開發人員指南

AWS Cloud Development Kit (AWS CDK) V2

版本 2

AWS Cloud Development Kit (AWS CDK) V2: 開發人員指南

Table of Contents

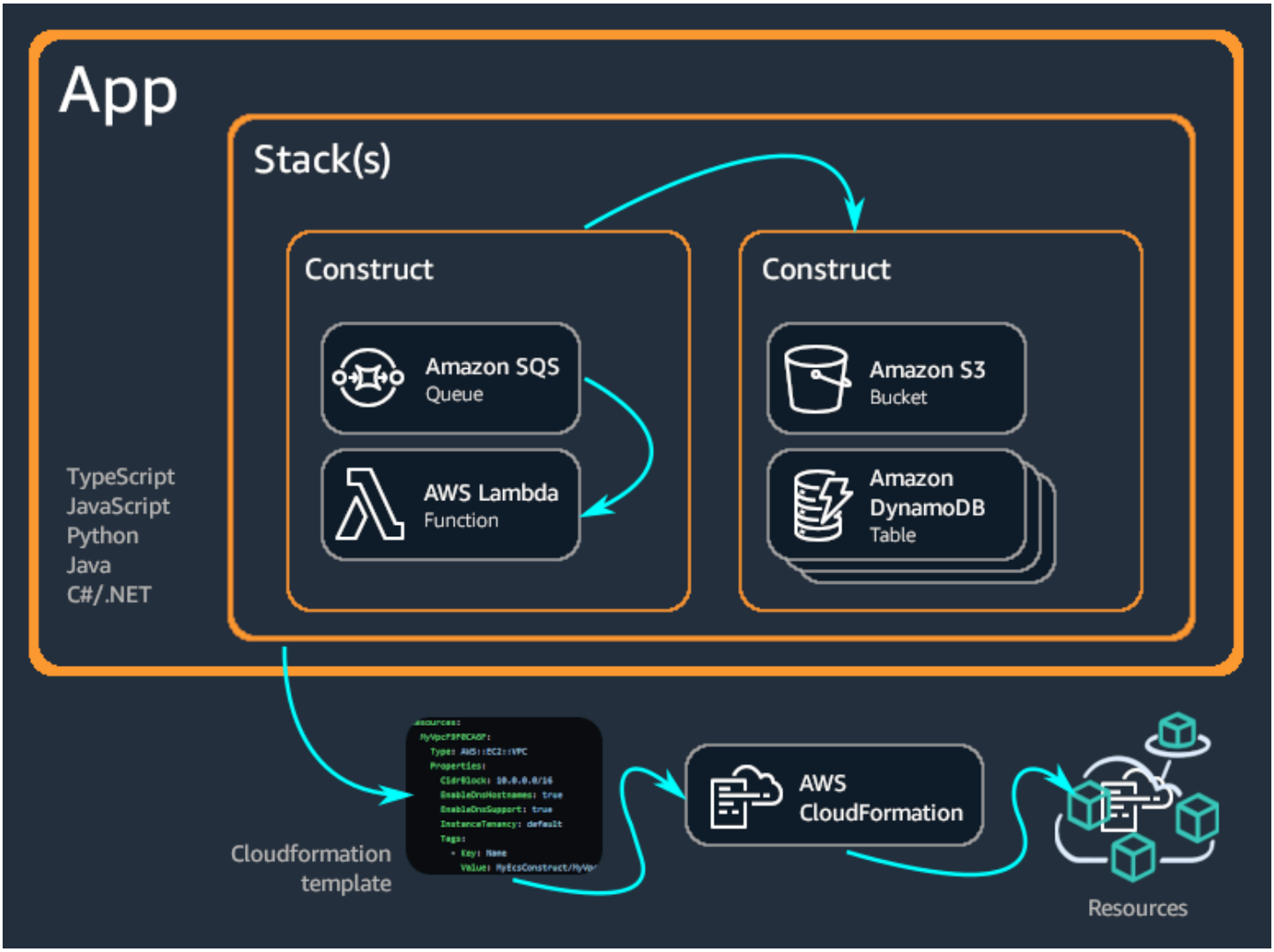
什麼是 AWS CDK？

這 AWS Cloud Development Kit (AWS CDK) 是一個開放原始碼軟體開發架構，用於在程式碼中定義雲端基礎架構，並透過 AWS CloudFormation.

AWS CDK 由兩個主要部分組成：

- AWS CDK 建構程式庫 — 預先撰寫的模組化且可重複使用的程式碼集合 (稱為建構)，您可以使用、修改和整合以快速開發基礎結構。 AWS CDK 建構程式庫的目標是降低在建置應用程式時定義和整合 AWS 服務所需的複雜性 AWS。
- AWS CDK 工具包 — 與 CDK 應用程序交互的命令行工具。使用工 AWS CDK 具組來建立、管理和部署您的 AWS CDK 專案。

支 AWS CDK 撐TypeScript、JavaScriptPython、Java、C#/.Net、和Go。您可以使用任何這些支援的程式設計語言來定義稱為建構可重複使用的雲端元件。您可以將這些組合成堆疊和應用程式。然後，您將 CDK 應用程式部署 AWS CloudFormation 到佈建或更新您的資源。



主題

的好處 AWS CDK

使用在雲端中 AWS CDK 開發可靠、可擴充且具成本效益的應用程式，並具備程式設計語言的顯著表現力。這種方法產生了許多好處，包括：

開發和管理您的基礎架構即程式碼 (IaC)

練習基礎結構即程式碼，以程式設計、描述性和宣告式的方式建立、部署和維護基礎結構。有了 IaC，您可以像開發人員處理程式碼相同的方式來處理基礎 這導致了一種可擴展的結構化方法來管理基礎結構。要了解有關 IaC 的更多信息，請參閱 AWS 白皮書簡介中的基礎結構即 DevOps 代碼。

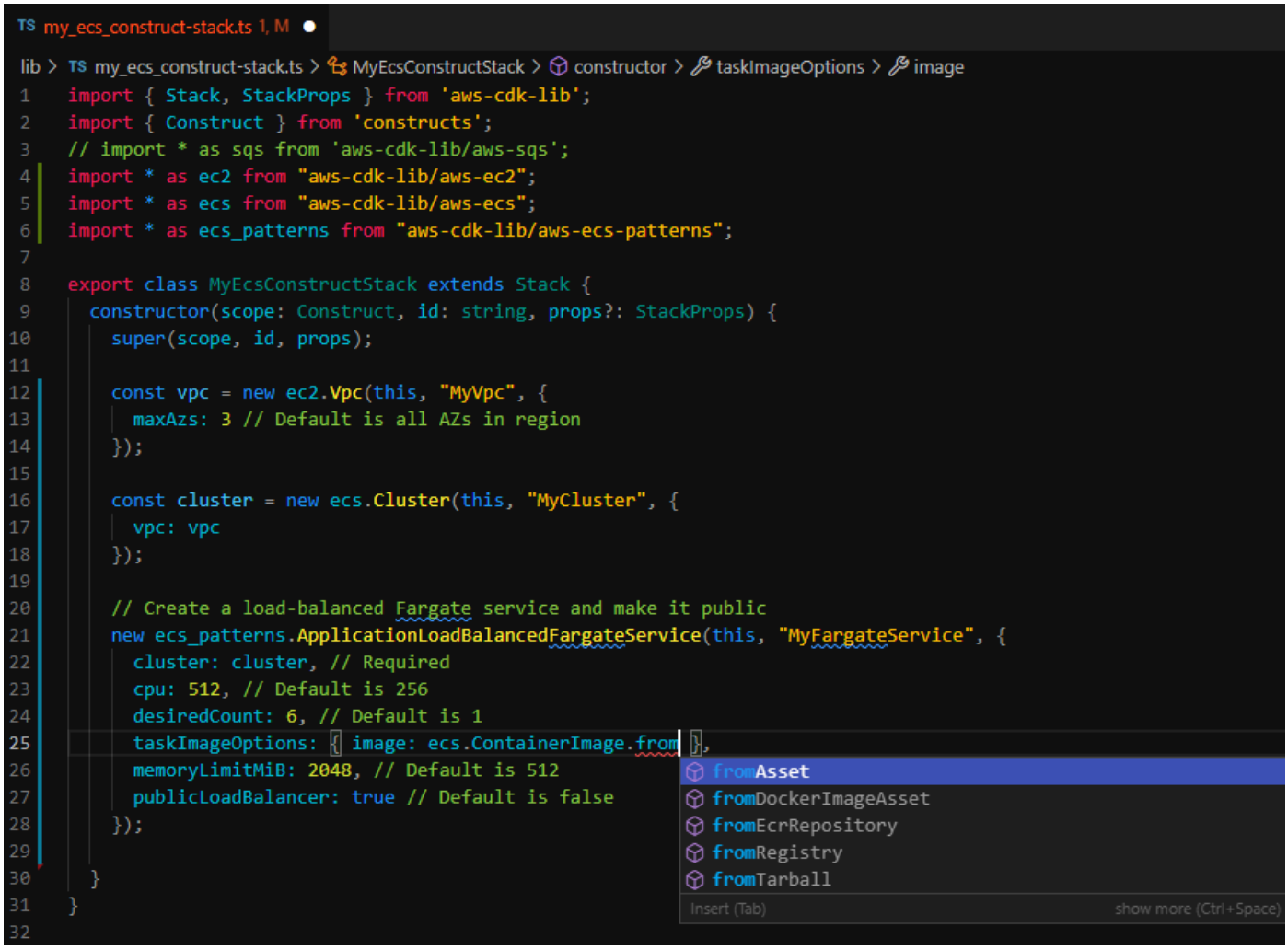
有了 AWS CDK，您可以將基礎結構、應用程式程式碼和組態集中在一個位置，確保您在每個里程碑都擁有完整、可雲端部署的系統。運用軟體工程最佳實務，例如程式碼檢閱、單元測試和原始檔控制，讓您的基礎架構更加穩固。

使用一般用途的程式設計語言定義雲端基礎架構

透過 AWS CDK，您可以使用下列任何一種程式設計語言來定義雲端基礎結構：TypeScriptJavaScriptPython、Java、C#/.Net、、和Go。選擇您偏好的語言，並使用參數、條件、迴圈、組合和繼承等程式設計元素來定義基礎結構所需的結果。

使用相同的程式設計語言來定義基礎結構和應用程式邏輯。

獲得在您偏好的 IDE (整合式開發環境) 中開發基礎結構的好處，例如語法醒目提示和智慧型程式碼完成。



透過部署基礎結 AWS CloudFormation

AWS CDK 與整合 AWS CloudFormation 以部署和佈建您的基礎架構 AWS。 AWS CloudFormation 是一個託管的 AWS 服務 ，提供對資源和屬性配置的廣泛支持，用於佈建服務 AWS。使用時 AWS CloudFormation，您可以預測和重複執行基礎結構部署，並發生錯誤的復原。如果您已經熟悉了 AWS CloudFormation，您不必在 AWS CDK開始使用時學習新的 IaC 管理服務。

使用建構快速開發應用程式

使用和共用稱為建構的可重複使用元件，加快開發速度。使用低階建構來定義個別 AWS CloudFormation 資源及其屬性。使用高階建構來快速定義應用程式的較大元件，為您的 AWS 資源提供合理且安全的預設值，以較少的程式碼定義更多的基礎結構。

建立您自己的建構，並針對您的獨特使用案例進行自訂，並在整個組織或甚至公眾共用這些建構。

的例子 AWS CDK

以下是使用建 AWS CDK 構程式庫建立具有 AWS Fargate (Fargate) 啟動類型的 Amazon Elastic Container Service (Amazon ECS) 服務的範例。如需此範例的詳細資訊，請參閱the section called “ECS”。

TypeScript

```
export class MyEcsConstructStack extends Stack {
  constructor(scope: App, id: string, props?: StackProps) {
    super(scope, id, props);

    const vpc = new ec2.Vpc(this, "MyVpc", {
      maxAzs: 3 // Default is all AZs in region
    });

    const cluster = new ecs.Cluster(this, "MyCluster", {
      vpc: vpc
    });

    // Create a load-balanced Fargate service and make it public
    new ecs_patterns.ApplicationLoadBalancedFargateService(this, "MyFargateService",
 {
      cluster: cluster, // Required
      cpu: 512, // Default is 256
      desiredCount: 6, // Default is 1
      taskImageOptions: { image: ecs.ContainerImage.fromRegistry("amazon/amazon-ecs-
sample") },
      memoryLimitMiB: 2048, // Default is 512
      publicLoadBalancer: true // Default is false
    });
  }
}
```

JavaScript

```
class MyEcsConstructStack extends Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    const vpc = new ec2.Vpc(this, "MyVpc", {
      maxAzs: 3 // Default is all AZs in region
```

```
    });

    const cluster = new ecs.Cluster(this, "MyCluster", {
      vpc: vpc
    });

    // Create a load-balanced Fargate service and make it public
    new ecs_patterns.ApplicationLoadBalancedFargateService(this, "MyFargateService",
 {
      cluster: cluster, // Required
      cpu: 512, // Default is 256
      desiredCount: 6, // Default is 1
      taskImageOptions: { image: ecs.ContainerImage.fromRegistry("amazon/amazon-ecs-
sample") },
      memoryLimitMiB: 2048, // Default is 512
      publicLoadBalancer: true // Default is false
    });
  }
}

module.exports = { MyEcsConstructStack }
```

Python

```
class MyEcsConstructStack(Stack):

    def __init__(self, scope: Construct, id: str, **kwargs) -> None:
        super().__init__(scope, id, **kwargs)

        vpc = ec2.Vpc(self, "MyVpc", max_azs=3)     # default is all AZs in region

        cluster = ecs.Cluster(self, "MyCluster", vpc=vpc)

        ecs_patterns.ApplicationLoadBalancedFargateService(self, "MyFargateService",
            cluster=cluster,            # Required
            cpu=512,                    # Default is 256
            desired_count=6,            # Default is 1
            task_image_options=ecs_patterns.ApplicationLoadBalancedTaskImageOptions(
                image=ecs.ContainerImage.from_registry("amazon/amazon-ecs-sample")),
            memory_limit_mib=2048,      # Default is 512
            public_load_balancer=True)  # Default is False
```

Java

```
public class MyEcsConstructStack extends Stack {

    public MyEcsConstructStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public MyEcsConstructStack(final Construct scope, final String id,
            StackProps props) {
        super(scope, id, props);

        Vpc vpc = Vpc.Builder.create(this, "MyVpc").maxAzs(3).build();

        Cluster cluster = Cluster.Builder.create(this, "MyCluster")
                .vpc(vpc).build();

        ApplicationLoadBalancedFargateService.Builder.create(this,
 "MyFargateService")
                .cluster(cluster)
                .cpu(512)
                .desiredCount(6)
                .taskImageOptions(
                       ApplicationLoadBalancedTaskImageOptions.builder()
                               .image(ContainerImage
                                       .fromRegistry("amazon/amazon-ecs-sample"))
                               .build()).memoryLimitMiB(2048)
                .publicLoadBalancer(true).build();
    }
}
```

C#

```
public class MyEcsConstructStack : Stack
{
    public MyEcsConstructStack(Construct scope, string id, IStackProps props=null) :
 base(scope, id, props)
    {
        var vpc = new Vpc(this, "MyVpc", new VpcProps
        {
            MaxAzs = 3
        });
```

```
        var cluster = new Cluster(this, "MyCluster", new ClusterProps
        {
            Vpc = vpc
        });

        new ApplicationLoadBalancedFargateService(this, "MyFargateService",
            new ApplicationLoadBalancedFargateServiceProps
        {
            Cluster = cluster,
            Cpu = 512,
            DesiredCount = 6,
            TaskImageOptions = new ApplicationLoadBalancedTaskImageOptions
            {
                Image = ContainerImage.FromRegistry("amazon/amazon-ecs-sample")
            },
            MemoryLimitMiB = 2048,
            PublicLoadBalancer = true,
        });
    }
}
```

Go

```
func NewMyEcsConstructStack(scope constructs.Construct, id string, props
 *MyEcsConstructStackProps) awscdk.Stack {

 var sprops awscdk.StackProps

 if props != nil {
  sprops = props.StackProps
 }

 stack := awscdk.NewStack(scope, &id, &sprops)

 vpc := awsec2.NewVpc(stack, jsii.String("MyVpc"), &awsec2.VpcProps{
  MaxAzs: jsii.Number(3), // Default is all AZs in region
 })

 cluster := awsecs.NewCluster(stack, jsii.String("MyCluster"), &awsecs.ClusterProps{
  Vpc: vpc,
 })
```

```
 awsecspatterns.NewApplicationLoadBalancedFargateService(stack,
 jsii.String("MyFargateService"),
  &awsecspatterns.ApplicationLoadBalancedFargateServiceProps{
   Cluster:        cluster,           // required
   Cpu:            jsii.Number(512),  // default is 256
   DesiredCount:   jsii.Number(5),    // default is 1
   MemoryLimitMiB: jsii.Number(2048), // Default is 512
   TaskImageOptions: &awsecspatterns.ApplicationLoadBalancedTaskImageOptions{
    Image: awsecs.ContainerImage_FromRegistry(jsii.String("amazon/amazon-ecs-
sample"), nil),
   },
   PublicLoadBalancer: jsii.Bool(true), // Default is false
  })

 return stack

}
```

這個類生成超過 500 行的 AWS CloudFormation 模板。部署 AWS CDK 應用程式會產生以下類型的 50 多種資源。

- AWS::EC2::EIP
- AWS::EC2::InternetGateway
- AWS::EC2::NatGateway
- AWS::EC2::Route
- AWS::EC2::RouteTable
- AWS::EC2::SecurityGroup
- AWS::EC2::Subnet
- AWS::EC2::SubnetRouteTableAssociation
- AWS::EC2::VPCGatewayAttachment
- AWS::EC2::VPC
- AWS::ECS::Cluster
- AWS::ECS::Service
- AWS::ECS::TaskDefinition
- AWS::ElasticLoadBalancingV2::Listener
- AWS::ElasticLoadBalancingV2::LoadBalancer

- AWS::ElasticLoadBalancingV2::TargetGroup
- AWS::IAM::Policy
- AWS::IAM::Role
- AWS::Logs::LogGroup

AWS CDK features

AWS CDKGitHub儲存庫

如需官方 AWS CDK GitHub儲存庫，請參閱 aws-cdk。在這裡，您可以提交問題，查看我們的許可證，跟踪版本等。

由於它 AWS CDK 是開源的，因此該團隊鼓勵您為使其成為更好的工具做出貢獻。如需詳細資訊，請參閱貢獻給 AWS Cloud Development Kit (AWS CDK).

AWS CDK API 參考資料

AWS CDK 建構程式庫提供 API 來定義您的 CDK 應用程式，並將 CDK 建構新增至應用程式。如需詳細資訊，請參閱 AWS CDK API 參考。

建構程式設計模型

構造編程模型（CPM）將背後的概念擴展 AWS CDK 到其他領域中。使用 CPM 的其他工具包括：

- 地形專用 CDK (CDK)
- 適用於庫伯尼特人的 CDK (CDK8s)
- 項目, 用於構建項目配置

構建樞紐

構造中心是一個在線註冊表，您可以在其中查找，發布和共享開源 AWS CDK 庫。

後續步驟

若要開始使用 AWS CDK，請參閱開始使用 AWS CDK。

進一步了解

若要繼續瞭解 AWS CDK，請參閱下列內容：

- AWS CDK 概念— 重要的概念和術語 AWS CDK.
- AWS CDK 工作坊 — 實踐工作坊，以學習和使用 AWS CDK.
- AWS CDK 模式 — 開放原始碼的 AWS 無伺服器架構模式集合， AWS CDK 由 AWS 專家打造。
- AWS CDK 代碼示例-示例 AWS CDK 項目的GitHub存儲庫。
- cdk.dev — 社區驅動的中心 AWS CDK，包括社區工作區。Slack
- 令人敬畏的 CDK — 包含 AWS CDK 開源項目，指南，博客和其他資源的精選列表的GitHub存儲庫。
- AWS 解決方案建構 — 經過審核的組態基礎架構即程式碼 (IaC) 模式，可輕鬆組裝到生產就緒應用程式中。
- AWS 開發人員工具部落格 — 已篩選的部落格文章 AWS CDK。
- AWS CDK 上 Stack Overflow — 用 aws-cd k 標記的問題。Stack Overflow
- AWS CDK 教學課程 AWS Cloud9 — 與 AWS Cloud9 開發環境 AWS CDK 搭配使用的教學課程。

若要進一步了解的相關主題 AWS CDK，請參閱下列內容：

- AWS CloudFormation 概念 — 由於專 AWS CDK 為搭配使用而建置 AWS CloudFormation，因此我們建議您學習並瞭解關鍵 AWS CloudFormation 概念。
- AWS 詞彙表 — 所用關鍵詞彙的定義 AWS。

若要深入了解與可用來簡化無伺服器應用程式開發和部署的相關工具，請參閱下列內容： AWS CDK

- AWS Serverless Application Model— 一種開放原始碼開發人員工具，可簡化並改善在上建置和執行無伺服器應用程式的 AWS體驗。
- AWSChalice-用於編寫無服務器應用程序的框架。Python

AWS CDK 概念

瞭解 AWS Cloud Development Kit (AWS CDK).

AWS CDK 和 IaC

這 AWS CDK 是一個開放原始碼架構，您可以使用程式碼來管理 AWS 基礎結構。這種方法被稱為基礎設施代碼（IaC）。透過以程式碼形式管理和佈建基礎結構，您可以像開發人員處理程式碼一樣處理基礎結構。這提供了許多好處，例如版本控制和可擴展性。要了解有關 IaC 的更多信息，請參閱什麼是基礎結構即代碼？

AWS CDK 而且 AWS CloudFormation

與 AWS CDK 緊密整合 AWS CloudFormation。 AWS CloudFormation 是一項完全受控的服務，可用來管理和佈建基礎結構 AWS。使用 AWS CloudFormation，您可以在範本中定義基礎結構並將其部署到 AWS CloudFormation. 然後， AWS CloudFormation 服務會根據範本上定義的組態佈建您的基礎結構。

AWS CloudFormation 模板是聲明性的，這意味著它們聲明了基礎結構的所需狀態或結果。使用 JSON 或 YAML，您可以定義 AWS 資源和屬性來宣告 AWS 基礎結構。資源代表上的許多服務， AWS 而屬性則代表您想要的這些服務組態。當您將範本部署到時 AWS CloudFormation，會依範本中所述佈建您的資源及其設定的屬性。

透過 AWS CDK，您可以使用一般用途的程式設計語言來管理基礎結構的命令。您可以定義達到所需狀態所需的邏輯或序列，而不是僅以宣告方式定義所需的狀態。例如，您可以使用`if`陳述式或條件迴圈來決定如何達到基礎結構所需的結束狀態。

使用建立的 AWS CDK 基礎結構最終會翻譯或合成為 AWS CloudFormation 範本，並使用 AWS CloudFormation 服務進行部署。因此，雖然 AWS CDK 提供了不同的方法來建立您的基礎結構，您仍然可以獲得的好處 AWS CloudFormation，例如廣泛的 AWS 資源配置支援和強大的部署程序。

若要深入瞭解 AWS CloudFormation，請參閱「什麼是 AWS CloudFormation？」 在《AWS CloudFormation 使用者指南》中。

AWS CDK 和抽象

使用時 AWS CloudFormation，您必須定義資源配置方式的每個細節。這提供了完全控制您的基礎架構的好處。但是，這需要您學習、瞭解和建立強大的範本，其中包含資源配置詳細資料以及資源之間的關係，例如權限和事件驅動的互動。

使用 AWS CDK，您可以對資源組態擁有相同的控制權。不過， AWS CDK 也提供強大的抽象概念，可加速並簡化基礎架構開發程序。例如，提供明智的 AWS CDK 預設組態的 include 建構，以及為您產生樣板程式碼的輔助程式方法。 AWS CDK 也提供工具，例如 AWS CDK 命令列介面 (AWS CDK CLI)，可為您執行基礎結構管理動作。

進一步了解核心 AWS CDK 概念

與互動 AWS CDK

與使用時 AWS CDK，您將主要與 AWS 建構程式庫和 AWS CDK CLI.

與開發 AWS CDK

AWS CDK 可以使用任何支援的程式設計語言來撰寫。您可以從 CDK 專案開始，該專案包含資料夾和檔案的結構，包括資產。在專案中，您可以建立 CDK 應用程式。在應用程序中，您可以定義一個堆棧，該堆棧直接表示 CloudFormation 堆棧。在堆疊中，您可以使用建構來定義 AWS 資源和屬性。

使用部署 AWS CDK

您可以將 CDK 應用程式部署到 AWS 環境中。在部署之前，您必須執行一次性啟動載入以準備環境。

進一步了解

若要進一步了解 AWS CDK 核心概念，請參閱本節中的主題。

支援的程式設計語言

對以下通用編程語言 AWS Cloud Development Kit (AWS CDK) 具有一流的支持：

- TypeScript
- JavaScript
- Python

- Java
- C#
- Go

理論上也可能使用其他JVM.NETCLR語言，但我們目前不提供官方支持。

> **Note**
>
> 本指南目前不包含指示或程式碼Go範例the section called “在圍棋”。

該 AWS CDK 開發在一種語言，TypeScript. 為了支持其他語言， AWS CDK 利用稱為JSII生成語言綁定的工具。

我們嘗試提供每種語言的慣例，以盡可能自然和直觀地進行開發。 AWS CDK 例如，我們會使用偏好語言的標準儲存庫來散佈「 AWS 建構程式庫」模組，然後您使用該語言的標準套件管理員來安裝它們。方法和屬性也會使用您語言的建議命名模式來命名。

以下是一些代碼示例：

TypeScript

```
const bucket = new s3.Bucket(this, 'MyBucket', {
  bucketName: 'my-bucket',
  versioned: true,
  websiteRedirect: {hostName: 'aws.amazon.com'}});
```

JavaScript

```
const bucket = new s3.Bucket(this, 'MyBucket', {
  bucketName: 'my-bucket',
  versioned: true,
  websiteRedirect: {hostName: 'aws.amazon.com'}});
```

Python

```
bucket = s3.Bucket("MyBucket", bucket_name="my-bucket", versioned=True,
            website_redirect=s3.RedirectTarget(host_name="aws.amazon.com"))
```

Java

```
Bucket bucket = Bucket.Builder.create(self, "MyBucket")
                      .bucketName("my-bucket")
                      .versioned(true)
                      .websiteRedirect(new RedirectTarget.Builder()
                          .hostName("aws.amazon.com").build())
                      .build();
```

C#

```
var bucket = new Bucket(this, "MyBucket", new BucketProps {
                      BucketName = "my-bucket",
                      Versioned  = true,
                      WebsiteRedirect = new RedirectTarget {
                              HostName = "aws.amazon.com"
                      }});
```

Go

```
bucket := awss3.NewBucket(scope, jsii.String("MyBucket"), &awss3.BucketProps {
 BucketName: jsii.String("my-bucket"),
 Versioned: jsii.Bool(true),
 WebsiteRedirect: &awss3.RedirectTarget {
  HostName: jsii.String("aws.amazon.com"),
 },
})
```

> **Note**
>
> 這些程式碼片段僅供說明之用。它們不完整，不會按原樣運行。

AWS 建構程式庫是使用每種語言的標準套件管理工具來散佈NPM，包括PyPiMaven、、和NuGet。我們也提供每種語言的 AWS CDK API 參考版本。

為了協助您以偏好 AWS CDK 的語言使用，本指南包含下列支援語言的主題：

- the section called “在 TypeScript”
- the section called “在 JavaScript”

- the section called “在 Python 中”
- the section called “在爪哇”
- the section called “在 C＃ 中”
- the section called “在圍棋”

TypeScript是支援的第一種語言 AWS CDK，而且大部分 AWS CDK 範例程式碼都是用來編寫的 TypeScript。本指南包含一個主題，專門說明如何調整TypeScript AWS CDK 程式碼，以便與其他支援的語言搭配使用。如需更多詳細資訊，請參閱 TypeScript與其他語言比 AWS CDK 較。

AWS CDK 項目

AWS Cloud Development Kit (AWS CDK) 專案代表包含 CDK 程式碼的檔案和資料夾。內容將根據您的編程語言而有所不同。

您可以手動建立 AWS CDK 專案，也可以使用指 AWS CDK 令行介面 (AWS CDK CLI) `cdk init` 指令建立專案。在本主題中，我們將參考 AWS CDK CLI 所建立之檔案和資料夾的專案結構和命名慣例。您可以自定義和組織 CDK 項目以滿足您的需求。

> **Note**
>
> 由建立的專案結構 AWS CDK CLI可能會隨著時間的推移而有所不同。

主題

- 通用檔案和資料夾
- 特定語言的檔案和資料夾

通用檔案和資料夾

.git

如果您已`git`安裝，則會 AWS CDK CLI自動為您的專案初始化Git儲存庫。目`.git`錄包含有關存放庫的資訊。

.gitignore

用Git來指定要忽略的檔案和資料夾的文字檔案。

README.md

文字檔案，提供您管理 AWS CDK 專案的基本指導和重要資訊。視需要修改此檔案，以記錄有關 CDK 專案的重要資訊。

cdk.json

的組態檔案 AWS CDK。該文件提供了 AWS CDK CLI有關如何運行您的應用程序的說明。

特定語言的檔案和資料夾

下列檔案和資料夾對於每種支援的程式設計語言都是唯一的。

TypeScript

以下是使用`cdk init --language typescript`指令在`my-cdk-ts-project`目錄中建立的範例專案：

```
my-cdk-ts-project
### .git
### .gitignore
### .npmignore
### README.md
### bin
#   ### my-cdk-ts-project.ts
### cdk.json
### jest.config.js
### lib
#   ### my-cdk-ts-project-stack.ts
### node_modules
### package-lock.json
### package.json
### test
#   ### my-cdk-ts-project.test.ts
### tsconfig.json
```

. N 米格諾

指定將套件發佈至npm登錄時要忽略哪些檔案和資料夾的檔案。此檔案類似於`.gitignore`，但是特定於npm套件。

斌 my-cdk-ts-project

應用程式檔案會定義您的 CDK 應用程式。CDK 專案可以包含一或多個應用程式檔案。應用程序文件存儲在文件bin夾中。

以下是定義 CDK 應用程式的基本應用程式檔案範例：

```
#!/usr/bin/env node
import 'source-map-support/register';
import * as cdk from 'aws-cdk-lib';
import { MyCdkTsProjectStack } from '../lib/my-cdk-ts-project-stack';

const app = new cdk.App();
new MyCdkTsProjectStack(app, 'MyCdkTsProjectStack');
```

傑斯特配置

的組態檔案Jest。 Jest是一種流行的JavaScript測試框架。

庫堆棧 my-cdk-ts-project

堆疊檔案會定義您的 CDK 堆疊。在堆疊中，您可以使用建構來定義 AWS 資源和屬性。

以下是定義 CDK 堆疊的基本堆疊檔案範例：

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';

export class MyCdkTsProjectStack extends cdk.Stack {
 constructor(scope: Construct, id: string, props?: cdk.StackProps) {
  super(scope, id, props);

  // code that defines your resources and properties go here
 }
}
```

節點模組 (_l)

專案中包含Node.js專案相依性的通用資料夾。

包-鎖定 .json

與檔案搭配使用以管理相依性版本的中繼資料package.json檔案。

包裝

專案中常用的中繼資料檔Node.js案。此檔案包含 CDK 專案的相關資訊，例如專案名稱、指令碼定義、相依性以及其他匯入專案層級資訊。

測試 my-cdk-ts-project

系統會建立測試資料夾來組織 CDK 專案的測試。也會建立範例測試檔案。

您可以在中編寫測試TypeScript並在運行測試之前使用Jest編譯TypeScript代碼。

tsconfig.json

在指定編譯器選項和項TypeScript目設置的項目中使用的配置文件。

JavaScript

以下是使用`cdk init --language javascript`指令在`my-cdk-js-project`目錄中建立的範例專案：

```
my-cdk-js-project
### .git
### .gitignore
### .npmignore
### README.md
### bin
#   ### my-cdk-js-project.js
### cdk.json
### jest.config.js
### lib
#   ### my-cdk-js-project-stack.js
### node_modules
### package-lock.json
### package.json
### test
    ### my-cdk-js-project.test.js
```

. N 米格諾

指定將套件發佈至npm登錄時要忽略哪些檔案和資料夾的檔案。此檔案類似於`.gitignore`，但是特定於npm套件。

垃圾桶 my-cdk-js-project

應用程式檔案會定義您的 CDK 應用程式。CDK 專案可以包含一或多個應用程式檔案。應用程序文件存儲在文件bin夾中。

以下是定義 CDK 應用程式的基本應用程式檔案範例：

```
#!/usr/bin/env node

const cdk = require('aws-cdk-lib');
const { MyCdkJsProjectStack } = require('../lib/my-cdk-js-project-stack');

const app = new cdk.App();
new MyCdkJsProjectStack(app, 'MyCdkJsProjectStack');
```

傑斯特配置

的組態檔案Jest。 Jest是一種流行的JavaScript測試框架。

版本庫 my-cdk-js-project-stack.js

堆疊檔案會定義您的 CDK 堆疊。在堆疊中，您可以使用建構來定義 AWS 資源和屬性。

以下是定義 CDK 堆疊的基本堆疊檔案範例：

```
const { Stack, Duration } = require('aws-cdk-lib');

class MyCdkJsProjectStack extends Stack {
 constructor(scope, id, props) {
  super(scope, id, props);

  // code that defines your resources and properties go here
 }
}

module.exports = { MyCdkJsProjectStack }
```

節點模組 (_l)

專案中包含Node.js專案相依性的通用資料夾。

包-鎖定 .json

與檔案搭配使用以管理相依性版本的中繼資料package.json檔案。

包裝

專案中常用的中繼資料檔Node.js案。此檔案包含 CDK 專案的相關資訊，例如專案名稱、指令碼定義、相依性以及其他匯入專案層級資訊。

測試 my-cdk-js-project

系統會建立測試資料夾來組織 CDK 專案的測試。也會建立範例測試檔案。

您可以在中編寫測試JavaScript並在運行測試之前使用Jest編譯JavaScript代碼。

Python

以下是使用`cdk init --language python`指令在`my-cdk-py-project`目錄中建立的範例專案：

```
my-cdk-py-project
### .git
### .gitignore
### .venv
### README.md
### app.py
### cdk.json
### my_cdk_py_project
#   ### __init__.py
#   ### my_cdk_py_project_stack.py
### requirements-dev.txt
### requirements.txt
### source.bat
### tests
    ### __init__.py
    ### unit
```

.venv

CDK CLI 會自動為您的專案建立虛擬環境。目`.venv`錄參照此虛擬環境。

app.py

應用程式檔案會定義您的 CDK 應用程式。CDK 專案可以包含一或多個應用程式檔案。

以下是定義 CDK 應用程式的基本應用程式檔案範例：

```
#!/usr/bin/env python3
```

```
import os

import aws_cdk as cdk

from my_cdk_py_project.my_cdk_py_project_stack import MyCdkPyProjectStack

app = cdk.App()
MyCdkPyProjectStack(app, "MyCdkPyProjectStack")

app.synth()
```

我的 CDK 項目

包含堆疊檔案的目錄。CDK 在此處CLI創建以下內容：

- `__init__.py` — 空的Python套件定義檔案。
- `my_cdk_py_project`— 定義您的 CDK 堆疊的檔案。然後，您可以使用建構在堆疊中定義 AWS 資源和屬性。

以下是堆棧文件的示例：

```
from aws_cdk import Stack

from constructs import Construct

class MyCdkPyProjectStack(Stack):
 def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
  super().__init__(scope, construct_id, **kwargs)

  # code that defines your resources and properties go here
```

requirements-dev.txt

文件類似`requirements.txt`，但用於管理依賴關係專門用於開發目的，而不是生產。

requirements.txt

在專案中Python用來指定和管理專案相依性的通用檔案。

source.bat

用於設置Python虛擬環境的 Batch 文件。Windows

測試

包含 CDK 專案測試的目錄。

以下是單元測試的示例：

```
import aws_cdk as core
import aws_cdk.assertions as assertions

from my_cdk_py_project.my_cdk_py_project_stack import MyCdkPyProjectStack

def test_sqs_queue_created():
 app = core.App()
 stack = MyCdkPyProjectStack(app, "my-cdk-py-project")
 template = assertions.Template.from_stack(stack)

 template.has_resource_properties("AWS::SQS::Queue", {
  "VisibilityTimeout": 300
 })
```

Java

以下是使用`cdk init --language java`指令在`my-cdk-java-project`目錄中建立的範例專案：

```
my-cdk-java-project
### .git
### .gitignore
### README.md
### cdk.json
### pom.xml
### src
    ### main
    ### test
```

pom.xml

包含 CDK 專案的設定資訊和中繼資料的檔案。此檔案是的一部分Maven。

src /主要

包含應用程式和堆疊檔案的目錄。

以下是應用程式檔案範例：

```
package com.myorg;
```

```
import software.amazon.awscdk.App;
import software.amazon.awscdk.Environment;
import software.amazon.awscdk.StackProps;

import java.util.Arrays;

public class MyCdkJavaProjectApp {
 public static void main(final String[] args) {
  App app = new App();

  new MyCdkJavaProjectStack(app, "MyCdkJavaProjectStack", StackProps.builder()
   .build());

  app.synth();
 }
}
```

以下是一個示例堆棧文件：

```
package com.myorg;

import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;

public class MyCdkJavaProjectStack extends Stack {
 public MyCdkJavaProjectStack(final Construct scope, final String id) {
  this(scope, id, null);
 }

 public MyCdkJavaProjectStack(final Construct scope, final String id, final
 StackProps props) {
  super(scope, id, props);

  // code that defines your resources and properties go here
 }
}
```

src /測試

包含測試檔案的目錄。以下是範例：

```
package com.myorg;
```

```
import software.amazon.awscdk.App;
import software.amazon.awscdk.assertions.Template;
import java.io.IOException;

import java.util.HashMap;

import org.junit.jupiter.api.Test;

public class MyCdkJavaProjectTest {

 @Test
 public void testStack() throws IOException {
  App app = new App();
  MyCdkJavaProjectStack stack = new MyCdkJavaProjectStack(app, "test");

  Template template = Template.fromStack(stack);

  template.hasResourceProperties("AWS::SQS::Queue", new HashMap<String, Number>()
 {{
   put("VisibilityTimeout", 300);
  }});
 }
}
```

C#

以下是使用`cdk init --language csharp`指令在`my-cdk-csharp-project`目錄中建立的範例專案：

```
my-cdk-csharp-project
### .git
### .gitignore
### README.md
### cdk.json
### src
    ### MyCdkCsharpProject
    ### MyCdkCsharpProject.sln
```

資產管理系統/MyCdkCsharpProject

包含應用程式和堆疊檔案的目錄。

以下是應用程式檔案範例：

```
using Amazon.CDK;
using System;
using System.Collections.Generic;
using System.Linq;

namespace MyCdkCsharpProject
{
 sealed class Program
 {
  public static void Main(string[] args)
  {
   var app = new App();
   new MyCdkCsharpProjectStack(app, "MyCdkCsharpProjectStack", new StackProps{});
   app.Synth();
  }
 }
}
```

以下是一個示例堆棧文件：

```
using Amazon.CDK;
using Constructs;

namespace MyCdkCsharpProject
{
 public class MyCdkCsharpProjectStack : Stack
 {
  internal MyCdkCsharpProjectStack(Construct scope, string id, IStackProps props
 = null) : base(scope, id, props)
  {
   // code that defines your resources and properties go here
  }
 }
}
```

該目錄還包含以下內容：

- `GlobalSuppressions.cs`— 用於在專案中抑制特定編譯器警告或錯誤的檔案。
- `.csproj`— 基於 XML 的文件，用於定義項目設置，依賴關係和構建配置。

SRC /. SLN MyCdkCsharpProject

Microsoft Visual Studio Solution File用於組織和管理相關項目。

Go

以下是使用cdk init --language go指令在my-cdk-go-project目錄中建立的範例專案：

```
my-cdk-go-project
### .git
### .gitignore
### README.md
### cdk.json
### go.mod
### my-cdk-go-project.go
### my-cdk-go-project_test.go
```

去. 國防部

包含模塊信息，用於管理Go項目的依賴關係和版本控制的文件。

my-cdk-go-project. 走

定義您的 CDK 應用程式和堆疊的檔案。

以下是範例：

```
package main
import (
 "github.com/aws/aws-cdk-go/awscdk/v2"
 "github.com/aws/constructs-go/constructs/v10"
 "github.com/aws/jsii-runtime-go"
)

type MyCdkGoProjectStackProps struct {
 awscdk.StackProps
}

func NewMyCdkGoProjectStack(scope constructs.Construct, id string, props
 *MyCdkGoProjectStackProps) awscdk.Stack {
 var sprops awscdk.StackProps
 if props != nil {
  sprops = props.StackProps
```

```
 }
 stack := awscdk.NewStack(scope, &id, &sprops)
 // The code that defines your resources and properties go here

  return stack
}

func main() {
 defer jsii.Close()
 app := awscdk.NewApp(nil)
 NewMyCdkGoProjectStack(app, "MyCdkGoProjectStack", &MyCdkGoProjectStackProps{
  awscdk.StackProps{
   Env: env(),
  },
 })
 app.Synth(nil)
}

func env() *awscdk.Environment {

 return nil
}
```

my-cdk-go-project_ 測試

定義範例測試的檔案。

以下是範例：

```
package main

import (
 "testing"

 "github.com/aws/aws-cdk-go/awscdk/v2"
 "github.com/aws/aws-cdk-go/awscdk/v2/assertions"
 "github.com/aws/jsii-runtime-go"
)

func TestMyCdkGoProjectStack(t *testing.T) {

 // GIVEN
 app := awscdk.NewApp(nil)
```

```
 // WHEN
 stack := NewMyCdkGoProjectStack(app, "MyStack", nil)

 // THEN
 template := assertions.Template_FromStack(stack, nil)
 template.HasResourceProperties(jsii.String("AWS::SQS::Queue"),
 map[string]interface{}{
  "VisibilityTimeout": 300,
 })
}
```

AWS CDK 应用

應用 AWS Cloud Development Kit (AWS CDK) 程式或應用程式是一或多個 CDK 堆疊的集合。堆棧是一個或多個構造的集合，它們定義了 AWS 資源和屬性。因此，堆棧和構造的整體分組稱為 CDK 應用程序。

主題

- 定義應用
- 建構樹
- 應用生命週期

定義應用

您可以在專案的應用程式檔案中定義應用程式執行個體來建立應用程式。若要執行此操作，您可以從「App建構資料庫」匯入並使用 AWS 建構。該App構造不需要任何初始化參數。它是唯一可以用作根的構造。

建構程式庫中的App和Stack類別是唯一的 AWS 建構。與其他結構相比，它們不會自行配置 AWS 資源。相反，它們用於為您的其他構造提供上下文。所有代表 AWS 資源的建構都必須在Stack建構的範圍內直接或間接定義。 Stack建構是在App建構範圍內定義的。

然後合成應用程序以創建堆棧的 AWS CloudFormation 模板。以下是範例：

TypeScript

```
const app = new App();
new MyFirstStack(app, 'hello-cdk');
```

```
app.synth();
```

JavaScript

```
const app = new App();
new MyFirstStack(app, 'hello-cdk');
app.synth();
```

Python

```
app = App()
MyFirstStack(app, "hello-cdk")
app.synth()
```

Java

```
App app = new App();
new MyFirstStack(app, "hello-cdk");
app.synth();
```

C#

```
var app = new App();
new MyFirstStack(app, "hello-cdk");
app.Synth();
```

Go

```
app := awscdk.NewApp(nil)

MyFirstStack(app, "MyFirstStack", &MyFirstStackProps{
  awscdk.StackProps{
    Env: env(),
  },
})

app.Synth(nil)
```

單個應用程序中的堆棧可以輕鬆引用彼此的資源和屬性。這會 AWS CDK 推斷堆疊之間的相依性，以便以正確的順序部署它們。您可以使用單一`cdk deploy`命令在應用程式中部署任何或所有堆疊。

建構樹

建構是使用傳遞給每個建構的scope引數，以App類別做為根目錄，在其他建構中定義建構。通過這種方式， AWS CDK 應用程序定義了稱為構造樹的結構層次結構。

這棵樹的根是你的應用程序，它是App類的一個實例。在應用程序中，您實例化一個或多個堆棧。在堆棧中，您實例化構造，它們本身可以實例化資源或其他構造，依此類推。

建構一律會在另一個建構的範圍內明確定義，這會在建構之間建立關係。幾乎總是，您應該傳遞this（使用 Python，self）作為範圍，表明新構造是當前構造的子構造。預期的模式是您從中派生構造 `Construct`，然後在其構造函數中實例化它使用的構造。

明確傳遞範圍允許每個構造將自己添加到樹中，這種行為完全包含在`Construct基類`中。它在支援的每種語言中都以相同的方式運作，而 AWS CDK 且不需要額外的自訂。

> **Important**
>
> 從技術上講，除了實例化構造this時，可以傳遞一些範圍。您可以在樹中的任何位置添加構造，甚至可以在同一應用程序中的另一個堆棧中添加構造。例如，您可以編寫一個 mixin 樣式函數，將構造添加到作為參數傳入的範圍中。這裡的實際困難是，您無法輕鬆確保為構造選擇的 ID 在其他人的範圍內是唯一的。這種做法也會讓您的程式碼更難以理解、維護和重複使用。找到一種表達自己的意圖的方法幾乎總是更好，而不是訴諸濫用論點。scope

會 AWS CDK 使用從樹狀結構根目錄到每個子建構的路徑中所有建構的 ID 來產生所需的唯一 ID。AWS CloudFormation這種方法意味著構造 ID 只需要在其範圍內是唯一的，而不是像本機那樣在整個堆棧中 AWS CloudFormation。但是，如果將構造移動到不同的範圍，則其生成的堆棧唯一 ID 會更改，並且 AWS CloudFormation 不會將其視為相同的資源。

建構樹與您在 AWS CDK 程式碼中定義的建構是分開的。但是，它可以通過任何構造的node屬性訪問，該屬性是對表示樹中該構造的節點的引用。每個節點都是`Node`執行個體，其屬性可讓您存取樹狀目錄的根目錄以及節點的父範圍和子項。

1. `node.children`— 構造的直接子.
2. `node.id`— 建構在其範圍內的識別碼。
3. `node.path`— 建構的完整路徑，包括其所有父項的 ID。
4. `node.root`— 建構樹 (應用程式) 的根目錄。

5. node.scope— 建構的範圍 (父項)，如果節點是根，則未定義。
6. node.scopes— 建構的所有父項，直到根。
7. node.uniqueId— 樹狀結構內此建構的唯一英數識別碼 (依預設，產生自node.path雜湊)。

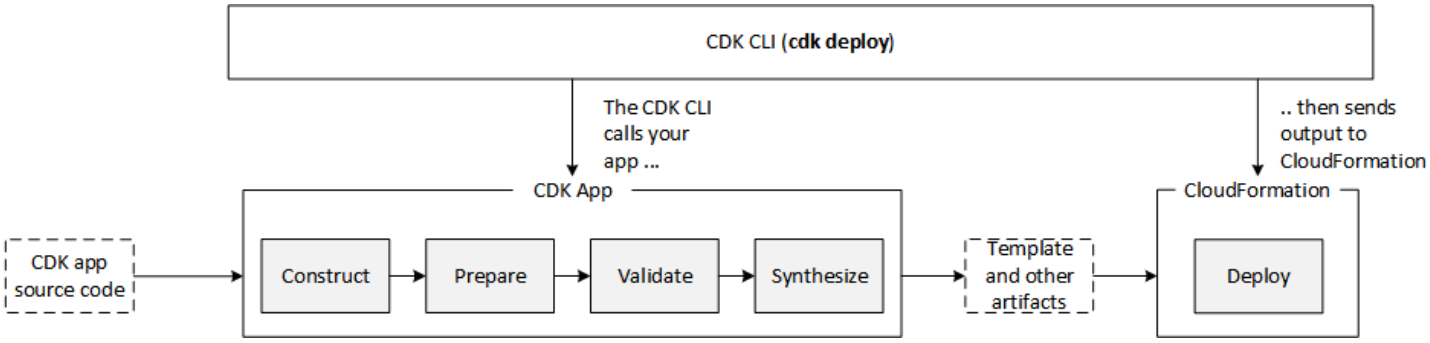
構造樹定義了一個隱含的順序，其中構造被合成到最終 AWS CloudFormation 模板中的資源。如果一個資源必須在另一個資源之前創建， AWS CloudFormation 或者 AWS 構造庫通常推斷依賴關係。然後，他們確保以正確的順序創建資源。

您還可以通過使用兩個節點之間添加明確的依賴關係node.addDependency()。如需詳細資訊，請參閱 AWS CDK API 參考中的相依性。

AWS CDK 提供了一種簡單的方法來訪問建構樹中的每個節點，並對每個節點執行操作。如需詳細資訊，請參閱 the section called “面向”。

應用生命週期

當您部署 CDK 應用程式時，會進行下列階段。這就是所謂的應用程序生命週期：



AWS CDK 應用程序在其生命週期中經歷以下階段。

- 構造（或初始化）-您的代碼實例化所有定義的構造，然後將它們鏈接在一起。在這個階段，所有的構造（應用程序，棧和它們的子構造）被實例化，並執行構造函數鏈。您的大多數應用程序代碼都在此階段執行。
- 準備 — 已實作該prepare方法的所有建構都會參與最後一輪修改，以設定其最終狀態。準備階段會自動發生。身為使用者，您看不到此階段的任何意見反應。很少需要使用「準備」掛鉤，通常不建議使用。在此階段變更建構樹時要非常小心，因為作業的順序可能會影響行為。
- 驗證 — 所有已實作該validate方法的建構都可以驗證自己，以確保它們處於正確部署的狀態。您將收到此階段發生的任何驗證失敗的通知。一般來說，我們建議盡快執行驗證（通常只要你得到一些輸入），並儘早拋出異常。儘早執行驗證可提高可靠性，因為堆棧跟踪將更加準確，並確保您的代碼可以繼續安全地執行。

- 合成-這是執行 AWS CDK 應用程序的最後階段。它是由調用觸發的`app.synth()`，它遍歷構造樹並在所有構造上調用該`synthesize`方法。實作的建構可`synthesize`以參與合成，並將部署成品發出至產生的雲端組件。這些成品包括 AWS CloudFormation 範本、 AWS Lambda 應用程式服務包、檔案和Docker影像資產，以及其他部署成品。 the section called “雲端組件”描述了這個階段的輸出。在大多數情況下，您不需要實現該`synthesize`方法。
- 部署 — 在這個階段中， AWS CDK CLI採用由綜合階段產生的部署成品雲端組件，並將其部署到 AWS 環境中。它會將資產上傳到 Amazon S3 和 Amazon ECR，或任何需要去的地方。然後，它會啟動 AWS CloudFormation 部署以部署應用程式並建立資源。

在 AWS CloudFormation 部署階段開始時，您的 AWS CDK 應用程式已完成並結束。這具有以下含義：

- 應用 AWS CDK 程式無法回應部署期間發生的事件，例如建立的資源或整個部署完成。若要在部署階段執行程式碼，您必須將其作為自訂資源插入 AWS CloudFormation 範本。有關向應用程序添加自定義資源的更多信息，請參閱AWS CloudFormation 模塊或自定義資源示例。
- 該 AWS CDK 應用程序可能必須使用在運行時無法知道的值。例如，如果 AWS CDK 應用程式使用自動產生的名稱定義 Amazon S3 儲存貯體，而您擷取 `bucket.bucketName` (Python:`bucket_name`) 屬性，則該值不是已部署儲存貯體的名稱。相反，你會得到一個`Token`價值。若要判斷特定值是否可用，請呼叫 `cdk.isUnresolved(value)` (Python:`is_unresolved`)。如需詳細資訊，請參閱 the section called “代幣”。

雲端組件

呼叫`app.synth()`是指示從應用程 AWS CDK 式合成雲端組件的原因。通常您不會直接與雲端組件互動。它們是包含將應用程序部署到雲環境所需的一切文件。例如，它包含應用程序中每個堆棧的 AWS CloudFormation 模板。它還包括您在應用程序中引用的任何文件資產或 Docker 圖像的副本。

如需如何格式化雲端組合的詳細資訊，請參閱雲端組合規格。

若要與應用 AWS CDK 程式建立的雲端組件互動，您通常會使用 AWS CDK CLI. 但是，任何可以讀取雲端組件格式的工具都可以用來部署您的應用程式。

執行應用程式

CDK CLI 需要知道如何執行您的 AWS CDK 應用程序。如果您使用該`cdk init`命令從模板創建項目，則應用程序的`cdk.json`文件包含一個`app`密鑰。這個鍵指定了寫入應用程序的語言的必要命令。如果您的語言需要編譯，命令行會在運行應用程序之前執行此步驟，因此您不會忘記這樣做。

TypeScript

