iam::123456789012:role/service-role/lambda-apigateway-role
```

使用調用函數 AWS CLI



在將函數與 API Gateway 整合之前，請確認已成功部署該函數。建立測試事件，其中包含 API Gateway API 將傳送給 Lambda 的參數，並使用 AWS CLI `invoke`命令執行您的函數。

若要使用以呼叫 Lambda 函數 AWS CLI

1. 將下面的 JSON 儲存為名為 `input.txt` 的檔案。

```
{
    "operation": "echo",
    "payload": {
        "somekey1": "somevalue1",
        "somekey2": "somevalue2"
    }
}
```

2. 執行下列 `invoke` AWS CLI 命令。

```
aws lambda invoke \
--function-name LambdaFunctionOverHttps \
--payload file://input.txt outputfile.txt \
```

```
--cli-binary-format raw-in-base64-out
```

如果您使用的是 AWS CLI 版本 2，則需要此cli-binary-format選項。若要讓此成為預設的設定，請執行 `aws configure set cli-binary-format raw-in-base64-out`。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

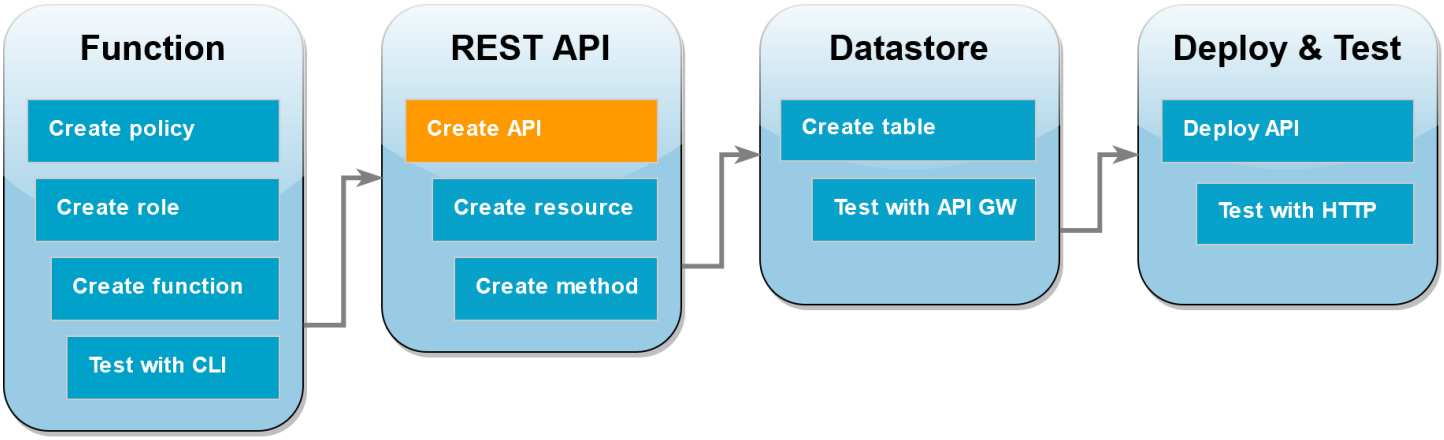
您應該會看到下列回應：

```
{
"StatusCode": 200,
"ExecutedVersion": "LATEST"
}
```

3. 確認函數已執行您在 JSON 測試事件中指定的 `echo` 操作。檢查 `outputfile.txt` 檔案，並確認包含下列內容：

```
{"somekey1": "somevalue1", "somekey2": "somevalue2"}
```

使用 API Gateway 建立 REST API



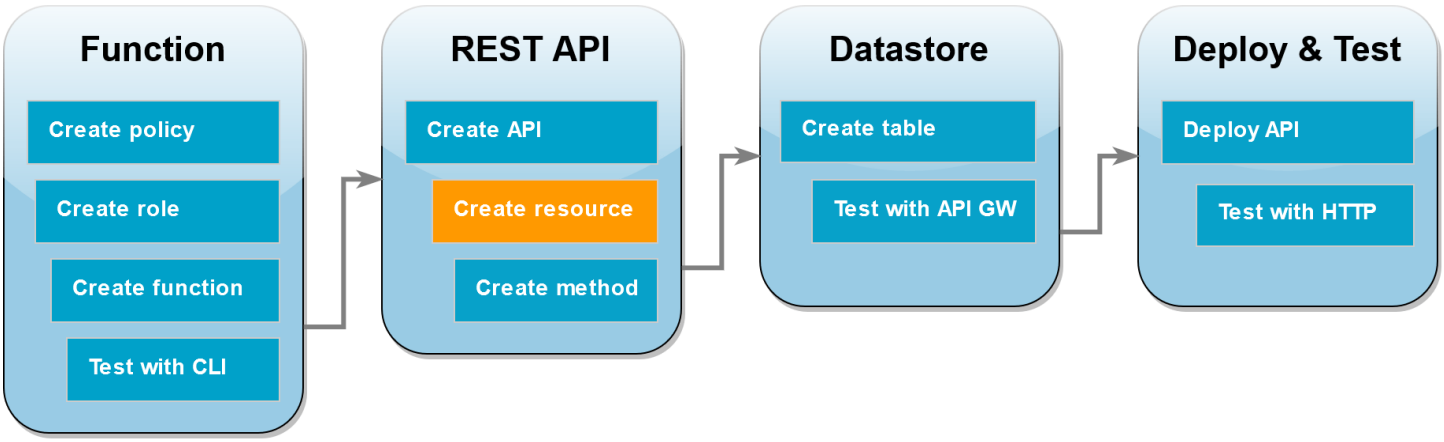
在此步驟中，您將建立用來調用 Lambda 函數的 API Gateway REST API。

若要建立 API

1. 開啟 API Gateway 主控台。
2. 選擇 建立 API 。
3. 在 REST API 方塊中，選擇 建置 。

4. 在 API 詳細資訊下，讓新增 API維持在已選取的狀態，然後對於 API 名稱，輸入 **`DynamoDBOperations`**。
5. 選擇 建立 API 。

在 REST API 上建立資源

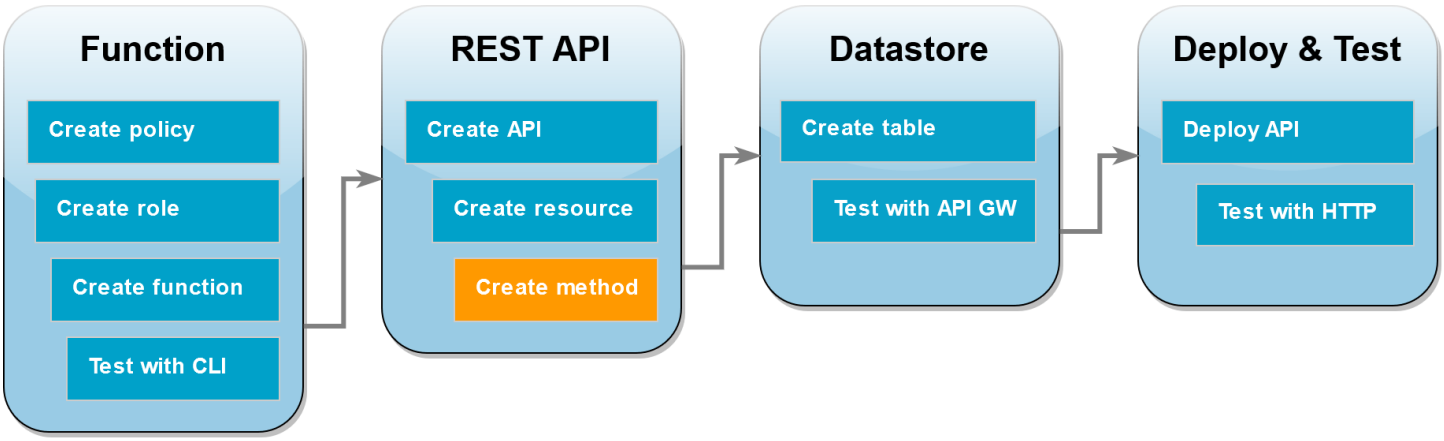


若要將 HTTP 方法新增到 API 中，首先需為該方法建立用來操作的資源。您可以在此建立資源來管理 DynamoDB 資料表。

若要建立資源

1. 在 API Gateway 主控台中，在 API 的資源頁面上，選擇建立資源。
2. 在資源詳細資訊中，針對資源名稱輸入 **`DynamoDBManager`**。
3. 選擇 建立資源 。

建立 HTTP POST 方法



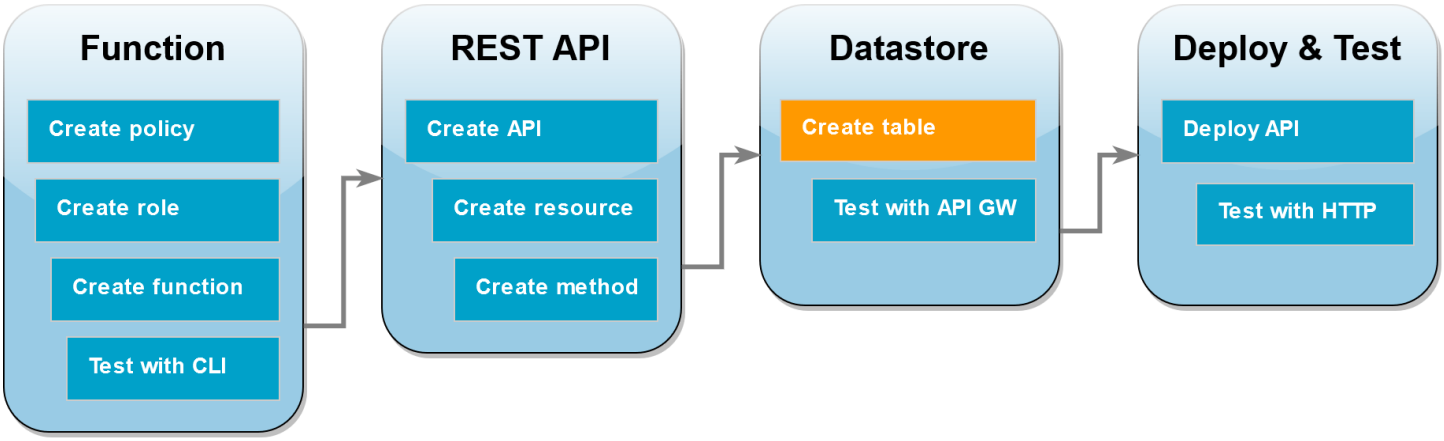
在此步驟中，您將為 DynamoDBManager 資源建立方法 (POST)。您需將此 POST 方法連結到 Lambda 函數，如此一來當方法收到 HTTP 請求，API Gateway 就會調用 Lambda 函數。

> **Note**
>
> 基於本教學課程的目的，會使用一個 HTTP 方法 (POST) 來調用單一 Lambda 函數，該函數會對 DynamoDB 資料表執行所有操作。在實際的應用程式中，最佳實務是針對每項操作使用不同的 Lambda 函數和 HTTP 方法。如需詳細資訊，請參閱無伺服器園地中的 The Lambda Monolith。

建立 POST 方法

1. 在 API 的資源頁面上，確定已反白選取 /DynamoDBManager 資源。然後，在方法窗格中，選擇建立方法。
2. 針對方法類型，選擇 POST。
3. 對於整合類型，讓 Lambda 函數維持在已選取的狀態。
4. 對於 Lambda 函數，請為函數 (LambdaFunctionOverHttps) 選擇 Amazon Resource Name (ARN)。
5. 選擇建立方法。

建立 DynamoDB 資料表



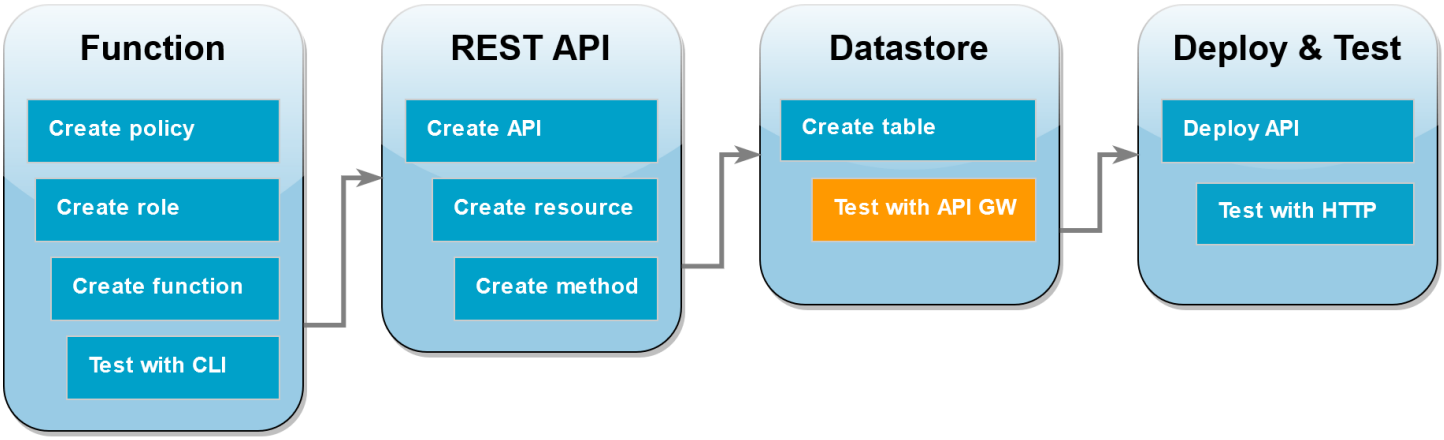
建立空白的 DynamoDB 資料表，Lambda 函數會對該資料表執行 CRUD 操作。

若要建立 DynamoDB 資料表

1. 開啟 DynamoDB 主控台的 資料表頁面 。

2. 選擇 建立資料表 。
3. 在 Table details (資料表詳細資訊) 下，執行下列動作：
   1. 對於 Table name (資料表名稱)，請輸入 **`lambda-apigateway`**。
   2. 對於 Partition key (分割區索引鍵)，輸入**`id`**，並保持資料類型設定為 String (字串)。
4. 在 Table settings (資料表設定) 下，保留 Default settings (預設設定)。
5. 選擇 建立資料表 。

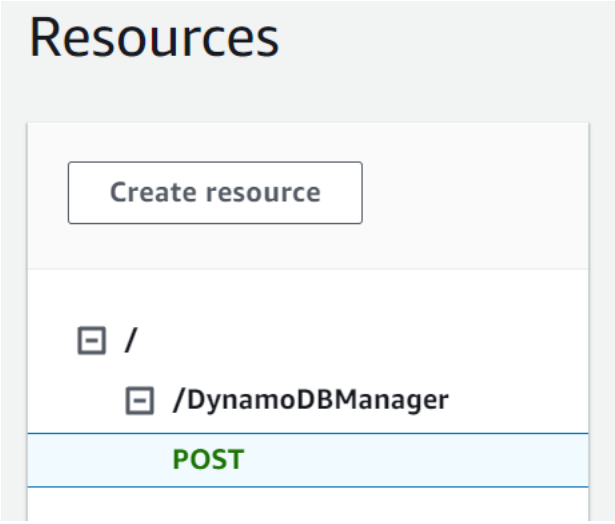
測試 API Gateway、Lambda 和 DynamoDB 的整合



您現在已準備好測試 API Gateway API 方法與 Lambda 函數和 DynamoDB 資料表的整合。使用 API Gateway 主控台，您可以利用主控台的測試功能，將請求直接傳送至您的 POST 方法。在此步驟中，首先需使用 `create` 操作將新項目新增至 DynamoDB 資料表，然後使用 `update` 操作來修改項目。

測試 1：在 DynamoDB 資料表中建立新項目

1. 在 API Gateway 主控台中，選擇您的 API (`DynamoDBOperations`)。
2. 選擇資`DynamoDBManager`源下的 POST 方法。



3. 選擇測試標籤。您可能需要選擇向右箭頭按鈕才能顯示此索引標籤。
4. 在測試方法下，讓查詢字串和標頭留空。對於請求主體，貼上下列 JSON：

```
{
  "operation": "create",
  "payload": {
    "Item": {
      "id": "1234ABCD",
      "number": 5
    }
  }
}
```

5. 選擇 測試 。

   測試完成時顯示的結果應該會顯示 200 狀態。此狀態碼表示 create 操作成功。

   若要確認，您可以檢查 DynamoDB 資料表現在是否包含新項目。

6. 開啟 DynamoDB 主控台的 資料表頁面 ，然後選擇 lambda-apigateway 資料表。
7. 選擇 探索資料表項目 。在 Items returned (傳回的項目) 窗格中，應該會看到一個包含 id 1234ABCD 和 number 5 的項目。

測試 2 ：更新 DynamoDB 資料表中的項目

1. 在 API Gateway 主控台中，返回到 POST 方法的測試分頁。
2. 在測試方法下，讓查詢字串和標頭留空。對於請求主體，貼上下列 JSON：

```
{
    "operation": "update",
    "payload": {
        "Key": {
            "id": "1234ABCD"
        },
        "AttributeUpdates": {
            "number": {
                "Value": 10
            }
        }
    }
}
```

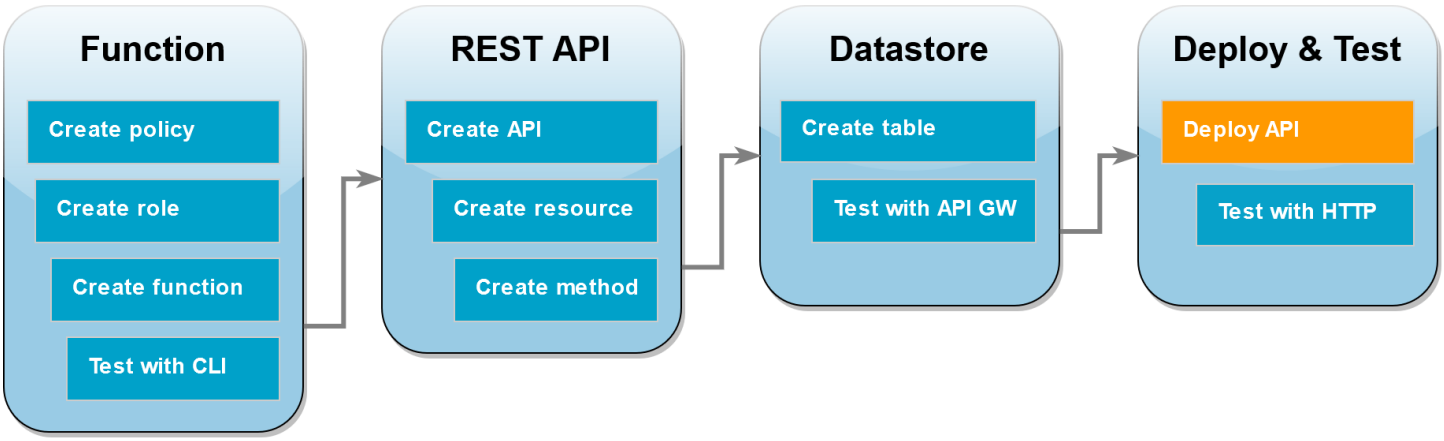
3. 選擇 測試 。

   測試完成時顯示的結果應該會顯示 200 狀態。此狀態碼表示 update 操作成功。

   若要確認，請檢查 DynamoDB 資料表中的項目是否已修改。

4. 開啟 DynamoDB 主控台的 資料表頁面 ，然後選擇 lambda-apigateway 資料表。
5. 選擇 探索資料表項目 。在 Items returned (傳回的項目) 窗格中，應該會看到一個包含 id 1234ABCD 和 number 10 的項目。

部署 API

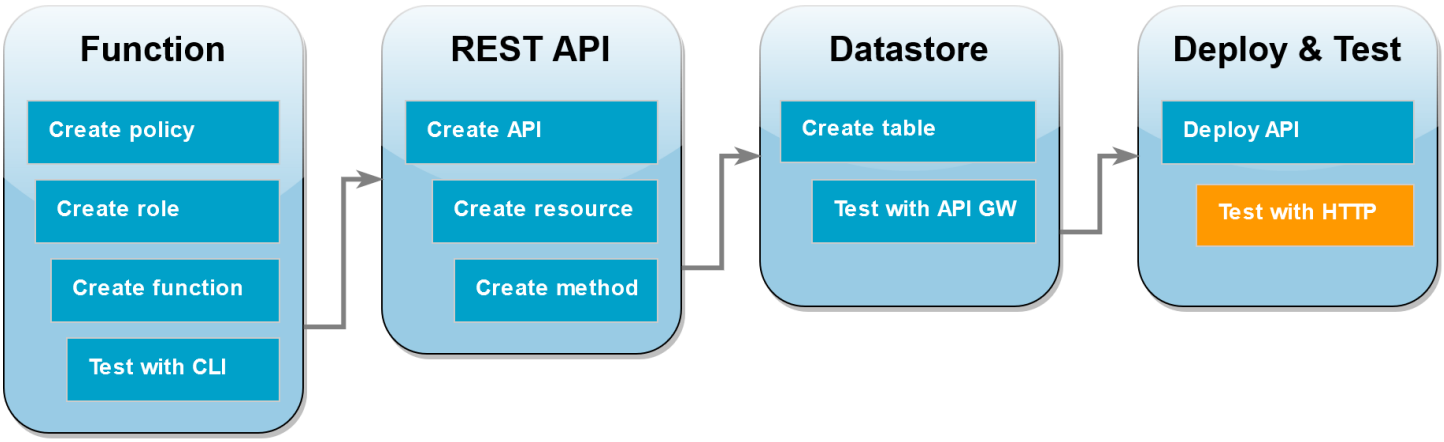


為了讓用戶端能呼叫您的 API，您必須建立部署並建立相關聯的階段。階段代表 API 的快照，包括其方法和整合項目。

部署 API

1. 開啟 API Gateway 主控台中的 API 頁面，然後選擇 DynamoDBOperations API。
2. 在 API 的資源頁面上，選擇部署 API。
3. 對於階段，請選擇*新增階段*，然後在階段名稱輸入 **test**。
4. 選擇部署。
5. 在階段詳細資訊窗格中，複製調用 URL。您將在下一個步驟中使用此資料來透過 HTTP 請求調用函數。

使用 curl 來透過 HTTP 請求調用函數



您現在可以透過向 API 發出 HTTP 請求來調用 Lambda 函數。在此步驟中，您將在 DynamoDB 資料表中建立新項目，然後將其刪除。

使用 curl 調用 Lambda 函數

1. 使用您在上個步驟中複製的調用 URL 執行下列 `curl` 命令。將 curl 與 `-d` (資料) 選項搭配使用時，系統會自動使用 HTTP POST 方法。

```
curl https://l8togsqxd8.execute-api.us-west-2.amazonaws.com/test/DynamoDBManager \
-d '{"operation": "create", "payload": {"Item": {"id": "5678EFGH", "number": 15}}}'
```

2. 若要驗證建立操作是否成功，請執行下列步驟：
   1. 開啟 DynamoDB 主控台的 資料表頁面 ，然後選擇 `lambda-apigateway` 資料表。
   2. 選擇 探索資料表項目 。在 Items returned (傳回的項目) 窗格中，應該會看到一個包含 id `5678EFGH` 和 number `15` 的項目。
3. 執行以下 `curl` 命令來刪除您剛剛建立的項目。使用您自己的調用 URL。

```
curl https://l8togsqxd8.execute-api.us-west-2.amazonaws.com/test/DynamoDBManager \
-d '{"operation": "delete", "payload": {"Key": {"id": "5678EFGH"}}}'
```

4. 確認刪除操作成功。在 DynamoDB 主控台 探索項目 頁面的 傳回的項目 窗格中，確認具有 id 5678EFGH 的項目已不存在於資料表中。

清除資源 (選用)

除非您想要保留為此教學課程建立的資源，否則您現在便可刪除。刪除您不再使用的 AWS 資源，您可以避免不必要的費用 AWS 帳戶。

若要刪除 Lambda 函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇您建立的函數。
3. 選擇 Actions (動作)、Delete (刪除)。
4. 在文字輸入欄位中輸入 **delete**，然後選擇 刪除 。

刪除執行角色

1. 開啟 IAM 主控台中的 角色頁面 。
2. 選取您建立的執行角色。
3. 選擇 刪除 。
4. 在文字輸入欄位中輸入角色的名稱，然後選擇 刪除 。

若要刪除 API

1. 開啟 API Gateway 主控台中的 API 頁面 。
2. 選取您建立的 API。
3. 選擇 動作 、刪除 。
4. 選擇 刪除 。

若要刪除 DynamoDB 資料表

1. 開啟 DynamoDB 主控台的 資料表頁面 。

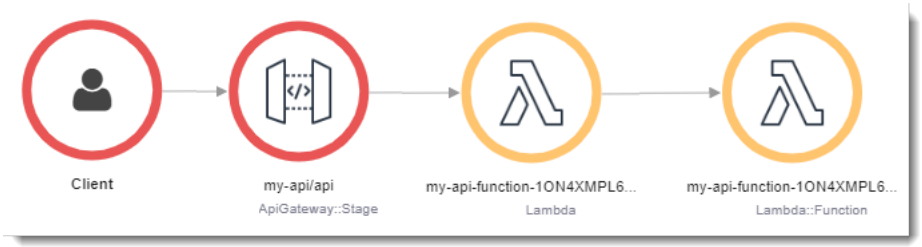
2. 選取您建立的資料表。
3. 選擇 刪除 。
4. 在文字方塊中輸入 **delete**。
5. 選擇 刪除資料表 。

使用 API Gateway API 處理 Lambda 錯誤

API Gateway 會將所有調用和函數錯誤視為內部錯誤。如果 Lambda API 拒絕調用請求，則 API Gateway 會傳回 500 錯誤代碼。如果函數執行但傳回錯誤，或傳回格式錯誤的回應，API Gateway 會傳回 502。在這兩種情況下，API Gateway 的回應主體為 `{"message": "Internal server error"}`。

> **Note**
>
> API Gateway 不會重試任何 Lambda 調用。如果 Lambda 傳回錯誤，API Gateway 會將錯誤回應傳回至用戶端。

下列範例顯示導致函數錯誤和 API Gateway 傳回 502 的請求的 X-Ray 流程圖。用戶端會收到一般錯誤訊息。



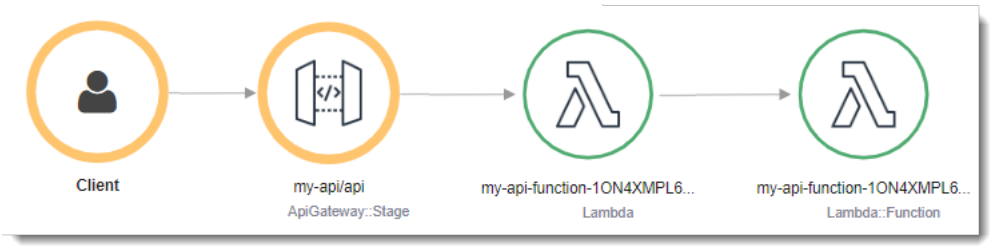
若要自訂錯誤回應，您必須在程式碼中發現錯誤，並以必要的格式格式化回應。

Example index.mjs - 格式錯誤

```
var formatError = function(error){
  var response = {
    "statusCode": error.statusCode,
    "headers": {
      "Content-Type": "text/plain",
      "x-amzn-ErrorType": error.code
    },
```

```
    "isBase64Encoded": false,
    "body": error.code + ": " + error.message
  }
  return response
}
```

API Gateway 將此回應轉換為具有自定義狀態碼和主體的 HTTP 錯誤。在流程圖中，函數節點是綠色的，因為它會處理錯誤。



AWS Lambda 搭配使用 AWS 應用程式編寫器

AWS 應用程式編寫器 是用於指定現代應用程序的可視化構建器。 AWS您可以透過 AWS 服務 在視覺化畫布中拖曳、分組和連線來設計應用程式架構。應用程式編寫器會從您的設計建立基礎設施即程式碼 (IaC) 範本，您可以使用 AWS SAM 或 AWS CloudFormation 部署。

匯出 Lambda 函數至應用程式編寫器

您可以使用 Lambda 主控台根據現有 Lambda 函數的組態建立新專案，開始使用應用程式編寫器。若要將函數的組態和程式碼匯出至應用程式編寫器以建立新專案，請執行下列動作：

1. 開啟 Lambda 主控台中的函數頁面。
2. 選取您希望使用做為應用程式編寫器專案的函數。
3. 在函數概觀窗格中，選擇匯出至應用程式編寫器。

   若要將函數的組態和程式碼匯出至應用程式編寫器，Lambda 會在您的帳戶中建立 Amazon S3 儲存貯體來暫時存放此資料。
4. 在對話方塊中，選擇確認並建立專案以接受此儲存貯體的預設名稱，並將函數的設定和程式碼匯出至應用程式編寫器。
5. (選擇性) 若要為 Lambda 建立的 Amazon S3 儲存貯體選擇其他名稱，請輸入新名稱，然後選擇確認並建立專案。Amazon S3 儲存貯體的名稱必須是全域唯一的，並遵循儲存貯體命名規則。
6. 若要在應用程式編寫器中儲存專案和函數檔案，請啟用本機同步模式。

> **Note**
>
> 如果您之前已使用匯出至應用程式編寫器功能，並使用預設名稱建立 Amazon S3 儲存貯體，則 Lambda 可以重複使用此儲存貯體 (如果儲存貯體仍然存在)。接受對話方塊中的預設儲存貯體名稱，以重新使用現有儲存貯體。

Amazon S3 傳輸儲存貯體組態

Lambda 建立用來傳輸函數組態的 Amazon S3 儲存貯體，會使用 AES 256 加密標準的自動加密物件。Lambda 也會將值區設定為使用儲存貯體擁有者條件，以確保只 AWS 帳戶 有您能夠將物件新增至值區。

Lambda 會將儲存貯體設定為在上傳物件 10 天後自動刪除物件。但是，Lambda 不會自動刪除儲存貯體本身。若要從您的值區刪除值區 AWS 帳戶，請依照刪除值區中的指示操作。預設值區名稱使用字首`lambdasam`、10 位數字的英數字串， AWS 區域 以及您在中建立函數的名稱：

```
lambdasam-06f22da95b-us-east-1
```

為了避免額外的費用被添加到您的 AWS 帳戶，我們建議您在完成將函數匯出到應用程式撰寫器後立即刪除 Amazon S3 儲存貯體。

適用標準 Amazon S3 定價。

所需的許可

若要使用 Lambda 與應用程式撰寫器整合功能，您需要特定許可才能下載 AWS SAM 範本並將函數的組態寫入 Amazon S3。

若要下載 AWS SAM 範本，您必須擁有使用下列 API 動作的權限：

- GetPolicy
- IAM: GetPolicy 版本
- IAM：GetRole
- IAM：GetRole政策
- IAM：ListAttachedRolePolicies
- IAM：ListRole政策
- IAM：ListRoles

您可以將AWSLambda_ReadOnlyAccess AWS 受管政策新增至 IAM 使用者角色，授予使用所有這些動作的權限。

若要讓 Lambda 將函數的組態寫入 Amazon S3，您必須擁有使用下列 API 動作的許可：

- 中三：PutObject
- 中三：CreateBucket
- S3：PutBucket加密
- 中三：PutBucketLifecycleConfiguration

如果您無法將函數組態匯出至應用程式編寫器，請檢查您的帳戶是否具有執行這些操作所需要的許可。如果您擁有所需要的許可，但仍然無法匯出函數組態，請檢查任何可能會限制 Amazon S3 存取的資源型政策。

其他資源

如需取得如何根據現有 Lambda 函數在應用程式編寫器中設計無伺服器應用程式的詳細教學課程，請參閱the section called “基礎設施即程式碼 (IaC)”。

若要使用應用程式撰寫器， AWS SAM 以及使用 Lambda 設計和部署完整的無伺服器應用程式，您也可以遵循AWS 無伺服器模式研討會中的AWS 應用程式編寫器 教學課程。

搭配 CloudWatch 日誌使用 Lambda

您可以使用 Lambda 函數來監控和分析來自 Amazon 日 CloudWatch 誌日誌串流的日誌。建立一個或多個日誌串流的訂閱，以在建立日誌或符合選用模式時叫用函數。使用函數來傳送通知或將日誌保存到資料庫或儲存體。

CloudWatch Logs 會以包含記錄資料的事件以非同步方式叫用您的函數。資料欄位的值是 Base64 編碼的 .gzip 封存。

Example CloudWatch 記錄訊息事件

```
{
  "awslogs": {
    "data":
 "ewogICAgIm1lc3NhZ2VUeXBlIjogIkRBVEFfTUVTU0FHRSIsCiAgICAib3duZXIiOiAiMTIzNDU2Nzg5MDEyIiwKICAgI
  }
}
```

若已解碼並解壓縮，日誌資料為具有下列結構的 JSON 文件：

Example CloudWatch 日誌消息數據（解碼）

```
{
    "messageType": "DATA_MESSAGE",
    "owner": "123456789012",
    "logGroup": "/aws/lambda/echo-nodejs",
    "logStream": "2019/03/13/[$LATEST]94fa867e5374431291a7fc14e2f56ae7",
    "subscriptionFilters": [
        "LambdaStream_cloudwatchlogs-node"
    ],
    "logEvents": [
        {
            "id": "34622316099697884706540976068822859012661220141643892546",
            "timestamp": 1552518348220,
            "message": "REPORT RequestId: 6234bffe-149a-b642-81ff-2e8e376d8aff
\tDuration: 46.84 ms\tBilled Duration: 47 ms \tMemory Size: 192 MB\tMax Memory Used: 72
 MB\t\n"
        }
    ]
}
```

AWS Lambda 搭配使用 AWS CloudFormation

在 AWS CloudFormation 範本中，您可以指定 Lambda 函數做為自訂資源的目標。在堆疊生命週期事件期間，使用自訂資源來處理參數、擷取組態值或呼叫其他 AWS 服務。

下列範例叫用在範本中其他地方定義的函數。

Example - 自訂資源定義

```
Resources:
  primerinvoke:
    Type: AWS::CloudFormation::CustomResource
    Version: "1.0"
    Properties:
      ServiceToken: !GetAtt primer.Arn
      FunctionName: !Ref randomerror
```

服務令牌是在您建立、更新或刪除堆疊時 AWS CloudFormation 呼叫的函數的 Amazon 資源名稱 (ARN)。您還可以包含其他屬性`FunctionName`，例如，按原樣 AWS CloudFormation 傳遞給您的函數。

AWS CloudFormation 使用包含回呼 URL 的事件以非同步方式叫用 Lambda 函數。

Example — AWS CloudFormation 消息事件

```
{
    "RequestType": "Create",
    "ServiceToken": "arn:aws:lambda:us-east-1:123456789012:function:lambda-error-
processor-primer-14ROR2T3JKU66",
    "ResponseURL": "https://cloudformation-custom-resource-response-useast1.s3-us-
east-1.amazonaws.com/arn%3Aaws%3Acloudformation%3Aus-east-1%3A123456789012%3Astack/
lambda-error-processor/1134083a-2608-1e91-9897-022501a2c456%7Cprimerinvoke
%7C5d478078-13e9-baf0-464a-7ef285ecc786?
AWSAccessKeyId=AKIAIOSFODNN7EXAMPLE&Expires=1555451971&Signature=28UijZePE5I4dvukKQqM
%2F9Rf1o4%3D",
    "StackId": "arn:aws:cloudformation:us-east-1:123456789012:stack/lambda-error-
processor/1134083a-2608-1e91-9897-022501a2c456",
    "RequestId": "5d478078-13e9-baf0-464a-7ef285ecc786",
    "LogicalResourceId": "primerinvoke",
    "ResourceType": "AWS::CloudFormation::CustomResource",
    "ResourceProperties": {
```

```
        "ServiceToken": "arn:aws:lambda:us-east-1:123456789012:function:lambda-error-
processor-primer-14ROR2T3JKU66",
        "FunctionName": "lambda-error-processor-randomerror-ZWUC391MQAJK"
    }
}
```

函式負責回傳回應到回呼 URL，說明呼叫成功或失敗。如需完整的回應語法，請參閱自訂資源回應物件。

Example — AWS CloudFormation 自訂資源回應

```
{
    "Status": "SUCCESS",
    "PhysicalResourceId": "2019/04/18/[$LATEST]b3d1bfc65f19ec610654e4d9b9de47a0",
    "StackId": "arn:aws:cloudformation:us-east-1:123456789012:stack/lambda-error-
processor/1134083a-2608-1e91-9897-022501a2c456",
    "RequestId": "5d478078-13e9-baf0-464a-7ef285ecc786",
    "LogicalResourceId": "primerinvoke"
}
```

AWS CloudFormation 提供一個名`cfn-response`為處理發送響應的庫。如果您在範本中定義函數，則可以依名稱要求資源庫。 AWS CloudFormation 然後將程式庫新增至為函數建立的部署套件。

如果自訂資源使用的函數已連接彈性網路介面，則請將下列資源新增至 VPC 政策，其中 **`region`** 是函數所在的區域 (不含破折號)。例如，`us-east-1` 為 `useast1`。這將允許自定義資源響應將信號發回到 AWS CloudFormation 堆棧的回調 URL。

```
arn:aws:s3:::cloudformation-custom-resource-response-region",
"arn:aws:s3:::cloudformation-custom-resource-response-region/*",
```

下列的範例函式會叫用第二個函式。如果呼叫成功，函式會傳送成功回應 AWS CloudFormation，而堆疊更新會繼續進行。範本使用提供的AWS::Serverless::Function資源類型 AWS Serverless Application Model。

Example -自定義資源功能

```
Transform: 'AWS::Serverless-2016-10-31'
Resources:
  primer:
    Type: AWS::Serverless::Function
    Properties:
```

```
      Handler: index.handler
      Runtime: nodejs16.x
      InlineCode: |
        var aws = require('aws-sdk');
        var response = require('cfn-response');
        exports.handler = function(event, context) {
            // For Delete requests, immediately send a SUCCESS response.
            if (event.RequestType == "Delete") {
                response.send(event, context, "SUCCESS");
                return;
            }
            var responseStatus = "FAILED";
            var responseData = {};
            var functionName = event.ResourceProperties.FunctionName
            var lambda = new aws.Lambda();
            lambda.invoke({ FunctionName: functionName }, function(err, invokeResult) {
                if (err) {
                    responseData = {Error: "Invoke call failed"};
                    console.log(responseData.Error + ":\n", err);
                }
                else responseStatus = "SUCCESS";
                response.send(event, context, responseStatus, responseData);
            });
        };
      Description: Invoke a function to create a log stream.
      MemorySize: 128
      Timeout: 8
      Role: !GetAtt role.Arn
      Tracing: Active
```

如果自定義資源調用的函數未在模板中定義，則可以`cfn-response`從用戶指南的 cfn-response 模塊中獲取源代碼。 AWS CloudFormation

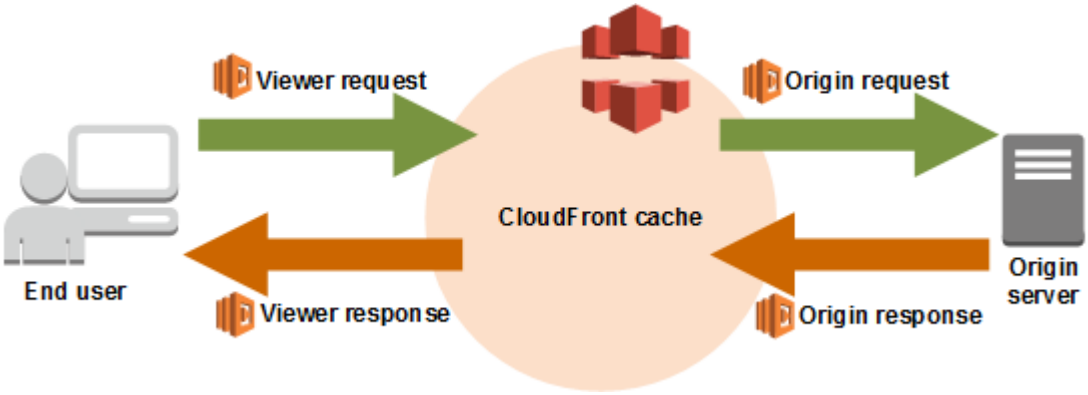
如需自訂資源的詳細資訊，請參閱 AWS CloudFormation 使用者指南中的自訂資源。

AWS Lambda搭配使用 CloudFront Lambda @Edge

Lambda @Edge 是一個擴充功AWS Lambda能，可讓您在 Amazon CloudFront 節點部署 Python 和 Node.js 函數。Lambda @Edge 的常見使用案例是使用函數來自訂您的 CloudFront 分發提供給最終使用者的內容。調用較靠近檢視器，而不是在原始伺服器上的函數，可大幅降低延遲並改善使用者體驗。

當您將 CloudFront 分發與 Lambda @Edge 函數產生關聯時，會在節 CloudFront 點 CloudFront 攔截請求和回應。 CloudFront 然後通過發送事件調用您的 Lambda 函數。發生下列事件時，您可以 CloudFront 叫用 Lambda 函數：

- CloudFront 收到來自檢視者的要求時 (檢視者要求)
- 在將請求 CloudFront 轉發到原始（原始請求）之前
- 當 CloudFront 收到來自來源的響應（原始響應）
- CloudFront 返回給查看者的響應之前（查看器響應）



> **Note**
>
> Lambda@Edge 支援一組有限的執行時間和功能。如需詳細資訊，請參閱 Amazon CloudFront 開發人員指南中對 Lambda 函數的要求和限制。

以下是 CloudFront 事件的範例。

Example CloudFront 訊息事件

```
{
  "Records": [
    {
```

```
      "cf": {
        "config": {
          "distributionId": "EDFDVBD6EXAMPLE"
        },
        "request": {
          "clientIp": "2001:0db8:85a3:0:0:8a2e:0370:7334",
          "method": "GET",
          "uri": "/picture.jpg",
          "headers": {
            "host": [
              {
                "key": "Host",
                "value": "d111111abcdef8.cloudfront.net"
              }
            ],
            "user-agent": [
              {
                "key": "User-Agent",
                "value": "curl/7.51.0"
              }
            ]
          }
        }
      }
    ]
}
```

如需使用 Lambda @Edge 的詳細資訊，請參閱 CloudFront 搭配使用 Lambda @Edge。

搭配使用 AWS Lambda 與 AWS CodeCommit

您可以為 AWS CodeCommit 儲存庫建立觸發條件，讓儲存庫中的事件叫用 Lambda 函數。例如，當建立分支或標籤，或是推送至現有分支時，您可以叫用 Lambda 函數。

Example AWS CodeCommit 訊息事件

```
{
    "Records": [
        {
            "awsRegion": "us-east-2",
            "codecommit": {
                "references": [
                    {
                        "commit": "5e493c6f3067653f3d04eca608b4901eb227078",
                        "ref": "refs/heads/master"
                    }
                ]
            },
            "eventId": "31ade2c7-f889-47c5-a937-1cf99e2790e9",
            "eventName": "ReferenceChanges",
            "eventPartNumber": 1,
            "eventSource": "aws:codecommit",
            "eventSourceARN": "arn:aws:codecommit:us-east-2:123456789012:lambda-
pipeline-repo",
            "eventTime": "2019-03-12T20:58:25.400+0000",
            "eventTotalParts": 1,
            "eventTriggerConfigId": "0d17d6a4-efeb-46f3-b3ab-a63741badeb8",
            "eventTriggerName": "index.handler",
            "eventVersion": "1.0",
            "userIdentityARN": "arn:aws:iam::123456789012:user/intern"
        }
    ]
}
```

如需詳細資訊，請參閱管理 AWS CodeCommit 儲存庫的觸發條件。

搭配使用 AWS Lambda 與 Amazon Cognito

Amazon Cognito 事件功能可讓您執行 Lambda 函數，藉此回應 Amazon Cognito 中的事件。Amazon Cognito 為您的 Web 和行動應用程式提供身分驗證、授權和使用者管理。您可叫用 Lambda 函數來回應 Amazon Cognito 中的重要事件。例如，您可以使用同步觸發事件，叫用每次同步處理資料集時就會發佈一次的 Lambda 函數。如需進一步了解及逐步說明範例，請參閱「行動開發」部落格中的介紹 Amazon Cognito 事件：同步觸發。

Example Amazon Cognito 訊息事件

```
{
  "datasetName": "datasetName",
  "eventType": "SyncTrigger",
  "region": "us-east-1",
  "identityId": "identityId",
  "datasetRecords": {
    "SampleKey2": {
      "newValue": "newValue2",
      "oldValue": "oldValue2",
      "op": "replace"
    },
    "SampleKey1": {
      "newValue": "newValue1",
      "oldValue": "oldValue1",
      "op": "replace"
    }
  },
  "identityPoolId": "identityPoolId",
  "version": 2
}
```

您可以使用 Amazon Cognito 事件訂閱組態來設定事件來源映射。如需事件來源映射及範例事件的詳細資訊，請參閱 Amazon Cognito 開發人員指南中的 Amazon Cognito 事件。

搭配使用 Lambda 與 Amazon Connect

您可以使用 Lambda 函數來處理來自 Amazon Connect 的請求。您可以使用 Amazon Connect 建立雲端聯絡中心。

Amazon Connect 將使用含有請求內文和中繼資料的事件，同步叫用您的 Lambda 函數。

Example Amazon Connect 請求事件

```
{
  "Details": {
      "ContactData": {
          "Attributes": {},
          "Channel": "VOICE",
          "ContactId": "4a573372-1f28-4e26-b97b-XXXXXXXXXXX",
          "CustomerEndpoint": {
              "Address": "+1234567890",
              "Type": "TELEPHONE_NUMBER"
          },
          "InitialContactId": "4a573372-1f28-4e26-b97b-XXXXXXXXXXX",
          "InitiationMethod": "INBOUND | OUTBOUND | TRANSFER | CALLBACK",
          "InstanceARN": "arn:aws:connect:aws-region:1234567890:instance/
c8c0e68d-2200-4265-82c0-XXXXXXXXXX",
          "PreviousContactId": "4a573372-1f28-4e26-b97b-XXXXXXXXXXX",
          "Queue": {
             "ARN": "arn:aws:connect:eu-west-2:111111111111:instance/cccccccc-bbbb-
dddd-eeee-ffffffffffff/queue/aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee",
              "Name": "PasswordReset"
           },
          "SystemEndpoint": {
              "Address": "+1234567890",
              "Type": "TELEPHONE_NUMBER"
          }
      },
      "Parameters": {
          "sentAttributeKey": "sentAttributeValue"
      }
  },
  "Name": "ContactFlowEvent"
}
```

有關如何搭配使用 Amazon Connect 與 Lambda 的資訊，請參閱 Amazon Connect 管理員指南中的叫用 Lambda 函數。

AWS Lambda 與 Amazon EC2 一起使用

您可以使用 AWS Lambda 來處理來自 Amazon 彈性運算雲端的生命週期事件，以及管理 Amazon EC2 資源。Amazon EC2 會針對生命週期CloudWatch 事件 EventBridge (例如執行個體變更狀態、Amazon 彈性區塊存放區磁碟區快照完成時或競價型執行個體排定終止) 傳送事件至 Amazon (事件)。您可以設定 EventBridge (CloudWatch 事件) 將這些事件轉寄至 Lambda 函數以進行處理。

EventBridge (CloudWatch 事件) 使用來自 Amazon EC2 的事件文件以非同步方式叫用您的 Lambda 函數。

Example 執行個體生命週期事件

```
{
    "version": "0",
    "id": "b6ba298a-7732-2226-xmpl-976312c1a050",
    "detail-type": "EC2 Instance State-change Notification",
    "source": "aws.ec2",
    "account": "111122223333",
    "time": "2019-10-02T17:59:30Z",
    "region": "us-east-1",
    "resources": [
        "arn:aws:ec2:us-east-1:111122223333:instance/i-0c314xmplcd5b8173"
    ],
    "detail": {
        "instance-id": "i-0c314xmplcd5b8173",
        "state": "running"
    }
}
```

如需設定事件的詳細資訊，請參閱使用 Lambda 與 Amazon EventBridge 排程。如需處理 Amazon EBS 快照通知的範例函數，請參閱 Amazon EBS 的EventBridge 排程器。

您也可以使用 AWS 開發套件，透過 Amazon EC2 API 管理執行個體和其他資源。

許可

若要處理來自 Amazon EC2 的生命週期事件， EventBridge (CloudWatch 事件) 需要許可才能叫用您的函數。此許可來自函數的資源型政策。如果您使用 EventBridge (E CloudWatch vents) 主控台設定事件觸發器，則主控台會代表您更新以資源為基礎的策略。否則，新增如下的聲明：

Example Amazon EC2 生命週期通知的資源型政策聲明

```
{
  "Sid": "ec2-events",
  "Effect": "Allow",
  "Principal": {
    "Service": "events.amazonaws.com"
  },
  "Action": "lambda:InvokeFunction",
  "Resource": "arn:aws:lambda:us-east-1:12456789012:function:my-function",
  "Condition": {
    "ArnLike": {
      "AWS:SourceArn": "arn:aws:events:us-east-1:12456789012:rule/*"
    }
  }
}
```

若要新增陳述式，請使用add-permission AWS CLI 指令。

```
aws lambda add-permission --action lambda:InvokeFunction --statement-id ec2-events \
--principal events.amazonaws.com --function-name my-function --source-arn
 'arn:aws:events:us-east-1:12456789012:rule/*'
```

如果您的函數使用 AWS 開發套件來管理 Amazon EC2 資源，請將 Amazon EC2 許可新增至函數的執行角色。

教學課程：設定 Lambda 函數以 ElastiCache 在 Amazon VPC 中存取 Amazon

若要了解如何將 Lambda 設定ElastiCache 為 ElastiCache 在 Amazon VPC 中存取 Amazon，請參閱 Redis 使用者指南中的 Lambda 教學課程。

處理應用程式負載平衡器請求

您可以使用 Lambda 函數處理來自 Application Load Balancer 的請求。Elastic Load Balancing 支援 Lambda 函數作為 Application Load Balancer 的目標。使用負載平衡器規則，根據路徑或標頭值將 HTTP 請求路由至特定函式。由您的 Lambda 函數處理請求並傳回 HTTP 回應。

Elastic Load Balancer 將使用含有請求內文和中繼資料的事件，同步叫用您的 Lambda 函數。

Example Application Load Balancer 請求事件

```
{
    "requestContext": {
        "elb": {
            "targetGroupArn": "arn:aws:elasticloadbalancing:us-
east-1:123456789012:targetgroup/lambda-279XGJDqGZ5rsrHC2Fjr/49e9d65c45c6791a"
        }
    },
    "httpMethod": "GET",
    "path": "/lambda",
    "queryStringParameters": {
        "query": "1234ABCD"
    },
    "headers": {
        "accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/
webp,image/apng,*/*;q=0.8",
        "accept-encoding": "gzip",
        "accept-language": "en-US,en;q=0.9",
        "connection": "keep-alive",
        "host": "lambda-alb-123578498.us-east-1.elb.amazonaws.com",
        "upgrade-insecure-requests": "1",
        "user-agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
 (KHTML, like Gecko) Chrome/71.0.3578.98 Safari/537.36",
        "x-amzn-trace-id": "Root=1-5c536348-3d683b8b04734faae651f476",
        "x-forwarded-for": "72.12.164.125",
        "x-forwarded-port": "80",
        "x-forwarded-proto": "http",
        "x-imforwards": "20"
    },
    "body": "",
    "isBase64Encoded": False
}
```

您的函數會處理事件，並將 JSON 格式的回應文件傳回給負載平衡器。Elastic Load Balancing 會將文件轉換成 HTTP 成功或錯誤回應，並傳回給使用者。

Example 回應文件格式

```
{
    "statusCode": 200,
    "statusDescription": "200 OK",
    "isBase64Encoded": False,
    "headers": {
        "Content-Type": "text/html"
    },
    "body": "<h1>Hello from Lambda!</h1>"
}
```

若要設定 Application Load Balancer 作為函數觸發條件，請授予 Elastic Load Balancing 執行函數的許可、建立目標群組將請求路由至函數，並且為負載平衡器加入規則以傳送請求至目標群組。

使用 `add-permission` 命令，將許可陳述式加入至函式以資源為基礎的政策。

```
aws lambda add-permission --function-name alb-function \
--statement-id load-balancer --action "lambda:InvokeFunction" \
--principal elasticloadbalancing.amazonaws.com
```

您應該會看到下列輸出：

```
{
    "Statement": "{\"Sid\":\"load-balancer\",\"Effect\":\"Allow\",\"Principal\":
{\"Service\":\"elasticloadbalancing.amazonaws.com\"},\"Action\":\"lambda:InvokeFunction
\",\"Resource\":\"arn:aws:lambda:us-west-2:123456789012:function:alb-function\"}"
}
```

如需有關設定 Application Load Balancer 接聽程式和目標群組的說明，請參閱 Application Load Balancer 使用者指南中的 Lambda 函數作為目標。

搭配使用 Amazon EFS 與 Lambda

Lambda 與 Amazon Elastic File System (Amazon EFS) 整合，以支援 Lambda 應用程式的安全共用檔案系統存取。您可以設定函數，以便透過 VPC 內的本機網路，使用 NFS 協定進行初始化時，掛載檔案系統。Lambda 會管理連線，並加密進出檔案系統的所有流量。

檔案系統和 Lambda 函數必須位於相同的區域中。帳戶中的 Lambda 函數可以掛載不同帳戶中的檔案系統。在此案例中，您可以在函數 VPC 和檔案系統 VPC 之間設定 VPC 對等互連。

> **Note**
>
> 若要設定函數以連線至檔案系統，請參閱設定 Lambda 函數的檔案系統存取權。

Amazon EFS 支援檔案鎖定， 以防止多個函數嘗試同時寫入同一個檔案系統時發生損毀。Amazon EFS 中的鎖定功能遵循適用於建議型鎖定的 NFS v4.1 協定，可讓您的應用程式使用整個檔案和位元組範圍鎖定。

Amazon EFS 提供一些選項，以根據應用程式的需求自訂您的檔案系統，以便在擴展時保持高效能。有三個主要因素需要考量：連線數量、輸送量 (以每秒 MiB 為單位) 和 IOPS。

> **配額**
>
> 如需檔案系統限制的詳細資訊，請參閱 Amazon Elastic File System 使用者指南中的 Amazon EFS 檔案系統的配額。

為避免擴展、輸送量和 IOPS 發生問題，請監控 Amazon EFS 傳送給 Amazon CloudWatch 的指標。如需在 Amazon EFS 中監控的概觀，請參閱 Amazon Elastic File System 使用者指南中的監控 Amazon EFS。

章節

- 連線
- 輸送量
- IOPS

連線

Amazon EFS 每個檔案系統最多支援 25,000 個連線。在初始化期間，函數的每個執行個體都會建立一個與其檔案系統的單一連線，該連線會在整個叫用中持續存在。這表示您可以跨一或多個連線到檔案系統的函數，達到 25,000 個並行。若要限制函數建立的連線數量，請使用保留的並行。

然而，當您在擴展時向函數程式碼或組態進行變更，函數執行個體數量會暫時增加超過目前並行。Lambda 會佈建新的執行個體，以處理新的要求，在就職行個體關閉其與檔案系統的連線之前，會有一些延遲。為了避免在部署期間達到連線數量上限，請使用輪流部署。透過輪流部署，每次進行變更時，都會將流量逐漸轉移至新版本。

如果您從其他服務 (例如 Amazon EC2) 連線到相同的檔案系統，您應該也會注意到 Amazon EFS 中連線的擴展行為。檔案系統支援在一次高載中建立最多 3,000 個連線，之後每分鐘支援 500 個新連線。

若要監控並觸發連線上的警示，請使用 `ClientConnections` 指標。

輸送量

擴展時，也有可能超過檔案系統的輸送量上限。在高載模式 (預設值) 中，檔案系統具有低基準輸送量，可隨其大小呈線性擴展。若要允許高載活動，會授與檔案系統高載額度，允許其使用 100 MIB/秒或更多的輸送量。額度會持續累計，而且每次讀取和寫入操作都會耗用這些額度。如果檔案系統用完額度，它會節流超出基準輸送量的讀取和寫入操作，這可能會導致叫用逾時。

> **Note**
>
> 如果您使用佈建的並行，即使在閒置時，您的函數也可以取用高載額度。使用佈建並行，在叫用函數之前，Lambda 會初始化函數的執行個體，並每隔幾個小時回收一次執行個體。如果您在初始化期間使用附加檔案系統上的檔案，則此活動可以使用高載額度。

若要監控並觸發輸送量的警示，請使用 `BurstCreditBalance` 指標。當您的函數並行數量很低，該指標應該增加，當並行數量很高，則應該減少。如果在低活動期間，指標持續減少或累計不足以涵蓋尖峰流量，您可能需要限制函數的並行數量，或啟用佈建的輸送量。

IOPS

每秒輸入/輸出操作數量 (IOPS) 是檔案系統處理的讀取和寫入操作數量的測量。在一般用途模式中，IOPS 受到限制，有利於較低的延遲，這對大多數應用程式很有幫助。

若要在一般用途模式中監控 IOPS 並警示，請使用 `PercentIOLimit` 指標。如果此指標達到 100%，您的函數可能會逾時等待讀取和寫入操作完成。

使用 Lambda 與 Amazon EventBridge 排程

Amazon EventBridge Scheduler 是無伺服器排程器，可讓您從單一中央受管服務建立、執行和管理任務。使用 EventBridge Scheduler，您可以使用循環模式的 cron 和速率運算式來建立排程，或設定一次性呼叫。您可以設定彈性的交付時段、定義重試次數上限，以及設定未處理事件的最長保留時間。

當您使用 Lambda 設定 EventBridge 排程器時， EventBridge 排程器會以非同步方式叫用您的 Lambda 函數。本頁說明如何使用 EventBridge 排程器在排程上叫用 Lambda 函數。

設定執行角色

當您建立新排程時， EventBridge 排程器必須具有代表您呼叫其目標 API 作業的權限。您可以使用執行角色將這些權限授與 EventBridge 「排程器」。排程執行角色所連接的許可政策會定義哪些是必要許可。這些權限取決於您希望 EventBridge 排程器叫用的目標 API。

當您使用「 EventBridge 排程器」主控台建立排程時，如下列程序所示，「 EventBridge 排程器」會根據您選取的目標自動設定執行角色。如果您想要使用其中一個 S EventBridge cheduler SDK、或來建立排程 AWS CloudFormation，您必須具有現有的執行角色，以授與 EventBridge 排程器呼叫目標所需的權限。 AWS CLI如需有關手動設定排程執行角色的詳細資訊，請參閱《EventBridge 排程器使用指南》中的 < 設定執行角色 > 。

建立排程

使用主控台建立排程

1. 在 https://console.aws.amazon.com/scheduler/home 打開 Amazon EventBridge 調度程序控制台。
2. 在排程頁面上，選擇建立排程。
3. 在指定排程詳細資訊頁面的排程名稱和描述區段中，執行以下動作：

   a. 在排程名稱中，輸入排程的名稱，例如 **`MyTestSchedule`**。

   b. (選用) 在描述中，輸入對排程的描述，例如 **`My first schedule`**。

   c. 針對排程群組，從下拉式清單中選擇排程群組。如果您沒有群組，請選擇預設值。若要建立排程群組，請選擇建立自己的排程。

      您可以使用排程群組，為不同群組的排程加上標籤。

4. • 選擇排程選項。

頻率	執行此作業...	
一次性排程 一次性排程只會在您指定的日期與時間調用目標一次。	針對日期和時間執行以下動作： • 依 YYYY/MM/DD 格式輸入有效日期。 • 依 hh:mm 格式輸入時間戳記 (24 小時)。 • 針對時區選擇時區。	

頻率	執行此作業...	
週期性排程 週期性排程會依您指定的頻率，使用 cron 或 Rate 運算式調用目標。	a. 在排程模式中，執行下列其中一項動作： • 若要使用 Cron 運算式定義排程，請選擇 Cron 排程，然後輸入 Cron 運算式。 • 若要使用 Rate 表達式定義排程，請選擇 Rate 排程，然後輸入 Rate 表達式。 如需 Cron 和費率運算式的詳細資訊，請參閱 Amazon 排程器使用者指南中的 EventBridge 排程器上的 EventBridge 排程類型。 b. 對於彈性時段，選擇關閉可關閉此選項，或者也能選擇其中一個預先定義的時間範圍。例如，如果您選擇 15 分鐘並設定週期性排程，每小時調用目標一次，則排程會在每小時一開始的 15 分鐘內執行。	

5. (選用) 如果您在上一個步驟中選擇週期性排程，請在時間範圍區段執行以下動作：

   a. 針對時區選擇時區。

   b. 對於開始日期和時間，依 YYYY/MM/DD 格式輸入有效日期，接著依 24 小時的 hh:mm 格式指定時間戳記。

   c. 對於結束日期和時間，依 YYYY/MM/DD 格式輸入有效日期，接著依 24 小時的 hh:mm 格式指定時間戳記。

6. 選擇下一步。
7. 在「選取目標」 頁面上，選擇 EventBridge 排程器呼叫的 AWS API 作業：
   a. 選擇AWS Lambda 調用。
   b. 在調用區段中，選取函數或選擇建立新的 Lambda 函數。
   c. (選用) 輸入 JSON 承載。如果您未輸入裝載， EventBridge Scheduler 會使用空白事件來叫用函數。
8. 選擇下一步。
9. 在設定頁面執行以下動作：
   a. 若要開啟排程，請在排程狀態底下切換到啟用排程。
   b. 若要設定排程的重試政策，請在重試政策和無效字母佇列 (DLQ) 底下執行以下動作：
      - 切換到重試。
      - 針對事件的保留時間上限，輸入 EventBridge 排程器必須保留未處理事件的最大小時數和最小時數。
      - 時間最長可設為 24 小時。
      - 針對重試次數上限，輸入目標傳回錯誤時， EventBridge 排程器重試排程的次數上限。

        最大值為重試 185 次。

      使用重試原則時，如果排程無法呼叫其目標， EventBridge 排程器會重新執行排程。一旦設定此功能，您就必須設定排程的最長保留時間和重試次數。
   c. 選擇 EventBridge 排程器儲存未傳遞事件的位置。

無效字母佇列 (DLQ) 選項	執行此作業...	
不儲存	選擇無。	
將活動儲存在建立排程的相同 AWS 帳戶 位置	a. 選擇選取我中的一個 Amazon SQS 佇列 AWS 帳戶 作為 DL Q。 b. 選擇 Amazon SQS 佇列的 Amazon Resource Name (ARN)。	

無效字母佇列 (DLQ) 選項	執行此作業...	
將活動儲存在與建立排程不同 AWS 帳戶 的位置	a. 選擇 「 AWS 帳戶 將其他地方的 Amazon SQS 佇列指定為 DL Q」。 b. 輸入 Amazon SQS 佇列的 Amazon Resource Name (ARN)。	

d. 若要使用由客戶管理的金鑰加密您的目標輸入，請在加密底下選擇自訂加密設定 (進階)。

   如果選擇此選項，請輸入現有的 KMS 金鑰 ARN，或選擇建立 AWS KMS key，以導覽至 AWS KMS 控制台。如需 EventBridge 排程器如何加密靜態資料的詳細資訊，請參閱 Amazon EventBridge 排程器使用者指南中的靜態加密。

e. 若要讓 EventBridge 排程器為您建立新的執行角色，請選擇 [為此排程建立新角色]。接著輸入角色名稱。如果您選擇此選項， EventBridge Scheduler 會將範本化目標所需的必要權限附加至角色。

10. 選擇下一步。
11. 在檢閱和建立排程頁面上，檢閱排程的詳細資訊。在每個區段中選擇編輯，即可返回該步驟並編輯其詳細資訊。
12. 選擇建立排程。

    您可以在排程頁面檢視新建立和現有的排程。在狀態欄底下，確認您的新排程狀態為已啟用。

若要確認 EventBridge 排程器是否呼叫函式，請檢查函式的 Amazon CloudWatch 記錄。

相關資源

如需有關 EventBridge 排程器的詳細資訊，請參閱下列內容：

- EventBridge 排程器使用指南
- EventBridge 排程器 API 參考
- EventBridge 排程器定價

搭配使用 AWS Lambda 與 AWS IoT

AWS IoT 提供網際網路連線裝置 (例如感應器) 和 AWS 雲端間的安全通訊。這可以讓您收集、存放和分析來自多個裝置的遙測資料。

您可以為您的裝置建立 AWS IoT 規則，使其和 AWS 服務互動。AWS IoT 規則引擎提供 SQL 類型的語言，可讓您從訊息酬載中選取資料，以及傳送資料至其他服務，例如 Amazon S3、Amazon DynamoDB 和 AWS Lambda。當您希望叫用其他 AWS 服務或第三方服務時，您可以定義規則來叫用 Lambda 函數。

當傳入的 IoT 訊息觸發規則時，AWS IoT 會以非同步方式叫用您的 Lambda 函數，並將資料從 IoT 訊息傳遞至函數。

以下範例示範從溫室感應器讀取濕度。資料列和資料行的值會識別感應器的位置。此範例事件是以 AWS IoT 規則教學中的溫室類型為基礎。

Example AWS IoT 訊息事件

```
{
    "row" : "10",
    "pos" : "23",
    "moisture" : "75"
}
```

針對非同步叫用，Lambda 會將訊息排入佇列，並且在您的函式傳回錯誤時重試。為您的函數設定目的地來保留函數無法處理的事件。

您需要授予許可，AWS IoT 服務才能叫用您的 Lambda 函數。使用 `add-permission` 命令，將許可陳述式加入至函式以資源為基礎的政策。

```
aws lambda add-permission --function-name my-function \
--statement-id iot-events --action "lambda:InvokeFunction" --principal
 iot.amazonaws.com
```

您應該會看到下列輸出：

```
{
    "Statement": "{\"Sid\":\"iot-events\",\"Effect\":\"Allow\",\"Principal\":
{\"Service\":\"iot.amazonaws.com\"},\"Action\":\"lambda:InvokeFunction\",\"Resource\":
\"arn:aws:lambda:us-east-1:123456789012:function:my-function\"}"
```

```
}
```

如需如何搭配使用 Lambda 與 AWS IoT 的詳細資訊，請參閱建立 AWS Lambda 規則。

AWS Lambda 與 Amazon 數據 Firehose 一起使用

Amazon 資料 Firehose 擷取、轉換串流資料，並將其載入下游服務，例如 Apache Flink 或 Amazon S3 的受管服務。您可以撰寫 Lambda 函數，在往下游傳送前請求其他自訂的資料處理程序。

Example Amazon 數據 Firehose 消息事件

```
{
  "invocationId": "invoked123",
  "deliveryStreamArn": "aws:lambda:events",
  "region": "us-west-2",
  "records": [
    {
      "data": "SGVsbG8gV29ybGQ=",
      "recordId": "record1",
      "approximateArrivalTimestamp": 1510772160000,
      "kinesisRecordMetadata": {
        "shardId": "shardId-000000000000",
        "partitionKey": "4d1ad2b9-24f8-4b9d-a088-76e9947c317a",
        "approximateArrivalTimestamp": "2012-04-23T18:25:43.511Z",
        "sequenceNumber": "49546986683135544286507457936321625675700192471156785154",
        "subsequenceNumber": ""
      }
    },
    {
      "data": "SGVsbG8gV29ybGQ=",
      "recordId": "record2",
      "approximateArrivalTimestamp": 151077216000,
      "kinesisRecordMetadata": {
        "shardId": "shardId-000000000001",
        "partitionKey": "4d1ad2b9-24f8-4b9d-a088-76e9947c318a",
        "approximateArrivalTimestamp": "2012-04-23T19:25:43.511Z",
        "sequenceNumber": "49546986683135544286507457936321625675700192471156785155",
        "subsequenceNumber": ""
      }
    }
  ]
}
```

如需詳細資訊，請參閱《Firehose 開發人員指南》中的 Amazon 資料 Firehose 資料轉換。

搭配使用 AWS Lambda 與 Amazon Lex

您可以使用 Amazon Lex 將對話機器人整合至您的應用程式中。Amazon Lex 機器人提供與使用者的對話界面。Amazon Lex 提供與 Lambda 的預先建置整合，使您能夠搭配使用 Amazon Lex 機器人與 Lambda 函數。

設定 Amazon Lex 機器人時，您可以指定 Lambda 函數執行驗證、履行或兩者。對於驗證，Amazon Lex 會在使用者每次回應後叫用 Lambda 函數。該 Lambda 函數可驗證回應，並在必要時向使用者提供修正意見回饋。為了實現意圖，Amazon Lex 會在機器人成功收集所有必要資訊及收到使用者確認後，叫用 Lambda 函數以履行使用者請求。

您可以管理 Lambda 函數的並行，以控制您服務之同時機器人對談的最大數量。如果函數處於最大並行狀態，Amazon Lex API 會傳回 HTTP 429 狀態碼 (太多請求)。

如果 Lambda 函數擲回例外狀況，API 會傳回 HTTP 424 狀態碼 (相依性失敗例外)。

Amazon Lex 機器人會同步叫用您的 Lambda 函數。事件參數包含機器人和對話方塊中每個插槽值的相關資訊。如需事件和回應欄位的定義，請參閱《Amazon Lex 開發人員指南》中的 Lambda 事件和回應格式。Amazon Lex 訊息事件中的`invocationSource`參數會指出 Lambda 函數是否應驗證輸入 (DialogCodeHook) 或履行意圖 (FulfillmentCodeHook)。

若要取得說明如何搭配使用 Lambda 與 Amazon Lex 的範例教學，請參閱 Amazon Lex 開發人員指南中的練習 1：使用藍圖建立 Amazon Lex 機器人。

角色和許可

您需要將服務連結角色設定為函數的執行角色。Amazon Lex 會定義具有預先定義許可的服務連結角色。當您使用主控台建立 Amazon Lex 機器人時，會自動建立服務連結角色。若要使用 AWS CLI 建立服務連結角色，請使用 `create-service-linked-role` 命令。

```
aws iam create-service-linked-role --aws-service-name lex.amazonaws.com
```

這個命令會建立下列角色。

```
{
  "Role": {
      "AssumeRolePolicyDocument": {
          "Version": "2012-10-17",
          "Statement": [
              {
```

```
                "Action": "sts:AssumeRole",
                "Effect": "Allow",
                "Principal": {
                    "Service": "lex.amazonaws.com"
                }
            }
        ]
    },
    "RoleName": "AWSServiceRoleForLexBots",
    "Path": "/aws-service-role/lex.amazonaws.com/",
    "Arn": "arn:aws:iam::account-id:role/aws-service-role/lex.amazonaws.com/
AWSServiceRoleForLexBots"
}
```

如果您的 Lambda 函數使用其他 AWS 服務，您需要將對應的許可新增至服務連結角色。

您可以使用資源型許可政策來允許 Amazon Lex 機器人調用 Lambda 函數。如果您使用 Amazon Lex 主控台，則會自動建立許可政策。在 AWS CLI 中，使用 Lambda `add-permission` 命令來設定許可。

若使用 Amazon Lex V2，請執行下列命令。在來源 ARN 中，以 AWS 區域 取代 Amazon Lex 機器人所在的 `us-east-1`，並使用您自己的 AWS 帳戶編號和機器人別名。

```
aws lambda add-permission \
    --function-name LexCodeHook \
    --statement-id LexInvoke-MyBot \
    --action lambda:InvokeFunction \
    --principal lex.amazonaws.com \
    --source-arn "arn:aws:lex:us-east-1:123456789012:bot-alias/MYBOT/MYBOTALIAS"
```

您也可以使用 Amazon Lex V1 調用 Lambda 函數。若使用 Amazon Lex V1，請執行下列命令。在來源 ARN 中，以 `us-east-1` 取代 Amazon Lex 意圖所在的 AWS 區域，並使用您自己的 AWS 帳戶編號和意圖名稱。

```
aws lambda add-permission \
    --function-name LexCodeHook \
    --statement-id LexInvoke-MyIntent \
    --action lambda:InvokeFunction \
    --principal lex.amazonaws.com \
```

```
    --source-arn "arn:aws:lex:us-east-1:123456789012 ID:intent:MYINTENT:*"
```

請注意，我們已停止維護 Amazon Lex V1。建議您使用 Amazon Lex V2。

AWS Lambda 與 Amazon RDS 一起使用

您可以直接或透過 Amazon RDS Proxy 將 Lambda 函數連線到 Amazon Relational Database Service (Amazon RDS)。直接連線適用於簡單的案例，生產環境則建議使用代理。資料庫代理管理許多共用資料庫連線，讓函數在不耗盡資料庫連線的情況下達到高並行層級。

我們建議將 Amazon RDS Proxy 用於 Lambda 函數，這些函數會頻繁進行短資料庫連線，或是開啟和關閉大量資料庫連線。

設定函數

在 Lambda 主控台中，您可以佈建和設定 Amazon RDS 資料庫執行個體和代理資源。如需詳細資訊，請參閱組態索引標籤下的 RDS 資料庫。或者，您也可以在 Amazon RDS 主控台中建立與設定 Lambda 函數的連線。

- 若要連線到資料庫，您的函數必須位於資料庫執行所在的相同 Amazon VPC 內。
- 您可以搭配 MySQL、MariaDB、PostgreSQL 或 Microsoft SQL Server 引擎，使用 Amazon RDS 資料庫。
- 您也可以搭配 MySQL 或 PostgreSQL 引擎，使用 Aurora DB 叢集。
- 您需要提供 Secrets Manager 秘密以用於資料庫身分驗證。
- IAM 角色必須提供使用秘密的許可，而受信任的政策必須允許 Amazon RDS 擔任該角色。
- 使用主控台設定 Amazon RDS 資源並將其連接到您的函數的 IAM 原則必須具有下列許可：

> **Note**
>
> 只有在將 Amazon RDS 代理設定為管理資料庫連線集區時，才需要 Amazon RDS 代理許可。

Example 許可政策

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "ec2:CreateSecurityGroup",
```

```
        "ec2:DescribeSecurityGroups",
        "ec2:DescribeSubnets",
        "ec2:DescribeVpcs",
        "ec2:AuthorizeSecurityGroupIngress",
        "ec2:AuthorizeSecurityGroupEgress",
        "ec2:RevokeSecurityGroupEgress",
        "ec2:CreateNetworkInterface",
        "ec2:DeleteNetworkInterface",
        "ec2:DescribeNetworkInterfaces"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "rds-db:connect",
        "rds:CreateDBProxy",
        "rds:CreateDBInstance",
        "rds:CreateDBSubnetGroup",
        "rds:DescribeDBClusters",
        "rds:DescribeDBInstances",
        "rds:DescribeDBSubnetGroups",
        "rds:DescribeDBProxies",
        "rds:DescribeDBProxyTargets",
        "rds:DescribeDBProxyTargetGroups",
        "rds:RegisterDBProxyTargets",
        "rds:ModifyDBInstance",
        "rds:ModifyDBProxy"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "lambda:CreateFunction",
        "lambda:ListFunctions",
        "lambda:UpdateFunctionConfiguration"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "iam:AttachRolePolicy",
```

```
        "iam:AttachPolicy",
        "iam:CreateRole",
        "iam:CreatePolicy"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "secretsmanager:GetResourcePolicy",
        "secretsmanager:GetSecretValue",
        "secretsmanager:DescribeSecret",
        "secretsmanager:ListSecretVersionIds",
        "secretsmanager:CreateSecret"
      ],
      "Resource": "*"
    }
  ]
}
```

Amazon RDS 會按資料庫執行個體大小收取代理程式的小時費率，請參閱 RDS 代理定價以了解詳細資訊。如需代理連線的一般詳細資訊，請參閱《Amazon RDS 使用者指南》中的使用 Amazon RDS Proxy。

> **Lambda 和 Amazon RDS 設定**
>
> Lambda 和 Amazon RDS 主控台都將協助您自動設定一些必要的資源，以便在 Lambda 和 Amazon RDS 之間建立連線。

使用 Lambda 函數 Connect 到 Amazon RDS 資料庫

下列程式碼範例示範如何實作連線至 Amazon RDS 資料庫的 Lambda 函數。該函數提出了一個簡單的數據庫請求，並返回結果。

Go

SDK for Go V2

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 連線至 Lambda 函數中的 Amazon RDS 資料庫。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
/*
Golang v2 code here.
*/

package main

import (
 "context"
 "database/sql"
 "encoding/json"
 "fmt"

 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/feature/rds/auth"
 _ "github.com/go-sql-driver/mysql"
)

type MyEvent struct {
 Name string `json:"name"`
}

func HandleRequest(event *MyEvent) (map[string]interface{}, error) {

 var dbName string = "DatabaseName"
 var dbUser string = "DatabaseUser"
 var dbHost string = "mysqldb.123456789012.us-east-1.rds.amazonaws.com"
 var dbPort int = 3306
 var dbEndpoint string = fmt.Sprintf("%s:%d", dbHost, dbPort)
```

```
 var region string = "us-east-1"

 cfg, err := config.LoadDefaultConfig(context.TODO())
 if err != nil {
  panic("configuration error: " + err.Error())
 }

 authenticationToken, err := auth.BuildAuthToken(
  context.TODO(), dbEndpoint, region, dbUser, cfg.Credentials)
 if err != nil {
  panic("failed to create authentication token: " + err.Error())
 }

 dsn := fmt.Sprintf("%s:%s@tcp(%s)/%s?tls=true&allowCleartextPasswords=true",
  dbUser, authenticationToken, dbEndpoint, dbName,
 )

 db, err := sql.Open("mysql", dsn)
 if err != nil {
  panic(err)
 }

 defer db.Close()

 var sum int
 err = db.QueryRow("SELECT ?+? AS sum", 3, 2).Scan(&sum)
 if err != nil {
  panic(err)
 }
 s := fmt.Sprint(sum)
 message := fmt.Sprintf("The selected sum is: %s", s)

 messageBytes, err := json.Marshal(message)
 if err != nil {
  return nil, err
 }

 messageString := string(messageBytes)
 return map[string]interface{}{
  "statusCode": 200,
  "headers":    map[string]string{"Content-Type": "application/json"},
  "body":       messageString,
 }, nil
}
```

```
func main() {
 lambda.Start(HandleRequest)
}
```

JavaScript

適用於 JavaScript (v2) 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 JavaScript 連接到一個 Lambda 函數中的 Amazon RDS 數據庫。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
/*
Node.js code here.
*/
// ES6+ example
import { Signer } from "@aws-sdk/rds-signer";
import mysql from 'mysql2/promise';

async function createAuthToken() {
  // Define connection authentication parameters
  const dbinfo = {

    hostname: process.env.ProxyHostName,
    port: process.env.Port,
    username: process.env.DBUserName,
    region: process.env.AWS_REGION,

  }

  // Create RDS Signer object
  const signer = new Signer(dbinfo);

  // Request authorization token from RDS, specifying the username
```

```
  const token = await signer.getAuthToken();
  return token;
}

async function dbOps() {

  // Obtain auth token
  const token = await createAuthToken();
  // Define connection configuration
  let connectionConfig = {
    host: process.env.ProxyHostName,
    user: process.env.DBUserName,
    password: token,
    database: process.env.DBName,
    ssl: 'Amazon RDS'
  }
  // Create the connection to the DB
  const conn = await mysql.createConnection(connectionConfig);
  // Obtain the result of the query
  const [res,] = await conn.execute('select ?+? as sum', [3, 2]);
  return res;

}

export const handler = async (event) => {
  // Execute database flow
  const result = await dbOps();
  // Return result
  return {
    statusCode: 200,
    body: JSON.stringify("The selected sum is: " + result[0].sum)
  }
};
```

處理來自 Amazon RDS 的事件通知

您可以使用 Lambda 來處理 Amazon RDS 資料庫的事件通知。Amazon RDS 會將通知傳送到 Amazon Simple Notification Service (Amazon SNS) 主題，您可以進行設定，透過該主題叫用 Lambda 函數。Amazon SNS 會將來自 Amazon RDS 的訊息包裝在自己的事件文件中，並將其傳送到函數。

如需設定 Amazon RDS 資料庫以傳送通知的詳細資訊，請參閱使用 Amazon RDS 事件通知。

Example Amazon SNS 事件中的 Amazon RDS 訊息

```
{
        "Records": [
          {
            "EventVersion": "1.0",
            "EventSubscriptionArn": "arn:aws:sns:us-east-2:123456789012:rds-
lambda:21be56ed-a058-49f5-8c98-aedd2564c486",
            "EventSource": "aws:sns",
            "Sns": {
              "SignatureVersion": "1",
              "Timestamp": "2023-01-02T12:45:07.000Z",
              "Signature": "tcc6faL2yUC6dgZdmrwh1Y4cGa/ebXEkAi6RibDsvpi
+tE/1+82j...65r==",
              "SigningCertUrl": "https://sns.us-east-2.amazonaws.com/
SimpleNotificationService-ac565b8b1a6c5d002d285f9598aa1d9b.pem",
              "MessageId": "95df01b4-ee98-5cb9-9903-4c221d41eb5e",
              "Message": "{\"Event Source\":\"db-instance\",\"Event Time\":\"2023-01-02
 12:45:06.000\",\"Identifier Link\":\"https://console.aws.amazon.com/rds/home?
region=eu-west-1#dbinstance:id=dbinstanceid\",\"Source ID\":\"dbinstanceid\",\"Event ID
\":\"http://docs.amazonwebservices.com/AmazonRDS/latest/UserGuide/USER_Events.html#RDS-
EVENT-0002\",\"Event Message\":\"Finished DB Instance backup\"}",
              "MessageAttributes": {},
              "Type": "Notification",
              "UnsubscribeUrl": "https://sns.us-east-2.amazonaws.com/?
Action=Unsubscribe&amp;SubscriptionArn=arn:aws:sns:us-east-2:123456789012:test-
lambda:21be56ed-a058-49f5-8c98-aedd2564c486",
              "TopicArn":"arn:aws:sns:us-east-2:123456789012:sns-lambda",
              "Subject": "RDS Notification Message"
            }
          }
        ]
      }
```

Lambda 和 Amazon RDS 教學

- 使用 Lambda 函數來存取 Amazon RDS – 在《Amazon RDS 誰用著指南》中，學習如何使用 Lambda 函數並透過 Amazon RDS Proxy 將資料寫入 Amazon RDS 資料庫。每當新增訊息，您的 Lambda 函數將從 Amazon SQS 佇列中讀取記錄，然後將新項目寫入資料庫中的資料表。

使用 Lambda 處理 Amazon S3 事件通知

您可以使用 Lambda 來處理來自 Amazon Simple Storage Service 的事件通知。建立或刪除物件時，Amazon S3 可以將事件傳送至 Lambda 函數。您在儲存貯體上設定通知設定，並授予 Amazon S3 許可以在函數以資源為基礎的許可政策上叫用函數。

> **Warning**
>
> 如果 Lambda 函數使用的是觸發的相同儲存貯體，這可能會造成函數在迴圈中執行。例如，如果儲存貯體在物件每次上傳時都觸發函數，且該函數會將物件上傳至儲存貯體，則函數會間接地觸發本身。若要避免此狀況，請使用兩個儲存貯體，或將觸發設定為僅套用至傳入物件所用的字首。

Amazon S3 使用包含物件詳細資訊的事件以非同步方式叫用您的函數。以下範例顯示當部署套裝服務上傳至 Amazon S3 時，Amazon S3 傳送的事件。

Example Amazon S3 通知事件

```
{
  "Records": [
    {
      "eventVersion": "2.1",
      "eventSource": "aws:s3",
      "awsRegion": "us-east-2",
      "eventTime": "2019-09-03T19:37:27.192Z",
      "eventName": "ObjectCreated:Put",
      "userIdentity": {
        "principalId": "AWS:AIDAINPONIXQXHT3IKHL2"
      },
      "requestParameters": {
        "sourceIPAddress": "205.255.255.255"
      },
      "responseElements": {
        "x-amz-request-id": "D82B88E5F771F645",
        "x-amz-id-2":
 "vlR7PnpV2Ce81l0PRw6jlUpck7Jo5ZsQjryTjKlc5aLWGVHPZLj5NeC6qMa0emYBDXOo6QBU0Wo="
      },
      "s3": {
        "s3SchemaVersion": "1.0",
        "configurationId": "828aa6fc-f7b5-4305-8584-487c791949c1",
```

```
        "bucket": {
          "name": "DOC-EXAMPLE-BUCKET",
          "ownerIdentity": {
            "principalId": "A3I5XTEXAMAI3E"
          },
          "arn": "arn:aws:s3:::lambda-artifacts-deafc19498e3f2df"
        },
        "object": {
          "key": "b21b84d653bb07b05b1e6b33684dc11b",
          "size": 1305107,
          "eTag": "b21b84d653bb07b05b1e6b33684dc11b",
          "sequencer": "0C0F6F405D6ED209E1"
        }
      }
    }
  ]
}
```

若要叫用您的函數，Amazon S3 需要該函數以資源為基礎政策的許可。當您在 Lambda 主控台中設定 Amazon S3 觸發條件時，主控台會修改以資源為基礎的政策，讓 Amazon S3 在儲存貯體名稱與帳戶 ID 相符時叫用該函數。如果您在 Amazon S3 中設定通知，您要使用 Lambda API 更新政策。您也可以使用 Lambda API 將許可授予另一個帳戶，或將許可限制為指定的別名。

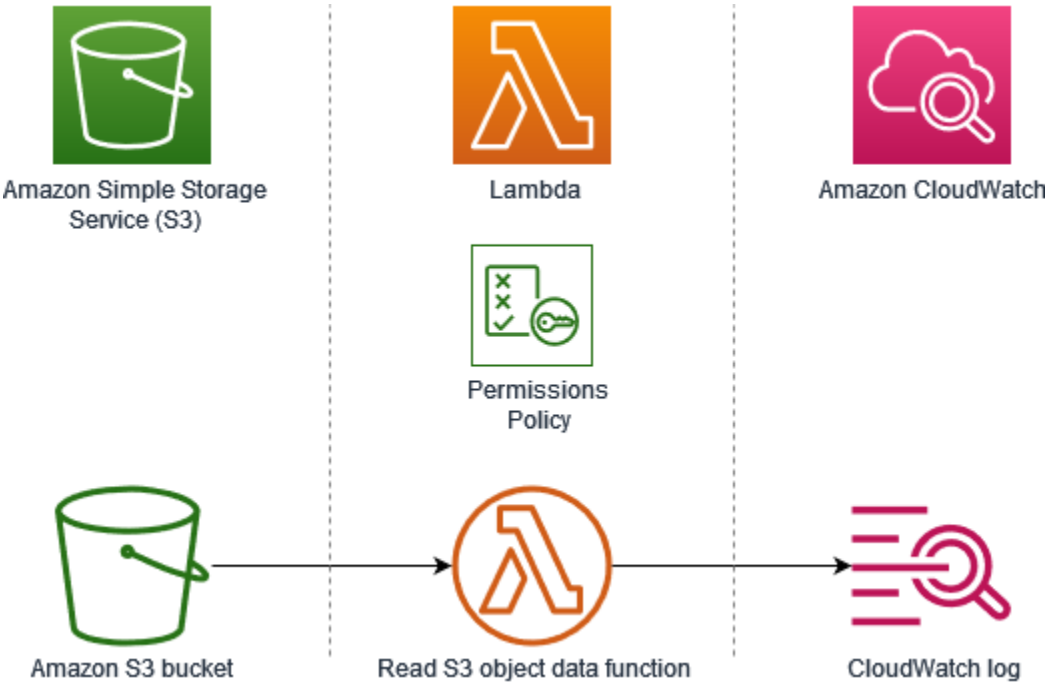
如果您的函數使用 AWS 開發套件來管理 Amazon S3 資源，則其執行角色也需要 Amazon S3 許可。

主題

- 教學課程：使用 Amazon S3 觸發條件調用 Lambda 函數
- 教學課程：使用 Amazon S3 觸發條件建立縮圖影像

教學課程：使用 Amazon S3 觸發條件調用 Lambda 函數

在本教學課程中，您將使用主控台建立 Lambda 函數，並設定 Amazon Simple Storage Service (Amazon S3) 儲存貯體的觸發條件。每次將物件新增到 Amazon S3 儲存貯體時，您的函數都會執行並將物件類型輸出到 Amazon CloudWatch 日誌。



本教學課程示範如何：

1. 建立 Amazon S3 儲存貯體。
2. 建立一個 Lambda 函數，用於傳回 Amazon S3 儲存貯體物件的物件類型。
3. 設定一個 Lambda 觸發條件，用於在有物件上傳至儲存貯體時調用函數。
4. 先使用虛擬事件測試函數，再使用觸發條件進行測試。

完成這些步驟後，您將了解如何設定 Lambda 函數，讓函數在 Amazon S3 儲存貯體中有物件新增或刪除時執行。您只能使用 AWS Management Console 來完成本教學課程。

必要條件

註冊一個 AWS 帳戶

如果您沒有 AWS 帳戶，請完成以下步驟來建立一個。

若要註冊成為 AWS 帳戶

1. 開啟 https://portal.aws.amazon.com/billing/signup。
2. 請遵循線上指示進行。

   部分註冊程序需接收來電，並在電話鍵盤輸入驗證碼。

當您註冊一個時 AWS 帳戶，將創建AWS 帳戶根使用者一個。根使用者有權存取該帳戶中的所有 AWS 服務 和資源。安全性最佳做法是將管理存取權指派給使用者，並僅使用 root 使用者來執行需要 root 使用者存取權的工作。

AWS 註冊過程完成後，會向您發送確認電子郵件。您可以隨時登錄 https://aws.amazon.com/ 並選擇我的帳戶，以檢視您目前的帳戶活動並管理帳戶。

建立具有管理權限的使用者

註冊後，請保護您的 AWS 帳戶 AWS 帳戶根使用者 AWS IAM Identity Center、啟用和建立系統管理使用者，這樣您就不會將 root 使用者用於日常工作。

保護您的 AWS 帳戶根使用者

1. 選擇 Root 使用者並輸入您的 AWS 帳戶 電子郵件地址，以帳戶擁有者身分登入。AWS Management Console在下一頁中，輸入您的密碼。

   如需使用根使用者登入的說明，請參閱 AWS 登入 使用者指南中的以根使用者身分登入。
2. 若要在您的根使用者帳戶上啟用多重要素驗證 (MFA)。

   如需指示，請參閱《IAM 使用者指南》中的為 AWS 帳戶 根使用者啟用虛擬 MFA 裝置 (主控台)。

建立具有管理權限的使用者

1. 啟用 IAM Identity Center。

   如需指示，請參閱 AWS IAM Identity Center 使用者指南中的啟用 AWS IAM Identity Center。
2. 在 IAM 身分中心中，將管理存取權授予使用者。

   若要取得有關使用 IAM Identity Center 目錄 做為身分識別來源的自學課程，請參閱《使用指南》IAM Identity Center 目錄中的「以預設值設定使用AWS IAM Identity Center 者存取」。

以具有管理權限的使用者身分登入

- 若要使用您的 IAM Identity Center 使用者簽署，請使用建立 IAM Identity Center 使用者時傳送至您電子郵件地址的簽署 URL。

如需使用 IAM 身分中心使用者登入的說明，請參閱使用AWS 登入 者指南中的登入 AWS 存取入口網站。

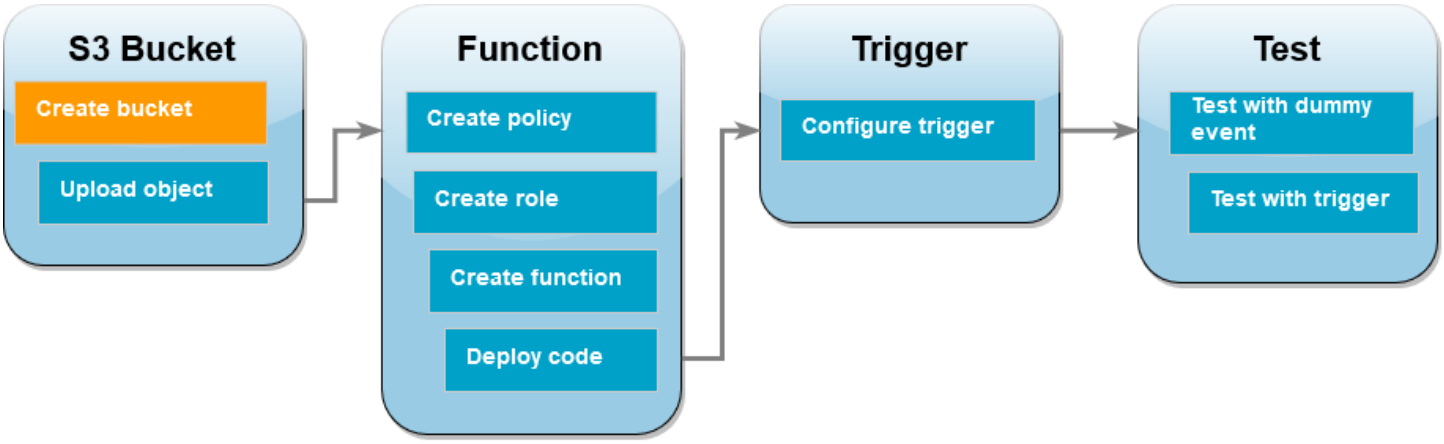
指派存取權給其他使用者

1. 在 IAM 身分中心中，建立遵循套用最低權限許可的最佳做法的權限集。

   如需指示，請參閱《AWS IAM Identity Center 使用指南》中的「建立權限集」。
2. 將使用者指派給群組，然後將單一登入存取權指派給群組。

   如需指示，請參閱《AWS IAM Identity Center 使用指南》中的「新增群組」。

建立 Amazon S3 儲存貯體



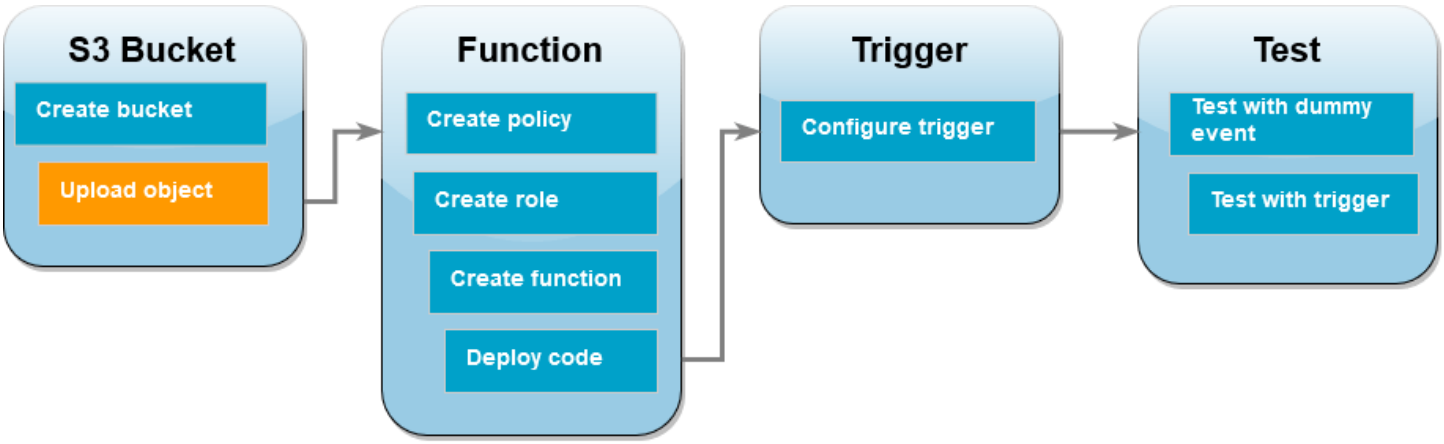
建立 Amazon S3 儲存貯體

1. 開啟 Amazon S3 主控台，然後選取儲存貯體頁面。
2. 選擇 建立儲存貯體 。
3. 在 General configuration (一般組態) 下，執行下列動作：

   a. 請在儲存貯體名稱輸入符合 Amazon S3 儲存貯體命名規則的全域唯一名稱。儲存貯體名稱只能包含小寫字母、數字、句點 (.) 和連字號 (-)。

   b. 對於 AWS 區域，選擇一個區域。在本教學課程稍後的階段，您必須在同一區域中建立 Lambda 函數。
4. 其他所有選項維持設為預設值，然後選擇建立儲存貯體。

將測試物件上傳至儲存貯體

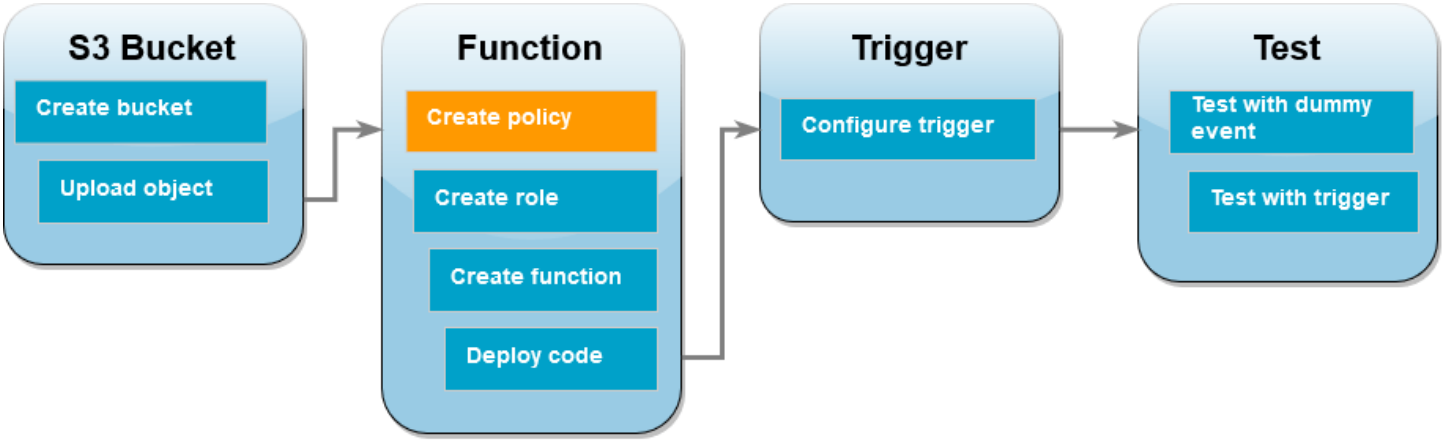


上傳測試物件

1. 開啟 Amazon S3 主控台的儲存貯體頁面，然後選擇您在上一個步驟中建立的儲存貯體。
2. 選擇上傳。
3. 選擇「新增檔案」，然後選取您要上載的物件。您可以選取任何檔案 (例如，`HappyFace.jpg`)。
4. 選擇開啟，然後選擇上傳。

稍後在教學課程中，您將使用此物件測試 Lambda 函數。

建立許可政策



建立允許 Lambda 從 Amazon S3 儲存貯體取得物件並寫入 Amazon CloudWatch 日誌的許可政策。

建立政策

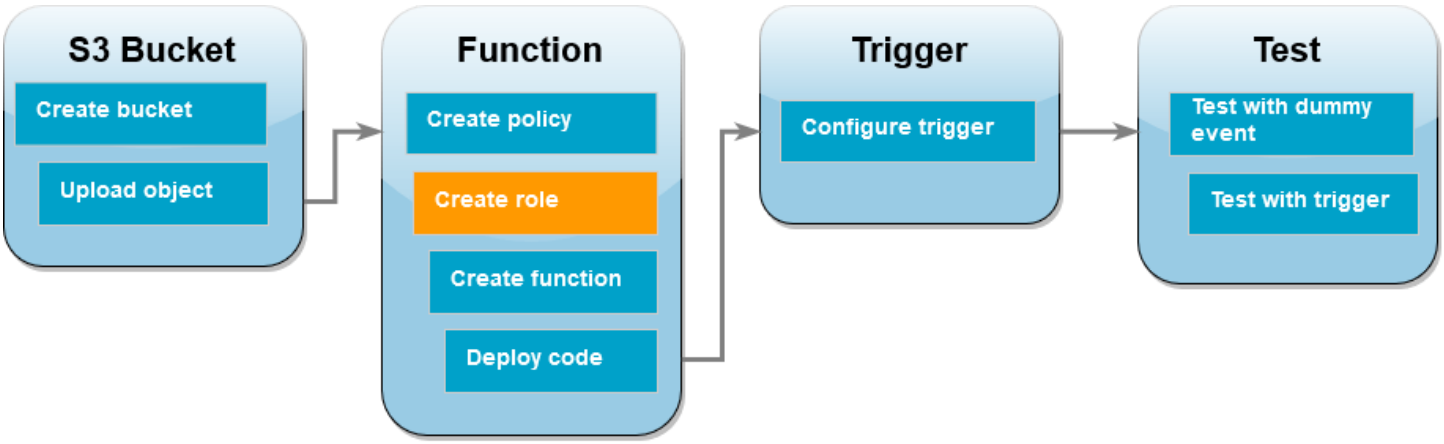
1. 開啟 IAM 主控台中的 政策 頁面。

2. 選擇 建立政策 。
3. 選擇 JSON 索引標籤，然後將下列政策貼到 JSON 編輯器。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:PutLogEvents",
                "logs:CreateLogGroup",
                "logs:CreateLogStream"
            ],
            "Resource": "arn:aws:logs:*:*:*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject"
            ],
            "Resource": "arn:aws:s3:::*/*"
        }
    ]
}
```

4. 選擇 下一步：標籤 。
5. 選擇 下一步：檢閱 。
6. 在 檢閱政策 下，針對政策 名稱 ，輸入 **s3-trigger-tutorial**。
7. 選擇 建立政策 。

建立執行角色

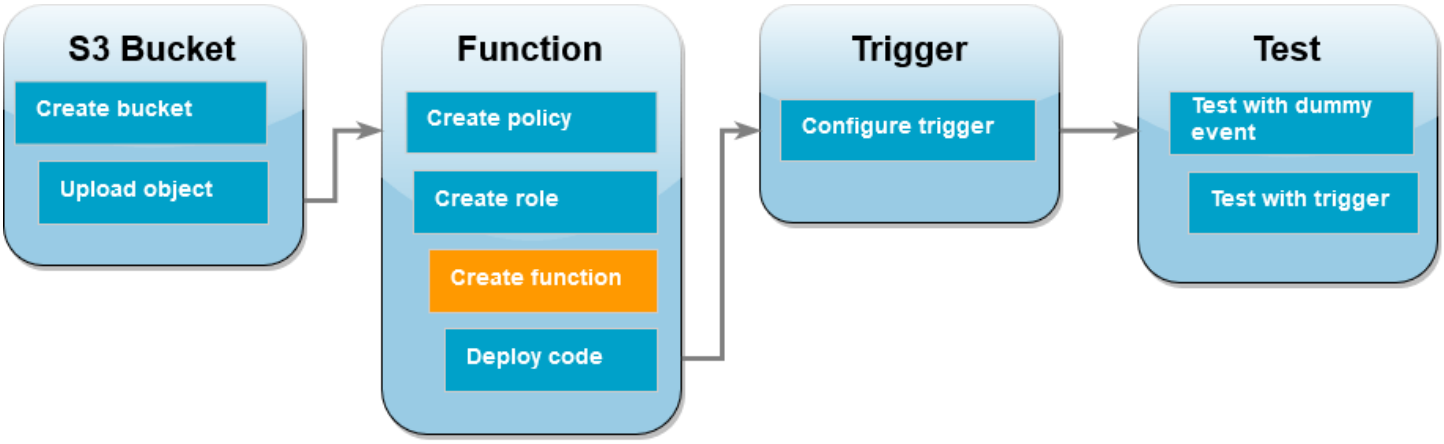


執行角色是一種 AWS Identity and Access Management (IAM) 角色，可授與 Lambda 函數存取 AWS 服務和資源的權限。在此步驟中，使用您在上一個步驟中建立的權限原則建立執行角色。

建立執行角色並附加自訂許可政策

1. 開啟 IAM 主控台中的 角色頁面 。
2. 選擇 建立角色。
3. 信任的實體類型請選擇 AWS 服務，使用案例則選擇 Lambda。
4. 選擇 下一步。
5. 在政策搜尋方塊中，輸入 **`s3-trigger-tutorial`**。
6. 在搜尋結果中，選取您建立的政策 (`s3-trigger-tutorial`)，然後選擇 下一步 。
7. 在 角色詳細資料 底下，角色名稱 請輸入 **`lambda-s3-trigger-role`**，然後選擇 建立角色 。

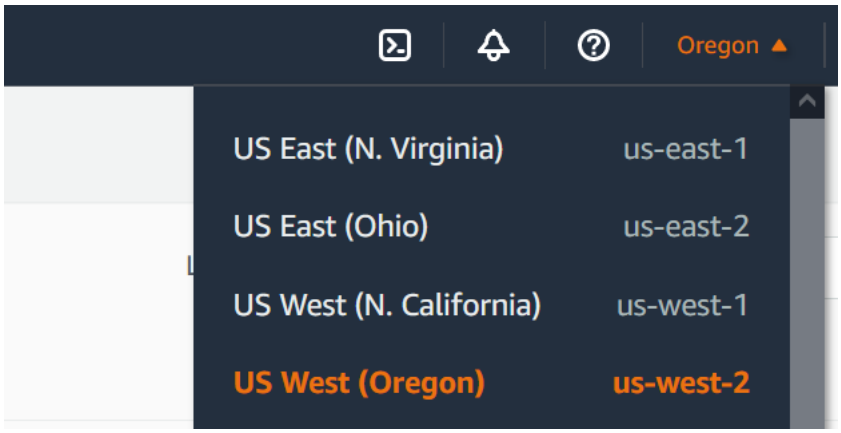
建立 Lambda 函式



使用 Python 3.12 執行階段，在主控台中建立 Lambda 函數。

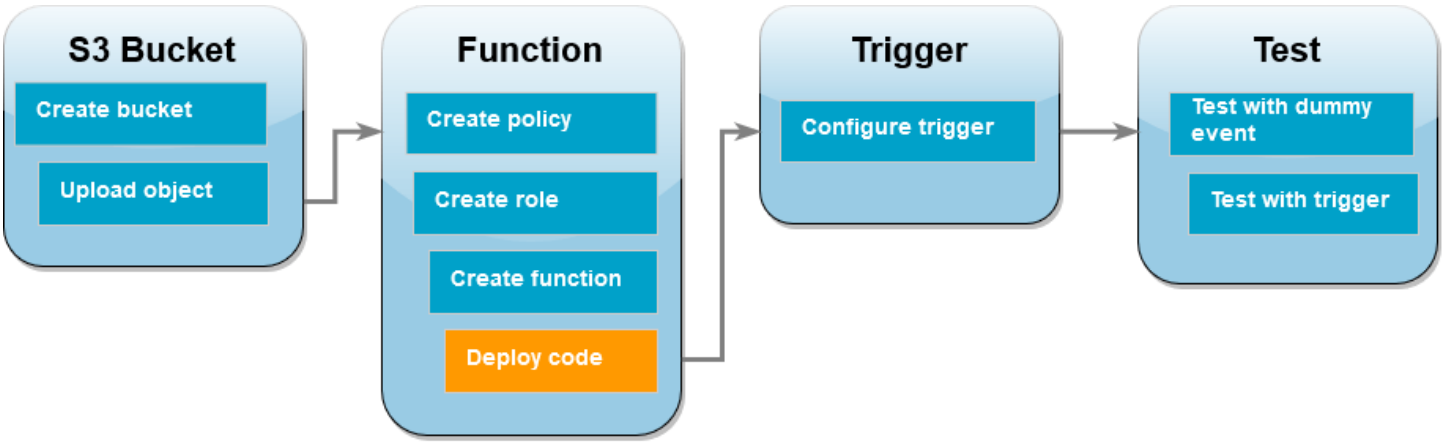
建立 Lambda 函數

1. 開啟 Lambda 主控台中的函數頁面。
2. 請確定您在建立 Amazon S3 儲存貯體的相同 AWS 區域 內容中工作。您可使用螢幕頂端的下拉式清單來變更區域。



3. 選擇建立函數。
4. 選擇從頭開始撰寫
5. 在基本資訊下，請執行下列動作：
   a. 在函數名稱輸入 `s3-trigger-tutorial`
   b. 對於「執行階段」，請選擇 Python 3.12。
   c. 對於 Architecture (架構)，選擇 x86_64。
6. 在變更預設執行角色索引標籤中，執行下列操作：
   a. 展開索引標籤，然後選擇使用現有角色。
   b. 選擇您之前建立的 `lambda-s3-trigger-role`。
7. 選擇建立函數。

部署函數程式碼



本教程使用 Python 3.12 運行時，但我們也提供了其他運行時的示例代碼文件。您可以在下列方塊中選取索引標籤，查看您感興趣的執行期程式碼。

Lambda 函數會從從 Amazon S3 接收到的`event`參數擷取上傳物件的金鑰名稱，以及儲存貯體的名稱。然後，函數會使用來自的 get_object 方法 AWS SDK for Python (Boto3) 來擷取物件的中繼資料，包括上載物件的內容類型 (MIME 類型)。

部署函數程式碼

1. 在下面的框中選擇 Python 選項卡並複製代碼。

   .NET

   AWS SDK for .NET

   > **Note**
   >
   > 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

   使用 .NET 搭配 Lambda 來使用 S3 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Threading.Tasks;
using Amazon.Lambda.Core;
using Amazon.S3;
```

```
using System;
using Amazon.Lambda.S3Events;
using System.Web;

// Assembly attribute to enable the Lambda function's JSON input to be
 converted into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJson

namespace S3Integration
{
    public class Function
    {
        private static AmazonS3Client _s3Client;
        public Function() : this(null)
        {
        }

        internal Function(AmazonS3Client s3Client)
        {
            _s3Client = s3Client ?? new AmazonS3Client();
        }

        public async Task<string> Handler(S3Event evt, ILambdaContext
 context)
        {
            try
            {
                if (evt.Records.Count <= 0)
                {
                    context.Logger.LogLine("Empty S3 Event received");
                    return string.Empty;
                }

                var bucket = evt.Records[0].S3.Bucket.Name;
                var key =
 HttpUtility.UrlDecode(evt.Records[0].S3.Object.Key);

                context.Logger.LogLine($"Request is for {bucket} and {key}");

                var objectResult = await _s3Client.GetObjectAsync(bucket,
 key);

                context.Logger.LogLine($"Returning {objectResult.Key}");
```

```
                return objectResult.Key;
            }
            catch (Exception e)
            {
                context.Logger.LogLine($"Error processing request -
 {e.Message}");

                return string.Empty;
            }
        }
    }
}
```

Go

SDK for Go V2

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 搭配 Lambda 來使用 S3 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "log"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/service/s3"
)

func handler(ctx context.Context, s3Event events.S3Event) error {
```

```
 sdkConfig, err := config.LoadDefaultConfig(ctx)
 if err != nil {
  log.Printf("failed to load default config: %s", err)
  return err
 }
 s3Client := s3.NewFromConfig(sdkConfig)

 for _, record := range s3Event.Records {
  bucket := record.S3.Bucket.Name
  key := record.S3.Object.URLDecodedKey
  headOutput, err := s3Client.HeadObject(ctx, &s3.HeadObjectInput{
   Bucket: &bucket,
   Key:    &key,
  })
  if err != nil {
   log.Printf("error getting head of object %s/%s: %s", bucket, key, err)
   return err
  }
  log.Printf("successfully retrieved %s/%s of type %s", bucket, key,
 *headOutput.ContentType)
 }

 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

適用於 Java 2.x 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Java 搭配 Lambda 來使用 S3 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import software.amazon.awssdk.services.s3.model.HeadObjectRequest;
import software.amazon.awssdk.services.s3.model.HeadObjectResponse;
import software.amazon.awssdk.services.s3.S3Client;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.S3Event;
import
 com.amazonaws.services.lambda.runtime.events.models.s3.S3EventNotification.S3EventNo

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Handler implements RequestHandler<S3Event, String> {
    private static final Logger logger =
 LoggerFactory.getLogger(Handler.class);
    @Override
    public String handleRequest(S3Event s3event, Context context) {
        try {
          S3EventNotificationRecord record = s3event.getRecords().get(0);
          String srcBucket = record.getS3().getBucket().getName();
          String srcKey = record.getS3().getObject().getUrlDecodedKey();

          S3Client s3Client = S3Client.builder().build();
          HeadObjectResponse headObject = getHeadObject(s3Client, srcBucket,
 srcKey);

          logger.info("Successfully retrieved " + srcBucket + "/" + srcKey +
 " of type " + headObject.contentType());

          return "Ok";
        } catch (Exception e) {
          throw new RuntimeException(e);
        }
    }

    private HeadObjectResponse getHeadObject(S3Client s3Client, String
 bucket, String key) {
        HeadObjectRequest headObjectRequest = HeadObjectRequest.builder()
```

```
                .bucket(bucket)
                .key(key)
                .build();
        return s3Client.headObject(headObjectRequest);
    }
}
```

JavaScript

適用於 JavaScript (v3) 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用使 Lambda JavaScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { S3Client, HeadObjectCommand } from "@aws-sdk/client-s3";

const client = new S3Client();

exports.handler = async (event, context) => {

    // Get the object from the event and show its content type
    const bucket = event.Records[0].s3.bucket.name;
    const key = decodeURIComponent(event.Records[0].s3.object.key.replace(/\
+/g, ' '));

    try {
        const { ContentType } = await client.send(new HeadObjectCommand({
            Bucket: bucket,
            Key: key,
        }));

        console.log('CONTENT TYPE:', ContentType);
        return ContentType;
```

```
    } catch (err) {
        console.log(err);
        const message = `Error getting object ${key} from bucket ${bucket}.
 Make sure they exist and your bucket is in the same region as this
 function.`;
        console.log(message);
        throw new Error(message);
    }
};
```

使用使 Lambda TypeScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { S3Event } from 'aws-lambda';
import { S3Client, HeadObjectCommand } from '@aws-sdk/client-s3';

const s3 = new S3Client({ region: process.env.AWS_REGION });

export const handler = async (event: S3Event): Promise<string | undefined> =>
 {
  // Get the object from the event and show its content type
  const bucket = event.Records[0].s3.bucket.name;
  const key = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/
g, ' '));
  const params = {
    Bucket: bucket,
    Key: key,
  };
  try {
    const { ContentType } = await s3.send(new HeadObjectCommand(params));
    console.log('CONTENT TYPE:', ContentType);
    return ContentType;
  } catch (err) {
    console.log(err);
    const message = `Error getting object ${key} from bucket ${bucket}. Make
 sure they exist and your bucket is in the same region as this function.`;
    console.log(message);
    throw new Error(message);
  }
};
```

PHP

適用於 PHP 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 PHP 使用 Lambda 使用 S3 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

use Bref\Context\Context;
use Bref\Event\S3\S3Event;
use Bref\Event\S3\S3Handler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';


class Handler extends S3Handler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    public function handleS3(S3Event $event, Context $context) : void
    {
        $this->logger->info("Processing S3 records");

        // Get the object from the event and show its content type
        $records = $event->getRecords();

        foreach ($records as $record)
        {
```

```
            $bucket = $record->getBucket()->getName();
            $key = urldecode($record->getObject()->getKey());

            try {
                $fileSize = urldecode($record->getObject()->getSize());
                echo "File Size: " . $fileSize . "\n";
                // TODO: Implement your custom processing logic here
            } catch (Exception $e) {
                echo $e->getMessage() . "\n";
                echo 'Error getting object ' . $key . ' from bucket ' .
 $bucket . '. Make sure they exist and your bucket is in the same region as
 this function.' . "\n";
                throw $e;
            }
        }
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

適用於 Python (Boto3) 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Python 搭配 Lambda 來使用 S3 事件。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
import json
import urllib.parse
import boto3

print('Loading function')
```

```
s3 = boto3.client('s3')


def lambda_handler(event, context):
    #print("Received event: " + json.dumps(event, indent=2))

    # Get the object from the event and show its content type
    bucket = event['Records'][0]['s3']['bucket']['name']
    key = urllib.parse.unquote_plus(event['Records'][0]['s3']['object']
['key'], encoding='utf-8')
    try:
        response = s3.get_object(Bucket=bucket, Key=key)
        print("CONTENT TYPE: " + response['ContentType'])
        return response['ContentType']
    except Exception as e:
        print(e)
        print('Error getting object {} from bucket {}. Make sure they
 exist and your bucket is in the same region as this function.'.format(key,
 bucket))
        raise e
```

Ruby

適用於 Ruby 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用紅寶石與 Lambda 一個 S3 事件。

```
require 'json'
require 'uri'
require 'aws-sdk'

puts 'Loading function'
```

```
def lambda_handler(event:, context:)
  s3 = Aws::S3::Client.new(region: 'region') # Your AWS region
  # puts "Received event: #{JSON.dump(event)}"

  # Get the object from the event and show its content type
  bucket = event['Records'][0]['s3']['bucket']['name']
  key = URI.decode_www_form_component(event['Records'][0]['s3']['object']
['key'], Encoding::UTF_8)
  begin
    response = s3.get_object(bucket: bucket, key: key)
    puts "CONTENT TYPE: #{response.content_type}"
    return response.content_type
  rescue StandardError => e
    puts e.message
    puts "Error getting object #{key} from bucket #{bucket}. Make sure they
 exist and your bucket is in the same region as this function."
    raise e
  end
end
```

Rust

適用於 Rust 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

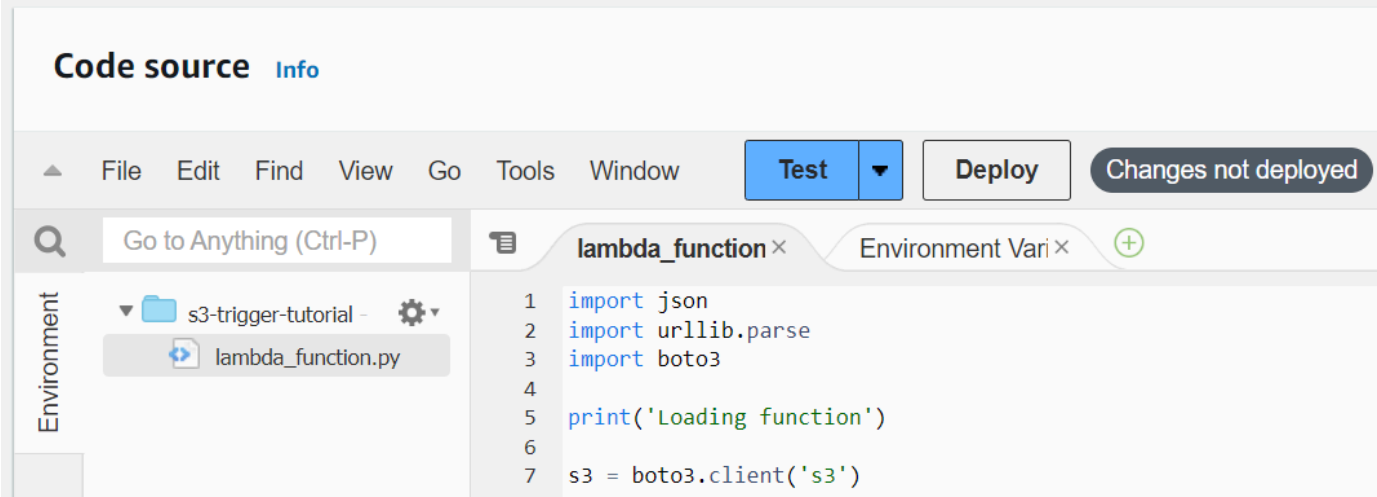
使用 Rust 搭配 Lambda 來使用 S3 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::s3::S3Event;
use aws_sdk_s3::{Client};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};


/// Main function
```

```
#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        .with_target(false)
        .without_time()
        .init();

    // Initialize the AWS SDK for Rust
    let config = aws_config::load_from_env().await;
    let s3_client = Client::new(&config);

    let res = run(service_fn(|request: LambdaEvent<S3Event>| {
        function_handler(&s3_client, request)
    })).await;

    res
}

async fn function_handler(
    s3_client: &Client,
    evt: LambdaEvent<S3Event>
) -> Result<(), Error> {
    tracing::info!(records = ?evt.payload.records.len(), "Received request
 from SQS");

    if evt.payload.records.len() == 0 {
        tracing::info!("Empty S3 event received");
    }

    let bucket =
 evt.payload.records[0].s3.bucket.name.as_ref().expect("Bucket name to
 exist");
    let key = evt.payload.records[0].s3.object.key.as_ref().expect("Object
 key to exist");

    tracing::info!("Request is for {} and object {}", bucket, key);

    let s3_get_object_result = s3_client
        .get_object()
        .bucket(bucket)
        .key(key)
        .send()
        .await;
```

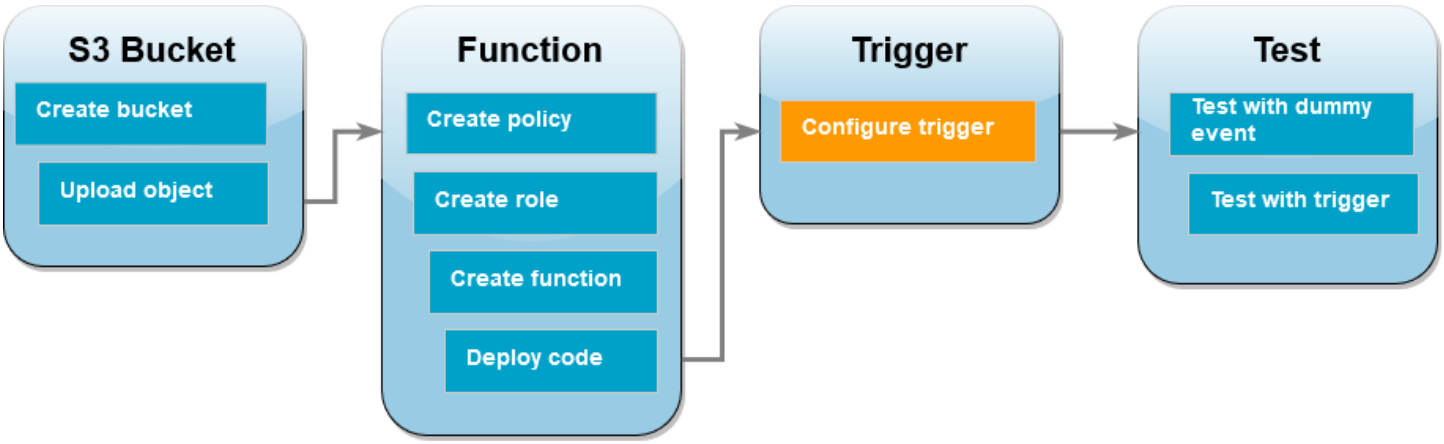
```
    match s3_get_object_result {
        Ok(_) => tracing::info!("S3 Get Object success, the s3GetObjectResult
 contains a 'body' property of type ByteStream"),
        Err(_) => tracing::info!("Failure with S3 Get Object request")
    }

    Ok(())
}
```

2. 在 Lambda 主控台的「程式碼原始碼」窗格中，將程式碼貼到 lambda_function.py 檔案中。



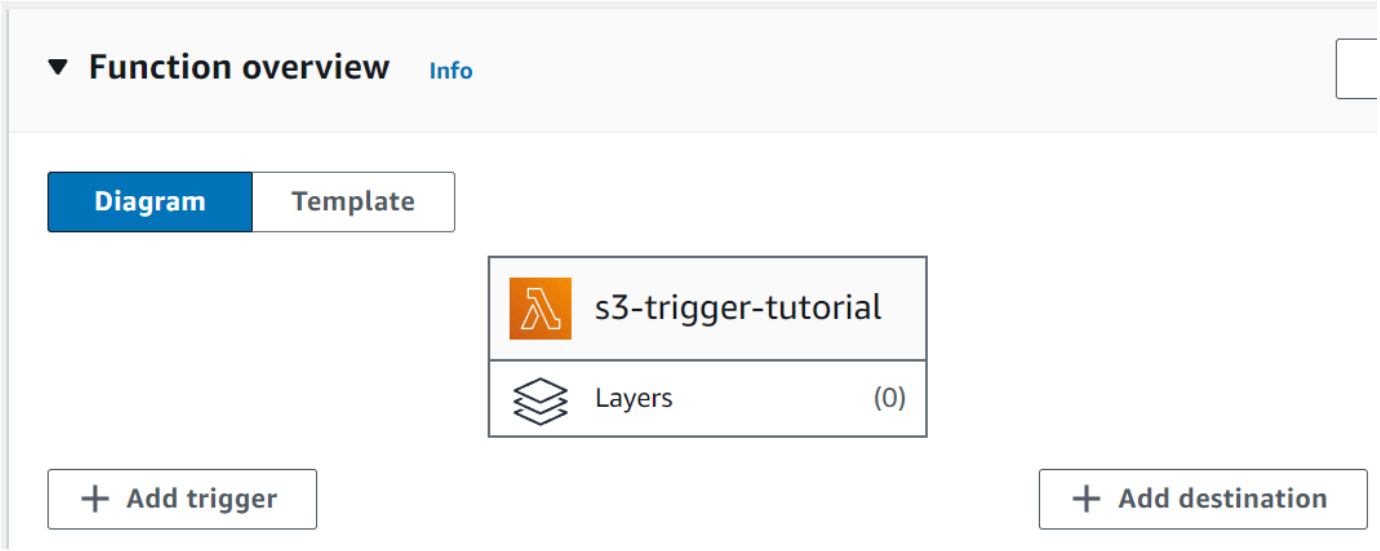
3. 選擇部署。

建立 Amazon S3 觸發條件



若要建立 Amazon S3 觸發條件

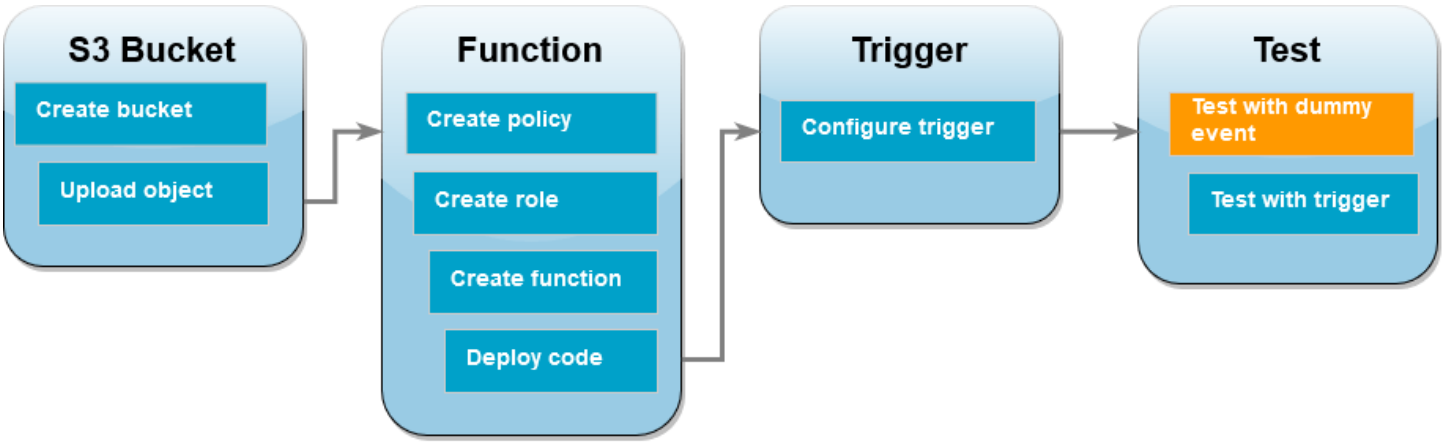
1. 在函數概觀窗格中，選擇新增觸發條件。



2. 選取 S3。
3. 在儲存貯體下，選取您在本教學課程中稍早建立的儲存貯體。
4. 在 [事件類型] 下，確定已選取 [所有物件建立事件]。
5. 在遞迴調用下，選取核取方塊，確認您了解不建議使用相同的 Amazon S3 儲存貯體進行輸入和輸出作業。
6. 選擇新增。

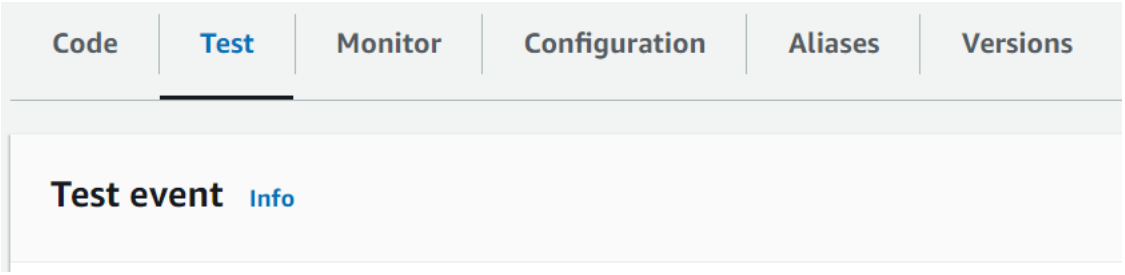
> **Note**
>
> 當您使用 Lambda 主控台為 Lambda 函數建立 Amazon S3 觸發器時，Amazon S3 會在您指定的儲存貯體上設定事件通知。在設定此事件通知之前，Amazon S3 會執行一系列檢查，以確認事件目標存在並具有必要的 IAM 政策。Amazon S3 也會對為該儲存貯體設定的任何其他事件通知執行這些測試。
>
> 由於這項檢查，如果儲存貯體先前已針對不再存在的資源設定事件目的地，或者針對沒有所需許可政策的資源設定事件目的地，Amazon S3 將無法建立新的事件通知。您會看到下列錯誤訊息，指出無法建立觸發器：
>
> ```
> An error occurred when creating the trigger: Unable to validate the following
>  destination configurations.
> ```
>
> 如果您先前使用相同儲存貯體為另一個 Lambda 函數設定了觸發器，且之後您已刪除該函數或修改了其權限政策，則可能會看到此錯誤。

使用虛擬事件來測試 Lambda 函數



使用虛擬事件來測試 Lambda 函數

1. 在函數的 Lambda 主控台頁面中，選擇 [測試] 索引標籤。



2. 事件名稱輸入 `MyTestEvent`。
3. 在事件 JSON 中，貼上下列測試事件。請務必取代這些值：

   - 將 `us-east-1` 取代為您用來建立 Amazon S3 儲存貯體的區域。
   - 將 `DOC-EXAMPLE-BUCKET` 的兩個執行個體取代為您的 Amazon S3 儲存貯體名稱。
   - 將 `test%2FKey` 取代為您之前上傳到儲存貯體的測試物件名稱 (例如 `HappyFace.jpg`)。

```
{
  "Records": [
    {
      "eventVersion": "2.0",
      "eventSource": "aws:s3",
      "awsRegion": "us-east-1",
      "eventTime": "1970-01-01T00:00:00.000Z",
      "eventName": "ObjectCreated:Put",
      "userIdentity": {
        "principalId": "EXAMPLE"
```

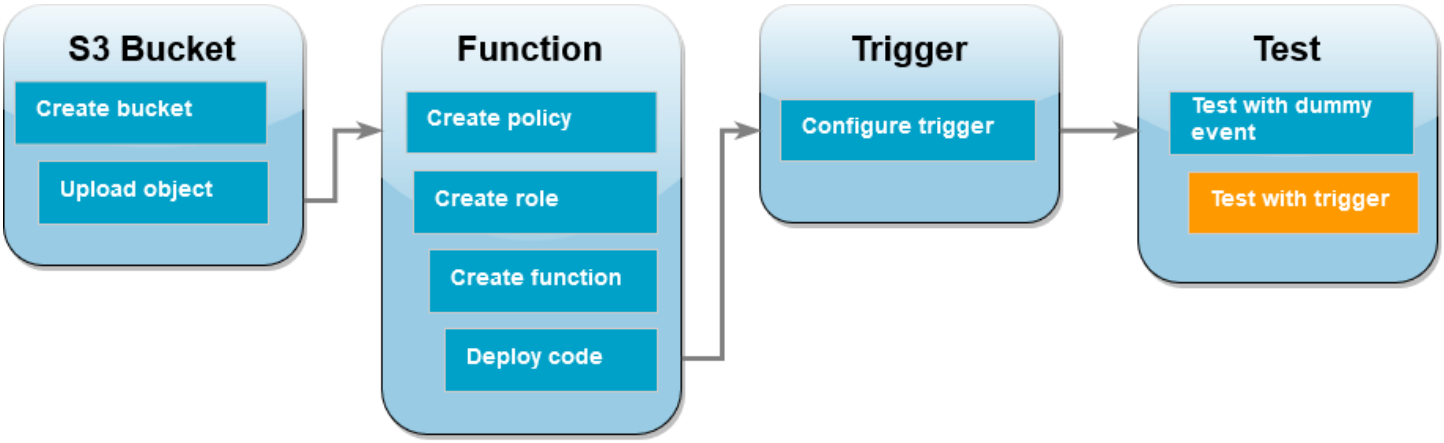
```
      },
      "requestParameters": {
        "sourceIPAddress": "127.0.0.1"
      },
      "responseElements": {
        "x-amz-request-id": "EXAMPLE123456789",
        "x-amz-id-2": "EXAMPLE123/5678abcdefghijklambdaisawesome/
mnopqrstuvwxyzABCDEFGH"
      },
      "s3": {
        "s3SchemaVersion": "1.0",
        "configurationId": "testConfigRule",
        "bucket": {
          "name": "DOC-EXAMPLE-BUCKET",
          "ownerIdentity": {
            "principalId": "EXAMPLE"
          },
          "arn": "arn:aws:s3:::DOC-EXAMPLE-BUCKET"
        },
        "object": {
          "key": "test%2Fkey",
          "size": 1024,
          "eTag": "0123456789abcdef0123456789abcdef",
          "sequencer": "0A1B2C3D4E5F678901"
        }
      }
    }
  ]
}
```

4. 選擇 Save (儲存)。
5. 選擇 測試。
6. 如果函數成功運作，您會在執行結果索引標籤中看到類似以下內容的輸出。

```
Response
"image/jpeg"

Function Logs
START RequestId: 12b3cae7-5f4e-415e-93e6-416b8f8b66e6 Version: $LATEST
2021-02-18T21:40:59.280Z    12b3cae7-5f4e-415e-93e6-416b8f8b66e6    INFO    INPUT
 BUCKET AND KEY:  { Bucket: 'DOC-EXAMPLE-BUCKET', Key: 'HappyFace.jpg' }
2021-02-18T21:41:00.215Z    12b3cae7-5f4e-415e-93e6-416b8f8b66e6    INFO    CONTENT
 TYPE: image/jpeg
```

```
END RequestId: 12b3cae7-5f4e-415e-93e6-416b8f8b66e6
REPORT RequestId: 12b3cae7-5f4e-415e-93e6-416b8f8b66e6    Duration: 976.25 ms
 Billed Duration: 977 ms    Memory Size: 128 MB    Max Memory Used: 90 MB    Init
 Duration: 430.47 ms

Request ID
12b3cae7-5f4e-415e-93e6-416b8f8b66e6
```

使用 Amazon S3 觸發條件測試 Lambda 函數



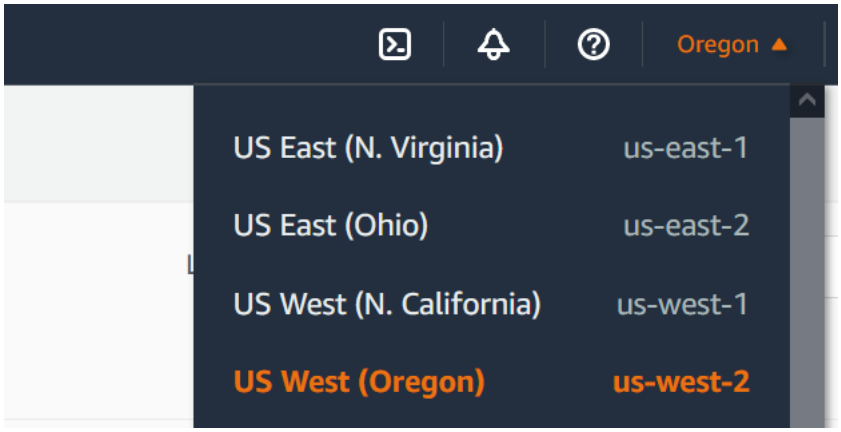
若要使用設定的觸發器測試您的功能，請使用主控台將物件上傳到 Amazon S3 儲存貯體。若要驗證 Lambda 函數是否如預期般執行，請使用 CloudWatch 記錄檢視函數的輸出。

將物件上傳至 Amazon S3 儲存貯體

1. 開啟 Amazon S3 主控台的「儲存貯體」頁面，然後選擇您先前建立的儲存貯體。
2. 選擇上傳。
3. 選擇新增檔案，然後使用檔案選擇器選擇您要上傳的物件。此物件可以是您自選的任何檔案。
4. 選擇開啟，然後選擇上傳。

若要使 CloudWatch 用記錄來驗證函數呼叫

1. 開啟 CloudWatch 主控台。
2. 請確定您使用的是建立 Lambda 函數的相同 AWS 區域 方式。您可使用螢幕頂端的下拉式清單來變更區域。



3. 選擇日誌，然後選擇日誌群組。
4. 為函數 (`/aws/lambda/s3-trigger-tutorial`) 選擇日誌群組名稱。
5. 在日誌串流下，選擇最新的日誌串流。
6. 如果您的函數是為了回應 Amazon S3 觸發器而正確叫用，您會看到類似以下內容的輸出。您看到的 `CONTENT TYPE` 取決於您上傳到儲存貯體的檔案類型。

```
2022-05-09T23:17:28.702Z 0cae7f5a-b0af-4c73-8563-a3430333cc10 INFO CONTENT
 TYPE: image/jpeg
```

清除您的資源

除非您想要保留為此教學課程建立的資源，否則您現在便可刪除。刪除您不再使用的 AWS 資源，您可以避免不必要的費用 AWS 帳戶。

若要刪除 Lambda 函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇您建立的函數。
3. 選擇 Actions (動作)、Delete (刪除)。
4. 在文字輸入欄位中輸入 **`delete`**，然後選擇 刪除 。

刪除執行角色

1. 開啟 IAM 主控台中的 角色頁面 。
2. 選取您建立的執行角色。
3. 選擇 刪除 。

4. 在文字輸入欄位中輸入角色的名稱，然後選擇 刪除 。

刪除 S3 儲存貯體

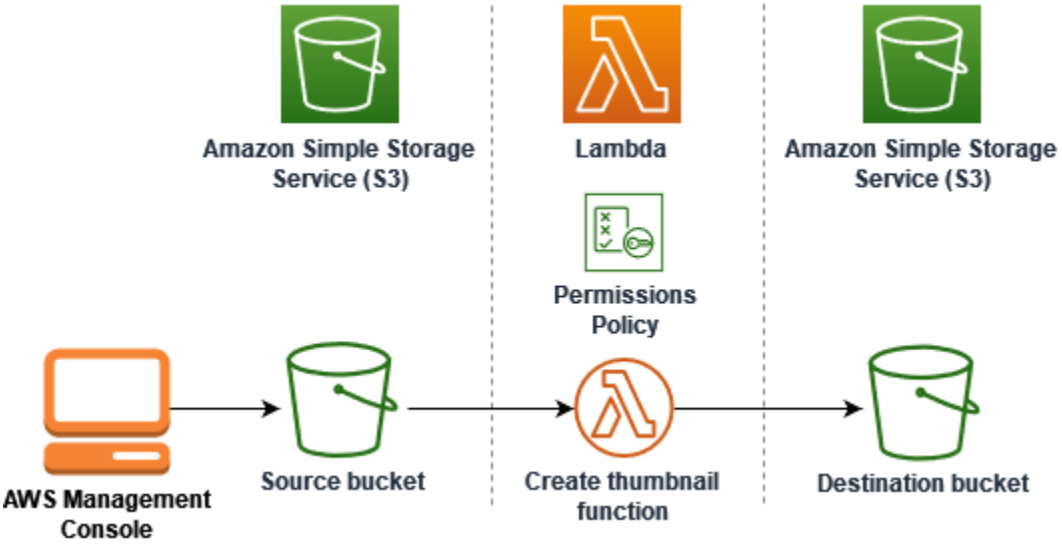
1. 開啟 Amazon S3 主控台。
2. 選擇您建立的儲存貯體。
3. 選擇 刪除 。
4. 在文字輸入欄位中輸入儲存貯體的名稱。
5. 選擇 刪除儲存貯體 。

後續步驟

在中教學課程：使用 Amazon S3 觸發條件建立縮圖影像，Amazon S3 觸發程序會叫用為上傳到儲存貯體的每個映像檔建立縮圖影像的函數。本教學課程需要中等程度 AWS 的 Lambda 網域知識。它示範如何使用 AWS Command Line Interface (AWS CLI) 建立資源，以及如何為函數及其相依性建立 .zip 檔案歸檔部署套件。

教學課程：使用 Amazon S3 觸發條件建立縮圖影像

在本教學課程中，您將建立和設定 Lambda 函數，它會調整新增至 Amazon Simple Storage Service (Amazon S3) 儲存貯體的映像。當您將映像檔案新增至儲存貯體時，Amazon S3 會調用 Lambda 函數。然後，函數會建立映像的縮圖版本，並將其輸出到不同 Amazon S3 儲存貯體。



請執行下列步驟以完成本教學課程：

1. 建立來源和目的地 Amazon S3 儲存貯體，並上傳範例映像。
2. 建立可調整映像大小並將縮圖輸出到 Amazon S3 儲存貯體的 Lambda 函數。
3. 設定一個 Lambda 觸發條件，在物件上傳至來源儲存貯體時調用函數。
4. 首先使用虛擬事件測試函數，然後透過將映像上傳到來源儲存貯體來測試函數。

完成這些步驟後，將了解如何使用 Lambda 在新增至 Amazon S3 儲存貯體的物件上執行檔案處理任務。您可以使用 AWS Command Line Interface (AWS CLI) 或完成本自學課程 AWS Management Console。

如果您正在尋找更簡單的範例來學習如何為 Lambda 設定 Amazon S3 觸發條件，則可以嘗試教學課程：使用 Amazon S3 觸發條件調用 Lambda 函數。

主題

必要條件

註冊一個 AWS 帳戶

如果您沒有 AWS 帳戶，請完成以下步驟來建立一個。

若要註冊成為 AWS 帳戶

1. 開啟 https://portal.aws.amazon.com/billing/signup。
2. 請遵循線上指示進行。

部分註冊程序需接收來電，並在電話鍵盤輸入驗證碼。

當您註冊一個時 AWS 帳戶，將創建AWS 帳戶根使用者一個。根使用者有權存取該帳戶中的所有 AWS 服務 和資源。安全性最佳做法是將管理存取權指派給使用者，並僅使用 root 使用者來執行需要 root 使用者存取權的工作。

AWS 註冊過程完成後，會向您發送確認電子郵件。您可以隨時登錄 https://aws.amazon.com/ 並選擇我的帳戶，以檢視您目前的帳戶活動並管理帳戶。

建立具有管理權限的使用者

註冊後，請保護您的 AWS 帳戶 AWS 帳戶根使用者 AWS IAM Identity Center、啟用和建立系統管理使用者，這樣您就不會將 root 使用者用於日常工作。

保護您的 AWS 帳戶根使用者

1. 選擇 Root 使用者並輸入您的 AWS 帳戶 電子郵件地址，以帳戶擁有者身分登入。AWS Management Console在下一頁中，輸入您的密碼。

   如需使用根使用者登入的說明，請參閱 AWS 登入 使用者指南中的以根使用者身分登入。
2. 若要在您的根使用者帳戶上啟用多重要素驗證 (MFA)。

   如需指示，請參閱《IAM 使用者指南》中的為 AWS 帳戶 根使用者啟用虛擬 MFA 裝置 (主控台)。

建立具有管理權限的使用者

1. 啟用 IAM Identity Center。

   如需指示，請參閱 AWS IAM Identity Center 使用者指南中的啟用 AWS IAM Identity Center。
2. 在 IAM 身分中心中，將管理存取權授予使用者。

   若要取得有關使用 IAM Identity Center 目錄 做為身分識別來源的自學課程，請參閱《使用指南》IAM Identity Center 目錄中的「以預設值設定使用AWS IAM Identity Center 者存取」。

以具有管理權限的使用者身分登入

- 若要使用您的 IAM Identity Center 使用者簽署，請使用建立 IAM Identity Center 使用者時傳送至您電子郵件地址的簽署 URL。

如需使用 IAM 身分中心使用者登入的說明，請參閱使用AWS 登入 者指南中的登入 AWS 存取入口網站。

指派存取權給其他使用者

1. 在 IAM 身分中心中，建立遵循套用最低權限許可的最佳做法的權限集。

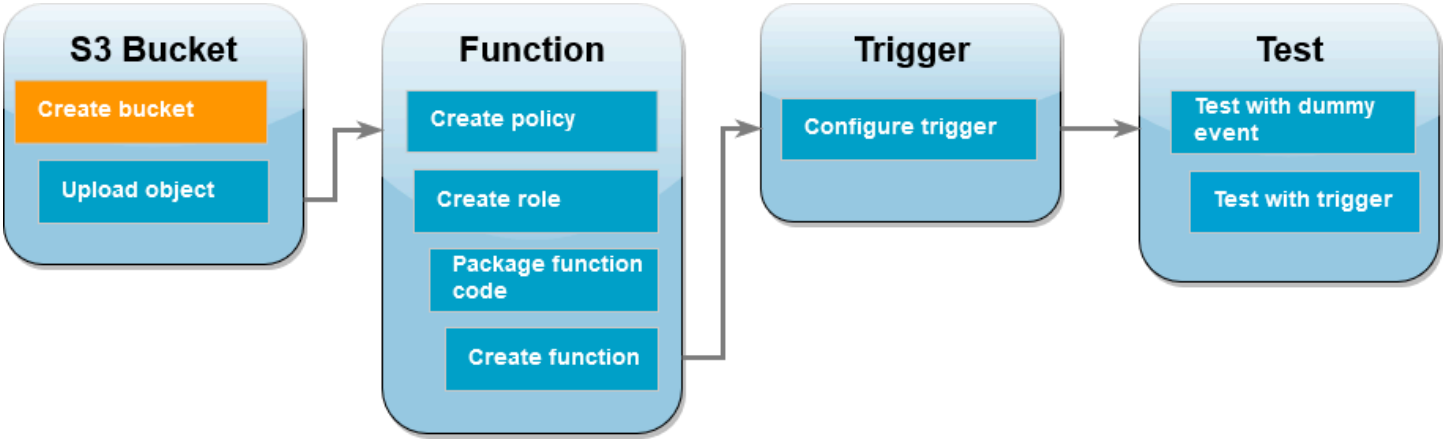
   如需指示，請參閱《AWS IAM Identity Center 使用指南》中的「建立權限集」。
2. 將使用者指派給群組，然後將單一登入存取權指派給群組。

   如需指示，請參閱《AWS IAM Identity Center 使用指南》中的「新增群組」。

如果您想要使用 AWS CLI 來完成自學課程，請安裝最新版本的 AWS Command Line Interface.

對於 Lambda 函數程式碼，您可以使用 Python 或 Node.js。為您想要使用的語言安裝語言支援工具和套件管理工具。

建立兩個 Amazon S3 儲存貯體



首先建立兩個 Amazon S3 儲存貯體。第一個儲存貯體是將接收映像上傳的來源儲存貯體。當您調用函數時，Lambda 會使用第二個儲存貯體來儲存已調整大小的縮圖。

AWS Management Console

建立 Amazon S3 儲存貯體 (主控台)

1. 開啟 Amazon S3 主控台的儲存貯體頁面。
2. 選擇 建立儲存貯體 。

3. 在 General configuration (一般組態) 下，執行下列動作：

   a. 請在儲存貯體名稱輸入符合 Amazon S3 儲存貯體命名規則的全域唯一名稱。儲存貯體名稱只能包含小寫字母、數字、句點 (.) 和連字號 (-)。

   b. 對於 AWS 區域，請選擇最接近您地理位置的 AWS 區域。稍後在教學課程中，您必須建立 Lambda 函數 AWS 區域，因此請記下您選擇的區域。

4. 其他所有選項維持設為預設值，然後選擇建立儲存貯體。

5. 重複步驟 1 到 4 以建立目的地儲存貯體。對於儲存貯體名稱，輸入 **DOC-EXAMPLE-SOURCE-BUCKET-resized**，其中 **DOC-EXAMPLE-SOURCE-BUCKET** 是您剛才建立的來源儲存貯體名稱。

AWS CLI

建立 Amazon S3 儲存貯體 (AWS CLI)

1. 執行下列 CLI 命令以建立來源儲存貯體。您為儲存貯體選擇的名稱必須是全域唯一的，並遵循 Amazon S3 儲存貯體命名規則。名稱只能包含小寫字母、數字、句點 (.) 和連字號 (-)。對於 `region` 和 `LocationConstraint`，請選擇最接近您地理位置的 AWS 區域。

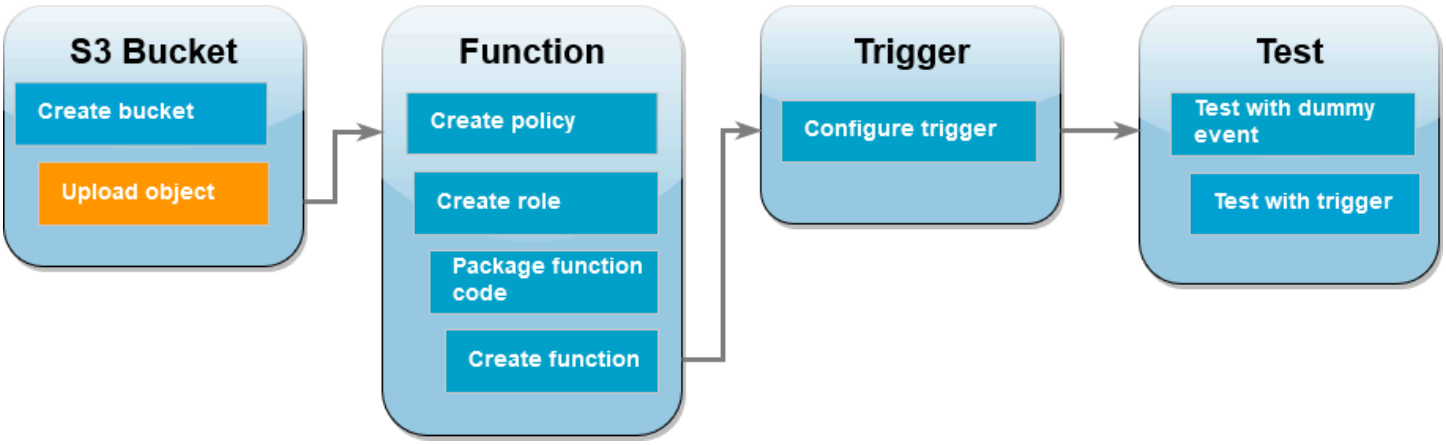
   ```
   aws s3api create-bucket --bucket DOC-EXAMPLE-SOURCE-BUCKET --region us-west-2 \
   --create-bucket-configuration LocationConstraint=us-west-2
   ```

   在教學課程稍後，您必須在與來源儲存貯體 AWS 區域 相同的位置建立 Lambda 函數，因此請記下您選擇的區域。

2. 執行下列命令以建立目的地儲存貯體。對於儲存貯體名稱，必須使用 **DOC-EXAMPLE-SOURCE-BUCKET-resized**，其中 **DOC-EXAMPLE-SOURCE-BUCKET** 是您在步驟 1 中建立的來源儲存貯體名稱。對於`region`和`LocationConstraint`，請選擇 AWS 區域 您用來建立來源值區的相同項目。

   ```
   aws s3api create-bucket --bucket DOC-EXAMPLE-SOURCE-BUCKET-resized --region us-
   west-2 \
   --create-bucket-configuration LocationConstraint=us-west-2
   ```

將測試映像上傳到來源儲存貯體



稍後在教學課程中，您將使用 AWS CLI 或 Lambda 主控台叫用 Lambda 函數來測試它。若要確認函數運作正常，來源儲存貯體需要包含測試映像。此映像可以是您選擇的任何 JPG 或 PNG 檔案。

AWS Management Console

將測試映像上傳到來源儲存貯體 (主控台)

1. 開啟 Amazon S3 主控台的儲存貯體頁面。
2. 選取您在上一個步驟所建立的來源儲存貯體。
3. 選擇上傳。
4. 選擇新增檔案，然後使用檔案選擇器選擇您要上傳的物件。
5. 選擇開啟，然後選擇上傳。

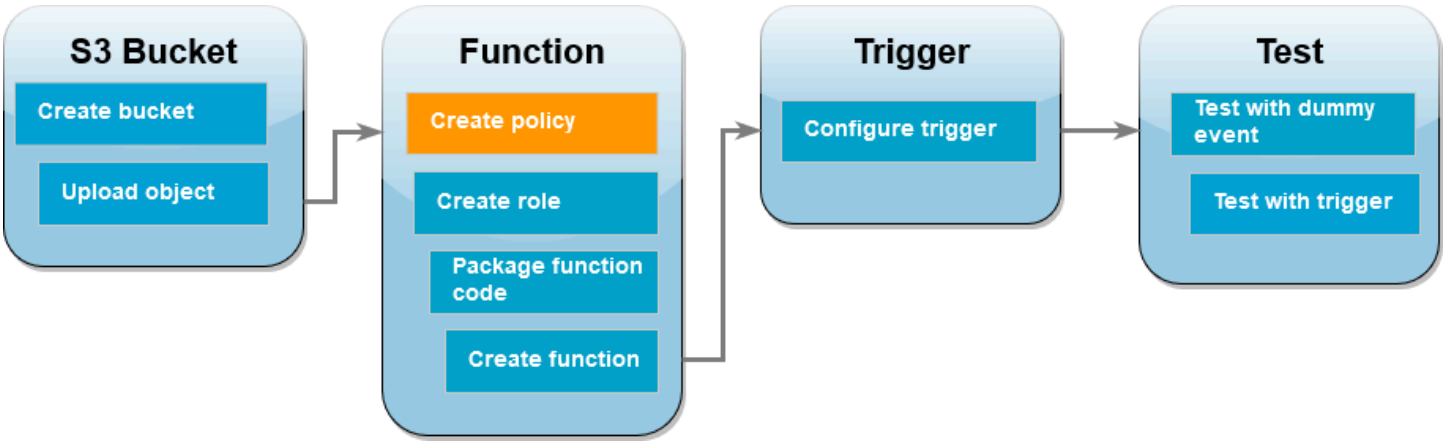
AWS CLI

將測試映像上傳到來源儲存貯體 (AWS CLI)

- 從包含要上傳之映像的目錄中，執行下列 CLI 命令。將 `--bucket` 參數取代為來源儲存貯體名稱。對於 `--key` 和 `--body` 參數，請使用測試映像的檔案名稱。

```
aws s3api put-object --bucket DOC-EXAMPLE-SOURCE-BUCKET --key HappyFace.jpg --
body ./HappyFace.jpg
```

建立許可政策



建立 Lambda 函數的第一個步驟是建立許可政策。此原則為您的函數提供存取其他 AWS 資源所需的權限。在本教學中，該政策為 Amazon S3 儲存貯體提供 Lambda 讀取和寫入許可，並允許其寫入 Amazon CloudWatch 日誌。

AWS Management Console

建立政策 (主控台)

1. 開啟 AWS Identity and Access Management (IAM) 主控台的「政策」頁面。
2. 選擇建立政策。
3. 選擇 JSON 索引標籤，然後將下列政策貼到 JSON 編輯器。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:PutLogEvents",
                "logs:CreateLogGroup",
                "logs:CreateLogStream"
            ],
            "Resource": "arn:aws:logs:*:*:*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject"
            ],
```

```
            "Resource": "arn:aws:s3:::*/*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:PutObject"
            ],
            "Resource": "arn:aws:s3:::*/*"
        }
    ]
}
```

4. 選擇下一步。
5. 在政策詳細資訊下，針對政策政策名稱 ，輸入 **LambdaS3Policy**。
6. 選擇建立政策。

AWS CLI

建立政策 (AWS CLI)

1. 將下面的 JSON 儲存在名為 policy.json 的檔案中。

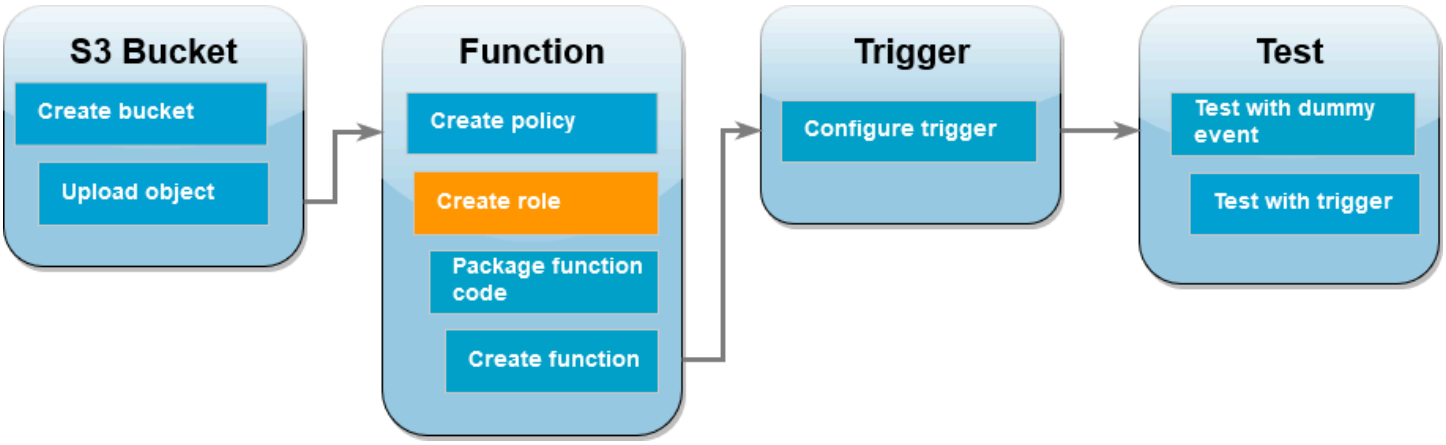
```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:PutLogEvents",
                "logs:CreateLogGroup",
                "logs:CreateLogStream"
            ],
            "Resource": "arn:aws:logs:*:*:*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject"
            ],
            "Resource": "arn:aws:s3:::*/*"
        },
        {
```

```
            "Effect": "Allow",
            "Action": [
                "s3:PutObject"
            ],
            "Resource": "arn:aws:s3:::*/*"
        }
    ]
}
```

2. 在儲存 JSON 政策文件的目錄中執行下列 CLI 命令。

```
aws iam create-policy --policy-name LambdaS3Policy --policy-document file://
policy.json
```

建立執行角色



執行角色是一種 IAM 角色，可授與 Lambda 函數存取 AWS 服務 和資源的權限。若要為函數提供 Amazon S3 儲存貯體的讀取和寫入存取權，請連接您在上個步驟建立的許可政策。

AWS Management Console

建立執行角色並連接許可政策 (主控台)

1. 開啟 (IAM) 主控台的角色頁面。
2. 選擇建立角色。
3. 對於可信實體類型，選取 AWS 服務，然後針對使用案例選取 Lambda。
4. 選擇下一步。
5. 執行下列動作，新增您在上個步驟中建立的許可政策：

a. 在政策搜尋方塊中，輸入 **LambdaS3Policy**。

b. 在搜尋結果中，選取 LambdaS3Policy 的核取方塊。

c. 選擇下一步。

6. 在角色詳細資訊下，針對角色名稱，輸入 **LambdaS3Role**。

7. 選擇建立角色。

AWS CLI

建立執行角色並連接許可政策 (AWS CLI)

1. 將下面的 JSON 儲存在名為 trust-policy.json 的檔案中。此信任原則允許 Lambda 使用角色的權限，方法是lambda.amazonaws.com授予服務主體呼叫 AWS Security Token Service (AWS STS) AssumeRole 動作的權限。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

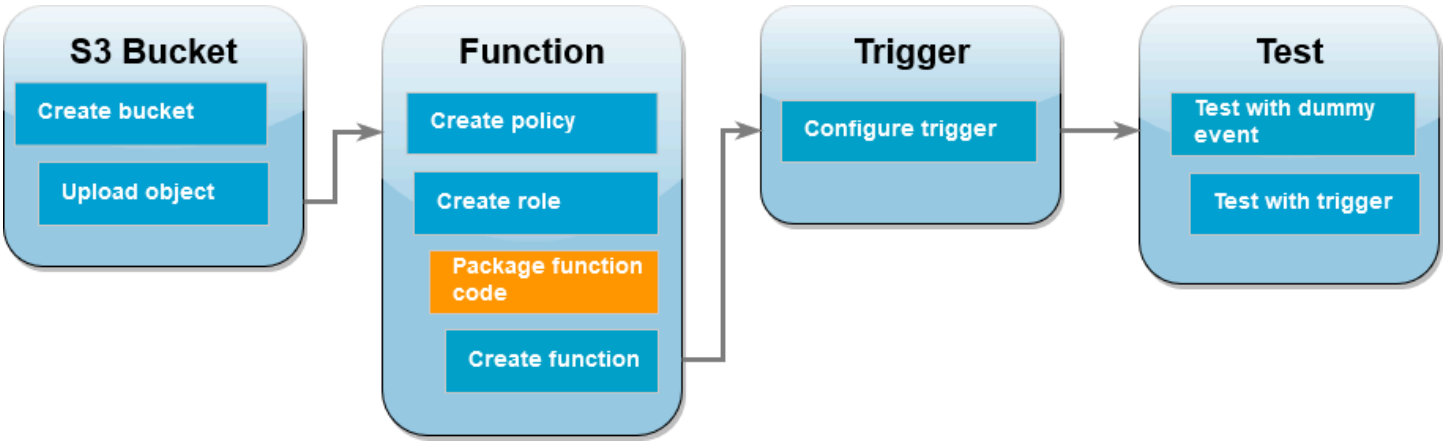
2. 在儲存 JSON 信任政策文件的目錄中，執行下列 CLI 命令，建立執行角色。

```
aws iam create-role --role-name LambdaS3Role --assume-role-policy-document
 file://trust-policy.json
```

3. 執行下列 CLI 命令，連接您在上個步驟中建立的許可政策。將保單 ARN 中的 AWS 帳戶 號碼取代為您自己的帳號。

```
aws iam attach-role-policy --role-name LambdaS3Role --policy-arn
 arn:aws:iam::123456789012:policy/LambdaS3Policy
```

建立函數部署套件



要建立函數，需建立包含函數程式碼和其相依項的部署套件。對於此 `CreateThumbnail` 函數，函數程式碼使用單獨的程式庫來調整映像大小。依照所選語言的指示，建立包含所需程式庫的部署套件。

Node.js

建立部署套件 (Node.js)

1. 為函數程式碼和相依項建立名為 `lambda-s3` 的目錄並導覽到該目錄。

```
mkdir lambda-s3
cd lambda-s3
```

2. 將以下函數程式碼儲存在名為 `index.mjs` 檔案中。請務必使`'us-west-2'`用您建立自己的來源和目的地值區的 AWS 區域 位置來取代。

```
// dependencies
import { S3Client, GetObjectCommand, PutObjectCommand } from '@aws-sdk/client-
s3';

import { Readable } from 'stream';

import sharp from 'sharp';
import util from 'util';


// create S3 client
const s3 = new S3Client({region: 'us-west-2'});

// define the handler function
```

```
export const handler = async (event, context) => {

// Read options from the event parameter and get the source bucket
console.log("Reading options from event:\n", util.inspect(event, {depth: 5}));
  const srcBucket = event.Records[0].s3.bucket.name;

// Object key may have spaces or unicode non-ASCII characters
const srcKey    = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/
g, " "));
const dstBucket = srcBucket + "-resized";
const dstKey    = "resized-" + srcKey;

// Infer the image type from the file suffix
const typeMatch = srcKey.match(/\.([^.]*)$/);
if (!typeMatch) {
  console.log("Could not determine the image type.");
  return;
}

// Check that the image type is supported
const imageType = typeMatch[1].toLowerCase();
if (imageType != "jpg" && imageType != "png") {
  console.log(`Unsupported image type: ${imageType}`);
  return;
}

// Get the image from the source bucket. GetObjectCommand returns a stream.
try {
  const params = {
    Bucket: srcBucket,
    Key: srcKey
  };
  var response = await s3.send(new GetObjectCommand(params));
  var stream = response.Body;

// Convert stream to buffer to pass to sharp resize function.
  if (stream instanceof Readable) {
    var content_buffer = Buffer.concat(await stream.toArray());

  } else {
    throw new Error('Unknown object stream type');
  }
```

```
} catch (error) {
  console.log(error);
  return;
}


// set thumbnail width. Resize will set the height automatically to maintain
 aspect ratio.
const width  = 200;

// Use the sharp module to resize the image and save in a buffer.
try {
  var output_buffer = await sharp(content_buffer).resize(width).toBuffer();

} catch (error) {
  console.log(error);
  return;
}

// Upload the thumbnail image to the destination bucket
try {
  const destparams = {
    Bucket: dstBucket,
    Key: dstKey,
    Body: output_buffer,
    ContentType: "image"
  };

  const putResult = await s3.send(new PutObjectCommand(destparams));

  } catch (error) {
    console.log(error);
    return;
  }

  console.log('Successfully resized ' + srcBucket + '/' + srcKey +
    ' and uploaded to ' + dstBucket + '/' + dstKey);
  };
```

3. 在 `lambda-s3` 目錄中，使用 npm 安裝 Sharp 程式庫。請注意，最新的 Sharp 版本 (0.33) 與 Lambda 不相容。安裝版本 0.32.6 以完成本教學課程。

```
npm install sharp@0.32.6
```

npm install 命令為您的模組建立一個 node_modules 目錄。在此步驟之後，目錄結構應如下所示。

```
lambda-s3
|- index.mjs
|- node_modules
|  |- base64js
|  |- bl
|  |- buffer
...
|- package-lock.json
|- package.json
```

4. 建立包含函數程式碼及其相依項 .zip 部署套件。在 MacOS 或 Linux 中，執行下列命令。

```
zip -r function.zip .
```

在 Windows 中，使用您偏好的 zip 公用程式建立 .zip 檔案。請確保 index.mjs、package.json 和 package-lock.json 檔案以及 node_modules 目錄全部都位於 .zip 檔案的根目錄。

Python

建立部署套件 (Python)

1. 將程式碼範例儲存為名為 lambda_function.py 的檔案。

```
import boto3
import os
import sys
import uuid
from urllib.parse import unquote_plus
from PIL import Image
import PIL.Image

s3_client = boto3.client('s3')

def resize_image(image_path, resized_path):
  with Image.open(image_path) as image:
    image.thumbnail(tuple(x / 2 for x in image.size))
```

```
    image.save(resized_path)

def lambda_handler(event, context):
  for record in event['Records']:
    bucket = record['s3']['bucket']['name']
    key = unquote_plus(record['s3']['object']['key'])
    tmpkey = key.replace('/', '')
    download_path = '/tmp/{}{}'.format(uuid.uuid4(), tmpkey)
    upload_path = '/tmp/resized-{}'.format(tmpkey)
    s3_client.download_file(bucket, key, download_path)
    resize_image(download_path, upload_path)
    s3_client.upload_file(upload_path, '{}-resized'.format(bucket), 'resized-
{}'.format(key))
```

2. 在建立 `lambda_function.py` 檔案的同一個目錄中，建立名為 `package` 的新目錄並安裝 Pillow (PIL) 程式庫和 AWS SDK for Python (Boto3)。雖然 Lambda Python 執行期包含 Boto3 SDK 的版本，但建議您將函數的所有相依項新增至部署套件，即使其包含在執行期中。如需詳細資訊，請參閱 Python 中的執行期相依項。

```
mkdir package
pip install \
--platform manylinux2014_x86_64 \
--target=package \
--implementation cp \
--python-version 3.9 \
--only-binary=:all: --upgrade \
pillow boto3
```

Pillow 程式庫中包含 C/C ++ 程式碼。使用 `--platform manylinux_2014_x86_64` 和 `--only-binary=:all:` 選項，pip 便會下載並安裝適用的 Pillow 版本，其中包含與 Amazon Linux 2 作業系統相容的預先編譯二進位檔。這可確保無論本機建置機器的作業系統和架構為何，您的部署套件都能在 Lambda 執行環境中運作。

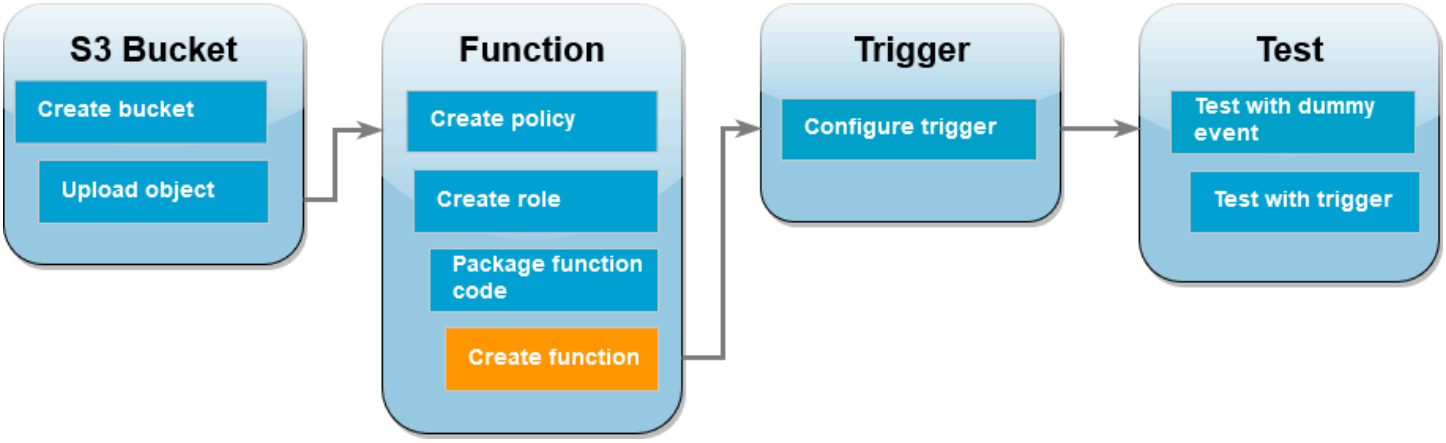
3. 建立一個包含應用程式碼以及 Pillow 和 Boto3 程式庫的 .zip 檔案。在 Linux 或 MacOS 中，使用命令列界面執行下列命令。

```
cd package
zip -r ../lambda_function.zip .
cd ..
zip lambda_function.zip lambda_function.py
```

在 Windows 中，使用您偏好的 zip 工具建立 `lambda_function.zip` 檔案。確保包含相依項的 `lambda_function.py` 檔案和資料夾全部都在 .zip 檔案的根目錄中。

您也可以使用 Python 虛擬環境建立部署套件。請參閱使用 .zip 封存檔部署 Python Lambda 函數

建立 Lambda 函式



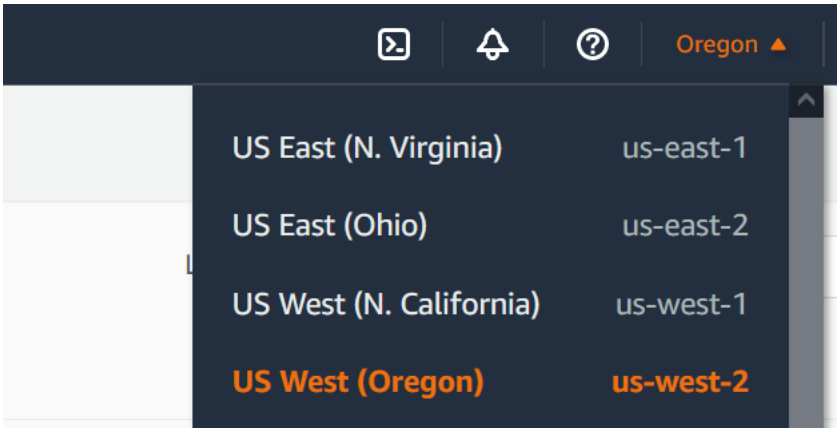
您可以使用 AWS CLI 或 Lambda 主控台建立 Lambda 函數。依照所選語言的指示建立函數。

AWS Management Console

建立函數的方式 (主控台)

要使用主控台建立 Lambda 函數，首先建立包含一些 'Hello world' 程式碼的基本函數。然後，透過上傳您在上一個步驟中建立的 .zip 或 JAR 檔案，將此程式碼取代為您自己的函數程式碼。

1. 開啟 Lambda 主控台中的函數頁面。
2. 請確定您在建立 Amazon S3 儲存貯體的相同 AWS 區域 內容中工作。可使用螢幕頂端的下拉式清單來變更區域。



3. 選擇建立函數。
4. 選擇 Author from scratch (從頭開始撰寫)。
5. 在基本資訊下，請執行下列動作：
   a. 針對 函數名稱 ，請輸入 **`CreateThumbnail`**。
   b. 對於執行期，請根據您為函數選擇的語言，選擇 Node.js 18.x 或 Python 3.9。
   c. 對於 Architecture (架構)，選擇 x86_64。
6. 在變更預設執行角色索引標籤中，執行下列操作：
   a. 展開索引標籤，然後選擇使用現有角色。
   b. 選擇您之前建立的 `LambdaS3Role`。
7. 選擇建立函數。

上傳函數程式碼 (主控台)

1. 在程式碼來源窗格中選擇上傳來源。
2. 選擇 .zip 檔案。
3. 選擇上傳。
4. 在檔案選擇器中，選取 .zip 檔案，並選擇開啟。
5. 選擇儲存。

AWS CLI

建立函數 (AWS CLI)

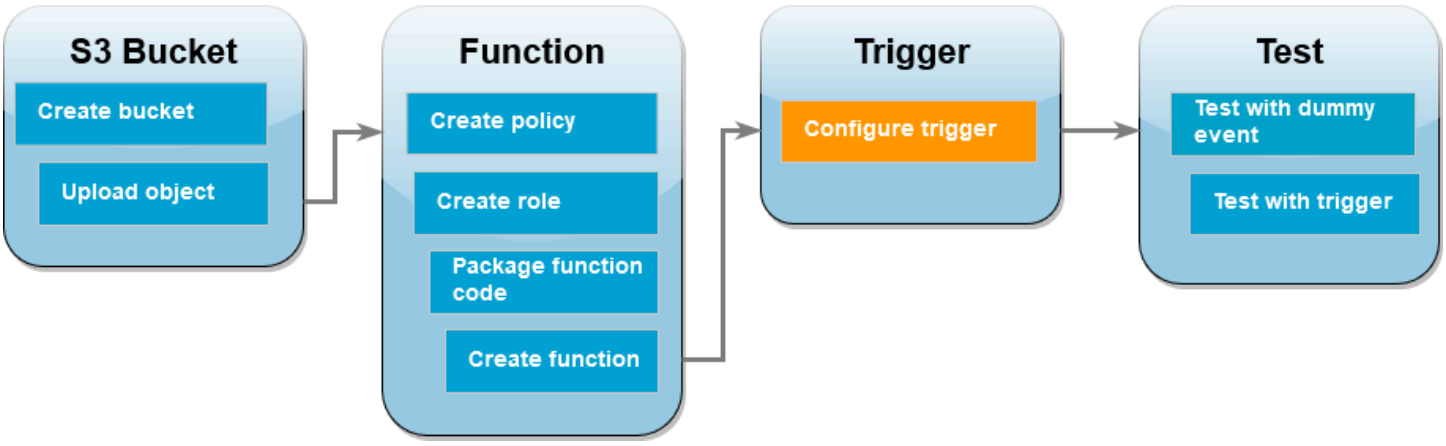
- 執行所選語言的 CLI 命令。對於`role`參數，請務必以您自己`123456789012`的 AWS 帳戶 ID 取代。對於 `region` 參數，將 `us-west-2` 取代為您建立 Amazon S3 儲存貯體所在的區域。
  - 對於 Node.js，請從包含 `function.zip` 檔案的目錄中執行下列命令。

```
aws lambda create-function --function-name CreateThumbnail \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs18.x \
--timeout 10 --memory-size 1024 \
--role arn:aws:iam::123456789012:role/LambdaS3Role --region us-west-2
```

  - 對於 Python，請從包含 `lambda_function.zip` 檔案的目錄中執行下列命令。

```
aws lambda create-function --function-name CreateThumbnail \
--zip-file fileb://lambda_function.zip --handler
 lambda_function.lambda_handler \
--runtime python3.9 --timeout 10 --memory-size 1024 \
--role arn:aws:iam::123456789012:role/LambdaS3Role --region us-west-2
```

設定 Amazon S3 以調用函數



若要在將映像上傳至來源儲存貯體時執行 Lambda 函數，您需要設定函數的觸發條件。可以使用主控台或 AWS CLI來設定 Amazon S3 觸發條件。

> **Important**
>
> 此程序會將 Amazon S3 儲存貯體設定為每次在儲存貯體中建立物件時即會調用您的函數。請務必僅在來源儲存貯體上進行設定。如果 Lambda 函數在進行調用的同一個儲存貯體中建立物件，則可以在一個迴圈中連續調用函數。這可能會導致未預期的費用會向您收取 AWS 帳戶.

AWS Management Console

設定 Amazon S3 觸發條件 (主控台)

1. 開啟 Lambda 主控台的函數頁面，然後選擇您的函數 (`CreateThumbnail`)。
2. 選擇 Add trigger (新增觸發條件)。
3. 選取 S3。
4. 在儲存貯體下，選取您的來源儲存貯體。
5. 在事件類型下，選取所有物件建立事件。
6. 在遞迴調用下，選取核取方塊，確認您了解不建議使用相同的 Amazon S3 儲存貯體進行輸入和輸出作業。您可以閱讀無伺服器園地中 導致 Lambda 函數失控的遞迴模式，進一步了解 Lambda 中的遞迴調用模式。
7. 選擇新增。

   當您使用 Lambda 主控台建立觸發條件時，Lambda 會自動建立資源型政策，為您選取的服務授予調用函數的許可。

AWS CLI

設定 Amazon S3 觸發條件 (AWS CLI)

1. 若要讓 Amazon S3 來源儲存貯體在新增映像檔案時調用函數，您首先需要使用資源型政策為函數設定許可。以資源為基礎的政策聲明提供了調用函數的其他 AWS 服務 權限。若要授予 Amazon S3 調用函數的許可，請執行下列 CLI 命令。請務必使用您自己的 AWS 帳戶 ID 取代`source-account`參數，並使用您自己的來源值區名稱。

   ```
   aws lambda add-permission --function-name CreateThumbnail \
   --principal s3.amazonaws.com --statement-id s3invoke --action
    "lambda:InvokeFunction" \
   --source-arn arn:aws:s3:::DOC-EXAMPLE-SOURCE-BUCKET \
   ```

```
--source-account 123456789012
```

使用此命令定義的政策允許 Amazon S3 僅在來源儲存貯體上發生動作時調用函數。

> **Note**
>
> 雖然 Amazon S3 儲存貯體名稱全域唯一，但是在使用資源型政策時，最佳實務是指定儲存貯體必須屬於您的帳戶。這是因為如果您刪除存儲桶，則另一個 AWS 帳戶 存儲桶可能使用相同的 Amazon 資源名稱（ARN）創建存儲桶。

2. 將下面的 JSON 儲存在名為 `notification.json` 的檔案中。套用至來源儲存貯體時，此 JSON 會設定儲存貯體，以便在每次新增新物件時傳送通知至 Lambda 函數。將 Lambda 函 AWS 帳戶 數 ARN AWS 區域 中的號碼取代為您自己的帳戶號碼和區域。

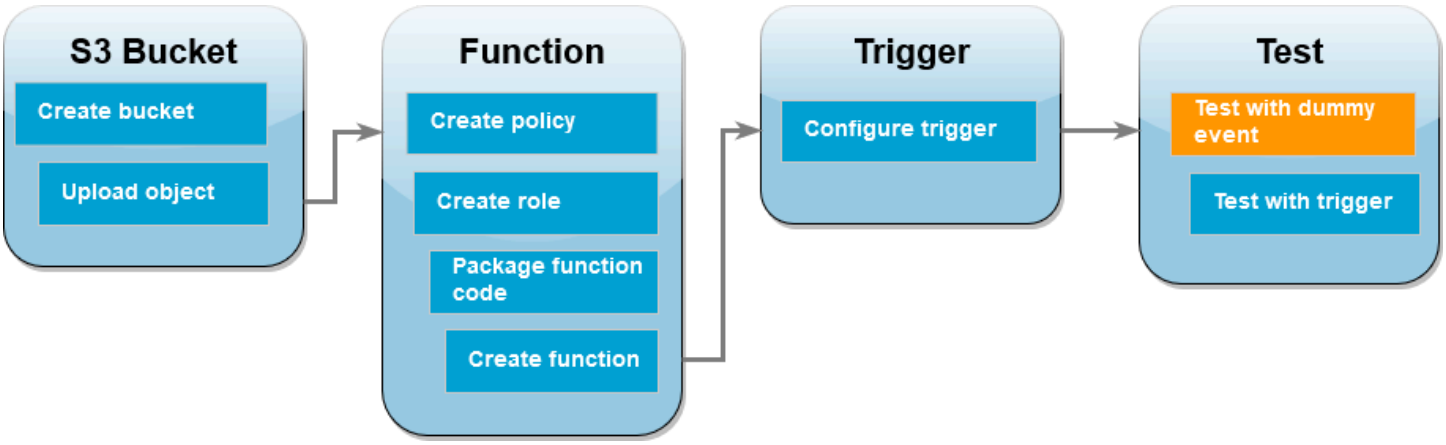
```
{
"LambdaFunctionConfigurations": [
    {
      "Id": "CreateThumbnailEventConfiguration",
      "LambdaFunctionArn": "arn:aws:lambda:us-
west-2:123456789012:function:CreateThumbnail",
      "Events": [ "s3:ObjectCreated:Put" ]
    }
  ]
}
```

3. 執行下列 CLI 命令，將您建立的 JSON 檔案中的通知設定套用至來源儲存貯體。用您自己的來源儲存貯體名稱取代 `DOC-EXAMPLE-SOURCE-BUCKET`。

```
aws s3api put-bucket-notification-configuration --bucket DOC-EXAMPLE-SOURCE-
BUCKET \
--notification-configuration file://notification.json
```

若要進一步了解`put-bucket-notification-configuration`命令和選`notification-configuration`項，請參閱 AWS CLI 命令參考put-bucket-notification-configuration中的。

使用虛擬事件來測試 Lambda 函數



在將映像檔案新增至 Amazon S3 來源儲存貯體以測試整個設定之前，可以透過使用虛擬事件調用 Lambda 函數來測試其是否正常運作。Lambda 中的事件是一種 JSON 格式的文件，它包含供函數處理的資料。Amazon S3 調用函數時，傳送至函數的事件會包含諸如儲存貯體名稱、儲存貯體 ARN 和物件金鑰等資訊。

AWS Management Console

使用虛擬事件來測試 Lambda 函數 (主控台)

1. 開啟 Lambda 主控台的函數頁面，然後選擇您的函數 (`CreateThumbnail`)。
2. 選擇測試標籤。
3. 若要建立測試事件，請在測試事件窗格中執行下列動作：
   a. 在測試事件動作下方，選取建立新事件。
   b. 事件名稱輸入 **`myTestEvent`**。
   c. 對於範本，選取 S3 Put。
   d. 將下列參數的值取代為您自己的值。
      - 對於`awsRegion`，請 AWS 區域 以您在中建立的 Amazon S3 儲存貯體取`us-east-1`代。
      - 對於 `name`，將 `DOC-EXAMPLE-BUCKET` 取代為 Amazon S3 來源儲存貯體的名稱。
      - 對於 `key`，將 `test%2Fkey` 取代為您在步驟 將測試映像上傳到來源儲存貯體 中上傳至來源儲存貯體之測試物件的檔案名稱。

```
{
```

```
  "Records": [
    {
      "eventVersion": "2.0",
      "eventSource": "aws:s3",
      "awsRegion": "us-east-1",
      "eventTime": "1970-01-01T00:00:00.000Z",
      "eventName": "ObjectCreated:Put",
      "userIdentity": {
        "principalId": "EXAMPLE"
      },
      "requestParameters": {
        "sourceIPAddress": "127.0.0.1"
      },
      "responseElements": {
        "x-amz-request-id": "EXAMPLE123456789",
        "x-amz-id-2": "EXAMPLE123/5678abcdefghijklambdaisawesome/
mnopqrstuvwxyzABCDEFGH"
      },
      "s3": {
        "s3SchemaVersion": "1.0",
        "configurationId": "testConfigRule",
        "bucket": {
          "name": "DOC-EXAMPLE-BUCKET",
          "ownerIdentity": {
            "principalId": "EXAMPLE"
          },
          "arn": "arn:aws:s3:::DOC-EXAMPLE-BUCKET"
        },
        "object": {
          "key": "test%2Fkey",
          "size": 1024,
          "eTag": "0123456789abcdef0123456789abcdef",
          "sequencer": "0A1B2C3D4E5F678901"
        }
      }
    }
  ]
}
```

e. 選擇儲存。

4. 在測試事件窗格中，選擇測試。

5. 若要檢查您的函數已建立大小经过调整的映像版本並將其存放在目標 Amazon S3 儲存貯體中，請執行以下操作：

a. 開啟 Amazon S3 主控台的儲存貯體頁面。

b. 選擇目標儲存貯體，並確認在物件窗格中列出已調整大小的檔案。

AWS CLI

使用虛擬事件來測試 Lambda 函數 (AWS CLI)

1. 將下面的 JSON 儲存在名為 dummyS3Event.json 的檔案中。將下列參數的值取代為您自己的值：

   1. 對於awsRegion，請 AWS 區域 以您在中建立的 Amazon S3 儲存貯體取us-west-2代。
   2. 對於 name，將 DOC-EXAMPLE-SOURCE-BUCKET 取代為 Amazon S3 來源儲存貯體的名稱。
   3. 對於 key，將 HappyFace.jpg 取代為您在步驟 將測試映像上傳到來源儲存貯體 中上傳至來源儲存貯體之測試物件的檔案名稱。

```
{
  "Records":[
    {
      "eventVersion":"2.0",
      "eventSource":"aws:s3",
      "awsRegion":"us-west-2",
      "eventTime":"1970-01-01T00:00:00.000Z",
      "eventName":"ObjectCreated:Put",
      "userIdentity":{
        "principalId":"AIDAJDPLRKLG7UEXAMPLE"
      },
      "requestParameters":{
        "sourceIPAddress":"127.0.0.1"
      },
      "responseElements":{
        "x-amz-request-id":"C3D13FE58DE4C810",
        "x-amz-id-2":"FMyUVURIY8/IgAtTv8xRjskZQpcIZ9KG4V5Wp6S7S/
JRWeUWerMUE5JgHvANOjpD"
      },
      "s3":{
        "s3SchemaVersion":"1.0",
        "configurationId":"testConfigRule",
        "bucket":{
```

```
          "name":"DOC-EXAMPLE-SOURCE-BUCKET",
          "ownerIdentity":{
            "principalId":"A3NL1KOZZKExample"
          },
          "arn":"arn:aws:s3:::DOC-EXAMPLE-SOURCE-BUCKET"
        },
        "object":{
          "key":"HappyFace.jpg",
          "size":1024,
          "eTag":"d41d8cd98f00b204e9800998ecf8427e",
          "versionId":"096fKKXTRTtl3on89fVO.nfljtsv6qko"
        }
      }
    }
  ]
}
```

2. 在您儲存 dummyS3Event.json 檔案的目錄中，透過執行下列 CLI 命令來調用函數。此命令透過將 RequestResponse 指定為調用類型參數的值來同步調用 Lambda 函數。若要進一步了解同步和非同步調用，請參閱調用 Lambda 函數。

```
aws lambda invoke --function-name CreateThumbnail \
--invocation-type RequestResponse --cli-binary-format raw-in-base64-out \
--payload file://dummyS3Event.json outputfile.txt
```

如果您使用的是版本 2，則需要此 cli-binary-format 選項 AWS CLI。若要讓此成為預設的設定，請執行 aws configure set cli-binary-format raw-in-base64-out。如需詳細資訊，請參閱《支援AWS CLI 的全域命令列選項》。

3. 確認函數已建立映像的縮圖版本，並將其儲存到目標 Amazon S3 儲存貯體。執行以下 CLI 命令，將 DOC-EXAMPLE-SOURCE-BUCKET-resized 取代為您自己的目的地儲存貯體的名稱。

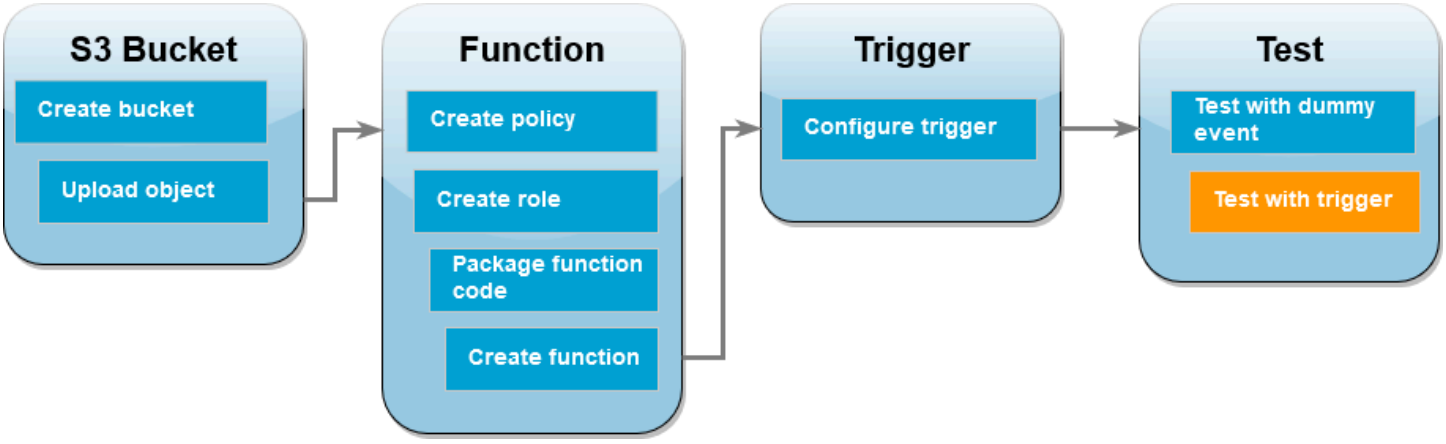
```
aws s3api list-objects-v2 --bucket DOC-EXAMPLE-SOURCE-BUCKET-resized
```

您應該會看到類似下列的輸出。Key 參數會顯示調整大小後的映像檔案名稱。

```
{
    "Contents": [
        {
            "Key": "resized-HappyFace.jpg",
            "LastModified": "2023-06-06T21:40:07+00:00",
            "ETag": "\"d8ca652ffe83ba6b721ffc20d9d7174a\"",
```

```
            "Size": 2633,
            "StorageClass": "STANDARD"
        }
    ]
}
```

使用 Amazon S3 觸發條件測試函數



既然您已確認 Lambda 函數運作正常，便可將映像檔案新增至 Amazon S3 來源儲存貯體來測試完整的設定。將映像新增至來源儲存貯體時，應該會自動調用 Lambda 函數。函數會建立已調整大小的檔案版本，並將其存放在目標儲存貯體中。

AWS Management Console

使用 Amazon S3 觸發條件測試 Lambda 函數 (主控台)

1. 若要將映像上傳到 Amazon S3 儲存貯體，請執行以下操作：
   a. 開啟 Amazon S3 主控台的儲存貯體頁面，並選擇來源儲存貯體。
   b. 選擇上傳。
   c. 選擇新增檔案，然後使用檔案選擇器選擇您要上傳的映像檔案。映像物件可以是任何 .jpg 或 .png 檔案。
   d. 選擇開啟，然後選擇上傳。
2. 透過執行下列操作，確認 Lambda 已在目標儲存貯體中儲存了調整大小後的映像檔案版本：
   a. 導覽回 Amazon S3 主控台的儲存貯體頁面，然後選擇目的地儲存貯體。
   b. 在物件窗格中，現在應該會看到兩個已調整大小的映像檔案，分別來自 Lambda 函數的每個測試。若要下載已調整大小的映像，請選取檔案，然後選擇下載。

AWS CLI

使用 Amazon S3 觸發條件測試 Lambda 函數 (AWS CLI)

1. 從包含要上傳之映像的目錄中，執行下列 CLI 命令。將 `--bucket` 參數取代為來源儲存貯體名稱。對於 `--key` 和 `--body` 參數，請使用測試映像的檔案名稱。測試映像可以是任何 .jpg 或 .png 檔案。

```
aws s3api put-object --bucket DOC-EXAMPLE-SOURCE-BUCKET --key SmileyFace.jpg --body ./SmileyFace.jpg
```

2. 確認函數已建立映像的縮圖版本，並將其儲存到目標 Amazon S3 儲存貯體。執行以下 CLI 命令，將 `DOC-EXAMPLE-SOURCE-BUCKET-resized` 取代為您自己的目的地儲存貯體的名稱。

```
aws s3api list-objects-v2 --bucket DOC-EXAMPLE-SOURCE-BUCKET-resized
```

如果函數成功運作，則您會看到類似以下內容的輸出。您的目標儲存貯體現在應包含兩個已調整大小的檔案。

```
{
    "Contents": [
        {
            "Key": "resized-HappyFace.jpg",
            "LastModified": "2023-06-07T00:15:50+00:00",
            "ETag": "\"7781a43e765a8301713f533d70968a1e\"",
            "Size": 2763,
            "StorageClass": "STANDARD"
        },
        {
            "Key": "resized-SmileyFace.jpg",
            "LastModified": "2023-06-07T00:13:18+00:00",
            "ETag": "\"ca536e5a1b9e32b22cd549e18792cdbc\"",
            "Size": 1245,
            "StorageClass": "STANDARD"
        }
    ]
}
```

清除您的資源

除非您想要保留為此教學課程建立的資源，否則您現在便可刪除。刪除您不再使用的 AWS 資源，您可以避免不必要的費用 AWS 帳戶。

若要刪除 Lambda 函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇您建立的函數。
3. 選擇 Actions (動作)、Delete (刪除)。
4. 在文字輸入欄位中輸入 **delete**，然後選擇 Delete (刪除)。

刪除您建立的政策

1. 開啟 IAM 主控台中的 Policies (政策) 頁面。
2. 選取您建立的策略 (AWSLambdaS3Policy)。
3. 選擇 政策動作，然後 刪除。
4. 選擇 刪除 。

若要刪除執行角色

1. 開啟 IAM 主控台中的 角色頁面 。
2. 選取您建立的執行角色。
3. 選擇 刪除 。
4. 在文字輸入欄位中輸入角色的名稱，然後選擇 刪除 。

刪除 S3 儲存貯體

1. 開啟 Amazon S3 主控台。
2. 選擇您建立的儲存貯體。
3. 選擇 刪除 。
4. 在文字輸入欄位中輸入儲存貯體的名稱。
5. 選擇 刪除儲存貯體 。

AWS Lambda 與 Amazon S3 批次操作搭配使用

您可以使用 Amazon S3 批次操作，在大型 Amazon S3 物件組合叫用 Lambda 函數。Amazon S3 會追蹤批次操作的進度、傳送通知以及儲存顯示每個動作狀態的完成報告。

若要執行批次操作，您可以建立 Amazon S3 批次操作任務。當您建立任務時，請提供資訊清單 (物件清單)，並設定要在這些物件上執行的動作。

當批次任務開始時，Amazon S3 會針對資訊清單中的每個物件同步叫用 Lambda 函數。事件參數包括儲存貯體和物件的名稱。

下列範例顯示 Amazon S3 針對在文件例子儲存貯體中名為 customerImage1.jpg 的物件傳送至 Lambda 函數的事件。

Example Amazon S3 批次請求事件

```
{
"invocationSchemaVersion": "1.0",
    "invocationId": "YXNkbGZqYWRmaiBhc2RmdW9hZHNmZGpmaGFzbGtkaGZza2RmaAo",
    "job": {
        "id": "f3cc4f60-61f6-4a2b-8a21-d07600c373ce"
    },
    "tasks": [
        {
            "taskId": "dGFza2lkZ29lc2hlcmUK",
            "s3Key": "customerImage1.jpg",
            "s3VersionId": "1",
            "s3BucketArn": "arn:aws:s3:::DOC-EXAMPLE-BUCKET"
        }
    ]
}
```

您的 Lambda 函數必須傳回一個 JSON 物件，並有如下例所示的欄位。您可以從事件參數複製 `invocationId` 和 `taskId`。您可以在 `resultString` 返回子串。Amazon S3 會在完成報告中儲存 `resultString` 值。

Example Amazon S3 批次請求回應

```
{
```

```
  "invocationSchemaVersion": "1.0",
  "treatMissingKeysAs" : "PermanentFailure",
  "invocationId" : "YXNkbGZqYWRmaiBhc2RmdW9hZHNmZGpmaGFzbGtkaGZza2RmaAo",
  "results": [
    {
      "taskId": "dGFza2lkZ29lc2hlcmUK",
      "resultCode": "Succeeded",
      "resultString": "[\"Alice\", \"Bob\"]"
    }
  ]
}
```

從 Amazon S3 批次操作叫用 Lambda 函數

您可以使用不合格或合格的函數 ARN 叫用 Lambda 函數。如果您要在整個批次任務中使用相同的函數版本，請在建立任務時，在 FunctionARN 參數中設定特定的函數版本。如果您設定別名或 $LATEST 限定詞，如果在任務執行期間更新別名或 $LATEST，批次任務就會立即開始呼叫新版函數。

請注意，您無法重複使用現有的 Amazon S3 事件型函數進行批次操作。這是因為 Amazon S3 批次操作將不同的事件參數傳遞給 Lambda 函數，並預期傳回有特定 JSON 結構的訊息。

在您針對 Amazon S3 批次任務建立的資源型政策中，請確認您已針對任務設定權限以叫用您的 Lambda 函數。

在函數的執行角色中，針對 Amazon S3 設定信任政策以在角色執行您的函數時取得該角色。

如果您的函數使用 AWS 開發套件來管理 Amazon S3 資源，則需要在執行角色中新增 Amazon S3 許可。

執行任務時，Amazon S3 會啟動多個函數執行個體以並行處理 Amazon S3 物件，直至達到函數的並行限制。Amazon S3 會限制執行個體的初始漸進測試，以避免較小任務造成過多成本。

如果 Lambda 函數傳回 TemporaryFailure 回應程式碼，Amazon S3 就會重試操作。

如需 Amazon S3 批次操作的詳細資訊，請參閱 Amazon S3 開發人員指南中的執行批次操作。

如需如何在 Amazon S3 批次操作中使用 Lambda 函數的範例，請參閱 Amazon S3 開發人員指南中的從 Amazon S3 批次操作中叫用 Lambda 函數。

使用 S3 Object Lambda 轉換 S3 物件

藉助 S3 Object Lambda，您可將自己的程式碼新增至 Amazon S3 GET、HEAD 和 LIST 請求，以便在資料傳回應用程式前對其做出修改和處理。您可以使用自訂程式碼修改標準 S3 GET、HEAD 和 LIST 請求返回的資料，以執行資料列篩選、動態調整影像大小、修訂機密資料以及更多動作。採用 AWS Lambda 函數，您的程式碼在由 AWS 全受管的基礎設施上運行，這就避免了建立和存儲資料的衍生副本或運行代理，並且全程無需對應用程式進行任何變更。

如需詳細資訊，請參閱使用 S3 Object Lambda 轉換物件。

教學課程

- 使用 Amazon S3 Object Lambda 轉換應用程式的資料
- 使用 Amazon S3 Object Lambda 和 Amazon Comprehend 來偵測和編輯 PII 資料
- 使用 Amazon S3 Object Lambda 在擷取影像時動態加上浮水印

搭配使用 AWS Lambda 與 Secrets Manager

您的 AWS Lambda 函數可以使用 Secrets Manager API 或任何AWS軟體開發套件 (SDK) 與 AWS Secrets Manager 互動。您也可以使用 AWS 參數和秘密 Lambda 延伸項目，擷取和快取 Lambda 函數中的 AWS Secrets Manager 秘密，無需使用 SDK。如需詳細資訊，請參閱在 AWS Lambda 函數中使用 AWS Secrets Manager 秘密。

搭配使用 AWS Lambda 與 Amazon SES

若您使用 Amazon SES 接收訊息，則可設定 Amazon SES，以便在收到訊息時呼叫 Lambda 函數。然後，該服務會透過傳遞內送電子郵件事件，進而叫用 Lambda 函數。事實上，此內送電子郵件事件是 Amazon SES 事件中的 Amazon SNS 訊息，其會以參數形式存在。

Example Amazon SES 訊息事件

```
{
  "Records": [
    {
      "eventVersion": "1.0",
      "ses": {
        "mail": {
          "commonHeaders": {
            "from": [
              "Jane Doe <janedoe@example.com>"
            ],
            "to": [
              "johndoe@example.com"
            ],
            "returnPath": "janedoe@example.com",
            "messageId": "<0123456789example.com>",
            "date": "Wed, 7 Oct 2015 12:34:56 -0700",
            "subject": "Test Subject"
          },
          "source": "janedoe@example.com",
          "timestamp": "1970-01-01T00:00:00.000Z",
          "destination": [
            "johndoe@example.com"
          ],
          "headers": [
            {
              "name": "Return-Path",
              "value": "<janedoe@example.com>"
            },
            {
              "name": "Received",
              "value": "from mailer.example.com (mailer.example.com [203.0.113.1])
 by inbound-smtp.us-west-2.amazonaws.com with SMTP id o3vrnil0e2ic for
 johndoe@example.com; Wed, 07 Oct 2015 12:34:56 +0000 (UTC)"
            },
            {
```

```
              "name": "DKIM-Signature",
              "value": "v=1; a=rsa-sha256; c=relaxed/relaxed; d=example.com;
 s=example; h=mime-version:from:date:message-id:subject:to:content-type;
 bh=jX3F0bCAI7sIbkHyy3mLYO28ieDQz2R0P8HwQkklFj4=; b=sQwJ+LMe9RjkesGu
+vqU56asvMhrLRRYrWCbV"
            },
            {
              "name": "MIME-Version",
              "value": "1.0"
            },
            {
              "name": "From",
              "value": "Jane Doe <janedoe@example.com>"
            },
            {
              "name": "Date",
              "value": "Wed, 7 Oct 2015 12:34:56 -0700"
            },
            {
              "name": "Message-ID",
              "value": "<0123456789example.com>"
            },
            {
              "name": "Subject",
              "value": "Test Subject"
            },
            {
              "name": "To",
              "value": "johndoe@example.com"
            },
            {
              "name": "Content-Type",
              "value": "text/plain; charset=UTF-8"
            }
          ],
          "headersTruncated": false,
          "messageId": "o3vrnil0e2ic28tr"
        },
        "receipt": {
          "recipients": [
            "johndoe@example.com"
          ],
          "timestamp": "1970-01-01T00:00:00.000Z",
          "spamVerdict": {
```

```
            "status": "PASS"
          },
          "dkimVerdict": {
            "status": "PASS"
          },
          "processingTimeMillis": 574,
          "action": {
            "type": "Lambda",
            "invocationType": "Event",
            "functionArn": "arn:aws:lambda:us-west-2:111122223333:function:Example"
          },
          "spfVerdict": {
            "status": "PASS"
          },
          "virusVerdict": {
            "status": "PASS"
          }
        }
      },
      "eventSource": "aws:ses"
    }
  ]
}
```

如需詳細資訊，請參閱《Amazon SES 開發人員指南》中的「Lambda 動作」。

使用 Amazon SNS 通知叫用 Lambda 函數

您可以使用 Lambda 函數來處理 Amazon Simple Notification Service (Amazon SNS) 通知。Amazon SNS 支援 Lambda 函數作為傳送至主題之訊息的目標。您可以將自己的函數訂閱至相同帳戶或其他 AWS 帳戶中的主題。如需詳細演練，請參閱the section called “教學課程”。

Lambda 僅支援標準 SNS 主題的 SNS 觸發器。不支援先進先出主題。

針對非同步叫用，Lambda 會將訊息排入佇列並處理重試。如果 Amazon SNS 無法連接至 Lambda 或訊息遭到拒絕，Amazon SNS 會在數小時中以增加的間隔重試。如需詳細資訊，請參閱 Amazon SNS 常見問答集中的可靠性。

> **Warning**
>
> Lambda 事件來源對應至少處理每個事件一次，並且可能會重複處理記錄。為了避免與重複事件相關的潛在問題，我們強烈建議您將函數代碼設為冪等。若要深入了解，請參閱 AWS 知識中心如何讓 Lambda 函數具有冪等性。

主題

- 使用主控台為 Lambda 函數新增 Amazon SNS 主題觸發器
- 為 Lambda 函數手動新增 Amazon SNS 主題觸發器
- SNS 事件形狀範例
- 教學課程：搭 AWS Lambda 配 Amazon 簡易通知服務使用

使用主控台為 Lambda 函數新增 Amazon SNS 主題觸發器

若要將 SNS 主題新增為 Lambda 函數的觸發器，最簡單的方法是使用 Lambda 主控台。當您透過主控台新增觸發器時，Lambda 會自動設定必要的權限和訂閱，以開始接收來自 SNS 主題的事件。

若要將 SNS 主題新增為 Lambda 函數 (主控台) 的觸發器

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇您要為其新增觸發程式的函數名稱。
3. 選擇 [組態]，然後選擇 [觸發器]。
4. 選擇 Add trigger (新增觸發條件)。

5. 在 [觸發器設定] 下方的下拉式功能表中，選擇 [SNS]。
6. 對於 SNS 主題，請選擇要訂閱的 SNS 主題。

為 Lambda 函數手動新增 Amazon SNS 主題觸發器

若要手動設定 Lambda 函數的 SNS 觸發器，您需要完成下列步驟：

- 為您的函數定義以資源為基礎的策略，以允許 SNS 調用它。
- 向 Amazon SNS 主題訂閱您的 Lambda 函數。

> **Note**
>
> 如果您的 SNS 主題和 Lambda 函數位於不同的 AWS 帳戶中，您還需要授予額外權限，以允許跨帳戶訂閱 SNS 主題。如需詳細資訊，請參閱授與 Amazon SNS 訂閱的跨帳戶權限。

您可以使用 AWS Command Line Interface (AWS CLI) 來完成這兩個步驟。首先，若要為允許 SNS 叫用的 Lambda 函數定義以資源為基礎的政策，請使用下列命令。 AWS CLI 請務必將的值取代為您`--function-name`的 Lambda 函數名稱，並將的值取代為您`--source-arn`的 SNS 主題 ARN。

```
aws lambda add-permission --function-name example-function \
    --source-arn arn:aws:sns:us-east-1:123456789012:sns-topic-for-lambda \
    --statement-id function-with-sns --action "lambda:InvokeFunction" \
    --principal sns.amazonaws.com
```

若要訂閱 SNS 主題的功能，請使用下列 AWS CLI 指令。將的值取代為您`--topic-arn`的 SNS 主題 ARN，並將的值取代為您`--notification-endpoint`的 Lambda 函數 ARN。

```
aws sns subscribe --protocol lambda \
    --region us-east-1 \
    --topic-arn arn:aws:sns:us-east-1:123456789012:sns-topic-for-lambda \
    --notification-endpoint arn:aws:lambda:us-east-1:123456789012:function:example-
function
```

SNS 事件形狀範例

Amazon SNS 使用包含訊息和中繼資料的事件，以非同步方式叫用您的函數。

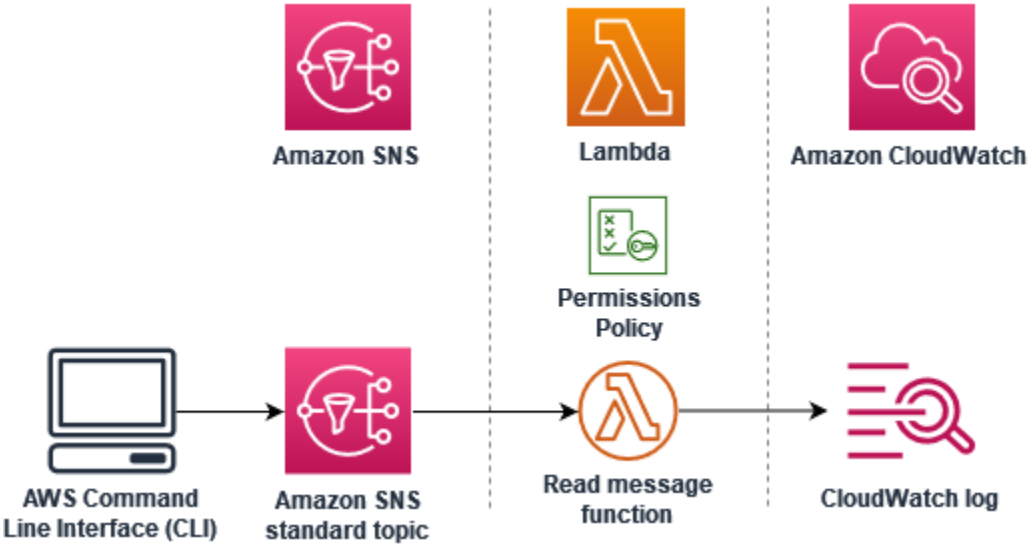
Example Amazon SNS 訊息事件

```
{
  "Records": [
    {
      "EventVersion": "1.0",
      "EventSubscriptionArn": "arn:aws:sns:us-east-1:123456789012:sns-lambda:21be56ed-
a058-49f5-8c98-aedd2564c486",
      "EventSource": "aws:sns",
      "Sns": {
        "SignatureVersion": "1",
        "Timestamp": "2019-01-02T12:45:07.000Z",
        "Signature": "tcc6faL2yUC6dgZdmrwh1Y4cGa/ebXEkAi6RibDsvpi+tE/1+82j...65r==",
        "SigningCertURL": "https://sns.us-east-1.amazonaws.com/
SimpleNotificationService-ac565b8b1a6c5d002d285f9598aa1d9b.pem",
        "MessageId": "95df01b4-ee98-5cb9-9903-4c221d41eb5e",
        "Message": "Hello from SNS!",
        "MessageAttributes": {
          "Test": {
            "Type": "String",
            "Value": "TestString"
          },
          "TestBinary": {
            "Type": "Binary",
            "Value": "TestBinary"
          }
        },
        "Type": "Notification",
        "UnsubscribeURL": "https://sns.us-east-1.amazonaws.com/?
Action=Unsubscribe&amp;SubscriptionArn=arn:aws:sns:us-east-1:123456789012:test-
lambda:21be56ed-a058-49f5-8c98-aedd2564c486",
        "TopicArn":"arn:aws:sns:us-east-1:123456789012:sns-lambda",
        "Subject": "TestInvoke"
      }
    }
  ]
}
```

教學課程：搭 AWS Lambda 配 Amazon 簡易通知服務使用

在本教學中，您可以使用一個 AWS 帳戶 Lambda 函數來單獨訂閱亞馬遜簡單通知服務 (Amazon SNS) 主題 AWS 帳戶。當您將訊息發佈到 Amazon SNS 主題時，Lambda 函數會讀取訊息的內容，並將

其輸出到 Amazon CloudWatch 日誌。若要完成此自學課程，請使用 AWS Command Line Interface (AWS CLI)。



請執行下列步驟以完成本教學課程：

- 在帳戶 A 中建立 Amazon SNS 主題。
- 在帳戶 B 中建立 Lambda 函數，以讀取主題的訊息。
- 在帳戶 B 中建立主題的訂閱。
- 將訊息發佈到帳戶 A 中的 Amazon SNS 主題，並確認帳戶 B 中的 Lambda 函數將這些訊息輸出到 CloudWatch 日誌。

完成這些步驟後，您將了解如何設定 Amazon SNS 主題來調用 Lambda 函數。您也將學習如何建立 AWS Identity and Access Management (IAM) 政策，為另一個資源提供呼叫 Lambda AWS 帳戶 的權限。

在此教學課程中，您會使用兩種獨立的 AWS 帳戶。這些指 AWS CLI 令透過使用兩個名為 `accountA` AND 的具名輪廓來說明這一點`accountB`，每個設定為使用不同的輪廓 AWS 帳戶。若要瞭解如何設定 AWS CLI 使用不同的設定檔，請參閱第 2 版使用AWS Command Line Interface 者指南中的組態和認證檔案設定。請務必為兩個設定檔設定相同 AWS 區域 的預設值。

如果您為兩者建立的 AWS CLI 紀要 AWS 帳戶 使用不同的名稱，或者如果您使用預設紀要和一個具名的設定檔，請依需要在以下步驟中修改 AWS CLI 指令。

必要條件

註冊一個 AWS 帳戶

如果您沒有 AWS 帳戶，請完成以下步驟來建立一個。

若要註冊成為 AWS 帳戶

1. 開啟 https://portal.aws.amazon.com/billing/signup。
2. 請遵循線上指示進行。

   部分註冊程序需接收來電，並在電話鍵盤輸入驗證碼。

   當您註冊時 AWS 帳戶，會建立AWS 帳戶根使用者一個。根使用者有權存取該帳戶中的所有 AWS 服務 和資源。安全性最佳做法是將管理存取權指派給使用者，並僅使用 root 使用者來執行需要 root 使用者存取權的工作。

AWS 註冊過程完成後，會向您發送確認電子郵件。您可以隨時登錄 https://aws.amazon.com/ 並選擇我的帳戶，以檢視您目前的帳戶活動並管理帳戶。

建立具有管理權限的使用者

註冊後，請保護您的 AWS 帳戶 AWS 帳戶根使用者 AWS IAM Identity Center、啟用和建立系統管理使用者，這樣您就不會將 root 使用者用於日常工作。

保護您的 AWS 帳戶根使用者

1. 選擇 Root 使用者並輸入您的 AWS 帳戶 電子郵件地址，以帳戶擁有者身分登入。AWS Management Console在下一頁中，輸入您的密碼。

   如需使用根使用者登入的說明，請參閱 AWS 登入 使用者指南中的以根使用者身分登入。
2. 若要在您的根使用者帳戶上啟用多重要素驗證 (MFA)。

   如需指示，請參閱《IAM 使用者指南》中的為 AWS 帳戶 根使用者啟用虛擬 MFA 裝置 (主控台)。

建立具有管理權限的使用者

1. 啟用 IAM Identity Center。

   如需指示，請參閱 AWS IAM Identity Center 使用者指南中的啟用 AWS IAM Identity Center。

2. 在 IAM 身分中心中，將管理存取權授予使用者。

   若要取得有關使用 IAM Identity Center 目錄 做為身分識別來源的自學課程，請參閱《使用指南》IAM Identity Center 目錄中的「以預設值設定使用AWS IAM Identity Center 者存取」。

以具有管理權限的使用者身分登入

- 若要使用您的 IAM Identity Center 使用者簽署，請使用建立 IAM Identity Center 使用者時傳送至您電子郵件地址的簽署 URL。

  如需使用 IAM 身分中心使用者登入的說明，請參閱使用AWS 登入 者指南中的登入 AWS 存取入口網站。

指派存取權給其他使用者

1. 在 IAM 身分中心中，建立遵循套用最低權限許可的最佳做法的權限集。

   如需指示，請參閱《AWS IAM Identity Center 使用指南》中的「建立權限集」。
2. 將使用者指派給群組，然後將單一登入存取權指派給群組。

   如需指示，請參閱《AWS IAM Identity Center 使用指南》中的「新增群組」。

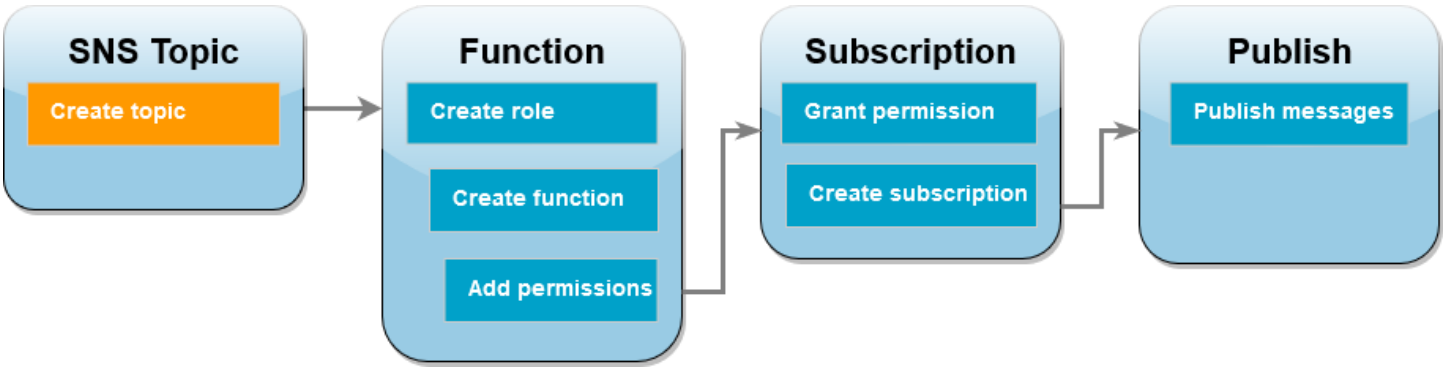
安裝 AWS Command Line Interface

如果您尚未安裝 AWS Command Line Interface，請按照安裝或更新最新版本的步驟進 AWS CLI行安裝。

本教學課程需使用命令列終端機或 Shell 來執行命令。在 Linux 和 macOS 中，使用您偏好的 Shell 和套件管理工具。

> **Note**
>
> 在 Windows 中，作業系統的內建終端不支援您常與 Lambda 搭配使用的某些 Bash CLI 命令 (例如 `zip`)。若要取得 Ubuntu 和 Bash 的 Windows 整合版本，請安裝適用於 Linux 的 Windows 子系統。

建立 Amazon SNS 主題 (帳戶 A)



若要建立 主題

- 在帳戶 A 中，使用下列 AWS CLI 命令建立 Amazon SNS 標準主題。

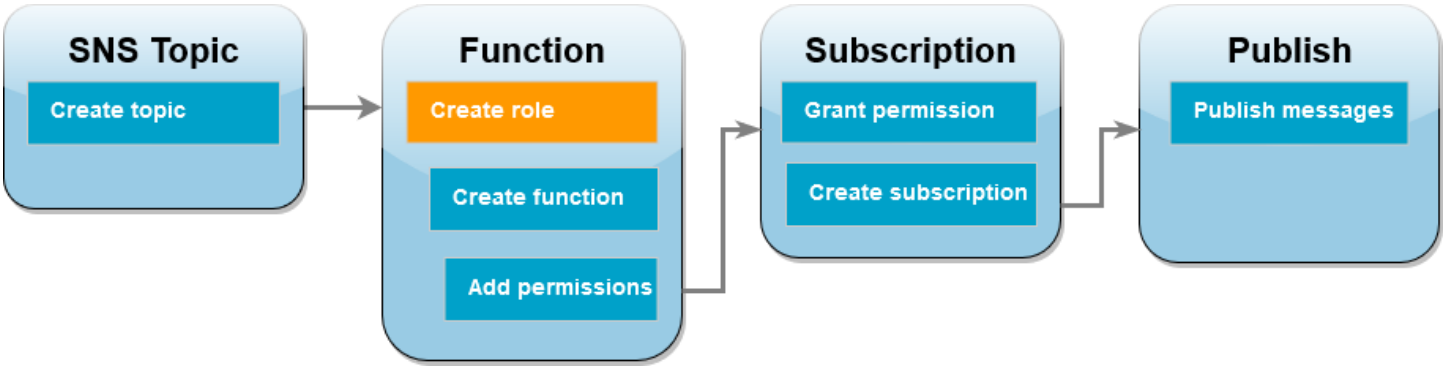
  ```
  aws sns create-topic --name sns-topic-for-lambda --profile accountA
  ```

  您應該會看到類似下列的輸出。

  ```
  {
      "TopicArn": "arn:aws:sns:us-west-2:123456789012:sns-topic-for-lambda"
  }
  ```

  記下主題的 Amazon Resource Name (ARN)。當您新增許可到 Lambda 函數以訂閱主題時，稍後會在教學課程中用上它。

建立函數執行角色 (帳戶 B)

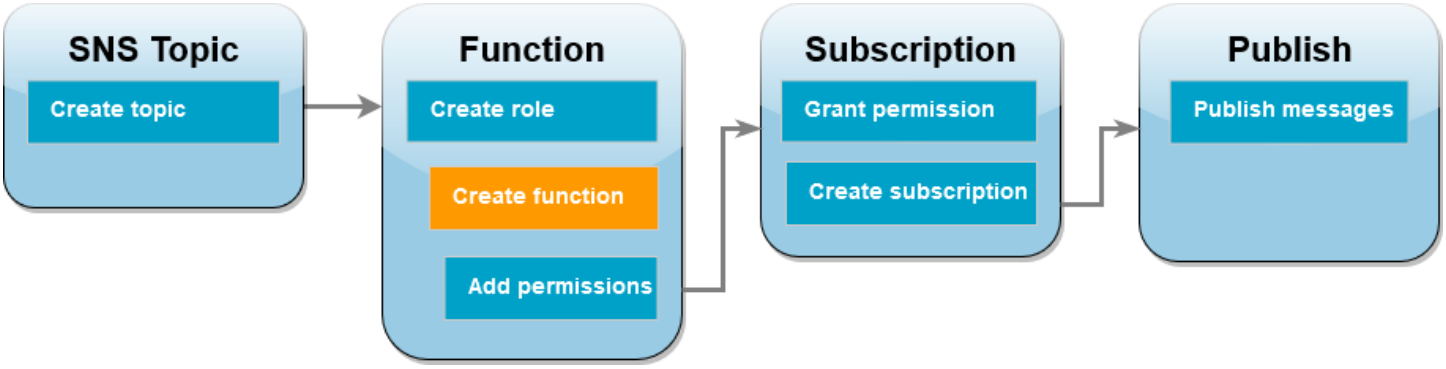


執行角色是一種 IAM 角色，可授與 Lambda 函數存取 AWS 服務和資源的權限。在帳戶 B 中建立函數之前，請先建立一個角色，該角色會授予函數將記錄寫入記錄檔的基本權限。 CloudWatch 我們將在稍後的步驟中新增要從 Amazon SNS 主題讀取的許可。

若要建立執行角色

1. 在帳戶 B 中開啟 IAM 主控台的角色頁面。
2. 選擇建立角色。
3. 針對信任的實體類型，請選擇 AWS 服務。
4. 針對使用案例，請選擇 Lambda。
5. 選擇下一步。
6. 透過下列步驟將基本許可政策新增至角色：
   a. 在許可政策搜尋方塊中，輸入 **AWSLambdaBasicExecutionRole**。
   b. 選擇下一步。
7. 執行下列動作來完成角色的建立：
   a. 在角色詳細資訊下方的角色名稱中輸入 **lambda-sns-role**。
   b. 選擇建立角色。

建立 Lambda 函數 (帳戶 B)



建立可處理 Amazon SQS 訊息的 Lambda 函數。函數代碼將每條記錄的消息內容記錄到 Amazon CloudWatch 日誌。

本教學課程使用 Node.js 18.x 執行期，但我們也有提供其他執行期語言的範例程式碼。您可以在下列方塊中選取索引標籤，查看您感興趣的執行期程式碼。您將在此步驟中使用的 JavaScript 程式碼位於JavaScript索引標籤中顯示的第一個範例中。

.NET

AWS SDK for .NET

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 .NET 搭配 Lambda 來使用 SNS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SNSEvents;


// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace SnsIntegration;

public class Function
{
    public async Task FunctionHandler(SNSEvent evnt, ILambdaContext context)
    {
        foreach (var record in evnt.Records)
        {
            await ProcessRecordAsync(record, context);
        }
        context.Logger.LogInformation("done");
    }

    private async Task ProcessRecordAsync(SNSEvent.SNSRecord record,
 ILambdaContext context)
    {
        try
        {
            context.Logger.LogInformation($"Processed record
 {record.Sns.Message}");
```

```
            // TODO: Do interesting work based on the new message
            await Task.CompletedTask;
        }
        catch (Exception e)
        {
            //You can use Dead Letter Queue to handle failures. By configuring a
 Lambda DLQ.
            context.Logger.LogError($"An error occurred");
            throw;
        }
    }
}
```

Go

SDK for Go V2

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 搭配 Lambda 來使用 SNS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "fmt"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, snsEvent events.SNSEvent) {
 for _, record := range snsEvent.Records {
  processMessage(record)
 }
```

```
 fmt.Println("done")
}

func processMessage(record events.SNSEventRecord) {
 message := record.SNS.Message
 fmt.Printf("Processed message: %s\n", message)
 // TODO: Process your record here
}

func main() {
 lambda.Start(handler)
}
```

Java

適用於 Java 2.x 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Java 與 Lambda 一起使用 SNS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SNSEvent;
import com.amazonaws.services.lambda.runtime.events.SNSEvent.SNSRecord;


import java.util.Iterator;
import java.util.List;

public class SNSEventHandler implements RequestHandler<SNSEvent, Boolean> {
    LambdaLogger logger;
```

```
    @Override
    public Boolean handleRequest(SNSEvent event, Context context) {
        logger = context.getLogger();
        List<SNSRecord> records = event.getRecords();
        if (!records.isEmpty()) {
            Iterator<SNSRecord> recordsIter = records.iterator();
            while (recordsIter.hasNext()) {
                processRecord(recordsIter.next());
            }
        }
        return Boolean.TRUE;
    }

    public void processRecord(SNSRecord record) {
        try {
            String message = record.getSNS().getMessage();
            logger.log("message: " + message);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

}
```

JavaScript

適用於 JavaScript (v3) 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使 Lambda JavaScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
```

```
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    await processMessageAsync(record);
  }
  console.info("done");
};

async function processMessageAsync(record) {
  try {
    const message = JSON.stringify(record.Sns.Message);
    console.log(`Processed message ${message}`);
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

使 Lambda TypeScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SNSEvent, Context, SNSHandler, SNSEventRecord } from "aws-lambda";

export const functionHandler: SNSHandler = async (
  event: SNSEvent,
  context: Context
): Promise<void> => {
  for (const record of event.Records) {
    await processMessageAsync(record);
  }
  console.info("done");
};

async function processMessageAsync(record: SNSEventRecord): Promise<any> {
  try {
    const message: string = JSON.stringify(record.Sns.Message);
    console.log(`Processed message ${message}`);
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
```

```
    throw err;
  }
}
```

PHP

適用於 PHP 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 PHP 搭配 Lambda 來使用 SNS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

/*
Since native PHP support for AWS Lambda is not available, we are utilizing Bref's
 PHP functions runtime for AWS Lambda.
For more information on Bref's PHP runtime for Lambda, refer to: https://bref.sh/
docs/runtimes/function

Another approach would be to create a custom runtime.
A practical example can be found here: https://aws.amazon.com/blogs/apn/aws-
lambda-custom-runtime-for-php-a-practical-example/
*/

// Additional composer packages may be required when using Bref or any other PHP
 functions runtime.
// require __DIR__ . '/vendor/autoload.php';

use Bref\Context\Context;
use Bref\Event\Sns\SnsEvent;
use Bref\Event\Sns\SnsHandler;

class Handler extends SnsHandler
{
```

```
    public function handleSns(SnsEvent $event, Context $context): void
    {
        foreach ($event->getRecords() as $record) {
            $message = $record->getMessage();

            // TODO: Implement your custom processing logic here
            // Any exception thrown will be logged and the invocation will be
 marked as failed

            echo "Processed Message: $message" . PHP_EOL;
        }
    }
}

return new Handler();
```

Python

適用於 Python (Boto3) 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Python 搭配 Lambda 來使用 SNS 事件。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event, context):
    for record in event['Records']:
        process_message(record)
    print("done")

def process_message(record):
    try:
        message = record['Sns']['Message']
        print(f"Processed message {message}")
        # TODO; Process your record here
```

```
    except Exception as e:
        print("An error occurred")
        raise e
```

Ruby

適用於 Ruby 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用紅寶石使用 Lambda 的 SNS 事件。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event:, context:)
  event['Records'].map { |record| process_message(record) }
end

def process_message(record)
  message = record['Sns']['Message']
  puts("Processing message: #{message}")
rescue StandardError => e
  puts("Error processing message: #{e}")
  raise
end
```

Rust

適用於 Rust 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Rust 搭配 Lambda 來使用 SNS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::sns::SnsEvent;
use aws_lambda_events::sns::SnsRecord;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};
use tracing::info;

// Built with the following dependencies:
//  aws_lambda_events = { version = "0.10.0", default-features = false, features
 = ["sns"] }
//  lambda_runtime = "0.8.1"
//  tokio = { version = "1", features = ["macros"] }
//  tracing = { version = "0.1", features = ["log"] }
//  tracing-subscriber = { version = "0.3", default-features = false, features =
 ["fmt"] }

async fn function_handler(event: LambdaEvent<SnsEvent>) -> Result<(), Error> {
    for event in event.payload.records {
        process_record(&event)?;
    }

    Ok(())
}

fn process_record(record: &SnsRecord) -> Result<(), Error> {
    info!("Processing SNS Message: {}", record.sns.message);

    // Implement your record handling code here.

    Ok(())
}
```

```
#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        .with_target(false)
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

建立函數

1. 建立專案的目錄，然後切換至該目錄。

```
mkdir sns-tutorial
cd sns-tutorial
```

2. 將範例程 JavaScript 式碼複製到名為的新檔案中`index.js`。
3. 使用以下 `zip` 命令建立部署套件。

```
zip function.zip index.js
```

4. 執行下列 AWS CLI 命令，在帳戶 B 中建立您的 Lambda 函數。

```
aws lambda create-function --function-name Function-With-SNS \
    --zip-file fileb://function.zip --handler index.handler --runtime nodejs18.x \
    --role arn:aws:iam::<AccountB_ID>:role/lambda-sns-role  \
    --timeout 60 --profile accountB
```

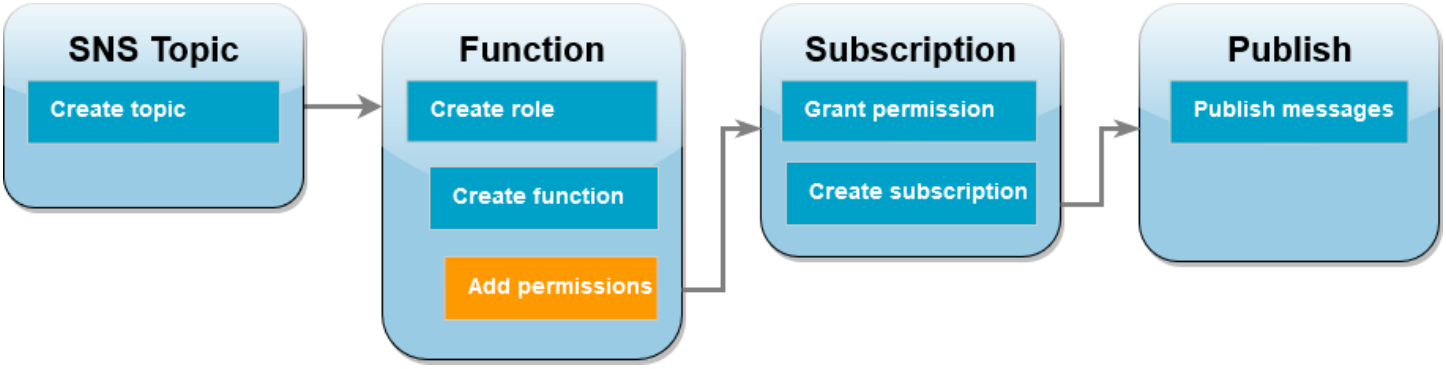
您應該會看到類似下列的輸出。

```
{
    "FunctionName": "Function-With-SNS",
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:Function-With-
SNS",
    "Runtime": "nodejs18.x",
    "Role": "arn:aws:iam::123456789012:role/lambda_basic_role",
    "Handler": "index.handler",
    ...
```

```
    "RuntimeVersionConfig": {
        "RuntimeVersionArn": "arn:aws:lambda:us-
west-2::runtime:7d5f06b69c951da8a48b926ce280a9daf2e8bb1a74fc4a2672580c787d608206"
    }
}
```

5. 記錄函數的 Amazon Resource Name (ARN)。當您新增許可以允許 Amazon SNS 調用您的函數時，稍後會在教學課程中用上它。

為函數新增許可 (帳戶 B)



若要使用 Amazon SNS 調用函數，您必須在以資源為基礎之政策的陳述式中授予許可。您可以使用 AWS CLI `add-permission`命令加入這個陳述式。

若要授予 Amazon SNS 調用您函數的許可

- 在帳戶 B 中，針對先前錄製的 Amazon SNS 主題使用 ARN 執行下列 AWS CLI 命令。

```
aws lambda add-permission --function-name Function-With-SNS \
    --source-arn arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda \
    --statement-id function-with-sns --action "lambda:InvokeFunction" \
    --principal sns.amazonaws.com --profile accountB
```

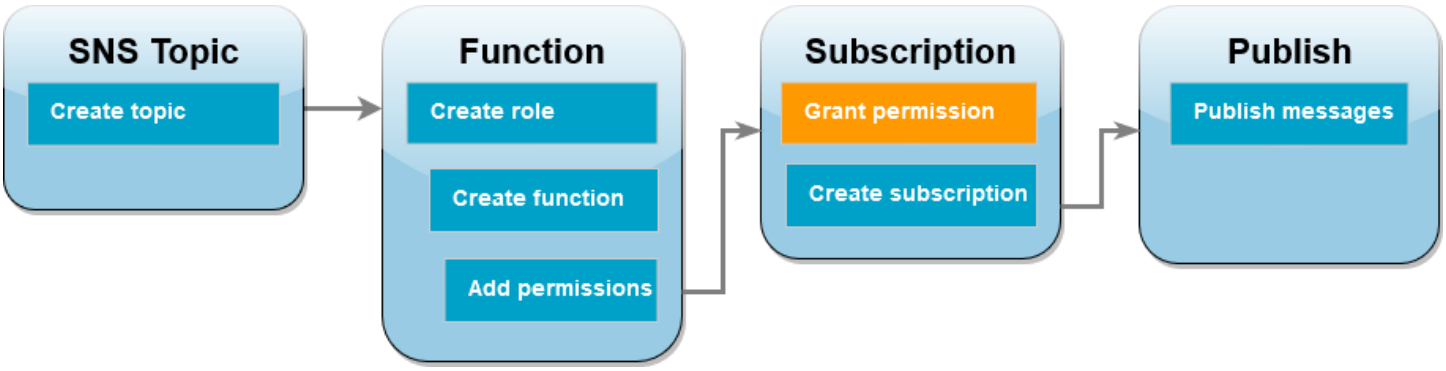
您應該會看到類似下列的輸出。

```
{
    "Statement": "{\"Condition\":{\"ArnLike\":{\"AWS:SourceArn\":
      \"arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda\"}},
      \"Action\":[\"lambda:InvokeFunction\"],
      \"Resource\":\"arn:aws:lambda:us-east-1:<AccountB_ID>:function:Function-With-
SNS\",
      \"Effect\":\"Allow\",\"Principal\":{\"Service\":\"sns.amazonaws.com\"},
```

```
      \"Sid\":\"function-with-sns\"}"
}
```

> **Note**
>
> 如果具有 Amazon SNS 主題的帳戶是在選擇加入中託管 AWS 區域，則需要在主體中指定區域。例如，如果您使用亞太區域 (香港)的 Amazon SNS 主題，則需要為主體指定 `sns.ap-east-1.amazonaws.com`，而不是 `sns.amazonaws.com`。

授予 Amazon SNS 訂閱的跨帳戶許可 (帳戶 A)



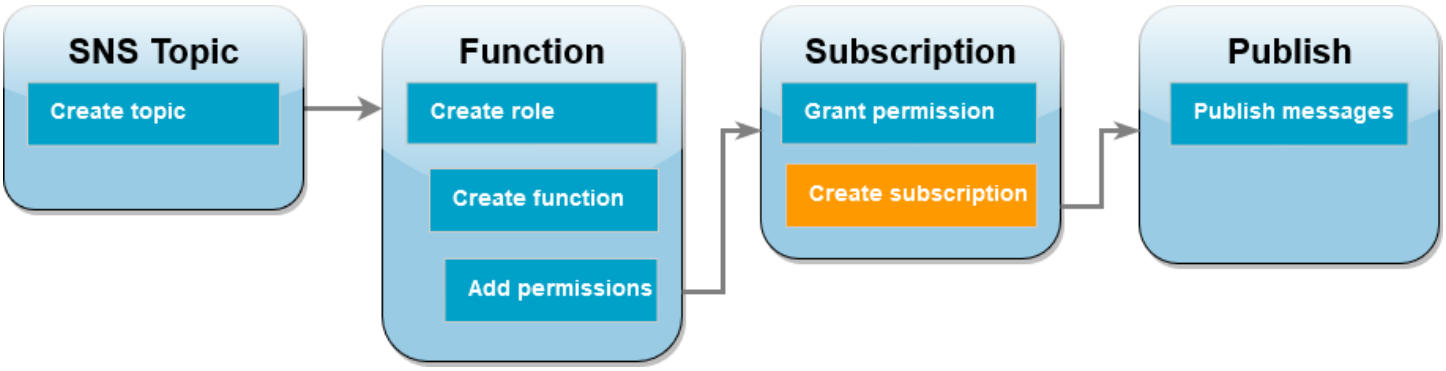
若要讓帳戶 B 中的 Lambda 函數訂閱您在帳戶 A 中建立的 Amazon SNS 主題，您必須向帳戶 B 授予訂閱您主題的許可。您可以使用 AWS CLI `add-permission`命令授與此權限。

若要向帳戶 B 授予訂閱主題的許可

- 在帳戶 A 中，執行下列 AWS CLI 命令。將 ARN 用於之前記錄的 Amazon SNS 主題。

```
aws sns add-permission --label lambda-access --aws-account-id <AccountB_ID> \
    --topic-arn arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda \
    --action-name Subscribe ListSubscriptionsByTopic --profile accountA
```

建立訂閱 (帳戶 B)



在帳戶 B 中，您現在讓 Lambda 函數訂閱您在教學課程開始時透過帳戶 A 建立的 Amazon SNS 主題。當訊息傳送至此主題 (`sns-topic-for-lambda`) 時，Amazon SNS 會調用您帳戶 B 中的 Lambda 函數 `Function-With-SNS`。

若要建立訂閱

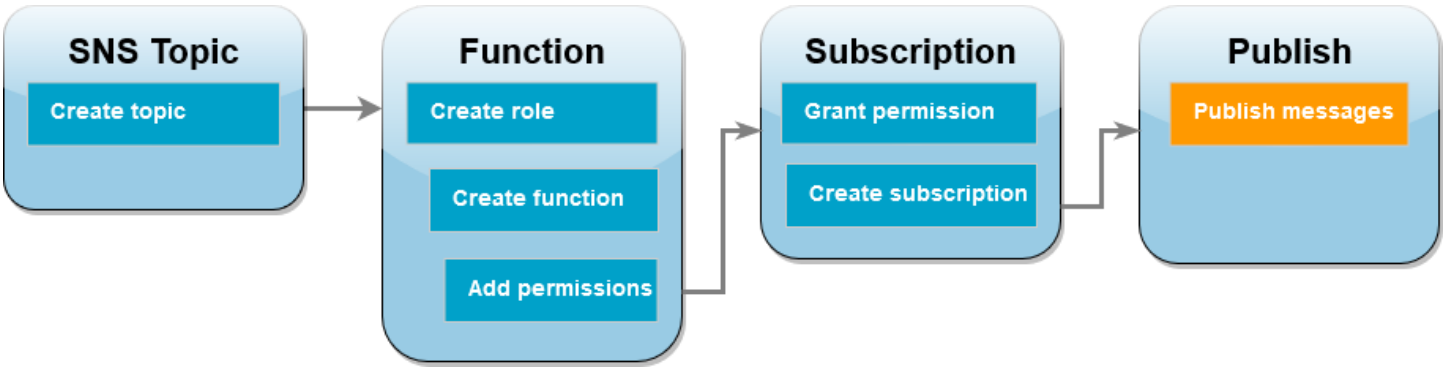
- 在帳戶 B 中，執行下列 AWS CLI 命令。使用您建立主題的預設區域，以及主題和 Lambda 函數的 ARN。

```
aws sns subscribe --protocol lambda \
    --region us-east-1 \
    --topic-arn arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda \
    --notification-endpoint arn:aws:lambda:us-
east-1:<AccountB_ID>:function:Function-With-SNS \
    --profile accountB
```

您應該會看到類似下列的輸出。

```
{
    "SubscriptionArn": "arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-
lambda:5d906xxxx-7c8x-45dx-a9dx-0484e31c98xx"
}
```

將訊息發佈至主題 (帳戶 A 和帳戶 B)



帳戶 B 中的 Lambda 函數已訂閱您帳戶 A 中的 Amazon SNS 主題後，是時候將訊息發佈至您的主題來測試您的設定了。若要確認 Amazon SNS 已叫用 Lambda 函數，您可以使用 CloudWatch 日誌來檢視函數的輸出。

若要將訊息發佈到您的主題並檢視函數的輸出

1. 輸入 `Hello World` 至文字檔，並儲存為 `message.txt`。
2. 從您儲存文字檔案的相同目錄中，在帳戶 A 中執行下列 AWS CLI 命令。 使用 ARN 為您自己的主題。

```
aws sns publish --message file://message.txt --subject Test \
    --topic-arn arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda \
    --profile accountA
```

   這將傳回具有唯一識別符的訊息 ID，表示 Amazon SNS 已接受訊息。接著，Amazon SNS 會嘗試將訊息傳遞給主題訂閱者。若要確認 Amazon SNS 已叫用 Lambda 函數，請使用 CloudWatch 日誌檢視函數的輸出：

3. 在帳戶 B 中，開啟 Amazon CloudWatch 主控台的日誌群組頁面。
4. 為函數 (`/aws/lambda/Function-With-SNS`) 選擇日誌群組名稱。
5. 選擇最新的日誌串流。
6. 如果您的函數有被正確調用，您將會看到類似下方的輸出，顯示您發佈到主題的訊息內容。

```
2023-07-31T21:42:51.250Z c1cba6b8-ade9-4380-aa32-d1a225da0e48 INFO Processed
 message Hello World
2023-07-31T21:42:51.250Z c1cba6b8-ade9-4380-aa32-d1a225da0e48 INFO done
```

清除您的資源

除非您想要保留為此教學課程建立的資源，否則您現在便可刪除。刪除您不再使用的 AWS 資源，您可以避免不必要的費用 AWS 帳戶。

在 帳戶 A 中，清除您的 Amazon SNS 主題。

刪除 Amazon SNS 主題

1. 在 Amazon SNS 主控台開啟 Topics (主題) 頁面。
2. 選擇您建立的主題。
3. 選擇 刪除 。
4. 在文字輸入欄位中輸入 **delete me**。
5. 選擇 刪除 。

在 帳戶 B 中，清除您的執行角色、Lambda 函數以及 Amazon SNS 訂閱。

刪除執行角色

1. 開啟 IAM 主控台中的 角色頁面 。
2. 選取您建立的執行角色。
3. 選擇 刪除 。
4. 在文字輸入欄位中輸入角色的名稱，然後選擇 刪除 。

若要刪除 Lambda 函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇您建立的函數。
3. 選擇 Actions (動作)、Delete (刪除)。
4. 在文字輸入欄位中輸入 **delete**，然後選擇 刪除 。

刪除 Amazon SNS 訂閱

1. 在 Amazon SNS 主控台開啟 Subscriptions (訂閱) 頁面。
2. 選取您建立的訂閱。
3. 選擇 刪除，刪除。

使用 AWS Lambda 函數的最佳做法

以下是使用 AWS Lambda的推薦最佳實務：

主題

如需 Lambda 應用程式最佳實務的詳細資訊，請參閱無伺服器園地中的 應用程式設計。您也可以聯絡您的 AWS 客戶團隊並要求架構審查。

函數程式碼

- 區隔 Lambda 處理常式與您的核心邏輯。能允許您製作更多可測單位的函式。在 Node.js 時，看起來會是這個樣子：

```
exports.myHandler = function(event, context, callback) {
 var foo = event.foo;
 var bar = event.bar;
 var result = MyLambdaFunction (foo, bar);

 callback(null, result);
}

function MyLambdaFunction (foo, bar) {
 // MyLambdaFunction logic here
}
```

- 請利用執行環境重新使用來改看函式的效能。在函式處理常式之外初始化 SDK 用戶端和資料庫連線，並在本機快取 /tmp 目錄中的靜態資產。由您函式的相同執行個體處理的後續叫用可以重複使用這些資源。這可藉由減少函數執行時間來節省成本。

若要避免叫用間洩漏潛在資料，請不要使用執行環境來儲存使用者資料、事件，或其他牽涉安全性的資訊。如果您的函式依賴無法存放在處理常式內記憶體中的可變狀態，請考慮為每個使用者建立個別函式或個別函式版本。

- 使用 Keep-Alive 指令維持持續連線的狀態。Lambda 會隨著時間的推移清除閒置連線。叫用函數時嘗試重複使用閒置連線將導致連線錯誤。若要維護持續連線，請使用與執行階段相關聯的 keep-alive (保持啟用) 指令。如需範例，請參閱在 Node.js 中重複使用 Keep-Alive 的連線。
- 使用環境變數將操作參數傳遞給您的函數。例如，如果您正在寫入到 Amazon S3 儲存貯體，而非對您正在寫入的儲存貯體名稱進行硬式編碼，請將儲存貯體名稱設定為環境變數。
- 控制函數部署套件內的相依性。 AWS Lambda 執行環境包含許多程式庫，例如 Node.js 和 Python 執行階段的 AWS SDK (完整清單可以在這裡找到:Lambda 執行期)。若要啟用最新的一組功能與安全更新，Lambda 會定期更新這些程式庫。這些更新可能會為您的 Lambda 函數行為帶來細微的變更。若要完全掌控您函式所使用的相依性，請利用部署套件封裝您的所有相依性。
- 將部署套件最小化至執行時間所必要的套件大小。這能減少您的部署套件被下載與呼叫前解壓縮的時間。對於以 Java 或 .NET Core 編寫的函數，請避免將整個 AWS SDK 庫作為部署包的一部分上傳。或者，選擇性倚賴取得您需要的軟體開發套件元件的模組 (例如 DynamoDB、Amazon S3 開發套件模組，以及 Lambda 核心程式庫)。
- 將相依性 `.jar` 檔案置於不同的 /lib 目錄，縮短 Lambda 解壓縮以 Java 撰寫之部署套件的時間。這比將您所有函式程式碼全部放入具大量 `.class` 檔案的單一 jar 更快速。如需說明，請參閱 使用 .zip 或 JAR 封存檔部署 Java Lambda 函數。
- 最小化依存項目的複雜性。偏好更簡易的框架，其可快速在執行環境啟動時載入。例如，偏好更簡易的 Java 相依性置入 (IoC) 架構如 Dagger 或 Guice，勝於複雜的架構如 Spring Framework。
- 避免在您的 Lambda 函數中使用遞迴程式碼，其中函數會自動呼叫自己，直到符合一些任意的標準。這會導致意外的函式呼叫量與升高的成本。若您不小心這樣做，當更新程式碼時，請立刻將函式的預留並行設為 `0`，以調節對函式的所有呼叫。
- 請勿在您的 Lambda 函數程式碼中使用未記錄的非公有 API。對於 AWS Lambda 受管執行階段，Lambda 會定期將安全性和功能更新套用至 Lambda 的內部 API。這些內部 API 更新可能是向後不相容的，這會導致意外結果，例如若您的函數依賴於這些非公有 API，則叫用失敗。請參閱 API 參考查看公開可用 API 的清單。
- 撰寫等冪程式碼。為函數撰寫等冪程式碼可確保採用相同方式來處理重複事件。程式碼應正確驗證事件並正常處理重複的事件。如需詳細資訊，請參閱 How do I make my Lambda function idempotent? (如何讓 Lambda 函數等冪？)。
- 避免使用 DNS 快取。Lambda 函數已快取 DNS 回應。如果您使用其他 DNS 快取，則可能會遇到連線逾時的情況。

此java.util.logging.Logger類別可間接啟用 JVM DNS 快取。若要覆寫預設設定，請在初始化之前將網路位址 .cache.ttl 設定為 0。 logger範例：

```
public class MyHandler {
  // first set TTL property
  static{
   java.security.Security.setProperty("networkaddress.cache.ttl" , "0");
  }
 // then instantiate logger
  var logger = org.apache.logging.log4j.LogManager.getLogger(MyHandler.class);
}
```

若要避免UnknownHostException失敗，建議您networkaddress.cache.negative.ttl將設定為 0。您可以使用AWS_LAMBDA_JAVA_NETWORKADDRESS_CACHE_NEGATIVE_TTL=0環境變數為 Lambda 函數設定此屬性。

停用 JVM DNS 快取並不會停用 Lambda 的受管理 DNS 快取。

函數組態

- 啟動您的 Lambda 函式的效能測試是確保您挑選最佳記憶體大小組態非常重要的一環。任何記憶體大小的增加能觸發相同的增加可用於函數的 CPU。函數的記憶體使用量是根據每次叫用來決定的，並且可以在 Amazon CloudWatch 中檢視。每一次呼叫會產生 REPORT: 項目，如下所示：

```
REPORT RequestId: 3604209a-e9a3-11e6-939a-754dd98c7be3 Duration: 12.34 ms Billed
 Duration: 100 ms Memory Size: 128 MB Max Memory Used: 18 MB
```

藉由分析 Max Memory Used: 欄位，您可以判斷您的函式是否需要更多記憶體，或是否過度佈建函式記憶體大小。

若要尋找適合您功能的記憶體配置，我們建議您使用開放原始碼 AWS Lambda 電源調整專案。如需詳細資訊，請參閱開啟AWS Lambda 電源調整 GitHub。

若要最佳化函數效能，我們也建議部署可充分利用進階向量延伸 2 (AVX2) 的程式庫。這可讓您處理繁重的工作負載，包括機器學習推論、媒體處理、高效能運算 (HPC)、科學模擬和財務建模。如需詳細資訊，請參閱使用 AVX2 建立更快的 AWS Lambda 函數。

- 對您的 Lambda 函數進行負載測試以判斷最佳的逾時值。分析您的函式執行多久時間是很重要的，如此您更能判斷任何相依服務的問題，這可能會增加超出預期的函式並行。當您的 Lambda 函數對可能無法處理 Lambda 擴展的資源發出網路呼叫時，這尤其重要。
- 設定 IAM 政策時，使用最嚴苛的許可。了解您的 Lambda 函數需要的資源與操作，並限制這些許可的執行角色。如需詳細資訊，請參閱 管理權限 AWS Lambda。
- 熟悉 Lambda 配額。承載大小、檔案描述項與 /tmp 空間，是在判斷執行時間資源限制時經常忽略的。
- 刪除您不再使用的 Lambda 函數。透過上述方法，未使用的函式不會對您的部署套件大小限制做不必要的計算。
- 如果您使用 Amazon Simple Queue Service 作為事件來源，請確保函數的預期叫用時間值不會超過佇列上的可見性逾時值。這同時適用於CreateFunction和UpdateFunctionConfiguration。
  - 在的情況下 CreateFunction，函數建立程序 AWS Lambda 將失敗。
  - 在的情況下 UpdateFunctionConfiguration，它可能會導致函數的重複調用。

功能擴充性

- 熟悉您的上游和下游輸送量限制。雖然 Lambda 函數可隨負載進行無縫擴展，但上游和下游相依性可能不具有相同的輸送量能力。如果您需要限制函數可擴展的高度，則可以在函數上配置保留並發性。
- 內置油門公差。如果您的同步函數因流量超過 Lambda 的擴展速率而遇到節流，您可以使用下列策略來改善節流容忍度：
  - 透過抖動來使用逾時、重試和輪詢。實作這些策略可以順利重試的呼叫，並協助確保 Lambda 可以在幾秒鐘內擴充，以將使用者限制降到最低。
  - 使用佈建的並行。佈建並行是 Lambda 配置給函數的預先初始化執行環境數目。Lambda 會在可用時使用佈建的並行處理傳入的要求。如果需要，Lambda 也可以將您的函數擴展到佈建的並行設定之外。設定佈建的並行功能會對您的帳戶產生額外費用 AWS 。

指標與警示

- 使用使用 Lambda 函數指標和 CloudWatch 警示，而不是從 Lambda 函數程式碼中建立或更新指標。這是追蹤 Lambda 函數運作狀態的更有效率方式，可讓您盡快在開發階段找出問題。例如，您可以根據預期的 Lambda 函數叫用持續時間來設定警示，以解決任何由函數程式碼導致的瓶頸或延遲。

- 利用您的記錄程式庫與 AWS Lambda 指標與維度，找出應用程式錯誤 (例如，ERR、ERROR、WARNING 等等)。
- 使用 AWS 成本異常偵測來偵測帳戶中的異常活動。成本異常偵測會使用機器學習來持續監控您的成本和使用量，同時盡可能減少誤報警示。「成本異常偵測」使用的資料可延遲長達 24 小時。 AWS Cost Explorer因此，使用後最多可能需要 24 小時才能偵測到異常。若要開始使用成本異常偵測，您必須先使用註冊 Cost Explorer。然後，存取成本異常偵測。

使用串流

- 使用不同的批次與記錄大小測試讓每個事件來源的輪詢頻率調整為函式可以多快完成作業。該CreateEventSourceMapping BatchSize 參數控制每次調用可以發送到函數的最大記錄數。更大的批次大小通常會更有效吸收更大集合的記錄的呼叫成本，增加您的傳輸量。

  Lambda 預設會在記錄可用時立即叫用函數。如果 Lambda 從事件來源中讀取的批次只有一筆記錄，Lambda 只會傳送一筆記錄至函數。為避免調用具有少量記錄的函數，您可設定批次間隔，請求事件來源緩衝記錄最長達五分鐘。調用函數之前，Lambda 會繼續從事件來源中讀取記錄，直到收集到完整批次、批次間隔到期或者批次達到 6 MB 的承載限制。如需詳細資訊，請參閱 批次處理行為。

  > **Warning**
  >
  > Lambda 事件來源對應至少處理每個事件一次，並且可能會重複處理記錄。為了避免與重複事件相關的潛在問題，我們強烈建議您將函數代碼設為冪等。若要深入了解，請參閱 AWS 知識中心如何讓 Lambda 函數具有冪等性。

- 新增碎片以增加 Kinesis 串流處理輸送量。Kinesis 串流由一個或多個碎片組成。Lambda 會使用最多一個並行呼叫，輪詢每個碎片。例如，若您的串流有 100 個有效碎片，則會有最多 100 個 Lambda 函數叫用正在同時執行。增加碎片的數量會直接增加最大並行 Lambda 函數叫用的數量，還會增加 Kinesis 串流處理輸送量。如果您正在增加 Kinesis 串流中碎片的數量，請確保為您的資料挑選良好的分割區索引鍵 (請參閱分割區索引鍵)，如此一來，相關的記錄會位於相同的碎片上，您的資料也會獲得妥善的分配。
- 使用 Amazon CloudWatch on IteratorAge 來判斷您的 Kinesis 串流是否正在處理中。例如，將 CloudWatch 鬧鐘的上限設定為 30000 (30 秒)。

安全最佳實務

- 使用監視與安全性最佳 AWS Lambda 作法相關的使用 AWS Security Hub。Security Hub 會透過安全控制來評估資源組態和安全標準，協助您遵守各種合規架構。如需使用 Security Hub 評估 Lambda 資源的詳細資訊，請參閱使用 AWS Security Hub 者指南中的AWS Lambda 控制項。
- 使用 Amazon Lambda 保護監控 GuardDuty Lambda 網路活動日誌 GuardDuty Lambda 保護可協助您識別潛在的安全威脅，當您的 AWS 帳戶. 例如，如果您的其中一個函數查詢與加密貨幣相關活動相關聯的 IP 地址。 GuardDuty 監控叫用 Lambda 函數時產生的網路活動記錄。若要進一步了解，請參閱 Amazon GuardDuty 使用者指南中的 Lambda 保護。

管理權限 AWS Lambda

您可以使用 AWS Identity and Access Management (IAM) 管理中的許可 AWS Lambda。使用 Lambda 函數時，需要考慮兩種主要的權限類別：

- Lambda 函數執行 API 動作和存取其他 AWS 資源所需的權限
- 其他 AWS 使用者和實體存取您的 Lambda 函數所需的權限

Lambda 函數通常需要存取其他 AWS 資源，並對這些資源執行各種 API 作業。例如，您可能有一個 Lambda 函數，可透過更新 Amazon DynamoDB 資料庫中的項目來回應事件。在這種情況下，您的函數需要訪問數據庫的權限，以及在該數據庫中放置或更新項目的權限。

您可以在稱為執行角色的特殊 IAM 角色中定義 Lambda 函數所需的許可。在此角色中，您可以附加政策，定義函數存取其他 AWS 資源所需的每個權限，以及從事件來源讀取。每個 Lambda 函數都必須具有執行角色。您的執行角色至少必須能夠存取 Amazon， CloudWatch 因為 Lambda 函數預設會 CloudWatch 記錄到日誌。您可以將AWSLambdaBasicExecutionRole受管理的原則附加至您的執行角色，以滿足此需求。

若要授與其他 AWS 帳戶、組織和服務存取 Lambda 資源的權限，您有以下幾個選項：

- 您可以使用以身分識別為基礎的政策授與其他使用者存取您的 Lambda 資源。以身分為基礎的政策可直接套用到使用者或與使用者相關連的群組和角色。
- 您可以使用以資源為基礎的政策授與其他帳戶和 AWS 服務存取 Lambda 資源的權限。當使用者嘗試存取 Lambda 資源時，Lambda 會同時考慮身分型政策 (針對使用者)，以及以資源型政策 (針對資源)。當亞馬遜簡單儲存 AWS 服務 (Amazon S3) 等服務呼叫您的 Lambda 函數時，Lambda 只會考慮以資源為基礎的政策。
- 您可以使用以屬性為基礎的存取控制 (ABAC) 模型來控制對 L ambda 函數的存取。使用 ABAC，您可以將標籤附加到 Lambda 函數，在特定 API 請求中傳遞標籤，或將它們附加到發出請求的 IAM 主體。在 IAM 政策的條件元素中指定相同的標籤，以控制函數存取。

在中 AWS，最佳做法是僅授與執行工作所需的權限 (最低權限權限)。若要在 Lambda 中實作此功能，我們建議從受AWS 管政策開始。您可以依原狀使用這些受管政策，或將其作為起點，撰寫限制程度更高的專屬政策。

為了協助您微調最低權限存取權限的許可，Lambda 提供了一些額外的條件，您可以在政策中加入。如需詳細資訊，請參閱 the section called “資源與條件”。

如需 IAM 的詳細資訊，請參閱《IAM 使用者指南》。

使用執行角色定義 Lambda 函數許可

Lambda 函數的執行角色是一種 AWS Identity and Access Management (IAM) 角色，可授與函數存取 AWS 服務和資源的權限。例如，您可以建立執行角色，該角色具有將日誌傳送到 Amazon CloudWatch 和將追蹤資料上傳到的權限 AWS X-Ray。本頁提供有關如何建立、檢視和管理 Lambda 函數執行角色的資訊。

當您調用函數時，Lambda 會自動擔任您的執行角色。您應該避免在函數代碼中手動調用`sts:AssumeRole`以承擔執行角色。如果您的使用案例請求角色擔任自己，則您必須將角色本身作為受信任主體包含在角色的信任政策中。如需如何修改角色信任政策的詳細資訊，請參閱《IAM 使用者指南》中的修改角色信任政策 (主控台)。

為了讓 Lambda 能夠正確承擔您的執行角色，角色的信任政策必須將 Lambda 服務主體 (`lambda.amazonaws.com`) 指定為受信任的服務。

主題

- 在 IAM 主控台中建立執行角色
- 建立和管理角色 AWS CLI
- 為 Lambda 執行角色授予最低權限存取權
- 檢視和更新執行角色中的權限
- 在執行角色中使用 AWS 受管理的原則
- 使用源函數 ARN 控制函數訪問行為

在 IAM 主控台中建立執行角色

根據預設，當您在 Lambda 主控台中建立函數時，Lambda 會建立具有最低許可的執行角色。具體而言，此執行角色包括受`AWSLambdaBasicExecutionRole`管政策，該政策為您的函數提供將事件記錄到 Amazon CloudWatch Logs 的基本許可。

您的函數通常需要額外的權限才能執行更有意義的工作。例如，您可能有一個 Lambda 函數，可透過更新 Amazon DynamoDB 資料庫中的項目來回應事件。您可以使用 IAM 主控台建立具有必要許可的執行角色。

若要在 IAM 主控台中建立執行角色

1. 在 IAM 主控台中開啟 角色頁面。

2. 選擇 建立角色。
3. 在受信任的實體類型下，選擇 AWS 服務。
4. 在 使用案例 下，選擇 Lambda。
5. 選擇下一步。
6. 選取您要附加至角色的 AWS 受管理原則。例如，如果您的函數需要存取 DynamoDB，請選取AWSLambdaDynamoDBExecutionRole受管理的原則。
7. 選擇下一步。
8. 輸入 角色名稱 ，然後選擇 建立角色 。

如需詳細指示，請參閱《IAM 使用者指南》中的為 AWS 服務建立角色 (主控台)。

創建執行角色後，將其附加到您的函數。在 Lambda 主控台中建立函數時，您可以將先前建立的任何執行角色附加至該函數。如果您要將新的執行角色附加至現有函數，請遵循中的步驟。

建立和管理角色 AWS CLI

若要使用 AWS Command Line Interface (AWS CLI) 建立執行角色，請使用create-role指令。使用此命令時，您可以指定內嵌信任政策。角色的信任政策授予指定主體擔任該角色的許可。在下列範例中，您授予 Lambda 服務主體擔任您的角色的許可。注意，JSON 字串中轉義引號的請求有所不同，這取決於您的 shell。

```
aws iam create-role --role-name lambda-ex --assume-role-policy-document '{"Version":
 "2012-10-17","Statement": [{ "Effect": "Allow", "Principal": {"Service":
 "lambda.amazonaws.com"}, "Action": "sts:AssumeRole"}]}'
```

您也可使用單獨的 JSON 檔案來定義角色的信任政策。在下列範例中，`trust-policy.json` 為當前目錄中的檔案。

Example trust-policy.json

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
```

```
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

```
aws iam create-role --role-name lambda-ex --assume-role-policy-document file://trust-policy.json
```

您應該會看到下列輸出：

```
{
    "Role": {
        "Path": "/",
        "RoleName": "lambda-ex",
        "RoleId": "AROAQFOXMPL6TZ6ITKWND",
        "Arn": "arn:aws:iam::123456789012:role/lambda-ex",
        "CreateDate": "2020-01-17T23:19:12Z",
        "AssumeRolePolicyDocument": {
            "Version": "2012-10-17",
            "Statement": [
                {
                    "Effect": "Allow",
                    "Principal": {
                        "Service": "lambda.amazonaws.com"
                    },
                    "Action": "sts:AssumeRole"
                }
            ]
        }
    }
}
```

使用 attach-policy-to-role 命令將許可新增至角色。下列命令會將`AWSLambdaBasicExecutionRole`受管理的原則新增至`lambda-ex`執行角色。

```
aws iam attach-role-policy --role-name lambda-ex --policy-arn arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
```

創建執行角色後，將其附加到您的函數。在 Lambda 主控台中建立函數時，您可以將先前建立的任何執行角色附加至該函數。如果您要將新的執行角色附加至現有函數，請遵循中的步驟。

為 Lambda 執行角色授予最低權限存取權

當您第一次為 Lambda 函數建立 IAM 角色時，有時可能會授予超出所需的許可。在生產環境中發佈您的函數之前，最佳實務是調整政策以僅包含必要的許可。如需詳細資訊，請參閱《IAM 使用者指南》中的套用最低權限許可。

使用 IAM Access Analyzer 來協助識別 IAM 執行角色政策的必要許可。IAM Access Analyzer 會檢閱您指定的日期範圍內的日 AWS CloudTrail 誌，並產生政策範本，並且僅使用該功能在該期間使用的許可來產生政策範本。您可以使用範本建立具有精細許可的受管政策，然後將其連接至 IAM 角色。如此一來，您只會針對特定使用案例授與角色與 AWS 資源互動所需的權限。

如需詳細資訊，請參閱《IAM 使用者指南》中的根據存取活動產生政策。

檢視和更新執行角色中的權限

本主題介紹如何檢視和更新函數的執行角色。

主題

- 檢視函數的執行角色
- 更新函數的執行角色

檢視函數的執行角色

若要檢視函數的執行角色，請使用 Lambda 主控台。

若要檢視函數的執行角色 (主控台)

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇函數的名稱。
3. 選擇 組態 ，然後選擇 許可 。
4. 在 [執行角色] 底下，您可以檢視目前作為函式執行角色使用的角色。為方便起見，您可以在 [資源摘要] 區段下檢視函數可存取的所有資源和動作。您也可以從下拉式清單中選擇服務，以查看與該服務相關的所有權限。

更新函數的執行角色

您可以隨時從函式的執行角色新增或移除許可，或將函式設定為使用不同的角色。如果您的函數需要存取任何其他服務或資源，您必須將必要的權限新增至執行角色。

當您向函數添加權限時，也對其代碼或配置執行簡單的更新。若您函數的執行中執行個體具有已過期的憑證，上述動作會強制停止並取代這些執行個體。

若要更新函數的執行角色，您可以使用 Lambda 主控台。

更新函數的執行角色（控制台）

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇函數的名稱。
3. 選擇 組態 ，然後選擇 許可 。
4. 在執行角色下，選擇編輯。
5. 如果您想要更新函數以使用不同的角色作為執行角色，請在 [現有角色] 下的下拉式功能表中選擇新角色。

   > **Note**
   >
   > 如果您想要更新現有執行角色中的許可，則只能在 AWS Identity and Access Management (IAM) 主控台中執行此操作。

   如果您想要建立新角色作為執行角色，請選擇 [執行角色] 下的 [從 AWS 原則範本建立新角色]。然後，在 [角色名稱] 底下輸入新角色的名稱，並在 [原則範本] 底下指定要附加至新角色的任何策略。
6. 選擇 Save (儲存)。

在執行角色中使用 AWS 受管理的原則

下列 AWS 受管政策提供使用 Lambda 功能所需的權限。

變更	描述	日期
AWSLambdaMSKExecutionRole-Lambda 添加了卡夫卡:DescribeClusterV2 權限這一政策.	`AWSLambdaMSKExecutionRole` 授予從 Amazon Managed Streaming for Apache Kafka (Amazon MSK) 叢集讀取和存取記錄、管理彈	2022 年 6 月 17 日

變更	描述	日期
	性網路界面 (ENI) 以及寫入日誌的權限。 CloudWatch	
AWSLambdaBasicExecutionRole— Lambda 開始追蹤此政策的變更。	`AWSLambdaBasicExecutionRole` 授予將日誌上傳至 CloudWatch 的許可。	2022 年 2 月 14 日
AWSLambdaDynamoDBExecutionRole— Lambda 開始追蹤此政策的變更。	`AWSLambdaDynamoDBExecutionRole` 授與從 Amazon DynamoDB 串流讀取記錄並寫入日誌的權限。 CloudWatch	2022 年 2 月 14 日
AWSLambdaKinesisExecutionRole— Lambda 開始追蹤此政策的變更。	`AWSLambdaKinesisExecutionRole` 授與從 Amazon Kinesis 資料串流讀取事件並寫入 CloudWatch 日誌的權限。	2022 年 2 月 14 日
AWSLambdaMSKExecutionRole— Lambda 開始追蹤此政策的變更。	`AWSLambdaMSKExecutionRole` 授予從 Amazon Managed Streaming for Apache Kafka (Amazon MSK) 叢集讀取和存取記錄、管理彈性網路界面 (ENI) 以及寫入日誌的權限。 CloudWatch	2022 年 2 月 14 日
AWSLambdaSQSQueueExecutionRole— Lambda 開始追蹤此政策的變更。	`AWSLambdaSQSQueueExecutionRole` 授與從 Amazon Simple Queue Service (Amazon SQS) 讀取訊息並寫入 CloudWatch 日誌的權限。	2022 年 2 月 14 日

變更	描述	日期
AWSLambdaVPCAccessExecutionRole— Lambda 開始追蹤此政策的變更。	AWSLambdaVPCAccessExecutionRole 授予在 Amazon VPC 內管理 ENI 並寫入日誌的許可。 CloudWatch	2022 年 2 月 14 日
AWSXRayDaemonWriteAccess— Lambda 開始追蹤此政策的變更。	AWSXRayDaemonWriteAccess 授予將追蹤資料上傳至 X-Ray 的許可。	2022 年 2 月 14 日
CloudWatchLambdaInsightsExecutionRolePolicy— Lambda 開始追蹤此政策的變更。	CloudWatchLambdaInsightsExecutionRolePolicy 授與將執行階段指標寫入 CloudWatch Lambda 見解的權限。	2022 年 2 月 14 日
亞馬遜 S3 ObjectLambdaExecutionRole 政策 — Lambda 開始追蹤此政策的變更。	AmazonS3ObjectLambdaExecutionRolePolicy 授予與亞馬遜簡單儲存服務 (Amazon S3) 物件 Lambda 互動並寫入 CloudWatch 日誌的許可。	2022 年 2 月 14 日

針對某些功能，Lambda 主控台會嘗試將遺漏的許可新增到您客戶受管政策中的執行角色。這些政策的數量可能會相當多。請在啟用功能前將相關受 AWS 管理˙政策新增到您的執行角色，以避免建立額外政策。

當您使用事件來源映射來調用函數時，Lambda 會使用執行角色來讀取事件資料。例如，Kinesis 的事件來源映射會從資料串流讀取事件，並將這些事件批次傳送到函數。

當服務在您的帳戶中擔任角色時，您可以在角色信任政策中包含 aws:SourceAccount 和 aws:SourceArn 全域條件內容金鑰，以將角色的存取限制為僅由預期資源產生的請求。如需詳細資訊，請參閱 AWS Security Token Service的預防跨服務混淆代理人。

除了 AWS 受管政策之外，Lambda 主控台還提供範本，用於建立具有其他使用案例權限的自訂原則。當您在 Lambda 主控台中建立函數時，您可以選擇使用來自一個或多個範本的許可建立新的執行角

色。當您從藍圖建立函式時，或是您設定需要存取其他服務的選項時，也會自動套用這些範本。範例範本可在本指南的GitHub儲存庫中找到。

使用源函數 ARN 控制函數訪問行為

Lambda 函數程式碼通常會向其他 AWS 服務發出 API 要求。若要提出這些請求，Lambda 會擔任函數的執行角色，以產生一組暫時性憑證。這些憑證在函數調用期間可當成環境變數使用。使用 AWS SDK 時，您不需要直接在程式碼中提供 SDK 的憑證。根據預設，憑證提供者鏈會依序檢查您可以設定憑證的每個位置，並選取第一個可用的位置，通常是環境變數 (`AWS_ACCESS_KEY_ID`、`AWS_SECRET_ACCESS_KEY` 和 `AWS_SESSION_TOKEN`)。

如果請求是來自執行環境中的 AWS API 請求，則 Lambda 會將來源函數 ARN 插入認證內容中。針對 Lambda 代表您在執行環境外發出的下列 AWS API 請求，Lambda 也會插入來源函數 ARN：

服務	動作	原因
CloudWatch 日誌	`CreateLogGroup`，`CreateLogStream`，`PutLogEvents`	將記錄檔儲存到 CloudWatch 記錄檔記錄群組
X-Ray	`PutTraceSegments`	將追蹤資料傳送到 X-Ray
Amazon EFS	`ClientMount`	將函數連接到 Amazon Elastic File System (Amazon EFS) 檔案系統

Lambda 代表您使用相同執行角色在執行環境之外進行的其他 AWS API 呼叫不包含來源函數 ARN。這類執行環境外的 API 呼叫範例包括：

- 呼叫 AWS Key Management Service (AWS KMS) 以自動加密和解密您的環境變數。
- 對 Amazon Elastic Compute Cloud (Amazon EC2) 發出的呼叫，目的是為已啟用 VPC 的函數建立彈性網絡介面 (ENI)。
- 呼叫 AWS 服務 (例如 Amazon Simple Queue Service (Amazon SQS)，以便從設定為事件來源映射的事件來源讀取。

您可以使用憑證內容中的來源函數 ARN，驗證對資源的呼叫是否來自特定 Lambda 函數的程式碼。若要驗證這一點，請在 IAM 身分型政策或服務控制政策 (SCP) 中使用`lambda:SourceFunctionArn`條件金鑰。

> **Note**
>
> 您無法在資源型政策中使用 `lambda:SourceFunctionArn` 條件索引鍵。

在您的身分型政策或 SCP 中使用此條件，您可以針對函數程式碼對其他服務執行的 API 動作實作安全性控制。 AWS 此操作具有一些關鍵的安全保護作用，例如，協助您識別憑證洩漏的來源。

> **Note**
>
> `lambda:SourceFunctionArn` 條件索引鍵與 `lambda:FunctionArn` 和 `aws:SourceArn` 條件索引鍵不同。`lambda:FunctionArn` 條件索引鍵僅適用於事件來源映射，會協助定義您的事件來源可以呼叫哪些函數。`aws:SourceArn`條件金鑰僅適用於 Lambda 函數作為目標資源的政策，並協助定義哪些其他 AWS 服務和資源可以叫用該函數。`lambda:SourceFunctionArn`條件金鑰可套用至任何以身分識別為基礎的政策或 SCP，以定義具有對其他資源進行特定 AWS API 呼叫之權限的特定 Lambda 函數。

若要在政策中使用 `lambda:SourceFunctionArn`，請將其作為任何 ARN 條件運算子的條件，納入到政策中。索引鍵的值必須為有效的 ARN。

例如，假設您的 Lambda 函數程式碼會進行 `s3:PutObject` 呼叫，該呼叫以特定 Amazon S3 儲存貯體為目標。您可能希望僅允許一個特定 Lambda 函數擁有對該儲存貯體的 `s3:PutObject` 存取權。在此情況下，您函數的執行角色應連接有類似以下的政策：

Example 授予特定 Lambda 函數存取權以存取 Amazon S3 資源的政策

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ExampleSourceFunctionArn",
            "Effect": "Allow",
            "Action": "s3:PutObject",
            "Resource": "arn:aws:s3:::lambda_bucket/*",
            "Condition": {
```

```
                "ArnEquals": {
                    "lambda:SourceFunctionArn": "arn:aws:lambda:us-
east-1:123456789012:function:source_lambda"
                }
            }
        }
    ]
}
```

此政策僅在來源為具有 ARN `arn:aws:lambda:us-east-1:123456789012:function:source_lambda` 的 Lambda 函數時允許 `s3:PutObject` 存取權。此政策不允許對任何其他呼叫身分的 `s3:PutObject` 存取權。即使不同函數或實體以相同執行角色進行 `s3:PutObject` 呼叫也是如此。

> **Note**
>
> `lambda:SourceFunctionARN` 條件索引鍵不支援 Lambda 函數版本或函數別名。如果您將 ARN 用於特定函數版本或別名，則您的函數將無權執行您指定的動作。務必為您的函數使用不符資格的 ARN，而不使用版本或別名後綴。

您也可以`lambda:SourceFunctionArn`在 SCP 中使用。例如，假設您要僅讓單一 Lambda 函數的程式碼或來自 Amazon 虛擬私有雲端 (VPC) 的呼叫能存取儲存貯體。以下 SCP 對此進行說明。

Example 在特定條件下拒絕對 Amazon S3 的存取的政策

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "s3:*"
            ],
            "Resource": "arn:aws:s3:::lambda_bucket/*",
            "Effect": "Deny",
            "Condition": {
                "StringNotEqualsIfExists": {
                    "aws:SourceVpc": [
                        "vpc-12345678"
                    ]
                }
```

```
            }
        },
        {
            "Action": [
                "s3:*"
            ],
            "Resource": "arn:aws:s3:::lambda_bucket/*",
            "Effect": "Deny",
            "Condition": {
                "ArnNotEqualsIfExists": {
                    "lambda:SourceFunctionArn": "arn:aws:lambda:us-
east-1:123456789012:function:source_lambda"
                }
            }
        }
    ]
}
```

除非 S3 動作來自具有 ARN `arn:aws:lambda:*:123456789012:function:source_lambda` 的特定 Lambda 函數，或是其來自指定的 VPC，否則此政策會拒絕所有 S3 動作。僅在請求中包含 `aws:SourceVpc` 索引鍵時，`StringNotEqualsIfExists` 運算子才會告訴 IAM 處理此條件。同樣地，僅在 `lambda:SourceFunctionArn` 存在時，IAM 才會考慮 `ArnNotEqualsIfExists` 運算子。

授與其他 AWS 實體存取您的 Lambda 函數

若要授與其他 AWS 帳戶、組織和服務存取 Lambda 資源的權限，您有以下幾個選項：

- 您可以使用以身分識別為基礎的政策授與其他使用者存取您的 Lambda 資源。以身分為基礎的政策可直接套用到使用者或與使用者相關連的群組和角色。
- 您可以使用以資源為基礎的政策授與其他帳戶和 AWS 服務存取 Lambda 資源的權限。當使用者嘗試存取 Lambda 資源時，Lambda 會同時考慮身分型政策 (針對使用者)，以及以資源型政策 (針對資源)。當亞馬遜簡單儲存 AWS 服務 (Amazon S3) 之類的服務呼叫您的 Lambda 函數時，Lambda 只會考慮以資源為基礎的政策。
- 您可以使用以屬性為基礎的存取控制 (ABAC) 模型來控制對 L ambda 函數的存取。使用 ABAC，您可以將標籤附加到 Lambda 函數，在特定 API 請求中傳遞標籤，或將它們附加到發出請求的 IAM 主體。在 IAM 政策的條件元素中指定相同的標籤，以控制函數存取。

為了協助您微調最低權限存取權限的許可，Lambda 提供了一些您可以在政策中加入的其他條件。如需更多詳細資訊，請參閱 the section called “資源與條件”。

在 Lambda 中使用以身分識別為基礎的 IAM 政策

您可以在 AWS Identity and Access Management (IAM) 中使用以身分識別為基礎的政策授與帳戶中的使用者存取 Lambda。以身分為基礎的政策可直接套用到使用者或與使用者相關連的群組和角色。您可以授予另一個帳戶的使用者許可，讓他們可以擔任您帳戶中的角色並存取 Lambda 資源。本頁面展示如何將身分識別型政策用於函數開發的範例。

Lambda 提供的 AWS 受管政策可授予 Lambda API 動作的存取權，並在某些情況下存取用於開發和管理 Lambda 資源的其他 AWS 服務。Lambda 會視需要更新這些受管政策，來確保您的使用者可在新功能發行時進行存取。

- AWSLambda_FullAccess— 授予 Lambda 動作和其他用於開發和維護 Lambda 資源的 AWS 服務的完整存取權。此原則是藉由設定先前原則的範圍來建立。AWSLambdaFullAccess
- AWSLambda_ReadOnlyAccess— 授予 Lambda 資源的唯讀存取權。此原則是藉由設定先前原則的範圍來建立。AWSLambdaReadOnlyAccess
- AWSLambdaRole— 授與叫用 Lambda 函數的權限。

AWS 受管政策授予 API 動作的權限，而不會限制使用者可以修改的 Lambda 函數或層。如需進行更精細的控制，您可以建立自己的政策，來限制使用者的許可範圍。

章節

撰寫將函數授與使用者權限的範例原則

使用身分型政策以允許使用者對 Lambda 函數執行操作。

> **Note**
>
> 對於定義為容器映像的函數，存取映像的使用者許可必須在 Amazon Elastic Container Registry 中設定。如需範例，請參閱 Amazon ECR 許可。

以下顯示具受限範圍之許可政策的範例。這可讓使用者建立和管理以指定字首 (`intern-`) 命名並以指定執行角色設定的 Lambda 函數。

Example 函數開發政策

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ReadOnlyPermissions",
            "Effect": "Allow",
            "Action": [
                "lambda:GetAccountSettings",
                "lambda:GetEventSourceMapping",
                "lambda:GetFunction",
                "lambda:GetFunctionConfiguration",
                "lambda:GetFunctionCodeSigningConfig",
                "lambda:GetFunctionConcurrency",
                "lambda:ListEventSourceMappings",
                "lambda:ListFunctions",
                "lambda:ListTags",
                "iam:ListRoles"
            ],
            "Resource": "*"
        },
```

```
        {
            "Sid": "DevelopFunctions",
            "Effect": "Allow",
            "NotAction": [
                "lambda:AddPermission",
                "lambda:PutFunctionConcurrency"
            ],
            "Resource": "arn:aws:lambda:*:*:function:intern-*"
        },
        {
            "Sid": "DevelopEventSourceMappings",
            "Effect": "Allow",
            "Action": [
                "lambda:DeleteEventSourceMapping",
                "lambda:UpdateEventSourceMapping",
                "lambda:CreateEventSourceMapping"
            ],
            "Resource": "*",
            "Condition": {
                "StringLike": {
                    "lambda:FunctionArn": "arn:aws:lambda:*:*:function:intern-*"
                }
            }
        },
        {
            "Sid": "PassExecutionRole",
            "Effect": "Allow",
            "Action": [
                "iam:ListRolePolicies",
                "iam:ListAttachedRolePolicies",
                "iam:GetRole",
                "iam:GetRolePolicy",
                "iam:PassRole",
                "iam:SimulatePrincipalPolicy"
            ],
            "Resource": "arn:aws:iam::*:role/intern-lambda-execution-role"
        },
        {
            "Sid": "ViewLogs",
            "Effect": "Allow",
            "Action": [
                "logs:*"
            ],
            "Resource": "arn:aws:logs:*:*:log-group:/aws/lambda/intern-*"
```

```
        }
    ]
}
```

會根據許可支援的資源和條件來將此政策中的許可組織為陳述式。

- ReadOnlyPermissions - Lambda 主控台會在您瀏覽和檢視函數時使用這些許可。他們不支援資源模式或條件。

```
            "Action": [
                "lambda:GetAccountSettings",
                "lambda:GetEventSourceMapping",
                "lambda:GetFunction",
                "lambda:GetFunctionConfiguration",
                "lambda:GetFunctionCodeSigningConfig",
                "lambda:GetFunctionConcurrency",
                "lambda:ListEventSourceMappings",
                "lambda:ListFunctions",
                "lambda:ListTags",
                "iam:ListRoles"
            ],
            "Resource": "*"
```

- DevelopFunctions - 使用會對字首為 intern- (除了 AddPermission 和 PutFunctionConcurrency) 的函數進行操作的 Lambda 動作。AddPermission 會在該函數中修改以資源為基礎的政策，並且可能會產生安全隱憂。PutFunctionConcurrency 會為某函數保留擴展容量並可從其他函數取走容量。

```
            "NotAction": [
                "lambda:AddPermission",
                "lambda:PutFunctionConcurrency"
            ],
            "Resource": "arn:aws:lambda:*:*:function:intern-*"
```

- DevelopEventSourceMappings – 在字首為 intern- 的函數上管理事件來源映射。這些動作會在事件來源映射上操作，但您可以使用條件來依函數進行限制。

```
            "Action": [
                "lambda:DeleteEventSourceMapping",
```

```
            "lambda:UpdateEventSourceMapping",
            "lambda:CreateEventSourceMapping"
        ],
        "Resource": "*",
        "Condition": {
            "StringLike": {
                "lambda:FunctionArn": "arn:aws:lambda:*:*:function:intern-*"
            }
        }
```

- PassExecutionRole - 檢視並僅傳遞名為 intern-lambda-execution-role 的角色，必須由具有 IAM 許可的使用者建立和管理該角色。當您將執行角色指派至函數時就會用到 PassRole。

```
        "Action": [
            "iam:ListRolePolicies",
            "iam:ListAttachedRolePolicies",
            "iam:GetRole",
            "iam:GetRolePolicy",
            "iam:PassRole",
            "iam:SimulatePrincipalPolicy"
        ],
        "Resource": "arn:aws:iam::*:role/intern-lambda-execution-role"
```

- ViewLogs— 使用 CloudWatch 記錄檢視前綴為之函數的記錄。intern-

```
        "Action": [
            "logs:*"
        ],
        "Resource": "arn:aws:logs:*:*:log-group:/aws/lambda/intern-*"
```

此政策可讓使用者開始使用 Lambda，而不需使其他使用者的資源承受風險。它不允許使用者設定要由其他服務觸發或呼叫其他 AWS 服務的函數，這需要更廣泛的 IAM 許可。它也不包括對不支持有限範圍政策的服務（例如 CloudWatch 和 X-Ray）的許可。對這些服務使用唯讀政策來為使用者提供指標和追蹤資料的存取。

為函數配置觸發器時，您需要訪問才能使用調用函數的 AWS 服務。例如，若要設定 Amazon S3 觸發，您需要使用 Amazon S3 動作的許可來管理儲存貯體通知。其中許多權限都包含在AWSLambdaFullAccess受管理的策略中。本指南的GitHub儲存庫提供範例原則。

撰寫授與使用層權限的範例原則

下列政策授予使用者建立 layer 以及搭配函數使用的許可。資源模式允許用戶在任何 AWS 區域和任何圖層版本中工作，只要圖層的名稱開頭為`test-`。

Example 圖層開發政策

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "PublishLayers",
            "Effect": "Allow",
            "Action": [
                "lambda:PublishLayerVersion"
            ],
            "Resource": "arn:aws:lambda:*:*:layer:test-*"
        },
        {
            "Sid": "ManageLayerVersions",
            "Effect": "Allow",
            "Action": [
                "lambda:GetLayerVersion",
                "lambda:DeleteLayerVersion"
            ],
            "Resource": "arn:aws:lambda:*:*:layer:test-*:*"
        }
    ]
}
```

您也可以使用 `lambda:Layer` 條件來在函數建立和設定期間強制執行 layer 的使用。例如，您可以防止使用者使用其他帳戶發佈的 layer。下列政策會將條件新增至 `CreateFunction`，且 `UpdateFunctionConfiguration` 動作需要來自帳戶 `123456789012` 指定的任何 layer。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ConfigureFunctions",
            "Effect": "Allow",
            "Action": [
                "lambda:CreateFunction",
```

```
                "lambda:UpdateFunctionConfiguration"
            ],
            "Resource": "*",
            "Condition": {
                "ForAllValues:StringLike": {
                    "lambda:Layer": [
                        "arn:aws:lambda:*:123456789012:layer:*:*"
                    ]
                }
            }
        }
    ]
}
```

若要確保條件是否套用，請確認沒有其他陳述式授予使用者這些動作的許可。

使用身分型原則實作跨帳戶存取

您可以將上述任何政策和陳述式套用到角色，您在之後可以將該角色與其他帳戶共用來讓該角色存取 Lambda 資源。與使用者不同，角色沒有身分驗證所需的登入資料。相反地，角色擁有信任政策，可指定誰可擔任角色並使用其許可。

您可以使用跨帳戶角色來提供您信任之帳戶對 Lambda 動作和資源的存取。如果您只想要授予許可來調用函數或使用 layer，請改用以資源為基礎的政策。

如需詳細資訊，請參閱《IAM 使用者指南》中的 IAM 角色。

使用 Lambda 中以資源為基礎的政策

Lambda 支援對 Lambda 函數和層使用資源型許可政策。以資源為基礎的策略可讓您以每個資源為基礎，將使用權限授與其他 AWS 帳號或組織。您還可以使用以資源為基礎的策略來允許 AWS 服務代表您調用您的函數。

針對 Lambda 函數，您可以授予帳戶叫用或管理函數的許可。您也可以使用單一資源型政策授予許可給在 AWS Organizations中的整個組織。您也可以使用以資源為基礎的政策，將調用權限授予 AWS 服務的調用權限，該服務會調用函數以響應您帳戶中的活動。

檢視函式以資源為基礎的政策

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。

3. 選擇 Configuration (組態)，然後選擇 Permissions (權限)。
4. 向下捲動至 Resource-based policy (資源型政策)，然後選擇 View policy document (檢視政策文件)。以資源為基礎的策略會顯示當其他帳戶或 AWS 服務嘗試存取函數時所套用的權限。下列範例顯示的陳述式允許 Amazon S3 針對帳戶 123456789012 中名稱為 DOC-EXAMPLE-BUCKET 的儲存貯體，叫用名為 my-function 的函數。

   Example 以資源為基礎政策

```
{
    "Version": "2012-10-17",
    "Id": "default",
    "Statement": [
        {
            "Sid": "lambda-allow-s3-my-function",
            "Effect": "Allow",
            "Principal": {
              "Service": "s3.amazonaws.com"
            },
            "Action": "lambda:InvokeFunction",
            "Resource":  "arn:aws:lambda:us-east-2:123456789012:function:my-
function",
            "Condition": {
              "StringEquals": {
                "AWS:SourceAccount": "123456789012"
              },
              "ArnLike": {
                "AWS:SourceArn": "arn:aws:s3:::DOC-EXAMPLE-BUCKET"
              }
            }
        }
     ]
}
```

對於 Lambda 層級，您只能在特定層級版本上使用以資源為基礎的政策，而不是在整個層級上使用。除了會將許可授予至單一帳戶或多個帳戶的政策，針對 layer，您也可以將許可授予至組織中的所有帳戶。

> **Note**
>
> 您只能在AddPermission和 AddLayerVersionPermissionAPI 動作範圍內更新 Lambda 資源以資源為基礎的政策。目前，您無法使用 JSON 為 Lambda 資源撰寫政策，或使用未映射至那些動作參數的條件。

以資源為基礎的政策適用於單一函式、版本、別名或 layer 版本。他們會將許可授予一或多個服務和帳戶。針對您想要多個資源之存取或使用以資源為基礎的政策不支援之 API 動作的信任帳戶，您可以使用跨帳戶角色。

主題

支援的 API 動作

下列 Lambda API 動作支援資源型政策：

- CreateAlias
- DeleteAlias
- DeleteFunction
- DeleteFunction并发性
- DeleteFunctionEventInvokeConfig
- DeleteProvisionedConcurrencyConfig
- GetAlias
- GetFunction
- GetFunction并发性
- GetFunction配置

- GetFunctionEventInvokeConfig
- GetPolicy
- GetProvisionedConcurrencyConfig
- Invoke
- ListAliases
- ListFunctionEventInvoke配置
- ListProvisionedConcurrencyConfigs
- ListTags
- ListVersionsByFunction
- PublishVersion
- PutFunction并发性
- PutFunctionEventInvokeConfig
- PutProvisionedConcurrencyConfig
- TagResource
- UntagResource
- UpdateAlias
- UpdateFunction代碼
- UpdateFunctionEventInvokeConfig

授與 AWS 服務的功能存取權

當您使用 AWS 服務來叫用函數時，您可以在資源型政策的陳述式中授與權限。您可以將陳述式套用至要叫用或管理的整個函數，或將陳述式限制在單一版本或別名。

> **Note**
>
> 當您透過Lambda 主控台將觸發新增至函數時，主控台會更新函數的以資源為基礎的政策來允許服務進行叫用。若要將許可授予無法在 Lambda 主控台中使用的其他帳戶或服務，您可使用 AWS CLI。

使用 `add-permission` 命令新增陳述式。最簡單的以資源為基礎的政策陳述式可讓服務叫用函式。下列命令可授予 Amazon SNS 叫用名為 `my-function` 之函數的許可。

```
aws lambda add-permission --function-name my-function --action lambda:InvokeFunction --
statement-id sns \
    --principal sns.amazonaws.com --output text
```

您應該會看到下列輸出：

```
{"Sid":"sns","Effect":"Allow","Principal":
{"Service":"sns.amazonaws.com"},"Action":"lambda:InvokeFunction","Resource":"arn:aws:lambda:us-
east-2:123456789012:function:my-function"}
```

這可讓 Amazon SNS 叫用函數的 `lambda:Invoke` API，但不會限制會觸發叫用的 Amazon SNS 主題。若要確保您的函式只會被特定的資源叫用，請使用 `source-arn` 選項來指定資源的 Amazon Resource Name (ARN)。下列命令僅允許 Amazon SNS 為名為 `my-topic` 的主題訂閱叫用函數。

```
aws lambda add-permission --function-name my-function --action lambda:InvokeFunction --
statement-id sns-my-topic \
    --principal sns.amazonaws.com --source-arn arn:aws:sns:us-east-2:123456789012:my-
topic
```

有些服務可在不同的帳戶中叫用函式。如果您指定的來源 ARN 中有帳戶 ID，您可以放心。然而，對於 Amazon S3，來源是一種儲存貯體，其 ARN 沒有帳戶 ID。您可以刪除該儲存貯體，且其他帳戶可使用相同的名稱建立儲存貯體。透過您的帳戶 ID 使用 `source-account` 選項來確保只有帳戶中的資源可叫用該函式。

```
aws lambda add-permission --function-name my-function --action lambda:InvokeFunction --
statement-id s3-account \
    --principal s3.amazonaws.com --source-arn arn:aws:s3:::DOC-EXAMPLE-BUCKET --source-
account 123456789012
```

授予組織對函數的存取

若要將權限授與中的組織 AWS Organizations，請將組織 ID 指定為`principal-org-id`。下列AddPermission AWS CLI 命令會將呼叫存取權授與組織`o-a1b2c3d4e5f`中的所有使用者。

```
aws lambda add-permission --function-name example \
    --statement-id PrincipalOrgIDExample --action lambda:InvokeFunction \
    --principal * --principal-org-id o-a1b2c3d4e5f
```

> **Note**
>
> 在此命令中，Principal 是 *。這意味著組織 o-a1b2c3d4e5f 中的所有使用者都會取得函數叫用許可。如果將 AWS 帳戶或角色指定為Principal，則只有該主參與者會取得函數叫用權限，但前提是它們也是o-a1b2c3d4e5f組織的一部分。

此命令建立類似以下的資源型政策：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "PrincipalOrgIDExample",
            "Effect": "Allow",
            "Principal": "*",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:example",
            "Condition": {
                "StringEquals": {
                    "aws:PrincipalOrgID": "o-a1b2c3d4e5f"
                }
            }
        }
    ]
}
```

如需詳細資訊，請參閱 AWS Identity and Access Management 使用者指南中的 aws: PrincipalOrg ID。

授予函式對其他帳戶的存取

若要授與其他 AWS 帳戶的權限，請將帳號 ID 指定為principal. 以下範例會授予帳戶 111122223333 以 prod 別名叫用 my-function 的許可。

```
aws lambda add-permission --function-name my-function:prod --statement-id xaccount --
action lambda:InvokeFunction \
    --principal 111122223333 --output text
```

您應該會看到下列輸出：

```
{"Sid":"xaccount","Effect":"Allow","Principal":
{"AWS":"arn:aws:iam::111122223333:root"},"Action":"lambda:InvokeFunction","Resource":"arn:aws:l
east-2:123456789012:function:my-function"}
```

以資源為基礎的政策會授予其他帳戶存取函數的許可，但不允許該帳戶中的使用者超過其許可。另一個帳戶中的使用者必須具有相對應的使用者許可才能使用 Lambda API。

若要限制另一個帳戶中使用者或角色的存取，請指定身分的完整 ARN 作為主體。例如 `arn:aws:iam::123456789012:user/developer`。

別名會限制其他帳戶可叫用的版本。這需要其他帳戶在函式 ARN 中包含別名。

```
aws lambda invoke --function-name arn:aws:lambda:us-west-2:123456789012:function:my-
function:prod out
```

您應該會看到下列輸出：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "1"
}
```

然後，函數擁有者可以更新別名以指向新版本，而無需呼叫者變更他們叫用函數的方式。這可確保另一個帳戶不需要變更其程式碼來使用新版本，而且它只具有叫用與別名關聯之函數版本的許可。

您可以為在現有函式上操作之大部分 API 動作授予跨帳戶存取。例如，您可以授予對 `lambda:ListAliases` 的存取，來讓帳戶取得別名的清單，或 `lambda:GetFunction` 讓他們下載函式程式碼。個別新增每個許可，或使用 `lambda:*` 來授予對指定函式進行所有動作的存取。

若要授予其他帳戶多個動作或是不在函數上操作之動作的許可，我們建議您請使用 IAM 角色。

授予 Layer 對其他帳戶的存取

如欲將 layer 使用許可授予其他帳戶，請透過 add-layer-version-permission 命令將陳述式新增至 layer 版本的許可政策。在各陳述式中，您可將許可授予單一帳戶、所有帳戶或某個組織。

以下列範例會授予帳戶 111122223333 存取 `bash-runtime` layer 第 2 版的許可。

```
aws lambda add-layer-version-permission --layer-name bash-runtime --statement-id
 xaccount \
```

```
--action lambda:GetLayerVersion --principal 111122223333 --version-number 2 --output
 text
```

您應該會看到類似下列的輸出：

```
e210ffdc-e901-43b0-824b-5fcd0dd26d16    {"Sid":"xaccount","Effect":"Allow","Principal":
{"AWS":"arn:aws:iam::111122223333:root"},"Action":"lambda:GetLayerVersion","Resource":"arn:aws:
east-1:123456789012:layer:bash-runtime:2"}
```

許可僅適用於單一層版本。每次建立新的層版本時均需重複此程序。

若要將許可授予組織中的所有帳戶，請使用 organization-id 選項。以下範例授予組織內的所有帳戶使用第 3 版 Layer 的許可。

```
aws lambda add-layer-version-permission --layer-name my-layer \
    --statement-id engineering-org --version-number 3 --principal '*' \
    --action lambda:GetLayerVersion --organization-id o-t194hfs8cz --output text
```

您應該會看到下列輸出：

```
b0cd9796-d4eb-4564-939f-de7fe0b42236    {"Sid":"engineering-
org","Effect":"Allow","Principal":"*","Action":"lambda:GetLayerVersion","Resource":"arn:aws:lam
east-2:123456789012:layer:my-layer:3","Condition":{"StringEquals":
{"aws:PrincipalOrgID":"o-t194hfs8cz"}}}"
```

若要將權限授與所有 AWS 帳戶，請使*用主參與者，並省略組織 ID。如有多個帳戶或組織，您需要加入多個陳述式。

清理以資源為基礎的政策

若要檢查函式的以資源為基礎的政策，請使用 get-policy 命令。

```
aws lambda get-policy --function-name my-function --output text
```

您應該會看到下列輸出：

```
{"Version":"2012-10-17","Id":"default","Statement":
[{"Sid":"sns","Effect":"Allow","Principal":
{"Service":"s3.amazonaws.com"},"Action":"lambda:InvokeFunction","Resource":"arn:aws:lambda:us-
east-2:123456789012:function:my-function","Condition":{"ArnLike":
```

```
{"AWS:SourceArn":"arn:aws:sns:us-east-2:123456789012:lambda*"}}}]}      7c681fc9-
b791-4e91-acdf-eb847fdaa0f0
```

針對版本和別名，在函式名稱的後方附加版本編號或別名。

```
aws lambda get-policy --function-name my-function:PROD
```

若要將許可從函式中移除，請使用 remove-permission。

```
aws lambda remove-permission --function-name example --statement-id sns
```

使用 get-layer-version-policy 指令檢視層級上的許可。

```
aws lambda get-layer-version-policy --layer-name my-layer --version-number 3 --output
 text
```

您應該會看到下列輸出：

```
b0cd9796-d4eb-4564-939f-de7fe0b42236    {"Sid":"engineering-
org","Effect":"Allow","Principal":"*","Action":"lambda:GetLayerVersion","Resource":"arn:aws:lam
west-2:123456789012:layer:my-layer:3","Condition":{"StringEquals":
{"aws:PrincipalOrgID":"o-t194hfs8cz"}}}"
```

使用 remove-layer-version-permission 以從策略中移除陳述式。

```
aws lambda remove-layer-version-permission --layer-name my-layer --version-number 3 --
statement-id engineering-org
```

在 Lambda 中使用基於屬性的存取控制

搭配屬性型存取控制 (ABAC)，您可以使用標籤來控制對 Lambda 函數的存取。您可以將標籤附加到 Lambda 函數，在特定 API 請求中傳遞標籤，或將它們附加到發出請求的 AWS Identity and Access Management (IAM) 主體。有關如何 AWS 授予以屬性為基礎的存取權的詳細資訊，請參閱 IAM 使用者指南中的使用標籤控制 AWS 資源的存取。

您可以使用 ABAC 來授予最低權限，無需在 IAM 政策中指定 Amazon Resource Name (ARN) 或 ARN 模式。可以在 IAM 政策 的條件元素中指定標籤，來控制存取。由於您無需在建立新函數時更新 IAM 政策，因此使用 ABAC 可以更輕鬆地擴展。可以將標籤新增至新函數來控制存取。

在 Lambda 中，標籤會在函數層級上工作。層、程式碼簽署組態或事件來源映射不支援標籤。在您標記函數時，這些標籤會套用至與該函數相關聯的所有版本和別名。如需有關如何標記函數的詳細資訊，請參閱在 Lambda 函數上使用標籤。

您可以使用以下條件索引鍵來控制函數動作：

- aws：ResourceTag/標籤鍵：根據連接到 L ambda 函數的標籤控制訪問。
- aws：RequestTag/tag-key：要求標籤出現在請求中，例如在創建新函數時。
- aws: PrincipalTag /tag-key：根據附加到其 IAM 使用者或角色的標籤，控制 IAM 主體 (發出請求的人員) 可以執行的動作。
- aws:TagKeys: 控制是否可以在請求中使用特定的標籤密鑰。

如需支援 ABAC 的 Lambda 動作完整清單，請參閱 支援的函數動作，並檢查資料表中的 Condition (條件) 欄位。

以下步驟會示範使用 ABAC 設定許可的一種方法。在此範例案例中，您將會建立四個 IAM 許可政策。然後，您會將這些政策連接至新的 IAM 角色。最後，您將建立 IAM 使用者，並授予該使用者擔任新角色的許可。

主題

- 必要條件
- 步驟 1：要求新函數具有標籤
- 步驟 2：依據連接至 Lambda 函數和 IAM 主體的標籤來允許動作
- 步驟 3：授予 List 許可
- 步驟 4：授予 IAM 許可
- 步驟 5：建立 IAM 角色
- 步驟 6：建立 IAM 使用者
- 步驟 7：測試許可
- 步驟 8：清理資源

必要條件

請確定您擁有 Lambda 執行角色。您將會在授予 IAM 許可以及建立 Lambda 函數時使用此角色。

步驟 1：要求新函數具有標籤

在搭配使用 ABAC 和 Lambda 時，最佳實務是要求所有函數都具有標籤。這有助於確保您的 ABAC 許可政策如預期般運作。

建立與以下範例類似的 IAM 政策。此政策使用 aws: RequestTag /tag-key、aws: ResourceTag /tag-key 和 aws: TagKeys 條件金鑰來要求新函數和建立函數的 IAM 主體都有標籤。`projectForAllValues` 修飾詞會確保 `project` 是唯一受允許的標籤。如果您不納入 `ForAllValues` 修飾詞，則只要使用者也傳遞 `project`，其便也可新增其他標籤至函數。

Example – 要求新函數具有標籤

```
{
  "Version": "2012-10-17",
  "Statement": {
    "Effect": "Allow",
    "Action": [
      "lambda:CreateFunction",
      "lambda:TagResource"
    ],
    "Resource": "arn:aws:lambda:*:*:function:*",
    "Condition": {
      "StringEquals": {
        "aws:RequestTag/project": "${aws:PrincipalTag/project}",
        "aws:ResourceTag/project": "${aws:PrincipalTag/project}"
      },
      "ForAllValues:StringEquals": {
        "aws:TagKeys": "project"
      }
    }
  }
}
```

步驟 2：依據連接至 Lambda 函數和 IAM 主體的標籤來允許動作

使用 aws: ResourceTag /tag-key 條件鍵建立第二個 IAM 政策，以要求主體的標籤與附加到函數的標籤相符。以下範例政策會允許具有 `project` 標籤的主體呼叫具有 `project` 標籤的函數。如果函數有任何其他標籤，則會該動作將遭拒。

Example – 要求函數和 IAM 主體的標籤相符

```
{
```

```
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "lambda:InvokeFunction",
        "lambda:GetFunction"
      ],
      "Resource": "arn:aws:lambda:*:*:function:*",
      "Condition": {
        "StringEquals": {
          "aws:ResourceTag/project": "${aws:PrincipalTag/project}"
        }
      }
    }
  ]
}
```

步驟 3：授予 List 許可

建立允許主體列出 Lambda 函數和 IAM 角色的政策。此政策會允許主體在主控台以及呼叫 API 動作時查看所有 Lambda 函數和 IAM 角色。

Example – 授予 Lambda 和 IAM 的 List 許可

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllResourcesLambdaNoTags",
      "Effect": "Allow",
      "Action": [
        "lambda:GetAccountSettings",
        "lambda:ListFunctions",
        "iam:ListRoles"
      ],
      "Resource": "*"
    }
  ]
}
```

步驟 4：授予 IAM 許可

建立允許 iam: 的政策PassRole。在您將執行角色指派給函數時，便會需要此許可。在以下範例政策中，用您 Lambda 執行角色的 ARN 來取代範例 ARN。

> **Note**
>
> 請勿將政策中的 `ResourceTag` 條件金鑰與 `iam:PassRole` 動作搭配使用。您不能使用該 IAM 角色的標籤來控制可傳遞該角色的人員。如需將角色傳遞至服務所需權限的詳細資訊，請參閱授與使用者將角色傳遞至 AWS 服務的權限。

Example – 授予傳遞執行角色的許可

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "VisualEditor0",
      "Effect": "Allow",
      "Action": [
        "iam:PassRole"
      ],
      "Resource": "arn:aws:iam::111122223333:role/lambda-ex"
    }
  ]
}
```

步驟 5：建立 IAM 角色

最佳實務為使用角色來委派許可。建立名為 `abac-project-role` 的 IAM 角色：

- 在步驟 1：選取可信任實體時：選擇 AWS account (帳戶)，然後選擇 This account (此帳戶)。
- 在步驟 2：新增許可時：連接您於前一步中建立的四個 IAM 政策。
- 在步驟 3：命名、檢閱和建立時：選擇 Add tag (新增標籤)。在 Key (索引鍵) 欄位，輸入 `project`。請勿輸入值。

步驟 6：建立 IAM 使用者

建立名為 abac-test-user 的 IAM 使用者。在 Set permissions (設定許可) 區段中，選擇 Attach existing policies directly (直接連接現有政策)，接著選擇 Create policy (建立政策)。輸入以下政策定義。以您的 AWS 帳戶 ID 來取代 *111122223333*。此政策允許 abac-test-user 擔任 abac-project-role。

Example – 允許 IAM 使用者擔任 ABAC 角色

```
{
  "Version": "2012-10-17",
  "Statement": {
    "Effect": "Allow",
    "Action": "sts:AssumeRole",
    "Resource": "arn:aws:iam::111122223333:role/abac-project-role"
  }
}
```

步驟 7：測試許可

1. 登入 AWS 主控台的身分abac-test-user。如需詳細資訊，請參閱以 IAM 使用者身分登入。
2. 切換到 abac-project-role 角色。如需詳細資訊，請參閱切換到角色 (主控台)。
3. 建立 Lambda 函數。
   - 在 Permissions (許可) 下，選擇 Change default execution role (變更預設執行角色)，然後在 Execution role (執行角色) 中選擇 Use an existing role (使用現有角色)。選擇與您在 步驟 4：授予 IAM 許可 中使用的相同執行角色。
   - 在 Advanced settings (進階設定)下，選擇 Enable tags (啟用標籤)，然後選擇 Add new tag (新增標籤)。在 Key (索引鍵) 欄位，輸入 project。請勿輸入值。
4. 測試函數。
5. 建立第二個 Lambda 函數，並新增一個不同的標籤，例如 environment。此操作應會失敗，因為您在 步驟 1：要求新函數具有標籤 中建立的 ABAC 政策僅允許主體建立具有 project 標籤的函數。
6. 建立無標籤的第三個函數。此操作應會失敗，因為您在 步驟 1：要求新函數具有標籤 中建立的 ABAC 政策不允許主體建立無標籤的函數。

此授權策略允許您控制存取，且無需為每個新使用者建立新政策。若要將授予新使用者存取權限，只需要向其授予擔任與其受指派專案相對應角色的許可。

步驟 8：清理資源

刪除 IAM 角色

1. 開啟 IAM 主控台中的 角色頁面 。
2. 選取您在步驟 5 中建立的角色。
3. 選擇刪除。
4. 若要確認刪除，請在文字輸入欄位中輸入角色名稱。
5. 選擇刪除。