```
{
  "app": "npx ts-node --prefer-ts-exts bin/my-app.ts"
}
```

JavaScript

```
{
  "app": "node bin/my-app.js"
}
```

Python

```
{
    "app": "python app.py"
}
```

Java

```
{
  "app": "mvn -e -q compile exec:java"
}
```

C#

```
{
  "app": "dotnet run -p src/MyApp/MyApp.csproj"
}
```

Go

```
{
  "app": "go mod download && go run my-app.go"
}
```

如果您沒有使用 CDK 創建項目CLI，或者您想覆蓋中給出的命令行cdk.json，則可以在發出cdk命令時使用該--app選項。

```
$ cdk --app 'executable' cdk-command ...
```

命令的#執行部分指示應運行以執行 CDK 應用程序的命令。使用引號，如圖所示，因為這些命令包含空格。*cdk-##*是一個子命令，synth或者告deploy訴 CDK 你想用你的應用程序做CLI什麼。請按照該子命令所需的任何其他選項進行操作。

也 AWS CDK CLI可以直接與已經合成的雲端組件互動。要做到這一點，傳遞雲程序集存儲在其中的目錄--app。以下範例列出儲存於下的雲端組合中定義的堆疊`./my-cloud-assembly`。

```
$ cdk --app ./my-cloud-assembly ls
```

堆疊

AWS Cloud Development Kit (AWS CDK) 堆疊是定義 AWS 資源的一個或多個建構的集合。每個 CDK 堆疊代表 CDK 應用程式中的一個 AWS CloudFormation 堆疊。在部署時，堆疊內的建構會佈建為單一單元 (稱為 AWS CloudFormation 堆疊)。若要深入瞭解 AWS CloudFormation 堆疊，請參閱使用AWS CloudFormation 者指南中的使用堆疊。

由於 CDK 堆疊是透過 AWS CloudFormation 堆疊實作的，因此適用 AWS CloudFormation 配額和限制。若要深入了解，請參閱AWS CloudFormation 配額。

主題

- 定義堆疊
- 使用堆疊

定義堆疊

堆疊是在應用程式內容中定義的。您可以使用「 AWS 建構程式庫」中的`Stack`類別來定義堆疊。您可以使用下列任何一種方式來定義堆疊：

- 直接在應用程序的範圍內。
- 間接由樹中的任何構造。

下列範例會定義包含兩個堆疊的 CDK 應用程式：

TypeScript

```
const app = new App();
```

```
new MyFirstStack(app, 'stack1');
new MySecondStack(app, 'stack2');

app.synth();
```

JavaScript

```
const app = new App();

new MyFirstStack(app, 'stack1');
new MySecondStack(app, 'stack2');

app.synth();
```

Python

```
app = App()

MyFirstStack(app, 'stack1')
MySecondStack(app, 'stack2')

app.synth()
```

Java

```
App app = new App();

new MyFirstStack(app, "stack1");
new MySecondStack(app, "stack2");

app.synth();
```

C#

```
var app = new App();

new MyFirstStack(app, "stack1");
new MySecondStack(app, "stack2");

app.Synth();
```

下列範例是在個別檔案上定義堆疊的常見模式。在這裡，我們擴展或繼承Stack類，並定義一個接受scopeid、和的構造函數props。然後，我們使用接收的scope、id和調super用基Stack類構造函數props。

TypeScript

```
class HelloCdkStack extends Stack {
  constructor(scope: App, id: string, props?: StackProps) {
    super(scope, id, props);

    //...
  }
}
```

JavaScript

```
class HelloCdkStack extends Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    //...
  }
}
```

Python

```
class HelloCdkStack(Stack):

    def __init__(self, scope: Construct, id: str, **kwargs) -> None:
        super().__init__(scope, id, **kwargs)

        # ...
```

Java

```
public class HelloCdkStack extends Stack {
    public HelloCdkStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public HelloCdkStack(final Construct scope, final String id, final StackProps
 props) {
```

```
        super(scope, id, props);

        // ...
    }
}
```

C#

```
public class HelloCdkStack : Stack
{
    public HelloCdkStack(Construct scope, string id, IStackProps props=null) :
 base(scope, id, props)
    {
        //...
    }
}
```

Go

```
func HelloCdkStack(scope constructs.Construct, id string, props *HelloCdkStackProps)
 awscdk.Stack {
 var sprops awscdk.StackProps
 if props != nil {
  sprops = props.StackProps
 }
 stack := awscdk.NewStack(scope, &id, &sprops)

  return stack
}
```

下列範例宣告名為的堆疊類別包`MyFirstStack`含單一 Amazon S3 儲存貯體。

TypeScript

```
class MyFirstStack extends Stack {
  constructor(scope: Construct, id: string, props?: StackProps) {
    super(scope, id, props);

    new s3.Bucket(this, 'MyFirstBucket');
  }
}
```

JavaScript

```
class MyFirstStack extends Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    new s3.Bucket(this, 'MyFirstBucket');
  }
}
```

Python

```
class MyFirstStack(Stack):

    def __init__(self, scope: Construct, id: str, **kwargs):
        super().__init__(scope, id, **kwargs)

        s3.Bucket(self, "MyFirstBucket")
```

Java

```
public class MyFirstStack extends Stack {
    public MyFirstStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public MyFirstStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        new Bucket(this, "MyFirstBucket");
    }
}
```

C#

```
public class MyFirstStack : Stack
{
    public MyFirstStack(Stack scope, string id, StackProps props = null) :
 base(scope, id, props)
    {
        new Bucket(this, "MyFirstBucket");
```

```
    }
}
```

Go

```
func MyFirstStack(scope constructs.Construct, id string, props *MyFirstStackProps)
 awscdk.Stack {
  var sprops awscdk.StackProps
  if props != nil {
    sprops = props.StackProps
  }
  stack := awscdk.NewStack(scope, &id, &sprops)

  s3.NewBucket(stack, jsii.String("MyFirstBucket"), &s3.BucketProps{})
  return stack
}
```

但是，此代碼只聲明了一個堆棧。為了使堆棧實際合成到 AWS CloudFormation 模板中並進行部署，必須實例化它。而且，像所有 CDK 結構一樣，它必須在某些情況下實例化。就App是上下文。

如果您使用的是標準 AWS CDK 開發模板，則堆棧將在實例化對象的同一文件中實例化。App

TypeScript

以專案資料夾中的專案 (例如`hello-cdk.ts`) 命名的檔`bin`案。

JavaScript

以專案資料夾中的專案 (例如`hello-cdk.js`) 命名的檔`bin`案。

Python

專案主目錄`app.py`中的檔案。

Java

名為的文件*`ProjectName`*`App.java`，例如`HelloCdkApp.java`，嵌套在`src/main`目錄下深處。

C#

例如`src\`*`ProjectName`*，`Program.cs`在下命名的檔案`src\HelloCdk\Program.cs`。

堆棧 API

堆棧對象提供了豐富的 API，包括以下內容：

- `Stack.of(construct)`— 傳回定義建構之 Stack 的靜態方法。如果您需要從可重複使用的構造中與堆棧進行交互，這非常有用。如果在範圍內找不到堆棧，則調用失敗。
- `stack.stackName`（Python：`stack_name`）-返回堆棧的物理名稱。如前所述，所有 AWS CDK 堆棧都具有 AWS CDK 可以在合成過程中解析的物理名稱。
- `stack.region`和 `stack.account` — 傳回 AWS 區域和帳戶，分別將部署此堆疊。這些屬性會傳回下列其中一項：
  - 定義堆疊時明確指定的帳戶或區域
  - 字符串編碼的令牌，解析為帳戶和 Region 的 AWS CloudFormation 虛擬參數，以指示此堆棧與環境無關

  如需如何判斷堆疊環境的相關資訊，請參閱the section called “環境”。
- `stack.addDependency(stack)`（Python: `stack.add_dependency(stack)`-可用於顯式定義兩個堆棧之間的依賴關係順序。一次部署多個堆疊時，cdk deploy命令會遵守此順序。
- `stack.tags`— 傳回可用來新增或移除堆疊層級標籤的一TagManager個。這個標籤管理器標記堆棧中的所有資源，並且在創建堆棧時也標記堆棧本身 AWS CloudFormation。
- `stack.partition`，`stack.urlSuffix`（Python:`url_suffix`），`stack.stackId`（Python:`stack_`和`stack.notificationArn`（Python:`notification_arn`）-返回解析為相應 AWS CloudFormation 虛擬參數的令牌，例如`{ "Ref": "AWS::Partition" }`。這些令牌與特定的堆棧對象相關聯，以便 AWS CDK 框架可以識別跨堆棧引用。
- `stack.availabilityZones`(Python:`availability_zones`) — 傳回部署此堆疊之環境中可用的一組可用區域。對於與環境無關的堆疊，這一定會傳回具有兩個可用區域的陣列。對於環境特定的堆疊，會 AWS CDK 查詢環境並傳回您指定之區域中可用的確切可用區域集。
- `stack.parseArn(arn)`和 `stack.formatArn(comps)` (Python:`parse_arn`,`format_arn`) — 可用於使用 Amazon 資源名稱 (ARN).
- `stack.toJsonString(obj)`(Python:`to_json_string`) — 可用於將任意物件格式化為可嵌入 AWS CloudFormation 範本中的 JSON 字串。該對象可以包括令牌，屬性和引用，這些標記和引用僅在部署期間解析。
- `stack.templateOptions`(Python:`template_options`) — 用來指定堆疊的 AWS CloudFormation 範本選項，例如「轉換」、「描述」和「中繼資料」。

使用堆疊

若要列出 CDK 應用程式中的所有堆疊，請使用指cdk ls令。前面的例子將輸出以下內容：

```
stack1
stack2
```

堆疊會部署為 AWS CloudFormation 堆疊的一部分到環 AWS 境中。環境涵蓋了一個特定的 AWS 帳戶和 AWS 區域.

當您針對具有多個堆疊的應用程式執行cdk synth命令時，雲端組件會針對每個堆疊執行個體包含個別的範本。即使兩個堆疊是相同類別的實體，也會將 AWS CDK 它們當做兩個個別範本發出。

您可以通過在cdk synth命令中指定堆棧名稱來合成每個模板。下面的例子合成了堆棧 1 的模板。

```
$ cdk synth stack1
```

這種方法在概念上與模 AWS CloudFormation 板通常使用方式不同，模板可以多次部署並通過AWS CloudFormation 參數參數化。雖然 AWS CloudFormation 參數可以在中定義 AWS CDK，但通常不鼓勵參數，因為 AWS CloudFormation 參數只能在部署期間解析。這意味著您無法在代碼中確定它們的值。

例如，要根據參數值在應用程序中有條件地包含資源，則必須設置AWS CloudFormation 條件並使用它標記資源。 AWS CDK 採用了一種在合成時解析具體模板的方法。因此，您可以使用 if 陳述式來檢查值，以判斷是否應該定義資源或應套用某些行為。

> **Note**
>
> 在合成期間 AWS CDK 提供盡可能多的分辨率，以使您的編程語言的慣用和自然使用。

像任何其他構造，堆棧可以組成組合在一起。下列程式碼顯示由三個堆疊組成的服務範例：控制平面、資料平面和監視堆疊。服務結構定義了兩次：一次用於測試版環境，一次用於生產環境。

TypeScript

```
import { App, Stack } from 'aws-cdk-lib';
import { Construct } from 'constructs';
```

```
interface EnvProps {
  prod: boolean;
}

// imagine these stacks declare a bunch of related resources
class ControlPlane extends Stack {}
class DataPlane extends Stack {}
class Monitoring extends Stack {}

class MyService extends Construct {

  constructor(scope: Construct, id: string, props?: EnvProps) {

    super(scope, id);

    // we might use the prod argument to change how the service is configured
    new ControlPlane(this, "cp");
    new DataPlane(this, "data");
    new Monitoring(this, "mon");  }
}

const app = new App();
new MyService(app, "beta");
new MyService(app, "prod", { prod: true });

app.synth();
```

JavaScript

```
const { App, Stack } = require('aws-cdk-lib');
const { Construct } = require('constructs');

// imagine these stacks declare a bunch of related resources
class ControlPlane extends Stack {}
class DataPlane extends Stack {}
class Monitoring extends Stack {}

class MyService extends Construct {

  constructor(scope, id, props) {

    super(scope, id);
```

```
    // we might use the prod argument to change how the service is configured
    new ControlPlane(this, "cp");
    new DataPlane(this, "data");
    new Monitoring(this, "mon");
  }
}

const app = new App();
new MyService(app, "beta");
new MyService(app, "prod", { prod: true });

app.synth();
```

Python

```
from aws_cdk import App, Stack
from constructs import Construct

# imagine these stacks declare a bunch of related resources
class ControlPlane(Stack): pass
class DataPlane(Stack): pass
class Monitoring(Stack): pass

class MyService(Construct):

  def __init__(self, scope: Construct, id: str, *, prod=False):

    super().__init__(scope, id)

    # we might use the prod argument to change how the service is configured
    ControlPlane(self, "cp")
    DataPlane(self, "data")
    Monitoring(self, "mon")

app = App();
MyService(app, "beta")
MyService(app, "prod", prod=True)

app.synth()
```

Java

```
package com.myorg;
```

```
import software.amazon.awscdk.App;
import software.amazon.awscdk.Stack;
import software.constructs.Construct;

public class MyApp {

    // imagine these stacks declare a bunch of related resources
    static class ControlPlane extends Stack {
        ControlPlane(Construct scope, String id) {
            super(scope, id);
        }
    }

    static class DataPlane extends Stack {
        DataPlane(Construct scope, String id) {
            super(scope, id);
        }
    }

    static class Monitoring extends Stack {
        Monitoring(Construct scope, String id) {
            super(scope, id);
        }
    }

    static class MyService extends Construct {
        MyService(Construct scope, String id) {
            this(scope, id, false);
        }

        MyService(Construct scope, String id, boolean prod) {
            super(scope, id);

            // we might use the prod argument to change how the service is
 configured
            new ControlPlane(this, "cp");
            new DataPlane(this, "data");
            new Monitoring(this, "mon");
        }
    }

    public static void main(final String argv[]) {
        App app = new App();
```

```
        new MyService(app, "beta");
        new MyService(app, "prod", true);

        app.synth();
    }
}
```

C#

```
using Amazon.CDK;
using Constructs;

// imagine these stacks declare a bunch of related resources
public class ControlPlane : Stack {
    public ControlPlane(Construct scope, string id=null) : base(scope, id) { }
}

public class DataPlane : Stack {
    public DataPlane(Construct scope, string id=null) : base(scope, id) { }
}

public class Monitoring : Stack
{
    public Monitoring(Construct scope, string id=null) : base(scope, id) { }
}

public class MyService : Construct
{
    public MyService(Construct scope, string id, Boolean prod=false) : base(scope,
 id)
    {
        // we might use the prod argument to change how the service is configured
        new ControlPlane(this, "cp");
        new DataPlane(this, "data");
        new Monitoring(this, "mon");
    }
}

class Program
{
    static void Main(string[] args)
    {
```

```
        var app = new App();
        new MyService(app, "beta");
        new MyService(app, "prod", prod: true);
        app.Synth();
    }
}
```

此 AWS CDK 應用程序最終由六個堆棧組成，每個環境三個堆棧：

```
$ cdk ls

betacpDA8372D3
betadataE23DB2BA
betamon632BD457
prodcp187264CE
proddataF7378CE5
prodmon631A1083
```

堆 AWS CloudFormation 疊的實體名稱會 AWS CDK 根據樹狀結構中堆疊的建構路徑自動決定。默認情況下，堆棧的名稱是從Stack對象的構造 ID 派生的。但是，您可以使用 stackName prop (在 Python 中stack_name) 來指定明確的名稱，如下所示。

TypeScript

```
new MyStack(this, 'not:a:stack:name', { stackName: 'this-is-stack-name' });
```

JavaScript

```
new MyStack(this, 'not:a:stack:name', { stackName: 'this-is-stack-name' });
```

Python

```
MyStack(self, "not:a:stack:name", stack_name="this-is-stack-name")
```

Java

```
new MyStack(this, "not:a:stack:name", StackProps.builder()
    .StackName("this-is-stack-name").build());
```

C#

```
new MyStack(this, "not:a:stack:name", new StackProps
{
    StackName = "this-is-stack-name"
});
```

巢狀堆疊

該NestedStack構造提供了一種圍繞堆棧 AWS CloudFormation 500 資源限制的方法。嵌套堆棧算作在包含它的堆棧中只有一個資源。不過，它最多可以包含 500 個資源，包括其他巢狀堆疊。

嵌套堆棧的範圍必須是Stack或NestedStack構造。嵌套堆棧不需要在其父堆棧中詞法聲明。實例化嵌套堆棧時，只需要將父堆棧作為第一個參數（scope）傳遞。除了這個限制之外，定義嵌套堆棧中的構造的工作原理與普通堆棧中的結構完全相同。

在合成時，嵌套堆棧合成為其自己的 AWS CloudFormation 模板，該模板在部署時將其上傳到 AWS CDK 臨時存儲桶。巢狀堆疊繫結至其父系堆疊，不會視為獨立的部署成品。它們不是由列出cdk list，而且不能由部署cdk deploy。

父堆棧和嵌套堆棧之間的引用會自動轉換為生成的 AWS CloudFormation 模板中的堆棧參數和輸出，就像任何跨堆棧引用一樣。

> **Warning**
>
> 部署巢狀堆疊之前，不會顯示安全狀況的變更。此資訊僅針對頂層堆疊顯示。

建構

構造是 AWS Cloud Development Kit (AWS CDK) 應用程序的基本構建塊。建構是應用程式中的元件，代表一或多個 AWS CloudFormation 資源及其組態。您可以透過匯入和設定建構來逐步建置應用程式。

建構是您匯入 CDK 應用程式的類別。您可以從「AWS 建構資料庫」取得建構。您也可以建立和散發自己的建構，或使用協力廠商開發人員所建立的建構。

構造是構造編程模型（CPM）的一部分。它們可以與其他工具一起使用，例如 CDK 用於 Terraform (CDKTF)、CDK Kubernetes (CDK8s) 和. Projen

主題

AWS 建構程式庫

AWS 構造庫包含由 AWS開發和維護的構造的集合。它被組織成包含代表所有可用資源的結構的各種模塊。 AWS如需參考資訊，請參閱 AWS CDK API 參考資料。

主 CDK 包被調用`aws-cdk-lib`，它包含了大部分的 AWS 構造庫。它還包含基類，例如`Stack`和`App`。

主要 CDK 套件的實際套件名稱會因語言而有所不同。

TypeScript

安裝	`npm install aws-cdk-lib`
Import	`import * as cdk from 'aws-cdk-lib';`

JavaScript

安裝	`npm install aws-cdk-lib`
Import	`const cdk = require('aws-cdk-lib');`

Python

安裝	`python -m pip install aws-cdk-lib`

Import	`import aws_cdk as cdk`

Java

在中`pom.xml`，新增	Group `##. ###.`; artifact `aws-cdk-lib`
Import	`import software.amazon.awscdk.App;`

C#

安裝	dotnet add package Amazon.CDK.Lib
Import	`using Amazon.CDK;`

Go

安裝	go get github.com/aws/aws-cdk-go/awscdk/v2
Import	`import ( "github.com/aws/aws-cdk-go/awscdk/v2" )`

> **Note**
>
> 如果您使用建立 CDK 專案`cdk init`，則不需要手動安裝`aws-cdk-lib`。

AWS 建構程式庫也包含具有`Construct`基底類別的`constructs`套件。它是在它自己的軟件包，因為除了其他基於構造的工具使用它 AWS CDK，包括用於地形的 CDK 和 Kubernetes 的 CDK。

許多第三方也發佈了與. AWS CDK造訪建構中心，探索 AWS CDK 建構合作夥伴生態系統。

建構圖層

「 AWS 建構資源庫」中的建構分為三個層級。每個級別都提供了不斷增加的抽象水平。抽象化越高，配置越容易，需要更少的專業知識。抽象越低，可用的自定義越多，需要更多的專業知識。

層級 1 (L1) 建構

L1 結構，也稱為 CFN 資源，是最低層級的建構，並且不提供抽象。每個 L1 建構會直接對應至單一 AWS CloudFormation 資源。使用 L1 建構時，您可以匯入代表特定 AWS CloudFormation 資源的建構。然後，您可以在建構執行個體中定義資源的屬性。

當您熟悉 AWS CloudFormation 並需要完全控制定義 AWS 資源屬性時，L1 建構非常適合使用。

在「 AWS 建構程式庫」中，L1 建構的名稱開頭為`Cfn`，後面接著它所代表之 AWS CloudFormation 資源的識別碼。例如，`CfnBucket`建構是代表`AWS::S3::Bucket` AWS CloudFormation 資源的 L1 建構。

L1 建構是從AWS CloudFormation 資源規格產生的。如果中存在資源 AWS CloudFormation，它將以 L1 建構的 AWS CDK 形式提供。新的資源或屬性可能需要長達一週的時間才能在 AWS 建構資料庫中使用。若要取得更多資訊，請參閱《AWS CloudFormation 使用指南》中的AWS 資源和屬性類型參考。

層級 2 (L2) 建構

L2 構造，也稱為策劃構造，由 CDK 團隊精心開發，並且通常是使用最廣泛的構造類型。L2 構造直接映射到單個 AWS CloudFormation 資源，類似於 L1 構造。與 L1 結構相比，L2 結構透過直覺式意向型 API 提供更高層級的抽象。L2 結構包括合理的默認屬性配置，最佳實踐安全策略，並為您生成大量的樣板代碼和粘合邏輯。

L2 建構也為大多數資源提供協助程式方法，讓定義屬性、權限、資源之間以事件為基礎的互動等變得更簡單快速。

此`s3.Bucket`類別是亞馬遜簡單儲存服務 (Amazon S3) 儲存貯體資源的 L2 建構範例。

「 AWS 建構資料庫」包含指定為穩定且可供生產使用的 L2 建構。對於正在開發的 L2 構造，它們被指定為實驗性並在單獨的模塊中提供。

層級 3 (L3) 建構

L3 結構，也稱為模式，是最高層級的抽象。每個 L3 建構都可以包含設定為共同運作以完成應用程式中特定工作或服務的資源集合。L3 結構用於為應用程序中的特定用例創建整個 AWS 體系結構。

為了提供完整的系統設計或更大系統的重要部分，L3 構造提供了自以為是的默認屬性配置。它們圍繞著解決問題並提供解決方案的特定方法構建。透過 L3 建構，您可以使用最少的輸入和程式碼，快速建立和設定多個資源。

此ecsPatterns.ApplicationLoadBalancedFargateService類別是 L3 建構的範例，代表在 Amazon 彈性容器 AWS Fargate 服務 (Amazon ECS) 叢集上執行且由應用程式負載平衡器前端執行的服務。

與 L2 建構類似，已準備好可供生產使用的 L3 建構包含在「建構資料庫」中。 AWS 正在開發的那些在單獨的模塊中提供。

定義建構

合成

組合是通過構造定義更高級別抽象的關鍵模式。高層建構可以由任意數目的較低層級建構組成。從自下而上的觀點，您可以使用建構來組織您要部署的個別 AWS 資源。您可以根據自己的目的使用任何方便的抽象，並根據需要提供盡可能多的級別。

使用構成，您可以定義可重複使用的組件，並像任何其他代碼一樣共享 例如，團隊可以定義一個建構來實作公司對 Amazon DynamoDB 表的最佳實務，包括備份、全域複寫、自動擴展和監控。團隊可以在內部與其他團隊共享構造，也可以公開共享。

團隊可以像使用任何其他庫包一樣使用構造。更新庫後，開發人員可以訪問新版本的改進和錯誤修復，類似於任何其他代碼庫。

初始化

建構模組在延伸 `Construct` 基本類別的類別中實作。您可以藉由實體化類別來定義建構。所有建構模組在初始化時都採用以下三個參數：

- scope — 建構的父項或擁有者。這可以是一個堆棧或另一個構造。範圍決定建構在建構樹中的位置。您通常應該傳遞 `this` (`self`in Python) 代表目前物件的範圍。
- id — 在範圍內必須是唯一的識別碼。該標識符作為構造中定義的所有內容的命名空間。它用於生成唯一標識符，例如資源名稱和 AWS CloudFormation 邏輯 ID。

  標識符只需要在一個範圍內是唯一的。這使您可以實例化和重複使用構造，而不必擔心它們可能包含的構造和標識符，並允許將構造組成為更高級別的抽象。此外，範圍使得可以一次引用所有構造組。範例包括標記，或指定建構的部署位置。

- prop — 一組屬性或關鍵字引數 (視語言而定)，用於定義建構的初始配置。較高層級的建構會提供更多預設值，如果所有 prop 元素都是選用的，您可以完全省略 prop 參數。

組態

大多數構造接受`props`作為它們的第三個參數（或者在 Python 中，關鍵字參數），這是一個定義構造配置的名稱/值集合。下列範例會定義啟用 AWS Key Management Service (AWS KMS) 加密和靜態網站託管的值區。由於它沒有明確指定加密金鑰，因此`Bucket`建構會定義新金鑰`kms.Key`並將其與值區建立關聯。

TypeScript

```
new s3.Bucket(this, 'MyEncryptedBucket', {
  encryption: s3.BucketEncryption.KMS,
  websiteIndexDocument: 'index.html'
});
```

JavaScript

```
new s3.Bucket(this, 'MyEncryptedBucket', {
  encryption: s3.BucketEncryption.KMS,
  websiteIndexDocument: 'index.html'
});
```

Python

```
s3.Bucket(self, "MyEncryptedBucket", encryption=s3.BucketEncryption.KMS,
    website_index_document="index.html")
```

Java

```
Bucket.Builder.create(this, "MyEncryptedBucket")
        .encryption(BucketEncryption.KMS_MANAGED)
        .websiteIndexDocument("index.html").build();
```

C#

```
new Bucket(this, "MyEncryptedBucket", new BucketProps
{
    Encryption = BucketEncryption.KMS_MANAGED,
```

```
    WebsiteIndexDocument = "index.html"
});
```

Go

```
awss3.NewBucket(stack, jsii.String("MyEncryptedBucket"), &awss3.BucketProps{
 Encryption: awss3.BucketEncryption_KMS,
 WebsiteIndexDocument: jsii.String("index.html"),
})
```

與建構互動

構造是擴展基本構造類的類。實體化建構之後，建構物件會公開一組方法和屬性，這些方法和屬性可讓您與建構互動，並將其作為系統其他部分的參考傳遞。

該 AWS CDK 框架沒有對構造的 API 施加任何限制。作者可以定義他們想要的任何 API。但是，包含在「 AWS AWS 建構資料庫」中的建構，例如`s3.Bucket`，請遵循指導方針和一般模式。這可在所有 AWS 資源中提供一致的體驗。

大多數建 AWS 構都有一組授與方法，您可以使用這些方法將該建構的 AWS Identity and Access Management (IAM) 許可授與主體。下列範例授予 IAM 群組從 Amazon S3 儲存貯體讀取的`data-science`權限`raw-data`。

TypeScript

```
const rawData = new s3.Bucket(this, 'raw-data');
const dataScience = new iam.Group(this, 'data-science');
rawData.grantRead(dataScience);
```

JavaScript

```
const rawData = new s3.Bucket(this, 'raw-data');
const dataScience = new iam.Group(this, 'data-science');
rawData.grantRead(dataScience);
```

Python

```
raw_data = s3.Bucket(self, 'raw-data')
data_science = iam.Group(self, 'data-science')
raw_data.grant_read(data_science)
```

Java

```
Bucket rawData = new Bucket(this, "raw-data");
Group dataScience = new Group(this, "data-science");
rawData.grantRead(dataScience);
```

C#

```
var rawData = new Bucket(this, "raw-data");
var dataScience = new Group(this, "data-science");
rawData.GrantRead(dataScience);
```

Go

```
 rawData := awss3.NewBucket(stack, jsii.String("raw-data"), nil)
 dataScience := awsiam.NewGroup(stack, jsii.String("data-science"), nil)
 rawData.GrantRead(dataScience, nil)
```

另一種常見的模式是讓 AWS 建構從其他地方提供的資料中設定資源的屬性之一。屬性可以包括 Amazon 資源名稱 (ARN)、名稱或網址。

下列程式碼會透過環境變數中的佇列 URL 定義 AWS Lambda 函數，並將其與 Amazon 簡單佇列服務 (Amazon SQS) 佇列產生關聯。

TypeScript

```
const jobsQueue = new sqs.Queue(this, 'jobs');
const createJobLambda = new lambda.Function(this, 'create-job', {
  runtime: lambda.Runtime.NODEJS_18_X,
  handler: 'index.handler',
  code: lambda.Code.fromAsset('./create-job-lambda-code'),
  environment: {
    QUEUE_URL: jobsQueue.queueUrl
  }
});
```

JavaScript

```
const jobsQueue = new sqs.Queue(this, 'jobs');
const createJobLambda = new lambda.Function(this, 'create-job', {
```

```
  runtime: lambda.Runtime.NODEJS_18_X,
  handler: 'index.handler',
  code: lambda.Code.fromAsset('./create-job-lambda-code'),
  environment: {
    QUEUE_URL: jobsQueue.queueUrl
  }
});
```

Python

```
jobs_queue = sqs.Queue(self, "jobs")
create_job_lambda = lambda_.Function(self, "create-job",
    runtime=lambda_.Runtime.NODEJS_18_X,
    handler="index.handler",
    code=lambda_.Code.from_asset("./create-job-lambda-code"),
    environment=dict(
        QUEUE_URL=jobs_queue.queue_url
    )
)
```

Java

```
final Queue jobsQueue = new Queue(this, "jobs");
Function createJobLambda = Function.Builder.create(this, "create-job")
                .handler("index.handler")
                .code(Code.fromAsset("./create-job-lambda-code"))
                .environment(java.util.Map.of(   // Map.of is Java 9 or later
                    "QUEUE_URL", jobsQueue.getQueueUrl())
                .build();
```

C#

```
var jobsQueue = new Queue(this, "jobs");
var createJobLambda = new Function(this, "create-job", new FunctionProps
{
    Runtime = Runtime.NODEJS_18_X,
    Handler = "index.handler",
    Code = Code.FromAsset(@".\create-job-lambda-code"),
    Environment = new Dictionary<string, string>
    {
        ["QUEUE_URL"] = jobsQueue.QueueUrl
    }
```

```
});
```

Go

```
createJobLambda := awslambda.NewFunction(stack, jsii.String("create-job"),
&awslambda.FunctionProps{
 Runtime: awslambda.Runtime_NODEJS_18_X(),
 Handler: jsii.String("index.handler"),
 Code:    awslambda.Code_FromAsset(jsii.String(".\\create-job-lambda-code"), nil),
 Environment: &map[string]*string{
  "QUEUE_URL": jsii.String(*jobsQueue.QueueUrl()),
 },
})
```

如需「 AWS 建構程式庫」中最常見 API 模式的相關資訊，請參閱the section called “資源”。

應用程序和堆棧構造

建構程式庫中的App和Stack類別是唯一的 AWS 建構。與其他結構相比，它們不會自行配置 AWS 資源。相反，它們用於為您的其他構造提供上下文。所有代表 AWS 資源的建構都必須在Stack建構的範圍內直接或間接定義。 Stack建構是在App建構範圍內定義的。

若要深入瞭解 CDK 應用程式，請參閱AWS CDK 应用。若要深入瞭解 CDK 堆疊，請參閱堆疊。

下列範例會定義具有單一堆疊的應用程式。在堆疊中，L2 建構可用來設定 Amazon S3 儲存貯體資源。

TypeScript

```
import { App, Stack, StackProps } from 'aws-cdk-lib';
import * as s3 from 'aws-cdk-lib/aws-s3';

class HelloCdkStack extends Stack {
  constructor(scope: App, id: string, props?: StackProps) {
    super(scope, id, props);

    new s3.Bucket(this, 'MyFirstBucket', {
      versioned: true
    });
  }
}
```

```
const app = new App();
new HelloCdkStack(app, "HelloCdkStack");
```

JavaScript

```
const { App , Stack } = require('aws-cdk-lib');
const s3 = require('aws-cdk-lib/aws-s3');

class HelloCdkStack extends Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    new s3.Bucket(this, 'MyFirstBucket', {
      versioned: true
    });
  }
}

const app = new App();
new HelloCdkStack(app, "HelloCdkStack");
```

Python

```
from aws_cdk import App, Stack
import aws_cdk.aws_s3 as s3
from constructs import Construct

class HelloCdkStack(Stack):

    def __init__(self, scope: Construct, id: str, **kwargs) -> None:
        super().__init__(scope, id, **kwargs)

        s3.Bucket(self, "MyFirstBucket", versioned=True)

app = App()
HelloCdkStack(app, "HelloCdkStack")
```

Java

HelloCdkStack.java檔案中定義的堆疊：

```
import software.constructs.Construct;
```

```
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.services.s3.*;

public class HelloCdkStack extends Stack {
    public HelloCdkStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public HelloCdkStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        Bucket.Builder.create(this, "MyFirstBucket")
            .versioned(true).build();
    }
}
```

HelloCdkApp.java檔案中定義的應用程式：

```
import software.amazon.awscdk.App;
import software.amazon.awscdk.StackProps;

public class HelloCdkApp {
    public static void main(final String[] args) {
        App app = new App();

        new HelloCdkStack(app, "HelloCdkStack", StackProps.builder()
                .build());

        app.synth();
    }
}
```

C#

```
using Amazon.CDK;
using Amazon.CDK.AWS.S3;

namespace HelloCdkApp
{
    internal static class Program
    {
```

```
        public static void Main(string[] args)
        {
            var app = new App();
            new HelloCdkStack(app, "HelloCdkStack");
            app.Synth();
        }
    }

    public class HelloCdkStack : Stack
    {
        public HelloCdkStack(Construct scope, string id, IStackProps props=null) :
 base(scope, id, props)
        {
            new Bucket(this, "MyFirstBucket", new BucketProps { Versioned = true });
        }
    }
}
```

Go

```
func NewHelloCdkStack(scope constructs.Construct, id string, props
 *HelloCdkStackProps) awscdk.Stack {
 var sprops awscdk.StackProps
 if props != nil {
  sprops = props.StackProps
 }
 stack := awscdk.NewStack(scope, &id, &sprops)

 awss3.NewBucket(stack, jsii.String("MyFirstBucket"), &awss3.BucketProps{
  Versioned: jsii.Bool(true),
 })

 return stack
}
```

使用建構

使用 L1 建構

L1 建構會直接對應至個別 AWS CloudFormation 資源。您必須提供資源的必要配置。

在這個例子中，我們使用 CfnBucket L1 構造創建一個bucket對象：

TypeScript

```
const bucket = new s3.CfnBucket(this, "MyBucket", {
  bucketName: "MyBucket"
});
```

JavaScript

```
const bucket = new s3.CfnBucket(this, "MyBucket", {
  bucketName: "MyBucket"
});
```

Python

```
bucket = s3.CfnBucket(self, "MyBucket", bucket_name="MyBucket")
```

Java

```
CfnBucket bucket = new CfnBucket.Builder().bucketName("MyBucket").build();
```

C#

```
var bucket = new CfnBucket(this, "MyBucket", new CfnBucketProps
{
    BucketName= "MyBucket"
});
```

Go

```
 awss3.NewCfnBucket(stack, jsii.String("MyBucket"), &awss3.CfnBucketProps{
  BucketName: jsii.String("MyBucket"),
 })
```

不是簡單布林值、字串、數字或容器的建構屬性，在支援的語言中會以不同的方式處理。

TypeScript

```
const bucket = new s3.CfnBucket(this, "MyBucket", {
  bucketName: "MyBucket",
  corsConfiguration: {
```

```
    corsRules: [{
          allowedOrigins: ["*"],
          allowedMethods: ["GET"]
    }]
  }
});
```

JavaScript

```
const bucket = new s3.CfnBucket(this, "MyBucket", {
  bucketName: "MyBucket",
  corsConfiguration: {
    corsRules: [{
          allowedOrigins: ["*"],
          allowedMethods: ["GET"]
    }]
  }
});
```

Python

在 Python 中，這些屬性由定義為 L1 構造的內部類別的類型來表示。例如，a 的可選屬cors_configuration性CfnBucket需要類型的包裝函式CfnBucket.CorsConfigurationProperty。在這裡，我們cors_configuration在一個CfnBucket實例上定義。

```
bucket = CfnBucket(self, "MyBucket", bucket_name="MyBucket",
    cors_configuration=CfnBucket.CorsConfigurationProperty(
        cors_rules=[CfnBucket.CorsRuleProperty(
            allowed_origins=["*"],
            allowed_methods=["GET"]
        )]
    )
)
```

Java

在 Java 中，這些屬性由定義為 L1 構造的內部類別的類型來表示。例如，a 的可選屬corsConfiguration性CfnBucket需要類型的包裝函式CfnBucket.CorsConfigurationProperty。在這裡，我們corsConfiguration在一個CfnBucket實例上定義。

```
CfnBucket bucket = CfnBucket.Builder.create(this, "MyBucket")
                        .bucketName("MyBucket")
                        .corsConfiguration(new
 CfnBucket.CorsConfigurationProperty.Builder()
                            .corsRules(Arrays.asList(new
 CfnBucket.CorsRuleProperty.Builder()
                                .allowedOrigins(Arrays.asList("*"))
                                .allowedMethods(Arrays.asList("GET"))
                                .build()))
                            .build())
                        .build();
```

C#

在 C# 中，這些屬性由定義為 L1 構造的內部類型表示。例如，a 的可選屬CorsConfiguration性CfnBucket需要類型的包裝函式CfnBucket.CorsConfigurationProperty。在這裡，我們CorsConfiguration在一個CfnBucket實例上定義。

```
var bucket = new CfnBucket(this, "MyBucket", new CfnBucketProps
{
    BucketName = "MyBucket",
    CorsConfiguration = new CfnBucket.CorsConfigurationProperty
    {
        CorsRules = new object[] {
            new CfnBucket.CorsRuleProperty
            {
                AllowedOrigins = new string[] { "*" },
                AllowedMethods = new string[] { "GET" },
            }
        }
    }
});
```

Go

在 Go 中，這些類型是使用 L1 建構、底線和屬性名稱的名稱來命名。例如，a 的可選屬CorsConfiguration性CfnBucket需要類型的包裝函式CfnBucket_CorsConfigurationProperty。在這裡，我們CorsConfiguration在一個CfnBucket實例上定義。

```
 awss3.NewCfnBucket(stack, jsii.String("MyBucket"), &awss3.CfnBucketProps{
```

```
  BucketName: jsii.String("MyBucket"),
  CorsConfiguration: &awss3.CfnBucket_CorsConfigurationProperty{
   CorsRules: []awss3.CorsRule{
    awss3.CorsRule{
     AllowedOrigins: jsii.Strings("*"),
     AllowedMethods: &[]awss3.HttpMethods{"GET"},
    },
   },
  },
 })
```

Important

您不能將 L2 屬性類型與 L1 建構搭配使用，反之亦然。使用 L1 建構時，請務必使用您正在使用的 L1 建構所定義的類型。請勿使用其他 L1 建構中的型別 (有些類型可能具有相同的名稱，但它們不是相同的類型)。
我們的一些特定於語言的 API 引用目前在 L1 屬性類型的路徑中存在錯誤，或者根本不記錄這些類。我們希望能盡快解決這個問題。同時，請記住，這些類型始終是與它們一起使用的 L1 構造的內部類。

使用 L2 建構

在下列範例中，我們透過從 `Bucket`L2 建構建立物件來定義 Amazon S3 儲存貯體：

TypeScript

```
import * as s3 from 'aws-cdk-lib/aws-s3';

// "this" is HelloCdkStack
new s3.Bucket(this, 'MyFirstBucket', {
  versioned: true
});
```

JavaScript

```
const s3 = require('aws-cdk-lib/aws-s3');

// "this" is HelloCdkStack
new s3.Bucket(this, 'MyFirstBucket', {
```

```
  versioned: true
});
```

Python

```
import aws_cdk.aws_s3 as s3

# "self" is HelloCdkStack
s3.Bucket(self, "MyFirstBucket", versioned=True)
```

Java

```
import software.amazon.awscdk.services.s3.*;

public class HelloCdkStack extends Stack {
    public HelloCdkStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public HelloCdkStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        Bucket.Builder.create(this, "MyFirstBucket")
                .versioned(true).build();
    }
}
```

C#

```
using Amazon.CDK.AWS.S3;

// "this" is HelloCdkStack
new Bucket(this, "MyFirstBucket", new BucketProps
{
    Versioned = true
});
```

Go

```
import (
 "github.com/aws/aws-cdk-go/awscdk/v2/awss3"
```

```
 "github.com/aws/jsii-runtime-go"
)

// stack is HelloCdkStack
awss3.NewBucket(stack, jsii.String("MyFirstBucket"), &awss3.BucketProps{
  Versioned: jsii.Bool(true),
 })>
```

`MyFirstBucket`不是所 AWS CloudFormation 建立值區的名稱。它是給 CDK 應用程序上下文中的新構造的邏輯標識符。實體名稱值將用於命名資源。 AWS CloudFormation

使用第三方建構

構建中心是一種資源 AWS，可幫助您發現來自第三方和開源 CDK 社區的其他構造。

編寫自己的構造

除了使用現有的結構之外，您還可以編寫自己的構造，並讓任何人在其應用程序中使用它們。所有建構在中都是相等的。 AWS CDK來自構造庫的 AWS 構造被視為與通過NPM、Maven或發布的第三方庫中的 contstruct 相同。PyPI發佈至公司內部套件儲存庫的建構也會以相同的方式處理。

若要宣告新建構，請在`constructs`封裝中建立擴充 Con struc t 基底類別的類別，然後遵循初始設定式引數的模式。

下列範例顯示如何宣告代表 Amazon S3 儲存貯體的建構。S3 儲存貯體會在每次有人上傳檔案到其中時，傳送 Amazon 簡單通知服務 (Amazon SNS) 通知。

TypeScript

```
export interface NotifyingBucketProps {
  prefix?: string;
}

export class NotifyingBucket extends Construct {
  constructor(scope: Construct, id: string, props: NotifyingBucketProps = {}) {
    super(scope, id);
    const bucket = new s3.Bucket(this, 'bucket');
    const topic = new sns.Topic(this, 'topic');
    bucket.addObjectCreatedNotification(new s3notify.SnsDestination(topic),
      { prefix: props.prefix });
  }
```

```
}
```

JavaScript

```
class NotifyingBucket extends Construct {
  constructor(scope, id, props = {}) {
    super(scope, id);
    const bucket = new s3.Bucket(this, 'bucket');
    const topic = new sns.Topic(this, 'topic');
    bucket.addObjectCreatedNotification(new s3notify.SnsDestination(topic),
      { prefix: props.prefix });
  }
}

module.exports = { NotifyingBucket }
```

Python

```
class NotifyingBucket(Construct):

    def __init__(self, scope: Construct, id: str, *, prefix=None):
        super().__init__(scope, id)
        bucket = s3.Bucket(self, "bucket")
        topic = sns.Topic(self, "topic")
        bucket.add_object_created_notification(s3notify.SnsDestination(topic),
            s3.NotificationKeyFilter(prefix=prefix))
```

Java

```
public class NotifyingBucket extends Construct {

    public NotifyingBucket(final Construct scope, final String id) {
        this(scope, id, null, null);
    }

    public NotifyingBucket(final Construct scope, final String id, final BucketProps
 props) {
        this(scope, id, props, null);
    }

    public NotifyingBucket(final Construct scope, final String id, final String
 prefix) {
        this(scope, id, null, prefix);
```

```
    }

    public NotifyingBucket(final Construct scope, final String id, final BucketProps
 props, final String prefix) {
        super(scope, id);

        Bucket bucket = new Bucket(this, "bucket");
        Topic topic = new Topic(this, "topic");
        if (prefix != null)
            bucket.addObjectCreatedNotification(new SnsDestination(topic),
                NotificationKeyFilter.builder().prefix(prefix).build());
     }
}
```

C#

```
public class NotifyingBucketProps : BucketProps
{
    public string Prefix { get; set; }
}

public class NotifyingBucket : Construct
{
    public NotifyingBucket(Construct scope, string id, NotifyingBucketProps props =
 null) : base(scope, id)
    {
        var bucket = new Bucket(this, "bucket");
        var topic = new Topic(this, "topic");
        bucket.AddObjectCreatedNotification(new SnsDestination(topic), new
 NotificationKeyFilter
        {
            Prefix = props?.Prefix
        });
    }
}
```

Go

```
type NotifyingBucketProps struct {
 awss3.BucketProps
 Prefix *string
}
```

```
func NewNotifyingBucket(scope constructs.Construct, id *string, props
 *NotifyingBucketProps) awss3.Bucket {
 var bucket awss3.Bucket
 if props == nil {
  bucket = awss3.NewBucket(scope, jsii.String(*id+"Bucket"), nil)
 } else {
  bucket = awss3.NewBucket(scope, jsii.String(*id+"Bucket"), &props.BucketProps)
 }
 topic := awssns.NewTopic(scope, jsii.String(*id+"Topic"), nil)
 if props == nil {
  bucket.AddObjectCreatedNotification(awss3notifications.NewSnsDestination(topic))
 } else {
  bucket.AddObjectCreatedNotification(awss3notifications.NewSnsDestination(topic),
 &awss3.NotificationKeyFilter{
   Prefix: props.Prefix,
  })
 }
 return bucket
}
```

> **Note**
>
> 我們的NotifyingBucket構造不是從繼承Bucket而是從Construct. 我們使用組合而不是繼承，將 Amazon S3 儲存貯體和 Amazon SNS 主題捆綁在一起。一般來說，在開發 AWS CDK 構造時，組合優於繼承。

構NotifyingBucket造函數具有典型的構造簽名：scopeid，和props。最後一個引數是可選的 (取得預設值{})props，因為所有 prop 都是選用的。（基Construct類不採用props參數。）您可以在您的應用程序中定義此構造的一個實例props，例如：

TypeScript

```
new NotifyingBucket(this, 'MyNotifyingBucket');
```

JavaScript

```
new NotifyingBucket(this, 'MyNotifyingBucket');
```

Python

```
NotifyingBucket(self, "MyNotifyingBucket")
```

Java

```
new NotifyingBucket(this, "MyNotifyingBucket");
```

C#

```
new NotifyingBucket(this, "MyNotifyingBucket");
```

Go

```
NewNotifyingBucket(stack, jsii.String("MyNotifyingBucket"), nil)
```

或者您可以使用`props`（在 Java 中是一個附加參數）來指定要過濾的路徑前綴，例如：

TypeScript

```
new NotifyingBucket(this, 'MyNotifyingBucket', { prefix: 'images/' });
```

JavaScript

```
new NotifyingBucket(this, 'MyNotifyingBucket', { prefix: 'images/' });
```

Python

```
NotifyingBucket(self, "MyNotifyingBucket", prefix="images/")
```

Java

```
new NotifyingBucket(this, "MyNotifyingBucket", "/images");
```

C#

```
new NotifyingBucket(this, "MyNotifyingBucket", new NotifyingBucketProps
{
```

```
    Prefix = "/images"
});
```

Go

```
NewNotifyingBucket(stack, jsii.String("MyNotifyingBucket"), &NotifyingBucketProps{
 Prefix: jsii.String("images/"),
})
```

通常，您還希望在構造中公開某些屬性或方法。在構造後面隱藏一個主題並不是很有用，因為構造的用戶無法訂閱它。新增`topic`屬性可讓取用者存取內部主題，如下列範例所示：

TypeScript

```
export class NotifyingBucket extends Construct {
  public readonly topic: sns.Topic;

  constructor(scope: Construct, id: string, props: NotifyingBucketProps) {
    super(scope, id);
    const bucket = new s3.Bucket(this, 'bucket');
    this.topic = new sns.Topic(this, 'topic');
    bucket.addObjectCreatedNotification(new s3notify.SnsDestination(this.topic),
 { prefix: props.prefix });
  }
}
```

JavaScript

```
class NotifyingBucket extends Construct {

  constructor(scope, id, props) {
    super(scope, id);
    const bucket = new s3.Bucket(this, 'bucket');
    this.topic = new sns.Topic(this, 'topic');
    bucket.addObjectCreatedNotification(new s3notify.SnsDestination(this.topic),
 { prefix: props.prefix });
  }
}

module.exports = { NotifyingBucket };
```

Python

```
class NotifyingBucket(Construct):

    def __init__(self, scope: Construct, id: str, *, prefix=None, **kwargs):
        super().__init__(scope, id)
        bucket = s3.Bucket(self, "bucket")
        self.topic = sns.Topic(self, "topic")
        bucket.add_object_created_notification(s3notify.SnsDestination(self.topic),
            s3.NotificationKeyFilter(prefix=prefix))
```

Java

```
public class NotifyingBucket extends Construct {

    public Topic topic = null;

    public NotifyingBucket(final Construct scope, final String id) {
        this(scope, id, null, null);
    }

    public NotifyingBucket(final Construct scope, final String id, final BucketProps
 props) {
        this(scope, id, props, null);
    }

    public NotifyingBucket(final Construct scope, final String id, final String
 prefix) {
        this(scope, id, null, prefix);
    }

    public NotifyingBucket(final Construct scope, final String id, final BucketProps
 props, final String prefix) {
        super(scope, id);

        Bucket bucket = new Bucket(this, "bucket");
        topic = new Topic(this, "topic");
        if (prefix != null)
            bucket.addObjectCreatedNotification(new SnsDestination(topic),
                NotificationKeyFilter.builder().prefix(prefix).build());
     }
}
```

C#