若要刪除 IAM 使用者

1. 開啟 IAM 主控台的「使用者」頁面。
2. 選取您在步驟 6 中建立的 IAM 使用者。
3. 選擇刪除。
4. 若要確認刪除，請在文字輸入欄位中輸入使用者名稱。
5. 選擇刪除使用者。

若要刪除 Lambda 函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇您建立的函數。
3. 選擇 Actions (動作)、Delete (刪除)。
4. 在文字輸入欄位中輸入 **`delete`**，然後選擇 刪除 。

微調政策的「資源」和「條件」部分

您可以在 AWS Identity and Access Management (IAM) 政策中指定資源和條件，來限制使用者許可的範圍。政策中的每個動作都支援資源和條件類型組合，組合內容取決於動作的行為。

每個 IAM 政策陳述式授予在資源上執行動作的許可。當動作不在具名資源上執行動作，或是當您授予對所有資源執行動作的許可，政策中資源的值是萬用字元 (*)。對於許多動作，您可以透過指定資源的 Amazon Resource Name (ARN) 或符合多個資源的 ARN 模式，來限制使用者可以修改的資源。

若要依照資源限制許可，請依照 ARN 指定資源。

Lambda 資源 ARN 格式

- 函數 - arn:aws:lambda:*us-west-2*:*123456789012*:function:*my-function*
- 函數版本 - arn:aws:lambda:*us-west-2*:*123456789012*:function:*my-function*:*1*
- 函數別名 - arn:aws:lambda:*us-west-2*:*123456789012*:function:*my-function*:*TEST*
- 事件來源映射 - arn:aws:lambda:*us-west-2*:*123456789012*:event-source-mapping:*fa123456-14a1-4fd2-9fec-83de64ad683de6d47*
- 層 - arn:aws:lambda:*us-west-2*:*123456789012*:layer:*my-layer*
- 層版本 - arn:aws:lambda:*us-west-2*:*123456789012*:layer:*my-layer*:*1*

例如，下列原則允許中的使用者叫 AWS 帳戶 123456789012用美國西部 (奧勒岡) AWS 區域my-function中名為的函數。

Example 呼叫函數政策

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Invoke",
            "Effect": "Allow",
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:my-function"
        }
    ]
}
```

此為特例，其中動作識別符 (lambda:InvokeFunction) 與 API 操作 (Invoke) 不同。對於其他動作，動作識別符就是開頭為 lambda: 的操作名稱。

章節

- 瞭解原則中的「條件」區段
- 在策略的「資源」部分中引用函數
- 支援的函數動作
- 支援的事件來源對應動作

- 支援的圖層動作

瞭解原則中的「條件」區段

條件是選用的政策元素，會套用額外的邏輯來判斷是否允許動作。除了所有動作皆支援的一般條件之外，Lambda 還會定義您可用來限制某些動作之其他參數值的條件類型。

例如，`lambda:Principal` 條件讓您可在函數的資源型政策中，限制使用者可授予叫用存取權限的服務或帳號。以下政策會讓使用者可將許可授予 Amazon Simple Notification Service (Amazon SNS) 主題以叫用名為 `test` 的函數。

Example 管理函數政策許可

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ManageFunctionPolicy",
            "Effect": "Allow",
            "Action": [
                "lambda:AddPermission",
                "lambda:RemovePermission"
            ],
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:test:*",
            "Condition": {
                "StringEquals": {
                    "lambda:Principal": "sns.amazonaws.com"
                }
            }
        }
    ]
}
```

條件要求委託人是 Amazon SNS 而不是其他服務或帳戶。資源模式要求函數名稱是 `test` 並包含版本編號或別名。例如 `test:v1`。

如需 Lambda 和其他 AWS 服務的資源和條件的詳細資訊，請參閱服務授權參考中服 AWS 務的動作、資源和條件金鑰。

在策略的「資源」部分中引用函數

您可以使用 Amazon Resource Name (ARN) 來參考政策陳述式中的 Lambda 函數。函數 ARN 的格式取決於您是參考整個函數 (不合格)、函數版本或別名 (合格)。

進行 Lambda API 呼叫時，使用者可以在參數中傳遞版本 ARN 或別名 ARN，或在GetFunction`FunctionName`參數中設定值，來指定版本或別名。GetFunction`Qualifier`Lambda 會比較 IAM 政策中的資源元素與 API 呼叫中傳遞的 `FunctionName` 和 `Qualifier`，從而作出授權決策。如果不符，Lambda 會拒絕該請求。

無論是允許還是拒絕對函數執行操作，都必須在政策陳述式中使用正確的函數 ARN 類型，才能達到預期的結果。例如，如果您的政策引用了不合格的 ARN，Lambda 會接受參考不合格 ARN 的請求，但拒絕參考合格 ARN 的請求。

> **Note**
>
> 不能使用萬用字元 (*) 以讓帳戶 ID 相符。如需有關已接受語法的詳細資訊，請參閱《IAM 使用者指南》中的 IAM JSON 政策參考。

Example 允許叫用不合格的 ARN

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:myFunction"
        }
    ]
}
```

如果您的政策引用了特定合格的 ARN，Lambda 會接受參考該 ARN 的請求，但拒絕參考不合格 ARN 或不同的合格 ARN 的請求，例如 `myFunction:2`。

Example 允許叫用特定的合格 ARN

```
{
```

```
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:myFunction:1"
        }
    ]
}
```

如果您的政策使用 **:*** 參考任何合格的 ARN，Lambda 會接受任何合格的 ARN，但拒絕參考不合格 ARN 的請求。

Example 允許叫用任何合格的 ARN

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:myFunction:*"
        }
    ]
}
```

如果您的政策使用 * 參考任何 ARN，Lambda 會接受任何合格或不合格的 ARN。

Example 允許叫用任何合格或不合格的 ARN

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:myFunction*"
        }
    ]
}
```

支援的函數動作

您可依據函數、版本或別名 ARN，將在函數上操作的動作限於特定函數，如下表所述。會將不支援資源限制的動作授予給所有資源 (*)。

函數動作

動作	資源	條件
AddPermission RemovePermission	函式 函式版本 函式別名	lambda:Principal aws:ResourceTag/${TagKey} lambda:FunctionUrlAuthType
Invoke 許可：lambda:InvokeFunction	函式 函式版本 函式別名	aws:ResourceTag/${TagKey} lambda:EventSourceToken
CreateFunction	函式	lambda:CodeSigningConfigArn lambda:Layer lambda:VpcIds lambda:SubnetIds lambda:SecurityGroupIds aws:ResourceTag/${TagKey} aws:RequestTag/${TagKey} aws:TagKeys
UpdateFunction配置	函式	lambda:CodeSigningConfigArn lambda:Layer

動作	資源	條件
		lambda:VpcIds lambda:SubnetIds lambda:SecurityGroupIds aws:ResourceTag/${TagKey}

動作	資源	條件
CreateAlias	函式	aws:ResourceTag/${TagKey}
DeleteAlias		
DeleteFunction		
DeleteFunctionCodeSigningConfig		
DeleteFunction并发性		
GetAlias		
GetFunction		
GetFunctionCodeSigningConfig		
GetFunction并发性		
GetFunction配置		
GetPolicy		
ListProvisionedConcurrencyConfigs		
ListAliases		
ListTags		
ListVersionsByFunction		
PublishVersion		
PutFunctionCodeSigningConfig		
PutFunction并发性		
UpdateAlias		
UpdateFunction代碼		

動作	資源	條件
CreateFunctionUrlConfig DeleteFunctionUrlConfig GetFunctionUrlConfig UpdateFunctionUrlConfig	函式 函式別名	lambda:FunctionUrlAuthType lambda:FunctionArn aws:ResourceTag/${TagKey}
ListFunctionUrlConfigs	函式	lambda:FunctionUrlAuthType
DeleteFunctionEventInvokeConfig GetFunctionEventInvokeConfig ListFunctionEventInvoke配置 PutFunctionEventInvokeConfig UpdateFunctionEventInvokeConfig	函式	aws:ResourceTag/${TagKey}
DeleteProvisionedConcurrencyConfig GetProvisionedConcurrencyConfig PutProvisionedConcurrencyConfig	函式別名 函式版本	aws:ResourceTag/${TagKey}
GetAccount設定 ListFunctions	*	無
TagResource	函式	aws:ResourceTag/${TagKey} aws:RequestTag/${TagKey} aws:TagKeys
UntagResource	函式	aws:ResourceTag/${TagKey} aws:TagKeys

支援的事件來源對應動作

對於事件來源映射，您可將 Delete 和 Update 許可限制至特定事件來源。`lambda:FunctionArn` 條件可讓您限制使用者可以設定事件來源叫用哪些函數。

對於這些動作，資源是事件來源映射，所以 Lambda 提供一個條件，可讓您根據事件來源映射叫用的函數來限制許可。

事件來源映射動作

動作	資源	條件
DeleteEventSourceMapping UpdateEventSourceMapping	事件來源映射	`lambda:FunctionArn`
CreateEventSourceMapping GetEventSourceMapping	*	`lambda:FunctionArn`
ListEventSourceMappings	*	無

支援的圖層動作

圖層動作可讓您限制使用者可以管理或搭配函數使用的圖層。與圖層使用和許可相關的動作在圖層版本上執行作業，而 `PublishLayerVersion` 在圖層名稱上執行作業。您可以搭配使用萬用字元，來限制使用者可依照名稱使用的 layer。

> **Note**
>
> 「GetLayer版本」動作也涵蓋了 GetLayerVersionByArn。Lambda 不支援 `GetLayerVersionByArn` 作為 IAM 動作。

圖層動作

動作	資源	條件
AddLayerVersionPermission	Layer 版本	無

動作	資源	條件
RemoveLayerVersionPermission GetLayer版本 GetLayerVersionPolicy DeleteLayer版本		
ListLayer版本 PublishLayer版本	Layer	無
ListLayers	*	無

AWS Lambda 中的安全性

雲端安全是 AWS 最重視的一環。身為 AWS 客戶的您，將能從資料中心和網路架構的建置中獲益，以滿足組織最為敏感的安全要求。

安全是 AWS 與您共同的責任。共同的責任模型 將此描述為雲端 本身 的安全和雲端內部的安全：

- 雲端本身的安全 – AWS 負責保護在 AWS Cloud 中執行 AWS 服務的基礎設施。AWS 也提供您可安全使用的服務。在 AWS 合規計畫中，第三方稽核員會定期測試並驗證我們的安全功效。若要進一步瞭解適用於 AWS Lambda 的合規計劃，請參閱 合規計劃範圍內的 AWS 服務。
- 雲端內部的安全 – 您的責任取決於所使用的 AWS 服務。您也必須對其他因素負責，包括資料的機密性、您公司的請求和適用法律和法規。

本文件有助於您了解如何在使用 Lambda 時套用共同責任模型。下列主題將示範如何設定 Lambda 以達到您的安全和合規目標。您也會了解如何使用其他 AWS 服務來協助監控並保護 Lambda 資源。

如需有關將安全性原則套用至 Lambda 應用程式的詳細資訊，請參閱無伺服器園地中的安全性。

主題

- 資料保護 AWS Lambda
- AWS Lambda 的身分和存取權管理
- 為 Lambda 函數和層建立治理策略
- AWS Lambda 的合規驗證
- AWS Lambda 中的恢復能力
- AWS Lambda 中的基礎設施安全

資料保護 AWS Lambda

AWS 共用責任模型適用於中的資料保護 AWS Lambda。如此模型所述， AWS 負責保護執行所有 AWS 雲端. 您負責維護在此基礎設施上託管內容的控制權。您也同時負責所使用 AWS 服務 的安全組態和管理任務。如需資料隱私權的詳細資訊，請參閱資料隱私權常見問答集。如需歐洲資料保護的相關資訊，請參閱 AWS 安全性部落格上的 AWS 共同責任模型和 GDPR 部落格文章。

基於資料保護目的，我們建議您使用 AWS IAM Identity Center 或 AWS Identity and Access Management (IAM) 保護 AWS 帳戶 登入資料並設定個別使用者。如此一來，每個使用者都只會獲得授與完成其任務所必須的許可。我們也建議您採用下列方式保護資料：

- 每個帳戶均要使用多重要素驗證 (MFA)。
- 使用 SSL/TLS 與 AWS 資源進行通訊。我們需要 TLS 1.2 並建議使用 TLS 1.3。
- 使用設定 API 和使用者活動記錄 AWS CloudTrail。
- 使用 AWS 加密解決方案以及其中的所有默認安全控制 AWS 服務。
- 使用進階的受管安全服務 (例如 Amazon Macie)，協助探索和保護儲存在 Amazon S3 的敏感資料。
- 如果透過命令列介面或 API 存取時需要經 AWS 過 FIPS 140-2 驗證的加密模組，請使用 FIPS 端點。如需 FIPS 和 FIPS 端點的相關資訊，請參閱聯邦資訊處理標準 (FIPS) 140-2 概觀。

我們強烈建議您絕對不要將客戶的電子郵件地址等機密或敏感資訊，放在標籤或自由格式的文字欄位中，例如名稱 欄位。這包括當您使用主控台、API 或 AWS 開發套件 AWS 服務 使用 Lambda 或其他工作時。 AWS CLI您在標籤或自由格式文字欄位中輸入的任何資料都可能用於計費或診斷日誌。如果您提供外部伺服器的 URL，我們強烈建議請勿在驗證您對該伺服器請求的 URL 中包含憑證資訊。

章節

傳輸中加密

Lambda API 端點僅支援透過 HTTPS 的安全連線。當您使用 AWS Management Console、AWS 開發套件或 Lambda API 管理 Lambda 資源時，所有通訊都會使用傳輸層安全性 (TLS) 進行加密。如需完整的 API 端點清單，請參閱《 AWS 一般參考》中的AWS 區域和端點。

當您將您的函數連線至檔案系統時，Lambda 會針對所有連線使用傳輸中加密。如需詳細資訊，請參閱 Amazon Elastic File System 使用者指南中的 Amazon EFS 的資料加密。

使用環境變數時，您可以啟用主控台加密協助程式以使用用戶端加密來保護傳輸中的環境變數。如需詳細資訊，請參閱 保護 Lambda 環境變數。

靜態加密

Lambda 總是在靜態時加密環境變數。根據預設，Lambda 會使 AWS KMS key 用 Lambda 在您的帳戶中建立的來加密您的環境變數。這 AWS 受管金鑰 被命名為`aws/lambda`。

依據每個函數，您可以選擇將 Lambda 設定為使用客戶管理的金鑰，而非使用預設的 AWS 受管金鑰來加密您的環境變數。如需詳細資訊，請參閱 保護 Lambda 環境變數。

Lambda 一律加密您上傳到 Lambda 的檔案，包括部署套件和層封存。

Amazon CloudWatch 日誌和預設 AWS X-Ray 也會加密資料，並可設定為使用客戶受管金鑰。如需詳細資訊，請參閱中的 CloudWatch 記錄檔和資料保護中的加密記錄資料 AWS X-Ray。

AWS Lambda 的身分和存取權管理

AWS Identity and Access Management (IAM) 是一種 AWS 服務，讓管理員能夠安全地控制對 AWS 資源的存取權。IAM 管理員會控制誰經過身分身分驗證 (已登入) 並得到授權 (具有許可) 來使用 Lambda 資源。IAM 是一種您可以免費使用的 AWS 服務。

主題

- 物件
- 使用身分驗證
- 使用政策管理存取權
- AWS Lambda 搭配 IAM 的運作方式
- AWS Lambda 的身分型政策範例
- AWS Lambda 的 AWS 受管政策
- 對 AWS Lambda 身分與存取進行疑難排解

物件

AWS Identity and Access Management (IAM) 的使用方式會不同，取決於您在 Lambda 中所執行的工作。

服務使用者 – 如果您使用 Lambda 執行任務，您的管理員會為您提供您需要的憑證和許可。隨著您為了執行作業而使用的 Lambda 功能數量變多，您可能會需要額外的許可。瞭解存取許可的管理方式可協助您向管理員請求正確的許可。若您無法存取 Lambda 中的某項功能，請參閱 對 AWS Lambda 身分與存取進行疑難排解。

服務管理員 – 如果您負責公司內的 Lambda 資源，您可能具備 Lambda 的完整存取權。您的任務是判斷服務使用者應存取的 Lambda 功能及資源。接著，您必須將請求提交給您的 IAM 管理員，來變更您服務使用者的許可。檢閱此頁面上的資訊，了解 IAM 的基本概念。若要進一步了解貴公司如何搭配使用 IAM 與 Lambda，請參閱 AWS Lambda 搭配 IAM 的運作方式。

IAM 管理員 – 如果您是 IAM 管理員，建議您掌握如何撰寫政策以管理 Lambda 存取權的詳細資訊。若要檢視您可以在 IAM 中使用的範例 Lambda 身分型政策，請參閱 AWS Lambda 的身分型政策範例。

使用身分驗證

身分驗證是使用身分憑證登入 AWS 的方式。您必須以 AWS 帳戶根使用者、IAM 使用者身分，或擔任 IAM 角色進行 驗證 (登入至 AWS)。

您可以使用透過身分來源 AWS IAM Identity Center 提供的憑證，以聯合身分登入 AWS。(IAM Identity Center) 使用者、貴公司的單一登入身分驗證和您的 Google 或 Facebook 憑證都是聯合身分的範例。您以聯合身分登入時，您的管理員先前已設定使用 IAM 角色的聯合身分。您 AWS 藉由使用聯合進行存取時，您會間接擔任角色。

根據您的使用者類型，您可以登入 AWS Management Console 或 AWS 存取入口網站。如需登入至 AWS 的相關資訊，請參閱《AWS 登入 使用者指南》中的如何登入您的 AWS 帳戶。

如果您是以程式設計的方式存取 AWS，AWS 提供軟體開發套件 (SDK) 和命令列介面 (CLI)，以便使用您的憑證透過密碼編譯方式簽署您的請求。如果您不使用 AWS 工具，您必須自行簽署請求。如需使用建議的方法自行簽署請求的相關資訊，請參閱《IAM 使用者指南》中的簽署 AWS API 請求。

無論您使用何種身分驗證方法，您可能都需要提供額外的安全性資訊。例如，AWS 建議您使用多重要素驗證 (MFA) 以提高帳戶的安全。如需更多資訊，請參閱《AWS IAM Identity Center 使用者指南》中的多重要素驗證和《IAM 使用者指南》中的在 AWS 中使用多重要素驗證 (MFA)。

AWS 帳戶 根使用者

如果是建立 AWS 帳戶，您會先有一個登入身分，可以完整存取帳戶中所有 AWS 服務 與資源。此身分稱為 AWS 帳戶 根使用者，使用建立帳戶時所使用的電子郵件地址和密碼即可登入並存取。強烈建議您不要以根使用者處理日常作業。保護您的根使用者憑證，並將其用來執行只能由根使用者執行的任務。如需這些任務的完整清單，了解需以根使用者登入的任務，請參閱《IAM 使用者指南》中的需要根使用者憑證的任務。

聯合身分

最佳實務是要求人類使用者 (包括需要管理員存取權的使用者)搭配身分提供者使用聯合功能，使用暫時憑證來存取 AWS 服務。

聯合身分是來自您企業使用者目錄的使用者、Web 身分供應商、AWS Directory Service、Identity Center 目錄或透過身分來源提供的憑證來存取 AWS 服務 的任何使用者。聯合身分存取 AWS 帳戶時，會擔任角色，並由角色提供暫時憑證。

對於集中式存取權管理，我們建議您使用 AWS IAM Identity Center。您可以在 IAM Identity Center 中建立使用者和群組，也可以連線並同步到自己身分來源中的一組使用者和群組，以便在您的所有 AWS 帳戶和應用程式中使用。如需 IAM Identity Center 的相關資訊，請參閱《AWS IAM Identity Center 使用者指南》中的什麼是 IAM Identity Center？。

IAM 使用者和群組

IAM 使用者是您 AWS 帳戶 中的一種身分，具備單一人員或應用程式的特定許可。建議您盡可能依賴暫時憑證，而不是擁有建立長期憑證 (例如密碼和存取金鑰)的 IAM 使用者。但是如果特定使用案例需要擁有長期憑證的 IAM 使用者，建議您輪換存取金鑰。如需詳細資訊，請參閱《IAM 使用者指南》中的為需要長期憑證的使用案例定期輪換存取金鑰。

IAM 群組是一種指定 IAM 使用者集合的身分。您無法以群組身分登入。您可以使用群組來一次為多名使用者指定許可。群組可讓管理大量使用者許可的過程變得更為容易。例如，您可以擁有一個名為 IAMAdmins 的群組，並給予該群組管理 IAM 資源的許可。

使用者與角色不同。使用者只會與單一人員或應用程式建立關聯，但角色的目的是在由任何需要它的人員取得。使用者擁有永久的長期憑證，但角色僅提供暫時憑證。若要進一步了解，請參閱《IAM 使用者指南》中的建立 IAM 使用者 (而非角色) 的時機。

IAM 角色

IAM 角色是您 AWS 帳戶 中的一種身分，具備特定許可。它類似 IAM 使用者，但不與特定的人員相關聯。您可以在 AWS Management Console 中透過切換角色來暫時取得 IAM 角色。您可以透過呼叫 AWS CLI 或 AWS API 操作，或是使用自訂 URL 來取得角色。如需使用角色的方法的相關資訊，請參閱《IAM 使用者指南》中的使用 IAM 角色。

使用暫時憑證的 IAM 角色在下列情況中非常有用：

- 聯合身分使用者存取 — 如需向聯合身分指派許可，請建立角色，並為角色定義許可。當聯合身分進行身分驗證時，該身分會與角色建立關聯，並取得由角色定義的許可。如需有關聯合角色的詳細資訊，請參閱《IAM 使用者指南》https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles_create_for-idp.html中的為第三方身分提供者建立角色。如果您使用 IAM Identity Center，則需要設定許可集。為控制身分驗證後可以存取的內容，IAM Identity Center 將許可集與 IAM 中的角色相關聯。如需有關許可集的資訊，請參閱《AWS IAM Identity Center 使用者指南》中的許可集。
- 暫時 IAM 使用者許可 – IAM 使用者或角色可以擔任 IAM 角色來暫時針對特定任務採用不同的許可。
- 跨帳戶存取權 – 您可以使用 IAM 角色，允許不同帳戶中的某人 (信任的委託人)存取您帳戶中的資源。角色是授予跨帳戶存取權的主要方式。但是，針對某些 AWS 服務，您可以將政策直接連接到資

源 (而非使用角色作為代理)。若要了解跨帳戶存取角色和資源型政策間的差異，請參閱《IAM 使用者指南》中的 IAM 角色與資源類型政策的差異。

- 跨服務存取 – 有些 AWS 服務 會使用其他 AWS 服務 中的功能。例如，當您在服務中進行呼叫時，該服務通常會在 Amazon EC2 中執行應用程式或將物件儲存在 Amazon Simple Storage Service (Amazon S3) 中。服務可能會使用呼叫主體的許可、使用服務角色或使用服務連結角色來執行此作業。
  - 轉發存取工作階段 (FAS)：當您使用 IAM 使用者或角色在 AWS 中執行動作時，系統會將您視為主體。當您使用某些服務時，您可能會執行一個動作，而該動作之後會在不同的服務中啟動另一個動作。FAS 使用主體的許可呼叫 AWS 服務，搭配請求 AWS 服務 以向下游服務發出請求。只有在服務收到需要與其他 AWS 服務 或資源互動才能完成的請求之後，才會提出 FAS 請求。在此情況下，您必須具有執行這兩個動作的許可。如需提出 FAS 請求時的政策詳細資訊，請參閱《轉發存取工作階段》。
  - 服務角色：服務角色是服務擔任的 IAM 角色，可代表您執行動作。IAM 管理員可以從 IAM 內建立、修改和刪除服務角色。如需詳細資訊，請參閱《IAM 使用者指南》中的建立角色以委派許可給 AWS 服務 服務。
  - 服務連結角色 – 服務連結角色是一種連結到 AWS 服務 的服務角色類型。服務可以擔任代表您執行動作的角色。服務連結角色會顯示在您的 AWS 帳戶 中，並由該服務所擁有。IAM 管理員可以檢視，但不能編輯服務連結角色的許可。
- 在 Amazon EC2 上執行的應用程式 – 針對在 EC2 執行個體上執行並提出 AWS CLI 和 AWS API 請求的應用程式，您可以使用 IAM 角色來管理暫時憑證。這是在 EC2 執行個體內儲存存取金鑰的較好方式。如需指派 AWS 角色給 EC2 執行個體並提供其所有應用程式使用，您可以建立連接到執行個體的執行個體設定檔。執行個體設定檔包含該角色，並且可讓 EC2 執行個體上執行的程式取得暫時憑證。如需詳細資訊，請參閱《IAM 使用者指南》中的利用 IAM 角色來授予許可給 Amazon EC2 執行個體上執行的應用程式。

如需了解是否要使用 IAM 角色或 IAM 使用者，請參閱《IAM 使用者指南》中的建立 IAM 角色 (而非使用者) 的時機。

使用政策管理存取權

您可以透過建立政策並將其附加到 AWS 身分或資源，在 AWS 中控制存取。政策是 AWS 中的一個物件，當其和身分或資源建立關聯時，便可定義其許可。AWS 會在主體 (使用者、根使用者或角色工作階段)發出請求時評估這些政策。政策中的許可，決定是否允許或拒絕請求。大部分政策以 JSON 文件形式儲存在 AWS 中。如需 JSON 政策文件結構和內容的詳細資訊，請參閱《IAM 使用者指南》中的 JSON 政策概觀。

管理員可以使用 AWS JSON 政策來指定誰可以存取哪些內容。也就是說，哪個主體在什麼條件下可以對什麼資源執行哪些動作。

預設情況下，使用者和角色沒有許可。若要授與使用者對其所需資源執行動作的許可，IAM 管理員可以建立 IAM 政策。然後，管理員可以將 IAM 政策新增至角色，使用者便能擔任這些角色。

IAM 政策定義該動作的許可，無論您使用何種方法來執行操作。例如，假設您有一個允許 `iam:GetRole` 動作的政策。具備該政策的使用者便可以從 AWS Management Console、AWS CLI 或 AWS API 取得角色資訊。

身分型政策

身分型政策是可以附加到身分 (例如 IAM 使用者、使用者群組或角色)的 JSON 許可政策文件。這些政策可控制身分在何種條件下能對哪些資源執行哪些動作。若要了解如何建立身分類型政策，請參閱《IAM 使用者指南》中的建立 IAM 政策。

身分型政策可進一步分類成內嵌政策或受管政策。內嵌政策會直接內嵌到單一使用者、群組或角色。受管政策則是獨立的政策，您可以將這些政策附加到 AWS 帳戶 中的多個使用者、群組和角色。受管政策包含 AWS 管理政策和客戶管理政策。如需瞭解如何在受管政策及內嵌政策間選擇，請參閱 IAM 使用者指南中的在受管政策和內嵌政策間選擇。

資源型政策

資源型政策是連接到資源的 JSON 政策文件。資源型政策的最常見範例是 IAM 角色信任政策和 Amazon S3 儲存貯體政策。在支援資源型政策的服務中，服務管理員可以使用它們來控制對特定資源的存取權。對於附加政策的資源，政策會定義指定的主體可以對該資源執行的動作以及在何種條件下執行的動作。您必須在資源型政策中指定主體。主體可以包括帳戶、使用者、角色、聯合身分使用者或 AWS 服務。

資源型政策是位於該服務中的內嵌政策。您無法在資源型政策中使用來自 IAM 的 AWS 受管政策。

存取控制清單 (ACL)

存取控制清單 (ACL) 可控制哪些委託人 (帳戶成員、使用者或角色)擁有存取某資源的許可。ACL 類似於資源型政策，但它們不使用 JSON 政策文件格式。

Amazon Simple Storage Service (Amazon S3)、AWS WAF 和 Amazon VPC 是支援 ACL 的服務範例。若要進一步了解 ACL，請參閱《Amazon Simple Storage Service 開發人員指南》中的存取控制清單 (ACL) 概觀。

其他政策類型

AWS 支援其他較少見的政策類型。這些政策類型可設定較常見政策類型授與您的最大許可。

- 許可界限 – 許可範圍是一種進階功能，可供您設定身分型政策能授予 IAM 實體 (IAM 使用者或角色) 的最大許可。您可以為實體設定許可界限。所產生的許可會是實體的身分型政策和其許可界限的交集。會在 `Principal` 欄位中指定使用者或角色的資源型政策則不會受到許可界限限制。所有這類政策中的明確拒絕都會覆寫該允許。如需許可範圍的更多相關資訊，請參閱《IAM 使用者指南》中的 IAM 實體許可範圍。
- 服務控制政策 (SCP) – SCP 是 JSON 政策，可指定 AWS Organizations 中組織或組織單位 (OU) 的最大許可。AWS Organizations 服務可用來分組和集中管理您企業所擁有的多個 AWS 帳戶。若您啟用組織中的所有功能，您可以將服務控制政策 (SCP) 套用到任何或所有帳戶。SCP 會限制成員帳戶中實體的許可，包括每個 AWS 帳戶根使用者。如需組織和 SCP 的更多相關資訊，請參閱《AWS Organizations 使用者指南》中的 SCP 運作方式。
- 工作階段政策 – 工作階段政策是一種進階政策，您可以在透過編寫程式的方式建立角色或聯合使用者的暫時工作階段時，作為參數傳遞。所產生工作階段的許可會是使用者或角色的身分型政策和工作階段政策的交集。許可也可以來自資源型政策。所有這類政策中的明確拒絕都會覆寫該允許。如需更多資訊，請參閱《IAM 使用者指南》中的工作階段政策。

多種政策類型

將多種政策類型套用到請求時，其結果形成的許可會更為複雜、更加難以理解。若要了解 AWS 在涉及多種政策類型時如何判斷是否允許一項請求，請參閱《IAM 使用者指南》中的政策評估邏輯。

AWS Lambda 搭配 IAM 的運作方式

在您使用 IAM 管理 Lambda 的存取權限之前，請先了解哪些 IAM 功能可與 Lambda 搭配使用。

您可搭配 AWS Lambda 使用的 IAM 功能

IAM 功能	Lambda 支持
身分型政策	是
資源型政策	是
政策動作	是

IAM 功能	Lambda 支持
政策資源	是
政策條件索引鍵 (服務特定)	是
ACL	否
ABAC(政策中的標籤)	部分
臨時憑證	是
轉送存取工作階段 (FAS)	否
服務角色	是
服務連結角色	部分

若要深入瞭解 Lambda 和其他AWS服務如何搭配大多數 IAM 功能使用，請參閱 IAM 使用者指南中的可與 IAM 搭配使用的AWS服務。

Lambda 的基於身分識別的政策

支援身分型政策	是

身分型政策是可以連接到身分 (例如 IAM 使用者、使用者群組或角色) 的 JSON 許可政策文件。這些政策可控制身分在何種條件下能對哪些資源執行哪些動作。若要瞭解如何建立身分類型政策，請參閱《IAM 使用者指南》中的建立 IAM 政策。

使用 IAM 身分型政策，您可以指定允許或拒絕的動作和資源，以及在何種條件下允許或拒絕動作。您無法在身分型政策中指定主體，因為這會套用至附加的使用者或角色。如要瞭解您在 JSON 政策中使用的所有元素，請參閱 IAM 使用者指南中的 IAM JSON 政策元素參考。

Lambda 的基於身分識別的政策範例

若要檢視以 Lambda 身分識別為基礎的政策範例，請參閱。AWS Lambda 的身分型政策範例

以資源為基礎的政策

支援以資源基礎的政策	是

資源型政策是附加到資源的 JSON 政策文件。資源型政策的最常見範例是 IAM 角色信任政策和 Amazon S3 儲存貯體政策。在支援資源型政策的服務中，服務管理員可以使用它們來控制對特定資源的存取權。對於附加政策的資源，政策會定義指定的主體可以對該資源執行的動作以及在何種條件下執行的動作。您必須在資源型政策中指定主體。主體可以包括帳戶、使用者、角色、聯合身分使用者或 AWS 服務。

若要啟用跨帳戶存取權，您可以指定在其他帳戶內的所有帳戶或 IAM 實體，作為資源型政策的主體。新增跨帳戶主體至資源型政策，只是建立信任關係的一半。當主體和資源在不同的 AWS 帳戶 中時，受信任帳戶中的 IAM 管理員也必須授與主體實體 (使用者或角色) 存取資源的許可。其透過將身分型政策附加到實體來授予許可。不過，如果資源型政策會為相同帳戶中的主體授與存取，這時就不需要額外的身分型政策。如需詳細資訊，請參閱 IAM 使用者指南中的 IAM 角色與資源型政策有何差異。

您可以將以資源為基礎的政策附加到 Lambda 函數或層。此原則定義哪些主體可以對函數或層執行動作。

若要瞭解如何將以資源為基礎的政策附加至函數或層，請參閱使用 Lambda 中以資源為基礎的政策。

Lambda 的政策動作

支援政策動作	是

管理員可以使用 AWS JSON 政策來指定誰可以存取哪些內容。也就是說，哪個主體在什麼條件下可以對什麼資源執行哪些動作。

JSON 政策的 `Action` 元素描述您可以用來允許或拒絕政策中存取的動作。政策動作的名稱通常會和相關聯的 AWS API 操作相同。有一些例外狀況，例如沒有相符的 API 操作的僅限許可動作。也有一些操作需要政策中的多個動作。這些額外的動作稱為相依動作。

政策會使用動作來授與執行相關聯操作的許可。

若要查看 Lambda 動作清單，請參閱服務授權參考AWS Lambda中所定義的動作。

Lambda 中的政策動作會在動作之前使用下列前置詞：

```
lambda
```

如需在單一陳述式中指定多個動作，請用逗號分隔。

```
"Action": [
      "lambda:action1",
      "lambda:action2"
         ]
```

若要檢視以 Lambda 身分識別為基礎的政策範例，請參閱。AWS Lambda 的身分型政策範例

Lambda 的政策資源

支援政策資源	是

管理員可以使用 AWS JSON 政策來指定誰可以存取哪些內容。也就是說，哪個主體在什麼條件下可以對什麼資源執行哪些動作。

Resource JSON 政策元素可指定要套用動作的物件。陳述式必須包含 Resource 或 NotResource 元素。最佳實務是使用其 Amazon Resource Name (ARN) 來指定資源。您可以針對支援特定資源類型的動作 (稱為資源層級許可) 來這麼做。

對於不支援資源層級許可的動作 (例如列出作業)，請使用萬用字元 (*) 來表示陳述式適用於所有資源。

```
"Resource": "*"
```

若要查看 Lambda 資源類型及其 ARN 的清單，請參閱服務授權參考AWS Lambda中所定義的資源類型。若要了解您可以使用哪些動作指定每個資源的 ARN，請參閱 AWS Lambda 定義的動作。

若要檢視以 Lambda 身分識別為基礎的政策範例，請參閱。AWS Lambda 的身分型政策範例

Lambda 的政策條件索引鍵

支援服務特定政策條件索引鍵	是

管理員可以使用 AWS JSON 政策來指定誰可以存取哪些內容。也就是說，哪個主體在什麼條件下可以對什麼資源執行哪些動作。

Condition 元素 (或 Condition 區塊)可讓您指定使陳述式生效的條件。Condition 元素是選用項目。您可以建立使用條件運算子的條件運算式 (例如等於或小於)，來比對政策中的條件和請求中的值。

若您在陳述式中指定多個 Condition 元素，或是在單一 Condition 元素中指定多個索引鍵，AWS 會使用邏輯 AND 操作評估他們。若您為單一條件索引鍵指定多個值，AWS 會使用邏輯 OR 操作評估條件。必須符合所有條件，才會授與陳述式的許可。

您也可以在指定條件時使用預留位置變數。例如，您可以只在使用者使用其 IAM 使用者名稱標記時，將存取資源的許可授與該 IAM 使用者。如需更多資訊，請參閱《IAM 使用者指南》中的 IAM 政策元素：變數和標籤。

AWS 支援全域條件索引鍵和服務特定的條件索引鍵。若要查看 AWS 全域條件索引鍵，請參閱《IAM 使用者指南》中的 AWS 全域條件內容索引鍵。

若要查看 Lambda 條件金鑰清單，請參閱服務授權參考AWS Lambda中的條件金鑰。若要了解您可以針對何種動作及資源使用條件索引鍵，請參閱 AWS Lambda 定義的動作。

若要檢視以 Lambda 身分識別為基礎的政策範例，請參閱。AWS Lambda 的身分型政策範例

在 Lambda 語中的 ACL

支援 ACL	否

存取控制清單 (ACL) 可控制哪些主體 (帳戶成員、使用者或角色) 擁有存取某資源的許可。ACL 類似於資源型政策，但它們不使用 JSON 政策文件格式。

阿巴克與 Lambda

支援 ABAC (政策中的標籤)	部分

屬性型存取控制 (ABAC) 是一種授權策略，可根據屬性來定義許可。在 AWS 中，這些屬性稱為標籤。您可以將標籤附加到 IAM 實體 (使用者或角色)，以及許多 AWS 資源。為實體和資源加上標籤是 ABAC 的第一步。您接著要設計 ABAC 政策，允許在主體的標籤與其嘗試存取的資源標籤相符時操作。

ABAC 在成長快速的環境中相當有幫助，並能在政策管理變得繁瑣時提供協助。

若要根據標籤控制存取，請使用 aws:ResourceTag/*key-name*、aws:RequestTag/*key-name* 或 aws:TagKeys 條件索引鍵，在政策的條件元素中，提供標籤資訊。

如果服務支援每個資源類型的全部三個條件索引鍵，則對該服務而言，值為 Yes。如果服務僅支援某些資源類型的全部三個條件索引鍵，則值為 Partial。

如需 ABAC 的詳細資訊，請參閱《IAM 使用者指南》中的什麼是 ABAC?。如要查看含有設定 ABAC 步驟的教學課程，請參閱《IAM 使用者指南》中的使用屬性型存取控制 (ABAC)。

如需標記 Lambda 資源的詳細資訊，請參閱在 Lambda 中使用基於屬性的存取控制。

搭配 Lambda 使用臨時登入

支援臨時憑證	是

您使用臨時憑證進行登入時，某些 AWS 服務 無法運作。如需詳細資訊，包括那些 AWS 服務 搭配臨時憑證運作，請參閱《IAM 使用者指南》中的可搭配 IAM 運作的 AWS 服務。

如果您使用使用者名稱和密碼之外的任何方法登入 AWS Management Console，則您正在使用臨時憑證。例如，當您使用公司的單一登入(SSO)連結存取 AWS 時，該程序會自動建立臨時憑證。當您以使用者身分登入主控台，然後切換角色時，也會自動建立臨時憑證。如需切換角色的詳細資訊，請參閱 IAM 使用者指南中的切換至角色 (主控台)。

您可使用 AWS CLI 或 AWS API，手動建立臨時憑證。接著，您可以使用這些臨時憑證來存取 AWS。AWS 建議您動態產生臨時憑證，而非使用長期存取金鑰。如需詳細資訊，請參閱 IAM 中的暫時性安全憑證。

Lambda 的轉寄存取會話

支援轉寄存取工作階段 (FAS)	否

當您使用 IAM 使用者或角色在 AWS 中執行動作時，系統會將您視為委託人。使用某些服務時，您可能會執行某個動作，進而在不同服務中啟動另一個動作。FAS 使用主體的許可呼叫 AWS 服務，搭配請求 AWS 服務 以向下游服務發出請求。只有在服務收到需要與其他 AWS 服務 或資源互動才能完成

的請求之後，才會提出 FAS 請求。在此情況下，您必須具有執行這兩個動作的許可。如需提出 FAS 請求時的政策詳細資訊，請參閱《轉發存取工作階段》。

Lambda 的服務角色

支援服務角色	是

服務角色是服務擔任的 IAM 角色，可代您執行動作。IAM 管理員可以從 IAM 內建立、修改和刪除服務角色。如需詳細資訊，請參閱《IAM 使用者指南》中的建立角色以委派許可給 AWS 服務 服務。

在 Lambda 中，服務角色稱為執行角色。

> **Warning**
>
> 變更執行角色的權限可能會中斷 Lambda 功能。

Lambda 的服務連結角色

支援服務連結角色	部分

服務連結角色是一種連結到 AWS 服務 的服務角色類型。服務可以擔任代表您執行動作的角色。服務連結角色會顯示在您的 AWS 帳戶 中，並由該服務所擁有。IAM 管理員可以檢視，但不能編輯服務連結角色的許可。

Lambda 沒有服務連結角色，但 Lambda@Edge 則有。如需詳細資訊，請參閱 Amazon CloudFront 開發人員指南中的 Lambda @Edge 的服務連結角色。

如需建立或管理服務連結角色的詳細資訊，請參閱可搭配 IAM 運作的 AWS 服務。在表格中尋找服務，其中包含服務連結角色欄中的 Yes。選擇 Yes (是) 連結，以檢視該服務的服務連結角色文件。

AWS Lambda 的身分型政策範例

根據預設，使用者和角色不具備建立或修改 Lambda 資源的許可。他們也無法使用 AWS Management Console、AWS Command Line Interface (AWS CLI) 或 AWS API 執行任務。若要授與使用者對其所需資源執行動作的許可，IAM 管理員可以建立 IAM 政策。然後，管理員可以將 IAM 政策新增至角色，使用者便能擔任這些角色。

若要了解如何使用這些範例 JSON 政策文件建立 IAM 身分型政策，請參閱《IAM 使用者指南》中的建立 IAM 政策。

如需 Lambda 定義的動作和資源類型的詳細資訊，包括每個資源類型的 ARN 格式，請參閱服務授權參考AWS Lambda中的動作、資源和條件金鑰。

主題

- 政策最佳實務
- 使用 Lambda 主控台
- 允許使用者檢視他們自己的許可

政策最佳實務

身分型政策會判斷您帳戶中的某個人員是否可以建立、存取或刪除 Lambda 資源。這些動作可能會讓您的 AWS 帳戶 產生費用。當您建立或編輯身分型政策時，請遵循下列準則及建議事項：

- 開始使用 AWS 受管政策並朝向最低權限許可的目標邁進：如需開始授予許可給使用者和工作負載，請使用 AWS 受管政策，這些政策會授予許可給許多常用案例。它們可在您的 AWS 帳戶 中使用。我們建議您定義特定於使用案例的 AWS 客戶管理政策，以便進一步減少許可。如需更多資訊，請參閱 IAM 使用者指南中的 AWS 受管政策或任務職能的 AWS 受管政策。
- 套用最低許可許可 – 設定 IAM 政策的許可時，請僅授予執行任務所需的權限。為實現此目的，您可以定義在特定條件下可以對特定資源採取的動作，這也稱為最低權限許可。如需使用 IAM 套用許可的更多相關資訊，請參閱 IAM 使用者指南中的 IAM 中的政策和許可。
- 使用 IAM 政策中的條件進一步限制存取權 – 您可以將條件新增至政策，以限制動作和資源的存取。例如，您可以撰寫政策條件，指定必須使用 SSL 傳送所有請求。您也可以使用條件來授予對服務動作的存取權，前提是透過特定 AWS 服務 (例如 AWS CloudFormation)使用條件。如需更多資訊，請參閱《IAM 使用者指南》中的 IAM JSON 政策元素：條件。
- 使用 IAM Access Analyzer 驗證 IAM 政策，確保許可安全且可正常運作 – IAM Access Analyzer 驗證新政策和現有政策，確保這些政策遵從 IAM 政策語言 (JSON) 和 IAM 最佳實務。IAM Access Analyzer 提供 100 多項政策檢查及切實可行的建議，可協助您編寫安全且實用的政策。如需更多資訊，請參閱 IAM 使用者指南中的 IAM Access Analyzer 政策驗證。
- 需要多重要素驗證 (MFA)：如果存在需要 AWS 帳戶中 IAM 使用者或根使用者的情況，請開啟 MFA 提供額外的安全性。如需在呼叫 API 操作時請求 MFA，請將 MFA 條件新增至您的政策。如需更多資訊，請參閱 IAM 使用者指南中的設定 MFA 保護的 API 存取。

有關 IAM 中最佳實務的更多相關資訊，請參閱 IAM 使用者指南中的 IAM 最佳安全實務。

使用 Lambda 主控台

若要存取 AWS Lambda 主控台，您必須擁有最低的一組許可。這些權限必須允許您列出和檢視AWS 帳戶. 如果您建立比最基本必要許可更嚴格的身分型政策，則對於具有該政策的實體 (使用者或角色) 而言，主控台就無法如預期運作。

對於僅呼叫 AWS CLI 或 AWS API 的使用者，您不需要允許其最基本主控台許可。反之，只需允許存取符合他們嘗試執行之 API 操作的動作就可以了。

如需針對函式開發授予最低存取權的範例政策，請參閱 撰寫將函數授與使用者權限的範例原則。除了 Lambda API，Lambda 主控台會使用其他服務來顯示觸發組態，並讓您新增新的觸發。如果您的使用者搭配其他服務使用 Lambda，他們就必須也要存取這些服務。如需透過 Lambda 設定其他服務的詳細資訊，請參閱 使用來自其 AWS 他服務的事件叫用 Lambda。

允許使用者檢視他們自己的許可

此範例會示範如何建立政策，允許 IAM 使用者檢視附加到他們使用者身分的內嵌及受管政策。此政策包含在主控台上，或是使用 AWS CLI 或 AWS API 透過編寫程式的方式完成此動作的許可。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
                "iam:GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
        },
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam:ListAttachedGroupPolicies",
```

```
                "iam:ListGroupPolicies",
                "iam:ListPolicyVersions",
                "iam:ListPolicies",
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

AWS Lambda 的 AWS 受管政策

AWS 管理的政策是由 AWS 建立和管理的獨立政策。AWS 管理的政策的設計在於為許多常見使用案例提供許可，如此您就可以開始將許可指派給使用者、群組和角色。

請謹記，AWS 管理的政策可能不會授予您特定使用案例的最低權限許可，因為它們可供所有 AWS 客戶使用。我們建議您定義使用案例專屬的客戶管理政策，以便進一步減少許可。

您無法更改 AWS 管理的政策中定義的許可。如果 AWS 更新 AWS 管理的政策中定義的許可，更新會影響政策連接的所有主體身分 (使用者、群組和角色)。在推出新的 AWS 服務 或有新的 API 操作可供現有服務使用時，AWS 很可能會更新 AWS 管理的政策。

如需詳細資訊，請參閱《IAM 使用者指南》中的 AWS 受管政策。

主題

- AWS受管理的策略： AWSLambda_FullAccess
- AWS受管理的策略： AWSLambda_ReadOnlyAccess
- AWS受管理的策略： AWSLambdaBasicExecutionRole
- AWS受管理的策略： AWSLambdaDynamoDBExecutionRole
- AWS受管理的策略： AWSLambdaENIManagementAccess
- AWS受管理的策略： AWSLambdaExecute
- AWS受管政策： AWSLambdaInvocation
- AWS受管理的策略： AWSLambdaKinesisExecutionRole
- AWS受管理的策略： AWSLambdaMSKExecutionRole
- AWS受管理的策略： AWSLambdaRole

- AWS受管理的策略： AWSLambdaSQSQueueExecutionRole
- AWS受管理的策略： AWSLambdaVPCAccessExecutionRole
- Lambda AWS 受管政策更新項目

AWS受管理的策略： AWSLambda_FullAccess

此政策也會授予完整存取權給 Lambda 動作。它還會將許可授予用於開發和維護 Lambda 資源的其他 AWS 服務。

您可以將 AWSLambda_FullAccess 政策連接至使用者、群組與角色。

許可詳細資訊

此政策包含以下許可：

- lambda：允許委託人完整存取 Lambda。
- cloudformation：允許委託人描述 AWS CloudFormation 堆疊，並列出這些堆疊中的資源。
- cloudwatch— 允許主體列出 Amazon CloudWatch 指標並取得指標資料。
- ec2：允許委託人描述安全群組、子網路和 VPC。
- iam：允許委託人取得政策、政策版本、角色、角色政策、連接角色政策以及角色清單。此政策也允許委託人將角色傳遞給 Lambda。將執行角色指派給函數時，便會使用 PassRole 許可。
- kms：允許委託人列出別名。
- logs— 允許主體描述 Amazon CloudWatch 日誌群組。對於與 Lambda 函數相關聯的日誌群組，此政策可讓委託人描述日誌串流、取得日誌事件，以及篩選日誌事件。
- states：允許委託人描述及列出 AWS Step Functions 狀態機器。
- tag：允許委託人根據其標籤取得資源。
- xray：允許委託人取得 AWS X-Ray 追蹤摘要，並擷取 ID 指定的追蹤清單。

如需有關此原則的詳細資訊，包括 JSON 政策文件和政策版本，請參閱AWS受管理原則參考指南AWSLambda_FullAccess中的。

AWS受管理的策略： AWSLambda_ReadOnlyAccess

此政策授予 Lambda 資源和其他 AWS 資源的唯讀存取權，可用於開發和維護 Lambda 資源。

您可以將 AWSLambda_ReadOnlyAccess 政策連接至使用者、群組與角色。

許可詳細資訊

此政策包含以下許可：

- `lambda`：允許委託人取得和列出所有資源。
- `cloudformation`：允許委託人描述和列出 AWS CloudFormation 堆疊，並列出這些堆疊中的資源。
- `cloudwatch`— 允許主體列出 Amazon CloudWatch 指標並取得指標資料。
- `ec2`：允許委託人描述安全群組、子網路和 VPC。
- `iam`：允許委託人取得政策、政策版本、角色、角色政策、連接角色政策以及角色清單。
- `kms`：允許委託人列出別名。
- `logs`— 允許主體描述 Amazon CloudWatch 日誌群組。對於與 Lambda 函數相關聯的日誌群組，此政策可讓委託人描述日誌串流、取得日誌事件，以及篩選日誌事件。
- `states`：允許委託人描述及列出 AWS Step Functions 狀態機器。
- `tag`：允許委託人根據其標籤取得資源。
- `xray`：允許委託人取得 AWS X-Ray 追蹤摘要，並擷取 ID 指定的追蹤清單。

如需有關此原則的詳細資訊，包括 JSON 政策文件和政策版本，請參閱AWS受管理原則參考指南AWSLambda_ReadOnlyAccess中的。

AWS受管理的策略： AWSLambdaBasicExecutionRole

此原則會授與將記錄檔上傳至記錄 CloudWatch 檔的權限。

您可以將 `AWSLambdaBasicExecutionRole` 政策連接至使用者、群組與角色。

如需有關此原則的詳細資訊，包括 JSON 政策文件和政策版本，請參閱AWS受管理原則參考指南AWSLambdaBasicExecutionRole中的。

AWS受管理的策略： AWSLambdaDynamoDBExecutionRole

此政策授予從 Amazon DynamoDB 串流讀取記錄並寫入日誌的權限。 CloudWatch

您可以將 `AWSLambdaDynamoDBExecutionRole` 政策連接至使用者、群組與角色。

如需有關此原則的詳細資訊，包括 JSON 政策文件和政策版本，請參閱AWS受管理原則參考指南AWSLambdaDynamoDBExecutionRole中的。

AWS受管理的策略： AWSLambdaENIManagementAccess

此政策授予建立、描述和刪除已啟用 VPC 之 Lambda 函數所使用之彈性網路界面的許可。

您可以將 `AWSLambdaENIManagementAccess` 政策連接至使用者、群組與角色。

如需有關此原則的詳細資訊，包括 JSON 政策文件和政策版本，請參閱AWS受管理原則參考指南AWSLambdaENIManagementAccess中的。

AWS受管理的策略： AWSLambdaExecute

此政策授予PUT和存GET取 Amazon 簡單儲存服務，以及對 CloudWatch 日誌的完整存取權。

您可以將 `AWSLambdaExecute` 政策連接至使用者、群組與角色。

如需有關此原則的詳細資訊，包括 JSON 政策文件和政策版本，請參閱AWS受管理原則參考指南AWSLambdaExecute中的。

AWS受管政策： AWSLambdaInvocation

此政策授予 Amazon DynamoDB Streams 的讀取存取權。

您可以將 `AWSLambdaInvocation-DynamoDB` 政策連接至使用者、群組與角色。

如需有關此原則的詳細資訊，包括 JSON 政策文件和政策版本，請參閱AWS受管原則參考指南中的AWSLambdaInvocation-DynamoDB。

AWS受管理的策略： AWSLambdaKinesisExecutionRole

此政策授予從 Amazon Kinesis 資料串流讀取事件並寫入 CloudWatch 日誌的權限。

您可以將 `AWSLambdaKinesisExecutionRole` 政策連接至使用者、群組與角色。

如需有關此原則的詳細資訊，包括 JSON 政策文件和政策版本，請參閱AWS受管理原則參考指南AWSLambdaKinesisExecutionRole中的。

AWS受管理的策略： AWSLambdaMSKExecutionRole

此政策授予讀取和存取來自 Apache Kafka 叢集之 Amazon 受管串流記錄、管理彈性網路界面以及寫入日誌的權限。 CloudWatch

您可以將 `AWSLambdaMSKExecutionRole` 政策連接至使用者、群組與角色。

如需有關此原則的詳細資訊，包括 JSON 政策文件和政策版本，請參閱AWS受管理原則參考指南AWSLambdaMSKExecutionRole中的。

AWS受管理的策略： AWSLambdaRole

此政策授予調用 Lambda 函數的許可。

您可以將 `AWSLambdaRole` 政策連接至使用者、群組與角色。

如需有關此原則的詳細資訊，包括 JSON 政策文件和政策版本，請參閱AWS受管理原則參考指南AWSLambdaRole中的。

AWS受管理的策略： AWSLambdaSQSQueueExecutionRole

此政策授予讀取和刪除 Amazon 簡單佇列服務佇列中訊息的權限，並授予對 CloudWatch 日誌的寫入許可。

您可以將 `AWSLambdaSQSQueueExecutionRole` 政策連接至使用者、群組與角色。

如需有關此原則的詳細資訊，包括 JSON 政策文件和政策版本，請參閱AWS受管理原則參考指南AWSLambdaSQSQueueExecutionRole中的。

AWS受管理的策略： AWSLambdaVPCAccessExecutionRole

此政策授予管理 Amazon Virtual Private Cloud 中彈性網路界面和寫入 CloudWatch 日誌的許可。

您可以將 `AWSLambdaVPCAccessExecutionRole` 政策連接至使用者、群組與角色。

如需有關此原則的詳細資訊，包括 JSON 政策文件和政策版本，請參閱AWS受管理原則參考指南AWSLambdaVPCAccessExecutionRole中的。

Lambda AWS 受管政策更新項目

變更	描述	日期
AWSLambdaVPCAccessExecutionRole— 變更	Lambda 已更新`AWSLambdaVPCAccessExecutionRole` 政策以允許執行此動作`ec2:DescribeSubnets` 。	2024年1月5日

變更	描述	日期
AWSLambda_ReadOnly Access— 變更	Lambda 已更新允許委託人列出 AWS CloudFormation 堆疊的 AWSLambda_ReadOnly Access 政策。	2023 年 7 月 27 日
AWS Lambda 已開始追蹤變更	AWS Lambda 已開始追蹤其 AWS 管理的政策的變更。	2023 年 7 月 27 日

對 AWS Lambda 身分與存取進行疑難排解

請使用以下資訊來協助您診斷和修正使用 Lambda 和 IAM 時可能遇到的常見問題。

主題

- 我未獲授權，不得在 Lambda 中執行動作
- 我沒有授權執行 iam：PassRole
- 我想讓我以外的人員存取我AWS 帳戶的 Lambda 資源

我未獲授權，不得在 Lambda 中執行動作

如果您收到錯誤，告知您未獲授權執行動作，您的政策必須更新，允許您執行動作。

下列範例錯誤會在mateojackson IAM 使用者嘗試使用主控台檢視一個虛構 *my-example-widget* 資源的詳細資訊，但卻無虛構 lambda:*GetWidget* 許可時發生。

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 lambda:GetWidget on resource: my-example-widget
```

在此情況下，必須更新 mateojackson 使用者的政策，允許使用 lambda:*GetWidget* 動作存取 *my-example-widget* 資源。

如需任何協助，請聯絡您的 AWS 管理員。您的管理員提供您的登入憑證。

我沒有授權執行 iam：PassRole

如果您收到錯誤，告知您無權執行 iam:PassRole 動作，則必須更新您的政策，以允許您將角色傳遞至 Lambda。

有些 AWS 服務 允許您傳遞現有的角色至該服務，而無須建立新的服務角色或服務連結角色。如需執行此作業，您必須擁有將角色傳遞至該服務的許可。

當名為 marymajor 的 IAM 使用者嘗試使用主控台在 Lambda 中執行動作時，發生下列範例錯誤。但是，動作請求服務具備服務角色授予的許可。Mary 沒有將角色傳遞至該服務的許可。

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

在這種情況下，Mary 的政策必須更新，允許她執行 iam:PassRole 動作。

如需任何協助，請聯絡您的 AWS 管理員。您的管理員提供您的登入憑證。

我想讓我以外的人員存取我AWS 帳戶的 Lambda 資源

您可以建立一個角色，讓其他帳戶中的使用者或您的組織外部的人員存取您的資源。您可以指定要允許哪些信任物件取得該角色。針對支援基於資源的政策或存取控制清單 (ACL) 的服務，您可以使用那些政策來授予人員存取您的資源的許可。

如需進一步了解，請參閱以下內容：

- 若要了解 Lambda 是否支援這些功能，請參閱 AWS Lambda 搭配 IAM 的運作方式。
- 若要了解如何存取您擁有的所有 AWS 帳戶 所提供的資源，請參閱《IAM 使用者指南》中的將存取權提供給您所擁有的另一個 AWS 帳戶 中的 IAM 使用者。
- 如需了解如何將資源的存取權提供給第三方 AWS 帳戶，請參閱《IAM 使用者指南》中的將存取權提供給第三方擁有的 AWS 帳戶。
- 如需了解如何透過聯合身分提供存取權，請參閱《IAM 使用者指南》中的將存取權提供給在外部進行身分驗證的使用者 (聯合身分)。
- 若要了解使用角色和資源型政策進行跨帳戶存取之間的差異，請參閱《IAM 使用者指南》中的 IAM 角色與資源型政策的差異。

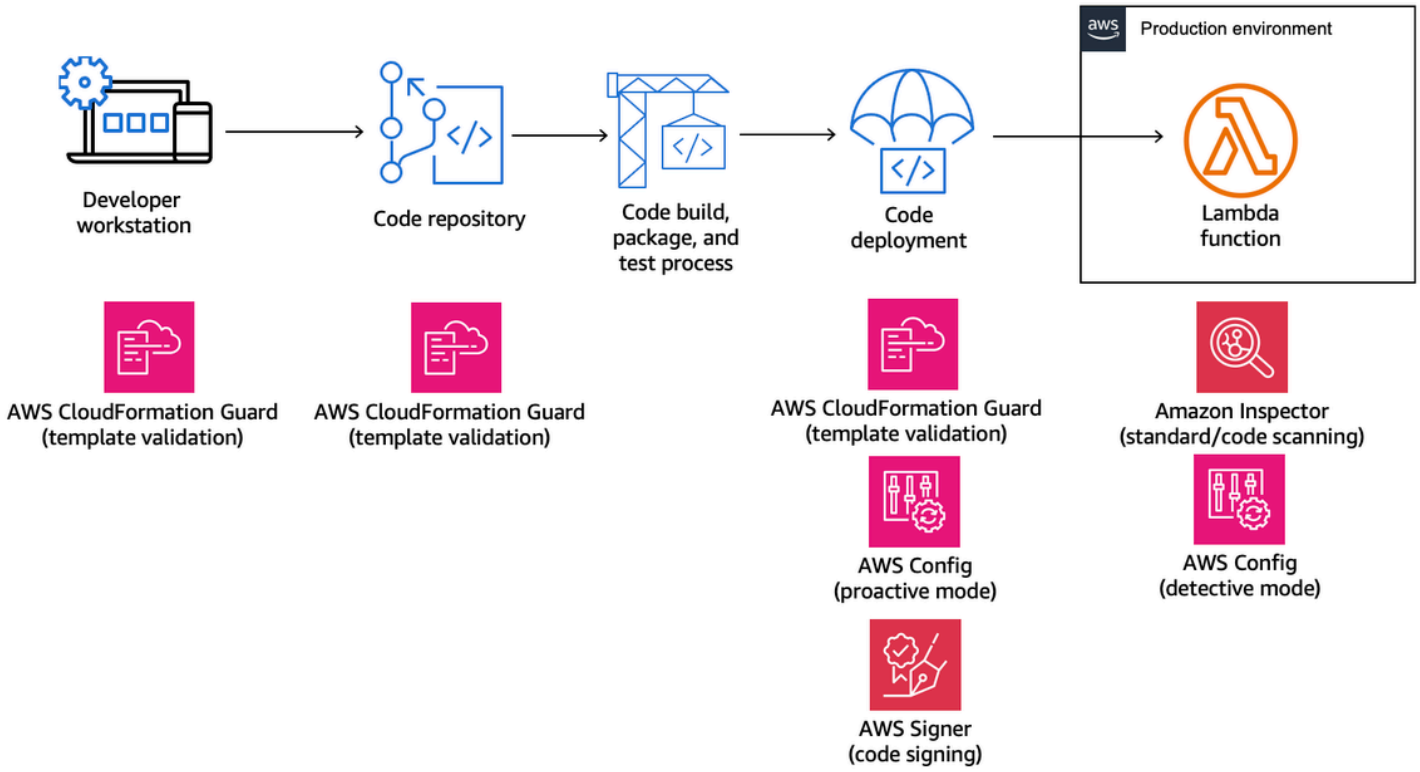
為 Lambda 函數和層建立治理策略

若要建置和部署無伺服器、雲端原生應用程式，您必須透過適當的控管和防護機制來確保敏捷性和加速上市。您要設定商業級別優先順序，可以強調敏捷性做為首要任務，或者透過控管、防護機制和控制項來強調風險規避。實際上，您不能採用「非此即彼」的策略，而要選擇能在軟體開發週期中平衡敏捷性和防護機制的「兼顧」策略。無論這些要求位於公司生命週期的哪個階段，控管功能都有可能成為您的流程和工具鏈當中的一項實作要求。

以下是一些組織可能為 Lambda 實作的控管控制項範例：

- Lambda 函數不可公開存取。
- Lambda 函數必須附加到 VPC。
- Lambda 函數不應使用已棄用的執行期。
- Lambda 函數必須有一組必要標籤進行標記。
- Lambda 層不可在組織外部存取。
- 附加安全群組的 Lambda 函數必須在函數和安全群組之間有配對的標籤。
- 具有附加層的 Lambda 函數必須使用核准的版本
- Lambda 環境變數必須使用客戶自管金鑰進行靜態加密。

下圖是深入控管策略的範例，此類策略會在軟體開發和部署過程當中實作控制項和政策：



下列主題說明如何針對新創公司和企業在組織中實作開發和部署 Lambda 函數的控制項。您的組織可能有現成的工具。下列主題採用模組化方法來實作這些控制項，可讓您挑選實際需要的元件。

主題

- 使用 AWS CloudFormation Guard 的 Lambda 主動式控制項

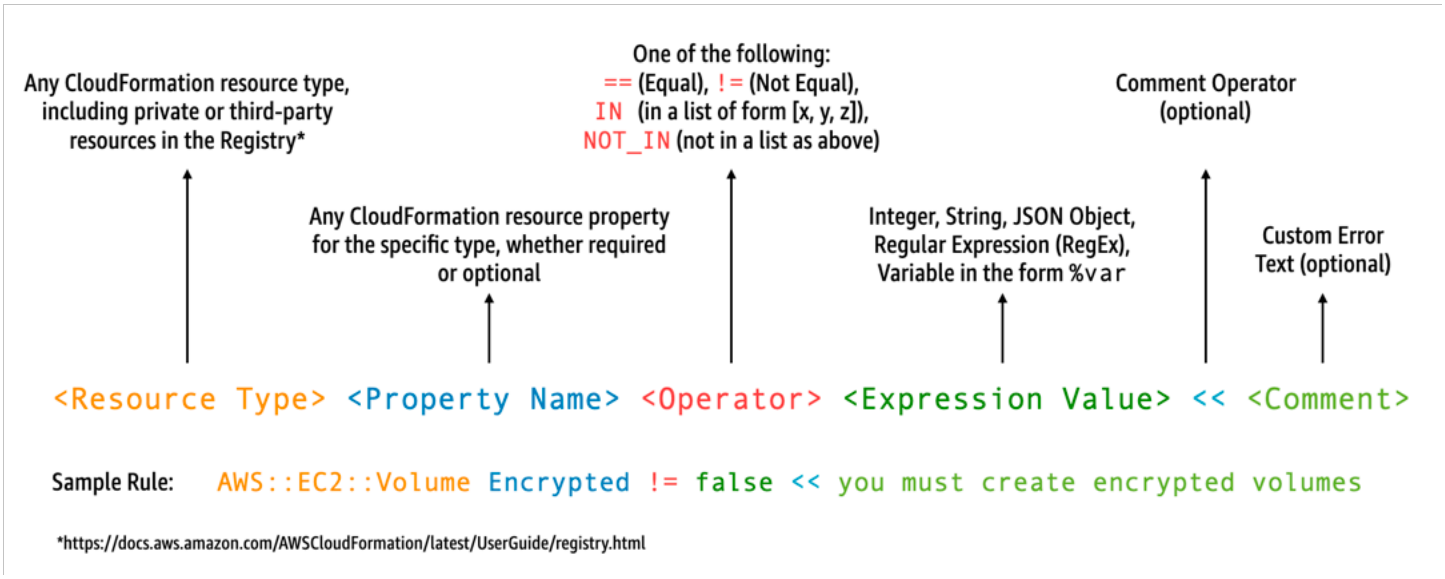
- 使用 Lambda 實作預防性控制 AWS Config
- 偵測不合規的 Lambda 部署和組態 AWS Config
- 使用 AWS Signer 的 Lambda 程式碼簽署
- 使用 Amazon Inspector 將 Lambda 的安全評估
- 實作 Lambda 安全性與合規性的可觀測性

使用 AWS CloudFormation Guard 的 Lambda 主動式控制項

AWS CloudFormation Guard是開放原始碼、一般用途的 policy-as-code 評估工具。透過比照政策規則驗證基礎設施即程式碼 (IaC) 範本和服務組合，它可被用於實現預防性控管與合規。這些規則還可以依據您的團隊或組織要求進行自訂。對於 Lambda 函數，Guard 規則可透過在建立或更新 Lambda 函數時定義所需的必要屬性設定，控制資源建立和組態更新。

合規管理員會定義部署和更新 Lambda 函數所需的控制項和控管政策清單。平台管理員在 CI/CD 管道中實作控制項，做為程式碼儲存庫的預遞交驗證 Webhook，並為開發人員提供用於在本機工作站驗證範本和程式碼的命令列工具。開發人員使用命令列工具編寫程式碼並驗證範本，然後將程式碼遞交到儲存庫。儲存庫隨後會在將這些程式碼部署到 AWS 環境前透過 CI/CD 管道自動對其進行驗證。

Guard 讓您可以使用特定領域的語言，如下編寫您的規則並實作您的控制項。



例如，假設您希望確定開發人員僅選擇最新的執行期。您可以指定兩種不同的政策，一種用來識別已棄用的執行期，另一種則用來確定即將棄用的執行期。為此，您可能要編寫以下 `etc/rules.guard` 檔案：

```
let lambda_functions = Resources.*[
    Type == "AWS::Lambda::Function"
]

rule lambda_already_deprecated_runtime when %lambda_functions !empty {
    %lambda_functions {
        Properties {
            when Runtime exists {
```

```
                Runtime !in ["dotnetcore3.1", "nodejs12.x", "python3.6", "python2.7",
 "dotnet5.0", "dotnetcore2.1", "ruby2.5", "nodejs10.x", "nodejs8.10", "nodejs4.3",
 "nodejs6.10", "dotnetcore1.0", "dotnetcore2.0", "nodejs4.3-edge", "nodejs"] <<Lambda
 function is using a deprecated runtime.>>
            }
        }
    }
}

rule lambda_soon_to_be_deprecated_runtime when %lambda_functions !empty {
    %lambda_functions {
        Properties {
            when Runtime exists {
                Runtime !in ["nodejs16.x", "nodejs14.x", "python3.7", "java8",
 "dotnet7", "go1.x", "ruby2.7", "provided"] <<Lambda function is using a runtime that
 is targeted for deprecation.>>
            }
        }
    }
}
```

現在假設您撰寫下列定義 Lambda 函數的`iac/lambda.yaml` CloudFormation 範本：

```
  Fn:
    Type: AWS::Lambda::Function
    Properties:
      Runtime: python3.7
      CodeUri: src
      Handler: fn.handler
      Role: !GetAtt FnRole.Arn
      Layers:
        - arn:aws:lambda:us-east-1:111122223333:layer:LambdaInsightsExtension:35
```

在安裝 Guard 公用程式後，驗證您的範本：

```
cfn-guard validate --rules etc/rules.guard --data iac/lambda.yaml
```

輸出如下：

```
lambda.yaml Status = FAIL
FAILED rules
rules.guard/lambda_soon_to_be_deprecated_runtime
```

```
---
Evaluating data lambda.yaml against rules rules.guard
Number of non-compliant resources 1
Resource = Fn {
  Type      = AWS::Lambda::Function
  Rule = lambda_soon_to_be_deprecated_runtime {
    ALL {
      Check =  Runtime not IN
 ["nodejs16.x","nodejs14.x","python3.7","java8","dotnet7","go1.x","ruby2.7","provided"]
 {
        ComparisonError {
          Message          = Lambda function is using a runtime that is targeted for
 deprecation.
          Error            = Check was not compliant as property [/Resources/
Fn/Properties/Runtime[L:88,C:15]] was not present in [(resolved, Path=[L:0,C:0]
 Value=["nodejs16.x","nodejs14.x","python3.7","java8","dotnet7","go1.x","ruby2.7","provided"])]
        }
          PropertyPath    = /Resources/Fn/Properties/Runtime[L:88,C:15]
          Operator        = NOT IN
          Value           = "python3.7"
          ComparedWith    =
 [["nodejs16.x","nodejs14.x","python3.7","java8","dotnet7","go1.x","ruby2.7","provided"]]
          Code:
               86.  Fn:
               87.    Type: AWS::Lambda::Function
               88.    Properties:
               89.      Runtime: python3.7
               90.      CodeUri: src
               91.      Handler: fn.handler

      }
    }
  }
}
```

Guard 可讓您的開發人員透過本機開發工作站即可了解，他們需要更新範本以便使用組織允許的執行期。這發生在遞交至程式碼儲存庫之前，隨後會使 CI/CD 管道中的檢查失敗。因此，您的開發人員將取得此回饋，以便了解如何開發合乎規範的範本，並且將他們的時間轉移到編寫可創造商業價值的程式碼。此控制可套用到本機的開發人員工作站，在預遞交驗證 Webhook 和/或在部署前的 CI/CD 管道中套用。

警告

如果您使用 AWS Serverless Application Model (AWS SAM) 範本定義 Lambda 函數，注意您需要如下更新 Guard 規則以便搜尋 `AWS::Serverless::Function` 資源類型。

```
let lambda_functions = Resources.*[
    Type == "AWS::Serverless::Function"
]
```

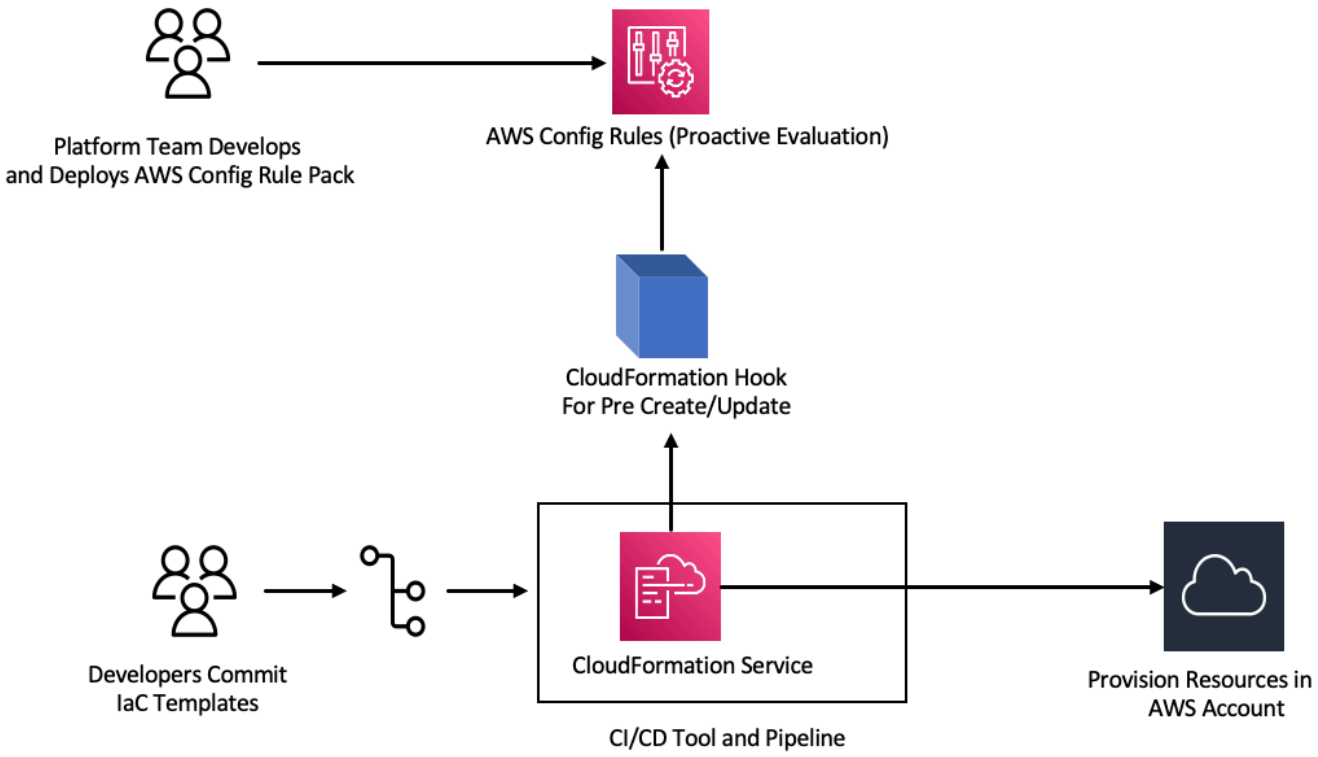
Guard 還預期屬性會包含在資源定義當中。同時，AWS SAM 範本允許在單獨的全域區段指定屬性。「全域」區段裡定義的屬性不使用您的 Guard 規則進行驗證。

如 Guard 故障診斷文件中所述，應注意 Guard 不支援簡寫形式的內部函數 (例如 `!GetAtt` 或 `!Sub`)，而要求使用展開形式：`Fn::GetAtt` 和 `Fn::Sub`。(先前的範例不會評估 Role 屬性，因此為了簡便起見而使用簡寫形式的內部函數。)

使用 Lambda 實作預防性控制 AWS Config

儘早在開發過程中確保無伺服器應用程式的合規性至關重要。在本主題中，我們將說明如何使用 AWS Config 實作預防性控制項。這可讓您在開發過程的早期實作合規檢查，並且使您可以在 CI/CD 管道中實作相同的控制項。這也會在集中管理的規則儲存庫中將您的控制項標準化，以便您可以在 AWS 帳戶之間一致地套用控制項。

例如，假設您的合規管理員定義了一項需求，以確保所有 Lambda 函數都包含 AWS X-Ray 追蹤。透 AWS Config過主動模式，您可以在部署之前對 Lambda 函數資源執行合規性檢查，降低部署設定不正確的 Lambda 函數的風險，並透過更快速地將基礎架構作為程式碼範本提供意見回饋，從而節省開發人員時間。以下是具 AWS Config有預防性控制的流程視覺化：



考慮所有 Lambda 函數都必須啟用追蹤功能的要求。作為回應，平台團隊確定是否需要在所有帳戶中主動執行特定 AWS Config 規則。此規則會將任何未設定 X-Ray 追蹤組態的 Lambda 函數標記為不合規資源。小組會開發規則、將其封裝在一致性套件中，並在所有帳戶中部署一致性套件，以確保組織中的所有 AWS 帳戶均勻套用這些控制項。您可以採用 AWS CloudFormation Guard 2.x.x 語法編寫該規則，它將採用以下形式：

```
rule name when condition { assertion }
```

以下是檢查以確保 Lambda 函數已啟用追蹤功能的範例 Guard 規則：

```
rule lambda_tracing_check {
  when configuration.tracingConfig exists {
      configuration.tracingConfig.mode == "Active"
  }
}
```

平台團隊會強制每個 AWS CloudFormation 部署都呼叫預先建立/更新勾點，以採取進一步的動作。他們會全權負責開發此勾點和設定管道、強化合規規則的集中控制，並確保在所有部署中一致地套用這些規則。若要開發、封裝和註冊勾點，請參閱 CloudFormation 命令列介面 (CFN-CLI) 文件中的開發 AWS CloudFormation 掛接。您可以使用 CloudFormation CLI 來建立掛接專案：

```
cfn init
```

此命令會要求您提供有關勾點專案的一些基本資訊，並在其中建立包含下列檔案的專案：

```
README.md
<hook-name>.json
rpdk.log
src/handler.py
template.yml
hook-role.yaml
```

做為勾點開發人員，您需要在 <hook-name>.json 組態檔案中新增所需的目標資源類型。在下列組態中，勾點設定為在使用建立任何 Lambda 函數之前執行 CloudFormation。您也可以為 preUpdate 和 preDelete 動作新增類似的處理常式。

```
    "handlers": {
        "preCreate": {
            "targetNames": [
                "AWS::Lambda::Function"
            ],
            "permissions": []
        }
    }
```

您還需要確保 CloudFormation 鉤子具有調 AWS Config 用 API 的適當權限。您可以透過更新名為 hook-role.yaml 的角色定義檔案來執行該動作。根據預設，角色定義檔案具有下列信任原則，可 CloudFormation 讓您擔任該角色。

```
      AssumeRolePolicyDocument:
        Version: '2012-10-17'
        Statement:
          - Effect: Allow
            Principal:
              Service:
                - hooks.cloudformation.amazonaws.com
                - resources.cloudformation.amazonaws.com
```

若要允許此勾點呼叫組態 API，您必須將下列許可新增至政策陳述式。然後，您可以使用cfn submit命令提交鉤子專案，在此命令為您 CloudFormation 建立具有所需權限的角色。

```
      Policies:
        - PolicyName: HookTypePolicy
          PolicyDocument:
            Version: '2012-10-17'
            Statement:
              - Effect: Allow
                Action:
                  - "config:Describe*"
                  - "config:Get*"
                  - "config:List*"
                  - "config:SelectResourceConfig"
                Resource: "*
```

接下來，您需要在 src/handler.py 檔案中編寫 Lambda 函數。在此檔案中，您將在啟動該專案時發現已建立的名為 preCreate、preUpdate 和 preDelete 的方法。您的目標是撰寫一個常見的、可重複使用的函式，以主動模式呼叫 AWS Config StartResourceEvaluation API 使用 AWS SDK for Python (Boto3). 此 API 呼叫會將資源屬性做為輸入，並且比照規則定義評估資源。

```
def validate_lambda_tracing_config(resource_type, function_properties:
 MutableMapping[str, Any]) -> ProgressEvent:
  LOG.info("Fetching proactive data")
  config_client = boto3.client('config')
  resource_specs = {
      'ResourceId': 'MyFunction',
      'ResourceType': resource_type,
```

```
      'ResourceConfiguration': json.dumps(function_properties),
      'ResourceConfigurationSchemaType': 'CFN_RESOURCE_SCHEMA'
  }
  LOG.info("Resource Specifications:", resource_specs)
  eval_response = config_client.start_resource_evaluation(EvaluationMode='PROACTIVE',
 ResourceDetails=resource_specs, EvaluationTimeout=60)
  ResourceEvaluationId = eval_response.ResourceEvaluationId
  compliance_response =
 config_client.get_compliance_details_by_resource(ResourceEvaluationId=ResourceEvaluationId)
  LOG.info("Compliance Verification:",
 compliance_response.EvaluationResults[0].ComplianceType)
  if "NON_COMPLIANT" == compliance_response.EvaluationResults[0].ComplianceType:
      return ProgressEvent(status=OperationStatus.FAILED, message="Lambda function
 found with no tracing enabled : FAILED", errorCode=HandlerErrorCode.NonCompliant)
  else:
      return ProgressEvent(status=OperationStatus.SUCCESS, message="Lambda function
 found with tracing enabled : PASS.")
```

現在，您可以從處理常式為預先建立的勾點呼叫常用的函數。以下是處理常式的範例：

```
@hook.handler(HookInvocationPoint.CREATE_PRE_PROVISION)
def pre_create_handler(
        session: Optional[SessionProxy],
        request: HookHandlerRequest,
        callback_context: MutableMapping[str, Any],
        type_configuration: TypeConfigurationModel
) -> ProgressEvent:
    LOG.info("Starting execution of the hook")
    target_name = request.hookContext.targetName
    LOG.info("Target Name:", target_name)
    if "AWS::Lambda::Function" == target_name:
        return validate_lambda_tracing_config(target_name,
            request.hookContext.targetModel.get("resourceProperties")
        )
    else:
        raise exceptions.InvalidRequest(f"Unknown target type: {target_name}")
```

完成此步驟後，您可以註冊鉤子並將其配置為監聽所有 AWS Lambda 函數創建事件。

開發人員會使用 Lambda 為無伺服器微服務準備基礎設施即程式碼 (IaC) 範本。此準備工作包括遵守內部標準，然後進行本機測試並將範本遞交至儲存庫。以下是範例 IaC 範本：

```
  MyLambdaFunction:
```

```
  Type: 'AWS::Lambda::Function'
  Properties:
    Handler: index.handler
    Role: !GetAtt LambdaExecutionRole.Arn
    FunctionName: MyLambdaFunction
    Code:
      ZipFile: |
        import json

        def handler(event, context):
            return {
                'statusCode': 200,
                'body': json.dumps('Hello World!')
            }
    Runtime: python3.8
    TracingConfig:
        Mode: PassThrough
    MemorySize: 256
    Timeout: 10
```

作為 CI/CD 程序的一部分，當 CloudFormation 範本部署時， CloudFormation服務會在佈建資源類型之前叫用預先建立/更新勾點。AWS::Lambda::Function掛接會利用以主動模式執行的 AWS Config 規則來驗證 Lambda 函數組態是否包含強制追蹤組態。來自勾點的回應將決定下一個步驟。如果符合標準，勾點會發出成功的訊號，並 CloudFormation 繼續佈建資源。如果沒有， CloudFormation 堆疊部署會失敗，管線會立即停止，系統會記錄詳細資料以供後續檢閱。合規通知將傳送至相關的利害關係人。

您可以在控制台中找到掛接成功/失敗信息： CloudFormation

Stack info | Events | Resources | Outputs | Parameters | Template | Change sets

Events (19)

Search events

Timestamp	Logical ID	Status	Status reason	Hook invocations
2023-08-29 23:50:23 UTC-0500	HookTestStack	ROLLBACK_COMPLETE	-	-
2023-08-29 23:50:22 UTC-0500	LambdaExecutionRole	DELETE_COMPLETE	-	-
2023-08-29 23:50:21 UTC-0500	MyApi	DELETE_COMPLETE	-	-
2023-08-29 23:50:20 UTC-0500	LambdaExecutionRole	DELETE_IN_PROGRESS	-	-
2023-08-29 23:50:20 UTC-0500	MyLambdaFunction	DELETE_COMPLETE	-	-
2023-08-29 23:50:20 UTC-0500	MyApi	DELETE_IN_PROGRESS	-	-
2023-08-29 23:50:18 UTC-0500	HookTestStack	ROLLBACK_IN_PROGRESS	The following resource(s) failed to create: [MyLambdaFunction]. Rollback requested by user.	-
2023-08-29 23:50:17 UTC-0500	MyLambdaFunction	CREATE_FAILED	The following hook(s) failed: [AWSSamples::LambdaTracingCheck::Hook]	-
2023-08-29 23:50:17 UTC-0500	MyLambdaFunction	CREATE_IN_PROGRESS	-	AWSSamples::LambdaTracingCheck::Hook
2023-08-29 23:50:16 UTC-0500	MyLambdaFunction	CREATE_IN_PROGRESS	-	AWSSamples::LambdaTracingCheck::Hook
2023-08-29 23:50:15 UTC-0500	MyLambdaFunction	CREATE_IN_PROGRESS	-	-
2023-08-29 23:50:14 UTC-0500	LambdaExecutionRole	CREATE_COMPLETE	-	-
2023-08-29 23:49:59 UTC-0500	MyApi	CREATE_COMPLETE	-	-
2023-08-29 23:49:59 UTC-0500	MyApi	CREATE_IN_PROGRESS	Resource creation Initiated	-
2023-08-29 23:49:58 UTC-0500	LambdaExecutionRole	CREATE_IN_PROGRESS	Resource creation Initiated	-
2023-08-29 23:49:58 UTC-0500	LambdaExecutionRole	CREATE_IN_PROGRESS	-	-
2023-08-29 23:49:58 UTC-0500	MyApi	CREATE_IN_PROGRESS	-	-
2023-08-29 23:49:55 UTC-0500	HookTestStack	CREATE_IN_PROGRESS	User Initiated	-
2023-08-29 23:49:50 UTC-0500	HookTestStack	REVIEW_IN_PROGRESS	User Initiated	-

如果您已為 CloudFormation 勾點啟用記錄，您可以擷取勾點評估結果。以下是狀態為失敗的勾點範例日誌，表示 Lambda 函數未啟用 X-Ray：

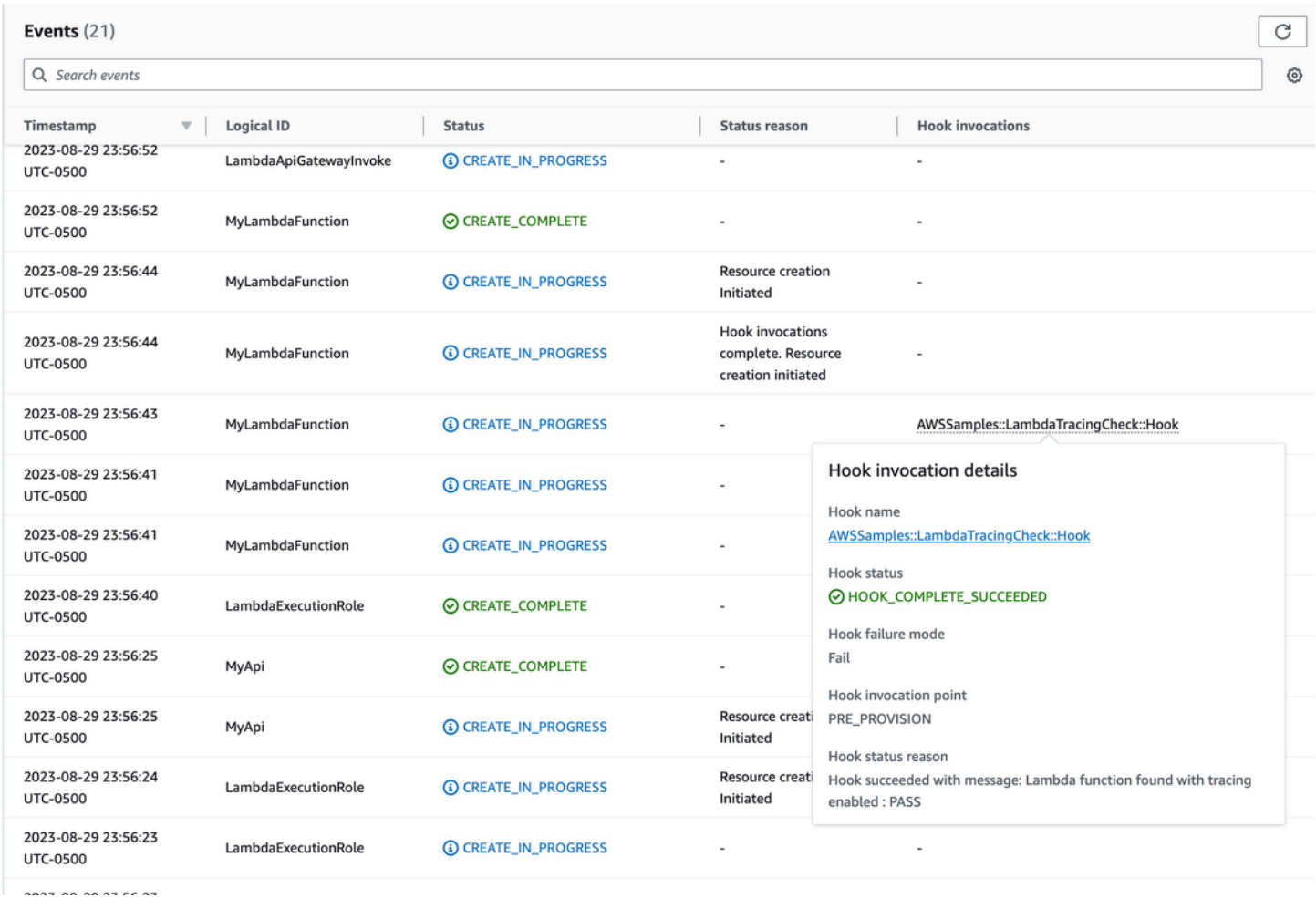
2023-08-29T23:50:17.574-05:00 ProgressEvent(status=<OperationStatus.FAILED: 'FAILED'>, errorCode=<HandlerErrorCode.NonCompliant: 'NonCompliant'…

```
ProgressEvent(status=<OperationStatus.FAILED: 'FAILED'>, errorCode=<HandlerErrorCode.NonCompliant: 'NonCompliant'>, message='Lambda
function found with no tracing enabled : FAILED', result=None, callbackContext=None, callbackDelaySeconds=0, resourceModel=None,
resourceModels=None, nextToken=None)
```

Copy

No newer events at this moment. *Auto retry paused.* Resume

如果開發人員選擇將 IaC 變更為將 `TracingConfig Mode` 值更新為 `Active` 並重新部署，勾點將成功執行且堆疊將繼續建立 Lambda 資源。



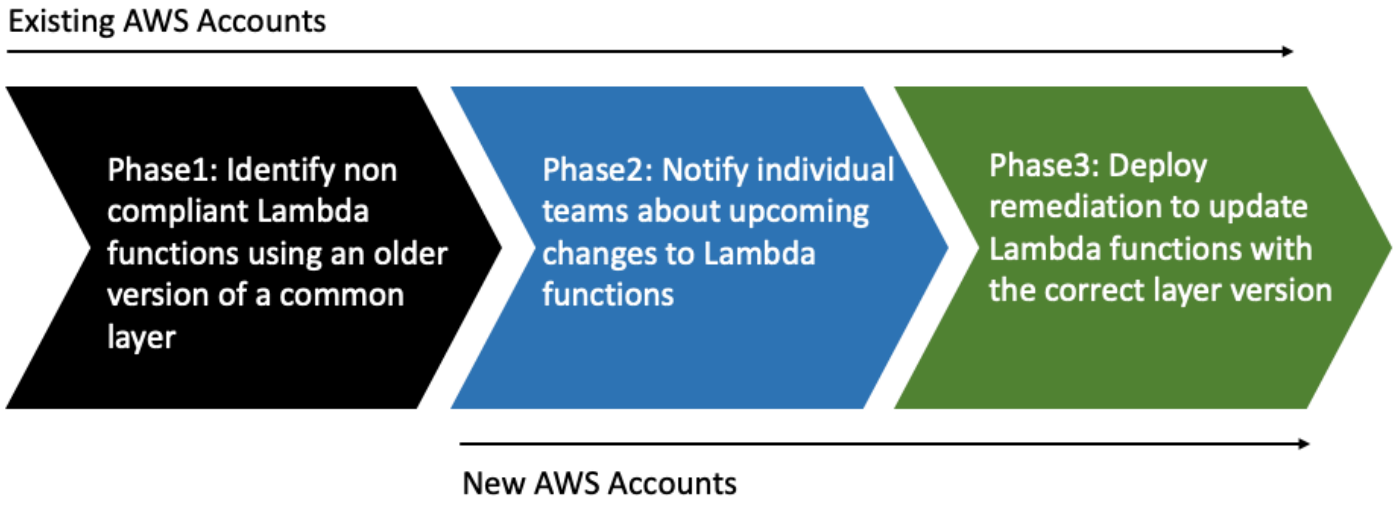
如此一來，您就可以在帳戶 AWS Config 中開發和部署無伺服器資源時，以主動模式實作預防性控制。AWS 透過將 AWS Config 規則整合到 CI/CD 管道，您可以識別並有針對性地封鎖不合規的資源部署，例如缺少主動追蹤組態的 Lambda 函數。如此可確保只有符合最新控管原則的資源才會部署到您的 AWS 環境中。

偵測不合規的 Lambda 部署和組態 AWS Config

除了主動評估之外，還 AWS Config 可以反動地偵測不符合您控管原則的資源部署和組態。這一點很重要，因為控管政策會隨著您的組織學習和實作新的最佳實務而演進。

請考慮在部署或更新 Lambda 函數時設定全新政策的情形：所有 Lambda 函數都必須始終使用特定且經過核准的 Lambda 層版本。您可以將 AWS Config 設為監控新的或更新函數的層組態。如果 AWS Config 偵測到未使用核准圖層版本的函數，則會將該函數標記為不相容的資源。您可以選擇配置 AWS Config 為自動修復資源，方法是使用自動 AWS Systems Manager 化文件指定修復動作。例如，您可以使用 Python 撰寫自動化文件 AWS SDK for Python (Boto3)，該文件會將不相容的函數更新為指向核准的圖層版本。因此， AWS Config 作為偵探和糾正控制，自動化合規管理。

我們將此過程分解為三個重要的實作階段：



階段 1：確定存取資源

首先 AWS Config 在您的帳戶中啟用，並將其設定為記錄 AWS Lambda 函數。這允許觀察 AWS Config 何時創建或更新 Lambda 函數。然後，您可以設定自訂政策規則，以檢查使用 AWS CloudFormation Guard 語法的特定策略違規情況。Guard 規則採用下列一般形式：

```
rule name when condition { assertion }
```

以下是一條範例規則，用於檢查確保層未被設為舊的層版本：

```
rule desiredlayer when configuration.layers !empty {
```

```
    some configuration.layers[*].arn != CONFIG_RULE_PARAMETERS.OldLayerArn
}
```

我們來了解一下規則語法和結構：

- 規則名稱：在提供的範例中，規則的名稱為 `desiredlayer`。
- 條件：此子句指定應該檢查規則的條件。在提供的範例中，條件為 `configuration.layers ! empty`。這意味著只有當組態中的 `layers` 屬性不為空時，才應評估資源。
- 聲明：在 `when` 子句後，聲明將決定檢查什麼規則。聲明 `some configuration.layers[*].arn != CONFIG_RULE_PARAMETERS.OldLayerArn` 會檢查任何 Lambda 層 ARN 是否與 `OldLayerArn` 值不相符。如果不相符，聲明為 true 且規則通過；否則，它將會失敗。

`CONFIG_RULE_PARAMETERS`是使用 AWS Config 規則配置的一組特殊參數。在這種情況下，`OldLayerArn` 是 `CONFIG_RULE_PARAMETERS` 內的一項參數。它允許使用者提供其認為舊的或已棄用的特定 ARN 值，然後，規則會檢查任何 Lambda 函數是否使用這個舊的 ARN 值。

階段 2：視覺化與設計

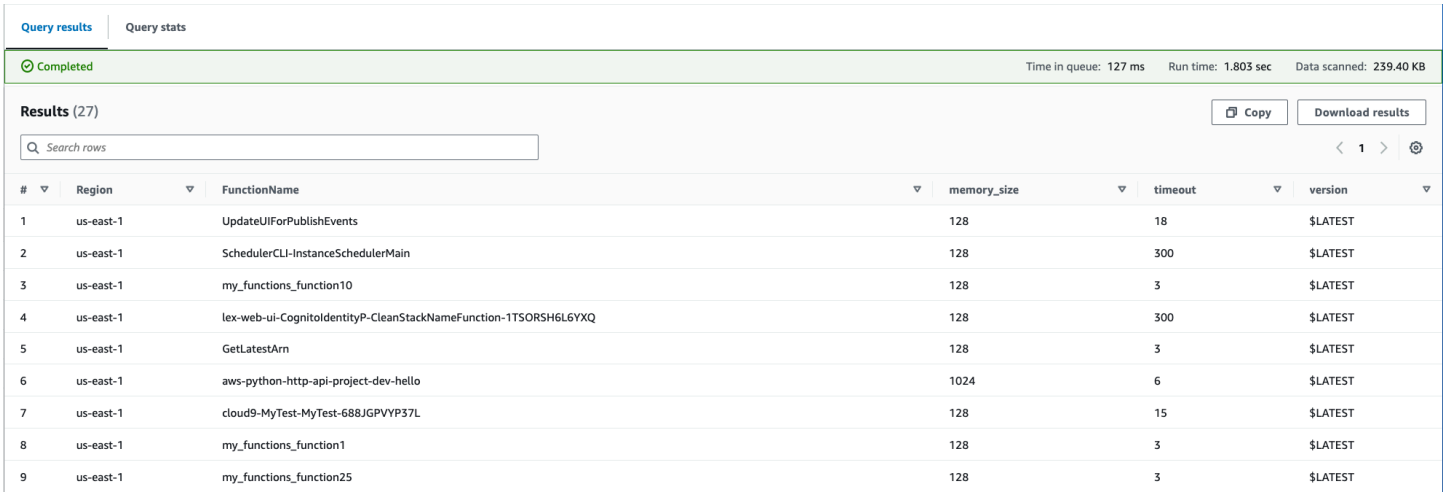
AWS Config 收集組態資料並將該資料存放在 Amazon Simple Storage Service (Amazon S3) 儲存貯體中。您可以使用 Amazon Athena 直接從 S3 儲存貯體查詢這些資料。藉助 Athena，您可以在組織層級彙整此類資料，為所有帳戶產生資源組態的整體全貌。若要設定資源組態資料的彙總，請參閱 AWS 雲端營運和管理部落格 QuickSight上的使用 Athena 和 Amazon 將資 AWS Config 料視覺化。

以下是使用特定層 ARN 識別所有 Lambda 函數的範例 Athena 查詢：

```
WITH unnested AS (
    SELECT
      item.awsaccountid AS account_id,
      item.awsregion AS region,
      item.configuration AS lambda_configuration,
      item.resourceid AS resourceid,
      item.resourcename AS resourcename,
      item.configuration AS configuration,
      json_parse(item.configuration) AS lambda_json
    FROM
      default.aws_config_configuration_snapshot,
      UNNEST(configurationitems) as t(item)
    WHERE
```

```
      "dt" = 'latest'
      AND item.resourcetype = 'AWS::Lambda::Function'
  )

  SELECT DISTINCT
    region as Region,
    resourcename as FunctionName,
    json_extract_scalar(lambda_json, '$.memorySize') AS memory_size,
    json_extract_scalar(lambda_json, '$.timeout') AS timeout,
    json_extract_scalar(lambda_json, '$.version') AS version
  FROM
    unnested
  WHERE
    lambda_configuration LIKE '%arn:aws:lambda:us-
east-1:111122223333:layer:AnyGovernanceLayer:24%'
```

以下是查詢的結果：



Query results | Query stats

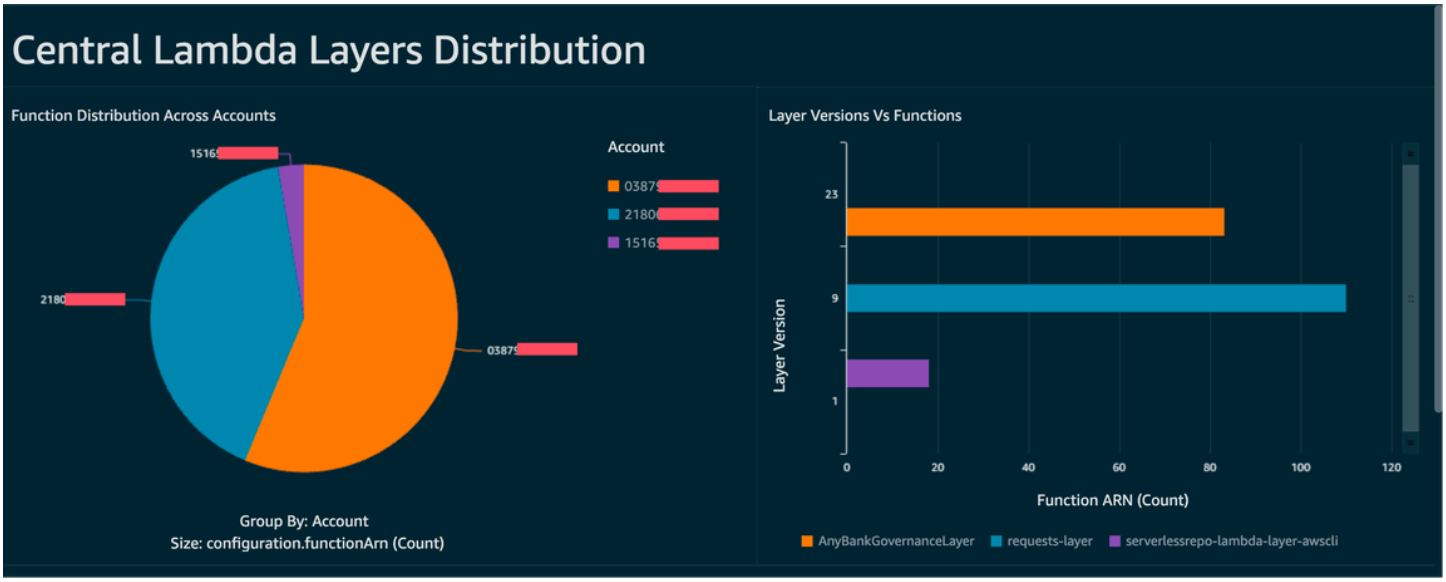
Completed — Time in queue: 127 ms — Run time: 1.803 sec — Data scanned: 239.40 KB

Results (27) — Copy — Download results

Search rows — 1

#	Region	FunctionName	memory_size	timeout	version
1	us-east-1	UpdateUIForPublishEvents	128	18	$LATEST
2	us-east-1	SchedulerCLI-InstanceSchedulerMain	128	300	$LATEST
3	us-east-1	my_functions_function10	128	3	$LATEST
4	us-east-1	lex-web-ui-CognitoIdentityP-CleanStackNameFunction-1TSORSH6L6YXQ	128	300	$LATEST
5	us-east-1	GetLatestArn	128	3	$LATEST
6	us-east-1	aws-python-http-api-project-dev-hello	1024	6	$LATEST
7	us-east-1	cloud9-MyTest-MyTest-688JGPVYP37L	128	15	$LATEST
8	us-east-1	my_functions_function1	128	3	$LATEST
9	us-east-1	my_functions_function25	128	3	$LATEST

透過整個組織彙總的 AWS Config 資料，您就可以使用 Amazon 建立儀表板 QuickSight。透過將您的 Athena 結果匯入 Amazon QuickSight，您可以視覺化 Lambda 函數如何遵守層版本規則。此儀表板可以突出顯示合規和不合規的資源，依下一個區段中所述協助您確定強制政策。下圖是一個範例儀表板，它會報告在組織內部套用至函數的層版本的分佈情況。



階段 3：實作與強制執行

您現在可以選擇性地將您在階段 1 中建立的層版本規則與透過 Systems Manager 自動化文件執行的修補動作配對，該自動化文件是您使用 AWS SDK for Python (Boto3)編寫的 Python 指令碼。此指令碼會針對每個 Lambda 函數呼叫UpdateFunction設定 API 動作，並使用新的層 ARN 更新函數組態。或者，您可以讓指令碼提交提取請求至程式碼儲存庫，以更新層 ARN。這樣，未來的程式碼部署也會使用正確的層 ARN 進行更新。

使用 AWS Signer 的 Lambda 程式碼簽署

AWS Signer 是一項全受管程式碼簽署服務，可讓您對照數位簽章驗證您的程式碼，以確認該程式碼未經變更且來自受信任的發布者。AWS Signer 可與 AWS Lambda 搭配使用，以便驗證函數和層在部署到您的 AWS 環境前未經變更。這可以保護您的組織免受惡意行為者侵害，這些行為者可能已取得憑證以建立新的或更新現有的函數。

若要為 Lambda 函數設定程式碼簽署，請先建立啟用版本控制的 S3 儲存貯體。之後，使用 AWS Signer 建立簽署設定檔，將 Lambda 指定為平台，然後指定簽署設定檔的有效天數。範例：

```
Signer:
  Type: AWS::Signer::SigningProfile
  Properties:
    PlatformId: AWSLambda-SHA384-ECDSA
    SignatureValidityPeriod:
      Type: DAYS
      Value: !Ref pValidDays
```

接下來使用簽署設定檔和 Lambda 來建立簽署組態。當簽署組態遇到與預期的數位簽章不相符的成品時，您必須指定要執行何種操作：警告 (但允許部署) 或強制執行 (並封鎖部署)。以下範例設為強制執行並封鎖部署。

```
SigningConfig:
  Type: AWS::Lambda::CodeSigningConfig
  Properties:
    AllowedPublishers:
      SigningProfileVersionArns:
        - !GetAtt Signer.ProfileVersionArn
    CodeSigningPolicies:
      UntrustedArtifactOnDeployment: Enforce
```

您現有的 AWS Signer 設定為由 Lambda 封鎖不受信任的部署。假設您已經完成功能請求的編碼，現在已準備好部署函數。第一個步驟是壓縮程式碼及適當相依性，然後使用您建立的簽署設定檔簽署成品。為此，您可以上傳壓縮成品到 S3 儲存貯體，然後開始簽署任務。

```
aws signer start-signing-job \
--source 's3={bucketName=your-versioned-bucket,key=your-prefix/your-zip-
artifact.zip,version=QyaJ3c4qa50LXV.9VaZgXHlsGbvCXxpT}' \
--destination 's3={bucketName=your-versioned-bucket,prefix=your-prefix/}' \
--profile-name your-signer-id
```

您將取得如下輸出，其中 jobId 是在目的地儲存貯體和字首中建立的物件，jobOwner 是執行該任務的 12 位數 AWS 帳戶 ID。

```
{
    "jobId": "87a3522b-5c0b-4d7d-b4e0-4255a8e05388",
    "jobOwner": "111122223333"
  }
```

現在，您可以使用已簽署的 S3 物件和您建立的程式碼簽署組態來部署您的函數。

```
  Fn:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: s3://your-versioned-bucket/your-prefix/87a3522b-5c0b-4d7d-
b4e0-4255a8e05388.zip
      Handler: fn.handler
      Role: !GetAtt FnRole.Arn
      CodeSigningConfigArn: !Ref pSigningConfigArn
```

或者，您還可以使用原始未簽署的來源壓縮成品測試函數部署。部署會失敗並顯示以下訊息：

```
Lambda cannot deploy the function. The function or layer might be signed using a
 signature that the client is not configured to accept. Check the provided signature
 for unsigned.
```

若您使用 AWS Serverless Application Model (AWS SAM) 建置與部署您的函數，套件命令將處理壓縮成品上傳到 S3，還會開始簽署任務並取得已簽署成品。您可以使用以下命令和參數來執行此動作：

```
sam package -t your-template.yaml \
--output-template-file your-output.yaml \
--s3-bucket your-versioned-bucket \
--s3-prefix your-prefix \
--signing-profiles your-signer-id
```

AWS Signer 會協助您確認部署到帳戶中的壓縮成品是否受信任用於部署。您可以在 CI/CD 管道中包含上述程序，並要求所有函數都使用先前主題中所述的技術附加程式碼簽署組態。透過搭配 Lambda 函數部署使用程式碼簽署，您可以防止可能已取得憑證的惡意行為者建立或更新函數，將惡意程式碼注入函數中。

使用 Amazon Inspector 將 Lambda 的安全評估

Amazon Inspector 是一項漏洞管理服務，可持續掃描工作負載，以尋找已知的軟體漏洞和意外的網路暴露。Amazon Inspector 會建立調查結果以描述漏洞、識別受影響的資源、評定漏洞嚴重性並提供修補指引。

Amazon Inspector 支援為 Lambda 函數和層提供持續的自動化安全性漏洞評估。Amazon Inspector 為 Lambda 提供兩種掃描類型：

- Lambda 標準掃描 (預設)：掃描 Lambda 函數及其層內的應用程式相依性，以尋找套件漏洞。
- Lambda 程式碼掃描：掃描函數和層內的自訂應用程式程式碼，以尋找程式碼漏洞。您可以啟動 Lambda 標準掃描，或同時啟動 Lambda 標準掃描和 Lambda 程式碼掃描。

若要啟用 Amazon Inspector，請導覽至 Amazon Inspector 主控台，展開設定區段，然後選擇帳戶管理。在帳戶索引標籤上，選擇啟動，然後選取其中一個掃描選項。

您可以為多個帳戶啟用 Amazon Inspector，並在設定 Amazon Inspector 時為特定帳戶的組織委派管理 Amazon Inspector 的許可。啟用時，您需要透過建立以下角色來授予 Amazon Inspector 許可：`AWSServiceRoleForAmazonInspector2`。Amazon Inspector 主控台允許您使用一鍵式選項建立此角色。

對於 Lambda 標準掃描，Amazon Inspector 會在下列情況下啟動 Lambda 函數的漏洞掃描。

- 一旦 Amazon Inspector 發現現有 Lambda 函數。
- 當您部署新的 Lambda 函數時。
- 當您對現有的 Lambda 函數或其層的應用程式程式碼或相依性部署更新時。
- 每當 Amazon Inspector 新增一個常見漏洞和暴露 (CVE) 項目到其資料庫，而該 CVE 與您的函數相關時。

對於 Lambda 程式碼掃描，Amazon Inspector 會使用自動推理和機器學習來評估您的 Lambda 函數應用程式程式碼，以分析您的應用程式程式碼是否符合整體安全規範。若 Amazon Inspector 在您的 Lambda 函數應用程式程式碼中偵測到漏洞，Amazon Inspector 會產生一份詳細的程式碼漏洞調查結果。如需可能偵測的清單，請參閱 Amazon 偵測 CodeGuru 器程式庫。

若要檢視調查結果，請前往 Amazon Inspector 主控台。在調查結果選單上，選擇依 Lambda 函數以顯示對 Lambda 函數執行的安全性掃描結果。

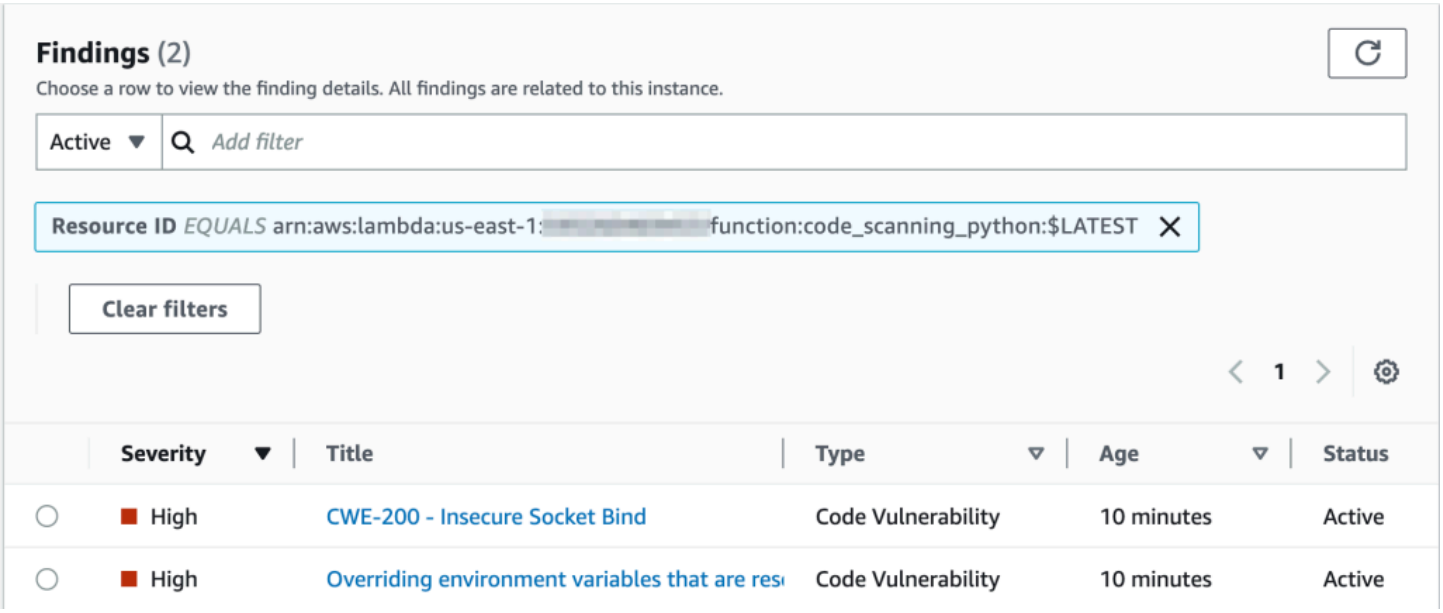
若要從標準掃描中排除 Lambda 函數，請使用以下鍵值對標記函數：

- `Key:InspectorExclusion`
- `Value:LambdaStandardScanning`

若要從程式碼掃描中排除 Lambda 函數，請使用以下鍵值對標記函數：

- `Key:InspectorCodeExclusion`
- `Value:LambdaCodeScanning`

例如，如下圖所示，Amazon Inspector 會自動偵測漏洞並將發現的漏洞分類為程式碼漏洞類型，這表示這些漏洞存在於函數的程式碼中，而不是在其中一個程式碼相依程式庫中。您可以一次檢查一個特定函數或多個函數的這些詳細資訊。



您可以深入研究每項結果，並了解如何修補問題。

Overriding environment variables that are reserved by AWS Lambda might lead to unexpected behavior.

Finding ID: arn:aws:inspector2:us-east-1:[redacted]:finding/[redacted]

Overriding environment variables that are reserved by AWS Lambda might lead to unexpected behavior or failure of the Lambda function.

Finding overview	
AWS account ID	[redacted]
Severity	High
Type	Code Vulnerability
Detector name	Override of reserved variable names in a Lambda function
Relevant CWE	--
Rule ID	Rule-434311
Detector tags	#availability, #aws-python-sdk, #aws-lambda, #data-integrity, #maintainability, #security, #security-context, #python
Fix available	Yes
Created at	March 29, 2023 10:08 AM (UTC-04:00)

Vulnerability details	
File path	lambda_function.py

Vulnerability location

```
3  import socket
4
5  def lambda_handler(event, context):
6
7      # print("Scenario 1");
8      os.environ['_HANDLER'] = 'hello'
9      # print("Scenario 1 ends")
10
11     # print("Scenario 2");
12     s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
13     s.bind(('',0))
```

Suggested remediation

Your code attempts to override an environment variable that is reserved by the Lambda runtime environment. This can lead to unexpected behavior and might break the execution of your Lambda function.

在使用您的 Lambda 函數時，確認您遵守 Lambda 函數的命名慣例。如需詳細資訊，請參閱 使用 Lambda 環境變數來設定程式碼中的值。

您要負責執行自己接受的修補建議。在接受前，請務必檢閱修補建議。您可能需要編輯修補建議，以確保您的程式碼符合您的預期。

實作 Lambda 安全性與合規性的可觀測性

AWS Config 是尋找和修復不合規的 AWS Serverless 資源的有用工具。您對無伺服器資源所做的每項變更都會記錄在 AWS Config 中。此外，AWS Config 還可讓您將組態快照資料存放在 S3 上。您可以使用 Amazon Athena 和 Amazon 製 QuickSight 作儀表板並查看AWS Config數據。在 偵測不合規的 Lambda 部署和組態 AWS Config 中，我們討論了如何視覺化 Lambda 層等特定組態。本主題將展開介紹這些概念。

Lambda 組態的可見性

您可以使用查詢來提取重要的組態，如 AWS 帳戶 ID、區域、AWS X-Ray 追蹤組態、VPC 組態、記憶體大小、執行期和標籤等。以下是您可以用來從 Athena 提取這些資訊的範例查詢：

```
WITH unnested AS (
    SELECT
      item.awsaccountid AS account_id,
      item.awsregion AS region,
      item.configuration AS lambda_configuration,
      item.resourceid AS resourceid,
      item.resourcename AS resourcename,
      item.configuration AS configuration,
      json_parse(item.configuration) AS lambda_json
    FROM
      default.aws_config_configuration_snapshot,
      UNNEST(configurationitems) as t(item)
    WHERE
      "dt" = 'latest'
      AND item.resourcetype = 'AWS::Lambda::Function'
  )

  SELECT DISTINCT
    account_id,
    tags,
    region as Region,
    resourcename as FunctionName,
    json_extract_scalar(lambda_json, '$.memorySize') AS memory_size,
    json_extract_scalar(lambda_json, '$.timeout') AS timeout,
    json_extract_scalar(lambda_json, '$.runtime') AS version
    json_extract_scalar(lambda_json, '$.vpcConfig.SubnetIds') AS vpcConfig
    json_extract_scalar(lambda_json, '$.tracingConfig.mode') AS tracingConfig
  FROM
    unnested
```

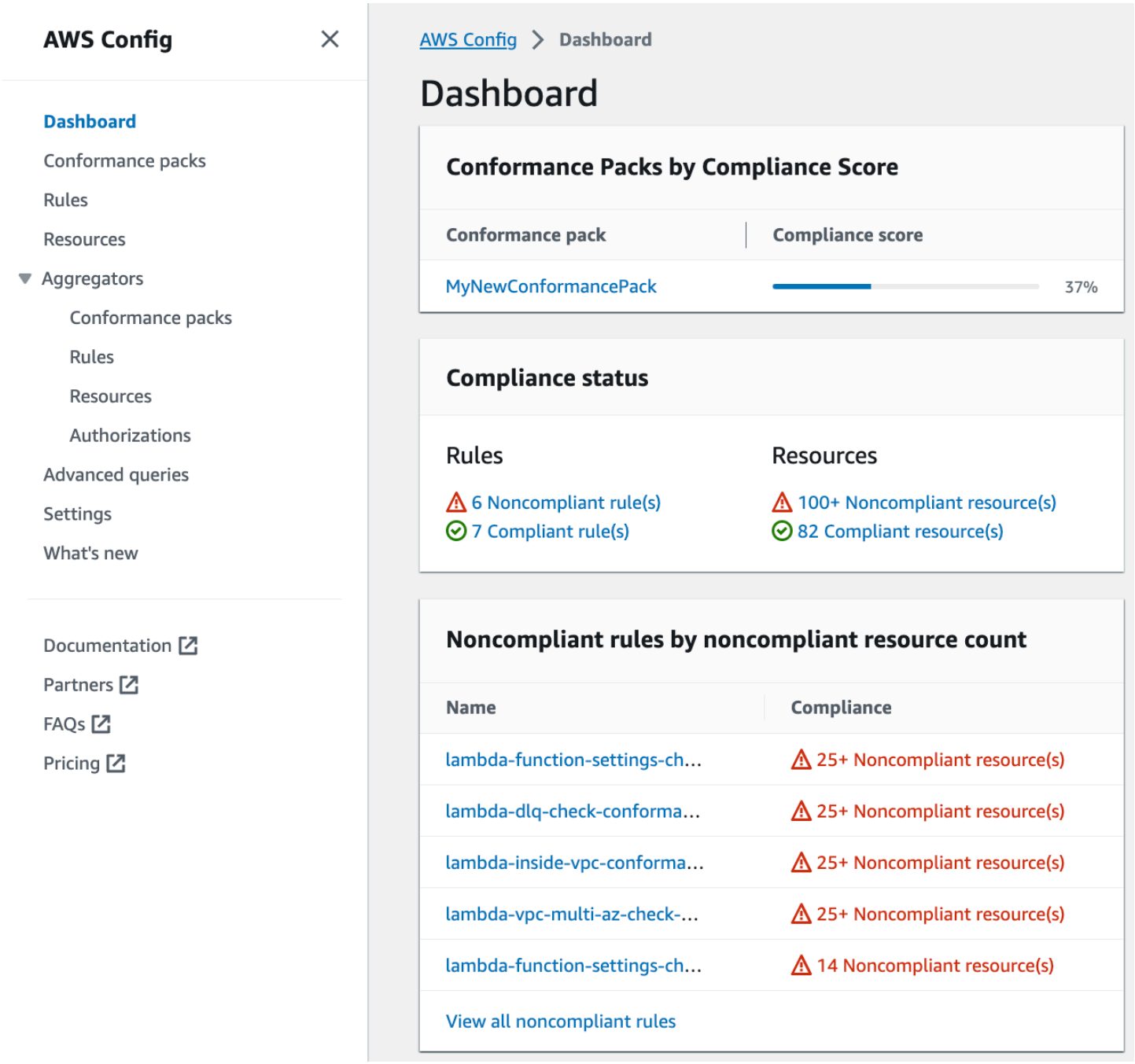
您可以使用查詢建立 Amazon QuickSight 儀表板並將資料視覺化。若要彙總AWS資源組態資料、在 Athena 建立表格，並在 Athena 的資料上建立 Amazon QuickSight 儀表板，請參閱AWS雲端營運和管理部落格 QuickSight上的使用 Athena 和 Amazon 將資AWS Config料視覺化。注意，此查詢還會擷取函數的標籤資訊。這可讓您更深入地了解工作負載和環境，尤其當您使用自訂標籤時。

如需有關可採取之動作的詳細資訊，請參閱本主題稍後的 處理可觀測性調查結果 區段。

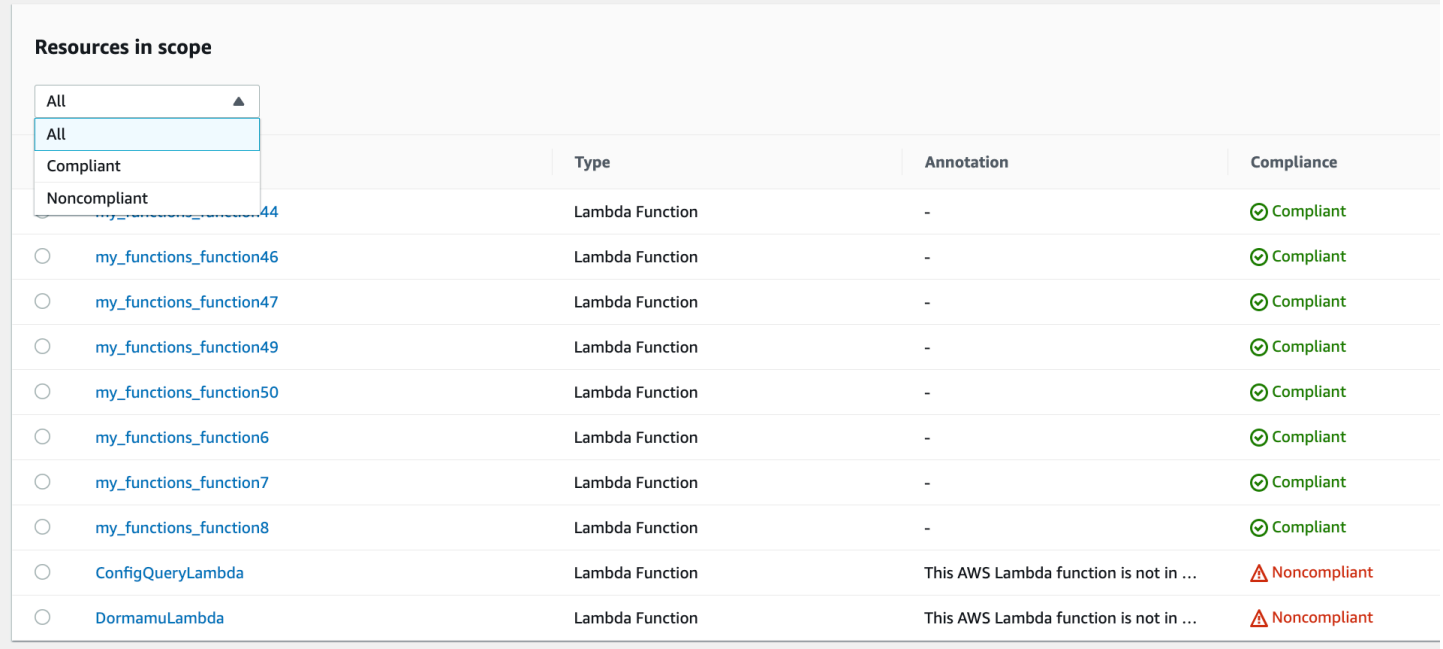
Lambda 合規可見性

使用由 AWS Config 產生的資料，您可以建立用於監控合規性的組織層級儀表板。這允許您一致地追蹤與監控：

- 依合規分數劃分的合規性套件
- 依不合規資源劃分的規則
- 合規狀態

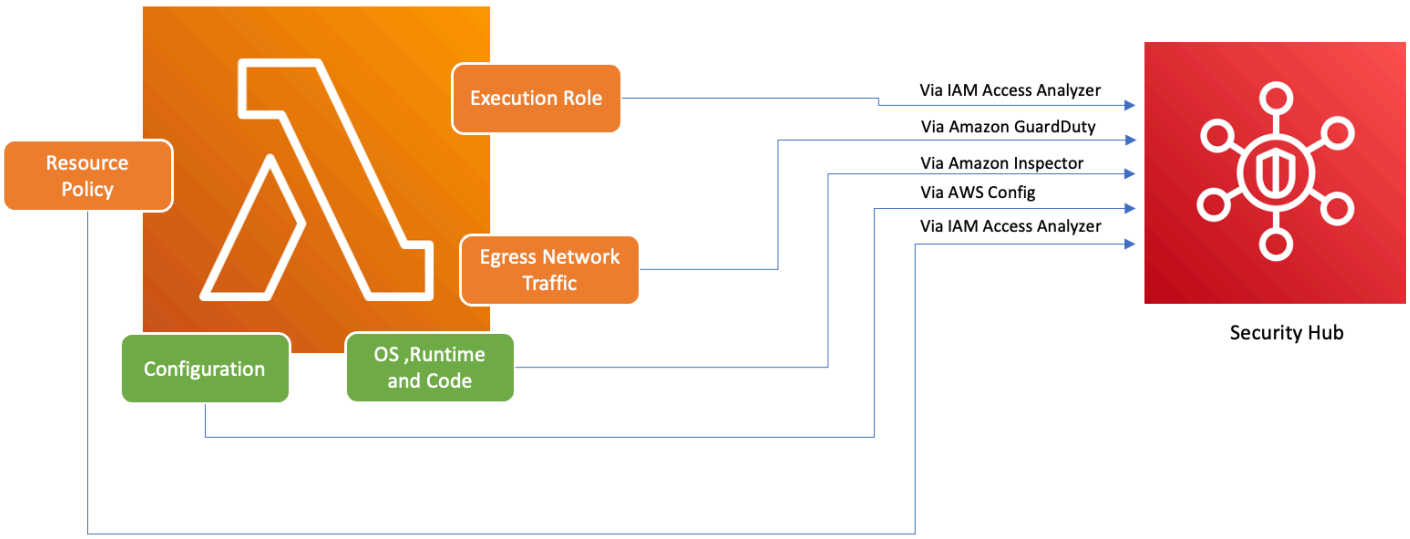


檢查每條規則，以便為該規則識別不合規的資源。例如，如果您的組織強制所有 Lambda 函數都必須與 VPC 相關聯，而且您已部署 AWS Config 規則來識別合規性，則可以在上述清單中選取 `lambda-inside-vpc` 規則。



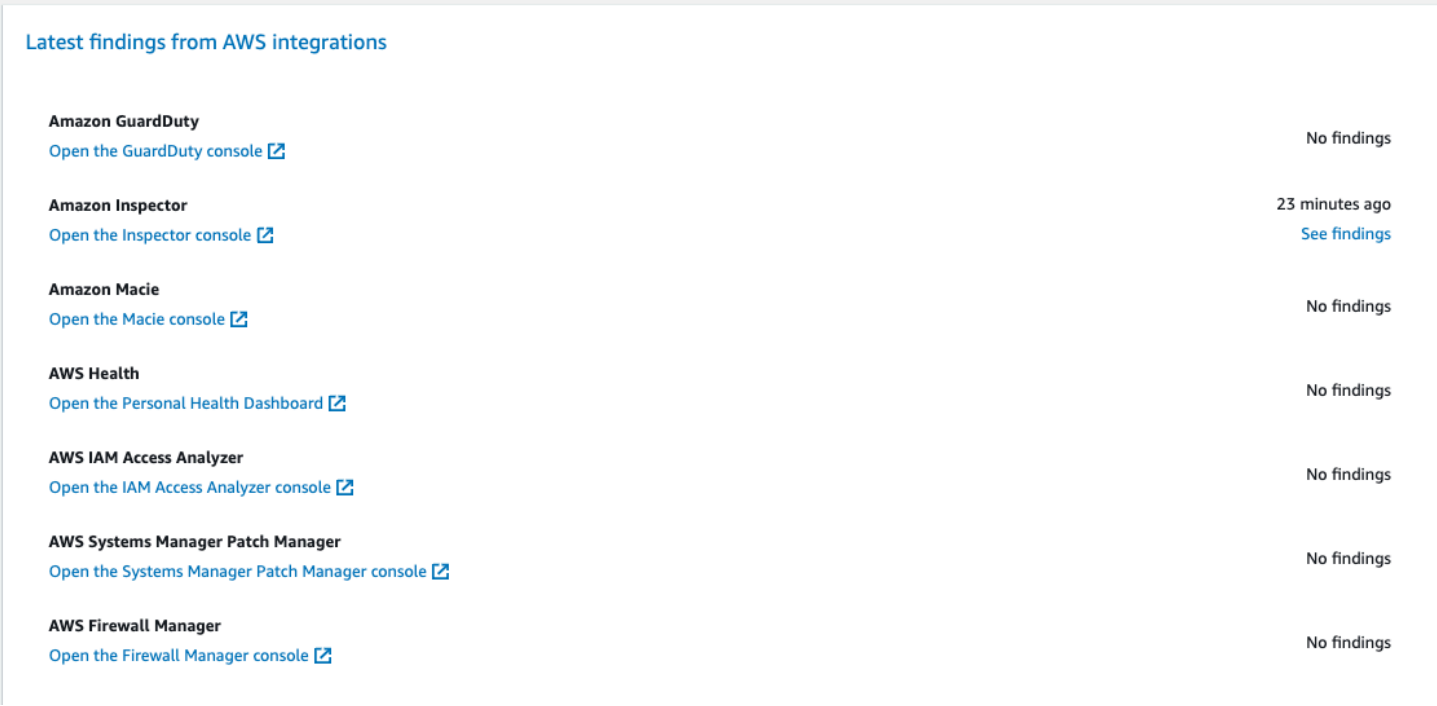
如需有關可採取之動作的詳細資訊，請參閱下方的 處理可觀測性調查結果 區段。

使用 Security Hub 的 Lambda 函數邊界可見性

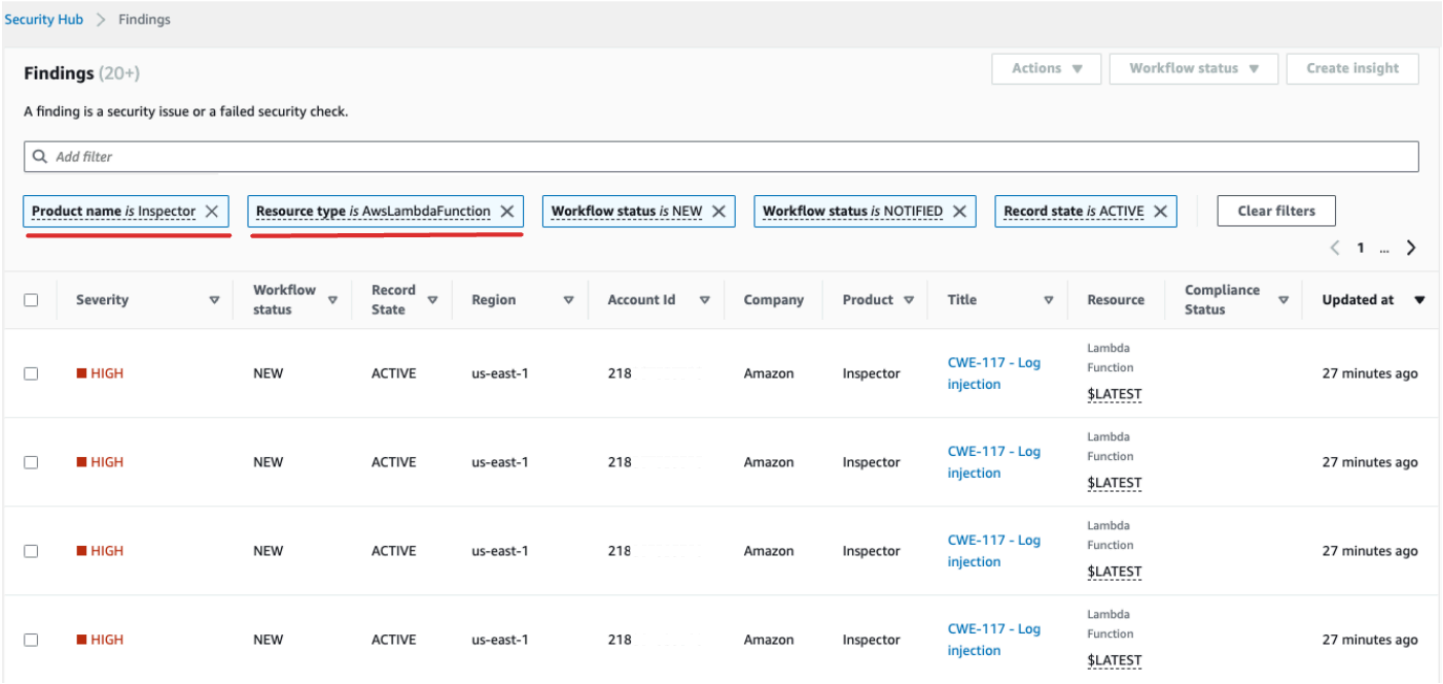


為了確保包括 Lambda 在內的 AWS 服務得以安全使用，AWS 引入了基礎安全最佳實務 v1.0.0。這組最佳實務提供清晰的指導方針以保護 AWS 環境中的資源和資料，強調維護強有力安全狀況的重要性。AWS Security Hub 透過提供統一的安全性與合規性中心對此進行補強。它彙總、組織和優先處理來自 Amazon Inspector 和 Amazon 等多個AWS服務的安全發現結果。AWS Identity and Access Management Access Analyzer GuardDuty

如果您有 Security Hub、Amazon Inspector、IAM 存取分析器，並在AWS組織內 GuardDuty 啟用，Security Hub 會自動彙總這些服務的發現項目。例如，我們來考慮一下 Amazon Inspector。使用 Security Hub，您可以有效地識別 Lambda 函數中的程式碼和套件漏洞。在 Security Hub 主控台中，導覽至標有來自 AWS 整合的最新調查結果的底部區段。您可以在此檢視和分析來自多項整合 AWS 服務的調查結果。



若要查看詳細資訊，請選擇第二欄中的查看調查結果連結。這會顯示依產品 (如 Amazon Inspector) 篩選的調查結果清單。若要將搜尋限定於 Lambda 函數，請將 `ResourceType` 設為 `AwsLambdaFunction`。這會顯示 Amazon Inspector 中與 Lambda 函數相關的調查結果。



對於 GuardDuty，您可以識別可疑的網路流量模式。此類異常可能表明 Lambda 函數中存在潛在的惡意程式碼。

藉助 IAM Access Analyzer，您可以檢查政策，特別是那些包含條件陳述式的政策，這些陳述式會授予函數存取外部實體的權限。此外，IAM 存取分析器會評估使用 Lambda API 中的AddPermission作業時所設定的許可和`EventSourceToken`.