```
public class NotifyingBucket : Construct
{
    public readonly Topic topic;

    public NotifyingBucket(Construct scope, string id, NotifyingBucketProps props =
 null) : base(scope, id)
    {
        var bucket = new Bucket(this, "bucket");
        topic = new Topic(this, "topic");
        bucket.AddObjectCreatedNotification(new SnsDestination(topic), new
 NotificationKeyFilter
        {
            Prefix = props?.Prefix
        });
    }
}
```

Go

要在 Go 中執行此操作，我們需要一些額外的管道。我們原來的NewNotifyingBucket函數返回了一個awss3.Bucket. 我們需要通過創建一個NotifyingBucket結構Bucket來擴展以包含一個topic成員。然後，我們的函數將返回這種類型。

```
type NotifyingBucket struct {
 awss3.Bucket
 topic awssns.Topic
}

func NewNotifyingBucket(scope constructs.Construct, id *string, props
 *NotifyingBucketProps) NotifyingBucket {
 var bucket awss3.Bucket
 if props == nil {
  bucket = awss3.NewBucket(scope, jsii.String(*id+"Bucket"), nil)
 } else {
  bucket = awss3.NewBucket(scope, jsii.String(*id+"Bucket"), &props.BucketProps)
 }
 topic := awssns.NewTopic(scope, jsii.String(*id+"Topic"), nil)
 if props == nil {
  bucket.AddObjectCreatedNotification(awss3notifications.NewSnsDestination(topic))
 } else {
```

```
  bucket.AddObjectCreatedNotification(awss3notifications.NewSnsDestination(topic),
 &awss3.NotificationKeyFilter{
   Prefix: props.Prefix,
  })
 }
 var nbucket NotifyingBucket
 nbucket.Bucket = bucket
 nbucket.topic = topic
 return nbucket
}
```

現在，消費者可以訂閱該主題，例如：

TypeScript

```
const queue = new sqs.Queue(this, 'NewImagesQueue');
const images = new NotifyingBucket(this, '/images');
images.topic.addSubscription(new sns_sub.SqsSubscription(queue));
```

JavaScript

```
const queue = new sqs.Queue(this, 'NewImagesQueue');
const images = new NotifyingBucket(this, '/images');
images.topic.addSubscription(new sns_sub.SqsSubscription(queue));
```

Python

```
queue = sqs.Queue(self, "NewImagesQueue")
images = NotifyingBucket(self, prefix="Images")
images.topic.add_subscription(sns_sub.SqsSubscription(queue))
```

Java

```
NotifyingBucket images = new NotifyingBucket(this, "MyNotifyingBucket", "/images");
images.topic.addSubscription(new SqsSubscription(queue));
```

C#

```
var queue = new Queue(this, "NewImagesQueue");
var images = new NotifyingBucket(this, "MyNotifyingBucket", new NotifyingBucketProps
{
```

```
    Prefix = "/images"
});
images.topic.AddSubscription(new SqsSubscription(queue));
```

Go

```
queue := awssqs.NewQueue(stack, jsii.String("NewImagesQueue"), nil)
images := NewNotifyingBucket(stack, jsii.String("MyNotifyingBucket"),
&NotifyingBucketProps{
 Prefix: jsii.String("/images"),
})
images.topic.AddSubscription(awssnssubscriptions.NewSqsSubscription(queue, nil))
```

進一步了解

以下影片提供 CDK 結構的全面概述，並說明如何在 CDK 應用程式中使用它們。

CDK 構造解釋

環境

環境是目標 AWS 區域 ， AWS 帳戶 而堆疊會部署至。CDK 應用程序中的所有堆棧都與環境（ ）env明確或隱式關聯。

主題

- 設定環境
- 引導環境

設定環境

對於生產堆棧，我們建議您使用env屬性為應用程序中的每個堆棧明確指定環境。下列範例會為其兩個不同的堆疊指定不同的環境。

TypeScript

```
const envEU  = { account: '2383838383', region: 'eu-west-1' };
const envUSA = { account: '8373873873', region: 'us-west-2' };

new MyFirstStack(app, 'first-stack-us', { env: envUSA });
```

```
new MyFirstStack(app, 'first-stack-eu', { env: envEU });
```

JavaScript

```
const envEU  = { account: '2383838383', region: 'eu-west-1' };
const envUSA = { account: '8373873873', region: 'us-west-2' };

new MyFirstStack(app, 'first-stack-us', { env: envUSA });
new MyFirstStack(app, 'first-stack-eu', { env: envEU });
```

Python

```
env_EU = cdk.Environment(account="8373873873", region="eu-west-1")
env_USA = cdk.Environment(account="2383838383", region="us-west-2")

MyFirstStack(app, "first-stack-us", env=env_USA)
MyFirstStack(app, "first-stack-eu", env=env_EU)
```

Java

```
public class MyApp {

    // Helper method to build an environment
    static Environment makeEnv(String account, String region) {
        return Environment.builder()
                .account(account)
                .region(region)
                .build();
    }

    public static void main(final String argv[]) {
        App app = new App();

        Environment envEU = makeEnv("8373873873", "eu-west-1");
        Environment envUSA = makeEnv("2383838383", "us-west-2");

        new MyFirstStack(app, "first-stack-us", StackProps.builder()
                .env(envUSA).build());
        new MyFirstStack(app, "first-stack-eu", StackProps.builder()
                .env(envEU).build());

        app.synth();
    }
```

```
}
```

C#

```
Amazon.CDK.Environment makeEnv(string account, string region)
{
    return new Amazon.CDK.Environment
    {
        Account = account,
        Region = region
    };
}

var envEU = makeEnv(account: "8373873873", region: "eu-west-1");
var envUSA = makeEnv(account: "2383838383", region: "us-west-2");

new MyFirstStack(app, "first-stack-us", new StackProps { Env=envUSA });
new MyFirstStack(app, "first-stack-eu", new StackProps { Env=envEU });
```

當您對目標帳戶和 Region 進行硬式編碼 (如上述範例所示) 時，堆疊一律會部署到該特定帳戶和區域。若要將堆疊部署到不同的目標，但要在合成時判斷目標，您的堆疊可以使用 AWS CDK CLI 提供的兩個環境變數:CDK_DEFAULT_ACCOUNT和CDK_DEFAULT_REGION。這些變數是根據使用--profile選項指定的 AWS 描述檔設定，如果您未指定，則為預設設定 AWS 檔。

下列程式碼片段顯示如何存取從堆疊中 AWS CDK CLI 傳遞的帳戶和區域。

TypeScript

通過 Node 的process對象訪問環境變量。

> **Note**
>
> 您需要在process中使用該DefinitelyTyped模組 TypeScript。 cdk init為您安裝此模塊。但是，如果您正在處理在添加之前創建的項目，或者您沒有使用cdk init.
>
> ```
> npm install @types/node
> ```

```
new MyDevStack(app, 'dev', {
```

```
  env: {
    account: process.env.CDK_DEFAULT_ACCOUNT,
    region: process.env.CDK_DEFAULT_REGION
}});
```

JavaScript

通過 Node 的process對象訪問環境變量。

```
new MyDevStack(app, 'dev', {
  env: {
    account: process.env.CDK_DEFAULT_ACCOUNT,
    region: process.env.CDK_DEFAULT_REGION
}});
```

Python

使用os模組的environ字典來存取環境變數。

```
import os
MyDevStack(app, "dev", env=cdk.Environment(
    account=os.environ["CDK_DEFAULT_ACCOUNT"],
    region=os.environ["CDK_DEFAULT_REGION"]))
```

Java

用System.getenv()於取得環境變數的值。

```
public class MyApp {

    // Helper method to build an environment
    static Environment makeEnv(String account, String region) {
        account = (account == null) ? System.getenv("CDK_DEFAULT_ACCOUNT") :
 account;
        region = (region == null) ? System.getenv("CDK_DEFAULT_REGION") : region;

        return Environment.builder()
                .account(account)
                .region(region)
                .build();
    }

    public static void main(final String argv[]) {
```

```
        App app = new App();

        Environment envEU = makeEnv(null, null);
        Environment envUSA = makeEnv(null, null);

        new MyDevStack(app, "first-stack-us", StackProps.builder()
                .env(envUSA).build());
        new MyDevStack(app, "first-stack-eu", StackProps.builder()
                .env(envEU).build());

        app.synth();
    }
}
```

C#

用System.Environment.GetEnvironmentVariable()於取得環境變數的值。

```
Amazon.CDK.Environment makeEnv(string account=null, string region=null)
{
    return new Amazon.CDK.Environment
    {
        Account = account ??
 System.Environment.GetEnvironmentVariable("CDK_DEFAULT_ACCOUNT"),
        Region = region ??
 System.Environment.GetEnvironmentVariable("CDK_DEFAULT_REGION")
    };
}

new MyDevStack(app, "dev", new StackProps { Env = makeEnv() });
```

AWS 區域 使用區域代碼指定。如需清單，請參閱區域端點。

AWS CDK 完全不指定屬性和使CDK_DEFAULT_ACCOUNT用和指定env屬性之間的區別。CDK_DEFAULT_REGION前者意味著堆棧應該合成一個與環境無關的模板。在這種堆疊中定義的建構不能使用有關其環境的任何資訊。例如，您不能編寫類似的代碼if (stack.region === 'us-east-1')或使用框架設施，如 vpc.From Lookup（Python:from_lookup），它需要查詢您的帳戶。AWS 在您指定明確的環境之前，這些功能完全不起作用；若要使用它們，您必須指定env。

當您使用CDK_DEFAULT_ACCOUNT和傳遞環境時CDK_DEFAULT_REGION，堆疊將部署在合成時由 AWS CDK CLI 決定的帳戶和區域中。這使環境依賴的代碼可以工作，但這也意味著合成的模板可能會

根據其合成的機器，用戶或會話而有所不同。這種行為通常在開發過程中可以接受甚至可取，但它可能是生產使用的反模式。

您可以根據需env要使用任何有效的表達式進行設置。例如，您可以撰寫堆疊來支援兩個額外的環境變數，讓您在合成時覆寫帳戶和 Region。我們會打電話給這些CDK_DEPLOY_ACCOUNT，CDK_DEPLOY_REGION在這裡，但你可以命名他們任何你喜歡的東西，因為它們不是由 AWS CDK. 在下面的堆棧的環境中，如果設置了替代環境變量，則使用它們。如果未設定它們，它們會退回至 AWS CDK.

TypeScript

```
new MyDevStack(app, 'dev', {
  env: {
    account: process.env.CDK_DEPLOY_ACCOUNT || process.env.CDK_DEFAULT_ACCOUNT,
    region: process.env.CDK_DEPLOY_REGION || process.env.CDK_DEFAULT_REGION
}});
```

JavaScript

```
new MyDevStack(app, 'dev', {
  env: {
    account: process.env.CDK_DEPLOY_ACCOUNT || process.env.CDK_DEFAULT_ACCOUNT,
    region: process.env.CDK_DEPLOY_REGION || process.env.CDK_DEFAULT_REGION
}});
```

Python

```
MyDevStack(app, "dev", env=cdk.Environment(
    account=os.environ.get("CDK_DEPLOY_ACCOUNT", os.environ["CDK_DEFAULT_ACCOUNT"]),
    region=os.environ.get("CDK_DEPLOY_REGION", os.environ["CDK_DEFAULT_REGION"])
```

Java

```
public class MyApp {

    // Helper method to build an environment
    static Environment makeEnv(String account, String region) {
        account = (account == null) ? System.getenv("CDK_DEPLOY_ACCOUNT") : account;
        region = (region == null) ? System.getenv("CDK_DEPLOY_REGION") : region;
        account = (account == null) ? System.getenv("CDK_DEFAULT_ACCOUNT") :
 account;
        region = (region == null) ? System.getenv("CDK_DEFAULT_REGION") : region;
```

```
        return Environment.builder()
                .account(account)
                .region(region)
                .build();
    }

    public static void main(final String argv[]) {
        App app = new App();

        Environment envEU = makeEnv(null, null);
        Environment envUSA = makeEnv(null, null);

        new MyDevStack(app, "first-stack-us", StackProps.builder()
                .env(envUSA).build());
        new MyDevStack(app, "first-stack-eu", StackProps.builder()
                .env(envEU).build());

        app.synth();
    }
}
```

C#

```
Amazon.CDK.Environment makeEnv(string account=null, string region=null)
{
    return new Amazon.CDK.Environment
    {
        Account = account ??
            System.Environment.GetEnvironmentVariable("CDK_DEPLOY_ACCOUNT") ??
            System.Environment.GetEnvironmentVariable("CDK_DEFAULT_ACCOUNT"),
        Region = region ??
            System.Environment.GetEnvironmentVariable("CDK_DEPLOY_REGION") ??
            System.Environment.GetEnvironmentVariable("CDK_DEFAULT_REGION")
    };
}

new MyDevStack(app, "dev", new StackProps { Env = makeEnv() });
```

通過這種方式聲明堆棧的環境，您可以編寫一個簡短的腳本或批處理文件，如下所示以從命令行參數設置變量，然後調用`cdk deploy`。前兩個以外的任何引數都會傳遞給，`cdk deploy`並且可以用來指定命令列選項或堆疊。

macOS/Linux

```
#!/usr/bin/env bash
if [[ $# -ge 2 ]]; then
    export CDK_DEPLOY_ACCOUNT=$1
    export CDK_DEPLOY_REGION=$2
    shift; shift
    npx cdk deploy "$@"
    exit $?
else
    echo 1>&2 "Provide account and region as first two args."
    echo 1>&2 "Additional args are passed through to cdk deploy."
    exit 1
fi
```

將腳本另存為`cdk-deploy-to.sh`，然後執行`chmod +x cdk-deploy-to.sh`以使其可執行。

Windows

```
@findstr /B /V @ %~dpnx0 > %~dpn0.ps1 && powershell -ExecutionPolicy Bypass
 %~dpn0.ps1 %*
@exit /B %ERRORLEVEL%
if ($args.length -ge 2) {
    $env:CDK_DEPLOY_ACCOUNT, $args = $args
    $env:CDK_DEPLOY_REGION,  $args = $args
    npx cdk deploy $args
    exit $lastExitCode
} else {
    [console]::error.writeline("Provide account and region as first two args.")
    [console]::error.writeline("Additional args are passed through to cdk deploy.")
    exit 1
}
```

該腳本的視窗版本用於 PowerShell 提供與 MacOS /Linux 版本相同的功能。它還包含指令，允許它作為一個批處理文件運行，以便它可以很容易地從命令行調用。它應該被保存為`cdk-deploy-to.bat`。當調用批處理文件時，`cdk-deploy-to.ps1`將創建該文件。

然後，您可以編寫其他腳本來調用「部署到」腳本以部署到特定環境（甚至每個腳本的多個環境）：

macOS/Linux

```
#!/usr/bin/env bash
```

```
# cdk-deploy-to-test.sh
./cdk-deploy-to.sh 123457689 us-east-1 "$@"
```

Windows

```
@echo off
rem cdk-deploy-to-test.bat
cdk-deploy-to 135792469 us-east-1 %*
```

部署到多個環境時，請考慮是否要在部署失敗後繼續部署至其他環境。下列範例可避免在第一個生產環境失敗時部署至第二個生產環境。

macOS/Linux

```
#!/usr/bin/env bash
# cdk-deploy-to-prod.sh
./cdk-deploy-to.sh 135792468 us-west-1 "$@" || exit
./cdk-deploy-to.sh 246813579 eu-west-1 "$@"
```

Windows

```
@echo off
rem cdk-deploy-to-prod.bat
cdk-deploy-to 135792469 us-west-1 %* || exit /B
cdk-deploy-to 245813579 eu-west-1 %*
```

開發人員仍然可以使用正常`cdk deploy`命令部署到自己的 AWS 環境進行開發。

如果在實例化堆棧時未指定環境，則該堆棧被認為與環境無關。 AWS CloudFormation 從這種堆棧合成的模板將嘗試在環境相關的屬性（例如`stack.account`，`stack.region`和`stack.availabilityZones`（Python：）上使用部署時間分辨率。`availability_zones`

使用cdk deploy部署與環境無關的堆疊時， AWS CDK CLI將使用指定的設定檔來決定要部署的位 AWS CLI 置。如果未指定任何設定檔，則會使用預設設定檔。 AWS CDK CLI遵循類似的通訊協定，以決 AWS CLI 定在您的 AWS 帳戶中執行作業時要使用哪些 AWS 認證。如需詳細資訊，請參閱 the section called “AWS CDK 工具包”。

在與環境無關的堆疊中，使用可用區域的任何建構都會看到兩個可用區域，允許堆疊部署到任何區域。

引導環境

您必須啟動要將 CDK 堆疊部署到的每個環境。啟動安裝會準備環境以進行部署。如需進一步了解，請參閱引導。

引導

啟動安裝是準備部署環境的程序。啟動安裝是一次性動作，您必須針對將資源部署到的每個環境執行此動作。

主題

引導環境

> **Important**
>
> 您可能會對已啟動載入的資源中儲存的資料產生 AWS 費用。

啟動安裝可在您的環境中佈建資源，例如用於存放檔案的 Amazon 簡單儲存服務 (Amazon S3) 儲存貯體，以及授予執行部署所需許可的 AWS Identity and Access Management (IAM) 角色。這些資源會在 AWS CloudFormation 堆疊中佈建，稱為啟動程序堆疊。它通常被命名`CDKToolkit`。就像任何 AWS CloudFormation 堆疊一樣，一旦部署，它就會出現在環境的 AWS CloudFormation 主控台中。

> **Note**
>
> CDK v2 使用現代的引導模板。在 v2 中不支援 CDK v1 中的舊版範本。

環境是獨立的。如果您想要部署到多個環境，則必須個別啟動載入每個環境。

如果您嘗試將 CDK 應用程式部署到尚未啟動載入的環境中，您將收到錯誤訊息，提醒您啟動環境。

使用 CDK Pipelines 進行引導

如果您使用 CDK Pipelines 部署到其他帳戶的環境中，並且收到類似下列的訊息：

```
Policy contains a statement with one or more invalid principals
```

此錯誤訊息表示其他環境中不存在適當的 IAM 角色。最有可能的原因是環境尚未啟動載入。啟動環境，然後再試一次。

> **Note**
>
> 如果環境已啟動載入，請勿刪除並重新建立環境的啟動程序堆疊。刪除啟動程序堆疊將刪除最初佈建在環境中以支援 CDK 部署的 AWS 資源。這將導致管道停止工作。相反，請嘗試通過再次運行 CDK CLI `cdk bootstrap` 命令將啟動程序堆棧更新為新版本。

如何引導

啟動環境時， AWS CloudFormation 範本會部署至特定環境。此範本會在您的帳戶中佈建資源，以準備您的環境以進行部署。

引導模板接受自定義引導資源某些方面的參數。如需詳細資訊，請參閱 the section called “自定義引導”。

您可以透過下列任何一種方式引導：

- 使用 AWS CDK CLI 命令。cdk bootstrap這是最簡單的方法，如果你只有幾個環境來引導，效果很好。
- AWS CDK CLI使用其他 AWS CloudFormation 部署工具部署所提供的範本。這可讓您使用 AWS CloudFormation StackSets 或以 AWS Control Tower 及 AWS CloudFormation 控制台或 AWS CLI. 您可以在部署之前對範本進行一些小的修改。這種方法更加靈活，適合大規模部署。

引導一個環境不止一次不是錯誤。如果您啟動的環境已經啟動載入，則必要時會升級其啟動程序堆疊。否則，什麼都不會發生。

引導與 AWS CDKCLI

使用指cdk bootstrap令啟動一或多個 AWS 環境。

下列範例會啟動兩個環境：

```
$ cdk bootstrap aws://ACCOUNT-NUMBER-1/REGION-1 aws://ACCOUNT-NUMBER-2/REGION-2 ...
```

下列範例顯示啟動載入環境的多種方式。如第二個範例所示，指定環境時，aws://前置字元是選用的。

```
$ cdk bootstrap aws://123456789012/us-east-1
$ cdk bootstrap 123456789012/us-east-1 123456789012/us-west-1
```

當您執行時cdk bootstrap，CDK CLI 一律會在目前目錄中合成 CDK 應用程式。如果您沒有指定至少一個環境，CDK CLI 會啟動應用程式中參照的所有環境。

對於與環境無關的堆疊，CDK CLI 將嘗試從預設來源判斷環境。這可以是使用--profile選項、環境變數或預設 AWS CLI 來源指定的環境。如果找到，則會啟動載入環境。

例如，以下命令使用prod AWS 配置文件合成當前 AWS CDK 應用程序，然後啟動其環境。

```
$ cdk bootstrap --profile prod
```

從模板引導 AWS CloudFormation

您可以取得並部署啟動程序 AWS CloudFormation 範本來啟動環境。

若要在檔案中取得此範本的副本bootstrap-template.yaml，請執行下列命令：

macOS/Linux

```
$ cdk bootstrap --show-template > bootstrap-template.yaml
```

Windows

在 Windows 上， PowerShell 必須用來保留模板的編碼。

```
powershell "cdk bootstrap --show-template | Out-File -encoding utf8 bootstrap-
template.yaml"
```

該模板也可以在AWS CDK GitHub 存儲庫中使用。

使用 CDK CLI 或您偏好的範本部署機制來部署此 AWS CloudFormation 範本。若要使用 CDK CLI 進行部署，請執行cdk bootstrap --template ***TEMPLATE_FILENAME***。您也可以使用執行下列命令來部署它，或使用 AWS CloudFormation Stack Sets 一次部署到一或多個帳戶。 AWS CLI

macOS/Linux

```
aws cloudformation create-stack \
  --stack-name CDKToolkit \
  --template-body file://path/to/bootstrap-template.yaml \
  --capabilities CAPABILITY_NAMED_IAM \
  --region us-west-1
```

Windows

```
aws cloudformation create-stack ^
  --stack-name CDKToolkit ^
  --template-body file://path/to/bootstrap-template.yaml ^
  --capabilities CAPABILITY_NAMED_IAM ^
  --region us-west-1
```

自定義引導

有兩種方法可以自訂環境中資源的啟動載入：

- 搭配指令使用`cdk bootstrap`指令行參數。這可讓您修改範本的幾個方面。
- 修改預設啟動程序範本並自行部署。這使您可以更完整地控制引導資源。

下列命令列選項與 CDK 搭配使用時 CLIcdk bootstrap，會提供啟動載入範本的常用調整：

- --bootstrap-bucket-name 覆寫 Amazon S3 儲存貯體的名稱。可能需要變更您的 CDK 應用程式 (請參閱the section called “堆疊合成器”)。
- --bootstrap-kms-key-id覆寫用於加密 S3 儲存貯體的 AWS KMS 金鑰。
- --cloudformation-execution-policies指定受管理原則的 ARN，這些原則應附加至部署角色在堆疊部署 AWS CloudFormation 期間所承擔的部署角色。依預設，堆疊會使用該`AdministratorAccess`原則以完整的管理員權限部署。

  原則 ARN 必須以單一字串引數的形式傳遞，並以逗號分隔個別 ARN。例如：

```
--cloudformation-execution-policies "arn:aws:iam::aws:policy/
AWSLambda_FullAccess,arn:aws:iam::aws:policy/AWSCodeDeployFullAccess".
```

> **Important**
>
> 若要避免部署失敗，請確定您指定的原則足以讓您在啟動載入的環境中執行的任何部署。

- --qualifier是新增至啟動程序堆疊中所有資源名稱的字串。限定詞可讓您在相同環境中佈建多個啟動程序堆疊時避免資源名稱衝突。預設值為 hnb659fds (此值沒有意義)。

  更改限定符還要求您的 CDK 應用程序將更改後的值傳遞給堆棧合成器。如需詳細資訊，請參閱 the section called “堆疊合成器”。
- --tags將一個或多個 AWS CloudFormation 標籤添加到啟動程序堆棧。
- --trust列出可能部署到正在啟動載入之環境中的 AWS 帳戶。

  啟動載入另一個環境中的 CDK 管線將部署到的環境時，請使用此旗標。執行引導載入的帳戶始終受信任。
- --trust-for-lookup列出可從啟動載入的環境中查詢內容資訊的 AWS 帳戶。

  使用此旗標可授與帳戶合成將部署到環境中的堆疊的權限，而不會實際授予他們直接部署這些堆疊的權限。
- --termination-protection防止啟動程序堆疊被刪除。若要取得更多資訊，請參閱《AWS CloudFormation 使用指南》中的〈保護堆疊不被刪除〉。

> **Important**
>
> 現代啟動程序範本有效地授予`--trust`清單中任`--cloudformation-execution-policies`何 AWS 帳戶所隱含的權限。默認情況下，這將擴展讀取和寫入啟動載入帳戶中的任何資源的權限。確保使用您熟悉的策略和受信任帳戶配置引導堆棧。

自訂範本

當您需要比 CDK 提供的更多自訂項目時，您CLI可以修改啟動程序範本以滿足您的需求。首先，您可以使用--show-template選項取得範本。以下是範例：

```
$ cdk bootstrap --show-template
```

您所做的任何修改都必須遵守引導模板合同。若要確保稍後不會被使用預設範本執行的cdk bootstrap使用者意外覆寫您的自訂，請變更 BootstrapVariant template 參數的預設值。CDK CLI 只允許使用與目前部署的範本具有相同BootstrapVariant且版本等於或更高版本的範本覆寫啟動程序堆疊。

然後the section called “從模板引導 AWS CloudFormation”，您可以按照中所述或使用部署修改過的範本cdk bootstrap --template。

```
$ cdk bootstrap --template bootstrap-template.yaml
```

引導模板差異

如前所述， AWS CDK v1 支持兩個引導模板，傳統和現代。CDK V2 僅支持現代模板。作為參考，以下是這兩個模板之間的高層次差異。

功能	舊版 (僅限 v1)	現代（V1 和 V2）
跨帳戶部署	不允許	允許
AWS CloudFormation 許可	使用目前使用者的權限進行部署 (由 AWS 設定檔、環境變數等決定)	使用佈建啟動程序堆疊時指定的權限進行部署 (例如，使--trust用)
版本控制	只有一個版本的啟動程序堆棧可用	Bootstrap 堆棧已版本化; 可以在將 future 的版本中添加新資源，並且 AWS CDK 應用程序可能需要最低版本
資源 *	Amazon S3 儲存貯體	Amazon S3 儲存貯體
		AWS KMS key
		IAM 角色
		Amazon ECR 儲存庫
		用於版本控制的 SSM 參數

功能	舊版 (僅限 v1)	現代（V1 和 V2）
資源命名	自動產生	確定性
貯體加密	預設金鑰	客戶受管金鑰

* 我們將根據需要向引導模板添加其他資源。

必須升級使用舊版範本啟動載入的環境，才能透過重新啟動載入來使用 CDK v2 的現代範本。在刪除舊版值區之前，至少重新部署環境中的所有 AWS CDK 應用程式一次。

堆疊合成器

您的 AWS CDK 應用程序需要了解可用的引導資源，以便成功合成可以部署的堆棧。堆棧合成器是一 AWS CDK 個控制堆棧模板如何合成的類。這包括它如何使用引導載入資源（例如，它如何引用存儲在引導存儲桶中的資產）。

內建 AWS CDK的堆疊合成器稱`DefaultStackSynthesizer`為。它包括跨帳戶部署和 CDK Pipelines 部署的功能。

當您使用該`synthesizer`屬性實例化堆棧時，可以將堆棧合成器傳遞給堆棧。

TypeScript

```
new MyStack(this, 'MyStack', {
  // stack properties
  synthesizer: new DefaultStackSynthesizer({
    // synthesizer properties
  }),
});
```

JavaScript

```
new MyStack(this, 'MyStack', {
  // stack properties
  synthesizer: new DefaultStackSynthesizer({
    // synthesizer properties
  }),
});
```

Python

```
MyStack(self, "MyStack",
    # stack properties
    synthesizer=DefaultStackSynthesizer(
        # synthesizer properties
))
```

Java

```
new MyStack(app, "MyStack", StackProps.builder()
    // stack properties
  .synthesizer(DefaultStackSynthesizer.Builder.create()
    // synthesizer properties
    .build())
  .build();
```

C#

```
new MyStack(app, "MyStack", new StackProps
// stack properties
{
    Synthesizer = new DefaultStackSynthesizer(new DefaultStackSynthesizerProps
    {
        // synthesizer properties
    })
});
```

如果您未提供該synthesizer屬性，DefaultStackSynthesizer則使用。

定制合成

根據您對引導模板所做的更改，您可能還需要自定義合成。DefaultStackSynthesizer可以使用如下所述的屬性來自訂。

如果這些屬性都沒有提供您需要的自定義，則可以將合成器編寫為實現的類IStackSynthesizer（可能是衍生自DefaultStackSynthesizer）。

變更限定詞

限定符被添加到引導資源的名稱中，以區分單獨的引導程序堆棧中的資源。若要在相同環境 (AWS 帳戶和區域) 中部署兩個不同版本的啟動程式堆疊，堆疊必須具有不同的限定詞。

此功能用於 CDK 本身的自動化測試之間的名稱隔離。除非您可以非常精確地將授予 AWS CloudFormation 執行角色的 IAM 許可範圍縮小，否則在單個帳戶中擁有兩個不同的引導堆疊沒有權限隔離好處。因此，通常不需要變更此值。

若要變更限定詞，請使`DefaultStackSynthesizer`用屬性實例化合成器來配置：

TypeScript

```
new MyStack(this, 'MyStack', {
  synthesizer: new DefaultStackSynthesizer({
    qualifier: 'MYQUALIFIER',
  }),
});
```

JavaScript

```
new MyStack(this, 'MyStack', {
  synthesizer: new DefaultStackSynthesizer({
    qualifier: 'MYQUALIFIER',
  }),
})
```

Python

```
MyStack(self, "MyStack",
    synthesizer=DefaultStackSynthesizer(
        qualifier="MYQUALIFIER"
))
```

Java

```
new MyStack(app, "MyStack", StackProps.builder()
  .synthesizer(DefaultStackSynthesizer.Builder.create()
    .qualifier("MYQUALIFIER")
    .build())
```

```
  .build();
```

C#

```
new MyStack(app, "MyStack", new StackProps
{
    Synthesizer = new DefaultStackSynthesizer(new DefaultStackSynthesizerProps
    {
        Qualifier = "MYQUALIFIER"
    })
});
```

或者通過將限定符配置為上下文鍵入cdk.json。

```
{
  "app": "...",
  "context": {
    "@aws-cdk/core:bootstrapQualifier": "MYQUALIFIER"
  }
}
```

變更資源名稱

所有其他DefaultStackSynthesizer屬性與引導模板中的資源的名稱有關。只有在修改啟動程序範本並變更資源名稱或命名配置時，才需要提供這些屬性中的任何一個。

所有屬性都接受特殊的預留位置${Qualifier}${AWS::Partition}、${AWS::AccountId}、和${AWS::Region}。這些預留位置會分別取代為qualifier參數值，以及堆疊環境的分 AWS 割區、帳戶 ID 和 Region 值。

下列範例會顯示最常用的屬性DefaultStackSynthesizer及其預設值，就像您正在實體化合成器一樣。如需完整清單，請參閱 DefaultStackSynthesizerProps。

TypeScript

```
new DefaultStackSynthesizer({
  // Name of the S3 bucket for file assets
  fileAssetsBucketName: 'cdk-${Qualifier}-assets-${AWS::AccountId}-${AWS::Region}',
  bucketPrefix: '',
```

```
  // Name of the ECR repository for Docker image assets
  imageAssetsRepositoryName: 'cdk-${Qualifier}-container-assets-${AWS::AccountId}-
${AWS::Region}',

  // ARN of the role assumed by the CLI and Pipeline to deploy here
  deployRoleArn: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-deploy-role-${AWS::AccountId}-${AWS::Region}',
  deployRoleExternalId: '',

  // ARN of the role used for file asset publishing (assumed from the CLI role)
  fileAssetPublishingRoleArn: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-file-publishing-role-${AWS::AccountId}-${AWS::Region}',
  fileAssetPublishingExternalId: '',

  // ARN of the role used for Docker asset publishing (assumed from the CLI role)
  imageAssetPublishingRoleArn: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-image-publishing-role-${AWS::AccountId}-${AWS::Region}',
  imageAssetPublishingExternalId: '',

  // ARN of the role passed to CloudFormation to execute the deployments
  cloudFormationExecutionRole: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-cfn-exec-role-${AWS::AccountId}-${AWS::Region}',

  // ARN of the role used to look up context information in an environment
  lookupRoleArn: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-lookup-role-${AWS::AccountId}-${AWS::Region}',
  lookupRoleExternalId: '',

  // Name of the SSM parameter which describes the bootstrap stack version number
  bootstrapStackVersionSsmParameter: '/cdk-bootstrap/${Qualifier}/version',

  // Add a rule to every template which verifies the required bootstrap stack
 version
  generateBootstrapVersionRule: true,

})
```

JavaScript

```
new DefaultStackSynthesizer({
  // Name of the S3 bucket for file assets
  fileAssetsBucketName: 'cdk-${Qualifier}-assets-${AWS::AccountId}-${AWS::Region}',
  bucketPrefix: '',
```

```
  // Name of the ECR repository for Docker image assets
  imageAssetsRepositoryName: 'cdk-${Qualifier}-container-assets-${AWS::AccountId}-
${AWS::Region}',

  // ARN of the role assumed by the CLI and Pipeline to deploy here
  deployRoleArn: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-deploy-role-${AWS::AccountId}-${AWS::Region}',
  deployRoleExternalId: '',

  // ARN of the role used for file asset publishing (assumed from the CLI role)
  fileAssetPublishingRoleArn: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-file-publishing-role-${AWS::AccountId}-${AWS::Region}',
  fileAssetPublishingExternalId: '',

  // ARN of the role used for Docker asset publishing (assumed from the CLI role)
  imageAssetPublishingRoleArn: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-image-publishing-role-${AWS::AccountId}-${AWS::Region}',
  imageAssetPublishingExternalId: '',

  // ARN of the role passed to CloudFormation to execute the deployments
  cloudFormationExecutionRole: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-cfn-exec-role-${AWS::AccountId}-${AWS::Region}',

  // ARN of the role used to look up context information in an environment
  lookupRoleArn: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-lookup-role-${AWS::AccountId}-${AWS::Region}',
  lookupRoleExternalId: '',

  // Name of the SSM parameter which describes the bootstrap stack version number
  bootstrapStackVersionSsmParameter: '/cdk-bootstrap/${Qualifier}/version',

  // Add a rule to every template which verifies the required bootstrap stack
 version
  generateBootstrapVersionRule: true,
})
```

Python

```
DefaultStackSynthesizer(
  # Name of the S3 bucket for file assets
  file_assets_bucket_name="cdk-${Qualifier}-assets-${AWS::AccountId}-
${AWS::Region}",
```

```
  bucket_prefix="",

  # Name of the ECR repository for Docker image assets
  image_assets_repository_name="cdk-${Qualifier}-container-assets-${AWS::AccountId}-
${AWS::Region}",

  # ARN of the role assumed by the CLI and Pipeline to deploy here
  deploy_role_arn="arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-deploy-role-${AWS::AccountId}-${AWS::Region}",
  deploy_role_external_id="",

  # ARN of the role used for file asset publishing (assumed from the CLI role)
  file_asset_publishing_role_arn="arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-file-publishing-role-${AWS::AccountId}-${AWS::Region}",
  file_asset_publishing_external_id="",

  # ARN of the role used for Docker asset publishing (assumed from the CLI role)
  image_asset_publishing_role_arn="arn:${AWS::Partition}:iam::
${AWS::AccountId}:role/cdk-${Qualifier}-image-publishing-role-${AWS::AccountId}-
${AWS::Region}",
  image_asset_publishing_external_id="",

  # ARN of the role passed to CloudFormation to execute the deployments
  cloud_formation_execution_role="arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-cfn-exec-role-${AWS::AccountId}-${AWS::Region}",

  # ARN of the role used to look up context information in an environment
  lookup_role_arn="arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-lookup-role-${AWS::AccountId}-${AWS::Region}",
  lookup_role_external_id="",

  # Name of the SSM parameter which describes the bootstrap stack version number
  bootstrap_stack_version_ssm_parameter="/cdk-bootstrap/${Qualifier}/version",

  # Add a rule to every template which verifies the required bootstrap stack version
  generate_bootstrap_version_rule=True,
)
```

Java

```
DefaultStackSynthesizer.Builder.create()
    // Name of the S3 bucket for file assets
```

```
    .fileAssetsBucketName("cdk-${Qualifier}-assets-${AWS::AccountId}-
${AWS::Region}")
    .bucketPrefix('')

    // Name of the ECR repository for Docker image assets
    .imageAssetsRepositoryName("cdk-${Qualifier}-container-assets-${AWS::AccountId}-
${AWS::Region}")

    // ARN of the role assumed by the CLI and Pipeline to deploy here
    .deployRoleArn("arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-deploy-role-${AWS::AccountId}-${AWS::Region}")
    .deployRoleExternalId("")

    // ARN of the role used for file asset publishing (assumed from the CLI role)
    .fileAssetPublishingRoleArn("arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-file-publishing-role-${AWS::AccountId}-${AWS::Region}")
    .fileAssetPublishingExternalId("")

    // ARN of the role used for Docker asset publishing (assumed from the CLI role)
    .imageAssetPublishingRoleArn("arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-image-publishing-role-${AWS::AccountId}-${AWS::Region}")
    .imageAssetPublishingExternalId("")

    // ARN of the role passed to CloudFormation to execute the deployments
    .cloudFormationExecutionRole("arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-cfn-exec-role-${AWS::AccountId}-${AWS::Region}")

    .lookupRoleArn("arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-lookup-role-${AWS::AccountId}-${AWS::Region}")
    .lookupRoleExternalId("")

    // Name of the SSM parameter which describes the bootstrap stack version number
    .bootstrapStackVersionSsmParameter("/cdk-bootstrap/${Qualifier}/version")

    // Add a rule to every template which verifies the required bootstrap stack
 version
    .generateBootstrapVersionRule(true)
.build()
```

C#

```
new DefaultStackSynthesizer(new DefaultStackSynthesizerProps
{
```

```
    // Name of the S3 bucket for file assets
    FileAssetsBucketName = "cdk-${Qualifier}-assets-${AWS::AccountId}-
${AWS::Region}",
    BucketPrefix = "",

    // Name of the ECR repository for Docker image assets
    ImageAssetsRepositoryName = "cdk-${Qualifier}-container-assets-
${AWS::AccountId}-${AWS::Region}",

    // ARN of the role assumed by the CLI and Pipeline to deploy here
    DeployRoleArn = "arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-deploy-role-${AWS::AccountId}-${AWS::Region}",
    DeployRoleExternalId = "",

    // ARN of the role used for file asset publishing (assumed from the CLI role)
    FileAssetPublishingRoleArn = "arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-file-publishing-role-${AWS::AccountId}-${AWS::Region}",
    FileAssetPublishingExternalId = "",

    // ARN of the role used for Docker asset publishing (assumed from the CLI role)
    ImageAssetPublishingRoleArn = "arn:${AWS::Partition}:iam::
${AWS::AccountId}:role/cdk-${Qualifier}-image-publishing-role-${AWS::AccountId}-
${AWS::Region}",
    ImageAssetPublishingExternalId = "",

    // ARN of the role passed to CloudFormation to execute the deployments
    CloudFormationExecutionRole = "arn:${AWS::Partition}:iam::
${AWS::AccountId}:role/cdk-${Qualifier}-cfn-exec-role-${AWS::AccountId}-
${AWS::Region}",

    LookupRoleArn = "arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-lookup-role-${AWS::AccountId}-${AWS::Region}",
    LookupRoleExternalId = "",

    // Name of the SSM parameter which describes the bootstrap stack version number
    BootstrapStackVersionSsmParameter = "/cdk-bootstrap/${Qualifier}/version",

    // Add a rule to every template which verifies the required bootstrap stack
 version
    GenerateBootstrapVersionRule = true,
})
```

引導模板合同

引導堆棧的要求取決於正在使用的堆棧合成器。如果您編寫自己的堆疊合成器，則可以完全控制合成器所需的引導資源以及合成器如何找到它們。

本節介紹DefaultStackSynthesizer了引導模板的期望。

版本控制

範本應包含一個資源，以建立具有已知名稱的 SSM 參數，以及反映範本版本的輸出。

```
Resources:
  CdkBootstrapVersion:
    Type: AWS::SSM::Parameter
    Properties:
      Type: String
      Name:
        Fn::Sub: '/cdk-bootstrap/${Qualifier}/version'
      Value: 4
Outputs:
  BootstrapVersion:
    Value:
      Fn::GetAtt: [CdkBootstrapVersion, Value]
```

角色

DefaultStackSynthesizer需要五個 IAM 角色用於五個不同的目的。如果您未使用預設角色，則必須告訴合成器您要使用之角色的 ARN。

角色如下：

- 部署角色由 AWS CDK Toolkit 承擔，並由部署 AWS CodePipeline 到環境中。它AssumeRolePolicy控制誰可以部署到環境中。在範本中，您可以看到此角色所需的權限。
- 「 AWS CDK 工具組」會假設查詢角色在環境中執行前後關聯查詢。它AssumeRolePolicy控制誰可以部署到環境中。您可以在範本中看到此角色所需的權限。
- AWS CDK Toolkit 和 AWS CodeBuild 專案會假設檔案發佈角色和影像發佈角色，以將資產發佈到環境中。它們分別用來寫入 S3 儲存貯體和 ECR 儲存庫。這些角色需要這些資源的寫入權限。
- AWS CloudFormation 執行角色會傳遞 AWS CloudFormation 給以執行實際部署。其權限是部署將在其下執行的權限。權限會作為列出受管理原則 ARN 的參數傳遞至堆疊。

輸出

該 AWS CDK 工具包要求啟動程序堆棧上存在以下 CloudFormation 輸出。

- `BucketName`：檔案資產值區的名稱
- `BucketDomainName`：網域名稱格式的檔案資產儲存貯體
- `BootstrapVersion`：啟動程序堆疊的目前版本

模板歷史

引導程序模板是版本化的，並隨著時間的 AWS CDK 推移而發展。如果您提供自己的引導模板，請使用規範的默認模板保持最新狀態。您要確保範本可繼續使用所有 CDK 功能。

> **Note**
>
> 默認情況下，引導模板的早期版本 AWS KMS key 在每個引導式環境中創建了一個。若要避免 KMS 金鑰收費，請使用`--no-bootstrap-customer-key`. 目前的預設值為無 KMS 金鑰，這有助於避免這些費用。

本節包含在每個版本中所做的變更清單。

範本版本	AWS CDK 版本	變更
1	1.40.0	具有值區、金鑰、儲存庫和角色的初始範本版本。
2	1.45.0	將資產發佈角色分割為個別的檔案和影像發佈角色。
3	1.46.0	新增`FileAssetKeyArn` 匯出，以便能夠將解密權限新增至資產取用者。
4	1.61.0	AWS KMS 現在透過 Amazon S3 隱含許可，不再需要 `FileAsetKeyArn` 。添加 `CdkBootstrapVersion`

範本版本	AWS CDK 版本	變更
		SSM 參數，以便在不知道堆棧名稱的情況下驗證啟動程序堆棧版本。
5	1.87.0	部署角色可以讀取 SSM 參數。
6	1.108.0	新增與部署角色分開的查詢角色。
6	1.109.0	將`aws-cdk:bootstrap-role` 標籤附加至部署、檔案發佈和映像發佈角色。
7	1.110.0	部署角色無法再直接讀取目標帳戶中的值區。不過，這個角色實際上是系統管理員，而且總是可以使用其 AWS CloudFormation 權限讓值區可讀取)。
8	1.114.0	查閱角色具有目標環境的完整唯讀權限，並具有`aws-cdk:bootstrap-role` 標籤。
9	2.1.0	修正 Amazon S3 資產上傳不被常見參考的加密 SCP 拒絕的問題。
10	2.4.0	Amazon ECR 現 ScanOnPush 在預設為啟用。
11	2.18.0	新增允許 Lambda 從 Amazon ECR 存放庫提取的政策，以便在重新啟動過程中存活。

範本版本	AWS CDK 版本	變更
12	2.20.0	添加對實驗性的支持cdk import。
13	2.25.0	使啟動載具建立的 Amazon ECR 儲存庫中的容器映像不可變。
14	2.34.0	依預設，在儲存庫層級關閉 Amazon ECR 映像掃描，以允許不支援映像掃描的啟動載入區域。
15	2.60.0	無法標記 KMS 金鑰。
16	2.69.0	找到 KMS .2 的位址 Security Hub。
17	2.72.0	尋找到 ECR.3 的位址 Security Hub。
18	2.80.0	針對版本 16 所做的還原變更，因為它們不適用於所有分割區，因此不建議使用。
19	2.106.1	還原對 18 版所做的變更，其中 AccessControl 屬性已從範本中移除。(#27964)
20	2.119.0	將`ssm:GetParameters` 動作新增至部 AWS CloudFormation 署 IAM 角色。如需詳細資訊，請參閱 #28336。

Security Hub 發現項

如果您正在使用 AWS Security Hub，則可能會看到 AWS CDK 引導過程創建的某些資源的發現結果報告。Security Hub 發現項目可協助您尋找應該仔細檢查準確性和安全性的資源組態。我們已透過 AWS Security 檢閱這些特定的資源組態，並確信它們不會構成安全性問題。

[KMS.2] IAM 主體不應具有允許對所有 KMS 金鑰進行解密動作的 IAM 內嵌政策

部署角色 (預設名稱`cdk-hnb659fds-deploy-role-ACCOUNT-REGION`) 具有讀取存放在 Amazon S3 中的加密資料的權限。該政策本身並未授予任何資料的權限：只能解密從 Amazon S3 讀取的資料，並且只能從明確允許 Deploy Role 透過儲存貯體政策讀取資料的儲存貯體，以及明確允許 Deploy Role 使用其金鑰政策使用這些資料進行解密的金鑰。此陳述式用於允許 AWS CDK 管道執行跨帳戶部署。

為什麼 Security Hub 標記這個？ 原則包含一個`Resource: *`結合`Condition`子句；Security Hub 正在標記*. 這*是必要的，因為在帳戶啟動載入時，由 P AWS CDK ipelines 針對 CodePipeline Artifact 存儲桶創建的 AWS KMS 密鑰尚不存在，因此我們無法引用其 ARN。此外，Security Hub 在其推理中不包括政策聲明中的`Condition`子句。

如果我想解決這個發現怎麼辦？ 只要 AWS KMS 金鑰上的資源原則不是不必要的寬容性，目前的角色原則就不允許部署角色存取任何超過應有的資料。如果您仍然想擺脫發現，則可以通過以下兩種方式之一自定義引導程序堆棧（使用上面概述的過程）來完成此操作：

- 如果您不使用 AWS CDK 管道進行跨帳戶部署：從部署角色`Sid: PipelineCrossAccountArtifactsBucket`中移除陳述式；或
- 如果您使用 AWS CDK 管道進行跨帳戶部署：部署 AWS CDK 管道之後，請查詢成 Artifact 儲存貯體的 AWS KMS Key ARN，並以實際的 `Resource: *` Key ARN 取代`Sid: PipelineCrossAccountArtifactsBucket`陳述式。

資源

資源是您設定要在應用程式 AWS 服務 中使用的項目。資源是的一項功能 AWS CloudFormation。透過在 AWS CloudFormation 範本中配置資源及其屬性，您可以部署 AWS CloudFormation 到佈建資源。使用 AWS Cloud Development Kit (AWS CDK)，您可以透過建構來配置資源。然後，您部署 CDK 應用程序，其中涉及合成 AWS CloudFormation 模板並部署 AWS CloudFormation 到佈建資源。

主題

- 使用建構配置資源

- 參考資源
- 資源實體名稱
- 傳遞唯一資源識別碼
- 授與資源之間的權限
- 資源指標和警示
- 網路流量
- 事件處理
- 移除政策

使用建構配置資源

如中所述the section called “建構”，AWS CDK 提供了一個豐富的建構函式庫 (稱為AWS 建構)，代表所有 AWS 資源。

若要使用對應的建構來建立資源的執行個體，請傳入範圍作為第一個引數、建構的邏輯 ID，以及一組組態屬性 (prop)。例如，以下說明如何使用建構程式庫中的 SQ S.Queue 建構建立具有 AWS KMS 加密功能的 Amazon SQS 佇列。 AWS

TypeScript

```
import * as sqs from '@aws-cdk/aws-sqs';

new sqs.Queue(this, 'MyQueue', {
    encryption: sqs.QueueEncryption.KMS_MANAGED
});
```

JavaScript

```
const sqs = require('@aws-cdk/aws-sqs');

new sqs.Queue(this, 'MyQueue', {
    encryption: sqs.QueueEncryption.KMS_MANAGED
});
```

Python

```
import aws_cdk.aws_sqs as sqs
```

```
sqs.Queue(self, "MyQueue", encryption=sqs.QueueEncryption.KMS_MANAGED)
```

Java

```
import software.amazon.awscdk.services.sqs.*;

Queue.Builder.create(this, "MyQueue").encryption(
        QueueEncryption.KMS_MANAGED).build();
```

C#

```
using Amazon.CDK.AWS.SQS;

new Queue(this, "MyQueue", new QueueProps
{
    Encryption = QueueEncryption.KMS_MANAGED
});
```

某些配置 prop 是可選的，並且在許多情況下具有默認值。在某些情況下，所有 prop 都是可選的，最後一個參數可以完全省略。

資源屬性

「AWS 建構程式庫」中的大多數資源都會公開屬性，這些屬性會在部署階段透過 AWS CloudFormation。屬性會以屬性的形式顯示在資源類別上，並以類型名稱做為前置詞。下列範例說明如何使用 `queueUrl` (Python:`queue_url`) 屬性取得 Amazon SQS 佇列的網址。

TypeScript

```
import * as sqs from '@aws-cdk/aws-sqs';

const queue = new sqs.Queue(this, 'MyQueue');
const url = queue.queueUrl; // => A string representing a deploy-time value
```

JavaScript

```
const sqs = require('@aws-cdk/aws-sqs');

const queue = new sqs.Queue(this, 'MyQueue');
const url = queue.queueUrl; // => A string representing a deploy-time value
```

Python

```
import aws_cdk.aws_sqs as sqs

queue = sqs.Queue(self, "MyQueue")
url = queue.queue_url # => A string representing a deploy-time value
```

Java

```
Queue queue = new Queue(this, "MyQueue");
String url = queue.getQueueUrl();    // => A string representing a deploy-time value
```

C#

```
var queue = new Queue(this, "MyQueue");
var url = queue.QueueUrl; // => A string representing a deploy-time value
```

如需有the section called “代幣”關如何將部署時間屬性 AWS CDK 編碼為字串的資訊，請參閱。

參考資源

配置資源時，您通常必須引用另一個資源的屬性。範例如下：

- 亞馬遜彈性容器服務 (Amazon ECS) 資源需要對其執行所在叢集的參考。
- Amazon CloudFront 分發需要參考包含源代碼的亞馬遜簡單存儲服務（Amazon S3）存儲桶。

您可以使用下列任一方式參考資源：

- 通過傳遞 CDK 應用程序中定義的資源，無論是在同一堆棧中還是在不同的堆棧中
- 通過傳遞引用您 AWS 帳戶中定義的資源的代理對象，該對象是從資源的唯一標識符（例如 ARN）創建的

如果建構的屬性代表另一個資源的建構，則其類型為建構的介面類型。例如，Amazon ECS 建構採用類型`clusterecs.ICluster`的屬性。另一個例子是採用類型屬性 `sourceBucket` (Python:`source_bucket`) 的 CloudFront 分佈結構`s3.IBucket`。

您可以直接傳遞在同一 AWS CDK 應用程序中定義的適當類型的任何資源對象。下列範例會定義 Amazon ECS 叢集，然後使用它來定義 Amazon ECS 服務。

TypeScript

```
const cluster = new ecs.Cluster(this, 'Cluster', { /*...*/ });

const service = new ecs.Ec2Service(this, 'Service', { cluster: cluster });
```

JavaScript

```
const cluster = new ecs.Cluster(this, 'Cluster', { /*...*/ });

const service = new ecs.Ec2Service(this, 'Service', { cluster: cluster });
```

Python

```
cluster = ecs.Cluster(self, "Cluster")

service = ecs.Ec2Service(self, "Service", cluster=cluster)
```

Java

```
Cluster cluster = new Cluster(this, "Cluster");
Ec2Service service = new Ec2Service(this, "Service",
        new Ec2ServiceProps.Builder().cluster(cluster).build());
```

C#

```
var cluster = new Cluster(this, "Cluster");
var service = new Ec2Service(this, "Service", new Ec2ServiceProps { Cluster =
 cluster });
```

引用不同堆棧中的資源

您可以引用不同堆棧中的資源，只要它們是在同一個應用程序中定義的，並且位於相同的 AWS 環境中。通常使用以下模式：

- 將對構造的引用存儲為生成資源的堆棧的屬性。（要獲取對當前構造堆棧的引用，請使用`Stack.of(this)`。）
- 將此引用傳遞給堆棧的構造函數，該構造函數將資源作為參數或屬性消耗。然後消費堆棧將其作為屬性傳遞給需要它的任何構造。

下面的例子定義了一個堆棧stack1。此堆疊定義 Amazon S3 儲存貯體，並將儲存貯體建構的參考存放為堆疊的屬性。然後，應用程序定義了第二個堆棧stack2，它在實例化時接受存儲桶。 stack2例如，可能會定義一個使用存儲桶進行數據存儲的 AWS Glue 表。

TypeScript

```
const prod = { account: '123456789012', region: 'us-east-1' };

const stack1 = new StackThatProvidesABucket(app, 'Stack1', { env: prod });

// stack2 will take a property { bucket: IBucket }
const stack2 = new StackThatExpectsABucket(app, 'Stack2', {
  bucket: stack1.bucket,
  env: prod
});
```

JavaScript

```
const prod = { account: '123456789012', region: 'us-east-1' };

const stack1 = new StackThatProvidesABucket(app, 'Stack1', { env: prod });

// stack2 will take a property { bucket: IBucket }
const stack2 = new StackThatExpectsABucket(app, 'Stack2', {
  bucket: stack1.bucket,
  env: prod
});
```

Python

```
prod = core.Environment(account="123456789012", region="us-east-1")

stack1 = StackThatProvidesABucket(app, "Stack1", env=prod)

# stack2 will take a property "bucket"
stack2 = StackThatExpectsABucket(app, "Stack2", bucket=stack1.bucket, env=prod)
```

Java

```
// Helper method to build an environment
static Environment makeEnv(String account, String region) {
```

```
    return Environment.builder().account(account).region(region)
            .build();
}

App app = new App();

Environment prod = makeEnv("123456789012", "us-east-1");

StackThatProvidesABucket stack1 = new StackThatProvidesABucket(app, "Stack1",
        StackProps.builder().env(prod).build());

// stack2 will take an argument "bucket"
StackThatExpectsABucket stack2 = new StackThatExpectsABucket(app, "Stack,",
        StackProps.builder().env(prod).build(), stack1.bucket);
```

C#

```
Amazon.CDK.Environment makeEnv(string account, string region)
{
    return new Amazon.CDK.Environment { Account = account, Region = region };
}

var prod = makeEnv(account: "123456789012", region: "us-east-1");

var stack1 = new StackThatProvidesABucket(app, "Stack1", new StackProps { Env =
 prod });

// stack2 will take a property "bucket"
var stack2 = new StackThatExpectsABucket(app, "Stack2", new StackProps { Env = prod,
    bucket = stack1.Bucket});
```

如果 AWS CDK 判斷資源位於相同環境中，但位於不同的堆疊中，它會自動合成產生堆疊中的 AWS CloudFormation 匯出，並在消耗堆疊ImportValue中合成 Fn::，以將該資訊從一個堆疊傳輸到另一個堆疊。

解決相依性死鎖

從不同堆棧中的一個堆棧引用資源創建兩個堆棧之間的依賴關係。這樣可以確保它們以正確的順序進行部署。堆棧部署後，這種依賴關係是具體的。之後，從使用堆疊中移除共用資源的使用可能會導致非預期的部署失敗。如果兩個堆棧之間存在另一個依賴關係，強制以相同的順序部署它們，則會發生這種情況。如果 CDK Toolkit 簡單地選擇生產堆棧以首先部署，則也可能在沒有依賴性的情況下發生。匯

AWS CloudFormation 出會從產生堆疊中移除，因為它不再需要，但匯出的資源仍在使用堆疊中使用，因為尚未部署其更新。因此，部署生產者堆疊會失敗。

若要中斷此鎖死，請移除使用耗用堆疊中共用資源的使用。（這將從生產堆棧中刪除自動導出。）接下來，使用與自動生成的導出完全相同的邏輯 ID 手動將相同的導出添加到生產堆棧中。移除使用堆疊中共用資源的使用，並部署這兩個堆疊。然後，刪除手動導出（如果不再需要共享資源），然後再次部署這兩個堆棧。堆疊的exportValue()方法是為此目的建立手動匯出的便利方法。(請參閱連結方法參考中的範例)。

參考帳戶中的 AWS 資源

假設您想使用應用程序中 AWS 帳戶中已經可 AWS CDK 用的資源。這可能是透過主控台、 AWS SDK、直接使 AWS CloudFormation用或在不同 AWS CDK 應用程式中定義的資源。您可以將資源的 ARN（或其他標識屬性或屬性組）轉換為代理對象。代理對象通過調用資源的類的靜態工廠方法作為對資源的引用。

當您創建這樣的代理時，外部資源不會成為您的 AWS CDK 應用程序的一部分。因此，您在 AWS CDK 應用程式中對 Proxy 所做的變更不會影響已部署的資源。然而，代理可以傳遞給需要該類型資源的任何 AWS CDK 方法。

下列範例顯示如何使用 ARN arn: awn: s3:: 以現有儲存貯體為基礎來參考儲存貯體my-bucket-name，以及以具有特定 ID 的現有 VPC 為基礎的 Amazon 虛擬私有雲。

TypeScript

```
// Construct a proxy for a bucket by its name (must be same account)
s3.Bucket.fromBucketName(this, 'MyBucket', 'my-bucket-name');

// Construct a proxy for a bucket by its full ARN (can be another account)
s3.Bucket.fromBucketArn(this, 'MyBucket', 'arn:aws:s3:::my-bucket-name');

// Construct a proxy for an existing VPC from its attribute(s)
ec2.Vpc.fromVpcAttributes(this, 'MyVpc', {
  vpcId: 'vpc-1234567890abcde',
});
```

JavaScript

```
// Construct a proxy for a bucket by its name (must be same account)
s3.Bucket.fromBucketName(this, 'MyBucket', 'my-bucket-name');
```

```
// Construct a proxy for a bucket by its full ARN (can be another account)
s3.Bucket.fromBucketArn(this, 'MyBucket', 'arn:aws:s3:::my-bucket-name');

// Construct a proxy for an existing VPC from its attribute(s)
ec2.Vpc.fromVpcAttributes(this, 'MyVpc', {
  vpcId: 'vpc-1234567890abcde'
});
```

Python

```
# Construct a proxy for a bucket by its name (must be same account)
s3.Bucket.from_bucket_name(self, "MyBucket", "my-bucket-name")

# Construct a proxy for a bucket by its full ARN (can be another account)
s3.Bucket.from_bucket_arn(self, "MyBucket", "arn:aws:s3:::my-bucket-name")

# Construct a proxy for an existing VPC from its attribute(s)
ec2.Vpc.from_vpc_attributes(self, "MyVpc", vpc_id="vpc-1234567890abcdef")
```

Java

```
// Construct a proxy for a bucket by its name (must be same account)
Bucket.fromBucketName(this, "MyBucket", "my-bucket-name");

// Construct a proxy for a bucket by its full ARN (can be another account)
Bucket.fromBucketArn(this, "MyBucket",
        "arn:aws:s3:::my-bucket-name");

// Construct a proxy for an existing VPC from its attribute(s)
Vpc.fromVpcAttributes(this, "MyVpc", VpcAttributes.builder()
        .vpcId("vpc-1234567890abcdef").build());
```

C#

```
// Construct a proxy for a bucket by its name (must be same account)
Bucket.FromBucketName(this, "MyBucket", "my-bucket-name");

// Construct a proxy for a bucket by its full ARN (can be another account)
Bucket.FromBucketArn(this, "MyBucket", "arn:aws:s3:::my-bucket-name");

// Construct a proxy for an existing VPC from its attribute(s)
```

```
Vpc.FromVpcAttributes(this, "MyVpc", new VpcAttributes
{
    VpcId = "vpc-1234567890abcdef"
});
```

讓我們仔細看看該Vpc.fromLookup()方法。由於ec2.Vpc建構很複雜，因此您可能需要多種方式選取要與 CDK 應用程式搭配使用的 VPC。為了解決這個問題，VPC 建構具有fromLookup靜態方法 (Python:from_lookup)，可讓您透過在合成時查詢 AWS 帳戶來查詢所需的 Amazon VPC。

若要使用Vpc.fromLookup()，合成堆疊的系統必須能夠存取擁有 Amazon VPC 的帳戶。這是因為 CDK 工具組會查詢帳戶，以便在合成時找到合適的 Amazon VPC。

此外，僅Vpc.fromLookup()適用於使用明確帳戶和區域定義的堆疊 (請參閱the section called “環境”)。如果 AWS CDK 嘗試從與環境無關的堆疊查詢 Amazon VPC，則 CDK 工具組不知道要查詢哪個環境以尋找 VPC。

您必須提供足以唯一識別 AWS 帳戶中 VPC 的Vpc.fromLookup()屬性。例如，只能有一個預設 VPC，因此將 VPC 指定為預設值就足夠了。

TypeScript

```
ec2.Vpc.fromLookup(this, 'DefaultVpc', {
  isDefault: true
});
```

JavaScript

```
ec2.Vpc.fromLookup(this, 'DefaultVpc', {
  isDefault: true
});
```

Python

```
ec2.Vpc.from_lookup(self, "DefaultVpc", is_default=True)
```

Java

```
Vpc.fromLookup(this, "DefaultVpc", VpcLookupOptions.builder()
        .isDefault(true).build());
```

C#

```
Vpc.FromLookup(this, id = "DefaultVpc", new VpcLookupOptions { IsDefault = true });
```

您也可以使用該tags內容按標籤查詢 VPC。您可以在建立 Amazon VPC 時使用 AWS CloudFormation 或將標籤新增至其。 AWS CDK您可以在建立後隨時使用 AWS Management Console AWS CLI、或 AWS SDK 編輯標籤。除了您自己新增的任何標籤之外，還會 AWS CDK 自動將下列標籤新增至其建立的所有 VPC。

- 名稱 — VPC 的名稱。
- aws-cdk: 子網路名稱 — 子網路的名稱。
- aws-cdk: 子網路類型 — 子網路的類型：「公用」、「私用」或「隔離」。

TypeScript

```
ec2.Vpc.fromLookup(this, 'PublicVpc',
    {tags: {'aws-cdk:subnet-type': "Public"}});
```

JavaScript

```
ec2.Vpc.fromLookup(this, 'PublicVpc',
    {tags: {'aws-cdk:subnet-type': "Public"}});
```

Python

```
ec2.Vpc.from_lookup(self, "PublicVpc",
    tags={"aws-cdk:subnet-type": "Public"})
```

Java

```
Vpc.fromLookup(this, "PublicVpc", VpcLookupOptions.builder()
        .tags(java.util.Map.of("aws-cdk:subnet-type", "Public"))  // Java 9 or later
        .build());
```

C#

```
Vpc.FromLookup(this, id = "PublicVpc", new VpcLookupOptions
```

```
    { Tags = new Dictionary<string, string> { ["aws-cdk:subnet-type"] =
 "Public" });
```

的結果`Vpc.fromLookup()`會快取在專`cdk.context.json`案的檔案中。(請參閱 the section called "Context")。將此檔案提交至版本控制，以便您的應用程式繼續參考相同的 Amazon VPC。即使您稍後以可能導致選取不同 VPC 的方式變更 VPC 的屬性，此功能仍然有效。如果您要在無法存取定義 VPC 的 AWS 帳戶 (例如 CDK Pipelines) 的環境中部署堆疊，這一點就特別重要。

儘管您可以在使用 AWS CDK 應用程序中定義的類似資源的任何位置使用外部資源，但無法對其進行修改。例如，在外部調用 `addToResourcePolicy` (Python:`add_to_resource_policy`) 什麼都`s3.Bucket`不做。

資源實體名稱

中的資源邏輯名稱與 AWS CloudFormation 在部署資源之 AWS Management Console 後顯示的資源名稱不同 AWS CloudFormation。 AWS CDK 呼叫這些姓氏的實體名稱。

例如， AWS CloudFormation 可能會使用上一個範例`Stack2MyBucket4DD88B4F`中的邏輯 ID 以實體名稱建立 Amazon S3 儲存貯體`stack2mybucket4dd88b4f-iuv1rbv9z3to`。

您可以在建立表示資源的建構時，使用屬性 <resourceType>Name 來指定實體名稱。下列範例會使用實體名稱建立 Amazon S3 儲存貯體`my-bucket-name`。

TypeScript

```
const bucket = new s3.Bucket(this, 'MyBucket', {
  bucketName: 'my-bucket-name',
});
```

JavaScript

```
const bucket = new s3.Bucket(this, 'MyBucket', {
  bucketName: 'my-bucket-name'
});
```

Python

```
bucket = s3.Bucket(self, "MyBucket", bucket_name="my-bucket-name")
```

Java

```
Bucket bucket = Bucket.Builder.create(this, "MyBucket")
        .bucketName("my-bucket-name").build();
```

C#

```
var bucket = new Bucket(this, "MyBucket", new BucketProps { BucketName = "my-bucket-
name" });
```

在中為資源指定實體名稱有一些缺點 AWS CloudFormation。最重要的是，對需要取代資源的已部署資源所做的任何變更 (例如，在建立之後不可變的資源屬性變更) 如果資源已指定實體名稱，就會失敗。如果您最終處於該狀態，唯一的解決方案是刪除 AWS CloudFormation 堆棧，然後再次部署 AWS CDK 應用程序。如需詳細資訊，請參閱AWS CloudFormation 文件。

在某些情況下，例如建立具有跨環境參照的 AWS CDK 應用程式時，需要實體名稱 AWS CDK 才能正常運作。在這些情況下，如果您不想打擾自己想出一個物理名稱，則可以為您 AWS CDK 命名它。若要這樣做，請使用特殊值`PhysicalName.GENERATE_IF_NEEDED`，如下所示。

TypeScript

```
const bucket = new s3.Bucket(this, 'MyBucket', {
  bucketName: core.PhysicalName.GENERATE_IF_NEEDED,
});
```

JavaScript

```
const bucket = new s3.Bucket(this, 'MyBucket', {
  bucketName: core.PhysicalName.GENERATE_IF_NEEDED
});
```

Python

```
bucket = s3.Bucket(self, "MyBucket",
                           bucket_name=core.PhysicalName.GENERATE_IF_NEEDED)
```

Java

```
Bucket bucket = Bucket.Builder.create(this, "MyBucket")
```

```
        .bucketName(PhysicalName.GENERATE_IF_NEEDED).build();
```

C#

```
var bucket = new Bucket(this, "MyBucket", new BucketProps
    { BucketName = PhysicalName.GENERATE_IF_NEEDED });
```

傳遞唯一資源識別碼

如前一節所述，您應該盡可能透過參照傳遞資源。但是，在某些情況下，您別無選擇，只能通過其中一個屬性引用資源。範例使用案例包括下列各項：

- 當您使用低級 AWS CloudFormation 資源時。
- 當您需要將資源公開給 AWS CDK 應用程式的執行階段元件時，例如透過環境變數參考 Lambda 函數時。

這些識別碼可作為資源的屬性使用，如下所示。

TypeScript

```
bucket.bucketName
lambdaFunc.functionArn
securityGroup.groupArn
```

JavaScript

```
bucket.bucketName
lambdaFunc.functionArn
securityGroup.groupArn
```

Python

```
bucket.bucket_name
lambda_func.function_arn
security_group_arn
```

Java

Java AWS CDK 綁定使用的屬性吸氣方法。

```
bucket.getBucketName()
lambdaFunc.getFunctionArn()
securityGroup.getGroupArn()
```

C#

```
bucket.BucketName
lambdaFunc.FunctionArn
securityGroup.GroupArn
```

下列範例顯示如何將產生的值區名稱傳遞給 AWS Lambda 函數。

TypeScript

```
const bucket = new s3.Bucket(this, 'Bucket');

new lambda.Function(this, 'MyLambda', {
  // ...
  environment: {
    BUCKET_NAME: bucket.bucketName,
  },
});
```

JavaScript

```
const bucket = new s3.Bucket(this, 'Bucket');

new lambda.Function(this, 'MyLambda', {
  // ...
  environment: {
    BUCKET_NAME: bucket.bucketName
  }
});
```

Python

```
bucket = s3.Bucket(self, "Bucket")

lambda.Function(self, "MyLambda", environment=dict(BUCKET_NAME=bucket.bucket_name))
```

Java

```
final Bucket bucket = new Bucket(this, "Bucket");

Function.Builder.create(this, "MyLambda")
        .environment(java.util.Map.of(    // Java 9 or later
                "BUCKET_NAME", bucket.getBucketName()))
        .build();
```

C#

```
var bucket = new Bucket(this, "Bucket");

new Function(this, "MyLambda", new FunctionProps
{
    Environment = new Dictionary<string, string>
    {
        ["BUCKET_NAME"] = bucket.BucketName
    }
});
```

授與資源之間的權限

透過提供簡單的意向型 API 來表示權限需求，較高層級的建構可達成最低權限的權限。例如，許多 L2 建構都提供授予方法，您可以使用這些方法授與實體 (例如 IAM 角色或使用者) 使用資源的權限，而無需手動建立 IAM 權限陳述式。

下列範例會建立許可，以允許 Lambda 函數的執行角色讀取和寫入特定 Amazon S3 儲存貯體的物件。如果 Amazon S3 儲存貯體使用 AWS KMS 金鑰加密，此方法也會授與 Lambda 函數執行角色的許可，以便使用金鑰進行解密。

TypeScript

```
if (bucket.grantReadWrite(func).success) {
  // ...
}
```

JavaScript

```
if ( bucket.grantReadWrite(func).success) {
  // ...
```

```
}
```

Python

```
if bucket.grant_read_write(func).success:
    # ...
```

Java

```
if (bucket.grantReadWrite(func).getSuccess()) {
    // ...
}
```

C#

```
if (bucket.GrantReadWrite(func).Success)
{
    // ...
}
```

授予方法返回一個`iam.Grant`對象。使用`Grant`物件的`success`屬性來判斷授權是否已有效套用 (例如，可能尚未套用至外部資源)。您也可以使用`Grant`物件的 `assertSuccess` (Python:`assert_success`) 方法來強制授權已成功套用。

如果特定的使用案例無法使用特定的授權方法，您可以使用一般授權方法來定義具有指定動作清單的新授權。

下列範例顯示如何授與 Lambda 函數存取權限存取 Amazon DynamoDB 動`CreateBackup`作。

TypeScript

```
table.grant(func, 'dynamodb:CreateBackup');
```

JavaScript

```
table.grant(func, 'dynamodb:CreateBackup');
```

Python

```
table.grant(func, "dynamodb:CreateBackup")
```

Java

```
table.grant(func, "dynamodb:CreateBackup");
```

C#

```
table.Grant(func, "dynamodb:CreateBackup");
```

執行程式碼時，許多資源 (例如 Lambda 函數) 都需要扮演角色。組態特性可讓您指定`iam.IRole`。如果未指定角色，函數會自動建立專門用於此用途的角色。然後，您可以在資源上使用授與方法，將語句添加到角色。

授予方法是使用較低級別的 API 建立的，用於處理 IAM 政策。原則模型為PolicyDocument物件。使用`addToRolePolicy`方法 (Python:) 將陳述式直接新增至角色 (或建構的附加角色`add_to_role_policy`)，或使用 (Python:) 方法新增至資源的`Bucket`政策`addToResourcePolicy` (例如政策`add_to_resource_policy`)。

資源指標和警示

許多資源都會發出可用於設定監控儀表板和警示的 CloudWatch 指標。較高層級的建構具有度量方法，可讓您存取量度，而無需查詢要使用的正確名稱。

下列範例顯示如何在 `ApproximateNumberOfMessagesNotVisible` Amazon SQS 佇列超過 100 時定義警示。

TypeScript

```
import * as cw from '@aws-cdk/aws-cloudwatch';
import * as sqs from '@aws-cdk/aws-sqs';
import { Duration } from '@aws-cdk/core';

const queue = new sqs.Queue(this, 'MyQueue');

const metric = queue.metricApproximateNumberOfMessagesNotVisible({
  label: 'Messages Visible (Approx)',
  period: Duration.minutes(5),
  // ...
});
metric.createAlarm(this, 'TooManyMessagesAlarm', {
  comparisonOperator: cw.ComparisonOperator.GREATER_THAN_THRESHOLD,
  threshold: 100,
```

```
  // ...
});
```

JavaScript

```
const cw = require('@aws-cdk/aws-cloudwatch');
const sqs = require('@aws-cdk/aws-sqs');
const { Duration } = require('@aws-cdk/core');

const queue = new sqs.Queue(this, 'MyQueue');

const metric = queue.metricApproximateNumberOfMessagesNotVisible({
  label: 'Messages Visible (Approx)',
  period: Duration.minutes(5)
  // ...
});
metric.createAlarm(this, 'TooManyMessagesAlarm', {
  comparisonOperator: cw.ComparisonOperator.GREATER_THAN_THRESHOLD,
  threshold: 100
  // ...
});
```

Python

```
import aws_cdk.aws_cloudwatch as cw
import aws_cdk.aws_sqs as sqs
from aws_cdk.core import Duration

queue = sqs.Queue(self, "MyQueue")
metric = queue.metric_approximate_number_of_messages_not_visible(
    label="Messages Visible (Approx)",
    period=Duration.minutes(5),
    # ...
)
metric.create_alarm(self, "TooManyMessagesAlarm",
    comparison_operator=cw.ComparisonOperator.GREATER_THAN_THRESHOLD,
    threshold=100,
    # ...
)
```

Java

```
import software.amazon.awscdk.core.Duration;
```

```
import software.amazon.awscdk.services.sqs.Queue;
import software.amazon.awscdk.services.cloudwatch.Metric;
import software.amazon.awscdk.services.cloudwatch.MetricOptions;
import software.amazon.awscdk.services.cloudwatch.CreateAlarmOptions;
import software.amazon.awscdk.services.cloudwatch.ComparisonOperator;

Queue queue = new Queue(this, "MyQueue");

Metric metric = queue
        .metricApproximateNumberOfMessagesNotVisible(MetricOptions.builder()
                .label("Messages Visible (Approx)")
                .period(Duration.minutes(5)).build());

metric.createAlarm(this, "TooManyMessagesAlarm", CreateAlarmOptions.builder()
                .comparisonOperator(ComparisonOperator.GREATER_THAN_THRESHOLD)
                .threshold(100)
                // ...
                .build());
```

C#

```
using cdk = Amazon.CDK;
using cw = Amazon.CDK.AWS.CloudWatch;
using sqs = Amazon.CDK.AWS.SQS;

var queue = new sqs.Queue(this, "MyQueue");
var metric = queue.MetricApproximateNumberOfMessagesNotVisible(new cw.MetricOptions
{
    Label = "Messages Visible (Approx)",
    Period = cdk.Duration.Minutes(5),
    // ...
});
metric.CreateAlarm(this, "TooManyMessagesAlarm", new cw.CreateAlarmOptions
{
    ComparisonOperator = cw.ComparisonOperator.GREATER_THAN_THRESHOLD,
    Threshold = 100,
    // ..
});
```

如果特定測量結果沒有方法，您可以使用一般測量結果方法來手動指定測量結果名稱。

指標也可以新增至 CloudWatch 儀表板。請參閱 CloudWatch。

網路流量

在許多情況下，您必須在網路上啟用權限才能讓應用程式運作，例如當計算基礎結構需要存取持續性層時。建立或接聽連線的資源會公開啟用流量流程的方法，包括設定安全性群組規則或網路 ACL。

IconnecTable 資源具有一個`connections`屬性，該屬性是通往網路流量規則組態的閘道。

您可以使用`allow`方法讓資料在指定的網路路徑上流動。下列範例會啟用從 Amazon EC2 Auto Scaling 群組連線至網路和傳入連線的 HTTPS 連線`fleet2`。

TypeScript

```
import * as asg from '@aws-cdk/aws-autoscaling';
import * as ec2 from '@aws-cdk/aws-ec2';

const fleet1: asg.AutoScalingGroup = asg.AutoScalingGroup(/*...*/);

// Allow surfing the (secure) web
fleet1.connections.allowTo(new ec2.Peer.anyIpv4(), new ec2.Port({ fromPort: 443,
 toPort: 443 }));

const fleet2: asg.AutoScalingGroup = asg.AutoScalingGroup(/*...*/);
fleet1.connections.allowFrom(fleet2, ec2.Port.AllTraffic());
```

JavaScript

```
const asg = require('@aws-cdk/aws-autoscaling');
const ec2 = require('@aws-cdk/aws-ec2');

const fleet1 = asg.AutoScalingGroup();

// Allow surfing the (secure) web
fleet1.connections.allowTo(new ec2.Peer.anyIpv4(), new ec2.Port({ fromPort: 443,
 toPort: 443 }));

const fleet2 = asg.AutoScalingGroup();
fleet1.connections.allowFrom(fleet2, ec2.Port.AllTraffic());
```

Python

```
import aws_cdk.aws_autoscaling as asg
import aws_cdk.aws_ec2 as ec2
```

```
fleet1 = asg.AutoScalingGroup( ... )

# Allow surfing the (secure) web
fleet1.connections.allow_to(ec2.Peer.any_ipv4(),
  ec2.Port(PortProps(from_port=443, to_port=443)))

fleet2 = asg.AutoScalingGroup( ... )
fleet1.connections.allow_from(fleet2, ec2.Port.all_traffic())
```

Java

```
import software.amazon.awscdk.services.autoscaling.AutoScalingGroup;
import software.amazon.awscdk.services.ec2.Peer;
import software.amazon.awscdk.services.ec2.Port;

AutoScalingGroup fleet1 = AutoScalingGroup.Builder.create(this, "MyFleet")
        /* ... */.build();

// Allow surfing the (secure) Web
fleet1.getConnections().allowTo(Peer.anyIpv4(),
        Port.Builder.create().fromPort(443).toPort(443).build());

AutoScalingGroup fleet2 = AutoScalingGroup.Builder.create(this, "MyFleet2")
        /* ... */.build();
fleet1.getConnections().allowFrom(fleet2, Port.allTraffic());
```

C#

```
using cdk = Amazon.CDK;
using asg = Amazon.CDK.AWS.AutoScaling;
using ec2 = Amazon.CDK.AWS.EC2;

// Allow surfing the (secure) Web
var fleet1 = new asg.AutoScalingGroup(this, "MyFleet", new asg.AutoScalingGroupProps
 { /* ... */ });
fleet1.Connections.AllowTo(ec2.Peer.AnyIpv4(), new ec2.Port(new ec2.PortProps
  { FromPort = 443, ToPort = 443 });

var fleet2 = new asg.AutoScalingGroup(this, "MyFleet2", new
 asg.AutoScalingGroupProps { /* ... */ });
fleet1.Connections.AllowFrom(fleet2, ec2.Port.AllTraffic());
```

某些資源具有與其相關聯的預設連接埠。範例包括公用連接埠上負載平衡器的接聽程式，以及資料庫引擎接受 Amazon RDS 資料庫執行個體連線的連接埠。在這種情況下，您可以強制執行嚴格的網路控制，而不必手動指定連接埠。若要這麼做，請使用`allowDefaultPortFrom`和`allowToDefaultPort`方法 (Python:`allow_default_port_from`,`allow_to_default_port`)。

下列範例顯示如何啟用來自任何 IPV4 位址的連線，以及如何啟用 Auto Scaling 群組的連線以存取資料庫。

TypeScript

```
listener.connections.allowDefaultPortFromAnyIpv4('Allow public access');

fleet.connections.allowToDefaultPort(rdsDatabase, 'Fleet can access database');
```

JavaScript

```
listener.connections.allowDefaultPortFromAnyIpv4('Allow public access');

fleet.connections.allowToDefaultPort(rdsDatabase, 'Fleet can access database');
```

Python

```
listener.connections.allow_default_port_from_any_ipv4("Allow public access")

fleet.connections.allow_to_default_port(rds_database, "Fleet can access database")
```

Java

```
listener.getConnections().allowDefaultPortFromAnyIpv4("Allow public access");

fleet.getConnections().AllowToDefaultPort(rdsDatabase, "Fleet can access database");
```

C#

```
listener.Connections.AllowDefaultPortFromAnyIpv4("Allow public access");

fleet.Connections.AllowToDefaultPort(rdsDatabase, "Fleet can access database");
```

事件處理

某些資源可以充當事件來源。使用方addEventNotification法 (Python:add_event_notification) 將事件目標註冊到資源所發出的特定事件類型。除此之外，addXxxNotification方法還提供了一種簡單的方法來註冊常見事件類型的處理常式。

下列範例示範如何在物件新增至 Amazon S3 儲存貯體時觸發 Lambda 函數。

TypeScript

```
import * as s3nots from '@aws-cdk/aws-s3-notifications';

const handler = new lambda.Function(this, 'Handler', { /*…*/ });
const bucket = new s3.Bucket(this, 'Bucket');
bucket.addObjectCreatedNotification(new s3nots.LambdaDestination(handler));
```

JavaScript

```
const s3nots = require('@aws-cdk/aws-s3-notifications');

const handler = new lambda.Function(this, 'Handler', { /*…*/ });
const bucket = new s3.Bucket(this, 'Bucket');
bucket.addObjectCreatedNotification(new s3nots.LambdaDestination(handler));
```

Python

```
import aws_cdk.aws_s3_notifications as s3_nots

handler = lambda_.Function(self, "Handler", ...)
bucket = s3.Bucket(self, "Bucket")
bucket.add_object_created_notification(s3_nots.LambdaDestination(handler))
```

Java

```
import software.amazon.awscdk.services.s3.Bucket;
import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.s3.notifications.LambdaDestination;

Function handler = Function.Builder.create(this, "Handler")/* ... */.build();
Bucket bucket = new Bucket(this, "Bucket");
```

```
bucket.addObjectCreatedNotification(new LambdaDestination(handler));
```

C#

```
using lambda = Amazon.CDK.AWS.Lambda;
using s3 = Amazon.CDK.AWS.S3;
using s3Nots = Amazon.CDK.AWS.S3.Notifications;

var handler = new lambda.Function(this, "Handler", new lambda.FunctionProps { .. });
var bucket = new s3.Bucket(this, "Bucket");
bucket.AddObjectCreatedNotification(new s3Nots.LambdaDestination(handler));
```

移除政策

維護持續性資料的資源 (例如資料庫、Amazon S3 儲存貯體和 Amazon ECR 登錄) 都有移除政策。移除原則會指出當包含永久性物件的 AWS CDK 堆疊銷毀時，是否要刪除持續性物件。指定移除原則的值可透過 AWS CDK `core`模組中的`RemovalPolicy`列舉取得。

> **Note**
>
> 除了那些永久存儲數據的資源之外，也可能具有`removalPolicy`用於不同目的的的資源。例如，Lambda 函數版本會使用`removalPolicy`屬性來判斷部署新版本時是否保留指定版本。與 Amazon S3 儲存貯體或 DynamoDB 表格上的移除政策相比，這些檔案具有不同的意義和預設值。

Value	意義
`RemovalPolicy###`	Keep the contents of the resource when destroying the stack (default). The resource is orphaned from the stack and must be deleted manually. If you attempt to re-deploy the stack while the resource still exists, you will receive an error message due to a name conflict.
`RemovalPolicy. ###`	The resource will be destroyed along with the stack.

AWS CloudFormation 不會移除包含檔案的 Amazon S3 儲存貯體，即使其移除政策設為也是如此DESTROY。嘗試這樣做是一個 AWS CloudFormation 錯誤。若要在銷毀值區之前先 AWS CDK 刪除值區中的所有檔案，請將值區的autoDeleteObjects屬性設定為true。

以下是使用和設定為建立 Amazon S3 儲存貯DESTROY體RemovalPolicyautoDeleteOjbects的範例true。

TypeScript

```
import * as cdk from '@aws-cdk/core';
import * as s3 from '@aws-cdk/aws-s3';

export class CdkTestStack extends cdk.Stack {
  constructor(scope: cdk.Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    const bucket = new s3.Bucket(this, 'Bucket', {
      removalPolicy: cdk.RemovalPolicy.DESTROY,
      autoDeleteObjects: true
    });
  }
}
```

JavaScript

```
const cdk = require('@aws-cdk/core');
const s3 = require('@aws-cdk/aws-s3');

class CdkTestStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    const bucket = new s3.Bucket(this, 'Bucket', {
      removalPolicy: cdk.RemovalPolicy.DESTROY,
      autoDeleteObjects: true
    });
  }
}

module.exports = { CdkTestStack }
```

Python

```
import aws_cdk.core as cdk
import aws_cdk.aws_s3 as s3

class CdkTestStack(cdk.stack):
    def __init__(self, scope: cdk.Construct, id: str, **kwargs):
        super().__init__(scope, id, **kwargs)

        bucket = s3.Bucket(self, "Bucket",
            removal_policy=cdk.RemovalPolicy.DESTROY,
            auto_delete_objects=True)
```

Java

```
software.amazon.awscdk.core.*;
import software.amazon.awscdk.services.s3.*;

public class CdkTestStack extends Stack {
    public CdkTestStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public CdkTestStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        Bucket.Builder.create(this, "Bucket")
                .removalPolicy(RemovalPolicy.DESTROY)
                .autoDeleteObjects(true).build();
    }
}
```

C#

```
using Amazon.CDK;
using Amazon.CDK.AWS.S3;

public CdkTestStack(Construct scope, string id, IStackProps props) : base(scope, id,
 props)
{
    new Bucket(this, "Bucket", new BucketProps {
        RemovalPolicy = RemovalPolicy.DESTROY,
```

```
        AutoDeleteObjects = true
    });
}
```

您也可以透過`applyRemovalPolicy()`方法將移除原則直接套用至基礎 AWS CloudFormation 資源。此方法適用於某些在其 L2 資源 prop 中沒有`removalPolicy`屬性的有狀態資源。範例如下：

- AWS CloudFormation 堆疊
- Amazon Cognito 使用者集區
- Amazon DocumentDB 實例
- Amazon EC2 卷
- Amazon OpenSearch 服務域
- Amazon FSx 檔案系統
- Amazon SQS 佇列

TypeScript

```
const resource = bucket.node.findChild('Resource') as cdk.CfnResource;
resource.applyRemovalPolicy(cdk.RemovalPolicy.DESTROY);
```

JavaScript

```
const resource = bucket.node.findChild('Resource');
resource.applyRemovalPolicy(cdk.RemovalPolicy.DESTROY);
```

Python

```
resource = bucket.node.find_child('Resource')
resource.apply_removal_policy(cdk.RemovalPolicy.DESTROY);
```

Java

```
CfnResource resource = (CfnResource)bucket.node.findChild("Resource");
resource.applyRemovalPolicy(cdk.RemovalPolicy.DESTROY);
```

C#

```
var resource = (CfnResource)bucket.node.findChild('Resource');
resource.ApplyRemovalPolicy(cdk.RemovalPolicy.DESTROY);
```

> **Note**
>
> AWS CDK的`RemovalPolicy`翻譯為 AWS CloudFormation的`DeletionPolicy`。但是，中的預設值 AWS CDK 是保留資料，這與 AWS CloudFormation 預設值相反。

識別碼

構建 AWS Cloud Development Kit (AWS CDK) 應用程序時，您將使用許多類型的標識符和名稱。若要 AWS CDK 有效地使用並避免錯誤，請務必瞭解識別碼的類型。

識別碼在建立它們的範圍內必須是唯一的；在您的 AWS CDK 應用程式中，它們不需要是全域唯一的。

如果您嘗試在相同範圍內建立具有相同值的識別項，則 AWS CDK 會擲回例外狀況。

主題

- 建構識別碼
- 路徑
- 唯一 ID
- 邏輯識別碼

建構識別碼

最常見的標識符是實例化構造對象時作為第二個參數傳遞的標識符。`id`與所有標識符一樣，此標識符只需要在創建它的範圍內是唯一的，這是實例化構造對象時的第一個參數。

> **Note**
>
> 堆疊`id`的也是您在中用來參考它的識別碼the section called “AWS CDK 工具包”。

讓我們來看一個例子，我們有兩個結構MyBucket在我們的應用程序標識符。第一個是在具有標識符的堆棧的範圍內定義的Stack1。第二個在具有標識符的堆棧的範圍內定義Stack2。因為它們是在不同的範圍中定義的，所以這不會引起任何衝突，並且它們可以在同一個應用程序中並存而不會出現問題。

TypeScript

```
import { App, Stack, StackProps } from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as s3 from 'aws-cdk-lib/aws-s3';

class MyStack extends Stack {
  constructor(scope: Construct, id: string, props: StackProps = {}) {
    super(scope, id, props);

    new s3.Bucket(this, 'MyBucket');
  }
}

const app = new App();
new MyStack(app, 'Stack1');
new MyStack(app, 'Stack2');
```

JavaScript

```
const { App , Stack } = require('aws-cdk-lib');
const s3 = require('aws-cdk-lib/aws-s3');

class MyStack extends Stack {
  constructor(scope, id, props = {}) {
    super(scope, id, props);

    new s3.Bucket(this, 'MyBucket');
  }
}

const app = new App();
new MyStack(app, 'Stack1');
new MyStack(app, 'Stack2');
```

Python

```
from aws_cdk import App, Construct, Stack, StackProps
```

```
from constructs import Construct
from aws_cdk import aws_s3 as s3

class MyStack(Stack):

    def __init__(self, scope: Construct, id: str, **kwargs):

        super().__init__(scope, id, **kwargs)
        s3.Bucket(self, "MyBucket")

app = App()
MyStack(app, 'Stack1')
MyStack(app, 'Stack2')
```

Java

```
// MyStack.java
package com.myorg;

import software.amazon.awscdk.App;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.constructs.Construct;
import software.amazon.awscdk.services.s3.Bucket;

public class MyStack extends Stack {
    public MyStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public MyStack(final Construct scope, final String id, final StackProps props) {
        super(scope, id, props);
        new Bucket(this, "MyBucket");
    }
}

// Main.java
package com.myorg;

import software.amazon.awscdk.App;

public class Main {
    public static void main(String[] args) {
```

```
        App app = new App();
        new MyStack(app, "Stack1");
        new MyStack(app, "Stack2");
    }
}
```

C#

```
using Amazon.CDK;
using constructs;
using Amazon.CDK.AWS.S3;

public class MyStack : Stack
{
    public MyStack(Construct scope, string id, IStackProps props) : base(scope, id,
 props)
    {
        new Bucket(this, "MyBucket");
    }
}

class Program
{
    static void Main(string[] args)
    {
        var app = new App();
        new MyStack(app, "Stack1");
        new MyStack(app, "Stack2");
    }
}
```

路徑

AWS CDK 應用程式中的建構會形成植根於App類別的階層架構。我們將來自給定建構的 ID 集合、其父建構、其祖父系結構等引用到建構樹的根目錄，做為路徑。

通 AWS CDK 常會將範本中的路徑顯示為字串。層級中的 ID 會以斜線分隔，從根App執行個體 (通常是堆疊) 下的節點開始。例如，上一個程式碼範例中兩個 Amazon S3 儲存貯體資源的路徑為`Stack1/MyBucket`和`Stack2/MyBucket`。

您可以透過程式設計方式存取任何建構的路徑，如下列範例所示。這得到了myConstruct（或者my_construct，正如 Python 開發人員所寫的那樣）的路徑。由於 ID 在建立的範圍內必須是唯一的，因此它們的路徑在 AWS CDK 應用程式中永遠是唯一的。

TypeScript

```
const path: string = myConstruct.node.path;
```

JavaScript

```
const path = myConstruct.node.path;
```

Python

```
path = my_construct.node.path
```

Java

```
String path = myConstruct.getNode().getPath();
```

C#

```
string path = myConstruct.Node.Path;
```

唯一 ID

AWS CloudFormation 要求範本中的所有邏輯 ID 都是唯一的。因此， AWS CDK 必須能夠為應用程式中的每個建構產生唯一的識別碼。資源具有全局唯一的路徑（從堆棧到特定資源的所有範圍的名稱）。因此， AWS CDK 會透過串連路徑的元素並新增 8 位數的雜湊來產生必要的唯一識別碼。（散列是區分不同路徑所必需的，例如A/B/C和A/BC，這將導致相同的 AWS CloudFormation 標識符。 AWS CloudFormation 識別碼是字母數字，不能包含斜線或其他分隔符號字元。) AWS CDK 調用此字符串構造的唯一 ID。

通常，您的 AWS CDK 應用程序不需要了解唯一 ID。不過，您可以透過程式設計方式存取任何建構的唯一 ID，如下列範例所示。

TypeScript

```
const uid: string = Names.uniqueId(myConstruct);
```

JavaScript

```
const uid = Names.uniqueId(myConstruct);
```

Python

```
uid = Names.unique_id(my_construct)
```

Java

```
String uid = Names.uniqueId(myConstruct);
```

C#

```
string uid = Names.Uniqueid(myConstruct);
```

地址是另一種唯一標識符，可唯一區分 CDK 資源。從路徑的 SHA-1 哈希值派生，它不是人類可讀的。但是，它的常量，相對較短的長度（總是 42 個十六進制字符）使其在「傳統」唯一 ID 可能太長的情況下非常有用。某些結構可能會使用合成 AWS CloudFormation 模板中的地址，而不是唯一 ID。同樣，您的應用程序通常不需要了解其結構的地址，但是您可以按如下方式檢索構造的地址。

TypeScript

```
const addr: string = myConstruct.node.addr;
```

JavaScript

```
const addr = myConstruct.node.addr;
```

Python

```
addr = my_construct.node.addr
```

Java

```
String addr = myConstruct.getNode().getAddr();
```

C#

```
string addr = myConstruct.Node.Addr;
```

邏輯識別碼

唯一 ID 可做為代表 AWS 資源之建構所產生 AWS CloudFormation 範本中資源的邏輯識別碼 (或邏輯名稱)。

例如，在上一個範例中建立的 Amazon S3 儲存貯體`Stack2`會產生`AWS::S3::Bucket`資源。資源的邏輯 ID `Stack2MyBucket4DD88B4F` 位於產生的 AWS CloudFormation 範本中。(如需如何產生此識別碼的詳細資訊，請參閱the section called “唯一 ID”。)

邏輯 ID 穩定性

避免在資源建立後變更資源的邏輯 ID。 AWS CloudFormation 透過其邏輯 ID 識別資源。因此，如果您變更資源的邏輯 ID， AWS CloudFormation 會使用新的邏輯 ID 建立新資源，然後刪除現有的邏輯 ID。視資源類型而定，這可能會造成服務中斷、資料遺失或同時造成兩者。

代幣

令牌表示只能在應用程序生命週期中稍後解析的值。例如，您在 CDK 應用程式中定義的 Amazon 簡易儲存服務 (Amazon S3) 儲存貯體的名稱，只有在合成範 AWS CloudFormation 本時才會分配。如果你打印`bucket.bucketName`屬性，這是一個字符串，你會看到它包含如下內容：

```
${TOKEN[Bucket.Name.1234]}
```

這就是 AWS CDK 編碼令牌的方式，其值在施工時尚未知道，但稍後將可用。 AWS CDK 調用這些佔位符令牌。在這種情況下，它是編碼為字符串的令牌。

您可以傳遞此字符串，就好像它是存儲桶的名稱一樣。在下列範例中，值區名稱會指定為 AWS Lambda 函數的環境變數。

TypeScript

```
const bucket = new s3.Bucket(this, 'MyBucket');

const fn = new lambda.Function(stack, 'MyLambda', {
  // ...
  environment: {
    BUCKET_NAME: bucket.bucketName,
  }
});
```

JavaScript

```
const bucket = new s3.Bucket(this, 'MyBucket');

const fn = new lambda.Function(stack, 'MyLambda', {
  // ...
  environment: {
    BUCKET_NAME: bucket.bucketName
  }
});
```

Python

```
bucket = s3.Bucket(self, "MyBucket")

fn = lambda_.Function(stack, "MyLambda",
        environment=dict(BUCKET_NAME=bucket.bucket_name))
```

Java

```
final Bucket bucket = new Bucket(this, "MyBucket");

Function fn = Function.Builder.create(this, "MyLambda")
        .environment(java.util.Map.of(    // Map.of requires Java 9+
            "BUCKET_NAME", bucket.getBucketName()))
        .build();
```

C#

```
var bucket = new s3.Bucket(this, "MyBucket");
```

```
var fn = new Function(this, "MyLambda", new FunctionProps {
    Environment = new Dictionary<string, string>
    {
        ["BUCKET_NAME"] = bucket.BucketName
    }
});
```

當 AWS CloudFormation 模板最終合成時，令牌被渲染為 AWS CloudFormation 內{ "Ref": "MyBucket" }在。在部署階段， AWS CloudFormation 會以建立的值區的實際名稱取代此內建項目。

主題

- 令牌和令牌編碼
- 字符串編碼令牌
- 列表編碼令牌
- 数字编码令牌
- 懶惰值
- 轉換為 JSON

令牌和令牌編碼

令牌是實現 IreSolable 接口的對象，其中包含一個單resolve一的方法。在合成過程中 AWS CDK 調用此方法以產生 AWS CloudFormation 模板的最終值。令牌參與合成過程以產生任何類型的任意值。

> **Note**
>
> 您很少會直接使用IResolvable介面。您很可能只會看到字符串編碼版本的令牌。

其他函數通常只接受基本型別的引數，例如string或number。要在這些情況下使用令牌，您可以通過使用 CDK.Token 類上的靜態方法將它們編碼為三種類型之一。

- Token.asString生成一個字符串編碼（或調.toString()用令牌對象）
- Token.asList以產生清單編碼
- Token.asNumber以產生數值編碼

這些需要一個任意值，它可以是一個IResolvable，並將它們編碼為指示類型的原始值。

> **Important**
>
> 由於先前的任何一種類型都可能是編碼的 Token，因此在剖析或嘗試讀取其內容時請小心。例如，如果您嘗試剖析字串以從中擷取值，而該字串是編碼的 Token，則剖析會失敗。同樣地，如果您嘗試查詢陣列的長度或使用數字執行數學運算，您必須先驗證它們是否未編碼 Token。

若要檢查值中是否有未解析的權杖，請呼叫 Token.isUnresolved (Python:is_unresolved) 方法。

下列範例會驗證字串值 (可能是 Token) 的長度不超過 10 個字元。

TypeScript

```
if (!Token.isUnresolved(name) && name.length > 10) {
  throw new Error(`Maximum length for name is 10 characters`);
}
```

JavaScript

```
if ( !Token.isUnresolved(name) && name.length > 10) {
  throw ( new Error(`Maximum length for name is 10 characters`));
}
```

Python

```
if not Token.is_unresolved(name) and len(name) > 10:
    raise ValueError("Maximum length for name is 10 characters")
```

Java

```
if (!Token.isUnresolved(name) && name.length() > 10)
    throw new IllegalArgumentException("Maximum length for name is 10 characters");
```

C#

```
if (!Token.IsUnresolved(name) && name.Length > 10)
```

```
    throw new ArgumentException("Maximum length for name is 10 characters");
```

如果 name 是 Token，則不會執行驗證，並且在生命週期的稍後階段（例如部署期間）仍可能發生錯誤。

> **Note**
>
> 您可以使用令牌編碼來轉義類型系統。例如，您可以對在合成時產生數字值的令牌進行字符串編碼。如果您使用這些函數，您有責任確保您的模板在合成後解析為可用狀態。

字符串編碼令牌

字符串編碼令牌看起來如下所示。

```
${TOKEN[Bucket.Name.1234]}
```

它們可以像常規字符串一樣傳遞，並且可以連接起來，如以下示例所示。

TypeScript

```
const functionName = bucket.bucketName + 'Function';
```

JavaScript

```
const functionName = bucket.bucketName + 'Function';
```

Python

```
function_name = bucket.bucket_name + "Function"
```

Java

```
String functionName = bucket.getBucketName().concat("Function");
```

C#

```
string functionName = bucket.BucketName + "Function";
```

如果您的語言支援，您也可以使用字串內插補點，如下列範例所示。

TypeScript

```
const functionName = `${bucket.bucketName}Function`;
```

JavaScript

```
const functionName = `${bucket.bucketName}Function`;
```

Python

```
function_name = f"{bucket.bucket_name}Function"
```

Java

```
String functionName = String.format("%sFunction". bucket.getBucketName());
```

C#

```
string functionName = $"${bucket.bucketName}Function";
```

避免以其他方式操作字符串。例如，取得字串的子字串很可能會中斷字串 Token。

列表編碼令牌

列表編碼的令牌如下所示：

```
["#{TOKEN[Stack.NotificationArns.1234]}"]
```

與這些列表有關的唯一安全的事情就是將它們直接傳遞給其他構造。字符串列表形式的令牌不能連接，也不能從令牌中獲取元素。操作它們的唯一安全方法是使用 Fn.select 等 AWS CloudFormation 內在函數。

数字编码令牌

數字編碼的令牌是一組微小的負浮點數，如下所示。

```
-1.8881545897087626e+289
```

與列表令牌一樣，您無法修改數字值，因為這樣做可能會破壞數字令牌。唯一允許的操作是將值傳遞給另一個構造。

懶惰值

除了表示部署時間值 (例如 AWS CloudFormation 參數) 之外，Token 也常用來表示合成時間延遲值。這些是最終值將在合成完成之前確定的值，但不是在構建該值的時間點。使用令牌將文字字符串或數字值傳遞給另一個構造，而合成時的實際值可能取決於尚未發生的某些計算。

您可以使用Lazy類別上的靜態方法 (例如 lazy.String 和 lazy .Number) 來建構代表合成時間延遲值的記號。這些方法接受一個對象，其produce屬性是接受上下文參數的函數，並在調用時返回最終值。

下列範例會建立 Auto Scaling 群組，其容量是在建立之後決定的。

TypeScript