處理可觀測性調查結果

考慮到 Lambda 函數可能的廣泛組態及其獨特要求，一種標準化的自動修補解決方案可能不適用於所有情況。此外，在各種環境中，變更的實作方式也存在差異。如果您遇到任何看似不合規的組態，請考慮下列指導方針：

1. 標記策略

   我們建議實作全面的標記策略。每個 Lambda 函數都應使用關鍵資訊進行標記，例如：

   - 擁有者：對函數負責的人員或團隊。
   - 環境：生產、整備、開發或沙盒。
   - 應用程式：此函數所屬的更廣泛上下文 (如果適用)。

2. 擁有者外展

有別於自動執行重大變更 (如 VPC 組態調整)，主動聯絡不合規函數的擁有者 (透過擁有者標籤識別) 可為其提供充足的時間，以便其：

- 調整 Lambda 函數的不合規組態。
- 提供說明並請求例外狀況，或完善合規標準。

3. 維護組態管理資料庫 (CMDB)

   雖然標籤可以提供直接上下文，但維護集中式 CMDB 可以提供更深入的洞見。它可以保留有關每個 Lambda 函數的更精細資訊、它的相依性和其他關鍵中繼資料。CMDB 是稽核、合規性檢查和識別函數擁有者的重要資源。

隨著無伺服器基礎設施的不斷演進，採用主動的監控策略變得至關重要。使用 AWS Config、Security Hub 和 Amazon Inspector 等工具，可以快速識別潛在的異常或不合規的組態。然而，單純使用工具無法確保完全的合規性或最佳化組態。將這些工具與妥善記載的程序及最佳實務進行配對至關重要。

- 回饋迴圈：一旦執行修補步驟，務必確保有回饋迴圈。這意味著定期重新檢視不合規的資源，以確定它們是否已更新或仍存在相同的問題。
- 記錄：始終記錄可觀測性、所執行的動作以及任何授予的例外狀況。適當記錄不僅有助於稽核，還可協助強化該程序以便在未來提高合規性與安全性。
- 培訓和認知：確保所有利害關係人，尤其是 Lambda 函數擁有者，定期接受培訓並了解最佳實務、組織政策以及合規性要求。定期舉辦研討會、網路研討會或提供培訓課程，可進一步確保每個人在安全性和合規性方面取得共識。

總之，儘管工具和技術提供了強大的偵測與標記潛在問題的功能，但人的因素 (理解、溝通、培訓和記錄) 仍然是關鍵。它們共同構成一個強效的組合，可確保您的 Lambda 函數和更廣泛的基礎設施保持合規、安全並針對您的商業需求提供最佳化。

AWS Lambda 的合規驗證

在多個 AWS 合規計劃中，第三方稽核人員會評估 AWS Lambda 的安全與合規。這些計劃包括 SOC、PCI、FedRAMP、HIPAA 等等。

如需特定合規計劃範圍內的 AWS 服務清單，請參閱合規計劃內的 AWS 服務。如需一般資訊，請參閱 AWS 合規計劃。

您可使用 AWS Artifact 下載第三方稽核報告。如需詳細資訊，請參閱在 AWS Artifact 中下載報告。

您使用 Lambda 的合規責任，取決於資料的機密性、您公司的合規目標及適用法律和法規。您可以實作控管控制項，以確保公司的 Lambda 函數符合您的合規需求。如需更多詳細資訊，請參閱 為 Lambda 函數和層建立治理策略。

AWS Lambda 中的恢復能力

AWS 全球基礎設施是以 AWS 區域與可用區域為中心建置的。AWS區域提供多個分開且隔離的實際可用區域，它們以低延遲、高輸送量和高度備援聯網功能相互連結。透過可用區域，您所設計與操作的應用程式和資料庫，就能夠在可用區域之間自動容錯移轉，而不會發生中斷。可用區域的可用性、容錯能力和擴充能力，均較單一或多個資料中心的傳統基礎設施還高。

如需 AWS 區域與可用區域的詳細資訊，請參閱 AWS 全球基礎設施。

除了 AWS 全球基礎設施，Lambda 還提供數種功能，可協助支援資料的彈性和備份需求。

- 版本控制 - 您可以在 Lambda 中使用版本控制，在開發時儲存您函數的程式碼和組態。搭配別名，您可以使用版本控制來執行藍綠和輪流部署。如需詳細資訊，請參閱 Lambda 函數版本。
- 擴展 - 當您的函數在處理先前請求的同時收到請求，Lambda 會啟動函數的另一個執行個體來處理增加的負載。Lambda 會自動擴展以處理每個區域 1,000 個並行執行，如果需要，可以增加配額。如需詳細資訊，請參閱 了解 Lambda 展函數。
- 高可用性 - Lambda 會在多個可用區域中執行您的函數，確保單一區域的服務中斷時，其可用於處理事件。如果您將函式設定為連接到您帳戶中的虛擬私有雲端 (VPC)，請在多個可用區域中指定子網路，以確保高可用性。如需詳細資訊，請參閱 讓 Lambda 函數存取 Amazon VPC 中的資源。
- 預留並行 - 若要確保您的函數可隨時擴展以處理額外的請求，您可以為其預留並行。為函式設定預留並行，確保其可擴展為 (但不超過) 指定的並行叫用數目。這可確保您不會因為取用所有可用並行的其他函式而遺失請求。如需詳細資訊，請參閱 為函數配置保留並發。

- 重試 - 對於非同步叫用以及其他服務所觸發的叫用子集，Lambda 會在發生錯誤時重試，且重試之間有延遲。同步叫用函式的其他用戶端和 AWS 服務負責執行重試。如需詳細資訊，請參閱 了解 Lambda 中的重試行為。
- 無效字母佇列 - 對於非同步叫用，您可以設定 Lambda，以便在所有重試都失敗時，將請求傳送到無效字母佇列。無效字母佇列是可接收事件以便排除故障或重新處理的 Amazon SNS 主題或 Amazon SQS 佇列。如需詳細資訊，請參閱 無效字母佇列。

AWS Lambda 中的基礎設施安全

作為一種受管服務，AWS Lambda 受 AWS 全域網路安全的保護。如需 AWS 安全服務以及 AWS 如何保護基礎設施的相關資訊，請參閱 AWS 雲端安全。若要使用基礎設施安全性的最佳實務來設計您的 AWS 環境，請參閱安全支柱 AWS 架構良好的框架中的基礎設施保護。

您可使用 AWS 發佈的 API 呼叫，透過網路來存取 Lambda。用戶端必須支援下列項目：

- Transport Layer Security (TLS)。我們需要 TLS 1.2 並建議使用 TLS 1.3。
- 具備完美轉送私密(PFS)的密碼套件，例如 DHE (Ephemeral Diffie-Hellman)或 ECDHE (Elliptic Curve Ephemeral Diffie-Hellman)。現代系統(如 Java 7 和更新版本)大多會支援這些模式。

此外，請求必須使用存取金鑰 ID 和與 IAM 主體相關聯的私密存取金鑰來簽署。或者，您可以使用 AWS Security Token Service (AWS STS) 以產生暫時安全憑證以簽署請求。

監控與疑難排解 Lambda 函數

AWS Lambda 與其他 AWS 服務整合，協助您監控 Lambda 函數並進行疑難排解。Lambda 會代表您自動監控 Lambda 函數，並透過 Amazon 報告指標 CloudWatch。為了協助您在程式碼執行時對其進行監控，Lambda 會自動追蹤請求的次數、每個請求的調用持續時間、以及導致錯誤的請求次數。

您可以使用其他 AWS 服務對 Lambda 函數進行疑難排解。本節描述了如何使用這些 AWS 服務來監控、追蹤、偵錯及疑難排解您的 Lambda 函數和應用程式。如需有關每個執行階段中函數記錄和錯誤的詳細資訊，請參閱個別執行階段區段。

如需監控 Lambda 應用程式的詳細資訊，請參閱無伺服器園地中的監控和可觀測性。

章節

- 監控 Lambda 主控台上的函數
- 使用 Lambda 函數指標
- 使用 Amazon CloudWatch 日誌 AWS Lambda
- 使用記錄 AWS Lambda API 呼叫 AWS CloudTrail
- 使用視覺化 Lambda 函數叫用 AWS X-Ray
- 使用 Amazon CloudWatch Lambda 見解監控功能效
- 搭配 Lambda 函數使用 CodeGuru 效能分析工具
- 使用其他 AWS 服務的範例工作流程

監控 Lambda 主控台上的函數

Lambda 服務會代表您監控功能，並將指標傳送至 Amazon CloudWatch。Lambda 主控台會建立這些指標的監控圖表，並將其顯示在每個 Lambda 函數的 Monitoring (監控) 頁面上。

Lambda 主控台提供指標、日誌和追蹤的單一窗格檢視。主控台提供適用於全部窗格的時間範圍、時區和重新整理選項的篩選器。您可以輕鬆地關聯指標、日誌和追蹤，在疑難排解 Lambda 函數中的錯誤時，減少平均復原時間 (MTTR)。

定價

CloudWatch 擁有永久免費方案。除了免費方案閾值之外，還會 CloudWatch 收取指標、儀表板、警示、記錄和深入解析的費用。如需詳細資訊，請參閱 Amazon CloudWatch 定價。

使用 Lambda 主控台

您可以在 Lambda 主控台上監控您的 Lambda 函數和應用程式。

監控函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇 監控 索引標籤。

監測圖表的類型

下一節描述了 Lambda 主控台上的監控圖表。

Lambda 監控圖表

- Invocations (調用) - 調用函數的次數。
- Duration (持續時間) – 您的函數程式碼在處理事件時所花費的平均、最短、和最長時間。
- Error count and success rate (%) (錯誤計數和成功率 (%)) - 錯誤數以及完成時未發生錯誤的調用百分比。
- Throttles (調節) - 由於並行限制而導致調用失敗的次數。
- IteratorAge— 對於串流事件來源，指 Lambda 收到批次並叫用函數時，批次中最後一個項目的存留時間。

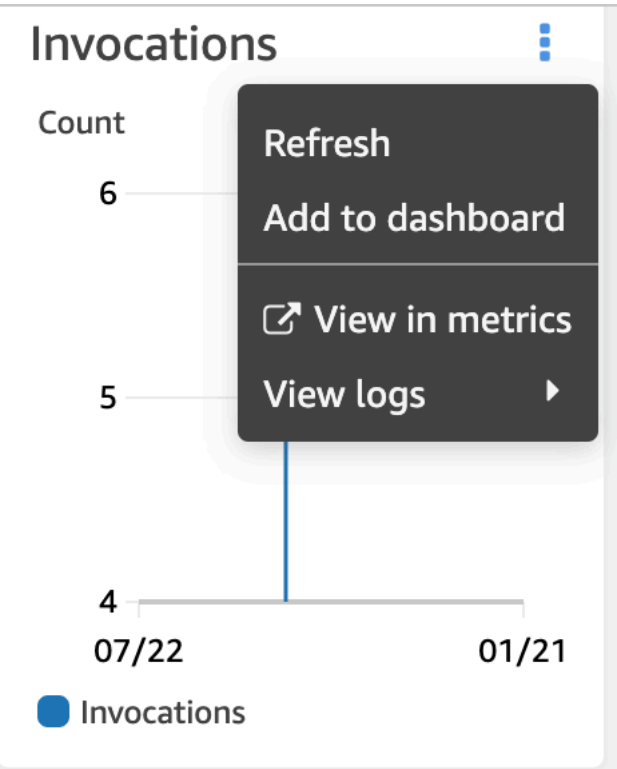
- Async delivery failures (非同步傳送失敗) - 當 Lambda 嘗試寫入至目的地或無效字母佇列時發生的錯誤數目。
- Concurrent executions (並行執行) - 處理事件的函數執行處理數目。

在 Lambda 主控台上檢視圖表

下節說明如何在 Lambda 主控台上檢視 CloudWatch 監控圖形，以及如何開啟指 CloudWatch 標儀表板。

檢視函數的監控圖表

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇 監控 索引標籤。
4. 從預先定義的時間範圍中選擇，或選擇自訂的時間範圍。
5. 若要查看中圖形的定義 CloudWatch，請選擇三個垂直點 (Widget 動作)，然後選擇「在量度中檢視」，在 CloudWatch 主控台上開啟「量度」儀表板。

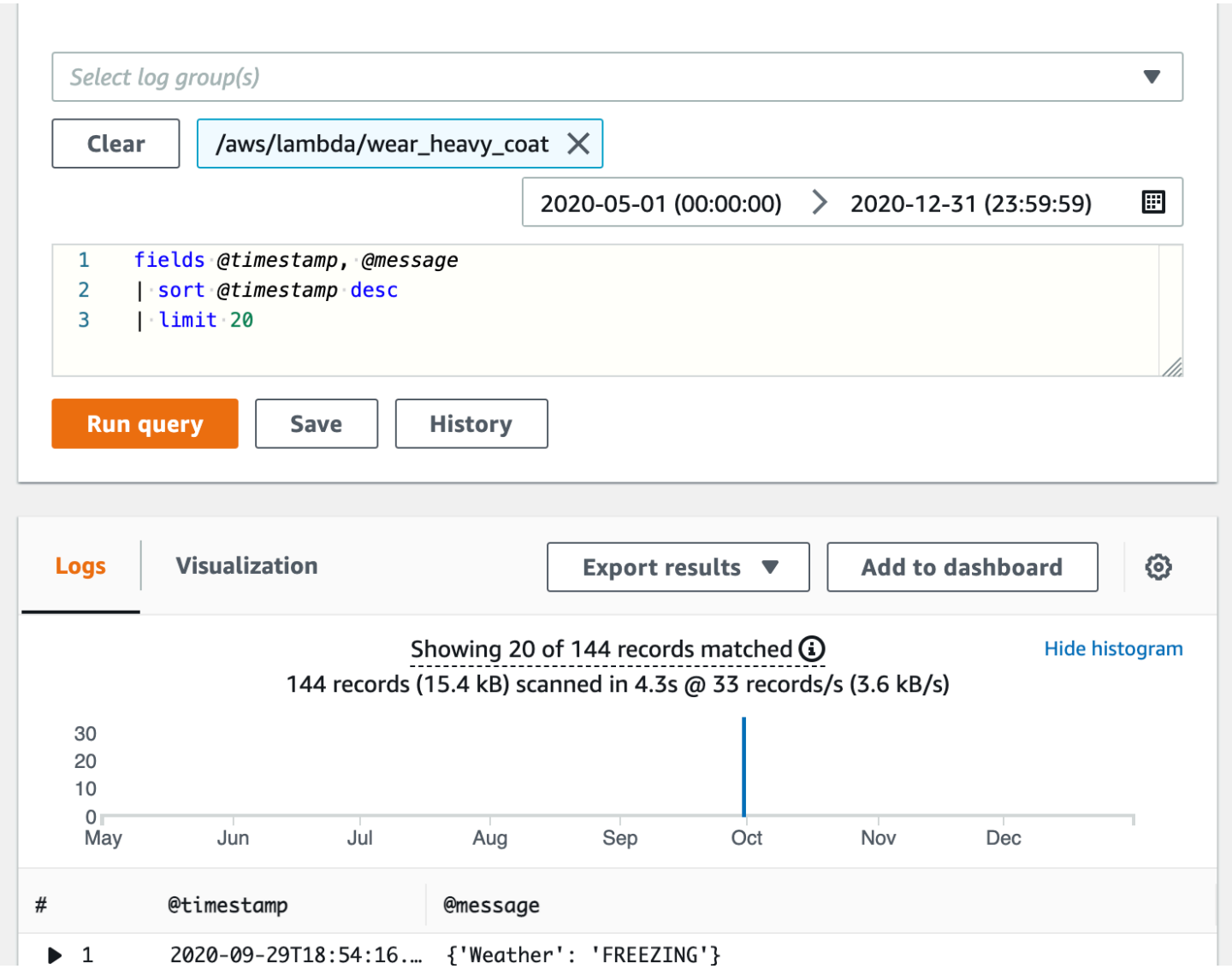


在 CloudWatch 記錄主控台上檢視查詢

下節說明如何在記錄主控台上檢視 CloudWatch Logs Insights 的報告，並將其新增至自訂儀表板。CloudWatch

檢視函數報告

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇 監控 索引標籤。
4. 選擇 [檢視登入] CloudWatch。
5. 選擇 View in Logs Insights (檢視日誌洞見)。
6. 從預先定義的時間範圍中選擇，或選擇自訂的時間範圍。
7. 選擇 Run query (執行查詢)。
8. (選用) 選擇 Save (儲存)。



後續步驟？

- 瞭解 Lambda 記錄和傳送 CloudWatch 的指標使用 Lambda 函數指標。
- 了解如何使用 CloudWatch Lambda 深入解析來收集和彙總 Lambda 函數執行階段效能指標和記錄使用 Amazon CloudWatch Lambda 見解監控功能效。

使用 Lambda 函數指標

當您的 AWS Lambda 函數完成處理事件時，Lambda 會將有關呼叫的指標傳送至 Amazon CloudWatch。啟用這些指標不會產生額外費用。

在 CloudWatch 主控台上，您可以使用這些指標建立圖形和儀表板。您可以設定警示來回應使用率、效能或錯誤率變更。Lambda 會以 1 分鐘 CloudWatch 的間隔傳送指標資料至。如需要更快速清楚了解 Lambda 函數，您可以建立高解析度 自訂指標 (如無伺服器園地所述)。自訂指標和 CloudWatch警示需支付費用。如需詳細資訊，請參閱 Amazon CloudWatch 定價。

此頁面說明主控台上可用的 Lambda 函數叫用、效能和並行指標。 CloudWatch

章節

- 在 CloudWatch 主控台上檢視指標
- 指標類型

在 CloudWatch 主控台上檢視指標

您可以使用主 CloudWatch 控台，依函數名稱、別名或版本來篩選和排序函數量度量。

若要在 CloudWatch 主控台上檢視指標

1. 開啟主 CloudWatch 控台的「指標」頁面 (AWS/Lambda命名空間)。
2. 在瀏覽索引標籤的指標下，選擇下列任一維度：
   - 依函數名稱 (FunctionName) - 檢視函數所有版本和別名的彙總指標。
   - 依資源 (Resource) - 檢視函數版本或別名的指標。
   - 執行版本 (ExecutedVersion) - 檢視別名與版本組合的指標。使用 ExecutedVersion 維度來比較兩個函式版本的錯誤率，這兩個版本都是加權別名的目標。
   - 跨所有函數 (無) — 檢視目前所有函數的彙總度量 AWS 區域。
3. 選擇指標，然後選擇新增至圖表或其他繪製選項。

根據預設，圖表會針對所有指標使用 Sum 統計資料。若要選擇不同的統計資料並自訂圖表，請使用 圖表化指標 標籤中的選項。

> **Note**
>
> 指標上的時間戳記會反映呼叫函數的時間。根據調用的持續時間，這可能是發出指標前的幾分鐘。例如，如果函數逾時 10 分鐘，請查看過去 10 分鐘之前以獲取精確的指標。

如需有關的詳細資訊 CloudWatch，請參閱 Amazon CloudWatch 使用者指南。

指標類型

下節說明 CloudWatch主控台上可用的 Lambda 指標類型。

呼叫指標

調用指標是 Lambda 函數調用結果的二進制指標。例如，若函數傳回錯誤，Lambda 會傳送具有值 1 的 `Errors` 指標。若要取得每分鐘發生的函數錯誤數目，請以一分鐘為其檢視 `Errors` 指標的 `Sum`。

> **Note**
>
> 檢視下列含有 `Sum` 統計資料的調用指標。

- `Invocations` – 您的函數程式碼被調用的次數，包含成功調用和造成函數錯誤的調用。如果呼叫請求受到調節，或導致呼叫錯誤，系統就不會記錄呼叫。`Invocations` 的值等於計費的請求數目。
- `Errors` - 導致函數錯誤的呼叫數目。函數錯誤包含程式碼擲回的例外，以及 Lambda 執行時間擲回的例外。執行時間會針對如逾時和組態錯誤等問題傳回錯誤。若要計算錯誤率，將 `Errors` 的值除以 `Invocations` 的值。請注意，錯誤指標上的時間戳記會反映調用函數的時間，而不是錯誤發生的時間。
- `DeadLetterErrors` - 如為非同步調用，則為 Lambda 嘗試傳送事件至無效字母佇列 (DLQ) 但失敗的次數。無效信件錯誤可能設定錯誤的資源或大小限制而發生。
- `DestinationDeliveryFailures`：如為非同步調用和支援的 事件來源映射，則為 Lambda 嘗試傳送事件至 目的地 但失敗的次數。事件來源映射方面，Lambda 支援串流來源 (DynamoDB 和 Kinesis) 的目的地。傳遞錯誤可能因權限錯誤、設定錯誤的資源或大小限制而發生。如果您設定的目的地是不受支援的類型 (例如 Amazon SQS FIFO 佇列或 Amazon SNS FIFO 主題)，也可能發生此錯誤。
- `Throttles` - 調節的調用請求次數。當所有函數執行個體正在處理請求且沒有並行可供擴展規模時，Lambda 會使用 `TooManyRequestsException` 錯誤拒絕其他請求。限流的請求和其他調用錯誤不會計為 `Invocations` 或 `Errors`。

- `OversizedRecordCount`：針對 Amazon DocumentDB 事件來源，您的函數從變更串流接收到大小超過 6 MB 的事件數量。Lambda 會捨棄訊息並發出此指標。
- `ProvisionedConcurrencyInvocations` - 使用佈建並行調用函數程式碼的次數。
- `ProvisionedConcurrencySpilloverInvocations` - 當所有佈建並行都處於使用中狀態時，使用標準並行調用函數程式碼的次數。
- `RecursiveInvocationsDropped`— Lambda 因偵測到您的函數是無限遞迴迴圈的一部分而停止呼叫函數的次數。 使用 Lambda 遞迴迴路偵測來防止無限迴圈透過追蹤支援 AWS SDK 新增的中繼資料，監控函數作為請求鏈一部分的呼叫次數。如果函數作為請求鏈的一部分被調用超過 16 次，則 Lambda 會放棄下一次調用。

效能指標

效能指標提供單一函數調用的效能詳細資料。例如，`Duration` 指標表示您的函式處理事件時以豪秒計算的時間。若要了解您的函數如何處理事件，請檢視這些有 `Average` 或 `Max` 統計資料的指標。

- `Duration` - 您的函數程式碼處理一個事件時所花費的時間。調用的計費期間是 `Duration` 四捨五入到最接近毫秒的值。`Duration` 不包含冷啟動時間。
- `PostRuntimeExtensionsDuration` - 在函數程式碼完成後，執行時間在執行程式碼進行擴展時所用的累計時間。
- `IteratorAge`：針對 DynamoDB、Kinesis 和 Amazon DocumentDB 事件來源，則為事件中最後一筆記錄的留存期。這個指標用來衡量串流接收記錄到事件來源映射將事件傳送至函數之間的時間。
- `OffsetLag` – 若為 Apache Kafka 和 Amazon Managed Streaming for Apache Kafka (Amazon MSK) 事件來源，則為寫入主題的最後一筆記錄與函數取用者群組處理的最後一筆記錄之間的偏移量差異。雖然 Kafka 主題會有多個分區，但這個指標可以在主題層級測量偏移延遲。

`Duration` 也支援百分位數 (p) 統計資料。使用百分位數排除偏差 `Average` 和 `Maximum` 統計資料的離群值。例如，p95 統計資料會顯示調用的最大持續時間為 95%，排除最慢的 5%。如需詳細資訊，請參閱 Amazon CloudWatch 使用者指南中的百分位數。

並行指標

Lambda 會將並行指標報告為跨函數、版本、別名或 AWS 區域處理事件的執行個體數彙總計數。若要查看您與達到並行限制的接近程度，請使用 `Max` 統計資料檢視這些指標。

- `ConcurrentExecutions` - 處理事件的函數執行個體數目。如果此數目達到區域的並行執行配額，或是函數的保留並行上限，Lambda 就會限流額外的調用請求。

- ProvisionedConcurrentExecutions - 使用佈建並行處理事件的函數執行個體數目。針對具有佈建並行之別名或版本的每次調用，Lambda 都會發出目前的計數。
- ProvisionedConcurrencyUtilization— 對於版本或別名，值ProvisionedConcurrentExecutions除以已配置的佈建並行總量。例如，如果您為函數配置佈建的並行 10，而您的ProvisionedConcurrentExecutions是 7，則您的ProvisionedConcurrencyUtilization是 0.7。
- UnreservedConcurrentExecutions - 若為區域，則是沒有保留並行的函數所處理的事件數目。
- ClaimedAccountConcurrency – 對於區域，無法用於隨需調用的並行數量。ClaimedAccountConcurrency 等於 UnreservedConcurrentExecutions 加上配置並行的數量 (亦即預留並行總數加上佈建並行總數)。如需詳細資訊，請參閱 使用 ClaimedAccountConcurrency 指標。

非同步調用指標

非同步調用指標提供有關來自事件來源和直接調用的非同步調用詳細資料。您可以設定閾值和警示，以便在發生某些變更時通知您。例如排入處理佇列的事件數目意外增加時 (AsyncEventsReceived)。或是某事件已等待處理很長一段時間時 (AsyncEventAge)。

- AsyncEventsReceived – Lambda 成功排入處理佇列的事件數目。此指標可讓您深入了解 Lambda 函數接收的事件數量。監控此指標並設定閾值警示以檢查問題。例如，偵測傳送至 Lambda 的不必要事件數量，並快速診斷因不正確的觸發程序或函數組態所造成的問題。AsyncEventsReceived 和 Invocations 之間的不相符項目可能表示處理差異、捨棄的事件或潛在的待處理佇列。
- AsyncEventAge – Lambda 成功將事件排入佇列到調用函數之間的時間。因調用失敗或限流而重試事件時，此指標的值會增加。監控此指標，並針對發生佇列累積時的不同統計資料設定閾值警示。若要對此指標的增加情形進行疑難排解，請查看 Errors 指標以識別函數錯誤，並查看 Throttles 指標以找出並行問題。
- AsyncEventsDropped – 在未成功執行函數的情況下捨棄的事件數目。如果您有設定無效字母佇列 (DLQ) 或 OnFailure 目的地，則事件會在捨棄之前傳送至該處。有多種原因會導致事件遭捨棄。例如，事件可能超過事件存留期上限、用盡重試次數上限，或是預留並行可能設定為 0。若要對捨棄事件的原因進行疑難排解，請查看 Errors 指標以識別函數錯誤，並查看 Throttles 指標來找出並行問題。

使用 Amazon CloudWatch 日誌 AWS Lambda

AWS Lambda 代表您自動監控 Lambda 函數，協助您疑難排解函數中的故障。只要函數的執行角色具有必要的許可，Lambda 就會擷取函數處理的所有請求的日誌，並將其傳送至 Amazon CloudWatch 日誌。

您可以在您的程式碼中插入記錄陳述式，以協助驗證您的程式碼如預期運作。Lambda 會自動與 CloudWatch 日誌整合，並將程式碼中的所有日 CloudWatch 誌傳送到與 Lambda 函數相關聯的日誌群組。

根據預設，Lambda 會將日誌傳送到名為 `/aws/lambda/<function name>` 的日誌群組。如果您希望函數將日誌傳送到另一個群組，可以使用 Lambda 主控台、 AWS Command Line Interface (AWS CLI) 或 Lambda API 進行設定。如需進一步了解，請參閱the section called “設定 CloudWatch 記錄群組”。

您可以使用 Lambda 主控台、主控台、 AWS Command Line Interface (AWS CLI) 或 CloudWatch API 來檢視 Lambda 函數的記錄。 CloudWatch

> **Note**
>
> 在函數調用後，日誌可能需要 5 到 10 分鐘才會顯示。

章節

必要條件

您的執行角色需要將日誌上傳到 CloudWatch 日誌的權限。您可以使用 Lambda 提供的`AWSLambdaBasicExecutionRole` AWS 受管政策新增 CloudWatch 記錄權限。執行以下命令，將此政策新增至您的角色：

```
aws iam attach-role-policy --role-name your-role --policy-arn arn:aws:iam::aws:policy/
service-role/AWSLambdaBasicExecutionRole
```

如需詳細資訊，請參閱 the section called “AWS 受管理政策”。

定價

使用 Lambda 記錄無須額外付費；不過，需支付標準 CloudWatch 記錄費用。如需詳細資訊，請參閱CloudWatch 定價。

設定 Lambda 函數的進階日誌控制項

為了讓您更妥善地控制您擷取、處理和使用函數日誌的方式，Lambda 提供下列日誌組態選項：

- 日誌格式 - 在純文字和結構化 JSON 格式之間為您的日誌進行選擇
- 記錄層級-針對 JSON 結構化記錄，選擇 Lambda 傳送至的記錄詳細資料層級 CloudWatch，例如「錯誤」、「除錯」或「資訊」
- 日誌組-選擇您的功能發送日誌的日誌組 CloudWatch

設定 JSON 和純文字日誌格式

將日誌輸出擷取為 JSON 索引鍵值組可以在偵錯函數時更容易搜尋和篩選。使用 JSON 格式的日誌檔，您也可以在日誌檔中新增標籤和內容資訊。這可以幫助您對大量日誌資料執行自動分析。除非您的開發工作流程仰賴使用純文字 Lambda 日誌的現有工具，否則建議您為日誌格式選取 JSON。

對於所有 Lambda 受管執行階段，您可以選擇是以非結構化純文字還是 JSON 格式將函數的系統 CloudWatch 記錄傳送至記錄。系統日誌是 Lambda 產生的日誌，有時也稱為平台事件日誌。

對於支援的執行期，當您使用其中一種支援的內建記錄方法時，Lambda 也能以結構化 JSON 格式輸出函數的應用程式日誌 (函數程式碼產生的日誌)。當您針對這些執行期設定函數的日誌格式時，您選擇的組態會同時套用至系統和應用程式日誌。

對於支援的執行期，如果您的函數使用支援的記錄程式庫或方法，則不需要變更 Lambda 現有程式碼，即可擷取結構化 JSON 中的記錄。

> **Note**
>
> 使用 JSON 日誌格式新增其他中繼資料，並將日誌訊息編碼為包含一系列索引鍵值組的 JSON 物件。因此，函數日誌訊息的大小可能會增加。

支援的執行期和記錄方法

Lambda 目前支援針對下列執行期輸出 JSON 結構化應用程式日誌的選項。

執行期	支援的版本
Java	除了 Amazon Linux 1 上的 Java 8 以外的所有 Java 執行期
Node.js	Node.js 16 及更高版本
Python	Python 3.7 及更高版本

若要讓 Lambda 以結構化 JSON 格式傳送函數 CloudWatch 的應用程式記錄檔，您的函數必須使用下列內建記錄工具來輸出記錄：

- Java - `LambdaLogger` 日誌或 Log4j2。
- Node.js - 主控台的方法 `console.trace`、`console.debug`、`console.log`、`console.info`、`console.error` 和 `console.warn`
- Python - 標準的 Python `logging` 程式庫

如需關於將進階日誌控制項與支援的執行期搭配使用的詳細資訊，請參閱 the section called “日誌”、the section called “日誌” 和 the section called “日誌”。

對於其他受管的 Lambda 執行期，Lambda 目前僅支援以結構化 JSON 格式擷取系統記錄。但是，您仍然可以在任何運行時使用記錄工具（例如 Powertools）捕獲結構化 JSON 格式的應用程序日誌記錄，用於 AWS Lambda 該輸出 JSON 格式的日誌輸出。

預設日誌格式

目前，所有 Lambda 執行期的預設日誌格式都是純文字。

如果您已經使用 Powertools 等日誌庫 AWS Lambda 來生成 JSON 結構化格式的函數日誌，則如果選擇 JSON 日誌格式，則不需要更改代碼。Lambda 不會對任何已經進行 JSON 編碼的日誌進行雙重編碼，因此您函數的應用程式日誌將繼續像以前一樣被擷取。

系統日誌的 JSON 格式

當您將函數的日誌檔格式設定為 JSON 時，每個系統日誌項目 (平台事件) 都會擷取為 JSON 物件，其中包含與下列索引鍵配對的索引鍵值：

- `"time"` - 產生日誌訊息的時間
- `"type"` - 記錄的事件類型
- `"record"` - 日誌輸出的內容

`"record"` 值的格式會根據日誌的事件類型而有所不同。如需更多資訊，請參閱the section called “遙測 API Event 物件類型”。如需有關指派給系統日誌事件的日誌層級的詳細資訊，請參閱 the section called “系統日誌層級事件映射”。

為了進行比較，下列兩個範例會以純文字和結構化 JSON 格式顯示相同的記錄輸出。請注意，在大多數情況下，系統日誌事件在 JSON 格式輸出時所包含的資訊會比以純文字輸出時更多。

Example 純文字：

```
2023-03-13 18:56:24.046000 fbe8c1   INIT_START  Runtime Version:
 python:3.9.v18  Runtime Version ARN: arn:aws:lambda:eu-
west-1::runtime:edb5a058bfa782cb9cedc6d534ac8b8c193bc28e9a9879d9f5ebaaf619cd0fc0
```

Example 結構化的 JSON：

```
{
  "time": "2023-03-13T18:56:24.046Z",
  "type": "platform.initStart",
  "record": {
    "initializationType": "on-demand",
    "phase": "init",
    "runtimeVersion": "python:3.9.v18",
    "runtimeVersionArn": "arn:aws:lambda:eu-
west-1::runtime:edb5a058bfa782cb9cedc6d534ac8b8c193bc28e9a9879d9f5ebaaf619cd0fc0"
  }
}
```

> **Note**
>
> the section called “遙測 API” 始終以 JSON 格式發出平台事件，例如 `START` 和 `REPORT`。設定 Lambda 傳送的系統記錄檔格式 CloudWatch 不會影響 Lambda 遙測 API 行為。

應用程式日誌的 JSON 格式

當您將函數的日誌格式設定為 JSON 時，使用支援的記錄程式庫和方法撰寫的應用程式日誌輸出會擷取為 JSON 物件，其中包含具有以下索引鍵的鍵值對。

- `"timestamp"` - 產生日誌訊息的時間
- `"level"` - 指派給訊息的日誌層級
- `"message"` - 日誌訊息的內容
- `"requestId"` (Python 和 Node.js) 或 `"AWSrequestId"` (Java) - 函數調用的唯一請求 ID

依據您的函數使用的執行期和記錄方法，此 JSON 物件還可能包含其他鍵值對。例如，在 Node.js 中，如果您的函數使用 `console` 方法來記錄使用多個引數的錯誤物件，JSON 物件將包含具有索引鍵 `errorMessage`、`errorType` 和 `stackTrace` 的額外鍵值對。若要進一步了解不同 Lambda 執行期中的 JSON 格式的日誌，請參閱 the section called “日誌”、the section called “日誌” 和 the section called “日誌”。

> **Note**
>
> Lambda 用於時間戳記值的關鍵對於系統日誌和應用程式日誌而言不同。對於系統記錄，Lambda 會使用索引鍵 `"time"` 來維持遙測 API 的一致性。對於應用程式日誌，Lambda 會遵循支援的執行期和使用 `"timestamp"` 的慣例。

為了進行比較，下列兩個範例會以純文字和結構化 JSON 格式顯示相同的記錄輸出。

Example 純文字：

```
2023-10-27T19:17:45.586Z 79b4f56e-95b1-4643-9700-2807f4e68189 INFO some log message
```

Example 結構化的 JSON：

```
{
```

```
    "timestamp":"2023-10-27T19:17:45.586Z",
    "level":"INFO",
    "message":"some log message",
    "requestId":"79b4f56e-95b1-4643-9700-2807f4e68189"
}
```

設定函數的日誌格式

若要設定函數的記錄格式，您可以使用 Lambda 主控台或 AWS Command Line Interface (AWS CLI)。您也可以使用和設UpdateFunction定 Lambda API 命令、 AWS Serverless Application Model (AWS SAM) AWS::Serverless::Function資源CreateFunction和資源來設定函數的 AWS CloudFormation AWS::Lambda::Function記錄格式。

更改函數的日誌格式不會影響存儲在日誌中的現有 CloudWatch 日誌。只有新的日誌檔會使用更新的格式。

如果您將函數的日誌格式變更為 JSON 但未設定日誌層級，Lambda 會自動將函數的應用程序日誌層級和系統日誌層級設為 INFO。這表示 Lambda 只會傳送層級 INFO 和更低層級的記錄輸出至 CloudWatch 記錄。若要進一步了解應用程式和系統日誌層級篩選，請參閱 the section called “日誌層級篩選”

> **Note**
>
> 對於 Python 執行期，當函數的日誌格式設定為純文字時，預設的日誌層級設定為 WARN。這表示 Lambda 只會將 WARN 層級和更低層級的記錄輸出傳送至 CloudWatch 記錄檔。將函數的日誌格式變更為 JSON 會改變此預設行為。若要進一步了解以 Python 記錄日誌，請參閱 the section called “日誌”。

對於發出內嵌指標格式 (EMF) 日誌的 Node.js 函數，將函數的日誌格式更改為 JSON 可能會導 CloudWatch 致無法識別您的指標。

> **Important**
>
> 如果您的函數使用 Powertools of AWS Lambda (TypeScript) 或開源 EMF 客戶端庫發出 EMF 日誌，請將您的 Powertools 和 EMF 庫更新為最新版本，以確保 CloudWatch 可以繼續正確解析日誌。如果您切換到 JSON 日誌格式，我們也建議您進行測試，以確保與函數的內嵌指標相容。如需有關發出 EMF 日誌檔之 node.js 函數的進一步建議，請參閱 the section called “搭配結構化 JSON 日誌使用內嵌指標格式 (EMF) 用戶端程式庫”。

若要設定函數的日誌格式 (主控台)

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇一個函數。
3. 在函數組態頁面上，選擇監視和操作工具。
4. 在日誌組態窗格中，選擇編輯。
5. 在日誌內容之下，針對日誌檔格式選取文字或 JSON。
6. 選擇儲存。

若要變更現有函數的日誌格式 (AWS CLI)

- 若要變更現有函數的日誌格式，請使用 `update-function-configuration` 命令。將 `LoggingConfig` 中 `LogFormat` 選項設定為 `JSON` 或 `Text`。

```
aws lambda update-function-configuration \
--function-name myFunction --logging-config LogFormat=JSON
```

若要在建立函數 (AWS CLI) 時設定記錄格式

- 若要在建立新函數時設定日誌格式，請使用 `create-function` 命令中的 `--logging-config` 選項。將 `LogFormat` 設定為 `JSON` 或 `Text`。下列範例命令會使用 Node.js 18 執行期建立函數，以結構化 JSON 輸出日誌檔。

  如果您在建立函數時未指定日誌格式，Lambda 會針對您選取的執行期版本使用預設日誌格式。如需有關預設記錄格式的資訊，請參閱 the section called “預設日誌格式”。

```
aws lambda create-function --function-name myFunction --runtime nodejs18.x \
--handler index.handler --zip-file fileb://function.zip \
--role arn:aws:iam::123456789012:role/LambdaRole --logging-config LogFormat=JSON
```

日誌層級篩選

Lambda 可以篩選函數的記錄，只有特定詳細資料層級或更低層級的記錄檔才會傳送至 CloudWatch 記錄。您可以針對函數的系統日誌 (Lambda 產生的日誌) 和應用程式日誌 (函數程式碼產生的日誌) 分別設定日誌層級篩選。

對於 the section called “支援的執行期和記錄方法”，您無需對函數程式碼進行任何變更，Lambda 即可篩選函數的應用程式日誌。

對於所有其他執行期和記錄方法，您的函數程式碼必須將日誌事件輸出至 `stdout` 或 `stderr` 作為 JSON 格式的物件，其中包含與索引鍵 `"level"` 配對的索引鍵值)。例如，Lambda 會將下列輸出解譯為 `stdout` 作為 DEBUG 層級日誌檔。

```
print('{"level": "debug", "msg": "my debug log", "timestamp":
 "2023-11-02T16:51:31.587199Z"}')
```

如果 `"level"` 值欄位無效或遺失，Lambda 會將日誌輸出指派層級 INFO。若要讓 Lambda 使用時間戳記欄位，您必須以有效的 RFC 3339 時間戳記格式指定時間。如果您沒有提供有效的時間戳記，Lambda 會為日誌指派層級 INFO，並為您新增時間戳記。

命名時間戳記索引鍵時，請遵循您使用的執行期慣例。Lambda 支援受管理執行期使用的大多數通用命名慣例。例如，在使用 .NET 執行期的函數中，Lambda 會辨識 索引鍵 `"Timestamp"`。

> **Note**
>
> 若要使用日誌層級篩選，您的函數必須設定為使用 JSON 記錄格式。所有 Lambda 受管執行期的預設日誌格式目前都是純文字。若要瞭解如何將函數的日誌格式設定為 JSON，請參閱 the section called “設定函數的日誌格式”。

對於應用程式日誌 (由函數程式碼生成的日誌)，您可以在以下日誌層級之間進行選擇。

日誌層級	標準用量
TRACE (大多數詳細資訊)	用於追蹤程式碼執行路徑的最精細資訊
DEBUG	系統偵錯的詳細資訊
INFO	記錄函數正常操作的訊息
WARN	有關可能導致未解決意外行為的潛在錯誤的消息
ERROR	有關阻止程式碼按預期執行的問題的訊息
FATAL (最少詳細資訊)	有關導致應用程式停止運作的嚴重錯誤訊息

當您選取記錄層級時，Lambda 會將該層級及較低層級的記錄傳送至 CloudWatch 記錄檔。例如，如果您將函數的應用程式日誌層級設定為 WARN，Lambda 就不會在 INFO 和 DEBUG 層級傳送日誌輸出。日誌篩選的預設應用程式日誌層級為 INFO。

當 Lambda 篩選函數的應用程式日誌時，沒有層級的日誌訊息將被指派日誌等級 INFO。

對於系統日誌檔 (Lambda 服務產生的日誌檔)，您可以在下列日誌層級進行選擇。

日誌層級	用量
DEBUG (大多數詳細資訊)	系統偵錯的詳細資訊
INFO	記錄函數正常操作的訊息
WARN (最少詳細資訊)	有關可能導致未解決意外行為的潛在錯誤的消息

當您選取日誌層級時，Lambda 會在該層級 (含) 或更低層級傳送日誌。例如，如果您將函數的系統日誌層級設定為 INFO，Lambda 不會在 DEBUG 層級傳送日誌輸出。

根據預設，Lambda 會將系統日誌層級設定為 INFO。透過此設定，Lambda 會自動將訊息傳送`"start"`並`"report"`記錄到 CloudWatch。若要接收更多或更少詳細的系統日誌，請將日誌層級變更為 DEBUG 或 WARN。若要查看 Lambda 映射不同系統日誌事件的記錄層級清單，請參閱 the section called “系統日誌層級事件映射”。

設定日誌層級篩選

若要為您的函數設定應用程式和系統記錄層級篩選，您可以使用 Lambda 主控台或 AWS Command Line Interface (AWS CLI)。您也可以使用和設UpdateFunction定 Lambda API 命令、 AWS Serverless Application Model (AWS SAM) AWS::Serverless::Function資源CreateFunction和資源來設定函數的 AWS CloudFormation AWS::Lambda::Function記錄層級。

請注意，如果您在程式碼中設定函數的日誌層級，此設定會優先於您的任何其他日誌層級設定。例如，如果您使用 Python `logging` `setLevel()` 方法將函數的記錄層級設定為 INFO，則此設定的優先級將高於您使用 Lambda 主控台設定的 WARN 層級。

若要設定現有函數的應用程式或系統日誌層級 (主控台)

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 在函數組態頁面上，選擇監視和操作工具。

4. 在日誌組態窗格中，選擇編輯。
5. 在日誌內容之下，針對日誌檔格式，確保已選取 JSON
6. 使用選項按鈕，為您的函數選擇所需的應用程式日誌層級和系統日誌層級。
7. 選擇儲存。

若要設定現有函數的應用程式或系統日誌層級 (AWS CLI)

- 若要變更現有函數的應用程式或系統日誌層級，請使用 `update-function-configuration` 命令。設定 `--system-log-level` 為 `DEBUG`、`INFO` 或 `WARN` 之一。設定 `--application-log-level` 為 `DEBUG`、`INFO`、`WARN`、`ERROR` 或 `FATAL` 之一。

```
aws lambda update-function-configuration \
--function-name myFunction --system-log-level WARN \
--application-log-level ERROR
```

若要在建立函數時設定日誌層級篩選

- 若要在建立新函數時設定日誌層級篩選，請使用 create-function 命令中的 `--system-log-level` 和 `--application-log-level` 選項。設定 `--system-log-level` 為 `DEBUG`、`INFO` 或 `WARN` 之一。設定 `--application-log-level` 為 `DEBUG`、`INFO`、`WARN`、`WARN` 或 `FATAL` 之一。

```
aws lambda create-function --function-name myFunction --runtime nodejs18.x \
--handler index.handler --zip-file fileb://function.zip \
--role arn:aws:iam::123456789012:role/LambdaRole --system-log-level WARN \
--application-log-level ERROR
```

系統日誌層級事件映射

對於 Lambda 產生的系統層級日誌事件，以下資料表定義指派給每個事件的日誌層級。若要進一步瞭解資料表中所列事件，請參閱 the section called “Event 結構描述參考”

事件名稱	條件	指派的日誌層級
initStart	runtimeVersion已設定	INFO

事件名稱	條件	指派的日誌層級
initStart	runtimeVersion未設定	DEBUG
初始化 RuntimeDone	status=success	DEBUG
初始化 RuntimeDone	status!=success	WARN
initReport	initializationType=snapstart	INFO
initReport	initializationType!=snapstart	DEBUG
initReport	status!=success	WARN
restoreStart	runtimeVersion已設定	INFO
restoreStart	runtimeVersion未設定	DEBUG
恢復 RuntimeDone	status=success	DEBUG
恢復 RuntimeDone	status!=success	WARN
restoreReport	status=success	INFO
restoreReport	status!=success	WARN
入門	-	INFO
runtimeDone	status=success	DEBUG
runtimeDone	status!=success	WARN
報告	status=success	INFO
報告	status!=success	WARN
副檔名	state=success	INFO
副檔名	state!=success	WARN
logSubscription	-	INFO

事件名稱	條件	指派的日誌層級
telemetrySubscription	-	INFO
logsDropped	-	WARN

> **Note**
>
> the section called “遙測 API” 始終發出一組完整的平台事件。設定 Lambda 傳送的系統記錄層級 CloudWatch 不會影響 Lambda 遙測 API 行為。

使用自訂執行期的應用程式日誌層級篩選

當您為函數設定應用程式日誌層級篩選時，Lambda 會在幕後使用 AWS_LAMBDA_LOG_LEVEL 環境變數在執行期設定應用程式日誌層級。Lambda 也會使用 AWS_LAMBDA_LOG_FORMAT 環境變數來設定函數的日誌格式。您可以使用這些變數，將 Lambda 進階日誌控制項整合至自訂執行期。

若要使用自訂執行階段搭配 Lambda 主控台和 Lambda API 來設定函數的記錄設定，請設定您的自訂執行階段以檢查這些環境變數的值。 AWS CLI然後，您可以根據您選取的日誌格式和日誌層級來設定執行期的日誌程式。

設定 CloudWatch 記錄群組

根據預設， CloudWatch 會在第一次叫用函數時自動建立/aws/lambda/<function name>為函數命名的記錄群組。若要將函數設定為將日誌傳送到現有的日誌群組，或為您的函數建立新的日誌群組，您可以使用 Lambda 主控台或 AWS CLI. 您也可以使用和設定 Lambda API 命令CreateFunction和UpdateFunction AWS Serverless Application Model (AWS SAM):: 無伺服器:AWS: 函數資源來設定自訂記錄群組。

您可以設定多個 Lambda 函數，將記錄傳送至相同的記 CloudWatch 錄群組。例如，您可以使用單一日誌群組來儲存組成特定應用程式之所有 Lambda 函數的記錄。當您針對 Lambda 函數使用自訂日誌群組時，Lambda 建立的日誌串流會包含函數名稱和函數版本。如此可確保日誌訊息和函數之間的映射會被保留，即使您對多個函數使用相同的日誌群組也是如此。

自訂記錄群組的記錄資料流命名格式遵循下列慣例：

```
YYYY/MM/DD/<function_name>[<function_version>][<execution_environment_GUID>]
```

請注意，設定自訂記錄群組時，您為記錄群組選取的名稱必須遵循CloudWatch 記錄檔命名規則。此外，自訂日誌群組名稱不得以字串 aws/ 開頭。如果您以 aws/ 開頭建立自訂日誌群組，Lambda 將無法建立日誌群組。因此，您的函數的日誌將不會發送到 CloudWatch。

若要變更函數的日誌群組 (主控台)

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 在函數組態頁面上，選擇監視和操作工具。
4. 在日誌組態窗格中，選擇編輯。
5. 在 [記錄群組] 窗格中，對於記CloudWatch 錄群組，選擇 [自訂]。
6. 在 [自訂記錄群組] 下，輸入您希望函數傳送記錄的目標記錄群組名稱。 CloudWatch 如果您輸入現有日誌群組的名稱，則您的函數將使用該群組。如果沒有具有您輸入名稱的日誌群組，則 Lambda 會以該名稱為您的函數建立新的日誌群組。

若要變更函數的日誌群組 (AWS CLI)

- 若要變更現有函數的日誌群組，請使用 `update-function-configuration` 命令。如果您指定現有日誌群組的名稱，則您的函數將使用該群組。如果沒有具有您指定名稱的日誌群組，則 Lambda 會以該名稱為您的函數建立新的日誌群組。

```
aws lambda update-function-configuration \
--function-name myFunction --log-group myLogGroup
```

若要在建立函數 (AWS CLI) 時指定自訂日誌群組

- 若要在使用建立新 Lambda 函數時指定自訂記錄群組 AWS CLI，請使用`--log-group`選項。如果您指定現有日誌群組的名稱，則您的函數將使用該群組。如果沒有具有您指定名稱的日誌群組，則 Lambda 會以該名稱為您的函數建立新的日誌群組。

  下列範例命令會建立 Node.js Lambda 函數，該函數會將日誌檔傳送至名為 myLogGroup 的日誌群組。

```
aws lambda create-function --function-name myFunction --runtime nodejs18.x \
--handler index.handler --zip-file fileb://function.zip \
--role arn:aws:iam::123456789012:role/LambdaRole --log-group myLogGroup
```

執行角色許可

為了讓您的功能將日誌發送到 CloudWatch 日誌，它必須具有`logs:PutLogEvents`權限。使用 Lambda 主控台設定函數的日誌群組時，如果函數沒有此許可，Lambda 預設會將其新增至函數的執行角色。Lambda 新增此權限時，會授予函數將記錄傳送至任何日誌 CloudWatch 記錄群組的權限。

若要防止 Lambda 自動更新函數的執行角色並改為手動編輯，請展開許可，然後取消勾選新增所需許可。

使用設定函數的日誌群組時 AWS CLI，Lambda 不會自動新增`logs:PutLogEvents`權限。如果函數的執行角色尚不具備許可，請將其新增至函數的執行角色。此權限包含在AWSLambdaBasicExecutionRole受管理的策略中。

使用 Lambda 主控台存取日誌

若要使用 Lambda 主控台檢視日誌

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇 監控 。
4. 選擇 [檢視登入] CloudWatch。

使用存取記錄 AWS CLI

這 AWS CLI 是一種開放原始碼工具，可讓您使用命令列殼層中的命令與 AWS 服務互動。若要完成本節中的步驟，您必須執行下列各項：

- AWS Command Line Interface (AWS CLI) 第二版
- AWS CLI -快速配置 `aws configure`

您可以透過 AWS CLI，使用 `--log-type` 命令選項來擷取要調用的日誌。其回應將包含 `LogResult` 欄位，內含該次調用的 base64 編碼日誌 (最大達 4 KB)。

Example 擷取日誌 ID

下列範例顯示如何從名稱為 `my-function` 的函數的 `LogResult` 欄位來擷取日誌 ID。

```
aws lambda invoke --function-name my-function out --log-type Tail
```

您應該會看到下列輸出：

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example 解碼日誌

在相同的命令提示中，使用 base64 公用程式來解碼日誌。下列範例顯示如何擷取 my-function 的 base64 編碼日誌。

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

如果您使用的是 AWS CLI 版本 2，則需要此cli-binary-format選項。若要讓此成為預設的設定，請執行 aws configure set cli-binary-format raw-in-base64-out。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

您應該會看到下列輸出：

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

該 base64 公用程式可在 Linux、macOS 和 Ubuntu on Windows 上使用。macOS 使用者可能需要使用 base64 -D。

Example get-logs.sh 指令碼

在相同的命令提示中，使用下列指令碼下載最後五個日誌事件。該指令碼使用 sed 以從輸出檔案移除引述，並休眠 15 秒以使日誌可供使用。輸出包括來自 Lambda 的回應以及來自 get-log-events 命令的輸出。

複製下列程式碼範例的內容，並將您的 Lambda 專案目錄儲存為 `get-logs.sh`。

如果您使用的是 AWS CLI 版本 2，則需要此cli-binary-format選項。若要讓此成為預設的設定，請執行 `aws configure set cli-binary-format raw-in-base64-out`。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS 和 Linux (僅限)

在相同的命令提示中，macOS 和 Linux 使用者可能需要執行下列命令，以確保指令碼可執行。

```
chmod -R 755 get-logs.sh
```

Example 擷取最後五個記錄事件

在相同的命令提示中，執行下列指令碼以取得最後五個日誌事件。

```
./get-logs.sh
```

您應該會看到下列輸出：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
```

```
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```


執行階段函數記錄

若要偵錯並驗證程式碼是否如預期般運作，您可以使用程式設計語言的標準記錄功能輸出記錄檔。Lambda 執行階段會將函數的記錄輸出上傳至 CloudWatch 記錄。如需語言的專屬說明，請參閱以下主題：

- AWS Lambda Node.js 中的函數日誌記錄
- AWS Lambda 函數日誌記 Python
- AWS Lambda 紅寶石中的函數登錄
- AWS Lambda Java 中的函數日誌記錄
- AWS Lambda 函數登錄 Go
- C# 中的 Lambda 函數日誌記錄

- AWS Lambda 功能登錄 PowerShell

後續步驟？

- 在 Amazon CloudWatch 使用者指南中的監 CloudWatch 控系統、應用程式和自訂日誌檔中，進一步了解日誌群組以及透過主控台存取日誌群組。

使用記錄 AWS Lambda API 呼叫 AWS CloudTrail

AWS Lambda 與AWS CloudTrail提供使用者、角色或 AWS 服務. CloudTrail 將 Lambda 的 API 呼叫擷取為事件。擷取的呼叫包括從 Lambda 主控台進行的呼叫，以及針對 Lambda API 操作的程式碼呼叫。使用收集的資訊 CloudTrail，您可以判斷向 Lambda 發出的請求、提出請求的來源 IP 位址、提出要求的時間以及其他詳細資訊。

每一筆事件或日誌專案都會包含產生請求者的資訊。身分資訊可協助您判斷下列事項：

- 該請求是使用根使用者還是使用者憑證提出。
- 請求是否代表 IAM 身分中心使用者提出。
- 提出該請求時，是否使用了特定角色或聯合身分使用者的暫時安全憑證。
- 該請求是否由另一項 AWS 服務服務提出。

CloudTrail 在您創建帳戶 AWS 帳戶 時處於活動狀態，並且您自動可以訪問 CloudTrail 事件歷史記錄。 CloudTrail 事件歷史記錄提供了過去 90 天中記錄的管理事件的可查看，可搜索，可下載和不可變的記錄。 AWS 區域若要取得更多資訊，請參閱《使用指南》中的〈AWS CloudTrail 使用 CloudTrail 事件歷程〉。查看活動歷史記錄不 CloudTrail收取任何費用。

如需過 AWS 帳戶 去 90 天內持續的事件記錄，請建立追蹤或 CloudTrailLake 事件資料存放區。

CloudTrail 小徑

追蹤可 CloudTrail 將日誌檔交付到 Amazon S3 儲存貯體。使用建立的所有系統線 AWS Management Console 都是多區域。您可以使用建立單一區域或多區域系統線。 AWS CLI建議您建立多區域追蹤，因為您會擷取帳戶 AWS 區域 中的所有活動。如果您建立單一區域追蹤，則只能檢視追蹤記錄中的 AWS 區域事件。如需有關追蹤的詳細資訊，請參閱《AWS CloudTrail 使用指南》中的「為您的建立追蹤」 AWS 帳戶和「為組織建立追蹤」。

您可以透 CloudTrail 過建立追蹤，免費將一份正在進行的管理事件副本傳遞到 Amazon S3 儲存貯體，但是需要支付 Amazon S3 儲存費用。如需有關 CloudTrail 定價的詳細資訊，請參閱AWS CloudTrail 定價。如需 Amazon S3 定價的相關資訊，請參閱 Amazon S3 定價。

CloudTrail 湖泊事件資料存放區

CloudTrail L@@ ake 可讓您針對事件執行 SQL 型查詢。 CloudTrail 湖將基於行的 JSON 格式的現有事件轉換為 Apache ORC 格式。ORC 是一種單欄式儲存格式，針對快速擷取資料進行了最佳化。系統會將事件彙總到事件資料存放區中，事件資料存放區是事件的不可變集合，其依據為您透過套用進階事件選取器選取的條件。套用於事件資料存放區的選取器控制哪些事件持續存在並可供

您查詢。若要取得有關 CloudTrail Lake 的更多資訊，請參閱使用指南中的〈AWS CloudTrail 使用 AWS CloudTrail Lake〉。

CloudTrail Lake 事件資料存放區和查詢會產生費用。建立事件資料存放區時，您可以選擇要用於事件資料存放區的定價選項。此定價選項將決定擷取和儲存事件的成本，以及事件資料存放區的預設和最長保留期。如需有關 CloudTrail 定價的詳細資訊，請參閱AWS CloudTrail 定價。

Lambda 資料事件 CloudTrail

資料事件提供在資源上或在資源中執行的資源操作的相關資訊 (例如，讀取或寫入 Amazon S3 物件)。這些也稱為資料平面操作。資料事件通常是大量資料的活動。根據預設， CloudTrail 不會記錄大多數資料事件，且 CloudTrail 事件歷程記錄不會記錄這些事件。

支援的服務預設會記錄其中一個 CloudTrail 資料事件`LambdaESMDisabled`。若要進一步了解如何使用此事件協助疑難排解 Lambda 事件來源對應的問題，請參閱the section called “用來疑 CloudTrail 難排解已停用的 Lambda 事件”。

資料事件需支付額外的費用。如需有關 CloudTrail 定價的詳細資訊，請參閱AWS CloudTrail 定價。

您可以使用 CloudTrail 主控台或 CloudTrail API 作業記錄`AWS::Lambda::Function`資源類型的資料事件。 AWS CLI有關如何記錄資料事件的詳細資訊，請參閱AWS CloudTrail 使用《使用指南》 AWS Command Line Interface中的記錄資料事件 AWS Management Console和記錄資料事件。

下表列出您可以記錄其資料事件的 Lambda 資源類型。[資料事件類型 (主控台)] 欄顯示可從主控台的 [資料事件類型 CloudTrail] 清單中選擇的值。resource .type 值欄會顯示**`resources.type`**值，您可以在使用或 API 設定進階事件選取器時指定這個值。 AWS CLI CloudTrail 記錄到資料 CloudTrail欄中的資料 API 會顯示 CloudTrail 針對資源類型記錄的 API 呼叫。

資料事件類型 (主控台)	resources.type 值	記錄到的資料 API CloudTrail
Lambda	`AWS::Lambda::Function`	Invoke

您可以設定進階事件選取器來篩選`eventName`、`readOnly`、和`resources.ARN`欄位，以僅記錄對您很重要的事件。下列範例是資料事件組態的 JSON 檢視，此檢視僅記錄特定函數的事件。如需這些欄位的詳細資訊，請參閱 AWS CloudTrail API 參考AdvancedFieldSelector中的。

```
[
```

```
  {
    "name": "function-invokes",
    "fieldSelectors": [
      {
        "field": "eventCategory",
        "equals": [
          "Data"
        ]
      },
      {
        "field": "resources.type",
        "equals": [
          "AWS::Lambda::Function"
        ]
      },
      {
        "field": "resources.ARN",
        "equals": [
          "arn:aws:lambda:us-east-1:111122223333:function:hello-world"
        ]
      }
    ]
  }
]
```

Lambda 管理事件 CloudTrail

管理事件提供有關在您的資源上執行的管理作業的資訊 AWS 帳戶。這些也稱為控制平面操作。依預設，會 CloudTrail 記錄管理事件。

Lambda 支援將下列動作記錄為記 CloudTrail 錄檔中的管理事件。

> **Note**
>
> 在 CloudTrail 記錄檔中，`eventName`可能包含日期和版本資訊，但仍參考相同的公用 API 動作。例如，動`GetFunction`作會顯示為`GetFunction20150331v2`。下列清單指定事件名稱與 API 動作名稱不同的時間。

- AddLayerVersionPermission
- AddPermission(活動名稱:`AddPermission20150331v2`)

- CreateAlias(活動名稱:CreateAlias20150331)
- CreateEventSourceMapping(活動名稱:CreateEventSourceMapping20150331)
- CreateFunction(活動名稱:CreateFunction20150331)

  (Environment和ZipFile參數會從的 CloudTrail 記錄中省略CreateFunction。)
- CreateFunctionUrlConfig
- DeleteAlias(活動名稱:DeleteAlias20150331)
- DeleteCodeSigningConfig
- DeleteEventSourceMapping(活動名稱:DeleteEventSourceMapping20150331)
- DeleteFunction(活動名稱:DeleteFunction20150331)
- DeleteFunction並行 (事件名稱:DeleteFunctionConcurrency20171031)
- DeleteFunctionUrlConfig
- DeleteProvisionedConcurrencyConfig
- GetAlias(活動名稱:GetAlias20150331)
- GetEventSourceMapping
- GetFunction
- GetFunctionUrlConfig
- GetFunction配置
- GetLayerVersionPolicy
- GetPolicy
- ListEventSourceMappings
- ListFunctions
- ListFunctionUrlConfigs
- PublishLayer版本 (事件名稱:PublishLayerVersion20181031)

  (的 CloudTrail 記錄中省略此ZipFile參數PublishLayerVersion。)
- PublishVersion(活動名稱:PublishVersion20150331)
- PutFunction並行 (事件名稱:PutFunctionConcurrency20171031)
- PutFunctionCodeSigningConfig
- PutFunctionEventInvokeConfig
- PutProvisionedConcurrencyConfig

- PutRuntimeManagementConfig
- RemovePermission(活動名稱:RemovePermission20150331v2)
- TagResource(活動名稱:TagResource20170331v2)
- UntagResource(活動名稱:UntagResource20170331v2)
- UpdateAlias(活動名稱:UpdateAlias20150331)
- UpdateCodeSigningConfig
- UpdateEventSourceMapping(活動名稱:UpdateEventSourceMapping20150331)
- UpdateFunction代碼 (事件名稱:UpdateFunctionCode20150331v2)

  (的 CloudTrail 記錄中省略此ZipFile參數UpdateFunctionCode。)
- UpdateFunction組態 (事件名稱:UpdateFunctionConfiguration20150331v2)

  (的 CloudTrail 記錄中省略此Environment參數UpdateFunctionConfiguration。)
- UpdateFunctionEventInvokeConfig
- UpdateFunctionUrlConfig

用來疑 CloudTrail 難排解已停用的 Lambda 事件

當您使用 UpdateEventSourceMappingAPI 動作變更事件來源對應的狀態時，API 呼叫會記錄為中的管理事件 CloudTrail。由於錯誤，事件來源對映也可以直接轉換至Disabled狀態。

對於下列服務，Lambda 會將資LambdaESMDisabled料事件發佈 CloudTrail 到事件來源轉換為「已停用」狀態時：

- Amazon Simple Queue Service (Amazon SQS)
- Amazon DynamoDB
- Amazon Kinesis

Lambda 不支援任何其他事件來源對應類型的此事件。

若要在支援服務的事件來源對應轉換到Disabled狀態時接收警示，請 CloudWatch 使用該LambdaESMDisabled CloudTrail 事件在 Amazon 中設定警示。若要深入瞭解如何設定 CloudWatch 鬧鐘，請參閱建立 CloudTrail 事件 CloudWatch 警示：範例。

LambdaESMDisabled事件訊息中的serviceEventDetails實體包含下列其中一個錯誤代碼。

RESOURCE_NOT_FOUND

請求中指定的資源不存在。

FUNCTION_NOT_FOUND

附加至事件來源的函數不存在。

REGION_NAME_NOT_VALID

提供給事件來源或函數的區域名稱無效。

AUTHORIZATION_ERROR

權限尚未設定，或設定錯誤。

FUNCTION_IN_FAILED_STATE

函數程式碼無法編譯、發生無法復原的例外狀況，或發生錯誤的部署。

Lambda 事件範例

事件代表來自任何來源的單一請求，並包括有關請求的 API 操作，操作的日期和時間，請求參數等信息。 CloudTrail 日誌文件不是公共 API 調用的有序堆棧跟踪，因此事件不會以任何特定順序顯示。

下列範例顯示`GetFunction`和`DeleteFunction`動作的 CloudTrail 記錄項目。

> **Note**
>
> `eventName` 可能包含日期與版本資訊，如 `"GetFunction20150331"`，但仍參照至相同的公有 API。

```
{
  "Records": [
    {
      "eventVersion": "1.03",
      "userIdentity": {
        "type": "IAMUser",
        "principalId": "A1B2C3D4E5F6G7EXAMPLE",
        "arn": "arn:aws:iam::111122223333:user/myUserName",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "userName": "myUserName"
```

```
      },
      "eventTime": "2015-03-18T19:03:36Z",
      "eventSource": "lambda.amazonaws.com",
      "eventName": "GetFunction",
      "awsRegion": "us-east-1",
      "sourceIPAddress": "127.0.0.1",
      "userAgent": "Python-httplib2/0.8 (gzip)",
      "errorCode": "AccessDenied",
      "errorMessage": "User: arn:aws:iam::111122223333:user/myUserName is not
 authorized to perform: lambda:GetFunction on resource: arn:aws:lambda:us-
west-2:111122223333:function:other-acct-function",
      "requestParameters": null,
      "responseElements": null,
      "requestID": "7aebcd0f-cda1-11e4-aaa2-e356da31e4ff",
      "eventID": "e92a3e85-8ecd-4d23-8074-843aabfe89bf",
      "eventType": "AwsApiCall",
      "recipientAccountId": "111122223333"
    },
    {
      "eventVersion": "1.03",
      "userIdentity": {
        "type": "IAMUser",
        "principalId": "A1B2C3D4E5F6G7EXAMPLE",
        "arn": "arn:aws:iam::111122223333:user/myUserName",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "userName": "myUserName"
      },
      "eventTime": "2015-03-18T19:04:42Z",
      "eventSource": "lambda.amazonaws.com",
      "eventName": "DeleteFunction20150331",
      "awsRegion": "us-east-1",
      "sourceIPAddress": "127.0.0.1",
      "userAgent": "Python-httplib2/0.8 (gzip)",
      "requestParameters": {
        "functionName": "basic-node-task"
      },
      "responseElements": null,
      "requestID": "a2198ecc-cda1-11e4-aaa2-e356da31e4ff",
      "eventID": "20b84ce5-730f-482e-b2b2-e8fcc87ceb22",
      "eventType": "AwsApiCall",
      "recipientAccountId": "111122223333"
    }
  ]
```

```
}
```

若要取得有關 CloudTrail 記錄內容的資訊，請參閱AWS CloudTrail 使用指南中的CloudTrail記錄內容。

使用視覺化 Lambda 函數叫用 AWS X-Ray

您可以使 AWS X-Ray 用視覺化應用程式的元件、識別效能瓶頸，以及疑難排解導致錯誤的要求。您的 Lambda 函數會將追蹤資料傳送至 X-Ray，而且 X-Ray 會處理資料，以產生服務映射和可搜尋的追蹤摘要。

如果您已在調用函數的服務中啟用 X-Ray 追蹤，則 Lambda 會自動將追蹤傳送至 X-Ray。上游服務 (例如 Amazon API Gateway) 或在 Amazon EC2 上託管並利用 X-Ray 開發套件進行檢測的應用程式會取樣傳入要求，並新增追蹤標頭，告知 Lambda 是否傳送追蹤。來自上游訊息生產者的追蹤 (例如 Amazon SQS) 會自動連結至來自下游 Lambda 函數的追蹤，以建立整個應用程式的 end-to-end 檢視。如需詳細資訊，請參閱《AWS X-Ray 開發人員指南》中的追蹤事件導向應用程式。

> **Note**
>
> Lambda 函數若具有 Amazon Managed Streaming for Apache Kafka (Amazon MSK)、自我管理的 Apache Kafka、帶有 ActiveMQ 和 RabbitMQ 的 Amazon MQ，或是 Amazon DocumentDB 事件來源映射，目前不支援 X-Ray 追蹤。

若要使用控制台在 Lambda 函數上切換主動追蹤，請按照下列步驟操作：

開啟主動追蹤

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇 組態 ，然後選擇 監控和操作工具 。
4. 選擇 編輯 。
5. 在 X-Ray 下，打開 主動追蹤 。
6. 選擇 儲存 。

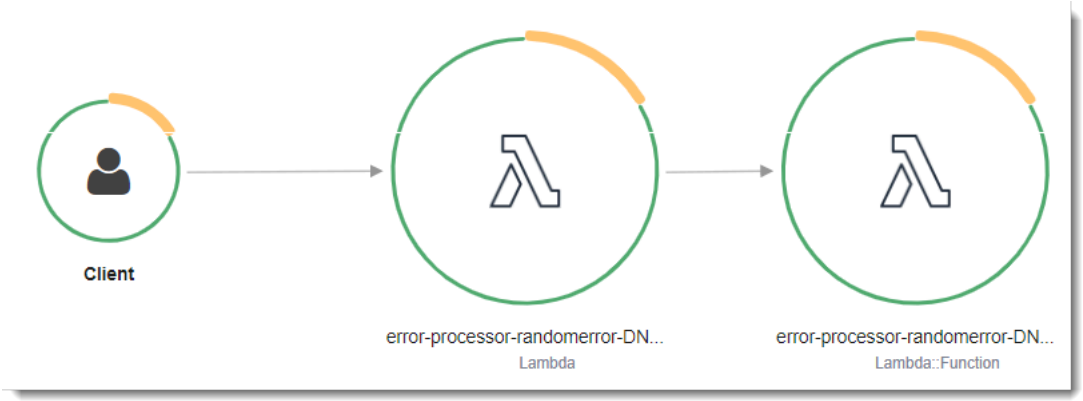
> **定價**
>
> 作為免費方案的一部分，您可以每月免費使用 X-Ray 追蹤，最多達到一定限制。 AWS 達到閾值後，X-Ray 會收取追蹤儲存及擷取的費用。如需詳細資訊，請參閱 AWS X-Ray 定價。

您的函數需要將追蹤資料上傳至 X-Ray 的許可。當您在 Lambda 主控台中啟用追蹤時，Lambda 會將必要的許可新增至函數的執行角色。否則，請將AWSXRayDaemonWriteAccess原則新增至執行角色。

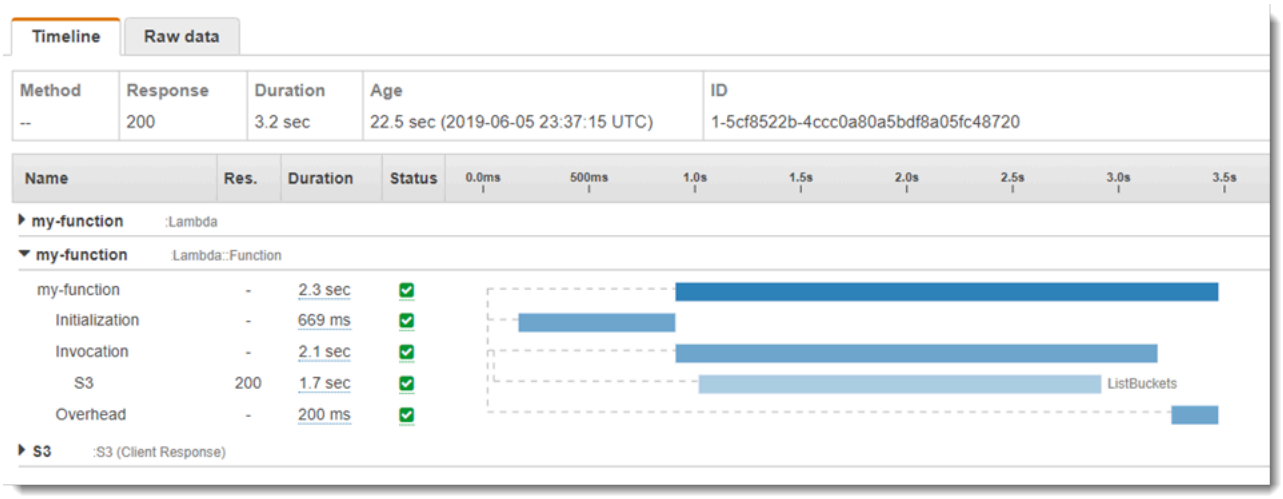
X-Ray 無法追蹤應用程式的所有請求。X-Ray 會套用取樣演算法以確保追蹤的效率，同時仍提供所有請求的代表範本。取樣率為每秒 1 次請求和 5% 的額外請求。

> **Note**
>
> 您無法針對函數設定 X-Ray 取樣率。

在 X-Ray 中，追蹤會記錄一或多個服務所處理之要求的相關資訊。Lambda 會記錄每個追蹤 2 個區段，在服務圖表上建立兩個節點。下列影像會強調顯示這兩個節點：



左側第一個節點代表接收調用請求的 Lambda 服務。第二個節點代表您特定的 Lambda 函數。下列範例顯示了具有這 2 個區段的追蹤。兩者都被命名為我的函數，但一個具有的起源AWS::Lambda，另一個具有的AWS::Lambda::Function起源。如果AWS::Lambda區段顯示錯誤，表示 Lambda 服務發生問題。如果AWS::Lambda::Function區段顯示錯誤，表示您的函數發生問題。



函數段（`AWS::Lambda::Function`）帶有`Initialization`，`Invocation`，`Restore`（Lambda SnapStart僅）和`Overhead`的子段。如需詳細資訊，請參閱 Lambda 執行環境生命週期。

> **Note**
>
> X-Ray 會將 Lambda 函數中未處理的例外狀況視為 `Error` 狀態。X-Ray 僅會在 Lambda 發生內部伺服器錯誤時記錄 `Fault` 狀態。如需詳細資訊，請參閱《X-Ray 開發人員指南》中的錯誤、故障和例外狀況。

`Initialization` 子區段代表 Lambda 執行環境生命週期的初始化階段。在此階段過程中，Lambda 會使用您已設定的資源建立或解除凍結執行環境，下載函數程式碼和所有層，初始化擴充功能，初始化執行時間，以及執行函數的初始化程式碼。

`Invocation` 子區段表示調用階段，其中 Lambda 調用函數處理常式。這從執行時間和延伸註冊開始，並在執行時間準備傳送響應時結束。

(僅限 Lambda SnapStart) `Restore` 子區段會顯示 Lambda 還原快照、載入執行期 (JVM) 和執行任何 `afterRestore` 執行期掛鉤所需的時間。還原快照的程序可能包括在 MicroVM 以外的活動上花費的時間。此時間在 `Restore` 子區段中報告。您不需要為在 MicroVM 外還原快照所花費的時間付費。

`Overhead` 子區段表示當執行時間傳送響應和下一次調用信號之間的時間發生的階段。在此期間，執行時間會完成與調用相關的所有工作，並準備凍結沙盒。

> **Note**
>
> 偶爾您可能會注意到 X-Ray 追蹤中的函數初始化和調用階段之間存在很大的差距。對於使用佈建並行的函數，這是因為 Lambda 會在調用之前才初始化函數執行個體。對於使用未預留 (隨需) 並行的函數，即使沒有調用，Lambda 也可能會主動初始化函數執行個體。從視覺上看，這兩種情況都會顯示為初始化和調用階段之間的時間差距。

> **Important**
>
> 在 Lambda 中，您可以使用 X-Ray 開發套件來擴充 `Invocation` 子區段與下游呼叫、註釋和中繼資料的其他子區段。您無法直接存取函數區段，或在處理常式調用範圍之外記錄完成的工作。

如需在 Lambda 追蹤的語言特定簡介，請參閱下列主題：

- 在中檢測 Node.js 程式碼 AWS Lambda
- 檢測 Python 代碼 AWS Lambda
- 檢測 Ruby 代碼 AWS Lambda
- 在中檢測 Java 程式碼 AWS Lambda
- 檢測 Go 代碼 AWS Lambda
- 檢測 C# 代碼 AWS Lambda

如需支援使用中檢測的服務的完整清單，請參閱 AWS X-Ray 開發人員指南中的支援 AWS 服務。

章節

- 執行角色許可
- AWS X-Ray 守護進程
- 透過 Lambda API 啟用主動追蹤
- 啟用作用中追蹤 AWS CloudFormation

執行角色許可

Lambda 需要下列許可才能傳送追蹤資料到 X-Ray。新增許可到您的函數的執行角色。

- X 射線:PutTrace區段
- X 射線：PutTelemetry記錄

這些權限包含在AWSXRayDaemonWriteAccess受管理的策略中。

AWS X-Ray 守護進程

X-Ray 開發套件不是直接將追蹤資料傳送至 X-Ray API，而是使用協助程序。 AWS X-Ray 常駐程式是在 Lambda 環境中執行的應用程式，會偵聽包含區段和子區段的 UDP 流量。它會緩衝傳入的資料並分批寫入 X-Ray，減少追蹤調用所需的處理和記憶體負荷。

Lambda 執行時間允許常駐程式最多達 3% 的函數設定記憶體或 16 MB (以較大者為準)。如果您的函數在調用過程中耗盡內存，則執行時間會首先終止常駐程式以釋放內存。

常駐程式程序由 Lambda 完全管理，且無法由使用者設定。函數調用所產生的所有區段都會記錄在與 Lambda 函數相同的帳戶中。無法將常駐程式設定為將它們重新導向至任何其他帳戶。

如需詳細資訊，請參閱《 X-Ray 開發人員指南》中的 X-Ray 常駐程式。

透過 Lambda API 啟用主動追蹤

若要使用 AWS CLI 或 AWS SDK 管理追蹤組態，請使用下列 API 作業：

- UpdateFunction配置
- GetFunction配置
- CreateFunction

下列範例 AWS CLI 命令可在名為 my- function 的函式上啟用主動追蹤。

```
aws lambda update-function-configuration \
--function-name my-function \
--tracing-config Mode=Active
```

追蹤模式是您發布函數版本時版本特定組態的一部分。您無法變更已發佈版本上的追蹤模式。

啟用作用中追蹤 AWS CloudFormation

若要啟動 AWS CloudFormation 範本中的`AWS::Lambda::Function`資源追蹤，請使用`TracingConfig`屬性。

Example function-inline.yml - 追蹤組態

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

對於 AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` 資源，請使用`Tracing`屬性。

Example template.yml - 追蹤組態

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

使用 Amazon CloudWatch Lambda 見解監控功能效

Amazon CloudWatch Lambda 見解會為您的無伺服器應用程式收集和彙總 Lambda 函數執行時期效能指標和日誌。本頁說明如何啟用和使用 Lambda Insights 來診斷 Lambda 函數的問題。

章節

Lambda Insights 如何監控無伺服器應用程式

CloudWatch Lambda 洞察是一種監控和故障排除解決方案，適用於在上 AWS Lambda執行的無伺服此解決方案會收集、彙總和摘要系統層級的指標，包括 CPU 時間、記憶體、磁碟和網路使用量。它也會收集、彙總和摘要診斷資訊，例如冷啟動和 Lambda 工作人員關閉，協助您隔離 Lambda 函數問題並快速加以解決。

Lambda 深入解析使用全新的 CloudWatch Lambda 見解延伸模組，該延伸模組是以 Lambda 層形式 當您在支援的執行階段的 Lambda 函數上啟用此延伸模組時，它會收集系統層級指標，並針對該 Lambda 函數的每次叫用發出單一效能記錄事件。 CloudWatch 使用內嵌指標格式從記錄事件擷取指標。如需詳細資訊，請參閱使用 AWS Lambda 擴充功能。

Lambda Insights 層會擴展 `/aws/lambda-insights/` 日誌群組的 `CreateLogStream` 和 `PutLogEvents`。

定價

當您為 Lambda 函數啟用 Lambda 深入解析時，Lambda 洞見會報告每個函數 8 個指標，而且每次函數呼叫都會傳送約 1 KB 的記錄資料給。 CloudWatch您只需要為 Lambda Insights 報告的函數指標和

記錄付費。沒有最低費用或強制性的服務使用政策。如果未叫用函數，則不需為 Lambda Insights 支付費用。如需定價範例，請參閱 Amazon CloudWatch 定價。

支援的執行期

您可以使用 Lambda Insights 搭配任何支援 Lambda 擴展功能的執行時間。

在 Lambda 主控台中啟用 Lambda Insights

您可以對新函數和現有 Lambda 函數啟用 Lambda Insights 增強型監控功能。當您在 Lambda 主控台的函數中針對支援的執行時間啟用 Lambda Insights 時，Lambda 會將 Lambda Insights 擴展功能新增至您的函數，並驗證或嘗試將 `CloudWatchLambdaInsightsExecutionRolePolicy` 政策連接至函數的執行角色。

若要在 Lambda 主控台中啟用 Lambda Insights

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇函數。
3. 選擇 Configuration (組態) 索引標籤。
4. 在左側菜單中，選擇監視和操作工具。
5. 在 [其他監視工具] 窗格中，選擇 [編輯]。
6. 在 CloudWatch Lambda 深入解析下，開啟增強型監控。
7. 選擇儲存。

以程式設計方式啟用 Lambda Insights

您也可 Lambda 使用 AWS Command Line Interface (AWS CLI)、 AWS Serverless Application Model (SAM) CLI 或 AWS Cloud Development Kit (AWS CDK). AWS CloudFormation當您以程式設計方式在受支援的執行階段的函數上啟用 Lambda Insights 時，會將`CloudWatchLambdaInsightsExecutionRolePolicy`原則 CloudWatch 附加至函數的執行角色。

如需詳細資訊，請參閱 Amazon CloudWatch 使用者指南中的 Lambda 洞察入門。

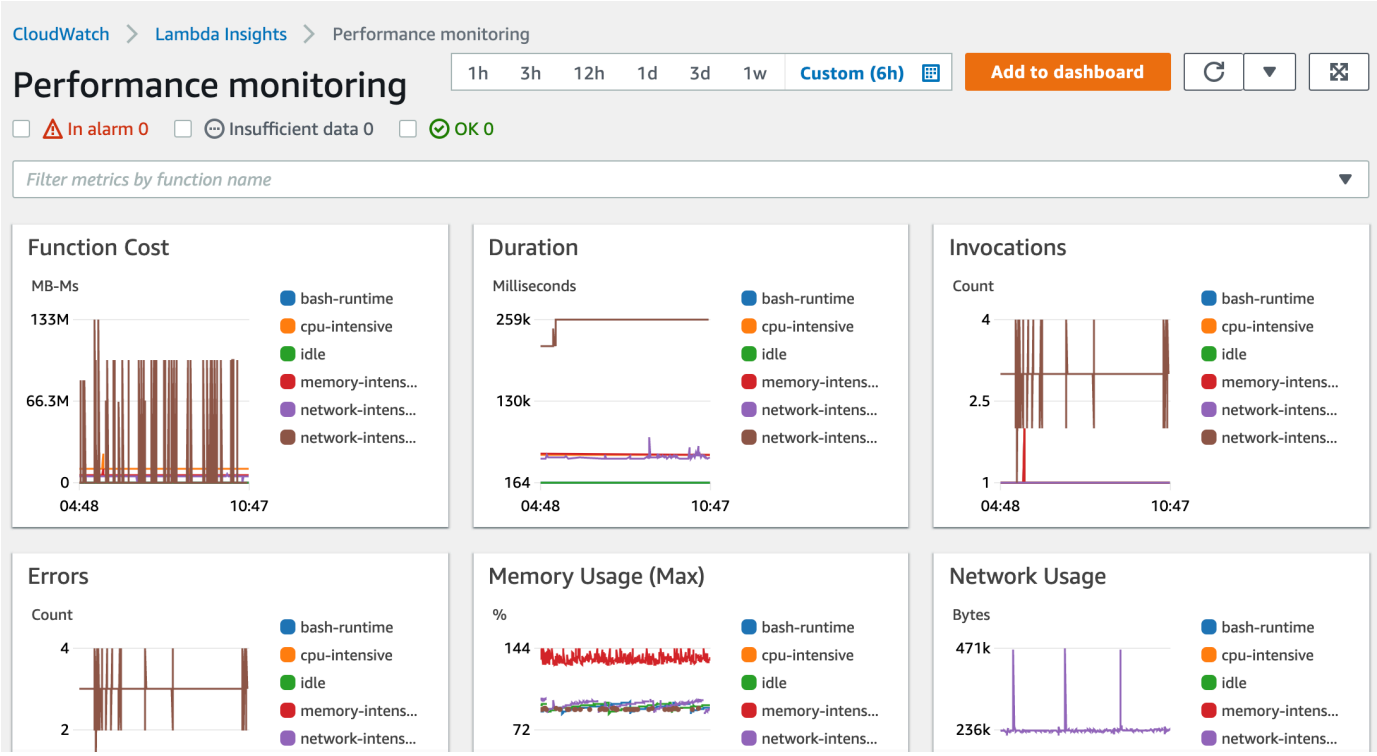
使用 Lambda Insights 儀表板

Lambda 洞見儀表板在 CloudWatch 主控台中有兩個檢視：多功能概觀和單一功能檢視。多功能概觀彙總了目前 AWS 帳戶和區域中 Lambda 函數的執行階段指標。單一函數檢視會顯示單一 Lambda 函數的可用執行時間指標。

您可以使用 CloudWatch 主控台中的 Lambda 見解儀表板多功能概觀來識別使用過度和未充分利用的 Lambda 函數。您可以使用 CloudWatch 主控台中的 Lambda 見解儀表板單一功能檢視，對個別請求進行疑難排解。

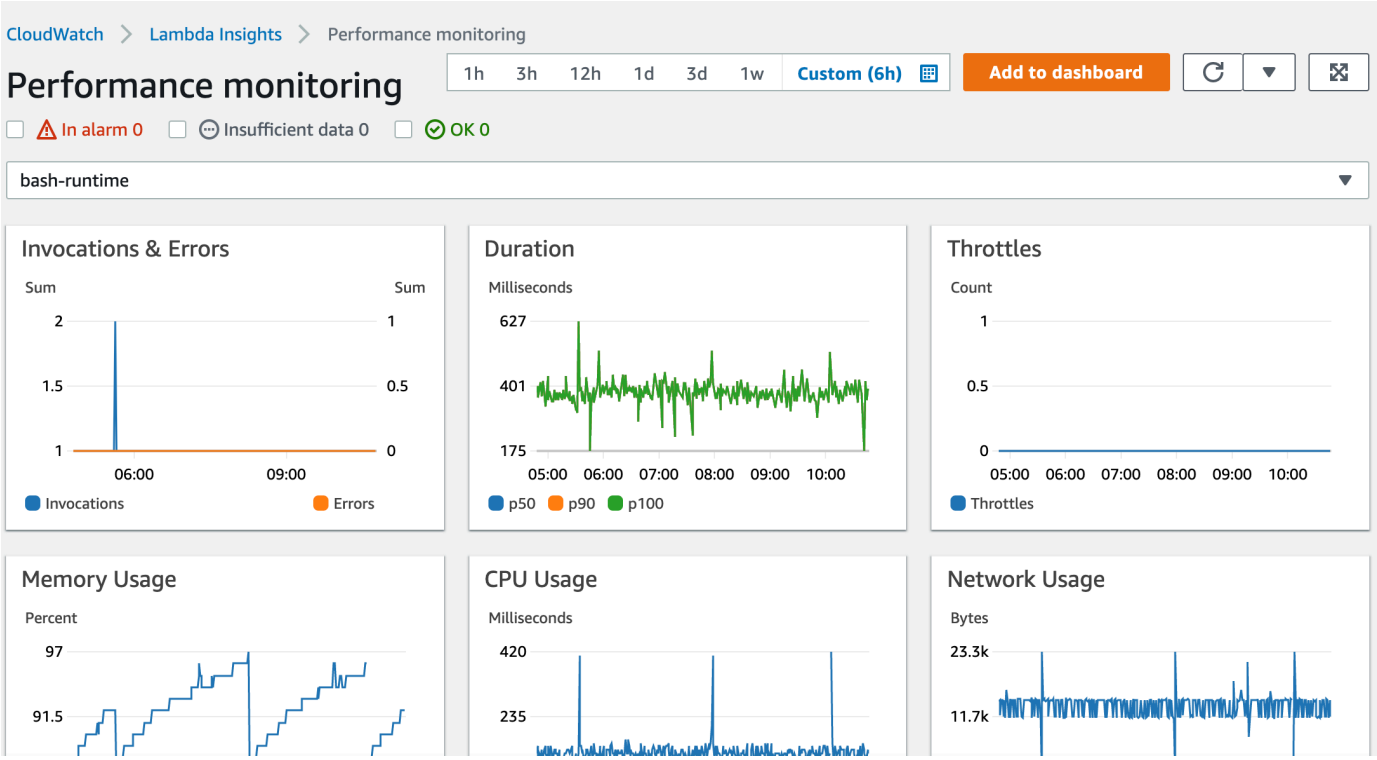
檢視所有函式的執行階段指標

1. 開啟主控台中的「多功能」頁 CloudWatch 面。
2. 從預先定義的時間範圍中選擇，或選擇自訂的時間範圍。
3. (選擇性) 選擇「新增至儀表板」，將 Widget 新增至 CloudWatch 儀表板。



檢視單一函式的執行階段指標

1. 開啟主 CloudWatch 控台中的單一功能頁面。
2. 從預先定義的時間範圍中選擇，或選擇自訂的時間範圍。
3. (選擇性) 選擇「新增至儀表板」，將 Widget 新增至 CloudWatch 儀表板。



如需詳細資訊，請參閱在 CloudWatch 儀表板上建立和使用小器具。

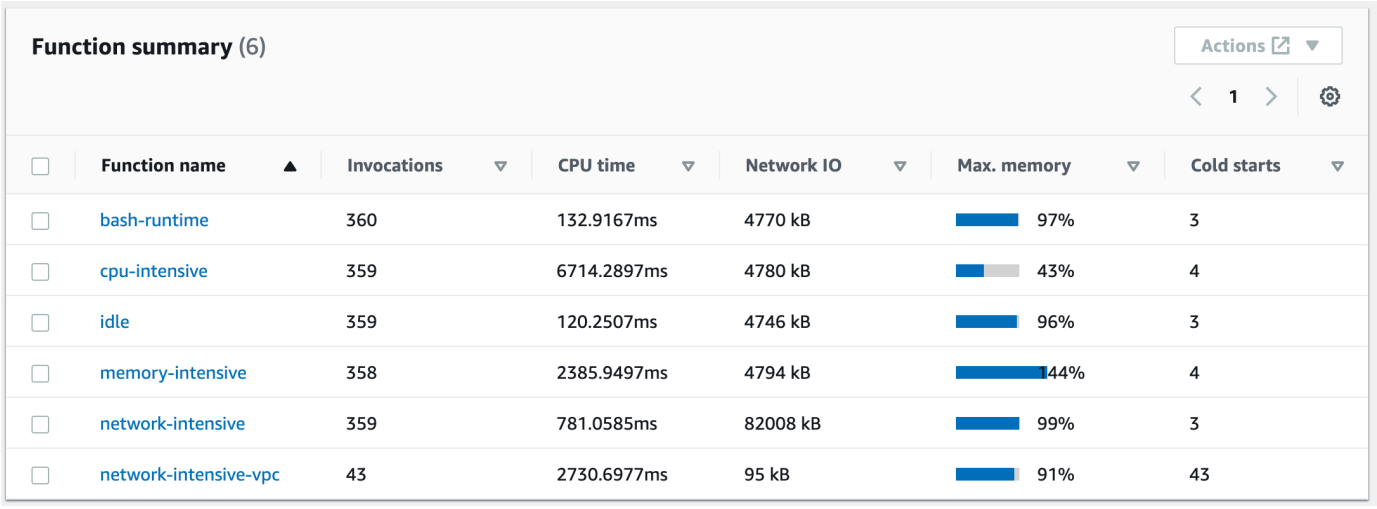
偵測函式異常的工作流程範例

您可以使用 Lambda Insights 儀表板上的多函數概觀來識別和偵測您的函數是否有運算記憶體異常。例如，如果多函式概觀指出某個函式正在使用大量記憶體，您可以在記憶體使用量窗格中檢視詳細的記憶體使用量。然後，您可以移至「指標」儀表板以啟用異常偵測或建立警示。

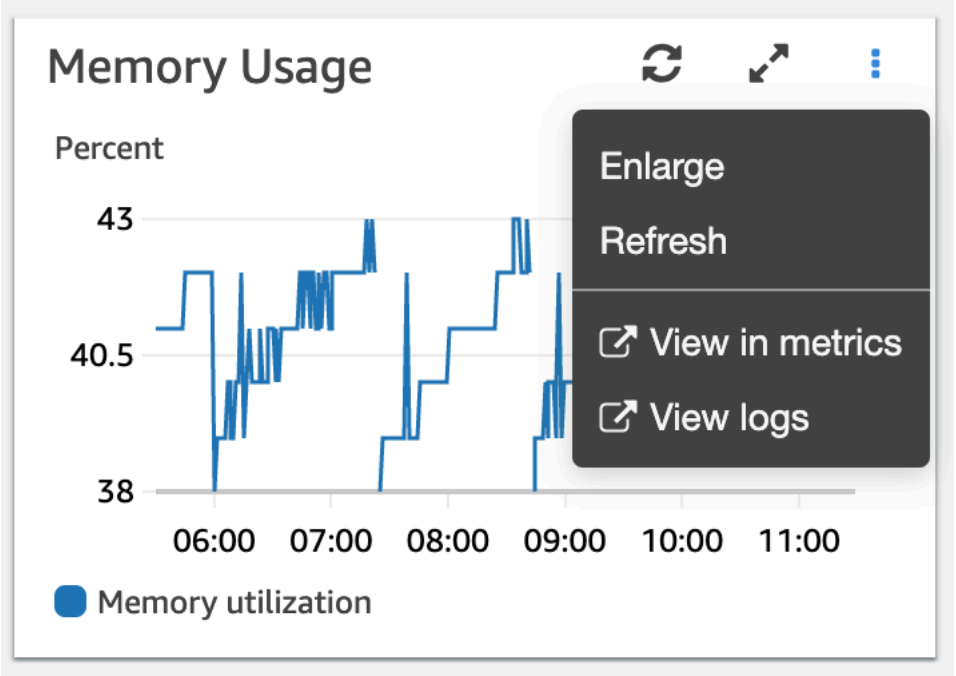
啟用對函式的異常偵測

1. 開啟主控台中的「多功能」頁 CloudWatch 面。
2. 在函式摘要下方，選擇您的函式名稱。

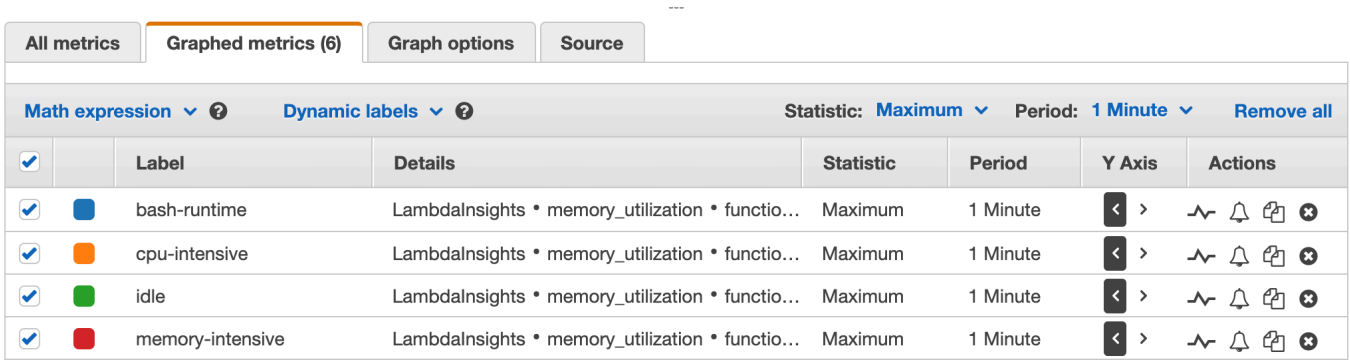
   單一函式檢視隨即開啟，其中包含函式執行階段指標。



3. 在記憶體使用量窗格中，選擇三個垂直點，然後選擇在指標中檢視以開啟指標儀表板。



4. 在圖形化指標索引標籤的動作欄中，選擇第一個圖示，以啟用對函式的異常偵測。



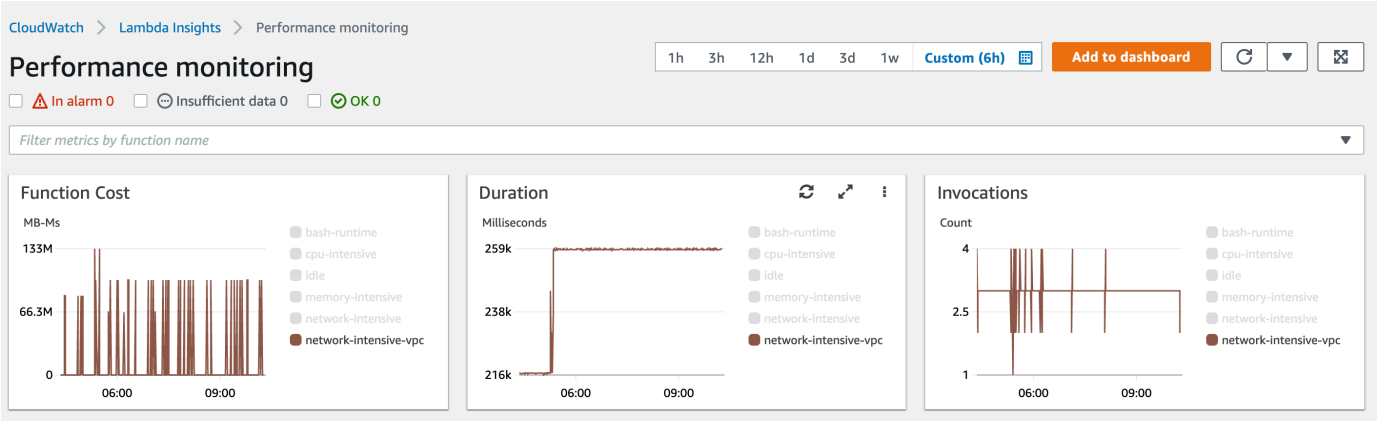
如需詳細資訊，請參閱使用 CloudWatch 異常偵測。

使用查詢故障排除函式的範例工作流程

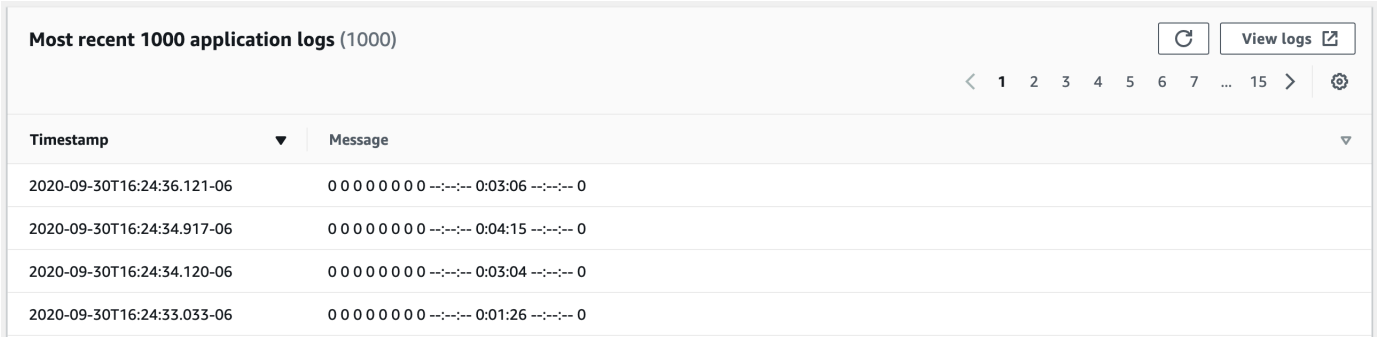
您可以使用 Lambda Insights 儀表板上的單一函數檢視來識別函數持續時間中峰值的根本原因。例如，如果多函式概觀指出函式持續時間大幅增加，您可以暫停或選擇持續時間窗格中的每個函式，以判斷哪個函式造成增加。然後，您可以移至單一函式檢視並檢閱應用程式日誌，以判斷根本原因。

在函式上執行查詢

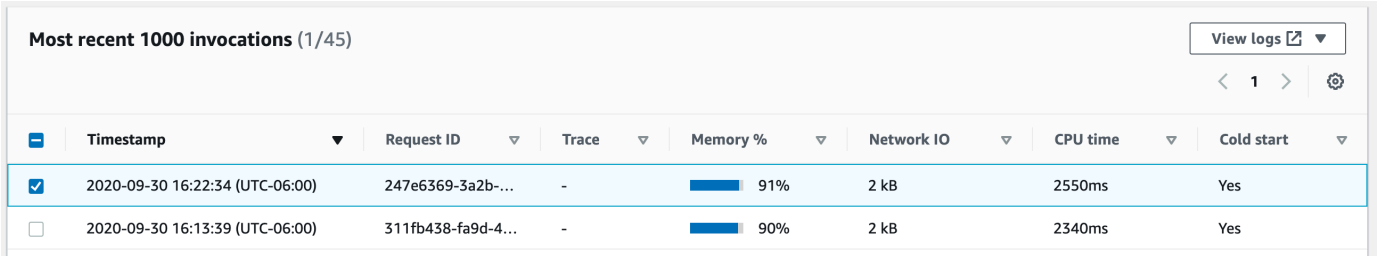
1. 開啟主控台中的「多功能」頁 CloudWatch 面。
2. 在持續時間窗格中，選擇您的函式以篩選持續時間量度。



3. 開啟單一函式頁面。
4. 選擇依函式名稱篩選指標下拉式清單，然後選擇您的函式。
5. 若要檢視 最近 1000 個應用程式日誌，請選擇應用程式日誌索引標籤。
6. 檢閱時間戳記和訊息，以識別您要故障排除的叫用請求。



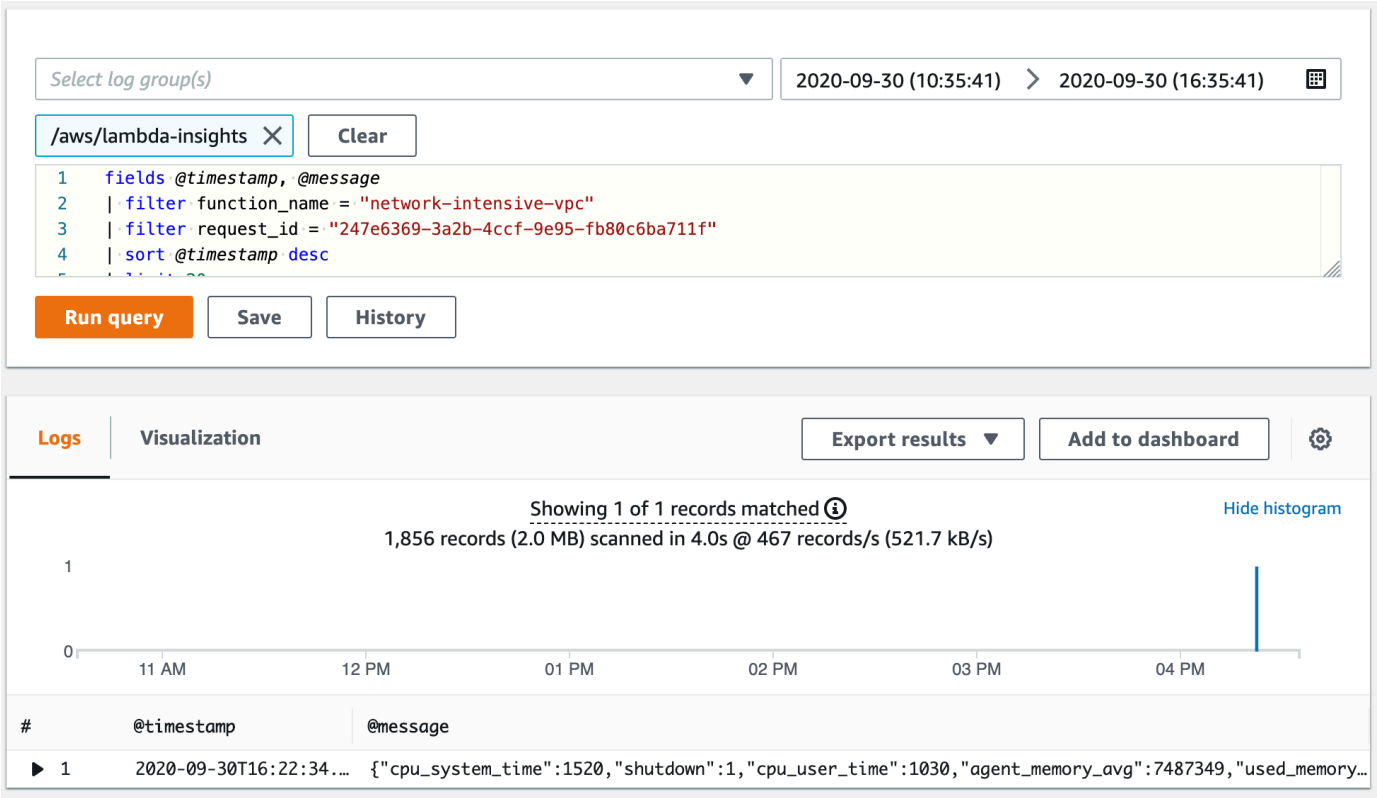
7. 若要顯示最近的 1000 次叫用，請選擇叫用索引標籤。
8. 針對您要故障排除的叫用請求，選取時間戳記或訊息。



9. 選擇檢視日誌的下拉式清單，然後選擇檢視效能日誌。

   您的函式自動產生的查詢會在日誌深入資訊儀表板中開啟。

10. 選擇執行查詢以產生叫用請求的日誌訊息。



後續步驟？

- 在 Amazon CloudWatch 使用者指南中的建立儀表板中了解如何建立 CloudWatch 日誌儀表板。
- 在 Amazon CloudWatch 使用者指南中的「新增查詢到儀表板」或「匯出查詢結果」中，了解如何將查詢新增至 CloudWatch 日誌儀表板。

搭配 Lambda 函數使用 CodeGuru 效能分析工具

您可以使用 Amazon 效能分 CodeGuru 析工具，深入瞭解 Lambda 函數的執行階段效能。本頁說明如何從 Lambda 主控台啟動 CodeGuru 效能分析工具。

章節

支援的執行期

如果函數的執行階段為 Python 3.8、Python 3.9、Java 8，搭配 Amazon Linux 2、Java 11 或 Java 17，您可以從 Lambda 主控台啟動 CodeGuru 效能分析工具。對於其他執行階段版本，您可以手動啟動 CodeGuru 效能評測工具。

- 對於 Java 執行時間，請參閱分析在 AWS Lambda 上執行的 Java 應用程式。
- 對於 Python 執行時間，請參閱分析在 AWS Lambda 上執行的 Python 應用程式。

> **Note**
>
> CodeGuru 效能分析工具目前僅支援使用 x 86_64 架構的函數。

從 Lambda 主控台啟用 CodeGuru 效能分析工具

本節說明如何從 Lambda 主控台啟動 CodeGuru 效能分析工具。

若要從 Lambda 主控台啟動 CodeGuru 效能分析工具

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇函數。
3. 選擇 Configuration (組態) 索引標籤。
4. 在 Monitoring and operations tools (監控和操作工具) 窗格中，選擇 Edit (編輯)。
5. 在 Amazon CodeGuru 效能分析工具下，開啟程式碼分析。

6. 選擇儲存。

啟動之後，CodeGuru 會自動建立名稱的效能分析工具群組。aws-lambda-<your-function-name>您可以從 CodeGuru 控制台更改名稱。

當您從 Lambda 主控台啟動 CodeGuru 效能分析工具時會發生什麼情況？

當您從主控台啟用 CodeGuru 效能分析工具時，Lambda 會自動代表您執行下列動作：

- Lambda 會將效能 CodeGuru 分析工具圖層新增至您的函數。如需詳細資訊，請參閱 Amazon CodeGuru 效能評測工具使用者指南中的使用AWS Lambda圖層。
- Lambda 也會將環境變數新增到您的函數中。具體的值會根據執行時間而有所不同。

  環境變數

執行時間	金鑰	值
java8.al2、java11	JAVA_TOOL_OPTIONS	-javaagent:/opt/codeguru-profiler-java-agent-standalone.jar
python3.8、python3.9	AWS_LAMBDA_EXEC_WRAPPER	/opt/codeguru_profiler_lambda_exec

- Lambda 會將 AmazonCodeGuruProfilerAgentAccess 政策新增至函數的執行角色。

> **Note**
>
> 當您從主控台停用 CodeGuru 效能分析工具時，Lambda 會自動從您的函數中移除效能分 CodeGuru 析工具層。不過，Lambda 並不會從您的執行角色中移除環境變數或 AmazonCodeGuruProfilerAgentAccess 政策。

後續步驟？

- 請參閱 Amazon Profiler 使用者指南中的使用視覺效果，進一步了解效能分析工具群組所收集的資 CodeGuru 料。

使用其他 AWS 服務的範例工作流程

AWS Lambda 與其他 AWS 服務整合，以協助您監控、追蹤、偵錯和疑難排解 Lambda 函數。此頁面顯示您可以與 AWS X-Ray 和 AWS Trusted Advisor 搭配使用的工作流程，來追蹤 Lambda 函數並對其進行疑難排解。

章節

必要條件

下一節說明使用 AWS X-Ray 和 Trusted Advisor 來對 Lambda 函數進行疑難排解的步驟。

使用 AWS X-Ray

必須在 Lambda 主控台上啟用 AWS X-Ray 才能完成此頁面上的 AWS X-Ray 工作流程。如果您的執行角色沒有必要的許可，Lambda 主控台會嘗試將它們新增至您的執行角色。

若要在 Lambda 主控台上啟用 AWS X-Ray

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇函數。
3. 選擇 Configuration (組態) 索引標籤。
4. 在監控工具窗格上，選擇編輯。
5. 在 AWS X-Ray 下，開啟 Active tracing (作用中追蹤)。
6. 選擇儲存。

使用 AWS Trusted Advisor

AWS Trusted Advisor 會檢查您的 AWS 環境，並在節省成本、提升系統可用性與效能以及協助填補安全漏洞時向您提出建議。您可以使用 Trusted Advisor 檢查，以評估 AWS 帳戶中的 Lambda 函數和應用程式。此檢查提供建議採取的步驟和提供更多資訊的資源。

- 如需適用於 Trusted Advisor 檢查的 AWS 支援方案的詳細資訊，請參閱支援方案。
- 如需有關檢查 Lambda 的詳細資訊，請參閱 AWS Trusted Advisor 最佳實務檢查清單。
- 如需有關如何使用 Trusted Advisor 主控台的詳細資訊，請參閱開始使用 AWS Trusted Advisor。
- 如需如何允許和拒絕主控台對 Trusted Advisor 的存取權說明，請參閱 IAM 政策範例。

定價

- 使用 AWS X-Ray，您只需根據已記錄、擷取及掃描的追蹤數量，按使用量付費。如需詳細資訊，請參閱 AWS X-Ray 定價。
- AWS 商業和企業支援訂閱隨附 Trusted Advisor 成本最佳化檢查。如需詳細資訊，請參閱 AWS Trusted Advisor 定價。

用於檢視追蹤地圖的範例 AWS X-Ray 工作流程

如果您已啟用AWS X-Ray，您可以在 CloudWatch 主控台上檢視軌跡地圖。追蹤地圖會將您的服務端點和資源顯示為「節點」並重點標示每個節點及其連線的流量、延遲及錯誤。

您可以選擇節點來查看與該服務部分相關聯的相互關聯指標、日誌及追蹤詳情。這可以讓您調查問題，以及問題對應用程式造成的影響。

若要使用 CloudWatch 主控台檢視軌跡對應和追蹤

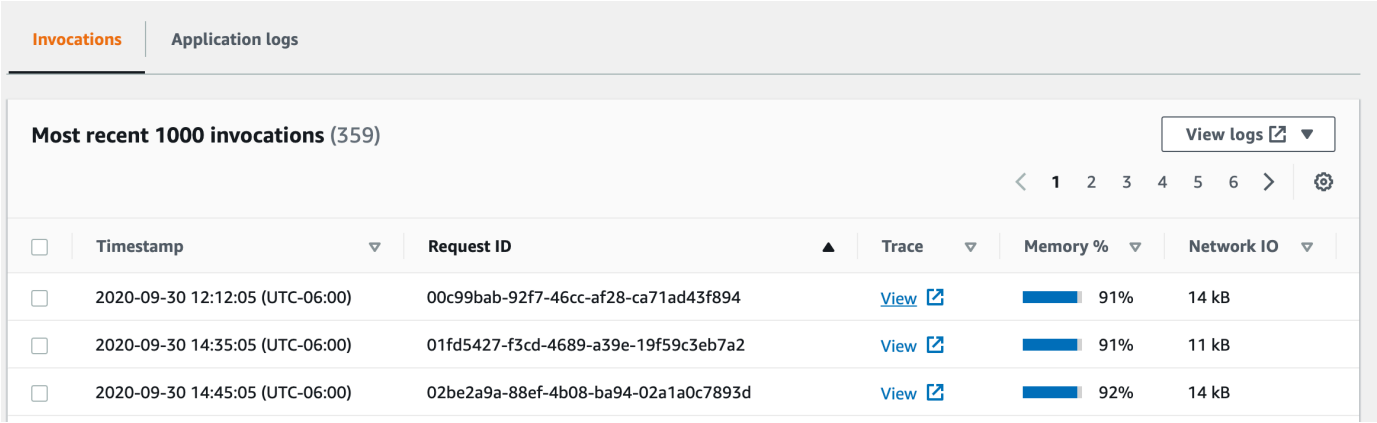
1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇 Monitoring (監控)。
4. 選擇檢視 X-Ray 追蹤。
5. 在左側導覽窗格中，選擇 X-Ray 追蹤下方的追蹤地圖。
6. 從預先定義的時間範圍中選擇，或選擇自訂的時間範圍。
7. 若要對請求進行疑難排解，請選擇篩選條件。

檢視追蹤詳細資訊的範例 AWS X-Ray 工作流程

如果您已啟用AWS X-Ray，則可以使用 CloudWatch Lambda Insights 儀表板上的單一功能檢視來顯示函數叫用錯誤的分散式追蹤資料。例如，如果應用程式日誌訊息顯示錯誤，您可以開啟追蹤地圖，以查看分散式追蹤資料和處理交易的其他服務。

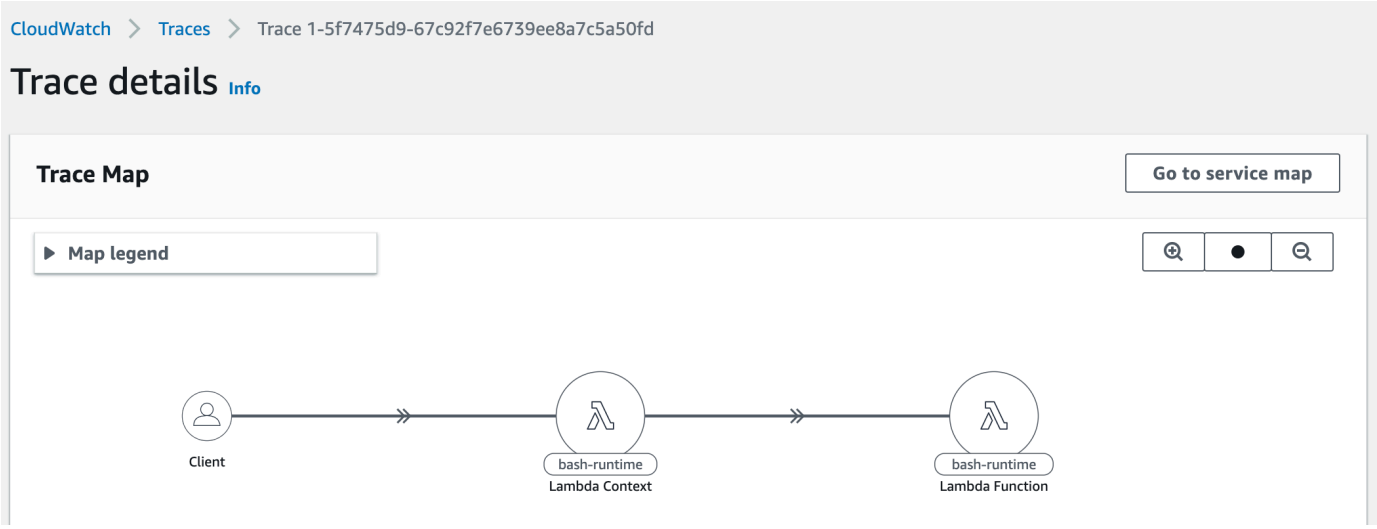
檢視函數的追蹤詳細資訊

1. 在 CloudWatch 主控台中開啟單一功能檢視。
2. 選擇 Application logs (應用程式日誌) 索引標籤。
3. 使用 Timestamp (時間戳記) 和 Message (訊息)，以識別您要疑難排解的叫用請求。
4. 若要顯示最近的 1000 次叫用，請選擇叫用索引標籤。



5. 選擇 Request ID (請求 ID) 欄，依字母遞增順序排序項目。
6. 在 Trace (追蹤) 欄中，選擇 View (檢視)。

追蹤詳細資訊頁面會在追蹤檢視中開啟。



檢視建議的 AWS Trusted Advisor 工作流程範例

Trusted Advisor 會在所有 AWS 區域中檢查 Lambda 函數，以識別最可能節省成本的函數，並提供可行的最佳化建議。它會分析 Lambda 使用量資料，例如函數執行時間、計費持續時間、使用的記憶體、設定的記憶體、逾時組態和錯誤。

例如，具有高錯誤率檢查的 Lambda 函數建議您使用AWS X-Ray或 CloudWatch 偵測 Lambda 函數的錯誤。

檢查具有高錯誤率的函數

1. 開啟 Trusted Advisor 主控台。
2. 選擇 Cost Optimization (成本最佳化) 類別。
3. 向下捲動至 AWS Lambda Functions with High Error Rate (具有高錯誤率的 Lambda 函數)。展開區段以查看結果和建議的動作。

後續步驟？

- 參閱使用 X-Ray 追蹤地圖，進一步了解如何整合追蹤、指標、日誌和警示。
- 在使用 Trusted Advisor 作為 Web 服務中進一步了解如何取得 Trusted Advisor 檢查的清單。

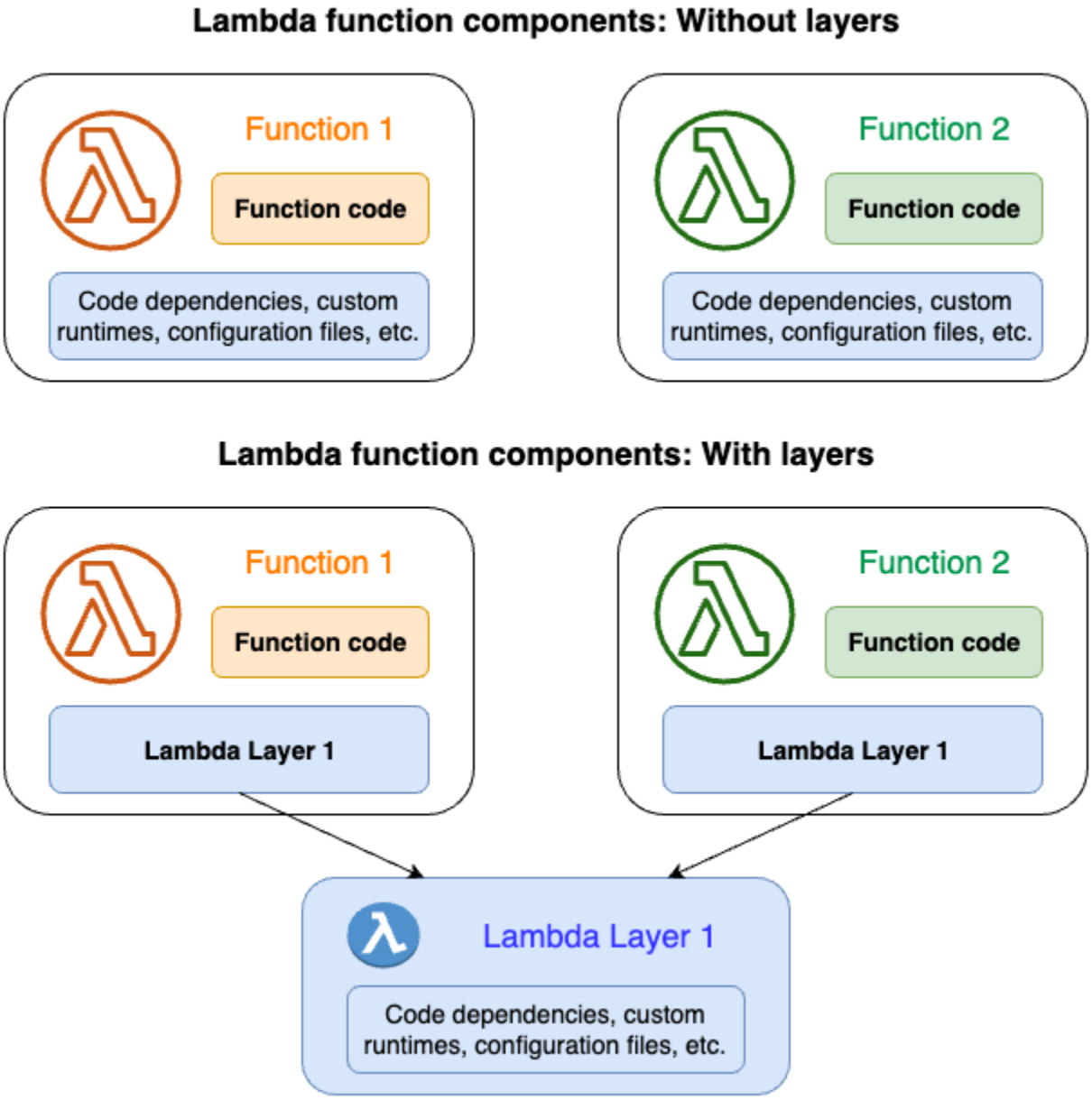
使用層管理 Lambda 相依性

Lambda 層是含有補充程式碼或資料的 .zip 封存檔。層通常具備程式庫相依性、自訂執行期或組態檔案。

以下是您可能會考慮使用層的多種原因：

- 縮減部署套件的大小。切勿將所有函數的相依項以及函數程式碼加入部署套件，而是將它們放在層裡面。這可以使部署套件在容量小的情況下同時保持條理。
- 若要將核心函數邏輯與相依項分隔開來。您可以透過層獨立於函數程式碼更新函數相依項，反之亦然。這能夠促進關注點分離的原則，且有助於您將重心放在函數邏輯上。
- 若要跨多個函數共享相依項。建立層後，您可以將其套用於帳戶中的函數，數量無任何限制。如果沒有使用層，則必須在每個個別的部署套件中加入相同的相依項。
- 若要使用 Lambda 主控台程式碼編輯器。程式碼編輯器是快速測試次要函數程式碼更新的實用工具。不過，如果您的部署套件太大，便無法使用編輯器。使用層可以縮減套件的大小，並取得程式碼編輯器的使用權限。

下圖會說明共用相依項的兩個函數之間的概略架構差異。一個函數使用 Lambda 層，而另一個函數則不使用。



將層新增至 Lambda 函數時，Lambda 會將層內容擷取至函數執行環境中的 `/opt` 目錄。所有原生支援的 Lambda 執行期皆包含 `/opt` 目錄中特定目錄的路徑。如此一來，您的函數便可以存取您的層內容。如需有關這類特定路徑以及如何正確封裝層的詳細資訊，請參閱 the section called “封裝層”。

每個函數最多可包含五個圖層。此外，您只能將層與 部署為 .zip 封存檔 的 Lambda 函數搭配使用。對於定義為容器映像的函數，您可以在建立容器映像時封裝偏好的執行期和所有程式碼相依項。如需詳細資訊，請參閱運 AWS 算部落格上的使用容器映像中的 Lambda 層和擴充功能。

主題

如何使用層

若要建立層，請將相依項封裝到 .zip 檔案中，方法類似於您建立一般部署套件的方式。更具體來說，建立和使用層的一般程序包括以下三個步驟：

- 首先，封裝層內容。這表示您必須建立一個 .zip 封存檔。如需詳細資訊，請參閱 the section called “封裝層”。
- 接著，在 Lambda 中建立層。如需詳細資訊，請參閱 the section called “建立和刪除層”。
- 將層新增到您的函數中。如需詳細資訊，請參閱 the section called “新增層”。

層和層的版本

層版本是特定層版本不可變的快照。建立新層時，Lambda 會建立版本編號為 1 的新層版本。每次將更新發佈至層時，Lambda 都會遞增版本編號並建立新的層版本。

每個層版本皆由唯一的 Amazon Resource Name (ARN) 進行識別。向函數新增層時，您必須指定要使用的確切層版本。

封裝層內容

Lambda 層是含有補充程式碼或資料的 .zip 封存檔。層通常具備程式庫相依性、自訂執行期或組態檔案。

本節會說明如何正確封裝層內容。若要進一步了解有關層的概念性資訊以及您可能會考慮使用的原因，請參閱 Lambda 層。

建立層的第一步是將所有層內容綁定至 .zip 封存檔。由於 Lambda 函數是在 Amazon Linux 上執行，因此您的層內容必須能夠在 Linux 環境中編譯和建置。

若要確保您的圖層內容在 Linux 環境中正常運作，我們建議您使用 Docker 或 AWS Cloud9. AWS Cloud9 是一個基於雲的集成開發環境（IDE），提供對 Linux 服務器的內置訪問以運行和測試代碼。如需詳細資訊，請參閱 AWS 運算部落格上的 使用 Lambda 層來簡化您的開發程序。

主題

- 每個 Lambda 執行時間的層路徑

每個 Lambda 執行時間的層路徑

將層新增至函數時，Lambda 會將層內容載入該執行環境的 `/opt` 目錄。在每一次 Lambda 執行期中，PATH 變數已包含 `/opt` 目錄中的特定資料夾路徑。若要確保PATH變數會取得圖層內容，您的圖層 .zip 檔案應該在下列資料夾路徑中具有其相依性：

每個 Lambda 執行時間的層路徑

執行期	路徑
Node.js	`nodejs/node_modules`
	`nodejs/node14/node_modules` (NODE_PATH)
	`nodejs/node16/node_modules` (NODE_PATH)
	`nodejs/node18/node_modules` (NODE_PATH)
Python	`python`
	`python/lib/` *`python3.x`* `/site-packages` (網站目錄)

執行期	路徑
Java	`java/lib` (`CLASSPATH`)
Ruby	`ruby/gems/3.2.0` (`GEM_PATH`)
所有執行時間	`ruby/lib` (`RUBYLIB`)
	`bin` (`PATH`)
	`lib` (`LD_LIBRARY_PATH`)

下列範例展示如何在圖層 .zip 封存中建構資料夾。

Node.js

Example 適用於 Node.js 的 AWS X-Ray 開發套件的檔案結構

```
xray-sdk.zip
# nodejs/node_modules/aws-xray-sdk
```

Python

Example 請求庫的文件結構

```
layer_content.zip
# python
    # lib
        # python3.11
            # site-packages
                # requests
                # <other_dependencies> (i.e. dependencies of the requests package)
                # ...
```

Ruby

Example JSON gem 的檔案結構

```
json.zip
# ruby/gems/2.7.0/
```

```
                | build_info
                | cache
                | doc
                | extensions
                | gems
                | # json-2.1.0
                # specifications
                  # json-2.1.0.gemspec
```

Java

Example Jackson JAR 檔案的檔案結構

```
layer_content.zip
# java
    # lib
        # jackson-core-2.17.0.jar
        # <other potential dependencies>
        # ...
```

All

Example JQ 程式庫的檔案結構

```
jq.zip
# bin/jq
```

如需有關封裝、建立和新增圖層的語言特定說明，請參閱下列頁面：

- Python – the section called “圖層”
- 爪哇 — the section called “圖層”

在 Lambda 中建立和刪除層

Lambda 層是含有補充程式碼或資料的 .zip 封存檔。層通常具備程式庫相依性、自訂執行期或組態檔案。

本節會說明如何在 Lambda 中建立和刪除層。若要進一步了解有關層的概念性資訊以及您可能會考慮使用的原因，請參閱 Lambda 層。

封裝層內容 後，下一步是在 Lambda 中建立層。本節會示範如何僅使用 Lambda 主控台或 Lambda API 建立和刪除層。若要使用 AWS CloudFormation 建立層，請參閱 the section called “具有的圖層 AWS CloudFormation”。若要使用 AWS Serverless Application Model (AWS SAM) 建立層，請參閱 the section called “具有的圖層 AWS SAM”。

主題

- 建立圖層
- 刪除圖層版本

建立圖層

若要建立層，您可以從本機電腦或 Amazon Simple Storage Service (Amazon S3) 中上傳 .zip 封存檔。設定函數的執行環境時，Lambda 會將層內容擷取到 `/opt` 目錄中。

層可以有一個或多個 層版本。建立層時，Lambda 將層版本設定為版本 1。您可以隨時變更既有層版本的許可。不過，若要更新程式碼或進行其他組態變更，您必須建立新的層版本。

建立圖層 (主控台)

1. 開啟 Lambda 主控台中的 層頁面。
2. 選擇 建立圖層。
3. 在 Layer configuration (圖層組態) 下，為 Name (名稱) 輸入圖層的名稱。
4. (選用) 在 Description (說明) 中，輸入 Layer 的說明。
5. 若要上傳 Layer 程式碼，請執行下列其中一個動作：
   - 若要從電腦上傳 .zip 檔案，請選擇 Upload a .zip file (上傳 .zip 檔案)。然後，選擇 Upload (上傳) 以選取您的本機 .zip 檔案。
   - 若要從 Amazon S3 上傳檔案，請選擇 Upload a file from Amazon S3 (從 Amazon S3 上傳檔案)。然後，對於 Amazon S3 連結 URL，輸入檔案的連結。

6. (選用) 對於相容架構，選擇一個值或兩個值。如需詳細資訊，請參閱 the section called “指令集 (ARM/x86)”。
7. (選擇性) 在 相容執行期 中選擇相容於您的層的執行期。
8. (選擇性) 在 License (授權) 中，輸入任何必要的授權資訊。
9. 選擇建立。

或者，您也可以使用 PublishLayerVersionAPI 建立圖層。例如，您可以使用 `publish-layer-version` AWS Command Line Interface (CLI) 命令並指定名稱、指示和 .zip 封存檔。授權資訊、相容執行期以及相容架構參數皆為選填。

```
aws lambda publish-layer-version --layer-name my-layer \
    --description "My layer" \
    --license-info "MIT" \
    --zip-file fileb://layer.zip \
    --compatible-runtimes python3.10 python3.11 \
    --compatible-architectures "arm64" "x86_64"
```

您應該會看到類似下列的輸出：

```
{
    "Content": {
        "Location": "https://awslambda-us-east-2-layers.s3.us-east-2.amazonaws.com/
snapshots/123456789012/my-layer-4aaa2fbb-ff77-4b0a-ad92-5b78a716a96a?
versionId=27iWyA73cCAYqyH...",
        "CodeSha256": "tv9jJO+rPbXUUXuRKi7CwHzKtLDkDRJLB3cC3Z/ouXo=",
        "CodeSize": 169
    },
    "LayerArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-layer",
    "LayerVersionArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-layer:1",
    "Description": "My layer",
    "CreatedDate": "2023-11-14T23:03:52.894+0000",
    "Version": 1,
    "CompatibleArchitectures": [
        "arm64",
        "x86_64"
     ],
    "LicenseInfo": "MIT",
    "CompatibleRuntimes": [
        "python3.10",
        "python3.11"
```

```
    ]
}
```

每次呼叫 `publish-layer-version` 時都會建立一個新版本的層。

刪除圖層版本

若要刪除圖層版本，請使用 DeleteLayerVersionAPI。例如，您可以使用 `delete-layer-version` CLI 命令並指定層的名稱和版本。

```
aws lambda delete-layer-version --layer-name my-layer --version-number 1
```

刪除層版本之後，您無法再設定 Lambda 函數以便使用它。不過，凡已使用該版本的任何函式均能繼續對其進行存取。此外，Lambda 永遠不會重複使用層名稱的版本編號。

為函數新增層

Lambda 層是含有補充程式碼或資料的 .zip 封存檔。層通常具備程式庫相依性、自訂執行期或組態檔案。

本節會說明如何將層新增至 Lambda 函數。若要進一步了解有關層的概念性資訊以及您可能會考慮使用的原因，請參閱 Lambda 層。

您必須先執行下列動作，才能設定 Lambda 函數以使用層：

- 封裝層內容
- 在 Lambda 中建立層
- 確保您具有在圖層版本上調GetLayerVersion用 API 的權限。對於 AWS 帳戶 中的函數，您必須在您的使用者政策中具備此許可。若要在其他帳號中使用圖層，其他帳戶的擁有者必須在資源型策略中授與您的帳戶許可。如需範例，請參閱 the section called “授予 Layer 對其他帳戶的存取”。

您最多可以將五個層新增至 Lambda 函數。函數和所有圖層的解壓縮大小總計不得超過解壓縮部署套件大小 250 MB 的配額。如需詳細資訊，請參閱 Lambda 配額。

您的函數可以繼續使用您已新增的任何層版本，即使該層版本已被刪除，或您存取層的許可被撤銷後也是如此。但是，您不能建立使用已刪除圖層版本的新函數。

> **Note**
>
> 確定您新增至函數的層與函數的執行期和指令集架構相容。

新增層至函數 (主控台)

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇要設定的函數。
3. 在 Layers (層) 下，選擇 Add a layer (新增層)
4. 在選擇層下方選擇層來源：
   a. 若是 AWS 層 或 自訂層 層來源，請從下拉式功能表中選擇層。在 Version (版本) 中，從下拉式選單中選擇層版本。
   b. 若是指定 ARN 層來源，請在文字方塊中輸入 ARN，然後選擇驗證。接著選擇新增。

新增層的順序即 Lambda 將層內容合併至執行環境的順序。您可以使用主控台來變更層合併順序。

若要更新函數的層合併順序 (主控台)

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇要設定的函數。
3. 在 Layers (層) 下方，選擇 Edit (編輯)
4. 選擇其中一個層。
5. 選擇 Merge earlier (先前合併) 或 Merge later (稍後合併) 來調整層的順序。
6. 選擇儲存。

層已設定版本控制。每個層版本的內容都是不可變的。層擁有者可發行新的層版本，以提供更新內容。您可以使用主控台來更新函數附加的層版本。

若要更新函數的層版本 (主控台)

1. 開啟 Lambda 主控台中的 層頁面。
2. 選擇您要更新版本的層。
3. 選擇使用此版本的函數標籤。
4. 選擇您要修改的函數，然後選擇編輯。
5. 在層版本中選擇要變更的層版本。
6. 選擇 Update functions (更新函數)。

無法跨 AWS 帳戶更新函數的層版本。

主題

從您的函數存取層內容

如果您的 Lambda 函數包含層，Lambda 會將層內容擷取到函數執行環境中的 `/opt` 目錄。Lambda 按函數列出的順序 (從低到高) 擷取層。Lambda 會合併名稱相同的資料夾。如果同一個檔案出現在多個圖層中，則函數會使用最後擷取的圖層中的版本。

每個 Lambda 執行期會將特定的 /opt 目錄資料夾新增至 PATH 變數。您的函數程式碼可存取層內容，無需指定路徑。如需 Lambda 執行環境中路徑設定的詳細資訊，請參閱 the section called “定義執行時間環境變數”。

請參閱 the section called “每個 Lambda 執行時間的層路徑” 以了解建立層時要加入程式庫的位置。

如果您使用的是 Node.js 或 Python 執行期，則可以使用 Lambda 主控台內建的程式碼編輯器。您應當可以將已新增為層的任何程式庫匯入目前的函數。

尋找圖層資訊

若要在帳戶中尋找與函數執行階段相容的圖層，請使用 ListLayersAPI。舉例來說，您可以使用下列 `list-layers` AWS Command Line Interface (CLI) 命令：

```
aws lambda list-layers --compatible-runtime python3.9
```

您應該會看到類似下列的輸出：

```
{
    "Layers": [
        {
            "LayerName": "my-layer",
            "LayerArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-layer",
            "LatestMatchingVersion": {
                "LayerVersionArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-
layer:2",
                "Version": 2,
                "Description": "My layer",
                "CreatedDate": "2023-11-15T00:37:46.592+0000",
                "CompatibleRuntimes": [
                    "python3.9",
                    "python3.10",
                    "python3.11",
                ]
            }
        }
    ]
}
```

若要在您的帳戶中列出所有層，請忽略 `--compatible-runtime` 選項。回應詳細資訊會顯示各個層的最新版本。

您也可以使用 ListLayerVersionsAPI 取得圖層的最新版本。舉例來說，您可以使用下列 `list-layer-versions` CLI 命令：

```
aws lambda list-layer-versions --layer-name my-layer
```

您應該會看到類似下列的輸出：

```
{
    "LayerVersions": [
        {
            "LayerVersionArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-
layer:2",
            "Version": 2,
            "Description": "My layer",
            "CreatedDate": "2023-11-15T00:37:46.592+0000",
            "CompatibleRuntimes": [
                "java11"
            ]
        },
        {
            "LayerVersionArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-
layer:1",
            "Version": 1,
            "Description": "My layer",
            "CreatedDate": "2023-11-15T00:27:46.592+0000",
            "CompatibleRuntimes": [
                "java11"
            ]
        }
    ]
}
```

AWS CloudFormation 與圖層一起使用

您可以使用 AWS CloudFormation 建立圖層，並將圖層與 Lambda 函數產生關聯。下列範例範本會建立名為 `my-lambda-layer` 的層，並使用 Layers 屬性將該層連接至 Lambda 函數。

```
---
Description: CloudFormation Template for Lambda Function with Lambda Layer
Resources:
  MyLambdaLayer:
    Type: AWS::Lambda::LayerVersion
    Properties:
      LayerName: my-lambda-layer
      Description: My Lambda Layer
      Content:
        S3Bucket: DOC-EXAMPLE-BUCKET
        S3Key: my-layer.zip
      CompatibleRuntimes:
        - python3.9
        - python3.10
        - python3.11

  MyLambdaFunction:
    Type: AWS::Lambda::Function
    Properties:
      FunctionName: my-lambda-function
      Runtime: python3.9
      Handler: index.handler
      Timeout: 10
      Policies:
        - AWSLambdaBasicExecutionRole
        - AWSLambda_ReadOnlyAccess
        - AWSXrayWriteOnlyAccess
      Layers:
        - !Ref MyLambdaLayer
```

AWS SAM 與圖層一起使用

您可以使用 AWS Serverless Application Model (AWS SAM) 在應用程式中自動建立圖層。`AWS::Serverless::LayerVersion` 資源類型會建立可從 Lambda 函數組態參考的圖層版本。

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Description: AWS SAM Template for Lambda Function with Lambda Layer

Resources:
  MyLambdaLayer:
    Type: AWS::Serverless::LayerVersion
    Properties:
      LayerName: my-lambda-layer
      Description: My Lambda Layer
      ContentUri: s3://DOC-EXAMPLE-BUCKET/my-layer.zip
      CompatibleRuntimes:
        - python3.9
        - python3.10
        - python3.11

  MyLambdaFunction:
    Type: AWS::Serverless::Function
    Properties:
      FunctionName: MyLambdaFunction
      Runtime: python3.9
      Handler: app.handler
      CodeUri: s3://DOC-EXAMPLE-BUCKET/my-function
      Layers:
        - !Ref MyLambdaLayer
```

使用 Lambda 擴充功能擴充功能擴充

您可以使用 Lambda 擴展功能來增強 Lambda 函數。例如，使用 Lambda 擴展將函數與您偏好的監控、可觀度、安全性和治理工具整合。您可以從 AWS Lambda 合作夥伴提供的廣泛工具集中選擇，也可以建立自己的 Lambda 擴展。

Lambda 支援外部和內部擴展。外部延伸項目會在執行環境中作為獨立的處理序執行，並在完全處理函式叫用之後繼續執行。由於延伸項目會以單獨的程序執行，因此您可以使用與函數不同的語言來撰寫。所有 Lambda 執行期 支援延伸。

內部延伸項目會執行作為執行階段程序的一部分。您的函式透過使用包裝函式指令碼或進行中的機制存取內部延伸項目，例如 `JAVA_TOOL_OPTIONS`。如需詳細資訊，請參閱 修改執行階段環境。

您可以使用 Lambda 主控台、() 或基礎架構即程式碼 AWS Command Line Interface (IaC AWS CLI) 服務和工具 (例如 AWS CloudFormation、 AWS Serverless Application Model (AWS SAM) 和 Terraform，將擴充功能新增至函數。

您需要依據延伸項目所耗用的執行時間付費 (以 1 毫秒為單位)。安裝自己的延伸項目不需花費任何費用。如需延伸項目的定價資訊，請參閱AWS Lambda 定價。如需合作夥伴延伸項目的定價資訊，請參閱合作夥伴的網站。如需官方合作夥伴擴充功能清單，請參閱 the section called “延伸合作夥伴”。

如需擴充功能的教學課程以及如何搭配 Lambda 函數來使用，請參閱 AWS Lambda 擴充功能研討會。

主題

- 執行環境
- 對效能和資源的影響
- 許可
- 設定 Lambda 延伸模組
- AWS Lambda 擴充功能合
- 使用 Lambda 擴充功能 API 建立擴充功能
- Lambda 遙測 API

執行環境

Lambda 會在執行環境中叫用您的函數，該環境可提供安全且隔離的執行時間環境。執行環境會管理執行函數所需的資源，並為函數的執行時間和延伸項目提供生命週期支援。

執行環境的生命週期包含下列階段：

- **Init**：在此階段中，Lambda 會使用設定的資源建立或解除凍結執行環境，下載函數程式碼和所有層，初始化所有擴展功能，初始化執行時間，然後執行函數的初始化程式碼 (主處理常式之外的程式碼)。此 Init 階段發生在第一次調用期間，或者在函數調用之前發生 (如果您已啟用佈建並行)。

  Init 階段分為三個子階段：Extension init、Runtime init 以及 Function init。這些子階段可確保所有擴展功能和執行時間在函數程式碼執行之前完成其設定任務。

  在 Lambda SnapStart 啟動的情況下，發佈函數版本時會發生 Init 階段。Lambda 會儲存初始化執行環境的記憶體和磁碟狀態快照、保留加密的快照，並快取以進行低延遲存取。如果您有 beforeCheckpoint 執行階段掛鉤，那麼程式碼會在 Init 階段結束時執行。
- **Restore**(SnapStart 僅限)：當您第一次叫用SnapStart函數且函數擴充時，Lambda 會從持續快照恢復新的執行環境，而不是從頭開始初始化函數。如果您有 afterRestore() 執行階段掛鉤，程式碼會在 Restore 階段結束時執行。您需支付 afterRestore() 執行階段掛鉤期間的費用。執行階段 (JVM) 必須載入，且 afterRestore() 執行階段掛鉤必須在逾時限制 (10 秒) 內完成。否則，你會得到一個 SnapStartTimeoutException. Restore 階段完成時，Lambda 會調用函數處理常式 (調用階段)。
- **Invoke**：在此階段中，Lambda 會調用函數處理常式。函數執行完成之後，Lambda 會準備處理另一個函數調用。
- **Shutdown**：如果 Lambda 函數在一段時間內未收到任何調用，就會觸發此階段。在 Shutdown 階段中，Lambda 會關閉執行時間、警示擴展功能以便它們完全停止，然後移除環境。Lambda 會傳送 Shutdown 事件到每個擴展功能，這會告訴擴展功能環境即將關閉。

在 Init 階段期間，Lambda 會將包含擴展功能的圖層擷取到執行環境中的 /opt 目錄。Lambda 在 /opt/extensions/ 目錄中尋找擴展功能，將每個檔案解譯為啟動擴展的可執行引導程序，並平行啟動所有擴展。

對效能和資源的影響

函式延伸項目的大小會計入部署套件的大小限制。針對 .zip 封存檔，函數和所有延伸項目解壓縮後的總大小不能超過解壓縮後 250 MB 的部署套件大小限制。

延伸項目可能會影響您的函式效能，因為它們會共用 CPU、記憶體和儲存體等函式資源。例如，如果延伸項目執行運算密集的作業，您可能會看到函數的執行持續時間增加。

在 Lambda 叫用函數之前，每個擴展必須完成其初始化。因此，耗用大量初始化時間的延伸可能會增加函式叫用的延遲。

若要測量延伸項目在函式執行後所花費的額外時間，您可以使用 `PostRuntimeExtensionsDuration` 函式指標。若要測量使用的記憶體的增加情況，您可以使用 `MaxMemoryUsed` 指標。若要了解特定延伸項目的影響，您可以並排執行不同版本的函式。

許可

延伸項目可存取與函式相同的資源。因為延伸項目是在與函式相同的環境中執行的，所以許可會在函式和延伸項目之間共用。

對於 .zip 檔案封存，您可以建立 AWS CloudFormation 範本，以簡化將相同擴充功能設定 AWS Identity and Access Management (包括 (IAM) 權限) 附加至多個函數的工作。

設定 Lambda 延伸模組

設定延伸 (.zip 檔案封存)

您可以將擴展功能作為 Lambda 層新增至函數。使用圖層可讓您在整個組織或整個 Lambda 開發人員社群中共用擴展功能。您可以將一或多個延伸項目新增至圖層。您可以為函式最多註冊 10 個延伸項目。

您可以使用與任何圖層相同的方法將延伸項目新增到您的函式中。如需詳細資訊，請參閱 Lambda 層。

將延伸項目新增到您的函式 (主控台)

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 如果尚未選取，請選擇 Code (程式碼) 標籤。
4. 在 Layers 下方，選擇 Edit (編輯)。
5. 在選擇圖層中，選擇指定 ARN。
6. 在指定 ARN中，輸入延伸圖層的 Amazon Resource Name (ARN)。
7. 選擇新增。

在容器映像中使用延伸項目

您可以將延伸項目新增至容器映像中。ENTRYPOINT 容器映像設定指定函數的主要程序。在 Dockerfile 中進行 ENTRYPOINT 設定，或設定為函數組態覆寫。

您可以在容器中執行多個程序。Lambda 會管理主程序的生命週期和任何額外程序。Lambda 會使用 Extensions API 來管理擴展生命週期。

範例：新增外部延伸項目

外部擴展會在不同於 Lambda 函數的程序中執行。Lambda 會在 `/opt/extensions/` 目錄中開始每個擴展的程序。Lambda 使用 Extensions API 來管理擴展生命週期。函數執行完成後，Lambda 會將 `Shutdown` 事件傳送至每個外部擴展。

Example 將外部延伸項目新增至 Python 基礎映像

```
FROM public.ecr.aws/lambda/python:3.11
```

```
# Copy and install the app
COPY /app /app
WORKDIR /app
RUN pip install -r requirements.txt

# Add an extension from the local directory into /opt
ADD my-extension.zip /opt
CMD python ./my-function.py
```

後續步驟

若要深入了解延伸項目，我們建議您使用下列資源：

- 如需基礎工作範例，請參閱 AWS 運算部落格上的建置 AWS Lambda 的延伸項目。
- 如需 AWS Lambda 合作夥伴提供的延伸項目相關資訊，請參閱 AWS 運算部落格上的 AWS Lambda 延伸項目簡介。
- 若要檢視可用的範例擴充功能和包裝函式指令碼，請參閱AWS LambdaAWS範例 GitHub 儲存區域上的

AWS Lambda 擴充功能合

AWS Lambda 已與多個第三方實體合作，提供與 Lambda 函數整合的擴充功能。下列清單詳細介紹了可供您隨時使用的第三方延伸。

- AppDynamics – 提供 Node.js 或 Python Lambda 函數的自動檢測，並提供關於函數效能的可視性和提醒。
- Check Point CloudGuard – 以延伸為基礎的執行時間解決方案，為無伺服器應用程式提供完整的生命週期安全性。
- Datadog – 透過使用指標、追蹤和日誌，提供對無伺服器應用程式的全面、即時可視性。
- Dynatrace – 提供追蹤和指標可視性，並充分利用 AI 在整個應用程式堆疊中進行自動錯誤偵測和根本原因分析。
- Elastic – 提供應用程式效能監控 (APM)，以使用相關聯的追蹤、指標和日誌識別並解決根本原因問題。
- Epsagon – 監聽調用事件，存放追蹤，並將其平行傳送至 Lambda 函數執行。
- Fastly – 保護您的 Lambda 函數免受可疑活動的影響，例如注入式攻擊、透過憑證填充的帳戶接管、惡意機器人和 API 濫用。
- HashiCorp Vault – 管理機密，並使其可供開發人員在函數程式碼中使用，而不讓函數保存庫知道。
- Honeycomb – 用於對應用程式堆疊偵錯的可觀察性工具。
- Lumigo – Profile Lambda 函數會調用並收集用於排解無伺服器和微型服務環境問題的指標。
- New Relic – 與 Lambda 功能一起執行，自動收集、增強遙測，並將其傳輸至 New Relic 的統一可觀察性平台。
- Sedai – 一個由 AI/ML 提供支援的自主雲端管理平台，可為雲端操作團隊提供持續最佳化，以最大限度地大規模節省雲端成本、提高效能和可用性。
- Sentry – 診斷、修復和最佳化 Lambda 函數的效能。
- Site24x7 – 實現 Lambda 環境的即時可觀察性
- Splunk – 收集高解析度、低延遲指標，以高效且有效率地監控 Lambda 函數。
- Sumo Logic – 提供對無伺服器應用程式運作狀態和效能的可視性。
- Thundra – 提供非同步遙測報告，如追蹤、指標和日誌。
- Salt 安全性 — 透過對不同執行階段的自動化設定和支援，簡化 Lambda 函數的 API 狀態控管和 API 安全性。

AWS 受管理擴充

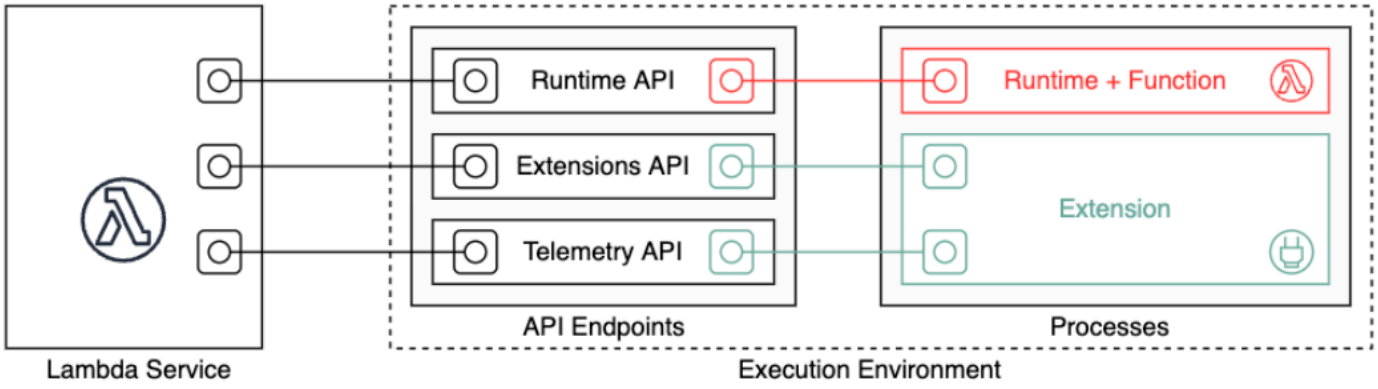
AWS 提供自己的受管理擴充功能，包括：

- AWS AppConfig— 使用功能標誌和動態資料來更新 Lambda 函數。您還可以使用此延伸來更新其他動態組態，如維運調節和調校。
- Amazon CodeGuru Profiler — 精確定位應用程式最昂貴的程式碼行，並提供改善程式碼的建議，藉此改善應用程式效能並降低成本。
- CloudWatch Lambda 深入解析 — 透過自動化儀表板監控、疑難排解及最佳化 Lambda 函數的效能。
- AWS 適用於 ADOT 的發行版 — 啟用將追蹤資料傳送至 AWS 監控服務 OpenTelemetry (例如 AWS X-Ray) 的功能，以及傳送至支援蜂巢狀態和 Lightstep OpenTelemetry 等支援的目的地。
- AWS 參數和機密 — 可讓客戶安全地從AWS Systems Manager 參數存放區擷取參數和機密AWS Secrets Manager。

如需其他延伸範例和示範專案，請參閱 AWS Lambda 延伸。

使用 Lambda 擴充功能 API 建立擴充功能

Lambda 函數作者使用延伸項目將 Lambda 與其偏好的監控、可觀度、安全性和治理工具整合在一起。函數作者可以使用合作AWS 夥伴和開放原始碼專案的擴充功能。 AWS如需使用擴充功能的詳細資訊，請參閱 AWS Compute 部落格上的AWS Lambda 擴充功能簡介 本節說明如何使用 Lambda 延伸 API、Lambda 執行環境生命週期以及 Lambda 延伸 API 參考。



作為延伸項目的作者，您可以使用 Lambda Extensions API 以便深入整合到 Lambda 執行環境中。您的延伸項目可以註冊函數和執行環境生命週期事件。為了回應這些事件，您可以啟動新程序、執行邏輯，以及控制並參與 Lambda 生命週期的所有階段：初始化、調用和關閉。此外，您可以使用 Runtime Logs API 來接收日誌串流。

延伸項目會在執行環境中作為獨立的處理序執行，並在完全處理函數調用之後繼續執行。由於延伸項目會以程序執行，因此您可以使用與函數不同的語言來撰寫。我們建議您使用編譯語言實作延伸項目。在這種情況下，延伸項目是獨立的二進位檔案，會與支援的執行時間相容。所有 Lambda 執行期 支援延伸。如果您使用非編譯語言，請確定您在延伸項目中包含相容的執行階段。

Lambda 也支援內部延伸項目。內部延伸項目會作為獨立執行緒在執行時間程序中執行。執行時間會啟動並停止內部延伸項目。與 Lambda 環境整合的另一種方法是使用特定語言的環境變數和包裝函數指令碼。您可使用這些來設定執行時間環境，並修改執行時間程序的啟動行為。

您可以使用兩種方法將延伸項目新增至函數。對於部署為 .zip 封存檔案的函數，您可以將延伸項目部署為 layer。對於定義為容器映像的函數，可以將延伸項目新增至容器映像。

> **Note**
>
> 如需擴充功能和包裝函式指令碼範例，請參閱 AWS 範例 GitHub 儲存庫上的AWS Lambda 擴充

主題

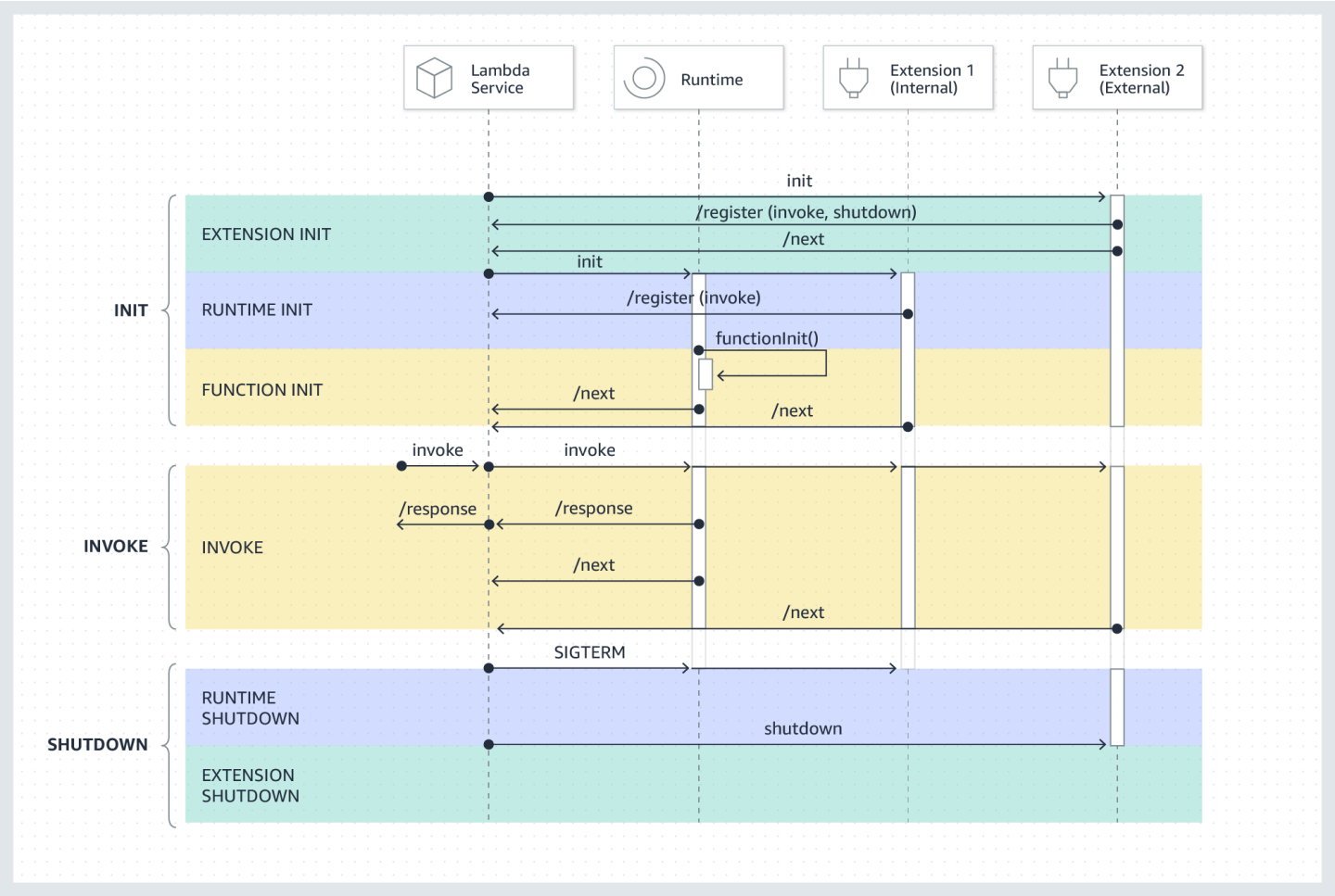
Lambda 執行環境生命週期

執行環境的生命週期包含下列階段：

- **Init**：在此階段中，Lambda 會使用設定的資源建立或解除凍結執行環境，下載函數程式碼和所有層，初始化所有擴展功能，初始化執行時間，然後執行函數的初始化程式碼 (主處理常式之外的程式碼)。此 `Init` 階段發生在第一次調用期間，或者在函數調用之前發生 (如果您已啟用佈建並行)。

  `Init` 階段分為三個子階段：`Extension init`、`Runtime init` 以及 `Function init`。這些子階段可確保所有擴展功能和執行時間在函數程式碼執行之前完成其設定任務。
- **Invoke**：在此階段中，Lambda 會調用函數處理常式。函數執行完成之後，Lambda 會準備處理另一個函數調用。
- **Shutdown**：如果 Lambda 函數在一段時間內未收到任何調用，就會觸發此階段。在 `Shutdown` 階段中，Lambda 會關閉執行時間、警示擴展功能以便它們完全停止，然後移除環境。Lambda 會傳送 `Shutdown` 事件到每個擴展功能，這會告訴擴展功能環境即將關閉。

每個階段都會從 Lambda 到執行時間和所有已註冊延伸項目的事件開始。執行時間和每個已註冊延伸項目都會透過傳送 `Next` API 請求來表示已完成。當每個處理已完成且沒有擱置的事件時，Lambda 凍結執行環境。



主題

初始化階段

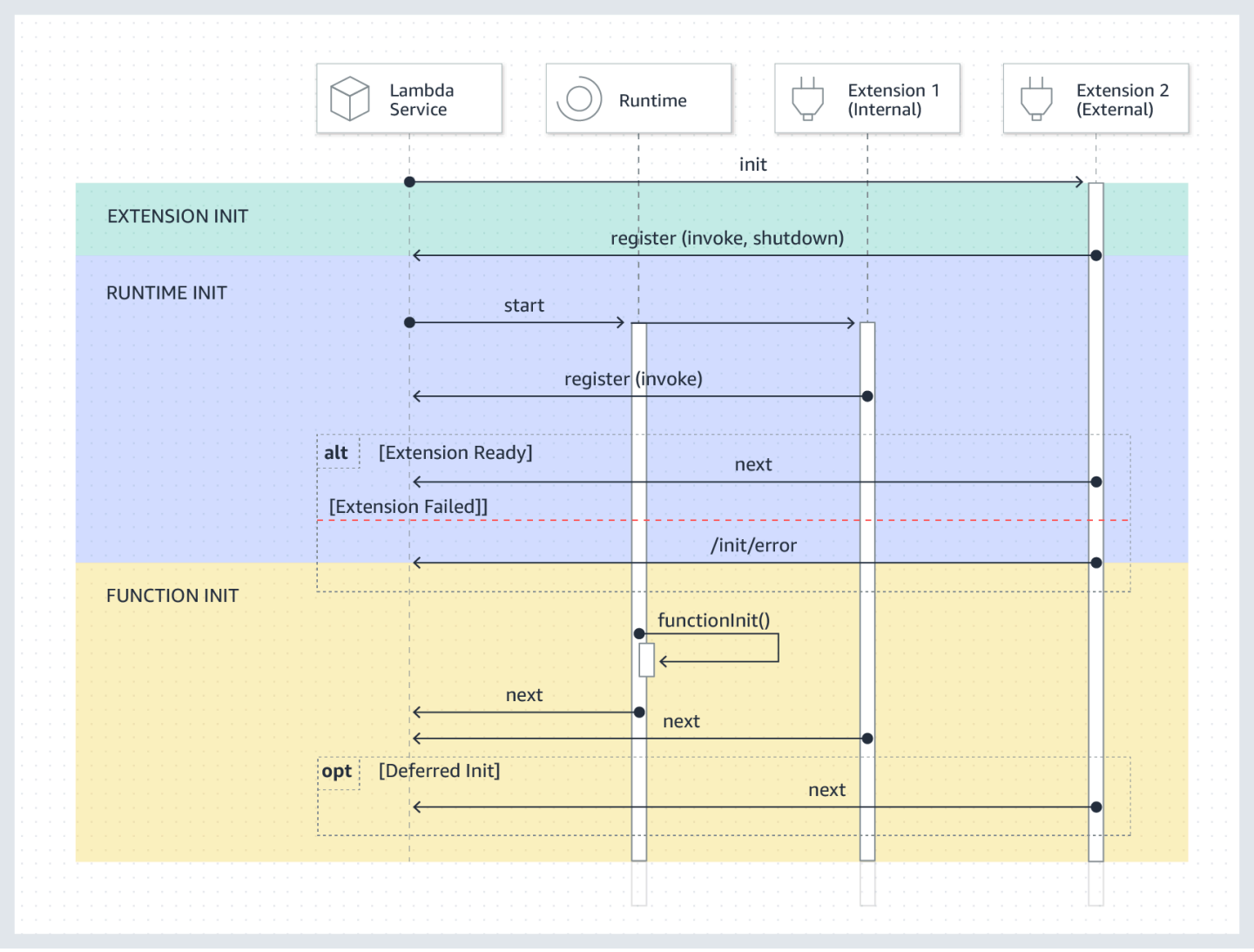
在 `Extension init` 階段中，每個延伸項目都需要向 Lambda 註冊才能接收事件。Lambda 會使用延伸項目的完整檔案名稱來驗證延伸項目是否已完成啟動程序。因此，每個 `Register` API 呼叫必須包含延伸項目完整檔案名稱的 `Lambda-Extension-Name` 標頭。

您可以為函式最多註冊 10 個延伸項目。此限制會透過 `Register` API 呼叫強制執行。

在每個延伸項目註冊後，Lambda 會啟動 `Runtime init` 階段。執行時間程序會呼叫 `functionInit` 以啟動 `Function init` 階段。

`Init` 階段會在執行階段之後完成，每個已註冊延伸項目都會透過傳送 `Next` API 請求來表示已完成。

> **Note**
>
> 延伸項目可以在 `Init` 階段的任何時候完成初始化。



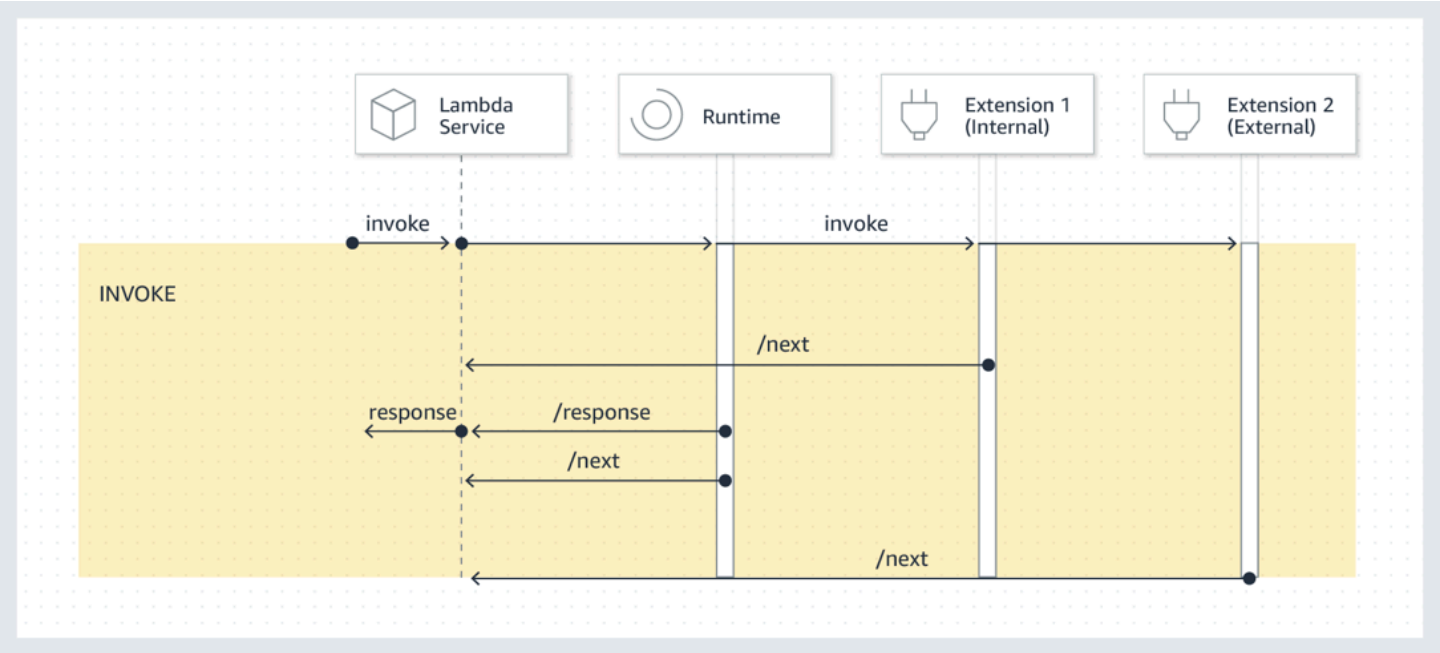
調用階段

當調用 Lambda 函數來回應 Next API 請求時，Lambda 會將 Invoke 事件傳送至執行時間，並傳送至針對該 Invoke 事件已註冊的每個延伸項目。

在調用期間，外部延伸項目會與函式平行執行。函式完成後，它們也會繼續執行。這可讓您擷取診斷資訊，或將日誌、指標和追蹤傳送至您選擇的位置。

從執行時間中收到函數回應之後，即使延伸項目仍在執行中，Lambda 也會將回應傳回給用戶端。

Invoke 階段會在執行階段後結束，所有延伸項目訊號都透過傳送 Next API 請求完成。



每個承載：傳送到執行時間 (和 Lambda 函數) 的事件載有整個請求、標頭 (例如 RequestId) 和承載。傳送至每個延伸項目的事件會包含描述事件內容的中繼資料。此生命週期事件包括事件的類型、函數逾時的時間 (deadlineMs) requestId、調用函數的 Amazon Resource Name (ARN) 以及追蹤標頭。

想要存取函式事件主體的延伸項目，您可以使用與延伸項目通訊的執行階段 SDK。函式開發人員會使用執行階段內 SDK，在調用函式時將承載傳送至延伸項目。

以下是承載範例：

```
{
    "eventType": "INVOKE",
    "deadlineMs": 676051,
    "requestId": "3da1f2dc-3222-475e-9205-e2e6c6318895",
```