```
let actualValue: number;

new AutoScalingGroup(this, 'Group', {
  desiredCapacity: Lazy.numberValue({
    produce(context) {
      return actualValue;
    }
  })
});

// At some later point
actualValue = 10;
```

JavaScript

```
let actualValue;

new AutoScalingGroup(this, 'Group', {
  desiredCapacity: Lazy.numberValue({
    produce(context) {
      return (actualValue);
    }
  })
```

```
});

// At some later point
actualValue = 10;
```

Python

```
class Producer:
    def __init__(self, func):
        self.produce = func

actual_value = None

AutoScalingGroup(self, "Group",
    desired_capacity=Lazy.number_value(Producer(lambda context: actual_value))
)

# At some later point
actual_value = 10
```

Java

```
double actualValue = 0;

class ProduceActualValue implements INumberProducer {

    @Override
    public Number produce(IResolveContext context) {
        return actualValue;
    }
}

AutoScalingGroup.Builder.create(this, "Group")
    .desiredCapacity(Lazy.numberValue(new ProduceActualValue())).build();

// At some later point
actualValue = 10;
```

C#

```
public class NumberProducer : INumberProducer
{
```

```
    Func<Double> function;

    public NumberProducer(Func<Double> function)
    {
        this.function = function;
    }

    public Double Produce(IResolveContext context)
    {
        return function();
    }
}

double actualValue = 0;

new AutoScalingGroup(this, "Group", new AutoScalingGroupProps
{
    DesiredCapacity = Lazy.NumberValue(new NumberProducer(() => actualValue))
});

// At some later point
actualValue = 10;
```

轉換為 JSON

有時您想要生成任意數據的 JSON 字符串，並且您可能不知道數據是否包含令牌。要正確 JSON 編碼任何數據結構，無論它是否包含令牌，請使用該方法堆棧。 toJsonString，如下列範例所示。

TypeScript

```
const stack = Stack.of(this);
const str = stack.toJsonString({
  value: bucket.bucketName
});
```

JavaScript

```
const stack = Stack.of(this);
const str = stack.toJsonString({
  value: bucket.bucketName
});
```

Python

```
stack = Stack.of(self)
string = stack.to_json_string(dict(value=bucket.bucket_name))
```

Java

```
Stack stack = Stack.of(this);
String stringVal = stack.toJsonString(java.util.Map.of(    // Map.of requires Java
 9+
        put("value", bucket.getBucketName())));
```

C#

```
var stack = Stack.Of(this);
var stringVal = stack.ToJsonString(new Dictionary<string, string>
{
    ["value"] = bucket.BucketName
});
```

參數

參數是在部署時提供的自訂值。參數是的一個特徵 AWS CloudFormation。由於 AWS Cloud Development Kit (AWS CDK) 合成了 AWS CloudFormation 模板，因此它還提供了部署時間參數的支持。

主題

- 關於參數
- 定義參數
- 使用參數
- 使用參數部署

關於參數

使用 AWS CDK，您可以定義參數，然後可以在您建立的建構的屬性中使用這些參數。您也可以部署包含參數的堆疊。

使用 AWS CDK Toolkit 部署 AWS CloudFormation 範本時，您可以在命令列上提供參數值。如果您透過 AWS CloudFormation 主控台部署範本，系統會提示您輸入參數值。

一般而言，我們建議您不要將 AWS CloudFormation 參數與 AWS CDK. 將值傳遞到 AWS CDK 應用程序的常用方法是上下文值和環境變量。由於它們在合成時不可用，因此參數值無法輕鬆用於 CDK 應用程序中的流量控制和其他目的。

> **Note**
>
> 要使用參數進行控制流程，您可以使用`CfnCondition`構造，儘管與本機語`if`句相比，這很尷尬。

使用參數需要您注意您編寫的代碼在部署時以及在合成時的行為方式。這使得很難理解和推理您的 AWS CDK 應用程序，在許多情況下幾乎沒有好處。

通常，最好讓 CDK 應用程序以明確定義的方式接受必要的信息，並直接使用它在 CDK 應用程序中聲明構造。理想的 AWS CDK—生產 AWS CloudFormation 範本是具體的，在部署時不會指定任何值。

但是，有些使用案例中的 AWS CloudFormation 參數是唯一適合的。例如，如果您有不同的團隊定義和部署基礎結構，則可以使用參數來使產生的範本更為有用。此外，由於 AWS CDK 支援 AWS CloudFormation 參數，您可以 AWS CDK 搭配使用 AWS CloudFormation 範本的 AWS 服務 (例如 Service Catalog) 使用。這些 AWS 服務會使用參數來設定要部署的範本。

定義參數

使用`CfnParameter`類別來定義參數。您至少需要為大多數參數指定類型和描述，儘管兩者在技術上都是可選的。當系統提示使用者在 AWS CloudFormation 主控台中輸入參數值時，會顯示說明。如需有關可用類型的詳細資訊，請參閱類型。

> **Note**
>
> 您可以在任何範圍內定義參數。不過，我們建議您在堆疊層級定義參數，以便在重構程式碼時不會變更其邏輯 ID。

TypeScript

```
const uploadBucketName = new CfnParameter(this, "uploadBucketName", {
  type: "String",
```

```
  description: "The name of the Amazon S3 bucket where uploaded files will be
 stored."});
```

JavaScript

```
const uploadBucketName = new CfnParameter(this, "uploadBucketName", {
  type: "String",
  description: "The name of the Amazon S3 bucket where uploaded files will be
 stored."});
```

Python

```
upload_bucket_name = CfnParameter(self, "uploadBucketName", type="String",
    description="The name of the Amazon S3 bucket where uploaded files will be
 stored.")
```

Java

```
CfnParameter uploadBucketName = CfnParameter.Builder.create(this,
 "uploadBucketName")
        .type("String")
        .description("The name of the Amazon S3 bucket where uploaded files will be
 stored")
        .build();
```

C#

```
var uploadBucketName = new CfnParameter(this, "uploadBucketName", new
 CfnParameterProps
{
    Type = "String",
    Description = "The name of the Amazon S3 bucket where uploaded files will be
 stored"
});
```

使用參數

`CfnParameter`實例通過令牌向您的 AWS CDK 應用程序暴露其價值。像所有令牌一樣，參數的令牌在合成時解析。但是它解析為對 AWS CloudFormation 模板中定義的參數的引用（將在部署時解析），而不是具體值。

您可以擷取 Token 做為Token類別的執行個體，或以字串、字串清單或數字編碼方式擷取。您的選擇取決於您要搭配使用參數之類別或方法所需的值種類。

TypeScript

Property	kind of value
`value`	## class instance
`valueAsList`	The token represented as a string list
`valueAsNumber`	The token represented as a number
`valueAsString`	The token represented as a string

JavaScript

Property	kind of value
`value`	## class instance
`valueAsList`	The token represented as a string list
`valueAsNumber`	The token represented as a number
`valueAsString`	The token represented as a string

Python

Property	kind of value
`value`	## class instance
`###`	The token represented as a string list
`## _ #`	The token represented as a number
`#####`	The token represented as a string

Java

Property	kind of value
#####	## class instance
getValueAs####	The token represented as a string list
getValueAs## ()	The token represented as a number
getValueAs#####	The token represented as a string

C#

Property	kind of value
Value	## class instance
ValueAsList	The token represented as a string list
ValueAsNumber	The token represented as a number
ValueAsString	The token represented as a string

例如，若要在Bucket定義中使用參數：

TypeScript

```
const bucket = new Bucket(this, "myBucket",
  { bucketName: uploadBucketName.valueAsString});
```

JavaScript

```
const bucket = new Bucket(this, "myBucket",
  { bucketName: uploadBucketName.valueAsString});
```

Python

```
bucket = Bucket(self, "myBucket",
```

```
    bucket_name=upload_bucket_name.value_as_string)
```

Java

```
Bucket bucket = Bucket.Builder.create(this, "myBucket")
        .bucketName(uploadBucketName.getValueAsString())
        .build();
```

C#

```
var bucket = new Bucket(this, "myBucket")
{
    BucketName = uploadBucketName.ValueAsString
};
```

使用參數部署

包含參數的生成模板可以通過 AWS CloudFormation 控制台以通常的方式部署。系統會提示您輸入每個參數的值。

T AWS CDK oolkit (cdk命令列工具) 也支援在部署時指定參數。您可以在--parameters標誌後面的命令行上提供這些內容。您可以部署使用uploadBucketName參數的堆疊，如下列範例所示。

```
cdk deploy MyStack --parameters uploadBucketName=uploadbucket
```

若要定義多個參數，請使用多個--parameters旗標。

```
cdk deploy MyStack --parameters uploadBucketName=upbucket --parameters
 downloadBucketName=downbucket
```

如果您要部署多個堆疊，則可以為每個堆疊指定不同的參數值。若要這麼做，請在參數名稱前加上堆疊名稱和冒號。

```
cdk deploy MyStack YourStack --parameters MyStack:uploadBucketName=uploadbucket --
parameters YourStack:uploadBucketName=upbucket
```

依預設，會 AWS CDK 保留先前部署的參數值，並在後續部署中使用它們 (如果未明確指定)。使用--no-previous-parameters旗標可要求指定所有參數。

標記

標籤是您可以新增至應用程式中建構的 AWS CDK 資訊索引鍵值元素。套用至指定建構的標籤也會套用至其所有可加標籤的子系。標籤包含在從您的應用程序合成的 AWS CloudFormation 模板中，並應用於其部署的 AWS 資源。出於以下目的，您可以使用標籤來識別和分類資源：

- 簡化管理
- 成本分配
- 存取控制
- 您設計的任何其他目的

> **Tip**
>
> 有關如何在資源中使用標籤的詳細 AWS 資訊，請參閱AWS 白皮書中的標記 AWS 資源的最佳做法。

主題

使用標籤

此`Tags`類別包含靜態方法`of()`，您可以透過該方法將標籤新增至指定的建構，或從中移除標籤。

- `Tags.of(SCOPE).add()`將新標籤套用至指定的建構及其所有子項。
- `Tags.of(SCOPE).remove()`從給定的建構及其任何子系中移除標籤，包括子建構可能套用至其本身的標籤。

> **Note**
>
> 標記是使用來實現的the section called “面向”。方面是一種將操作（例如標記）應用於給定範圍內的所有構造的方法。

下列範例會將含值值的標籤鍵套用至建構。

TypeScript

```
Tags.of(myConstruct).add('key', 'value');
```

JavaScript

```
Tags.of(myConstruct).add('key', 'value');
```

Python

```
Tags.of(my_construct).add("key", "value")
```

Java

```
Tags.of(myConstruct).add("key", "value");
```

C#

```
Tags.Of(myConstruct).Add("key", "value");
```

下列範例會從建構中刪除標籤索引鍵。

TypeScript

```
Tags.of(myConstruct).remove('key');
```

JavaScript

```
Tags.of(myConstruct).remove('key');
```

Python

```
Tags.of(my_construct).remove("key")
```

Java

```
Tags.of(myConstruct).remove("key");
```

C#

```
Tags.Of(myConstruct).Remove("key");
```

如果您使用的是Stage建構，請在Stage樓層或更低層級套用標籤。標籤不會跨越Stage邊界套用。

標記優先權

遞迴 AWS CDK 套用和移除標籤。如果發生衝突，具有最高優先順序的標籤操作將獲勝。(使用選用priority性質設定優先順序。) 如果兩個操作的優先順序相同，則最接近建構樹底部的標籤操作將獲勝。根據預設，套用標籤的優先順序為 100 (直接新增至 AWS CloudFormation 資源的標籤除外，其優先順序為 50)。移除標籤的預設優先順序為 200。

以下內容將優先順序為 300 的標籤套用至建構。

TypeScript

```
Tags.of(myConstruct).add('key', 'value', {
  priority: 300
});
```

JavaScript

```
Tags.of(myConstruct).add('key', 'value', {
  priority: 300
});
```

Python

```
Tags.of(my_construct).add("key", "value", priority=300)
```

Java

```
Tags.of(myConstruct).add("key", "value", TagProps.builder()
        .priority(300).build());
```

C#

```
Tags.Of(myConstruct).Add("key", "value", new TagProps { Priority = 300 });
```

可選屬性

標籤支援properties微調標籤套用至資源或從資源中移除的方式。所有屬性皆是選用。

`applyToLaunchedInstances`(Python:`apply_to_launched_instances`)

僅適用於加入 ()。依預設，標籤會套用至在「自動調整比例」群組中啟動的執行個體。將此屬性設定為 false 可忽略在「Auto Scaling」群組中啟動的執行個體。

`includeResourceTypes`/`excludeResourceTypes`(Python:`include_resource_types`/`exclude_res`

使用這些標籤僅可根據資源類型在資源子集上 AWS CloudFormation 操作標籤。依預設，此作業會套用至建構子樹狀結構中的所有資源，但是可以透過包含或排除某些資源類型來變更此作業。如果同時指定兩者，排除的優先順序會高於包含。

`priority`

使用此選項可設定此作業相對於其他`Tags.add()`和作`Tags.remove()`業的優先順序。較高的值優先於較低的值。新增作業的預設值為 100 (直接套用至 AWS CloudFormation 資源的標籤為 50)，移除作業的預設值為 200。

下列範例會將值和優先順序為 100 的標籤 tagname 套用至建構AWS::Xxx::Yyy中類型的資源。它不會將標籤套用至在 Amazon EC2 Auto Scaling 群組中啟動的執行個體，或套用至類型的資源AWS::Xxx::Zzz。（這些是兩種任意但不同 AWS CloudFormation 資源類型的佔位符。）

TypeScript

```
Tags.of(myConstruct).add('tagname', 'value', {
  applyToLaunchedInstances: false,
  includeResourceTypes: ['AWS::Xxx::Yyy'],
  excludeResourceTypes: ['AWS::Xxx::Zzz'],
```

```
  priority: 100,
});
```

JavaScript

```
Tags.of(myConstruct).add('tagname', 'value', {
  applyToLaunchedInstances: false,
  includeResourceTypes: ['AWS::Xxx::Yyy'],
  excludeResourceTypes: ['AWS::Xxx::Zzz'],
  priority: 100
});
```

Python

```
Tags.of(my_construct).add("tagname", "value",
    apply_to_launched_instances=False,
    include_resource_types=["AWS::Xxx::Yyy"],
    exclude_resource_types=["AWS::Xxx::Zzz"],
    priority=100)
```

Java

```
Tags.of(myConstruct).add("key", "value", TagProps.builder()
                .applyToLaunchedInstances(false)
                .includeResourceTypes(Arrays.asList("AWS::Xxx::Yyy"))
                .excludeResourceTypes(Arrays.asList("AWS::Xxx::Zzz"))
                .priority(100).build());
```

C#

```
Tags.Of(myConstruct).Add("tagname", "value", new TagProps
{
    ApplyToLaunchedInstances = false,
    IncludeResourceTypes = ["AWS::Xxx::Yyy"],
    ExcludeResourceTypes = ["AWS::Xxx::Zzz"],
    Priority = 100
});
```

下列範例會從建構AWS::Xxx::Yyy中類型的資源中移除優先順序為 200 的標籤 tagname，但不會從類型AWS::Xxx::Zzz的資源中移除。

TypeScript

```
Tags.of(myConstruct).remove('tagname', {
  includeResourceTypes: ['AWS::Xxx::Yyy'],
  excludeResourceTypes: ['AWS::Xxx::Zzz'],
  priority: 200,
});
```

JavaScript

```
Tags.of(myConstruct).remove('tagname', {
  includeResourceTypes: ['AWS::Xxx::Yyy'],
  excludeResourceTypes: ['AWS::Xxx::Zzz'],
  priority: 200
});
```

Python

```
Tags.of(my_construct).remove("tagname",
    include_resource_types=["AWS::Xxx::Yyy"],
    exclude_resource_types=["AWS::Xxx::Zzz"],
    priority=200,)
```

Java

```
Tags.of((myConstruct).remove("tagname", TagProps.builder()
        .includeResourceTypes(Arrays.asList("AWS::Xxx::Yyy"))
        .excludeResourceTypes(Arrays.asList("AWS::Xxx::Zzz"))
        .priority(100).build());
```

C#

```
Tags.Of(myConstruct).Remove("tagname", new TagProps
{
    IncludeResourceTypes = ["AWS::Xxx::Yyy"],
    ExcludeResourceTypes = ["AWS::Xxx::Zzz"],
    Priority = 100
});
```

範例

下列範例會將具StackType有值的標籤索引鍵新增TheBest至在Stack具名內建立的任何資源MarketingSystem。然後它會從除 Amazon EC2 VPC 子網路以外的所有資源中再次移除它。結果是只有子網路套用了標籤。

TypeScript

```
import { App, Stack, Tags } from 'aws-cdk-lib';

const app = new App();
const theBestStack = new Stack(app, 'MarketingSystem');

// Add a tag to all constructs in the stack
Tags.of(theBestStack).add('StackType', 'TheBest');

// Remove the tag from all resources except subnet resources
Tags.of(theBestStack).remove('StackType', {
  excludeResourceTypes: ['AWS::EC2::Subnet']
});
```

JavaScript

```
const { App, Stack, Tags } = require('aws-cdk-lib');

const app = new App();
const theBestStack = new Stack(app, 'MarketingSystem');

// Add a tag to all constructs in the stack
Tags.of(theBestStack).add('StackType', 'TheBest');

// Remove the tag from all resources except subnet resources
Tags.of(theBestStack).remove('StackType', {
  excludeResourceTypes: ['AWS::EC2::Subnet']
});
```

Python

```
from aws_cdk import App, Stack, Tags

app = App();
the_best_stack = Stack(app, 'MarketingSystem')
```

```
# Add a tag to all constructs in the stack
Tags.of(the_best_stack).add("StackType", "TheBest")

# Remove the tag from all resources except subnet resources
Tags.of(the_best_stack).remove("StackType",
    exclude_resource_types=["AWS::EC2::Subnet"])
```

Java

```
import software.amazon.awscdk.App;
import software.amazon.awscdk.Tags;

// Add a tag to all constructs in the stack
Tags.of(theBestStack).add("StackType", "TheBest");

// Remove the tag from all resources except subnet resources
Tags.of(theBestStack).remove("StackType", TagProps.builder()
        .excludeResourceTypes(Arrays.asList("AWS::EC2::Subnet"))
        .build());
```

C#

```
using Amazon.CDK;

var app = new App();
var theBestStack = new Stack(app, 'MarketingSystem');

// Add a tag to all constructs in the stack
Tags.Of(theBestStack).Add("StackType", "TheBest");

// Remove the tag from all resources except subnet resources
Tags.Of(theBestStack).Remove("StackType", new TagProps
{
    ExcludeResourceTypes = ["AWS::EC2::Subnet"]
});
```

下面的代碼實現了相同的結果。考慮哪種方法（包含或排除）使您的意圖更清晰。

TypeScript

```
Tags.of(theBestStack).add('StackType', 'TheBest',
```

```
  { includeResourceTypes: ['AWS::EC2::Subnet']});
```

JavaScript

```
Tags.of(theBestStack).add('StackType', 'TheBest',
  { includeResourceTypes: ['AWS::EC2::Subnet']});
```

Python

```
Tags.of(the_best_stack).add("StackType", "TheBest",
    include_resource_types=["AWS::EC2::Subnet"])
```

Java

```
Tags.of(theBestStack).add("StackType", "TheBest", TagProps.builder()
        .includeResourceTypes(Arrays.asList("AWS::EC2::Subnet"))
        .build());
```

C#

```
Tags.Of(theBestStack).Add("StackType", "TheBest", new TagProps {
    IncludeResourceTypes = ["AWS::EC2::Subnet"]
});
```

為單一建構加標籤

`Tags.of(scope).add(key, value)`是將標籤加入至中建構的標準方式 AWS CDK。它的樹步行行為，遞歸地標記給定範圍內的所有可標記資源，幾乎總是你想要的。但是，有時您需要標記特定的任意構造（或構造）。

一種這種情況涉及應用標籤，其值是從被標記的構造的某些屬性派生的標籤。標準標記方法遞歸地將相同的鍵和值應用於範圍中的所有匹配資源。但是，在這裡，每個標記構造的值可能不同。

標籤是使用方面來實現的，CDK 會在您使用指定的範圍內為每個構造調用`Tags.of(scope)`標籤的`visit()`方法。我們可以`Tag.visit()`直接調用以將標籤應用於單個構造。

TypeScript

```
new cdk.Tag(key, value).visit(scope);
```

JavaScript

```
new cdk.Tag(key, value).visit(scope);
```

Python

```
cdk.Tag(key, value).visit(scope)
```

Java

```
Tag.Builder.create(key, value).build().visit(scope);
```

C#

```
new Tag(key, value).Visit(scope);
```

您可以標記範圍下的所有建構，但是讓標籤的值衍生自每個建構的屬性。要做到這一點，寫一個方面，並在方面的`visit()`方法中應用標籤，如前面的例子所示。然後，使用將縱橫比添加到所需的範圍`Aspects.of(scope).add(aspect)`。

下列範例會將標籤套用至包含資源路徑之堆疊中的每個資源。

TypeScript

```
class PathTagger implements cdk.IAspect {
  visit(node: IConstruct) {
    new cdk.Tag("aws-cdk-path", node.node.path).visit(node);
  }
}

stack = new MyStack(app);
cdk.Aspects.of(stack).add(new PathTagger())
```

JavaScript

```
class PathTagger {
  visit(node) {
    new cdk.Tag("aws-cdk-path", node.node.path).visit(node);
```

```
  }
}

stack = new MyStack(app);
cdk.Aspects.of(stack).add(new PathTagger())
```

Python

```
@jsii.implements(cdk.IAspect)
class PathTagger:
    def visit(self, node: IConstruct):
        cdk.Tag("aws-cdk-path", node.node.path).visit(node)

stack = MyStack(app)
cdk.Aspects.of(stack).add(PathTagger())
```

Java

```
final class PathTagger implements IAspect {
 public void visit(IConstruct node) {
  Tag.Builder.create("aws-cdk-path", node.getNode().getPath()).build().visit(node);
 }
}

stack stack = new MyStack(app);
Aspects.of(stack).add(new PathTagger());
```

C#

```
public class PathTagger : IAspect
{
    public void Visit(IConstruct node)
    {
        new Tag("aws-cdk-path", node.Node.Path).Visit(node);
    }
}

var stack = new MyStack(app);
Aspects.Of(stack).Add(new PathTagger);
```

> **Tip**
>
> 條件式標記的邏輯 (包括優先順序、資源類型Tag等) 會內建於類別中。將標籤套用至任意資源時，您可以使用這些功能；如果條件不符合，則不會套用標籤。此外，該Tag類僅標記可標記資源，因此在應用標籤之前不需要測試構造是否可標記。

資產

資產是可以捆綁到 AWS CDK 庫和應用程序中的本地文件，目錄或 Docker 映像。例如，資產可能是包含 AWS Lambda 函數處理常式程式碼的目錄。資產可以代表應用程序需要操作的任何成品。

下面的教程視頻提供了 CDK 資產的全面概述，並解釋了如何在你的 insfrastructure 作為代碼（ IaC ）使用它們。

CDK 資產說明

您可以透過特定 AWS 建構公開的 API 來新增資產。例如，當您定義 Lambda.Function 建構時，程式碼屬性可讓您傳遞資產 (目錄)。 `Function`使用 assets 來捆綁目錄的內容，並將其用於函數的代碼。同樣，ECS。 ContainerImage.fromasSet 會在定義 Amazon ECS 任務定義時，使用從本機目錄建立的泊塢視窗映像。

詳細資產

當您參考應用程式中的資產時，從應用程式合成的雲端組件會包含中繼資料資訊以及 AWS CDK CLI 指示。這些說明包括在本機磁碟上尋找資產的位置，以及要根據資產類型執行的捆綁類型，例如要壓縮的目錄 (zip) 或要建置的 Docker 映像。

產 AWS CDK 生資產的來源雜湊。這可以在構建時使用，以確定資產的內容是否已更改。

依預設，會在雲端組件目錄中 AWS CDK 建立資產的複本，預設為`cdk.out`，在來源雜湊下。這樣，雲組件是獨立的，因此，如果它移動到不同的主機進行部署，它仍然可以部署。如需詳細資訊，請參閱 the section called “雲端組件”。

當部 AWS CDK 署參考資產的應用程式時 (直接透過應用程式程式碼或透過程式庫)， AWS CDK CLI 會先準備資產並將其發佈到 Amazon S3 儲存貯體或 Amazon ECR 儲存庫。S3 儲存貯體或儲存庫是在啟動載入期間建立的。) 只有這樣才會部署在堆疊中定義的資源。

本節介紹框架中可用的低級 API。

資產類型

AWS CDK 支援下列類型的資產：

Amazon S3 資產

這些是 AWS CDK 上傳到 Amazon S3 的本機檔案和目錄。

Docker 影像

這些是 AWS CDK 上傳到 Amazon ECR 的碼頭圖像。

這些資產類型將在下列各節中說明。

Amazon S3 資產

您可以將本機檔案和目錄定義為資產，並透過 aws- S3 資產模組將其 AWS CDK 封裝和上傳到 Amazon S3。

下列範例會定義本機目錄資產和檔案資產。

TypeScript

```
import { Asset } from 'aws-cdk-lib/aws-s3-assets';

// Archived and uploaded to Amazon S3 as a .zip file
const directoryAsset = new Asset(this, "SampleZippedDirAsset", {
  path: path.join(__dirname, "sample-asset-directory")
});

// Uploaded to Amazon S3 as-is
const fileAsset = new Asset(this, 'SampleSingleFileAsset', {
  path: path.join(__dirname, 'file-asset.txt')
});
```

JavaScript

```
const { Asset } = require('aws-cdk-lib/aws-s3-assets');

// Archived and uploaded to Amazon S3 as a .zip file
const directoryAsset = new Asset(this, "SampleZippedDirAsset", {
```

```
  path: path.join(__dirname, "sample-asset-directory")
});

// Uploaded to Amazon S3 as-is
const fileAsset = new Asset(this, 'SampleSingleFileAsset', {
  path: path.join(__dirname, 'file-asset.txt')
});
```

Python

```
import os.path
dirname = os.path.dirname(__file__)

from aws_cdk.aws_s3_assets import Asset

# Archived and uploaded to Amazon S3 as a .zip file
directory_asset = Asset(self, "SampleZippedDirAsset",
  path=os.path.join(dirname, "sample-asset-directory")
)

# Uploaded to Amazon S3 as-is
file_asset = Asset(self, 'SampleSingleFileAsset',
  path=os.path.join(dirname, 'file-asset.txt')
)
```

Java

```
import java.io.File;

import software.amazon.awscdk.services.s3.assets.Asset;

// Directory where app was started
File startDir = new File(System.getProperty("user.dir"));

// Archived and uploaded to Amazon S3 as a .zip file
Asset directoryAsset = Asset.Builder.create(this, "SampleZippedDirAsset")
                .path(new File(startDir, "sample-asset-
directory").toString()).build();

// Uploaded to Amazon S3 as-is
Asset fileAsset = Asset.Builder.create(this, "SampleSingleFileAsset")
                .path(new File(startDir, "file-asset.txt").toString()).build();
```

C#

```
using System.IO;
using Amazon.CDK.AWS.S3.Assets;

// Archived and uploaded to Amazon S3 as a .zip file
var directoryAsset = new Asset(this, "SampleZippedDirAsset", new AssetProps
{
    Path = Path.Combine(Directory.GetCurrentDirectory(), "sample-asset-directory")
});

// Uploaded to Amazon S3 as-is
var fileAsset = new Asset(this, "SampleSingleFileAsset", new AssetProps
{
    Path = Path.Combine(Directory.GetCurrentDirectory(), "file-asset.txt")
});
```

Go

```
dirName, err := os.Getwd()
if err != nil {
  panic(err)
}

awss3assets.NewAsset(stack, jsii.String("SampleZippedDirAsset"),
 &awss3assets.AssetProps{
  Path: jsii.String(path.Join(dirName, "sample-asset-directory")),
})

awss3assets.NewAsset(stack, jsii.String("SampleSingleFileAsset"),
 &awss3assets.AssetProps{
  Path: jsii.String(path.Join(dirName, "file-asset.txt")),
})
```

在大多數情況下，您不需要直接在`aws-s3-assets`模塊中使用 API。支援資產的模組 (例如) 具有便利的方法`aws-lambda`，以便您可以使用資產。對於 Lambda 函數，fromasSet () 靜態方法可讓您在本機檔案系統中指定目錄或 .zip 檔案。

Lambda 函數示例

常見的使用案例是使用處理常式程式碼建立 Lambda 函數作為 Amazon S3 資產。

下列範例使用 Amazon S3 資產在本機目錄中定義 Python 處理常式`handler`。它也會建立以本機目錄資產做為`code`屬性的 Lambda 函數。以下是處理程序的 Python 代碼。

```
def lambda_handler(event, context):
  message = 'Hello World!'
  return {
    'message': message
  }
```

主要應用程式的 AWS CDK 程式碼應如下所示。

TypeScript

```
import * as cdk from 'aws-cdk-lib';
import { Constructs } from 'constructs';
import * as lambda from 'aws-cdk-lib/aws-lambda';
import * as path from 'path';

export class HelloAssetStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    new lambda.Function(this, 'myLambdaFunction', {
      code: lambda.Code.fromAsset(path.join(__dirname, 'handler')),
      runtime: lambda.Runtime.PYTHON_3_6,
      handler: 'index.lambda_handler'
    });
  }
}
```

JavaScript

```
const cdk = require('aws-cdk-lib');
const lambda = require('aws-cdk-lib/aws-lambda');
const path = require('path');

class HelloAssetStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    new lambda.Function(this, 'myLambdaFunction', {
      code: lambda.Code.fromAsset(path.join(__dirname, 'handler')),
      runtime: lambda.Runtime.PYTHON_3_6,
```

```
      handler: 'index.lambda_handler'
    });
  }
}

module.exports = { HelloAssetStack }
```

Python

```
from aws_cdk import Stack
from constructs import Construct
from aws_cdk import aws_lambda as lambda_

import os.path
dirname = os.path.dirname(__file__)

class HelloAssetStack(Stack):
    def __init__(self, scope: Construct, id: str, **kwargs):
        super().__init__(scope, id, **kwargs)

        lambda_.Function(self, 'myLambdaFunction',
            code=lambda_.Code.from_asset(os.path.join(dirname, 'handler')),
            runtime=lambda_.Runtime.PYTHON_3_6,
            handler="index.lambda_handler")
```

Java

```
import java.io.File;

import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.lambda.Runtime;

public class HelloAssetStack extends Stack {

    public HelloAssetStack(final App scope, final String id) {
        this(scope, id, null);
    }

    public HelloAssetStack(final App scope, final String id, final StackProps props)
 {
        super(scope, id, props);
```

```
        File startDir = new File(System.getProperty("user.dir"));

        Function.Builder.create(this, "myLambdaFunction")
                .code(Code.fromAsset(new File(startDir, "handler").toString()))
                .runtime(Runtime.PYTHON_3_6)
                .handler("index.lambda_handler").build();
    }
}
```

C#

```
using Amazon.CDK;
using Amazon.CDK.AWS.Lambda;
using System.IO;

public class HelloAssetStack : Stack
{
    public HelloAssetStack(Construct scope, string id, StackProps props) :
 base(scope, id, props)
    {
        new Function(this, "myLambdaFunction", new FunctionProps
        {
            Code = Code.FromAsset(Path.Combine(Directory.GetCurrentDirectory(),
 "handler")),
            Runtime = Runtime.PYTHON_3_6,
            Handler = "index.lambda_handler"
        });
    }
}
```

Go

```
import (
  "os"
  "path"

  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/aws-cdk-go/awscdk/v2/awslambda"
  "github.com/aws/aws-cdk-go/awscdk/v2/awss3assets"
  "github.com/aws/constructs-go/constructs/v10"
  "github.com/aws/jsii-runtime-go"
)
```

```
func HelloAssetStack(scope constructs.Construct, id string, props
 *HelloAssetStackProps) awscdk.Stack {
  var sprops awscdk.StackProps
  if props != nil {
    sprops = props.StackProps
  }
  stack := awscdk.NewStack(scope, &id, &sprops)

  dirName, err := os.Getwd()
  if err != nil {
    panic(err)
  }

  awslambda.NewFunction(stack, jsii.String("myLambdaFunction"),
 &awslambda.FunctionProps{
    Code: awslambda.AssetCode_FromAsset(jsii.String(path.Join(dirName, "handler")),
 &awss3assets.AssetOptions{}),
    Runtime: awslambda.Runtime_PYTHON_3_6(),
    Handler: jsii.String("index.lambda_handler"),
  })

  return stack
}
```

該Function方法使用 assets 來捆綁目錄的內容，並將其用於函數的代碼。

> **Tip**
>
> Java .jar 文件是具有不同擴展名的 ZIP 文件。這些檔案會按原樣上傳至 Amazon S3，但是當部署為 Lambda 函數時，會擷取它們包含的檔案，而您可能不想要這些檔案。若要避免這種情況，請將.jar檔案放在目錄中，並將該目錄指定為資產。

部署時間屬性範例

Amazon S3 資產類型也會公開 AWS CDK 程式庫和應用程式中可參考的部署時間屬性。 AWS CDK CLI 指令cdk synth會將資產屬性顯示為 AWS CloudFormation 參數。

下列範例會使用部署時間屬性，將影像資產的位置作為環境變數傳遞至 Lambda 函數。（文件的種類並不重要; 這裡使用的 PNG 圖像只是一個例子。）

TypeScript

```
import { Asset } from 'aws-cdk-lib/aws-s3-assets';
import * as path from 'path';

const imageAsset = new Asset(this, "SampleAsset", {
  path: path.join(__dirname, "images/my-image.png")
});

new lambda.Function(this, "myLambdaFunction", {
  code: lambda.Code.asset(path.join(__dirname, "handler")),
  runtime: lambda.Runtime.PYTHON_3_6,
  handler: "index.lambda_handler",
  environment: {
    'S3_BUCKET_NAME': imageAsset.s3BucketName,
    'S3_OBJECT_KEY': imageAsset.s3ObjectKey,
    'S3_OBJECT_URL': imageAsset.s3ObjectUrl
  }
});
```

JavaScript

```
const { Asset } = require('aws-cdk-lib/aws-s3-assets');
const path = require('path');

const imageAsset = new Asset(this, "SampleAsset", {
  path: path.join(__dirname, "images/my-image.png")
});

new lambda.Function(this, "myLambdaFunction", {
  code: lambda.Code.asset(path.join(__dirname, "handler")),
  runtime: lambda.Runtime.PYTHON_3_6,
  handler: "index.lambda_handler",
  environment: {
    'S3_BUCKET_NAME': imageAsset.s3BucketName,
    'S3_OBJECT_KEY': imageAsset.s3ObjectKey,
    'S3_OBJECT_URL': imageAsset.s3ObjectUrl
  }
});
```

Python

```
import os.path
```

```
import aws_cdk.aws_lambda as lambda_
from aws_cdk.aws_s3_assets import Asset

dirname = os.path.dirname(__file__)

image_asset = Asset(self, "SampleAsset",
    path=os.path.join(dirname, "images/my-image.png"))

lambda_.Function(self, "myLambdaFunction",
    code=lambda_.Code.asset(os.path.join(dirname, "handler")),
    runtime=lambda_.Runtime.PYTHON_3_6,
    handler="index.lambda_handler",
    environment=dict(
        S3_BUCKET_NAME=image_asset.s3_bucket_name,
        S3_OBJECT_KEY=image_asset.s3_object_key,
        S3_OBJECT_URL=image_asset.s3_object_url))
```

Java

```
import java.io.File;

import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.lambda.Runtime;
import software.amazon.awscdk.services.s3.assets.Asset;

public class FunctionStack extends Stack {
    public FunctionStack(final App scope, final String id, final StackProps props) {
        super(scope, id, props);

        File startDir = new File(System.getProperty("user.dir"));

        Asset imageAsset = Asset.Builder.create(this, "SampleAsset")
                .path(new File(startDir, "images/my-image.png").toString()).build())

        Function.Builder.create(this, "myLambdaFunction")
                .code(Code.fromAsset(new File(startDir, "handler").toString()))
                .runtime(Runtime.PYTHON_3_6)
                .handler("index.lambda_handler")
                .environment(java.util.Map.of(    // Java 9 or later
                    "S3_BUCKET_NAME", imageAsset.getS3BucketName(),
```

```
                    "S3_OBJECT_KEY", imageAsset.getS3ObjectKey(),
                    "S3_OBJECT_URL", imageAsset.getS3ObjectUrl()))
                .build();
    }
}
```

C#

```
using Amazon.CDK;
using Amazon.CDK.AWS.Lambda;
using Amazon.CDK.AWS.S3.Assets;
using System.IO;
using System.Collections.Generic;

var imageAsset = new Asset(this, "SampleAsset", new AssetProps
{
    Path = Path.Combine(Directory.GetCurrentDirectory(), @"images\my-image.png")
});

new Function(this, "myLambdaFunction", new FunctionProps
{
    Code = Code.FromAsset(Path.Combine(Directory.GetCurrentDirectory(), "handler")),
    Runtime = Runtime.PYTHON_3_6,
    Handler = "index.lambda_handler",
    Environment = new Dictionary<string, string>
    {
        ["S3_BUCKET_NAME"] = imageAsset.S3BucketName,
        ["S3_OBJECT_KEY"] = imageAsset.S3ObjectKey,
        ["S3_OBJECT_URL"] = imageAsset.S3ObjectUrl
    }
});
```

Go

```
import (
  "os"
  "path"

  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/aws-cdk-go/awscdk/v2/awslambda"
  "github.com/aws/aws-cdk-go/awscdk/v2/awss3assets"
)
```

```
dirName, err := os.Getwd()
if err != nil {
  panic(err)
}

imageAsset := awss3assets.NewAsset(stack, jsii.String("SampleAsset"),
 &awss3assets.AssetProps{
  Path: jsii.String(path.Join(dirName, "images/my-image.png")),
})

awslambda.NewFunction(stack, jsii.String("myLambdaFunction"),
 &awslambda.FunctionProps{
  Code: awslambda.AssetCode_FromAsset(jsii.String(path.Join(dirName, "handler"))),
  Runtime: awslambda.Runtime_PYTHON_3_6(),
  Handler: jsii.String("index.lambda_handler"),
  Environment: &map[string]*string{
    "S3_BUCKET_NAME": imageAsset.S3BucketName(),
    "S3_OBJECT_KEY": imageAsset.S3ObjectKey(),
    "S3_URL": imageAsset.S3ObjectUrl(),
  },
})
```

許可

如果您直接透過 aws-S3 資產模組、IAM 角色、使用者或群組使用 Amazon S3 資產，而且您需要在執行階段讀取資產，請透過 As set.grantRead 方法授予這些資產 IAM 許可。

下列範例授與 IAM 群組對檔案資產的讀取權限。

TypeScript

```
import { Asset } from 'aws-cdk-lib/aws-s3-assets';
import * as path from 'path';

const asset = new Asset(this, 'MyFile', {
  path: path.join(__dirname, 'my-image.png')
});

const group = new iam.Group(this, 'MyUserGroup');
asset.grantRead(group);
```

JavaScript

```
const { Asset } = require('aws-cdk-lib/aws-s3-assets');
const path = require('path');

const asset = new Asset(this, 'MyFile', {
  path: path.join(__dirname, 'my-image.png')
});

const group = new iam.Group(this, 'MyUserGroup');
asset.grantRead(group);
```

Python

```
from aws_cdk.aws_s3_assets import Asset
import aws_cdk.aws_iam as iam

import os.path
dirname = os.path.dirname(__file__)

        asset = Asset(self, "MyFile",
            path=os.path.join(dirname, "my-image.png"))

        group = iam.Group(self, "MyUserGroup")
        asset.grant_read(group)
```

Java

```
import java.io.File;

import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.services.iam.Group;
import software.amazon.awscdk.services.s3.assets.Asset;

public class GrantStack extends Stack {
    public GrantStack(final App scope, final String id, final StackProps props) {
        super(scope, id, props);

        File startDir = new File(System.getProperty("user.dir"));

        Asset asset = Asset.Builder.create(this, "SampleAsset")
                .path(new File(startDir, "images/my-image.png").toString()).build();
```

```
        Group group = new Group(this, "MyUserGroup");
        asset.grantRead(group);   }
}
```

C#

```
using Amazon.CDK;
using Amazon.CDK.AWS.IAM;
using Amazon.CDK.AWS.S3.Assets;
using System.IO;

var asset = new Asset(this, "MyFile", new AssetProps {
    Path = Path.Combine(Path.Combine(Directory.GetCurrentDirectory(), @"images\my-
image.png"))
});

var group = new Group(this, "MyUserGroup");
asset.GrantRead(group);
```

Go

```
import (
  "os"
  "path"

  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/aws-cdk-go/awscdk/v2/awsiam"
  "github.com/aws/aws-cdk-go/awscdk/v2/awss3assets"
)

dirName, err := os.Getwd()
if err != nil {
  panic(err)
}

asset := awss3assets.NewAsset(stack, jsii.String("MyFile"), &awss3assets.AssetProps{
  Path: jsii.String(path.Join(dirName, "my-image.png")),
})

group := awsiam.NewGroup(stack, jsii.String("MyUserGroup"), &awsiam.GroupProps{})

asset.GrantRead(group)
```

泊塢視窗影像資產

AWS CDK 支持通過模塊將本地 Docker 圖像捆綁為資產。aws-ecr-assets

下列範例會定義在本機建置並推送至 Amazon ECR 的 Docker 映像檔。映像檔是從本機 Docker 內容目錄 (使用 Docker 檔案) 建立，並透過 AWS CDK CLI 或應用程式的 CI/CD 管道上傳至 Amazon ECR。圖像可以在您的 AWS CDK 應用程序中自然引用。

TypeScript

```
import { DockerImageAsset } from 'aws-cdk-lib/aws-ecr-assets';

const asset = new DockerImageAsset(this, 'MyBuildImage', {
  directory: path.join(__dirname, 'my-image')
});
```

JavaScript

```
const { DockerImageAsset } = require('aws-cdk-lib/aws-ecr-assets');

const asset = new DockerImageAsset(this, 'MyBuildImage', {
  directory: path.join(__dirname, 'my-image')
});
```

Python

```
from aws_cdk.aws_ecr_assets import DockerImageAsset

import os.path
dirname = os.path.dirname(__file__)

asset = DockerImageAsset(self, 'MyBuildImage',
    directory=os.path.join(dirname, 'my-image'))
```

Java

```
import software.amazon.awscdk.services.ecr.assets.DockerImageAsset;

File startDir = new File(System.getProperty("user.dir"));

DockerImageAsset asset = DockerImageAsset.Builder.create(this, "MyBuildImage")
```

```
            .directory(new File(startDir, "my-image").toString()).build();
```

C#

```
using System.IO;
using Amazon.CDK.AWS.ECR.Assets;

var asset = new DockerImageAsset(this, "MyBuildImage", new DockerImageAssetProps
{
    Directory = Path.Combine(Directory.GetCurrentDirectory(), "my-image")
});
```

Go

```
import (
  "os"
  "path"

  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/aws-cdk-go/awscdk/v2/awsecrassets"
)

dirName, err := os.Getwd()
if err != nil {
  panic(err)
}

asset := awsecrassets.NewDockerImageAsset(stack, jsii.String("MyBuildImage"),
 &awsecrassets.DockerImageAssetProps{
  Directory: jsii.String(path.Join(dirName, "my-image")),
})
```

該my-image目錄必須包含一個碼頭文件。 AWS CDK CLI 會從中建立 Docker 映像my-image，將其推送到 Amazon ECR 儲存庫，並將存放庫的名稱指定為堆疊的 AWS CloudFormation 參數。Docker 映像資產類型會公開可在 AWS CDK 程式庫和應用程式中參考的部署時間屬性。 AWS CDK CLI 指令cdk synth會將資產屬性顯示為 AWS CloudFormation 參數。

Amazon ECS 任務定義示例

一個常見的使用案例是創建一個 Amazon ECS TaskDefinition來運行碼頭容器。下列範例會指定在本機 AWS CDK 建置並推送至 Amazon ECR 的 Docker 映像資產的位置。

TypeScript

```
import * as ecs from 'aws-cdk-lib/aws-ecs';
import * as ecr_assets from 'aws-cdk-lib/aws-ecr-assets';
import * as path from 'path';

const taskDefinition = new ecs.FargateTaskDefinition(this, "TaskDef", {
  memoryLimitMiB: 1024,
  cpu: 512
});

const asset = new ecr_assets.DockerImageAsset(this, 'MyBuildImage', {
  directory: path.join(__dirname, 'my-image')
});

taskDefinition.addContainer("my-other-container", {
  image: ecs.ContainerImage.fromDockerImageAsset(asset)
});
```

JavaScript

```
const ecs = require('aws-cdk-lib/aws-ecs');
const ecr_assets = require('aws-cdk-lib/aws-ecr-assets');
const path = require('path');

const taskDefinition = new ecs.FargateTaskDefinition(this, "TaskDef", {
  memoryLimitMiB: 1024,
  cpu: 512
});

const asset = new ecr_assets.DockerImageAsset(this, 'MyBuildImage', {
  directory: path.join(__dirname, 'my-image')
});

taskDefinition.addContainer("my-other-container", {
  image: ecs.ContainerImage.fromDockerImageAsset(asset)
});
```

Python

```
import aws_cdk.aws_ecs as ecs
import aws_cdk.aws_ecr_assets as ecr_assets
```

```
import os.path
dirname = os.path.dirname(__file__)

task_definition = ecs.FargateTaskDefinition(self, "TaskDef",
    memory_limit_mib=1024,
    cpu=512)

asset = ecr_assets.DockerImageAsset(self, 'MyBuildImage',
    directory=os.path.join(dirname, 'my-image'))

task_definition.add_container("my-other-container",
    image=ecs.ContainerImage.from_docker_image_asset(asset))
```

Java

```
import java.io.File;

import software.amazon.awscdk.services.ecs.FargateTaskDefinition;
import software.amazon.awscdk.services.ecs.ContainerDefinitionOptions;
import software.amazon.awscdk.services.ecs.ContainerImage;

import software.amazon.awscdk.services.ecr.assets.DockerImageAsset;

File startDir = new File(System.getProperty("user.dir"));

FargateTaskDefinition taskDefinition = FargateTaskDefinition.Builder.create(
        this, "TaskDef").memoryLimitMiB(1024).cpu(512).build();

DockerImageAsset asset = DockerImageAsset.Builder.create(this, "MyBuildImage")
            .directory(new File(startDir, "my-image").toString()).build();

taskDefinition.addContainer("my-other-container",
        ContainerDefinitionOptions.builder()
            .image(ContainerImage.fromDockerImageAsset(asset))
            .build();
```

C#

```
using System.IO;
using Amazon.CDK.AWS.ECS;
using Amazon.CDK.AWS.Ecr.Assets;
```

```
var taskDefinition = new FargateTaskDefinition(this, "TaskDef", new
 FargateTaskDefinitionProps
{
    MemoryLimitMiB = 1024,
    Cpu = 512
});

var asset = new DockerImageAsset(this, "MyBuildImage", new DockerImageAssetProps
{
    Directory = Path.Combine(Directory.GetCurrentDirectory(), "my-image")
});

taskDefinition.AddContainer("my-other-container", new ContainerDefinitionOptions
{
    Image = ContainerImage.FromDockerImageAsset(asset)
});
```

Go

```
import (
  "os"
  "path"

  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/aws-cdk-go/awscdk/v2/awsecrassets"
  "github.com/aws/aws-cdk-go/awscdk/v2/awsecs"
)

dirName, err := os.Getwd()
if err != nil {
  panic(err)
}

taskDefinition := awsecs.NewTaskDefinition(stack, jsii.String("TaskDef"),
 &awsecs.TaskDefinitionProps{
  MemoryMiB: jsii.String("1024"),
  Cpu: jsii.String("512"),
})

asset := awsecrassets.NewDockerImageAsset(stack, jsii.String("MyBuildImage"),
 &awsecrassets.DockerImageAssetProps{
  Directory: jsii.String(path.Join(dirName, "my-image")),
})
```

```
taskDefinition.AddContainer(jsii.String("MyOtherContainer"),
 &awsecs.ContainerDefinitionOptions{
  Image: awsecs.ContainerImage_FromDockerImageAsset(asset),
})
```

部署時間屬性範例

下列範例顯示如何使用部署時間屬性，以`repository`及如`imageUri`何使用啟動類型建立 Amazon ECS 任務定義。 AWS Fargate 請注意，Amazon ECR 存放庫查詢需要映像的標籤，而不是其 URI，因此我們會從資產 URI 的末尾進行剪取。

TypeScript

```
import * as ecs from 'aws-cdk-lib/aws-ecs';
import * as path from 'path';
import { DockerImageAsset } from 'aws-cdk-lib/aws-ecr-assets';

const asset = new DockerImageAsset(this, 'my-image', {
  directory: path.join(__dirname, "..", "demo-image")
});

const taskDefinition = new ecs.FargateTaskDefinition(this, "TaskDef", {
  memoryLimitMiB: 1024,
  cpu: 512
});

taskDefinition.addContainer("my-other-container", {
  image: ecs.ContainerImage.fromEcrRepository(asset.repository,
 asset.imageUri.split(":").pop())
});
```

JavaScript

```
const ecs = require('aws-cdk-lib/aws-ecs');
const path = require('path');
const { DockerImageAsset } = require('aws-cdk-lib/aws-ecr-assets');

const asset = new DockerImageAsset(this, 'my-image', {
  directory: path.join(__dirname, "..", "demo-image")
});
```

```
const taskDefinition = new ecs.FargateTaskDefinition(this, "TaskDef", {
  memoryLimitMiB: 1024,
  cpu: 512
});

taskDefinition.addContainer("my-other-container", {
  image: ecs.ContainerImage.fromEcrRepository(asset.repository,
 asset.imageUri.split(":").pop())
});
```

Python

```
import aws_cdk.aws_ecs as ecs
from aws_cdk.aws_ecr_assets import DockerImageAsset

import os.path
dirname = os.path.dirname(__file__)

asset = DockerImageAsset(self, 'my-image',
    directory=os.path.join(dirname, "..", "demo-image"))

task_definition = ecs.FargateTaskDefinition(self, "TaskDef",
    memory_limit_mib=1024, cpu=512)

task_definition.add_container("my-other-container",
    image=ecs.ContainerImage.from_ecr_repository(
        asset.repository, asset.image_uri.rpartition(":")[-1]))
```

Java

```
import java.io.File;

import software.amazon.awscdk.services.ecr.assets.DockerImageAsset;

import software.amazon.awscdk.services.ecs.FargateTaskDefinition;
import software.amazon.awscdk.services.ecs.ContainerDefinitionOptions;
import software.amazon.awscdk.services.ecs.ContainerImage;

File startDir = new File(System.getProperty("user.dir"));

DockerImageAsset asset = DockerImageAsset.Builder.create(this, "my-image")
            .directory(new File(startDir, "demo-image").toString()).build();
```

```
FargateTaskDefinition taskDefinition = FargateTaskDefinition.Builder.create(
        this, "TaskDef").memoryLimitMiB(1024).cpu(512).build();