```
    "invokedFunctionArn": "arn:aws:lambda:us-
east-1:123456789012:function:ExtensionTest",
    "tracing": {
        "type": "X-Amzn-Trace-Id",
        "value":
 "Root=1-5f35ae12-0c0fec141ab77a00bc047aa2;Parent=2be948a625588e32;Sampled=1"
    }
 }
```

持續時間限制：該函數的逾時設置限制了整個 Invoke 階段的持續時間。例如，如果您將函式逾時設定為 360 秒，則函式和所有延伸項目都需要在 360 秒內完成。請注意，沒有獨立的調用後階段。持續時間是執行階段和所有擴充功能呼叫完成所需的總時間，直到函數和所有擴充功能完成執行之後才會計算。

效能影響和延伸項目負擔：延伸項目可能會影響函式效能。身為延伸項目的作者，您可以控制延伸項目的效能影響。例如，如果延伸項目執行運算密集型操作，函式的持續時間便會延長，因為延伸項目和函式程式碼會共用相同的 CPU 資源。此外，如果您的延伸項目在函式調用完成後執行大量作業，函式的持續時間就會增加，因為 Invoke 階段會持續進行，直到所有延伸項目表示它們已完成為止。

> **Note**
>
> Lambda 會根據函數的記憶體設定來分配 CPU 功率。您可能會在較低的記憶體設定下看到執行和初始化持續時間增加，因為函數和延伸項目程序正在競爭相同的 CPU 資源。若要減少執行和初始化持續時間，請嘗試增加記憶體設定。

為了協助識別延伸在 Invoke 階段上造成的效能衝擊，Lambda 會輸出 PostRuntimeExtensionsDuration 指標。此指標會測量執行階段 Next API 請求與上次延伸 Next API 請求之間所花費的累計時間。若要測量使用的記憶體增加的情況，請使用 MaxMemoryUsed 指標。如需函式指標的詳細資訊，請參閱使用 Lambda 函數指標。

函式開發人員可以並排執行不同版本的函式，以了解特定延伸項目的影響。我們建議延伸項目作者發佈預期的資源消耗，以便函式開發人員更容易選擇合適的延伸項目。

關閉階段

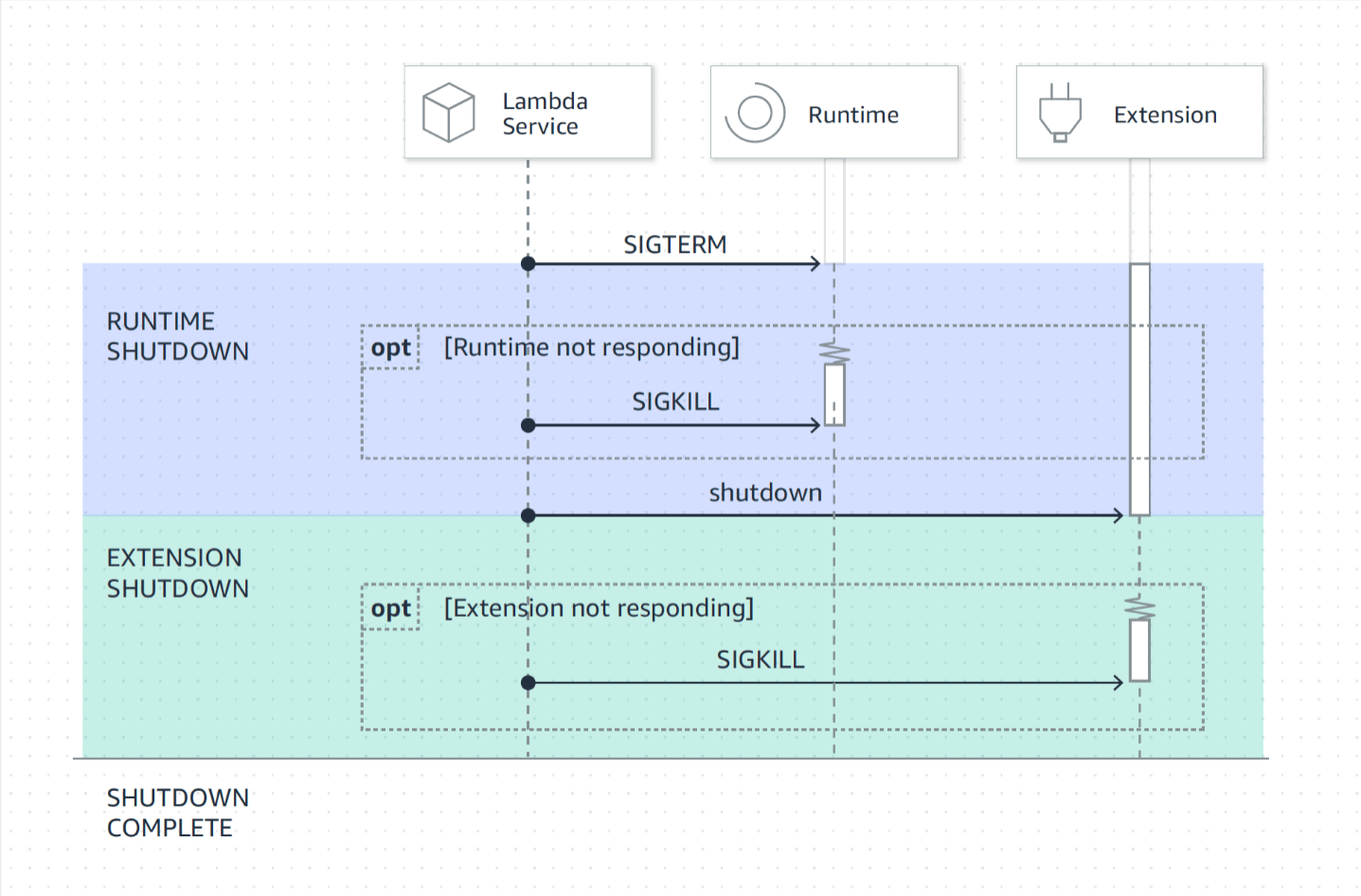
當 Lambda 即將關閉執行時間，它會將 Shutdown 事件傳送至每個已註冊外部延伸。延伸項目可以使用此時間進行最終清理工作。系統會傳送 Shutdown 事件以回應 Next API 請求。

持續時間限制：Shutdown 階段的最長持續時間視已註冊延伸項目的組態而定：

- 0 毫秒 - 沒有註冊延伸的函數
- 500 毫秒 - 具有已註冊內部延伸項目的函數
- 2,000 毫秒 - 具有一個或多個已註冊外部延伸項目的函數

對於具有外部延伸項目的函數，Lambda 會保留最多 300 毫秒 (具有內部延伸項目的執行時間 500 毫秒)，以便執行時間程序執行正常關閉。Lambda 會分配 2000 毫秒限制的其餘部分，以關閉外部延伸項目。

如果執行時間或延伸項目未在限制內回應 Shutdown 事件，Lambda 會使用 SIGKILL 訊號結束程序。



事件承載：Shutdown 事件會包含關閉的原因和剩餘時間 (以毫秒為單位)。

shutdownReason 包含以下值：

- SPINDOWN - 正常關閉
- TIMEOUT - 持續時間限制逾時
- FAILURE - 錯誤條件，例如 out-of-memory 事件

```
{
  "eventType": "SHUTDOWN",
  "shutdownReason": "reason for shutdown",
  "deadlineMs": "the time and date that the function times out in Unix time
 milliseconds"
}
```

許可與組態

延伸項目會在與 Lambda 函數相同的執行環境中執行。延伸項目也會與函式共用資源，例如 CPU、記憶體和 `/tmp` 儲存磁碟。此外，擴充功能使用與函數相同的 AWS Identity and Access Management (IAM) 角色和安全性內容。

檔案系統和網路存取許可：延伸項目在與函式執行階段相同的檔案系統和網路名稱命名空間中執行。這表示延伸項目必須與相關的作業系統相容。如果延伸項目需要任何其他傳出網路流量規則，您必須將這些規則套用至函數組態。

> **Note**
>
> 由於函式程式碼目錄是唯讀的目錄，因此延伸項目無法修改函式程式碼。

環境變數：延伸項目可以存取函式的環境變數，但下列特定於執行階段程序的變數除外：

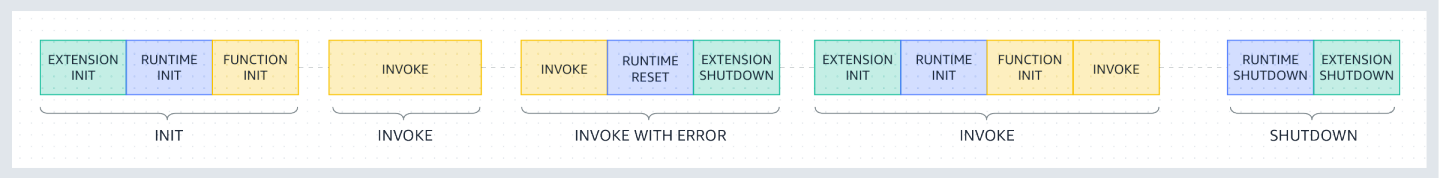
- `AWS_EXECUTION_ENV`
- `AWS_LAMBDA_LOG_GROUP_NAME`
- `AWS_LAMBDA_LOG_STREAM_NAME`
- `AWS_XRAY_CONTEXT_MISSING`
- `AWS_XRAY_DAEMON_ADDRESS`
- `LAMBDA_RUNTIME_DIR`
- `LAMBDA_TASK_ROOT`
- `_AWS_XRAY_DAEMON_ADDRESS`
- `_AWS_XRAY_DAEMON_PORT`
- `_HANDLER`

失敗處理

初始化失敗：如果延伸失敗，Lambda 會重新啟動執行環境以強制執行一致的行為，並促進延伸項目快速失敗。此外，對於某些客戶，延伸項目必須符合任務關鍵需求，例如日誌、安全性、治理和遙測收集。

調用失敗 (例如記憶體不足、函式逾時)：因為延伸項目程序與執行時間共享資源，所以記憶體耗盡會對其產生影響。當執行階段失敗時，所有延伸項目和執行階段本身都會參與此 Shutdown 階段。此外，執行時間會自動重新啟動，這可能是為了要做為目前調用的一部分，或是透過延遲的重新初始化機制重新啟動。

如果在 Invoke 期間發生失敗 (例如函數逾時或執行時間錯誤)，則 Lambda 服務會執行重設。重設的行為會與 Shutdown 事件一樣。首先，Lambda 關閉執行時間，然後將 Shutdown 事件傳送給每個已註冊外部延伸項目。事件會包括關閉的原因。如果將此環境用於新的調用，延伸項目和執行階段會重新初始化為下次調用的一部分。



如需上圖更為詳細的說明，請參閱 調用階段期間出現的故障。

擴充功能記錄：Lambda 會將擴充功能的記錄輸出傳送至 CloudWatch 記錄。Lambda 也會在 Init 期間為每個延伸項目產生額外的日誌事件。日誌事件會記錄成功時的名稱和註冊偏好設定 (事件、設定)，或失敗時的失敗原因。

故障排除延伸項目

- 如果 Register 請求失敗，請確定 Register API 呼叫中的 Lambda-Extension-Name 標頭包含延伸項目的完整檔案名稱。
- 如果內部延伸項目的 Register 請求失敗，請確定請求不會註冊 Shutdown 事件。

Extensions API 參考

對於 Extensions API 版本 2020-01-01 的 OpenAPI 規範可以在這裡找到：extensions-api.zip

您可以從 AWS_LAMBDA_RUNTIME_API 環境變數擷取 API 端點的值。若要傳送 Register 請求，請在每個 API 路徑之前使用前綴 2020-01-01/。例如：

```
http://${AWS_LAMBDA_RUNTIME_API}/2020-01-01/extension/register
```

API 方法

- 登錄
- 下一頁
- 初次化錯誤
- 結束錯誤

登錄

在 Extension init 階段中，所有延伸項目都需要向 Lambda 註冊才能接收事件。Lambda 會使用延伸項目的完整檔案名稱來驗證延伸項目是否已完成啟動程序。因此，每個 Register API 呼叫必須包含延伸項目完整檔案名稱的 Lambda-Extension-Name 標頭。

執行階段程序會啟動和停止內部延伸項目，因此不允許它們註冊 Shutdown 事件。

路徑 – /extension/register

方法 – POST

請求標頭

- Lambda-Extension-Name - 延伸項目的完整檔案名稱。必要：是。類型：字串
- Lambda-Extension-Accept-Feature - 用來在註冊期間指定選用的延伸項目特色。必要：否。類型：逗號分隔的字串。可使用此設定來指定的特色：
  - accountId - 如果指定此項，延伸項目註冊回應將包含與待註冊延伸項目之 Lambda 函數相關聯的帳戶 ID。

請求內文參數

- events - 要註冊的事件陣列。必要：否。類型：字串陣列。有效字串：INVOKE、SHUTDOWN。

回應標頭

- Lambda-Extension-Identifier - 產生所有後續請求所需的唯一代理程式識別符 (UUID 字串)。

回應代碼

- 200 - 回應主體包含的函數名稱、函數版本和處理常式名稱。
- 400 - 錯誤請求
- 403 - 禁止
- 500 - 容器錯誤。不可復原的狀態。延伸項目應會立即結束。

Example 範例請求主體

```
{
    'events': [ 'INVOKE', 'SHUTDOWN']
}
```

Example 範例回應主題

```
{
    "functionName": "helloWorld",
    "functionVersion": "$LATEST",
    "handler": "lambda_function.lambda_handler"
}
```

Example 具有選用 accountId 特色的回應內文範例

```
{
    "functionName": "helloWorld",
    "functionVersion": "$LATEST",
    "handler": "lambda_function.lambda_handler",
    "accountId": "123456789012"
}
```

下一頁

延伸項目會傳送 Next API 請求以接收下一個事件，可以是 Invoke 事件或 Shutdown 事件。回應主體包含承載，這是包含事件資料的 JSON 文件。

延伸項目會傳送 Next API 請求，以表示它已準備好接收新事件。這是一個封鎖調用。

請勿在 GET 調用上設定逾時，因為延伸項目可以暫停一段時間，直到有事件傳回為止。

路徑 – /extension/event/next

方法 – GET

請求標頭

- Lambda-Extension-Identifier - 延伸項目的唯一識別符 (UUID 字串)。必要：是。類型：UUID 字串。

回應標頭

- Lambda-Extension-Event-Identifier - 延伸項目的唯一識別符 (UUID 字串)。

回應代碼

- 200 - 回應包含下個事件 (EventInvoke 或 EventShutdown) 的相關資訊。
- 403 - 禁止
- 500 - 容器錯誤。不可復原的狀態。延伸項目應會立即結束。

初次化錯誤

延伸項目會使用此方法向 Lambda 報告初始化錯誤。當延伸項目註冊後無法初始化時會調用它。Lambda 收到錯誤後，進一步的 API 呼叫沒有成功。延伸項目應該在收到 Lambda 的回應之後退出。

路徑 – /extension/init/error

方法 – POST

請求標頭

- Lambda-Extension-Identifier - 延伸項目的唯一識別符。必要：是。類型：UUID 字串。
- Lambda-Extension-Function-Error-Type - 擴展遇到的錯誤類型。必要：是。此標頭包含一個字串值。Lambda 可接受任何字串，但我們建議使用格式 <category.reason>。例如：
  - 副檔名。 NoSuch處理器
  - 找到副檔名 .API KeyNot
  - 副檔名。 ConfigInvalid
  - 副檔名。 UnknownReason

請求內文參數

- ErrorRequest - 關於錯誤的資訊。必要：否。

此欄位是具有下列結構的 JSON 物件：

```
{
      errorMessage: string (text description of the error),
      errorType: string,
      stackTrace: array of strings
}
```

請注意，Lambda 接受 errorType 的任何值。

下列範例顯示 Lambda 函數錯誤訊息，其中函數無法剖析調用中提供的事件資料。

Example 函數錯誤

```
{
      "errorMessage" : "Error parsing event data.",
      "errorType" : "InvalidEventDataException",
      "stackTrace": [ ]
}
```

回應代碼

- 202 - 已接受
- 400 - 錯誤請求
- 403 - 禁止
- 500 - 容器錯誤。不可復原的狀態。延伸項目應會立即結束。

結束錯誤

延伸項目會使用此方法，在結束前向 Lambda 報告錯誤。當您遇到意外失敗時呼叫它。Lambda 收到錯誤後，進一步的 API 呼叫沒有成功。延伸項目應該在收到 Lambda 的回應之後退出。

路徑 – /extension/exit/error

方法 – POST

請求標頭

- `Lambda-Extension-Identifier` - 延伸項目的唯一識別符。必要：是。類型：UUID 字串。
- `Lambda-Extension-Function-Error-Type` - 擴展遇到的錯誤類型。必要：是。此標頭包含一個字串值。Lambda 可接受任何字串，但我們建議使用格式 <category.reason>。例如：
  - 副檔名。 NoSuch處理器
  - 找到副檔名 .API KeyNot
  - 副檔名。 ConfigInvalid
  - 副檔名。 UnknownReason

請求內文參數

- `ErrorRequest` - 關於錯誤的資訊。必要：否。

此欄位是具有下列結構的 JSON 物件：

```
{
      errorMessage: string (text description of the error),
      errorType: string,
      stackTrace: array of strings
}
```

請注意，Lambda 接受 `errorType` 的任何值。

下列範例顯示 Lambda 函數錯誤訊息，其中函數無法剖析調用中提供的事件資料。

Example 函數錯誤

```
{
      "errorMessage" : "Error parsing event data.",
      "errorType" : "InvalidEventDataException",
      "stackTrace": [ ]
}
```

回應代碼

- 202 - 已接受
- 400 - 錯誤請求

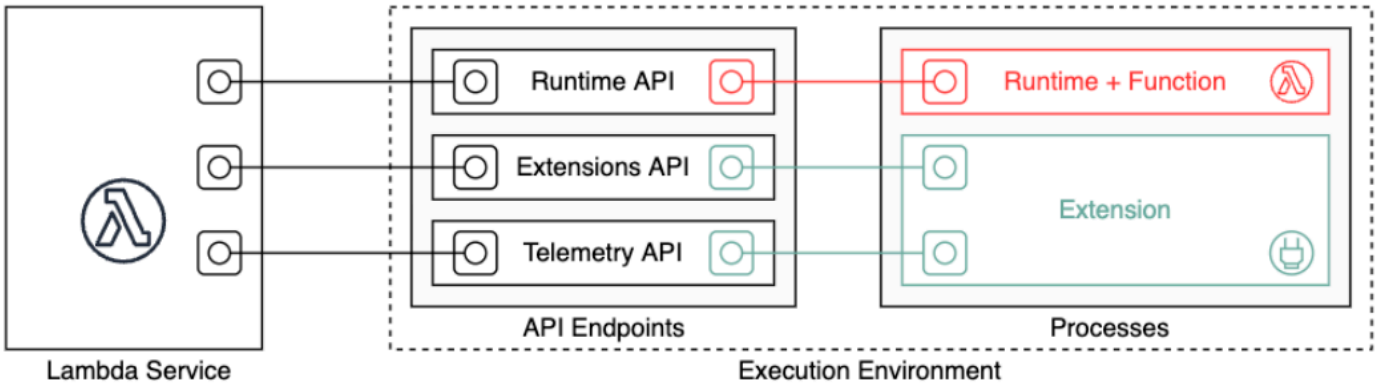
- 403 - 禁止
- 500 - 容器錯誤。不可復原的狀態。延伸項目應會立即結束。

Lambda 遙測 API

遙測 API 可讓您的延伸項目直接從 Lambda 接收遙測資料。在函數初始化和調用期間，Lambda 會自動擷取遙測資料，包括日誌、平台指標和平台追蹤。遙測 API 使延伸項目可以近乎即時地從 Lambda 直接存取這些遙測資料。

在 Lambda 執行環境中，您可以讓 Lambda 延伸項目訂閱遙測串流。訂閱後，Lambda 會自動將所有遙測資料傳送至您的延伸項目。然後，您便能靈活處理、篩選和傳送資料到偏好目的地，例如 Amazon Simple Storage Service (Amazon S3) 儲存貯體或第三方可觀測性工具提供者。

下圖顯示延伸 API 和遙測 API 如何從執行環境中將延伸項目連結至 Lambda。另外，執行期 API 也會將執行期和函數連接至 Lambda。



> **Important**
>
> Lambda 遙測 API 優先於 Lambda 日誌 API。雖然日誌 API 仍然正常運作，但我們建議您未來只使用遙測 API。您可以使用遙測 API 或日誌 API 訂閱遙測串流的延伸項目。使用其中一個 API 進行訂閱後，若嘗試使用其他 API 進行任何訂閱，都會傳回錯誤。

延伸項目可使用遙測 API 來訂閱三個不同的遙測串流：

- 平台遙測 – 日誌、指標和追蹤，描述與執行環境執行階段生命週期、延伸項目生命週期和函數調用相關的事件和錯誤。
- 函數日誌 – Lambda 函數程式碼產生的自訂日誌。
- 延伸項目日誌 - Lambda 延伸項目程式碼產生的自訂日誌。

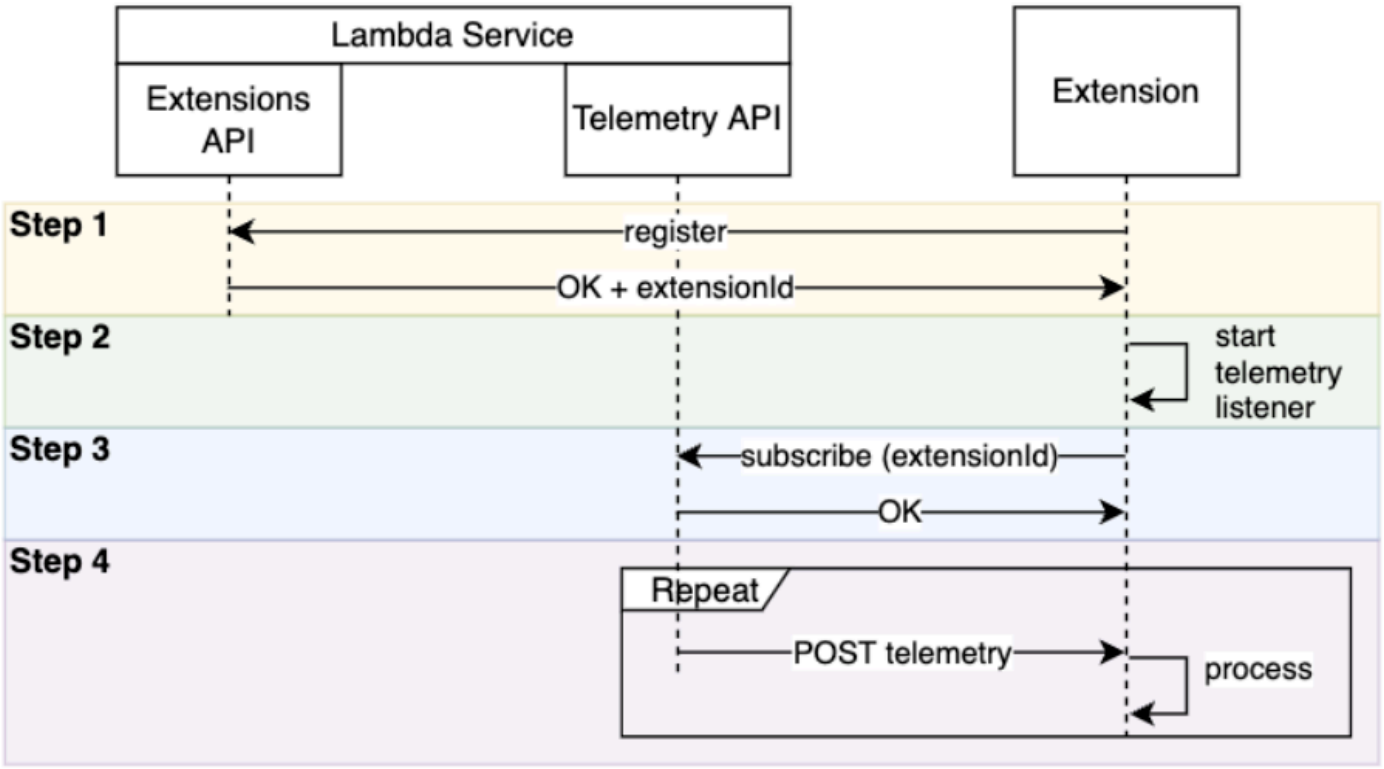
> **Note**
>
> Lambda 會將日誌和指標傳送至 X-Ray (如果您已啟動追蹤)，即使有擴充功能訂閱遙測串流也一樣。 CloudWatch

章節

使用遙測 API 建立延伸項目

Lambda 延伸項目會在執行環境中做為獨立的程序執行。延伸項目可以在函數調用完成後繼續執行。由於延伸項目為獨立的處理序，因此您可以使用與函數程式碼不同的語言來撰寫。我們建議您使用 Golang 或 Rust 等編譯語言編寫延伸項目。在這種情況下，延伸項目是獨立的二進位檔案，且與任何支援的執行階段相容。

下圖說明建立延伸項目的四步驟程序，讓延伸項目使用遙測 API 接收和處理遙測資料。



以下是各步驟的詳細說明：

1. 使用 the section called “Extensions API” 註冊延伸項目。這會為您提供 `Lambda-Extension-Identifier`，接著需要在下列步驟中使用。如需有關如何註冊延伸項目的詳細資訊，請參閱：the section called “註冊延伸項目”。
2. 建立遙測接聽程式。這可以是基本的 HTTP 或 TCP 伺服器。Lambda 使用遙測接聽程式的 URI 來將遙測資料傳送至延伸項目。如需詳細資訊，請參閱 the section called “建立遙測接聽程式”。
3. 使用遙測 API 中的訂閱 API，為您的延伸項目訂閱需要的遙測串流。在此步驟中，您需要遙測接聽程式的 URI。如需詳細資訊，請參閱 the section called “將訂閱請求傳送至遙測 API”。
4. 透過遙測接聽程式從 Lambda 取得遙測資料。您可以對這些資料執行任何自訂處理，例如將資料分派到 Amazon S3 或外部可檢視性服務。

> **Note**
>
> Lambda 函數的執行環境可以在其生命週期中多次啟動和停止。一般來說，延伸項目程式碼會在函數調用期間執行，並且在關閉階段執行最多 2 秒。我們建議在遙測傳送到您的接聽程式時

進行批次處理。然後，使用 Invoke 和 Shutdown 生命週期事件將每個批次傳送到所需的目的地。

註冊延伸項目

您必須先註冊 Lambda 延伸項目，才能訂閱遙測資料。註冊會在延伸功能初始化階段進行。下列範例顯示註冊延伸項目的 HTTP 請求。

```
POST http://${AWS_LAMBDA_RUNTIME_API}/2020-01-01/extension/register
 Lambda-Extension-Name: lambda_extension_name
{
    'events': [ 'INVOKE', 'SHUTDOWN']
}
```

如果請求成功，訂閱者會收到 HTTP 200 成功回應。回應標頭包含 Lambda-Extension-Identifier。回應內文包含函數的其他屬性。

```
HTTP/1.1 200 OK
Lambda-Extension-Identifier: a1b2c3d4-5678-90ab-cdef-EXAMPLE11111
{
    "functionName": "lambda_function",
    "functionVersion": "$LATEST",
    "handler": "lambda_handler",
    "accountId": "123456789012"
}
```

如需更多資訊，請參閱 the section called “Extensions API 參考”。

建立遙測接聽程式

Lambda 延伸項目必須具有可處理遙測 API 所傳入請求的接聽程式。下列程式碼顯示以 Golang 實作的遙測接聽程式實作範例：

```
// Starts the server in a goroutine where the log events will be sent
func (s *TelemetryApiListener) Start() (string, error) {
 address := listenOnAddress()
 l.Info("[listener:Start] Starting on address", address)
 s.httpServer = &http.Server{Addr: address}
 http.HandleFunc("/", s.http_handler)
 go func() {
```

```
  err := s.httpServer.ListenAndServe()
  if err != http.ErrServerClosed {
   l.Error("[listener:goroutine] Unexpected stop on Http Server:", err)
   s.Shutdown()
  } else {
   l.Info("[listener:goroutine] Http Server closed:", err)
  }
 }()
 return fmt.Sprintf("http://%s/", address), nil
}

// http_handler handles the requests coming from the Telemetry API.
// Everytime Telemetry API sends log events, this function will read them from the
 response body
// and put into a synchronous queue to be dispatched later.
// Logging or printing besides the error cases below is not recommended if you have
 subscribed to
// receive extension logs. Otherwise, logging here will cause Telemetry API to send new
 logs for
// the printed lines which may create an infinite loop.
func (s *TelemetryApiListener) http_handler(w http.ResponseWriter, r *http.Request) {
 body, err := ioutil.ReadAll(r.Body)
 if err != nil {
  l.Error("[listener:http_handler] Error reading body:", err)
  return
 }

 // Parse and put the log messages into the queue
 var slice []interface{}
 _ = json.Unmarshal(body, &slice)

 for _, el := range slice {
  s.LogEventsQueue.Put(el)
 }

 l.Info("[listener:http_handler] logEvents received:", len(slice), " LogEventsQueue
 length:", s.LogEventsQueue.Len())
 slice = nil
}
```

指定目的地通訊協定

使用遙測 API 訂閱以接收遙測資料時，除了目的地 URI 之外，您還可以指定目的地通訊協定：

```
{
    "destination": {
        "protocol": "HTTP",
        "URI": "http://sandbox.localdomain:8080"
    }
}
```

Lambda 接受兩種通訊協定用於接收遙測資料：

- HTTP (建議) - Lambda 會以 JSON 格式的記錄陣列將遙測資料傳遞至本機 HTTP 端點 (`http://sandbox.localdomain:${PORT}/${PATH}`)。$PATH 為選用參數。Lambda 僅支援 HTTP，不支援 HTTPS。Lambda 會透過 POST 請求傳遞遙測資料。
- TCP - Lambda 使用以換行分隔的 JSON (NDJSON) 格式將遙測資料傳遞至 TCP 連接埠。

> **Note**
>
> 強烈建議使用 HTTP，而不是使用 TCP。若使用 TCP，Lambda 平台無法確認何時將遙測資料傳遞至應用程式層。因此，如果您的延伸項目損毀，遙測資料可能會遺失。HTTP 沒有此限制。

訂閱以接收遙測資料之前，需先建立本機 HTTP 接聽程式或 TCP 連接埠。在設定期間，請注意下列事項：

- Lambda 只會將遙測資料傳送至執行環境內的目的地。
- 如果沒有接聽程式，或者如果 POST 請求遇到錯誤，則 Lambda 會重新嘗試傳送遙測資料 (使用回詢)。如果遙測接聽程式損毀，會在 Lambda 重新啟動執行環境之後繼續接收遙測資料。
- Lambda 保留連接埠 9001。沒有其他連接埠編號限制或建議。

設定記憶體使用量和緩衝

隨著訂閱用戶數量增加，執行環境的記憶體使用量會線性增加。訂閱會耗用記憶體資源，因為每個訂閱都會開啟新的記憶體緩衝區來存放遙測資料。緩衝區記憶體用量會計入執行環境中的整體記憶體耗用量。

透過遙測 API 來訂閱以接收遙測資料時，您可以選擇先緩衝遙測資料再批次傳遞給訂閱用戶。若要最佳化記憶體用量，您可以指定緩衝組態：

```
{
    "buffering": {
        "maxBytes": 256*1024,
        "maxItems": 1000,
        "timeoutMs": 100
    }
}
```

緩衝組態設定

參數	描述	預設值和限制
maxBytes	記憶體中要緩衝的遙測資料量上限 (位元組)。	預設：262,144 下限：262,144 上限：1,048,576
maxItems	記憶體中要緩衝的事件數目上限。	預設：10,000 下限：1,000 上限：10,000
timeoutMs	緩衝一個批次的時間上限 (毫秒)。	預設：1,000 下限：25 上限：30,000

當設定緩衝時，請記住以下幾點：

- 如果有任何輸入串流關閉，則 Lambda 會排清日誌。例如，如果執行期損毀，就可能會發生這種情況。
- 每個訂閱用戶可以在訂閱請求中自訂其緩衝組態。
- 決定讀取資料的緩衝區大小時，請預期接收的有效負載大小為 2 * maxBytes + metadataBytes (其中 maxBytes 是緩衝設定的元件)。若要評估要考量的 metadataBytes 數量，請檢閱下列中繼資料。Lambda 會將類似的中繼資料新增至每筆記錄：

```
{
```

```
    "time": "2022-08-20T12:31:32.123Z",
    "type": "function",
    "record": "Hello World"
}
```

- 如果訂閱者處理傳入遙測資料的速度不夠快，或是函數程式碼產生了極大量的日誌，Lambda 可能會捨棄記錄，以確保記憶體使用率維持在限制範圍內。發生這種情況時，Lambda 會傳送 `platform.logsDropped` 事件。

將訂閱請求傳送至遙測 API

Lambda 延伸項目可透過將訂閱請求傳送至遙測 API 來訂閱以接收遙測資料。訂閱請求應包含您希望延伸項目訂閱的事件類型相關資訊。此外，請求可以包含傳遞目的地資訊和緩衝組態。

傳送訂閱請求之前，您必須有延伸功能 ID (`Lambda-Extension-Identifier`)。向延伸 API 註冊您的延伸項目時，您可從 API 回應中取得延伸項目 ID。

訂閱會在延伸項目初始化階段進行。下列範例顯示訂閱全部三個遙測串流的 HTTP 請求：平台遙測、函數日誌和延伸項目日誌。

```
PUT http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry HTTP/1.1
{
   "schemaVersion": "2022-12-13",
   "types": [
        "platform",
        "function",
        "extension"
   ],
   "buffering": {
        "maxItems": 1000,
        "maxBytes": 256*1024,
        "timeoutMs": 100
   },
   "destination": {
        "protocol": "HTTP",
        "URI": "http://sandbox.localdomain:8080"
   }
}
```

如果請求成功，訂閱者會收到 HTTP 200 成功回應。

```
HTTP/1.1 200 OK
```

```
"OK"
```

輸入遙測 API 訊息

使用遙測 API 訂閱之後，延伸項目會自動透過 POST 請求開始接收來自 Lambda 的遙測資料。每個 POST 請求主體包含 Event 對象的數組。每個 Event 項目都有以下結構描述：

```
{
   time: String,
   type: String,
   record: Object
}
```

- time 屬性定義了 Lambda 平台產生事件的時間。這與事件實際發生的時間不同。time 的字串值是 ISO 8601 格式的時間戳記。
- type 屬性定義了事件類型。下表說明所有可能的值。
- record 屬性定義了包含遙測資料的 JSON 物件。此 JSON 物件的結構描述取決於 type。

下表摘要說明 Event 物件的所有類型，以及每個事件類型之遙測 API Event 結構描述參考的連結。

遙測 API 訊息類型

類別	事件類型	描述	事件記錄結構描述
平台事件	platform.initStart	函數初始化已開始。	the section called “platform.initStart ” 結構描述
平台事件	platform.initRuntimeDone	函數初始化已完成。	the section called “platform.initRuntimeDone ” 結構描述
平台事件	platform.initReport	函數初始化報告。	the section called “platform.initReport ” 結構描述

類別	事件類型	描述	事件記錄結構描述
平台事件	platform.start	函數調用已開始。	the section called “platform.start ” 結構描述
平台事件	platform.runtimeDone	執行階段已完成處理的事件，結果為成功或失敗。	the section called “platform.runtimeDone ” 結構描述
平台事件	platform.report	函數調用報告。	the section called “platform.report ” 結構描述
平台事件	platform.restoreStart	執行時間還原已開始。	the section called “platform.restoreStart ” 結構描述
平台事件	platform.restoreRuntimeDone	執行時間還原已完成。	the section called “platform.restoreRuntimeDone ” 結構描述
平台事件	platform.restoreReport	執行時間還原報告。	the section called “platform.restoreReport ” 結構描述
平台事件	platform.telemetrySubscription	延伸項目已訂閱遙測 API。	the section called “platform.telemetrySubscription ” 結構描述

類別	事件類型	描述	事件記錄結構描述
平台事件	`platform.logsDropped`	Lambda 已捨棄日誌項目。	the section called “`platform.logsDropped`” 結構描述
函數日誌	`function`	函數程式碼的日誌行。	the section called “`function`” 結構描述
延伸項目日誌	`extension`	延伸項目程式碼的日誌行。	the section called “`extension`” 結構描述

Lambda 遙測 API 參考

使用 Lambda 遙測 API 端點來讓延伸項目訂閱遙測串流。您可以從 AWS_LAMBDA_RUNTIME_API 環境變數中擷取遙測 API 端點。若要傳送 API 請求，請附加 API 版本 (2022-07-01/) 和telemetry/。例如：

```
http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry/
```

如需訂閱回應版本 2022-12-13 的 OpenAPI 規格 (OAS) 定義，請參閱：

- 郵telemetry-api-http-schema編
- TCP — telemetry-api-tcp-schema. 壓縮

API 操作

- 訂閱

訂閱

若要訂閱遙測串流，Lambda 延伸項目可以傳送訂閱 API 請求。

- 路徑 – /telemetry
- Method – PUT
- 標頭
  - Content-Type: application/json
- 請求內文參數
  - schemaVersion
    - 必要：是
    - 類型：String
    - 有效值："2022-12-13" 或 "2022-07-01"
  - destination – 定義遙測事件目的地和事件傳遞通訊協定的組態設定。
    - 必要：是
    - 類型：物件

```
{
    "protocol": "HTTP",
```

```
    "URI": "http://sandbox.localdomain:8080"
}
```

- protocol – Lambda 用來傳送遙測資料的通訊協定。
  - 必要：是
  - 類型：String
  - 有效值："HTTP"|"TCP"
- URI – 要傳送遙測資料的目的地 URI。
  - 必要：是
  - 類型：String
- 如需詳細資訊，請參閱 the section called “指定目的地通訊協定”。

- types – 您希望延伸項目訂閱的遙測類型。
  - 必要：是
  - 類型：字串陣列
  - 有效值："platform"|"function"|"extension"
- buffering – 事件緩衝的組態設定。
  - 必要：否
  - 類型：物件

```
{
   "buffering": {
        "maxItems": 1000,
        "maxBytes": 256*1024,
        "timeoutMs": 100
   }
}
```

  - maxItems – 記憶體中要緩衝的事件數目上限。
    - 必要：否
    - 類型：整數
    - 預設：1,000
    - 下限：1,000
    - 上限：10,000
  - maxBytes – 記憶體中要緩衝的遙測資料量上限 (位元組)。

    - 必要：否
    - 類型：整數
    - 預設：262,144
    - 下限：262,144
    - 上限：1,048,576
  - timeoutMs - 緩衝批次處理的時間上限 (毫秒)。
    - 必要：否
    - 類型：整數
    - 預設：1,000
    - 下限：25
    - 上限：30,000
  - 如需詳細資訊，請參閱 the section called “設定記憶體使用量和緩衝”。

訂閱 API 請求範例

```
PUT http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry HTTP/1.1
{
   "schemaVersion": "2022-12-13",
   "types": [
        "platform",
        "function",
        "extension"
   ],
   "buffering": {
        "maxItems": 1000,
        "maxBytes": 256*1024,
        "timeoutMs": 100
   },
   "destination": {
        "protocol": "HTTP",
        "URI": "http://sandbox.localdomain:8080"
   }
}
```

如果請求成功，延伸項目會收到 HTTP 200 成功回應：

```
HTTP/1.1 200 OK
```

```
"OK"
```

如果訂閱請求失敗，延伸項目會收到錯誤回應。例如：

```
HTTP/1.1 400 OK
{
    "errorType": "ValidationError",
    "errorMessage": "URI port is not provided; types should not be empty"
}
```

以下是延伸項目可能收到的其他幾個回應代碼：

- 200 - 請求已成功完成
- 202 - 已接受請求。本機測試環境中的訂閱請求回應
- 400 - 錯誤的請求
- 500 - 服務錯誤

Lambda 遙測 API **Event** 結構描述參考

使用 Lambda 遙測 API 端點來讓延伸項目訂閱遙測串流。您可以從 AWS_LAMBDA_RUNTIME_API 環境變數中擷取遙測 API 端點。若要傳送 API 請求，請附加 API 版本 (2022-07-01/) 和telemetry/。例如：

```
http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry/
```

如需訂閱回應版本 2022-12-13 的 OpenAPI 規格 (OAS) 定義，請參閱：

- 郵telemetry-api-http-schema編
- TCP — telemetry-api-tcp-schema. 壓縮

下表摘要說明遙測 API 支援的所有 Event 物件類型。

遙測 API 訊息類型

類別	事件類型	描述	事件記錄結構描述
平台事件	platform.initStart	函數初始化已開始。	the section called “platform.initStart ” 結構描述
平台事件	platform.initRuntimeDone	函數初始化已完成。	the section called “platform.initRuntimeDone ” 結構描述
平台事件	platform.initReport	函數初始化報告。	the section called “platform.initReport ” 結構描述
平台事件	platform.start	函數調用已開始。	the section called “platform.start ” 結構描述

類別	事件類型	描述	事件記錄結構描述
平台事件	platform.runtimeDone	執行階段已完成處理的事件，結果為成功或失敗。	the section called “platform.runtimeDone ”結構描述
平台事件	platform.report	函數調用報告。	the section called “platform.report ”結構描述
平台事件	platform.restoreStart	執行時間還原已開始。	the section called “platform.restoreStart ”結構描述
平台事件	platform.restoreRuntimeDone	執行時間還原已完成。	the section called “platform.restoreRuntimeDone ”結構描述
平台事件	platform.restoreReport	執行時間還原報告。	the section called “platform.restoreReport ”結構描述
平台事件	platform.telemetrySubscription	延伸項目已訂閱遙測 API。	the section called “platform.telemetrySubscription ”結構描述
平台事件	platform.logsDropped	Lambda 已捨棄日誌項目。	the section called “platform.logsDropped ”結構描述

類別	事件類型	描述	事件記錄結構描述
函數日誌	`function`	函數程式碼的日誌行。	the section called “`function`” 結構描述
延伸項目日誌	`extension`	延伸項目程式碼的日誌行。	the section called “`extension `” 結構描述

內容

- Span
- Status
- TraceContext
- TracingType

遙測 API **Event** 物件類型

本節詳細說明 Lambda 遙測 API 支援的 Event 物件類型。在事件描述中，問號 (?) 表示該屬性可能不存在於物件中。

platform.initStart

platform.initStart 事件表示函數初始化階段已開始。platform.initStart Event 物件的結構如下：

```
Event: Object
- time: String
- type: String = platform.initStart
- record: PlatformInitStart
```

PlatformInitStart 物件具有下列屬性：

- functionName – String
- functionVersion – String
- initializationType – the section called “InitType” 物件
- instanceId? – String
- instanceMaxMemory? – Integer
- phase – the section called “InitPhase” 物件
- runtimeVersion? – String
- runtimeVersionArn? – String

以下是 platform.initStart 類型 Event 的範例：

```
{
    "time": "2022-10-12T00:00:15.064Z",
    "type": "platform.initStart",
```

```
    "record": {
        "initializationType": "on-demand",
        "phase": "init",
        "runtimeVersion": "nodejs-14.v3",
        "runtimeVersionArn": "arn",
        "functionName": "myFunction",
        "functionVersion": "$LATEST",
        "instanceId": "82561ce0-53dd-47d1-90e0-c8f5e063e62e",
        "instanceMaxMemory": 256
    }
}
```

platform.initRuntimeDone

platform.initRuntimeDone 事件表示函數初始化階段已完成。platform.initRuntimeDone Event 物件的結構如下：

```
Event: Object
- time: String
- type: String = platform.initRuntimeDone
- record: PlatformInitRuntimeDone
```

PlatformInitRuntimeDone 物件具有下列屬性：

- initializationType – the section called “InitType” 物件
- phase – the section called “InitPhase” 物件
- status – the section called “Status” 物件
- spans? – the section called “Span” 物件的清單

以下是 platform.initRuntimeDone 類型 Event 的範例：

```
{
    "time": "2022-10-12T00:01:15.000Z",
    "type": "platform.initRuntimeDone",
    "record": {
        "initializationType": "on-demand"
        "status": "success",
        "spans": [
            {
                "name": "someTimeSpan",
```

```
                "start": "2022-06-02T12:02:33.913Z",
                "durationMs": 70.5
            }
        ]
    }
}
```

platform.initReport

platform.initReport 事件包含函數初始化階段的整體報告。platform.initReport Event 物件的結構如下：

```
Event: Object
- time: String
- type: String = platform.initReport
- record: PlatformInitReport
```

PlatformInitReport 物件具有下列屬性：

- errorType? - 字串
- initializationType – the section called “InitType” 物件
- phase – the section called “InitPhase” 物件
- metrics – the section called “InitReportMetrics” 物件
- spans? – the section called “Span” 物件的清單
- status – the section called “Status” 物件

以下是 platform.initReport 類型 Event 的範例：

```
{
    "time": "2022-10-12T00:01:15.000Z",
    "type": "platform.initReport",
    "record": {
        "initializationType": "on-demand",
        "status": "success",
        "phase": "init",
        "metrics": {
            "durationMs": 125.33
        },
        "spans": [
```

```
            {
                "name": "someTimeSpan",
                "start": "2022-06-02T12:02:33.913Z",
                "durationMs": 90.1
            }
        ]
    }
}
```

platform.start

platform.start 事件表示函數調用階段已開始。platform.start Event 物件的結構如下：

```
Event: Object
- time: String
- type: String = platform.start
- record: PlatformStart
```

PlatformStart 物件具有下列屬性：

- requestId – String
- version? – String
- tracing? – the section called “TraceContext”

以下是 platform.start 類型 Event 的範例：

```
{
    "time": "2022-10-12T00:00:15.064Z",
    "type": "platform.start",
    "record": {
        "requestId": "6d68ca91-49c9-448d-89b8-7ca3e6dc66aa",
        "version": "$LATEST",
        "tracing": {
            "spanId": "54565fb41ac79632",
            "type": "X-Amzn-Trace-Id",
            "value":
 "Root=1-62e900b2-710d76f009d6e7785905449a;Parent=0efbd19962d95b05;Sampled=1"
        }
    }
}
```

platform.runtimeDone

platform.runtimeDone 事件表示函數調用階段已完成。platform.runtimeDone Event 物件的結構如下：

```
Event: Object
- time: String
- type: String = platform.runtimeDone
- record: PlatformRuntimeDone
```

PlatformRuntimeDone 物件具有下列屬性：

- errorType? – String
- metrics? – the section called “RuntimeDoneMetrics” 物件
- requestId – String
- status – the section called “Status” 物件
- spans? – the section called “Span” 物件的清單
- tracing? – the section called “TraceContext” 物件

以下是 platform.runtimeDone 類型 Event 的範例：

```
{
    "time": "2022-10-12T00:01:15.000Z",
    "type": "platform.runtimeDone",
    "record": {
        "requestId": "6d68ca91-49c9-448d-89b8-7ca3e6dc66aa",
        "status": "success",
        "tracing": {
            "spanId": "54565fb41ac79632",
            "type": "X-Amzn-Trace-Id",
            "value":
 "Root=1-62e900b2-710d76f009d6e7785905449a;Parent=0efbd19962d95b05;Sampled=1"
        },
        "spans": [
            {
                "name": "someTimeSpan",
                "start": "2022-08-02T12:01:23:521Z",
                "durationMs": 80.0
            }
```

```
        ],
        "metrics": {
            "durationMs": 140.0,
            "producedBytes": 16
        }
    }
}
```

platform.report

platform.report 事件包含函數初始化階段的整體報告。platform.report Event 物件的結構如下：

```
Event: Object
- time: String
- type: String = platform.report
- record: PlatformReport
```

PlatformReport 物件具有下列屬性：

- metrics – the section called “ReportMetrics” 物件
- requestId – String
- spans? – the section called “Span” 物件的清單
- status – the section called “Status” 物件
- tracing? – the section called “TraceContext” 物件

以下是 platform.report 類型 Event 的範例：

```
{
    "time": "2022-10-12T00:01:15.000Z",
    "type": "platform.report",
    "record": {
        "metrics": {
            "billedDurationMs": 694,
            "durationMs": 693.92,
            "initDurationMs": 397.68,
            "maxMemoryUsedMB": 84,
            "memorySizeMB": 128
        },
```

```
        "requestId": "6d68ca91-49c9-448d-89b8-7ca3e6dc66aa",
    }
}
```

platform.restoreStart

platform.restoreStart 事件表示函數環境還原事件已開始。在環境還原事件中，Lambda 會從快取的快照建立環境，而不是從頭開始初始化。如需詳細資訊，請參閱 Lambda SnapStart。platform.restoreStart Event 物件的結構如下：

```
Event: Object
- time: String
- type: String = platform.restoreStart
- record: PlatformRestoreStart
```

PlatformRestoreStart 物件具有下列屬性：

- functionName – String
- functionVersion – String
- instanceId? – String
- instanceMaxMemory? – String
- runtimeVersion? – String
- runtimeVersionArn? – String

以下是 platform.restoreStart 類型 Event 的範例：

```
{
    "time": "2022-10-12T00:00:15.064Z",
    "type": "platform.restoreStart",
    "record": {
        "runtimeVersion": "nodejs-14.v3",
        "runtimeVersionArn": "arn",
        "functionName": "myFunction",
        "functionVersion": "$LATEST",
        "instanceId": "82561ce0-53dd-47d1-90e0-c8f5e063e62e",
        "instanceMaxMemory": 256
    }
}
```

platform.restoreRuntimeDone

platform.restoreRuntimeDone 事件表示函數環境還原事件已完成。在環境還原事件中，Lambda 會從快取的快照建立環境，而不是從頭開始初始化。如需詳細資訊，請參閱 Lambda SnapStart。platform.restoreRuntimeDone Event 物件的結構如下：

```
Event: Object
- time: String
- type: String = platform.restoreRuntimeDone
- record: PlatformRestoreRuntimeDone
```

PlatformRestoreRuntimeDone 物件具有下列屬性：

- errorType? – String
- spans? – the section called “Span” 物件的清單
- status – the section called “Status” 物件

以下是 platform.restoreRuntimeDone 類型 Event 的範例：

```
{
    "time": "2022-10-12T00:00:15.064Z",
    "type": "platform.restoreRuntimeDone",
    "record": {
        "status": "success",
        "spans": [
            {
                "name": "someTimeSpan",
                "start": "2022-08-02T12:01:23:521Z",
                "durationMs": 80.0
            }
        ]
    }
}
```

platform.restoreReport

platform.restoreReport 事件包含函數還原事件的整體報告。platform.restoreReport Event 物件的結構如下：

```
Event: Object
- time: String
```

```
- type: String = platform.restoreReport
- record: PlatformRestoreReport
```

PlatformRestoreReport 物件具有下列屬性：

- errorType? - 字串
- metrics? – the section called “RestoreReportMetrics” 物件
- spans? – the section called “Span” 物件的清單
- status – the section called “Status” 物件

以下是 platform.restoreReport 類型 Event 的範例：

```
{
    "time": "2022-10-12T00:00:15.064Z",
    "type": "platform.restoreReport",
    "record": {
        "status": "success",
        "metrics": {
            "durationMs": 15.19
        },
        "spans": [
            {
                "name": "someTimeSpan",
                "start": "2022-08-02T12:01:23:521Z",
                "durationMs": 30.0
            }
        ]
    }
}
```

platform.extension

extension 事件包含延伸項目程式碼的日誌。extension Event 物件的結構如下：

```
Event: Object
- time: String
- type: String = extension
- record: {}
```

PlatformExtension 物件具有下列屬性：

- events – String 清單
- name – String
- state – String

以下是 platform.extension 類型 Event 的範例：

```
{
    "time": "2022-10-12T00:02:15.000Z",
    "type": "platform.extension",
    "record": {
        "events": [ "INVOKE", "SHUTDOWN" ],
        "name": "my-telemetry-extension",
        "state": "Ready"
    }
}
```

platform.telemetrySubscription

platform.telemetrySubscription 事件包含延伸項目訂閱的相關資訊。platform.telemetrySubscription Event 物件的結構如下：

```
Event: Object
- time: String
- type: String = platform.telemetrySubscription
- record: PlatformTelemetrySubscription
```

PlatformTelemetrySubscription 物件具有下列屬性：

- name – String
- state – String
- types – String 清單

以下是 platform.telemetrySubscription 類型 Event 的範例：

```
{
    "time": "2022-10-12T00:02:35.000Z",
    "type": "platform.telemetrySubscription",
    "record": {
```

```
        "name": "my-telemetry-extension",
        "state": "Subscribed",
        "types": [ "platform", "function" ]
    }
}
```

platform.logsDropped

platform.logsDropped 事件包含已捨棄事件的相關資訊。當函數以過高的速率輸出日誌，Lambda 無法處理日誌時，Lambda 會發出platform.logsDropped事件。當 Lambda 無法以函數產生的速率將記錄檔傳送至 CloudWatch 或傳送至訂閱遙測 API 的擴充功能時，會捨棄記錄檔以防止函數的執行速度變慢。platform.logsDropped Event 物件的結構如下：

```
Event: Object
- time: String
- type: String = platform.logsDropped
- record: PlatformLogsDropped
```

PlatformLogsDropped 物件具有下列屬性：

- droppedBytes – Integer
- droppedRecords – Integer
- reason – String

以下是 platform.logsDropped 類型 Event 的範例：

```
{
    "time": "2022-10-12T00:02:35.000Z",
    "type": "platform.logsDropped",
    "record": {
        "droppedBytes": 12345,
        "droppedRecords": 123,
        "reason": "Some logs were dropped because the downstream consumer is slower
 than the logs production rate"
    }
}
```

function

function 事件包含函數程式碼的日誌。function Event 物件的結構如下：

```
Event: Object
- time: String
- type: String = function
- record: {}
```

`record` 欄位的格式取決於函數的日誌檔是以純文字格式還是 JSON 格式而定。若要進一步瞭解日誌格式設定選項，請參閱 the section called “設定 JSON 和純文字日誌格式”

以下是日誌格式為純文字時類型 `function` 的範例 `Event`。

```
{
    "time": "2022-10-12T00:03:50.000Z",
    "type": "function",
    "record": "[INFO] Hello world, I am a function!"
}
```

以下是日誌格式為 JSON 時的類型 `function` 的範例 `Event`。

```
{
    "time": "2022-10-12T00:03:50.000Z",
    "type": "function",
    "record": {
        "timestamp": "2022-10-12T00:03:50.000Z",
        "level": "INFO",
        "requestId": "79b4f56e-95b1-4643-9700-2807f4e68189",
        "message": "Hello world, I am a function!"
    }
}
```

> **Note**
>
> 如果您使用的描述版本比 2022-12-13 版本舊，則即使函數的 "record" 日誌格式配置為 JSON，也始終將其呈現為字串。

extension

`extension` 事件包含延伸項目程式碼的日誌。`extension` `Event` 物件的結構如下：

```
Event: Object
```

```
- time: String
- type: String = extension
- record: {}
```

record 欄位的格式取決於函數的日誌檔是以純文字格式還是 JSON 格式而定。若要進一步瞭解日誌格式設定選項，請參閱 the section called “設定 JSON 和純文字日誌格式”

以下是日誌格式為純文字時類型 extension 的範例 Event。

```
{
    "time": "2022-10-12T00:03:50.000Z",
    "type": "extension",
    "record": "[INFO] Hello world, I am an extension!"
}
```

以下是日誌格式為 JSON 時的類型 extension 的範例 Event。

```
{
    "time": "2022-10-12T00:03:50.000Z",
    "type": "extension",
    "record": {
       "timestamp": "2022-10-12T00:03:50.000Z",
       "level": "INFO",
       "requestId": "79b4f56e-95b1-4643-9700-2807f4e68189",
       "message": "Hello world, I am an extension!"
    }
}
```

> **Note**
>
> 如果您使用的描述版本比 2022-12-13 版本舊，則即使函數的 "record" 日誌格式配置為 JSON，也始終將其呈現為字串。

共用物件類型

本節詳細說明 Lambda 遙測 API 支援的共用物件類型。

InitPhase

字串列舉，描述初始化步驟發生時的階段。多數情況下，Lambda 會在 `init` 階段執行函數初始化程式碼。但是在某些錯誤情況下，Lambda 可能會在 `invoke` 階段重新執行函數初始化程式碼。(這稱為隱藏的初始化。)

- 類型 – `String`
- 有效值 – `init|invoke|snap-start`

InitReportMetrics

包含初始化階段相關指標的物件。

- 類型 – `Object`

`InitReportMetrics` 物件的結構如下：

```
InitReportMetrics: Object
- durationMs: Double
```

下列為 `InitReportMetrics` 物件的範例:

```
{
    "durationMs": 247.88
}
```

InitType

字串列舉，描述 Lambda 如何初始化環境。

- 類型 – `String`
- 有效值 – `on-demand|provisioned-concurrency`

ReportMetrics

包含已完成階段相關指標的物件。

- 類型 – `Object`

ReportMetrics 物件的結構如下：

```
ReportMetrics: Object
- billedDurationMs: Integer
- durationMs: Double
- initDurationMs?: Double
- maxMemoryUsedMB: Integer
- memorySizeMB: Integer
- restoreDurationMs?: Double
```

下列為 ReportMetrics 物件的範例:

```
{
    "billedDurationMs": 694,
    "durationMs": 693.92,
    "initDurationMs": 397.68,
    "maxMemoryUsedMB": 84,
    "memorySizeMB": 128
}
```

RestoreReportMetrics

包含已完成還原階段相關指標的物件。

- 類型 – Object

RestoreReportMetrics 物件的結構如下：

```
RestoreReportMetrics: Object
- durationMs: Double
```

下列為 RestoreReportMetrics 物件的範例:

```
{
    "durationMs": 15.19
}
```

RuntimeDoneMetrics

包含調用階段相關指標的物件。

- 類型 – Object

RuntimeDoneMetrics 物件的結構如下：

```
RuntimeDoneMetrics: Object
- durationMs: Double
- producedBytes?: Integer
```

下列為 RuntimeDoneMetrics 物件的範例:

```
{
    "durationMs": 200.0,
    "producedBytes": 15
}
```

Span

包含跨度詳細資訊的物件。跨度表示追蹤中的工作或作業單位。如需有關範圍的詳細資訊，請參閱 OpenTelemetry 文件網站追蹤 API 頁面上的跨距。

Lambda 針對 platform.RuntimeDone 事件支援下列跨度：

- responseLatency 跨度描述 Lambda 函數開始傳送回應所花費的時間。
- responseDuration 跨度描述 Lambda 函數完成傳送整個回應所花費的時間。
- runtimeOverhead 跨度描述 Lambda 執行期表示已準備好處理下一個函數調用所花費的時間。這是執行期傳回函數回應後，呼叫下一個調用 API 所花費的時間。

下列為 responseLatency 跨度物件的範例:

```
{
        "name": "responseLatency",
        "start": "2022-08-02T12:01:23.521Z",
        "durationMs": 23.02
      }
```

Status

描述初始化或呼叫階段狀態的物件。如果狀態為 failure 或 error，則 Status 物件也會包含描述錯誤的 errorType 欄位。

- 類型 – Object
- 有效狀態值 – success|failure|error|timeout

TraceContext

描述追蹤屬性的物件。

- 類型 – Object

TraceContext 物件的結構如下：

```
TraceContext: Object
- spanId?: String
- type: TracingType enum
- value: String
```

下列為 TraceContext 物件的範例:

```
{
    "spanId": "073a49012f3c312e",
    "type": "X-Amzn-Trace-Id",
    "value":
 "Root=1-62e900b2-710d76f009d6e7785905449a;Parent=0efbd19962d95b05;Sampled=1"
}
```

TracingType

字串列舉，描述 the section called “TraceContext” 物件中追蹤的類型。

- 類型 – String
- 有效值 – X-Amzn-Trace-Id

將 Lambda 遙測 API **Event** 物件轉換為 OpenTelemetry跨度

AWS Lambda遙測 API 結構描述在語義上與 OpenTelemetry (oTel) 相容。這表示您可以將AWS Lambda遙測 API `Event` 物件轉換為 OpenTelemetry (oTel) 跨度。轉換時，不應將單一 `Event` 物件映射到單一 oTel 跨度。而是應該在單一 oTel 跨度中顯示與生命週期階段相關的全部三個事件。例如，`start`、`runtimeDone` 和 `runtimeReport` 事件代表單一函數叫用。將這三個事件做為一個單一的 oTel 跨度呈現。

您可以使用跨度事件或子 (巢狀) 跨度轉換事件。此頁面上的表格針對這兩種做法，說明遙測 API 結構描述屬性與 oTel 跨度屬性之間的映射。如需 Otel 範圍的詳細資訊，請參閱 OpenTelemetry 文件網站追蹤 API 頁面上的範圍。

章節

透過跨度事件映射到 OTel 跨度

在下表中，e 代表來自遙測來源的事件。

映射 ***Start*** 事件

OpenTelemetry	Lambda 遙測 API 結構描述
`Span.Name`	延伸項目會根據 `type` 欄位產生此值。
`Span.StartTime`	請使用 `e.time`。
`Span.EndTime`	N/A，因為事件尚未完成。
`Span.Kind`	設定為 `Server`。
`Span.Status`	設定為 `Unset`。
`Span.TraceId`	剖析 `e.tracing.value` 中找到的 AWS X-Ray 標頭，然後使用 `TraceId` 值。
`Span.ParentId`	剖析 `e.tracing.value` 中找到的 X-Ray 標頭，然後使用 `Parent` 值。

OpenTelemetry	Lambda 遙測 API 結構描述
Span.SpanId	使用 e.tracing.spanId （如果可用)。若無法使用，請產生一個新的 SpanId。
Span.SpanContext.TraceState	X-Ray 追蹤內容則為 N/A。
Span.SpanContext.TraceFlags	剖析 e.tracing.value 中找到的 X-Ray 標頭，然後使用 Sampled 值。
Span.Attributes	延伸項目可以在此處新增任何自訂值。

映射 ***RuntimeDone** 事件

OpenTelemetry	Lambda 遙測 API 結構描述
Span.Name	延伸項目會根據 type 欄位產生值。
Span.StartTime	使用相符 *Start 事件中的 e.time。 或使用 e.time - e.metrics.durationMs 。
Span.EndTime	N/A，因為事件尚未完成。
Span.Kind	設定為 Server。
Span.Status	如果 e.status 不等於 success，則設定為 Error。 否則，請設定為 Ok。
Span.Events[]	請使用 e.spans[] 。
Span.Events[i].Name	請使用 e.spans[i].name 。
Span.Events[i].Time	請使用 e.spans[i].start 。
Span.TraceId	剖析 e.tracing.value 中找到的 AWS X-Ray 標頭，然後使用 TraceId 值。

OpenTelemetry	Lambda 遙測 API 結構描述
Span.ParentId	剖析 e.tracing.value 中找到的 X-Ray 標頭，然後使用 Parent 值。
Span.SpanId	使用 *Start 事件的相同 SpanId。如果無法使用，則使用 e.tracing.spanId 或產生新的 SpanId。
Span.SpanContext.TraceState	X-Ray 追蹤內容則為 N/A。
Span.SpanContext.TraceFlags	剖析 e.tracing.value 中找到的 X-Ray 標頭，然後使用 Sampled 值。
Span.Attributes	延伸項目可以在此處新增任何自訂值。

映射 ***Report** 事件

OpenTelemetry	Lambda 遙測 API 結構描述
Span.Name	延伸項目會根據 type 欄位產生值。
Span.StartTime	使用相符 *Start 事件中的 e.time。 或使用 e.time - e.metrics.durationMs 。
Span.EndTime	請使用 e.time。
Span.Kind	設定為 Server。
Span.Status	使用與 *RuntimeDone 事件相同的值。
Span.TraceId	剖析 e.tracing.value 中找到的 AWS X-Ray 標頭，然後使用 TraceId 值。
Span.ParentId	剖析 e.tracing.value 中找到的 X-Ray 標頭，然後使用 Parent 值。

OpenTelemetry	Lambda 遙測 API 結構描述
Span.SpanId	使用 *Start 事件的相同 SpanId。如果無法使用，則使用 e.tracing.spanId 或產生新的 SpanId。
Span.SpanContext.TraceState	X-Ray 追蹤內容則為 N/A。
Span.SpanContext.TraceFlags	剖析 e.tracing.value 中找到的 X-Ray 標頭，然後使用 Sampled 值。
Span.Attributes	延伸項目可以在此處新增任何自訂值。

透過子跨度映射到 OTel 跨度

下表說明如何針對 *RuntimeDone 跨度，透過子 (巢狀) 跨度將 Lambda 遙測 API 事件轉換為 OTel 跨度。如需 *Start 和 *Report 映射，請參閱「the section called “透過跨度事件映射到 OTel 跨度”」中的表格，因為對子跨度來說是相同的。在本表中，e 代表來自遙測來源的事件。

映射 ***RuntimeDone** 事件

OpenTelemetry	Lambda 遙測 API 結構描述
Span.Name	延伸項目會根據 type 欄位產生值。
Span.StartTime	使用相符 *Start 事件中的 e.time。 或使用 e.time - e.metrics.durationMs 。
Span.EndTime	N/A，因為事件尚未完成。
Span.Kind	設定為 Server。
Span.Status	如果 e.status 不等於 success，則設定為 Error。 否則，請設定為 Ok。

OpenTelemetry	Lambda 遙測 API 結構描述
Span.TraceId	剖析 e.tracing.value 中找到的 AWS X-Ray 標頭，然後使用 TraceId 值。
Span.ParentId	剖析 e.tracing.value 中找到的 X-Ray 標頭，然後使用 Parent 值。
Span.SpanId	使用 *Start 事件的相同 SpanId。如果無法使用，則使用 e.tracing.spanId 或產生新的 SpanId。
Span.SpanContext.TraceState	X-Ray 追蹤內容則為 N/A。
Span.SpanContext.TraceFlags	剖析 e.tracing.value 中找到的 X-Ray 標頭，然後使用 Sampled 值。
Span.Attributes	延伸項目可以在此處新增任何自訂值。
ChildSpan[i].Name	請使用 e.spans[i].name 。
ChildSpan[i].StartTime	請使用 e.spans[i].start 。
ChildSpan[i].EndTime	請使用 e.spans[i].start + e.spans[i].durations 。
ChildSpan[i].Kind	與父項 Span.Kind 相同。
ChildSpan[i].Status	與父項 Span.Status 相同。
ChildSpan[i].TraceId	與父項 Span.TraceId 相同。
ChildSpan[i].ParentId	使用父項 Span.SpanId 。
ChildSpan[i].SpanId	產生新的 SpanId。
ChildSpan[i].SpanContext.TraceState	X-Ray 追蹤內容則為 N/A。

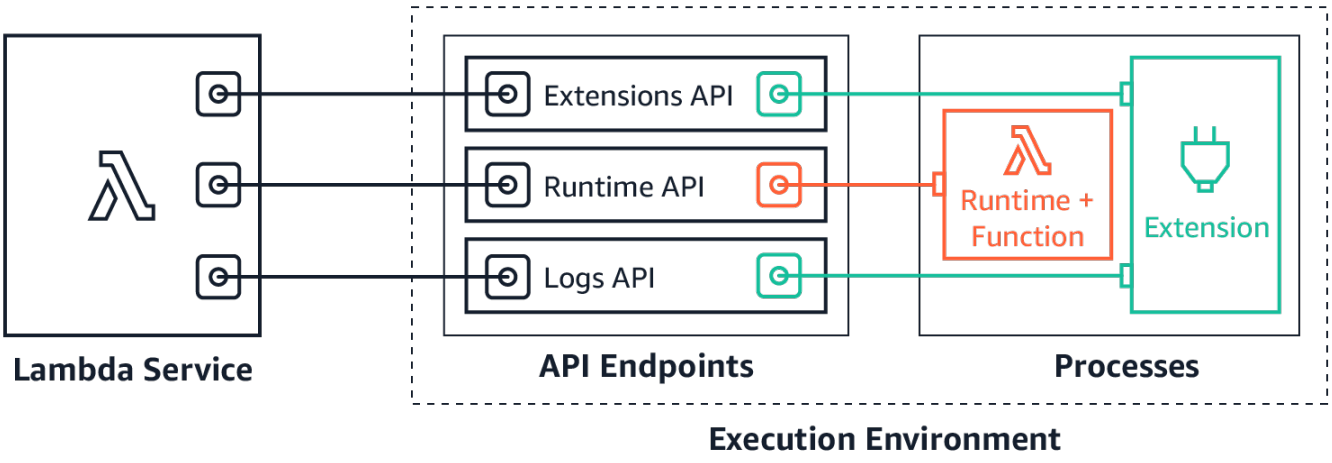
OpenTelemetry	Lambda 遙測 API 結構描述
`ChildSpan[i].SpanContext.TraceFlags`	與父項 `Span.SpanContext.TraceFlags` 相同。

Lambda Logs API

> **Important**
>
> Lambda 遙測 API 優先於 Lambda 日誌 API。雖然日誌 API 仍然正常運作，但我們建議您未來只使用遙測 API。您可以使用遙測 API 或日誌 API 訂閱遙測串流的延伸項目。使用其中一個 API 進行訂閱後，若嘗試使用其他 API 進行任何訂閱，都會傳回錯誤。

Lambda 會自動擷取執行階段日誌並將其串流至 Amazon CloudWatch 此日誌串流包含函數程式碼和延伸項目產生的日誌，以及 Lambda 作為函數調用一部分產生的日誌。

Lambda 延伸項目可以使用 Lambda Runtime Logs API，直接從 Lambda 執行環境中訂閱記錄串流。Lambda 會將日誌串流至延伸項目，延伸項目隨後可處理、篩選日誌並將其傳送至任何偏好目的地。



Logs API 允許延伸項目訂閱三種不同的日誌串流：

- Lambda 函數產生並寫入到 `stdout` 或 `stderr` 的函數日誌。
- 延伸項目程式碼產生的延伸項目日誌。
- Lambda 平台日誌，它記錄與調用和延伸項目相關的事件和錯誤。

> **Note**
>
> Lambda 會將所有記錄傳送至 CloudWatch，即使擴充功能訂閱一或多個日誌串流也是如此。

主題

訂閱接收日誌

Lambda 延伸項目可透過傳送訂閱請求至 Logs API 來訂閱接收日誌。

若要訂閱接收日誌，您需要延伸項目識別符 (`Lambda-Extension-Identifier`)。首先註冊延伸項目以接收延伸項目識別符。然後在初始化期間訂閱 Logs API。初始化階段完成後，Lambda 不會處理訂閱請求。

> **Note**
>
> Logs API 訂閱為等冪操作。重複的訂閱請求不會導致重複訂閱。

記憶體用量

隨著訂閱者數量的增加，記憶體用量會線性增加。訂閱會耗用記憶體資源，因為每個訂閱都會開啟新的記憶體緩衝區來存放日誌。為了協助最佳化記憶體用量，您可以調整緩衝組態。緩衝區記憶體用量會計入執行環境中的整體記憶體耗用量。

目的地協定

您可以選擇下列其中一個協定來接收日誌：

1. HTTP (建議) - Lambda 會以 JSON 格式的記錄陣列將日誌傳遞至本機 HTTP 端點 (`http://sandbox.localdomain:${PORT}/${PATH}`)。$PATH 為選用參數。請注意，僅支援 HTTP，而不是 HTTPS。您可以選擇透過 PUT 或 POST 接收日誌。
2. TCP - Lambda 以換行分隔的 JSON (NDJSON) 格式將日誌傳遞至 TCP 連接埠。

建議使用 HTTP，而不是使用 TCP。使用 TCP，Lambda 平台無法確認何時將日誌傳遞至應用程式層。因此，如果您的延伸項目損毀，可能會遺失日誌。HTTP 不會共享此限制。

我們也建議您先設定本機 HTTP 接聽程式或 TCP 連接埠，然後再訂閱接收日誌。在設定期間，請注意下列事項：

- Lambda 只會將日誌傳送至執行環境內的目的地。
- 如果沒有接聽程式，或者如果 POST 或 PUT 請求導致錯誤，則 Lambda 會重新嘗試傳送日誌 (使用輪詢)。如果日誌訂閱者當機，則在 Lambda 重新啟動執行環境之後會繼續接收日誌。
- Lambda 保留連接埠 9001。沒有其他連接埠編號限制或建議。

緩衝組態

Lambda 可以緩衝日誌並將其提供給訂閱者。您可以透過指定下列選用欄位，在訂閱請求中設定此行為。請注意，Lambda 會對您未指定的任何欄位使用預設值。

- timeoutMs - 緩衝批次處理的時間上限 (毫秒)。預設：1,000。下限：25。上限：30,000。
- maxBytes - 記憶體中要緩衝的日誌大小上限 (位元組)。預設：262,144。下限：262,144。上限：1,048,576。
- maxItems – 記憶體中要緩衝的事件數目上限。預設：10,000。下限：1,000。上限：10,000。

在緩衝組態期間，請注意下列幾點：

- 如果任何輸入串流被關閉，例如，如果執行時間崩潰，Lambda 會清除日誌。
- 每個訂閱者可以在訂閱請求中指定不同的緩衝組態。
- 考慮讀取資料所需的緩衝區大小。預計接收最大為 `2*maxBytes+metadata` 的承載，其中 `maxBytes` 在訂閱請求中進行設定。例如，Lambda 將下列中繼資料位元組新增至每個記錄：

```
{
"time": "2020-08-20T12:31:32.123Z",
"type": "function",
"record": "Hello World"
}
```

- 如果訂閱者無法足夠快速地處理傳入日誌，Lambda 可能會丟棄日誌，以確保記憶體使用率受到限制。若要指示丟棄的記錄數量，Lambda 會傳送 `platform.logsDropped` 日誌。

訂閱範例

訂閱平台和函數日誌的請求如下列範例所示。

```
PUT http://${AWS_LAMBDA_RUNTIME_API}/2020-08-15/logs HTTP/1.1
{ "schemaVersion": "2020-08-15",
  "types": [
      "platform",
      "function"
    ],
  "buffering": {
      "maxItems": 1000,
      "maxBytes": 262144,
      "timeoutMs": 100
    },
  "destination": {
    "protocol": "HTTP",
    "URI": "http://sandbox.localdomain:8080/lambda_logs"
  }
}
```

如果請求成功，訂閱者會收到 HTTP 200 成功回應。

```
HTTP/1.1 200 OK
"OK"
```

Logs API 的範本程式碼

如需示範如何將日誌傳送至自訂目的地的範本程式碼，請參閱 AWS 運算部落格上的使用 AWS Lambda 延伸項目將日誌傳送至自訂目的地。

如需說明如何開發基本 Lambda 擴充功能及訂閱記錄 API 的 Python 和 Go 程式碼範例，請參閱AWS 範例 GitHub 儲存庫上的AWS Lambda擴充功能。如需建置 Lambda 延伸項目的詳細資訊，請參閱 the section called “Extensions API”。

Logs API 參考

您可以從 `AWS_LAMBDA_RUNTIME_API` 環境變數中擷取 Logs API 端點。若要傳送 API 請求，請在 API 路徑之前使用前綴 `2020-08-15/`。例如：

```
http://${AWS_LAMBDA_RUNTIME_API}/2020-08-15/logs
```

應用程式介面版本的 OpenAPI 規格可在此處取得：.zip logs-api-request

訂閱

若要訂閱 Lambda 執行環境中可用的一個或多個日誌串流，延伸項目會傳送 Subscribe API 請求。

路徑 – `/logs`

方法 – PUT

主體參數

`destination` – 請參閱the section called “目的地協定”。必要：是。類型：字串。

`buffering` – 請參閱the section called “緩衝組態”。必要：否。類型：字串。

`types` - 要接收的日誌類型陣列。必要：是。類型：字串陣列。有效值：「平台」、「函數」、「延伸項目」。

`schemaVersion` - 必要：否。預設值："2020-08-15"。將延伸項目設定為 "2021-03-18"，可接收 `platform.runtimeDone` 訊息。

回應參數

訂閱回應的 OpenAPI 規範 (版本 2020-08-15)，適用於 HTTP 和 TCP 通訊協定：

- 拉logs-api-http-response鍊
- TCP: logs-api-tcp-response. 壓縮

回應代碼

- 200 - 請求已成功完成
- 202 - 已接受請求。在本機測試期間回應訂閱請求。
- 4XX - 錯誤請求
- 500 - 服務錯誤

如果請求成功，訂閱者會收到 HTTP 200 成功回應。

```
HTTP/1.1 200 OK
"OK"
```

如果請求失敗，訂閱者會收到錯誤回應。例如：

```
HTTP/1.1 400 OK
{
    "errorType": "Logs.ValidationError",
    "errorMessage": URI port is not provided; types should not be empty"
}
```

日誌訊息

Logs API 允許延伸項目訂閱三種不同的日誌串流：

- 函數 - Lambda 函數產生並寫入到 `stdout` 或 `stderr` 的日誌。
- 延伸項目 - 延伸項目程式碼產生的日誌。
- 平台 - 執行時間平台產生的日誌，它們記錄與調用和延伸相關的事件和錯誤。

主題

函數日誌

Lambda 函數和內部延伸項目會產生函數日誌並將它們寫入 `stdout` 或 `stderr`。

下列範例顯示函數日誌訊息的格式。{ "time": "2020-08-20T12:31:32.123Z", "type": "function", "record": "ERROR encountered. Stack trace:\n\my-function (line 10)\n" }

延伸項目日誌

延伸項目可產生延伸日誌。日誌格式與函數日誌的格式相同。

平台日誌

Lambda 會產生平台事件 (例如 `platform.start`、`platform.end` 以及 `platform.fault`) 的日誌訊息。

或者，您可以訂閱包含 `platform.runtimeDone` 日誌訊息的 2021-03-18 版本的 Logs API 結構描述。

平台日誌訊息範例

下面的範例顯示了平台開始和平台結束日誌。這些日誌指出 RequestID 所指定之調用的調用開始時間和調用結束時間。

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.start",
    "record": {"requestId": "6f7f0961f83442118a7af6fe80b88d56"}
}
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.end",
    "record": {"requestId": "6f7f0961f83442118a7af6fe80b88d56"}
}
```

該平台。 initRuntimeDone日誌信息顯示子階段的狀態，Runtime init子階段是「初始化」生命周期階段的一部分。Runtime init 成功時，執行階段會傳送 /next 執行階段 API 請求 (針對 on-demand 和 provisioned-concurrency 初始化類型) 或 restore/next (針對 snap-start 初始化類型)。下面的例子顯示了一個成功的平台。 initRuntimeDonesnap-start初始化類型的日誌消息。

```
{
  "time":"2022-07-17T18:41:57.083Z",
  "type":"platform.initRuntimeDone",
  "record":{
      "initializationType":"snap-start",
      "status":"success"
  }
}
```

platform.initReport 日誌訊息會顯示 Init 階段持續的時間長度，以及您須為此階段支付的費用。當初始化類型為 provisioned-concurrency，Lambda 會在調用期間傳送此訊息。當初始化類型為 snap-start，Lambda 會在還原快照之後傳送此訊息。下列範例顯示 snap-start 初始化類型的 platform.initReport 日誌訊息。

```
{
  "time":"2022-07-17T18:41:57.083Z",
  "type":"platform.initReport",
  "record":{
      "initializationType":"snap-start",
      "metrics":{
```

```
          "durationMs":731.79,
          "billedDurationMs":732
          }
  }
}
```

平台報告日誌包含 RequestID 所指定之調用的相關指標。只有調用包含冷啟動時，initDurationMs 欄位才會包含在日誌中。如果 AWS X-Ray 追蹤處於作用中狀態，則日誌包含 X-Ray 中繼資料。下列範例顯示包含冷啟動之調用的平台報告日誌。

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.report",
    "record": {"requestId": "6f7f0961f83442118a7af6fe80b88d56",
        "metrics": {"durationMs": 101.51,
            "billedDurationMs": 300,
            "memorySizeMB": 512,
            "maxMemoryUsedMB": 33,
            "initDurationMs": 116.67
        }
    }
}
```

平台故障日誌會擷取執行時間或執行環境錯誤。平台錯誤日誌訊息如下列範例所示。

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.fault",
    "record": "RequestId: d783b35e-a91d-4251-af17-035953428a2c Process exited before
 completing request"
}
```

當延伸項目註冊延伸項目 API 時，Lambda 會產生平台延伸項目日誌。平台延伸項目訊息如下列範例所示。

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.extension",
    "record": {"name": "Foo.bar",
        "state": "Ready",
        "events": ["INVOKE", "SHUTDOWN"]
```

```
     }
}
```

當延伸項目訂閱日誌 API 時，Lambda 會產生平台日誌訂閱日誌。日誌訂閱訊息如下列範例所示。

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.logsSubscription",
    "record": {"name": "Foo.bar",
        "state": "Subscribed",
        "types": ["function", "platform"],
    }
}
```

當延伸項目無法處理正在接收的日誌數量時，Lambda 產生的日誌會捨棄平台日誌。`platform.logsDropped` 日誌訊息如下列範例所示。

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.logsDropped",
    "record": {"reason": "Consumer seems to have fallen behind as it has not
 acknowledged receipt of logs.",
        "droppedRecords": 123,
        "droppedBytes" 12345
    }
}
```

platform.restoreStart 日誌訊息會顯示 `Restore` 階段開始的時間 (僅限 `snap-start` 初始化類型)。範例：

```
{
  "time":"2022-07-17T18:43:44.782Z",
  "type":"platform.restoreStart",
  "record":{}
}
```

platform.restoreReport 日誌訊息會顯示 `Restore` 階段持續的時間長度，以及您須為此階段付費的毫秒數 (僅限 `snap-start` 初始化類型)。範例：

```
{
  "time":"2022-07-17T18:43:45.936Z",
```

```
  "type":"platform.restoreReport",
  "record":{
      "metrics":{
          "durationMs":70.87,
          "billedDurationMs":13
      }
  }
}
```

平台 **runtimeDone** 訊息

如果您在訂閱請求中將結構描述版本設定為 "2021-03-18"，在函數調用成功完成或發生錯誤之後，Lambda 會傳送 `platform.runtimeDone` 訊息。延伸項目可以使用此訊息停止此函數調用的所有遙測集合。

結構描述版本 2021-03-18 中的日誌事件類型的 OpenAPI 規範可參閱此文件：schema-2021-03-18.zip

當執行時間傳送 `Next` 或 `Error` 執行時間 API 請求時，Lambda 會產生 `platform.runtimeDone` 日誌訊息。`platform.runtimeDone` 日誌會通知 Logs API 的使用者，告知他們函數調用完成。延伸項目可以使用此資訊來決定何時傳送該調用期間收集的所有遙測。

範例

當函數調用完成時，在執行時間傳送 NEXT 請求之後，Lambda 會傳送 `platform.runtimeDone` 訊息。下列範例顯示每個狀態值的訊息：成功、失敗和逾時。

Example 成功訊息範例

```
{
    "time": "2021-02-04T20:00:05.123Z",
    "type": "platform.runtimeDone",
    "record": {
       "requestId":"6f7f0961f83442118a7af6fe80b88",
       "status": "success"
    }
}
```

Example 失敗訊息範例

```
{
   "time": "2021-02-04T20:00:05.123Z",
   "type": "platform.runtimeDone",
```

```
   "record": {
      "requestId":"6f7f0961f83442118a7af6fe80b88",
      "status": "failure"
   }
}
```

Example 逾時訊息範例

```
{
   "time": "2021-02-04T20:00:05.123Z",
   "type": "platform.runtimeDone",
   "record": {
      "requestId":"6f7f0961f83442118a7af6fe80b88",
      "status": "timeout"
  }
}
```

Example 示例平台。 restoreRuntimeDone 訊息 (僅限**snap-start**初始化類型)

該平台。 restoreRuntimeDone日誌消息顯示Restore階段是否成功。執行階段傳送 restore/next 執行階段 API 請求時，Lambda 會產生此訊息。可能的狀態有三種：成功、失敗和逾時。下面的例子顯示了一個成功的平台。 restoreRuntimeDone日誌消息。

```
{
  "time":"2022-07-17T18:43:45.936Z",
  "type":"platform.restoreRuntimeDone",
  "record":{
      "status":"success"
  }
}
```

針對 Lambda 中的問題進行故障診斷

下列主題提供您在使用 Lambda API、主控台或工具時可能遭遇錯誤和問題的故障診斷建議。如果您發現未列在此處的問題，您可以使用此頁面上的 Feedback (意見回饋) 按鈕來報告。

如需更多故障診段建議和常見支援問題的解答，請瀏覽 AWS 知識中心。

如需有關偵錯和疑難排解 Lambda 應用程式的詳細資訊，請參閱無伺服器園地中的偵錯。

主題

針對 Lambda 中的部署問題進行疑難排解

當您更新函數時，Lambda 會透過啟動包含更新程式碼或設定的函數新執行個體，來部署變更。部署錯誤會導致您無法使用新版本，而造成這類錯誤的可能原因包含您部署套件、程式碼、許可或工具的問題。

當您直接使用 Lambda API 或使用用戶端 (例如) 將更新部署到函數時 AWS CLI，您可以直接在輸出中看到來自 Lambda 的錯誤。如果您使用 AWS CloudFormation、 AWS CodeDeploy或等服務 AWS CodePipeline，請在該服務的日誌或事件串流中尋找 Lambda 的回應。

下列主題提供您在使用 Lambda API、主控台或工具時可能遭遇錯誤和問題的故障診斷建議。如果您發現未列在此處的問題，您可以使用此頁面上的 Feedback (意見回饋) 按鈕來報告。

如需更多故障診段建議和常見支援問題的解答，請瀏覽 AWS 知識中心。

如需有關偵錯和疑難排解 Lambda 應用程式的詳細資訊，請參閱無伺服器園地中的偵錯。

主題

- 一般：未定義的方法處理常式
- Lambda：分層轉換失敗
- Lambda： InvalidParameterValueException或 RequestEntityTooLargeException
- Lambda： InvalidParameterValueException
- Lambda：並行和記憶體配額

一般：許可遭拒/無法載入此類檔案

錯誤：EACCES：拒絕許可，開啟 '/var/task/index.js'

錯誤：無法載入這類檔案 – 函數

錯誤：[Errno 13] 拒絕許可：'/var/task/function.py'

Lambda 執行時間需有許可才能讀取部署套裝服務中的檔案。在 Linux 權限八進制標記法中，Lambda 需要 644 個權限來處理不可執行的檔案 (rw-r--r--)，而目錄和可執行檔需要 755 個權限 ()。rwxr-xr-x

在 Linux 和 MacOS 中，使用 chmod 命令變更部署套件中檔案和目錄的檔案許可。例如，若要提供可執行檔正確的許可，請執行下列命令。

```
chmod 755 <filepath>
```

若要在 Windows 中變更檔案許可，請參閱 Microsoft Windows 文件的 Set, View, Change, or Remove Permissions on an Object。

一般：呼叫時發生錯誤 UpdateFunctionCode

錯誤：調用 UpdateFunctionCode操作時發生錯誤（ RequestEntityTooLargeException ）

當您將部署套裝服務或 Layer 存檔直接上傳至 Lambda 時，ZIP 檔案的大小限制為 50 MB。若要上傳更大的檔案，請將它存放在 Amazon S3 中並使用 S3Bucket 和 S3Key 參數。

> **Note**
>
> 當您使用 AWS CLI、 AWS SDK 或其他方式直接上傳檔案時，二進位 ZIP 檔案會轉換為 base64，這會將檔案的大小增加約 30%。若要允許此操作，以及請求中其他參數的大小，Lambda 套用的實際請求大小限制會更大。因此，50 MB 的限制是概略值。

Amazon S3：錯誤代碼 PermanentRedirect。

錯誤：發生錯誤 GetObject。S3 錯誤代碼： PermanentRedirect。S3 錯誤訊息：儲存貯體位於此區域：us-east-2。請使用此區域重試請求

當您從 Amazon S3 儲存貯體上傳函數的部署套件時，儲存貯體必須位於與函數相同的區域。當您在呼叫時指定 Amazon S3 物件 UpdateFunctionCode，或使用套件並在 AWS CLI 或 AWS SAM CLI 中部署命令時，可能會發生此問題。為開發應用程式的每個區域建立部署成品儲存貯體。

一般：找不到、無法載入、無法匯入、找不到類別、沒有此類檔案或目錄

錯誤： Cannot find module 'function' (找不到 'function' 模組)

錯誤：無法載入這類檔案 – 函數

錯誤： Unable to import module 'function' (無法匯入 'function' 模組)

錯誤： Class not found: function.Handler (找不到類別：function.Handler)

錯誤： fork/exec /var/task/function: no such file or directory (fork/exec /var/task/function：找不到檔案或目錄)

錯誤： Unable to load type 'Function.Handler' from assembly 'Function'. (無法從 'Function' 組件載入 'Function.Handler' 類型。)

您函數處理器組態中的檔案或類別名稱與您的程式碼不相符。如需詳細資訊，請參閱下一節。

一般：未定義的方法處理常式

錯誤： index.handler is undefined or not exported (index.handler 未定義或尚未匯出)

錯誤： Handler 'handler' missing on module 'function' ('function' 模組上找不到 'handler' 處理器)

錯誤：未定義的方法「處理程序」用於 # ：0 LambdaHandler

錯誤： No public method named handleRequest with appropriate method signature found on class function.Handler (在 function.Handler 類別上找不到具備適當方法簽章，名為 handleRequest 的公有方法)

錯誤： Unable to find method 'handleRequest' in type 'Function.Handler' from assembly 'Function' (在來自 'Function' 組件的 'Function.Handler' 類型中找不到 'handleRequest' 方法)

您函數處理器組態中的處理器方法名稱與您的程式碼不相符。每個執行時間都會定義處理器的命名慣例，例如 *filename*.*methodname*。處理器是您函數程式碼中的方法，執行時間會在調用您的函數時執行該方法。

針對某些語言，Lambda 提供了程式庫，其中包含預期處理器方法具備特定名稱的界面。如需每種語言的處理器命名詳細資訊，請參閱下列主題。

- 使用 Node.js 建置 Lambda 函數
- 使用 Python 建置 Lambda 函數
- 使用 Ruby 建置 Lambda 函數
- 使用 Java 建置 Lambda 函數
- 使用 Go 建置 Lambda 函數
- 使用 C# 建置 Lambda 函數
- 使用建置 Lambda 函數 PowerShell

Lambda：分層轉換失敗

錯誤：Lambda 分層轉換失敗。如需有關解決此問題的建議，請參閱《Lambda 使用者指南》中的「Lambda 部署問題疑難排解」頁面。

當您使用分層設定 Lambda 函數，Lambda 會將該分層與函數程式碼合併。如果此程序無法完成，Lambda 便會傳回此錯誤。如果出現此錯誤，請執行下列步驟：

- 從分層中刪除所有未使用的檔案
- 刪除分層中的所有符號連結
- 重新命名任何與函數分層中目錄名稱相同的所有檔案

Lambda： InvalidParameterValueException或 RequestEntityTooLargeException

錯誤:InvalidParameterValueException: Lambda 無法設定您的環境變數，因為您提供的環境變數超過 4KB 的限制。測量的字串:{"A1":" USFE cyPiPn Y5 7 X 5 B...

錯誤:RequestEntityTooLargeException: 請求必須小於 5120 字節的 UpdateFunctionConfiguration 操作

儲存在函數組態中的變數物件大小上限不得超過 4,096 個位元組。這包括金鑰名稱、值、引號、逗號和括號。HTTP 請求主體的大小總計也受到限制。

```
{
    "FunctionName": "my-function",
    "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
    "Runtime": "nodejs20.x",
    "Role": "arn:aws:iam::123456789012:role/lambda-role",
    "Environment": {
        "Variables": {
            "BUCKET": "DOC-EXAMPLE-BUCKET",
            "KEY": "file.txt"
        }
    },
    ...
}
```

在此範例中，物件是 39 個字元，並在其存放為字串 `{"BUCKET":"DOC-EXAMPLE-BUCKET","KEY":"file.txt"}` (不含空格) 時，最多佔用 39 個位元組。環境變數值中每個標準 ASCII 字元使用一個位元組。每個延伸的 ASCII 字元和 Unicode 字元可以使用 2 個位元組到 4 個位元組。

Lambda： InvalidParameterValueException

錯誤:InvalidParameterValueException: Lambda 無法設定您的環境變數，因為您提供的環境變數包含目前不支援修改的保留金鑰。

Lambda 保留一些環境變數金鑰以供內部使用。例如，執行時間使用的 `AWS_REGION` 可決定目前區域，而且不可置換。但是，執行時間使用的其他變數，例如 `PATH`，可在函數組態中擴充。如需完整清單，請參閱定義執行時間環境變數。

Lambda：並行和記憶體配額

錯誤： ConcurrentExecutions 為函數指定將帳戶的 UnreservedConcurrentExecution 低於其最小值

錯誤:'MemorySize' 值無法滿足限制:成員的值必須小於或等於 3008

當您超過帳戶的並行或記憶體配額時，就會發生這些錯誤。新 AWS 帳戶減少了並發性和內存配額。若要解決與並行相關的錯誤，您可以請求提高配額。您無法請求增加記憶體配額。

- 並行：如果您嘗試使用保留或佈建並行建立函數，或者您的每個函式並行要求 (PutFunctionConcurrency) 超過帳戶的並行配額，則可能會收到錯誤訊息。

- 記憶體：如果分配給函數的記憶體數量超過帳戶的記憶體配額，就會發生錯誤。

針對 Lambda 中的調用問題進行疑難排解

當您調用 Lambda 函數時，Lambda 會先驗證請求並檢查擴展容量，再將事件傳送到您的函數，或是事件佇列 (針對非同步調用)。導致調用錯誤的可能原因包含請求參數、事件結構、函數設定、使用者許可、資源許可或限制等問題。

如果您直接調用函數，您會在 Lambda 的回應中看到調用錯誤。如果您透過事件來源映射或是其他服務以非同步方式調用您的函數，您可能會在日誌、無效字母佇列或是失敗事件目的地上找到錯誤。錯誤處理選項和重試行為會因您調用函數的方式，以及錯誤的類型而有所不同。

如需 `Invoke` 操作可以傳回的錯誤類型清單，請參閱調用。

IAM：拉姆達：InvokeFunction 未授權

錯誤：用戶：ARN：aw：IAM：：123456789012：用戶/開發人員未被授權執行：lambda：在資源上：我的函數 InvokeFunction

您的使用者或您擔任的角色必須有調用函數的許可。此要求也適用於 Lambda 函數及其他調用函數的運算資源。將 AWS 受管理的策略添加AWSLambdaRole到您的用戶，或添加允許對目標功能`lambda:InvokeFunction`執行操作的自定義策略。

> **Note**
>
> IAM 動作的名稱 (`lambda:InvokeFunction`) 指的是 `Invoke` Lambda API 操作。

如需詳細資訊，請參閱管理權限 AWS Lambda。

Lambda：找不到有效的引導程序（運行時。 InvalidEntrypoint)

錯誤：找不到有效的引導程序：[/var/task/bootstrap /opt/bootstrap]

當部署套件的根層級不包含名為 `bootstrap` 的可執行檔時，通常會發生此錯誤。例如，如果您使用 .zip 檔案部署 `provided.al2023` 函數，則 `bootstrap` 檔案必須位於 .zip 檔案的根層級，而不在目錄中。

Lambda：無法執行操作 ResourceConflictException

錯誤ResourceConflictException:: 此時無法執行作業。函數量前處於下列狀態：擱置中

當您在建立時將函數連接到 Virtual Private Cloud (VPC) 時，函數會在 Lambda 建立彈性網路界面的同時，進入 Pending 狀態。在此期間，您無法調用或修改函數。如果您在建立後將函數連接到 VPC，則可以在更新擱置時調用該函數，但無法修改其程式碼或組態。

如需詳細資訊，請參閱Lambda 函數狀態。

Lambda：函數卡在待定狀態

錯誤： 函數停留在 *Pending* 狀態幾分鐘的時間。

如果函數卡在 Pending 狀態的時間超過六分鐘，請呼叫下列其中一個 API 操作來解除封鎖：

- UpdateFunctionCode
- UpdateFunctionConfiguration
- PublishVersion

Lambda 會取消待處理的操作並將該函數放入 Failed 狀態。您接著可以嘗試另一個更新。

Lambda：一個函數正在使用所有並行

問題：單一函數正在使用所有可用的並行，造成其他函數遭到調節。

若要將區域中 AWS 帳戶的可用並行分割為集 AWS 區，請使用保留的並行功能。預留並行可確保函數一律會擴展至其受到指派的並行，且函數擴展的並行也不會超過其受到指派的並行。

一般：無法使用其他帳戶或服務調用函數

問題：您可以直接調用函數，但當其他服務或帳戶調用該函數時，它不會執行。

您在函數以資源為基礎的政策中，授予其他服務和帳戶調用函數的許可。如果調用者屬於另一個帳戶，則該使用者也必須具備函數的調用許可。

一般：函數調用正在循環

問題： 在迴圈中連續調用函數。

這通常發生在您的函數管理觸發它的相同 AWS 服務中的資源時。例如，可以建立函數，將物件存放在所設定的 Amazon Simple Storage Service (Amazon S3) 儲存貯體中，該儲存貯體具有再次調用函數的通知。若要停止執行函式，請將可用的並行值減少為零，這會限制所有 future 的呼叫。然後，識別造成遞迴調用的程式碼路徑或組態錯誤。Lambda 會自動偵測並停止某些服 AWS 務和 SDK 的遞迴迴圈。如需詳細資訊，請參閱 the section called “遞迴迴圈偵測”。

Lambda：具有佈建並行的別名路由

問題： 在別名路由期間佈建並行溢出調用。

Lambda 使用簡單的概率模型來在兩個函數版本之間分配流量。在流量較低時，您可能會看到每個版本已設定流量百分比與實際流量百分比之間，存在很大差異。如果您的函數使用佈建並行，透過在別名路由作用期間設定較高數目的已佈建並行執行個體，則可以避免溢出調用。

Lambda：使用佈建並行的冷啟動

問題：啟用佈建的並行後，您會看到冷啟動。

當函數上的並行執行次數少於或等於已設定的佈建並行層級，則不應該發生任何冷啟動。若要協助您確認佈建的並行是否能正常運作，請執行下列動作：

- 請在函數版本或別名上檢查佈建的並行是否啟用。

  > **Note**
  >
  > 函數的未發佈版本 ($LATEST) 無法設定佈建並行。

- 確保您的觸發條件調用的是正確的函數版本或別名。例如，如果您使用的是 Amazon API Gateway，請檢查 API Gateway 調用的函數版本或別名具有佈建的並行，而不是 $LATEST。若要確認正在使用佈建的並行，您可以檢查 ProvisionedConcurrencyInvocations Amazon 指 CloudWatch 標。非零值表示函數正在初始化執行環境上處理調用。
- 檢查指標，判斷您的函數並行是否超過設定的佈建並行層級。ProvisionedConcurrencySpilloverInvocations CloudWatch非零值表示所有已佈建的並行處於使用中的狀態，而某些調用會在冷啟動時發生。
- 檢查調用頻率 (每秒請求數)。具有佈建並行的函數，每個佈建並行的請求速率上限為每秒 10 個。例如，設定為具備 100 個佈建並行的函數每秒可以處理 1,000 個請求。如果調用速率超過每秒 1,000 個請求，就可能會發生冷啟動。

Lambda：新版本的冷啟動

問題：在部署新版的函數時，您會看到冷啟動。

當您更新函數別名時，Lambda 會根據別名上設定的權重，自動將佈建的並行移至新版本。

錯誤：KMSDisabledException：Lambda 無法解密環境變數，因為使用的 KMS 金鑰已停用。請檢查函數的 KMS 金鑰設定。

如果您的 AWS Key Management Service (AWS KMS) 金鑰已停用，或者允許 Lambda 使用金鑰的授權遭到撤銷，就會發生此錯誤。如果授權遺失，請將函數設定為使用不同的金鑰。然後重新分配自訂金鑰以重新建立授權。

EFS：函數無法掛載 EFS 檔案系統

錯誤：EFSMountFailureException：此函數無法使用存取點陣列掛載 EFS 檔案系統：AW：彈性檔案系統：美國東部：123456789012：存取點 /fsap-015cxplb72b40 5fd。

對檔案系統的掛載要求已遭拒。檢查函數的許可，並確認其檔案系統和存取點是否存在，且可供使用。

EFS：函數無法連線到 EFS 檔案系統

錯誤：EFSMountConnectivityException：此功能無法使用存取點陣列連線到 Amazon EFS 檔案系統：AWS：彈性檔案系統：美國東部：123456789012：存取點 /fsap-015cxplb72b40 5fd。請檢查您的網路組態，並再試一次。

函數無法使用 NFS 通訊協定 (TCP 連接埠 2049) 建立函數檔案系統的連線。為 VPC 的子網路檢查安全群組與路由組態。

如果您在更新函數的 VPC 組態設定後看到這些錯誤，請嘗試卸載並重新掛載檔案系統。

EFS：因為逾時，函數無法掛載 EFS 檔案系統

錯誤：EFSMountTimeoutException：由於掛載逾時，此函數無法使用存取點掛載 EFS 檔案系統，而無法使用存取點 {arn: aw: 彈性檔案系統:美國東部:2:123456789012: 存取點 /fsap-015cxplb72b405fd}。

函數可以連線至函數的檔案系統，但掛載操作逾時。稍後再試一次，並考慮限制函數的並行數量，以減少檔案系統的負載。

Lambda：Lambda 偵測到耗時太久的 IO 程序

EFSIOException：函數執行個體已停止，因為 Lambda 偵測到耗時太久的 IO 處理序。

先前的調用逾時，而且 Lambda 無法終止函數處理常式。當附加的檔案系統用完高載額度，且基準輸送量不足時，可能會發生此問題。若要增加輸送量，您可以增加檔案系統的大小，或使用佈建的輸送量。如需詳細資訊，請參閱輸送量。

針對 Lambda 中的執行問題進行疑難排解

當 Lambda 執行時間執行您的函數程式碼時，可能會在已經處理事件一段時間的函數執行個體上處理事件，或是需要初始化新的執行個體。函數初始化期間、您的處理器程式碼處理事件時，或是您的函數傳回回應時 (或是無法傳回時)，都可能會發生錯誤。

造成函數執行錯誤的可能原因包含了您程式碼、函數組態、下游資源或是許可的問題。如果您直接叫用您的函數，您會在 Lambda 的回應中看到函數錯誤。如果您透過事件來源映射或是其他服務以非同步方式叫用您的函數，您可能會在日誌、無效字母佇列或是失敗的目的地上找到錯誤。錯誤處理選項和重試行為會因您叫用函數的方式，以及錯誤的類型而有所不同。

當您的函數程式碼或 Lambda 執行時間傳回錯誤時，Lambda 回應中的狀態碼將會是 200 OK。名為 `X-Amz-Function-Error` 的標頭指示回應中存在錯誤。400 和 500 系列的狀態碼為叫用錯誤預留。

Lambda：執行時間太長

問題：函數執行時間過長。

如果您的程式碼在 Lambda 中執行的時間比在本機機器上長，可能是因為函數可用的記憶體或處理能力受到限制。設定函數使用額外記憶體以同時增加記憶體和 CPU。

Lambda：日誌或追蹤沒有出現

問題：記錄檔不會出現在 CloudWatch 記錄檔中。

問題：追蹤不會出現在中 AWS X-Ray。

您的功能需要許可才能調用 CloudWatch 日誌和 X-Ray。更新其執行角色以授予許可。新增下列受管政策來啟用日誌和追蹤。

- AWSLambdaBasicExecutionRole

- AWSXRayDaemonWriteAccess

當您向函數添加權限時，也對其代碼或配置執行簡單的更新。若您函數的執行中執行個體具有已過期的憑證，上述動作會強制停止並取代這些執行個體。

> **Note**
>
> 在函數調用後，日誌可能需要 5 到 10 分鐘才會顯示。

Lambda：並非所有函數的日誌都會顯示

問題：即使我的權限是正確的，在 CloudWatch 日誌中缺少功能日誌

如果您 AWS 帳戶 達到其CloudWatch 日誌配額限制，請 CloudWatch 節流功能日誌記錄。發生這種情況時，您的函數輸出的某些日誌可能不會出現在 CloudWatch 日誌中。

如果您的函數以過高的速率輸出日誌，Lambda 處理它們，這也可能導致日誌輸出不出現在 CloudWatch 日誌中。當 Lambda 無法以函數產生記錄的速度傳送記錄檔時，會捨棄記錄檔以防止函數的執行速度變慢。 CloudWatch

如果您的函數設定為使用 JSON 格式的記錄，Lambda 會在卸除 CloudWatch 記錄檔時嘗試將`logsDropped`事件傳送至記錄檔。但是，當 CloudWatch 調節函數的日誌記錄時，此事件可能無法到達 CloudWatch 日誌，因此 Lambda 丟棄日誌時不會始終看到記錄。

若要檢查您是否 AWS 帳戶 已達到 CloudWatch 記錄配額限制，請執行下列動作：

1. 開啟 Service Quotas 主控台。
2. 在導覽窗格中，選擇 AWS services (AWS 服務)。
3. 從AWS 服務清單中搜尋 Amazon CloudWatch 日誌。
4. 在 Service Quotas 清單中，選擇 `CreateLogGroup throttle limit in transactions per second`、`CreateLogStream throttle limit in transactions per second` 和 `PutLogEvents throttle limit in transactions per second` 配額，以檢視您的使用率。

您也可以設定 CloudWatch 警示，在您的帳戶使用率超過您為這些配額指定的限制時提醒您。如需詳細資訊，請參閱根據靜態閾值建立 CloudWatch 警示。

如果記 CloudWatch 錄檔的預設配額限制不足以滿足您的使用案例，您可以要求提高配額。

Lambda：該函數在執行完成之前傳回

問題：(Node.js) 函數在程式碼完成執行前傳回

包括 AWS SDK 在內的許多程式庫都會以非同步方式運作。當您進行網路呼叫或是執行需要等待回應的其他操作時，程式庫會傳回稱為 promise 的物件，在背景追蹤操作的進度。

若要等待 promise 解析為回應，請使用 `await` 關鍵字。這會阻止您的處理器在包含回應的 promise 解析為物件前執行。如果您不需要在程式碼中使用回應中的資料，您可以直接將 promise 傳回執行時間。

有些程式庫不會傳回 promise，但是可以包裝在傳回 promise 的程式碼中。如需詳細資訊，請參閱 在 Node.js 中 Lambda 義函數處理常式。

AWS SDK：版本和更新

問題：執行階段中包含的 AWS SDK 不是最新版本

問題：執行階段中包含的 AWS SDK 會自動更新

指令碼語言的執行階段包括 AWS SDK，並會定期更新至最新版本。每個執行時間目前的版本都會列在執行時間頁面上。若要使用較新版本的 AWS SDK，或將函數鎖定到特定版本，您可以將程式庫與函數程式碼結合在一起，或建立 Lambda 層。如需建立包含相依性部署套件的詳細資訊，請參閱下列主題：

Node.js

- 使用 .zip 封存檔部署 Node.js Lambda 函數

Python

- 使用 .zip 封存檔部署 Python Lambda 函數

Ruby

- 使用 Ruby Lambda 函數的 .zip 封存檔

Java

- 使用 .zip 或 JAR 封存檔部署 Java Lambda 函數

Go

- 使用 .zip 封存檔部署 Go Lambda 函數

C#

- 使用 .zip 封存檔建置和部署 C# Lambda 函數

PowerShell

- 使用 .zip 檔案封存部署 PowerShell Lambda 函數

Python：程式庫的載入不正確

問題：(Python) 有些程式庫無法從部署套件正確載入

使用以 C 或 C++ 撰寫延伸模組的程式庫必須在處理器架構與 Lambda 相同的環境中編譯 (Amazon Linux)。如需更多詳細資訊，請參閱 使用 .zip 封存檔部署 Python Lambda 函數。

針對 Lambda 中的聯網問題進行疑難排解

根據預設，Lambda 會在擁有 AWS 服務及網際網路連線能力的內部 virtual private cloud (VPC) 中執行您的函數。若要存取本機網路資源，您可以設定您的函數，使其連接到您帳戶中的 VPC。當您使用此功能時，您可以使用 Amazon Virtual Private Cloud (Amazon VPC) 資源管理函數的網際網路存取能力及網路連線能力。

網路連線錯誤可能是由於 VPC 的路由設定、安全群組規則、 AWS Identity and Access Management (IAM) 角色許可或網路位址轉譯 (NAT) 發生問題，或是 IP 位址或網路介面等資源可用性造成的。視問題而定，如果請求無法到達其目標，您可能會看到特定錯誤或逾時。

VPC：函數會失去網際網路存取或逾時

問題：Lambda 函數在連線到 VPC 之後失去網際網路存取能力。

錯誤：Error: connect ETIMEDOUT 176.32.98.189:443 (錯誤：連線 ETIMEDOUT 176.32.98.189:443)

錯誤：Error: Task timed out after 10.00 seconds (錯誤：任務在 10.00 秒後逾時)

錯誤:ReadTimeoutError: 讀取逾時。 (讀取逾時 = 15)

當您將函數連線到 VPC 時，所有傳出請求都會通過 VPC。若要連線到網際網路，請設定您的 VPC 從函數的子網路將傳出流量傳送到公有子網路中的 NAT 閘道。如需詳細資訊和範例 VPC 組態，請參閱 the section called “VPC 功能的網際網路存取”。

如果某些 TCP 連線逾時，這可能是由於封包分段造成的。Lambda 函數無法處理傳入的分段 TCP 請求，因為 Lambda 不支援 TCP 或 ICMP 的 IP 分段。

VPC：功能需要在不使用互聯網的情況下訪問 AWS 服務

問題：您的 Lambda 函數需要在不使用網際網路的情況下存取 AWS 服務。

若要從沒有網際網路存取的私有子網路將函數連線到 AWS 服務，請使用 VPC 端點。

VPC：已達到彈性網絡介面限制

錯誤：ENILimitReachedException：已達到功能 VPC 的 elastic network interface 限制。

當您將 Lambda 函數連線到 VPC 時，Lambda 會為每個連接到函數的子網路和安全群組組合建立彈性網絡介面。預設服務配額為每個 VPC 250 個網路介面。若要請求提升配額，您可以使用 Service Quotas 主控台。

EC2：具有「lambda」類型的彈性網路介面

錯誤代碼：客戶端。 OperationNotPermitted

錯誤訊息：無法針對此類型的介面修改安全群組

如果您嘗試修改由 Lambda 所管理的彈性網絡介面 (ENI)，將會收到此錯誤訊息。`ModifyNetworkInterfaceAttribute` 不包含在 Lambda 為更新彈性網路介面上的操作所建立的 Lambda API 中。

AWS Lambda 應用

AWS Lambda 應用程式是 Lambda 函數、事件來源和其他一起工作以執行工作的資源的組合。您可以使用 AWS CloudFormation 和其他工具將應用程式的元件收集到單一套件中，以單一資源的方式部署和管理。應用程式可讓您的 Lambda 專案變得可攜式，並可讓您與其他開發人員工具 (例如 AWS CodePipeline AWS CodeBuild、和 AWS Serverless Application Model 命令列介面 (AWS SAM CLI) 整合。

AWS Serverless Application Repository 提供 Lambda 應用程式的集合，只需按幾下滑鼠即可在您的帳戶中部署這些應用程式。存放庫包含 ready-to-use 應用程式和範例，您可以將這些應用程式和範例用作自己專案的起點。您也可以提交自己的專案以納入其中。

AWS CloudFormation 可讓您建立範本，定義應用程式的資源，並可讓您以堆疊來管理應用程式。您可以更安全地新增或修改應用程式堆疊中的資源。如果更新的任何部分失敗， AWS CloudFormation 會自動復原至先前的組態。使用 AWS CloudFormation 參數，您可以從相同的範本為應用程式建立多個環境。 AWS SAM AWS CloudFormation 以專注於 Lambda 應用程式開發的簡化語法進行擴充。

AWS CLI 與 AWS SAM CLI 為命令列工具，可管理 Lambda 應用程式堆疊。除了使用 AWS CloudFormation API 管理應用程式堆疊的命令外，還 AWS CLI 支援更高層級的命令，可簡化工作，例如上傳部署套件和更新範本。 AWS SAM CLI 提供其他功能，包括驗證範本、在本機測試，以及與 CI/CD 系統整合。

建立應用程式時，您可以使用 CodeCommit 或 AWS CodeStar 連線來建立其 Git 儲存庫 GitHub。 CodeCommit 可讓您使用 IAM 主控台管理使用者的安全殼層金鑰和 HTTP 登入資料。CodeConnections 使您能夠連接到您的 GitHub 帳戶。如需連線的詳細資訊，請參閱開發人員工具主控台使用者指南中的什麼是連線？

如需設計 Lambda 應用程式的詳細資訊，請參閱無伺服器園地中的應用程式設計。

主題

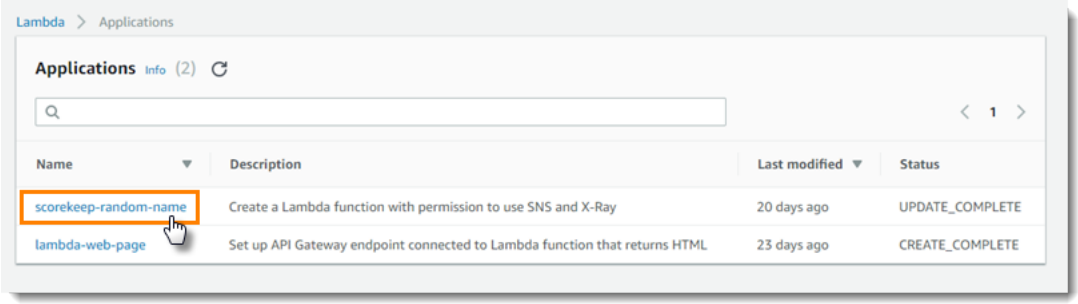
- 在 AWS Lambda 主控台中管理應用程式
- 為 Lambda 函數建立滾動式部署
- 搭配 Kubernetes 使用 Lambda

在 AWS Lambda 主控台中管理應用程式

AWS Lambda 主控台可協助您監控和管理您的 Lambda 應用程式。Applications (應用程式) 選單可列出具有 Lambda 函數的 AWS CloudFormation 堆疊。此功能表包含的堆疊包括您使用 AWS CloudFormation 主控台、AWS Serverless Application Repository、AWS CLI 或 AWS SAM CLI 在 AWS CloudFormation 中啟動的堆疊。

若要檢視 Lambda 應用程式

1. 開啟 Lambda 主控台中的 Applications (應用程式) 頁面。
2. 選擇應用程式。



此概觀說明有關您應用程式的以下資訊。

- AWS CloudFormation 範本或 SAM 範本 - 定義應用程式的範本。
- Resources (資源) - 在您的應用程式範本中定義的 AWS 資源。若要管理應用程式的 Lambda 函數，請在清單中選擇函數名稱。

監控應用程式

監控索引標籤會顯示 Amazon CloudWatch 儀表板，其中包含應用程式中資源的彙總指標。

若要監控 Lambda 應用程式

1. 開啟 Lambda 主控台中的 Applications (應用程式) 頁面。
2. 選擇 Monitoring (監控)。

在預設情況下，Lambda 主控台會顯示基本儀表板。您可以在應用程式範本中定義自訂儀表板以自訂此頁面。當您的範本包含一或多個儀表板時，此頁面會顯示您的儀表板，而非預設的儀表板。您可以使用頁面右上角的下拉式功能表切換不同的儀表板。

自訂監控儀表板

透過將一個或多個具有AWS::CloudWatch::Dashboard資源類型的 Amazon CloudWatch 儀表板新增至應用程式範本，以自訂您的應用程式監控頁面。以下範例以單一小工具建立儀表板，此工具可繪製呈現 `my-function` 函數叫用次數的圖形。

Example 函數儀表板範本

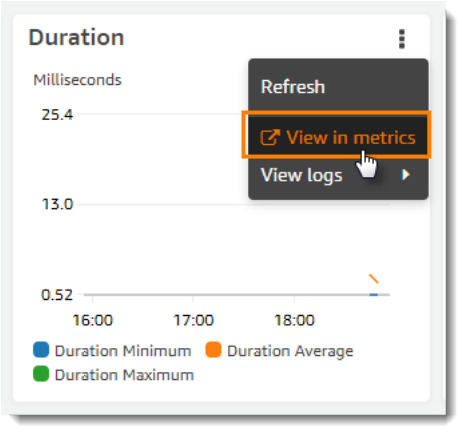
```
Resources:
  MyDashboard:
    Type: AWS::CloudWatch::Dashboard
    Properties:
      DashboardName: my-dashboard
      DashboardBody: |
        {
            "widgets": [
                {
                    "type": "metric",
                    "width": 12,
                    "height": 6,
                    "properties": {
                        "metrics": [
                            [
                                "AWS/Lambda",
                                "Invocations",
                                "FunctionName",
                                "my-function",
                                {
                                    "stat": "Sum",
                                    "label": "MyFunction"
                                }
                            ],
                            [
                                {
                                    "expression": "SUM(METRICS())",
                                    "label": "Total Invocations"
                                }
                            ]
                        ],
                        "region": "us-east-1",
                        "title": "Invocations",
                        "view": "timeSeries",
                        "stacked": false
```

```
                }
            }
        ]
    }
```

您可以從 CloudWatch 主控台取得預設監控儀表板中任何小工具的定義。

檢視小工具定義

1. 開啟 Lambda 主控台中的 Applications (應用程式) 頁面。
2. 選擇具有標準儀表板的應用程式。
3. 選擇 Monitoring (監控)。
4. 在任何小工具上，從下拉式功能表中選擇 View in metrics (檢視指標)。



5. 選擇 Source (來源)。

如需有關編寫 CloudWatch 儀表板和小器具的詳細資訊，請參閱 Amazon CloudWatch API 參考中的儀表板主體結構和語法。

為 Lambda 函數建立滾動式部署

使用滾動部署以控制與引入新版本 Lambda 函數的相關風險。在滾動部署中，系統會自動部署新版本的函數，並逐漸將增加的流量傳送到新版本。流量和增加速率是您可以設定的參數。

您可以使用 AWS CodeDeploy 和來設定滾動式部署 AWS SAM。 CodeDeploy 是一項服務，可將應用程式部署到亞馬遜運算平台 (例如 Amazon EC2 和 AWS Lambda. 如需詳細資訊，請參閱什麼是 CodeDeploy？ 。透過使 CodeDeploy 用部署 Lambda 函數，您可以輕鬆監控部署狀態，並在偵測到任何問題時啟動復原。

AWS SAM 是建置無伺服器應用程式的開放原始碼架構。您可以建立 AWS SAM 範本 (YAML 格式)，以指定滾動式部署所需元件的組態。 AWS SAM 使用範本來建立和設定元件。如需詳細資訊，請參閱什麼是 AWS SAM？。

在滾動式部署中， AWS SAM 執行下列工作：

- 這會配置您的 Lambda 函數並建立別名。

  別名路由組態是實作滾動部署的基本功能。
- 它創建一個 CodeDeploy 應用程序和部署組。

  部署群組會管理滾動部署和轉返 (如有需要)。
- 這會在您建立 Lambda 函數的新版本時進行偵測。
- 它會觸 CodeDeploy 發啟動新版本的部署。

範例 AWS SAM Lambda 範本

下列範例顯示 AWS SAM 範本，以進行簡單的滾動部署。

```
AWSTemplateFormatVersion : '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: A sample SAM template for deploying Lambda functions.

Resources:
# Details about the myDateTimeFunction Lambda function
  myDateTimeFunction:
    Type: AWS::Serverless::Function
    Properties:
```

```
      Handler: myDateTimeFunction.handler
      Runtime: nodejs18.x
# Creates an alias named "live" for the function, and automatically publishes when you
 update the function.
      AutoPublishAlias: live
      DeploymentPreference:
# Specifies the deployment configuration
          Type: Linear10PercentEvery2Minutes
```

此範本會定義名為 myDateTimeFunction 的 Lambda 函數，且具備下列屬性。

AutoPublishAlias

AutoPublishAlias 屬性會建立名為 live 的別名。此外，當您為該函數儲存新代碼時， AWS SAM 架構會自動偵測。之後，架構會發佈一個新的函數版本並更新 live 別名以指向新版本。

DeploymentPreference

DeploymentPreference屬性決定 CodeDeploy 應用程式將流量從原始版本的 Lambda 函數轉移到新版本的速率。值 Linear10PercentEvery2Minutes 每隔兩分鐘會將額外 10% 的流量轉移到新版本。

如需預先定義的部署組態清單，請參閱部署組態。

如需如何 CodeDeploy 搭配 Lambda 函數使用的詳細教學課程，請參閱使用部署更新的 Lambda 函數 CodeDeploy。

搭配 Kubernetes 使用 Lambda

您可以使用 Kubernetes 專用 AWS 控制器 (ACK) 或 Crossplane，透過 Kubernetes API 來部署和管理 Lambda 函數。

Kubernetes 專用 AWS 控制器 (ACK)

您可以使用 ACK，從 Kubernetes API 部署和管理 AWS 資源。通過 ACK，AWS為 Lambda，Amazon Elastic Container Registry (Amazon ECR)，Amazon 簡單存儲服務（Amazon S3）和亞馬遜等服務提供開源自定義控制器。AWS SageMaker每個支援的 AWS 服務都有專屬自訂控制器。在您的 Kubernetes 叢集中，為每個要使用的 AWS 服務安裝控制器。接著建立自訂資源定義 (CRD)以定義 AWS 資源。

建議您使用 Helm 3.8 或更新版本，安裝 ACK 控制器。每個 ACK 控制器都有專屬 Helm Chart，這會安裝控制器、CRD 和 Kubernetes RBAC 規則。如需詳細資訊，請參閱 ACK 文件中的 Install an ACK Controller。

建立 ACK 自訂資源之後，您可以像使用任何其他內建 Kubernetes 物件一樣加以使用。例如，您可以使用偏好的 Kubernetes 工具鏈 (包括 kubectl)，部署和管理 Lambda 函數。

以下是一些透過 ACK 佈建 Lambda 函數的範例使用案例：

- 您的組織使用角色型存取控制 (RBAC) 和服務帳戶的 IAM 角色，建立許可界限。您可以使用 ACK，為 Lambda 重複使用此安全模型，不須建立新的使用者和政策。
- 您的組織具有使用 Kubernetes 資訊清單將資源部署到 Amazon Elastic Kubernetes Service (Amazon EKS) 叢集的 DevOps 程序。您可以透過 ACK，使用清單檔案佈建 Lambda 函數，不須建立另外的基礎設施即程式碼範本。

如需使用 ACK 的詳細資訊，請參閱 ACK 文件中的 Lambda 教學課程。

Crossplane

Crossplane 是開放原始碼雲端原生運算基金會 (CNCF) 專案，使用 Kubernetes 來管理雲端基礎設施資源。開發人員可以透過 Crossplane 請求基礎設施，不須了解其複雜性。平台團隊保留基礎設施佈建和管理方式的控制權。

您可以使用 Crossplane，透過偏好的 Kubernetes 工具鏈 (例如 kubectl) 以及任何可將清單檔案部署到 Kubernetes 的 CI/CD 管道，部署和管理 Lambda 函數。以下是一些透過 Crossplane 佈建 Lambda 函數的範例使用案例：

- 您的組織希望確保 Lambda 函數具有正確標籤來強制遵循法規。平台團隊可以使用 Crossplane 組合，透過 API 抽象來定義此政策。開發人員隨後可以使用這些抽象來部署具標籤的 Lambda 函數。
- 您的專案會 GitOps 搭配使用 Kubernetes。在此模型中，Kubernetes 會持續協調 git 儲存庫 (所需狀態) 與叢集內執行的資源 (目前狀態)。如果有差異， GitOps 程序會自動對叢集進行變更。您可以 GitOps 搭配 Kubernetes 使用，透過跨平台部署和管理 Lambda 函數，並使用熟悉的 Kubernetes 工具和概念 (例如 CRD 和控制器) 來部署和管理 Lambda 函數。

若要進一步了解如何搭配 Lambda 使用 Crossplane，請參閱下列內容：

- AWS Blueprints for Crossplane：此儲存庫包括如何使用 Crossplane 部署 AWS 資源 (包括 Lambda 函數) 的範例。

> **Note**
>
> 目前正在積極開發 AWS Blueprints for Crossplane，因此不得用於生產環境。

- 使用 Amazon EKS 和 Crossplane 部署 Lambda：此影片展示透過 Crossplane 部署 AWS 無伺服器架構的進階範例，並從開發人員和平台的角度探索設計。

Lambda 範例應用程式

本指南的 GitHub 儲存庫包含示範如何使用各種語言和 AWS 服務的範例應用程式。每個範例應用程式都包含可輕鬆部署和清理的指令碼、 AWS SAM 範本和支援資源。

Node.js

以 Node.js 編寫的範例 Lambda 應用程式

- 空白 nodejs-一個 Node.js 函數，顯示日誌記錄，環境變量， AWS X-Ray 跟踪，圖層，單元測試和 SDK 的使用。 AWS
- nodejs-apig - 具有公有 API 端點的函數，它會處理來自 API Gateway 的事件並傳回 HTTP 回應。
- efs-nodejs - 在 Amazon VPC 中使用 Amazon EFS 檔案系統的函數。此範例包含設為與 Lambda 搭配使用的 VPC、檔案系統、掛載目標以及存取點。

Python

以 Python 編寫的範例 Lambda 應用程式

- 空白蟒蛇- 一個 Python 函數，顯示日誌記錄，環境變量， AWS X-Ray 跟踪，圖層，單元測試和 SDK 的使用。 AWS

Ruby

以 Ruby 編寫的範例 Lambda 應用程式

- 空白紅寶石 — 一個 Ruby 函數，顯示日誌記錄，環境變量， AWS X-Ray 跟踪，圖層，單元測試和 SDK 的使用。 AWS
- 適用於 AWS Lambda 的 Ruby 程式碼範例 — 以 Ruby 撰寫的程式碼範例，示範如何與 AWS Lambda 互動。

Java

以 Java 編寫的範例 Lambda 應用程式

- java17-examples – 一個 Java 函數，示範如何使用 Java 記錄來表示輸入事件資料物件。

- java-basic - 具有單元測試和變數日誌組態的最小 Java 函數集合。
- java-events - Java 函數集合，其中包含如何處理來自各種服務 (例如 Amazon API Gateway、Amazon SQS 和 Amazon Kinesis) 事件的骨架程式碼。這些函數使用最新版 aws-lambda-java-events 程式庫 (3.0.0 及更新版)。這些範例不需要 AWS SDK 作為相依性。
- s3-java - 一種 Java 函數，它處理來自 Amazon S3 的通知事件，並使用 Java Class Library (JCL) 以從上傳的映像檔案建立縮圖。
- 使用 API Gateway 調用 Lambda 函數 - 一個 Java 函數，其可掃描包含員工資訊的 Amazon DynamoDB 資料表。然後，其會使用 Amazon Simple Notification Service 向員工傳送文字訊息，慶祝他們的工作週年紀念日。此範例使用 API Gateway 調用函數。

在 Lambda 上執行熱門 Java 框架

- 彈簧雲函數示例- 來自 Spring 的一個示例，演示瞭如何使用 S pring 雲函數框架來創建 Lambda 函數。 AWS
- 無伺服器 Spring Boot 應用程式示範 — 示範如何在受管理的 Java 執行階段中設定典型的 Spring Boot 應用程式 SnapStart，不論是否使用自訂執行階段，或設定為 GraalVM 原生映像檔的範例。
- 無伺服器微型應用程式示範 — 示範如何在受管理的 Java 執行階段中使用 Micronaut 的範例 SnapStart，或是使用自訂執行階段的 GraalVM 原生映像檔。請參閱《Micronaut/Lambda 指南》以進一步瞭解。
- 無伺服器 Quarkus 應用程式示範 — 示範如何在受管理的 Java 執行階段中使用 Quarkus SnapStart，或是使用自訂執行階段作為 GraalVM 原生映像檔的範例。若要深入了解，請參閱「夸克斯/Lambda」指南和「夸克斯/指南」。SnapStart

Go

Lambda 為 Go 執行時間提供下列範例應用程式：

以 Go 編寫的範例 Lambda 應用程式

- go-al2：傳回公有 IP 地址的「hello world」函數。此應用程式使用 `provided.al2` 自訂執行期。
- 空白移動 — Go 函數，顯示 Lambda 的 Go 程式庫、記錄、環境變數和 SDK 的使用方式 AWS 。此應用程式使用 `go1.x` 執行期。

C#

以 C# 編寫的範例 Lambda 應用程式

- blank-csharp - 一種 C# 函數，它示範如何使用 Lambda 的 .NET 程式庫、記錄、環境變數、AWS X-Ray 追蹤、單元測試和 AWS 開發套件。
- blank-csharp-with-layer – C# 函數，使用 .NET CLI 建立封裝函數相依項的層。
- ec2-spot – 在 Amazon EC2 中管理 Spot 執行個體請求的函數。

PowerShell

Lambda 提供下列範例應用程式 PowerShell：

- 空白電源外殼-顯示使用日誌記錄，環境變量和 SDK 的 PowerShell 函數。 AWS

若要部署範例應用程式，請依照其 README 檔案中的指示。若要進一步了解應用程式架構和使用案例，請閱讀本章各主題。

主題

- 空白功能示例應用 AWS Lambda

空白功能示例應用 AWS Lambda

Blank 函數範例應用程式利用一個呼叫 Lambda API 的函數來示範 Lambda 中的一般操作。它顯示了日誌記錄，環境變量， AWS X-Ray 跟踪，層，單元測試和 AWS SDK 的使用。探索此應用程式可了解如何使用您的程式設計語言建置 Lambda 函數，或以它做為自有專案的起點。

此範例應用程式有下列幾種語言的變體：

變體

- Node.js – blank-nodejs。
- Python – blank-python。
- Ruby – blank-ruby。
- Java – blank-java。
- Go – blank-go。
- C# – blank-csharp。
- PowerShell — 空白電源外殼。

本主題中的範例聚焦於 Node.js 版本的程式碼，但詳細資訊通常適用於所有變體。

您可以使用 AWS CLI 和在幾分鐘內部署範例 AWS CloudFormation。依照 README 中的指示進行下載、設定，以及在您的帳戶中部署。

章節

- 架構和處理常式程式碼
- 使用 AWS CloudFormation 和的部署自動化 AWS CLI
- 儀器與 AWS X-Ray
- 使用 Layer 的相依性管理

架構和處理常式程式碼

範例應用程式包含函式程式碼、 AWS CloudFormation 範本和支援資源。當您部署範例時，您會使用下列 AWS 服務：

- AWS Lambda — 運行函數代碼，將日誌發送到 CloudWatch 日誌，並將跟踪數據發送到 X-Ray。函數也會呼叫 Lambda API，以取得有關目前區域中帳戶配額和用量的詳細資訊。

- AWS X-Ray - 收集追蹤資料、編製追蹤索引以供搜尋，以及產生服務映射。
- Amazon CloudWatch — 存儲日誌和指標。
- AWS Identity and Access Management (IAM) — 授予許可權。
- Amazon Simple Storage Service (Amazon S3)— 在部署期間存放函數的部署套件。
- AWS CloudFormation - 建立應用程式資源及部署函數程式碼。

每項服務均需收取標準費用。如需詳細資訊，請參閱 AWS 定價。

函數程式碼顯示處理事件的基本工作流程。處理常式接受一個 Amazon Simple Queue Service (Amazon SQS) 事件作為輸入，並逐一處理事件內含的記錄，記錄每個訊息的內容。它會記錄事件、內容物件和環境變數的內容。然後，它會使用 AWS SDK 進行呼叫，並將回應傳送回 Lambda 執行階段。

Example blank-nodejs/function/index.js - 處理常式程式碼

```
// Handler
exports.handler = async function(event, context) {
    event.Records.forEach(record => {
        console.log(record.body);
    });

    console.log('## ENVIRONMENT VARIABLES: ' + serialize(process.env));
    console.log('## CONTEXT: ' + serialize(context));
    console.log('## EVENT: ' + serialize(event));

    return getAccountSettings();
};

// Use SDK client
var getAccountSettings = function() {
    return lambda.send(new GetAccountSettingsCommand());
};

var serialize = function(object) {
    return JSON.stringify(object, null, 2);
};
```

範例應用程式不包含傳送事件的 Amazon SQS 佇列，但會使用來自 Amazon SQS (event.json) 的事件來說明事件的處理方式。若要將 Amazon SQS 佇列新增至您的應用程式，請參閱 搭配 Amazon SQS 使用 Lambda。

使用 AWS CloudFormation 和的部署自動化 AWS CLI

範例應用程式的資源是在 AWS CloudFormation 範本中定義，並使用 AWS CLI. 專案包含簡單的 Shell 指令碼，可將應用程式的設定、部署、叫用和拆卸程序自動化。

應用程式範本使用 AWS Serverless Application Model (AWS SAM) 資源類型來定義模型。 AWS SAM 透過自動化定義執行角色、API 和其他資源，簡化無伺服器應用程式的範本建立作業。

範本會定義應用程式堆疊中的資源。這包括函數、其執行角色，以及提供函數程式庫依賴項的 Lambda 層。堆疊不包含部署期間所 AWS CLI 使用的值區或記 CloudWatch 錄檔群組。

Example blank-nodejs/template.yml - 無伺服器資源

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Description: An AWS Lambda application that calls the Lambda API.
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Handler: index.handler
      Runtime: nodejs20.x
      CodeUri: function/.
      Description: Call the AWS Lambda API
      Timeout: 10
      # Function's execution role
      Policies:
        - AWSLambdaBasicExecutionRole
        - AWSLambda_ReadOnlyAccess
        - AWSXrayWriteOnlyAccess
      Tracing: Active
      Layers:
        - !Ref libs
  libs:
    Type: AWS::Serverless::LayerVersion
    Properties:
      LayerName: blank-nodejs-lib
      Description: Dependencies for the blank sample app.
      ContentUri: lib/.
      CompatibleRuntimes:
        - nodejs20.x
```

當您部署應用程式時，會 AWS SAM 將轉換 AWS CloudFormation 套用至範本，以產生具有標準類型 (例如AWS::Lambda::Function和) 的 AWS CloudFormation 範本AWS::IAM::Role。

Example 已處理範本

```
{
  "AWSTemplateFormatVersion": "2010-09-09",
  "Description": "An AWS Lambda application that calls the Lambda API.",
  "Resources": {
    "function": {
      "Type": "AWS::Lambda::Function",
      "Properties": {
        "Layers": [
          {
            "Ref": "libs32xmpl61b2"
          }
        ],
        "TracingConfig": {
          "Mode": "Active"
        },
        "Code": {
          "S3Bucket": "lambda-artifacts-6b000xmpl1e9bf2a",
          "S3Key": "3d3axmpl473d249d039d2d7a37512db3"
        },
        "Description": "Call the AWS Lambda API",
        "Tags": [
          {
            "Value": "SAM",
            "Key": "lambda:createdBy"
          }
        ],
```

在此範例中，Code 屬性指定 Amazon S3 儲存貯體中的物件。這一項對應到專案範本中 CodeUri 屬性的本機路徑：

```
      CodeUri: function/.
```

為了將專案檔案上傳至 Amazon S3，部署指令碼應使用 AWS CLI中的命令。cloudformation package 命令預先處理範本、上傳成品，並以 Amazon S3 物件位置取代本機路徑。指cloudformation deploy令會使用 AWS CloudFormation 變更集部署已處理的範本。

Example blank-nodejs/3-deploy.sh - 封裝與部署

```
#!/bin/bash
set -eo pipefail
ARTIFACT_BUCKET=$(cat bucket-name.txt)
aws cloudformation package --template-file template.yml --s3-bucket $ARTIFACT_BUCKET --
output-template-file out.yml
aws cloudformation deploy --template-file out.yml --stack-name blank-nodejs --
capabilities CAPABILITY_NAMED_IAM
```

第一次執行此指令碼時，會建立名為的 AWS CloudFormation 堆疊`blank-nodejs`。如果您變更函數程式碼或範本，您可以再次執行它以更新堆疊。

清除指令碼 (blank-nodejs/5-cleanup.sh) 會刪除堆疊，並選擇性地刪除部署儲存貯體和函數日誌。

儀器與 AWS X-Ray

範例函數已設定為可使用 AWS X-Ray 進行追蹤。若將追蹤模式設定為作用中，Lambda 會記錄叫用子集的計時資訊，並將其傳送至 X-Ray。X-Ray 處理資料以產生服務映射，顯示用戶端節點和兩個服務節點。

第一個服務節點 (`AWS::Lambda`) 代表 Lambda 服務，它會驗證叫用請求並將其傳送給函數。第二個節點 `AWS::Lambda::Function` 代表函數本身。

為了記錄其他詳細資訊，範例函數使用 X-Ray 開發套件。只要對函數程式碼進行最少的變更，X-Ray SDK 就會記錄使用 AWS SDK 對 AWS 服務進行的呼叫的詳細資訊。

Example blank-nodejs/function/index.js – 檢測

```
const AWSXRay = require('aws-xray-sdk-core');
const { LambdaClient, GetAccountSettingsCommand } = require('@aws-sdk/client-lambda');

// Create client outside of handler to reuse
const lambda = AWSXRay.captureAWSv3Client(new LambdaClient());
```

檢測 AWS SDK 用戶端會在服務對應中新增一個額外的節點，並在追蹤中加入更多詳細資訊。在此範例中，服務映射所顯示的範例函數會呼叫 Lambda API 以取得目前區域中儲存體和並行用量的詳細資訊。

追蹤會顯示呼叫的計時詳細資訊，針對函數初始化、呼叫和額外負荷各有不同子區段。呼叫子區段具有 AWS SDK 呼叫 API 作業的子區段。`GetAccountSettings`

您可以將 X-Ray 開發套件和其他程式庫包含在函數的部署套件中，或將它們分別部署在 Lambda 層。對於 Node.js、紅寶石和 Python，Lambda 執行階段會在執行環境中包含 AWS 開發套件。

使用 Layer 的相依性管理

您可以在本機安裝程式庫，並將它們包含在您上傳到 Lambda 的部署套件中，但這有其缺點。較大型的檔案會導致部署時間增加，並且可能令您無法在 Lambda 主控台中測試函數程式碼的變更。為了保持小部署套件，並避免上傳未變更的相依性，範例應用程式會建立一個 Lambda 層，並將其與函數建立關聯。

Example blank-nodejs/template.yml - 相依性層

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Handler: index.handler
      Runtime: nodejs20.x
      CodeUri: function/.
      Description: Call the AWS Lambda API
      Timeout: 10
      # Function's execution role
      Policies:
        - AWSLambdaBasicExecutionRole
        - AWSLambda_ReadOnlyAccess
        - AWSXrayWriteOnlyAccess
      Tracing: Active
      Layers:
        - !Ref libs
  libs:
    Type: AWS::Serverless::LayerVersion
    Properties:
      LayerName: blank-nodejs-lib
      Description: Dependencies for the blank sample app.
      ContentUri: lib/.
      CompatibleRuntimes:
        - nodejs20.x
```

2-build-layer.sh 指令碼使用 npm 來安裝函數的相依性，並將它們放置在具有 Lambda 執行時間所需結構的資料夾中。

Example 2-build-layer.sh - 準備該層

```
#!/bin/bash
set -eo pipefail
mkdir -p lib/nodejs
rm -rf node_modules lib/nodejs/node_modules
npm install --production
mv node_modules lib/nodejs/
```

第一次部署範例應用程式時，會將層與函數程式碼分開 AWS CLI 封裝，並同時部署兩者。若是後續部署，則只有當 `lib` 資料夾內容已變更時，才會上傳 Layer 存檔。

搭配 AWS 開發套件使用 Lambda

AWS 軟件開發套件（SDK）可用於許多流行的編程語言。每個 SDK 都提供 API、程式碼範例和說明文件，讓開發人員能夠更輕鬆地以偏好的語言建置應用程式。

SDK 文件	代碼範例
AWS SDK for C++	AWS SDK for C++ 程式碼範例
AWS CLI	AWS CLI 程式碼範例
AWS SDK for Go	AWS SDK for Go 程式碼範例
AWS SDK for Java	AWS SDK for Java 程式碼範例
AWS SDK for JavaScript	AWS SDK for JavaScript 程式碼範例
適用於 Kotlin 的 AWS SDK	適用於 Kotlin 的 AWS SDK 程式碼範例
AWS SDK for .NET	AWS SDK for .NET 程式碼範例
AWS SDK for PHP	AWS SDK for PHP 程式碼範例
AWS Tools for PowerShell	PowerShell 程式碼範例的工具
AWS SDK for Python (Boto3)	AWS SDK for Python (Boto3) 程式碼範例
AWS SDK for Ruby	AWS SDK for Ruby 程式碼範例
適用於 Rust 的 AWS SDK	適用於 Rust 的 AWS SDK 程式碼範例
適用於 SAP ABAP 的 AWS SDK	適用於 SAP ABAP 的 AWS SDK 程式碼範例
適用於 Swift 的 AWS SDK	適用於 Swift 的 AWS SDK 程式碼範例

如需 Lambda 專屬範例，請參閱 使用 AWS 開發套件的 Lambda 程式碼範例。

可用性範例

找不到所需的內容嗎？ 請使用本頁面底部的提供意見回饋連結申請程式碼範例。

使用 AWS 開發套件的 Lambda 程式碼範例

下列程式碼範例說明如何搭配 AWS 軟體開發套件 (SDK) 使用 Lambda。

Actions 是大型程式的程式碼摘錄，必須在內容中執行。雖然動作會告訴您如何呼叫個別服務函數，但您可以在其相關情境和跨服務範例中查看內容中的動作。

Scenarios (案例) 是向您展示如何呼叫相同服務中的多個函數來完成特定任務的程式碼範例。

Cross-service examples (跨服務範例) 是跨多個 AWS 服務執行的應用程式範例。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含入門相關資訊和舊版 SDK 的詳細資訊。

開始使用

Hello Lambda

下列程式碼範例示範如何開始使用 Lambda。

.NET

AWS SDK for .NET

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
namespace LambdaActions;

using Amazon.Lambda;

public class HelloLambda
{
    static async Task Main(string[] args)
    {
        var lambdaClient = new AmazonLambdaClient();

        Console.WriteLine("Hello AWS Lambda");
```

```
        Console.WriteLine("Let's get started with AWS Lambda by listing your
 existing Lambda functions:");

        var response = await lambdaClient.ListFunctionsAsync();
        response.Functions.ForEach(function =>
        {

 Console.WriteLine($"{function.FunctionName}\t{function.Description}");
        });
    }
}
```

- 如需 API 詳細資訊，請參閱 AWS SDK for .NET API 參考ListFunctions中的。

C++

適用於 C++ 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

C MakeLists .txt 的 CMake 文件的代碼。

```
# Set the minimum required version of CMake for this project.
cmake_minimum_required(VERSION 3.13)

# Set the AWS service components used by this project.
set(SERVICE_COMPONENTS lambda)

# Set this project's name.
project("hello_lambda")

# Set the C++ standard to use to build this target.
# At least C++ 11 is required for the AWS SDK for C++.
set(CMAKE_CXX_STANDARD 11)

# Use the MSVC variable to determine if this is a Windows build.
```

```
set(WINDOWS_BUILD ${MSVC})

if (WINDOWS_BUILD) # Set the location where CMake can find the installed
 libraries for the AWS SDK.
    string(REPLACE ";" "/aws-cpp-sdk-all;" SYSTEM_MODULE_PATH
 "${CMAKE_SYSTEM_PREFIX_PATH}/aws-cpp-sdk-all")
    list(APPEND CMAKE_PREFIX_PATH ${SYSTEM_MODULE_PATH})
endif ()

# Find the AWS SDK for C++ package.
find_package(AWSSDK REQUIRED COMPONENTS ${SERVICE_COMPONENTS})

if (WINDOWS_BUILD AND AWSSDK_INSTALL_AS_SHARED_LIBS)
     # Copy relevant AWS SDK for C++ libraries into the current binary directory
 for running and debugging.

     # set(BIN_SUB_DIR "/Debug") # if you are building from the command line you
 may need to uncomment this
                                     # and set the proper subdirectory to the
 executables' location.

     AWSSDK_CPY_DYN_LIBS(SERVICE_COMPONENTS ""
 ${CMAKE_CURRENT_BINARY_DIR}${BIN_SUB_DIR})
endif ()

add_executable(${PROJECT_NAME}
        hello_lambda.cpp)

target_link_libraries(${PROJECT_NAME}
        ${AWSSDK_LINK_LIBRARIES})
```

hello_lambda.cpp 來源檔案的程式碼。

```
#include <aws/core/Aws.h>
#include <aws/lambda/LambdaClient.h>
#include <aws/lambda/model/ListFunctionsRequest.h>
#include <iostream>

/*
 *  A "Hello Lambda" starter application which initializes an AWS Lambda (Lambda)
 client and lists the Lambda functions.
 *
```

```
 *  main function
 *
 *  Usage: 'hello_lambda'
 *
 */

int main(int argc, char **argv) {
    Aws::SDKOptions options;
    // Optionally change the log level for debugging.
//   options.loggingOptions.logLevel = Utils::Logging::LogLevel::Debug;
    Aws::InitAPI(options); // Should only be called once.
    int result = 0;
    {
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region (overrides config file).
        // clientConfig.region = "us-east-1";

        Aws::Lambda::LambdaClient lambdaClient(clientConfig);
        std::vector<Aws::String> functions;
        Aws::String marker; // Used for pagination.

        do {
            Aws::Lambda::Model::ListFunctionsRequest request;
            if (!marker.empty()) {
                request.SetMarker(marker);
            }

            Aws::Lambda::Model::ListFunctionsOutcome outcome =
 lambdaClient.ListFunctions(
                    request);

            if (outcome.IsSuccess()) {
                const Aws::Lambda::Model::ListFunctionsResult
 &listFunctionsResult = outcome.GetResult();
                std::cout << listFunctionsResult.GetFunctions().size()
                          << " lambda functions were retrieved." << std::endl;

                for (const Aws::Lambda::Model::FunctionConfiguration
 &functionConfiguration: listFunctionsResult.GetFunctions()) {
                    functions.push_back(functionConfiguration.GetFunctionName());
                    std::cout << functions.size() << "  "
                              << functionConfiguration.GetDescription() <<
 std::endl;
                    std::cout << "   "
```

```
                                <<
 Aws::Lambda::Model::RuntimeMapper::GetNameForRuntime(
                                        functionConfiguration.GetRuntime()) << ": "
                                << functionConfiguration.GetHandler()
                                << std::endl;
                }
                marker = listFunctionsResult.GetNextMarker();
            } else {
                std::cerr << "Error with Lambda::ListFunctions. "
                          << outcome.GetError().GetMessage()
                          << std::endl;
                result = 1;
                break;
            }
        } while (!marker.empty());
    }


    Aws::ShutdownAPI(options); // Should only be called once.
    return result;
}
```

- 如需 API 詳細資訊，請參閱 AWS SDK for C++ API 參考ListFunctions中的。

Go

SDK for Go V2

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
package main

import (
 "context"
 "fmt"
```

```
 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
)

// main uses the AWS SDK for Go (v2) to create an AWS Lambda client and list up
 to 10
// functions in your account.
// This example uses the default settings specified in your shared credentials
// and config files.
func main() {
 sdkConfig, err := config.LoadDefaultConfig(context.TODO())
 if err != nil {
  fmt.Println("Couldn't load default configuration. Have you set up your AWS
 account?")
  fmt.Println(err)
  return
 }
 lambdaClient := lambda.NewFromConfig(sdkConfig)

 maxItems := 10
 fmt.Printf("Let's list up to %v functions for your account.\n", maxItems)
 result, err := lambdaClient.ListFunctions(context.TODO(),
 &lambda.ListFunctionsInput{
  MaxItems: aws.Int32(int32(maxItems)),
 })
 if err != nil {
  fmt.Printf("Couldn't list functions for your account. Here's why: %v\n", err)
  return
 }
 if len(result.Functions) == 0 {
  fmt.Println("You don't have any functions!")
 } else {
  for _, function := range result.Functions {
   fmt.Printf("\t%v\n", *function.FunctionName)
  }
 }
}
```

- 如需 API 詳細資訊，請參閱 AWS SDK for Go API 參考ListFunctions中的。

Java

適用於 Java 2.x 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
package com.example.lambda;

import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.lambda.LambdaClient;
import software.amazon.awssdk.services.lambda.model.LambdaException;
import software.amazon.awssdk.services.lambda.model.ListFunctionsResponse;
import software.amazon.awssdk.services.lambda.model.FunctionConfiguration;
import java.util.List;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 */
public class ListLambdaFunctions {
    public static void main(String[] args) {
        Region region = Region.US_WEST_2;
        LambdaClient awsLambda = LambdaClient.builder()
                .region(region)
                .build();

        listFunctions(awsLambda);
        awsLambda.close();
    }

    public static void listFunctions(LambdaClient awsLambda) {
        try {
            ListFunctionsResponse functionResult = awsLambda.listFunctions();
```

```
            List<FunctionConfiguration> list = functionResult.functions();
            for (FunctionConfiguration config : list) {
                System.out.println("The function name is " +
 config.functionName());
            }

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

- 如需 API 詳細資訊，請參閱 AWS SDK for Java 2.x API 參考ListFunctions中的。

JavaScript

適用於 JavaScript (v3) 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
import { LambdaClient, paginateListFunctions } from "@aws-sdk/client-lambda";

const client = new LambdaClient({});

export const helloLambda = async () => {
  const paginator = paginateListFunctions({ client }, {});
  const functions = [];

  for await (const page of paginator) {
    const funcNames = page.Functions.map((f) => f.FunctionName);
    functions.push(...funcNames);
  }

  console.log("Functions:");
  console.log(functions.join("\n"));
  return functions;
```

```
};
```

- 如需 API 詳細資訊，請參閱 AWS SDK for JavaScript API 參考ListFunctions中的。

程式碼範例

- 使用 AWS SDK 的 Lambda 動作
  - 搭CreateAlias配 AWS 開發套件或 CLI 使用
  - 搭CreateFunction配 AWS 開發套件或 CLI 使用
  - 搭DeleteAlias配 AWS 開發套件或 CLI 使用
  - 搭DeleteFunction配 AWS 開發套件或 CLI 使用
  - 搭DeleteFunctionConcurrency配 AWS 開發套件或 CLI 使用
  - 搭DeleteProvisionedConcurrencyConfig配 AWS 開發套件或 CLI 使用
  - 搭GetAccountSettings配 AWS 開發套件或 CLI 使用
  - 搭GetAlias配 AWS 開發套件或 CLI 使用
  - 搭GetFunction配 AWS 開發套件或 CLI 使用
  - 搭GetFunctionConcurrency配 AWS 開發套件或 CLI 使用
  - 搭GetFunctionConfiguration配 AWS 開發套件或 CLI 使用
  - 搭GetPolicy配 AWS 開發套件或 CLI 使用
  - 搭GetProvisionedConcurrencyConfig配 AWS 開發套件或 CLI 使用
  - 搭Invoke配 AWS 開發套件或 CLI 使用
  - 搭ListFunctions配 AWS 開發套件或 CLI 使用
  - 搭ListProvisionedConcurrencyConfigs配 AWS 開發套件或 CLI 使用
  - 搭ListTags配 AWS 開發套件或 CLI 使用
  - 搭ListVersionsByFunction配 AWS 開發套件或 CLI 使用
  - 搭PublishVersion配 AWS 開發套件或 CLI 使用
  - 搭PutFunctionConcurrency配 AWS 開發套件或 CLI 使用
  - 搭PutProvisionedConcurrencyConfig配 AWS 開發套件或 CLI 使用
  - 搭RemovePermission配 AWS 開發套件或 CLI 使用
  - 搭TagResource配 AWS 開發套件或 CLI 使用
  - 搭UntagResource配 AWS 開發套件或 CLI 使用

- 使用排程事件來調用 Lambda 函數

使用 AWS SDK 的 Lambda 動作

下列程式碼範例示範如何使用 AWS SDK 執行個別 Lambda 動作。這些摘錄會呼叫 Lambda API，是必須在內容中執行之大型程式的程式碼摘錄。每個範例都包含一個連結 GitHub，您可以在其中找到設定和執行程式碼的指示。

下列範例僅包含最常使用的動作。如需完整清單，請參閱《AWS Lambda API 參考》。

範例

- 搭CreateAlias配 AWS 開發套件或 CLI 使用
- 搭CreateFunction配 AWS 開發套件或 CLI 使用
- 搭DeleteAlias配 AWS 開發套件或 CLI 使用
- 搭DeleteFunction配 AWS 開發套件或 CLI 使用
- 搭DeleteFunctionConcurrency配 AWS 開發套件或 CLI 使用
- 搭DeleteProvisionedConcurrencyConfig配 AWS 開發套件或 CLI 使用
- 搭GetAccountSettings配 AWS 開發套件或 CLI 使用
- 搭GetAlias配 AWS 開發套件或 CLI 使用
- 搭GetFunction配 AWS 開發套件或 CLI 使用
- 搭GetFunctionConcurrency配 AWS 開發套件或 CLI 使用
- 搭GetFunctionConfiguration配 AWS 開發套件或 CLI 使用
- 搭GetPolicy配 AWS 開發套件或 CLI 使用
- 搭GetProvisionedConcurrencyConfig配 AWS 開發套件或 CLI 使用
- 搭Invoke配 AWS 開發套件或 CLI 使用
- 搭ListFunctions配 AWS 開發套件或 CLI 使用
- 搭ListProvisionedConcurrencyConfigs配 AWS 開發套件或 CLI 使用
- 搭ListTags配 AWS 開發套件或 CLI 使用
- 搭ListVersionsByFunction配 AWS 開發套件或 CLI 使用
- 搭PublishVersion配 AWS 開發套件或 CLI 使用
- 搭PutFunctionConcurrency配 AWS 開發套件或 CLI 使用
- 搭PutProvisionedConcurrencyConfig配 AWS 開發套件或 CLI 使用

- 搭RemovePermission配 AWS 開發套件或 CLI 使用
- 搭TagResource配 AWS 開發套件或 CLI 使用
- 搭UntagResource配 AWS 開發套件或 CLI 使用
- 搭UpdateAlias配 AWS 開發套件或 CLI 使用
- 搭UpdateFunctionCode配 AWS 開發套件或 CLI 使用
- 搭UpdateFunctionConfiguration配 AWS 開發套件或 CLI 使用

搭**CreateAlias**配 AWS 開發套件或 CLI 使用

下列程式碼範例會示範如何使用CreateAlias。

CLI

AWS CLI

若要建立 Lambda 函數的別名

下列create-alias範例會建立名稱LIVE指向 my-function Lambda 函數第 1 版的別名。

```
aws lambda create-alias \
    --function-name my-function \
    --description "alias for live version of function" \
    --function-version 1 \
    --name LIVE
```

輸出：

```
{
    "FunctionVersion": "1",
    "Name": "LIVE",
    "AliasArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:LIVE",
    "RevisionId": "873282ed-4cd3-4dc8-a069-d0c647e470c6",
    "Description": "alias for live version of function"
}
```

如需詳細資訊，請參閱 AWS Lambda 開發人員指南中的設定AWS Lambda 函數別名。

- 如需 API 詳細資訊，請參閱AWS CLI 命令參考CreateAlias中的。

PowerShell

適用的工具 PowerShell

範例 1：此範例會針對指定的版本和路由組態建立新的 Lambda 別名，以指定其接收的叫用要求百分比。

```
New-LMAlias -FunctionName "MylambdaFunction123" -
RoutingConfig_AdditionalVersionWeight @{Name="1";Value="0.6} -Description "Alias
 for version 4" -FunctionVersion 4 -Name "PowershellAlias"
```

- 如需 API 詳細資訊，請參閱AWS Tools for PowerShell 指令程CreateAlias式參考中的。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭CreateFunction配 AWS 開發套件或 CLI 使用

下列程式碼範例會示範如何使用CreateFunction。

動作範例是大型程式的程式碼摘錄，必須在內容中執行。您可以在下列程式碼範例的內容中看到此動作：

- 開始使用函數

.NET

AWS SDK for .NET

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
    /// <summary>
    /// Creates a new Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the function.</param>
```

```
    /// <param name="s3Bucket">The Amazon Simple Storage Service (Amazon S3)
    /// bucket where the zip file containing the code is located.</param>
    /// <param name="s3Key">The Amazon S3 key of the zip file.</param>
    /// <param name="role">The Amazon Resource Name (ARN) of a role with the
    /// appropriate Lambda permissions.</param>
    /// <param name="handler">The name of the handler function.</param>
    /// <returns>The Amazon Resource Name (ARN) of the newly created
    /// Lambda function.</returns>
    public async Task<string> CreateLambdaFunctionAsync(
        string functionName,
        string s3Bucket,
        string s3Key,
        string role,
        string handler)
    {
        // Defines the location for the function code.
        // S3Bucket - The S3 bucket where the file containing
        //            the source code is stored.
        // S3Key    - The name of the file containing the code.
        var functionCode = new FunctionCode
        {
            S3Bucket = s3Bucket,
            S3Key = s3Key,
        };

        var createFunctionRequest = new CreateFunctionRequest
        {
            FunctionName = functionName,
            Description = "Created by the Lambda .NET API",
            Code = functionCode,
            Handler = handler,
            Runtime = Runtime.Dotnet6,
            Role = role,
        };

        var reponse = await
_lambdaService.CreateFunctionAsync(createFunctionRequest);
        return reponse.FunctionArn;
    }
```

- 如需 API 詳細資訊，請參閱 AWS SDK for .NET API 參考CreateFunction中的。

C++

適用於 C++ 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region in which the bucket was created
 (overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::Lambda::LambdaClient client(clientConfig);

        Aws::Lambda::Model::CreateFunctionRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        request.SetDescription(LAMBDA_DESCRIPTION); // Optional.
#if USE_CPP_LAMBDA_FUNCTION
        request.SetRuntime(Aws::Lambda::Model::Runtime::provided_al2);
        request.SetTimeout(15);
        request.SetMemorySize(128);

        // Assume the AWS Lambda function was built in Docker with same
 architecture
        // as this code.
#if  defined(__x86_64__)
        request.SetArchitectures({Aws::Lambda::Model::Architecture::x86_64});
#elif defined(__aarch64__)
        request.SetArchitectures({Aws::Lambda::Model::Architecture::arm64});
#else
#error "Unimplemented architecture"
#endif // defined(architecture)
#else
        request.SetRuntime(Aws::Lambda::Model::Runtime::python3_8);
#endif
        request.SetRole(roleArn);
        request.SetHandler(LAMBDA_HANDLER_NAME);
        request.SetPublish(true);
        Aws::Lambda::Model::FunctionCode code;
```

```
        std::ifstream ifstream(INCREMENT_LAMBDA_CODE.c_str(),
                               std::ios_base::in | std::ios_base::binary);
        if (!ifstream.is_open()) {
            std::cerr << "Error opening file " << INCREMENT_LAMBDA_CODE << "." <<
 std::endl;

#if USE_CPP_LAMBDA_FUNCTION
            std::cerr
                    << "The cpp Lambda function must be built following the
 instructions in the cpp_lambda/README.md file. "
                    << std::endl;
#endif
            deleteIamRole(clientConfig);
            return false;
        }

        Aws::StringStream buffer;
        buffer << ifstream.rdbuf();

        code.SetZipFile(Aws::Utils::ByteBuffer((unsigned char *)
 buffer.str().c_str(),
                                               buffer.str().length()));
        request.SetCode(code);

        Aws::Lambda::Model::CreateFunctionOutcome outcome =
 client.CreateFunction(
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda function was successfully created. " <<
 seconds
                      << " seconds elapsed." << std::endl;
            break;
        }

        else {
            std::cerr << "Error with CreateFunction. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
            deleteIamRole(clientConfig);
            return false;
        }
```

- 如需 API 詳細資訊，請參閱 AWS SDK for C++ API 參考CreateFunction中的。

CLI

AWS CLI

若要建立 Lambda 函數

下列 `create-function` 範例會建立名為 `my-function` 的 Lambda 函數。

```
aws lambda create-function \
    --function-name my-function \
    --runtime nodejs18.x \
    --zip-file fileb://my-function.zip \
    --handler my-function.handler \
    --role arn:aws:iam::123456789012:role/service-role/MyTestFunction-role-
tges6bf4
```

`my-function.zip` 的內容：

```
This file is a deployment package that contains your function code and any
 dependencies.
```

輸出：

```
{
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "CodeSha256": "PFn4S+er27qk+UuZSTKEQfNKG/XNn7QJs90mJgq6oH8=",
    "FunctionName": "my-function",
    "CodeSize": 308,
    "RevisionId": "873282ed-4cd3-4dc8-a069-d0c647e470c6",
    "MemorySize": 128,
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "Version": "$LATEST",
    "Role": "arn:aws:iam::123456789012:role/service-role/MyTestFunction-role-
zgur6bf4",
    "Timeout": 3,
    "LastModified": "2023-10-14T22:26:11.234+0000",
    "Handler": "my-function.handler",
    "Runtime": "nodejs18.x",
```

```
    "Description": ""
}
```

如需詳細資訊，請參閱《AWS Lambda 開發人員指南》中的 AWS Lambda 函數組態。

- 如需 API 詳細資訊，請參閱AWS CLI 命令參考CreateFunction中的。

Go

SDK for Go V2

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// CreateFunction creates a new Lambda function from code contained in the
 zipPackage
// buffer. The specified handlerName must match the name of the file and function
// contained in the uploaded code. The role specified by iamRoleArn is assumed by
// Lambda and grants specific permissions.
// When the function already exists, types.StateActive is returned.
// When the function is created, a lambda.FunctionActiveV2Waiter is used to wait
 until the
// function is active.
func (wrapper FunctionWrapper) CreateFunction(functionName string, handlerName
 string,
 iamRoleArn *string, zipPackage *bytes.Buffer) types.State {
 var state types.State
 _, err := wrapper.LambdaClient.CreateFunction(context.TODO(),
 &lambda.CreateFunctionInput{
  Code:         &types.FunctionCode{ZipFile: zipPackage.Bytes()},
```

```
  FunctionName: aws.String(functionName),
  Role:         iamRoleArn,
  Handler:      aws.String(handlerName),
  Publish:      true,
  Runtime:      types.RuntimePython38,
 })
 if err != nil {
  var resConflict *types.ResourceConflictException
  if errors.As(err, &resConflict) {
   log.Printf("Function %v already exists.\n", functionName)
   state = types.StateActive
  } else {
   log.Panicf("Couldn't create function %v. Here's why: %v\n", functionName, err)
  }
 } else {
  waiter := lambda.NewFunctionActiveV2Waiter(wrapper.LambdaClient)
  funcOutput, err := waiter.WaitForOutput(context.TODO(),
 &lambda.GetFunctionInput{
   FunctionName: aws.String(functionName)}, 1*time.Minute)
  if err != nil {
   log.Panicf("Couldn't wait for function %v to be active. Here's why: %v\n",
 functionName, err)
  } else {
   state = funcOutput.Configuration.State
  }
 }
 return state
}
```

- 如需 API 詳細資訊，請參閱 AWS SDK for Go API 參考CreateFunction中的。

Java

適用於 Java 2.x 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
import software.amazon.awssdk.core.SdkBytes;
import software.amazon.awssdk.core.waiters.WaiterResponse;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.lambda.LambdaClient;
import software.amazon.awssdk.services.lambda.model.CreateFunctionRequest;
import software.amazon.awssdk.services.lambda.model.FunctionCode;
import software.amazon.awssdk.services.lambda.model.CreateFunctionResponse;
import software.amazon.awssdk.services.lambda.model.GetFunctionRequest;
import software.amazon.awssdk.services.lambda.model.GetFunctionResponse;
import software.amazon.awssdk.services.lambda.model.LambdaException;
import software.amazon.awssdk.services.lambda.model.Runtime;
import software.amazon.awssdk.services.lambda.waiters.LambdaWaiter;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.InputStream;

/**
 * This code example requires a ZIP or JAR that represents the code of the
 * Lambda function.
 * If you do not have a ZIP or JAR, please refer to the following document:
 *
 * https://github.com/aws-doc-sdk-examples/tree/master/javav2/usecases/
creating_workflows_stepfunctions
 *
 * Also, set up your development environment, including your credentials.
 *
 * For information, see this documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 */

public class CreateFunction {
    public static void main(String[] args) {

        final String usage = """

                Usage:
                    <functionName> <filePath> <role> <handler>\s

                Where:
                    functionName - The name of the Lambda function.\s
```

```
                    filePath - The path to the ZIP or JAR where the code is
located.\s
                    role - The role ARN that has Lambda permissions.\s
                    handler - The fully qualified method name (for example,
example.Handler::handleRequest). \s
                """;

        if (args.length != 4) {
            System.out.println(usage);
            System.exit(1);
        }

        String functionName = args[0];
        String filePath = args[1];
        String role = args[2];
        String handler = args[3];
        Region region = Region.US_WEST_2;
        LambdaClient awsLambda = LambdaClient.builder()
                .region(region)
                .build();

        createLambdaFunction(awsLambda, functionName, filePath, role, handler);
        awsLambda.close();
    }

    public static void createLambdaFunction(LambdaClient awsLambda,
            String functionName,
            String filePath,
            String role,
            String handler) {

        try {
            LambdaWaiter waiter = awsLambda.waiter();
            InputStream is = new FileInputStream(filePath);
            SdkBytes fileToUpload = SdkBytes.fromInputStream(is);

            FunctionCode code = FunctionCode.builder()
                    .zipFile(fileToUpload)
                    .build();

            CreateFunctionRequest functionRequest =
CreateFunctionRequest.builder()
                    .functionName(functionName)
                    .description("Created by the Lambda Java API")
```

```
                    .code(code)
                    .handler(handler)
                    .runtime(Runtime.JAVA8)
                    .role(role)
                    .build();

            // Create a Lambda function using a waiter.
            CreateFunctionResponse functionResponse =
 awsLambda.createFunction(functionRequest);
            GetFunctionRequest getFunctionRequest = GetFunctionRequest.builder()
                    .functionName(functionName)
                    .build();
            WaiterResponse<GetFunctionResponse> waiterResponse =
 waiter.waitUntilFunctionExists(getFunctionRequest);
            waiterResponse.matched().response().ifPresent(System.out::println);
            System.out.println("The function ARN is " +
 functionResponse.functionArn());

        } catch (LambdaException | FileNotFoundException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

- 如需 API 詳細資訊，請參閱 AWS SDK for Java 2.x API 參考CreateFunction中的。

JavaScript

適用於 JavaScript (v3) 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
const createFunction = async (funcName, roleArn) => {
  const client = new LambdaClient({});
  const code = await readFile(`${dirname}../functions/${funcName}.zip`);
```

```
  const command = new CreateFunctionCommand({
    Code: { ZipFile: code },
    FunctionName: funcName,
    Role: roleArn,
    Architectures: [Architecture.arm64],
    Handler: "index.handler", // Required when sending a .zip file
    PackageType: PackageType.Zip, // Required when sending a .zip file
    Runtime: Runtime.nodejs16x, // Required when sending a .zip file
  });

  return client.send(command);
};
```

- 如需 API 詳細資訊，請參閱 AWS SDK for JavaScript API 參考CreateFunction中的。

Kotlin

適用於 Kotlin 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
suspend fun createNewFunction(
    myFunctionName: String,
    s3BucketName: String,
    myS3Key: String,
    myHandler: String,
    myRole: String
): String? {
    val functionCode =
        FunctionCode {
            s3Bucket = s3BucketName
            s3Key = myS3Key
        }

    val request =
        CreateFunctionRequest {
            functionName = myFunctionName
```

```
            code = functionCode
            description = "Created by the Lambda Kotlin API"
            handler = myHandler
            role = myRole
            runtime = Runtime.Java8
        }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        val functionResponse = awsLambda.createFunction(request)
        awsLambda.waitUntilFunctionActive {
            functionName = myFunctionName
        }
        return functionResponse.functionArn
    }
}
```

- 有關 API 的詳細信息，請參閱 AWS SDK CreateFunction中的 Kotlin API 參考。

PHP

適用於 PHP 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
    public function createFunction($functionName, $role, $bucketName, $handler)
    {
        //This assumes the Lambda function is in an S3 bucket.
        return $this->customWaiter(function () use ($functionName, $role,
$bucketName, $handler) {
            return $this->lambdaClient->createFunction([
                'Code' => [
                    'S3Bucket' => $bucketName,
                    'S3Key' => $functionName,
                ],
                'FunctionName' => $functionName,
                'Role' => $role['Arn'],
                'Runtime' => 'python3.9',
```

```
                'Handler' => "$handler.lambda_handler",
            ]);
        });
    }
```

- 如需 API 詳細資訊，請參閱 AWS SDK for PHP API 參考CreateFunction中的。

PowerShell

適用的工具 PowerShell

範例 1：此範例會建立 AWS Lambda MyFunction 中名為的新 C# (dotnetcore1.0 執行階段) 函數，從本機檔案系統上的 zip 檔案提供函數的編譯二進位檔案 (可以使用相對或絕對路徑)。C# Lambda 函數指定使用指定函式的處理常式: AssemblyName: 命名空間。 ClassName：：MethodName。您應該適當替換處理程序規範的程序集名稱（不帶 .dll 後綴），命名空間，類名和方法名稱部分。新函數將具有環境變量 'envvar1' 和 'envvar2' 從提供的值設置。

```
Publish-LMFunction -Description "My C# Lambda Function" `
        -FunctionName MyFunction `
        -ZipFilename .\MyFunctionBinaries.zip `
        -Handler "AssemblyName::Namespace.ClassName::MethodName" `
        -Role "arn:aws:iam::123456789012:role/LambdaFullExecRole" `
        -Runtime dotnetcore1.0 `
        -Environment_Variable @{ "envvar1"="value";"envvar2"="value" }
```

輸出：

```
CodeSha256       : /NgBMd...gq71I=
CodeSize         : 214784
DeadLetterConfig :
Description      : My C# Lambda Function
Environment      : Amazon.Lambda.Model.EnvironmentResponse
FunctionArn      : arn:aws:lambda:us-west-2:123456789012:function:ToUpper
FunctionName     : MyFunction
Handler          : AssemblyName::Namespace.ClassName::MethodName
KMSKeyArn        :
LastModified     : 2016-12-29T23:50:14.207+0000
MemorySize       : 128
Role             : arn:aws:iam::123456789012:role/LambdaFullExecRole
Runtime          : dotnetcore1.0
```

```
Timeout          : 3
Version          : $LATEST
VpcConfig        :
```

範例 2：除了函數二進位檔首先上傳到 Amazon S3 儲存貯體 (必須與預期的 Lambda 函數位於相同區域) 之外，此範例與前一個範例類似，然後在建立函數時參照產生的 S3 物件。

```
Write-S3Object -BucketName mybucket -Key MyFunctionBinaries.zip -File .
\MyFunctionBinaries.zip
Publish-LMFunction -Description "My C# Lambda Function" `
        -FunctionName MyFunction `
        -BucketName mybucket `
        -Key MyFunctionBinaries.zip `
        -Handler "AssemblyName::Namespace.ClassName::MethodName" `
        -Role "arn:aws:iam::123456789012:role/LambdaFullExecRole" `
        -Runtime dotnetcore1.0 `
        -Environment_Variable @{ "envvar1"="value";"envvar2"="value" }
```

- 如需 API 詳細資訊，請參閱AWS Tools for PowerShell 指令程CreateFunction式參考中的。

Python

適用於 Python (Boto3) 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def create_function(
        self, function_name, handler_name, iam_role, deployment_package
    ):
        """
```

```
        Deploys a Lambda function.