// extract the tag from the asset's image URI for use in ECR repo lookup
String imageUri = asset.getImageUri();
String imageTag = imageUri.substring(imageUri.lastIndexOf(":") + 1);

taskDefinition.addContainer("my-other-container",
        ContainerDefinitionOptions.builder().image(ContainerImage.fromEcrRepository(
                asset.getRepository(), imageTag)).build());
```

C#

```
using System.IO;
using Amazon.CDK.AWS.ECS;
using Amazon.CDK.AWS.ECR.Assets;

var asset = new DockerImageAsset(this, "my-image", new DockerImageAssetProps {
    Directory = Path.Combine(Directory.GetCurrentDirectory(), "demo-image")
});

var taskDefinition = new FargateTaskDefinition(this, "TaskDef", new
 FargateTaskDefinitionProps
{
    MemoryLimitMiB = 1024,
    Cpu = 512
});

taskDefinition.AddContainer("my-other-container", new ContainerDefinitionOptions
{
    Image = ContainerImage.FromEcrRepository(asset.Repository,
 asset.ImageUri.Split(":").Last())
});
```

Go

```
import (
  "os"
  "path"

  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/aws-cdk-go/awscdk/v2/awsecrassets"
  "github.com/aws/aws-cdk-go/awscdk/v2/awsecs"
```

```
)

dirName, err := os.Getwd()
if err != nil {
  panic(err)
}

asset := awsecrassets.NewDockerImageAsset(stack, jsii.String("MyImage"),
 &awsecrassets.DockerImageAssetProps{
  Directory: jsii.String(path.Join(dirName, "demo-image")),
})

taskDefinition := awsecs.NewFargateTaskDefinition(stack, jsii.String("TaskDef"),
 &awsecs.FargateTaskDefinitionProps{
  MemoryLimitMiB: jsii.Number(1024),
  Cpu: jsii.Number(512),
})

taskDefinition.AddContainer(jsii.String("MyOtherContainer"),
 &awsecs.ContainerDefinitionOptions{
  Image: awsecs.ContainerImage_FromEcrRepository(asset.Repository(),
 asset.ImageTag()),
})
```

構建參數示例

當 AWS CDK CLI 在部署期間建置映像時，您可以透過 buildArgs (Python:build_args) 屬性選項，為 Docker 建置步驟提供自訂的建置引數。

TypeScript

```
const asset = new DockerImageAsset(this, 'MyBuildImage', {
  directory: path.join(__dirname, 'my-image'),
  buildArgs: {
    HTTP_PROXY: 'http://10.20.30.2:1234'
  }
});
```

JavaScript

```
const asset = new DockerImageAsset(this, 'MyBuildImage', {
  directory: path.join(__dirname, 'my-image'),
```

```
  buildArgs: {
    HTTP_PROXY: 'http://10.20.30.2:1234'
  }
});
```

Python

```
asset = DockerImageAsset(self, "MyBulidImage",
    directory=os.path.join(dirname, "my-image"),
    build_args=dict(HTTP_PROXY="http://10.20.30.2:1234"))
```

Java

```
DockerImageAsset asset = DockerImageAsset.Builder.create(this, "my-image"),
            .directory(new File(startDir, "my-image").toString())
            .buildArgs(java.util.Map.of(    // Java 9 or later
                "HTTP_PROXY", "http://10.20.30.2:1234"))
            .build();
```

C#

```
var asset = new DockerImageAsset(this, "MyBuildImage", new DockerImageAssetProps {
    Directory = Path.Combine(Directory.GetCurrentDirectory(), "my-image"),
    BuildArgs = new Dictionary<string, string>
    {
        ["HTTP_PROXY"] = "http://10.20.30.2:1234"
    }
});
```

Go

```
dirName, err := os.Getwd()
if err != nil {
  panic(err)
}

asset := awsecrassets.NewDockerImageAsset(stack, jsii.String("MyBuildImage"),
 &awsecrassets.DockerImageAssetProps{
  Directory: jsii.String(path.Join(dirName, "my-image")),
  BuildArgs: &map[string]*string{
    "HTTP_PROXY": jsii.String("http://10.20.30.2:1234"),
  },
```

```
})
```

許可

如果您使用支援 Docker 影像資產的模組 (例如 aws-ecs)，則當您直接或透過使用資產時，會為您 AWS CDK 管理權限。ContainerImage fromEcrRepository（Python：`from_ecr_repository`）。如果您直接使用 Docker 影像資產，請確定使用主體具有提取影像的權限。

在大多數情況下，您應該使用資產. 存儲庫. `grant_pull` 這會修改主體的 IAM 政策，使其能夠從此存放庫提取映像。 如果提取映像的主體不在同一個帳戶中，或者它是不承擔您帳戶中角色的 AWS 服務 (例如 AWS CodeBuild)，則您必須授與資源策略的提取權限，而不是對主體的策略授與提取權限。 使用資產存儲庫。 addToResource原則方法 (Python:`add_to_resource_policy`) 授與適當的主體權限。

AWS CloudFormation 資源元數據

> **Note**
>
> 本節僅適用於建構作者。在某些情況下，工具需要知道某些 CFN 資源正在使用本地資產。例如，您可以使用 AWS SAM CLI 在本機叫用 Lambda 函數以進行偵錯。如需詳細資訊，請參閱 the section called “AWS SAM 整合”。

若要啟用此類使用案例，外部工具會在 AWS CloudFormation 資源上查詢一組中繼資料項目：

- `aws:asset:path`— 指向資產的本機路徑。
- `aws:asset:property`— 使用資產的資源屬性名稱。

使用這兩個中繼資料項目，工具可以識別特定資源使用的資產，並啟用進階本機體驗。

若要將這些中繼資料項目新增至資源，請使用 `asset.addResourceMetadata` (Python:`add_resource_metadata`) 方法。

許可

AWS 構造庫使用一些常見的，廣泛實施的習慣用法來管理訪問和權限。IAM 模塊為您提供了使用這些習語所需的工具。

AWS CDK 用 AWS CloudFormation 於部署變更。每個部署都涉及啟動部 AWS CloudFormation 署的 actor (開發人員或自動化系統)。在執行此操作的過程中，實行者將採用一個或多個 IAM 身分 (使用者或角色)，並選擇性地將角色傳遞給 AWS CloudFormation。

如果您 AWS IAM Identity Center 以使用者身分驗證，則單一登入提供者會提供短期工作階段登入資料，以授權您做為預先定義的 IAM 角色。若要瞭解如何從 IAM 身分中心身分驗 AWS 證 AWS CDK 取得登入資料，請參閱 AWS SDK 和工具參考指南中的了解 IAM 身分中心身分驗證。

主體

IAM 主體是經過驗證的 AWS 實體，代表可呼叫 AWS API 的使用者、服務或應用程式。「 AWS 建構物件庫」支援以數種彈性的方式指定主參與者，以授與他們存取您的 AWS 資源。

在安全性內容中，術語「主體」特別指的是經過驗證的實體，例如使用者。群組和角色等物件不代表使用者 (和其他已驗證的實體)，而是為了授與權限而間接識別它們。

例如，如果您建立 IAM 群組，您可以授與該群組 (以及其成員) 對 Amazon RDS 表的寫入存取權。不過，群組本身並不是主參與者，因為它不代表單一實體 (也無法登入群組)。

在 CDK 的 IAM 程式庫中，直接或間接識別主體的類別會實作IPrincipal介面，從而允許在存取政策中互換使用這些物件。但是，並非所有這些都是安全性意義上的主體。這些物件包括：

1. IAM 資源Role，例如User、和 Group
2. 服務主體 () `new iam.ServicePrincipal('service.amazonaws.com')`
3. 同盟主參與者 () `new iam.FederatedPrincipal('cognito-identity.amazonaws.com')`
4. 帳戶主體 (`new iam.AccountPrincipal('0123456789012')`)
5. 規範使用者主體 () `new iam.CanonicalUserPrincipal('79a59d[...]7ef2be')`
6. AWS Organizations 校長 () `new iam.OrganizationPrincipal('org-id')`
7. 任意 ARN 主體 () `new iam.ArnPrincipal(res.arn)`
8. 一個`iam.CompositePrincipal(principal1, principal2, ...)`信任多個主體

授權

每個代表可存取之資源的建構 (例如 Amazon S3 儲存貯體或 Amazon DynamoDB 表) 都有授與其他實體存取權的方法。所有這些方法都有名稱以 grant 開頭。

例如，Amazon S3 儲存貯體具有方法grantRead和 grantReadWrite (Python:grant_read,grant_read_write) 可分別啟用從實體到儲存貯體的讀取和讀取/寫入存取。實體不需要確切知道執行這些操作需要哪些 Amazon S3 IAM 許可。

授予方法的第一個參數始終是 IGRAN Table 類型。此介面代表可授與權限的實體。也就是說，它代表具有角色的資源，例如 IAM 物件RoleUser、和Group.

其他實體也可以被授與權限。例如，在本主題稍後，我們將展示如何授與 CodeBuild專案存取 Amazon S3 儲存貯體的權限。一般而言，關聯的角色是透過被授與存取權之實體上的role屬性取得。

使用執行角色的資源 (例如lambda.Function) 也會實作IGrantable，因此您可以直接授與他們存取權，而不是授與其角色的存取權。例如，如果bucket是 Amazon S3 儲存貯體，且function是 Lambda 函數，則下列程式碼會授與該儲存貯體的函數讀取存取權限。

TypeScript

```
bucket.grantRead(function);
```

JavaScript

```
bucket.grantRead(function);
```

Python

```
bucket.grant_read(function)
```

Java

```
bucket.grantRead(function);
```

C#

```
bucket.GrantRead(function);
```

有時候，必須在部署堆疊時套用權限。其中一種情況是，當您將 AWS CloudFormation 自訂資源存取權授予其他資源時。自訂資源將在部署期間叫用，因此在部署時必須具有指定的權限。

另一種情況是，當服務驗證您傳遞給該服務的角色是否已套用正確的原則。（許多 AWS 服務這樣做是為了確保您沒有忘記設置策略。）在這些情況下，如果套用權限太晚，部署可能會失敗。

若要強制授權的權限在建立另一個資源之前套用，您可以新增授權本身的相依性，如下所示。儘管 grant 方法的返回值通常被丟棄，但實際上每個授予方法都會返回一個`iam.Grant`對象。

TypeScript

```
const grant = bucket.grantRead(lambda);
const custom = new CustomResource(...);
custom.node.addDependency(grant);
```

JavaScript

```
const grant = bucket.grantRead(lambda);
const custom = new CustomResource(...);
custom.node.addDependency(grant);
```

Python

```
grant = bucket.grant_read(function)
custom = CustomResource(...)
custom.node.add_dependency(grant)
```

Java

```
Grant grant = bucket.grantRead(function);
CustomResource custom = new CustomResource(...);
custom.node.addDependency(grant);
```

C#

```
var grant = bucket.GrantRead(function);
var custom = new CustomResource(...);
custom.node.AddDependency(grant);
```

角色

IAM 套件包含代表 IAM 角色的`Role`建構。下面的代碼創建一個新的角色，信任 Amazon EC2 服務。

TypeScript

```
import * as iam from 'aws-cdk-lib/aws-iam';
```

```
const role = new iam.Role(this, 'Role', {
  assumedBy: new iam.ServicePrincipal('ec2.amazonaws.com'),   // required
});
```

JavaScript

```
const iam = require('aws-cdk-lib/aws-iam');

const role = new iam.Role(this, 'Role', {
  assumedBy: new iam.ServicePrincipal('ec2.amazonaws.com')   // required
});
```

Python

```
import aws_cdk.aws_iam as iam

role = iam.Role(self, "Role",
          assumed_by=iam.ServicePrincipal("ec2.amazonaws.com")) # required
```

Java

```
import software.amazon.awscdk.services.iam.Role;
import software.amazon.awscdk.services.iam.ServicePrincipal;

Role role = Role.Builder.create(this, "Role")
        .assumedBy(new ServicePrincipal("ec2.amazonaws.com")).build();
```

C#

```
using Amazon.CDK.AWS.IAM;

var role = new Role(this, "Role", new RoleProps
{
    AssumedBy = new ServicePrincipal("ec2.amazonaws.com"),   // required
});
```

您可以呼叫角色的addToPolicy方法 (Python:add_to_policy)，並傳入定義要新增規則的方法，將權限新增至角色。PolicyStatement陳述式會新增至角色的預設原則；如果沒有陳述式，則會建立一個陳述式。

下列範例會在授權服務的條件下，將Deny政策陳述式新增至動作ec2:SomeAction的角色以bucket及資源和 otherRole (Python:other_role) AWS CodeBuild。s3:AnotherAction

TypeScript

```
role.addToPolicy(new iam.PolicyStatement({
  effect: iam.Effect.DENY,
  resources: [bucket.bucketArn, otherRole.roleArn],
  actions: ['ec2:SomeAction', 's3:AnotherAction'],
  conditions: {StringEquals: {
    'ec2:AuthorizedService': 'codebuild.amazonaws.com',
}}}));
```

JavaScript

```
role.addToPolicy(new iam.PolicyStatement({
  effect: iam.Effect.DENY,
  resources: [bucket.bucketArn, otherRole.roleArn],
  actions: ['ec2:SomeAction', 's3:AnotherAction'],
  conditions: {StringEquals: {
    'ec2:AuthorizedService': 'codebuild.amazonaws.com'
}}}));
```

Python

```
role.add_to_policy(iam.PolicyStatement(
    effect=iam.Effect.DENY,
    resources=[bucket.bucket_arn, other_role.role_arn],
    actions=["ec2:SomeAction", "s3:AnotherAction"],
    conditions={"StringEquals": {
        "ec2:AuthorizedService": "codebuild.amazonaws.com"}}
))
```

Java

```
role.addToPolicy(PolicyStatement.Builder.create()
        .effect(Effect.DENY)
        .resources(Arrays.asList(bucket.getBucketArn(), otherRole.getRoleArn()))
        .actions(Arrays.asList("ec2:SomeAction", "s3:AnotherAction"))
        .conditions(java.util.Map.of(    // Map.of requires Java 9 or later
            "StringEquals", java.util.Map.of(
```

```
                "ec2:AuthorizedService", "codebuild.amazonaws.com")))
        .build());
```

C#

```
role.AddToPolicy(new PolicyStatement(new PolicyStatementProps
{
    Effect = Effect.DENY,
    Resources = new string[] { bucket.BucketArn, otherRole.RoleArn },
    Actions = new string[] { "ec2:SomeAction", "s3:AnotherAction" },
    Conditions = new Dictionary<string, object>
    {
        ["StringEquals"] = new Dictionary<string, string>
        {
            ["ec2:AuthorizedService"] = "codebuild.amazonaws.com"
        }
    }
}));
```

在前面的例子中，我們創建了一個新的PolicyStatement內聯 addToPolicy (Python:add_to_policy) 調用。您也可以傳入現有的政策聲明或您已修改的政策聲明。PolicyStatement物件有許多新增主參與者、資源、條件和動作的方法。

如果您使用的建構需要角色才能正常運作，則可以執行下列其中一項操作：

- 實例化構造對象時傳遞現有的角色。
- 讓建構為您建立新角色，信任適當的服務主體。下列範例會使用這樣的建構： CodeBuild 專案。

TypeScript

```
import * as codebuild from 'aws-cdk-lib/aws-codebuild';

// imagine roleOrUndefined is a function that might return a Role object
// under some conditions, and undefined under other conditions
const someRole: iam.IRole | undefined = roleOrUndefined();

const project = new codebuild.Project(this, 'Project', {
  // if someRole is undefined, the Project creates a new default role,
  // trusting the codebuild.amazonaws.com service principal
  role: someRole,
```

```
});
```

JavaScript

```
const codebuild = require('aws-cdk-lib/aws-codebuild');

// imagine roleOrUndefined is a function that might return a Role object
// under some conditions, and undefined under other conditions
const someRole = roleOrUndefined();

const project = new codebuild.Project(this, 'Project', {
  // if someRole is undefined, the Project creates a new default role,
  // trusting the codebuild.amazonaws.com service principal
  role: someRole
});
```

Python

```
import aws_cdk.aws_codebuild as codebuild

# imagine role_or_none is a function that might return a Role object
# under some conditions, and None under other conditions
some_role = role_or_none();

project = codebuild.Project(self, "Project",
# if role is None, the Project creates a new default role,
# trusting the codebuild.amazonaws.com service principal
role=some_role)
```

Java

```
import software.amazon.awscdk.services.iam.Role;
import software.amazon.awscdk.services.codebuild.Project;

// imagine roleOrNull is a function that might return a Role object
// under some conditions, and null under other conditions
Role someRole = roleOrNull();

// if someRole is null, the Project creates a new default role,
// trusting the codebuild.amazonaws.com service principal
Project project = Project.Builder.create(this, "Project")
        .role(someRole).build();
```

C#

```
using Amazon.CDK.AWS.CodeBuild;

// imagine roleOrNull is a function that might return a Role object
// under some conditions, and null under other conditions
var someRole = roleOrNull();

// if someRole is null, the Project creates a new default role,
// trusting the codebuild.amazonaws.com service principal
var project = new Project(this, "Project", new ProjectProps
{
    Role = someRole
});
```

建立物件之後，角色 (無論是傳入的角色或建構所建立的預設角色) 都可作為屬性使用`role`。但是，此屬性在外部資源上不可用。因此，這些建構具有 `addToRolePolicy` (Python:`add_to_role_policy`) 方法。

如果構造是外部資源，則該方法不執行任何操作，否則它會調用該`role`屬性的 `addToPolicy` (Python:`add_to_policy`) 方法。這樣可以避免明確處理未定義大小寫的麻煩。

下面的例子演示：

TypeScript

```
// project is imported into the CDK application
const project = codebuild.Project.fromProjectName(this, 'Project', 'ProjectName');

// project is imported, so project.role is undefined, and this call has no effect
project.addToRolePolicy(new iam.PolicyStatement({
  effect: iam.Effect.ALLOW,   // ... and so on defining the policy
}));
```

JavaScript

```
// project is imported into the CDK application
const project = codebuild.Project.fromProjectName(this, 'Project', 'ProjectName');

// project is imported, so project.role is undefined, and this call has no effect
project.addToRolePolicy(new iam.PolicyStatement({
```

```
  effect: iam.Effect.ALLOW   // ... and so on defining the policy
}));
```

Python

```
# project is imported into the CDK application
project = codebuild.Project.from_project_name(self, 'Project', 'ProjectName')

# project is imported, so project.role is undefined, and this call has no effect
project.add_to_role_policy(iam.PolicyStatement(
  effect=iam.Effect.ALLOW,   # ... and so on defining the policy
)
```

Java

```
// project is imported into the CDK application
Project project = Project.fromProjectName(this, "Project", "ProjectName");

// project is imported, so project.getRole() is null, and this call has no effect
project.addToRolePolicy(PolicyStatement.Builder.create()
        .effect(Effect.ALLOW)   // .. and so on defining the policy
        .build());
```

C#

```
// project is imported into the CDK application
var project = Project.FromProjectName(this, "Project", "ProjectName");

// project is imported, so project.role is null, and this call has no effect
project.AddToRolePolicy(new PolicyStatement(new PolicyStatementProps
{
    Effect = Effect.ALLOW, // ... and so on defining the policy
}));
```

資源政策

中 AWS的一些資源 (例如 Amazon S3 儲存貯體和 IAM 角色) 也有資源政策。這些構造有一個addToResourcePolicy方法（Python:add_to_resource_policy），它需要一個PolicyStatement作為它的參數。每個新增至資源策略的政策陳述式都必須至少指定一個主體。

在下列範例中，Amazon S3 儲存貯體bucket授予具有自身s3:SomeAction許可的角色。

TypeScript

```
bucket.addToResourcePolicy(new iam.PolicyStatement({
  effect: iam.Effect.ALLOW,
  actions: ['s3:SomeAction'],
  resources: [bucket.bucketArn],
  principals: [role]
}));
```

JavaScript

```
bucket.addToResourcePolicy(new iam.PolicyStatement({
  effect: iam.Effect.ALLOW,
  actions: ['s3:SomeAction'],
  resources: [bucket.bucketArn],
  principals: [role]
}));
```

Python

```
bucket.add_to_resource_policy(iam.PolicyStatement(
    effect=iam.Effect.ALLOW,
    actions=["s3:SomeAction"],
    resources=[bucket.bucket_arn],
    principals=role))
```

Java

```
bucket.addToResourcePolicy(PolicyStatement.Builder.create()
        .effect(Effect.ALLOW)
        .actions(Arrays.asList("s3:SomeAction"))
        .resources(Arrays.asList(bucket.getBucketArn()))
        .principals(Arrays.asList(role))
        .build());
```

C#

```
bucket.AddToResourcePolicy(new PolicyStatement(new PolicyStatementProps
{
    Effect = Effect.ALLOW,
    Actions = new string[] { "s3:SomeAction" },
    Resources = new string[] { bucket.BucketArn },
```

```
    Principals = new IPrincipal[] { role }
}));
```

使用外部 IAM 物件

如果您已在應用程式外定義 IAM 使用者、主體、群組或角色，則可以在 AWS CDK 應用程 AWS CDK 式中使用該 IAM 物件。若要這麼做，請使用其 ARN 或其名稱來建立對它的參考。(使用者、群組和角色的名稱。) 然後，返回的引用可用於授予權限或構建策略語句，如前面所述。

- 對於使用者，請撥打`User.fromUserArn()`或`User.fromUserName()`。`User.fromUserAttributes()`也可以使用，但目前提供的功能與`User.fromUserArn()`.
- 對於主體，實例化一`ArnPrincipal`個對象。
- 對於群組，請撥打電話`Group.fromGroupArn()`或`Group.fromGroupName()`。
- 對於角色，請撥打`Role.fromRoleArn()`或`Role.fromRoleName()`。

原則 (包括受管理的策略) 可以使用下列方法，以類似的方式使用。您可以在需要 IAM 政策的任何地方使用對這些物件的參考。

- `Policy.fromPolicyName`
- `ManagedPolicy.fromManagedPolicyArn`
- `ManagedPolicy.fromManagedPolicyName`
- `ManagedPolicy.fromAwsManagedPolicyName`

> **Note**
>
> 與所有對外部 AWS 資源的引用一樣，您無法在 CDK 應用程序中修改外部 IAM 對象。

執行期內容

上下文值是可以與應用程序，堆棧或構造關聯的鍵值對。它們可以從文件 (通常或在您的項目目錄中) `cdk.json`或`cdk.context.json`在命令行上提供給您的應用程序。

CDK 工具包使用上下文緩存在合成期間從您的 AWS 帳戶中檢索到的值。值包括帳戶中的可用區域，或 Amazon EC2 執行個體目前可用的亞馬遜機器映像 (AMI) ID。由於這些值是由您的 AWS 帳戶提供

的，因此它們可以在 CDK 應用程式的執行之間進行變更。這使得它們成為意外改變的潛在來源。CDK Toolkit 的快取行為會「凍結」CDK 應用程式的這些值，直到您決定接受新值為止。

想像下面沒有上下文緩存的情況。假設您指定了「最新的 Amazon Linux」作為 Amazon EC2 實例的 AMI，並發布了此 AMI 的新版本。然後，下次您部署 CDK 堆疊時，您已經部署的執行個體將會使用過時的 (「錯誤」) AMI，而且需要升級。升級將導致用新的實例替換所有現有實例，這可能是意外且不希望的。

相反，CDK 會將您帳戶的可用 AMI 記錄在項目的`cdk.context.json`文件中，並將存儲值用於 future 的綜合操作。如此一來，AMI 清單就不再是潛在的變更來源。您還可以確保堆棧將始終合成到相同的 AWS CloudFormation 模板中。

並非所有上下文值都是從您的 AWS 環境中緩存的值。 the section called “功能旗標”也是上下文值。您也可以建立自己的內容值，供應用程式或建構使用。

上下文鍵是字符串。值可以是 JSON 支持的任何類型：數字，字符串，數組或對象。

> **Tip**
>
> 如果您的構造創建自己的上下文值，請將庫的包名稱合併到其鍵中，以便它們不會與其他軟件包的上下文值衝突。

許多上下文值都與特定 AWS 環境相關聯，並且給定的 CDK 應用程序可以部署在多個環境中。這些值的鍵包括 AWS 帳戶和區域，以便來自不同環境的值不會發生衝突。

下列內容索引鍵說明了使用的格式 AWS CDK，包括帳戶與區域。

```
availability-zones:account=123456789012:region=eu-central-1
```

> **Important**
>
> 快取的內容值由 AWS CDK 及其建構管理，包括您可以撰寫的建構。請勿透過手動編輯檔案來新增或變更快取的內容值。不過，`cdk.context.json`偶爾檢閱以查看快取的值可能會很有用。不代表快取值的內容值應儲存在的`context`索引鍵下`cdk.json`。這樣，當緩存值被清除時，它們不會被清除。

上下文值的來源

上下文值可以通過六種不同的方式提供給您的 AWS CDK 應用程序：

- 自動從當前 AWS 帳戶。
- 通過--context選項的cdk命令。（這些值始終是字符串。）
- 在專案的cdk.context.json檔案中。
- 在項目cdk.json文件的context密鑰中。
- 在你的~/.cdk.json文件的context密鑰。
- 在您的 AWS CDK 應用程序中使用該construct.node.setContext()方法。

專案檔案cdk.context.json是 AWS CDK 快取從您 AWS 帳戶擷取的內容值的位置。例如，引入新的可用區域時，此做法可避免部署發生非預期的變更。 AWS CDK 不會將內容資料寫入列出的任何其他檔案。

> **Important**
>
> 因為它們是應用程序狀態的一部分，cdk.json並且cdk.context.json必須與應用程序的源代碼的其餘部分一起致力於源代碼控制。否則，在其他環境 (例如 CI 管線) 中的部署可能會產生不一致的結果。

前後關聯值的範圍是建立它們的建構；它們對子建構可見，但父項或同層級則不可見。由 AWS CDK Toolkit (cdk命令) 設定的前後關聯值可以自動設定、從檔案或從--context選項設定。這些來源的前後關聯值會在App建構上隱含地設定。因此，應用程序中每個堆棧中的每個構造都可以看到它們。

您的應用程序可以使用該construct.node.tryGetContext方法讀取上下文值。如果在當前構造或其任何父項上找不到請求的條目，則結果為undefined。（或者，結果可能是您的語言的等價物，例如None在 Python 中。）

內容方法

AWS CDK 支援數種內容方法，可讓 AWS CDK 應用程式從 AWS 環境中取得內容資訊。例如，您可以使用堆疊。可用區域方法，取得指定 AWS 帳戶和區域中可用的可用區域清單。

以下是上下文方法：

HostedZone. 從查找

獲取您帳戶中的託管區域。

堆疊. 可用性區域

取得支援的可用區域。

StringParameter.valueFromLookup

從目前區域的 Amazon EC2 Systems Manager 參數存放區取得值。

來自查找

取得您帳戶中現有的 Amazon 虛擬私有雲。

LookupMachineImage

查詢機器映像檔，以便與 Amazon 虛擬私有雲中的 NAT 執行個體搭配使用。

如果所需的上下文值不可用， AWS CDK 應用程序會通知 CDK Toolkit 缺少上下文信息。接下來，CLI 會查詢目前 AWS 帳戶中的資訊，並將產生的內容資訊儲存在`cdk.context.json`檔案中。然後，它使用上下文值再次執行 AWS CDK 應用程序。

檢視及管理前後關聯

使用cdk context指令檢視和管理`cdk.context.json`檔案中的資訊。若要查看此資訊，請不要使用任何選項的cdk context指令。輸出應該是如下所示。

```
Context found in cdk.json:

##############################################################################################
# # # Key                                                                  # Value
                                     #
##############################################################################################
# 1 # availability-zones:account=123456789012:region=eu-central-1 # [ "eu-central-1a",
 "eu-central-1b", "eu-central-1c" ]   #
##############################################################################################
# 2 # availability-zones:account=123456789012:region=eu-west-1    # [ "eu-west-1a",
 "eu-west-1b", "eu-west-1c" ]            #
##############################################################################################

Run cdk context --reset KEY_OR_NUMBER to remove a context key. If it is a cached value,
 it will be refreshed on the next cdk synth.
```

要刪除上下文值，請運行cdk context --reset，並指定該值的對應鍵或數字。下列範例會移除與前述範例中第二個索引鍵對應的值。此值代表歐洲 (愛爾蘭) 區域的可用區域清單。

```
cdk context --reset 2
```

```
Context value
availability-zones:account=123456789012:region=eu-west-1
reset. It will be refreshed on the next SDK synthesis run.
```

因此，如果您想要更新至最新版本的 Amazon Linux AMI，請使用上述範例對內容值進行控制更新並重設。然後，再次合成並部署您的應用程序。

```
cdk synth
```

若要清除應用程式的所有儲存內容值，請執行cdk context --clear下列步驟。

```
cdk context --clear
```

只有儲存在中的前後關聯值cdk.context.json可以重設或清除。 AWS CDK 不會接觸其他上下文值。因此，若要防止使用這些指令重設前後關聯值，您可以將值複製到cdk.json。

AWS CDK 工具包**--context**標誌

在合成或部署期間，使用--context（ -c簡稱 ）選項將運行時上下文值傳遞給 CDK 應用程序。

```
cdk synth --context key=value MyStack
```

若要指定多個內容值，請重複--context此選項任意次數，每次提供一個索引鍵值配對。

```
cdk synth --context key1=value1 --context key2=value2 MyStack
```

合成多個堆棧時，指定的上下文值將傳遞給所有堆棧。要為單個堆棧提供不同的上下文值，請為值使用不同的鍵，或者使用多個cdk synth或cdk deploy命令。

從命令行傳遞的上下文值始終是字符串。如果值通常是其他類型，您的程式碼必須準備好轉換或剖析該值。您可能有以其他方式提供的非字串內容值 (例如，在中cdk.context.json)。若要確定這種值如預期般運作，請在轉換之前確認該值是字串。

範例

以下是使用 AWS CDK 內容使用現有 Amazon VPC 的範例。

TypeScript

```
import * as cdk from 'aws-cdk-lib';
import * as ec2 from 'aws-cdk-lib/aws-ec2';
import { Construct } from 'constructs';

export class ExistsVpcStack extends cdk.Stack {

  constructor(scope: Construct, id: string, props?: cdk.StackProps) {

    super(scope, id, props);

    const vpcid = this.node.tryGetContext('vpcid');
    const vpc = ec2.Vpc.fromLookup(this, 'VPC', {
      vpcId: vpcid,
    });

    const pubsubnets = vpc.selectSubnets({subnetType: ec2.SubnetType.PUBLIC});

    new cdk.CfnOutput(this, 'publicsubnets', {
      value: pubsubnets.subnetIds.toString(),
    });
  }
}
```

JavaScript

```
const cdk = require('aws-cdk-lib');
const ec2 = require('aws-cdk-lib/aws-ec2');

class ExistsVpcStack extends cdk.Stack {

  constructor(scope, id, props) {

    super(scope, id, props);

    const vpcid = this.node.tryGetContext('vpcid');
    const vpc = ec2.Vpc.fromLookup(this, 'VPC', {
      vpcId: vpcid
```

```
    });

    const pubsubnets = vpc.selectSubnets({subnetType: ec2.SubnetType.PUBLIC});

    new cdk.CfnOutput(this, 'publicsubnets', {
      value: pubsubnets.subnetIds.toString()
    });
  }
}

module.exports = { ExistsVpcStack }
```

Python

```
import aws_cdk as cdk
import aws_cdk.aws_ec2 as ec2
from constructs import Construct

class ExistsVpcStack(cdk.Stack):

    def __init__(scope: Construct, id: str, **kwargs):

        super().__init__(scope, id, **kwargs)

        vpcid = self.node.try_get_context("vpcid")
        vpc = ec2.Vpc.from_lookup(self, "VPC", vpc_id=vpcid)

        pubsubnets = vpc.select_subnets(subnetType=ec2.SubnetType.PUBLIC)

        cdk.CfnOutput(self, "publicsubnets",
            value=pubsubnets.subnet_ids.to_string())
```

Java

```
import software.amazon.awscdk.CfnOutput;

import software.amazon.awscdk.services.ec2.Vpc;
import software.amazon.awscdk.services.ec2.VpcLookupOptions;
import software.amazon.awscdk.services.ec2.SelectedSubnets;
import software.amazon.awscdk.services.ec2.SubnetSelection;
import software.amazon.awscdk.services.ec2.SubnetType;
import software.constructs.Construct;
```

```
public class ExistsVpcStack extends Stack {
    public ExistsVpcStack(Construct context, String id) {
        this(context, id, null);
    }

    public ExistsVpcStack(Construct context, String id, StackProps props) {
        super(context, id, props);

        String vpcId = (String)this.getNode().tryGetContext("vpcid");
        Vpc vpc = (Vpc)Vpc.fromLookup(this, "VPC", VpcLookupOptions.builder()
                .vpcId(vpcId).build());

        SelectedSubnets pubSubNets = vpc.selectSubnets(SubnetSelection.builder()
                .subnetType(SubnetType.PUBLIC).build());

        CfnOutput.Builder.create(this, "publicsubnets")
                .value(pubSubNets.getSubnetIds().toString()).build();

    }
}
```

C#

```
using Amazon.CDK;
using Amazon.CDK.AWS.EC2;
using Constructs;

class ExistsVpcStack : Stack
{
    public ExistsVpcStack(Construct scope, string id, StackProps props) :
 base(scope, id, props)
    {
        var vpcId = (string)this.Node.TryGetContext("vpcid");
        var vpc = Vpc.FromLookup(this, "VPC", new VpcLookupOptions
        {
            VpcId = vpcId
        });

        SelectedSubnets pubSubNets = vpc.SelectSubnets([new SubnetSelection
        {
            SubnetType = SubnetType.PUBLIC
        }]);
```

```
        new CfnOutput(this, "publicsubnets", new CfnOutputProps {
            Value = pubSubNets.SubnetIds.ToString()
        });
    }
}
```

您可以使cdk diff用在命令行上查看傳遞上下文值的效果：

```
cdk diff -c vpcid=vpc-0cb9c31031d0d3e22
```

```
Stack ExistsvpcStack
Outputs
[+] Output publicsubnets publicsubnets:
 {"Value":"subnet-06e0ea7dd302d3e8f,subnet-01fc0acfb58f3128f"}
```

您可以檢視產生的前後關聯值，如下所示。

```
cdk context -j
```

```
{
  "vpc-provider:account=123456789012:filter.vpc-id=vpc-0cb9c31031d0d3e22:region=us-
east-1": {
    "vpcId": "vpc-0cb9c31031d0d3e22",
    "availabilityZones": [
      "us-east-1a",
      "us-east-1b"
    ],
    "privateSubnetIds": [
      "subnet-03ecfc033225be285",
      "subnet-0cded5da53180ebfa"
    ],
    "privateSubnetNames": [
      "Private"
    ],
    "privateSubnetRouteTableIds": [
      "rtb-0e955393ced0ada04",
      "rtb-05602e7b9f310e5b0"
    ],
    "publicSubnetIds": [
      "subnet-06e0ea7dd302d3e8f",
      "subnet-01fc0acfb58f3128f"
```

```
    ],
    "publicSubnetNames": [
      "Public"
    ],
    "publicSubnetRouteTableIds": [
      "rtb-00d1fdfd823c82289",
      "rtb-04bb1969b42969bcb"
    ]
  }
}
```

功能旗標

AWS CDK 使用功能標誌來啟用可能在發行版本中破壞的行為。旗標會儲存為 `cdk.json` (或`~/.cdk.json`) 中的the section called “Context”值。cdk context --reset或cdk context --clear指令不會移除它們。

功能旗標預設為停用。未指定旗標的現有專案將繼續像以前的 AWS CDK 版本一樣運作。使用cdk init包含旗標建立的新專案，可啟用建立專案的發行版本中所有可用的功能。編輯`cdk.json`以停用您偏好先前行為的任何旗標。您也可以在升級後新增旗標以啟用新行為 AWS CDK。

您可以在中的 AWS CDK GitHub 存放庫中找到目前所有功能旗標的清單`FEATURE_FLAGS.md`。如需`CHANGELOG`在該版本中新增的任何新功能旗標的說明，請參閱指定版本中的。

還原為 v1 行為

在 CDK v2 中，某些功能旗標的預設值已變更為 v1。您可以將這些設定回`false`來還原為特定 AWS CDK v1 行為。使用該`cdk diff`命令檢查對合成模板的更改，以查看是否需要這些標誌中的任何一個。

`@aws-cdk/core:newStyleStackSynthesis`

使用新的堆棧合成方法，該方法假定具有眾所周知名稱的引導資源。需要現代引導，但反過來允許 CI/CD 通過 CDK Pipelines 和跨帳戶部署開箱即用。

`@aws-cdk/aws-apigateway:usagePlanKeyOrderInsensitiveId`

如果您的應用程式使用多個 Amazon API Gateway API 金鑰，並將其與使用計劃建立關聯。

`@aws-cdk/aws-rds:lowercaseDbIdentifier`

如果您的應用程式使用 Amazon RDS 資料庫執行個體或資料庫叢集，並明確指定這些叢集的識別碼。

`@aws-cdk/aws-cloudfront:defaultSecurityPolicyTLSv1.2_2021`

如果您的應用程式使用 TLS_V1_2_2019 安全性原則搭配發行版本。 Amazon CloudFront 根據預設，CDK v2 會使用安全性原則。

`@aws-cdk/core:stackRelativeExports`

如果您的應用程序使用多個堆棧，並且您引用另一個堆棧中的資源，這將確定是否使用絕對路徑或相對路徑來構建 AWS CloudFormation 導出。

`@aws-cdk/aws-lambda:recognizeVersionProps`

如果設定為`false`，CDK 會在偵測 Lambda 函數是否已變更時包含中繼資料。如果只有中繼資料已變更，這可能會導致部署失敗，因為不允許重複的版本。如果您已對應用程式中的所有 Lambda 函數進行至少一次變更，則不需要還原此旗標。

在`cdk.json`中還原這些旗標的語法如下所示。

```
{
  "context": {
    "@aws-cdk/core:newStyleStackSynthesis": false,
    "@aws-cdk/aws-apigateway:usagePlanKeyOrderInsensitiveId": false,
    "@aws-cdk/aws-cloudfront:defaultSecurityPolicyTLSv1.2_2021": false,
    "@aws-cdk/aws-rds:lowercaseDbIdentifier": false,
    "@aws-cdk/core:stackRelativeExports": false,
    "@aws-cdk/aws-lambda:recognizeVersionProps": false
  }
}
```

面向

方面是一種將操作應用於給定範圍內的所有構造的方法。該方面可以修改結構，例如通過添加標籤。或者它可以驗證有關構造狀態的內容，例如確保所有存儲桶都已加密。

若要將縱橫比套用至建構及相同範圍內的所有建構，請`Aspects.of(SCOPE).add()`使用新的縱橫比呼叫，如下列範例所示。

TypeScript

```
Aspects.of(myConstruct).add(new SomeAspect(...));
```

JavaScript

```
Aspects.of(myConstruct).add(new SomeAspect(...));
```

Python

```
Aspects.of(my_construct).add(SomeAspect(...))
```

Java

```
Aspects.of(myConstruct).add(new SomeAspect(...));
```

C#

```
Aspects.Of(myConstruct).add(new SomeAspect(...));
```

Go

```
awscdk.Aspects_Of(stack).Add(awscdk.NewTag(...))
```

AWS CDK 使用方面來標記資源，但框架也可以用於其他目的。例如，您可以使用它來驗證或變更更高層級建構為您定義的 AWS CloudFormation 資源。

各個方面的細節

各個方面採用訪問者模式。一個方面是一個實現以下接口的類。

TypeScript

```
interface IAspect {
   visit(node: IConstruct): void;}
```

JavaScript

JavaScript 沒有作為語言功能的接口。因此，一個方面只是具有接受要操作的節點的`visit`方法的類的一個實例。

Python

Python 沒有接口作為一種語言功能。因此，一個方面只是具有接受要操作的節點的`visit`方法的類的一個實例。

Java

```
public interface IAspect {
    public void visit(Construct node);
}
```

C#

```
public interface IAspect
{
    void Visit(IConstruct node);
}
```

Go

```
type IAspect interface {
  Visit(node constructs.IConstruct)
}
```

當您調用時`Aspects.of(SCOPE).add(...)`，構造將方面添加到內部方面列表中。您可以使用取得清單`Aspects.of(SCOPE)`。

在準備階段，會以由上而下的順序 AWS CDK 呼叫建構的物件及其每個子項的`visit`方法。

該`visit`方法可以自由地更改構造中的任何內容。在強類型語言中，在訪問特定於構造的屬性或方法之前，將接收的構造轉換為更特定的類型。

方面不會跨越`Stage`構造邊界傳播，因為在定義之後`Stages`是獨立且不可變的。如果您希望它們訪問. `Stage` `Stage`

範例

下列範例會驗證堆疊中建立的所有值區是否已啟用版本控制。該方面將錯誤註釋添加到驗證失敗的構造中。這會導致synth作業失敗，並防止部署產生的雲端組件。

TypeScript

```
class BucketVersioningChecker implements IAspect {
  public visit(node: IConstruct): void {
    // See that we're dealing with a CfnBucket
```

```
    if (node instanceof s3.CfnBucket) {

      // Check for versioning property, exclude the case where the property
      // can be a token (IResolvable).
      if (!node.versioningConfiguration
        || (!Tokenization.isResolvable(node.versioningConfiguration)
            && node.versioningConfiguration.status !== 'Enabled')) {
        Annotations.of(node).addError('Bucket versioning is not enabled');
      }
    }
  }
}

// Later, apply to the stack
Aspects.of(stack).add(new BucketVersioningChecker());
```

JavaScript

```
class BucketVersioningChecker {
   visit(node) {
    // See that we're dealing with a CfnBucket
    if ( node instanceof s3.CfnBucket) {

      // Check for versioning property, exclude the case where the property
      // can be a token (IResolvable).
      if (!node.versioningConfiguration
        || !Tokenization.isResolvable(node.versioningConfiguration)
            && node.versioningConfiguration.status !== 'Enabled')) {
        Annotations.of(node).addError('Bucket versioning is not enabled');
      }
    }
  }
}

// Later, apply to the stack
Aspects.of(stack).add(new BucketVersioningChecker());
```

Python

```
@jsii.implements(cdk.IAspect)
class BucketVersioningChecker:

  def visit(self, node):
```

```
    # See that we're dealing with a CfnBucket
    if isinstance(node, s3.CfnBucket):

      # Check for versioning property, exclude the case where the property
      # can be a token (IResolvable).
      if (not node.versioning_configuration or
              not Tokenization.is_resolvable(node.versioning_configuration)
                  and node.versioning_configuration.status != "Enabled"):
          Annotations.of(node).add_error('Bucket versioning is not enabled')

# Later, apply to the stack
Aspects.of(stack).add(BucketVersioningChecker())
```

Java

```
public class BucketVersioningChecker implements IAspect
{
    @Override
    public void visit(Construct node)
    {
        // See that we're dealing with a CfnBucket
        if (node instanceof CfnBucket)
        {
            CfnBucket bucket = (CfnBucket)node;
            Object versioningConfiguration = bucket.getVersioningConfiguration();
            if (versioningConfiguration == null ||
                    !Tokenization.isResolvable(versioningConfiguration.toString())
 &&
                    !versioningConfiguration.toString().contains("Enabled"))
                Annotations.of(bucket.getNode()).addError("Bucket versioning is not
 enabled");
        }
    }
}


// Later, apply to the stack
Aspects.of(stack).add(new BucketVersioningChecker());
```

C#

```
class BucketVersioningChecker : Amazon.Jsii.Runtime.Deputy.DeputyBase, IAspect
{
```

```
    public void Visit(IConstruct node)
    {
        // See that we're dealing with a CfnBucket
        if (node is CfnBucket)
        {
            var bucket = (CfnBucket)node;
            if (bucket.VersioningConfiguration is null ||
                    !Tokenization.IsResolvable(bucket.VersioningConfiguration) &&
                    !bucket.VersioningConfiguration.ToString().Contains("Enabled"))
                Annotations.Of(bucket.Node).AddError("Bucket versioning is not
 enabled");
        }
    }
}

// Later, apply to the stack
Aspects.Of(stack).add(new BucketVersioningChecker());
```

開始使用 AWS CDK

安裝 AWS CDK CLI並建立您的第一個 CDK 應用程式，開始使用。 AWS Cloud Development Kit (AWS CDK)

主題

必要條件

推薦資源

在開始使用之前 AWS CDK，我們建議您對以下內容進行基本了解：

- 簡介 AWS CDK. 如需進一步了解，請參閱 什麼是 AWS CDK？
- 背後的核心概念 AWS CDK. 如需進一步了解，請參閱 AWS CDK 概念。
- 您 AWS 服務 要使用管理的 AWS CDK.
- AWS Identity and Access Management。 如需詳細資訊，請參閱什麼是 IAM？ 什麼是 IAM 身分中心？
- AWS CloudFormation 因為 AWS CDK 利用該 AWS CloudFormation 服務來提供 CDK 中創建的資源。如需進一步了解，請參閱什麼是 AWS CloudFormation？
- 您計劃與. 一起使用的支援程式設計語言 AWS CDK。

準備您的本地環境

無論您偏好的語言為何，所有 AWS CDK 開發人員都需要 Node.js14.15.0 或更新版本。所有支持的編程語言都使用相同的後端，該後端運行在上Node.js。我們建議使用主動長期支援的版本。您的組織可能有不同的建議。

> **Important**
>
> Node.js 版本 13.0.0 到 13.6.0 與其相依性的相容 AWS CDK 性問題不相容。

其他必要條件取決於您開發 AWS CDK 應用程式所使用的語言，如下所示。

TypeScript

- TypeScript 3.8 或更高版本 (`npm -g install typescript`)

JavaScript

沒有其他要求

Python

- Python 3.7 或更高版本包括`pip`和 `virtualenv`

Java

- Java 開發工具包（JDK）8（又名 1.8）或更高版本
- 阿帕奇 Maven 3.5 或更高版本

推薦使用 Java IDE（我們在本指南中的一些示例中使用了 Eclipse）。IDE 必須能夠導入 Maven 項目。檢查以確保您的項目設置為使用 Java 1.8。將 JAVA_HOME 環境變數設定為您已安裝 JDK 的路徑。

C#

.NET 核心 3.1 或更新版本，或 .NET 6.0 或更新版本。

視覺工作室 2019（任何版本）或視覺工作室代碼推薦。

Go

轉到 1.1.8 或更高版本。

如需詳細資訊，請參閱您語言的先決條件一節：

- the section called “在 TypeScript”
- the section called “在 JavaScript”

- the section called “在 Python 中”
- the section called “在爪哇”
- the section called “在 C＃ 中”
- the section called “在圍棋”

> **第三方語言棄用**
>
> 每種語言版本僅在EOL（生命週期結束）之前受到支持，並且如有更改，恕不另行通知。

步驟 1：建立 AWS 帳戶

如果您不熟悉 AWS，則必須註冊 AWS 帳戶 並建立管理使用者。如需詳細資訊，請參閱 IAM 使用者指南中的使用 IAM 進行設定。

與之互動時 AWS，您可以指定 AWS 安全登入資料以驗證您的身分，以及您是否有權存取要求的資源。 AWS 使用安全認證來驗證和授權您的請求。若要進一步了解，請參閱 IAM 使用者指南中的AWS 安全登入資料。

步驟 2：設定程式設計存取

AWS CDK 在您的本機環境中進行開發時，您將仰賴與 AWS CDK CLI資源互動 AWS 服務 並管理資 AWS 源。若要使用 AWS CDK CLI，您必須設定程式設計存取。若要深入了解可以設定程式設計存取的不同方式，請參閱 AWS SDK 和工具參考指南中的驗證和存取。

對於沒有雇主授予身份驗證方法的新用戶，我們建議使用 AWS IAM Identity Center。此方法包括安裝 AWS Command Line Interface (AWS CLI) 並將其用於配置和登入 AWS 存取入口網站。若要使用 IAM 身分中心設定程式設計存取，請參閱 AWS SDK 和工具參考指南中的 IAM 身分中心身分驗證。完成之後，您的環境應包含下列元素：

- 您可以在 AWS CLI執行應用程式之前啟動 AWS 存取入口網站工作階段。
- 具有設定AWSconfig檔的共用檔案，其中包含可從中參照的一組組態值 AWS CDK。[default]若要尋找此檔案的位置，請參閱 AWS SDK 和工具參考指南中的共用檔案位置。
- 共用config檔案會region設定設定。這會設定 AWS 要求 AWS 區域 使 AWS CDK 用的預設值。
- AWS CDK 使用設定檔的 SSO 權杖提供者組態，在傳送要求之前取得認證 AWS。這個sso_role_name值是連接到 IAM 身分中心權限集的 IAM 角色，應該允許存取應用程式中 AWS 服務 使用的角色。

下列範例config檔案顯示使用 SSO 權杖提供者組態設定的預設設定檔。設定檔的sso_session設定是指具名sso-session區段。此sso-session區段包含用來啟動 AWS 存取入口網站工作階段的設定。

```
[default]
sso_session = my-sso
sso_account_id = 111122223333
sso_role_name = SampleRole
region = us-east-1
output = json

[sso-session my-sso]
sso_region = us-east-1
sso_start_url = https://provided-domain.awsapps.com/start
sso_registration_scopes = sso:account:access
```

啟動 AWS 存取入口網站會話

在存取之前 AWS 服務，您需要使用有效的存 AWS 取入口網站工作階段， AWS CDK 才能使用 IAM 身分中心驗證來解析登入資料。根據您配置的會話長度，您的訪問最終將過期，並且 AWS CDK 將遇到身份驗證錯誤。在中執行下列命令 AWS CLI 以登入 AWS 存取入口網站。

```
aws sso login
```

如果您的 SSO 權杖提供者組態使用具名的設定檔而非預設設定檔，則命令為aws sso login --profile *NAME*。使用--profile選項或AWS_PROFILE環境變數發出指cdk令時，也請指定此設定檔。

若要測試您是否已有作用中的工作階段，請執行下列 AWS CLI 命令。

```
aws sts get-caller-identity
```

對此命令的回應，應報告共用 config 檔案中設定的 IAM Identity Center 帳戶和許可集合。

> **Note**
>
> 如果您已經擁有作用中的 AWS 存取入口網站工作階段並執行aws sso login，則不需要提供認證。

登入程序可能會提示您允許 AWS CLI 存取您的資料。由 AWS CLI 於建置在適用於 Python 的 SDK 之上，因此權限訊息可能包含`botocore`名稱的變體。

步驟 3：安裝 AWS CDKCLI

使用以下 Package 管理器命令 AWS CDK CLI全局安Node裝。

```
npm install -g aws-cdk
```

> **Note**
>
> 如果您收到權限錯誤，並具有系統管理員訪問權限，請嘗試`sudo npm install -g aws-cdk`。

執行下列命令以確認安裝成功。 AWS CDK CLI應該輸出版本號：

```
cdk --version
```

如果您收到錯誤訊息，請嘗試執行下列命令 AWS CDK CLI來解除安裝：

```
npm uninstall -g aws-cdk
```

然後，重複步驟以重新安裝 AWS CDK CLI.

如果仍然收到錯誤訊息，請從目前專案和全域`node-modules`資料夾中刪除該`node-modules`資料夾。若要找到此資料夾，請執行`npm config get prefix`。

AWS CDK CLI會從您在先前步驟中設定的來源取得安全認證。

> **Note**
>
> CDK 工具包 v2 適用於現有的 CDK v1 項目。但是，它無法初始化新的 CDK v1 專案。看看你是the section called “新先決條件”否需要能夠做到這一點。

步驟 4：引導您的環境

您計劃部署資源的每個 AWS 環境都必須啟動載入。

若要啟動程序，請執行下列命令：

```
cdk bootstrap aws://ACCOUNT-NUMBER/REGION
```

> **Tip**
>
> 如果您沒有 AWS 帳號方便使用，可以從 AWS Management Console. 或者，如果您已 AWS CLI 安裝，下列指令會顯示您的預設帳戶資訊，包括帳號。
>
> ```
> aws sts get-caller-identity
> ```
>
> 如果您在本機 AWS 組態中建立了命名的設定檔，您可以使用`--profile`此選項來顯示特定設定檔的帳戶資訊。下列範例顯示如何顯示 prod 設定檔的帳戶資訊。
>
> ```
> aws sts get-caller-identity --profile prod
> ```
>
> 若要顯示預設「區域」，請使用`aws configure get`。
>
> ```
> aws configure get region
> aws configure get region --profile prod
> ```

可選 AWS CDK 工具

這AWS Toolkit for Visual Studio Code是適用於 Visual Studio 程式碼的開放原始碼外掛程式，可協助您在上建立、偵錯和部署應用程式 AWS。此工具組提供開發 AWS CDK 應用程式的整合體驗。它包括 AWS CDK 資源管理器功能列出您的 AWS CDK 項目並瀏覽 CDK 應用程序的各種組件。安裝外掛程式並進一步瞭解如何使用 AWS CDK 檔案總管。

後續步驟

現在您已經安裝了 AWS CDK CLI，請使用它來構建您的第一個 AWS CDK 應用程序。

若要進一步瞭解如何使用偏好 AWS CDK 的程式設計語言，請參閱使用支援的 AWS CDK 程式設計語言。

AWS CDK 這是一個開源項目。若要貢獻，請參閱貢獻給 AWS Cloud Development Kit (AWS CDK).

進一步了解

若要進一步了解 AWS CDK，請參閱下列內容：

- CDK 工作坊 — 深入實踐工作坊。
- API 參考 — 探索您將使用的 AWS 服務 可用建構。
- 構建中心 — 從 CDK 社區中查找構造。
- AWS CDK 範例 — 探索 AWS CDK 專案的程式碼範例。

您的第一個 AWS CDK 應用

AWS Cloud Development Kit (AWS CDK) 通過構建您的第一個 CDK 應用程序開始使用。

在開始本自學課程之前，我們建議您完成下列步驟：

- 什麼是 AWS CDK？如需「」的簡介，請參閱 AWS CDK。
- 請參AWS CDK 概念閱以瞭解的核心概念 AWS CDK。
- 請瀏覽先決條件和 AWS CDK 設定步驟，位於開始使用 AWS CDK。

主題

- 後續步驟

關於本教學

在本教程中，您將創建和部署一個簡單的應用程 AWS CDK 序。此應用程序包含一個具有單個 Amazon Simple Storage Service (Amazon S3) 存儲桶資源的堆棧。通過本教程，您將學習以下內容：

- 專案的結 AWS CDK 構。
- 如何創建一個 AWS CDK 應用程序。
- 如何使用「 AWS 建構程式庫」來定義應用程式、堆疊和 AWS 資源。
- 如何使用 CDK CLI 來合成、比較、部署和刪除您的 CDK 應用程式。
- 如何修改和重新部署 CDK 應用程序以更新已部署的資源。

標準 AWS CDK 開發工作流程包含下列步驟：

1. 建立您的 AWS CDK 應用程式 — 在這裡，您將使用提供的範本 AWS CDK CLI。
2. 定義您的堆疊和資源 — 使用建構來定義應用程式中的堆疊和 AWS 資源。
3. 建立您的應用程式 — 此步驟為選用步驟。如有必要， AWS CDK CLI會自動執行此步驟。建議執行此步驟以識別語法和類型錯誤。
4. 合成堆疊 — 此步驟會為應用程式中的每個堆疊建立 AWS CloudFormation 範本。此步驟對於識別定義的 AWS 資源中的邏輯錯誤非常有用。
5. 部署您的應用程式 — 使用佈建資源部署 AWS CloudFormation 至您的 AWS 環境。在部署過程中，您將識別應用程序的任何權限問題。

通過典型的工作流程，您將返回並重複先前的步驟來修改或調試您的應用程序。

我們建議您對 AWS CDK 專案使用版本控制。

步驟 1：建立應用程式

CDK 應用程序應該位於自己的目錄中，並具有自己的本地模塊依賴關係。在您的開發機器上，創建一個新目錄。以下是建立新`hello-cdk`目錄的範例：

```
$ mkdir hello-cdk
$ cd hello-cdk
```

> **Important**
>
> 一定要命名你的項目目錄`hello-cdk`，完全如下所示。 AWS CDK 專案範本會使用目錄名稱來命名產生的程式碼中的項目。如果您使用不同的名稱，本教學課程中的程式碼將無法運作。

接下來，從您的新目錄中，使用cdk init命令初始化應用程序。使用`--language`選項指定app範本和您偏好的程式設計語言。以下是範例：

TypeScript

```
cdk init app --language typescript
```

JavaScript

```
cdk init app --language javascript
```

Python

```
cdk init app --language python
```

創建應用程序後，還要輸入以下兩個命令。這些激活應用程序的 Python 虛擬環境並安裝 AWS CDK 核心依賴項。

```
source .venv/bin/activate
python -m pip install -r requirements.txt
```

Java

```
cdk init app --language java
```

如果您使用的是 IDE，您現在可以開啟或匯入專案。例如，在 Eclipse 中，選擇「檔案 > 匯入 > Maven > 現有的 Maven 專案」。請確定專案設定已設定為使用 Java 8 (1.8)。

C#

```
cdk init app --language csharp
```

如果您使用的是 Visual Studio，請在`src`目錄中開啟解決方案檔案。

Go

```
cdk init app --language go
```

建立應用程式之後，也請輸入下列指令來安裝應用程式所需的「 AWS 建構程式庫」模組。

```
go get
```

該cdk init命令會在hello-cdk目錄中創建許多文件和文件夾，以幫助您組織 AWS CDK 應用程序的源代碼。總的來說，這就是所謂的 AWS CDK 項目。花點時間探索 CDK 項目。

如果您已Git安裝，則使用建立的每個專案cdk init也會初始化為Git儲存庫。

步驟 2：建置應用程式

在大多數程式設計環境中，您可以在進行變更後建置或編譯程式碼 這不是必要的， AWS CDK 因為 CDK CLI 會自動執行此步驟。不過，當您想要 catch 語法並輸入錯誤時，您仍然可以手動建置。以下是範例：

TypeScript

```
npm run build
```

JavaScript

不需要建置步驟。

Python

不需要建置步驟。

Java

```
mvn compile -q
```

或者在日食中按控制-B（其他 Java IDE 可能會有所不同）

C#

```
dotnet build src
```

或者在視覺工作室中按 F6

Go

```
go build
```

步驟 3：在應用程序中列出堆棧

通過在應用中列出堆棧來驗證您的應用程序已正確創建。執行下列命令：

```
cdk ls
```

應該會顯示輸出HelloCdkStack。如果您沒有看到此輸出，請確認您位於專案的正確工作目錄中，然後再試一次。如果您仍然看不到堆疊，請重複the section called “步驟 1：建立應用程式”並再試一次。

第 4 步：添加一個 Amazon S3 存儲桶

此時，您的 CDK 應用程序包含一個堆棧。接下來，您將在堆疊中定義 Amazon Simple Storage Service (Amazon S3) 儲存貯體資源。若要這麼做，您將從「建構資料庫」匯入並使用 `Bucket` L2 AWS 建構。

透過匯入Bucket建構並定義 Amazon S3 儲存貯體資源來修改您的 CDK 應用程式。以下是範例：

TypeScript

在 `lib/hello-cdk-stack.ts` 中：

```
import * as cdk from 'aws-cdk-lib';
import { aws_s3 as s3 } from 'aws-cdk-lib';

export class HelloCdkStack extends cdk.Stack {
  constructor(scope: cdk.App, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    new s3.Bucket(this, 'MyFirstBucket', {
      versioned: true
    });
  }
}
```

JavaScript

在 `lib/hello-cdk-stack.js` 中：

```
const cdk = require('aws-cdk-lib');
const s3 = require('aws-cdk-lib/aws-s3');

class HelloCdkStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    new s3.Bucket(this, 'MyFirstBucket', {
      versioned: true
    });
  }
}

module.exports = { HelloCdkStack }
```

Python

在 hello_cdk/hello_cdk_stack.py 中：

```
import aws_cdk as cdk
import aws_cdk.aws_s3 as s3

class HelloCdkStack(cdk.Stack):

    def __init__(self, scope: cdk.App, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        bucket = s3.Bucket(self, "MyFirstBucket", versioned=True)
```

Java

在 src/main/java/com/myorg/HelloCdkStack.java 中：

```
package com.myorg;

import software.amazon.awscdk.*;
import software.amazon.awscdk.services.s3.Bucket;

public class HelloCdkStack extends Stack {
    public HelloCdkStack(final App scope, final String id) {
        this(scope, id, null);
    }
```

```
    public HelloCdkStack(final App scope, final String id, final StackProps props) {
        super(scope, id, props);

        Bucket.Builder.create(this, "MyFirstBucket")
            .versioned(true).build();
    }
}
```

C#

在 src/HelloCdk/HelloCdkStack.cs 中：

```
using Amazon.CDK;
using Amazon.CDK.AWS.S3;

namespace HelloCdk
{
    public class HelloCdkStack : Stack
    {
        public HelloCdkStack(App scope, string id, IStackProps props=null) :
 base(scope, id, props)
        {
            new Bucket(this, "MyFirstBucket", new BucketProps
            {
                Versioned = true
            });
        }
    }
}
```

Go

在 hello-cdk.go 中：

```
package main

import (
 "github.com/aws/aws-cdk-go/awscdk/v2"
 "github.com/aws/aws-cdk-go/awscdk/v2/awss3"
 "github.com/aws/constructs-go/constructs/v10"
 "github.com/aws/jsii-runtime-go"
)
```

```
type HelloCdkStackProps struct {
 awscdk.StackProps
}

func NewHelloCdkStack(scope constructs.Construct, id string, props
 *HelloCdkStackProps) awscdk.Stack {
 var sprops awscdk.StackProps
 if props != nil {
  sprops = props.StackProps
 }
 stack := awscdk.NewStack(scope, &id, &sprops)

 awss3.NewBucket(stack, jsii.String("MyFirstBucket"), &awss3.BucketProps{
  Versioned: jsii.Bool(true),
 })

 return stack
}

func main() {
 defer jsii.Close()

 app := awscdk.NewApp(nil)

 NewHelloCdkStack(app, "HelloCdkStack", &HelloCdkStackProps{
  awscdk.StackProps{
   Env: env(),
  },
 })

 app.Synth(nil)
}

func env() *awscdk.Environment {
 return nil
}
```

讓我們來看看Bucket構造仔細看看。像所有的構造，Bucket類有三個參數：

- scope — 將Stack類別定義為Bucket建構的父項。定義 AWS 資源的所有構造都在堆棧的範圍內創建。您可以在構造內定義構造，創建一個層次結構（樹）。在這裡，在大多數情況下，範圍是this（selfinPython）。

- Id — 應 AWS CDK 用程式`Bucket`內的邏輯 ID。此 ID 加上以堆疊中儲存貯體位置為基礎的雜湊，可在部署期間唯一識別值區。當您在應用程序中更新構造並重新部署以更新已部署的資源時， AWS CDK 還會引用此 ID。在這裡，您的邏輯 ID 是`MyFirstBucket`。值區也可以具有使用`bucketName`屬性指定的名稱。這與邏輯 ID 不同。
- prop — 定義值區屬性的值組合。在這裡，您將`versioned`屬性定義為`true`，以便為值區中的檔案啟用版本控制。

  Prop 在支援的語言中以不同的方式表示 AWS CDK。

  - in TypeScript and `props` 是一個參數 JavaScript，並且您傳入包含所需屬性的對象。
  - 在 Python 中，道具作為關鍵字參數傳遞。
  - 在 Java 中，提供了一個生成器來傳遞道具。有兩個：一個 forBucketProps，第二個用於`Bucket`讓您在一步中構建構體及其 prop 對象。此代碼使用後者。
  - 在 C# 中，您可以使用`BucketProps`對象初始化器實例化一個對象，並將其作為第三個參數傳遞。

  如果構造的 prop 是可選的，則可以完全省略`props`參數。

所有結構都採用相同的三個參數，因此當您了解新結構時，很容易保持面向。正如你所期望的那樣，你可以子類任何構造來擴展它以滿足您的需求，或者如果你想改變它的默認值。

步驟 5：合成範本 AWS CloudFormation

合成應用程式的 AWS CloudFormation 範本，如下所示：

```
cdk synth
```

如果您的應用程序包含多個堆棧，則必須指定要合成的堆棧。由於您的應用程序包含單個堆棧，因此 CDK CLI 會自動檢測要合成的堆棧。

如果您未執行`cdk synth`，CDK CLI 會在您部署時自動執行此步驟。不過，我們建議您在每次部署之前執行此步驟。

> **Tip**
>
> 如果您收到類似的錯誤訊息`--app is required ...`，請檢查執行 CDK CLI 指令的目錄。你應該在你的主應用程序目錄。

該`cdk synth`命令運行您的應用程序。這將為您的應用程序中的每個堆棧創建一個 AWS CloudFormation 模板。CDK CLI 會在命令列中顯示範本的 YAML 格式版本，並在目錄中儲存範本的 JSON 格式版本。`cdk.out`以下是命令行輸出的片段，其中顯示了在 AWS CloudFormation 模板中定義的存儲桶：

```
Resources:
  MyFirstBucketB8884501:
    Type: AWS::S3::Bucket
    Properties:
      VersioningConfiguration:
        Status: Enabled
    UpdateReplacePolicy: Retain
    DeletionPolicy: Retain
    Metadata:...
```

> **Note**
>
> 依預設，每個產生的範本都包含一個`AWS::CDK::Metadata`資源。該 AWS CDK 團隊使用此元數據來深入了解使 AWS CDK 用情況並找到改進的方法。如需詳細資訊，包括如何選擇退出版本報告，請參閱版本報告。

產生的範本可透過 AWS CloudFormation 主控台或任何 AWS CloudFormation 部署工具進行部署。您也可以使用 CDK CLI 進行部署。在下一個步驟中，您可以使用 CDK CLI 進行部署。

步驟 6：部署您的堆疊

若要將 CDK 堆疊部署到 AWS CloudFormation 使用 CDKCLI，請執行下列命令：

```
cdk deploy
```

> **Important**
>
> 在部署之前，您必須執行 AWS 環境的一次性啟動載入。如需指示，請參閱啟動您的環境。

類似於`cdk synth`，您不必指定 AWS CDK 堆棧，因為應用程序包含單個堆棧。

如果您的代碼存在安全隱患，CDK CLI 將輸出摘要。您將需要確認它們才能繼續部署。本教程中的應用程序沒有這些含義。

執行之後`cdk deploy`，CDK CLI 會在部署堆疊時顯示進度資訊。完成後，您可以前往主AWS CloudFormation 控台檢視您的`HelloCdkStack`堆疊。您也可以前往 Amazon S3 主控台檢視您的`MyFirstBucket`資源。

恭喜您！您已使用 AWS CDK. 接下來，您將修改您的應用程序並重新部署以更新資源。

步驟 7：修改您的應用程式

在此步驟中，您將修改 Amazon S3 儲存貯體，方法是將其設定為在刪除堆疊時自動刪除。此修改涉及變更值區的`RemovalPolicy`屬性。您還將配置`autoDeleteObjects`屬性以配置 CDK CLI 以在銷毀值區之前從值區中刪除對象。依預設， AWS CloudFormation 不會刪除包含物件的 Amazon S3 儲存貯體。

請使用下列範例修改資源：

TypeScript

更新 `lib/hello-cdk-stack.ts`。

```
new s3.Bucket(this, 'MyFirstBucket', {
  versioned: true,
  removalPolicy: cdk.RemovalPolicy.DESTROY,
  autoDeleteObjects: true
});
```

JavaScript

更新 `lib/hello-cdk-stack.js`。

```
new s3.Bucket(this, 'MyFirstBucket', {
  versioned: true,
  removalPolicy: cdk.RemovalPolicy.DESTROY,
  autoDeleteObjects: true
});
```

Python

更新 `hello_cdk/hello_cdk_stack.py`。

```
bucket = s3.Bucket(self, "MyFirstBucket",
    versioned=True,
```

```
    removal_policy=cdk.RemovalPolicy.DESTROY,
    auto_delete_objects=True)
```

Java

更新 src/main/java/com/myorg/HelloCdkStack.java。

```
Bucket.Builder.create(this, "MyFirstBucket")
        .versioned(true)
        .removalPolicy(RemovalPolicy.DESTROY)
        .autoDeleteObjects(true)
        .build();
```

C#

更新 src/HelloCdk/HelloCdkStack.cs。

```
new Bucket(this, "MyFirstBucket", new BucketProps
{
    Versioned = true,
    RemovalPolicy = RemovalPolicy.DESTROY,
    AutoDeleteObjects = true
});
```

Go

更新 hello-cdk.go。

```
   awss3.NewBucket(stack, jsii.String("MyFirstBucket"), &awss3.BucketProps{
  Versioned:         jsii.Bool(true),
  RemovalPolicy:     awscdk.RemovalPolicy_DESTROY,
  AutoDeleteObjects: jsii.Bool(true),
 })
```

目前，您的程式碼變更尚未對部署的 Amazon S3 儲存貯體資源進行任何直接更新。您的代碼定義了資源的所需狀態。若要修改已部署的資源，您將使用 CDK CLI 將所需的狀態合成為新 AWS CloudFormation 範本。然後，您會將新 AWS CloudFormation 範本部署為變更集。變更集只會進行必要的變更，以達到新的所需狀態。

若要查看這些變更，請使用cdk diff指令。執行下列命令：

```
cdk diff
```

CDK 會CLI查詢您的 AWS 帳戶 帳戶以取得HelloCdkStack堆疊的最新 AWS CloudFormation 範本。然後，它將最新模板與剛從您的應用程序合成的模板進行比較。輸出應如下所示。

```
Stack HelloCdkStack
IAM Statement Changes
###############################################################################################
#   # Resource                     # Effect # Action                      # Principal
                  # Condition #
###############################################################################################
# + # ${Custom::S3AutoDeleteObject # Allow  # sts:AssumeRole              #
 Service:lambda.amazonaws.com #           #
#   # sCustomResourceProvider/Role #        #                            #
                  #           #
#   # .Arn}                        #        #                            #
                  #           #
###############################################################################################
# + # ${MyFirstBucket.Arn}         # Allow  # s3:DeleteObject*            # AWS:
${Custom::S3AutoDeleteOb #           #
#   # ${MyFirstBucket.Arn}/*       #        # s3:GetBucket*               #
 jectsCustomResourceProvider/ #           #
#   #                              #        # s3:GetObject*               # Role.Arn}
                  #           #
#   #                              #        # s3:List*                    #
                  #           #
###############################################################################################
IAM Policy Changes
###############################################################################################
#   # Resource                                                # Managed Policy ARN
                               #
###############################################################################################
# + # ${Custom::S3AutoDeleteObjectsCustomResourceProvider/Ro # {"Fn::Sub":"arn:
${AWS::Partition}:iam::aws:policy/serv #
#   # le}                                                     # ice-role/
AWSLambdaBasicExecutionRole"}                 #
###############################################################################################
(NOTE: There may be security-related changes not in this list. See https://github.com/
aws/aws-cdk/issues/1299)

Parameters
[+] Parameter
 AssetParameters/4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392/
```

```
S3Bucket
 AssetParameters4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392S3BucketBF7A7F3
 {"Type":"String","Description":"S3 bucket for asset
 \"4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392\""}
[+] Parameter
 AssetParameters/4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392/
S3VersionKey
 AssetParameters4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392S3VersionKeyFAF
 {"Type":"String","Description":"S3 key for asset version
 \"4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392\""}
[+] Parameter
 AssetParameters/4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392/
ArtifactHash
 AssetParameters4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392ArtifactHashE56
 {"Type":"String","Description":"Artifact hash for asset
 \"4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392\""}

Resources
[+] AWS::S3::BucketPolicy MyFirstBucket/Policy MyFirstBucketPolicy3243DEFD
[+] Custom::S3AutoDeleteObjects MyFirstBucket/AutoDeleteObjectsCustomResource
 MyFirstBucketAutoDeleteObjectsCustomResourceC52FCF6E
[+] AWS::IAM::Role Custom::S3AutoDeleteObjectsCustomResourceProvider/Role
 CustomS3AutoDeleteObjectsCustomResourceProviderRole3B1BD092
[+] AWS::Lambda::Function Custom::S3AutoDeleteObjectsCustomResourceProvider/Handler
 CustomS3AutoDeleteObjectsCustomResourceProviderHandler9D90184F
[~] AWS::S3::Bucket MyFirstBucket MyFirstBucketB8884501
 ## [~] DeletionPolicy
 #   ## [-] Retain
 #   ## [+] Delete
 ## [~] UpdateReplacePolicy
     ## [-] Retain
     ## [+] Delete
```

這個差異有四個部分：

- IAM 陳述式變更和 IAM 政策變更 — 由於您在 Amazon S3 儲存貯體上設定了`AutoDeleteObjects`屬性，因此存在這些許可變更。自動刪除功能會使用自訂資源來刪除值區中的物件，然後再刪除值區本身。IAM 物件會將自訂資源的程式碼存取權授予儲存貯體。
- 參數 — AWS CDK 使用這些項目來尋找自訂資源的 AWS Lambda 函數資產。
- 資源 — 此堆疊中的新資源和已變更的資源。我們可以看到前面提到的 IAM 對象，自定義資源以及其關聯的 Lambda 函數正在添加。我們還可以看到存儲桶

的DeletionPolicy和UpdateReplacePolicy屬性正在更新。這些允許存儲桶與堆棧一起刪除，並用新的替換。

您可能會注意到我們RemovalPolicy在 AWS CDK 應用程序中指定，但在生成的 AWS CloudFormation 模板中獲得了DeletionPolicy屬性。這是因為屬性 AWS CDK 使用不同的名稱。AWS CDK 預設值是在刪除堆疊時保留值區，而 AWS CloudFormation 預設值是將其刪除。如需詳細資訊，請參閱 the section called “移除政策”。

要查看您的新 AWS CloudFormation 模板，您可以運行cdk synth。通過對 CDK 應用程序進行一些更改，與原始 AWS CloudFormation AWS CloudFormation 模板相比，您的新模板現在包含了許多其他代碼行。

接下來，通過運行以下命令部署您的應用程序：

```
cdk deploy
```

AWS CDK 將通知您有關我們在差異中已經看到的安全策略更改。輸入y以核准變更並部署更新的堆疊。CDK CLI 會部署您的堆疊以進行所需的變更。下面是一個示例輸出：

```
HelloCdkStack: deploying...
[0%] start: Publishing
 4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392:current
[100%] success: Published
 4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392:current
HelloCdkStack: creating CloudFormation changeset...
 0/5 | 4:32:31 PM | UPDATE_IN_PROGRESS   | AWS::CloudFormation::Stack  | HelloCdkStack
 User Initiated
 0/5 | 4:32:36 PM | CREATE_IN_PROGRESS   | AWS::IAM::Role
       | Custom::S3AutoDeleteObjectsCustomResourceProvider/Role
 (CustomS3AutoDeleteObjectsCustomResourceProviderRole3B1BD092)
 1/5 | 4:32:36 PM | UPDATE_COMPLETE      | AWS::S3::Bucket             | MyFirstBucket
 (MyFirstBucketB8884501)
 1/5 | 4:32:36 PM | CREATE_IN_PROGRESS   | AWS::IAM::Role
       | Custom::S3AutoDeleteObjectsCustomResourceProvider/Role
 (CustomS3AutoDeleteObjectsCustomResourceProviderRole3B1BD092) Resource creation
 Initiated
 3/5 | 4:32:54 PM | CREATE_COMPLETE      | AWS::IAM::Role
       | Custom::S3AutoDeleteObjectsCustomResourceProvider/Role
 (CustomS3AutoDeleteObjectsCustomResourceProviderRole3B1BD092)
```

```
 3/5 | 4:32:56 PM | CREATE_IN_PROGRESS   | AWS::Lambda::Function
      | Custom::S3AutoDeleteObjectsCustomResourceProvider/Handler
 (CustomS3AutoDeleteObjectsCustomResourceProviderHandler9D90184F)
 3/5 | 4:32:56 PM | CREATE_IN_PROGRESS   | AWS::S3::BucketPolicy       | MyFirstBucket/
Policy (MyFirstBucketPolicy3243DEFD)
 3/5 | 4:32:56 PM | CREATE_IN_PROGRESS   | AWS::Lambda::Function
      | Custom::S3AutoDeleteObjectsCustomResourceProvider/Handler
 (CustomS3AutoDeleteObjectsCustomResourceProviderHandler9D90184F) Resource creation
 Initiated
 3/5 | 4:32:57 PM | CREATE_COMPLETE      | AWS::Lambda::Function
      | Custom::S3AutoDeleteObjectsCustomResourceProvider/Handler
 (CustomS3AutoDeleteObjectsCustomResourceProviderHandler9D90184F)
 3/5 | 4:32:57 PM | CREATE_IN_PROGRESS   | AWS::S3::BucketPolicy       | MyFirstBucket/
Policy (MyFirstBucketPolicy3243DEFD) Resource creation Initiated
 4/5 | 4:32:57 PM | CREATE_COMPLETE      | AWS::S3::BucketPolicy       | MyFirstBucket/
Policy (MyFirstBucketPolicy3243DEFD)
 4/5 | 4:32:59 PM | CREATE_IN_PROGRESS   | Custom::S3AutoDeleteObjects
 | MyFirstBucket/AutoDeleteObjectsCustomResource/Default
 (MyFirstBucketAutoDeleteObjectsCustomResourceC52FCF6E)
 5/5 | 4:33:06 PM | CREATE_IN_PROGRESS   | Custom::S3AutoDeleteObjects
 | MyFirstBucket/AutoDeleteObjectsCustomResource/Default
 (MyFirstBucketAutoDeleteObjectsCustomResourceC52FCF6E) Resource creation Initiated
 5/5 | 4:33:06 PM | CREATE_COMPLETE      | Custom::S3AutoDeleteObjects
 | MyFirstBucket/AutoDeleteObjectsCustomResource/Default
 (MyFirstBucketAutoDeleteObjectsCustomResourceC52FCF6E)
 5/5 | 4:33:08 PM | UPDATE_COMPLETE_CLEA | AWS::CloudFormation::Stack  | HelloCdkStack
 6/5 | 4:33:09 PM | UPDATE_COMPLETE      | AWS::CloudFormation::Stack  | HelloCdkStack

 #  HelloCdkStack

Stack ARN:
arn:aws:cloudformation:REGION:ACCOUNT:stack/HelloCdkStack/UNIQUE-ID
```

第 8 步：銷毀應用程序的資源

現在您已經完成此教學課程，您可以刪除已部署的 AWS CloudFormation 堆疊及與其相關聯的所有資源。這是最好的做法，可以將不必要的成本降至最低並保持環境清潔。執行下列命令：

```
cdk destroy
```

輸入y以核准變更並刪除堆疊。

> **Note**
>
> 如果您沒有更改存儲桶`RemovalPolicy`，堆棧刪除將成功完成，但存儲桶將變為孤立（不再與堆棧關聯）。

後續步驟

恭喜您！您已完成本教學課程，並已使 AWS CDK 用成功建立、修改和刪除中的資源 AWS 雲端。您現在可以開始使用 AWS CDK.

若要進一步瞭解如何使用偏好 AWS CDK 的程式設計語言，請參閱使用支援的 AWS CDK 程式設計語言。

如需其他資源，請參閱下列內容：

- 嘗試 CDK 工作坊，進行更深入的導覽，涉及更複雜的項目。
- 深入了解the section called “環境”、、、the section called “資產”the section called “許可”the section called “Context”the section called “參數”、和等概念the section called “自訂建構”。
- 請參閱 API 參考資料，開始探索適用於您最愛 AWS 服務的 CDK 結構。
- 訪問構建中心以發現由 AWS 和其他人創建的構造。
- 探索使用 AWS CDK.

AWS CDK 這是一個開源項目。若要貢獻，請參閱貢獻 AWS Cloud Development Kit (AWS CDK)。

從 AWS CDK 第 1 版遷移到第 AWS CDK 2 版

的第 2 版 AWS Cloud Development Kit (AWS CDK) 旨在讓您以偏好的程式設計語言將基礎結構寫成程式碼變得更加容易。本主題說明的 v1 和 v2 之間的變更 AWS CDK。

> **Tip**
>
> 若要識別使用 AWS CDK v1 部署的堆疊，請使用 awscd k-v1 堆疊尋找工具公用程式。

從 AWS CDK v1 到 CDK V2 的主要變化如下。

- AWS CDK v2 將 AWS 構造庫的穩定部分（包括核心庫）合併到單個包中`aws-cdk-lib`。開發人員不再需要為他們使用的個別 AWS 服務安裝其他套件。這種單一封裝方法也表示您不需要同步處理各種 CDK 程式庫套件的版本。

  代表中確切可用資源的 L1 (CFnxxxx) 建構一律被視為穩定 AWS CloudFormation，因此包含在中。`aws-cdk-lib`
- 我們仍在與社群合作以開發新的 L2 或 L3 建構的實驗模組，並未包含在其中。`aws-cdk-lib`相反地，它們會以個別套件的形式散發。實驗性套件會以`alpha`字尾和語意版本號碼命名。語義版本號與它們兼容的 AWS 構造庫的第一個版本匹配，也帶有`alpha`後綴。結構在被指定為穩定`aws-cdk-lib`之後移入，允許主構造庫遵守嚴格的語義版本控制。

  穩定性是在服務層級指定的。例如，如果我們開始為 Amazon 創建一個或多個 L2 構造 AppFlow，這在撰寫本文中只有 L1 構造，它們首先出現在一個名為的模塊中。`@aws-cdk/aws-appflow-alpha`然後，`aws-cdk-lib`當我們覺得新結構滿足客戶的基本需求時，他們就會轉移到。

  一旦一個模塊已被指定為穩定和納入`aws-cdk-lib`, 新的 API 使用在下一個 bullet 中描述的「BetAN」約定添加.

  每個實驗模組的新版本都會隨著 AWS CDK. 但是，在大多數情況下，它們不需要保持同步。您可以隨時升級`aws-cdk-lib`或實驗模塊。例外是，當兩個或多個相關的實驗模塊相互依賴時，它們必須是相同的版本。
- 對於正在添加新功能的穩定模塊，新的 API（無論是全新的構造還是現有構造上的新方法或屬性）在工作進行中時都會收到`Beta1`後綴。(在需要`Beta2`中斷變更時，接著是`Beta3`、等。) 當 API 指定為穩定時，會新增不含尾碼的 API 版本。然後，除了最新的（無論是測試版還是 final）之外的所有方法都被棄用。

例如，如果我們添加一個新的方法`grantPower()`到一個構造，它最初顯示為`grantPowerBeta1()`。如果需要中斷變更 (例如，新的必要參數或屬性)，則會命名方法的下一個版本`grantPowerBeta2()`，依此類推。當工作完成並完成 API 時，將添加該方法`grantPower()`（沒有後綴），並且 BetAN 方法將被棄用。

所有測試版 API 都會保留在構造庫中，直到下一個主要版本（3.0）發布為止，並且它們的簽名不會更改。如果您使用它們，您將看到棄用警告，因此您應該儘早移至 API 的最終版本。但是，future 的 AWS CDK 2.x 版本不會破壞您的應用程式。

- 已將類`Construct`別與相關型別一起從萃取 AWS CDK 到單獨的程式庫中。這樣做是為了支持將「構造編程模型」應用於其他領域的努力。如果您正在編寫自己的構造或使用相關的 API，則必須將`constructs`模塊聲明為依賴項，並對導入進行較小的更改。如果您正在使用高級功能（例如連接到 CDK 應用程序生命週期），則可能需要更多更改。如需完整詳細資訊，請參閱 RFC。
- AWS CDK v1.x 及其建構程式庫中已過時的屬性、方法和類型已從 CDK v2 API 中完全移除。在大多數支援的語言中，這些 API 會在 v1.x 版下產生警告，因此您可能已移轉至替代 API。有關 CDK v1.x 中已淘汰的 API 的完整清單，請參閱上。 GitHub
- 在 CDK v2 中預設會啟用 AWS CDK v1.x 中的功能旗標所控制的行為。不再需要較早的功能標誌，並且在大多數情況下不支持它們。在非常特定的情況下，還有一些可以讓您恢復到 CDK v1 行為。如需詳細資訊，請參閱 the section called “更新功能旗標”。
- 使用 CDK v2 時，您部署到的環境必須使用現代的啟動程序堆疊啟動載入。不再支援舊版啟動程序堆疊 (v1 下的預設值)。CDK v2 此外還需要現代堆棧的新版本。若要升級您現有的環境，請重新啟動它們。不再需要設置任何功能標誌或環境變量來使用現代引導堆棧。

> **Important**
>
> 現代啟動程序範本有效地授予`--trust`清單中任`--cloudformation-execution-policies`何 AWS 帳戶所隱含的權限。默認情況下，這將擴展讀取和寫入啟動載入帳戶中的任何資源的權限。確保使用您熟悉的策略和受信任的帳戶配置啟動載入堆棧。

新先決條件

AWS CDK v2 的大多數需求與 AWS CDK v1.x 的要求相同。此處列出其他需求。

- 對於 TypeScript 開發人員，需要 TypeScript 3.8 或更高版本。

- CDK 工具組的新版本需要與 CDK v2 搭配使用。現在 CDK v2 已正式推出，v2 是安裝 CDK 工具組時的預設版本。它與 CDK v1 項目向後兼容，因此除非要創建 CDK v1 項目，否則您不需要保持安裝早期版本。要升級，請發出問題`npm install -g aws-cdk`。

從 AWS CDK v2 開發人員預覽升級

如果您使用的是 CDK v2 開發人員預覽版，則您的專案在候選發行版本上具有相依性 AWS CDK，例如`2.0.0-rc1`. 將這些更新為`2.0.0`，然後更新項目中安裝的模塊。

TypeScript

`npm install` 或 `yarn install`

JavaScript

`npm install` 或 `yarn install`

Python

```
python -m pip install -r requirements.txt
```

Java

```
mvn package
```

C#

```
dotnet restore
```

Go

```
go get
```

更新相依性之後，`npm update -g aws-cdk`請將 CDK 工具組更新為發行版本的問題。

從 AWS CDK 第 1 版遷移到 CDK V2

若要將您的應用程式遷移至 AWS CDK v2，請先更新中的功能旗標`cdk.json`。然後更新您的應用程序的依賴關係，並根據需要為其編寫的編程語言導入。

更新到最近的 v1

我們看到許多客戶在一個步驟中從舊版 AWS CDK v1 升級到最新版本的 v2。雖然當然有可能這樣做，但是您將在多年的變更中進行升級 (不幸的是，可能並非所有人都有相同數量的演進測試我們今天)，也可以跨版本升級使用新的預設值和不同的程式碼組織。

為了獲得最安全的升級體驗並更輕鬆地診斷任何未預期變更的來源，我們建議您將這兩個步驟分開：首先升級到最新的 v1 版本，然後再切換到 v2。

更新功能旗標

`cdk.json`如果存在，請從以下 v1 功能標誌中刪除，因為它們在 AWS CDK v2 中默認情況下都處於活動狀態。如果它們的舊效果對您的基礎結構很重要，您將需要進行源代碼更改。如需詳細資訊，請參閱上 GitHub的旗標清單。

- `@aws-cdk/core:enableStackNameDuplicates`
- `aws-cdk:enableDiffNoFail`
- `@aws-cdk/aws-ecr-assets:dockerIgnoreSupport`
- `@aws-cdk/aws-secretsmanager:parseOwnedSecretName`
- `@aws-cdk/aws-kms:defaultKeyPolicies`
- `@aws-cdk/aws-s3:grantWriteWithoutAcl`
- `@aws-cdk/aws-efs:defaultEncryptionAtRest`

您可以將少數 v1 功能旗標設定為，以恢復到特定的 AWS CDK v1 行為； GitHub 如需完整的參考資料，請參閱the section called “還原為 v1 行為”或上的清單。`false`

對於這兩種類型的標誌，請使用`cdk diff`命令檢查對合成範本的變更，以查看這些旗標的變更是否會影響您的基礎結構。

CDK 工具包兼容性

CDK V2 需要 CDK 工具包的 v2 或更高版本。此版本與 CDK v1 應用程式向後相容。因此，您可以在所有 AWS CDK 專案中使用單一全域安裝的 CDK Toolkit 版本，無論它們使用 v1 還是 v2。一個例外是 CDK 工具包 v2 只創建 CDK v2 項目。

如果您需要同時建立 v1 和 v2 CDK 專案，請勿在全球範圍內安裝 CDK 工具組 v2。（如果您已經安裝了它，請將其刪除：`npm remove -g aws-cdk`。）若要叫用 CDK 工具組，請視npx需要使用來執行 CDK 工具組的 v1 或 v2。

```
npx aws-cdk@1.x init app --language typescript
npx aws-cdk@2.x init app --language typescript
```

Tip

設置指令行別名，以便您可以使用cdk和cdk1指令呼叫所需版本的 CDK Toolkit。

macOS/Linux

```
alias cdk1="npx aws-cdk@1.x"
alias cdk="npx aws-cdk@2.x"
```

Windows

```
doskey cdk1=npx aws-cdk@1.x $*
doskey cdk=npx aws-cdk@2.x $*
```

更新依賴關係和導入

更新應用程式的相依性，然後安裝新套件。最後，更新代碼中的導入。

TypeScript

應用程式

對於 CDK 應用程序，更新`package.json`如下。刪除對 v1 風格的單個穩定模塊的依賴關係，並為`aws-cdk-lib`您的應用程序建立所需的最低版本（2.0.0 在這裡）。

實驗結構是在單獨的，獨立版本化的包中提供，其名稱結尾為`alpha`和 alpha 版本號。Alpha 版本號碼對應於與它們相容`aws-cdk-lib`的第一個發行版本。在這裡，我們已經固定`aws-codestar`到 2.0.0-α。

```
{
  "dependencies": {
    "aws-cdk-lib": "^2.0.0",
    "@aws-cdk/aws-codestar-alpha": "2.0.0-alpha.1",
    "constructs": "^10.0.0"
  }
```

```
}
```

建構程式庫

對於構造庫，請為aws-cdk-lib您的應用程序創建所需的最低版本（2.0.0 在這裡），並按package.json如下方式更新。

請注意，既aws-cdk-lib顯示為對等依賴關係和開發依賴關係。

```
{
  "peerDependencies": {
    "aws-cdk-lib": "^2.0.0",
    "constructs": "^10.0.0"
  },
  "devDependencies": {
    "aws-cdk-lib": "^2.0.0",
    "constructs": "^10.0.0",
    "typescript": "~3.9.0"
  }
}
```

> **Note**
>
> 在發布與 v2 兼容的庫時，您應該對庫的版本號執行主要版本升級，因為這對於庫消費者來說是一個突破性的更改。使用單一程式庫無法同時支援 CDK v1 和 v2。若要繼續支援仍在使用 v1 的客戶，您可以 parallel 維護舊版，或為 v2 建立新套件。
> 您希望持續支援 AWS CDK v1 客戶的時間取決於您。您可以從 CDK v1 本身的生命週期中獲得提示，該生命週期將於 2022 年 6 月 1 日進入維護，並將於 2023 年 6 end-of-life 月 1 日到達。如需完整詳細資訊，請參閱AWS CDK 維護政策。

資料庫和應用程式

透過執行npm install或來安裝新的相依性yarn install。

將您的匯入變更為Construct從新constructs模組匯入、核心類型 (例如App和Stack從頂層) 匯入aws-cdk-lib，以及從下命名空間使用之服務的穩定「建構程式庫」aws-cdk-lib 模組。

```
import { Construct } from 'constructs';
import { App, Stack } from 'aws-cdk-lib';                // core constructs
import { aws_s3 as s3 } from 'aws-cdk-lib';              // stable module
```

```
import * as codestar from '@aws-cdk/aws-codestar-alpha';  // experimental module
```

JavaScript

更新package.json如下。刪除對 v1 風格的單個穩定模塊的依賴關係，並為aws-cdk-lib您的應用程序建立所需的最低版本（2.0.0 在這裡）。

實驗結構是在單獨的，獨立版本化的包中提供，其名稱結尾為alpha和 alpha 版本號。Alpha 版本號碼對應於與它們相容aws-cdk-lib的第一個發行版本。在這裡，我們已經固定aws-codestar到 2.0.0-α。

```
{
  "dependencies": {
    "aws-cdk-lib": "^2.0.0",
    "@aws-cdk/aws-codestar-alpha": "2.0.0-alpha.1",
    "constructs": "^10.0.0"
  }
}
```

透過執行npm install或來安裝新的相依性yarn install。

更改應用程序的導入以執行以下操作：

- Construct從新constructs模組匯入
- 從最上層匯入核心類型Stack，例如App和 aws-cdk-lib
- 從命名空間匯入「AWS 建構程式庫」模組 aws-cdk-lib

```
const { Construct } = require('constructs');
const { App, Stack } = require('aws-cdk-lib');              // core constructs
const s3 = require('aws-cdk-lib').aws_s3;                   // stable module
const codestar = require('@aws-cdk/aws-codestar-alpha');    // experimental module
```

Python

更新requirements.txt或定install_requires義，setup.py如下所示。刪除對 v1 風格的各個穩定模塊的依賴關係。

實驗結構是在單獨的，獨立版本化的包中提供，其名稱結尾為alpha和 alpha 版本號。Alpha 版本號碼對應於與它們相容aws-cdk-lib的第一個發行版本。在這裡，我們已經固定aws-codestar到 2.0.0α。

```
install_requires=[
     "aws-cdk-lib>=2.0.0",
     "constructs>=10.0.0",
     "aws-cdk.aws-codestar-alpha>=2.0.0alpha1",
     # ...
],
```

> **Tip**
>
> 使用卸載已安裝在應用程序虛擬環境中的任何其他版本的 AWS CDK 模塊`pip uninstall`。然後使用安裝新的依賴關係`python -m pip install -r requirements.txt`。

更改應用程序的導入以執行以下操作：

- `Construct`從新`constructs`模組匯入
- 從最上層匯入核心類型`Stack`，例如`App`和 `aws_cdk`
- 從命名空間匯入「 AWS 建構程式庫」模組 `aws_cdk`

```
from constructs import Construct
from aws_cdk import App, Stack                    # core constructs
from aws_cdk import aws_s3 as s3                  # stable module
import aws_cdk.aws_codestar_alpha as codestar     # experimental module

# ...

class MyConstruct(Construct):
    # ...

class MyStack(Stack):
    # ...

s3.Bucket(...)
```

Java

在中`pom.xml`，移除穩定模組的所有`software.amazon.awscdk`相依性，並將它們取代為`software.constructs` (in`Construct`) 和`software.amazon.awscdk`.

實驗結構是在單獨的，獨立版本化的包中提供，其名稱結尾為alpha和 alpha 版本號。Alpha 版本號碼對應於與它們相容aws-cdk-lib的第一個發行版本。在這裡，我們已經固定aws-codestar到 2.0.0-α。

```
<dependency>
    <groupId>software.amazon.awscdk</groupId>
    <artifactId>aws-cdk-lib</artifactId>
    <version>2.0.0</version>
</dependency><dependency>
    <groupId>software.amazon.awscdk</groupId>
    <artifactId>code-star-alpha</artifactId>
    <version>2.0.0-alpha.1</version>
</dependency>
<dependency>
    <groupId>software.constructs</groupId>
    <artifactId>constructs</artifactId>
    <version>10.0.0</version>
</dependency>
```

通過運行安裝新的依賴關係mvn package。

變更您的程式碼以執行下列動作：

- Construct從新software.constructs資料庫匯入
- 從匯入核心類別 StackApp，例如和 software.amazon.awscdk
- 從匯入服務建構 software.amazon.awscdk.services

```
import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.App;
import software.amazon.awscdk.services.s3.Bucket;
import software.amazon.awscdk.services.codestar.alpha.GitHubRepository;
```

C#

升級 C# CDK 應用程式的相依性最直接的方法是手動編輯.csproj檔案。移除所有穩定的Amazon.CDK.*套件參考，並以Amazon.CDK.Lib和Constructs套件的參照取代它們。

實驗結構是在單獨的，獨立版本化的包中提供，其名稱結尾為alpha和 alpha 版本號。Alpha 版本號碼對應於與它們相容aws-cdk-lib的第一個發行版本。在這裡，我們已經固定aws-codestar到 2.0.0-α。

```
<PackageReference Include="Amazon.CDK.Lib" Version="2.0.0" />
<PackageReference Include="Amazon.CDK.AWS.Codestar.Alpha" Version="2.0.0-alpha.1" />
<PackageReference Include="Constructs" Version="10.0.0" />
```

通過運行安裝新的依賴關係dotnet restore。

更改源文件中的導入，如下所示。

```
using Constructs;                   // for Construct class
using Amazon.CDK;                   // for core classes like App and Stack
using Amazon.CDK.AWS.S3;            // for stable constructs like Bucket
using Amazon.CDK.Codestar.Alpha;    // for experimental constructs
```

Go

go get將您的依賴項更新到最新版本並更新項目.mod文件的問題。

部署前先測試移轉的應用程式

部署堆疊之前，請使用cdk diff來檢查資源是否有未預期的變更。不會對邏輯 ID 進行變更 (造成資源取代)。

預期的變更包括但不限於：

- CDKMetadata資源的變更。
- 已更新資產雜湊。
- 與新樣式堆疊合成有關的變更。如果您的應用程式在 v1 中使用舊版堆疊合成器，則適用。CDK v2 不支援舊版堆疊合成器。)
- 添加一個CheckBootstrapVersion規則。

非預期的變更通常不會因為本身升級至 AWS CDK v2 而造成。通常，它們是先前由功能標誌更改的不推薦使用行為的結果。這是從早於 1.85.x 的 CDK 版本進行升級的症狀。您會看到升級至最新 v1.x 版本的相同變更。您通常可以通過執行以下操作來解決此問題：

1. 將您的應用程式升級至最新的 v1.x 版
2. 移除功能旗標
3. 視需要修改您的程式碼
4. 部署
5. 升級至第 2 版

> **Note**
>
> 如果您升級的應用程式在兩階段升級後無法部署，請回報問題。

當您準備好在應用程式中部署堆疊時，請考慮先部署副本，以便對其進行測試。要做到這一點最簡單的方法是將其部署到不同的區域。不過，您也可以變更堆疊的 ID。測試後，請務必使用銷毀測試副本cdk destroy。

故障診斷

TypeScript **`'from' expected`**或匯入**`';' expected`**時發生錯誤

升級至 TypeScript 3.8 或更新版本。

運行「cdk 引導程序」

如果您看到類似下列的錯誤：

```
#  MyStack failed: Error: MyStack: SSM parameter /cdk-bootstrap/hnb659fds/version not
 found. Has the environment been bootstrapped? Please run 'cdk bootstrap' (see https://
docs.aws.amazon.com/cdk/latest/guide/bootstrapping.html)
    at CloudFormationDeployments.validateBootstrapStackVersion (.../aws-cdk/lib/api/
cloudformation-deployments.ts:323:13)
    at processTicksAndRejections (internal/process/task_queues.js:97:5)
MyStack: SSM parameter /cdk-bootstrap/hnb659fds/version not found. Has the environment
 been bootstrapped? Please run 'cdk bootstrap' (see https://docs.aws.amazon.com/cdk/
latest/guide/bootstrapping.html)
```

AWS CDK v2 需要更新的啟動程序堆疊，此外，所有 v2 部署都需要啟動程序資源。（使用 v1，您可以在沒有引導載入的情況下部署簡單堆棧。）如需完整詳細資訊，請參閱 the section called “引導”。

正在尋找 v1 堆疊

將您的 CDK 應用程式從 v1 遷移至 v2 時，您可能想要識別使用 v1 建立的已部署 AWS CloudFormation 堆疊。若要這麼做，請執行下列命令：