        :param function_name: The name of the Lambda function.
        :param handler_name: The fully qualified name of the handler function.
This
                             must include the file name and the function name.
        :param iam_role: The IAM role to use for the function.
        :param deployment_package: The deployment package that contains the
function
                                   code in .zip format.
        :return: The Amazon Resource Name (ARN) of the newly created function.
        """
        try:
            response = self.lambda_client.create_function(
                FunctionName=function_name,
                Description="AWS Lambda doc example",
                Runtime="python3.8",
                Role=iam_role.arn,
                Handler=handler_name,
                Code={"ZipFile": deployment_package},
                Publish=True,
            )
            function_arn = response["FunctionArn"]
            waiter = self.lambda_client.get_waiter("function_active_v2")
            waiter.wait(FunctionName=function_name)
            logger.info(
                "Created function '%s' with ARN: '%s'.",
                function_name,
                response["FunctionArn"],
            )
        except ClientError:
            logger.error("Couldn't create function %s.", function_name)
            raise
        else:
            return function_arn
```

- 如需 API 的詳細資訊，請參閱AWS 開發套件CreateFunction中的 Python (博托 3) API 參考。

Ruby

適用於 Ruby 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
class LambdaWrapper
  attr_accessor :lambda_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Deploys a Lambda function.
  #
  # @param function_name: The name of the Lambda function.
  # @param handler_name: The fully qualified name of the handler function. This
  #                      must include the file name and the function name.
  # @param role_arn: The IAM role to use for the function.
  # @param deployment_package: The deployment package that contains the function
  #                            code in .zip format.
  # @return: The Amazon Resource Name (ARN) of the newly created function.
  def create_function(function_name, handler_name, role_arn, deployment_package)
    response = @lambda_client.create_function({
                                                role: role_arn.to_s,
                                                function_name: function_name,
                                                handler: handler_name,
                                                runtime: "ruby2.7",
                                                code: {
                                                  zip_file: deployment_package
                                                },
                                                environment: {
                                                  variables: {
                                                    "LOG_LEVEL" => "info"
                                                  }
                                                }
```

```
                                                  })
    @lambda_client.wait_until(:function_active_v2, { function_name:
function_name}) do |w|
      w.max_attempts = 5
      w.delay = 5
    end
    response
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error creating #{function_name}:\n #{e.message}")
  rescue Aws::Waiters::Errors::WaiterFailed => e
    @logger.error("Failed waiting for #{function_name} to activate:\n
#{e.message}")
  end
```

- 如需 API 詳細資訊，請參閱 AWS SDK for Ruby API 參考CreateFunction中的。

Rust

適用於 Rust 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
    /**
     * Create a function, uploading from a zip file.
     */
    pub async fn create_function(&self, zip_file: PathBuf) -> Result<String,
anyhow::Error> {
        let code = self.prepare_function(zip_file, None).await?;

        let key = code.s3_key().unwrap().to_string();

        let role = self.create_role().await.map_err(|e| anyhow!(e))?;

        info!("Created iam role, waiting 15s for it to become active");
        tokio::time::sleep(Duration::from_secs(15)).await;

        info!("Creating lambda function {}", self.lambda_name);
```

```
        let _ = self
            .lambda_client
            .create_function()
            .function_name(self.lambda_name.clone())
            .code(code)
            .role(role.arn())
            .runtime(aws_sdk_lambda::types::Runtime::Providedal2)
            .handler("_unused")
            .send()
            .await
            .map_err(anyhow::Error::from)?;

        self.wait_for_function_ready().await?;

        self.lambda_client
            .publish_version()
            .function_name(self.lambda_name.clone())
            .send()
            .await?;

        Ok(key)
    }

    /**
     * Upload function code from a path to a zip file.
     * The zip file must have an AL2 Linux-compatible binary called `bootstrap`.
     * The easiest way to create such a zip is to use `cargo lambda build --
output-format Zip`.
     */
    async fn prepare_function(
        &self,
        zip_file: PathBuf,
        key: Option<String>,
    ) -> Result<FunctionCode, anyhow::Error> {
        let body = ByteStream::from_path(zip_file).await?;

        let key = key.unwrap_or_else(|| format!("{}_code", self.lambda_name));

        info!("Uploading function code to s3://{}/{}", self.bucket, key);
        let _ = self
            .s3_client
            .put_object()
            .bucket(self.bucket.clone())
            .key(key.clone())
```

```
            .body(body)
            .send()
            .await?;

        Ok(FunctionCode::builder()
            .s3_bucket(self.bucket.clone())
            .s3_key(key)
            .build())
    }
```

- 如需 API 的詳細資訊，請參閱 AWS SDK CreateFunction中的 Rust API 參考資料。

SAP ABAP

適用於 SAP ABAP 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
    TRY.
        lo_lmd->createfunction(
            iv_functionname = iv_function_name
            iv_runtime = `python3.9`
            iv_role = iv_role_arn
            iv_handler = iv_handler
            io_code = io_zip_file
            iv_description = 'AWS Lambda code example'
        ).
        MESSAGE 'Lambda function created.' TYPE 'I'.
      CATCH /aws1/cx_lmdcodesigningcfgno00.
        MESSAGE 'Code signing configuration does not exist.' TYPE 'E'.
      CATCH /aws1/cx_lmdcodestorageexcdex.
        MESSAGE 'Maximum total code size per account exceeded.' TYPE 'E'.
      CATCH /aws1/cx_lmdcodeverification00.
        MESSAGE 'Code signature failed one or more validation checks for
signature mismatch or expiration.' TYPE 'E'.
      CATCH /aws1/cx_lmdinvalidcodesigex.
        MESSAGE 'Code signature failed the integrity check.' TYPE 'E'.
```

```
      CATCH /aws1/cx_lmdinvparamvalueex.
        MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
      CATCH /aws1/cx_lmdresourceconflictex.
        MESSAGE 'Resource already exists or another operation is in progress.'
TYPE 'E'.
      CATCH /aws1/cx_lmdresourcenotfoundex.
        MESSAGE 'The requested resource does not exist.' TYPE 'E'.
      CATCH /aws1/cx_lmdserviceexception.
        MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
TYPE 'E'.
      CATCH /aws1/cx_lmdtoomanyrequestsex.
        MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
    ENDTRY.
```

- 如需 API 詳細資訊，請參閱 AWS SDK CreateFunction中的 SAP ABAP API 參考資料。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭DeleteAlias配 AWS 開發套件或 CLI 使用

下列程式碼範例會示範如何使用DeleteAlias。

CLI

AWS CLI

若要刪除 Lambda 函數的別名

下列delete-alias範例會刪除 L my-function ambda 函數LIVE中指定的別名。

```
aws lambda delete-alias \
    --function-name my-function \
    --name LIVE
```

此命令不會產生輸出。

如需詳細資訊，請參閱 AWS Lambda 開發人員指南中的設定AWS Lambda 函數別名。

- 如需 API 詳細資訊，請參閱AWS CLI 命令參考DeleteAlias中的。

PowerShell

適用的工具 PowerShell

範例 1：此範例會刪除命令中提到的 Lambda 函數別名。

```
Remove-LMAlias -FunctionName "MylambdaFunction123" -Name "NewAlias"
```

- 如需 API 詳細資訊，請參閱AWS Tools for PowerShell 指令程DeleteAlias式參考中的。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭DeleteFunction配 AWS 開發套件或 CLI 使用

下列程式碼範例會示範如何使用DeleteFunction。

動作範例是大型程式的程式碼摘錄，必須在內容中執行。您可以在下列程式碼範例的內容中看到此動作：

- 開始使用函數

.NET

AWS SDK for .NET

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
    /// <summary>
    /// Delete an AWS Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to
    /// delete.</param>
    /// <returns>A Boolean value that indicates the success of the action.</
returns>
    public async Task<bool> DeleteFunctionAsync(string functionName)
```

```
    {
        var request = new DeleteFunctionRequest
        {
            FunctionName = functionName,
        };

        var response = await _lambdaService.DeleteFunctionAsync(request);

        // A return value of NoContent means that the request was processed.
        // In this case, the function was deleted, and the return value
        // is intentionally blank.
        return response.HttpStatusCode == System.Net.HttpStatusCode.NoContent;
    }
```

- 如需 API 詳細資訊，請參閱 AWS SDK for .NET API 參考DeleteFunction中的。

C++

適用於 C++ 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::Lambda::LambdaClient client(clientConfig);

    Aws::Lambda::Model::DeleteFunctionRequest request;
    request.SetFunctionName(LAMBDA_NAME);

    Aws::Lambda::Model::DeleteFunctionOutcome outcome = client.DeleteFunction(
            request);

    if (outcome.IsSuccess()) {
```

```
        std::cout << "The lambda function was successfully deleted." <<
 std::endl;
    }
    else {
        std::cerr << "Error with Lambda::DeleteFunction. "
                  << outcome.GetError().GetMessage()
                  << std::endl;
    }
```

- 如需 API 詳細資訊，請參閱 AWS SDK for C++ API 參考DeleteFunction中的。

CLI

AWS CLI

範例 1：若要依函數名稱刪除 Lambda 函數

下列 delete-function 範例會透過指定函數的名稱來刪除名為 my-function 的 Lambda 函數。

```
aws lambda delete-function \
    --function-name my-function
```

此命令不會產生輸出。

範例 2：若要依函數 ARN 刪除 Lambda 函數

下列 delete-function 範例會透過指定函數的 ARN 來刪除名為 my-function 的 Lambda 函數。

```
aws lambda delete-function \
    --function-name arn:aws:lambda:us-west-2:123456789012:function:my-function
```

此命令不會產生輸出。

範例 3：若要依部分函數 ARN 刪除 Lambda 函數

下列 delete-function 範例會透過指定函數的部分 ARN 來刪除名為 my-function 的 Lambda 函數。

```
aws lambda delete-function \
```

```
    --function-name 123456789012:function:my-function
```

此命令不會產生輸出。

如需詳細資訊，請參閱《AWS Lambda 開發人員指南》中的 AWS Lambda 函數組態。

- 如需 API 詳細資訊，請參閱AWS CLI 命令參考DeleteFunction中的。

Go

SDK for Go V2

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// DeleteFunction deletes the Lambda function specified by functionName.
func (wrapper FunctionWrapper) DeleteFunction(functionName string) {
 _, err := wrapper.LambdaClient.DeleteFunction(context.TODO(),
 &lambda.DeleteFunctionInput{
  FunctionName: aws.String(functionName),
 })
 if err != nil {
  log.Panicf("Couldn't delete function %v. Here's why: %v\n", functionName, err)
 }
}
```

- 如需 API 詳細資訊，請參閱 AWS SDK for Go API 參考DeleteFunction中的。

Java

適用於 Java 2.x 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
import software.amazon.awssdk.services.lambda.LambdaClient;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.lambda.model.DeleteFunctionRequest;
import software.amazon.awssdk.services.lambda.model.LambdaException;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 */
public class DeleteFunction {
    public static void main(String[] args) {
        final String usage = """

                Usage:
                    <functionName>\s

                Where:
                    functionName - The name of the Lambda function.\s
                """;

        if (args.length != 1) {
            System.out.println(usage);
            System.exit(1);
        }

        String functionName = args[0];
        Region region = Region.US_EAST_1;
```

```
        LambdaClient awsLambda = LambdaClient.builder()
                .region(region)
                .build();

        deleteLambdaFunction(awsLambda, functionName);
        awsLambda.close();
    }

    public static void deleteLambdaFunction(LambdaClient awsLambda, String
 functionName) {
        try {
            DeleteFunctionRequest request = DeleteFunctionRequest.builder()
                    .functionName(functionName)
                    .build();

            awsLambda.deleteFunction(request);
            System.out.println("The " + functionName + " function was deleted");

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

- 如需 API 詳細資訊，請參閱 AWS SDK for Java 2.x API 參考DeleteFunction中的。

JavaScript

適用於 JavaScript (v3) 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
/**
 * @param {string} funcName
 */
const deleteFunction = (funcName) => {
```

```
  const client = new LambdaClient({});
  const command = new DeleteFunctionCommand({ FunctionName: funcName });
  return client.send(command);
};
```

- 如需 API 詳細資訊，請參閱 AWS SDK for JavaScript API 參考DeleteFunction中的。

Kotlin

適用於 Kotlin 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
suspend fun delLambdaFunction(myFunctionName: String) {
    val request =
        DeleteFunctionRequest {
            functionName = myFunctionName
        }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        awsLambda.deleteFunction(request)
        println("$myFunctionName was deleted")
    }
}
```

- 有關 API 的詳細信息，請參閱 AWS SDK DeleteFunction中的 Kotlin API 參考。

PHP

適用於 PHP 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
public function deleteFunction($functionName)
{
    return $this->lambdaClient->deleteFunction([
        'FunctionName' => $functionName,
    ]);
}
```

- 如需 API 詳細資訊，請參閱 AWS SDK for PHP API 參考DeleteFunction中的。

PowerShell

適用的工具 PowerShell

範例 1：此範例會刪除 Lambda 函數的特定版本

```
Remove-LMFunction -FunctionName "MylambdaFunction123" -Qualifier '3'
```

- 如需 API 詳細資訊，請參閱AWS Tools for PowerShell 指令程DeleteFunction式參考中的。

Python

適用於 Python (Boto3) 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def delete_function(self, function_name):
        """
        Deletes a Lambda function.

        :param function_name: The name of the function to delete.
        """
        try:
            self.lambda_client.delete_function(FunctionName=function_name)
        except ClientError:
            logger.exception("Couldn't delete function %s.", function_name)
            raise
```

- 如需 API 的詳細資訊，請參閱AWS 開發套件DeleteFunction中的 Python (博托 3) API 參考。

Ruby

適用於 Ruby 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
class LambdaWrapper
  attr_accessor :lambda_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end
```

```
  # Deletes a Lambda function.
  # @param function_name: The name of the function to delete.
  def delete_function(function_name)
    print "Deleting function: #{function_name}..."
    @lambda_client.delete_function(
      function_name: function_name
    )
    print "Done!".green
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error deleting #{function_name}:\n #{e.message}")
  end
```

- 如需 API 詳細資訊，請參閱 AWS SDK for Ruby API 參考[DeleteFunction](#)中的。

Rust

適用於 Rust 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在[AWS 設定和執行程式碼範例儲存庫](#)。

```
   /** Delete a function and its role, and if possible or necessary, its
associated code object and bucket. */
   pub async fn delete_function(
       &self,
       location: Option<String>,
   ) -> (
       Result<DeleteFunctionOutput, anyhow::Error>,
       Result<DeleteRoleOutput, anyhow::Error>,
       Option<Result<DeleteObjectOutput, anyhow::Error>>,
   ) {
       info!("Deleting lambda function {}", self.lambda_name);
       let delete_function = self
           .lambda_client
           .delete_function()
           .function_name(self.lambda_name.clone())
           .send()
           .await
```

```
            .map_err(anyhow::Error::from);

        info!("Deleting iam role {}", self.role_name);
        let delete_role = self
            .iam_client
            .delete_role()
            .role_name(self.role_name.clone())
            .send()
            .await
            .map_err(anyhow::Error::from);

        let delete_object: Option<Result<DeleteObjectOutput, anyhow::Error>> =
            if let Some(location) = location {
                info!("Deleting object {location}");
                Some(
                    self.s3_client
                        .delete_object()
                        .bucket(self.bucket.clone())
                        .key(location)
                        .send()
                        .await
                        .map_err(anyhow::Error::from),
                )
            } else {
                info!(?location, "Skipping delete object");
                None
            };

        (delete_function, delete_role, delete_object)
    }
```

- 如需 API 的詳細資訊，請參閱 AWS SDK DeleteFunction中的 Rust API 參考資料。

SAP ABAP

適用於 SAP ABAP 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
   TRY.
       lo_lmd->deletefunction( iv_functionname = iv_function_name ).
       MESSAGE 'Lambda function deleted.' TYPE 'I'.
     CATCH /aws1/cx_lmdinvparamvalueex.
       MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
     CATCH /aws1/cx_lmdresourceconflictex.
       MESSAGE 'Resource already exists or another operation is in progress.'
TYPE 'E'.
     CATCH /aws1/cx_lmdresourcenotfoundex.
       MESSAGE 'The requested resource does not exist.' TYPE 'E'.
     CATCH /aws1/cx_lmdserviceexception.
       MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
TYPE 'E'.
     CATCH /aws1/cx_lmdtoomanyrequestsex.
       MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
   ENDTRY.
```

- 如需 API 詳細資訊，請參閱 AWS SDK DeleteFunction中的 SAP ABAP API 參考資料。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭DeleteFunctionConcurrency配 AWS 開發套件或 CLI 使用

下列程式碼範例會示範如何使用DeleteFunctionConcurrency。

CLI

AWS CLI

從函數中移除保留的並行執行限制

下列delete-function-concurrency範例會從my-function函式刪除保留的並行執行限制。

```
aws lambda delete-function-concurrency \
    --function-name  my-function
```

此命令不會產生輸出。

如需詳細資訊，請參閱 Lambda 開發人員指南中的保留 Lambda 函數的AWS 並行處理。

- 如需 API 詳細資訊，請參閱AWS CLI 命令參考中的DeleteFunction並行。

PowerShell

適用的工具 PowerShell

範例 1：此範例會移除 Lambda 函數的函數並行處理。

```
Remove-LMFunctionConcurrency -FunctionName "MylambdaFunction123"
```

- 如需 API 詳細資訊，請參閱AWS Tools for PowerShell 指令程式參DeleteFunction考中的並行。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭DeleteProvisionedConcurrencyConfig配 AWS 開發套件或 CLI 使用

下列程式碼範例會示範如何使用DeleteProvisionedConcurrencyConfig。

CLI

AWS CLI

刪除已提供的並行組態

下列delete-provisioned-concurrency-config範例會刪除指定函數GREEN別名的佈建並行組態。

```
aws lambda delete-provisioned-concurrency-config \
    --function-name my-function \
    --qualifier GREEN
```

- 如需 API 詳細資訊，請參閱AWS CLI 命令參考DeleteProvisionedConcurrencyConfig中的。

PowerShell

適用的工具 PowerShell

範例 1：此範例會移除特定別名的「佈建並行組態」。

```
Remove-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
Qualifier "NewAlias1"
```

- 如需 API 詳細資訊，請參閱AWS Tools for PowerShell 指令程DeleteProvisionedConcurrencyConfig式參考中的。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭GetAccountSettings配 AWS 開發套件或 CLI 使用

下列程式碼範例會示範如何使用GetAccountSettings。

CLI

AWS CLI

擷取您在某個 AWS 區域中的帳戶詳細資訊

下列get-account-settings範例會顯示您帳戶的 Lambda 限制和用量資訊。

```
aws lambda get-account-settings
```

輸出：

```
{
    "AccountLimit": {
        "CodeSizeUnzipped": 262144000,
        "UnreservedConcurrentExecutions": 1000,
        "ConcurrentExecutions": 1000,
        "CodeSizeZipped": 52428800,
        "TotalCodeSize": 80530636800
    },
    "AccountUsage": {
        "FunctionCount": 4,
        "TotalCodeSize": 9426
```

```
    }
}
```

如需詳細資訊，請參閱 AWS Lambda 開發人員指南中的AWS Lambda 限制。

- 如需 API 詳細資訊，請參閱AWS CLI 命令參考中的GetAccount設定。

PowerShell

適用的工具 PowerShell

範例 1：此範例顯示以比較帳戶限制和帳戶使用情況

```
Get-LMAccountSetting | Select-Object
 @{Name="TotalCodeSizeLimit";Expression={$_.AccountLimit.TotalCodeSize}},
 @{Name="TotalCodeSizeUsed";Expression={$_.AccountUsage.TotalCodeSize}}
```

輸出：

```
TotalCodeSizeLimit TotalCodeSizeUsed
------------------ -----------------
       80530636800          15078795
```

- 如需 API 詳細資訊，請參閱AWS Tools for PowerShell 指令程式參考中的GetAccount設定。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭GetAlias配 AWS 開發套件或 CLI 使用

下列程式碼範例會示範如何使用GetAlias。

CLI

AWS CLI

若要擷取有關函數別名的詳細資訊

下列get-alias範例會顯示在 my-function Lambda 函數LIVE上命名之別名的詳細資料。

```
aws lambda get-alias \
```

```
    --function-name my-function \
    --name LIVE
```

輸出：

```
{
    "FunctionVersion": "3",
    "Name": "LIVE",
    "AliasArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:LIVE",
    "RevisionId": "594f41fb-b85f-4c20-95c7-6ca5f2a92c93",
    "Description": "alias for live version of function"
}
```

如需詳細資訊，請參閱 AWS Lambda 開發人員指南中的設定AWS Lambda 函數別名。

- 如需 API 詳細資訊，請參閱AWS CLI 命令參考GetAlias中的。

PowerShell

適用的工具 PowerShell

範例 1：此範例會擷取特定 Lambda 函數別名的「路由 Config 定」權重。

```
Get-LMAlias -FunctionName "MylambdaFunction123" -Name "newlabel1" -Select
 RoutingConfig
```

輸出：

```
AdditionalVersionWeights
------------------------
{[1, 0.6]}
```

- 如需 API 詳細資訊，請參閱AWS Tools for PowerShell 指令程GetAlias式參考中的。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭GetFunction配 AWS 開發套件或 CLI 使用

下列程式碼範例會示範如何使用GetFunction。

動作範例是大型程式的程式碼摘錄，必須在內容中執行。您可以在下列程式碼範例的內容中看到此動作：

- 開始使用函數

.NET

AWS SDK for .NET

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
    /// <summary>
    /// Gets information about a Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function for
    /// which to retrieve information.</param>
    /// <returns>Async Task.</returns>
    public async Task<FunctionConfiguration> GetFunctionAsync(string
functionName)
    {
        var functionRequest = new GetFunctionRequest
        {
            FunctionName = functionName,
        };

        var response = await _lambdaService.GetFunctionAsync(functionRequest);
        return response.Configuration;
    }
```

- 如需 API 詳細資訊，請參閱 AWS SDK for .NET API 參考GetFunction中的。

C++

適用於 C++ 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
       Aws::Client::ClientConfiguration clientConfig;
       // Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
       // clientConfig.region = "us-east-1";

   Aws::Lambda::LambdaClient client(clientConfig);

       Aws::Lambda::Model::GetFunctionRequest request;
       request.SetFunctionName(functionName);

       Aws::Lambda::Model::GetFunctionOutcome outcome =
client.GetFunction(request);

       if (outcome.IsSuccess()) {
           std::cout << "Function retrieve.\n" <<

outcome.GetResult().GetConfiguration().Jsonize().View().WriteReadable()
                     << std::endl;
       }
       else {
           std::cerr << "Error with Lambda::GetFunction. "
                     << outcome.GetError().GetMessage()
                     << std::endl;
       }
```

- 如需 API 詳細資訊，請參閱 AWS SDK for C++ API 參考GetFunction中的。

CLI

AWS CLI

若要擷取函數相關資訊

下列 get-function 範例顯示 my-function 函數的相關資訊。

```
aws lambda get-function \
    --function-name  my-function
```

輸出：

```
{
    "Concurrency": {
        "ReservedConcurrentExecutions": 100
    },
    "Code": {
        "RepositoryType": "S3",
        "Location": "https://awslambda-us-west-2-tasks.s3.us-
west-2.amazonaws.com/snapshots/123456789012/my-function..."
    },
    "Configuration": {
        "TracingConfig": {
            "Mode": "PassThrough"
        },
        "Version": "$LATEST",
        "CodeSha256": "5tT2qgzYUHoqwR616pZ2dpkn/0J1FrzJmlKidWaaCgk=",
        "FunctionName": "my-function",
        "VpcConfig": {
            "SubnetIds": [],
            "VpcId": "",
            "SecurityGroupIds": []
        },
        "MemorySize": 128,
        "RevisionId": "28f0fb31-5c5c-43d3-8955-03e76c5c1075",
        "CodeSize": 304,
        "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function",
        "Handler": "index.handler",
        "Role": "arn:aws:iam::123456789012:role/service-role/helloWorldPython-
role-uy3l9qyq",
        "Timeout": 3,
```

```
        "LastModified": "2019-09-24T18:20:35.054+0000",
        "Runtime": "nodejs10.x",
        "Description": ""
    }
}
```

如需詳細資訊，請參閱《AWS Lambda 開發人員指南》中的 AWS Lambda 函數組態。

- 如需 API 詳細資訊，請參閱AWS CLI 命令參考GetFunction中的。

Go

SDK for Go V2

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// GetFunction gets data about the Lambda function specified by functionName.
func (wrapper FunctionWrapper) GetFunction(functionName string) types.State {
 var state types.State
 funcOutput, err := wrapper.LambdaClient.GetFunction(context.TODO(),
 &lambda.GetFunctionInput{
  FunctionName: aws.String(functionName),
 })
 if err != nil {
  log.Panicf("Couldn't get function %v. Here's why: %v\n", functionName, err)
 } else {
  state = funcOutput.Configuration.State
 }
 return state
```

```
}
```

- 如需 API 詳細資訊，請參閱 AWS SDK for Go API 參考GetFunction中的。

JavaScript

適用於 JavaScript (v3) 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
const getFunction = (funcName) => {
  const client = new LambdaClient({});
  const command = new GetFunctionCommand({ FunctionName: funcName });
  return client.send(command);
};
```

- 如需 API 詳細資訊，請參閱 AWS SDK for JavaScript API 參考GetFunction中的。

PHP

適用於 PHP 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
    public function getFunction($functionName)
    {
        return $this->lambdaClient->getFunction([
            'FunctionName' => $functionName,
```

```
        ]);
    }
```

- 如需 API 詳細資訊，請參閱 AWS SDK for PHP API 參考GetFunction中的。

Python

適用於 Python (Boto3) 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def get_function(self, function_name):
        """
        Gets data about a Lambda function.

        :param function_name: The name of the function.
        :return: The function data.
        """
        response = None
        try:
            response =
 self.lambda_client.get_function(FunctionName=function_name)
        except ClientError as err:
            if err.response["Error"]["Code"] == "ResourceNotFoundException":
                logger.info("Function %s does not exist.", function_name)
            else:
                logger.error(
                    "Couldn't get function %s. Here's why: %s: %s",
                    function_name,
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
```

```
                )
                raise
        return response
```

- 如需 API 的詳細資訊，請參閱AWS 開發套件GetFunction中的 Python (博托 3) API 參考。

Ruby

適用於 Ruby 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
class LambdaWrapper
  attr_accessor :lambda_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Gets data about a Lambda function.
  #
  # @param function_name: The name of the function.
  # @return response: The function data, or nil if no such function exists.
  def get_function(function_name)
    @lambda_client.get_function(
      {
        function_name: function_name
      }
    )
  rescue Aws::Lambda::Errors::ResourceNotFoundException => e
    @logger.debug("Could not find function: #{function_name}:\n #{e.message}")
    nil
  end
```

- 如需 API 詳細資訊，請參閱 AWS SDK for Ruby API 參考GetFunction中的。

Rust

適用於 Rust 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
    /** Get the Lambda function with this Manager's name. */
    pub async fn get_function(&self) -> Result<GetFunctionOutput, anyhow::Error>
{
        info!("Getting lambda function");
        self.lambda_client
            .get_function()
            .function_name(self.lambda_name.clone())
            .send()
            .await
            .map_err(anyhow::Error::from)
    }
```

- 如需 API 的詳細資訊，請參閱 AWS SDK GetFunction中的 Rust API 參考資料。

SAP ABAP

適用於 SAP ABAP 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
    TRY.
        oo_result =  lo_lmd->getfunction( iv_functionname = iv_function_name ).
     " oo_result is returned for testing purposes. "
        MESSAGE 'Lambda function information retrieved.' TYPE 'I'.
      CATCH /aws1/cx_lmdinvparamvalueex.
        MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
      CATCH /aws1/cx_lmdserviceexception.
        MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
 TYPE 'E'.
      CATCH /aws1/cx_lmdtoomanyrequestsex.
        MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
    ENDTRY.
```

- 如需 API 詳細資訊，請參閱 AWS SDK GetFunction中的 SAP ABAP API 參考資料。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭GetFunctionConcurrency配 AWS 開發套件或 CLI 使用

下列程式碼範例會示範如何使用GetFunctionConcurrency。

CLI

AWS CLI

若要檢視函數的保留並行設定

下列get-function-concurrency範例會擷取指定函數的保留並行設定。

```
aws lambda get-function-concurrency \
    --function-name my-function
```

輸出：

```
{
    "ReservedConcurrentExecutions": 250
}
```

- 如需 API 詳細資訊，請參閱AWS CLI 命令參考中的GetFunction並行。

PowerShell

適用的工具 PowerShell

範例 1：此範例取得 Lambda 函數的保留並行處理

```
Get-LMFunctionConcurrency -FunctionName "MylambdaFunction123" -Select *
```

輸出：

```
ReservedConcurrentExecutions
----------------------------
100
```

- 如需 API 詳細資訊，請參閱AWS Tools for PowerShell 指令程式參GetFunction考中的並行。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭GetFunctionConfiguration配 AWS 開發套件或 CLI 使用

下列程式碼範例會示範如何使用GetFunctionConfiguration。

CLI

AWS CLI

若要擷取 Lambda 函數的版本特定設定

下列get-function-configuration範例會顯示my-function函數第 2 版的設定。

```
aws lambda get-function-configuration \
    --function-name  my-function:2
```

輸出：

```
{
    "FunctionName": "my-function",
    "LastModified": "2019-09-26T20:28:40.438+0000",
    "RevisionId": "e52502d4-9320-4688-9cd6-152a6ab7490d",
    "MemorySize": 256,
```

```
    "Version": "2",
    "Role": "arn:aws:iam::123456789012:role/service-role/my-function-role-
uy3l9qyq",
    "Timeout": 3,
    "Runtime": "nodejs10.x",
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "CodeSha256": "5tT2qgzYUHaqwR716pZ2dpkn/0J1FrzJmlKidWoaCgk=",
    "Description": "",
    "VpcConfig": {
        "SubnetIds": [],
        "VpcId": "",
        "SecurityGroupIds": []
    },
    "CodeSize": 304,
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:2",
    "Handler": "index.handler"
}
```

如需詳細資訊，請參閱《AWS Lambda 開發人員指南》中的 AWS Lambda 函數組態。

- 如需 API 詳細資訊，請參閱AWS CLI 命令參考中的GetFunction組態。

PowerShell

適用的工具 PowerShell

範例 1：此範例會傳回 Lambda 函數的版本特定組態。

```
Get-LMFunctionConfiguration -FunctionName "MylambdaFunction123" -Qualifier
 "PowershellAlias"
```

輸出：

```
CodeSha256                  : uWOW0R7z+f0VyLuUg7+/D08hkMFsq0SF4seuyUZJ/R8=
CodeSize                    : 1426
DeadLetterConfig            : Amazon.Lambda.Model.DeadLetterConfig
Description                 : Verson 3 to test Aliases
Environment                 : Amazon.Lambda.Model.EnvironmentResponse
FunctionArn                 : arn:aws:lambda:us-
east-1:123456789012:function:MylambdaFunction123
```

```
                                :PowershellAlias
FunctionName                : MylambdaFunction123
Handler                     : lambda_function.launch_instance
KMSKeyArn                   :
LastModified                : 2019-12-25T09:52:59.872+0000
LastUpdateStatus            : Successful
LastUpdateStatusReason      :
LastUpdateStatusReasonCode  :
Layers                      : {}
MasterArn                   :
MemorySize                  : 128
RevisionId                  : 5d7de38b-87f2-4260-8f8a-e87280e10c33
Role                        : arn:aws:iam::123456789012:role/service-role/lambda
Runtime                     : python3.8
State                       : Active
StateReason                 :
StateReasonCode             :
Timeout                     : 600
TracingConfig               : Amazon.Lambda.Model.TracingConfigResponse
Version                     : 4
VpcConfig                   : Amazon.Lambda.Model.VpcConfigDetail
```

- 如需 API 詳細資訊，請參閱AWS Tools for PowerShell 指令程式參考中的GetFunction組態。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭GetPolicy配 AWS 開發套件或 CLI 使用

下列程式碼範例會示範如何使用GetPolicy。

CLI

AWS CLI

擷取函數、版本或別名的以資源為基礎的 IAM 政策

下列get-policy範例顯示有關 my-function Lambda 函數的政策資訊。

```
aws lambda get-policy \
    --function-name my-function
```

輸出：

```
{
    "Policy": {
        "Version":"2012-10-17",
        "Id":"default",
        "Statement":
        [
            {
                "Sid":"iot-events",
                "Effect":"Allow",
                "Principal": {"Service":"iotevents.amazonaws.com"},
                "Action":"lambda:InvokeFunction",
                "Resource":"arn:aws:lambda:us-west-2:123456789012:function:my-
function"
            }
        ]
    },
    "RevisionId": "93017fc9-59cb-41dc-901b-4845ce4bf668"
}
```

如需詳細資訊，請參閱 Lambda 開發人員指南中的針對 AWS Lambda 使用以資源為基礎的 AWS

- 如需 API 詳細資訊，請參閱AWS CLI 命令參考GetPolicy中的。

PowerShell

適用的工具 PowerShell

範例 1：此範例顯示 Lambda 函數的函數政策

```
Get-LMPolicy -FunctionName test -Select Policy
```

輸出：

```
{"Version":"2012-10-17","Id":"default","Statement":
[{"Sid":"xxxx","Effect":"Allow","Principal":
{"Service":"sns.amazonaws.com"},"Action":"lambda:InvokeFunction","Resource":"arn:aws:lamb
east-1:123456789102:function:test"}]}
```

- 如需 API 詳細資訊，請參閱AWS Tools for PowerShell 指令程GetPolicy式參考中的。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭GetProvisionedConcurrencyConfig配 AWS 開發套件或 CLI 使用

下列程式碼範例會示範如何使用GetProvisionedConcurrencyConfig。

CLI

AWS CLI

若要檢視已提供的並行組態

下列get-provisioned-concurrency-config範例會針對指定函數的BLUE別名顯示已佈建並行組態的詳細資訊。

```
aws lambda get-provisioned-concurrency-config \
    --function-name my-function \
    --qualifier BLUE
```

輸出：

```
{
    "RequestedProvisionedConcurrentExecutions": 100,
    "AvailableProvisionedConcurrentExecutions": 100,
    "AllocatedProvisionedConcurrentExecutions": 100,
    "Status": "READY",
    "LastModified": "2019-12-31T20:28:49+0000"
}
```

- 如需 API 詳細資訊，請參閱AWS CLI 命令參考GetProvisionedConcurrencyConfig中的。

PowerShell

適用的工具 PowerShell

範例 1：此範例取得指定 Lambda 函數別名的佈建並行組態。

```
C:\>Get-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
Qualifier "NewAlias1"
```

輸出：

```
AllocatedProvisionedConcurrentExecutions : 0
AvailableProvisionedConcurrentExecutions : 0
LastModified                             : 2020-01-15T03:21:26+0000
RequestedProvisionedConcurrentExecutions : 70
Status                                   : IN_PROGRESS
StatusReason                             :
```

- 如需 API 詳細資訊，請參閱AWS Tools for PowerShell 指令程GetProvisionedConcurrencyConfig式參考中的。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭Invoke配 AWS 開發套件或 CLI 使用

下列程式碼範例會示範如何使用Invoke。

動作範例是大型程式的程式碼摘錄，必須在內容中執行。您可以在下列程式碼範例的內容中看到此動作：

- 開始使用函數

.NET

AWS SDK for .NET

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
    /// <summary>
    /// Invoke a Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to
    /// invoke.</param
    /// <param name="parameters">The parameter values that will be passed to the
function.</param>
```

```
    /// <returns>A System Threading Task.</returns>
    public async Task<string> InvokeFunctionAsync(
        string functionName,
        string parameters)
    {
        var payload = parameters;
        var request = new InvokeRequest
        {
            FunctionName = functionName,
            Payload = payload,
        };

        var response = await _lambdaService.InvokeAsync(request);
        MemoryStream stream = response.Payload;
        string returnValue =
System.Text.Encoding.UTF8.GetString(stream.ToArray());
        return returnValue;
    }
```

- 如需 API 的詳細資訊，請參閱《AWS SDK for .NET API 參考》中的「Invoke」。

C++

適用於 C++ 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::Lambda::LambdaClient client(clientConfig);

        Aws::Lambda::Model::InvokeRequest request;
        request.SetFunctionName(LAMBDA_NAME);
```

```
        request.SetLogType(logType);
        std::shared_ptr<Aws::IOStream> payload =
 Aws::MakeShared<Aws::StringStream>(
                "FunctionTest");
        *payload << jsonPayload.View().WriteReadable();
        request.SetBody(payload);
        request.SetContentType("application/json");
        Aws::Lambda::Model::InvokeOutcome outcome = client.Invoke(request);

        if (outcome.IsSuccess()) {
            invokeResult = std::move(outcome.GetResult());
            result = true;
            break;
        }

        else {
            std::cerr << "Error with Lambda::InvokeRequest. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
            break;
        }
```

- 如需 API 的詳細資訊，請參閱《AWS SDK for C++ API 參考》中的「Invoke」。

CLI

AWS CLI

範例 1：若要同步調用 Lambda 函數

下列 `invoke` 範例會同步調用 `my-function` 函數。如果您使用的是 AWS CLI 第 2 版，則需要此`cli-binary-format`選項。如需詳細資訊，請參閱《AWS 命令列介面使用者指南》中的 AWS CLI 支援的全域命令列選項。

```
aws lambda invoke \
    --function-name my-function \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "name": "Bob" }' \
    response.json
```

輸出：

```
{
    "ExecutedVersion": "$LATEST",
    "StatusCode": 200
}
```

如需詳細資訊，請參閱《AWS Lambda 開發人員指南》中的同步調用。

範例 2：若要非同步調用 Lambda 函數

下列 invoke 範例會非同步調用 my-function 函數。如果您使用的是 AWS CLI 第 2 版，則需要此cli-binary-format選項。如需詳細資訊，請參閱《AWS 命令列介面使用者指南》中的 AWS CLI 支援的全域命令列選項。

```
aws lambda invoke \
    --function-name my-function \
    --invocation-type Event \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "name": "Bob" }' \
    response.json
```

輸出：

```
{
    "StatusCode": 202
}
```

如需詳細資訊，請參閱《AWS Lambda 開發人員指南》中的非同步調用。

- 如需 API 詳細資訊，請參閱《AWS CLI 命令參考》中的 Invoke。

Go

SDK for Go V2

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// Invoke invokes the Lambda function specified by functionName, passing the
 parameters
// as a JSON payload. When getLog is true, types.LogTypeTail is specified, which
 tells
// Lambda to include the last few log lines in the returned result.
func (wrapper FunctionWrapper) Invoke(functionName string, parameters any, getLog
 bool) *lambda.InvokeOutput {
 logType := types.LogTypeNone
 if getLog {
  logType = types.LogTypeTail
 }
 payload, err := json.Marshal(parameters)
 if err != nil {
  log.Panicf("Couldn't marshal parameters to JSON. Here's why %v\n", err)
 }
 invokeOutput, err := wrapper.LambdaClient.Invoke(context.TODO(),
 &lambda.InvokeInput{
  FunctionName: aws.String(functionName),
  LogType:      logType,
  Payload:      payload,
 })
 if err != nil {
  log.Panicf("Couldn't invoke function %v. Here's why: %v\n", functionName, err)
 }
 return invokeOutput
}
```

- 如需 API 的詳細資訊，請參閱《AWS SDK for Go API 參考》中的「Invoke」。

Java

適用於 Java 2.x 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
import org.json.JSONObject;
import software.amazon.awssdk.auth.credentials.ProfileCredentialsProvider;
import software.amazon.awssdk.services.lambda.LambdaClient;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.lambda.model.InvokeRequest;
import software.amazon.awssdk.core.SdkBytes;
import software.amazon.awssdk.services.lambda.model.InvokeResponse;
import software.amazon.awssdk.services.lambda.model.LambdaException;

public class LambdaInvoke {

    /*
     * Function names appear as
     * arn:aws:lambda:us-west-2:335556666777:function:HelloFunction
     * you can retrieve the value by looking at the function in the AWS Console
     *
     * Also, set up your development environment, including your credentials.
     *
     * For information, see this documentation topic:
     *
     * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.
     * html
     */

    public static void main(String[] args) {
        final String usage = """

                Usage:
                    <functionName>\s

                Where:
```

```
                    functionName - The name of the Lambda function\s
                """";

        if (args.length != 1) {
            System.out.println(usage);
            System.exit(1);
        }

        String functionName = args[0];
        Region region = Region.US_WEST_2;
        LambdaClient awsLambda = LambdaClient.builder()
                .region(region)
                .build();

        invokeFunction(awsLambda, functionName);
        awsLambda.close();
    }

    public static void invokeFunction(LambdaClient awsLambda, String
functionName) {

        InvokeResponse res = null;
        try {
            // Need a SdkBytes instance for the payload.
            JSONObject jsonObj = new JSONObject();
            jsonObj.put("inputValue", "2000");
            String json = jsonObj.toString();
            SdkBytes payload = SdkBytes.fromUtf8String(json);

            // Setup an InvokeRequest.
            InvokeRequest request = InvokeRequest.builder()
                    .functionName(functionName)
                    .payload(payload)
                    .build();

            res = awsLambda.invoke(request);
            String value = res.payload().asUtf8String();
            System.out.println(value);

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
```

```
}
```

- 如需 API 的詳細資訊，請參閱《AWS SDK for Java 2.x API 參考》中的「Invoke」。

JavaScript

適用於 JavaScript (v3) 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
const invoke = async (funcName, payload) => {
  const client = new LambdaClient({});
  const command = new InvokeCommand({
    FunctionName: funcName,
    Payload: JSON.stringify(payload),
    LogType: LogType.Tail,
  });

  const { Payload, LogResult } = await client.send(command);
  const result = Buffer.from(Payload).toString();
  const logs = Buffer.from(LogResult, "base64").toString();
  return { logs, result };
};
```

- 如需 API 的詳細資訊，請參閱《AWS SDK for JavaScript API 參考》中的「Invoke」。

Kotlin

適用於 Kotlin 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
suspend fun invokeFunction(functionNameVal: String) {
    val json = """{"inputValue":"1000"}"""
    val byteArray = json.trimIndent().encodeToByteArray()
    val request =
        InvokeRequest {
            functionName = functionNameVal
            logType = LogType.Tail
            payload = byteArray
        }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        val res = awsLambda.invoke(request)
        println("${res.payload?.toString(Charsets.UTF_8)}")
        println("The log result is ${res.logResult}")
    }
}
```

- 如需 API 的詳細資訊，請參閱《適用於 Kotlin 的AWS 開發套件 API 參考》中的「Invoke」。

PHP

適用於 PHP 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
    public function invoke($functionName, $params, $logType = 'None')
    {
        return $this->lambdaClient->invoke([
            'FunctionName' => $functionName,
            'Payload' => json_encode($params),
            'LogType' => $logType,
        ]);
    }
```

- 如需 API 的詳細資訊，請參閱《AWS SDK for PHP API 參考》中的「Invoke」。

Python

適用於 Python (Boto3) 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def invoke_function(self, function_name, function_params, get_log=False):
        """
        Invokes a Lambda function.

        :param function_name: The name of the function to invoke.
        :param function_params: The parameters of the function as a dict. This
 dict
                                is serialized to JSON before it is sent to
 Lambda.
        :param get_log: When true, the last 4 KB of the execution log are
 included in
                        the response.
        :return: The response from the function invocation.
```

```
        """
        try:
            response = self.lambda_client.invoke(
                FunctionName=function_name,
                Payload=json.dumps(function_params),
                LogType="Tail" if get_log else "None",
            )
            logger.info("Invoked function %s.", function_name)
        except ClientError:
            logger.exception("Couldn't invoke function %s.", function_name)
            raise
        return response
```

- 如需 API 的詳細資訊，請參閱《適用於 Python (Boto3) 的AWS 開發套件 API 參考》中的 Invoke。

Ruby

適用於 Ruby 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
class LambdaWrapper
  attr_accessor :lambda_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Invokes a Lambda function.
  # @param function_name [String] The name of the function to invoke.
  # @param payload [nil] Payload containing runtime parameters.
  # @return [Object] The response from the function invocation.
```

```
  def invoke_function(function_name, payload = nil)
    params = { function_name: function_name}
    params[:payload] = payload unless payload.nil?
    @lambda_client.invoke(params)
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error executing #{function_name}:\n
 #{e.message}")
  end
```

- 如需 API 的詳細資訊，請參閱《AWS SDK for Ruby API 參考》中的「Invoke」。

Rust

適用於 Rust 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
    /** Invoke the lambda function using calculator InvokeArgs. */
    pub async fn invoke(&self, args: InvokeArgs) -> Result<InvokeOutput,
 anyhow::Error> {
        info!(?args, "Invoking {}", self.lambda_name);
        let payload = serde_json::to_string(&args)?;
        debug!(?payload, "Sending payload");
        self.lambda_client
            .invoke()
            .function_name(self.lambda_name.clone())
            .payload(Blob::new(payload))
            .send()
            .await
            .map_err(anyhow::Error::from)
    }

fn log_invoke_output(invoke: &InvokeOutput, message: &str) {
    if let Some(payload) = invoke.payload().cloned() {
        let payload = String::from_utf8(payload.into_inner());
        info!(?payload, message);
    } else {
```

```
        info!("Could not extract payload")
    }
    if let Some(logs) = invoke.log_result() {
        debug!(?logs, "Invoked function logs")
    } else {
        debug!("Invoked function had no logs")
    }
}
```

- 如需 API 詳細資訊，請參閱《AWS SDK for Rust API 參考》中的 Invoke。

SAP ABAP

適用於 SAP ABAP 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
    TRY.
        DATA(lv_json) = /aws1/cl_rt_util=>string_to_xstring(
          `{`  &&
            `"action": "increment",`  &&
            `"number": 10` &&
          `}`
        ).
        oo_result =  lo_lmd->invoke(                  " oo_result is returned for
testing purposes. "
                iv_functionname = iv_function_name
                iv_payload = lv_json
             ).
        MESSAGE 'Lambda function invoked.' TYPE 'I'.
      CATCH /aws1/cx_lmdinvparamvalueex.
        MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
      CATCH /aws1/cx_lmdinvrequestcontex.
        MESSAGE 'Unable to parse request body as JSON.' TYPE 'E'.
      CATCH /aws1/cx_lmdinvalidzipfileex.
        MESSAGE 'The deployment package could not be unzipped.' TYPE 'E'.
      CATCH /aws1/cx_lmdrequesttoolargeex.
```

```
        MESSAGE 'Invoke request body JSON input limit was exceeded by the request
payload.' TYPE 'E'.
      CATCH /aws1/cx_lmdresourceconflictex.
        MESSAGE 'Resource already exists or another operation is in progress.'
TYPE 'E'.
      CATCH /aws1/cx_lmdresourcenotfoundex.
        MESSAGE 'The requested resource does not exist.' TYPE 'E'.
      CATCH /aws1/cx_lmdserviceexception.
        MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
TYPE 'E'.
      CATCH /aws1/cx_lmdtoomanyrequestsex.
        MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
      CATCH /aws1/cx_lmdunsuppedmediatyp00.
        MESSAGE 'Invoke request body does not have JSON as its content type.'
TYPE 'E'.
    ENDTRY.
```

- 如需 API 詳細資訊，請參閱《適用於 SAP ABAP 的AWS SDK API 參考》中的 Invoke。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭`ListFunctions`配 AWS 開發套件或 CLI 使用

下列程式碼範例會示範如何使用`ListFunctions`。

動作範例是大型程式的程式碼摘錄，必須在內容中執行。您可以在下列程式碼範例的內容中看到此動作：

- 開始使用函數

.NET

AWS SDK for .NET

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
    /// <summary>
    /// Get a list of Lambda functions.
    /// </summary>
    /// <returns>A list of FunctionConfiguration objects.</returns>
    public async Task<List<FunctionConfiguration>> ListFunctionsAsync()
    {
        var functionList = new List<FunctionConfiguration>();

        var functionPaginator =
            _lambdaService.Paginators.ListFunctions(new ListFunctionsRequest());
        await foreach (var function in functionPaginator.Functions)
        {
            functionList.Add(function);
        }

        return functionList;
    }
```

- 如需 API 詳細資訊，請參閱 AWS SDK for .NET API 參考ListFunctions中的。

C++

適用於 C++ 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::Lambda::LambdaClient client(clientConfig);

    std::vector<Aws::String> functions;
    Aws::String marker;
```

```
   do {
       Aws::Lambda::Model::ListFunctionsRequest request;
       if (!marker.empty()) {
           request.SetMarker(marker);
       }

       Aws::Lambda::Model::ListFunctionsOutcome outcome = client.ListFunctions(
               request);

       if (outcome.IsSuccess()) {
           const Aws::Lambda::Model::ListFunctionsResult &result =
outcome.GetResult();
           std::cout << result.GetFunctions().size()
                     << " lambda functions were retrieved." << std::endl;

           for (const Aws::Lambda::Model::FunctionConfiguration
&functionConfiguration: result.GetFunctions()) {
               functions.push_back(functionConfiguration.GetFunctionName());
               std::cout << functions.size() << "  "
                         << functionConfiguration.GetDescription() << std::endl;
               std::cout << "   "
                         <<
Aws::Lambda::Model::RuntimeMapper::GetNameForRuntime(
                                 functionConfiguration.GetRuntime()) << ": "
                         << functionConfiguration.GetHandler()
                         << std::endl;
           }
           marker = result.GetNextMarker();
       }
       else {
           std::cerr << "Error with Lambda::ListFunctions. "
                     << outcome.GetError().GetMessage()
                     << std::endl;
       }
   } while (!marker.empty());
```

- 如需 API 詳細資訊，請參閱 AWS SDK for C++ API 參考ListFunctions中的。

CLI

AWS CLI

若要擷取 Lambda 函數的清單

下列 `list-functions` 範例會顯示目前使用者的所有函數清單。

```
aws lambda list-functions
```

輸出：

```
{
    "Functions": [
        {
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "$LATEST",
            "CodeSha256": "dBG9m8SGdmlEjw/JYXlhhvCrAv5TxvXsbL/RMr0fT/I=",
            "FunctionName": "helloworld",
            "MemorySize": 128,
            "RevisionId": "1718e831-badf-4253-9518-d0644210af7b",
            "CodeSize": 294,
            "FunctionArn": "arn:aws:lambda:us-
west-2:123456789012:function:helloworld",
            "Handler": "helloworld.handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/MyTestFunction-
role-zgur6bf4",
            "Timeout": 3,
            "LastModified": "2023-09-23T18:32:33.857+0000",
            "Runtime": "nodejs18.x",
            "Description": ""
        },
        {
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "$LATEST",
            "CodeSha256": "sU0cJ2/hOZevwV/lTxCuQqK3gDZP3i8gUoqUUVRmY6E=",
            "FunctionName": "my-function",
            "VpcConfig": {
                "SubnetIds": [],
```

```
                "VpcId": "",
                "SecurityGroupIds": []
            },
            "MemorySize": 256,
            "RevisionId": "93017fc9-59cb-41dc-901b-4845ce4bf668",
            "CodeSize": 266,
            "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function",
            "Handler": "index.handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/
helloWorldPython-role-uy3l9qyq",
            "Timeout": 3,
            "LastModified": "2023-10-01T16:47:28.490+0000",
            "Runtime": "nodejs18.x",
            "Description": ""
        },
        {
            "Layers": [
                {
                    "CodeSize": 41784542,
                    "Arn": "arn:aws:lambda:us-
west-2:420165488524:layer:AWSLambda-Python37-SciPy1x:2"
                },
                {
                    "CodeSize": 4121,
                    "Arn": "arn:aws:lambda:us-
west-2:123456789012:layer:pythonLayer:1"
                }
            ],
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "$LATEST",
            "CodeSha256": "ZQukCqxtkqFgyF2cU41Avj99TKQ/hNihPtDtRcc08mI=",
            "FunctionName": "my-python-function",
            "VpcConfig": {
                "SubnetIds": [],
                "VpcId": "",
                "SecurityGroupIds": []
            },
            "MemorySize": 128,
            "RevisionId": "80b4eabc-acf7-4ea8-919a-e874c213707d",
            "CodeSize": 299,
```

```
            "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
python-function",
            "Handler": "lambda_function.lambda_handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/my-python-
function-role-z5g7dr6n",
            "Timeout": 3,
            "LastModified": "2023-10-01T19:40:41.643+0000",
            "Runtime": "python3.11",
            "Description": ""
        }
    ]
}
```

如需詳細資訊，請參閱《AWS Lambda 開發人員指南》中的 AWS Lambda 函數組態。

- 如需 API 詳細資訊，請參閱AWS CLI 命令參考ListFunctions中的。

Go

SDK for Go V2

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// ListFunctions lists up to maxItems functions for the account. This function
 uses a
// lambda.ListFunctionsPaginator to paginate the results.
func (wrapper FunctionWrapper) ListFunctions(maxItems int)
 []types.FunctionConfiguration {
 var functions []types.FunctionConfiguration
```

```
 paginator := lambda.NewListFunctionsPaginator(wrapper.LambdaClient,
 &lambda.ListFunctionsInput{
  MaxItems: aws.Int32(int32(maxItems)),
 })
 for paginator.HasMorePages() && len(functions) < maxItems {
  pageOutput, err := paginator.NextPage(context.TODO())
  if err != nil {
   log.Panicf("Couldn't list functions for your account. Here's why: %v\n", err)
  }
  functions = append(functions, pageOutput.Functions...)
 }
 return functions
}
```

- 如需 API 詳細資訊，請參閱 AWS SDK for Go API 參考ListFunctions中的。

JavaScript

適用於 JavaScript (v3) 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
const listFunctions = () => {
  const client = new LambdaClient({});
  const command = new ListFunctionsCommand({});

  return client.send(command);
};
```

- 如需 API 詳細資訊，請參閱 AWS SDK for JavaScript API 參考ListFunctions中的。

PHP

適用於 PHP 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
    public function listFunctions($maxItems = 50, $marker = null)
    {
        if (is_null($marker)) {
            return $this->lambdaClient->listFunctions([
                'MaxItems' => $maxItems,
            ]);
        }

        return $this->lambdaClient->listFunctions([
            'Marker' => $marker,
            'MaxItems' => $maxItems,
        ]);
    }
```

- 如需 API 詳細資訊，請參閱 AWS SDK for PHP API 參考ListFunctions中的。

PowerShell

適用的工具 PowerShell

範例 1：此範例顯示所有具有排序程式碼大小的 Lambda 函數

```
Get-LMFunctionList | Sort-Object -Property CodeSize | Select-Object FunctionName,
 RunTime, Timeout, CodeSize
```

輸出：

```
FunctionName                                                  Runtime   Timeout
 CodeSize
```

```
------------                                                    -------   -------
 --------
test                                                            python2.7       3
   243
MylambdaFunction123                                             python3.8     600
   659
myfuncpython1                                                   python3.8     303
   675
```

- 如需 API 詳細資訊，請參閱AWS Tools for PowerShell 指令程ListFunctions式參考中的。

Python

適用於 Python (Boto3) 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def list_functions(self):
        """
        Lists the Lambda functions for the current account.
        """
        try:
            func_paginator = self.lambda_client.get_paginator("list_functions")
            for func_page in func_paginator.paginate():
                for func in func_page["Functions"]:
                    print(func["FunctionName"])
                    desc = func.get("Description")
                    if desc:
                        print(f"\t{desc}")
                    print(f"\t{func['Runtime']}: {func['Handler']}")
        except ClientError as err:
            logger.error(
```

```
                "Couldn't list functions. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
```

- 如需 API 的詳細資訊，請參閱AWS 開發套件ListFunctions中的 Python (博托 3) API 參考。

Ruby

適用於 Ruby 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
class LambdaWrapper
  attr_accessor :lambda_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Lists the Lambda functions for the current account.
  def list_functions
    functions = []
    @lambda_client.list_functions.each do |response|
      response["functions"].each do |function|
        functions.append(function["function_name"])
      end
    end
    functions
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error executing #{function_name}:\n
 #{e.message}")
  end
```

- 如需 API 詳細資訊，請參閱 AWS SDK for Ruby API 參考ListFunctions中的。

Rust

適用於 Rust 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
    /** List all Lambda functions in the current Region. */
    pub async fn list_functions(&self) -> Result<ListFunctionsOutput,
anyhow::Error> {
        info!("Listing lambda functions");
        self.lambda_client
            .list_functions()
            .send()
            .await
            .map_err(anyhow::Error::from)
    }
```

- 如需 API 的詳細資訊，請參閱 AWS SDK ListFunctions中的 Rust API 參考資料。

SAP ABAP

適用於 SAP ABAP 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
    TRY.
```

```
        oo_result =  lo_lmd->listfunctions( ).       " oo_result is returned for
 testing purposes. "
        DATA(lt_functions) = oo_result->get_functions( ).
        MESSAGE 'Retrieved list of Lambda functions.' TYPE 'I'.
      CATCH /aws1/cx_lmdinvparamvalueex.
        MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
      CATCH /aws1/cx_lmdserviceexception.
        MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
 TYPE 'E'.
      CATCH /aws1/cx_lmdtoomanyrequestsex.
        MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
    ENDTRY.
```

- 如需 API 詳細資訊，請參閱 AWS SDK ListFunctions中的 SAP ABAP API 參考資料。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭`ListProvisionedConcurrencyConfigs`配 AWS 開發套件或 CLI 使用

下列程式碼範例會示範如何使用`ListProvisionedConcurrencyConfigs`。

CLI

AWS CLI

取得佈建並行設定的清單

下列`list-provisioned-concurrency-configs`範例會列出針對指定函數佈建的並行設定。

```
aws lambda list-provisioned-concurrency-configs \
    --function-name my-function
```

輸出：

```
{
    "ProvisionedConcurrencyConfigs": [
        {
            "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-
function:GREEN",
```

```
            "RequestedProvisionedConcurrentExecutions": 100,
            "AvailableProvisionedConcurrentExecutions": 100,
            "AllocatedProvisionedConcurrentExecutions": 100,
            "Status": "READY",
            "LastModified": "2019-12-31T20:29:00+0000"
        },
        {
            "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-
function:BLUE",
            "RequestedProvisionedConcurrentExecutions": 100,
            "AvailableProvisionedConcurrentExecutions": 100,
            "AllocatedProvisionedConcurrentExecutions": 100,
            "Status": "READY",
            "LastModified": "2019-12-31T20:28:49+0000"
        }
    ]
}
```

- 如需 API 詳細資訊，請參閱AWS CLI 命令參考ListProvisionedConcurrencyConfigs中的。

PowerShell

適用的工具 PowerShell

範例 1：此範例會擷取 Lambda 函數的佈建並行組態清單。

```
Get-LMProvisionedConcurrencyConfigList -FunctionName "MylambdaFunction123"
```

- 如需 API 詳細資訊，請參閱AWS Tools for PowerShell 指令程ListProvisionedConcurrencyConfigs式參考中的。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭ListTags配 AWS 開發套件或 CLI 使用

下列程式碼範例會示範如何使用ListTags。

CLI

AWS CLI

若要擷取 Lambda 函數的標籤清單

下列`list-tags`範例會顯示附加至 `my-function` Lambda 函數的標籤。

```
aws lambda list-tags \
    --resource arn:aws:lambda:us-west-2:123456789012:function:my-function
```

輸出：

```
{
    "Tags": {
        "Category": "Web Tools",
        "Department": "Sales"
    }
}
```

如需詳細資訊，請參閱 Lambda 開發人員指南中的標記AWS Lambda 函數。

- 如需 API 詳細資訊，請參閱AWS CLI 命令參考ListTags中的。

PowerShell

適用的工具 PowerShell

範例 1：擷取目前在指定函數上設定的標籤及其值。

```
Get-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction"
```

輸出：

```
Key        Value
---        -----
California Sacramento
Oregon     Salem
Washington Olympia
```

- 如需 API 詳細資訊，請參閱AWS Tools for PowerShell 指令程ListTags式參考中的。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭ListVersionsByFunction配 AWS 開發套件或 CLI 使用

下列程式碼範例會示範如何使用ListVersionsByFunction。

CLI

AWS CLI

若要擷取函數版本清單

下列list-versions-by-function範例會顯示 my-function Lambda 函數的版本清單。

```
aws lambda list-versions-by-function \
    --function-name my-function
```

輸出：

```
{
    "Versions": [
        {
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "$LATEST",
            "CodeSha256": "sU0cJ2/hOZevwV/lTxCuQqK3gDZP3i8gUoqUUVRmY6E=",
            "FunctionName": "my-function",
            "VpcConfig": {
                "SubnetIds": [],
                "VpcId": "",
                "SecurityGroupIds": []
            },
            "MemorySize": 256,
            "RevisionId": "93017fc9-59cb-41dc-901b-4845ce4bf668",
            "CodeSize": 266,
            "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:$LATEST",
            "Handler": "index.handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/
helloWorldPython-role-uy3l9qyq",
            "Timeout": 3,
```

```
            "LastModified": "2019-10-01T16:47:28.490+0000",
            "Runtime": "nodejs10.x",
            "Description": ""
        },
        {
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "1",
            "CodeSha256": "5tT2qgzYUHoqwR616pZ2dpkn/0J1FrzJmlKidWaaCgk=",
            "FunctionName": "my-function",
            "VpcConfig": {
                "SubnetIds": [],
                "VpcId": "",
                "SecurityGroupIds": []
            },
            "MemorySize": 256,
            "RevisionId": "949c8914-012e-4795-998c-e467121951b1",
            "CodeSize": 304,
            "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:1",
            "Handler": "index.handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/
helloWorldPython-role-uy3l9qyq",
            "Timeout": 3,
            "LastModified": "2019-09-26T20:28:40.438+0000",
            "Runtime": "nodejs10.x",
            "Description": "new version"
        },
        {
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "2",
            "CodeSha256": "sU0cJ2/hOZevwV/lTxCuQqK3gDZP3i8gUoqUUVRmY6E=",
            "FunctionName": "my-function",
            "VpcConfig": {
                "SubnetIds": [],
                "VpcId": "",
                "SecurityGroupIds": []
            },
            "MemorySize": 256,
            "RevisionId": "cd669f21-0f3d-4e1c-9566-948837f2e2ea",
            "CodeSize": 266,
```

```
            "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:2",
            "Handler": "index.handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/
helloWorldPython-role-uy3l9qyq",
            "Timeout": 3,
            "LastModified": "2019-10-01T16:47:28.490+0000",
            "Runtime": "nodejs10.x",
            "Description": "newer version"
        }
    ]
}
```

如需詳細資訊，請參閱 AWS Lambda 開發人員指南中的設定AWS Lambda 函數別名。

- 如需 API 詳細資訊，請參閱AWS CLI 命令參考ListVersionsByFunction中的。

PowerShell

適用的工具 PowerShell

範例 1：此範例會傳回每個 Lambda 函數版本的特定版本組態清單。

```
Get-LMVersionsByFunction -FunctionName "MylambdaFunction123"
```

輸出：

```
FunctionName        Runtime   MemorySize Timeout CodeSize LastModified
      RoleName
------------        -------   ---------- ------- -------- ------------
      --------
MylambdaFunction123 python3.8        128     600      659
 2020-01-10T03:20:56.390+0000 lambda
MylambdaFunction123 python3.8        128       5     1426
 2019-12-25T09:19:02.238+0000 lambda
MylambdaFunction123 python3.8        128       5     1426
 2019-12-25T09:39:36.779+0000 lambda
MylambdaFunction123 python3.8        128     600     1426
 2019-12-25T09:52:59.872+0000 lambda
```

- 如需 API 詳細資訊，請參閱AWS Tools for PowerShell 指令程ListVersionsByFunction式參考中的。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭PublishVersion配 AWS 開發套件或 CLI 使用

下列程式碼範例會示範如何使用PublishVersion。

CLI

AWS CLI

若要發佈新版本的函數

下列publish-version範例會發佈新版本的 my-function Lambda 函數。

```
aws lambda publish-version \
    --function-name my-function
```

輸出：

```
{
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "CodeSha256": "dBG9m8SGdmlEjw/JYXlhhvCrAv5TxvXsbL/RMr0fT/I=",
    "FunctionName": "my-function",
    "CodeSize": 294,
    "RevisionId": "f31d3d39-cc63-4520-97d4-43cd44c94c20",
    "MemorySize": 128,
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:3",
    "Version": "2",
    "Role": "arn:aws:iam::123456789012:role/service-role/MyTestFunction-role-
zgur6bf4",
    "Timeout": 3,
    "LastModified": "2019-09-23T18:32:33.857+0000",
    "Handler": "my-function.handler",
    "Runtime": "nodejs10.x",
    "Description": ""
}
```

如需詳細資訊，請參閱 AWS Lambda 開發人員指南中的設定AWS Lambda 函數別名。

- 如需 API 詳細資訊，請參閱AWS CLI 命令參考PublishVersion中的。

PowerShell

適用的工具 PowerShell

範例 1：此範例會為 Lambda 函數程式碼的現有快照建立版本

```
Publish-LMVersion -FunctionName "MylambdaFunction123" -Description "Publishing
 Existing Snapshot of function code as a  new version through Powershell"
```

- 如需 API 詳細資訊，請參閱AWS Tools for PowerShell 指令程PublishVersion式參考中的。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭PutFunctionConcurrency配 AWS 開發套件或 CLI 使用

下列程式碼範例會示範如何使用PutFunctionConcurrency。

CLI

AWS CLI

若要設定函數的保留並行限制

下列put-function-concurrency範例會為函數設定 100 個保留的並行執行。my-function

```
aws lambda put-function-concurrency \
    --function-name  my-function  \
    --reserved-concurrent-executions 100
```

輸出：

```
{
    "ReservedConcurrentExecutions": 100
}
```

如需詳細資訊，請參閱 Lambda 開發人員指南中的保留 Lambda 函數的AWS 並行處理。

- 如需 API 詳細資訊，請參閱AWS CLI 命令參考中的PutFunction並行。

PowerShell

適用的工具 PowerShell

範例 1：此範例會整體套用「函數」的並行設定。

```
Write-LMFunctionConcurrency -FunctionName "MylambdaFunction123" -
ReservedConcurrentExecution 100
```

- 如需 API 詳細資訊，請參閱AWS Tools for PowerShell 指令程式參PutFunction考中的並行。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭PutProvisionedConcurrencyConfig配 AWS 開發套件或 CLI 使用

下列程式碼範例會示範如何使用PutProvisionedConcurrencyConfig。

CLI

AWS CLI

若要配置已佈建並行

下列put-provisioned-concurrency-config範例會為指定函數的BLUE別名配置 100 個佈建的並行。

```
aws lambda put-provisioned-concurrency-config \
    --function-name my-function \
    --qualifier BLUE \
    --provisioned-concurrent-executions 100
```

輸出：

```
{
    "Requested ProvisionedConcurrentExecutions": 100,
    "Allocated ProvisionedConcurrentExecutions": 0,
    "Status": "IN_PROGRESS",
```

```
    "LastModified": "2019-11-21T19:32:12+0000"
}
```

- 如需 API 詳細資訊，請參閱AWS CLI 命令參考PutProvisionedConcurrencyConfig中的。

PowerShell

適用的工具 PowerShell

範例 1：此範例將佈建的並行組態新增至函數的別名

```
Write-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
ProvisionedConcurrentExecution 20 -Qualifier "NewAlias1"
```

- 如需 API 詳細資訊，請參閱AWS Tools for PowerShell 指令程PutProvisionedConcurrencyConfig式參考中的。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭RemovePermission配 AWS 開發套件或 CLI 使用

下列程式碼範例會示範如何使用RemovePermission。

CLI

AWS CLI

若要從現有 Lambda 函數移除權限

下列remove-permission範例會移除叫用名為之函數的權限my-function。

```
aws lambda remove-permission \
    --function-name my-function \
    --statement-id sns
```

此命令不會產生輸出。

如需詳細資訊，請參閱 Lambda 開發人員指南中的針對 AWS Lambda 使用以資源為基礎的 AWS

- 如需 API 詳細資訊，請參閱AWS CLI 命令參考RemovePermission中的。

PowerShell

適用的工具 PowerShell

範例 1：此範例會移除指定 StatementId 的 Lambda 函數的函數政策。

```
$policy =  Get-LMPolicy -FunctionName "MylambdaFunction123" -Select Policy |
 ConvertFrom-Json| Select-Object -ExpandProperty Statement
Remove-LMPermission -FunctionName "MylambdaFunction123" -StatementId
 $policy[0].Sid
```

- 如需 API 詳細資訊，請參閱AWS Tools for PowerShell 指令程RemovePermission式參考中的。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭TagResource配 AWS 開發套件或 CLI 使用

下列程式碼範例會示範如何使用TagResource。

CLI

AWS CLI

若要將標籤新增至現有的 Lambda 函數

下列tag-resource範例會將含有索引鍵名稱DEPARTMENT和值的標籤新增Department A至指定的 Lambda 函數。

```
aws lambda tag-resource \
    --resource arn:aws:lambda:us-west-2:123456789012:function:my-function \
    --tags "DEPARTMENT=Department A"
```

此命令不會產生輸出。

如需詳細資訊，請參閱 Lambda 開發人員指南中的標記AWS Lambda 函數。

- 如需 API 詳細資訊，請參閱AWS CLI 命令參考TagResource中的。

PowerShell

適用的工具 PowerShell

範例 1：將三個標籤 (華盛頓、奧勒岡和加利福尼亞州) 及其關聯值新增至由其 ARN 識別的指定函數。

```
Add-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction" -Tag @{ "Washington" = "Olympia";
 "Oregon" = "Salem"; "California" = "Sacramento" }
```

- 如需 API 詳細資訊，請參閱AWS Tools for PowerShell 指令程TagResource式參考中的。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭UntagResource配 AWS 開發套件或 CLI 使用

下列程式碼範例會示範如何使用UntagResource。

CLI

AWS CLI

若要從現有 Lambda 函數移除標籤

下列untag-resource範例會從 my-function Lambda 函數移除含有索引鍵名稱DEPARTMENT標籤的標籤。

```
aws lambda untag-resource \
    --resource arn:aws:lambda:us-west-2:123456789012:function:my-function \
    --tag-keys DEPARTMENT
```

此命令不會產生輸出。

如需詳細資訊，請參閱 Lambda 開發人員指南中的標記AWS Lambda 函數。

- 如需 API 詳細資訊，請參閱AWS CLI 命令參考UntagResource中的。

PowerShell

適用的工具 PowerShell

例 1：從函數中刪除提供的標籤。除非指定-Force 參數，否則指令程式會在繼續之前提示確認。只要呼叫服務即可移除標籤。

```
Remove-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction" -TagKey
 "Washington","Oregon","California"
```

示例 2：從函數中刪除提供的標籤。除非指定-Force 參數，否則指令程式會在繼續之前提示確認。一旦調用服務是按照提供的標籤進行。

```
"Washington","Oregon","California" | Remove-LMResourceTag -Resource
 "arn:aws:lambda:us-west-2:123456789012:function:MyFunction"
```

- 如需 API 詳細資訊，請參閱AWS Tools for PowerShell 指令程UntagResource式參考中的。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭UpdateAlias配 AWS 開發套件或 CLI 使用

下列程式碼範例會示範如何使用UpdateAlias。

CLI

AWS CLI

更新函數別名

下列update-alias範例會將名稱為的別名LIVE更新為指向 my-function Lambda 函數的第 3 版。

```
aws lambda update-alias \
    --function-name my-function \
    --function-version 3 \
    --name LIVE
```

輸出：

```
{
    "FunctionVersion": "3",
    "Name": "LIVE",
    "AliasArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:LIVE",
    "RevisionId": "594f41fb-b85f-4c20-95c7-6ca5f2a92c93",
    "Description": "alias for live version of function"
}
```

如需詳細資訊，請參閱 AWS Lambda 開發人員指南中的設定AWS Lambda 函數別名。

- 如需 API 詳細資訊，請參閱AWS CLI 命令參考UpdateAlias中的。

PowerShell

適用的工具 PowerShell

範例 1：此範例會更新現有 Lambda 函數別名的組態。它更新了 RoutingConfiguration 值，將流量的 60%（0.6）轉移到版本 1

```
Update-LMAlias -FunctionName "MylambdaFunction123" -Description
 " Alias for version 2" -FunctionVersion 2 -Name "newlabel1" -
RoutingConfig_AdditionalVersionWeight @{Name="1";Value="0.6}
```

- 如需 API 詳細資訊，請參閱AWS Tools for PowerShell 指令程UpdateAlias式參考中的。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭UpdateFunctionCode配 AWS 開發套件或 CLI 使用

下列程式碼範例會示範如何使用UpdateFunctionCode。

動作範例是大型程式的程式碼摘錄，必須在內容中執行。您可以在下列程式碼範例的內容中看到此動作：

- 開始使用函數

.NET

AWS SDK for .NET

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
    /// <summary>
    /// Update an existing Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to update.</
param>
    /// <param name="bucketName">The bucket where the zip file containing
    /// the Lambda function code is stored.</param>
    /// <param name="key">The key name of the source code file.</param>
    /// <returns>Async Task.</returns>
    public async Task UpdateFunctionCodeAsync(
        string functionName,
        string bucketName,
        string key)
    {
        var functionCodeRequest = new UpdateFunctionCodeRequest
        {
            FunctionName = functionName,
            Publish = true,
            S3Bucket = bucketName,
            S3Key = key,
        };

        var response = await
 _lambdaService.UpdateFunctionCodeAsync(functionCodeRequest);
        Console.WriteLine($"The Function was last modified at
 {response.LastModified}.");
    }
```

- 如需 API 詳細資訊，請參閱 AWS SDK for .NET API 參考中的UpdateFunction程式碼。

C++

適用於 C++ 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region in which the bucket was created
 (overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::Lambda::LambdaClient client(clientConfig);

        Aws::Lambda::Model::UpdateFunctionCodeRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        std::ifstream ifstream(CALCULATOR_LAMBDA_CODE.c_str(),
                               std::ios_base::in | std::ios_base::binary);
        if (!ifstream.is_open()) {
            std::cerr << "Error opening file " << INCREMENT_LAMBDA_CODE << "." <<
 std::endl;

#if USE_CPP_LAMBDA_FUNCTION
            std::cerr
                    << "The cpp Lambda function must be built following the
 instructions in the cpp_lambda/README.md file. "
                    << std::endl;
#endif
            deleteLambdaFunction(client);
            deleteIamRole(clientConfig);
            return false;
        }

        Aws::StringStream buffer;
        buffer << ifstream.rdbuf();
        request.SetZipFile(
                Aws::Utils::ByteBuffer((unsigned char *) buffer.str().c_str(),
                                       buffer.str().length()));
        request.SetPublish(true);
```

```
        Aws::Lambda::Model::UpdateFunctionCodeOutcome outcome =
 client.UpdateFunctionCode(
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda code was successfully updated." <<
 std::endl;
        }
        else {
            std::cerr << "Error with Lambda::UpdateFunctionCode. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }
```

- 如需 API 詳細資訊，請參閱 AWS SDK for C++ API 參考中的UpdateFunction程式碼。

CLI

AWS CLI

若要更新 Lambda 函數的程式碼

下列 update-function-code 範例會使用指定 zip 檔案的內容替換 my-function 函數未發布 ($LATEST) 版本的程式碼。

```
aws lambda update-function-code \
    --function-name  my-function \
    --zip-file fileb://my-function.zip
```

輸出：

```
{
    "FunctionName": "my-function",
    "LastModified": "2019-09-26T20:28:40.438+0000",
    "RevisionId": "e52502d4-9320-4688-9cd6-152a6ab7490d",
    "MemorySize": 256,
    "Version": "$LATEST",
    "Role": "arn:aws:iam::123456789012:role/service-role/my-function-role-
uy3l9qyq",
    "Timeout": 3,
```

```
    "Runtime": "nodejs10.x",
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "CodeSha256": "5tT2qgzYUHaqwR716pZ2dpkn/0J1FrzJmlKidWoaCgk=",
    "Description": "",
    "VpcConfig": {
        "SubnetIds": [],
        "VpcId": "",
        "SecurityGroupIds": []
    },
    "CodeSize": 304,
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "Handler": "index.handler"
}
```

如需詳細資訊，請參閱《AWS Lambda 開發人員指南》中的 AWS Lambda 函數組態。

- 如需 API 詳細資訊，請參閱AWS CLI 命令參考中的UpdateFunction程式碼。

Go

SDK for Go V2

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// UpdateFunctionCode updates the code for the Lambda function specified by
 functionName.
```

```
// The existing code for the Lambda function is entirely replaced by the code in
 the
// zipPackage buffer. After the update action is called, a
 lambda.FunctionUpdatedV2Waiter
// is used to wait until the update is successful.
func (wrapper FunctionWrapper) UpdateFunctionCode(functionName string, zipPackage
 *bytes.Buffer) types.State {
 var state types.State
 _, err := wrapper.LambdaClient.UpdateFunctionCode(context.TODO(),
 &lambda.UpdateFunctionCodeInput{
  FunctionName: aws.String(functionName), ZipFile: zipPackage.Bytes(),
 })
 if err != nil {
  log.Panicf("Couldn't update code for function %v. Here's why: %v\n",
 functionName, err)
 } else {
  waiter := lambda.NewFunctionUpdatedV2Waiter(wrapper.LambdaClient)
  funcOutput, err := waiter.WaitForOutput(context.TODO(),
 &lambda.GetFunctionInput{
   FunctionName: aws.String(functionName)}, 1*time.Minute)
  if err != nil {
   log.Panicf("Couldn't wait for function %v to be active. Here's why: %v\n",
 functionName, err)
  } else {
   state = funcOutput.Configuration.State
  }
 }
 return state
}
```

- 如需 API 詳細資訊，請參閱 AWS SDK for Go API 參考中的UpdateFunction程式碼。

JavaScript

適用於 JavaScript (v3) 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
const updateFunctionCode = async (funcName, newFunc) => {
  const client = new LambdaClient({});
  const code = await readFile(`${dirname}../functions/${newFunc}.zip`);
  const command = new UpdateFunctionCodeCommand({
    ZipFile: code,
    FunctionName: funcName,
    Architectures: [Architecture.arm64],
    Handler: "index.handler", // Required when sending a .zip file
    PackageType: PackageType.Zip, // Required when sending a .zip file
    Runtime: Runtime.nodejs16x, // Required when sending a .zip file
  });

  return client.send(command);
};
```

- 如需 API 詳細資訊，請參閱 AWS SDK for JavaScript API 參考中的UpdateFunction程式碼。

PHP

適用於 PHP 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
    public function updateFunctionCode($functionName, $s3Bucket, $s3Key)
    {
        return $this->lambdaClient->updateFunctionCode([
            'FunctionName' => $functionName,
            'S3Bucket' => $s3Bucket,
            'S3Key' => $s3Key,
        ]);
    }
```

- 如需 API 詳細資訊，請參閱 AWS SDK for PHP API 參考中的UpdateFunction程式碼。

PowerShell

適用的工具 PowerShell

範例 1：以指定 zip 檔案中包含的新內容更新名為 MyFunction " 的函數。對於 C # .NET 核心 Lambda 函數，zip 文件應該包含已編譯的程序集。

```
Update-LMFunctionCode -FunctionName MyFunction -ZipFilename .\UpdatedCode.zip
```

範例 2：此範例與前一個範例類似，但使用包含已更新程式碼的 Amazon S3 物件來更新函數。

```
Update-LMFunctionCode -FunctionName MyFunction -BucketName mybucket -Key
 UpdatedCode.zip
```

- 如需 API 詳細資訊，請參閱AWS Tools for PowerShell 指令程式參考中的程式 UpdateFunction碼。

Python

適用於 Python (Boto3) 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def update_function_code(self, function_name, deployment_package):
        """
        Updates the code for a Lambda function by submitting a .zip archive that
 contains
        the code for the function.
```

```
        :param function_name: The name of the function to update.
        :param deployment_package: The function code to update, packaged as bytes
in
                                   .zip format.
        :return: Data about the update, including the status.
        """
        try:
            response = self.lambda_client.update_function_code(
                FunctionName=function_name, ZipFile=deployment_package
            )
        except ClientError as err:
            logger.error(
                "Couldn't update function %s. Here's why: %s: %s",
                function_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response
```

- 如需 API 詳細資訊，請參閱AWS 開發套件中的UpdateFunction程式碼 (Boto3) API 參考。

Ruby

適用於 Ruby 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
class LambdaWrapper
  attr_accessor :lambda_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
```

```
 end

 # Updates the code for a Lambda function by submitting a .zip archive that
contains
 # the code for the function.

 # @param function_name: The name of the function to update.
 # @param deployment_package: The function code to update, packaged as bytes in
 #                            .zip format.
 # @return: Data about the update, including the status.
 def update_function_code(function_name, deployment_package)
   @lambda_client.update_function_code(
     function_name: function_name,
     zip_file: deployment_package
   )
   @lambda_client.wait_until(:function_updated_v2, { function_name:
function_name}) do |w|
     w.max_attempts = 5
     w.delay = 5
   end
 rescue Aws::Lambda::Errors::ServiceException => e
   @logger.error("There was an error updating function code for:
#{function_name}:\n #{e.message}")
   nil
 rescue Aws::Waiters::Errors::WaiterFailed => e
   @logger.error("Failed waiting for #{function_name} to update:\n
#{e.message}")
 end
```

- 如需 API 詳細資訊，請參閱 AWS SDK for Ruby API 參考中的UpdateFunction程式碼。

Rust

適用於 Rust 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
    /** Given a Path to a zip file, update the function's code and wait for the
 update to finish. */
    pub async fn update_function_code(
        &self,
        zip_file: PathBuf,
        key: String,
    ) -> Result<UpdateFunctionCodeOutput, anyhow::Error> {
        let function_code = self.prepare_function(zip_file, Some(key)).await?;

        info!("Updating code for {}", self.lambda_name);
        let update = self
            .lambda_client
            .update_function_code()
            .function_name(self.lambda_name.clone())
            .s3_bucket(self.bucket.clone())
            .s3_key(function_code.s3_key().unwrap().to_string())
            .send()
            .await
            .map_err(anyhow::Error::from)?;

        self.wait_for_function_ready().await?;

        Ok(update)
    }

    /**
     * Upload function code from a path to a zip file.
     * The zip file must have an AL2 Linux-compatible binary called `bootstrap`.
     * The easiest way to create such a zip is to use `cargo lambda build --
output-format Zip`.
     */
    async fn prepare_function(
        &self,
        zip_file: PathBuf,
        key: Option<String>,
    ) -> Result<FunctionCode, anyhow::Error> {
        let body = ByteStream::from_path(zip_file).await?;

        let key = key.unwrap_or_else(|| format!("{}_code", self.lambda_name));

        info!("Uploading function code to s3://{}/{}", self.bucket, key);
        let _ = self
            .s3_client
```

```
            .put_object()
            .bucket(self.bucket.clone())
            .key(key.clone())
            .body(body)
            .send()
            .await?;

        Ok(FunctionCode::builder()
            .s3_bucket(self.bucket.clone())
            .s3_key(key)
            .build())
    }
```

- 如需 API 的詳細資訊，請參閱 AWS SDK 中的UpdateFunction程式碼以取得 Rust API 參考。

SAP ABAP

適用於 SAP ABAP 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
    TRY.
        oo_result = lo_lmd->updatefunctioncode(     " oo_result is returned for
testing purposes. "
              iv_functionname = iv_function_name
              iv_zipfile = io_zip_file
          ).

        MESSAGE 'Lambda function code updated.' TYPE 'I'.
      CATCH /aws1/cx_lmdcodesigningcfgno00.
        MESSAGE 'Code signing configuration does not exist.' TYPE 'E'.
      CATCH /aws1/cx_lmdcodestorageexcdex.
        MESSAGE 'Maximum total code size per account exceeded.' TYPE 'E'.
      CATCH /aws1/cx_lmdcodeverification00.
        MESSAGE 'Code signature failed one or more validation checks for
signature mismatch or expiration.' TYPE 'E'.
      CATCH /aws1/cx_lmdinvalidcodesigex.
```

```
        MESSAGE 'Code signature failed the integrity check.' TYPE 'E'.
      CATCH /aws1/cx_lmdinvparamvalueex.
        MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
      CATCH /aws1/cx_lmdresourceconflictex.
        MESSAGE 'Resource already exists or another operation is in progress.'
TYPE 'E'.
      CATCH /aws1/cx_lmdresourcenotfoundex.
        MESSAGE 'The requested resource does not exist.' TYPE 'E'.
      CATCH /aws1/cx_lmdserviceexception.
        MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
TYPE 'E'.
      CATCH /aws1/cx_lmdtoomanyrequestsex.
        MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
    ENDTRY.
```

- 如需 API 詳細資訊，請參閱 AWS SDK 中的UpdateFunction程式碼，以取得 SAP ABAP API 參考資料。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭UpdateFunctionConfiguration配 AWS 開發套件或 CLI 使用

下列程式碼範例會示範如何使用UpdateFunctionConfiguration。

動作範例是大型程式的程式碼摘錄，必須在內容中執行。您可以在下列程式碼範例的內容中看到此動作：

- 開始使用函數

.NET

AWS SDK for .NET

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
    /// <summary>
    /// Update the code of a Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the function to update.</param>
    /// <param name="functionHandler">The code that performs the function's
actions.</param>
    /// <param name="environmentVariables">A dictionary of environment
variables.</param>
    /// <returns>A Boolean value indicating the success of the action.</returns>
    public async Task<bool> UpdateFunctionConfigurationAsync(
        string functionName,
        string functionHandler,
        Dictionary<string, string> environmentVariables)
    {
        var request = new UpdateFunctionConfigurationRequest
        {
            Handler = functionHandler,
            FunctionName = functionName,
            Environment = new Amazon.Lambda.Model.Environment { Variables =
environmentVariables },
        };

        var response = await
_lambdaService.UpdateFunctionConfigurationAsync(request);

        Console.WriteLine(response.LastModified);

        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }
```

- 有關 API 詳細信息，請參閱 AWS SDK for .NET API 參考中的UpdateFunction配置。

C++

適用於 C++ 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
       Aws::Client::ClientConfiguration clientConfig;
       // Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
       // clientConfig.region = "us-east-1";

   Aws::Lambda::LambdaClient client(clientConfig);

       Aws::Lambda::Model::UpdateFunctionConfigurationRequest request;
       request.SetFunctionName(LAMBDA_NAME);
       Aws::Lambda::Model::Environment environment;
       environment.AddVariables("LOG_LEVEL", "DEBUG");
       request.SetEnvironment(environment);

       Aws::Lambda::Model::UpdateFunctionConfigurationOutcome outcome =
client.UpdateFunctionConfiguration(
               request);

       if (outcome.IsSuccess()) {
           std::cout << "The lambda configuration was successfully updated."
                     << std::endl;
           break;
       }

       else {
           std::cerr << "Error with Lambda::UpdateFunctionConfiguration. "
                     << outcome.GetError().GetMessage()
                     << std::endl;
       }
```

- 有關 API 詳細信息，請參閱 AWS SDK for C++ API 參考中的UpdateFunction配置。

CLI

AWS CLI

若要修改函數的組態

下列 update-function-configuration 範例會將 my-function 函數未發布 ($LATEST) 版本的記憶體大小修改為 256 MB。

```
aws lambda update-function-configuration \
    --function-name  my-function \
    --memory-size 256
```

輸出：

```
{
    "FunctionName": "my-function",
    "LastModified": "2019-09-26T20:28:40.438+0000",
    "RevisionId": "e52502d4-9320-4688-9cd6-152a6ab7490d",
    "MemorySize": 256,
    "Version": "$LATEST",
    "Role": "arn:aws:iam::123456789012:role/service-role/my-function-role-
uy3l9qyq",
    "Timeout": 3,
    "Runtime": "nodejs10.x",
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "CodeSha256": "5tT2qgzYUHaqwR716pZ2dpkn/0J1FrzJmlKidWoaCgk=",
    "Description": "",
    "VpcConfig": {
        "SubnetIds": [],
        "VpcId": "",
        "SecurityGroupIds": []
    },
    "CodeSize": 304,
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "Handler": "index.handler"
}
```

如需詳細資訊，請參閱《AWS Lambda 開發人員指南》中的 AWS Lambda 函數組態。

- 如需 API 詳細資訊，請參閱AWS CLI 命令參考中的UpdateFunction組態。

Go

SDK for Go V2

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// UpdateFunctionConfiguration updates a map of environment variables configured
 for
// the Lambda function specified by functionName.
func (wrapper FunctionWrapper) UpdateFunctionConfiguration(functionName string,
 envVars map[string]string) {
 _, err := wrapper.LambdaClient.UpdateFunctionConfiguration(context.TODO(),
 &lambda.UpdateFunctionConfigurationInput{
  FunctionName: aws.String(functionName),
  Environment:  &types.Environment{Variables: envVars},
 })
 if err != nil {
  log.Panicf("Couldn't update configuration for %v. Here's why: %v",
 functionName, err)
 }
}
```

- 有關 API 詳細信息，請參閱 AWS SDK for Go API 參考中的UpdateFunction配置。

JavaScript

適用於 JavaScript (v3) 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
const updateFunctionConfiguration = (funcName) => {
  const client = new LambdaClient({});
  const config = readFileSync(`${dirname}../functions/config.json`).toString();
  const command = new UpdateFunctionConfigurationCommand({
    ...JSON.parse(config),
    FunctionName: funcName,
  });
  return client.send(command);
};
```

- 有關 API 詳細信息，請參閱 AWS SDK for JavaScript API 參考中的UpdateFunction配置。

PHP

適用於 PHP 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
    public function updateFunctionConfiguration($functionName, $handler,
 $environment = '')
    {
        return $this->lambdaClient->updateFunctionConfiguration([
            'FunctionName' => $functionName,
            'Handler' => "$handler.lambda_handler",
            'Environment' => $environment,
```

```
        ]);
    }
```

- 有關 API 詳細信息，請參閱 AWS SDK for PHP API 參考中的UpdateFunction配置。

PowerShell

適用的工具 PowerShell

範例 1：此範例更新現有的 Lambda 函數組態

```
Update-LMFunctionConfiguration -FunctionName "MylambdaFunction123" -Handler
 "lambda_function.launch_instance" -Timeout 600 -Environment_Variable
 @{ "envvar1"="value";"envvar2"="value" } -Role arn:aws:iam::123456789101:role/
service-role/lambda -DeadLetterConfig_TargetArn arn:aws:sns:us-east-1:
 123456789101:MyfirstTopic
```

- 如需 API 詳細資訊，請參閱AWS Tools for PowerShell 指令程式參考中的UpdateFunction組態。

Python

適用於 Python (Boto3) 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def update_function_configuration(self, function_name, env_vars):
        """
        Updates the environment variables for a Lambda function.
```

```
        :param function_name: The name of the function to update.
        :param env_vars: A dict of environment variables to update.
        :return: Data about the update, including the status.
        """
        try:
            response = self.lambda_client.update_function_configuration(
                FunctionName=function_name, Environment={"Variables": env_vars}
            )
        except ClientError as err:
            logger.error(
                "Couldn't update function configuration %s. Here's why: %s: %s",
                function_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response
```

- 如需 API 的詳細資訊，請參閱在 AWS SDK 中進行UpdateFunction設定 (Boto3) API 參考。

Ruby

適用於 Ruby 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
class LambdaWrapper
  attr_accessor :lambda_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end
```

```
 # Updates the environment variables for a Lambda function.
 # @param function_name: The name of the function to update.
 # @param log_level: The log level of the function.
 # @return: Data about the update, including the status.
 def update_function_configuration(function_name, log_level)
   @lambda_client.update_function_configuration({
                                                  function_name: function_name,
                                                  environment: {
                                                    variables: {
                                                      "LOG_LEVEL" => log_level
                                                    }
                                                  }
                                                })
   @lambda_client.wait_until(:function_updated_v2, { function_name:
function_name}) do |w|
     w.max_attempts = 5
     w.delay = 5
   end
 rescue Aws::Lambda::Errors::ServiceException => e
   @logger.error("There was an error updating configurations for
#{function_name}:\n #{e.message}")
 rescue Aws::Waiters::Errors::WaiterFailed => e
   @logger.error("Failed waiting for #{function_name} to activate:\n
#{e.message}")
 end
```

- 有關 API 詳細信息，請參閱 AWS SDK for Ruby API 參考中的UpdateFunction配置。

Rust

適用於 Rust 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
   /** Update the environment for a function. */
   pub async fn update_function_configuration(
```

```
        &self,
        environment: Environment,
    ) -> Result<UpdateFunctionConfigurationOutput, anyhow::Error> {
        info!(
            ?environment,
            "Updating environment for {}", self.lambda_name
        );
        let updated = self
            .lambda_client
            .update_function_configuration()
            .function_name(self.lambda_name.clone())
            .environment(environment)
            .send()
            .await
            .map_err(anyhow::Error::from)?;

        self.wait_for_function_ready().await?;

        Ok(updated)
    }
```

- 如需 API 的詳細資訊，請參閱 AWS SDK 中的UpdateFunction設定以取得 Rust API 參考資料

SAP ABAP

適用於 SAP ABAP 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
    TRY.
        oo_result = lo_lmd->updatefunctionconfiguration(     " oo_result is
returned for testing purposes. "
              iv_functionname = iv_function_name
              iv_runtime = iv_runtime
              iv_description  = 'Updated Lambda function'
              iv_memorysize  = iv_memory_size
          ).
```

```
        MESSAGE 'Lambda function configuration/settings updated.' TYPE 'I'.
      CATCH /aws1/cx_lmdcodesigningcfgno00.
        MESSAGE 'Code signing configuration does not exist.' TYPE 'E'.
      CATCH /aws1/cx_lmdcodeverification00.
        MESSAGE 'Code signature failed one or more validation checks for
signature mismatch or expiration.' TYPE 'E'.
      CATCH /aws1/cx_lmdinvalidcodesigex.
        MESSAGE 'Code signature failed the integrity check.' TYPE 'E'.
      CATCH /aws1/cx_lmdinvparamvalueex.
        MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
      CATCH /aws1/cx_lmdresourceconflictex.
        MESSAGE 'Resource already exists or another operation is in progress.'
TYPE 'E'.
      CATCH /aws1/cx_lmdresourcenotfoundex.
        MESSAGE 'The requested resource does not exist.' TYPE 'E'.
      CATCH /aws1/cx_lmdserviceexception.
        MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
TYPE 'E'.
      CATCH /aws1/cx_lmdtoomanyrequestsex.
        MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
    ENDTRY.
```

- 如需 API 詳細資訊，請UpdateFunction參閱 AWS SDK 中針對 SAP ABAP API 參考資料的設定。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使用 AWS 開發套件的 Lambda 案例

下列程式碼範例說明如何使用 AWS SDK 在 Lambda 中實作常見案例。這些案例會向您展示如何呼叫 Lambda 中的多個函數來完成特定任務。每個案例都包含一個連結 GitHub，您可以在其中找到如何設定和執行程式碼的指示。

範例

- 使用開發套件，透過 Lambda 函數自動確認已知的 Amazon Cognito 認知使用者 AWS
- 使用開發套件，透過 Lambda 函數自動遷移已知的 Amazon Cognito 知使用者 AWS
- 開始使用開發套件 AWS 建立和叫用 Lambda 函數

- 使用開發套件 AWS 進行 Amazon Cognito 使用者身份驗證之後，使用 Lambda 函數寫入自訂活動

使用開發套件，透過 Lambda 函數自動確認已知的 Amazon Cognito 認知使用者 AWS

下列程式碼範例顯示如何使用 Lambda 函數自動確認已知的 Amazon Cognito 使用者。

- 設定使用者集區以呼叫PreSignUp觸發器的 Lambda 函數。
- 使用亞馬遜認可註冊用戶。
- Lambda 函數會掃描 DynamoDB 表格，並自動確認已知使用者。
- 以新使用者身分登入，然後清理資源。

Go

SDK for Go V2

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

在命令提示中執行互動式案例。

```
// AutoConfirm separates the steps of this scenario into individual functions so
 that
// they are simpler to read and understand.
type AutoConfirm struct {
 helper       IScenarioHelper
 questioner   demotools.IQuestioner
 resources    Resources
 cognitoActor *actions.CognitoActions
}

// NewAutoConfirm constructs a new auto confirm runner.
func NewAutoConfirm(sdkConfig aws.Config, questioner demotools.IQuestioner,
 helper IScenarioHelper) AutoConfirm {
 scenario := AutoConfirm{
```

```
  helper:       helper,
  questioner:   questioner,
  resources:    Resources{},
  cognitoActor: &actions.CognitoActions{CognitoClient:
 cognitoidentityprovider.NewFromConfig(sdkConfig)},
 }
 scenario.resources.init(scenario.cognitoActor, questioner)
 return scenario
}

// AddPreSignUpTrigger adds a Lambda handler as an invocation target for the
 PreSignUp trigger.
func (runner *AutoConfirm) AddPreSignUpTrigger(userPoolId string, functionArn
 string) {
 log.Printf("Let's add a Lambda function to handle the PreSignUp trigger from
 Cognito.\n" +
  "This trigger happens when a user signs up, and lets your function take action
 before the main Cognito\n" +
  "sign up processing occurs.\n")
 err := runner.cognitoActor.UpdateTriggers(
  userPoolId,
  actions.TriggerInfo{Trigger: actions.PreSignUp, HandlerArn:
 aws.String(functionArn)})
 if err != nil {
  panic(err)
 }
 log.Printf("Lambda function %v added to user pool %v to handle the PreSignUp
 trigger.\n",
  functionArn, userPoolId)
}

// SignUpUser signs up a user from the known user table with a password you
 specify.
func (runner *AutoConfirm) SignUpUser(clientId string, usersTable string)
 (string, string) {
 log.Println("Let's sign up a user to your Cognito user pool. When the user's
 email matches an email in the\n" +
  "DynamoDB known users table, it is automatically verified and the user is
 confirmed.")

 knownUsers, err := runner.helper.GetKnownUsers(usersTable)
 if err != nil {
  panic(err)
 }
```

```
 userChoice := runner.questioner.AskChoice("Which user do you want to use?\n",
 knownUsers.UserNameList())
 user := knownUsers.Users[userChoice]

 var signedUp bool
 var userConfirmed bool
 password := runner.questioner.AskPassword("Enter a password that has at least
 eight characters, uppercase, lowercase, numbers and symbols.\n"+
  "(the password will not display as you type):", 8)
 for !signedUp {
  log.Printf("Signing up user '%v' with email '%v' to Cognito.\n", user.UserName,
 user.UserEmail)
  userConfirmed, err = runner.cognitoActor.SignUp(clientId, user.UserName,
 password, user.UserEmail)
  if err != nil {
   var invalidPassword *types.InvalidPasswordException
   if errors.As(err, &invalidPassword) {
    password = runner.questioner.AskPassword("Enter another password:", 8)
   } else {
    panic(err)
   }
  } else {
   signedUp = true
  }
 }
 log.Printf("User %v signed up, confirmed = %v.\n", user.UserName, userConfirmed)

 log.Println(strings.Repeat("-", 88))

 return user.UserName, password
}

// SignInUser signs in a user.
func (runner *AutoConfirm) SignInUser(clientId string, userName string, password
 string) string {
 runner.questioner.Ask("Press Enter when you're ready to continue.")
 log.Printf("Let's sign in as %v...\n", userName)
 authResult, err := runner.cognitoActor.SignIn(clientId, userName, password)
 if err != nil {
  panic(err)
 }
 log.Printf("Successfully signed in. Your access token starts with: %v...\n",
 (*authResult.AccessToken)[:10])
 log.Println(strings.Repeat("-", 88))
```

```
 return *authResult.AccessToken
}

// Run runs the scenario.
func (runner *AutoConfirm) Run(stackName string) {
 defer func() {
  if r := recover(); r != nil {
   log.Println("Something went wrong with the demo.")
   runner.resources.Cleanup()
  }
 }()

 log.Println(strings.Repeat("-", 88))
 log.Printf("Welcome\n")

 log.Println(strings.Repeat("-", 88))

 stackOutputs, err := runner.helper.GetStackOutputs(stackName)
 if err != nil {
  panic(err)
 }
 runner.resources.userPoolId = stackOutputs["UserPoolId"]
 runner.helper.PopulateUserTable(stackOutputs["TableName"])

 runner.AddPreSignUpTrigger(stackOutputs["UserPoolId"],
 stackOutputs["AutoConfirmFunctionArn"])
 runner.resources.triggers = append(runner.resources.triggers, actions.PreSignUp)
 userName, password := runner.SignUpUser(stackOutputs["UserPoolClientId"],
 stackOutputs["TableName"])
 runner.helper.ListRecentLogEvents(stackOutputs["AutoConfirmFunction"])
 runner.resources.userAccessTokens = append(runner.resources.userAccessTokens,
  runner.SignInUser(stackOutputs["UserPoolClientId"], userName, password))

 runner.resources.Cleanup()

 log.Println(strings.Repeat("-", 88))
 log.Println("Thanks for watching!")
 log.Println(strings.Repeat("-", 88))
}
```

使用 Lambda 函數處理`PreSignUp`觸發器。

```
const TABLE_NAME = "TABLE_NAME"

// UserInfo defines structured user data that can be marshalled to a DynamoDB
 format.
type UserInfo struct {
 UserName  string `dynamodbav:"UserName"`
 UserEmail string `dynamodbav:"UserEmail"`
}

// GetKey marshals the user email value to a DynamoDB key format.
func (user UserInfo) GetKey() map[string]dynamodbtypes.AttributeValue {
 userEmail, err := attributevalue.Marshal(user.UserEmail)
 if err != nil {
  panic(err)
 }
 return map[string]dynamodbtypes.AttributeValue{"UserEmail": userEmail}
}

type handler struct {
 dynamoClient *dynamodb.Client
}

// HandleRequest handles the PreSignUp event by looking up a user in an Amazon
 DynamoDB table and
// specifying whether they should be confirmed and verified.
func (h *handler) HandleRequest(ctx context.Context, event
 events.CognitoEventUserPoolsPreSignup) (events.CognitoEventUserPoolsPreSignup,
 error) {
 log.Printf("Received presignup from %v for user '%v'", event.TriggerSource,
 event.UserName)
 if event.TriggerSource != "PreSignUp_SignUp" {
  // Other trigger sources, such as PreSignUp_AdminInitiateAuth, ignore the
 response from this handler.
  return event, nil
 }
 tableName := os.Getenv(TABLE_NAME)
 user := UserInfo{
  UserEmail: event.Request.UserAttributes["email"],
 }
 log.Printf("Looking up email %v in table %v.\n", user.UserEmail, tableName)
 output, err := h.dynamoClient.GetItem(ctx, &dynamodb.GetItemInput{
  Key:       user.GetKey(),
```

```
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Error looking up email %v.\n", user.UserEmail)
  return event, err
 }
 if output.Item == nil {
  log.Printf("Email %v not found. Email verification is required.\n",
 user.UserEmail)
  return event, err
 }

 err = attributevalue.UnmarshalMap(output.Item, &user)
 if err != nil {
  log.Printf("Couldn't unmarshal DynamoDB item. Here's why: %v\n", err)
  return event, err
 }

 if user.UserName != event.UserName {
  log.Printf("UserEmail %v found, but stored UserName '%v' does not match
 supplied UserName '%v'. Verification is required.\n",
   user.UserEmail, user.UserName, event.UserName)
 } else {
  log.Printf("UserEmail %v found with matching UserName %v. User is confirmed.
\n", user.UserEmail, user.UserName)
  event.Response.AutoConfirmUser = true
  event.Response.AutoVerifyEmail = true
 }

 return event, err
}

func main() {
 sdkConfig, err := config.LoadDefaultConfig(context.TODO())
 if err != nil {
  log.Panicln(err)
 }
 h := handler{
  dynamoClient: dynamodb.NewFromConfig(sdkConfig),
 }
 lambda.Start(h.HandleRequest)
}
```

建立執行一般工作的結構。

```
// IScenarioHelper defines common functions used by the workflows in this
 example.
type IScenarioHelper interface {
 Pause(secs int)
 GetStackOutputs(stackName string) (actions.StackOutputs, error)
 PopulateUserTable(tableName string)
 GetKnownUsers(tableName string) (actions.UserList, error)
 AddKnownUser(tableName string, user actions.User)
 ListRecentLogEvents(functionName string)
}

// ScenarioHelper contains AWS wrapper structs used by the workflows in this
 example.
type ScenarioHelper struct {
 questioner  demotools.IQuestioner
 dynamoActor *actions.DynamoActions
 cfnActor    *actions.CloudFormationActions
 cwlActor    *actions.CloudWatchLogsActions
 isTestRun   bool
}

// NewScenarioHelper constructs a new scenario helper.
func NewScenarioHelper(sdkConfig aws.Config, questioner demotools.IQuestioner)
 ScenarioHelper {
 scenario := ScenarioHelper{
  questioner:  questioner,
  dynamoActor: &actions.DynamoActions{DynamoClient:
 dynamodb.NewFromConfig(sdkConfig)},
  cfnActor:    &actions.CloudFormationActions{CfnClient:
 cloudformation.NewFromConfig(sdkConfig)},
  cwlActor:    &actions.CloudWatchLogsActions{CwlClient:
 cloudwatchlogs.NewFromConfig(sdkConfig)},
 }
 return scenario
}

// Pause waits for the specified number of seconds.
func (helper ScenarioHelper) Pause(secs int) {
```

```
 if !helper.isTestRun {
  time.Sleep(time.Duration(secs) * time.Second)
 }
}

// GetStackOutputs gets the outputs from the specified CloudFormation stack in a
 structured format.
func (helper ScenarioHelper) GetStackOutputs(stackName string)
 (actions.StackOutputs, error) {
 return helper.cfnActor.GetOutputs(stackName), nil
}

// PopulateUserTable fills the known user table with example data.
func (helper ScenarioHelper) PopulateUserTable(tableName string) {
 log.Printf("First, let's add some users to the DynamoDB %v table we'll use for
 this example.\n", tableName)
 err := helper.dynamoActor.PopulateTable(tableName)
 if err != nil {
  panic(err)
 }
}

// GetKnownUsers gets the users from the known users table in a structured
 format.
func (helper ScenarioHelper) GetKnownUsers(tableName string) (actions.UserList,
 error) {
 knownUsers, err := helper.dynamoActor.Scan(tableName)
 if err != nil {
  log.Printf("Couldn't get known users from table %v. Here's why: %v\n",
 tableName, err)
 }
 return knownUsers, err
}

// AddKnownUser adds a user to the known users table.
func (helper ScenarioHelper) AddKnownUser(tableName string, user actions.User) {
 log.Printf("Adding user '%v' with email '%v' to the DynamoDB known users
 table...\n",
  user.UserName, user.UserEmail)
 err := helper.dynamoActor.AddUser(tableName, user)
 if err != nil {
  panic(err)
 }
}
```

```
// ListRecentLogEvents gets the most recent log stream and events for the
 specified Lambda function and displays them.
func (helper ScenarioHelper) ListRecentLogEvents(functionName string) {
 log.Println("Waiting a few seconds to let Lambda write to CloudWatch Logs...")
 helper.Pause(10)
 log.Println("Okay, let's check the logs to find what's happened recently with
 your Lambda function.")
 logStream, err := helper.cwlActor.GetLatestLogStream(functionName)
 if err != nil {
  panic(err)
 }
 log.Printf("Getting some recent events from log stream %v\n",
 *logStream.LogStreamName)
 events, err := helper.cwlActor.GetLogEvents(functionName,
 *logStream.LogStreamName, 10)
 if err != nil {
  panic(err)
 }
 for _, event := range events {
  log.Printf("\t%v", *event.Message)
 }
 log.Println(strings.Repeat("-", 88))
}
```

創建一個包裝 Amazon Cognito 操作的結構。

```
type CognitoActions struct {
 CognitoClient *cognitoidentityprovider.Client
}



// Trigger and TriggerInfo define typed data for updating an Amazon Cognito
 trigger.
type Trigger int

const (
 PreSignUp Trigger = iota
```

```
 UserMigration
 PostAuthentication
)

type TriggerInfo struct {
 Trigger    Trigger
 HandlerArn *string
}

// UpdateTriggers adds or removes Lambda triggers for a user pool. When a trigger
 is specified with a `nil` value,
// it is removed from the user pool.
func (actor CognitoActions) UpdateTriggers(userPoolId string,
 triggers ...TriggerInfo) error {
 output, err := actor.CognitoClient.DescribeUserPool(context.TODO(),
 &cognitoidentityprovider.DescribeUserPoolInput{
  UserPoolId: aws.String(userPoolId),
 })
 if err != nil {
  log.Printf("Couldn't get info about user pool %v. Here's why: %v\n",
 userPoolId, err)
  return err
 }
 lambdaConfig := output.UserPool.LambdaConfig
 for _, trigger := range triggers {
  switch trigger.Trigger {
  case PreSignUp:
   lambdaConfig.PreSignUp = trigger.HandlerArn
  case UserMigration:
   lambdaConfig.UserMigration = trigger.HandlerArn
  case PostAuthentication:
   lambdaConfig.PostAuthentication = trigger.HandlerArn
  }
 }
 _, err = actor.CognitoClient.UpdateUserPool(context.TODO(),
 &cognitoidentityprovider.UpdateUserPoolInput{
  UserPoolId:   aws.String(userPoolId),
  LambdaConfig: lambdaConfig,
 })
 if err != nil {
  log.Printf("Couldn't update user pool %v. Here's why: %v\n", userPoolId, err)
 }
 return err
}
```

```
// SignUp signs up a user with Amazon Cognito.
func (actor CognitoActions) SignUp(clientId string, userName string, password
 string, userEmail string) (bool, error) {
 confirmed := false
 output, err := actor.CognitoClient.SignUp(context.TODO(),
 &cognitoidentityprovider.SignUpInput{
  ClientId: aws.String(clientId),
  Password: aws.String(password),
  Username: aws.String(userName),
  UserAttributes: []types.AttributeType{
   {Name: aws.String("email"), Value: aws.String(userEmail)},
  },
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't sign up user %v. Here's why: %v\n", userName, err)
  }
 } else {
  confirmed = output.UserConfirmed
 }
 return confirmed, err
}



// SignIn signs in a user to Amazon Cognito using a username and password
 authentication flow.
func (actor CognitoActions) SignIn(clientId string, userName string, password
 string) (*types.AuthenticationResultType, error) {
 var authResult *types.AuthenticationResultType
 output, err := actor.CognitoClient.InitiateAuth(context.TODO(),
 &cognitoidentityprovider.InitiateAuthInput{
  AuthFlow:       "USER_PASSWORD_AUTH",
  ClientId:       aws.String(clientId),
  AuthParameters: map[string]string{"USERNAME": userName, "PASSWORD": password},
 })
 if err != nil {
  var resetRequired *types.PasswordResetRequiredException
```

```
  if errors.As(err, &resetRequired) {
   log.Println(*resetRequired.Message)
  } else {
   log.Printf("Couldn't sign in user %v. Here's why: %v\n", userName, err)
  }
 } else {
  authResult = output.AuthenticationResult
 }
 return authResult, err
}



// ForgotPassword starts a password recovery flow for a user. This flow typically
 sends a confirmation code
// to the user's configured notification destination, such as email.
func (actor CognitoActions) ForgotPassword(clientId string, userName string)
 (*types.CodeDeliveryDetailsType, error) {
 output, err := actor.CognitoClient.ForgotPassword(context.TODO(),
 &cognitoidentityprovider.ForgotPasswordInput{
  ClientId: aws.String(clientId),
  Username: aws.String(userName),
 })
 if err != nil {
  log.Printf("Couldn't start password reset for user '%v'. Here;s why: %v\n",
 userName, err)
 }
 return output.CodeDeliveryDetails, err
}



// ConfirmForgotPassword confirms a user with a confirmation code and a new
 password.
func (actor CognitoActions) ConfirmForgotPassword(clientId string, code string,
 userName string, password string) error {
 _, err := actor.CognitoClient.ConfirmForgotPassword(context.TODO(),
 &cognitoidentityprovider.ConfirmForgotPasswordInput{
  ClientId:         aws.String(clientId),
  ConfirmationCode: aws.String(code),
  Password:         aws.String(password),
  Username:         aws.String(userName),
 })
 if err != nil {
```

```
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't confirm user %v. Here's why: %v", userName, err)
  }
 }
 return err
}



// DeleteUser removes a user from the user pool.
func (actor CognitoActions) DeleteUser(userAccessToken string) error {
 _, err := actor.CognitoClient.DeleteUser(context.TODO(),
 &cognitoidentityprovider.DeleteUserInput{
  AccessToken: aws.String(userAccessToken),
 })
 if err != nil {
  log.Printf("Couldn't delete user. Here's why: %v\n", err)
 }
 return err
}



// AdminCreateUser uses administrator credentials to add a user to a user pool.
 This method leaves the user
// in a state that requires they enter a new password next time they sign in.
func (actor CognitoActions) AdminCreateUser(userPoolId string, userName string,
 userEmail string) error {
 _, err := actor.CognitoClient.AdminCreateUser(context.TODO(),
 &cognitoidentityprovider.AdminCreateUserInput{
  UserPoolId:     aws.String(userPoolId),
  Username:       aws.String(userName),
  MessageAction:  types.MessageActionTypeSuppress,
  UserAttributes: []types.AttributeType{{Name: aws.String("email"), Value:
 aws.String(userEmail)}},
 })
 if err != nil {
  var userExists *types.UsernameExistsException
  if errors.As(err, &userExists) {
   log.Printf("User %v already exists in the user pool.", userName)
   err = nil
```

```
  } else {
   log.Printf("Couldn't create user %v. Here's why: %v\n", userName, err)
  }
 }
 return err
}



// AdminSetUserPassword uses administrator credentials to set a password for a
 user without requiring a
// temporary password.
func (actor CognitoActions) AdminSetUserPassword(userPoolId string, userName
 string, password string) error {
 _, err := actor.CognitoClient.AdminSetUserPassword(context.TODO(),
 &cognitoidentityprovider.AdminSetUserPasswordInput{
  Password:   aws.String(password),
  UserPoolId: aws.String(userPoolId),
  Username:   aws.String(userName),
  Permanent:  true,
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't set password for user %v. Here's why: %v\n", userName,
 err)
  }
 }
 return err
}
```

建立包裝 DynamoDB 動作的結構。

```
// DynamoActions encapsulates the Amazon Simple Notification Service (Amazon SNS)
 actions
// used in the examples.
type DynamoActions struct {
 DynamoClient *dynamodb.Client
```

```
}

// User defines structured user data.
type User struct {
 UserName  string
 UserEmail string
 LastLogin *LoginInfo `dynamodbav:",omitempty"`
}

// LoginInfo defines structured custom login data.
type LoginInfo struct {
 UserPoolId string
 ClientId   string
 Time       string
}

// UserList defines a list of users.
type UserList struct {
 Users []User
}

// UserNameList returns the usernames contained in a UserList as a list of
 strings.
func (users *UserList) UserNameList() []string {
 names := make([]string, len(users.Users))
 for i := 0; i < len(users.Users); i++ {
  names[i] = users.Users[i].UserName
 }
 return names
}

// PopulateTable adds a set of test users to the table.
func (actor DynamoActions) PopulateTable(tableName string) error {
 var err error
 var item map[string]types.AttributeValue
 var writeReqs []types.WriteRequest
 for i := 1; i < 4; i++ {
  item, err = attributevalue.MarshalMap(User{UserName: fmt.Sprintf("test_user_
%v", i), UserEmail: fmt.Sprintf("test_email_%v@example.com", i)})
  if err != nil {
   log.Printf("Couldn't marshall user into DynamoDB format. Here's why: %v\n",
 err)
   return err
  }
```

```
  writeReqs = append(writeReqs, types.WriteRequest{PutRequest:
 &types.PutRequest{Item: item}})
 }
 _, err = actor.DynamoClient.BatchWriteItem(context.TODO(),
 &dynamodb.BatchWriteItemInput{
  RequestItems: map[string][]types.WriteRequest{tableName: writeReqs},
 })
 if err != nil {
  log.Printf("Couldn't populate table %v with users. Here's why: %v\n",
 tableName, err)
 }
 return err
}

// Scan scans the table for all items.
func (actor DynamoActions) Scan(tableName string) (UserList, error) {
 var userList UserList
 output, err := actor.DynamoClient.Scan(context.TODO(), &dynamodb.ScanInput{
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Couldn't scan table %v for items. Here's why: %v\n", tableName,
 err)
 } else {
  err = attributevalue.UnmarshalListOfMaps(output.Items, &userList.Users)
  if err != nil {
   log.Printf("Couldn't unmarshal items into users. Here's why: %v\n", err)
  }
 }
 return userList, err
}

// AddUser adds a user item to a table.
func (actor DynamoActions) AddUser(tableName string, user User) error {
 userItem, err := attributevalue.MarshalMap(user)
 if err != nil {
  log.Printf("Couldn't marshall user to item. Here's why: %v\n", err)
 }
 _, err = actor.DynamoClient.PutItem(context.TODO(), &dynamodb.PutItemInput{
  Item:      userItem,
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Couldn't put item in table %v. Here's why: %v", tableName, err)
```

```
 }
 return err
}
```

建立包裝 CloudWatch 記錄動作的結構。

```
type CloudWatchLogsActions struct {
 CwlClient *cloudwatchlogs.Client
}

// GetLatestLogStream gets the most recent log stream for a Lambda function.
func (actor CloudWatchLogsActions) GetLatestLogStream(functionName string)
 (types.LogStream, error) {
 var logStream types.LogStream
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.DescribeLogStreams(context.TODO(),
 &cloudwatchlogs.DescribeLogStreamsInput{
  Descending:   aws.Bool(true),
  Limit:        aws.Int32(1),
  LogGroupName: aws.String(logGroupName),
  OrderBy:      types.OrderByLastEventTime,
 })
 if err != nil {
  log.Printf("Couldn't get log streams for log group %v. Here's why: %v\n",
 logGroupName, err)
 } else {
  logStream = output.LogStreams[0]
 }
 return logStream, err
}

// GetLogEvents gets the most recent eventCount events from the specified log
 stream.
func (actor CloudWatchLogsActions) GetLogEvents(functionName string,
 logStreamName string, eventCount int32) (
 []types.OutputLogEvent, error) {
 var events []types.OutputLogEvent
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.GetLogEvents(context.TODO(),
 &cloudwatchlogs.GetLogEventsInput{
```

```
  LogStreamName: aws.String(logStreamName),
  Limit:         aws.Int32(eventCount),
  LogGroupName:  aws.String(logGroupName),
 })
 if err != nil {
  log.Printf("Couldn't get log event for log stream %v. Here's why: %v\n",
 logStreamName, err)
 } else {
  events = output.Events
 }
 return events, err
}
```

創建一個包裝 AWS CloudFormation 動作的結構。

```
// StackOutputs defines a map of outputs from a specific stack.
type StackOutputs map[string]string

type CloudFormationActions struct {
 CfnClient *cloudformation.Client
}

// GetOutputs gets the outputs from a CloudFormation stack and puts them into a
 structured format.
func (actor CloudFormationActions) GetOutputs(stackName string) StackOutputs {
 output, err := actor.CfnClient.DescribeStacks(context.TODO(),
 &cloudformation.DescribeStacksInput{
  StackName: aws.String(stackName),
 })
 if err != nil || len(output.Stacks) == 0 {
  log.Panicf("Couldn't find a CloudFormation stack named %v. Here's why: %v\n",
 stackName, err)
 }
 stackOutputs := StackOutputs{}
 for _, out := range output.Stacks[0].Outputs {
  stackOutputs[*out.OutputKey] = *out.OutputValue
 }
 return stackOutputs
}
```

清理資源。

```
// Resources keeps track of AWS resources created during an example and handles
// cleanup when the example finishes.
type Resources struct {
 userPoolId       string
 userAccessTokens []string
 triggers         []actions.Trigger

 cognitoActor *actions.CognitoActions
 questioner   demotools.IQuestioner
}

func (resources *Resources) init(cognitoActor *actions.CognitoActions, questioner
 demotools.IQuestioner) {
 resources.userAccessTokens = []string{}
 resources.triggers = []actions.Trigger{}
 resources.cognitoActor = cognitoActor
 resources.questioner = questioner
}

// Cleanup deletes all AWS resources created during an example.
func (resources *Resources) Cleanup() {
 defer func() {
  if r := recover(); r != nil {
   log.Printf("Something went wrong during cleanup.\n%v\n", r)
   log.Println("Use the AWS Management Console to remove any remaining resources
 \n" +
    "that were created for this scenario.")
  }
 }()

 wantDelete := resources.questioner.AskBool("Do you want to remove all of the AWS
 resources that were created "+
  "during this demo (y/n)?", "y")
 if wantDelete {
  for _, accessToken := range resources.userAccessTokens {
   err := resources.cognitoActor.DeleteUser(accessToken)
   if err != nil {
    log.Println("Couldn't delete user during cleanup.")
```

```
    panic(err)
   }
   log.Println("Deleted user.")
  }
  triggerList := make([]actions.TriggerInfo, len(resources.triggers))
  for i := 0; i < len(resources.triggers); i++ {
   triggerList[i] = actions.TriggerInfo{Trigger: resources.triggers[i],
 HandlerArn: nil}
  }
  err := resources.cognitoActor.UpdateTriggers(resources.userPoolId,
 triggerList...)
  if err != nil {
   log.Println("Couldn't update Cognito triggers during cleanup.")
   panic(err)
  }
  log.Println("Removed Cognito triggers from user pool.")
 } else {
  log.Println("Be sure to remove resources when you're done with them to avoid
 unexpected charges!")
 }
}
```

- 如需 API 詳細資訊，請參閱《AWS SDK for Go API 參考》中的下列主題。
  - DeleteUser
  - InitiateAuth
  - SignUp
  - UpdateUser游泳池

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使用開發套件，透過 Lambda 函數自動遷移已知的 Amazon Cognito 知使用者 AWS

下列程式碼範例說明如何使用 Lambda 函數自動遷移已知的 Amazon Cognito 使用者。

- 設定使用者集區以呼叫`MigrateUser`觸發器的 Lambda 函數。
- 使用不在使用者集區中的使用者名稱和電子郵件登入 Amazon Cognito。

- Lambda 函數會掃描 DynamoDB 表格，並自動將已知的使用者移轉至使用者集區。
- 執行忘記密碼流程以重設已遷移使用者的密碼。
- 以新使用者身分登入，然後清理資源。

Go

SDK for Go V2

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

在命令提示中執行互動式案例。

```
import (
 "errors"
 "fmt"
 "log"
 "strings"
 "user_pools_and_lambda_triggers/actions"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider/types"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
)

// MigrateUser separates the steps of this scenario into individual functions so
 that
// they are simpler to read and understand.
type MigrateUser struct {
 helper       IScenarioHelper
 questioner   demotools.IQuestioner
 resources    Resources
 cognitoActor *actions.CognitoActions
}

// NewMigrateUser constructs a new migrate user runner.
```

```
func NewMigrateUser(sdkConfig aws.Config, questioner demotools.IQuestioner,
 helper IScenarioHelper) MigrateUser {
 scenario := MigrateUser{
  helper:       helper,
  questioner:   questioner,
  resources:    Resources{},
  cognitoActor: &actions.CognitoActions{CognitoClient:
 cognitoidentityprovider.NewFromConfig(sdkConfig)},
 }
 scenario.resources.init(scenario.cognitoActor, questioner)
 return scenario
}

// AddMigrateUserTrigger adds a Lambda handler as an invocation target for the
 MigrateUser trigger.
func (runner *MigrateUser) AddMigrateUserTrigger(userPoolId string, functionArn
 string) {
 log.Printf("Let's add a Lambda function to handle the MigrateUser trigger from
 Cognito.\n" +
  "This trigger happens when an unknown user signs in, and lets your function
 take action before Cognito\n" +
  "rejects the user.\n\n")
 err := runner.cognitoActor.UpdateTriggers(
  userPoolId,
  actions.TriggerInfo{Trigger: actions.UserMigration, HandlerArn:
 aws.String(functionArn)})
 if err != nil {
  panic(err)
 }
 log.Printf("Lambda function %v added to user pool %v to handle the MigrateUser
 trigger.\n",
  functionArn, userPoolId)

 log.Println(strings.Repeat("-", 88))
}

// SignInUser adds a new user to the known users table and signs that user in to
 Amazon Cognito.
func (runner *MigrateUser) SignInUser(usersTable string, clientId string) (bool,
 actions.User) {
 log.Println("Let's sign in a user to your Cognito user pool. When the username
 and email matches an entry in the\n" +
  "DynamoDB known users table, the email is automatically verified and the user
 is migrated to the Cognito user pool.")
```

```
 user := actions.User{}
 user.UserName = runner.questioner.Ask("\nEnter a username:")
 user.UserEmail = runner.questioner.Ask("\nEnter an email that you own. This
 email will be used to confirm user migration\n" +
  "during this example:")

 runner.helper.AddKnownUser(usersTable, user)

 var err error
 var resetRequired *types.PasswordResetRequiredException
 var authResult *types.AuthenticationResultType
 signedIn := false
 for !signedIn && resetRequired == nil {
  log.Printf("Signing in to Cognito as user '%v'. The expected result is a
 PasswordResetRequiredException.\n\n", user.UserName)
  authResult, err = runner.cognitoActor.SignIn(clientId, user.UserName, "_")
  if err != nil {
   if errors.As(err, &resetRequired) {
    log.Printf("\nUser '%v' is not in the Cognito user pool but was found in the
 DynamoDB known users table.\n"+
     "User migration is started and a password reset is required.",
 user.UserName)
   } else {
    panic(err)
   }
  } else {
   log.Printf("User '%v' successfully signed in. This is unexpected and probably
 means you have not\n"+
    "cleaned up a previous run of this scenario, so the user exist in the Cognito
 user pool.\n"+
    "You can continue this example and select to clean up resources, or manually
 remove\n"+
    "the user from your user pool and try again.", user.UserName)
   runner.resources.userAccessTokens = append(runner.resources.userAccessTokens,
 *authResult.AccessToken)
   signedIn = true
  }
 }

 log.Println(strings.Repeat("-", 88))
 return resetRequired != nil, user
}
```

```
// ResetPassword starts a password recovery flow.
func (runner *MigrateUser) ResetPassword(clientId string, user actions.User) {
 wantCode := runner.questioner.AskBool(fmt.Sprintf("In order to migrate the user
 to Cognito, you must be able to receive a confirmation\n"+
  "code by email at %v. Do you want to send a code (y/n)?", user.UserEmail), "y")
 if !wantCode {
  log.Println("To complete this example and successfully migrate a user to
 Cognito, you must enter an email\n" +
   "you own that can receive a confirmation code.")
  return
 }
 codeDelivery, err := runner.cognitoActor.ForgotPassword(clientId, user.UserName)
 if err != nil {
  panic(err)
 }
 log.Printf("\nA confirmation code has been sent to %v.",
 *codeDelivery.Destination)
 code := runner.questioner.Ask("Check your email and enter it here:")

 confirmed := false
 password := runner.questioner.AskPassword("\nEnter a password that has at least
 eight characters, uppercase, lowercase, numbers and symbols.\n"+
  "(the password will not display as you type):", 8)
 for !confirmed {
  log.Printf("\nConfirming password reset for user '%v'.\n", user.UserName)
  err = runner.cognitoActor.ConfirmForgotPassword(clientId, code, user.UserName,
 password)
  if err != nil {
   var invalidPassword *types.InvalidPasswordException
   if errors.As(err, &invalidPassword) {
    password = runner.questioner.AskPassword("\nEnter another password:", 8)
   } else {
    panic(err)
   }
  } else {
   confirmed = true
  }
 }
 log.Printf("User '%v' successfully confirmed and migrated.\n", user.UserName)
 log.Println("Signing in with your username and password...")
 authResult, err := runner.cognitoActor.SignIn(clientId, user.UserName, password)
 if err != nil {
  panic(err)
 }
```

```
 log.Printf("Successfully signed in. Your access token starts with: %v...\n",
 (*authResult.AccessToken)[:10])
 runner.resources.userAccessTokens = append(runner.resources.userAccessTokens,
 *authResult.AccessToken)

 log.Println(strings.Repeat("-", 88))
}

// Run runs the scenario.
func (runner *MigrateUser) Run(stackName string) {
 defer func() {
  if r := recover(); r != nil {
   log.Println("Something went wrong with the demo.")
   runner.resources.Cleanup()
  }
 }()

 log.Println(strings.Repeat("-", 88))
 log.Printf("Welcome\n")

 log.Println(strings.Repeat("-", 88))

 stackOutputs, err := runner.helper.GetStackOutputs(stackName)
 if err != nil {
  panic(err)
 }
 runner.resources.userPoolId = stackOutputs["UserPoolId"]

 runner.AddMigrateUserTrigger(stackOutputs["UserPoolId"],
 stackOutputs["MigrateUserFunctionArn"])
 runner.resources.triggers = append(runner.resources.triggers,
 actions.UserMigration)
 resetNeeded, user := runner.SignInUser(stackOutputs["TableName"],
 stackOutputs["UserPoolClientId"])
 if resetNeeded {
  runner.helper.ListRecentLogEvents(stackOutputs["MigrateUserFunction"])
  runner.ResetPassword(stackOutputs["UserPoolClientId"], user)
 }

 runner.resources.Cleanup()

 log.Println(strings.Repeat("-", 88))
 log.Println("Thanks for watching!")
 log.Println(strings.Repeat("-", 88))
```

```
}
```

使用 Lambda 函數處理MigrateUser觸發器。

```
const TABLE_NAME = "TABLE_NAME"

// UserInfo defines structured user data that can be marshalled to a DynamoDB
 format.
type UserInfo struct {
 UserName  string `dynamodbav:"UserName"`
 UserEmail string `dynamodbav:"UserEmail"`
}

type handler struct {
 dynamoClient *dynamodb.Client
}

// HandleRequest handles the MigrateUser event by looking up a user in an Amazon
 DynamoDB table and
// specifying whether they should be migrated to the user pool.
func (h *handler) HandleRequest(ctx context.Context, event
 events.CognitoEventUserPoolsMigrateUser)
 (events.CognitoEventUserPoolsMigrateUser, error) {
 log.Printf("Received migrate trigger from %v for user '%v'",
 event.TriggerSource, event.UserName)
 if event.TriggerSource != "UserMigration_Authentication" {
  return event, nil
 }
 tableName := os.Getenv(TABLE_NAME)
 user := UserInfo{
  UserName: event.UserName,
 }
 log.Printf("Looking up user '%v' in table %v.\n", user.UserName, tableName)
 filterEx := expression.Name("UserName").Equal(expression.Value(user.UserName))
 expr, err := expression.NewBuilder().WithFilter(filterEx).Build()
 if err != nil {
  log.Printf("Error building expression to query for user '%v'.\n",
 user.UserName)
  return event, err
 }
```

```
 output, err := h.dynamoClient.Scan(ctx, &dynamodb.ScanInput{
  TableName:                 aws.String(tableName),
  FilterExpression:          expr.Filter(),
  ExpressionAttributeNames:  expr.Names(),
  ExpressionAttributeValues: expr.Values(),
 })
 if err != nil {
  log.Printf("Error looking up user '%v'.\n", user.UserName)
  return event, err
 }
 if output.Items == nil || len(output.Items) == 0 {
  log.Printf("User '%v' not found, not migrating user.\n", user.UserName)
  return event, err
 }

 var users []UserInfo
 err = attributevalue.UnmarshalListOfMaps(output.Items, &users)
 if err != nil {
  log.Printf("Couldn't unmarshal DynamoDB items. Here's why: %v\n", err)
  return event, err
 }

 user = users[0]
 log.Printf("UserName '%v' found with email %v. User is migrated and must reset
 password.\n", user.UserName, user.UserEmail)
 event.CognitoEventUserPoolsMigrateUserResponse.UserAttributes =
 map[string]string{
  "email":          user.UserEmail,
  "email_verified": "true", // email_verified is required for the forgot password
 flow.
 }
 event.CognitoEventUserPoolsMigrateUserResponse.FinalUserStatus =
 "RESET_REQUIRED"
 event.CognitoEventUserPoolsMigrateUserResponse.MessageAction = "SUPPRESS"

 return event, err
}

func main() {
 sdkConfig, err := config.LoadDefaultConfig(context.TODO())
 if err != nil {
  log.Panicln(err)
 }
 h := handler{
```

```
  dynamoClient: dynamodb.NewFromConfig(sdkConfig),
 }
 lambda.Start(h.HandleRequest)
}
```

建立執行一般工作的結構。

```
// IScenarioHelper defines common functions used by the workflows in this
 example.
type IScenarioHelper interface {
 Pause(secs int)
 GetStackOutputs(stackName string) (actions.StackOutputs, error)
 PopulateUserTable(tableName string)
 GetKnownUsers(tableName string) (actions.UserList, error)
 AddKnownUser(tableName string, user actions.User)
 ListRecentLogEvents(functionName string)
}

// ScenarioHelper contains AWS wrapper structs used by the workflows in this
 example.
type ScenarioHelper struct {
 questioner  demotools.IQuestioner
 dynamoActor *actions.DynamoActions
 cfnActor    *actions.CloudFormationActions
 cwlActor    *actions.CloudWatchLogsActions
 isTestRun   bool
}

// NewScenarioHelper constructs a new scenario helper.
func NewScenarioHelper(sdkConfig aws.Config, questioner demotools.IQuestioner)
 ScenarioHelper {
 scenario := ScenarioHelper{
  questioner:  questioner,
  dynamoActor: &actions.DynamoActions{DynamoClient:
 dynamodb.NewFromConfig(sdkConfig)},
  cfnActor:    &actions.CloudFormationActions{CfnClient:
 cloudformation.NewFromConfig(sdkConfig)},
  cwlActor:    &actions.CloudWatchLogsActions{CwlClient:
 cloudwatchlogs.NewFromConfig(sdkConfig)},
 }
```

```
 return scenario
}

// Pause waits for the specified number of seconds.
func (helper ScenarioHelper) Pause(secs int) {
 if !helper.isTestRun {
  time.Sleep(time.Duration(secs) * time.Second)
 }
}

// GetStackOutputs gets the outputs from the specified CloudFormation stack in a
 structured format.
func (helper ScenarioHelper) GetStackOutputs(stackName string)
 (actions.StackOutputs, error) {
 return helper.cfnActor.GetOutputs(stackName), nil
}

// PopulateUserTable fills the known user table with example data.
func (helper ScenarioHelper) PopulateUserTable(tableName string) {
 log.Printf("First, let's add some users to the DynamoDB %v table we'll use for
 this example.\n", tableName)
 err := helper.dynamoActor.PopulateTable(tableName)
 if err != nil {
  panic(err)
 }
}

// GetKnownUsers gets the users from the known users table in a structured
 format.
func (helper ScenarioHelper) GetKnownUsers(tableName string) (actions.UserList,
 error) {
 knownUsers, err := helper.dynamoActor.Scan(tableName)
 if err != nil {
  log.Printf("Couldn't get known users from table %v. Here's why: %v\n",
 tableName, err)
 }
 return knownUsers, err
}

// AddKnownUser adds a user to the known users table.
func (helper ScenarioHelper) AddKnownUser(tableName string, user actions.User) {
 log.Printf("Adding user '%v' with email '%v' to the DynamoDB known users
 table...\n",
  user.UserName, user.UserEmail)
```

```
 err := helper.dynamoActor.AddUser(tableName, user)
 if err != nil {
  panic(err)
 }
}

// ListRecentLogEvents gets the most recent log stream and events for the
 specified Lambda function and displays them.
func (helper ScenarioHelper) ListRecentLogEvents(functionName string) {
 log.Println("Waiting a few seconds to let Lambda write to CloudWatch Logs...")
 helper.Pause(10)
 log.Println("Okay, let's check the logs to find what's happened recently with
 your Lambda function.")
 logStream, err := helper.cwlActor.GetLatestLogStream(functionName)
 if err != nil {
  panic(err)
 }
 log.Printf("Getting some recent events from log stream %v\n",
 *logStream.LogStreamName)
 events, err := helper.cwlActor.GetLogEvents(functionName,
 *logStream.LogStreamName, 10)
 if err != nil {
  panic(err)
 }
 for _, event := range events {
  log.Printf("\t%v", *event.Message)
 }
 log.Println(strings.Repeat("-", 88))
}
```

創建一個包裝 Amazon Cognito 操作的結構。

```
type CognitoActions struct {
 CognitoClient *cognitoidentityprovider.Client
}
```

```
// Trigger and TriggerInfo define typed data for updating an Amazon Cognito
 trigger.
type Trigger int

const (
 PreSignUp Trigger = iota
 UserMigration
 PostAuthentication
)

type TriggerInfo struct {
 Trigger    Trigger
 HandlerArn *string
}

// UpdateTriggers adds or removes Lambda triggers for a user pool. When a trigger
 is specified with a `nil` value,
// it is removed from the user pool.
func (actor CognitoActions) UpdateTriggers(userPoolId string,
 triggers ...TriggerInfo) error {
 output, err := actor.CognitoClient.DescribeUserPool(context.TODO(),
 &cognitoidentityprovider.DescribeUserPoolInput{
  UserPoolId: aws.String(userPoolId),
 })
 if err != nil {
  log.Printf("Couldn't get info about user pool %v. Here's why: %v\n",
 userPoolId, err)
  return err
 }
 lambdaConfig := output.UserPool.LambdaConfig
 for _, trigger := range triggers {
  switch trigger.Trigger {
  case PreSignUp:
   lambdaConfig.PreSignUp = trigger.HandlerArn
  case UserMigration:
   lambdaConfig.UserMigration = trigger.HandlerArn
  case PostAuthentication:
   lambdaConfig.PostAuthentication = trigger.HandlerArn
  }
 }
 _, err = actor.CognitoClient.UpdateUserPool(context.TODO(),
 &cognitoidentityprovider.UpdateUserPoolInput{
  UserPoolId:   aws.String(userPoolId),
  LambdaConfig: lambdaConfig,
```

```
 })
 if err != nil {
  log.Printf("Couldn't update user pool %v. Here's why: %v\n", userPoolId, err)
 }
 return err
}



// SignUp signs up a user with Amazon Cognito.
func (actor CognitoActions) SignUp(clientId string, userName string, password
 string, userEmail string) (bool, error) {
 confirmed := false
 output, err := actor.CognitoClient.SignUp(context.TODO(),
 &cognitoidentityprovider.SignUpInput{
  ClientId: aws.String(clientId),
  Password: aws.String(password),
  Username: aws.String(userName),
  UserAttributes: []types.AttributeType{
   {Name: aws.String("email"), Value: aws.String(userEmail)},
  },
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't sign up user %v. Here's why: %v\n", userName, err)
  }
 } else {
  confirmed = output.UserConfirmed
 }
 return confirmed, err
}



// SignIn signs in a user to Amazon Cognito using a username and password
 authentication flow.
func (actor CognitoActions) SignIn(clientId string, userName string, password
 string) (*types.AuthenticationResultType, error) {
 var authResult *types.AuthenticationResultType
 output, err := actor.CognitoClient.InitiateAuth(context.TODO(),
 &cognitoidentityprovider.InitiateAuthInput{
```

```
  AuthFlow:       "USER_PASSWORD_AUTH",
  ClientId:       aws.String(clientId),
  AuthParameters: map[string]string{"USERNAME": userName, "PASSWORD": password},
 })
 if err != nil {
  var resetRequired *types.PasswordResetRequiredException
  if errors.As(err, &resetRequired) {
   log.Println(*resetRequired.Message)
  } else {
   log.Printf("Couldn't sign in user %v. Here's why: %v\n", userName, err)
  }
 } else {
  authResult = output.AuthenticationResult
 }
 return authResult, err
}



// ForgotPassword starts a password recovery flow for a user. This flow typically
 sends a confirmation code
// to the user's configured notification destination, such as email.
func (actor CognitoActions) ForgotPassword(clientId string, userName string)
 (*types.CodeDeliveryDetailsType, error) {
 output, err := actor.CognitoClient.ForgotPassword(context.TODO(),
 &cognitoidentityprovider.ForgotPasswordInput{
  ClientId: aws.String(clientId),
  Username: aws.String(userName),
 })
 if err != nil {
  log.Printf("Couldn't start password reset for user '%v'. Here;s why: %v\n",
 userName, err)
 }
 return output.CodeDeliveryDetails, err
}



// ConfirmForgotPassword confirms a user with a confirmation code and a new
 password.
func (actor CognitoActions) ConfirmForgotPassword(clientId string, code string,
 userName string, password string) error {
 _, err := actor.CognitoClient.ConfirmForgotPassword(context.TODO(),
 &cognitoidentityprovider.ConfirmForgotPasswordInput{
```

```
  ClientId:         aws.String(clientId),
  ConfirmationCode: aws.String(code),
  Password:         aws.String(password),
  Username:         aws.String(userName),
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't confirm user %v. Here's why: %v", userName, err)
  }
 }
 return err
}



// DeleteUser removes a user from the user pool.
func (actor CognitoActions) DeleteUser(userAccessToken string) error {
 _, err := actor.CognitoClient.DeleteUser(context.TODO(),
 &cognitoidentityprovider.DeleteUserInput{
  AccessToken: aws.String(userAccessToken),
 })
 if err != nil {
  log.Printf("Couldn't delete user. Here's why: %v\n", err)
 }
 return err
}



// AdminCreateUser uses administrator credentials to add a user to a user pool.
 This method leaves the user
// in a state that requires they enter a new password next time they sign in.
func (actor CognitoActions) AdminCreateUser(userPoolId string, userName string,
 userEmail string) error {
 _, err := actor.CognitoClient.AdminCreateUser(context.TODO(),
 &cognitoidentityprovider.AdminCreateUserInput{
  UserPoolId:     aws.String(userPoolId),
  Username:       aws.String(userName),
  MessageAction:  types.MessageActionTypeSuppress,
  UserAttributes: []types.AttributeType{{Name: aws.String("email"), Value:
 aws.String(userEmail)}},
```

```
 })
 if err != nil {
  var userExists *types.UsernameExistsException
  if errors.As(err, &userExists) {
   log.Printf("User %v already exists in the user pool.", userName)
   err = nil
  } else {
   log.Printf("Couldn't create user %v. Here's why: %v\n", userName, err)
  }
 }
 return err
}



// AdminSetUserPassword uses administrator credentials to set a password for a
 user without requiring a
// temporary password.
func (actor CognitoActions) AdminSetUserPassword(userPoolId string, userName
 string, password string) error {
 _, err := actor.CognitoClient.AdminSetUserPassword(context.TODO(),
 &cognitoidentityprovider.AdminSetUserPasswordInput{
  Password:   aws.String(password),
  UserPoolId: aws.String(userPoolId),
  Username:   aws.String(userName),
  Permanent:  true,
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't set password for user %v. Here's why: %v\n", userName,
 err)
  }
 }
 return err
}
```

建立包裝 DynamoDB 動作的結構。

```
// DynamoActions encapsulates the Amazon Simple Notification Service (Amazon SNS)
 actions
// used in the examples.
type DynamoActions struct {
 DynamoClient *dynamodb.Client
}

// User defines structured user data.
type User struct {
 UserName  string
 UserEmail string
 LastLogin *LoginInfo `dynamodbav:",omitempty"`
}

// LoginInfo defines structured custom login data.
type LoginInfo struct {
 UserPoolId string
 ClientId   string
 Time       string
}

// UserList defines a list of users.
type UserList struct {
 Users []User
}

// UserNameList returns the usernames contained in a UserList as a list of
 strings.
func (users *UserList) UserNameList() []string {
 names := make([]string, len(users.Users))
 for i := 0; i < len(users.Users); i++ {
  names[i] = users.Users[i].UserName
 }
 return names
}

// PopulateTable adds a set of test users to the table.
func (actor DynamoActions) PopulateTable(tableName string) error {
 var err error
 var item map[string]types.AttributeValue
 var writeReqs []types.WriteRequest
 for i := 1; i < 4; i++ {
```

```
  item, err = attributevalue.MarshalMap(User{UserName: fmt.Sprintf("test_user_
%v", i), UserEmail: fmt.Sprintf("test_email_%v@example.com", i)})
  if err != nil {
   log.Printf("Couldn't marshall user into DynamoDB format. Here's why: %v\n",
 err)
   return err
  }
  writeReqs = append(writeReqs, types.WriteRequest{PutRequest:
 &types.PutRequest{Item: item}})
 }
 _, err = actor.DynamoClient.BatchWriteItem(context.TODO(),
 &dynamodb.BatchWriteItemInput{
  RequestItems: map[string][]types.WriteRequest{tableName: writeReqs},
 })
 if err != nil {
  log.Printf("Couldn't populate table %v with users. Here's why: %v\n",
 tableName, err)
 }
 return err
}

// Scan scans the table for all items.
func (actor DynamoActions) Scan(tableName string) (UserList, error) {
 var userList UserList
 output, err := actor.DynamoClient.Scan(context.TODO(), &dynamodb.ScanInput{
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Couldn't scan table %v for items. Here's why: %v\n", tableName,
 err)
 } else {
  err = attributevalue.UnmarshalListOfMaps(output.Items, &userList.Users)
  if err != nil {
   log.Printf("Couldn't unmarshal items into users. Here's why: %v\n", err)
  }
 }
 return userList, err
}

// AddUser adds a user item to a table.
func (actor DynamoActions) AddUser(tableName string, user User) error {
 userItem, err := attributevalue.MarshalMap(user)
 if err != nil {
  log.Printf("Couldn't marshall user to item. Here's why: %v\n", err)
```

```
 }
 _, err = actor.DynamoClient.PutItem(context.TODO(), &dynamodb.PutItemInput{
  Item:      userItem,
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Couldn't put item in table %v. Here's why: %v", tableName, err)
 }
 return err
}
```

建立包裝 CloudWatch 記錄動作的結構。

```
type CloudWatchLogsActions struct {
 CwlClient *cloudwatchlogs.Client
}

// GetLatestLogStream gets the most recent log stream for a Lambda function.
func (actor CloudWatchLogsActions) GetLatestLogStream(functionName string)
 (types.LogStream, error) {
 var logStream types.LogStream
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.DescribeLogStreams(context.TODO(),
 &cloudwatchlogs.DescribeLogStreamsInput{
  Descending:   aws.Bool(true),
  Limit:        aws.Int32(1),
  LogGroupName: aws.String(logGroupName),
  OrderBy:      types.OrderByLastEventTime,
 })
 if err != nil {
  log.Printf("Couldn't get log streams for log group %v. Here's why: %v\n",
 logGroupName, err)
 } else {
  logStream = output.LogStreams[0]
 }
 return logStream, err
}

// GetLogEvents gets the most recent eventCount events from the specified log
 stream.
```

```
func (actor CloudWatchLogsActions) GetLogEvents(functionName string,
 logStreamName string, eventCount int32) (
 []types.OutputLogEvent, error) {
 var events []types.OutputLogEvent
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.GetLogEvents(context.TODO(),
 &cloudwatchlogs.GetLogEventsInput{
  LogStreamName: aws.String(logStreamName),
  Limit:         aws.Int32(eventCount),
  LogGroupName:  aws.String(logGroupName),
 })
 if err != nil {
  log.Printf("Couldn't get log event for log stream %v. Here's why: %v\n",
 logStreamName, err)
 } else {
  events = output.Events
 }
 return events, err
}
```

創建一個包裝 AWS CloudFormation 動作的結構。

```
// StackOutputs defines a map of outputs from a specific stack.
type StackOutputs map[string]string

type CloudFormationActions struct {
 CfnClient *cloudformation.Client
}

// GetOutputs gets the outputs from a CloudFormation stack and puts them into a
 structured format.
func (actor CloudFormationActions) GetOutputs(stackName string) StackOutputs {
 output, err := actor.CfnClient.DescribeStacks(context.TODO(),
 &cloudformation.DescribeStacksInput{
  StackName: aws.String(stackName),
 })
 if err != nil || len(output.Stacks) == 0 {
  log.Panicf("Couldn't find a CloudFormation stack named %v. Here's why: %v\n",
 stackName, err)
 }
```

```
 stackOutputs := StackOutputs{}
 for _, out := range output.Stacks[0].Outputs {
  stackOutputs[*out.OutputKey] = *out.OutputValue
 }
 return stackOutputs
}
```

清理資源。

```
// Resources keeps track of AWS resources created during an example and handles
// cleanup when the example finishes.
type Resources struct {
 userPoolId       string
 userAccessTokens []string
 triggers         []actions.Trigger

 cognitoActor *actions.CognitoActions
 questioner   demotools.IQuestioner
}

func (resources *Resources) init(cognitoActor *actions.CognitoActions, questioner
 demotools.IQuestioner) {
 resources.userAccessTokens = []string{}
 resources.triggers = []actions.Trigger{}
 resources.cognitoActor = cognitoActor
 resources.questioner = questioner
}

// Cleanup deletes all AWS resources created during an example.
func (resources *Resources) Cleanup() {
 defer func() {
  if r := recover(); r != nil {
   log.Printf("Something went wrong during cleanup.\n%v\n", r)
   log.Println("Use the AWS Management Console to remove any remaining resources
 \n" +
    "that were created for this scenario.")
  }
 }()
```

```
 wantDelete := resources.questioner.AskBool("Do you want to remove all of the AWS
 resources that were created "+
  "during this demo (y/n)?", "y")
 if wantDelete {
  for _, accessToken := range resources.userAccessTokens {
   err := resources.cognitoActor.DeleteUser(accessToken)
   if err != nil {
    log.Println("Couldn't delete user during cleanup.")
    panic(err)
   }
   log.Println("Deleted user.")
  }
  triggerList := make([]actions.TriggerInfo, len(resources.triggers))
  for i := 0; i < len(resources.triggers); i++ {
   triggerList[i] = actions.TriggerInfo{Trigger: resources.triggers[i],
 HandlerArn: nil}
  }
  err := resources.cognitoActor.UpdateTriggers(resources.userPoolId,
 triggerList...)
  if err != nil {
   log.Println("Couldn't update Cognito triggers during cleanup.")
   panic(err)
  }
  log.Println("Removed Cognito triggers from user pool.")
 } else {
  log.Println("Be sure to remove resources when you're done with them to avoid
 unexpected charges!")
 }
}
```

- 如需 API 詳細資訊，請參閱《AWS SDK for Go API 參考》中的下列主題。
  - ConfirmForgot密碼
  - DeleteUser
  - ForgotPassword
  - InitiateAuth
  - SignUp
  - UpdateUser游泳池

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

開始使用開發套件 AWS 建立和叫用 Lambda 函數

下列程式碼範例示範如何：

- 建立 IAM 角色和 Lambda 函數，然後上傳處理常式程式碼。
- 調用具有單一參數的函數並取得結果。
- 更新函數程式碼並使用環境變數進行設定。
- 調用具有新參數的函數並取得結果。顯示傳回的執行日誌。
- 列出您帳戶的函數，然後清理相關資源。

如需詳細資訊，請參閱使用主控台建立 Lambda 函數。

.NET

AWS SDK for .NET

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

建立執行 Lambda 動作的方法。

```
namespace LambdaActions;

using Amazon.Lambda;
using Amazon.Lambda.Model;

/// <summary>
/// A class that implements AWS Lambda methods.
/// </summary>
public class LambdaWrapper
{
    private readonly IAmazonLambda _lambdaService;

    /// <summary>
```

```
    /// Constructor for the LambdaWrapper class.
    /// </summary>
    /// <param name="lambdaService">An initialized Lambda service client.</param>
    public LambdaWrapper(IAmazonLambda lambdaService)
    {
        _lambdaService = lambdaService;
    }

    /// <summary>
    /// Creates a new Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the function.</param>
    /// <param name="s3Bucket">The Amazon Simple Storage Service (Amazon S3)
    /// bucket where the zip file containing the code is located.</param>
    /// <param name="s3Key">The Amazon S3 key of the zip file.</param>
    /// <param name="role">The Amazon Resource Name (ARN) of a role with the
    /// appropriate Lambda permissions.</param>
    /// <param name="handler">The name of the handler function.</param>
    /// <returns>The Amazon Resource Name (ARN) of the newly created
    /// Lambda function.</returns>
    public async Task<string> CreateLambdaFunctionAsync(
        string functionName,
        string s3Bucket,
        string s3Key,
        string role,
        string handler)
    {
        // Defines the location for the function code.
        // S3Bucket - The S3 bucket where the file containing
        //            the source code is stored.
        // S3Key    - The name of the file containing the code.
        var functionCode = new FunctionCode
        {
            S3Bucket = s3Bucket,
            S3Key = s3Key,
        };

        var createFunctionRequest = new CreateFunctionRequest
        {
            FunctionName = functionName,
            Description = "Created by the Lambda .NET API",
            Code = functionCode,
            Handler = handler,
            Runtime = Runtime.Dotnet6,
```

```
            Role = role,
        };

        var reponse = await
 _lambdaService.CreateFunctionAsync(createFunctionRequest);
        return reponse.FunctionArn;
    }


    /// <summary>
    /// Delete an AWS Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to
    /// delete.</param>
    /// <returns>A Boolean value that indicates the success of the action.</
returns>
    public async Task<bool> DeleteFunctionAsync(string functionName)
    {
        var request = new DeleteFunctionRequest
        {
            FunctionName = functionName,
        };

        var response = await _lambdaService.DeleteFunctionAsync(request);

        // A return value of NoContent means that the request was processed.
        // In this case, the function was deleted, and the return value
        // is intentionally blank.
        return response.HttpStatusCode == System.Net.HttpStatusCode.NoContent;
    }


    /// <summary>
    /// Gets information about a Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function for
    /// which to retrieve information.</param>
    /// <returns>Async Task.</returns>
    public async Task<FunctionConfiguration> GetFunctionAsync(string
 functionName)
    {
        var functionRequest = new GetFunctionRequest
        {
            FunctionName = functionName,
```

```
        };

        var response = await _lambdaService.GetFunctionAsync(functionRequest);
        return response.Configuration;
    }


    /// <summary>
    /// Invoke a Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to
    /// invoke.</param
    /// <param name="parameters">The parameter values that will be passed to the
function.</param>
    /// <returns>A System Threading Task.</returns>
    public async Task<string> InvokeFunctionAsync(
        string functionName,
        string parameters)
    {
        var payload = parameters;
        var request = new InvokeRequest
        {
            FunctionName = functionName,
            Payload = payload,
        };

        var response = await _lambdaService.InvokeAsync(request);
        MemoryStream stream = response.Payload;
        string returnValue =
System.Text.Encoding.UTF8.GetString(stream.ToArray());
        return returnValue;
    }


    /// <summary>
    /// Get a list of Lambda functions.
    /// </summary>
    /// <returns>A list of FunctionConfiguration objects.</returns>
    public async Task<List<FunctionConfiguration>> ListFunctionsAsync()
    {
        var functionList = new List<FunctionConfiguration>();

        var functionPaginator =
            _lambdaService.Paginators.ListFunctions(new ListFunctionsRequest());
```

```
        await foreach (var function in functionPaginator.Functions)
        {
            functionList.Add(function);
        }

        return functionList;
    }


    /// <summary>
    /// Update an existing Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to update.</
param>
    /// <param name="bucketName">The bucket where the zip file containing
    /// the Lambda function code is stored.</param>
    /// <param name="key">The key name of the source code file.</param>
    /// <returns>Async Task.</returns>
    public async Task UpdateFunctionCodeAsync(
        string functionName,
        string bucketName,
        string key)
    {
        var functionCodeRequest = new UpdateFunctionCodeRequest
        {
            FunctionName = functionName,
            Publish = true,
            S3Bucket = bucketName,
            S3Key = key,
        };

        var response = await
 _lambdaService.UpdateFunctionCodeAsync(functionCodeRequest);
        Console.WriteLine($"The Function was last modified at
 {response.LastModified}.");
    }


    /// <summary>
    /// Update the code of a Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the function to update.</param>
    /// <param name="functionHandler">The code that performs the function's
 actions.</param>
```

```
    /// <param name="environmentVariables">A dictionary of environment
 variables.</param>
    /// <returns>A Boolean value indicating the success of the action.</returns>
    public async Task<bool> UpdateFunctionConfigurationAsync(
        string functionName,
        string functionHandler,
        Dictionary<string, string> environmentVariables)
    {
        var request = new UpdateFunctionConfigurationRequest
        {
            Handler = functionHandler,
            FunctionName = functionName,
            Environment = new Amazon.Lambda.Model.Environment { Variables =
 environmentVariables },
        };

        var response = await
 _lambdaService.UpdateFunctionConfigurationAsync(request);

        Console.WriteLine(response.LastModified);

        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }


}
```

建立可執行該案例的函數。

```
global using System.Threading.Tasks;
global using Amazon.IdentityManagement;
global using Amazon.Lambda;
global using LambdaActions;
global using LambdaScenarioCommon;
global using Microsoft.Extensions.DependencyInjection;
global using Microsoft.Extensions.Hosting;
global using Microsoft.Extensions.Logging;
global using Microsoft.Extensions.Logging.Console;
global using Microsoft.Extensions.Logging.Debug;
```

```
using Amazon.Lambda.Model;
using Microsoft.Extensions.Configuration;

namespace LambdaBasics;

public class LambdaBasics
{
    private static ILogger logger = null!;

    static async Task Main(string[] args)
    {
        // Set up dependency injection for the Amazon service.
        using var host = Host.CreateDefaultBuilder(args)
            .ConfigureLogging(logging =>
                logging.AddFilter("System", LogLevel.Debug)
                    .AddFilter<DebugLoggerProvider>("Microsoft",
 LogLevel.Information)
                    .AddFilter<ConsoleLoggerProvider>("Microsoft",
 LogLevel.Trace))
            .ConfigureServices((_, services) =>
            services.AddAWSService<IAmazonLambda>()
            .AddAWSService<IAmazonIdentityManagementService>()
            .AddTransient<LambdaWrapper>()
            .AddTransient<LambdaRoleWrapper>()
            .AddTransient<UIWrapper>()
        )
        .Build();

        var configuration = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            .AddJsonFile("settings.json") // Load test settings from .json file.
            .AddJsonFile("settings.local.json",
            true) // Optionally load local settings.
        .Build();


        logger = LoggerFactory.Create(builder => { builder.AddConsole(); })
            .CreateLogger<LambdaBasics>();

        var lambdaWrapper = host.Services.GetRequiredService<LambdaWrapper>();
        var lambdaRoleWrapper =
 host.Services.GetRequiredService<LambdaRoleWrapper>();
        var uiWrapper = host.Services.GetRequiredService<UIWrapper>();
```

```
        string functionName = configuration["FunctionName"]!;
        string roleName = configuration["RoleName"]!;
        string policyDocument = "{" +
            " \"Version\": \"2012-10-17\"," +
            " \"Statement\": [ " +
            "    {" +
            "        \"Effect\": \"Allow\"," +
            "        \"Principal\": {" +
            "            \"Service\": \"lambda.amazonaws.com\" " +
            "    }," +
            "        \"Action\": \"sts:AssumeRole\" " +
            "    }" +
            "]" +
        "}";

        var incrementHandler = configuration["IncrementHandler"];
        var calculatorHandler = configuration["CalculatorHandler"];
        var bucketName = configuration["BucketName"];
        var incrementKey = configuration["IncrementKey"];
        var calculatorKey = configuration["CalculatorKey"];
        var policyArn = configuration["PolicyArn"];

        uiWrapper.DisplayLambdaBasicsOverview();

        // Create the policy to use with the AWS Lambda functions and then attach
the
        // policy to a new role.
        var roleArn = await lambdaRoleWrapper.CreateLambdaRoleAsync(roleName,
policyDocument);

        Console.WriteLine("Waiting for role to become active.");
        uiWrapper.WaitABit(15, "Wait until the role is active before trying to
use it.");

        // Attach the appropriate AWS Identity and Access Management (IAM) role
policy to the new role.
        var success = await
lambdaRoleWrapper.AttachLambdaRolePolicyAsync(policyArn, roleName);
        uiWrapper.WaitABit(10, "Allow time for the IAM policy to be attached to
the role.");

        // Create the Lambda function using a zip file stored in an Amazon Simple
Storage Service
        // (Amazon S3) bucket.
```

```
        uiWrapper.DisplayTitle("Create Lambda Function");
        Console.WriteLine($"Creating the AWS Lambda function: {functionName}.");
        var lambdaArn = await lambdaWrapper.CreateLambdaFunctionAsync(
            functionName,
            bucketName,
            incrementKey,
            roleArn,
            incrementHandler);

        Console.WriteLine("Waiting for the new function to be available.");
        Console.WriteLine($"The AWS Lambda ARN is {lambdaArn}");

        // Get the Lambda function.
        Console.WriteLine($"Getting the {functionName} AWS Lambda function.");
        FunctionConfiguration config;
        do
        {
            config = await lambdaWrapper.GetFunctionAsync(functionName);
            Console.Write(".");
        }
        while (config.State != State.Active);

        Console.WriteLine($"\nThe function, {functionName} has been created.");
        Console.WriteLine($"The runtime of this Lambda function is
{config.Runtime}.");

        uiWrapper.PressEnter();

        // List the Lambda functions.
        uiWrapper.DisplayTitle("Listing all Lambda functions.");
        var functions = await lambdaWrapper.ListFunctionsAsync();
        DisplayFunctionList(functions);

        uiWrapper.DisplayTitle("Invoke increment function");
        Console.WriteLine("Now that it has been created, invoke the Lambda
increment function.");
        string? value;
        do
        {
            Console.Write("Enter a value to increment: ");
            value = Console.ReadLine();
        }
        while (string.IsNullOrEmpty(value));
```

```
        string functionParameters = "{" +
            "\"action\": \"increment\", " +
            "\"x\": \"" + value + "\"" +
        "}";
        var answer = await lambdaWrapper.InvokeFunctionAsync(functionName,
functionParameters);
        Console.WriteLine($"{value} + 1 = {answer}.");

        uiWrapper.DisplayTitle("Update function");
        Console.WriteLine("Now update the Lambda function code.");
        await lambdaWrapper.UpdateFunctionCodeAsync(functionName, bucketName,
calculatorKey);

        do
        {
            config = await lambdaWrapper.GetFunctionAsync(functionName);
            Console.Write(".");
        }
        while (config.LastUpdateStatus == LastUpdateStatus.InProgress);

        await lambdaWrapper.UpdateFunctionConfigurationAsync(
            functionName,
            calculatorHandler,
            new Dictionary<string, string> { { "LOG_LEVEL", "DEBUG" } });

        do
        {
            config = await lambdaWrapper.GetFunctionAsync(functionName);
            Console.Write(".");
        }
        while (config.LastUpdateStatus == LastUpdateStatus.InProgress);

        uiWrapper.DisplayTitle("Call updated function");
        Console.WriteLine("Now call the updated function...");

        bool done = false;

        do
        {
            string? opSelected;

            Console.WriteLine("Select the operation to perform:");
            Console.WriteLine("\t1. add");
            Console.WriteLine("\t2. subtract");
```

```
            Console.WriteLine("\t3. multiply");
            Console.WriteLine("\t4. divide");
            Console.WriteLine("\tOr enter \"q\" to quit.");
            Console.WriteLine("Enter the number (1, 2, 3, 4, or q) of the
operation you want to perform: ");
            do
            {
                Console.Write("Your choice? ");
                opSelected = Console.ReadLine();
            }
            while (opSelected == string.Empty);

            var operation = (opSelected) switch
            {
                "1" => "add",
                "2" => "subtract",
                "3" => "multiply",
                "4" => "divide",
                "q" => "quit",
                _ => "add",
            };

            if (operation == "quit")
            {
                done = true;
            }
            else
            {
                // Get two numbers and an action from the user.
                value = string.Empty;
                do
                {
                    Console.Write("Enter the first value: ");
                    value = Console.ReadLine();
                }
                while (value == string.Empty);

                string? value2;
                do
                {
                    Console.Write("Enter a second value: ");
                    value2 = Console.ReadLine();
                }
                while (value2 == string.Empty);
```

```
                functionParameters = "{" +
                    "\"action\": \"" + operation + "\", " +
                    "\"x\": \"" + value + "\"," +
                    "\"y\": \"" + value2 + "\"" +
                "}";

                answer = await lambdaWrapper.InvokeFunctionAsync(functionName,
functionParameters);
                Console.WriteLine($"The answer when we {operation} the two
numbers is: {answer}.");
            }

            uiWrapper.PressEnter();
        } while (!done);

        // Delete the function created earlier.

        uiWrapper.DisplayTitle("Clean up resources");
        // Detach the IAM policy from the IAM role.
        Console.WriteLine("First detach the IAM policy from the role.");
        success = await lambdaRoleWrapper.DetachLambdaRolePolicyAsync(policyArn,
roleName);
        uiWrapper.WaitABit(15, "Let's wait for the policy to be fully detached
from the role.");

        Console.WriteLine("Delete the AWS Lambda function.");
        success = await lambdaWrapper.DeleteFunctionAsync(functionName);
        if (success)
        {
            Console.WriteLine($"The {functionName} function was deleted.");
        }
        else
        {
            Console.WriteLine($"Could not remove the function {functionName}");
        }

        // Now delete the IAM role created for use with the functions
        // created by the application.
        Console.WriteLine("Now we can delete the role that we created.");
        success = await lambdaRoleWrapper.DeleteLambdaRoleAsync(roleName);
        if (success)
        {
            Console.WriteLine("The role has been successfully removed.");
```

```
        }
        else
        {
            Console.WriteLine("Couldn't delete the role.");
        }

        Console.WriteLine("The Lambda Scenario is now complete.");
        uiWrapper.PressEnter();

        // Displays a formatted list of existing functions returned by the
        // LambdaMethods.ListFunctions.
        void DisplayFunctionList(List<FunctionConfiguration> functions)
        {
            functions.ForEach(functionConfig =>
            {

 Console.WriteLine($"{functionConfig.FunctionName}\t{functionConfig.Description}");
            });
        }
    }
}


namespace LambdaActions;

using Amazon.IdentityManagement;
using Amazon.IdentityManagement.Model;

public class LambdaRoleWrapper
{
    private readonly IAmazonIdentityManagementService _lambdaRoleService;

    public LambdaRoleWrapper(IAmazonIdentityManagementService lambdaRoleService)
    {
        _lambdaRoleService = lambdaRoleService;
    }

    /// <summary>
    /// Attach an AWS Identity and Access Management (IAM) role policy to the
    /// IAM role to be assumed by the AWS Lambda functions created for the
 scenario.
    /// </summary>
    /// <param name="policyArn">The Amazon Resource Name (ARN) of the IAM
 policy.</param>
```

```
    /// <param name="roleName">The name of the IAM role to attach the IAM policy
 to.</param>
    /// <returns>A Boolean value indicating the success of the action.</returns>
    public async Task<bool> AttachLambdaRolePolicyAsync(string policyArn, string
 roleName)
    {
        var response = await _lambdaRoleService.AttachRolePolicyAsync(new
 AttachRolePolicyRequest { PolicyArn = policyArn, RoleName = roleName });
        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }

    /// <summary>
    /// Create a new IAM role.
    /// </summary>
    /// <param name="roleName">The name of the IAM role to create.</param>
    /// <param name="policyDocument">The policy document for the new IAM role.</
param>
    /// <returns>A string representing the ARN for newly created role.</returns>
    public async Task<string> CreateLambdaRoleAsync(string roleName, string
 policyDocument)
    {
        var request = new CreateRoleRequest
        {
            AssumeRolePolicyDocument = policyDocument,
            RoleName = roleName,
        };

        var response = await _lambdaRoleService.CreateRoleAsync(request);
        return response.Role.Arn;
    }

    /// <summary>
    /// Deletes an IAM role.
    /// </summary>
    /// <param name="roleName">The name of the role to delete.</param>
    /// <returns>A Boolean value indicating the success of the operation.</
returns>
    public async Task<bool> DeleteLambdaRoleAsync(string roleName)
    {
        var request = new DeleteRoleRequest
        {
            RoleName = roleName,
        };
```

```
        var response = await _lambdaRoleService.DeleteRoleAsync(request);
        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }

    public async Task<bool> DetachLambdaRolePolicyAsync(string policyArn, string
 roleName)
    {
        var response = await _lambdaRoleService.DetachRolePolicyAsync(new
 DetachRolePolicyRequest { PolicyArn = policyArn, RoleName = roleName });
        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }
}


namespace LambdaScenarioCommon;
public class UIWrapper
{
    public readonly string SepBar = new('-', Console.WindowWidth);

    /// <summary>
    /// Show information about the AWS Lambda Basics scenario.
    /// </summary>
    public void DisplayLambdaBasicsOverview()
    {
        Console.Clear();

        DisplayTitle("Welcome to AWS Lambda Basics");
        Console.WriteLine("This example application does the following:");
        Console.WriteLine("\t1. Creates an AWS Identity and Access Management
 (IAM) role that will be assumed by the functions we create.");
        Console.WriteLine("\t2. Attaches an IAM role policy that has Lambda
 permissions.");
        Console.WriteLine("\t3. Creates a Lambda function that increments the
 value passed to it.");
        Console.WriteLine("\t4. Calls the increment function and passes a
 value.");
        Console.WriteLine("\t5. Updates the code so that the function is a simple
 calculator.");
        Console.WriteLine("\t6. Calls the calculator function with the values
 entered.");
        Console.WriteLine("\t7. Deletes the Lambda function.");
        Console.WriteLine("\t7. Detaches the IAM role policy.");
        Console.WriteLine("\t8. Deletes the IAM role.");
        PressEnter();
```

```
    }

    /// <summary>
    /// Display a message and wait until the user presses enter.
    /// </summary>
    public void PressEnter()
    {
        Console.Write("\nPress <Enter> to continue. ");
        _ = Console.ReadLine();
        Console.WriteLine();
    }

    /// <summary>
    /// Pad a string with spaces to center it on the console display.
    /// </summary>
    /// <param name="strToCenter">The string to be centered.</param>
    /// <returns>The padded string.</returns>
    public string CenterString(string strToCenter)
    {
        var padAmount = (Console.WindowWidth - strToCenter.Length) / 2;
        var leftPad = new string(' ', padAmount);
        return $"{leftPad}{strToCenter}";
    }

    /// <summary>
    /// Display a line of hyphens, the centered text of the title and another
    /// line of hyphens.
    /// </summary>
    /// <param name="strTitle">The string to be displayed.</param>
    public void DisplayTitle(string strTitle)
    {
        Console.WriteLine(SepBar);
        Console.WriteLine(CenterString(strTitle));
        Console.WriteLine(SepBar);
    }

    /// <summary>
    /// Display a countdown and wait for a number of seconds.
    /// </summary>
    /// <param name="numSeconds">The number of seconds to wait.</param>
    public void WaitABit(int numSeconds, string msg)
    {
        Console.WriteLine(msg);
```

```
        // Wait for the requested number of seconds.
        for (int i = numSeconds; i > 0; i--)
        {
            System.Threading.Thread.Sleep(1000);
            Console.Write($"{i}...");
        }

        PressEnter();
    }
}
```

定義增量一個數字的 Lambda 處理常式。

```
using Amazon.Lambda.Core;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace LambdaIncrement;

public class Function
{

    /// <summary>
    /// A simple function increments the integer parameter.
    /// </summary>
    /// <param name="input">A JSON string containing an action, which must be
    /// "increment" and a string representing the value to increment.</param>
    /// <param name="context">The context object passed by Lambda containing
    /// information about invocation, function, and execution environment.</
param>
    /// <returns>A string representing the incremented value of the parameter.</
returns>
    public int FunctionHandler(Dictionary<string, string> input, ILambdaContext
 context)
    {
        if (input["action"] == "increment")
        {
            int inputValue = Convert.ToInt32(input["x"]);
```

```
            return inputValue + 1;
        }
        else
        {
            return 0;
        }
    }
}
```

定義可執行算術運算的第二個 Lambda 處理常式。

```
using Amazon.Lambda.Core;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace LambdaCalculator;

public class Function
{

    /// <summary>
    /// A simple function that takes two number in string format and performs
    /// the requested arithmetic function.
    /// </summary>
    /// <param name="input">JSON data containing an action, and x and y values.
    /// Valid actions include: add, subtract, multiply, and divide.</param>
    /// <param name="context">The context object passed by Lambda containing
    /// information about invocation, function, and execution environment.</
param>
    /// <returns>A string representing the results of the calculation.</returns>
    public int FunctionHandler(Dictionary<string, string> input, ILambdaContext
 context)
    {
        var action = input["action"];
        int x = Convert.ToInt32(input["x"]);
        int y = Convert.ToInt32(input["y"]);
        int result;
        switch (action)
```

```
        {
            case "add":
                result = x + y;
                break;
            case "subtract":
                result = x - y;
                break;
            case "multiply":
                result = x * y;
                break;
            case "divide":
                if (y == 0)
                {
                    Console.Error.WriteLine("Divide by zero error.");
                    result = 0;
                }
                else
                    result = x / y;
                break;
            default:
                Console.Error.WriteLine($"{action} is not a valid operation.");
                result = 0;
                break;
        }
        return result;
    }
}
```

- 如需 API 詳細資訊，請參閱《AWS SDK for .NET API 參考》中的下列主題。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunction代碼
  - UpdateFunction配置

C++

適用於 C++ 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
//! Get started with functions scenario.
/*!
 \param clientConfig: AWS client configuration.
 \return bool: Successful completion.
 */
bool AwsDoc::Lambda::getStartedWithFunctionsScenario(
        const Aws::Client::ClientConfiguration &clientConfig) {

    Aws::Lambda::LambdaClient client(clientConfig);

    // 1. Create an AWS Identity and Access Management (IAM) role for Lambda
 function.
    Aws::String roleArn;
    if (!getIamRoleArn(roleArn, clientConfig)) {
        return false;
    }

    // 2. Create a Lambda function.
    int seconds = 0;
    do {
        Aws::Lambda::Model::CreateFunctionRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        request.SetDescription(LAMBDA_DESCRIPTION); // Optional.
#if USE_CPP_LAMBDA_FUNCTION
        request.SetRuntime(Aws::Lambda::Model::Runtime::provided_al2);
        request.SetTimeout(15);
        request.SetMemorySize(128);

        // Assume the AWS Lambda function was built in Docker with same
 architecture
        // as this code.
#if  defined(__x86_64__)
```

```
        request.SetArchitectures({Aws::Lambda::Model::Architecture::x86_64});
#elif defined(__aarch64__)
        request.SetArchitectures({Aws::Lambda::Model::Architecture::arm64});
#else
#error "Unimplemented architecture"
#endif // defined(architecture)
#else
        request.SetRuntime(Aws::Lambda::Model::Runtime::python3_8);
#endif
        request.SetRole(roleArn);
        request.SetHandler(LAMBDA_HANDLER_NAME);
        request.SetPublish(true);
        Aws::Lambda::Model::FunctionCode code;
        std::ifstream ifstream(INCREMENT_LAMBDA_CODE.c_str(),
                               std::ios_base::in | std::ios_base::binary);
        if (!ifstream.is_open()) {
            std::cerr << "Error opening file " << INCREMENT_LAMBDA_CODE << "." <<
 std::endl;

#if USE_CPP_LAMBDA_FUNCTION
            std::cerr
                    << "The cpp Lambda function must be built following the
 instructions in the cpp_lambda/README.md file. "
                    << std::endl;
#endif
            deleteIamRole(clientConfig);
            return false;
        }

        Aws::StringStream buffer;
        buffer << ifstream.rdbuf();

        code.SetZipFile(Aws::Utils::ByteBuffer((unsigned char *)
 buffer.str().c_str(),
                                               buffer.str().length()));
        request.SetCode(code);

        Aws::Lambda::Model::CreateFunctionOutcome outcome =
 client.CreateFunction(
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda function was successfully created. " <<
 seconds
```

```
                      << " seconds elapsed." << std::endl;
            break;
        }
        else if (outcome.GetError().GetErrorType() ==
                 Aws::Lambda::LambdaErrors::INVALID_PARAMETER_VALUE &&
                 outcome.GetError().GetMessage().find("role") >= 0) {
            if ((seconds % 5) == 0) { // Log status every 10 seconds.
                std::cout
                        << "Waiting for the IAM role to become available as a
CreateFunction parameter. "
                        << seconds
                        << " seconds elapsed." << std::endl;

                std::cout << outcome.GetError().GetMessage() << std::endl;
            }
        }
        else {
            std::cerr << "Error with CreateFunction. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
            deleteIamRole(clientConfig);
            return false;
        }
        ++seconds;
        std::this_thread::sleep_for(std::chrono::seconds(1));
    } while (60 > seconds);

    std::cout << "The current Lambda function increments 1 by an input." <<
std::endl;

    // 3.  Invoke the Lambda function.
    {
        int increment = askQuestionForInt("Enter an increment integer: ");

        Aws::Lambda::Model::InvokeResult invokeResult;
        Aws::Utils::Json::JsonValue jsonPayload;
        jsonPayload.WithString("action", "increment");
        jsonPayload.WithInteger("number", increment);
        if (invokeLambdaFunction(jsonPayload, Aws::Lambda::Model::LogType::Tail,
                                 invokeResult, client)) {
            Aws::Utils::Json::JsonValue jsonValue(invokeResult.GetPayload());
            Aws::Map<Aws::String, Aws::Utils::Json::JsonView> values =
                    jsonValue.View().GetAllObjects();
            auto iter = values.find("result");
```

```
            if (iter != values.end() && iter->second.IsIntegerType()) {
                {
                    std::cout << INCREMENT_RESUlT_PREFIX
                              << iter->second.AsInteger() << std::endl;
                }
            }
            else {
                std::cout << "There was an error in execution. Here is the log."
                          << std::endl;
                Aws::Utils::ByteBuffer buffer =
 Aws::Utils::HashingUtils::Base64Decode(
                        invokeResult.GetLogResult());
                std::cout << "With log " << buffer.GetUnderlyingData() <<
 std::endl;
            }
        }
    }

    std::cout
            << "The Lambda function will now be updated with new code. Press
 return to continue, ";
    Aws::String answer;
    std::getline(std::cin, answer);

    // 4.  Update the Lambda function code.
    {
        Aws::Lambda::Model::UpdateFunctionCodeRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        std::ifstream ifstream(CALCULATOR_LAMBDA_CODE.c_str(),
                               std::ios_base::in | std::ios_base::binary);
        if (!ifstream.is_open()) {
            std::cerr << "Error opening file " << INCREMENT_LAMBDA_CODE << "." <<
 std::endl;

#if USE_CPP_LAMBDA_FUNCTION
            std::cerr
                    << "The cpp Lambda function must be built following the
 instructions in the cpp_lambda/README.md file. "
                    << std::endl;
#endif
            deleteLambdaFunction(client);
            deleteIamRole(clientConfig);
            return false;
        }
```

```
        Aws::StringStream buffer;
        buffer << ifstream.rdbuf();
        request.SetZipFile(
                Aws::Utils::ByteBuffer((unsigned char *) buffer.str().c_str(),
                                       buffer.str().length()));
        request.SetPublish(true);

        Aws::Lambda::Model::UpdateFunctionCodeOutcome outcome =
client.UpdateFunctionCode(
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda code was successfully updated." <<
std::endl;
        }
        else {
            std::cerr << "Error with Lambda::UpdateFunctionCode. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }
    }

    std::cout
            << "This function uses an environment variable to control the logging
level."
            << std::endl;
    std::cout
            << "UpdateFunctionConfiguration will be used to set the LOG_LEVEL to
DEBUG."
            << std::endl;
    seconds = 0;

    // 5.  Update the Lambda function configuration.
    do {
        ++seconds;
        std::this_thread::sleep_for(std::chrono::seconds(1));
        Aws::Lambda::Model::UpdateFunctionConfigurationRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        Aws::Lambda::Model::Environment environment;
        environment.AddVariables("LOG_LEVEL", "DEBUG");
        request.SetEnvironment(environment);
```

```
        Aws::Lambda::Model::UpdateFunctionConfigurationOutcome outcome =
 client.UpdateFunctionConfiguration(
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda configuration was successfully updated."
                      << std::endl;
            break;
        }

            // RESOURCE_IN_USE: function code update not completed.
        else if (outcome.GetError().GetErrorType() !=
                 Aws::Lambda::LambdaErrors::RESOURCE_IN_USE) {
            if ((seconds % 10) == 0) { // Log status every 10 seconds.
                std::cout << "Lambda function update in progress . After " <<
 seconds
                          << " seconds elapsed." << std::endl;
            }
        }
        else {
            std::cerr << "Error with Lambda::UpdateFunctionConfiguration. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }

    } while (0 < seconds);

    if (0 > seconds) {
        std::cerr << "Function failed to become active." << std::endl;
    }
    else {
        std::cout << "Updated function active after " << seconds << " seconds."
                  << std::endl;
    }

    std::cout
            << "\nThe new code applies an arithmetic operator to two variables, x
 an y."
            << std::endl;
    std::vector<Aws::String> operators = {"plus", "minus", "times", "divided-
by"};
    for (size_t i = 0; i < operators.size(); ++i) {
        std::cout << "   " << i + 1 << " " << operators[i] << std::endl;
    }
```

```
   // 6.  Invoke the updated Lambda function.
   do {
       int operatorIndex = askQuestionForIntRange("Select an operator index 1 -
4 ", 1,
                                                  4);
       int x = askQuestionForInt("Enter an integer for the x value ");
       int y = askQuestionForInt("Enter an integer for the y value ");

       Aws::Utils::Json::JsonValue calculateJsonPayload;
       calculateJsonPayload.WithString("action", operators[operatorIndex - 1]);
       calculateJsonPayload.WithInteger("x", x);
       calculateJsonPayload.WithInteger("y", y);
       Aws::Lambda::Model::InvokeResult calculatedResult;
       if (invokeLambdaFunction(calculateJsonPayload,
                                Aws::Lambda::Model::LogType::Tail,
                                calculatedResult, client)) {
           Aws::Utils::Json::JsonValue jsonValue(calculatedResult.GetPayload());
           Aws::Map<Aws::String, Aws::Utils::Json::JsonView> values =
                   jsonValue.View().GetAllObjects();
           auto iter = values.find("result");
           if (iter != values.end() && iter->second.IsIntegerType()) {
               std::cout << ARITHMETIC_RESULT_PREFIX << x << " "
                         << operators[operatorIndex - 1] << " "
                         << y << " is " << iter->second.AsInteger() <<
std::endl;
           }
           else if (iter != values.end() && iter->second.IsFloatingPointType())
{
               std::cout << ARITHMETIC_RESULT_PREFIX << x << " "
                         << operators[operatorIndex - 1] << " "
                         << y << " is " << iter->second.AsDouble() << std::endl;
           }
           else {
               std::cout << "There was an error in execution. Here is the log."
                         << std::endl;
               Aws::Utils::ByteBuffer buffer =
Aws::Utils::HashingUtils::Base64Decode(
                        calculatedResult.GetLogResult());
               std::cout << "With log " << buffer.GetUnderlyingData() <<
std::endl;
           }
       }
```

```
        answer = askQuestion("Would you like to try another operation? (y/n) ");
    } while (answer == "y");

    std::cout
            << "A list of the lambda functions will be retrieved. Press return to
continue, ";
    std::getline(std::cin, answer);

    // 7.  List the Lambda functions.

    std::vector<Aws::String> functions;
    Aws::String marker;

    do {
        Aws::Lambda::Model::ListFunctionsRequest request;
        if (!marker.empty()) {
            request.SetMarker(marker);
        }

        Aws::Lambda::Model::ListFunctionsOutcome outcome = client.ListFunctions(
                request);

        if (outcome.IsSuccess()) {
            const Aws::Lambda::Model::ListFunctionsResult &result =
outcome.GetResult();
            std::cout << result.GetFunctions().size()
                      << " lambda functions were retrieved." << std::endl;

            for (const Aws::Lambda::Model::FunctionConfiguration
&functionConfiguration: result.GetFunctions()) {
                functions.push_back(functionConfiguration.GetFunctionName());
                std::cout << functions.size() << "  "
                          << functionConfiguration.GetDescription() << std::endl;
                std::cout << "   "
                          <<
Aws::Lambda::Model::RuntimeMapper::GetNameForRuntime(
                                  functionConfiguration.GetRuntime()) << ": "
                          << functionConfiguration.GetHandler()
                          << std::endl;
            }
            marker = result.GetNextMarker();
        }
        else {
            std::cerr << "Error with Lambda::ListFunctions. "
```

```
                    << outcome.GetError().GetMessage()
                    << std::endl;
        }
    } while (!marker.empty());

    // 8.  Get a Lambda function.
    if (!functions.empty()) {
        std::stringstream question;
        question << "Choose a function to retrieve between 1 and " <<
functions.size()
                 << " ";
        int functionIndex = askQuestionForIntRange(question.str(), 1,

static_cast<int>(functions.size()));

        Aws::String functionName = functions[functionIndex - 1];

        Aws::Lambda::Model::GetFunctionRequest request;
        request.SetFunctionName(functionName);

        Aws::Lambda::Model::GetFunctionOutcome outcome =
client.GetFunction(request);

        if (outcome.IsSuccess()) {
            std::cout << "Function retrieve.\n" <<

outcome.GetResult().GetConfiguration().Jsonize().View().WriteReadable()
                    << std::endl;
        }
        else {
            std::cerr << "Error with Lambda::GetFunction. "
                    << outcome.GetError().GetMessage()
                    << std::endl;
        }
    }

    std::cout << "The resources will be deleted. Press return to continue, ";
    std::getline(std::cin, answer);

    // 9.  Delete the Lambda function.
    bool result = deleteLambdaFunction(client);

    // 10. Delete the IAM role.
    return result && deleteIamRole(clientConfig);
```

```
}

//! Routine which invokes a Lambda function and returns the result.
/*!
 \param jsonPayload: Payload for invoke function.
 \param logType: Log type setting for invoke function.
 \param invokeResult: InvokeResult object to receive the result.
 \param client: Lambda client.
 \return bool: Successful completion.
 */
bool
AwsDoc::Lambda::invokeLambdaFunction(const Aws::Utils::Json::JsonValue
 &jsonPayload,
                                     Aws::Lambda::Model::LogType logType,
                                     Aws::Lambda::Model::InvokeResult
 &invokeResult,
                                     const Aws::Lambda::LambdaClient &client) {
    int seconds = 0;
    bool result = false;
    /*
     * In this example, the Invoke function can be called before recently created
 resources are
     * available.  The Invoke function is called repeatedly until the resources
 are
     * available.
     */
    do {
        Aws::Lambda::Model::InvokeRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        request.SetLogType(logType);
        std::shared_ptr<Aws::IOStream> payload =
 Aws::MakeShared<Aws::StringStream>(
                "FunctionTest");
        *payload << jsonPayload.View().WriteReadable();
        request.SetBody(payload);
        request.SetContentType("application/json");
        Aws::Lambda::Model::InvokeOutcome outcome = client.Invoke(request);

        if (outcome.IsSuccess()) {
            invokeResult = std::move(outcome.GetResult());
            result = true;
            break;
        }
```

```
            // ACCESS_DENIED: because the role is not available yet.
            // RESOURCE_CONFLICT: because the Lambda function is being created or
 updated.
        else if ((outcome.GetError().GetErrorType() ==
                  Aws::Lambda::LambdaErrors::ACCESS_DENIED) ||
                 (outcome.GetError().GetErrorType() ==
                  Aws::Lambda::LambdaErrors::RESOURCE_CONFLICT)) {
            if ((seconds % 5) == 0) { // Log status every 10 seconds.
                std::cout << "Waiting for the invoke api to be available, status
 " <<
                          ((outcome.GetError().GetErrorType() ==
                            Aws::Lambda::LambdaErrors::ACCESS_DENIED ?
                            "ACCESS_DENIED" : "RESOURCE_CONFLICT")) << ". " <<
 seconds
                          << " seconds elapsed." << std::endl;
            }
        }
        else {
            std::cerr << "Error with Lambda::InvokeRequest. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
            break;
        }
        ++seconds;
        std::this_thread::sleep_for(std::chrono::seconds(1));
    } while (seconds < 60);

    return result;
}
```

- 如需 API 詳細資訊，請參閱《AWS SDK for C++ API 參考》中的下列主題。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunction代碼
  - UpdateFunction配置

Go

SDK for Go V2

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

建立互動式案例，示範如何開始使用 Lambda 函數。

```
// GetStartedFunctionsScenario shows you how to use AWS Lambda to perform the
 following
// actions:
//
//  1. Create an AWS Identity and Access Management (IAM) role and Lambda
 function, then upload handler code.
//  2. Invoke the function with a single parameter and get results.
//  3. Update the function code and configure with an environment variable.
//  4. Invoke the function with new parameters and get results. Display the
 returned execution log.
//  5. List the functions for your account, then clean up resources.
type GetStartedFunctionsScenario struct {
 sdkConfig       aws.Config
 functionWrapper actions.FunctionWrapper
 questioner      demotools.IQuestioner
 helper          IScenarioHelper
 isTestRun       bool
}

// NewGetStartedFunctionsScenario constructs a GetStartedFunctionsScenario
 instance from a configuration.
// It uses the specified config to get a Lambda client and create wrappers for
 the actions
// used in the scenario.
func NewGetStartedFunctionsScenario(sdkConfig aws.Config, questioner
 demotools.IQuestioner,
 helper IScenarioHelper) GetStartedFunctionsScenario {
 lambdaClient := lambda.NewFromConfig(sdkConfig)
 return GetStartedFunctionsScenario{
  sdkConfig:       sdkConfig,
```

```
  functionWrapper: actions.FunctionWrapper{LambdaClient: lambdaClient},
  questioner:      questioner,
  helper:          helper,
 }
}

// Run runs the interactive scenario.
func (scenario GetStartedFunctionsScenario) Run() {
 defer func() {
  if r := recover(); r != nil {
   log.Printf("Something went wrong with the demo.\n")
  }
 }()

 log.Println(strings.Repeat("-", 88))
 log.Println("Welcome to the AWS Lambda get started with functions demo.")
 log.Println(strings.Repeat("-", 88))

 role := scenario.GetOrCreateRole()
 funcName := scenario.CreateFunction(role)
 scenario.InvokeIncrement(funcName)
 scenario.UpdateFunction(funcName)
 scenario.InvokeCalculator(funcName)
 scenario.ListFunctions()
 scenario.Cleanup(role, funcName)

 log.Println(strings.Repeat("-", 88))
 log.Println("Thanks for watching!")
 log.Println(strings.Repeat("-", 88))
}

// GetOrCreateRole checks whether the specified role exists and returns it if it
 does.
// Otherwise, a role is created that specifies Lambda as a trusted principal.
// The AWSLambdaBasicExecutionRole managed policy is attached to the role and the
 role
// is returned.
func (scenario GetStartedFunctionsScenario) GetOrCreateRole() *iamtypes.Role {
 var role *iamtypes.Role
 iamClient := iam.NewFromConfig(scenario.sdkConfig)
 log.Println("First, we need an IAM role that Lambda can assume.")
 roleName := scenario.questioner.Ask("Enter a name for the role:",
 demotools.NotEmpty{})
 getOutput, err := iamClient.GetRole(context.TODO(), &iam.GetRoleInput{
```

```
 RoleName: aws.String(roleName)})
if err != nil {
 var noSuch *iamtypes.NoSuchEntityException
 if errors.As(err, &noSuch) {
  log.Printf("Role %v doesn't exist. Creating it....\n", roleName)
 } else {
  log.Panicf("Couldn't check whether role %v exists. Here's why: %v\n",
   roleName, err)
 }
} else {
 role = getOutput.Role
 log.Printf("Found role %v.\n", *role.RoleName)
}
if role == nil {
 trustPolicy := PolicyDocument{
  Version: "2012-10-17",
  Statement: []PolicyStatement{{
   Effect:    "Allow",
   Principal: map[string]string{"Service": "lambda.amazonaws.com"},
   Action:    []string{"sts:AssumeRole"},
  }},
 }
 policyArn := "arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole"
 createOutput, err := iamClient.CreateRole(context.TODO(), &iam.CreateRoleInput{
  AssumeRolePolicyDocument: aws.String(trustPolicy.String()),
  RoleName:                 aws.String(roleName),
 })
 if err != nil {
  log.Panicf("Couldn't create role %v. Here's why: %v\n", roleName, err)
 }
 role = createOutput.Role
 _, err = iamClient.AttachRolePolicy(context.TODO(), &iam.AttachRolePolicyInput{
  PolicyArn: aws.String(policyArn),
  RoleName:  aws.String(roleName),
 })
 if err != nil {
  log.Panicf("Couldn't attach a policy to role %v. Here's why: %v\n", roleName,
err)
 }
 log.Printf("Created role %v.\n", *role.RoleName)
 log.Println("Let's give AWS a few seconds to propagate resources...")
 scenario.helper.Pause(10)
}
log.Println(strings.Repeat("-", 88))
```

```
 return role
}

// CreateFunction creates a Lambda function and uploads a handler written in
 Python.
// The code for the Python handler is packaged as a []byte in .zip format.
func (scenario GetStartedFunctionsScenario) CreateFunction(role *iamtypes.Role)
 string {
 log.Println("Let's create a function that increments a number.\n" +
  "The function uses the 'lambda_handler_basic.py' script found in the \n" +
  "'handlers' directory of this project.")
 funcName := scenario.questioner.Ask("Enter a name for the Lambda function:",
 demotools.NotEmpty{})
 zipPackage := scenario.helper.CreateDeploymentPackage("lambda_handler_basic.py",
 fmt.Sprintf("%v.py", funcName))
 log.Printf("Creating function %v and waiting for it to be ready.", funcName)
 funcState := scenario.functionWrapper.CreateFunction(funcName,
 fmt.Sprintf("%v.lambda_handler", funcName),
  role.Arn, zipPackage)
 log.Printf("Your function is %v.", funcState)
 log.Println(strings.Repeat("-", 88))
 return funcName
}

// InvokeIncrement invokes a Lambda function that increments a number. The
 function
// parameters are contained in a Go struct that is used to serialize the
 parameters to
// a JSON payload that is passed to the function.
// The result payload is deserialized into a Go struct that contains an int
 value.
func (scenario GetStartedFunctionsScenario) InvokeIncrement(funcName string) {
 parameters := actions.IncrementParameters{Action: "increment"}
 log.Println("Let's invoke our function. This function increments a number.")
 parameters.Number = scenario.questioner.AskInt("Enter a number to increment:",
 demotools.NotEmpty{})
 log.Printf("Invoking %v with %v...\n", funcName, parameters.Number)
 invokeOutput := scenario.functionWrapper.Invoke(funcName, parameters, false)
 var payload actions.LambdaResultInt
 err := json.Unmarshal(invokeOutput.Payload, &payload)
 if err != nil {
  log.Panicf("Couldn't unmarshal payload from invoking %v. Here's why: %v\n",
   funcName, err)
 }
```

```
 log.Printf("Invoking %v with %v returned %v.\n", funcName, parameters.Number,
 payload)
 log.Println(strings.Repeat("-", 88))
}

// UpdateFunction updates the code for a Lambda function by uploading a simple
 arithmetic
// calculator written in Python. The code for the Python handler is packaged as a
// []byte in .zip format.
// After the code is updated, the configuration is also updated with a new log
// level that instructs the handler to log additional information.
func (scenario GetStartedFunctionsScenario) UpdateFunction(funcName string) {
 log.Println("Let's update the function to an arithmetic calculator.\n" +
  "The function uses the 'lambda_handler_calculator.py' script found in the \n" +
  "'handlers' directory of this project.")
 scenario.questioner.Ask("Press Enter when you're ready.")
 log.Println("Creating deployment package...")
 zipPackage :=
 scenario.helper.CreateDeploymentPackage("lambda_handler_calculator.py",
  fmt.Sprintf("%v.py", funcName))
 log.Println("...and updating the Lambda function and waiting for it to be
 ready.")
 funcState := scenario.functionWrapper.UpdateFunctionCode(funcName, zipPackage)
 log.Printf("Updated function %v. Its current state is %v.", funcName, funcState)
 log.Println("This function uses an environment variable to control logging
 level.")
 log.Println("Let's set it to DEBUG to get the most logging.")
 scenario.functionWrapper.UpdateFunctionConfiguration(funcName,
  map[string]string{"LOG_LEVEL": "DEBUG"})
 log.Println(strings.Repeat("-", 88))
}

// InvokeCalculator invokes the Lambda calculator function. The parameters are
 stored in a
// Go struct that is used to serialize the parameters to a JSON payload. That
 payload is then passed
// to the function.
// The result payload is deserialized to a Go struct that stores the result as
 either an
// int or float32, depending on the kind of operation that was specified.
func (scenario GetStartedFunctionsScenario) InvokeCalculator(funcName string) {
 wantInvoke := true
 choices := []string{"plus", "minus", "times", "divided-by"}
 for wantInvoke {
```

```
  choice := scenario.questioner.AskChoice("Select an arithmetic operation:\n",
 choices)
  x := scenario.questioner.AskInt("Enter a value for x:", demotools.NotEmpty{})
  y := scenario.questioner.AskInt("Enter a value for y:", demotools.NotEmpty{})
  log.Printf("Invoking %v %v %v...", x, choices[choice], y)
  calcParameters := actions.CalculatorParameters{
   Action: choices[choice],
   X:      x,
   Y:      y,
  }
  invokeOutput := scenario.functionWrapper.Invoke(funcName, calcParameters, true)
  var payload any
  if choice == 3 { // divide-by results in a float.
   payload = actions.LambdaResultFloat{}
  } else {
   payload = actions.LambdaResultInt{}
  }
  err := json.Unmarshal(invokeOutput.Payload, &payload)
  if err != nil {
   log.Panicf("Couldn't unmarshal payload from invoking %v. Here's why: %v\n",
    funcName, err)
  }
  log.Printf("Invoking %v with %v %v %v returned %v.\n", funcName,
   calcParameters.X, calcParameters.Action, calcParameters.Y, payload)
  scenario.questioner.Ask("Press Enter to see the logs from the call.")
  logRes, err := base64.StdEncoding.DecodeString(*invokeOutput.LogResult)
  if err != nil {
   log.Panicf("Couldn't decode log result. Here's why: %v\n", err)
  }
  log.Println(string(logRes))
  wantInvoke = scenario.questioner.AskBool("Do you want to calculate again? (y/
n)", "y")
 }
 log.Println(strings.Repeat("-", 88))
}

// ListFunctions lists up to the specified number of functions for your account.
func (scenario GetStartedFunctionsScenario) ListFunctions() {
 count := scenario.questioner.AskInt(
  "Let's list functions for your account. How many do you want to see?",
 demotools.NotEmpty{})
 functions := scenario.functionWrapper.ListFunctions(count)
 log.Printf("Found %v functions:", len(functions))
 for _, function := range functions {
```

```
  log.Printf("\t%v", *function.FunctionName)
 }
 log.Println(strings.Repeat("-", 88))
}

// Cleanup removes the IAM and Lambda resources created by the example.
func (scenario GetStartedFunctionsScenario) Cleanup(role *iamtypes.Role, funcName
 string) {
 if scenario.questioner.AskBool("Do you want to clean up resources created for
 this example? (y/n)",
  "y") {
  iamClient := iam.NewFromConfig(scenario.sdkConfig)
  policiesOutput, err := iamClient.ListAttachedRolePolicies(context.TODO(),
   &iam.ListAttachedRolePoliciesInput{RoleName: role.RoleName})
  if err != nil {
   log.Panicf("Couldn't get policies attached to role %v. Here's why: %v\n",
    *role.RoleName, err)
  }
  for _, policy := range policiesOutput.AttachedPolicies {
   _, err = iamClient.DetachRolePolicy(context.TODO(),
 &iam.DetachRolePolicyInput{
    PolicyArn: policy.PolicyArn, RoleName: role.RoleName,
   })
   if err != nil {
    log.Panicf("Couldn't detach policy %v from role %v. Here's why: %v\n",
     *policy.PolicyArn, *role.RoleName, err)
   }
  }
  _, err = iamClient.DeleteRole(context.TODO(), &iam.DeleteRoleInput{RoleName:
 role.RoleName})
  if err != nil {
   log.Panicf("Couldn't delete role %v. Here's why: %v\n", *role.RoleName, err)
  }
  log.Printf("Deleted role %v.\n", *role.RoleName)

  scenario.functionWrapper.DeleteFunction(funcName)
  log.Printf("Deleted function %v.\n", funcName)
 } else {
  log.Println("Okay. Don't forget to delete the resources when you're done with
 them.")
 }
}
```

建立包裝個別 Lambda 動作的結構。

```
// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}


// GetFunction gets data about the Lambda function specified by functionName.
func (wrapper FunctionWrapper) GetFunction(functionName string) types.State {
 var state types.State
 funcOutput, err := wrapper.LambdaClient.GetFunction(context.TODO(),
 &lambda.GetFunctionInput{
  FunctionName: aws.String(functionName),
 })
 if err != nil {
  log.Panicf("Couldn't get function %v. Here's why: %v\n", functionName, err)
 } else {
  state = funcOutput.Configuration.State
 }
 return state
}



// CreateFunction creates a new Lambda function from code contained in the
 zipPackage
// buffer. The specified handlerName must match the name of the file and function
// contained in the uploaded code. The role specified by iamRoleArn is assumed by
// Lambda and grants specific permissions.
// When the function already exists, types.StateActive is returned.
// When the function is created, a lambda.FunctionActiveV2Waiter is used to wait
 until the
// function is active.
func (wrapper FunctionWrapper) CreateFunction(functionName string, handlerName
 string,
 iamRoleArn *string, zipPackage *bytes.Buffer) types.State {
 var state types.State
```

```
 _, err := wrapper.LambdaClient.CreateFunction(context.TODO(),
 &lambda.CreateFunctionInput{
  Code:         &types.FunctionCode{ZipFile: zipPackage.Bytes()},
  FunctionName: aws.String(functionName),
  Role:         iamRoleArn,
  Handler:      aws.String(handlerName),
  Publish:      true,
  Runtime:      types.RuntimePython38,
 })
 if err != nil {
  var resConflict *types.ResourceConflictException
  if errors.As(err, &resConflict) {
   log.Printf("Function %v already exists.\n", functionName)
   state = types.StateActive
  } else {
   log.Panicf("Couldn't create function %v. Here's why: %v\n", functionName, err)
  }
 } else {
  waiter := lambda.NewFunctionActiveV2Waiter(wrapper.LambdaClient)
  funcOutput, err := waiter.WaitForOutput(context.TODO(),
 &lambda.GetFunctionInput{
   FunctionName: aws.String(functionName)}, 1*time.Minute)
  if err != nil {
   log.Panicf("Couldn't wait for function %v to be active. Here's why: %v\n",
 functionName, err)
  } else {
   state = funcOutput.Configuration.State
  }
 }
 return state
}



// UpdateFunctionCode updates the code for the Lambda function specified by
 functionName.
// The existing code for the Lambda function is entirely replaced by the code in
 the
// zipPackage buffer. After the update action is called, a
 lambda.FunctionUpdatedV2Waiter
// is used to wait until the update is successful.
func (wrapper FunctionWrapper) UpdateFunctionCode(functionName string, zipPackage
 *bytes.Buffer) types.State {
 var state types.State
```

```
 _, err := wrapper.LambdaClient.UpdateFunctionCode(context.TODO(),
 &lambda.UpdateFunctionCodeInput{
  FunctionName: aws.String(functionName), ZipFile: zipPackage.Bytes(),
 })
 if err != nil {
  log.Panicf("Couldn't update code for function %v. Here's why: %v\n",
 functionName, err)
 } else {
  waiter := lambda.NewFunctionUpdatedV2Waiter(wrapper.LambdaClient)
  funcOutput, err := waiter.WaitForOutput(context.TODO(),
 &lambda.GetFunctionInput{
   FunctionName: aws.String(functionName)}, 1*time.Minute)
  if err != nil {
   log.Panicf("Couldn't wait for function %v to be active. Here's why: %v\n",
 functionName, err)
  } else {
   state = funcOutput.Configuration.State
  }
 }
 return state
}



// UpdateFunctionConfiguration updates a map of environment variables configured
 for
// the Lambda function specified by functionName.
func (wrapper FunctionWrapper) UpdateFunctionConfiguration(functionName string,
 envVars map[string]string) {
 _, err := wrapper.LambdaClient.UpdateFunctionConfiguration(context.TODO(),
 &lambda.UpdateFunctionConfigurationInput{
  FunctionName: aws.String(functionName),
  Environment:  &types.Environment{Variables: envVars},
 })
 if err != nil {
  log.Panicf("Couldn't update configuration for %v. Here's why: %v",
 functionName, err)
 }
}



// ListFunctions lists up to maxItems functions for the account. This function
 uses a
```

```
// lambda.ListFunctionsPaginator to paginate the results.
func (wrapper FunctionWrapper) ListFunctions(maxItems int)
 []types.FunctionConfiguration {
 var functions []types.FunctionConfiguration
 paginator := lambda.NewListFunctionsPaginator(wrapper.LambdaClient,
 &lambda.ListFunctionsInput{
  MaxItems: aws.Int32(int32(maxItems)),
 })
 for paginator.HasMorePages() && len(functions) < maxItems {
  pageOutput, err := paginator.NextPage(context.TODO())
  if err != nil {
   log.Panicf("Couldn't list functions for your account. Here's why: %v\n", err)
  }
  functions = append(functions, pageOutput.Functions...)
 }
 return functions
}



// DeleteFunction deletes the Lambda function specified by functionName.
func (wrapper FunctionWrapper) DeleteFunction(functionName string) {
 _, err := wrapper.LambdaClient.DeleteFunction(context.TODO(),
 &lambda.DeleteFunctionInput{
  FunctionName: aws.String(functionName),
 })
 if err != nil {
  log.Panicf("Couldn't delete function %v. Here's why: %v\n", functionName, err)
 }
}



// Invoke invokes the Lambda function specified by functionName, passing the
 parameters
// as a JSON payload. When getLog is true, types.LogTypeTail is specified, which
 tells
// Lambda to include the last few log lines in the returned result.
func (wrapper FunctionWrapper) Invoke(functionName string, parameters any, getLog
 bool) *lambda.InvokeOutput {
 logType := types.LogTypeNone
 if getLog {
  logType = types.LogTypeTail
 }
```

```
 payload, err := json.Marshal(parameters)
 if err != nil {
  log.Panicf("Couldn't marshal parameters to JSON. Here's why %v\n", err)
 }
 invokeOutput, err := wrapper.LambdaClient.Invoke(context.TODO(),
 &lambda.InvokeInput{
  FunctionName: aws.String(functionName),
  LogType:      logType,
  Payload:      payload,
 })
 if err != nil {
  log.Panicf("Couldn't invoke function %v. Here's why: %v\n", functionName, err)
 }
 return invokeOutput
}



// IncrementParameters is used to serialize parameters to the increment Lambda
 handler.
type IncrementParameters struct {
 Action string `json:"action"`
 Number int    `json:"number"`
}

// CalculatorParameters is used to serialize parameters to the calculator Lambda
 handler.
type CalculatorParameters struct {
 Action string `json:"action"`
 X      int    `json:"x"`
 Y      int    `json:"y"`
}

// LambdaResultInt is used to deserialize an int result from a Lambda handler.
type LambdaResultInt struct {
 Result int `json:"result"`
}

// LambdaResultFloat is used to deserialize a float32 result from a Lambda
 handler.
type LambdaResultFloat struct {
 Result float32 `json:"result"`
}
```

建立實作函數的結構，以協助執行案例。

```
// IScenarioHelper abstracts I/O and wait functions from a scenario so that they
// can be mocked for unit testing.
type IScenarioHelper interface {
 Pause(secs int)
 CreateDeploymentPackage(sourceFile string, destinationFile string) *bytes.Buffer
}

// ScenarioHelper lets the caller specify the path to Lambda handler functions.
type ScenarioHelper struct {
 HandlerPath string
}

// Pause waits for the specified number of seconds.
func (helper *ScenarioHelper) Pause(secs int) {
 time.Sleep(time.Duration(secs) * time.Second)
}

// CreateDeploymentPackage creates an AWS Lambda deployment package from a source
 file. The
// deployment package is stored in .zip format in a bytes.Buffer. The buffer can
 be
// used to pass a []byte to Lambda when creating the function.
// The specified destinationFile is the name to give the file when it's deployed
 to Lambda.
func (helper *ScenarioHelper) CreateDeploymentPackage(sourceFile string,
 destinationFile string) *bytes.Buffer {
 var err error
 buffer := &bytes.Buffer{}
 writer := zip.NewWriter(buffer)
 zFile, err := writer.Create(destinationFile)
 if err != nil {
  log.Panicf("Couldn't create destination archive %v. Here's why: %v\n",
 destinationFile, err)
 }
 sourceBody, err := os.ReadFile(fmt.Sprintf("%v/%v", helper.HandlerPath,
 sourceFile))
 if err != nil {
  log.Panicf("Couldn't read handler source file %v. Here's why: %v\n",
```

```
   sourceFile, err)
 } else {
  _, err = zFile.Write(sourceBody)
  if err != nil {
   log.Panicf("Couldn't write handler %v to zip archive. Here's why: %v\n",
    sourceFile, err)
  }
 }
 err = writer.Close()
 if err != nil {
  log.Panicf("Couldn't close zip writer. Here's why: %v\n", err)
 }
 return buffer
}
```

定義增量一個數字的 Lambda 處理常式。

```
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)


def lambda_handler(event, context):
    """
    Accepts an action and a single number, performs the specified action on the
 number,
    and returns the result. The only allowable action is 'increment'.

    :param event: The event dict that contains the parameters sent when the
 function
                  is invoked.
    :param context: The context in which the function is called.
    :return: The result of the action.
    """
    result = None
    action = event.get("action")
    if action == "increment":
        result = event.get("number", 0) + 1
        logger.info("Calculated result of %s", result)
    else:
```

```
        logger.error("%s is not a valid action.", action)

    response = {"result": result}
    return response
```

定義可執行算術運算的第二個 Lambda 處理常式。

```
import logging
import os


logger = logging.getLogger()

# Define a list of Python lambda functions that are called by this AWS Lambda
 function.
ACTIONS = {
    "plus": lambda x, y: x + y,
    "minus": lambda x, y: x - y,
    "times": lambda x, y: x * y,
    "divided-by": lambda x, y: x / y,
}


def lambda_handler(event, context):
    """
    Accepts an action and two numbers, performs the specified action on the
 numbers,
    and returns the result.

    :param event: The event dict that contains the parameters sent when the
 function
                  is invoked.
    :param context: The context in which the function is called.
    :return: The result of the specified action.
    """
    # Set the log level based on a variable configured in the Lambda environment.
    logger.setLevel(os.environ.get("LOG_LEVEL", logging.INFO))
    logger.debug("Event: %s", event)

    action = event.get("action")
```

```
    func = ACTIONS.get(action)
    x = event.get("x")
    y = event.get("y")
    result = None
    try:
        if func is not None and x is not None and y is not None:
            result = func(x, y)
            logger.info("%s %s %s is %s", x, action, y, result)
        else:
            logger.error("I can't calculate %s %s %s.", x, action, y)
    except ZeroDivisionError:
        logger.warning("I can't divide %s by 0!", x)

    response = {"result": result}
    return response
```

- 如需 API 詳細資訊，請參閱《AWS SDK for Go API 參考》中的下列主題。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunction代碼
  - UpdateFunction配置

Java

適用於 Java 2.x 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
/*
 *  Lambda function names appear as:
 *
 *  arn:aws:lambda:us-west-2:335556666777:function:HelloFunction
 *
 *  To find this value, look at the function in the AWS Management Console.
 *
 *  Before running this Java code example, set up your development environment,
 including your credentials.
 *
 *  For more information, see this documentation topic:
 *
 *  https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 *
 *  This example performs the following tasks:
 *
 * 1. Creates an AWS Lambda function.
 * 2. Gets a specific AWS Lambda function.
 * 3. Lists all Lambda functions.
 * 4. Invokes a Lambda function.
 * 5. Updates the Lambda function code and invokes it again.
 * 6. Updates a Lambda function's configuration value.
 * 7. Deletes a Lambda function.
 */

public class LambdaScenario {
    public static final String DASHES = new String(new char[80]).replace("\0",
 "-");

    public static void main(String[] args) throws InterruptedException {
        final String usage = """

                Usage:
                    <functionName> <filePath> <role> <handler> <bucketName> <key>
\s

                Where:
                    functionName - The name of the Lambda function.\s
                    filePath - The path to the .zip or .jar where the code is
 located.\s
                    role - The AWS Identity and Access Management (IAM) service
 role that has Lambda permissions.\s
```

```
                    handler - The fully qualified method name (for example,
example.Handler::handleRequest).\s
                    bucketName - The Amazon Simple Storage Service (Amazon S3)
bucket name that contains the .zip or .jar used to update the Lambda function's
code.\s
                    key - The Amazon S3 key name that represents the .zip or .jar
(for example, LambdaHello-1.0-SNAPSHOT.jar).
                    """;

       if (args.length != 6) {
           System.out.println(usage);
           System.exit(1);
       }

       String functionName = args[0];
       String filePath = args[1];
       String role = args[2];
       String handler = args[3];
       String bucketName = args[4];
       String key = args[5];

       Region region = Region.US_WEST_2;
       LambdaClient awsLambda = LambdaClient.builder()
               .region(region)
               .build();

       System.out.println(DASHES);
       System.out.println("Welcome to the AWS Lambda example scenario.");
       System.out.println(DASHES);

       System.out.println(DASHES);
       System.out.println("1. Create an AWS Lambda function.");
       String funArn = createLambdaFunction(awsLambda, functionName, filePath,
role, handler);
       System.out.println("The AWS Lambda ARN is " + funArn);
       System.out.println(DASHES);

       System.out.println(DASHES);
       System.out.println("2. Get the " + functionName + " AWS Lambda
function.");
       getFunction(awsLambda, functionName);
       System.out.println(DASHES);

       System.out.println(DASHES);
```

```
        System.out.println("3. List all AWS Lambda functions.");
        listFunctions(awsLambda);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("4. Invoke the Lambda function.");
        System.out.println("*** Sleep for 1 min to get Lambda function ready.");
        Thread.sleep(60000);
        invokeFunction(awsLambda, functionName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("5. Update the Lambda function code and invoke it
again.");
        updateFunctionCode(awsLambda, functionName, bucketName, key);
        System.out.println("*** Sleep for 1 min to get Lambda function ready.");
        Thread.sleep(60000);
        invokeFunction(awsLambda, functionName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("6. Update a Lambda function's configuration value.");
        updateFunctionConfiguration(awsLambda, functionName, handler);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("7. Delete the AWS Lambda function.");
        LambdaScenario.deleteLambdaFunction(awsLambda, functionName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("The AWS Lambda scenario completed successfully");
        System.out.println(DASHES);
        awsLambda.close();
    }

    public static String createLambdaFunction(LambdaClient awsLambda,
            String functionName,
            String filePath,
            String role,
            String handler) {

        try {
            LambdaWaiter waiter = awsLambda.waiter();
```

```
            InputStream is = new FileInputStream(filePath);
            SdkBytes fileToUpload = SdkBytes.fromInputStream(is);

            FunctionCode code = FunctionCode.builder()
                    .zipFile(fileToUpload)
                    .build();

            CreateFunctionRequest functionRequest =
CreateFunctionRequest.builder()
                    .functionName(functionName)
                    .description("Created by the Lambda Java API")
                    .code(code)
                    .handler(handler)
                    .runtime(Runtime.JAVA8)
                    .role(role)
                    .build();

            // Create a Lambda function using a waiter
            CreateFunctionResponse functionResponse =
awsLambda.createFunction(functionRequest);
            GetFunctionRequest getFunctionRequest = GetFunctionRequest.builder()
                    .functionName(functionName)
                    .build();
            WaiterResponse<GetFunctionResponse> waiterResponse =
waiter.waitUntilFunctionExists(getFunctionRequest);
            waiterResponse.matched().response().ifPresent(System.out::println);
            return functionResponse.functionArn();

        } catch (LambdaException | FileNotFoundException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        return "";
    }

    public static void getFunction(LambdaClient awsLambda, String functionName) {
        try {
            GetFunctionRequest functionRequest = GetFunctionRequest.builder()
                    .functionName(functionName)
                    .build();

            GetFunctionResponse response =
awsLambda.getFunction(functionRequest);
```

```
            System.out.println("The runtime of this Lambda function is " +
response.configuration().runtime());

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    public static void listFunctions(LambdaClient awsLambda) {
        try {
            ListFunctionsResponse functionResult = awsLambda.listFunctions();
            List<FunctionConfiguration> list = functionResult.functions();
            for (FunctionConfiguration config : list) {
                System.out.println("The function name is " +
config.functionName());
            }

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    public static void invokeFunction(LambdaClient awsLambda, String
functionName) {

        InvokeResponse res;
        try {
            // Need a SdkBytes instance for the payload.
            JSONObject jsonObj = new JSONObject();
            jsonObj.put("inputValue", "2000");
            String json = jsonObj.toString();
            SdkBytes payload = SdkBytes.fromUtf8String(json);

            InvokeRequest request = InvokeRequest.builder()
                    .functionName(functionName)
                    .payload(payload)
                    .build();

            res = awsLambda.invoke(request);
            String value = res.payload().asUtf8String();
            System.out.println(value);
```

```
        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    public static void updateFunctionCode(LambdaClient awsLambda, String
functionName, String bucketName, String key) {
        try {
            LambdaWaiter waiter = awsLambda.waiter();
            UpdateFunctionCodeRequest functionCodeRequest =
UpdateFunctionCodeRequest.builder()
                    .functionName(functionName)
                    .publish(true)
                    .s3Bucket(bucketName)
                    .s3Key(key)
                    .build();

            UpdateFunctionCodeResponse response =
awsLambda.updateFunctionCode(functionCodeRequest);
            GetFunctionConfigurationRequest getFunctionConfigRequest =
GetFunctionConfigurationRequest.builder()
                    .functionName(functionName)
                    .build();

            WaiterResponse<GetFunctionConfigurationResponse> waiterResponse =
waiter
                    .waitUntilFunctionUpdated(getFunctionConfigRequest);
            waiterResponse.matched().response().ifPresent(System.out::println);
            System.out.println("The last modified value is " +
response.lastModified());

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    public static void updateFunctionConfiguration(LambdaClient awsLambda, String
functionName, String handler) {
        try {
            UpdateFunctionConfigurationRequest configurationRequest =
UpdateFunctionConfigurationRequest.builder()
                    .functionName(functionName)
```

```
                    .handler(handler)
                    .runtime(Runtime.JAVA11)
                    .build();

            awsLambda.updateFunctionConfiguration(configurationRequest);

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    public static void deleteLambdaFunction(LambdaClient awsLambda, String
 functionName) {
        try {
            DeleteFunctionRequest request = DeleteFunctionRequest.builder()
                    .functionName(functionName)
                    .build();

            awsLambda.deleteFunction(request);
            System.out.println("The " + functionName + " function was deleted");

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

- 如需 API 詳細資訊，請參閱《AWS SDK for Java 2.x API 參考》中的下列主題。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunction代碼
  - UpdateFunction配置

JavaScript

適用於 JavaScript (v3) 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

建立可授與 Lambda 權限寫入記錄的 AWS Identity and Access Management (IAM) 角色。

```
    log(`Creating role (${NAME_ROLE_LAMBDA})...`);
    const response = await createRole(NAME_ROLE_LAMBDA);

import { AttachRolePolicyCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
 * @param {string} policyArn
 * @param {string} roleName
 */
export const attachRolePolicy = (policyArn, roleName) => {
  const command = new AttachRolePolicyCommand({
    PolicyArn: policyArn,
    RoleName: roleName,
  });

  return client.send(command);
};
```

建立 Lambda 函數並上傳處理常式程式碼。

```
const createFunction = async (funcName, roleArn) => {
  const client = new LambdaClient({});
  const code = await readFile(`${dirname}../functions/${funcName}.zip`);

  const command = new CreateFunctionCommand({
    Code: { ZipFile: code },
```

```
    FunctionName: funcName,
    Role: roleArn,
    Architectures: [Architecture.arm64],
    Handler: "index.handler", // Required when sending a .zip file
    PackageType: PackageType.Zip, // Required when sending a .zip file
    Runtime: Runtime.nodejs16x, // Required when sending a .zip file
  });

  return client.send(command);
};
```

調用具有單一參數的函數並取得結果。

```
const invoke = async (funcName, payload) => {
  const client = new LambdaClient({});
  const command = new InvokeCommand({
    FunctionName: funcName,
    Payload: JSON.stringify(payload),
    LogType: LogType.Tail,
  });

  const { Payload, LogResult } = await client.send(command);
  const result = Buffer.from(Payload).toString();
  const logs = Buffer.from(LogResult, "base64").toString();
  return { logs, result };
};
```

更新函數程式碼並設定具有環境變數的 Lambda 環境。

```
const updateFunctionCode = async (funcName, newFunc) => {
  const client = new LambdaClient({});
  const code = await readFile(`${dirname}../functions/${newFunc}.zip`);
  const command = new UpdateFunctionCodeCommand({
    ZipFile: code,
    FunctionName: funcName,
    Architectures: [Architecture.arm64],
    Handler: "index.handler", // Required when sending a .zip file
    PackageType: PackageType.Zip, // Required when sending a .zip file
    Runtime: Runtime.nodejs16x, // Required when sending a .zip file
  });
```

```
  return client.send(command);
};

const updateFunctionConfiguration = (funcName) => {
  const client = new LambdaClient({});
  const config = readFileSync(`${dirname}../functions/config.json`).toString();
  const command = new UpdateFunctionConfigurationCommand({
    ...JSON.parse(config),
    FunctionName: funcName,
  });
  return client.send(command);
};
```

列出您帳戶的函數。

```
const listFunctions = () => {
  const client = new LambdaClient({});
  const command = new ListFunctionsCommand({});

  return client.send(command);
};
```

刪除 IAM 角色與 Lambda 函數。

```
import { DeleteRoleCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
 * @param {string} roleName
 */
export const deleteRole = (roleName) => {
  const command = new DeleteRoleCommand({ RoleName: roleName });
  return client.send(command);
};

/**
 * @param {string} funcName
 */
const deleteFunction = (funcName) => {
```

```
  const client = new LambdaClient({});
  const command = new DeleteFunctionCommand({ FunctionName: funcName });
  return client.send(command);
};
```

- 如需 API 詳細資訊，請參閱《AWS SDK for JavaScript API 參考》中的下列主題。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunction代碼
  - UpdateFunction配置

Kotlin

適用於 Kotlin 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
suspend fun main(args: Array<String>) {
    val usage = """
        Usage:
            <functionName> <role> <handler> <bucketName> <updatedBucketName>
 <key>

        Where:
            functionName - The name of the AWS Lambda function.
            role - The AWS Identity and Access Management (IAM) service role that
 has AWS Lambda permissions.
            handler - The fully qualified method name (for example,
 example.Handler::handleRequest).
```

```
            bucketName - The Amazon Simple Storage Service (Amazon S3) bucket
name that contains the ZIP or JAR used for the Lambda function's code.
            updatedBucketName - The Amazon S3 bucket name that contains the .zip
or .jar used to update the Lambda function's code.
            key - The Amazon S3 key name that represents the .zip or .jar file
(for example, LambdaHello-1.0-SNAPSHOT.jar).
            """

   if (args.size != 6) {
       println(usage)
       exitProcess(1)
   }

   val functionName = args[0]
   val role = args[1]
   val handler = args[2]
   val bucketName = args[3]
   val updatedBucketName = args[4]
   val key = args[5]

   println("Creating a Lambda function named $functionName.")
   val funArn = createScFunction(functionName, bucketName, key, handler, role)
   println("The AWS Lambda ARN is $funArn")

   // Get a specific Lambda function.
   println("Getting the $functionName AWS Lambda function.")
   getFunction(functionName)

   // List the Lambda functions.
   println("Listing all AWS Lambda functions.")
   listFunctionsSc()

   // Invoke the Lambda function.
   println("*** Invoke the Lambda function.")
   invokeFunctionSc(functionName)

   // Update the AWS Lambda function code.
   println("*** Update the Lambda function code.")
   updateFunctionCode(functionName, updatedBucketName, key)

   // println("*** Invoke the function again after updating the code.")
   invokeFunctionSc(functionName)

   // Update the AWS Lambda function configuration.
```

```
    println("Update the run time of the function.")
    updateFunctionConfiguration(functionName, handler)

    // Delete the AWS Lambda function.
    println("Delete the AWS Lambda function.")
    delFunction(functionName)
}

suspend fun createScFunction(
    myFunctionName: String,
    s3BucketName: String,
    myS3Key: String,
    myHandler: String,
    myRole: String
): String {
    val functionCode =
        FunctionCode {
            s3Bucket = s3BucketName
            s3Key = myS3Key
        }

    val request =
        CreateFunctionRequest {
            functionName = myFunctionName
            code = functionCode
            description = "Created by the Lambda Kotlin API"
            handler = myHandler
            role = myRole
            runtime = Runtime.Java8
        }

    // Create a Lambda function using a waiter
    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        val functionResponse = awsLambda.createFunction(request)
        awsLambda.waitUntilFunctionActive {
            functionName = myFunctionName
        }
        return functionResponse.functionArn.toString()
    }
}

suspend fun getFunction(functionNameVal: String) {
    val functionRequest =
        GetFunctionRequest {
```

```
            functionName = functionNameVal
        }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        val response = awsLambda.getFunction(functionRequest)
        println("The runtime of this Lambda function is
 ${response.configuration?.runtime}")
    }
}

suspend fun listFunctionsSc() {
    val request =
        ListFunctionsRequest {
            maxItems = 10
        }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        val response = awsLambda.listFunctions(request)
        response.functions?.forEach { function ->
            println("The function name is ${function.functionName}")
        }
    }
}

suspend fun invokeFunctionSc(functionNameVal: String) {
    val json = """{"inputValue":"1000"}"""
    val byteArray = json.trimIndent().encodeToByteArray()
    val request =
        InvokeRequest {
            functionName = functionNameVal
            payload = byteArray
            logType = LogType.Tail
        }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        val res = awsLambda.invoke(request)
        println("The function payload is
 ${res.payload?.toString(Charsets.UTF_8)}")
    }
}

suspend fun updateFunctionCode(
    functionNameVal: String?,
    bucketName: String?,
```

```
    key: String?
) {
    val functionCodeRequest =
        UpdateFunctionCodeRequest {
            functionName = functionNameVal
            publish = true
            s3Bucket = bucketName
            s3Key = key
        }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        val response = awsLambda.updateFunctionCode(functionCodeRequest)
        awsLambda.waitUntilFunctionUpdated {
            functionName = functionNameVal
        }
        println("The last modified value is " + response.lastModified)
    }
}

suspend fun updateFunctionConfiguration(
    functionNameVal: String?,
    handlerVal: String?
) {
    val configurationRequest =
        UpdateFunctionConfigurationRequest {
            functionName = functionNameVal
            handler = handlerVal
            runtime = Runtime.Java11
        }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        awsLambda.updateFunctionConfiguration(configurationRequest)
    }
}

suspend fun delFunction(myFunctionName: String) {
    val request =
        DeleteFunctionRequest {
            functionName = myFunctionName
        }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        awsLambda.deleteFunction(request)
        println("$myFunctionName was deleted")
```

```
    }
}
```

- 如需 API 詳細資訊，請參閱《AWS 適用於 Kotlin 的 SDK API 參考》中的下列主題。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunction代碼
  - UpdateFunction配置

PHP

適用於 PHP 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
namespace Lambda;

use Aws\S3\S3Client;
use GuzzleHttp\Psr7\Stream;
use Iam\IAMService;

class GettingStartedWithLambda
{
    public function run()
    {
        echo("\n");
        echo("---------------------------------------\n");
        print("Welcome to the AWS Lambda getting started demo using PHP!\n");
        echo("---------------------------------------\n");
```

```
        $clientArgs = [
            'region' => 'us-west-2',
            'version' => 'latest',
            'profile' => 'default',
        ];
        $uniqid = uniqid();

        $iamService = new IAMService();
        $s3client = new S3Client($clientArgs);
        $lambdaService = new LambdaService();

        echo "First, let's create a role to run our Lambda code.\n";
        $roleName = "test-lambda-role-$uniqid";
        $rolePolicyDocument = "{
            \"Version\": \"2012-10-17\",
            \"Statement\": [
                {
                    \"Effect\": \"Allow\",
                    \"Principal\": {
                        \"Service\": \"lambda.amazonaws.com\"
                    },
                    \"Action\": \"sts:AssumeRole\"
                }
            ]
        }";
        $role = $iamService->createRole($roleName, $rolePolicyDocument);
        echo "Created role {$role['RoleName']}.\n";

        $iamService->attachRolePolicy(
            $role['RoleName'],
            "arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole"
        );
        echo "Attached the AWSLambdaBasicExecutionRole to {$role['RoleName']}.
\n";

        echo "\nNow let's create an S3 bucket and upload our Lambda code there.
\n";
        $bucketName = "test-example-bucket-$uniqid";
        $s3client->createBucket([
            'Bucket' => $bucketName,
        ]);
        echo "Created bucket $bucketName.\n";

        $functionName = "doc_example_lambda_$uniqid";
```

```
        $codeBasic = __DIR__ . "/lambda_handler_basic.zip";
        $handler = "lambda_handler_basic";
        $file = file_get_contents($codeBasic);
        $s3client->putObject([
            'Bucket' => $bucketName,
            'Key' => $functionName,
            'Body' => $file,
        ]);
        echo "Uploaded the Lambda code.\n";

        $createLambdaFunction = $lambdaService->createFunction($functionName,
 $role, $bucketName, $handler);
        // Wait until the function has finished being created.
        do {
            $getLambdaFunction = $lambdaService-
>getFunction($createLambdaFunction['FunctionName']);
        } while ($getLambdaFunction['Configuration']['State'] == "Pending");
        echo "Created Lambda function {$getLambdaFunction['Configuration']
['FunctionName']}.\n";

        sleep(1);

        echo "\nOk, let's invoke that Lambda code.\n";
        $basicParams = [
            'action' => 'increment',
            'number' => 3,
        ];
        /** @var Stream $invokeFunction */
        $invokeFunction = $lambdaService->invoke($functionName, $basicParams)
['Payload'];
        $result = json_decode($invokeFunction->getContents())->result;
        echo "After invoking the Lambda code with the input of
 {$basicParams['number']} we received $result.\n";

        echo "\nSince that's working, let's update the Lambda code.\n";
        $codeCalculator = "lambda_handler_calculator.zip";
        $handlerCalculator = "lambda_handler_calculator";
        echo "First, put the new code into the S3 bucket.\n";
        $file = file_get_contents($codeCalculator);
        $s3client->putObject([
            'Bucket' => $bucketName,
            'Key' => $functionName,
            'Body' => $file,
        ]);
```

```
        echo "New code uploaded.\n";

        $lambdaService->updateFunctionCode($functionName, $bucketName,
 $functionName);
        // Wait for the Lambda code to finish updating.
        do {
            $getLambdaFunction = $lambdaService-
>getFunction($createLambdaFunction['FunctionName']);
        } while ($getLambdaFunction['Configuration']['LastUpdateStatus'] !==
 "Successful");
        echo "New Lambda code uploaded.\n";

        $environment = [
            'Variable' => ['Variables' => ['LOG_LEVEL' => 'DEBUG']],
        ];
        $lambdaService->updateFunctionConfiguration($functionName,
 $handlerCalculator, $environment);
        do {
            $getLambdaFunction = $lambdaService-
>getFunction($createLambdaFunction['FunctionName']);
        } while ($getLambdaFunction['Configuration']['LastUpdateStatus'] !==
 "Successful");
        echo "Lambda code updated with new handler and a LOG_LEVEL of DEBUG for
 more information.\n";

        echo "Invoke the new code with some new data.\n";
        $calculatorParams = [
            'action' => 'plus',
            'x' => 5,
            'y' => 4,
        ];
        $invokeFunction = $lambdaService->invoke($functionName,
 $calculatorParams, "Tail");
        $result = json_decode($invokeFunction['Payload']->getContents())->result;
        echo "Indeed, {$calculatorParams['x']} + {$calculatorParams['y']} does
 equal $result.\n";
        echo "Here's the extra debug info: ";
        echo base64_decode($invokeFunction['LogResult']) . "\n";

        echo "\nBut what happens if you try to divide by zero?\n";
        $divZeroParams = [
            'action' => 'divide',
            'x' => 5,
            'y' => 0,
```

```
        ];
        $invokeFunction = $lambdaService->invoke($functionName, $divZeroParams,
 "Tail");
        $result = json_decode($invokeFunction['Payload']->getContents())->result;
        echo "You get a |$result| result.\n";
        echo "And an error message: ";
        echo base64_decode($invokeFunction['LogResult']) . "\n";

        echo "\nHere's all the Lambda functions you have in this Region:\n";
        $listLambdaFunctions = $lambdaService->listFunctions(5);
        $allLambdaFunctions = $listLambdaFunctions['Functions'];
        $next = $listLambdaFunctions->get('NextMarker');
        while ($next != false) {
            $listLambdaFunctions = $lambdaService->listFunctions(5, $next);
            $next = $listLambdaFunctions->get('NextMarker');
            $allLambdaFunctions = array_merge($allLambdaFunctions,
 $listLambdaFunctions['Functions']);
        }
        foreach ($allLambdaFunctions as $function) {
            echo "{$function['FunctionName']}\n";
        }

        echo "\n\nAnd don't forget to clean up your data!\n";

        $lambdaService->deleteFunction($functionName);
        echo "Deleted Lambda function.\n";
        $iamService->deleteRole($role['RoleName']);
        echo "Deleted Role.\n";
        $deleteObjects = $s3client->listObjectsV2([
            'Bucket' => $bucketName,
        ]);
        $deleteObjects = $s3client->deleteObjects([
            'Bucket' => $bucketName,
            'Delete' => [
                'Objects' => $deleteObjects['Contents'],
            ]
        ]);
        echo "Deleted all objects from the S3 bucket.\n";
        $s3client->deleteBucket(['Bucket' => $bucketName]);
        echo "Deleted the bucket.\n";
    }
}
```

- 如需 API 詳細資訊，請參閱《AWS SDK for PHP API 參考》中的下列主題。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunction代碼
  - UpdateFunction配置

Python

適用於 Python (Boto3) 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

定義增量一個數字的 Lambda 處理常式。

```
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)


def lambda_handler(event, context):
    """
    Accepts an action and a single number, performs the specified action on the
 number,
    and returns the result. The only allowable action is 'increment'.

    :param event: The event dict that contains the parameters sent when the
 function
                  is invoked.
    :param context: The context in which the function is called.
    :return: The result of the action.
    """
```

```
    result = None
    action = event.get("action")
    if action == "increment":
        result = event.get("number", 0) + 1
        logger.info("Calculated result of %s", result)
    else:
        logger.error("%s is not a valid action.", action)

    response = {"result": result}
    return response
```

定義可執行算術運算的第二個 Lambda 處理常式。

```
import logging
import os


logger = logging.getLogger()

# Define a list of Python lambda functions that are called by this AWS Lambda
 function.
ACTIONS = {
    "plus": lambda x, y: x + y,
    "minus": lambda x, y: x - y,
    "times": lambda x, y: x * y,
    "divided-by": lambda x, y: x / y,
}


def lambda_handler(event, context):
    """
    Accepts an action and two numbers, performs the specified action on the
 numbers,
    and returns the result.

    :param event: The event dict that contains the parameters sent when the
 function
                  is invoked.
    :param context: The context in which the function is called.
    :return: The result of the specified action.
```

```
    """
    # Set the log level based on a variable configured in the Lambda environment.
    logger.setLevel(os.environ.get("LOG_LEVEL", logging.INFO))
    logger.debug("Event: %s", event)

    action = event.get("action")
    func = ACTIONS.get(action)
    x = event.get("x")
    y = event.get("y")
    result = None
    try:
        if func is not None and x is not None and y is not None:
            result = func(x, y)
            logger.info("%s %s %s is %s", x, action, y, result)
        else:
            logger.error("I can't calculate %s %s %s.", x, action, y)
    except ZeroDivisionError:
        logger.warning("I can't divide %s by 0!", x)

    response = {"result": result}
    return response
```

建立可包裝 Lambda 動作的函數。

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    @staticmethod
    def create_deployment_package(source_file, destination_file):
        """
        Creates a Lambda deployment package in .zip format in an in-memory
 buffer. This
        buffer can be passed directly to Lambda when creating the function.

        :param source_file: The name of the file that contains the Lambda handler
                            function.
```

```
        :param destination_file: The name to give the file when it's deployed to
Lambda.
        :return: The deployment package.
        """
        buffer = io.BytesIO()
        with zipfile.ZipFile(buffer, "w") as zipped:
            zipped.write(source_file, destination_file)
        buffer.seek(0)
        return buffer.read()

    def get_iam_role(self, iam_role_name):
        """
        Get an AWS Identity and Access Management (IAM) role.

        :param iam_role_name: The name of the role to retrieve.
        :return: The IAM role.
        """
        role = None
        try:
            temp_role = self.iam_resource.Role(iam_role_name)
            temp_role.load()
            role = temp_role
            logger.info("Got IAM role %s", role.name)
        except ClientError as err:
            if err.response["Error"]["Code"] == "NoSuchEntity":
                logger.info("IAM role %s does not exist.", iam_role_name)
            else:
                logger.error(
                    "Couldn't get IAM role %s. Here's why: %s: %s",
                    iam_role_name,
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
                raise
        return role

    def create_iam_role_for_lambda(self, iam_role_name):
        """
        Creates an IAM role that grants the Lambda function basic permissions. If
a
        role with the specified name already exists, it is used for the demo.

        :param iam_role_name: The name of the role to create.
```

```
        :return: The role and a value that indicates whether the role is newly
 created.
        """
        role = self.get_iam_role(iam_role_name)
        if role is not None:
            return role, False

        lambda_assume_role_policy = {
            "Version": "2012-10-17",
            "Statement": [
                {
                    "Effect": "Allow",
                    "Principal": {"Service": "lambda.amazonaws.com"},
                    "Action": "sts:AssumeRole",
                }
            ],
        }
        policy_arn = "arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole"

        try:
            role = self.iam_resource.create_role(
                RoleName=iam_role_name,
                AssumeRolePolicyDocument=json.dumps(lambda_assume_role_policy),
            )
            logger.info("Created role %s.", role.name)
            role.attach_policy(PolicyArn=policy_arn)
            logger.info("Attached basic execution policy to role %s.", role.name)
        except ClientError as error:
            if error.response["Error"]["Code"] == "EntityAlreadyExists":
                role = self.iam_resource.Role(iam_role_name)
                logger.warning("The role %s already exists. Using it.",
 iam_role_name)
            else:
                logger.exception(
                    "Couldn't create role %s or attach policy %s.",
                    iam_role_name,
                    policy_arn,
                )
                raise

        return role, True

    def get_function(self, function_name):
```

```
        """
        Gets data about a Lambda function.

        :param function_name: The name of the function.
        :return: The function data.
        """
        response = None
        try:
            response =
self.lambda_client.get_function(FunctionName=function_name)
        except ClientError as err:
            if err.response["Error"]["Code"] == "ResourceNotFoundException":
                logger.info("Function %s does not exist.", function_name)
            else:
                logger.error(
                    "Couldn't get function %s. Here's why: %s: %s",
                    function_name,
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
                raise
        return response


    def create_function(
        self, function_name, handler_name, iam_role, deployment_package
    ):
        """
        Deploys a Lambda function.

        :param function_name: The name of the Lambda function.
        :param handler_name: The fully qualified name of the handler function.
This
                             must include the file name and the function name.
        :param iam_role: The IAM role to use for the function.
        :param deployment_package: The deployment package that contains the
function
                                   code in .zip format.
        :return: The Amazon Resource Name (ARN) of the newly created function.
        """
        try:
            response = self.lambda_client.create_function(
                FunctionName=function_name,
                Description="AWS Lambda doc example",
```

```
                Runtime="python3.8",
                Role=iam_role.arn,
                Handler=handler_name,
                Code={"ZipFile": deployment_package},
                Publish=True,
            )
            function_arn = response["FunctionArn"]
            waiter = self.lambda_client.get_waiter("function_active_v2")
            waiter.wait(FunctionName=function_name)
            logger.info(
                "Created function '%s' with ARN: '%s'.",
                function_name,
                response["FunctionArn"],
            )
        except ClientError:
            logger.error("Couldn't create function %s.", function_name)
            raise
        else:
            return function_arn


    def delete_function(self, function_name):
        """
        Deletes a Lambda function.

        :param function_name: The name of the function to delete.
        """
        try:
            self.lambda_client.delete_function(FunctionName=function_name)
        except ClientError:
            logger.exception("Couldn't delete function %s.", function_name)
            raise


    def invoke_function(self, function_name, function_params, get_log=False):
        """
        Invokes a Lambda function.

        :param function_name: The name of the function to invoke.
        :param function_params: The parameters of the function as a dict. This
dict
                                is serialized to JSON before it is sent to
Lambda.
```

```
        :param get_log: When true, the last 4 KB of the execution log are
included in
                            the response.
        :return: The response from the function invocation.
        """
        try:
            response = self.lambda_client.invoke(
                FunctionName=function_name,
                Payload=json.dumps(function_params),
                LogType="Tail" if get_log else "None",
            )
            logger.info("Invoked function %s.", function_name)
        except ClientError:
            logger.exception("Couldn't invoke function %s.", function_name)
            raise
        return response


    def update_function_code(self, function_name, deployment_package):
        """
        Updates the code for a Lambda function by submitting a .zip archive that
contains
        the code for the function.

        :param function_name: The name of the function to update.
        :param deployment_package: The function code to update, packaged as bytes
in
                                   .zip format.
        :return: Data about the update, including the status.
        """
        try:
            response = self.lambda_client.update_function_code(
                FunctionName=function_name, ZipFile=deployment_package
            )
        except ClientError as err:
            logger.error(
                "Couldn't update function %s. Here's why: %s: %s",
                function_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response
```

```
    def update_function_configuration(self, function_name, env_vars):
        """
        Updates the environment variables for a Lambda function.

        :param function_name: The name of the function to update.
        :param env_vars: A dict of environment variables to update.
        :return: Data about the update, including the status.
        """
        try:
            response = self.lambda_client.update_function_configuration(
                FunctionName=function_name, Environment={"Variables": env_vars}
            )
        except ClientError as err:
            logger.error(
                "Couldn't update function configuration %s. Here's why: %s: %s",
                function_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response


    def list_functions(self):
        """
        Lists the Lambda functions for the current account.
        """
        try:
            func_paginator = self.lambda_client.get_paginator("list_functions")
            for func_page in func_paginator.paginate():
                for func in func_page["Functions"]:
                    print(func["FunctionName"])
                    desc = func.get("Description")
                    if desc:
                        print(f"\t{desc}")
                    print(f"\t{func['Runtime']}: {func['Handler']}")
        except ClientError as err:
            logger.error(
                "Couldn't list functions. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
```

```
            )
            raise
```

建立可執行該案例的函數。

```
class UpdateFunctionWaiter(CustomWaiter):
    """A custom waiter that waits until a function is successfully updated."""

    def __init__(self, client):
        super().__init__(
            "UpdateSuccess",
            "GetFunction",
            "Configuration.LastUpdateStatus",
            {"Successful": WaitState.SUCCESS, "Failed": WaitState.FAILURE},
            client,
        )

    def wait(self, function_name):
        self._wait(FunctionName=function_name)


def run_scenario(lambda_client, iam_resource, basic_file, calculator_file,
 lambda_name):
    """
    Runs the scenario.

    :param lambda_client: A Boto3 Lambda client.
    :param iam_resource: A Boto3 IAM resource.
    :param basic_file: The name of the file that contains the basic Lambda
 handler.
    :param calculator_file: The name of the file that contains the calculator
 Lambda handler.
    :param lambda_name: The name to give resources created for the scenario, such
 as the
                        IAM role and the Lambda function.
    """
    logging.basicConfig(level=logging.INFO, format="%(levelname)s: %(message)s")

    print("-" * 88)
```

```
    print("Welcome to the AWS Lambda getting started with functions demo.")
    print("-" * 88)

    wrapper = LambdaWrapper(lambda_client, iam_resource)

    print("Checking for IAM role for Lambda...")
    iam_role, should_wait = wrapper.create_iam_role_for_lambda(lambda_name)
    if should_wait:
        logger.info("Giving AWS time to create resources...")
        wait(10)

    print(f"Looking for function {lambda_name}...")
    function = wrapper.get_function(lambda_name)
    if function is None:
        print("Zipping the Python script into a deployment package...")
        deployment_package = wrapper.create_deployment_package(
            basic_file, f"{lambda_name}.py"
        )
        print(f"...and creating the {lambda_name} Lambda function.")
        wrapper.create_function(
            lambda_name, f"{lambda_name}.lambda_handler", iam_role,
deployment_package
        )
    else:
        print(f"Function {lambda_name} already exists.")
    print("-" * 88)

    print(f"Let's invoke {lambda_name}. This function increments a number.")
    action_params = {
        "action": "increment",
        "number": q.ask("Give me a number to increment: ", q.is_int),
    }
    print(f"Invoking {lambda_name}...")
    response = wrapper.invoke_function(lambda_name, action_params)
    print(
        f"Incrementing {action_params['number']} resulted in "
        f"{json.load(response['Payload'])}"
    )
    print("-" * 88)

    print(f"Let's update the function to an arithmetic calculator.")
    q.ask("Press Enter when you're ready.")
    print("Creating a new deployment package...")
    deployment_package = wrapper.create_deployment_package(
```

```
        calculator_file, f"{lambda_name}.py"
    )
    print(f"...and updating the {lambda_name} Lambda function.")
    update_waiter = UpdateFunctionWaiter(lambda_client)
    wrapper.update_function_code(lambda_name, deployment_package)
    update_waiter.wait(lambda_name)
    print(f"This function uses an environment variable to control logging
level.")
    print(f"Let's set it to DEBUG to get the most logging.")
    wrapper.update_function_configuration(
        lambda_name, {"LOG_LEVEL": logging.getLevelName(logging.DEBUG)}
    )

    actions = ["plus", "minus", "times", "divided-by"]
    want_invoke = True
    while want_invoke:
        print(f"Let's invoke {lambda_name}. You can invoke these actions:")
        for index, action in enumerate(actions):
            print(f"{index + 1}: {action}")
        action_params = {}
        action_index = q.ask(
            "Enter the number of the action you want to take: ",
            q.is_int,
            q.in_range(1, len(actions)),
        )
        action_params["action"] = actions[action_index - 1]
        print(f"You've chosen to invoke 'x {action_params['action']} y'.")
        action_params["x"] = q.ask("Enter a value for x: ", q.is_int)
        action_params["y"] = q.ask("Enter a value for y: ", q.is_int)
        print(f"Invoking {lambda_name}...")
        response = wrapper.invoke_function(lambda_name, action_params, True)
        print(
            f"Calculating {action_params['x']} {action_params['action']}
{action_params['y']} "
            f"resulted in {json.load(response['Payload'])}"
        )
        q.ask("Press Enter to see the logs from the call.")
        print(base64.b64decode(response["LogResult"]).decode())
        want_invoke = q.ask("That was fun. Shall we do it again? (y/n) ",
q.is_yesno)
    print("-" * 88)

    if q.ask(
```

```
        "Do you want to list all of the functions in your account? (y/n) ",
 q.is_yesno
    ):
        wrapper.list_functions()
    print("-" * 88)

    if q.ask("Ready to delete the function and role? (y/n) ", q.is_yesno):
        for policy in iam_role.attached_policies.all():
            policy.detach_role(RoleName=iam_role.name)
        iam_role.delete()
        print(f"Deleted role {lambda_name}.")
        wrapper.delete_function(lambda_name)
        print(f"Deleted function {lambda_name}.")

    print("\nThanks for watching!")
    print("-" * 88)


if __name__ == "__main__":
    try:
        run_scenario(
            boto3.client("lambda"),
            boto3.resource("iam"),
            "lambda_handler_basic.py",
            "lambda_handler_calculator.py",
            "doc_example_lambda_calculator",
        )
    except Exception:
        logging.exception("Something went wrong with the demo!")
```

- 如需 API 詳細資訊，請參閱下列《適用於 Python (Boto3) 的AWS 開發套件 API 參考》中的下列主題。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunction代碼
  - UpdateFunction配置

Ruby

適用於 Ruby 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

為能夠寫入日誌的 Lambda 函數設定先決條件 IAM 許可。

```
  # Get an AWS Identity and Access Management (IAM) role.
  #
  # @param iam_role_name: The name of the role to retrieve.
  # @param action: Whether to create or destroy the IAM apparatus.
  # @return: The IAM role.
  def manage_iam(iam_role_name, action)
    role_policy = {
      'Version': "2012-10-17",
      'Statement': [
        {
          'Effect': "Allow",
          'Principal': {
            'Service': "lambda.amazonaws.com"
          },
          'Action': "sts:AssumeRole"
        }
      ]
    }
    case action
    when "create"
      role = $iam_client.create_role(
        role_name: iam_role_name,
        assume_role_policy_document: role_policy.to_json
      )
      $iam_client.attach_role_policy(
        {
          policy_arn: "arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole",
          role_name: iam_role_name
        }
      )
```

```
      $iam_client.wait_until(:role_exists, { role_name: iam_role_name }) do |w|
        w.max_attempts = 5
        w.delay = 5
      end
      @logger.debug("Successfully created IAM role: #{role['role']['arn']}")
      @logger.debug("Enforcing a 10-second sleep to allow IAM role to activate
 fully.")
      sleep(10)
      return role, role_policy.to_json
    when "destroy"
      $iam_client.detach_role_policy(
        {
          policy_arn: "arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole",
          role_name: iam_role_name
        }
      )
      $iam_client.delete_role(
        role_name: iam_role_name
      )
      @logger.debug("Detached policy & deleted IAM role: #{iam_role_name}")
    else
      raise "Incorrect action provided. Must provide 'create' or 'destroy'"
    end
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error creating role or attaching policy:\n
 #{e.message}")
  end
```

定義 Lambda 處理常式，該處理常式以作為調用參數提供的數字遞增。

```
require "logger"

# A function that increments a whole number by one (1) and logs the result.
# Requires a manually-provided runtime parameter, 'number', which must be Int
#
# @param event [Hash] Parameters sent when the function is invoked
# @param context [Hash] Methods and properties that provide information
# about the invocation, function, and execution environment.
# @return incremented_number [String] The incremented number.
def lambda_handler(event:, context:)
  logger = Logger.new($stdout)
```

```
  log_level = ENV["LOG_LEVEL"]
  logger.level = case log_level
                 when "debug"
                   Logger::DEBUG
                 when "info"
                   Logger::INFO
                 else
                   Logger::ERROR
                 end
  logger.debug("This is a debug log message.")
  logger.info("This is an info log message. Code executed successfully!")
  number = event["number"].to_i
  incremented_number = number + 1
  logger.info("You provided #{number.round} and it was incremented to
 #{incremented_number.round}")
  incremented_number.round.to_s
end
```

將您的 Lambda 函數壓縮到部署套件中。

```
  # Creates a Lambda deployment package in .zip format.
  # This zip can be passed directly as a string to Lambda when creating the
 function.
  #
  # @param source_file: The name of the object, without suffix, for the Lambda
 file and zip.
  # @return: The deployment package.
  def create_deployment_package(source_file)
    Dir.chdir(File.dirname(__FILE__))
    if File.exist?("lambda_function.zip")
      File.delete("lambda_function.zip")
      @logger.debug("Deleting old zip: lambda_function.zip")
    end
    Zip::File.open("lambda_function.zip", create: true) {
      |zipfile|
      zipfile.add("lambda_function.rb", "#{source_file}.rb")
    }
    @logger.debug("Zipping #{source_file}.rb into: lambda_function.zip.")
    File.read("lambda_function.zip").to_s
  rescue StandardError => e
    @logger.error("There was an error creating deployment package:\n
 #{e.message}")
```

```
  end
```

建立新 Lambda 函數。

```
 # Deploys a Lambda function.
 #
 # @param function_name: The name of the Lambda function.
 # @param handler_name: The fully qualified name of the handler function. This
 #                      must include the file name and the function name.
 # @param role_arn: The IAM role to use for the function.
 # @param deployment_package: The deployment package that contains the function
 #                            code in .zip format.
 # @return: The Amazon Resource Name (ARN) of the newly created function.
 def create_function(function_name, handler_name, role_arn, deployment_package)
   response = @lambda_client.create_function({
                                               role: role_arn.to_s,
                                               function_name: function_name,
                                               handler: handler_name,
                                               runtime: "ruby2.7",
                                               code: {
                                                 zip_file: deployment_package
                                               },
                                               environment: {
                                                 variables: {
                                                   "LOG_LEVEL" => "info"
                                                 }
                                               }
                                             })
   @lambda_client.wait_until(:function_active_v2, { function_name:
function_name}) do |w|
     w.max_attempts = 5
     w.delay = 5
   end
   response
 rescue Aws::Lambda::Errors::ServiceException => e
   @logger.error("There was an error creating #{function_name}:\n #{e.message}")
 rescue Aws::Waiters::Errors::WaiterFailed => e
   @logger.error("Failed waiting for #{function_name} to activate:\n
#{e.message}")
 end
```

使用選用的執行階段參數調用 Lambda 函數。

```
 # Invokes a Lambda function.
 # @param function_name [String] The name of the function to invoke.
 # @param payload [nil] Payload containing runtime parameters.
 # @return [Object] The response from the function invocation.
 def invoke_function(function_name, payload = nil)
   params = { function_name: function_name}
   params[:payload] = payload unless payload.nil?
   @lambda_client.invoke(params)
 rescue Aws::Lambda::Errors::ServiceException => e
   @logger.error("There was an error executing #{function_name}:\n
#{e.message}")
 end
```

更新 Lambda 函數的組態以注入新的環境變數。

```
 # Updates the environment variables for a Lambda function.
 # @param function_name: The name of the function to update.
 # @param log_level: The log level of the function.
 # @return: Data about the update, including the status.
 def update_function_configuration(function_name, log_level)
   @lambda_client.update_function_configuration({
                                                  function_name: function_name,
                                                  environment: {
                                                    variables: {
                                                      "LOG_LEVEL" => log_level
                                                    }
                                                  }
                                                })
   @lambda_client.wait_until(:function_updated_v2, { function_name:
function_name}) do |w|
     w.max_attempts = 5
     w.delay = 5
   end
 rescue Aws::Lambda::Errors::ServiceException => e
   @logger.error("There was an error updating configurations for
#{function_name}:\n #{e.message}")
 rescue Aws::Waiters::Errors::WaiterFailed => e
   @logger.error("Failed waiting for #{function_name} to activate:\n
#{e.message}")
 end
```

使用包含不同程式碼的不同部署套件來更新 Lambda 函數的程式碼。

```
 # Updates the code for a Lambda function by submitting a .zip archive that
contains
 # the code for the function.

 # @param function_name: The name of the function to update.
 # @param deployment_package: The function code to update, packaged as bytes in
 #                            .zip format.
 # @return: Data about the update, including the status.
 def update_function_code(function_name, deployment_package)
   @lambda_client.update_function_code(
     function_name: function_name,
     zip_file: deployment_package
   )
   @lambda_client.wait_until(:function_updated_v2, { function_name:
function_name}) do |w|
     w.max_attempts = 5
     w.delay = 5
   end
 rescue Aws::Lambda::Errors::ServiceException => e
   @logger.error("There was an error updating function code for:
#{function_name}:\n #{e.message}")
   nil
 rescue Aws::Waiters::Errors::WaiterFailed => e
   @logger.error("Failed waiting for #{function_name} to update:\n
#{e.message}")
 end
```

使用內建分頁程式列出所有現有的 Lambda 函數。

```
 # Lists the Lambda functions for the current account.
 def list_functions
   functions = []
   @lambda_client.list_functions.each do |response|
     response["functions"].each do |function|
       functions.append(function["function_name"])
     end
   end
   functions
```

```
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error executing #{function_name}:\n
#{e.message}")
  end
```

刪除特定 Lambda 函數。

```
  # Deletes a Lambda function.
  # @param function_name: The name of the function to delete.
  def delete_function(function_name)
    print "Deleting function: #{function_name}..."
    @lambda_client.delete_function(
      function_name: function_name
    )
    print "Done!".green
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error deleting #{function_name}:\n #{e.message}")
  end
```

- 如需 API 詳細資訊，請參閱《AWS SDK for Ruby API 參考》中的下列主題。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunction代碼
  - UpdateFunction配置

Rust

適用於 Rust 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

在此案例中使用具有相依項的 Cargo.toml。

```
[package]
name = "lambda-code-examples"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/
reference/manifest.html

[dependencies]
aws-config = { version = "1.0.1", features = ["behavior-version-latest"] }
aws-sdk-ec2 = { version = "1.3.0" }
aws-sdk-iam = { version = "1.3.0" }
aws-sdk-lambda = { version = "1.3.0" }
aws-sdk-s3 = { version = "1.4.0" }
aws-smithy-types = { version = "1.0.1" }
aws-types = { version = "1.0.1" }
clap = { version = "~4.4", features = ["derive"] }
tokio = { version = "1.20.1", features = ["full"] }
tracing-subscriber = { version = "0.3.15", features = ["env-filter"] }
tracing = "0.1.37"
serde_json = "1.0.94"
anyhow = "1.0.71"
uuid = { version = "1.3.3", features = ["v4"] }
lambda_runtime = "0.8.0"
serde = "1.0.164"
```

一個公用程式集合，可簡化此案例的 Lambda 呼叫。此檔案是套件中的 src/ations.rs。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

use anyhow::anyhow;
use aws_sdk_iam::operation::{create_role::CreateRoleError,
 delete_role::DeleteRoleOutput};
use aws_sdk_lambda::{
    operation::{
        delete_function::DeleteFunctionOutput, get_function::GetFunctionOutput,
        invoke::InvokeOutput, list_functions::ListFunctionsOutput,
        update_function_code::UpdateFunctionCodeOutput,
        update_function_configuration::UpdateFunctionConfigurationOutput,
```

```
    },
    primitives::ByteStream,
    types::{Environment, FunctionCode, LastUpdateStatus, State},
};
use aws_sdk_s3::{
    error::ErrorMetadata,
    operation::{delete_bucket::DeleteBucketOutput,
 delete_object::DeleteObjectOutput},
    types::CreateBucketConfiguration,
};
use aws_smithy_types::Blob;
use serde::{ser::SerializeMap, Serialize};
use std::{path::PathBuf, str::FromStr, time::Duration};
use tracing::{debug, info, warn};

/* Operation describes  */
#[derive(Clone, Copy, Debug, Serialize)]
pub enum Operation {
    #[serde(rename = "plus")]
    Plus,
    #[serde(rename = "minus")]
    Minus,
    #[serde(rename = "times")]
    Times,
    #[serde(rename = "divided-by")]
    DividedBy,
}

impl FromStr for Operation {
    type Err = anyhow::Error;

    fn from_str(s: &str) -> Result<Self, Self::Err> {
        match s {
            "plus" => Ok(Operation::Plus),
            "minus" => Ok(Operation::Minus),
            "times" => Ok(Operation::Times),
            "divided-by" => Ok(Operation::DividedBy),
            _ => Err(anyhow!("Unknown operation {s}")),
        }
    }
}

impl ToString for Operation {
    fn to_string(&self) -> String {
```

```
        match self {
            Operation::Plus => "plus".to_string(),
            Operation::Minus => "minus".to_string(),
            Operation::Times => "times".to_string(),
            Operation::DividedBy => "divided-by".to_string(),
        }
    }
}

/**
 * InvokeArgs will be serialized as JSON and sent to the AWS Lambda handler.
 */
#[derive(Debug)]
pub enum InvokeArgs {
    Increment(i32),
    Arithmetic(Operation, i32, i32),
}

impl Serialize for InvokeArgs {
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
    where
        S: serde::Serializer,
    {
        match self {
            InvokeArgs::Increment(i) => serializer.serialize_i32(*i),
            InvokeArgs::Arithmetic(o, i, j) => {
                let mut map: S::SerializeMap =
 serializer.serialize_map(Some(3))?;
                map.serialize_key(&"op".to_string())?;
                map.serialize_value(&o.to_string())?;
                map.serialize_key(&"i".to_string())?;
                map.serialize_value(&i)?;
                map.serialize_key(&"j".to_string())?;
                map.serialize_value(&j)?;
                map.end()
            }
        }
    }
}

/** A policy document allowing Lambda to execute this function on the account's
 behalf. */
const ROLE_POLICY_DOCUMENT: &str = r#"{
    "Version": "2012-10-17",
```

```
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": { "Service": "lambda.amazonaws.com" },
            "Action": "sts:AssumeRole"
        }
    ]
}"#;

/**
 * A LambdaManager gathers all the resources necessary to run the Lambda example
 scenario.
 * This includes instantiated aws_sdk clients and details of resource names.
 */
pub struct LambdaManager {
    iam_client: aws_sdk_iam::Client,
    lambda_client: aws_sdk_lambda::Client,
    s3_client: aws_sdk_s3::Client,
    lambda_name: String,
    role_name: String,
    bucket: String,
    own_bucket: bool,
}

// These unit type structs provide nominal typing on top of String parameters for
 LambdaManager::new
pub struct LambdaName(pub String);
pub struct RoleName(pub String);
pub struct Bucket(pub String);
pub struct OwnBucket(pub bool);

impl LambdaManager {
    pub fn new(
        iam_client: aws_sdk_iam::Client,
        lambda_client: aws_sdk_lambda::Client,
        s3_client: aws_sdk_s3::Client,
        lambda_name: LambdaName,
        role_name: RoleName,
        bucket: Bucket,
        own_bucket: OwnBucket,
    ) -> Self {
        Self {
            iam_client,
            lambda_client,
```

```
            s3_client,
            lambda_name: lambda_name.0,
            role_name: role_name.0,
            bucket: bucket.0,
            own_bucket: own_bucket.0,
        }
    }

    /**
     * Load the AWS configuration from the environment.
     * Look up lambda_name and bucket if none are given, or generate a random
name if not present in the environment.
     * If the bucket name is provided, the caller needs to have created the
bucket.
     * If the bucket name is generated, it will be created.
     */
    pub async fn load_from_env(lambda_name: Option<String>, bucket:
Option<String>) -> Self {
        let sdk_config = aws_config::load_from_env().await;
        let lambda_name = LambdaName(lambda_name.unwrap_or_else(|| {
            std::env::var("LAMBDA_NAME").unwrap_or_else(|_|
"rust_lambda_example".to_string())
        }));
        let role_name = RoleName(format!("{}_role", lambda_name.0));
        let (bucket, own_bucket) =
            match bucket {
                Some(bucket) => (Bucket(bucket), false),
                None => (
                    Bucket(std::env::var("LAMBDA_BUCKET").unwrap_or_else(|_| {
                        format!("rust-lambda-example-{}", uuid::Uuid::new_v4())
                    })),
                    true,
                ),
            };

        let s3_client = aws_sdk_s3::Client::new(&sdk_config);

        if own_bucket {
            info!("Creating bucket for demo: {}", bucket.0);
            s3_client
                .create_bucket()
                .bucket(bucket.0.clone())
                .create_bucket_configuration(
                    CreateBucketConfiguration::builder()
```

```
 .location_constraint(aws_sdk_s3::types::BucketLocationConstraint::from(
                            sdk_config.region().unwrap().as_ref(),
                        ))
                        .build(),
                )
                .send()
                .await
                .unwrap();
        }

        Self::new(
            aws_sdk_iam::Client::new(&sdk_config),
            aws_sdk_lambda::Client::new(&sdk_config),
            s3_client,
            lambda_name,
            role_name,
            bucket,
            OwnBucket(own_bucket),
        )
    }

    // snippet-start:[lambda.rust.scenario.prepare_function]
    /**
     * Upload function code from a path to a zip file.
     * The zip file must have an AL2 Linux-compatible binary called `bootstrap`.
     * The easiest way to create such a zip is to use `cargo lambda build --
output-format Zip`.
     */
    async fn prepare_function(
        &self,
        zip_file: PathBuf,
        key: Option<String>,
    ) -> Result<FunctionCode, anyhow::Error> {
        let body = ByteStream::from_path(zip_file).await?;

        let key = key.unwrap_or_else(|| format!("{}_code", self.lambda_name));

        info!("Uploading function code to s3://{}/{}", self.bucket, key);
        let _ = self
            .s3_client
            .put_object()
            .bucket(self.bucket.clone())
            .key(key.clone())
```

```
            .body(body)
            .send()
            .await?;

        Ok(FunctionCode::builder()
            .s3_bucket(self.bucket.clone())
            .s3_key(key)
            .build())
    }
    // snippet-end:[lambda.rust.scenario.prepare_function]

    // snippet-start:[lambda.rust.scenario.create_function]
    /**
     * Create a function, uploading from a zip file.
     */
    pub async fn create_function(&self, zip_file: PathBuf) -> Result<String,
anyhow::Error> {
        let code = self.prepare_function(zip_file, None).await?;

        let key = code.s3_key().unwrap().to_string();

        let role = self.create_role().await.map_err(|e| anyhow!(e))?;

        info!("Created iam role, waiting 15s for it to become active");
        tokio::time::sleep(Duration::from_secs(15)).await;

        info!("Creating lambda function {}", self.lambda_name);
        let _ = self
            .lambda_client
            .create_function()
            .function_name(self.lambda_name.clone())
            .code(code)
            .role(role.arn())
            .runtime(aws_sdk_lambda::types::Runtime::Providedal2)
            .handler("_unused")
            .send()
            .await
            .map_err(anyhow::Error::from)?;

        self.wait_for_function_ready().await?;

        self.lambda_client
            .publish_version()
            .function_name(self.lambda_name.clone())
```

```
            .send()
            .await?;

        Ok(key)
    }
    // snippet-end:[lambda.rust.scenario.create_function]

    /**
     * Create an IAM execution role for the managed Lambda function.
     * If the role already exists, use that instead.
     */
    async fn create_role(&self) -> Result<aws_sdk_iam::types::Role,
CreateRoleError> {
        info!("Creating execution role for function");
        let get_role = self
            .iam_client
            .get_role()
            .role_name(self.role_name.clone())
            .send()
            .await;
        if let Ok(get_role) = get_role {
            if let Some(role) = get_role.role {
                return Ok(role);
            }
        }

        let create_role = self
            .iam_client
            .create_role()
            .role_name(self.role_name.clone())
            .assume_role_policy_document(ROLE_POLICY_DOCUMENT)
            .send()
            .await;

        match create_role {
            Ok(create_role) => match create_role.role {
                Some(role) => Ok(role),
                None => Err(CreateRoleError::generic(
                    ErrorMetadata::builder()
                        .message("CreateRole returned empty success")
                        .build(),
                )),
            },
            Err(err) => Err(err.into_service_error()),
```

```
        }
    }

    /**
     * Poll `is_function_ready` with a 1-second delay. It returns when the
function is ready or when there's an error checking the function's state.
     */
    pub async fn wait_for_function_ready(&self) -> Result<(), anyhow::Error> {
        info!("Waiting for function");
        while !self.is_function_ready(None).await? {
            info!("Function is not ready, sleeping 1s");
            tokio::time::sleep(Duration::from_secs(1)).await;
        }
        Ok(())
    }

    /**
     * Check if a Lambda function is ready to be invoked.
     * A Lambda function is ready for this scenario when its state is active and
its LastUpdateStatus is Successful.
     * Additionally, if a sha256 is provided, the function must have that as its
current code hash.
     * Any missing properties or failed requests will be reported as an Err.
     */
    async fn is_function_ready(
        &self,
        expected_code_sha256: Option<&str>,
    ) -> Result<bool, anyhow::Error> {
        match self.get_function().await {
            Ok(func) => {
                if let Some(config) = func.configuration() {
                    if let Some(state) = config.state() {
                        info!(?state, "Checking if function is active");
                        if !matches!(state, State::Active) {
                            return Ok(false);
                        }
                    }
                    match config.last_update_status() {
                        Some(last_update_status) => {
                            info!(?last_update_status, "Checking if function is
ready");
                            match last_update_status {
                                LastUpdateStatus::Successful => {
                                    // continue
```

```
                                }
                                LastUpdateStatus::Failed |
LastUpdateStatus::InProgress => {
                                    return Ok(false);
                                }
                                unknown => {
                                    warn!(
                                        status_variant = unknown.as_str(),
                                        "LastUpdateStatus unknown"
                                    );
                                    return Err(anyhow!(
                                        "Unknown LastUpdateStatus, fn config is
{config:?}"
                                    ));
                                }
                            }
                        }
                        None => {
                            warn!("Missing last update status");
                            return Ok(false);
                        }
                    };
                    if expected_code_sha256.is_none() {
                        return Ok(true);
                    }
                    if let Some(code_sha256) = config.code_sha256() {
                        return Ok(code_sha256 ==
expected_code_sha256.unwrap_or_default());
                    }
                }
            }
            Err(e) => {
                warn!(?e, "Could not get function while waiting");
            }
        }
        Ok(false)
    }

    // snippet-start:[lambda.rust.scenario.get_function]
    /** Get the Lambda function with this Manager's name. */
    pub async fn get_function(&self) -> Result<GetFunctionOutput, anyhow::Error>
{
        info!("Getting lambda function");
        self.lambda_client
```

```
            .get_function()
            .function_name(self.lambda_name.clone())
            .send()
            .await
            .map_err(anyhow::Error::from)
    }
    // snippet-end:[lambda.rust.scenario.get_function]

    // snippet-start:[lambda.rust.scenario.list_functions]
    /** List all Lambda functions in the current Region. */
    pub async fn list_functions(&self) -> Result<ListFunctionsOutput,
anyhow::Error> {
        info!("Listing lambda functions");
        self.lambda_client
            .list_functions()
            .send()
            .await
            .map_err(anyhow::Error::from)
    }
    // snippet-end:[lambda.rust.scenario.list_functions]

    // snippet-start:[lambda.rust.scenario.invoke]
    /** Invoke the lambda function using calculator InvokeArgs. */
    pub async fn invoke(&self, args: InvokeArgs) -> Result<InvokeOutput,
anyhow::Error> {
        info!(?args, "Invoking {}", self.lambda_name);
        let payload = serde_json::to_string(&args)?;
        debug!(?payload, "Sending payload");
        self.lambda_client
            .invoke()
            .function_name(self.lambda_name.clone())
            .payload(Blob::new(payload))
            .send()
            .await
            .map_err(anyhow::Error::from)
    }
    // snippet-end:[lambda.rust.scenario.invoke]

    // snippet-start:[lambda.rust.scenario.update_function_code]
    /** Given a Path to a zip file, update the function's code and wait for the
update to finish. */
    pub async fn update_function_code(
        &self,
        zip_file: PathBuf,
```

```
        key: String,
    ) -> Result<UpdateFunctionCodeOutput, anyhow::Error> {
        let function_code = self.prepare_function(zip_file, Some(key)).await?;

        info!("Updating code for {}", self.lambda_name);
        let update = self
            .lambda_client
            .update_function_code()
            .function_name(self.lambda_name.clone())
            .s3_bucket(self.bucket.clone())
            .s3_key(function_code.s3_key().unwrap().to_string())
            .send()
            .await
            .map_err(anyhow::Error::from)?;

        self.wait_for_function_ready().await?;

        Ok(update)
    }
    // snippet-end:[lambda.rust.scenario.update_function_code]

    // snippet-start:[lambda.rust.scenario.update_function_configuration]
    /** Update the environment for a function. */
    pub async fn update_function_configuration(
        &self,
        environment: Environment,
    ) -> Result<UpdateFunctionConfigurationOutput, anyhow::Error> {
        info!(
            ?environment,
            "Updating environment for {}", self.lambda_name
        );
        let updated = self
            .lambda_client
            .update_function_configuration()
            .function_name(self.lambda_name.clone())
            .environment(environment)
            .send()
            .await
            .map_err(anyhow::Error::from)?;

        self.wait_for_function_ready().await?;

        Ok(updated)
    }
```

```
    // snippet-end:[lambda.rust.scenario.update_function_configuration]

    // snippet-start:[lambda.rust.scenario.delete_function]
    /** Delete a function and its role, and if possible or necessary, its
associated code object and bucket. */
    pub async fn delete_function(
        &self,
        location: Option<String>,
    ) -> (
        Result<DeleteFunctionOutput, anyhow::Error>,
        Result<DeleteRoleOutput, anyhow::Error>,
        Option<Result<DeleteObjectOutput, anyhow::Error>>,
    ) {
        info!("Deleting lambda function {}", self.lambda_name);
        let delete_function = self
            .lambda_client
            .delete_function()
            .function_name(self.lambda_name.clone())
            .send()
            .await
            .map_err(anyhow::Error::from);

        info!("Deleting iam role {}", self.role_name);
        let delete_role = self
            .iam_client
            .delete_role()
            .role_name(self.role_name.clone())
            .send()
            .await
            .map_err(anyhow::Error::from);

        let delete_object: Option<Result<DeleteObjectOutput, anyhow::Error>> =
            if let Some(location) = location {
                info!("Deleting object {location}");
                Some(
                    self.s3_client
                        .delete_object()
                        .bucket(self.bucket.clone())
                        .key(location)
                        .send()
                        .await
                        .map_err(anyhow::Error::from),
                )
            } else {
```

```
                info!(?location, "Skipping delete object");
                None
            };

        (delete_function, delete_role, delete_object)
    }
    // snippet-end:[lambda.rust.scenario.delete_function]

    pub async fn cleanup(
        &self,
        location: Option<String>,
    ) -> (
        (
            Result<DeleteFunctionOutput, anyhow::Error>,
            Result<DeleteRoleOutput, anyhow::Error>,
            Option<Result<DeleteObjectOutput, anyhow::Error>>,
        ),
        Option<Result<DeleteBucketOutput, anyhow::Error>>,
    ) {
        let delete_function = self.delete_function(location).await;

        let delete_bucket = if self.own_bucket {
            info!("Deleting bucket {}", self.bucket);
            if delete_function.2.is_none() ||
 delete_function.2.as_ref().unwrap().is_ok() {
                Some(
                    self.s3_client
                        .delete_bucket()
                        .bucket(self.bucket.clone())
                        .send()
                        .await
                        .map_err(anyhow::Error::from),
                )
            } else {
                None
            }
        } else {
            info!("No bucket to clean up");
            None
        };

        (delete_function, delete_bucket)
    }
}
```

```
/**
 * Testing occurs primarily as an integration test running the `scenario` bin
 successfully.
 * Each action relies deeply on the internal workings and state of Amazon Simple
 Storage Service (Amazon S3), Lambda, and IAM working together.
 * It is therefore infeasible to mock the clients to test the individual actions.
 */
#[cfg(test)]
mod test {
    use super::{InvokeArgs, Operation};
    use serde_json::json;

    /** Make sure that the JSON output of serializing InvokeArgs is what's
 expected by the calculator. */
    #[test]
    fn test_serialize() {
        assert_eq!(json!(InvokeArgs::Increment(5)), 5);
        assert_eq!(
            json!(InvokeArgs::Arithmetic(Operation::Plus, 5, 7)).to_string(),
            r#"{"op":"plus","i":5,"j":7}"#.to_string(),
        );
    }
}
```

從前端到後端執行該案例的二進位檔案，使用命令列旗標來控制某些行為。此檔案是套件中的 src/bin/scenario.rs。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

/*
## Service actions

Service actions wrap the SDK call, taking a client and any specific parameters
 necessary for the call.

* CreateFunction
* GetFunction
* ListFunctions
* Invoke
* UpdateFunctionCode
```

```
* UpdateFunctionConfiguration
* DeleteFunction

## Scenario
A scenario runs at a command prompt and prints output to the user on the result
 of each service action. A scenario can run in one of two ways: straight through,
 printing out progress as it goes, or as an interactive question/answer script.

## Getting started with functions

Use an SDK to manage AWS Lambda functions: create a function, invoke it, update
 its code, invoke it again, view its output and logs, and delete it.

This scenario uses two Lambda handlers:
_Note: Handlers don't use AWS SDK API calls._

The increment handler is straightforward:

1. It accepts a number, increments it, and returns the new value.
2. It performs simple logging of the result.

The arithmetic handler is more complex:
1. It accepts a set of actions ['plus', 'minus', 'times', 'divided-by'] and two
 numbers, and returns the result of the calculation.
2. It uses an environment variable to control log level (such as DEBUG, INFO,
 WARNING, ERROR).
It logs a few things at different levels, such as:
    * DEBUG: Full event data.
    * INFO: The calculation result.
    * WARN~ING~: When a divide by zero error occurs.
    * This will be the typical `RUST_LOG` variable.


The steps of the scenario are:

1. Create an AWS Identity and Access Management (IAM) role that meets the
 following requirements:
    * Has an assume_role policy that grants 'lambda.amazonaws.com' the
 'sts:AssumeRole' action.
    * Attaches the 'arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole' managed role.
    * _You must wait for ~10 seconds after the role is created before you can use
 it!_
```

```
2. Create a function (CreateFunction) for the increment handler by packaging it
 as a zip and doing one of the following:
    * Adding it with CreateFunction Code.ZipFile.
    * --or--
    * Uploading it to Amazon Simple Storage Service (Amazon S3) and adding it
 with CreateFunction Code.S3Bucket/S3Key.
    * _Note: Zipping the file does not have to be done in code._
    * If you have a waiter, use it to wait until the function is active.
 Otherwise, call GetFunction until State is Active.
3. Invoke the function with a number and print the result.
4. Update the function (UpdateFunctionCode) to the arithmetic handler by
 packaging it as a zip and doing one of the following:
    * Adding it with UpdateFunctionCode ZipFile.
    * --or--
    * Uploading it to Amazon S3 and adding it with UpdateFunctionCode S3Bucket/
S3Key.
5. Call GetFunction until Configuration.LastUpdateStatus is 'Successful' (or
 'Failed').
6. Update the environment variable by calling UpdateFunctionConfiguration and
 pass it a log level, such as:
    * Environment={'Variables': {'RUST_LOG': 'TRACE'}}
7. Invoke the function with an action from the list and a couple of values.
 Include LogType='Tail' to get logs in the result. Print the result of the
 calculation and the log.
8. [Optional] Invoke the function to provoke a divide-by-zero error and show the
 log result.
9. List all functions for the account, using pagination (ListFunctions).
10. Delete the function (DeleteFunction).
11. Delete the role.

Each step should use the function created in Service Actions to abstract calling
 the SDK.
 */

use aws_sdk_lambda::{operation::invoke::InvokeOutput, types::Environment};
use clap::Parser;
use std::{collections::HashMap, path::PathBuf};
use tracing::{debug, info, warn};
use tracing_subscriber::EnvFilter;

use lambda_code_examples::actions::{
    InvokeArgs::{Arithmetic, Increment},
    LambdaManager, Operation,
};
```

```
#[derive(Debug, Parser)]
pub struct Opt {
    /// The AWS Region.
    #[structopt(short, long)]
    pub region: Option<String>,

    // The bucket to use for the FunctionCode.
    #[structopt(short, long)]
    pub bucket: Option<String>,

    // The name of the Lambda function.
    #[structopt(short, long)]
    pub lambda_name: Option<String>,

    // The number to increment.
    #[structopt(short, long, default_value = "12")]
    pub inc: i32,

    // The left operand.
    #[structopt(long, default_value = "19")]
    pub num_a: i32,

    // The right operand.
    #[structopt(long, default_value = "23")]
    pub num_b: i32,

    // The arithmetic operation.
    #[structopt(short, long, default_value = "plus")]
    pub operation: Operation,

    #[structopt(long)]
    pub cleanup: Option<bool>,

    #[structopt(long)]
    pub no_cleanup: Option<bool>,
}

fn code_path(lambda: &str) -> PathBuf {
    PathBuf::from(format!("../target/lambda/{lambda}/bootstrap.zip"))
}

// snippet-start:[lambda.rust.scenario.log_invoke_output]
fn log_invoke_output(invoke: &InvokeOutput, message: &str) {
```

```
    if let Some(payload) = invoke.payload().cloned() {
        let payload = String::from_utf8(payload.into_inner());
        info!(?payload, message);
    } else {
        info!("Could not extract payload")
    }
    if let Some(logs) = invoke.log_result() {
        debug!(?logs, "Invoked function logs")
    } else {
        debug!("Invoked function had no logs")
    }
}
// snippet-end:[lambda.rust.scenario.log_invoke_output]

async fn main_block(
    opt: &Opt,
    manager: &LambdaManager,
    code_location: String,
) -> Result<(), anyhow::Error> {
    let invoke = manager.invoke(Increment(opt.inc)).await?;
    log_invoke_output(&invoke, "Invoked function configured as increment");

    let update_code = manager
        .update_function_code(code_path("arithmetic"), code_location.clone())
        .await?;

    let code_sha256 = update_code.code_sha256().unwrap_or("Unknown SHA");
    info!(?code_sha256, "Updated function code with arithmetic.zip");

    let arithmetic_args = Arithmetic(opt.operation, opt.num_a, opt.num_b);
    let invoke = manager.invoke(arithmetic_args).await?;
    log_invoke_output(&invoke, "Invoked function configured as arithmetic");

    let update = manager
        .update_function_configuration(
            Environment::builder()
                .set_variables(Some(HashMap::from([(
                    "RUST_LOG".to_string(),
                    "trace".to_string(),
                )])))
                .build(),
        )
        .await?;
    let updated_environment = update.environment();
```

```
    info!(?updated_environment, "Updated function configuration");

    let invoke = manager
        .invoke(Arithmetic(opt.operation, opt.num_a, opt.num_b))
        .await?;
    log_invoke_output(
        &invoke,
        "Invoked function configured as arithmetic with increased logging",
    );

    let invoke = manager
        .invoke(Arithmetic(Operation::DividedBy, opt.num_a, 0))
        .await?;
    log_invoke_output(
        &invoke,
        "Invoked function configured as arithmetic with divide by zero",
    );

    Ok::<(), anyhow::Error>(())
}

#[tokio::main]
async fn main() {
    tracing_subscriber::fmt()
        .without_time()
        .with_file(true)
        .with_line_number(true)
        .with_env_filter(EnvFilter::from_default_env())
        .init();

    let opt = Opt::parse();
    let manager = LambdaManager::load_from_env(opt.lambda_name.clone(),
 opt.bucket.clone()).await;

    let key = match manager.create_function(code_path("increment")).await {
        Ok(init) => {
            info!(?init, "Created function, initially with increment.zip");
            let run_block = main_block(&opt, &manager, init.clone()).await;
            info!(?run_block, "Finished running example, cleaning up");
            Some(init)
        }
        Err(err) => {
            warn!(?err, "Error happened when initializing function");
            None
```

```
        }
    };

    if Some(false) == opt.cleanup || Some(true) == opt.no_cleanup {
        info!("Skipping cleanup")
    } else {
        let delete = manager.cleanup(key).await;
        info!(?delete, "Deleted function & cleaned up resources");
    }
}
```

- 如需 API 詳細資訊，請參閱《適用於 Rust 的AWS SDK API 參考》中的下列主題。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunction代碼
  - UpdateFunction配置

SAP ABAP

適用於 SAP ABAP 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

```
   TRY.
       "Create an AWS Identity and Access Management (IAM) role that grants AWS
Lambda permission to write to logs."
       DATA(lv_policy_document) = `{` &&
           `"Version":"2012-10-17",` &&
                 `"Statement": [` &&
```

```
                    `{` &&
                      `"Effect": "Allow",` &&
                      `"Action": [` &&
                        `"sts:AssumeRole"` &&
                      `],` &&
                      `"Principal": {` &&
                        `"Service": [` &&
                          `"lambda.amazonaws.com"` &&
                        `]` &&
                      `}` &&
                    `}` &&
                  `]` &&
                `}`.
        TRY.
            DATA(lo_create_role_output) =  lo_iam->createrole(
                    iv_rolename = iv_role_name
                    iv_assumerolepolicydocument = lv_policy_document
                    iv_description = 'Grant lambda permission to write to logs'
                ).
            MESSAGE 'IAM role created.' TYPE 'I'.
            WAIT UP TO 10 SECONDS.            " Make sure that the IAM role is
 ready for use. "
          CATCH /aws1/cx_iamentityalrdyexex.
            MESSAGE 'IAM role already exists.' TYPE 'E'.
          CATCH /aws1/cx_iaminvalidinputex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_iammalformedplydocex.
            MESSAGE 'Policy document in the request is malformed.' TYPE 'E'.
        ENDTRY.

        TRY.
            lo_iam->attachrolepolicy(
                iv_rolename  = iv_role_name
                iv_policyarn = 'arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole'
            ).
            MESSAGE 'Attached policy to the IAM role.' TYPE 'I'.
          CATCH /aws1/cx_iaminvalidinputex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_iamnosuchentityex.
            MESSAGE 'The requested resource entity does not exist.' TYPE 'E'.
          CATCH /aws1/cx_iamplynotattachableex.
            MESSAGE 'Service role policies can only be attached to the service-
linked role for their service.' TYPE 'E'.
```

```
          CATCH /aws1/cx_iamunmodableentityex.
            MESSAGE 'Service that depends on the service-linked role is not
 modifiable.' TYPE 'E'.
        ENDTRY.

        " Create a Lambda function and upload handler code. "
        " Lambda function performs 'increment' action on a number. "
        TRY.
            lo_lmd->createfunction(
                 iv_functionname = iv_function_name
                 iv_runtime = `python3.9`
                 iv_role = lo_create_role_output->get_role( )->get_arn( )
                 iv_handler = iv_handler
                 io_code = io_initial_zip_file
                 iv_description = 'AWS Lambda code example'
             ).
            MESSAGE 'Lambda function created.' TYPE 'I'.
          CATCH /aws1/cx_lmdcodestorageexcdex.
            MESSAGE 'Maximum total code size per account exceeded.' TYPE 'E'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'E'.
        ENDTRY.

        " Verify the function is in Active state "
        WHILE lo_lmd->getfunction( iv_functionname = iv_function_name )-
>get_configuration( )->ask_state( ) <> 'Active'.
          IF sy-index = 10.
            EXIT.               " Maximum 10 seconds. "
          ENDIF.
          WAIT UP TO 1 SECONDS.
        ENDWHILE.

        "Invoke the function with a single parameter and get results."
        TRY.
            DATA(lv_json) = /aws1/cl_rt_util=>string_to_xstring(
              `{`  &&
                `"action": "increment",`  &&
                `"number": 10` &&
              `}`
            ).
            DATA(lo_initial_invoke_output) =  lo_lmd->invoke(
                       iv_functionname = iv_function_name
```

```
                        iv_payload = lv_json
                    ).
            ov_initial_invoke_payload = lo_initial_invoke_output->get_payload( ).
           " ov_initial_invoke_payload is returned for testing purposes. "
            DATA(lo_writer_json) = cl_sxml_string_writer=>create( type =
 if_sxml=>co_xt_json ).
            CALL TRANSFORMATION id SOURCE XML ov_initial_invoke_payload RESULT
 XML lo_writer_json.
            DATA(lv_result) = cl_abap_codepage=>convert_from( lo_writer_json-
>get_output( ) ).
            MESSAGE 'Lambda function invoked.' TYPE 'I'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdinvrequestcontex.
            MESSAGE 'Unable to parse request body as JSON.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'E'.
          CATCH /aws1/cx_lmdunsuppedmediatyp00.
            MESSAGE 'Invoke request body does not have JSON as its content type.'
 TYPE 'E'.
        ENDTRY.

        " Update the function code and configure its Lambda environment with an
 environment variable. "
        " Lambda function is updated to perform 'decrement' action also. "
        TRY.
            lo_lmd->updatefunctioncode(
                  iv_functionname = iv_function_name
                  iv_zipfile = io_updated_zip_file
              ).
            WAIT UP TO 10 SECONDS.            " Make sure that the update is
 completed. "
            MESSAGE 'Lambda function code updated.' TYPE 'I'.
          CATCH /aws1/cx_lmdcodestorageexcdex.
            MESSAGE 'Maximum total code size per account exceeded.' TYPE 'E'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'E'.
        ENDTRY.

        TRY.
            DATA lt_variables TYPE /aws1/
cl_lmdenvironmentvaria00=>tt_environmentvariables.
```

```
            DATA ls_variable LIKE LINE OF lt_variables.
            ls_variable-key = 'LOG_LEVEL'.
            ls_variable-value = NEW /aws1/cl_lmdenvironmentvaria00( iv_value =
 'info' ).
            INSERT ls_variable INTO TABLE lt_variables.

            lo_lmd->updatefunctionconfiguration(
                  iv_functionname = iv_function_name
                  io_environment = NEW /aws1/cl_lmdenvironment( it_variables =
 lt_variables )
              ).
            WAIT UP TO 10 SECONDS.            " Make sure that the update is
 completed. "
            MESSAGE 'Lambda function configuration/settings updated.' TYPE 'I'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourceconflictex.
            MESSAGE 'Resource already exists or another operation is in
 progress.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'E'.
        ENDTRY.

        "Invoke the function with new parameters and get results. Display the
 execution log that's returned from the invocation."
        TRY.
            lv_json = /aws1/cl_rt_util=>string_to_xstring(
              `{`  &&
                `"action": "decrement",`  &&
                `"number": 10` &&
              `}`
            ).
            DATA(lo_updated_invoke_output) =  lo_lmd->invoke(
                       iv_functionname = iv_function_name
                       iv_payload = lv_json
                  ).
            ov_updated_invoke_payload = lo_updated_invoke_output->get_payload( ).
           " ov_updated_invoke_payload is returned for testing purposes. "
            lo_writer_json = cl_sxml_string_writer=>create( type =
 if_sxml=>co_xt_json ).
            CALL TRANSFORMATION id SOURCE XML ov_updated_invoke_payload RESULT
 XML lo_writer_json.
            lv_result = cl_abap_codepage=>convert_from( lo_writer_json-
>get_output( ) ).
```

```
            MESSAGE 'Lambda function invoked.' TYPE 'I'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdinvrequestcontex.
            MESSAGE 'Unable to parse request body as JSON.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'E'.
          CATCH /aws1/cx_lmdunsuppedmediatyp00.
            MESSAGE 'Invoke request body does not have JSON as its content type.'
 TYPE 'E'.
        ENDTRY.

        " List the functions for your account. "
        TRY.
            DATA(lo_list_output) = lo_lmd->listfunctions( ).
            DATA(lt_functions) = lo_list_output->get_functions( ).
            MESSAGE 'Retrieved list of Lambda functions.' TYPE 'I'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
        ENDTRY.

        " Delete the Lambda function. "
        TRY.
            lo_lmd->deletefunction( iv_functionname = iv_function_name ).
            MESSAGE 'Lambda function deleted.' TYPE 'I'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'E'.
        ENDTRY.

        " Detach role policy. "
        TRY.
            lo_iam->detachrolepolicy(
                iv_rolename  = iv_role_name
                iv_policyarn = 'arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole'
            ).
            MESSAGE 'Detached policy from the IAM role.' TYPE 'I'.
          CATCH /aws1/cx_iaminvalidinputex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_iamnosuchentityex.
            MESSAGE 'The requested resource entity does not exist.' TYPE 'E'.
          CATCH /aws1/cx_iamplynotattachableex.
```

```
            MESSAGE 'Service role policies can only be attached to the service-
linked role for their service.' TYPE 'E'.
          CATCH /aws1/cx_iamunmodableentityex.
            MESSAGE 'Service that depends on the service-linked role is not
 modifiable.' TYPE 'E'.
        ENDTRY.

        " Delete the IAM role. "
        TRY.
            lo_iam->deleterole( iv_rolename = iv_role_name ).
            MESSAGE 'IAM role deleted.' TYPE 'I'.
          CATCH /aws1/cx_iamnosuchentityex.
            MESSAGE 'The requested resource entity does not exist.' TYPE 'E'.
          CATCH /aws1/cx_iamunmodableentityex.
            MESSAGE 'Service that depends on the service-linked role is not
 modifiable.' TYPE 'E'.
        ENDTRY.

      CATCH /aws1/cx_rt_service_generic INTO lo_exception.
        DATA(lv_error) = lo_exception->get_longtext( ).
        MESSAGE lv_error TYPE 'E'.
    ENDTRY.
```

- 如需 API 詳細資訊，請參閱《適用於 SAP ABAP 的AWS SDK API 參考》中的下列主題。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunction代碼
  - UpdateFunction配置

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使用開發套件 AWS 進行 Amazon Cognito 使用者身份驗證之後，使用 Lambda 函數寫入自訂活動

下列程式碼範例示範如何在 Amazon Cognito 使用者身分驗證後，使用 Lambda 函數撰寫自訂活動資料。

- 使用管理員功能將使用者新增至使用者集區。
- 設定使用者集區以呼叫`PostAuthentication`觸發器的 Lambda 函數。
- 將新用戶登錄到 Amazon Cognito。
- Lambda 函數會將自訂資訊寫入 CloudWatch 日誌和 DynamoDB 資料表。
- 從 DynamoDB 表格取得並顯示自訂資料，然後清理資源。

Go

SDK for Go V2

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在AWS 設定和執行程式碼範例儲存庫。

在命令提示中執行互動式案例。

```
// ActivityLog separates the steps of this scenario into individual functions so
 that
// they are simpler to read and understand.
type ActivityLog struct {
 helper       IScenarioHelper
 questioner   demotools.IQuestioner
 resources    Resources
 cognitoActor *actions.CognitoActions
}

// NewActivityLog constructs a new activity log runner.
func NewActivityLog(sdkConfig aws.Config, questioner demotools.IQuestioner,
 helper IScenarioHelper) ActivityLog {
 scenario := ActivityLog{
```

```
  helper:       helper,
  questioner:   questioner,
  resources:    Resources{},
  cognitoActor: &actions.CognitoActions{CognitoClient:
 cognitoidentityprovider.NewFromConfig(sdkConfig)},
 }
 scenario.resources.init(scenario.cognitoActor, questioner)
 return scenario
}

// AddUserToPool selects a user from the known users table and uses administrator
 credentials to add the user to the user pool.
func (runner *ActivityLog) AddUserToPool(userPoolId string, tableName string)
 (string, string) {
 log.Println("To facilitate this example, let's add a user to the user pool using
 administrator privileges.")
 users, err := runner.helper.GetKnownUsers(tableName)
 if err != nil {
  panic(err)
 }
 user := users.Users[0]
 log.Printf("Adding known user %v to the user pool.\n", user.UserName)
 err = runner.cognitoActor.AdminCreateUser(userPoolId, user.UserName,
 user.UserEmail)
 if err != nil {
  panic(err)
 }
 pwSet := false
 password := runner.questioner.AskPassword("\nEnter a password that has at least
 eight characters, uppercase, lowercase, numbers and symbols.\n"+
  "(the password will not display as you type):", 8)
 for !pwSet {
  log.Printf("\nSetting password for user '%v'.\n", user.UserName)
  err = runner.cognitoActor.AdminSetUserPassword(userPoolId, user.UserName,
 password)
  if err != nil {
   var invalidPassword *types.InvalidPasswordException
   if errors.As(err, &invalidPassword) {
    password = runner.questioner.AskPassword("\nEnter another password:", 8)
   } else {
    panic(err)
   }
  } else {
   pwSet = true
```

```
  }
 }

 log.Println(strings.Repeat("-", 88))

 return user.UserName, password
}

// AddActivityLogTrigger adds a Lambda handler as an invocation target for the
 PostAuthentication trigger.
func (runner *ActivityLog) AddActivityLogTrigger(userPoolId string,
 activityLogArn string) {
 log.Println("Let's add a Lambda function to handle the PostAuthentication
 trigger from Cognito.\n" +
  "This trigger happens after a user is authenticated, and lets your function
 take action, such as logging\n" +
  "the outcome.")
 err := runner.cognitoActor.UpdateTriggers(
  userPoolId,
  actions.TriggerInfo{Trigger: actions.PostAuthentication, HandlerArn:
 aws.String(activityLogArn)})
 if err != nil {
  panic(err)
 }
 runner.resources.triggers = append(runner.resources.triggers,
 actions.PostAuthentication)
 log.Printf("Lambda function %v added to user pool %v to handle
 PostAuthentication Cognito trigger.\n",
  activityLogArn, userPoolId)

 log.Println(strings.Repeat("-", 88))
}

// SignInUser signs in as the specified user.
func (runner *ActivityLog) SignInUser(clientId string, userName string, password
 string) {
 log.Printf("Now we'll sign in user %v and check the results in the logs and the
 DynamoDB table.", userName)
 runner.questioner.Ask("Press Enter when you're ready.")
 authResult, err := runner.cognitoActor.SignIn(clientId, userName, password)
 if err != nil {
  panic(err)
 }
 log.Println("Sign in successful.",
```

```
  "The PostAuthentication Lambda handler writes custom information to CloudWatch
 Logs.")

 runner.resources.userAccessTokens = append(runner.resources.userAccessTokens,
 *authResult.AccessToken)
}

// GetKnownUserLastLogin gets the login info for a user from the Amazon DynamoDB
 table and displays it.
func (runner *ActivityLog) GetKnownUserLastLogin(tableName string, userName
 string) {
 log.Println("The PostAuthentication handler also writes login data to the
 DynamoDB table.")
 runner.questioner.Ask("Press Enter when you're ready to continue.")
 users, err := runner.helper.GetKnownUsers(tableName)
 if err != nil {
  panic(err)
 }
 for _, user := range users.Users {
  if user.UserName == userName {
   log.Println("The last login info for the user in the known users table is:")
   log.Printf("\t%+v", *user.LastLogin)
  }
 }
 log.Println(strings.Repeat("-", 88))
}

// Run runs the scenario.
func (runner *ActivityLog) Run(stackName string) {
 defer func() {
  if r := recover(); r != nil {
   log.Println("Something went wrong with the demo.")
   runner.resources.Cleanup()
  }
 }()

 log.Println(strings.Repeat("-", 88))
 log.Printf("Welcome\n")

 log.Println(strings.Repeat("-", 88))

 stackOutputs, err := runner.helper.GetStackOutputs(stackName)
 if err != nil {
  panic(err)
```

```
 }
 runner.resources.userPoolId = stackOutputs["UserPoolId"]
 runner.helper.PopulateUserTable(stackOutputs["TableName"])
 userName, password := runner.AddUserToPool(stackOutputs["UserPoolId"],
 stackOutputs["TableName"])

 runner.AddActivityLogTrigger(stackOutputs["UserPoolId"],
 stackOutputs["ActivityLogFunctionArn"])
 runner.SignInUser(stackOutputs["UserPoolClientId"], userName, password)
 runner.helper.ListRecentLogEvents(stackOutputs["ActivityLogFunction"])
 runner.GetKnownUserLastLogin(stackOutputs["TableName"], userName)

 runner.resources.Cleanup()

 log.Println(strings.Repeat("-", 88))
 log.Println("Thanks for watching!")
 log.Println(strings.Repeat("-", 88))
}
```

使用 Lambda 函數處理PostAuthentication觸發器。

```
const TABLE_NAME = "TABLE_NAME"

// LoginInfo defines structured login data that can be marshalled to a DynamoDB
 format.
type LoginInfo struct {
 UserPoolId string `dynamodbav:"UserPoolId"`
 ClientId   string `dynamodbav:"ClientId"`
 Time       string `dynamodbav:"Time"`
}

// UserInfo defines structured user data that can be marshalled to a DynamoDB
 format.
type UserInfo struct {
 UserName  string    `dynamodbav:"UserName"`
 UserEmail string    `dynamodbav:"UserEmail"`
 LastLogin LoginInfo `dynamodbav:"LastLogin"`
}

// GetKey marshals the user email value to a DynamoDB key format.
```

```
func (user UserInfo) GetKey() map[string]dynamodbtypes.AttributeValue {
 userEmail, err := attributevalue.Marshal(user.UserEmail)
 if err != nil {
  panic(err)
 }
 return map[string]dynamodbtypes.AttributeValue{"UserEmail": userEmail}
}

type handler struct {
 dynamoClient *dynamodb.Client
}

// HandleRequest handles the PostAuthentication event by writing custom data to
 the logs and
// to an Amazon DynamoDB table.
func (h *handler) HandleRequest(ctx context.Context,
 event events.CognitoEventUserPoolsPostAuthentication)
 (events.CognitoEventUserPoolsPostAuthentication, error) {
 log.Printf("Received post authentication trigger from %v for user '%v'",
 event.TriggerSource, event.UserName)
 tableName := os.Getenv(TABLE_NAME)
 user := UserInfo{
  UserName:  event.UserName,
  UserEmail: event.Request.UserAttributes["email"],
  LastLogin: LoginInfo{
   UserPoolId: event.UserPoolID,
   ClientId:   event.CallerContext.ClientID,
   Time:       time.Now().Format(time.UnixDate),
  },
 }
 // Write to CloudWatch Logs.
 fmt.Printf("%#v", user)

 // Also write to an external system. This examples uses DynamoDB to demonstrate.
 userMap, err := attributevalue.MarshalMap(user)
 if err != nil {
  log.Printf("Couldn't marshal to DynamoDB map. Here's why: %v\n", err)
 } else if len(userMap) == 0 {
  log.Printf("User info marshaled to an empty map.")
 } else {
  _, err := h.dynamoClient.PutItem(ctx, &dynamodb.PutItemInput{
   Item:      userMap,
   TableName: aws.String(tableName),
  })
```

```
  if err != nil {
   log.Printf("Couldn't write to DynamoDB. Here's why: %v\n", err)
  } else {
   log.Printf("Wrote user info to DynamoDB table %v.\n", tableName)
  }
 }

 return event, nil
}

func main() {
 sdkConfig, err := config.LoadDefaultConfig(context.TODO())
 if err != nil {
  log.Panicln(err)
 }
 h := handler{
  dynamoClient: dynamodb.NewFromConfig(sdkConfig),
 }
 lambda.Start(h.HandleRequest)
}
```

建立執行一般工作的結構。

```
// IScenarioHelper defines common functions used by the workflows in this
 example.
type IScenarioHelper interface {
 Pause(secs int)
 GetStackOutputs(stackName string) (actions.StackOutputs, error)
 PopulateUserTable(tableName string)
 GetKnownUsers(tableName string) (actions.UserList, error)
 AddKnownUser(tableName string, user actions.User)
 ListRecentLogEvents(functionName string)
}

// ScenarioHelper contains AWS wrapper structs used by the workflows in this
 example.
type ScenarioHelper struct {
 questioner  demotools.IQuestioner
 dynamoActor *actions.DynamoActions
 cfnActor    *actions.CloudFormationActions
```

```
 cwlActor    *actions.CloudWatchLogsActions
 isTestRun   bool
}

// NewScenarioHelper constructs a new scenario helper.
func NewScenarioHelper(sdkConfig aws.Config, questioner demotools.IQuestioner)
 ScenarioHelper {
 scenario := ScenarioHelper{
  questioner:  questioner,
  dynamoActor: &actions.DynamoActions{DynamoClient:
 dynamodb.NewFromConfig(sdkConfig)},
  cfnActor:    &actions.CloudFormationActions{CfnClient:
 cloudformation.NewFromConfig(sdkConfig)},
  cwlActor:    &actions.CloudWatchLogsActions{CwlClient:
 cloudwatchlogs.NewFromConfig(sdkConfig)},
 }
 return scenario
}

// Pause waits for the specified number of seconds.
func (helper ScenarioHelper) Pause(secs int) {
 if !helper.isTestRun {
  time.Sleep(time.Duration(secs) * time.Second)
 }
}

// GetStackOutputs gets the outputs from the specified CloudFormation stack in a
 structured format.
func (helper ScenarioHelper) GetStackOutputs(stackName string)
 (actions.StackOutputs, error) {
 return helper.cfnActor.GetOutputs(stackName), nil
}

// PopulateUserTable fills the known user table with example data.
func (helper ScenarioHelper) PopulateUserTable(tableName string) {
 log.Printf("First, let's add some users to the DynamoDB %v table we'll use for
 this example.\n", tableName)
 err := helper.dynamoActor.PopulateTable(tableName)
 if err != nil {
  panic(err)
 }
}
```

```
// GetKnownUsers gets the users from the known users table in a structured
 format.
func (helper ScenarioHelper) GetKnownUsers(tableName string) (actions.UserList,
 error) {
 knownUsers, err := helper.dynamoActor.Scan(tableName)
 if err != nil {
  log.Printf("Couldn't get known users from table %v. Here's why: %v\n",
 tableName, err)
 }
 return knownUsers, err
}

// AddKnownUser adds a user to the known users table.
func (helper ScenarioHelper) AddKnownUser(tableName string, user actions.User) {
 log.Printf("Adding user '%v' with email '%v' to the DynamoDB known users
 table...\n",
  user.UserName, user.UserEmail)
 err := helper.dynamoActor.AddUser(tableName, user)
 if err != nil {
  panic(err)
 }
}

// ListRecentLogEvents gets the most recent log stream and events for the
 specified Lambda function and displays them.
func (helper ScenarioHelper) ListRecentLogEvents(functionName string) {
 log.Println("Waiting a few seconds to let Lambda write to CloudWatch Logs...")
 helper.Pause(10)
 log.Println("Okay, let's check the logs to find what's happened recently with
 your Lambda function.")
 logStream, err := helper.cwlActor.GetLatestLogStream(functionName)
 if err != nil {
  panic(err)
 }
 log.Printf("Getting some recent events from log stream %v\n",
 *logStream.LogStreamName)
 events, err := helper.cwlActor.GetLogEvents(functionName,
 *logStream.LogStreamName, 10)
 if err != nil {
  panic(err)
 }
 for _, event := range events {
  log.Printf("\t%v", *event.Message)
 }
```

```
 log.Println(strings.Repeat("-", 88))
}
```

創建一個包裝 Amazon Cognito 操作的結構。

```
type CognitoActions struct {
 CognitoClient *cognitoidentityprovider.Client
}



// Trigger and TriggerInfo define typed data for updating an Amazon Cognito
 trigger.
type Trigger int

const (
 PreSignUp Trigger = iota
 UserMigration
 PostAuthentication
)

type TriggerInfo struct {
 Trigger    Trigger
 HandlerArn *string
}

// UpdateTriggers adds or removes Lambda triggers for a user pool. When a trigger
 is specified with a `nil` value,
// it is removed from the user pool.
func (actor CognitoActions) UpdateTriggers(userPoolId string,
 triggers ...TriggerInfo) error {
 output, err := actor.CognitoClient.DescribeUserPool(context.TODO(),
 &cognitoidentityprovider.DescribeUserPoolInput{
  UserPoolId: aws.String(userPoolId),
 })
 if err != nil {
  log.Printf("Couldn't get info about user pool %v. Here's why: %v\n",
 userPoolId, err)
  return err
```

```
 }
 lambdaConfig := output.UserPool.LambdaConfig
 for _, trigger := range triggers {
  switch trigger.Trigger {
  case PreSignUp:
   lambdaConfig.PreSignUp = trigger.HandlerArn
  case UserMigration:
   lambdaConfig.UserMigration = trigger.HandlerArn
  case PostAuthentication:
   lambdaConfig.PostAuthentication = trigger.HandlerArn
  }
 }
 _, err = actor.CognitoClient.UpdateUserPool(context.TODO(),
 &cognitoidentityprovider.UpdateUserPoolInput{
  UserPoolId:   aws.String(userPoolId),
  LambdaConfig: lambdaConfig,
 })
 if err != nil {
  log.Printf("Couldn't update user pool %v. Here's why: %v\n", userPoolId, err)
 }
 return err
}



// SignUp signs up a user with Amazon Cognito.
func (actor CognitoActions) SignUp(clientId string, userName string, password
 string, userEmail string) (bool, error) {
 confirmed := false
 output, err := actor.CognitoClient.SignUp(context.TODO(),
 &cognitoidentityprovider.SignUpInput{
  ClientId: aws.String(clientId),
  Password: aws.String(password),
  Username: aws.String(userName),
  UserAttributes: []types.AttributeType{
   {Name: aws.String("email"), Value: aws.String(userEmail)},
  },
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't sign up user %v. Here's why: %v\n", userName, err)
```

```
  }
 } else {
  confirmed = output.UserConfirmed
 }
 return confirmed, err
}



// SignIn signs in a user to Amazon Cognito using a username and password
 authentication flow.
func (actor CognitoActions) SignIn(clientId string, userName string, password
 string) (*types.AuthenticationResultType, error) {
 var authResult *types.AuthenticationResultType
 output, err := actor.CognitoClient.InitiateAuth(context.TODO(),
 &cognitoidentityprovider.InitiateAuthInput{
  AuthFlow:       "USER_PASSWORD_AUTH",
  ClientId:       aws.String(clientId),
  AuthParameters: map[string]string{"USERNAME": userName, "PASSWORD": password},
 })
 if err != nil {
  var resetRequired *types.PasswordResetRequiredException
  if errors.As(err, &resetRequired) {
   log.Println(*resetRequired.Message)
  } else {
   log.Printf("Couldn't sign in user %v. Here's why: %v\n", userName, err)
  }
 } else {
  authResult = output.AuthenticationResult
 }
 return authResult, err
}



// ForgotPassword starts a password recovery flow for a user. This flow typically
 sends a confirmation code
// to the user's configured notification destination, such as email.
func (actor CognitoActions) ForgotPassword(clientId string, userName string)
 (*types.CodeDeliveryDetailsType, error) {
 output, err := actor.CognitoClient.ForgotPassword(context.TODO(),
 &cognitoidentityprovider.ForgotPasswordInput{
  ClientId: aws.String(clientId),
  Username: aws.String(userName),
```

```
 })
 if err != nil {
  log.Printf("Couldn't start password reset for user '%v'. Here;s why: %v\n",
 userName, err)
 }
 return output.CodeDeliveryDetails, err
}



// ConfirmForgotPassword confirms a user with a confirmation code and a new
 password.
func (actor CognitoActions) ConfirmForgotPassword(clientId string, code string,
 userName string, password string) error {
 _, err := actor.CognitoClient.ConfirmForgotPassword(context.TODO(),
 &cognitoidentityprovider.ConfirmForgotPasswordInput{
  ClientId:         aws.String(clientId),
  ConfirmationCode: aws.String(code),
  Password:         aws.String(password),
  Username:         aws.String(userName),
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't confirm user %v. Here's why: %v", userName, err)
  }
 }
 return err
}



// DeleteUser removes a user from the user pool.
func (actor CognitoActions) DeleteUser(userAccessToken string) error {
 _, err := actor.CognitoClient.DeleteUser(context.TODO(),
 &cognitoidentityprovider.DeleteUserInput{
  AccessToken: aws.String(userAccessToken),
 })
 if err != nil {
  log.Printf("Couldn't delete user. Here's why: %v\n", err)
 }
 return err
```

```
}



// AdminCreateUser uses administrator credentials to add a user to a user pool.
 This method leaves the user
// in a state that requires they enter a new password next time they sign in.
func (actor CognitoActions) AdminCreateUser(userPoolId string, userName string,
 userEmail string) error {
 _, err := actor.CognitoClient.AdminCreateUser(context.TODO(),
 &cognitoidentityprovider.AdminCreateUserInput{
  UserPoolId:     aws.String(userPoolId),
  Username:       aws.String(userName),
  MessageAction:  types.MessageActionTypeSuppress,
  UserAttributes: []types.AttributeType{{Name: aws.String("email"), Value:
 aws.String(userEmail)}},
 })
 if err != nil {
  var userExists *types.UsernameExistsException
  if errors.As(err, &userExists) {
   log.Printf("User %v already exists in the user pool.", userName)
   err = nil
  } else {
   log.Printf("Couldn't create user %v. Here's why: %v\n", userName, err)
  }
 }
 return err
}



// AdminSetUserPassword uses administrator credentials to set a password for a
 user without requiring a
// temporary password.
func (actor CognitoActions) AdminSetUserPassword(userPoolId string, userName
 string, password string) error {
 _, err := actor.CognitoClient.AdminSetUserPassword(context.TODO(),
 &cognitoidentityprovider.AdminSetUserPasswordInput{
  Password:   aws.String(password),
  UserPoolId: aws.String(userPoolId),
  Username:   aws.String(userName),
  Permanent:  true,
 })
 if err != nil {
```

```
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't set password for user %v. Here's why: %v\n", userName,
 err)
  }
 }
 return err
}
```

建立包裝 DynamoDB 動作的結構。

```
// DynamoActions encapsulates the Amazon Simple Notification Service (Amazon SNS)
 actions
// used in the examples.
type DynamoActions struct {
 DynamoClient *dynamodb.Client
}

// User defines structured user data.
type User struct {
 UserName  string
 UserEmail string
 LastLogin *LoginInfo `dynamodbav:",omitempty"`
}

// LoginInfo defines structured custom login data.
type LoginInfo struct {
 UserPoolId string
 ClientId   string
 Time       string
}

// UserList defines a list of users.
type UserList struct {
 Users []User
}
```

```
// UserNameList returns the usernames contained in a UserList as a list of
 strings.
func (users *UserList) UserNameList() []string {
 names := make([]string, len(users.Users))
 for i := 0; i < len(users.Users); i++ {
  names[i] = users.Users[i].UserName
 }
 return names
}

// PopulateTable adds a set of test users to the table.
func (actor DynamoActions) PopulateTable(tableName string) error {
 var err error
 var item map[string]types.AttributeValue
 var writeReqs []types.WriteRequest
 for i := 1; i < 4; i++ {
  item, err = attributevalue.MarshalMap(User{UserName: fmt.Sprintf("test_user_
%v", i), UserEmail: fmt.Sprintf("test_email_%v@example.com", i)})
  if err != nil {
   log.Printf("Couldn't marshall user into DynamoDB format. Here's why: %v\n",
 err)
   return err
  }
  writeReqs = append(writeReqs, types.WriteRequest{PutRequest:
 &types.PutRequest{Item: item}})
 }
 _, err = actor.DynamoClient.BatchWriteItem(context.TODO(),
 &dynamodb.BatchWriteItemInput{
  RequestItems: map[string][]types.WriteRequest{tableName: writeReqs},
 })
 if err != nil {
  log.Printf("Couldn't populate table %v with users. Here's why: %v\n",
 tableName, err)
 }
 return err
}

// Scan scans the table for all items.
func (actor DynamoActions) Scan(tableName string) (UserList, error) {
 var userList UserList
 output, err := actor.DynamoClient.Scan(context.TODO(), &dynamodb.ScanInput{
  TableName: aws.String(tableName),
 })
 if err != nil {
```

```
  log.Printf("Couldn't scan table %v for items. Here's why: %v\n", tableName,
 err)
 } else {
  err = attributevalue.UnmarshalListOfMaps(output.Items, &userList.Users)
  if err != nil {
   log.Printf("Couldn't unmarshal items into users. Here's why: %v\n", err)
  }
 }
 return userList, err
}

// AddUser adds a user item to a table.
func (actor DynamoActions) AddUser(tableName string, user User) error {
 userItem, err := attributevalue.MarshalMap(user)
 if err != nil {
  log.Printf("Couldn't marshall user to item. Here's why: %v\n", err)
 }
 _, err = actor.DynamoClient.PutItem(context.TODO(), &dynamodb.PutItemInput{
  Item:      userItem,
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Couldn't put item in table %v. Here's why: %v", tableName, err)
 }
 return err
}
```

建立包裝 CloudWatch 記錄動作的結構。

```
type CloudWatchLogsActions struct {
 CwlClient *cloudwatchlogs.Client
}

// GetLatestLogStream gets the most recent log stream for a Lambda function.
func (actor CloudWatchLogsActions) GetLatestLogStream(functionName string)
 (types.LogStream, error) {
 var logStream types.LogStream
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.DescribeLogStreams(context.TODO(),
 &cloudwatchlogs.DescribeLogStreamsInput{
```

```
  Descending:   aws.Bool(true),
  Limit:        aws.Int32(1),
  LogGroupName: aws.String(logGroupName),
  OrderBy:      types.OrderByLastEventTime,
 })
 if err != nil {
  log.Printf("Couldn't get log streams for log group %v. Here's why: %v\n",
 logGroupName, err)
 } else {
  logStream = output.LogStreams[0]
 }
 return logStream, err
}

// GetLogEvents gets the most recent eventCount events from the specified log
 stream.
func (actor CloudWatchLogsActions) GetLogEvents(functionName string,
 logStreamName string, eventCount int32) (
 []types.OutputLogEvent, error) {
 var events []types.OutputLogEvent
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.GetLogEvents(context.TODO(),
 &cloudwatchlogs.GetLogEventsInput{
  LogStreamName: aws.String(logStreamName),
  Limit:         aws.Int32(eventCount),
  LogGroupName:  aws.String(logGroupName),
 })
 if err != nil {
  log.Printf("Couldn't get log event for log stream %v. Here's why: %v\n",
 logStreamName, err)
 } else {
  events = output.Events
 }
 return events, err
}
```

創建一個包裝 AWS CloudFormation 動作的結構。

```
// StackOutputs defines a map of outputs from a specific stack.
type StackOutputs map[string]string
```

```
type CloudFormationActions struct {
 CfnClient *cloudformation.Client
}

// GetOutputs gets the outputs from a CloudFormation stack and puts them into a
 structured format.
func (actor CloudFormationActions) GetOutputs(stackName string) StackOutputs {
 output, err := actor.CfnClient.DescribeStacks(context.TODO(),
 &cloudformation.DescribeStacksInput{
  StackName: aws.String(stackName),
 })
 if err != nil || len(output.Stacks) == 0 {
  log.Panicf("Couldn't find a CloudFormation stack named %v. Here's why: %v\n",
 stackName, err)
 }
 stackOutputs := StackOutputs{}
 for _, out := range output.Stacks[0].Outputs {
  stackOutputs[*out.OutputKey] = *out.OutputValue
 }
 return stackOutputs
}
```

清理資源。

```
// Resources keeps track of AWS resources created during an example and handles
// cleanup when the example finishes.
type Resources struct {
 userPoolId       string
 userAccessTokens []string
 triggers         []actions.Trigger

 cognitoActor *actions.CognitoActions
 questioner   demotools.IQuestioner
}

func (resources *Resources) init(cognitoActor *actions.CognitoActions, questioner
 demotools.IQuestioner) {
 resources.userAccessTokens = []string{}
 resources.triggers = []actions.Trigger{}
```

```
 resources.cognitoActor = cognitoActor
 resources.questioner = questioner
}

// Cleanup deletes all AWS resources created during an example.
func (resources *Resources) Cleanup() {
 defer func() {
  if r := recover(); r != nil {
   log.Printf("Something went wrong during cleanup.\n%v\n", r)
   log.Println("Use the AWS Management Console to remove any remaining resources
 \n" +
    "that were created for this scenario.")
  }
 }()

 wantDelete := resources.questioner.AskBool("Do you want to remove all of the AWS
 resources that were created "+
  "during this demo (y/n)?", "y")
 if wantDelete {
  for _, accessToken := range resources.userAccessTokens {
   err := resources.cognitoActor.DeleteUser(accessToken)
   if err != nil {
    log.Println("Couldn't delete user during cleanup.")
    panic(err)
   }
   log.Println("Deleted user.")
  }
  triggerList := make([]actions.TriggerInfo, len(resources.triggers))
  for i := 0; i < len(resources.triggers); i++ {
   triggerList[i] = actions.TriggerInfo{Trigger: resources.triggers[i],
 HandlerArn: nil}
  }
  err := resources.cognitoActor.UpdateTriggers(resources.userPoolId,
 triggerList...)
  if err != nil {
   log.Println("Couldn't update Cognito triggers during cleanup.")
   panic(err)
  }
  log.Println("Removed Cognito triggers from user pool.")
 } else {
  log.Println("Be sure to remove resources when you're done with them to avoid
 unexpected charges!")
 }
}
```

- 如需 API 詳細資訊，請參閱《AWS SDK for Go API 參考》中的下列主題。
  - AdminCreate使用者
  - AdminSetUserPassword
  - DeleteUser
  - InitiateAuth
  - UpdateUser游泳池

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使 AWS 用 SDK 的 Lambda 無伺服器範例

下列程式碼範例說明如何搭配 AWS 開發套件使用 Lambda。

範例

- 在 Lambda 函數中連接到 Amazon RDS 數據庫
- 使用 Kinesis 觸發條件調用 Lambda 函數
- 從 DynamoDB 觸發程序叫用 Lambda 函數
- 從 Amazon DocumentDB 觸發器調用 Lambda 函數
- 使用 Amazon S3 觸發條件調用 Lambda 函數
- 使用 Amazon SNS 觸發條件調用 Lambda 函數
- 使用 Amazon SQS 觸發條件調用 Lambda 函數
- 使用 Kinesis 觸發條件報告 Lambda 函數的批次項目失敗
- 使用 DynamoDB 觸發程序報告 Lambda 函數的批次項目失敗
- 使用 Amazon SQS 觸發條件報告 Lambda 函數的批次項目失敗

在 Lambda 函數中連接到 Amazon RDS 數據庫

下列程式碼範例會示範如何實作連線至 RDS 資料庫的 Lambda 函數。該函數提出了一個簡單的數據庫請求，並返回結果。

Go

SDK for Go V2

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 連線至 Lambda 函數中的 Amazon RDS 資料庫。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
/*
Golang v2 code here.
*/

package main

import (
 "context"
 "database/sql"
 "encoding/json"
 "fmt"

 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/feature/rds/auth"
 _ "github.com/go-sql-driver/mysql"
)

type MyEvent struct {
 Name string `json:"name"`
}

func HandleRequest(event *MyEvent) (map[string]interface{}, error) {

 var dbName string = "DatabaseName"
 var dbUser string = "DatabaseUser"
 var dbHost string = "mysqldb.123456789012.us-east-1.rds.amazonaws.com"
 var dbPort int = 3306
 var dbEndpoint string = fmt.Sprintf("%s:%d", dbHost, dbPort)
```

```
 var region string = "us-east-1"

 cfg, err := config.LoadDefaultConfig(context.TODO())
 if err != nil {
  panic("configuration error: " + err.Error())
 }

 authenticationToken, err := auth.BuildAuthToken(
  context.TODO(), dbEndpoint, region, dbUser, cfg.Credentials)
 if err != nil {
  panic("failed to create authentication token: " + err.Error())
 }

 dsn := fmt.Sprintf("%s:%s@tcp(%s)/%s?tls=true&allowCleartextPasswords=true",
  dbUser, authenticationToken, dbEndpoint, dbName,
 )

 db, err := sql.Open("mysql", dsn)
 if err != nil {
  panic(err)
 }

 defer db.Close()

 var sum int
 err = db.QueryRow("SELECT ?+? AS sum", 3, 2).Scan(&sum)
 if err != nil {
  panic(err)
 }
 s := fmt.Sprint(sum)
 message := fmt.Sprintf("The selected sum is: %s", s)

 messageBytes, err := json.Marshal(message)
 if err != nil {
  return nil, err
 }

 messageString := string(messageBytes)
 return map[string]interface{}{
  "statusCode": 200,
  "headers":    map[string]string{"Content-Type": "application/json"},
  "body":       messageString,
 }, nil
}
```

```
func main() {
 lambda.Start(HandleRequest)
}
```

JavaScript

適用於 JavaScript (v2) 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 JavaScript 連接到一個 Lambda 函數中的 Amazon RDS 數據庫。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
/*
Node.js code here.
*/
// ES6+ example
import { Signer } from "@aws-sdk/rds-signer";
import mysql from 'mysql2/promise';

async function createAuthToken() {
  // Define connection authentication parameters
  const dbinfo = {

    hostname: process.env.ProxyHostName,
    port: process.env.Port,
    username: process.env.DBUserName,
    region: process.env.AWS_REGION,

  }

  // Create RDS Signer object
  const signer = new Signer(dbinfo);

  // Request authorization token from RDS, specifying the username
```

```
  const token = await signer.getAuthToken();
  return token;
}

async function dbOps() {

  // Obtain auth token
  const token = await createAuthToken();
  // Define connection configuration
  let connectionConfig = {
    host: process.env.ProxyHostName,
    user: process.env.DBUserName,
    password: token,
    database: process.env.DBName,
    ssl: 'Amazon RDS'
  }
  // Create the connection to the DB
  const conn = await mysql.createConnection(connectionConfig);
  // Obtain the result of the query
  const [res,] = await conn.execute('select ?+? as sum', [3, 2]);
  return res;

}

export const handler = async (event) => {
  // Execute database flow
  const result = await dbOps();
  // Return result
  return {
    statusCode: 200,
    body: JSON.stringify("The selected sum is: " + result[0].sum)
  }
};
```

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使用 Kinesis 觸發條件調用 Lambda 函數

下列程式碼範例示範如何實作 Lambda 函數，以便接收在收到來自 Kinesis 串流的訊息時觸發的事件。此函數會擷取 Kinesis 承載、從 Base64 解碼，並記錄記錄內容。

.NET

AWS SDK for .NET

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 .NET 搭配 Lambda 來使用 Kinesis 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
using AWS.Lambda.Powertools.Logging;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace KinesisIntegrationSampleCode;

public class Function
{
    // Powertools Logger requires an environment variables against your function
    // POWERTOOLS_SERVICE_NAME
    [Logging(LogEvent = true)]
    public async Task FunctionHandler(KinesisEvent evnt, ILambdaContext context)
    {
        if (evnt.Records.Count == 0)
        {
            Logger.LogInformation("Empty Kinesis Event received");
            return;
        }

        foreach (var record in evnt.Records)
        {
            try
            {
```

```
                Logger.LogInformation($"Processed Event with EventId:
 {record.EventId}");
                string data = await GetRecordDataAsync(record.Kinesis, context);
                Logger.LogInformation($"Data: {data}");
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex)
            {
                Logger.LogError($"An error occurred {ex.Message}");
                throw;
            }
        }
        Logger.LogInformation($"Successfully processed {evnt.Records.Count}
 records.");
    }

    private async Task<string> GetRecordDataAsync(KinesisEvent.Record record,
 ILambdaContext context)
    {
        byte[] bytes = record.Data.ToArray();
        string data = Encoding.UTF8.GetString(bytes);
        await Task.CompletedTask; //Placeholder for actual async work
        return data;
    }
}
```

Go

SDK for Go V2

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 搭配 Lambda 來使用 Kinesis 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main
```

```
import (
 "context"
 "log"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, kinesisEvent events.KinesisEvent) error {
 if len(kinesisEvent.Records) == 0 {
  log.Printf("empty Kinesis event received")
  return nil
 }

 for _, record := range kinesisEvent.Records {
  log.Printf("processed Kinesis event with EventId: %v", record.EventID)
  recordDataBytes := record.Kinesis.Data
  recordDataText := string(recordDataBytes)
  log.Printf("record data: %v", recordDataText)
  // TODO: Do interesting work based on the new data
 }
 log.Printf("successfully processed %v records", len(kinesisEvent.Records))
 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

適用於 Java 2.x 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Java 搭配 Lambda 來使用 Kinesis 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;

public class Handler implements RequestHandler<KinesisEvent, Void> {
    @Override
    public Void handleRequest(final KinesisEvent event, final Context context) {
        LambdaLogger logger = context.getLogger();
        if (event.getRecords().isEmpty()) {
            logger.log("Empty Kinesis Event received");
            return null;
        }
        for (KinesisEvent.KinesisEventRecord record : event.getRecords()) {
            try {
                logger.log("Processed Event with EventId: "+record.getEventID());
                String data = new String(record.getKinesis().getData().array());
                logger.log("Data:"+ data);
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex) {
                logger.log("An error occurred:"+ex.getMessage());
                throw ex;
            }
        }
        logger.log("Successfully processed:"+event.getRecords().size()+"
 records");
        return null;
    }

}
```

JavaScript

適用於 JavaScript (v3) 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Lambda 使用 JavaScript的 Kinesis 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    try {
      console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      console.log(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      console.error(`An error occurred ${err}`);
      throw err;
    }
  }
  console.log(`Successfully processed ${event.Records.length} records.`);
};

async function getRecordDataAsync(payload) {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

使用 Lambda 使用 TypeScript的 Kinesis 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import {
  KinesisStreamEvent,
```

```
  Context,
  KinesisStreamHandler,
  KinesisStreamRecordPayload,
} from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
  serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (
  event: KinesisStreamEvent,
  context: Context
): Promise<void> => {
  for (const record of event.Records) {
    try {
      logger.info(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      logger.info(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      logger.error(`An error occurred ${err}`);
      throw err;
    }
    logger.info(`Successfully processed ${event.Records.length} records.`);
  }
};

async function getRecordDataAsync(
  payload: KinesisStreamRecordPayload
): Promise<string> {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

PHP

適用於 PHP 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 PHP 使用 Lambda 消耗 Kinesis 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kinesis\KinesisEvent;
use Bref\Event\Kinesis\KinesisHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends KinesisHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleKinesis(KinesisEvent $event, Context $context): void
    {
        $this->logger->info("Processing records");
        $records = $event->getRecords();
        foreach ($records as $record) {
            $data = $record->getData();
```

```
            $this->logger->info(json_encode($data));
            // TODO: Do interesting work based on the new data

            // Any exception thrown will be logged and the invocation will be
 marked as failed
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

適用於 Python (Boto3) 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Python 搭配 Lambda 來使用 Kinesis 事件。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
import base64
def lambda_handler(event, context):

    for record in event['Records']:
        try:
            print(f"Processed Kinesis Event - EventID: {record['eventID']}")
            record_data = base64.b64decode(record['kinesis']
['data']).decode('utf-8')
            print(f"Record Data: {record_data}")
            # TODO: Do interesting work based on the new data
        except Exception as e:
            print(f"An error occurred {e}")
            raise e
```

```
    print(f"Successfully processed {len(event['Records'])} records.")
```

Ruby

適用於 Ruby 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用紅寶石使用 Lambda 消耗 Kinesis 事件。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'aws-sdk'

def lambda_handler(event:, context:)
  event['Records'].each do |record|
    begin
      puts "Processed Kinesis Event - EventID: #{record['eventID']}"
      record_data = get_record_data_async(record['kinesis'])
      puts "Record Data: #{record_data}"
      # TODO: Do interesting work based on the new data
    rescue => err
      $stderr.puts "An error occurred #{err}"
      raise err
    end
  end
  puts "Successfully processed #{event['Records'].length} records."
end

def get_record_data_async(payload)
  data = Base64.decode64(payload['data']).force_encoding('UTF-8')
  # Placeholder for actual async work
  # You can use Ruby's asynchronous programming tools like async/await or fibers
 here.
  return data
end
```

Rust

適用於 Rust 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Rust 使用 Lambda 消耗 Kinesis 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::kinesis::KinesisEvent;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<KinesisEvent>) -> Result<(), Error>
 {
    if event.payload.records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }

    event.payload.records.iter().for_each(|record| {
        tracing::info!("EventId:
 {}",record.event_id.as_deref().unwrap_or_default());

        let record_data = std::str::from_utf8(&record.kinesis.data);

        match record_data {
            Ok(data) => {
                // log the record data
                tracing::info!("Data: {}", data);
            }
            Err(e) => {
                tracing::error!("Error: {}", e);
            }
        }
    });

    tracing::info!(
        "Successfully processed {} records",
```

```
        event.payload.records.len()
    );

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

從 DynamoDB 觸發程序叫用 Lambda 函數

下列程式碼範例示範如何實作 Lambda 函數，該函數會接收透過從 DynamoDB 串流接收記錄而觸發的事件。此函數會擷取 DynamoDB 承載並記錄記錄內容。

.NET

AWS SDK for .NET

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 .NET 與 Lambda 一起使用 DynamoDB 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
```

```
// SPDX-License-Identifier: Apache-2.0
using System.Text.Json;
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace AWSLambda_DDB;

public class Function
{
    public void FunctionHandler(DynamoDBEvent dynamoEvent, ILambdaContext
 context)
    {
        context.Logger.LogInformation($"Beginning to process
 {dynamoEvent.Records.Count} records...");

        foreach (var record in dynamoEvent.Records)
        {
            context.Logger.LogInformation($"Event ID: {record.EventID}");
            context.Logger.LogInformation($"Event Name: {record.EventName}");

            context.Logger.LogInformation(JsonSerializer.Serialize(record));
        }

        context.Logger.LogInformation("Stream processing complete.");
    }
}
```

Go

SDK for Go V2

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 使用與 Lambda 一起使用 DynamoDB 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-lambda-go/events"
 "fmt"
)

func HandleRequest(ctx context.Context, event events.DynamoDBEvent) (*string,
 error) {
 if len(event.Records) == 0 {
  return nil, fmt.Errorf("received empty event")
 }

 for _, record := range event.Records {
   LogDynamoDBRecord(record)
 }

 message := fmt.Sprintf("Records processed: %d", len(event.Records))
 return &message, nil
}

func main() {
 lambda.Start(HandleRequest)
}

func LogDynamoDBRecord(record events.DynamoDBEventRecord){
 fmt.Println(record.EventID)
```

```
 fmt.Println(record.EventName)
 fmt.Printf("%+v\n", record.Change)
}
```

Java

適用於 Java 2.x 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Java 與 Lambda 一起使用 DynamoDB 事件。

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import
 com.amazonaws.services.lambda.runtime.events.DynamodbEvent.DynamodbStreamRecord;
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class example implements RequestHandler<DynamodbEvent, Void> {

    private static final Gson GSON = new
 GsonBuilder().setPrettyPrinting().create();

    @Override
    public Void handleRequest(DynamodbEvent event, Context context) {
        System.out.println(GSON.toJson(event));
        event.getRecords().forEach(this::logDynamoDBRecord);
        return null;
    }

    private void logDynamoDBRecord(DynamodbStreamRecord record) {
        System.out.println(record.getEventID());
        System.out.println(record.getEventName());
        System.out.println("DynamoDB Record: " +
 GSON.toJson(record.getDynamodb()));
    }
```

```
}
```

JavaScript

適用於 JavaScript (v3) 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用使 Lambda. JavaScript

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(record => {
        logDynamoDBRecord(record);
    });
};

const logDynamoDBRecord = (record) => {
    console.log(record.eventID);
    console.log(record.eventName);
    console.log(`DynamoDB Record: ${JSON.stringify(record.dynamodb)}`);
};
```

使用使 Lambda. TypeScript

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
export const handler = async (event, context) => {
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(record => {
        logDynamoDBRecord(record);
    });
}
const logDynamoDBRecord = (record) => {
```

```
    console.log(record.eventID);
    console.log(record.eventName);
    console.log(`DynamoDB Record: ${JSON.stringify(record.dynamodb)}`);
};
```

PHP

適用於 PHP 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 PHP 與 Lambda 一起使用 DynamoDB 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\DynamoDb\DynamoDbEvent;
use Bref\Event\DynamoDb\DynamoDbHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends DynamoDbHandler
{
    private StderrLogger $logger;

    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
```

```
     */
    public function handleDynamoDb(DynamoDbEvent $event, Context $context): void
    {
        $this->logger->info("Processing DynamoDb table items");
        $records = $event->getRecords();

        foreach ($records as $record) {
            $eventName = $record->getEventName();
            $keys = $record->getKeys();
            $old = $record->getOldImage();
            $new = $record->getNewImage();

            $this->logger->info("Event Name:".$eventName."\n");
            $this->logger->info("Keys:". json_encode($keys)."\n");
            $this->logger->info("Old Image:". json_encode($old)."\n");
            $this->logger->info("New Image:". json_encode($new));

            // TODO: Do interesting work based on the new data

            // Any exception thrown will be logged and the invocation will be
 marked as failed
        }

        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords items");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

適用於 Python (Boto3) 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用使用 Python 與 Lambda 一起使用動 DynamoDB 事件。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0

import json

def lambda_handler(event, context):
    print(json.dumps(event, indent=2))

    for record in event['Records']:
        log_dynamodb_record(record)

def log_dynamodb_record(record):
    print(record['eventID'])
    print(record['eventName'])
    print(f"DynamoDB Record: {json.dumps(record['dynamodb'])}")
```

Ruby

適用於 Ruby 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用紅寶石與 Lambda 一起使用 DynamoDB 事件。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0

def lambda_handler(event:, context:)
    return 'received empty event' if event['Records'].empty?

    event['Records'].each do |record|
      log_dynamodb_record(record)
    end

    "Records processed: #{event['Records'].length}"
  end
```

```
  def log_dynamodb_record(record)
    puts record['eventID']
    puts record['eventName']
    puts "DynamoDB Record: #{JSON.generate(record['dynamodb'])}"
  end
```

Rust

適用於 Rust 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Rust 與 Lambda 一起使用 DynamoDB 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

use lambda_runtime::{service_fn, tracing, Error, LambdaEvent};
use aws_lambda_events::{
    event::dynamodb::{Event, EventRecord},
   };


// Built with the following dependencies:
//lambda_runtime = "0.11.1"
//serde_json = "1.0"
//tokio = { version = "1", features = ["macros"] }
//tracing = { version = "0.1", features = ["log"] }
//tracing-subscriber = { version = "0.3", default-features = false, features =
 ["fmt"] }
//aws_lambda_events = "0.15.0"

async fn function_handler(event: LambdaEvent<Event>) ->Result<(), Error> {

    let records = &event.payload.records;
    tracing::info!("event payload: {:?}",records);
```

```
    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }

    for record in records{
        log_dynamo_dbrecord(record);
    }

    tracing::info!("Dynamo db records processed");

    // Prepare the response
    Ok(())

}

fn log_dynamo_dbrecord(record: &EventRecord)-> Result<(), Error>{
    tracing::info!("EventId: {}", record.event_id);
    tracing::info!("EventName: {}", record.event_name);
    tracing::info!("DynamoDB Record: {:?}", record.change );
    Ok(())

}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
    .with_max_level(tracing::Level::INFO)
    .with_target(false)
    .without_time()
    .init();

    let func = service_fn(function_handler);
    lambda_runtime::run(func).await?;
    Ok(())

}
```

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

從 Amazon DocumentDB 觸發器調用 Lambda 函數

下列程式碼範例會示範如何實作 Lambda 函數，此函數會接收由 DocumentDB 變更串流接收記錄所觸發的事件。該函數檢索 DocumentDB 有效載荷和記錄的內容。

Go

SDK for Go V2

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用圍棋使用 Lambda 使用 Amazon DocumentDB 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

package main

import (
 "context"
 "encoding/json"
 "fmt"

 "github.com/aws/aws-lambda-go/lambda"
)

type Event struct {
 Events []Record `json:"events"`
}

type Record struct {
 Event struct {
  OperationType string `json:"operationType"`
  NS            struct {
   DB   string `json:"db"`
   Coll string `json:"coll"`
  } `json:"ns"`
  FullDocument interface{} `json:"fullDocument"`
```

```
 } `json:"event"`
}

func main() {
 lambda.Start(handler)
}

func handler(ctx context.Context, event Event) (string, error) {
 fmt.Println("Loading function")
 for _, record := range event.Events {
  logDocumentDBEvent(record)
 }

 return "OK", nil
}

func logDocumentDBEvent(record Record) {
 fmt.Printf("Operation type: %s\n", record.Event.OperationType)
 fmt.Printf("db: %s\n", record.Event.NS.DB)
 fmt.Printf("collection: %s\n", record.Event.NS.Coll)
 docBytes, _ := json.MarshalIndent(record.Event.FullDocument, "", "  ")
 fmt.Printf("Full document: %s\n", string(docBytes))
}
```

JavaScript

適用於 JavaScript (v3) 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Amazon DocumentDB 事件 Lambda 使用. JavaScript

```
console.log('Loading function');
exports.handler = async (event, context) => {
    event.events.forEach(record => {
        logDocumentDBEvent(record);
    });
```

```
    return 'OK';
};

const logDocumentDBEvent = (record) => {
    console.log('Operation type: ' + record.event.operationType);
    console.log('db: ' + record.event.ns.db);
    console.log('collection: ' + record.event.ns.coll);
    console.log('Full document:', JSON.stringify(record.event.fullDocument, null,
 2));
};
```

Python

適用於 Python (Boto3) 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Python 使用 Lambda 消費 Amazon DocumentDB 事件。

```
import json

def lambda_handler(event, context):
    for record in event.get('events', []):
        log_document_db_event(record)
    return 'OK'

def log_document_db_event(record):
    event_data = record.get('event', {})
    operation_type = event_data.get('operationType', 'Unknown')
    db = event_data.get('ns', {}).get('db', 'Unknown')
    collection = event_data.get('ns', {}).get('coll', 'Unknown')
    full_document = event_data.get('fullDocument', {})

    print(f"Operation type: {operation_type}")
    print(f"db: {db}")
    print(f"collection: {collection}")
```

```
    print("Full document:", json.dumps(full_document, indent=2))
```

Ruby

適用於 Ruby 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用紅寶石使用 Lambda 消費 Amazon DocumentDB 事件。

```
require 'json'

def lambda_handler(event:, context:)
  event['events'].each do |record|
    log_document_db_event(record)
  end
  'OK'
end

def log_document_db_event(record)
  event_data = record['event'] || {}
  operation_type = event_data['operationType'] || 'Unknown'
  db = event_data.dig('ns', 'db') || 'Unknown'
  collection = event_data.dig('ns', 'coll') || 'Unknown'
  full_document = event_data['fullDocument'] || {}

  puts "Operation type: #{operation_type}"
  puts "db: #{db}"
  puts "collection: #{collection}"
  puts "Full document: #{JSON.pretty_generate(full_document)}"
end
```

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使用 Amazon S3 觸發條件調用 Lambda 函數

下列程式碼範例顯示如何實作 Lambda 函數來接收上傳物件至 S3 儲存貯體時觸發的事件。此函數會從事件參數擷取 S3 儲存貯體名稱和物件金鑰，並呼叫 Amazon S3 API 以擷取和記錄物件的內容類型。

.NET

AWS SDK for .NET

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 .NET 搭配 Lambda 來使用 S3 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Threading.Tasks;
using Amazon.Lambda.Core;
using Amazon.S3;
using System;
using Amazon.Lambda.S3Events;
using System.Web;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace S3Integration
{
    public class Function
    {
        private static AmazonS3Client _s3Client;
        public Function() : this(null)
        {
        }

        internal Function(AmazonS3Client s3Client)
        {
```

```
            _s3Client = s3Client ?? new AmazonS3Client();
        }

        public async Task<string> Handler(S3Event evt, ILambdaContext context)
        {
            try
            {
                if (evt.Records.Count <= 0)
                {
                    context.Logger.LogLine("Empty S3 Event received");
                    return string.Empty;
                }

                var bucket = evt.Records[0].S3.Bucket.Name;
                var key = HttpUtility.UrlDecode(evt.Records[0].S3.Object.Key);

                context.Logger.LogLine($"Request is for {bucket} and {key}");

                var objectResult = await _s3Client.GetObjectAsync(bucket, key);

                context.Logger.LogLine($"Returning {objectResult.Key}");

                return objectResult.Key;
            }
            catch (Exception e)
            {
                context.Logger.LogLine($"Error processing request -
 {e.Message}");

                return string.Empty;
            }
        }
    }
}
```

Go

SDK for Go V2

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 搭配 Lambda 來使用 S3 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "log"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/service/s3"
)

func handler(ctx context.Context, s3Event events.S3Event) error {
 sdkConfig, err := config.LoadDefaultConfig(ctx)
 if err != nil {
  log.Printf("failed to load default config: %s", err)
  return err
 }
 s3Client := s3.NewFromConfig(sdkConfig)

 for _, record := range s3Event.Records {
  bucket := record.S3.Bucket.Name
  key := record.S3.Object.URLDecodedKey
  headOutput, err := s3Client.HeadObject(ctx, &s3.HeadObjectInput{
   Bucket: &bucket,
   Key:    &key,
  })
  if err != nil {
   log.Printf("error getting head of object %s/%s: %s", bucket, key, err)
```

```
   return err
  }
  log.Printf("successfully retrieved %s/%s of type %s", bucket, key,
 *headOutput.ContentType)
 }

 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

適用於 Java 2.x 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Java 搭配 Lambda 來使用 S3 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import software.amazon.awssdk.services.s3.model.HeadObjectRequest;
import software.amazon.awssdk.services.s3.model.HeadObjectResponse;
import software.amazon.awssdk.services.s3.S3Client;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.S3Event;
import
 com.amazonaws.services.lambda.runtime.events.models.s3.S3EventNotification.S3EventNotifi

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

```
public class Handler implements RequestHandler<S3Event, String> {
    private static final Logger logger = LoggerFactory.getLogger(Handler.class);
    @Override
    public String handleRequest(S3Event s3event, Context context) {
        try {
          S3EventNotificationRecord record = s3event.getRecords().get(0);
          String srcBucket = record.getS3().getBucket().getName();
          String srcKey = record.getS3().getObject().getUrlDecodedKey();

          S3Client s3Client = S3Client.builder().build();
          HeadObjectResponse headObject = getHeadObject(s3Client, srcBucket,
 srcKey);

          logger.info("Successfully retrieved " + srcBucket + "/" + srcKey + " of
 type " + headObject.contentType());

          return "Ok";
        } catch (Exception e) {
          throw new RuntimeException(e);
        }
    }

    private HeadObjectResponse getHeadObject(S3Client s3Client, String bucket,
 String key) {
        HeadObjectRequest headObjectRequest = HeadObjectRequest.builder()
                .bucket(bucket)
                .key(key)
                .build();
        return s3Client.headObject(headObjectRequest);
    }
}
```

JavaScript

適用於 JavaScript (v3) 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用使 Lambda JavaScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { S3Client, HeadObjectCommand } from "@aws-sdk/client-s3";

const client = new S3Client();

exports.handler = async (event, context) => {

    // Get the object from the event and show its content type
    const bucket = event.Records[0].s3.bucket.name;
    const key = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/g,
 ' '));

    try {
        const { ContentType } = await client.send(new HeadObjectCommand({
            Bucket: bucket,
            Key: key,
        }));

        console.log('CONTENT TYPE:', ContentType);
        return ContentType;

    } catch (err) {
        console.log(err);
        const message = `Error getting object ${key} from bucket ${bucket}. Make
 sure they exist and your bucket is in the same region as this function.`;
        console.log(message);
        throw new Error(message);
    }
};
```

使用使 Lambda TypeScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { S3Event } from 'aws-lambda';
import { S3Client, HeadObjectCommand } from '@aws-sdk/client-s3';

const s3 = new S3Client({ region: process.env.AWS_REGION });
```

```
export const handler = async (event: S3Event): Promise<string | undefined> => {
  // Get the object from the event and show its content type
  const bucket = event.Records[0].s3.bucket.name;
  const key = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/g, '
 '));
  const params = {
    Bucket: bucket,
    Key: key,
  };
  try {
    const { ContentType } = await s3.send(new HeadObjectCommand(params));
    console.log('CONTENT TYPE:', ContentType);
    return ContentType;
  } catch (err) {
    console.log(err);
    const message = `Error getting object ${key} from bucket ${bucket}. Make sure
 they exist and your bucket is in the same region as this function.`;
    console.log(message);
    throw new Error(message);
  }
};
```

PHP

適用於 PHP 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 PHP 使用 Lambda 使用 S3 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

use Bref\Context\Context;
use Bref\Event\S3\S3Event;
use Bref\Event\S3\S3Handler;
use Bref\Logger\StderrLogger;
```

```
require __DIR__ . '/vendor/autoload.php';


class Handler extends S3Handler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    public function handleS3(S3Event $event, Context $context) : void
    {
        $this->logger->info("Processing S3 records");

        // Get the object from the event and show its content type
        $records = $event->getRecords();

        foreach ($records as $record)
        {
            $bucket = $record->getBucket()->getName();
            $key = urldecode($record->getObject()->getKey());

            try {
                $fileSize = urldecode($record->getObject()->getSize());
                echo "File Size: " . $fileSize . "\n";
                // TODO: Implement your custom processing logic here
            } catch (Exception $e) {
                echo $e->getMessage() . "\n";
                echo 'Error getting object ' . $key . ' from bucket ' .
 $bucket . '. Make sure they exist and your bucket is in the same region as this
 function.' . "\n";
                throw $e;
            }
        }
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

適用於 Python (Boto3) 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Python 搭配 Lambda 來使用 S3 事件。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
import json
import urllib.parse
import boto3

print('Loading function')

s3 = boto3.client('s3')


def lambda_handler(event, context):
    #print("Received event: " + json.dumps(event, indent=2))

    # Get the object from the event and show its content type
    bucket = event['Records'][0]['s3']['bucket']['name']
    key = urllib.parse.unquote_plus(event['Records'][0]['s3']['object']['key'],
 encoding='utf-8')
    try:
        response = s3.get_object(Bucket=bucket, Key=key)
        print("CONTENT TYPE: " + response['ContentType'])
        return response['ContentType']
    except Exception as e:
        print(e)
        print('Error getting object {} from bucket {}. Make sure they exist and
 your bucket is in the same region as this function.'.format(key, bucket))
        raise e
```

Ruby

適用於 Ruby 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用紅寶石與 Lambda 一個 S3 事件。

```
require 'json'
require 'uri'
require 'aws-sdk'

puts 'Loading function'

def lambda_handler(event:, context:)
  s3 = Aws::S3::Client.new(region: 'region') # Your AWS region
  # puts "Received event: #{JSON.dump(event)}"

  # Get the object from the event and show its content type
  bucket = event['Records'][0]['s3']['bucket']['name']
  key = URI.decode_www_form_component(event['Records'][0]['s3']['object']['key'],
 Encoding::UTF_8)
  begin
    response = s3.get_object(bucket: bucket, key: key)
    puts "CONTENT TYPE: #{response.content_type}"
    return response.content_type
  rescue StandardError => e
    puts e.message
    puts "Error getting object #{key} from bucket #{bucket}. Make sure they exist
 and your bucket is in the same region as this function."
    raise e
  end
end
```

Rust

適用於 Rust 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Rust 搭配 Lambda 來使用 S3 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::s3::S3Event;
use aws_sdk_s3::{Client};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};


/// Main function
#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        .with_target(false)
        .without_time()
        .init();

    // Initialize the AWS SDK for Rust
    let config = aws_config::load_from_env().await;
    let s3_client = Client::new(&config);

    let res = run(service_fn(|request: LambdaEvent<S3Event>| {
        function_handler(&s3_client, request)
    })).await;

    res
}

async fn function_handler(
    s3_client: &Client,
    evt: LambdaEvent<S3Event>
) -> Result<(), Error> {
```

```
    tracing::info!(records = ?evt.payload.records.len(), "Received request from
 SQS");

    if evt.payload.records.len() == 0 {
        tracing::info!("Empty S3 event received");
    }

    let bucket = evt.payload.records[0].s3.bucket.name.as_ref().expect("Bucket
 name to exist");
    let key = evt.payload.records[0].s3.object.key.as_ref().expect("Object key to
 exist");

    tracing::info!("Request is for {} and object {}", bucket, key);

    let s3_get_object_result = s3_client
        .get_object()
        .bucket(bucket)
        .key(key)
        .send()
        .await;

    match s3_get_object_result {
        Ok(_) => tracing::info!("S3 Get Object success, the s3GetObjectResult
 contains a 'body' property of type ByteStream"),
        Err(_) => tracing::info!("Failure with S3 Get Object request")
    }

    Ok(())
}
```

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使用 Amazon SNS 觸發條件調用 Lambda 函數

下列程式碼範例顯示如何實作 Lambda 函數，以便接收在收到來自 SNS 主題的訊息時觸發的事件。函數會從事件參數擷取訊息，並記錄每一則訊息的內容。

.NET

AWS SDK for .NET

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 .NET 搭配 Lambda 來使用 SNS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SNSEvents;


// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace SnsIntegration;

public class Function
{
    public async Task FunctionHandler(SNSEvent evnt, ILambdaContext context)
    {
        foreach (var record in evnt.Records)
        {
            await ProcessRecordAsync(record, context);
        }
        context.Logger.LogInformation("done");
    }

    private async Task ProcessRecordAsync(SNSEvent.SNSRecord record,
 ILambdaContext context)
    {
        try
        {
            context.Logger.LogInformation($"Processed record
 {record.Sns.Message}");
```

```
            // TODO: Do interesting work based on the new message
            await Task.CompletedTask;
        }
        catch (Exception e)
        {
            //You can use Dead Letter Queue to handle failures. By configuring a
 Lambda DLQ.
            context.Logger.LogError($"An error occurred");
            throw;
        }
    }
}
```

Go

SDK for Go V2

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 搭配 Lambda 來使用 SNS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "fmt"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, snsEvent events.SNSEvent) {
 for _, record := range snsEvent.Records {
  processMessage(record)
 }
```

```
 fmt.Println("done")
}

func processMessage(record events.SNSEventRecord) {
 message := record.SNS.Message
 fmt.Printf("Processed message: %s\n", message)
 // TODO: Process your record here
}

func main() {
 lambda.Start(handler)
}
```

Java

適用於 Java 2.x 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Java 與 Lambda 一起使用 SNS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SNSEvent;
import com.amazonaws.services.lambda.runtime.events.SNSEvent.SNSRecord;


import java.util.Iterator;
import java.util.List;

public class SNSEventHandler implements RequestHandler<SNSEvent, Boolean> {
    LambdaLogger logger;
```

```
    @Override
    public Boolean handleRequest(SNSEvent event, Context context) {
        logger = context.getLogger();
        List<SNSRecord> records = event.getRecords();
        if (!records.isEmpty()) {
            Iterator<SNSRecord> recordsIter = records.iterator();
            while (recordsIter.hasNext()) {
                processRecord(recordsIter.next());
            }
        }
        return Boolean.TRUE;
    }

    public void processRecord(SNSRecord record) {
        try {
            String message = record.getSNS().getMessage();
            logger.log("message: " + message);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

}
```

JavaScript

適用於 JavaScript (v3) 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

透過使 Lambda JavaScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
```

```
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    await processMessageAsync(record);
  }
  console.info("done");
};

async function processMessageAsync(record) {
  try {
    const message = JSON.stringify(record.Sns.Message);
    console.log(`Processed message ${message}`);
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

透過使 Lambda TypeScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SNSEvent, Context, SNSHandler, SNSEventRecord } from "aws-lambda";

export const functionHandler: SNSHandler = async (
  event: SNSEvent,
  context: Context
): Promise<void> => {
  for (const record of event.Records) {
    await processMessageAsync(record);
  }
  console.info("done");
};

async function processMessageAsync(record: SNSEventRecord): Promise<any> {
  try {
    const message: string = JSON.stringify(record.Sns.Message);
    console.log(`Processed message ${message}`);
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
```

```
    throw err;
  }
}
```

PHP

適用於 PHP 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 PHP 搭配 Lambda 來使用 SNS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

/*
Since native PHP support for AWS Lambda is not available, we are utilizing Bref's
 PHP functions runtime for AWS Lambda.
For more information on Bref's PHP runtime for Lambda, refer to: https://bref.sh/
docs/runtimes/function

Another approach would be to create a custom runtime.
A practical example can be found here: https://aws.amazon.com/blogs/apn/aws-
lambda-custom-runtime-for-php-a-practical-example/
*/

// Additional composer packages may be required when using Bref or any other PHP
 functions runtime.
// require __DIR__ . '/vendor/autoload.php';

use Bref\Context\Context;
use Bref\Event\Sns\SnsEvent;
use Bref\Event\Sns\SnsHandler;

class Handler extends SnsHandler
{
```

```
    public function handleSns(SnsEvent $event, Context $context): void
    {
        foreach ($event->getRecords() as $record) {
            $message = $record->getMessage();

            // TODO: Implement your custom processing logic here
            // Any exception thrown will be logged and the invocation will be
 marked as failed

            echo "Processed Message: $message" . PHP_EOL;
        }
    }
}

return new Handler();
```

Python

適用於 Python (Boto3) 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Python 搭配 Lambda 來使用 SNS 事件。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event, context):
    for record in event['Records']:
        process_message(record)
    print("done")

def process_message(record):
    try:
        message = record['Sns']['Message']
        print(f"Processed message {message}")
        # TODO; Process your record here
```

```
    except Exception as e:
        print("An error occurred")
        raise e
```

Ruby

適用於 Ruby 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用紅寶石使用與 Lambda 的 SNS 事件。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event:, context:)
  event['Records'].map { |record| process_message(record) }
end

def process_message(record)
  message = record['Sns']['Message']
  puts("Processing message: #{message}")
rescue StandardError => e
  puts("Error processing message: #{e}")
  raise
end
```

Rust

適用於 Rust 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Rust 搭配 Lambda 來使用 SNS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::sns::SnsEvent;
use aws_lambda_events::sns::SnsRecord;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};
use tracing::info;

// Built with the following dependencies:
//  aws_lambda_events = { version = "0.10.0", default-features = false, features
 = ["sns"] }
//  lambda_runtime = "0.8.1"
//  tokio = { version = "1", features = ["macros"] }
//  tracing = { version = "0.1", features = ["log"] }
//  tracing-subscriber = { version = "0.3", default-features = false, features =
 ["fmt"] }

async fn function_handler(event: LambdaEvent<SnsEvent>) -> Result<(), Error> {
    for event in event.payload.records {
        process_record(&event)?;
    }

    Ok(())
}

fn process_record(record: &SnsRecord) -> Result<(), Error> {
    info!("Processing SNS Message: {}", record.sns.message);

    // Implement your record handling code here.

    Ok(())
}
```

```
#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        .with_target(false)
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使用 Amazon SQS 觸發條件調用 Lambda 函數

下列程式碼範例示範如何實作 Lambda 函數，以便接收在收到來自 SQS 佇列的訊息時觸發的事件。函數會從事件參數擷取訊息，並記錄每一則訊息的內容。

.NET

AWS SDK for .NET

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 .NET 搭配 Lambda 來使用 SQS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SQSEvents;


// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
```

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace SqsIntegrationSampleCode
{
    public async Task FunctionHandler(SQSEvent evnt, ILambdaContext context)
    {
        foreach (var message in evnt.Records)
        {
            await ProcessMessageAsync(message, context);
        }

        context.Logger.LogInformation("done");
    }

    private async Task ProcessMessageAsync(SQSEvent.SQSMessage message,
 ILambdaContext context)
    {
        try
        {
            context.Logger.LogInformation($"Processed message {message.Body}");

            // TODO: Do interesting work based on the new message
            await Task.CompletedTask;
        }
        catch (Exception e)
        {
            //You can use Dead Letter Queue to handle failures. By configuring a
 Lambda DLQ.
            context.Logger.LogError($"An error occurred");
            throw;
        }

    }
}
```

Go

SDK for Go V2

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 搭配 Lambda 來使用 SQS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package integration_sqs_to_lambda

import (
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(event events.SQSEvent) error {
 for _, record := range event.Records {
  err := processMessage(record)
  if err != nil {
   return err
  }
 }
 fmt.Println("done")
 return nil
}

func processMessage(record events.SQSMessage) error {
 fmt.Printf("Processed message %s\n", record.Body)
 // TODO: Do interesting work based on the new message
 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

適用於 Java 2.x 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Java 搭配 Lambda 來使用 SQS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SQSEvent;
import com.amazonaws.services.lambda.runtime.events.SQSEvent.SQSMessage;

public class Function implements RequestHandler<SQSEvent, Void> {
    @Override
    public Void handleRequest(SQSEvent sqsEvent, Context context) {
        for (SQSMessage msg : sqsEvent.getRecords()) {
            processMessage(msg, context);
        }
        context.getLogger().log("done");
        return null;
    }

    private void processMessage(SQSMessage msg, Context context) {
        try {
            context.getLogger().log("Processed message " + msg.getBody());

            // TODO: Do interesting work based on the new message

        } catch (Exception e) {
            context.getLogger().log("An error occurred");
            throw e;
        }
```

```
    }
}
```

JavaScript

適用於 JavaScript (v3) 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 SQS 事件與 Lambda 用 JavaScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const message of event.Records) {
    await processMessageAsync(message);
  }
  console.info("done");
};

async function processMessageAsync(message) {
  try {
    console.log(`Processed message ${message.body}`);
    // TODO: Do interesting work based on the new message
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

使用 SQS 事件與 Lambda 用 TypeScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SQSEvent, Context, SQSHandler, SQSRecord } from "aws-lambda";
```

```
export const functionHandler: SQSHandler = async (
  event: SQSEvent,
  context: Context
): Promise<void> => {
  for (const message of event.Records) {
    await processMessageAsync(message);
  }
  console.info("done");
};

async function processMessageAsync(message: SQSRecord): Promise<any> {
  try {
    console.log(`Processed message ${message.body}`);
    // TODO: Do interesting work based on the new message
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

PHP

適用於 PHP 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 PHP 與 Lambda 一起使用 SQS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
```

```
use Bref\Event\InvalidLambdaEvent;
use Bref\Event\Sqs\SqsEvent;
use Bref\Event\Sqs\SqsHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends SqsHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws InvalidLambdaEvent
     */
    public function handleSqs(SqsEvent $event, Context $context): void
    {
        foreach ($event->getRecords() as $record) {
            $body = $record->getBody();
            // TODO: Do interesting work based on the new message
        }
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

適用於 Python (Boto3) 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Python 搭配 Lambda 來使用 SQS 事件。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event, context):
    for message in event['Records']:
        process_message(message)
    print("done")

def process_message(message):
    try:
        print(f"Processed message {message['body']}")
        # TODO: Do interesting work based on the new message
    except Exception as err:
        print("An error occurred")
        raise err
```

Ruby

適用於 Ruby 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Ruby 搭配 Lambda 來使用 SQS 事件。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event:, context:)
  event['Records'].each do |message|
    process_message(message)
  end
  puts "done"
end

def process_message(message)
  begin
    puts "Processed message #{message['body']}"
    # TODO: Do interesting work based on the new message
```

```
  rescue StandardError => err
    puts "An error occurred"
    raise err
  end
end
```

Rust

適用於 Rust 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Rust 與 Lambda 一起使用 SQS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::sqs::SqsEvent;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<SqsEvent>) -> Result<(), Error> {
    event.payload.records.iter().for_each(|record| {
        // process the record
        tracing::info!("Message body: {}",
 record.body.as_deref().unwrap_or_default())
    });

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
```

```
        .init();

    run(service_fn(function_handler)).await
}
```

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使用 Kinesis 觸發條件報告 Lambda 函數的批次項目失敗

下列程式碼範例示範如何針對接收來自 Kinesis 串流之事件的 Lambda 函數，實作部分批次回應。此函數會在回應中報告批次項目失敗，指示 Lambda 稍後重試這些訊息。

.NET

AWS SDK for .NET

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 .NET 搭配 Lambda 報告 Kinesis 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text;
using System.Text.Json.Serialization;
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
using AWS.Lambda.Powertools.Logging;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace KinesisIntegration;

public class Function
```

```
{
    // Powertools Logger requires an environment variables against your function
    // POWERTOOLS_SERVICE_NAME
    [Logging(LogEvent = true)]
    public async Task<StreamsEventResponse> FunctionHandler(KinesisEvent evnt,
 ILambdaContext context)
    {
        if (evnt.Records.Count == 0)
        {
            Logger.LogInformation("Empty Kinesis Event received");
            return new StreamsEventResponse();
        }

        foreach (var record in evnt.Records)
        {
            try
            {
                Logger.LogInformation($"Processed Event with EventId:
 {record.EventId}");
                string data = await GetRecordDataAsync(record.Kinesis, context);
                Logger.LogInformation($"Data: {data}");
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex)
            {
                Logger.LogError($"An error occurred {ex.Message}");
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                return new StreamsEventResponse
                {
                    BatchItemFailures = new
 List<StreamsEventResponse.BatchItemFailure>
                    {
                        new StreamsEventResponse.BatchItemFailure
 { ItemIdentifier = record.Kinesis.SequenceNumber }
                    }
                };
            }
        }
        Logger.LogInformation($"Successfully processed {evnt.Records.Count}
 records.");
        return new StreamsEventResponse();
```

```
    }

    private async Task<string> GetRecordDataAsync(KinesisEvent.Record record,
 ILambdaContext context)
    {
        byte[] bytes = record.Data.ToArray();
        string data = Encoding.UTF8.GetString(bytes);
        await Task.CompletedTask; //Placeholder for actual async work
        return data;
    }
}

public class StreamsEventResponse
{
    [JsonPropertyName("batchItemFailures")]
    public IList<BatchItemFailure> BatchItemFailures { get; set; }
    public class BatchItemFailure
    {
        [JsonPropertyName("itemIdentifier")]
        public string ItemIdentifier { get; set; }
    }
}
```

Go

SDK for Go V2

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 使用 Lambda 報告 Kinesis 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "fmt"
```

```
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, kinesisEvent events.KinesisEvent)
 (map[string]interface{}, error) {
 batchItemFailures := []map[string]interface{}{}

 for _, record := range kinesisEvent.Records {
  curRecordSequenceNumber := ""

  // Process your record
  if /* Your record processing condition here */ {
   curRecordSequenceNumber = record.Kinesis.SequenceNumber
  }

  // Add a condition to check if the record processing failed
  if curRecordSequenceNumber != "" {
   batchItemFailures = append(batchItemFailures, map[string]interface{}
{"itemIdentifier": curRecordSequenceNumber})
  }
 }

 kinesisBatchResponse := map[string]interface{}{
  "batchItemFailures": batchItemFailures,
 }
 return kinesisBatchResponse, nil
}

func main() {
 lambda.Start(handler)
}
```

Java

適用於 Java 2.x 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

透過使用 Java 的 Lambda 報告 Kinesis 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

public class ProcessKinesisRecords implements RequestHandler<KinesisEvent,
 StreamsEventResponse> {

    @Override
    public StreamsEventResponse handleRequest(KinesisEvent input, Context
 context) {

        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<>();
        String curRecordSequenceNumber = "";

        for (KinesisEvent.KinesisEventRecord kinesisEventRecord :
 input.getRecords()) {
            try {
                //Process your record
                KinesisEvent.Record kinesisRecord =
 kinesisEventRecord.getKinesis();
                curRecordSequenceNumber = kinesisRecord.getSequenceNumber();

            } catch (Exception e) {
```

```
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                batchItemFailures.add(new
 StreamsEventResponse.BatchItemFailure(curRecordSequenceNumber));
                return new StreamsEventResponse(batchItemFailures);
            }
        }

       return new StreamsEventResponse(batchItemFailures);
    }
}
```

JavaScript

適用於 JavaScript (v3) 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Javascript 搭配 Lambda 報告 Kinesis 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    try {
      console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      console.log(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      console.error(`An error occurred ${err}`);
      /* Since we are working with streams, we can return the failed item
 immediately.
           Lambda will immediately begin to retry processing from this failed
 item onwards. */
```

```
      return {
        batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
      };
    }
  }
  console.log(`Successfully processed ${event.Records.length} records.`);
  return { batchItemFailures: [] };
};

async function getRecordDataAsync(payload) {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

使用 Lambda 使用 TypeScript報告 Kinesis 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import {
  KinesisStreamEvent,
  Context,
  KinesisStreamHandler,
  KinesisStreamRecordPayload,
  KinesisStreamBatchResponse,
} from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
  serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (
  event: KinesisStreamEvent,
  context: Context
): Promise<KinesisStreamBatchResponse> => {
  for (const record of event.Records) {
    try {
      logger.info(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
```

```
      logger.info(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      logger.error(`An error occurred ${err}`);
      /* Since we are working with streams, we can return the failed item
 immediately.
           Lambda will immediately begin to retry processing from this failed
 item onwards. */
      return {
        batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
      };
    }
  }
  logger.info(`Successfully processed ${event.Records.length} records.`);
  return { batchItemFailures: [] };
};

async function getRecordDataAsync(
  payload: KinesisStreamRecordPayload
): Promise<string> {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

PHP

適用於 PHP 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 PHP 使用 Lambda 報告 Kinesis 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php
```

```
# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kinesis\KinesisEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): array
    {
        $kinesisEvent = new KinesisEvent($event);
        $this->logger->info("Processing records");
        $records = $kinesisEvent->getRecords();

        $failedRecords = [];
        foreach ($records as $record) {
            try {
                $data = $record->getData();
                $this->logger->info(json_encode($data));
                // TODO: Do interesting work based on the new data
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $failedRecords[] = $record->getSequenceNumber();
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");

        // change format for the response
        $failures = array_map(
```

```
            fn(string $sequenceNumber) => ['itemIdentifier' => $sequenceNumber],
            $failedRecords
        );

        return [
            'batchItemFailures' => $failures
        ];
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

適用於 Python (Boto3) 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

透過使用 Python 的 Lambda 報告 Kinesis 批次項目失敗。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def handler(event, context):
    records = event.get("Records")
    curRecordSequenceNumber = ""

    for record in records:
        try:
            # Process your record
            curRecordSequenceNumber = record["kinesis"]["sequenceNumber"]
        except Exception as e:
            # Return failed record's sequence number
            return {"batchItemFailures":[{"itemIdentifier":
 curRecordSequenceNumber}]}

    return {"batchItemFailures":[]}
```

Ruby

適用於 Ruby 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Ruby 使用 Lambda 報告 Kinesis 批次項目失敗。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'aws-sdk'

def lambda_handler(event:, context:)
  batch_item_failures = []

  event['Records'].each do |record|
    begin
      puts "Processed Kinesis Event - EventID: #{record['eventID']}"
      record_data = get_record_data_async(record['kinesis'])
      puts "Record Data: #{record_data}"
      # TODO: Do interesting work based on the new data
    rescue StandardError => err
      puts "An error occurred #{err}"
      # Since we are working with streams, we can return the failed item
 immediately.
      # Lambda will immediately begin to retry processing from this failed item
 onwards.
      return { batchItemFailures: [{ itemIdentifier: record['kinesis']
['sequenceNumber'] }] }
    end
  end

  puts "Successfully processed #{event['Records'].length} records."
  { batchItemFailures: batch_item_failures }
end
```

```
def get_record_data_async(payload)
  data = Base64.decode64(payload['data']).force_encoding('utf-8')
  # Placeholder for actual async work
  sleep(1)
  data
end
```

Rust

適用於 Rust 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Rust 使用 Lambda 報告 Kinesis 批次項目故障。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::{
    event::kinesis::KinesisEvent,
    kinesis::KinesisEventRecord,
    streams::{KinesisBatchItemFailure, KinesisEventResponse},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<KinesisEvent>) ->
 Result<KinesisEventResponse, Error> {
    let mut response = KinesisEventResponse {
        batch_item_failures: vec![],
    };

    if event.payload.records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(response);
    }

    for record in &event.payload.records {
        tracing::info!(
            "EventId: {}",
```

```
            record.event_id.as_deref().unwrap_or_default()
        );

        let record_processing_result = process_record(record);

        if record_processing_result.is_err() {
            response.batch_item_failures.push(KinesisBatchItemFailure {
                item_identifier: record.kinesis.sequence_number.clone(),
            });
            /* Since we are working with streams, we can return the failed item
 immediately.
            Lambda will immediately begin to retry processing from this failed
 item onwards. */
            return Ok(response);
        }
    }

    tracing::info!(
        "Successfully processed {} records",
        event.payload.records.len()
    );

    Ok(response)
}

fn process_record(record: &KinesisEventRecord) -> Result<(), Error> {
    let record_data = std::str::from_utf8(record.kinesis.data.as_slice());

    if let Some(err) = record_data.err() {
        tracing::error!("Error: {}", err);
        return Err(Error::from(err));
    }

    let record_data = record_data.unwrap_or_default();

    // do something interesting with the data
    tracing::info!("Data: {}", record_data);

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
```

```
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使用 DynamoDB 觸發程序報告 Lambda 函數的批次項目失敗

下列程式碼範例說明如何針對接收來自 DynamoDB 串流之事件的 Lambda 函數實作部分批次回應。此函數會在回應中報告批次項目失敗，指示 Lambda 稍後重試這些訊息。

.NET

AWS SDK for .NET

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 .NET 報告使用 Lambda 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text.Json;
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
```

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace AWSLambda_DDB;

public class Function
{
    public StreamsEventResponse FunctionHandler(DynamoDBEvent dynamoEvent,
 ILambdaContext context)

    {
        context.Logger.LogInformation($"Beginning to process
 {dynamoEvent.Records.Count} records...");
        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 List<StreamsEventResponse.BatchItemFailure>();
        StreamsEventResponse streamsEventResponse = new StreamsEventResponse();

        foreach (var record in dynamoEvent.Records)
        {
            try
            {
                var sequenceNumber = record.Dynamodb.SequenceNumber;
                context.Logger.LogInformation(sequenceNumber);
            }
            catch (Exception ex)
            {
                context.Logger.LogError(ex.Message);
                batchItemFailures.Add(new StreamsEventResponse.BatchItemFailure()
 { ItemIdentifier = record.Dynamodb.SequenceNumber });
            }
        }

        if (batchItemFailures.Count > 0)
        {
            streamsEventResponse.BatchItemFailures = batchItemFailures;
        }

        context.Logger.LogInformation("Stream processing complete.");
        return streamsEventResponse;
    }
}
```

Go

SDK for Go V2

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 使用 Lambda 報告批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

type BatchItemFailure struct {
 ItemIdentifier string `json:"ItemIdentifier"`
}

type BatchResult struct {
 BatchItemFailures []BatchItemFailure `json:"BatchItemFailures"`
}

func HandleRequest(ctx context.Context, event events.DynamoDBEvent)
 (*BatchResult, error) {
 var batchItemFailures []BatchItemFailure
 curRecordSequenceNumber := ""

 for _, record := range event.Records {
  // Process your record
  curRecordSequenceNumber = record.Change.SequenceNumber
 }

 if curRecordSequenceNumber != "" {
  batchItemFailures = append(batchItemFailures, BatchItemFailure{ItemIdentifier:
 curRecordSequenceNumber})
```

```
 }

 batchResult := BatchResult{
  BatchItemFailures: batchItemFailures,
 }

 return &batchResult, nil
}

func main() {
 lambda.Start(HandleRequest)
}
```

Java

適用於 Java 2.x 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Java 使用 Lambda 報告批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;
import com.amazonaws.services.lambda.runtime.events.models.dynamodb.StreamRecord;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

public class ProcessDynamodbRecords implements RequestHandler<DynamodbEvent,
 Serializable> {

    @Override
```

```
    public StreamsEventResponse handleRequest(DynamodbEvent input, Context
 context) {

        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<>();
        String curRecordSequenceNumber = "";

        for (DynamodbEvent.DynamodbStreamRecord dynamodbStreamRecord :
 input.getRecords()) {
          try {
                //Process your record
                StreamRecord dynamodbRecord = dynamodbStreamRecord.getDynamodb();
                curRecordSequenceNumber = dynamodbRecord.getSequenceNumber();

            } catch (Exception e) {
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                batchItemFailures.add(new
 StreamsEventResponse.BatchItemFailure(curRecordSequenceNumber));
                return new StreamsEventResponse(batchItemFailures);
            }
        }

       return new StreamsEventResponse();
    }
}
```

JavaScript

適用於 JavaScript (v3) 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Lambda 使用報告批次項目失敗 JavaScript

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
export const handler = async (event) => {
  const records = event.Records;
  let curRecordSequenceNumber = "";

  for (const record of records) {
    try {
      // Process your record
      curRecordSequenceNumber = record.dynamodb.SequenceNumber;
    } catch (e) {
      // Return failed record's sequence number
      return { batchItemFailures: [{ itemIdentifier:
 curRecordSequenceNumber }] };
    }
  }

  return { batchItemFailures: [] };
};
```

使用 Lambda 使用報告批次項目失敗 TypeScript

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { DynamoDBBatchItemFailure, DynamoDBStreamEvent } from "aws-lambda";

export const handler = async (event: DynamoDBStreamEvent):
 Promise<DynamoDBBatchItemFailure[]> => {

    const batchItemsFailures: DynamoDBBatchItemFailure[] = []
    let curRecordSequenceNumber

    for(const record of event.Records) {
        curRecordSequenceNumber = record.dynamodb?.SequenceNumber

        if(curRecordSequenceNumber) {
            batchItemsFailures.push({
                itemIdentifier: curRecordSequenceNumber
            })
        }
    }
```

```
    return batchItemsFailures
}
```

PHP

適用於 PHP 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 PHP 使用 Lambda 報告批次項目失敗。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\DynamoDb\DynamoDbEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): array
```

```
    {
        $dynamoDbEvent = new DynamoDbEvent($event);
        $this->logger->info("Processing records");

        $records = $dynamoDbEvent->getRecords();
        $failedRecords = [];
        foreach ($records as $record) {
            try {
                $data = $record->getData();
                $this->logger->info(json_encode($data));
                // TODO: Do interesting work based on the new data
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $failedRecords[] = $record->getSequenceNumber();
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");

        // change format for the response
        $failures = array_map(
            fn(string $sequenceNumber) => ['itemIdentifier' => $sequenceNumber],
            $failedRecords
        );

        return [
            'batchItemFailures' => $failures
        ];
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

適用於 Python (Boto3) 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Python 使用 Lambda 報告批次項目失敗。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def handler(event, context):
    records = event.get("Records")
    curRecordSequenceNumber = ""

    for record in records:
        try:
            # Process your record
            curRecordSequenceNumber = record["dynamodb"]["SequenceNumber"]
        except Exception as e:
            # Return failed record's sequence number
            return {"batchItemFailures":[{"itemIdentifier":
 curRecordSequenceNumber}]}

    return {"batchItemFailures":[]}
```

Ruby

適用於 Ruby 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用紅寶石使用 Lambda 報告批次項目失敗。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event:, context:)
    records = event["Records"]
    cur_record_sequence_number = ""

    records.each do |record|
      begin
        # Process your record
        cur_record_sequence_number = record["dynamodb"]["SequenceNumber"]
      rescue StandardError => e
        # Return failed record's sequence number
        return {"batchItemFailures" => [{"itemIdentifier" =>
 cur_record_sequence_number}]}
      end
    end

    {"batchItemFailures" => []}
  end
```

Rust

適用於 Rust 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Rust 使用 Lambda 報告批次項目故障。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::{
    event::dynamodb::{Event, EventRecord, StreamRecord},
    streams::{DynamoDbBatchItemFailure, DynamoDbEventResponse},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};
```

```
/// Process the stream record
fn process_record(record: &EventRecord) -> Result<(), Error> {
    let stream_record: &StreamRecord = &record.change;

    // process your stream record here...
    tracing::info!("Data: {:?}", stream_record);

    Ok(())
}

/// Main Lambda handler here...
async fn function_handler(event: LambdaEvent<Event>) ->
 Result<DynamoDbEventResponse, Error> {
    let mut response = DynamoDbEventResponse {
        batch_item_failures: vec![],
    };

    let records = &event.payload.records;

    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(response);
    }

    for record in records {
        tracing::info!("EventId: {}", record.event_id);

        // Couldn't find a sequence number
        if record.change.sequence_number.is_none() {
            response.batch_item_failures.push(DynamoDbBatchItemFailure {
                item_identifier: Some("".to_string()),
            });
            return Ok(response);
        }

        // Process your record here...
        if process_record(record).is_err() {
            response.batch_item_failures.push(DynamoDbBatchItemFailure {
                item_identifier: record.change.sequence_number.clone(),
            });
            /* Since we are working with streams, we can return the failed item
 immediately.
```

```
            Lambda will immediately begin to retry processing from this failed
 item onwards. */
            return Ok(response);
        }
    }

    tracing::info!("Successfully processed {} record(s)", records.len());

    Ok(response)
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使用 Amazon SQS 觸發條件報告 Lambda 函數的批次項目失敗

下列程式碼範例示範如何針對接收來自 SQS 佇列之事件的 Lambda 函數，實作部分批次回應。此函數會在回應中報告批次項目失敗，指示 Lambda 稍後重試這些訊息。

.NET

AWS SDK for .NET

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 .NET 搭配 Lambda 報告 SQS 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SQSEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri
namespace sqsSample;

public class Function
{
    public async Task<SQSBatchResponse> FunctionHandler(SQSEvent evnt,
 ILambdaContext context)
    {
        List<SQSBatchResponse.BatchItemFailure> batchItemFailures = new
 List<SQSBatchResponse.BatchItemFailure>();
        foreach(var message in evnt.Records)
        {
            try
            {
                //process your message
                await ProcessMessageAsync(message, context);
            }
            catch (System.Exception)
            {
                //Add failed message identifier to the batchItemFailures list
                batchItemFailures.Add(new
 SQSBatchResponse.BatchItemFailure{ItemIdentifier=message.MessageId});
            }
```

```
        }
        return new SQSBatchResponse(batchItemFailures);
    }

    private async Task ProcessMessageAsync(SQSEvent.SQSMessage message,
 ILambdaContext context)
    {
        if (String.IsNullOrEmpty(message.Body))
        {
            throw new Exception("No Body in SQS Message.");
        }
        context.Logger.LogInformation($"Processed message {message.Body}");
        // TODO: Do interesting work based on the new message
        await Task.CompletedTask;
    }
}
```

Go

SDK for Go V2

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 使用 Lambda 報告 SQS 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "encoding/json"
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)
```

```
func handler(ctx context.Context, sqsEvent events.SQSEvent)
 (map[string]interface{}, error) {
 batchItemFailures := []map[string]interface{}{}

 for _, message := range sqsEvent.Records {

  if /* Your message processing condition here */ {
   batchItemFailures = append(batchItemFailures, map[string]interface{}
{"itemIdentifier": message.MessageId})
  }
 }

 sqsBatchResponse := map[string]interface{}{
  "batchItemFailures": batchItemFailures,
 }
 return sqsBatchResponse, nil
}

func main() {
 lambda.Start(handler)
}
```

Java

適用於 Java 2.x 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Java 搭配 Lambda 報告 SQS 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SQSEvent;
import com.amazonaws.services.lambda.runtime.events.SQSBatchResponse;
```

```
import java.util.ArrayList;
import java.util.List;

public class ProcessSQSMessageBatch implements RequestHandler<SQSEvent,
 SQSBatchResponse> {
    @Override
    public SQSBatchResponse handleRequest(SQSEvent sqsEvent, Context context) {

        List<SQSBatchResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<SQSBatchResponse.BatchItemFailure>();
        String messageId = "";
        for (SQSEvent.SQSMessage message : sqsEvent.getRecords()) {
            try {
                //process your message
                messageId = message.getMessageId();
            } catch (Exception e) {
                //Add failed message identifier to the batchItemFailures list
                batchItemFailures.add(new
 SQSBatchResponse.BatchItemFailure(messageId));
            }
        }
        return new SQSBatchResponse(batchItemFailures);
    }
}
```

JavaScript

適用於 JavaScript (v3) 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 JavaScript Lambda 報告 SQS 批次項目失敗

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
export const handler = async (event, context) => {
    const batchItemFailures = [];
```

```
    for (const record of event.Records) {
        try {
            await processMessageAsync(record, context);
        } catch (error) {
            batchItemFailures.push({ itemIdentifier: record.messageId });
        }
    }

    return { batchItemFailures };
};

async function processMessageAsync(record, context) {
    if (record.body && record.body.includes("error")) {
        throw new Error("There is an error in the SQS Message.");
    }
    console.log(`Processed message: ${record.body}`);
}
```

使用 TypeScript Lambda 報告 SQS 批次項目失敗

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SQSEvent, SQSBatchResponse, Context, SQSBatchItemFailure, SQSRecord }
 from 'aws-lambda';

export const handler = async (event: SQSEvent, context: Context):
 Promise<SQSBatchResponse> => {
    const batchItemFailures: SQSBatchItemFailure[] = [];

    for (const record of event.Records) {
        try {
            await processMessageAsync(record);
        } catch (error) {
            batchItemFailures.push({ itemIdentifier: record.messageId });
        }
    }

    return {batchItemFailures: batchItemFailures};
};

async function processMessageAsync(record: SQSRecord): Promise<void> {
    if (record.body && record.body.includes("error")) {
        throw new Error('There is an error in the SQS Message.');
```

```
    }
    console.log(`Processed message ${record.body}`);
}
```

PHP

適用於 PHP 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 PHP 使用 Lambda 回報 SQS 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

use Bref\Context\Context;
use Bref\Event\Sqs\SqsEvent;
use Bref\Event\Sqs\SqsHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends SqsHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleSqs(SqsEvent $event, Context $context): void
    {
```

```
        $this->logger->info("Processing SQS records");
        $records = $event->getRecords();

        foreach ($records as $record) {
            try {
                // Assuming the SQS message is in JSON format
                $message = json_decode($record->getBody(), true);
                $this->logger->info(json_encode($message));
                // TODO: Implement your custom processing logic here
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $this->markAsFailed($record);
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords SQS records");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

適用於 Python (Boto3) 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Python 搭配 Lambda 報告 SQS 批次項目失敗。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0

def lambda_handler(event, context):
    if event:
        batch_item_failures = []
```

```
        sqs_batch_response = {}

        for record in event["Records"]:
            try:
                # process message
            except Exception as e:
                batch_item_failures.append({"itemIdentifier":
 record['messageId']})

        sqs_batch_response["batchItemFailures"] = batch_item_failures
        return sqs_batch_response
```

Ruby

適用於 Ruby 的開發套件

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Ruby 搭配 Lambda 報告 SQS 批次項目失敗。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'json'

def lambda_handler(event:, context:)
  if event
    batch_item_failures = []
    sqs_batch_response = {}

    event["Records"].each do |record|
      begin
        # process message
      rescue StandardError => e
        batch_item_failures << {"itemIdentifier" => record['messageId']}
      end
    end

    sqs_batch_response["batchItemFailures"] = batch_item_failures
```

```
    return sqs_batch_response
  end
end
```

Rust

適用於 Rust 的 SDK

> **Note**
>
> 還有更多關於 GitHub。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Rust 搭配 Lambda 報告 SQS 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::{
    event::sqs::{SqsBatchResponse, SqsEvent},
    sqs::{BatchItemFailure, SqsMessage},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn process_record(_: &SqsMessage) -> Result<(), Error> {
    Err(Error::from("Error processing message"))
}

async fn function_handler(event: LambdaEvent<SqsEvent>) ->
 Result<SqsBatchResponse, Error> {
    let mut batch_item_failures = Vec::new();
    for record in event.payload.records {
        match process_record(&record).await {
            Ok(_) => (),
            Err(_) => batch_item_failures.push(BatchItemFailure {
                item_identifier: record.message_id.unwrap(),
            }),
        }
    }

    Ok(SqsBatchResponse {
```

```
        batch_item_failures,
    })
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    run(service_fn(function_handler)).await
}
```

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使 AWS 用 SDK 的 Lambda 跨服務範例

下列範例應用程式使用 AWS SDK 來結合 Lambda 與其他 AWS 服務應用程式。每個範例都包含一個連結 GitHub，您可以在其中找到如何設定和執行應用程式的指示。

範例

- 建立 API Gateway REST API 以追蹤 COVID-19 資料
- 建立出借圖書館 REST API
- 使用 Step Functions 建立傳訊應用程式
- 建立相片資產管理應用程式，讓使用者以標籤管理相片
- 使用 API Gateway 建立 websocket 聊天應用程式
- 建立可分析客戶意見回饋並合成音訊的應用程式
- 從瀏覽器調用 Lambda 函數
- 使用 S3 物件 Lambda 為您的應用程式轉換資料
- 使用 API Gateway 來調用 Lambda 函數
- 使用 Step Functions 呼叫 Lambda 函數
- 使用排程事件來調用 Lambda 函數

建立 API Gateway REST API 以追蹤 COVID-19 資料

以下程式碼範例示範如何建立 REST API，此 API 使用虛構資料模擬追蹤美國 COVID-19 每日病例的系統。

Python

適用於 Python (Boto3) 的 SDK

示範如何搭配使用 AWS Chalice， AWS SDK for Python (Boto3) AWS Lambda以建立使用 Amazon API Gateway 和 Amazon DynamoDB 的無伺服器 REST API。REST API 使用虛構資料模擬追蹤美國 COVID-19 每日病例的系統。了解如何：

- 使用 AWS Chalice 定義 Lambda 函數中的路由，這些函數會呼叫以處理透過 API Gateway 傳送的 REST 要求。
- 使用 Lambda 函數在 DynamoDB 資料表中擷取和存放資料，以便為 REST 請求提供服務。
- 在 AWS CloudFormation 範本中定義資料表結構和資訊安全角色資源。
- 使用 AWS Chalice 並 CloudFormation 封裝和部署所有必要的資源。
- 用 CloudFormation 於清理所有創建的資源。

有關如何設置和運行的完整源代碼和說明，請參閱中的完整示例GitHub。

此範例中使用的服務

- API Gateway
- AWS CloudFormation
- DynamoDB
- Lambda

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

建立出借圖書館 REST API

下列程式碼範例顯示如何使用 Amazon Aurora 資料庫支援的 REST API 來建立出借圖書館，讓贊助人可以借書與還書。

Python

適用於 Python (Boto3) 的 SDK

示範如何 AWS SDK for Python (Boto3) 與 Amazon Relational Database Service 服務 (Amazon RDS) API 和 AWS Chalice 搭配使用，以建立由 Amazon Aurora 資料庫支援的 REST

API。Web 服務是完全無伺服器的，表示這是一種贊助人可以借書與還書的簡單出借圖書館。了解如何：

- 建立與管理無伺服器的 Aurora 資料庫叢集。
- 用 AWS Secrets Manager 於管理資料庫認證。
- 實作資料儲存層，該層使用 Amazon RDS 將資料移入和移出資料庫。
- 使用 AWS Chalice 將無伺服器 REST API 部署到 Amazon API Gateway 和. AWS Lambda
- 使用 Request 套件來將請求傳送到 Web 服務。

有關如何設置和運行的完整源代碼和說明，請參閱中的完整示例GitHub。

此範例中使用的服務

- API Gateway
- Aurora
- Lambda
- Secrets Manager

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使用 Step Functions 建立傳訊應用程式

下列程式碼範例會示範如何建立從資料庫資料表擷取訊息記錄的 AWS Step Functions Messenger 應用程式。

Python

適用於 Python (Boto3) 的 SDK

示範如何使用 AWS SDK for Python (Boto3) 與建立信 AWS Step Functions 使應用程式，以從 Amazon DynamoDB 表擷取訊息記錄，並透過 Amazon Simple Queue Service (Amazon SQS) 傳送這些記錄。狀態機與一個 AWS Lambda 功能集成在數據庫中掃描未發送的消息。

- 建立從 Amazon DynamoDB 資料表擷取和更新訊息記錄的狀態機器。
- 更新狀態機器定義，以便也向 Amazon Simple Queue Service (Amazon SQS) 傳送訊息。
- 開始和停用狀態機器執行。

- 使用服務整合從狀態機器連接至 Lambda、DynamoDB 和 Amazon SQS。

有關如何設置和運行的完整源代碼和說明，請參閱中的完整示例GitHub。

此範例中使用的服務

- DynamoDB
- Lambda
- Amazon SQS
- Step Functions

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

建立相片資產管理應用程式，讓使用者以標籤管理相片

下列程式碼範例示範如何建立無伺服器應用程式，讓使用者以標籤管理相片。

.NET

AWS SDK for .NET

顯示如何開發照片資產管理應用程式，以便使用 Amazon Rekognition 偵測圖片中的標籤，並將其儲存以供日後擷取。

有關如何設置和運行的完整源代碼和說明，請參閱中的完整示例 GitHub。

如要深入探索此範例的來源，請參閱 AWS 社群上的文章。

此範例中使用的服務

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

C++

適用於 C++ 的 SDK

顯示如何開發照片資產管理應用程式，以便使用 Amazon Rekognition 偵測圖片中的標籤，並將其儲存以供日後擷取。

有關如何設置和運行的完整源代碼和說明，請參閱中的完整示例 GitHub。

如要深入探索此範例的來源，請參閱 AWS 社群上的文章。

此範例中使用的服務

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Java

適用於 Java 2.x 的 SDK

顯示如何開發照片資產管理應用程式，以便使用 Amazon Rekognition 偵測圖片中的標籤，並將其儲存以供日後擷取。

有關如何設置和運行的完整源代碼和說明，請參閱中的完整示例 GitHub。

如要深入探索此範例的來源，請參閱 AWS 社群上的文章。

此範例中使用的服務

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3

- Amazon SNS

JavaScript

適用於 JavaScript (v3) 的開發套件

顯示如何開發照片資產管理應用程式，以便使用 Amazon Rekognition 偵測圖片中的標籤，並將其儲存以供日後擷取。

有關如何設置和運行的完整源代碼和說明，請參閱中的完整示例 GitHub。

如要深入探索此範例的來源，請參閱 AWS 社群上的文章。

此範例中使用的服務

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Kotlin

適用於 Kotlin 的 SDK

顯示如何開發照片資產管理應用程式，以便使用 Amazon Rekognition 偵測圖片中的標籤，並將其儲存以供日後擷取。

有關如何設置和運行的完整源代碼和說明，請參閱中的完整示例 GitHub。

如要深入探索此範例的來源，請參閱 AWS 社群上的文章。

此範例中使用的服務

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition

- Amazon S3
- Amazon SNS

PHP

適用於 PHP 的開發套件

顯示如何開發照片資產管理應用程式，以便使用 Amazon Rekognition 偵測圖片中的標籤，並將其儲存以供日後擷取。

有關如何設置和運行的完整源代碼和說明，請參閱中的完整示例 GitHub。

如要深入探索此範例的來源，請參閱 AWS 社群上的文章。

此範例中使用的服務

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Rust

適用於 Rust 的 SDK

顯示如何開發照片資產管理應用程式，以便使用 Amazon Rekognition 偵測圖片中的標籤，並將其儲存以供日後擷取。

有關如何設置和運行的完整源代碼和說明，請參閱中的完整示例 GitHub。

如要深入探索此範例的來源，請參閱 AWS 社群上的文章。

此範例中使用的服務

- API Gateway
- DynamoDB
- Lambda

- Amazon Rekognition
- Amazon S3
- Amazon SNS

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使用 API Gateway 建立 websocket 聊天應用程式

下列程式碼範例示範如何建立由建置於 Amazon API Gateway 上的 websocket API 提供服務的聊天應用程式。

Python

適用於 Python (Boto3) 的 SDK

示範如何使用 AWS SDK for Python (Boto3) 與 Amazon API Gateway V2 搭配使用，以建立與亞馬遜動態 B 整合的 AWS Lambda 網路通訊端 API。

- 建立由 API Gateway 提供服務的 websocket API。
- 定義 Lambda 處理常式，該常式將連接存放在 DynamoDB 中，並將訊息傳送給其他聊天參與者。
- 連接至 websocket 聊天應用程式，並使用 Websockets 套件傳送訊息。

有關如何設置和運行的完整源代碼和說明，請參閱中的完整示例GitHub。

此範例中使用的服務

- API Gateway
- DynamoDB
- Lambda

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

建立可分析客戶意見回饋並合成音訊的應用程式

下列程式碼範例示範如何建立應用程式，以分析客戶評論卡、從其原始語言進行翻譯、判斷對方情緒，以及透過翻譯後的文字產生音訊檔案。

.NET

AWS SDK for .NET

此範例應用程式會分析和存儲客戶的意見回饋卡。具體來說，它滿足了紐約市一家虛構飯店的需求。飯店以實體評論卡的形式收到賓客以各種語言撰寫的意見回饋。這些意見回饋透過 Web 用戶端上傳至應用程式。評論卡的影像上傳後，系統會執行下列步驟：

- 文字內容是使用 Amazon Textract 從影像中擷取。
- Amazon Comprehend 會決定擷取文字及其用語的情感。
- 擷取的文字內容會使用 Amazon Translate 翻譯成英文。
- Amazon Polly 會使用擷取的文字內容合成音訊檔案。

完整的應用程式可透過 AWS CDK 部署。如需原始程式碼和部署指示，請參閱中的專案 GitHub。

此範例中使用的服務

- Amazon Comprehend
- Lambda
- Amazon Polly
- Amazon Textract
- Amazon Translate

Java

適用於 Java 2.x 的 SDK

此範例應用程式會分析和存儲客戶的意見回饋卡。具體來說，它滿足了紐約市一家虛構飯店的需求。飯店以實體評論卡的形式收到賓客以各種語言撰寫的意見回饋。這些意見回饋透過 Web 用戶端上傳至應用程式。評論卡的影像上傳後，系統會執行下列步驟：

- 文字內容是使用 Amazon Textract 從影像中擷取。
- Amazon Comprehend 會決定擷取文字及其用語的情感。
- 擷取的文字內容會使用 Amazon Translate 翻譯成英文。
- Amazon Polly 會使用擷取的文字內容合成音訊檔案。

完整的應用程式可透過 AWS CDK 部署。如需原始程式碼和部署指示，請參閱中的專案 GitHub。

此範例中使用的服務

- Amazon Comprehend
- Lambda
- Amazon Polly
- Amazon Textract
- Amazon Translate

JavaScript

適用於 JavaScript (v3) 的開發套件

此範例應用程式會分析和存儲客戶的意見回饋卡。具體來說，它滿足了紐約市一家虛構飯店的需求。飯店以實體評論卡的形式收到賓客以各種語言撰寫的意見回饋。這些意見回饋透過 Web 用戶端上傳至應用程式。評論卡的影像上傳後，系統會執行下列步驟：

- 文字內容是使用 Amazon Textract 從影像中擷取。
- Amazon Comprehend 會決定擷取文字及其用語的情感。
- 擷取的文字內容會使用 Amazon Translate 翻譯成英文。
- Amazon Polly 會使用擷取的文字內容合成音訊檔案。

完整的應用程式可透過 AWS CDK 部署。如需原始程式碼和部署指示，請參閱中的專案 GitHub。下列摘錄顯示如何在 AWS SDK for JavaScript Lambda 函數內部使用。

```
import {
  ComprehendClient,
  DetectDominantLanguageCommand,
  DetectSentimentCommand,
} from "@aws-sdk/client-comprehend";

/**
 * Determine the language and sentiment of the extracted text.
 *
 * @param {{ source_text: string}} extractTextOutput
 */
export const handler = async (extractTextOutput) => {
  const comprehendClient = new ComprehendClient({});

  const detectDominantLanguageCommand = new DetectDominantLanguageCommand({
    Text: extractTextOutput.source_text,
```

```
  });

  // The source language is required for sentiment analysis and
  // translation in the next step.
  const { Languages } = await comprehendClient.send(
    detectDominantLanguageCommand,
  );

  const languageCode = Languages[0].LanguageCode;

  const detectSentimentCommand = new DetectSentimentCommand({
    Text: extractTextOutput.source_text,
    LanguageCode: languageCode,
  });

  const { Sentiment } = await comprehendClient.send(detectSentimentCommand);

  return {
    sentiment: Sentiment,
    language_code: languageCode,
  };
};
```

```
import {
  DetectDocumentTextCommand,
  TextractClient,
} from "@aws-sdk/client-textract";

/**
 * Fetch the S3 object from the event and analyze it using Amazon Textract.
 *
 * @param {import("@types/aws-lambda").EventBridgeEvent<"Object Created">}
 eventBridgeS3Event
 */
export const handler = async (eventBridgeS3Event) => {
  const textractClient = new TextractClient();

  const detectDocumentTextCommand = new DetectDocumentTextCommand({
    Document: {
      S3Object: {
        Bucket: eventBridgeS3Event.bucket,
        Name: eventBridgeS3Event.object,
      },
```

```
    },
  });

  // Textract returns a list of blocks. A block can be a line, a page, word, etc.
  // Each block also contains geometry of the detected text.
  // For more information on the Block type, see https://docs.aws.amazon.com/
textract/latest/dg/API_Block.html.
  const { Blocks } = await textractClient.send(detectDocumentTextCommand);

  // For the purpose of this example, we are only interested in words.
  const extractedWords = Blocks.filter((b) => b.BlockType === "WORD").map(
    (b) => b.Text,
  );

  return extractedWords.join(" ");
};
```

```
import { PollyClient, SynthesizeSpeechCommand } from "@aws-sdk/client-polly";
import { S3Client } from "@aws-sdk/client-s3";
import { Upload } from "@aws-sdk/lib-storage";

/**
 * Synthesize an audio file from text.
 *
 * @param {{ bucket: string, translated_text: string, object: string}}
 sourceDestinationConfig
 */
export const handler = async (sourceDestinationConfig) => {
  const pollyClient = new PollyClient({});

  const synthesizeSpeechCommand = new SynthesizeSpeechCommand({
    Engine: "neural",
    Text: sourceDestinationConfig.translated_text,
    VoiceId: "Ruth",
    OutputFormat: "mp3",
  });

  const { AudioStream } = await pollyClient.send(synthesizeSpeechCommand);

  const audioKey = `${sourceDestinationConfig.object}.mp3`;

  // Store the audio file in S3.
  const s3Client = new S3Client();
```

```
  const upload = new Upload({
    client: s3Client,
    params: {
      Bucket: sourceDestinationConfig.bucket,
      Key: audioKey,
      Body: AudioStream,
      ContentType: "audio/mp3",
    },
  });

  await upload.done();
  return audioKey;
};
```

```
import {
  TranslateClient,
  TranslateTextCommand,
} from "@aws-sdk/client-translate";

/**
 * Translate the extracted text to English.
 *
 * @param {{ extracted_text: string, source_language_code: string}}
 textAndSourceLanguage
 */
export const handler = async (textAndSourceLanguage) => {
  const translateClient = new TranslateClient({});

  const translateCommand = new TranslateTextCommand({
    SourceLanguageCode: textAndSourceLanguage.source_language_code,
    TargetLanguageCode: "en",
    Text: textAndSourceLanguage.extracted_text,
  });

  const { TranslatedText } = await translateClient.send(translateCommand);

  return { translated_text: TranslatedText };
};
```

此範例中使用的服務

- Amazon Comprehend
- Lambda

- Amazon Polly
- Amazon Textract
- Amazon Translate

Ruby

適用於 Ruby 的開發套件

此範例應用程式會分析和存儲客戶的意見回饋卡。具體來說，它滿足了紐約市一家虛構飯店的需求。飯店以實體評論卡的形式收到賓客以各種語言撰寫的意見回饋。這些意見回饋透過 Web 用戶端上傳至應用程式。評論卡的影像上傳後，系統會執行下列步驟：

- 文字內容是使用 Amazon Textract 從影像中擷取。
- Amazon Comprehend 會決定擷取文字及其用語的情感。
- 擷取的文字內容會使用 Amazon Translate 翻譯成英文。
- Amazon Polly 會使用擷取的文字內容合成音訊檔案。

完整的應用程式可透過 AWS CDK 部署。如需原始程式碼和部署指示，請參閱中的專案 GitHub。

此範例中使用的服務

- Amazon Comprehend
- Lambda
- Amazon Polly
- Amazon Textract
- Amazon Translate

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

從瀏覽器調用 Lambda 函數

下列程式碼範例會示範如何從瀏覽器叫用 AWS Lambda 函數。

JavaScript

適用於 JavaScript (v2) 的開發套件

您可以建立以瀏覽器為基礎的應用程式，使用 AWS Lambda 函數更新具有使用者選擇的 Amazon DynamoDB 表格。

有關如何設置和運行的完整源代碼和說明，請參閱中的完整示例GitHub。

此範例中使用的服務

- DynamoDB
- Lambda

適用於 JavaScript (v3) 的開發套件

您可以建立以瀏覽器為基礎的應用程式，使用 AWS Lambda 函數更新具有使用者選擇的 Amazon DynamoDB 表格。此應用程序使用 AWS SDK for JavaScript v3。

有關如何設置和運行的完整源代碼和說明，請參閱中的完整示例GitHub。

此範例中使用的服務

- DynamoDB
- Lambda

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使用 S3 物件 Lambda 為您的應用程式轉換資料

下列程式碼範例示範如何使用 S3 物件 Lambda 轉換應用程式的資料。

.NET

AWS SDK for .NET

示範如何將自訂程式碼新增至標準 S3 GET 請求，以修改從 S3 擷取的要求物件，讓物件符合要求用戶端或應用程式的需求。

有關如何設置和運行的完整源代碼和說明，請參閱中的完整示例GitHub。

此範例中使用的服務

- Lambda
- Amazon S3

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使用 API Gateway 來調用 Lambda 函數

下列程式碼範例說明如何建立 Amazon API Gateway 叫用的 AWS Lambda 函數。

Java

適用於 Java 2.x 的 SDK

示範如何使用 Lambda Java 執行階段 API 建立 AWS Lambda 函數。此範例會呼叫不同的 AWS 服務來執行特定使用案例。此範例示範如何建立 Amazon API Gateway 調用的 Lambda 函數，該函數會掃描 Amazon DynamoDB 資料表中的工作週年紀念日，並使用 Amazon Simple Notification Service (Amazon SNS) 傳送文字訊息給您的員工，在他們的週年紀念日向他們道賀。

有關如何設置和運行的完整源代碼和說明，請參閱中的完整示例GitHub。

此範例中使用的服務

- API Gateway
- DynamoDB
- Lambda
- Amazon SNS

JavaScript

適用於 JavaScript (v3) 的開發套件

示範如何使用 Lambda JavaScript 執行階段 API 建立 AWS Lambda 函數。此範例會呼叫不同的 AWS 服務來執行特定使用案例。此範例示範如何建立 Amazon API Gateway 調用的 Lambda 函數，該函數會掃描 Amazon DynamoDB 資料表中的工作週年紀念日，並使用 Amazon Simple

Notification Service (Amazon SNS) 傳送文字訊息給您的員工，在他們的週年紀念日向他們道賀。

有關如何設置和運行的完整源代碼和說明，請參閱中的完整示例GitHub。

此範例也可在 AWS SDK for JavaScript v3 開發人員指南中取得。

此範例中使用的服務

- API Gateway
- DynamoDB
- Lambda
- Amazon SNS

Python

適用於 Python (Boto3) 的開發套件

此範例顯示如何建立和使用目標為 AWS Lambda 函數的 Amazon API Gateway REST API。Lambda 處理常式會展示如何根據 HTTP 方法來路由；如何從查詢字串、標頭和本文中取得資料；以及如何傳回 JSON 回應。

- 部署 Lambda 函數。
- 建立 API Gateway REST API。
- 建立目標為 Lambda 函數的 REST 資源。
- 授與許可讓 API Gateway 調用 Lambda 函數。
- 使用 Request 套件來將請求傳送到 REST API。
- 清理示範期間建立的所有資源。

此範例最佳檢視於 GitHub。有關如何設置和運行的完整源代碼和說明，請參閱中的完整示例GitHub。

此範例中使用的服務

- API Gateway
- Lambda

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使用 Step Functions 呼叫 Lambda 函數

下列程式碼範例會示範如何建立依序叫用 AWS Lambda 函式的 AWS Step Functions 狀態機器。

Java

適用於 Java 2.x 的 SDK

說明如何使用 AWS Step Functions 和建立 AWS 無伺服器工作流程。 AWS SDK for Java 2.x每個工作流程步驟都是使用 AWS Lambda 函數來實作。

有關如何設置和運行的完整源代碼和說明，請參閱中的完整示例GitHub。

此範例中使用的服務

- DynamoDB
- Lambda
- Amazon SES
- Step Functions

JavaScript

適用於 JavaScript (v3) 的開發套件

說明如何使用 AWS Step Functions 和建立 AWS 無伺服器工作流程。 AWS SDK for JavaScript每個工作流程步驟都是使用 AWS Lambda 函數來實作。

Lambda 是一項運算服務，可讓您執行程式碼，而無需佈建或管理伺服器。Step Functions 是一種無伺服器協同運作服務，可讓您結合 Lambda 函數和其他 AWS 服務來建置關鍵業務應用程式。

有關如何設置和運行的完整源代碼和說明，請參閱中的完整示例GitHub。

此範例也可在 AWS SDK for JavaScript v3 開發人員指南中取得。

此範例中使用的服務

- DynamoDB
- Lambda
- Amazon SES

- Step Functions

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使用排程事件來調用 Lambda 函數

下列程式碼範例說明如何建立 Amazon EventBridge 排程事件所叫用的 AWS Lambda 函數。

Java

適用於 Java 2.x 的 SDK

說明如何建立叫用 AWS Lambda 函數的 Amazon EventBridge 排程事件。設定 EventBridge 為在叫用 Lambda 函數時使用 cron 運算式來排程。在此範例中，您會使用 Lambda Java 執行期 API 建立 Lambda 函數。此範例會呼叫不同的 AWS 服務來執行特定使用案例。此範例示範如何建立應用程式，將行動裝置文字訊息傳送給員工，在他們的週年紀念日向他們道賀。

有關如何設置和運行的完整源代碼和說明，請參閱中的完整示例GitHub。

此範例中使用的服務

- DynamoDB
- EventBridge
- Lambda
- Amazon SNS

JavaScript

適用於 JavaScript (v3) 的開發套件

說明如何建立叫用 AWS Lambda 函數的 Amazon EventBridge 排程事件。設定 EventBridge 為在叫用 Lambda 函數時使用 cron 運算式來排程。在此範例中，您可以使用 Lambda JavaScript 執行階段 API 建立 Lambda 函數。此範例會呼叫不同的 AWS 服務來執行特定使用案例。此範例示範如何建立應用程式，將行動裝置文字訊息傳送給員工，在他們的週年紀念日向他們道賀。

有關如何設置和運行的完整源代碼和說明，請參閱中的完整示例GitHub。

此範例也可在 AWS SDK for JavaScript v3 開發人員指南中取得。

此範例中使用的服務

- DynamoDB
- EventBridge
- Lambda
- Amazon SNS

Python

適用於 Python (Boto3) 的 SDK

此範例顯示如何將 AWS Lambda 函數註冊為已排程 Amazon EventBridge 事件的目標。Lambda 處理常式會將友善的訊息和完整事件資料寫入 Amazon CloudWatch 日誌，以供日後擷取。

- 部署 Lambda 函式。
- 建立 EventBridge 排程的事件，並使 Lambda 函數成為目標。
- 授予允許 EventBridge 叫用 Lambda 函數的權限。
- 列印記 CloudWatch 錄檔中的最新資料，以顯示排定呼叫的結果。
- 清理示範期間建立的所有資源。

此範例最佳檢視於 GitHub。有關如何設置和運行的完整源代碼和說明，請參閱中的完整示例GitHub。

此範例中使用的服務

- CloudWatch 日誌
- EventBridge
- Lambda

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱搭配 AWS 開發套件使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

Lambda 配額

> **Important**
>
> 新功 AWS 帳戶 能減少了並發性和內存配額。 AWS 根據您的使用情況自動提高這些配額。

運算與儲存

Lambda 會為運算和儲存資源的數量設定配額，您可以使用它們來執行並存放函數。並行執行和儲存的配額適用於每個 AWS 區域. 彈性網絡介面 (ENI) 配額適用於每個虛擬私有雲端 (VPC) (不受區域影響)。下列配額可以從其預設值增加。如需詳細資訊，請參閱《Service Quotas 使用者指南》中的請求提高配額。

資源	預設配額	最多可提高至
並行執行數	1,000	數萬
儲存已上傳的函數 (.zip 封存檔) 和層。每個函數版本和 layer 版本都會消耗儲存空間。 如需管理程式碼儲存的最佳實務，請參閱無伺服器園地中的 監控 Lambda 程式碼儲存。	75 GB	TB
儲存定義為容器映像的函數。這些映像會存放在 Amazon ECR 中。	請參閱 Amazon ECR 服務配額。	
每個 Virtual Private Cloud (VPC) 的彈性網路介面 **Note** 與其他服務 (例如 Amazon Elastic File System (Amazon EFS)) 共用此配額。請參閱 Amazon VPC 配額。	250	數千

如需有關並行及 Lambda 如何擴展函數並行以回應流量的詳細資訊，請參閱 了解 Lambda 展函數。

函數組態、部署和執行

以下配額可套用到函數組態、部署和執行。除非另有說明，否則無法變更。

> **Note**
>
> Lambda 文件、日誌訊息和主控台會使用縮寫 MB (而非 MiB) 來參考 1,024 KB。

資源	配額
函式記憶體分配	128 MB 至 10,240 MB，以 1 MB 遞增。 注意：Lambda 會按設定的記憶體數量比例來配置 CPU 功率。可使用 記憶體 (MB) 設定來增加或減少配置給函數的記憶體和 CPU 功率。在 1,769 MB 時，一個函數相當於一個 vCPU。
函數逾時	900 秒 (15 分鐘)
函數環境變數	對於函數相關聯的所有環境變量而言總計為 4 KB
函式以資源為基礎的政策	20 KB
函式 Layer	五層
函數並行擴展限制	對於每個函數，每 10 秒 1,000 個執行環境
調用承載 (請求和回應)	請求和回應各 6 MB (同步) 每個串流回應為 20 MB (同步. 串流回應的承載大小可以從預設值增加。 聯繫 AWS Support 以進一步諮詢。)

資源	配額
	256 KB (非同步) 請求明細行與表頭值的總大小總計為 1 MB
串流回應 的頻寬	函數回應的前 6 MB 不受限制 若回應大於 6 MB，則該回應的剩餘部分為 2 MBps
部署套件 (.zip 封存檔) 大小	50 MB (壓縮，可直接上傳) 250 MB (解壓縮) 此配額適用於您上傳的所有檔案，包括圖層和自訂執行期。 3 MB (主控台編輯器)
容器映像設定大小	16 KB
容器映像程式碼套件大小	10 GB (未壓縮影像大小上限，包括所有圖層)
測試事件 (主控台編輯器)	10
/tmp 目錄儲存	選擇介於 512 MB 與 10,240 MB 的數量，增量為 1 MB。
檔案描述項	1,024
執行程序/執行緒	1,024

Lambda API 請求

下列配額與 Lambda API 請求關聯。

資源	配額
每個區域的每函數調用請求數 (同步)	執行環境的每個執行個體每秒最多可處理 10 個請求。換句話說，總調用上限是並行上限的 10 倍。請參閱了解 Lambda 展函數。
每個區域的每函數調用請求數 (非同步)	執行環境的每個執行個體都可以處理無限數量的請求。換句話說，總調用上限僅取決於函數可用的並行。請參閱了解 Lambda 展函數。
每個函數版本或別名的調用頻率 (每秒請求數)	10 x 配置的佈建並行 Note 此配額僅適用於使用佈建並行的函數。
GetFunction API 請求	每秒 100 個請求。無法增加。
GetPolicy API 請求	每秒 15 個要求。無法增加。
控制平面 API 請求的其餘部分（不包括調用和 GetFunction請 GetPolicy求）	所有 API 每秒 15 個請求 (不是每個 API 每秒 15 個請求)。無法增加。

其他服務

其他服務 (例如 AWS Identity and Access Management (IAM)、Amazon CloudFront (Lambda @Edge) 和 Amazon Virtual Private Cloud (Amazon VPC) 的配額可能會影響您的 Lambda 函數。如需詳細資訊，請參閱 Amazon Web Services 一般參考 中的 AWS 服務 quotas 和 使用來自其 AWS 他服務的事件叫用 Lambda。

文件歷史記錄

下表說明 2018 年 5 月後 AWS Lambda 開發人員指南的重要變更。如需有關此文件更新的通知，您可以訂閱 RSS 摘要。

變更	描述	日期
新區域 SnapStart 的 Support	Lambda 現SnapStart已在下列區域提供：歐洲 (西班牙)、歐洲 (蘇黎世)、亞太區域 (墨爾本)、亞太區域 (海德拉巴) 和中東 (阿拉伯聯合大公國)。	2024 年 1 月 12 日
AWS 受管理策略更新	Service Quotas 更新了現有的 AWS 受管策略 (`AWSLambda VPCAccessExecution Role`)。	2024年1月5日
Python 3.12 執行期	Lambda 現在支援 Python 3.12 做為受管理的執行期和容器基礎映像。如需詳細資訊，請參閱 AWS 運算部落格中現已提供的 AWS Lambda Python 3.12 執行階段。	2023 年 12 月 14 日
Java 21 執行期	Lambda 現在支援 Java 21 做為受管理的執行期和容器基礎映像檔 (`java21`)。	2023 年 11 月 16 日
Node.js 20.x 執行期	Lambda 現在支援 Node.js 20 做為受管理的執行期和容器基礎映像檔 (`nodejs20.x`))。如需詳細資訊，請參閱 AWS 計算部落格 AWS Lambda上現在提供的 Node.js 20.x 執行階段。	2023 年 11 月 14 日

provided.al2023 執行期	Lambda 現在支援 Amazon Linux 2023 做為受管執行期和容器基礎映像。如需詳細資訊，請參閱 AWS 運算部落格 AWS Lambda上的介紹 Amazon Linux 2023 執行階段。	2023 年 11 月 9 日
IPv6 支援雙堆疊子網路	Lambda 現在也支援雙堆疊子網路的傳出 IPv6 流量。如需詳細資訊，請參閱 IPv6 支援。	2023 年 10 月 12 日
測試無伺服器函數和應用程式	瞭解在雲端中偵錯和自動測試無伺服器函數的技術。我們目前已在 Python 和 Typescript 程式語言區段加入測試章節和相關資源。如需詳細資訊，請參閱 測試無伺服器函數和應用程式。	2023 年 6 月 16 日
Ruby 3.2 執行期	Lambda 現在支援全新的 Ruby 3.2 執行期。如需詳細資訊，請參閱 使用 Ruby 建置 Lambda 函數。	2023 年 6 月 7 日
回應串流	Lambda 現在支援來自函數的串流回應。如需詳細資訊，請參閱設定 Lambda 函數以串流回應。	2023 年 4 月 6 日
非同步調用指標	Lambda 已發佈非同步調用指標。如需詳細資訊，請參閱非同步調用指標。	2023 年 2 月 9 日

執行階段版本控制項	Lambda 發佈了新的執行階段版本，包含安全性更新、錯誤修正和新功能。您現在可以控制何時將函數更新為新的執行階段版本。如需詳細資訊，請參閱 Lambda 執行階段更新。	2023 年 1 月 23 日
Lambda SnapStart	使用 Lambda SnapStart 可減少 Java 函數的啟動時間，而無需佈建額外資源或實作複雜的效能最佳化。如需詳細資訊，請參閱使用 Lambda 改善啟動效能 SnapStart。	2022 年 11 月 28 日
Node.js 18 執行階段	Lambda 現在支援 Node.js 18 的新執行階段。Node.js 18 使用 Amazon Linux 2。如需詳細資訊，請參閱使用 Node.js 建立 Lambda 函數。	2022 年 11 月 18 日
拉姆達：SourceFunctionArn 條件鍵	對於資 AWS 源，`lambda:SourceFunctionArn` 條件索引鍵會透過 Lambda 函數的 ARN 篩選對資源的存取。如需詳細資訊，請參閱使用 Lambda 執行環境憑證。	2022 年 7 月 1 日
Node.js 16 執行期	Lambda 現在支援 Node.js 16 的新執行期。Node.js 16 使用 Amazon Linux 2。如需詳細資訊，請參閱使用 Node.js 建立 Lambda 函數。	2022 年 5 月 11 日
Lambda 函數 URL	Lambda 現可支援函數 URL，這是 Lambda 函數專用的 HTTP(S) 端點。如需詳細資訊，請參閱 Lambda 函數 URL。	2022 年 4 月 6 日

AWS Lambda 控制台中的共享測試事件	Lambda 現可支援與相同 AWS 帳戶中的其他使用者共享測試事件。如需詳細資訊，請參閱在主控台中測試 Lambda 函數。	2022 年 3 月 16 日
PrincipalOrgId 以資源為基礎的政策	Lambda 現可支援將許可授予 AWS Organizations中的組織。如需詳細資訊，請參閱對 AWS Lambda使用以資源為基礎的政策。	2022 年 3 月 11 日
.NET 6 執行期	Lambda 現在支援 .NET 6 的新執行期。如需詳細資訊，請參閱 Lambda 執行期。	2022 年 2 月 23 日
篩選 Kinesis、DynamoDB 和 Amazon SQS 等事件來源的事件	Lambda 現在支援篩選 Kinesis、DynamoDB 和 Amazon SQS 等事件來源的事件 如需詳細資訊，請參閱 Lambda 事件篩選。	2021 年 11 月 24 日
Amazon MSK 的 mTLS 身分驗證和自我管理的 Apache Kafka 事件來源	Lambda 現在支援 Amazon MSK 的 mTLS 身分驗證和自我管理的 Apache Kafka 事件來源。如需詳細資訊，請參閱搭配使用 Lambda 與 Amazon MSK。	2021 年 11 月 19 日
Lambda on Graviton2	Lambda 現在支援 Graviton2 使用 arm64 架構運作。如需詳細資訊，請參閱 Lambda 指令集架構。	2021 年 9 月 29 日
Python 3.9 執行期	Lambda 現在支援 Python 3.9 的新執行期。如需詳細資訊，請參閱 Lambda 執行期。	2021 年 8 月 16 日

適用於 Node.js、Python 和 Java 的新執行期版本	新執行期版本適用於 Node.js、Python 和 Java。如需詳細資訊，請參閱 Lambda 執行期。	2021 年 7 月 21 日
支援 RabbitMQ 作為 Lambda 的事件來源	Lambda 現在支援 Amazon MQ for RabbitMQ 作為事件來源。Amazon MQ 是一種適用於 Apache ActiveMQ 和 RabbitMQ 的受管型訊息代理程式服務，可讓您輕鬆地在雲端設定及操作訊息代理程式。如需詳細資訊，請參閱搭配使用 Lambda 與 Amazon MQ。	2021 年 7 月 7 日
Lambda 上適用於自我管理型 Kafka 的 SASL/PLAIN 身分驗證	SASL/PLAIN 現在是 Lambda 上適用於自我管理型 Kafka 事件來源的受支援身分驗證機制。已在其自我管理型 Kafka 叢集上使用 SASL/PLAIN 的客戶現在可以輕鬆地使用 Lambda 來建置消費者應用程式，而不必修改身分驗證方式。如需詳細資訊，請參閱搭配使用 Lambda 與自我管理 Apache Kafka。	2021 年 6 月 29 日
Lambda Extensions API	Lambda 擴展的一般可用性。使用擴展來增強 Lambda 函數。您可以使用 Lambda 合作夥伴提供的擴展，也可以建立自己的 Lambda 擴展。如需詳細資訊，請參閱 Lambda Extensions API。	2021 年 5 月 24 日
全新 Lambda 主控台體驗	Lambda 主控台已重新設計，以改善效能和一致性。	2021 年 3 月 2 日

Node.js 14 執行期	Lambda 現在支援 Node.js 14 的新執行期。Node.js 14 使用 Amazon Linux 2。如需詳細資訊，請參閱使用 Node.js 建立 Lambda 函數。	2021 年 1 月 27 日
Lambda 容器映像	Lambda 現在支援定義為容器映像的函數。您可以結合容器工具的靈活性和 Lambda 的敏捷性和操作簡便性來建置應用程式。如需詳細資訊，請參閱將容器映像與 Lambda 搭配使用。	2020 年 12 月 1 日
Lambda 函數的程式碼簽署	Lambda 現在支援程式碼簽署。系統管理員可以將 Lambda 函數設定為只接受在部署時簽署的程式碼。Lambda 會檢查簽名，以確保程式碼不會遭到更改或竄改。此外，Lambda 可確保程式碼在接受部署之前，由受信任的開發人員簽署。如需詳細資訊，請參閱設定 Lambda 的程式碼簽署。	2020 年 11 月 23 日
預覽：Lambda Runtime Logs API	Lambda 現在支援 Runtime Logs API。Lambda 擴展可以使用 Logs API 來訂閱執行環境中的記錄串流。如需詳細資訊，請參閱 Lambda Runtime Logs API。	2020 年 11 月 12 日

Amazon MQ 的新事件來源	Lambda 現在支援 Amazon MQ 作為事件來源。使用 Lambda 函數來處理您的 Amazon MQ 訊息代理程式中的記錄。如需詳細資訊，請參閱搭配使用 Lambda 與 Amazon MQ。	2020 年 11 月 5 日
預覽：Lambda Extensions API	使用 Lambda 擴展來增強 Lambda 函數。您可以使用 Lambda 合作夥伴提供的擴展，也可以建立自己的 Lambda 擴展。如需詳細資訊，請參閱 Lambda Extensions API。	2020 年 10 月 8 日
AL2 支援 Java 8 和自訂執行期	Lambda 目前在 Amazon Linux 2 中支援 Java 8 和自訂執行期。如需詳細資訊，請參閱 Lambda 執行期。	2020 年 8 月 12 日
Amazon Managed Streaming for Apache Kafka 的新事件來源	Lambda 現在支援 Amazon MSK 作為事件來源。搭配使用 Lambda 函數與 Amazon MSK 來處理 Kafka 主題中的記錄。如需詳細資訊，請參閱搭配使用 Lambda 與 Amazon MSK。	2020 年 8 月 11 日
Amazon VPC 設定的 IAM 條件金鑰	您現在可以針對 VPC 設定使用 Lambda 特定條件金鑰。例如，您可以請求組織中的所有功能都連線到 VPC。您也可以指定函數的使用者可以和不可以使用的子網路和安全性群組。如需詳細資訊，請參閱針對 IAM 函數設定 VPC。	2020 年 8 月 10 日

Kinesis HTTP/2 串流取用程式的並行設定	您現在可以針對 Kinesis 取用者使用下列並行設定，搭配增強型散發 (HTTP/2 串流)：ParallelizationFactor、、、MaximumRetryAttempts 和。 MaximumRecordAgeIn Seconds DestinationConfig BisectBatchOnFunctionError 如需詳細資訊，請參閱AWS Lambda 搭配 Amazon Kinesis 使用。	2020 年 7 月 7 日
Kinesis HTTP/2 串流取用程式的批次間隔	您現在可以為 HTTP/2 串流設定批次視窗 (MaximumB atchingWindowIn秒)。Lambda 會從串流中讀取記錄，直到收集到完整批次，或直到批次間隔到期。如需詳細資訊，請參閱AWS Lambda 搭配 Amazon Kinesis 使用。	2020 年 6 月 18 日
支援 Amazon EFS 檔案系統	您現在可以將 Amazon EFS 檔案系統連線到您的 Lambda 函數，以獲得共用網路檔案存取權。如需詳細資訊，請參閱設定 Lambda 函數的檔案系統存取權。	2020 年 6 月 16 日
AWS CDK Lambda 主控台的範例應用程式	Lambda 主控台現在包含使 AWS Cloud Development Kit (AWS CDK) 用 TypeScript. 這 AWS CDK 是一個架構，可讓您在 Python TypeScript、Java 或 .NET 中定義應用程式資源。	2020 年 6 月 1 日

Support 於 .NET 核心 3.1.0 執行階段的支援 AWS Lambda	AWS Lambda 現在支援 .NET 核心 3.1.0 執行階段。如需詳細資訊，請參閱 .NET Core CLI。	2020 年 3 月 31 日
支援 API Gateway HTTP API	更新和擴增文件以便搭配使用 Lambda 與 API Gateway，包括對 HTTP API 的支援。已新增使用建立 API 和函數的範例應用程式 AWS CloudFormation。如需詳細資訊，請參閱搭配使用 Lambda 與 Amazon API Gateway。	2020 年 3 月 23 日
Ruby 2.7	Ruby 2.7 有新的執行期可用，ruby2.7 是第一個可使用 Amazon Linux 2 的 Ruby 執行期。如需詳細資訊，請參閱使用 Ruby 建立 Lambda 函數。	2020 年 2 月 19 日
並行指標	Lambda 現在會針對所有函數、別名和版本來報告 `ConcurrentExecutions` 指標。您可以在函式的監督頁面檢視此指標的圖表。先前，`ConcurrentExecutions` 只會報告帳戶層級和使用保留並行的函式。如需詳細資訊，請參閱 AWS Lambda 函數指標。	2020 年 2 月 18 日

函數狀態更新	預設情況下，所有函數皆會強制執行函數狀態。當您將函數連線到 VPC 時，Lambda 會建立共用的彈性網路介面。如此一來，您的函數即可在不建立其他網路介面的情況下輕鬆擴展規模。在此期間，您無法對函數執行其他操作，包括更新其組態和發佈版本。在某些情況下，調用也會受到影響。關於函數目前狀態的詳細資訊可從 Lambda API 中獲得。 此更新將分階段發佈。如需詳細資訊，請參閱 AWS 運算部落格上 VPC 網路的更新 Lambda 狀態生命週期。如需狀態的詳細資訊，請參閱 AWS Lambda 函數狀態。	2020 年 1 月 24 日
函數組態 API 輸出更新	為連線至 StateReason VPC 的函數新增原因代碼 LastUpdateStatusReasonCode (InvalidSubnet, InvalidSecurityGroup InvalidSecurityGroup) 和 (SubnetOutOFIP 位址,,)。 InvalidSubnet 如需狀態的詳細資訊，請參閱 AWS Lambda 函數狀態。	2020 年 1 月 20 日
佈建並行	您現在可以為函數版本或別名來配置佈建並行。佈建並行可讓函數在不造成延遲波動的情況下進行擴展。如需詳細資訊，請參閱管理 Lambda 函數的並行。	2019 年 12 月 3 日

建立資料庫代理	您現在可以使用 Lambda 主控台為 Lambda 函數建立資料庫代理。資料庫代理可讓函數在不耗盡資料庫連線的情況下達到高並行層級。如需詳細資訊，請參閱設定 Lambda 函數的資料庫存取權。	2019 年 12 月 3 日
持續時間指標支援百分位數	您現在可以根據百分位數來篩選持續時間指標。如需詳細資訊，請參閱 AWS Lambda 指標。	2019 年 11 月 26 日
增加串流事件來源的並行	DynamoDB 串流和 Kinesis 串流事件來源映射的新選項，可讓您從每個碎片一次處理多個批次。當您增加每個分片的並行批次數量時，函數的並行最高可以是串流中碎片數量的 10 倍。如需詳細資訊，請參閱 Lambda 事件來源映射。	2019 年 11 月 25 日
函數狀態	當您建立或更新函數時，函數會進入待定狀態，Lambda 會同時佈建資源以提供支援。如果您將函數連線至 VPC，Lambda 可立即建立共用的彈性網路介面，而不是在調用函數時建立網路介面。這可為連線至 VPC 的函數帶來較佳效能，但可能需要更新您的自動化。如需詳細資訊，請參閱 AWS Lambda 函數狀態。	2019 年 11 月 25 日

處理非同步調用選項時的錯誤	新的組態選項可用於非同步調用。您可以設定 Lambda，使其限制重試次數，並設定事件存留期的上限。如需詳細資訊，請參閱設定處理非同步調用時的錯誤。	2019 年 11 月 25 日
處理串流事件來源時的錯誤	有新的組態選項，可用於讀取自串流的事件來源映射。您可以設定 DynamoDB 串流和 Kinesis 串流事件來源映射以限制重試，並設定記錄保留期限的上限。發生錯誤時，您可以先將事件來源映射設定為分割批次，再進行重試，並將失敗批次的呼叫記錄傳送到佇列或主題。如需詳細資訊，請參閱 Lambda 事件來源映射。	2019 年 11 月 25 日
非同步調用的目的地	您現在可以設定 Lambda，使其將非同步調用的記錄傳送到另一個服務。呼叫記錄包含事件、內容和函數回應的詳細資訊。您可以將叫用記錄傳送至 SQS 佇列、SNS 主題、Lambda 函數或 EventBridge 事件匯流排。如需詳細資訊，請參閱設定非同步調用的目的地。	2019 年 11 月 25 日
適用於 Node.js、Python 和 Java 的新執行期	新的執行期可用於 Node.js 12、Python 3.8 和 Java 11。如需詳細資訊，請參閱 Lambda 執行期。	2019 年 11 月 18 日

FIFO 佇列支援 Amazon SQS 事件來源	您現在可以建立從先進先出 (FIFO) 佇列讀取內容的事件來源映射。先前，只支援標準佇列。如需詳細資訊，請參閱搭配使用 Lambda 與 Amazon SQS。	2019 年 11 月 18 日
在 Lambda 主控台中建立應用程式	現在通常可以在 Lambda 主控台中建立應用程式。如需指示，請參閱在 Lambda 主控台中管理應用程式。	2019 年 10 月 31 日
在 Lambda 主控台中建立應用程式 (Beta 版)	您現在可以在 Lambda 主控台中建立具有整合式持續交付管道的 Lambda 應用程式。主控台會提供範例應用程式，您可以使用這些應用程式做為自己專案的起點。在 AWS CodeCommit 和之間選擇 GitHub 用於原始檔控制。每次您將變更推送到儲存庫時，包含的管道便會自動建置和部署它們。如需指示，請參閱在 Lambda 主控台中管理應用程式。	2019 年 10 月 3 日

改善 VPC 連線函數的效能	Lambda 現在使用 Virtual Private Cloud (VPC)) 子網路中所有函數共享的新彈性網路介面類型。當您將函數連接到 VPC 時，Lambda 會為您選擇的每個安全群組和子網路組合建立網路介面。當有共享網路介面可供使用時，函數就不再需要在擴展時建立額外的網路介面。這大幅改善了啟動時間。如需詳細資訊，請參閱設定 Lambda 函數存取 VPC 中的資源。	2019 年 9 月 3 日
串流批次設定	您現在可以為 Amazon DynamoDB 和 Amazon Kinesis 事件來源映射設定批次時段。設定最多五分鐘的批次時段來緩衝傳入的記錄，直到完整批次可用為止。這可減少當串流較不活躍時調用函式的次數。	2019 年 8 月 29 日
CloudWatch 日誌洞察整合	Lambda 主控台中的監控頁面現在包含來自 Amazon CloudWatch 日誌洞察的報告。如需詳細資訊，請參閱AWS Lambda 主控台中的監視功能。	2019 年 6 月 18 日
Amazon Linux 2018.03	正在更新 Lambda 執行環境以使用 Amazon Linux 2018.03。如需詳細資訊，請參閱執行環境。	2019 年 5 月 21 日

Node.js 10	適用於 Node.js 10、nodejs 10.x. 的新執行期 這個執行期使用 Node.js 10.15，且系統會定期使用最新版的 Node.js 10 來更新此執行期。Node.js 10 也是第一個使用 Amazon Linux 2 的執行期。如需詳細資訊，請參閱使用 Node.js 建立 Lambda 函數。	2019 年 5 月 13 日
GetLayerVersionByArn API	使用 GetLayerVersionByArn API 下載圖層版本資訊，並以 ARN 版本作為輸入。相比之下 GetLayerVersion，您 GetLayerVersionByArn 可以直接使用 ARN，而不是剖析它來取得圖層名稱和版本號碼。	2019 年 4 月 25 日
Ruby	AWS Lambda 現在支援使用新執行階段的 Ruby 2.5。如需詳細資訊，請參閱使用 Ruby 建立 Lambda 函數。	2018 年 11 月 29 日
圖層	使用 Lambda 層，您可以單獨封裝和部署程式庫、自訂執行期及其他依存項目，與您的函數程式碼分開。與您的其他帳戶甚至世界各地的人共享您的 Layer。如需詳細資訊，請參閱 Lambda 層。	2018 年 11 月 29 日
自訂執行期	使用您慣用的程式設計語言建置自訂執行期以執行 Lambda 函數。如需詳細資訊，請參閱自訂 Lambda 執行期。	2018 年 11 月 29 日

Application Load Balancer 觸發	Elastic Load Balancing 現在支援 Lambda 函數作為 Application Load Balancer 的目標。如需詳細資訊，請參閱搭配使用 Lambda 與 Application Load Balancer。	2018 年 11 月 29 日
使用 Kinesis HTTP/2 串流消費者作為觸發條件	您可以使用 Kinesis HTTP/2 資料串流消費者，將事件傳送至 AWS Lambda。串流消費者有來自資料串流中每個碎片的專屬讀取傳輸量，並使用 HTTP/2 將延遲降至最低。如需詳細資訊，請參閱搭配使用 Lambda 與 Kinesis。	2018 年 11 月 19 日
Python 3.7	AWS Lambda 現在支援具有新執行階段的 Python 3.7。如需詳細資訊，請參閱使用 Python 建置 Lambda 函數。	2018 年 11 月 19 日
提高非同步函數呼叫的承載限制	非同步調用的承載大小上限已從 128 KB 提高至 256 KB，以符合來自 Amazon SNS 觸發的訊息大小上限。如需詳細資訊，請參閱 Lambda 配額。	2018 年 11 月 16 日
AWS GovCloud (美國東部) 區域	AWS Lambda 現在可在 AWS GovCloud (美國東部) 區域使用。	2018 年 11 月 12 日
將 AWS SAM 主題移至單獨的開發人員指南	許多主題著重於使用 AWS Serverless Application Model (AWS SAM) 建置無伺服器應用程式。這些主題已移至 AWS Serverless Application Model 開發人員指南。	2018 年 10 月 25 日

在主控台中檢視 Lambda 應用程式	您可以在 Lambda 主控台的應用程式頁面中檢視 Lambda 應用程式的狀態。此頁面顯示 AWS CloudFormation 堆疊的狀態。它包含多個頁面的連結，可讓您檢視有關堆疊中的資源的詳細資訊。您也可以檢視應用程式的彙總指標並建立自訂的監控儀表板。	2018 年 10 月 11 日
函數執行逾時限制	若要允許長時間執行的函數，可設定的最長執行逾時可從 5 分鐘增加為 15 分鐘。如需詳細資訊，請參閱 Lambda 限制。	2018 年 10 月 10 日
Support PowerShell 核心語言 AWS Lambda	AWS Lambda 現在支援 PowerShell 核心語言。如需詳細資訊，請參閱 PowerShell.	2018 年 9 月 11 日
Support 於 .NET 核心 2.1.0 執行階段的支援 AWS Lambda	AWS Lambda 現在支援 .NET 核心 2.1.0 執行階段。如需詳細資訊，請參閱 .NET Core CLI。	2018 年 7 月 9 日
現在可以透過 RSS 獲得更新	您現在可以訂閱 RSS 摘要，以追蹤本指南的發行版本。	2018 年 7 月 5 日
支援 Amazon SQS 作為事件來源	AWS Lambda 現在支援 Amazon Simple Queue Service (Amazon SQS) 做為事件來源。如需詳細資訊，請參閱調用 Lambda 函數。	2018 年 6 月 28 日
中國 (寧夏) 區域	AWS Lambda 現已在中國（寧夏）地區推出。如需有關 Lambda 區域和端點的詳細資訊，請參閱《AWS 一般參考》中的區域與端點。	2018 年 6 月 28 日

舊版更新

下表說明 2018 年六月前每個 AWS Lambda 開發人員指南版本的重要變更。

變更	描述	日期
執行期支援 Node.js 執行期 8.10	AWS Lambda 現在支援 Node.js 執行階段 8.10 版本。如需詳細資訊，請參閱 使用 Node.js 建置 Lambda 函數。	2018 年 4 月 2 日
函式和別名修訂 ID	AWS Lambda 現在支援函數版本和別名的修訂版 ID。在更新函式版本或別名資源時，您可以使用這些 ID 來追蹤和套用條件更新。	2018 年 1 月 25 日
Go 和 .NET 2.0 的執行期支援	AWS Lambda 已經添加了對 Go 和 .NET 2.0 的運行時支持。如需詳細資訊，請參閱 使用 Go 建置 Lambda 函數 及 使用 C# 建置 Lambda 函數。	2018 年 1 月 15 日
主控台重新設計	AWS Lambda 推出了全新的 Lambda 主控台來簡化您的體驗，並新增 Cloud9 程式碼編輯器來增強您的偵錯能力，並修改函數程式碼。如需詳細資訊，請參閱 使用 Lambda 主控台編輯器編輯代碼。	2017 年 11 月 30 日
為個別函式設定並行限制	AWS Lambda 現在支援對個別函數設定並行限制。如需詳細資訊，請參閱 為函數配置保留並發。	2017 年 11 月 30 日
使用別名轉移流量	AWS Lambda 現在支援使用別名轉移流量。如需詳細資訊，請參閱 為 Lambda 函數建立滾動式部署。	2017 年 11 月 28 日
逐步程式碼部署	AWS Lambda 現在支援利用程式碼部署，安全地部署新版 Lambda 函數。如需詳細資訊，請參閱逐步程式碼部署。	2017 年 11 月 28 日
中國 (北京) 區域	AWS Lambda 現已在中國（北京）地區提供。如需有關 Lambda 區域和端點的詳細資訊，請參閱《AWS 一般參考》中的區域與端點。	2017 年 11 月 9 日

變更	描述	日期
介紹 SAM Local	AWS Lambda 引進 SAM Local (現稱為 SAM CLI)，這是一種 AWS CLI 工具，可讓您在本機開發、測試和分析無伺服器應用程式，然後再將它們上傳到 Lambda 執行階段。如需詳細資訊，請參閱測試與偵錯無伺服器應用程式。	2017 年 8 月 11 日
加拿大 (中部) 區域	AWS Lambda 現已在加拿大 (中部) 區域推出。如需有關 Lambda 區域和端點的詳細資訊，請參閱《AWS 一般參考》中的區域與端點。	2017 年 6 月 22 日
南美洲 (聖保羅) 區域	AWS Lambda 現已在南美洲 (聖保羅) 地區推出。如需有關 Lambda 區域和端點的詳細資訊，請參閱《AWS 一般參考》中的區域與端點。	2017 年 6 月 6 日
AWS Lambda 支援 AWS X-Ray.	Lambda 推出對 X-Ray 的支援，它許您針對 Lambda 應用程式偵測、分析和最佳化效能問題。如需詳細資訊，請參閱 使用視覺化 Lambda 函數叫用 AWS X-Ray。	2017 年 4 月 19 日
亞太 (孟買) 區域	AWS Lambda 亞太區域 (孟買) 區域現已推出。如需有關 Lambda 區域和端點的詳細資訊，請參閱《AWS 一般參考》中的區域與端點。	2017 年 3 月 28 日
AWS Lambda 現在支援 Node.js 執行階段	AWS Lambda 增加了對 Node.js 運行時 v6.10 的支持。如需詳細資訊，請參閱 使用 Node.js 建置 Lambda 函數。	2017 年 3 月 22 日
歐洲 (倫敦) 區域	AWS Lambda 歐洲 (倫敦) 地區現已推出。如需有關 Lambda 區域和端點的詳細資訊，請參閱《AWS 一般參考》中的區域與端點。	2017 年 2 月 1 日
AWS Lambda 支援 .NET 執行階段、Lambda @Edge (預覽)、無效字母佇列，以及無伺服器應用程式的自動化部署。	AWS Lambda 增加了對 C# 的支持。如需詳細資訊，請參閱 使用 C# 建置 Lambda 函數。 Lambda @Edge 可讓您在節 AWS 點執行 Lambda 函數，以回應 CloudFront 事件。如需詳細資訊，請參閱 AWS Lambda搭配使用 CloudFront Lambda @Edge。	2016 年 12 月 3 日

變更	描述	日期
AWS Lambda 將 Amazon Lex 新增為支援的事件來源。	使用 Lambda 和 Amazon Lex，您便能針對 Slack 及 Facebook 之類的各種服務，快速建置聊天機器人。如需詳細資訊，請參閱 搭配使用 AWS Lambda 與 Amazon Lex。	2016 年 11 月 30 日
美國西部 (加利佛尼亞北部) 區域	AWS Lambda 現在可在美國西部 (加利佛尼亞北部) 區域使用。如需有關 Lambda 區域和端點的詳細資訊，請參閱《AWS 一般參考》中的區域與端點。	2016 年 11 月 21 日
介紹了用 AWS SAM 於建立和部署 Lambda 應用程式，以及使用 Lambda 函數組態設定的環境變數。	AWS SAM：您現在可以使用 AWS SAM 來定義在無伺服器應用程式中表示資源的語法。為了部署應用程式，只需指定需要的資源當做應用程式的一部分，以及於 AWS CloudFormation 範本檔案中與其相關聯的許可政策 (使用 JSON 或 YAML 撰寫)，封裝部署成品並部署範本。如需詳細資訊，請參閱 AWS Lambda 應用。 環境變數：您可以使用環境變數來指定您的函數程式碼外的 Lambda 函數的組態設定。如需詳細資訊，請參閱 使用 Lambda 環境變數來設定程式碼中的值。	2016 年 11 月 18 日
亞太 (首爾) 區域	AWS Lambda 亞太區域 (首爾) 區域現已推出。如需有關 Lambda 區域和端點的詳細資訊，請參閱《AWS 一般參考》中的區域與端點。	2016 年 8 月 29 日
亞太 (雪梨) 區域	亞太區域 (雪梨) 現在可以使用 Lambda。如需有關 Lambda 區域和端點的詳細資訊，請參閱《AWS 一般參考》中的區域與端點。	2016 年 6 月 23 日
更新至 Lambda 主控台	已更新 Lambda 主控台來簡化角色建立流程。	2016 年 6 月 23 日
AWS Lambda 現在支援 Node.js 執行階段 4.3 版	AWS Lambda 增加了對 Node.js 運行時版 4.3 的支持。如需詳細資訊，請參閱 使用 Node.js 建置 Lambda 函數。	2016 年 4 月 7 日
歐洲 (法蘭克福) 區域	歐洲 (法蘭克福) 區域現在可以使用 Lambda。如需有關 Lambda 區域和端點的詳細資訊，請參閱《AWS 一般參考》中的區域與端點。	2016 年 3 月 14 日

變更	描述	日期
VPC 支援	您現在可以設定 Lambda 函數來存取 VPC 中的資源。如需詳細資訊，請參閱 讓 Lambda 函數存取 Amazon VPC 中的資源。	2016 年 2 月 11 日
Lambda 執行期已更新。	已更新執行環境。	2015 年 11 月 4 日
版本控制支援、用於開發 Lambda 函數程式碼的 Python、已排程事件，以及增加執行期	您現在可以使用 Python 開發您的 Lambda 函式程式碼。如需詳細資訊，請參閱 使用 Python 建置 Lambda 函數。 版本控制：您可以維護一個或多個 Lambda 函式的版本。版本控制功能可以控制讓哪個 Lambda 函式版本在不同的環境中執行 (例如，開發、測試或生產)。如需詳細資訊，請參閱 Lambda 函數版本。 已排程事件：您也可使用 Lambda 主控台來設定 Lambda，以便在排程時間定期調用您的程式碼。您可以指定固定頻率 (時數、日數或週數)，或是指定一個 Cron 表達式。如需詳細資訊，請參閱 使用 Lambda 與 Amazon EventBridge 排程。 增加執行期：您現在可以設定 Lambda 函式執行五分鐘，允許更長的函式執行期，例如大量資料擷取和處理工作。	2015 年 10 月 8 日
支援 DynamoDB Streams	現在普遍都可使用 DynamoDB Streams，而且您可以在所有提供 DynamoDB 的地區使用它。您可以為您的資料表啟用 DynamoDB Streams，並使用 Lambda 函數當做資料表的觸發。觸發是您回應 DynamoDB 資料表更新所採取的自訂動作。如需範例演練，請參閱 教學課程： AWS Lambda 與 Amazon DynamoDB 串流搭配使用。	2015 年 7 月 14 日

變更	描述	日期
Lambda 現在支援使用與 REST 相容的用戶端調用 Lambda 函數。	到目前為止，若要從 Web、行動裝置或 IoT 應用程式叫用 Lambda 函數，您需要 AWS 開 AWS 發套件 (例如，Java 開發套件、適用於 Android 的 AWS 開發 AWS 套件或 iOS 版 SDK)。現在，Lambda 支援透過自訂的 API (您可使用 Amazon API Gateway 建立)，使用與 REST 相容的用戶端調用 Lambda 函數。您可以將請求傳送到 Lambda 函式端點 URL。您可以在端點上設定安全性，以允許開放存取，利用 AWS Identity and Access Management (IAM) 授權存取，或使用 API 金鑰來測量其他人對您的 Lambda 函數的存取。 如需入門練習範例，請參閱 使用 Amazon API Gateway 端點叫用 Lambda 函數。 如需 Amazon API Gateway 的詳細資訊，請參閱 https://aws.amazon.com/api-gateway/。	2015 年 7 月 9 日
Lambda 主控台現在提供藍圖，可輕鬆建立 Lambda 函數並進行測試。	Lambda 主控台提供一組藍圖。每個藍圖為您的 Lambda 函式提供範例事件來源組態和範本程式碼，而該函式則是用於輕鬆建立以 Lambda 為基礎的應用程式。所有的 Lambda 入門練習現在皆使用藍圖。如需詳細資訊，請參閱 開始使用 Lambda。	2015 年 7 月 9 日
Lambda 現在支援用 Java 撰寫 Lambda 函數。	您現在可以使用 Java 撰寫 Lambda 程式碼。如需詳細資訊，請參閱 使用 Java 建置 Lambda 函數。	2015 年 6 月 15 日
Lambda 現在支援在建立或更新 Lambda 函數時，指定 Amazon S3 物件作為函數 .zip 檔。	您可以將 Lambda 函式部署套件 (.zip 檔案) 上傳至相同區域內的 Amazon S3 儲存貯體，該處是您想要建立 Lambda 函式所在。然後，在建立或更新 Lambda 函式時，指定儲存貯體名稱和物件金鑰名稱。	2015 年 5 月 28 日

變更	描述	日期
Lambda 現在已普遍可用，並已增加行動後端的支援	Lambda 現在已普遍適用於生產用途。此版本也推出了新功能，可使用 Lambda 更輕鬆建置行動裝置、平板電腦及物聯網 (IoT) 後端，無需佈建或管理基礎設施便可自動擴展。Lambda 現在支援即時 (同步) 與非同步事件。其他功能包括更簡單的事件來源組態和管理。許可模型和程式設計模型已因推出 Lambda 函式適用的資源政策而更加簡化。 說明文件已隨之更新。如需詳細資訊，請參閱以下主題： 開始使用 Lambda AWS Lambda	2015 年 4 月 9 日
預覽版	AWS Lambda 開發人員指南的預覽版。	2014 年 11 月 13 日

本文為英文版的機器翻譯版本，如內容有任何歧義或不一致之處，概以英文版為準。