```
npx awscdk-v1-stack-finder
```

有關使用詳細信息，請參閱 awscdk-v1 堆棧查找器自述文件。

將現有資源和 AWS CloudFormation 範本移轉至 AWS CDK

CDK 移轉功能正在的預覽版本中， AWS CDK 且可能會有所變更。

使用命 AWS Cloud Development Kit (AWS CDK) 令列介面 (AWS CDK CLI) 將已部署的 AWS 資源、已部署的 AWS CloudFormation 堆疊和本機 AWS CloudFormation 範本移轉至 AWS CDK。

主題

遷移的運作方式

使用 AWS CDK CLI`cdk migrate`指令從下列來源移轉：

- 已部署的 AWS 資源。
- 已部署的 AWS CloudFormation 堆疊。
- 本機 AWS CloudFormation 範本。

已部署 AWS 資源

您可以從未與 AWS CloudFormation 堆疊關聯的特定環境 (AWS 帳戶 和 AWS 區域) 移轉已部署的 AWS 資源。

AWS CDK CLI利用 IaC 生成器服務來掃描 AWS 環境中的資源以收集資源詳細信息。若要進一步了解 IaC Generator，請參閱《AWS CloudFormation 使用者指南》中的〈產生現有資源的範本〉。

收集資源詳細資料後，會 AWS CDK CLI建立新的 CDK 應用程式，其中包含一個包含已移轉資源的單一堆疊。

已部署 AWS CloudFormation 堆疊

您可以將單一 AWS CloudFormation 堆疊移轉至新的 AWS CDK 應用程式。 AWS CDK CLI將擷取堆疊的 AWS CloudFormation 範本，並建立新的 CDK 應用程式。CDK 應用程序將由包含已遷移堆棧的單個堆 AWS CloudFormation 棧組成。

本地 AWS CloudFormation 模板

您可以從本端 AWS CloudFormation 樣板移轉。本機範本可能包含也可能不包含已部署的資源。AWS CDK CLI將創建一個新的 CDK 應用程序，其中包含一個包含您的資源的單個堆棧。

移轉後，您可以管理、修改和部署 CDK 應用程式， AWS CloudFormation 以佈建或更新資源。

CDK 遷移的好處

將資源遷移到 AWS CDK 過去一直是一個手動過程，需要時間和專業知識，甚 AWS CDK 至需要開始。 AWS CloudFormation 使用 CDK Migrate，可 AWS CDK CLI以在一小部分時間內為您完成大部分的遷移工作。CDK Migrate 將快速協助您開始使用在 AWS上 AWS CDK 開發和管理新的和現有的應用程式。

考量事項

一般考量

CDK 移轉與 CDK 匯入

該`cdk import`命令可以將部署的資源導入到新的或現有的 CDK 應用程序中。導入時，每個資源都必須在您的應用程序中手動定義為 L1 構造。我們建議您`cdk import`使用一次將一或多個資源匯入新的或現有的 CDK 應用程式。如需進一步了解，請參閱將現有資源匯入堆疊。

此命`cdk migrate`令會從已部署的資源、部署的 AWS CloudFormation 堆疊或本機 AWS CloudFormation 範本移轉至新的 CDK 應用程式。在移轉期間，會 AWS CDK CLI使`cdk import`用將資源匯入新的 CDK 應用程式。 AWS CDK CLI也會為您的每個資源產生 L1 建構。我們建議在從支援的移轉來源匯入新的應用 AWS CDK 程式`cdk migrate`時使用。

CDK 移轉僅建立 L1 建構

新創建的 CDK 應用程序將僅包含 L1 構造。您可以在遷移後向應用程序添加更高級別的構造。

CDK 遷移會建立包含單一堆疊的 CDK 應用程式

新創建的 CDK 應用程序將包含一個堆棧。

移轉已部署的資源時，所有移轉的資源都會包含在新 CDK 應用程式的單一堆疊中。

移轉 AWS CloudFormation 堆疊時，您只能將單一 AWS CloudFormation 堆疊移轉至新 CDK 應用程式中的單一堆疊。

移轉資產

專案資產 (例如程式 AWS Lambda 碼) 不會直接移轉至新的 CDK 應用程式。移轉後，您可以指定資產值以將其包含在 CDK 應用程式中。

移轉可設定狀態資源

遷移可設定狀態資源時，例如資料庫和 Amazon Simple Storage Service (Amazon S3) 儲存貯體，您通常會想要遷移現有資源，而不是建立新資源。

若要移轉和保留可設定狀態的資源，請執行下列動作：

- 確認您的可設定狀態資源支援匯入。如需詳細資訊，請參閱《AWS CloudFormation 使用指南》中的資源類型支援。
- 移轉後，請確認新 CDK 應用程式中已移轉資源的邏輯 ID 與已部署資源的邏輯 ID 相符。
- 如果是從 AWS CloudFormation 堆疊移轉，請確認新 CDK 應用程式中的堆疊名稱是否與 AWS CloudFormation 堆疊相符。
- 使用相同 AWS 區域 的 AWS 帳戶和已遷移的資源部署 CDK 應用程式。

從範本移轉時的 AWS CloudFormation 考量

CDK 移轉支援單一範本移轉

移轉樣 AWS CloudFormation 板時，您可以選取要移轉的單一樣板。不支援巢狀範本。

移轉具有內建函數的範本

從使用內建函數的 AWS CloudFormation 範本遷移時， AWS CDK CLI會嘗試將您的邏輯與類別一起遷移到 CDK 應用程式中。Fn要了解更多信息，請參閱 AWS Cloud Development Kit (AWS CDK) API 參考中的 Fn 類。

從已部署的資源移轉時的考量

掃描限制

掃描環境中的資源時，IAC 生成器對掃描時可以檢索的數據和配額限制具體限制。若要深入瞭解，請參閱AWS CloudFormation 使用者指南中的考量事項。

必要條件

使用`cdk migrate`指令之前，請執行下列動作：

1. 使用建立驗證 AWS。如需說明，請參閱步驟 2：設定程式設計存取。
2. 安裝或升級 AWS CDK CLI. 如需安裝指示，請參閱步驟 3：安裝 AWS CDKCLI。

開始使用 CDK 移轉

若要開始，請從您選擇的目錄執行 AWS CDK CLI`cdk migrate`命令。根據您執行的移轉類型，提供必要和選用的選項。

如需可搭配使用之選項的完整清單和說明`cdk migrate`，請參閱cdk migrate指令參考。

以下是您可能需要提供的一些重要選項。

Stack name (堆疊名稱)

唯一需要的選項是`--stack-name`。使用此選項可為移轉後將在應用 AWS CDK 程式內建立的堆疊指定名稱。堆棧名稱也將用作部署時 AWS CloudFormation 堆棧的名稱。

語言

用`--language`於指定新 CDK 應用程式的程式設計語言。

AWS 帳戶和 AWS 區域

會從預設來源 AWS CDK CLI擷取 AWS 帳戶和 AWS 區域 資訊。如需詳細資訊，請參閱 步驟 2：設定程式設計存取。您可以使用`--account`和`--region`選項`cdk migrate`來提供其他值。

新 CDK 專案的輸出目錄

依預設， AWS CDK CLI會在您的工作目錄中建立新的 CDK 專案，並使用您提供的值`--stack-name`來命名專案資料夾。如果目前存在具有相同名稱的資料夾，將 AWS CDK CLI會覆寫該資料夾。

您可以使用`--output-path`選項為新 CDK 專案資料夾指定不同的輸出路徑。

移轉來源

提供一個選項以指定要移轉的來源。

- `--from-path`— 從本端 AWS CloudFormation 範本移轉。
- `--from-scan`— 從 AWS 帳戶中部署的資源遷移和 AWS 區域.
- `--from-stack`— 從 AWS CloudFormation 堆疊移轉。

視您的移轉來源而定，您可以提供其他選項來自訂指`cdk migrate`令。

從 AWS CloudFormation 堆疊移轉

若要從已部署的 AWS CloudFormation 堆疊移轉，請提供選`--from-stack`項。提供已部署 AWS CloudFormation 堆疊的名稱`--stack-name`。以下是範例：

```
$ cdk migrate --from-stack --stack-name "myCloudFormationStack"
```

AWS CDK CLI將執行以下操作：

1. 擷取已部署堆疊的 AWS CloudFormation 範本。
2. 執行`cdk init`以初始化新的 CDK 應用程式。
3. 在包含已遷移堆疊的 CDK 應用程式中建立 AWS CloudFormation 堆疊。

當您從已部署的 AWS CloudFormation 堆疊移轉時，會 AWS CDK CLI嘗試將已部署的資源邏輯 ID 和已部署的 AWS CloudFormation 堆疊名稱與新 CDK 應用程式中的移轉資源和堆疊相符。

遷移後，您可以正常管理和修改 CDK 應用程序。部署時， AWS CloudFormation 會將部署識別為 AWS CloudFormation 堆疊更新，因為相符的 AWS CloudFormation 堆疊名稱。具有相符邏輯 ID 的資源將會更新。如需部署的詳細資訊，請參閱管理和部署您的 CDK 應用程式。

從 AWS CloudFormation 範本移轉

CDK 移轉支援從`JSON`或`YAML`格式化的 AWS CloudFormation 樣板進行移轉。

若要從本端 AWS CloudFormation 樣板移轉，請使用`--from-path`此選項並提供本端樣板的路徑。您還必須提供必要的`--stack-name`選項。以下是範例：

```
$ cdk migrate --from-path "./template.json" --stack-name "myCloudFormationStack"
```

AWS CDK CLI將執行以下操作：

1. 擷取您的本機 AWS CloudFormation 範本。
2. 執行`cdk init`以初始化新的 CDK 應用程式。
3. 在包含已遷移 AWS CloudFormation 範本的 CDK 應用程式中建立堆疊。

遷移後，您可以正常管理和修改 CDK 應用程序。在部署時，您有以下選項：

- 更新 AWS CloudFormation 堆疊 — 如果先前已部署本機 AWS CloudFormation 範本，您可以更新已部署的 AWS CloudFormation 堆疊。
- 部署新 AWS CloudFormation 堆疊 — 如果從未部署本機 AWS CloudFormation 範本，或者您想要從先前部署的範本建立新堆疊，則可以部署新 AWS CloudFormation 堆疊。

從 AWS SAM 範本移轉

若要從 AWS Serverless Application Model (AWS SAM) 範本移轉，您必須先將其轉換為 AWS CloudFormation 範本，或進行部署才能建立 AWS CloudFormation 堆疊。

若要將 AWS SAM 樣板轉換為 AWS CloudFormation，您可以使用 AWS SAM CLI`sam validate --debug`指令。在執行此命令之前，您可能必須`false`在`samconfig.toml`檔案中設定`lint`為。

若要轉換為 AWS CloudFormation 堆疊，請使用部署 AWS SAM 範本 AWS SAM CLI。然後從已部署的堆疊移轉。

從已部署的資源移轉

若要從已部署的 AWS 資源移轉，請提供`--from-scan`選項。您還必須提供必要的`--stack-name`選項。以下是範例：

```
$ cdk migrate --from-scan --stack-name "myCloudFormationStack"
```

AWS CDK CLI將執行以下操作：

1. 掃描您的帳戶以獲取資源和財產的詳細信息 — AWS CDK CLI 利用 IaC 發電機掃描您的帳戶並收集詳細信息.

2. 生成一個 AWS CloudFormation 模板-掃描後， AWS CDK CLI利用 IaC 生成器創建一個 AWS CloudFormation 模板。
3. 初始化新的 CDK 應用程序並遷移您的模板- AWS CDK CLI 運行`cdk init`以初始化新 AWS CDK 應用程序並將 AWS CloudFormation 模板作為單個堆棧遷移到 CDK 應用程序中。

使用篩選

默認情況下， AWS CDK CLI將掃描整個 AWS 環境並將資源遷移到 IAC 生成器的最大配額限制。您可以提供篩選條件， AWS CDK CLI以指定將資源從您的帳戶遷移至新 CDK 應用程式的條件。如需進一步了解，請參閱`--filter`。

使用 IAC 產生器掃描資源

根據您帳戶中的資源數量，掃描可能需要幾分鐘的時間。掃描過程中將顯示進度條。

支援的資源類型

AWS CDK CLI將遷移由 IaC 發生器支持的資源。如需完整清單，請參閱AWS CloudFormation 使用指南中的資源類型支援。

解析唯寫屬性

某些支援的資源包含唯寫屬性。這些屬性可以寫入，配置屬性，但不能由 IaC 生成器讀取或 AWS CloudFormation 獲取值。例如，出於安全原因，用於指定數據庫密碼的屬性可能是唯寫的。

在遷移過程中掃描資源時，IAC 生成器將檢測可能包含唯寫屬性的資源，並將它們分類為以下任何類型：

- `MUTUALLY_EXCLUSIVE_PROPERTIES`— 這些是特定資源的唯寫屬性，可互換並具有類似用途。需要其中一個互斥的屬性才能配置您的資源。例如，`AWS::Lambda::Function`資源的`S3BucketImageUri`、和`ZipFile`屬性是互斥的唯寫屬性。其中任何一個都可以用來指定函數資產，但您必須使用其中一個。
- `MUTUALLY_EXCLUSIVE_TYPES`— 這些是接受多種組態類型的必要唯寫內容。例如，`AWS::ApiGateway::RestApi`資源的`Body`屬性接受物件或字串類型。
- `UNSUPPORTED_PROPERTIES`— 這些是唯寫屬性，不屬於其他兩個類別。它們是可選的屬性或接受對象數組所需的屬性。

如需有關唯寫屬性以及 IAC 產生器在掃描已部署資源和建立 AWS CloudFormation 範本時如何管理這些屬性的詳細資訊，請參閱《使用者指南》中的 IAC 產生器和唯寫屬性。AWS CloudFormation

移轉之後，您必須在新的 CDK 應用程式中指定唯寫屬性值。 AWS CDK CLI將在 CDK 項目的文ReadMe件中附加一個警告部分，以記錄 IaC 生成器標識的任何只寫屬性。以下是範例：

```
# Welcome to your CDK TypeScript project
...
## Warnings
### Write-only properties
Write-only properties are resource property values that can be written to but can't be
 read by AWS CloudFormation or CDK Migrate. For more information, see [IaC generator
 and write-only properties](https://docs.aws.amazon.com/AWSCloudFormation/latest/
UserGuide/generate-IaC-write-only-properties.html).

Write-only properties discovered during migration are organized here by resource ID and
 categorized by write-only property type. Resolve write-only properties by providing
 property values in your CDK app. For guidance, see [Resolve write-only properties]
(https://docs.aws.amazon.com/cdk/v2/guide/migrate.html#migrate-resources-writeonly).
### MyLambdaFunction
- **UNSUPPORTED_PROPERTIES**:
  - SnapStart/ApplyOn: Applying SnapStart setting on function resource type.Possible
 values: [PublishedVersions, None]
This property can be replaced with other types
  - Code/S3ObjectVersion: For versioned objects, the version of the deployment package
 object to use.
This property can be replaced with other exclusive properties
- **MUTUALLY_EXCLUSIVE_PROPERTIES**:
  - Code/S3Bucket: An Amazon S3 bucket in the same AWS Region as your function. The
 bucket can be in a different AWS account.
This property can be replaced with other exclusive properties
  - Code/S3Key: The Amazon S3 key of the deployment package.
This property can be replaced with other exclusive properties
```

- 警告會在標題下組織，這些標題可識別與其相關聯的資源邏輯 ID。
- 警告會依類型分類。這些類型直接來自 IaC 發電機。

若要解析唯寫屬性

1. 識別要從 CDK 專案檔案的「警告」區段解析的唯寫性質。ReadMe在這裡，您可以記下 CDK 應用程式中可能包含唯寫屬性的資源，並識別探索到的唯寫屬性類型。

a. 針對MUTUALLY_EXCLUSIVE_PROPERTIES，決定要在 AWS CDK 應用程式中設定哪個互斥屬性。

b. 對於MUTUALLY_EXCLUSIVE_TYPES，決定您將使用哪一種可接受的類型來設定屬性。

c. 對於UNSUPPORTED_PROPERTIES，請確定屬性是選擇性還是必要的。然後，根據需要進行配置。

2. 使用 IAC 生成器和只寫屬性的指導來參考警告類型的含義。

3. 在您的 CDK 應用程序中，還將在應用程序的Props部分中指定要解析的只寫屬性值。在此提供正確的值。如需屬性說明和指引，您可以參考 AWS CDK API 參考。

   以下是遷移的 CDK 應用程序中的Props部分示例，其中包含兩個要解決的唯寫屬性：

```
export interface MyTestAppStackProps extends cdk.StackProps {
  /**
   * The Amazon S3 key of the deployment package.
   */
  readonly lambdaFunction00asdfasdfsadf008grk1CodeS3Keym8P82: string;
  /**
   * An Amazon S3 bucket in the same AWS Region as your function. The bucket can be
 in a different AWS account.
   */
  readonly lambdaFunction00asdfasdfsadf008grk1CodeS3Bucketzidw8: string;
}
```

解析所有唯寫屬性值之後，您就可以準備好進行部署。

移轉的 .json 檔案

AWS CDK CLI會在移轉期間在 AWS CDK 專migrate.json案中建立檔案。此檔案包含已部署資源的參考資訊。第一次部署 CDK 應用程式時，會 AWS CDK CLI使用此檔案來參考已部署的資源、將資源與新 AWS CloudFormation 堆疊相關聯，然後刪除檔案。

管理和部署您的 CDK 應用程式

移轉至時 AWS CDK，新的 CDK 應用程式可能無法立即部署就緒。本主題說明管理和部署新 CDK 應用程式時要考慮的行動項目。

準備部署

在部署之前，您必須準備 CDK 應用程序。

合成您的應用

使用指`cdk synth`令將 CDK 應用程式中的堆疊合成範本。 AWS CloudFormation

如果您是從已部署的 AWS CloudFormation 堆疊或範本移轉，則可以將合成的範本與移轉的範本進行比較，以驗證資源和屬性值。

要進一步了解 `cdk synth`，請參閱 合成堆疊。

執行差異

如果您是從部署的 AWS CloudFormation 堆疊移轉，您可以使用 cdk diff 指令與新 CDK 應用程式中的堆疊進行比較。

要了解有關 cdk 差異的更多信息，請參閱比較堆疊。

引導您的環境

如果您是第一次從 AWS 環境部署，請使用`cdk bootstrap`來準備您的環境。如需進一步了解，請參閱引導。

部署您的 CDK 應用程式

當您部署 CDK 應用程式時，會 AWS CDK CLI利用該 AWS CloudFormation 服務來佈建您的資源。資源被捆綁到 CDK 應用程序中的單個堆棧中，並作為單個 AWS CloudFormation 堆棧部署。

視您移轉來源的位置而定，您可以部署以建立新 AWS CloudFormation 堆疊或更新現有 AWS CloudFormation 堆疊。

部署以建立新 AWS CloudFormation 堆疊

如果您從已部署的資源移轉，則 AWS CDK CLI會在部署時自動建立新 AWS CloudFormation 堆疊。您部署的資源將包含在新 AWS CloudFormation 堆疊中。

如果您從未部署的本機 AWS CloudFormation 範本移轉，則 AWS CDK CLI會在部署時自動建立新 AWS CloudFormation 堆疊。

如果您從先前部署的已部署 AWS CloudFormation 堆疊或本機 AWS CloudFormation 範本移轉，您可以部署以建立新 AWS CloudFormation 堆疊。若要建立新堆疊，請執行下列動作：

- 部署至新 AWS 環境。這包括使用不同的 AWS 帳戶或部署到不同的帳戶 AWS 區域。
- 如果您想要將新堆疊部署到已移轉堆疊或範本的相同 AWS 環境，則必須將 CDK 應用程式中的堆疊名稱修改為新值。您還必須修改 CDK 應用程序中資源的所有邏輯 ID。然後，您可以部署到相同的環境，以建立新的堆疊和新資源。

部署以更新現有 AWS CloudFormation 堆疊

如果您從先前部署的已部署 AWS CloudFormation 堆疊或本機 AWS CloudFormation 範本移轉，則可以部署以更新現有 AWS CloudFormation 堆疊。

驗證 CDK 應用程式中的堆疊名稱是否與已 AWS CloudFormation 部署堆疊的堆疊名稱相符，並部署至相同的 AWS 環境。

使用支援的 AWS CDK 程式設計語言

使用以使用支援的程式設計語言 AWS Cloud Development Kit (AWS CDK)來定義您的 AWS 雲端 基礎結構。

主題

匯入建 AWS 構資源庫

包 AWS CDK 括 AWS 構造庫，按 AWS 服務組織的結構的集合。該庫的穩定構造在單個模塊中提供，由其TypeScript包名稱調用：`aws-cdk-lib`。實際的套件名稱因語言而異。

TypeScript

```
安裝                    #### aws-cdk-lib
匯入                    ## CDK = ###'aws-cdk-lib'#;
```

JavaScript

```
安裝                    #### aws-cdk-lib
匯入                    ## CDK = ###'aws-cdk-lib'#;
```

Python

安裝	`##-m #### aws-cdk-lib`
匯入	`# aws_cdk ### CDK`

Java

新增至 `pom.xml`	Group `##. ###.`; artifact `aws-cdk-lib`
匯入	`####. ###.` (for example)

C#

安裝	`##########.`
匯入	`##### .cdk;`

`construct`基底類別和支援程式碼位於`constructs`模組中。API 仍在進行細化的實驗結構作為單獨的模塊分發。

AWS CDK API 參考資料

AWS CDK API 參考提供程式庫中建構 (和其他元件) 的詳細文件。每種受支援的程式設計語言都會提供 API 參考的版本。

每個模塊的參考材料分為以下幾個部分。

- 概觀：使用中的服務所需知道的介紹性材料 AWS CDK，包括概念和範例。
- 建構：代表一或多個具體 AWS 資源的程式庫類別。這些是「策劃」（L2）資源或模式（L3 資源），它們提供具有理智默認值的高級接口。
- 類別：提供模組中建構所使用之功能的非建構類別。
- 結構：定義複合值結構的數據結構（屬性包），例如屬性（構造的`props`參數）和選項。
- 接口：名稱全部以「I」開頭的接口，定義相應構造或其他類的絕對最小功能。CDK 使用構造接口來表示 AWS 在應用程 AWS CDK 序外部定義並由諸如`Bucket.fromBucketArn()`.

- 枚舉：用於指定某些構造參數的命名值的集合。使用列舉值可讓 CDK 在合成期間檢查這些值的有效性。
- CloudFormation 資源：這些名稱以「Cfn」開頭的 L1 結構，代表規範中定義的資源完全相同。CloudFormation 它們會在每個 CDK 版本中自動從該規格產生。每個 L2 或 L3 建構都會封裝一或多個資源。 CloudFormation
- CloudFormation 屬性類型：定義每個 CloudFormation 資源之特性之具名值的集合。

接口與構造類比較

以特定的方式 AWS CDK 使用接口，即使您熟悉接口作為編程概念，也可能不明顯。

AWS CDK 支持使用 CDK 應用程序外定義的資源使用的方法，如`Bucket.fromBucketArn()`. 外部資源無法修改，並且可能無法使用 CDK 應用程序中定義的資源（例如`Bucket`類）具有所有可用的功能。接口代表 CDK 中針對特定 AWS 資源類型（包括外部資源）提供的最低功能。

在 CDK 應用程序中實例化資源時，應始終使用具體類，例如. `Bucket` 在您自己的一個結構中指定要接受的參數類型時，請使用接口類型，例如，`IBucket`如果您準備處理外部資源（也就是說，您不需要更改它們）。如果您需要 CDK 定義的建構，請指定您可以使用的最一般類型。

有些介面是與特定類別相關聯的屬性或選項組合的最低版本，而不是建構。當子類接受您將傳遞給父類的參數時，這種接口可能很有用。如果您需要一個或多個額外的屬性，你會想要實現或從這個接口，或從一個更具體的類型派生。

> **Note**
>
> 支援的某些程式設計語言 AWS CDK 沒有介面功能。在這些語言中，接口只是普通的類。您可以通過它們的名字來識別它們，它遵循初始「I」的模式，後跟其他構造的名稱（例如`IBucket`）。同樣的規則適用。

管理相依性

您 AWS CDK 應用程式或程式庫的相依性是使用套件管理工具來管理。這些工具通常與編程語言一起使用。

通常情況下， AWS CDK 支持該語言的標準或官方軟件包管理工具（如果有的話）。否則， AWS CDK 將支持該語言最受歡迎或廣泛支持的語言。您也可以使用其他工具，特別是如果它們與支持的工具一起使用。但是，對其他工具的官方支持是有限的。

AWS CDK 支援下列套件管理員：

語言	支援的套件管理工具
TypeScript/JavaScript	NPM（節點 Package 管理器）或紗線
Python	PIP (適用於 Python 的 Package 安裝程式)
Java	Maven
C#	NuGet
Go	圍棋模塊

當您使用 AWS CDK CLI`cdk init`指令建立新專案時，CDK 核心程式庫和穩定建構的相依性會自動指定。

如需管理支援之程式設計語言之相依性的詳細資訊，請參閱下列內容：

- 管理相依性 TypeScript.
- 管理相依性 JavaScript.
- 管理相依性 Python.
- 管理相依性 Java.
- 管理相依性 C#.
- 管理相依性 Go.

TypeScript與其他語言比 AWS CDK 較

TypeScript 是開發 AWS CDK 應用程式所支援的第一種語言。因此，大量的示例 CDK 代碼被寫入 TypeScript。如果您使用另一種語言進行開發，則比較 AWS CDK 代碼的實現方式 TypeScript 與您選擇的語言相比可能會很有用。這可以幫助您在整個文檔中使用示例。

導入模塊

TypeScript/JavaScript

TypeScript 支援匯入整個命名空間，或從命名空間匯入個別物件。每個命名空間都包含可與指定 AWS 服務搭配使用的建構和其他類別。

```
// Import main CDK library as cdk
import * as cdk from 'aws-cdk-lib';   // ES6 import preferred in TS
const cdk = require('aws-cdk-lib');   // Node.js require() preferred in JS

// Import specific core CDK classes
import { Stack, App } from 'aws-cdk-lib';
const { Stack, App } = require('aws-cdk-lib');

// Import AWS S3 namespace as s3 into current namespace
import { aws_s3 as s3 } from 'aws-cdk-lib';   // TypeScript
const s3 = require('aws-cdk-lib/aws-s3');     // JavaScript

// Having imported cdk already as above, this is also valid
const s3 = cdk.aws_s3;

// Now use s3 to access the S3 types
const bucket = s3.Bucket(...);

// Selective import of s3.Bucket
import { Bucket } from 'aws-cdk-lib/aws-s3';        // TypeScript
const { Bucket } = require('aws-cdk-lib/aws-s3');   // JavaScript

// Now use Bucket to instantiate an S3 bucket
const bucket = Bucket(...);
```

Python

像 TypeScript Python 支持命名空間模塊導入和選擇性導入。Python 中的命名空間看起來像 aws_cdk。 xxx，其中 xxx 代表 AWS 服務名稱，例如 Amazon S3 的 s3。（這些示例中使用了 Amazon S3）。

```
# Import main CDK library as cdk
import aws_cdk as cdk

# Selective import of specific core classes
from aws_cdk import Stack, App

# Import entire module as s3 into current namespace
import aws_cdk.aws_s3 as s3

# s3 can now be used to access classes it contains
bucket = s3.Bucket(...)
```

```
# Selective import of s3.Bucket into current namespace
from aws_cdk.s3 import Bucket

# Bucket can now be used to instantiate a bucket
bucket = Bucket(...)
```

Java

Java 的導入工作方 TypeScript式與. 每個 import 語句從給定的包或導入一個單一的類名，或者在該包中定義的所有類（使用*）。類可以使用任一類名本身，如果它已被導入，或包括它的包合格的類名進行訪問。

庫的命名類似`software.amazon.awscdk.services.xxx`於 AWS 構造庫（主庫是`software.amazon.awscdk`）。 AWS CDK 套件的 Maven 群組識別碼是`software.amazon.awscdk`。

```
// Make certain core classes available
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.App;

// Make all Amazon S3 construct library classes available
import software.amazon.awscdk.services.s3.*;

// Make only Bucket and EventType classes available
import software.amazon.awscdk.services.s3.Bucket;
import software.amazon.awscdk.services.s3.EventType;

// An imported class may now be accessed using the simple class name (assuming that
 name
// does not conflict with another class)
Bucket bucket = Bucket.Builder.create(...).build();

// We can always use the qualified name of a class (including its package) even
 without an
// import directive
software.amazon.awscdk.services.s3.Bucket bucket =
    software.amazon.awscdk.services.s3.Bucket.Builder.create(...)
        .build();

// Java 10 or later can use var keyword to avoid typing the type twice
var bucket =
    software.amazon.awscdk.services.s3.Bucket.Builder.create(...)
```

```
        .build();
```

C#

在 C# 中，您可以使用using指示詞匯入型別。有兩種樣式。一個使您可以通過使用它們的普通名稱訪問指定名稱空間中的所有類型。另一方面，您可以使用別名來引用命名空間本身。

包被命名Amazon.CDK.AWS.xxx為類似於 AWS 構造庫包。（核心模塊是Amazon.CDK。）

```
// Make CDK base classes available under cdk
using cdk = Amazon.CDK;

// Make all Amazon S3 construct library classes available
using Amazon.CDK.AWS.S3;

// Now we can access any S3 type using its name
var bucket = new Bucket(...);

// Import the S3 namespace under an alias
using s3 = Amazon.CDK.AWS.S3;

// Now we can access an S3 type through the namespace alias
var bucket = new s3.Bucket(...);

// We can always use the qualified name of a type (including its namespace) even
 without a
// using directive
var bucket = new Amazon.CDK.AWS.S3.Bucket(...)
```

Go

每個「AWS 建構程式庫」模組都是以 Go 套件的形式提供。

```
import (
    "github.com/aws/aws-cdk-go/awscdk/v2"           // CDK core package
    "github.com/aws/aws-cdk-go/awscdk/v2/awss3"     // AWS S3 construct library
 module
)

// now instantiate a bucket
bucket := awss3.NewBucket(...)
```

```
// use aliases for brevity/clarity
import (
    cdk "github.com/aws/aws-cdk-go/awscdk/v2"           // CDK core package
    s3  "github.com/aws/aws-cdk-go/awscdk/v2/awss3"     // AWS S3 construct library
 module
)

bucket := s3.NewBucket(...)
```

實例化構造

AWS CDK 在所有支援的語言中，建構類別的名稱都相同。大多數語言使用new關鍵字來實例化一個類（Python 和 Go 不）。此外，在大多數語言中，關鍵字this指的是當前實例。（self按慣例使用Python。）您應該將參數作為scope參數傳遞給當前實例的引用到您創建的每個構造。

構造的第三個參數是props，包含 AWS CDK 構建構所需屬性的對象。這個參數可能是可選的，但是當需要時，支持的語言以慣用方式處理它。屬性的名稱也會根據語言的標準命名模式進行調整。

TypeScript/JavaScript

```
// Instantiate default Bucket
const bucket = new s3.Bucket(this, 'MyBucket');

// Instantiate Bucket with bucketName and versioned properties
const bucket = new s3.Bucket(this, 'MyBucket', {
  bucketName: 'my-bucket',
   versioned: true,
});

// Instantiate Bucket with websiteRedirect, which has its own sub-properties
const bucket = new s3.Bucket(this, 'MyBucket', {
  websiteRedirect: {host: 'aws.amazon.com'}});
```

Python

Python 在實例化類時不使用new關鍵字。屬性參數使用關鍵字參數表示，並使用命名參數snake_case。

如果 prop 值本身就是一組屬性，則會以屬性命名的類別來表示，該類別會接受子屬性的關鍵字引數。

在 Python 中，當前實例被傳遞給方法作為第一個參數，這是self由約定命名。

```
# Instantiate default Bucket
bucket = s3.Bucket(self, "MyBucket")

# Instantiate Bucket with bucket_name and versioned properties
bucket = s3.Bucket(self, "MyBucket", bucket_name="my-bucket", versioned=true)

# Instantiate Bucket with website_redirect, which has its own sub-properties
bucket = s3.Bucket(self, "MyBucket", website_redirect=s3.WebsiteRedirect(
            host_name="aws.amazon.com"))
```

Java

在 Java 中，prop 參數由一個名為的類表示XxxxProps（例如，BucketProps對於Bucket構造的 prop）。您可以使用構建器模式構建道具參數。

每個XxxxProps類都有一個構建器。另外，每個建構都有一個方便的建構器，它們只需一個步驟即可建立道具和建構，如下列範例所示。

道具的名稱與中的名稱相同 TypeScript，使用camelCase。

```
// Instantiate default Bucket
Bucket bucket = Bucket(self, "MyBucket");

// Instantiate Bucket with bucketName and versioned properties
Bucket bucket = Bucket.Builder.create(self, "MyBucket")
                      .bucketName("my-bucket").versioned(true)
                      .build();

# Instantiate Bucket with websiteRedirect, which has its own sub-properties
Bucket bucket = Bucket.Builder.create(self, "MyBucket")
                      .websiteRedirect(new websiteRedirect.Builder()
                          .hostName("aws.amazon.com").build())
                      .build();
```

C#

在 C# 中，prop 使用對象初始值設定項指定到名為的類XxxxProps（例如，BucketProps對於Bucket構造的 prop）。

道具的名稱類似於 TypeScript，除了使用PascalCase。

實例化構造時使用var關鍵字很方便，因此您不需要鍵入兩次類名。但是，您的本地代碼樣式指南可能會有所不同。

```
// Instantiate default Bucket
var bucket = Bucket(self, "MyBucket");

// Instantiate Bucket with BucketName and Versioned properties
var bucket =  Bucket(self, "MyBucket", new BucketProps {
                      BucketName = "my-bucket",
                      Versioned  = true});

// Instantiate Bucket with WebsiteRedirect, which has its own sub-properties
var bucket = Bucket(self, "MyBucket", new BucketProps {
                      WebsiteRedirect = new WebsiteRedirect {
                              HostName = "aws.amazon.com"
                      }});
```

Go

要在 Go 中創建一個構造，請調用該函數，NewXxxxxx其中Xxxxxxx是構造的名稱。結構的屬性被定義為一個結構。

在 Go 中，所有構造參數都是指針，包括數字，布爾值和字符串等值。使用諸如jsii.String創建這些指針的便利功能。

```
// Instantiate default Bucket
bucket := awss3.NewBucket(stack, jsii.String("MyBucket"), nil)

// Instantiate Bucket with BucketName and Versioned properties
bucket1 := awss3.NewBucket(stack, jsii.String("MyBucket"), &awss3.BucketProps{
 BucketName: jsii.String("my-bucket"),
 Versioned:  jsii.Bool(true),
})

// Instantiate Bucket with WebsiteRedirect, which has its own sub-properties
bucket2 := awss3.NewBucket(stack, jsii.String("MyBucket"), &awss3.BucketProps{
 WebsiteRedirect: &awss3.RedirectTarget{
  HostName: jsii.String("aws.amazon.com"),
 }})
```

存取成員

它是常見的引用屬性或構造和其他 AWS CDK 類的屬性，並使用這些值作為，例如，輸入來構建其他構造。上述方法的命名差異也適用於此處。此外，在 Java 中，無法直接訪問成員。相反，提供了一個吸氣方法。

TypeScript/JavaScript

名字是camelCase。

```
bucket.bucketArn
```

Python

名字是snake_case。

```
bucket.bucket_arn
```

Java

為每個屬性提供了 getter 方法；這些camelCase名稱為。

```
bucket.getBucketArn()
```

C#

名字是PascalCase。

```
bucket.BucketArn
```

Go

名字是PascalCase。

```
bucket.BucketArn
```

枚舉常量

枚舉常量的作用域為一個類，並且在所有語言中具有帶下劃線的大寫名稱（有時稱為）。SCREAMING_SNAKE_CASE由於除 Go 以外的所有支持語言中的類名也使用相同的外殼，因此合格枚舉名稱在這些語言中也是相同的。

```
s3.BucketEncryption.KMS_MANAGED
```

在 Go 中，枚舉常量是模塊命名空間的屬性，編寫如下。

```
awss3.BucketEncryption_KMS_MANAGED
```

物件介面

AWS CDK 使用 TypeScript 物件介面來指示類別會實作預期的方法和屬性集。您可以辨識物件介面，因為其名稱以開頭I。具體類指示它通過使用implements關鍵字實現的接口。

TypeScript/JavaScript

> **Note**
>
> JavaScript 沒有界面功能。您可以忽略implements關鍵字和它後面的類名。

```
import { IAspect, IConstruct } from 'aws-cdk-lib';

class MyAspect implements IAspect {
  public visit(node: IConstruct) {
    console.log('Visited', node.node.path);
  }
}
```

Python

Python 沒有一個接口功能。但是，對於 AWS CDK 你可以通過裝飾你的類來指示接口實現@jsii.implements(interface)。

```
from aws_cdk import IAspect, IConstruct
```

```
import jsii

@jsii.implements(IAspect)
class MyAspect():
  def visit(self, node: IConstruct) -> None:
    print("Visited", node.node.path)
```

Java

```
import software.amazon.awscdk.IAspect;
import software.amazon.awscdk.IConstruct;

public class MyAspect implements IAspect {
    public void visit(IConstruct node) {
        System.out.format("Visited %s", node.getNode().getPath());
    }
}
```

C#

```
using Amazon.CDK;

public class MyAspect : IAspect
{
    public void Visit(IConstruct node)
    {
        System.Console.WriteLine($"Visited ${node.Node.Path}");
    }
}
```

Go

Go 結構不需要明確聲明它們實現的接口。Go 編譯器根據結構上可用的方法和屬性來確定實現。例如，在下列程式碼中，會`MyAspect`實作`IAspect`介面，因為它提供了採用建構的`Visit`方法。

```
type MyAspect struct {
}

func (a MyAspect) Visit(node constructs.IConstruct) {
 fmt.Println("Visited", *node.Node().Path())
}
```

使用 AWS CDK 中 TypeScript

TypeScript 是完全支援的用戶端語言， AWS Cloud Development Kit (AWS CDK) 且被認為是穩定的。TypeScript 使用 AWS CDK in 使用熟悉的工具，包括微軟的 TypeScript 編譯器 (`tsc`)、Node.js 和節點 Package 管理員 (npm)。如果您願意，您也可以使用 Yarn，儘管本指南中的示例使用 NPM。包括 AWS 構造庫的模塊是通過故宮存儲庫 npm js.org 分佈的。

您可以使用任何編輯器或 IDE。許多開 AWS CDK 發人員使用 Visual Studio 代碼（或其開放源代碼等效的 VScodium），它具有出色的支持。 TypeScript

主題

開始使用 TypeScript

若要使用 AWS CDK，您必須擁有 AWS 帳戶和認證，並已安裝 Node.js 和工 AWS CDK 具組。請參閱開始使用 AWS CDK。

您還需要 TypeScript 自己（版本 3.8 或更高版本）。如果您還沒有它，則可以使用npm.

```
npm install -g typescript
```

> **Note**
>
> 如果您收到權限錯誤，並具有系統管理員訪問權限，請嘗試`sudo npm install -g typescript`。

與常規保持 TypeScript 最新狀態`npm update -g typescript`。

> **Note**
>
> 第三方語言棄用：語言版本僅在供應商或社區共享其 EOL（生命週期終止）之前受到支持，並且可能會在事先通知的情況下進行更改。

建立專案

您可以在空目錄`cdk init`中呼叫來建立新 AWS CDK 專案。使用選`--language`項並指定`typescript`：

```
mkdir my-project
cd my-project
cdk init app --language typescript
```

建立專案也會安裝`aws-cdk-lib`模組及其相依性。

`cdk init`使用專案資料夾的名稱來命名專案的各種元素，包括類別、子資料夾和檔案。資料夾名稱中的連字號會轉換為底線。但是，名稱應該遵循 TypeScript 標識符的形式；例如，它不應以數字開頭或包含空格。

使用本地`tsc`和 `cdk`

在大多數情況下，本指南假設您全局安裝 TypeScript CDK Toolkit（`npm install -g typescript aws-cdk`），並且提供的命令示例（例如`cdk synth`）遵循此假設。這種方法可以很容易地將兩個組件保持在最新狀態，並且由於兩者都採用了嚴格的向後兼容性方法，因此始終使用最新版本的風險通常很小。

有些團隊更喜歡指定每個項目中的所有依賴關係，包括 TypeScript 編譯器和 CDK Toolkit 之類的工具。此作法可讓您將這些元件釘選到特定版本，並確保您團隊中的所有開發人員 (以及您的 CI/CD 環境) 都使用完全這些版本。這消除了可能的變更來源，有助於使組建和部署更加一致且可重複。

CDK 在 TypeScript 項目模板中包含了 CDK Toolkit 的依賴關係`package.json`，因此，如果您想使用此方法，則不需要對項目進行任何更改。 TypeScript 您需要做的就是使用稍微不同的命令來構建您的應用程序和發出`cdk`命令。

操作	使用全域工具	使用本機工具

初始化專案	`cdk init --language typescript`	`npx aws-cdk init --language typescript`
組建	`tsc`	`npm run build`
執行 CDK 工具組命令	`cdk ...`	`npm run cdk ...` or `npx aws-cdk ...`

`npx aws-cdk`執行在目前專案本機安裝的 CDK Toolkit 版本 (如果有的話)，則會退回至全域安裝 (如果有的話)。如果沒有全域安裝，請`npx`下載 CDK Toolkit 的暫存副本並執行該副本。您可以使用以下`@`語法指定 CDK 工具包的任意版本：`npx aws-cdk@2.0 --version`打印`2.0.0`。

> **Tip**
>
> 設定別名，以便您可以在本機 CDK Toolkit 安裝中使用`cdk`指令。
>
> macOS/Linux
>
> ```
> alias cdk="npx aws-cdk"
> ```
>
> Windows
>
> ```
> doskey cdk=npx aws-cdk $*
> ```

管理 AWS 建構程式庫模組

使用節點 Package 件管理員 (npm) 安裝和更新「 AWS 建構程式庫」模組，以供您的應用程式使用，以及您需要的其他套件。（`npm`如果您願意，您可以使用`yarn`而不是。）`npm`還會自動為這些模塊安裝依賴關係。

大多數 AWS CDK 構造都在主 CDK 包中，命名為`aws-cdk-lib`，這是由創建的新項目中的默認依賴項。`cdk init`「實驗」 AWS 構造庫模塊，其中更高級別的構造仍在開發中，命名為類似。`@aws-cdk/`*`SERVICE-NAME`*`-alpha`服務名稱具有 aws- 前綴。如果您不確定模塊的名稱，請在 NPM 上搜索它。

> **Note**
>
> CDK API 參考資料也會顯示套件名稱。

例如，以下指令會安裝的實驗模組 AWS CodeStar。

```
npm install @aws-cdk/aws-codestar-alpha
```

某些服務的「建構程式庫」支援位於多個命名空間中。例如，此外`aws-route53`，還有三個額外的 Amazon Route 53 命名空間`aws-route53-targets`、`aws-route53-patterns`、和`aws-route53resolver`。

您專案的相依性會維護於`package.json`. 您可以編輯此檔案，將部分或全部相依性鎖定到特定版本，或允許在特定條件下將它們更新為較新的版本。要根據您在中指定的規則將項目的 NPM 依賴項更新為最新的允許版本`package.json`：

```
npm update
```

在中 TypeScript，您可以使用與使用 NPM 安裝模組相同的名稱將模組匯入程式碼中。我們建議您在應用程式中匯入 AWS CDK 類別和 AWS 建構程式庫模組時採取下列作法。遵循這些準則將有助於使您的代碼與其他 AWS CDK 應用程序一致，以及更容易理解。

- 使用 ES6 風格的`import`指令，而不是。`require()`
- 一般而言，從中匯入個別類別`aws-cdk-lib`。

  ```
  import { App, Stack } from 'aws-cdk-lib';
  ```

- 如果您需要許多類別`aws-cdk-lib`，您可以使用的命名空間別名，`cdk`而不是匯入個別類別。避免兩者都做。

  ```
  import * as cdk from 'aws-cdk-lib';
  ```

- 一般而言，使用短命名空間別名匯入 AWS 服務結構。

  ```
  import { aws_s3 as s3 } from 'aws-cdk-lib';
  ```

管理相依性 TypeScript

在 TypeScript CDK 項目中，依賴關係在項目的主目錄中的`package.json`文件中指定。核心 AWS CDK 模組位於稱為的單一NPM套件中`aws-cdk-lib`。

當您使用安裝套件時npm install，NPM 會`package.json`為您記錄套件。

如果您願意，您可以使用紗線代替 NPM。但是，CDK 不支持紗線 plug-and-play模式，這是紗線 2 中的默認模式。將以下內容添加到項目的`.yarnrc.yml`文件中以關閉此功能。

```
nodeLinker: node-modules
```

CDK 應用

以下是`cdk init --language typescript`命令所產生的範例`package.json`檔案：

```
{
  "name": "my-package",
  "version": "0.1.0",
  "bin": {
    "my-package": "bin/my-package.js"
  },
  "scripts": {
    "build": "tsc",
    "watch": "tsc -w",
    "test": "jest",
    "cdk": "cdk"
  },
  "devDependencies": {
    "@types/jest": "^26.0.10",
    "@types/node": "10.17.27",
    "jest": "^26.4.2",
    "ts-jest": "^26.2.0",
    "aws-cdk": "2.16.0",
    "ts-node": "^9.0.0",
    "typescript": "~3.9.7"
  },
  "dependencies": {
    "aws-cdk-lib": "2.16.0",
    "constructs": "^10.0.0",
    "source-map-support": "^0.5.16"
  }
```

```
}
```

對於可部署的 CDK 應用程式，aws-cdk-lib必須在的一dependencies節中指定。package.json您可以使用脫字符號 (^) 版本編號說明符來表示您將接受比指定版本更新的版本，只要它們位於相同的主要版本中即可。

對於實驗結構，請為 alpha 構造庫模塊指定確切的版本，這些模塊具有可能更改的 API。不要使用 ^ 或 ~，因為這些模塊的更高版本可能會帶來 API 更改，從而破壞您的應用程序。

在的devDependencies章節中指定測試應用程式所需的程式庫版本和工具 (例如，jest測試架構) package.json。或者，使用 ^ 指定可接受更新的相容版本。

第三方構造庫

如果您要開發建構程式庫，請使用peerDependencies和devDependencies區段的組合來指定其相依性，如下列範例package.json檔案所示。

```
{
  "name": "my-package",
  "version": "0.0.1",
  "peerDependencies": {
    "aws-cdk-lib": "^2.14.0",
    "@aws-cdk/aws-appsync-alpha": "2.10.0-alpha",
    "constructs": "^10.0.0"
  },
  "devDependencies": {
    "aws-cdk-lib": "2.14.0",
    "@aws-cdk/aws-appsync-alpha": "2.10.0-alpha",
    "constructs": "10.0.0",
    "jsii": "^1.50.0",
    "aws-cdk": "^2.14.0"
  }
}
```

在中peerDependencies，使用脫字符號 (^) 來指定您的程式庫可使用aws-cdk-lib的最低版本。這樣可以最大限度地提高您的庫與一系列 CDK 版本的兼容性。為 alpha 構造庫模塊指定確切版本，這些模塊具有可能更改的 API。peerDependencies使用可確保node_modules樹中只有一個所有 CDK 庫的副本。

在中devDependencies，指定測試所需的工具和程式庫，選擇性地使用 ^ 表示可接受更新的相容版本。請確實指定 (不含 ^ 或 ~) 最低版本，以aws-cdk-lib及您公告程式庫相容的其他 CDK 套件。此

做法可確保您的測試會針對這些版本執行。這樣，如果您無意中使用了僅在較新版本中找到的功能，則測試可以 catch 它。

> **Warning**
>
> `peerDependencies`僅由 NPM 7 及更高版本自動安裝。如果您使用的是 NPM 6 或更早版本，或者如果您使用的是 Yarn，則必須在`devDependencies`. 否則，它們將不會被安裝，並且您將收到有關未解析的對等依賴關係的警告。

安裝和更新相依性

運行以下命令來安裝項目的依賴關係。

NPM

```
# Install the latest version of everything that matches the ranges in 'package.json'
npm install

# Install the same exact dependency versions as recorded in 'package-lock.json'
npm ci
```

Yarn

```
# Install the latest version of everything that matches the ranges in 'package.json'
yarn upgrade

# Install the same exact dependency versions as recorded in 'yarn.lock'
yarn install --frozen-lockfile
```

要更新已安裝的模塊，可以使用前面的npm install和yarn upgrade命令。其中一個命令會`node_modules`將中的套件更新為符合中規則的最新版本`package.json`。但是，它們不會`package.json`自行更新，您可能需要執行此操作來設置新的最低版本。如果您在裝載套件 GitHub，則可以將 Dependabot 版本更新設定為自動更新。`package.json`或使用 npm-check-updates。

> **Important**
>
> 根據設計，當您安裝或更新相依性時，NPM 和 Yarn 會選擇符合中`package.json`指定需求的每個套件的最新版本。總是存在這些版本可能被破壞的風險（無論是意外還是有意）。更新項目的依賴關係後進行徹底測試。

AWS CDK 在成語 TypeScript

道具

所有的 AWS 構造庫類都使用三個參數實例化：正在定義構造的範圍（它在構造樹中的父級），一個 id 和 prop。參數 prop 是構造用於配置其創建的 AWS 資源的鍵/值對的捆綁。其他類和方法也使用「屬性包」模式作為參數。

在中 TypeScript，的形狀`props`是使用介面來定義，告訴您必要引數和選用引數及其類型。這樣的接口是為每種`props`參數定義的，通常特定於單個構造或方法。例如，Bucket 建構 (在中`aws-cdk-lib/aws-s3 module`) 會指定符合`BucketProps`介面的`props`引數。

如果屬性本身就是物件 (例如的 websiteRedirect 屬性)`BucketProps`，則該物件將具有自己的介面，其形狀必須符合此介面，在此情況下`RedirectTarget`。

如果您正在對 AWS Construct Library 類進行子類別分類（或覆蓋接受 props-like 參數的方法），則可以從現有接口繼承以創建一個新的指定代碼所需的任何新道具。當調用父類或基本方法時，通常您可以傳遞收到的整個 prop 參數，因為對像中提供但未在接口中指定的任何屬性都將被忽略。

的 future 版本 AWS CDK 可能會巧合地添加一個新屬性，其中包含您用於自己屬性的名稱。傳遞您接收到繼承鏈的值可能會導致非預期的行為。通過移除或設置屬性時收到的道具的淺層副本更安全`undefined`。例如：

```
super(scope, name, {...props, encryptionKeys: undefined});
```

或者，命名您的屬性，以便清楚它們屬於您的構造。這樣，它們不太可能會與 future 版本中的屬性 AWS CDK 發生衝突。如果其中有很多，請使用單個適當命名的對象來容納它們。

缺少值

物件中缺少的值 (例如 prop) 具有`undefined`中的值 TypeScript。版本 3.7 的語言引入了簡化了使用這些值的運算符，使得在達到未定義值時更容易指定默認值和「短路」鏈接。如需這些功能的詳細資訊，請參閱 TypeScript 3.7 版本說明，特別是前兩個功能，選擇性鏈結和 Null 合併。

建置、合成和部署

一般而言，在建置和執行應用程式時，您應該位於專案的根目錄中。

Node.js 無法 TypeScript 直接執行，而是將您的應用程式轉換為 JavaScript 使用 TypeScript編譯器、`tsc`. 然後執行生成的 JavaScript 代碼。

只要需要運行您的應用程序，就會 AWS CDK 自動執行此操作。但是，手動編譯以檢查錯誤並運行測試可能很有用。若要手動編譯應 TypeScript 用程式，請發出問題`npm run build`。您還可能會發出進`npm run watch`入監視模式，在此模式中，每當您將更改保存到源文件時， TypeScript編譯器都會自動重建您的應用程序。

您可以使用以下命令單獨或一起合成和部署在 AWS CDK 應用程序中定義的堆棧。通常，當您發出它們時，您應該位於項目的主目錄中。

- `cdk synth`：從 AWS CDK 應用程序中的一個或多個堆棧中合成 AWS CloudFormation 模板。
- `cdk deploy`：將 AWS CDK 應用程序中一個或多個堆棧定義的資源部署到 AWS。

您可以在單一命令中指定要合成或部署的多個堆疊的名稱。如果您的應用程序只定義了一個堆棧，則不需要指定它。

```
cdk synth                 # app defines single stack
cdk deploy Happy Grumpy   # app defines two or more stacks; two are deployed
```

您也可以使用萬用字元 * (任意數目的字元) 和? （任何單個字符）以按模式識別堆棧。使用萬用字元時，請將模式括在引號中。否則，shell 可能會嘗試將其擴展到當前目錄中的文件的名稱，然後再將其傳遞到 AWS CDK Toolkit。

```
cdk synth "Stack?"    # Stack1, StackA, etc.
cdk deploy "*Stack"   # PipeStack, LambdaStack, etc.
```

> **Tip**
>
> 您不需要在部署堆疊之前明確合成堆疊；請`cdk deploy`執行此步驟，以確保部署最新的程式碼。

如需`cdk`指令的完整文件，請參閱the section called “AWS CDK 工具包”。

使用 AWS CDK 中的 JavaScript

JavaScript是完全支援的用戶端語言， AWS CDK 且被認為是穩定的。 JavaScript 使用 AWS Cloud Development Kit (AWS CDK) 中的使用熟悉的工具，包括 Node.js 和節點 Package 管理員 (npm)。如果您願意，您也可以使用 Yarn，儘管本指南中的示例使用 NPM。包括 AWS 構造庫的模塊是通過故宮存儲庫 npm js.org 分發的。

您可以使用任何編輯器或 IDE。許多開 AWS CDK 發人員使用 Visual Studio 代碼（或其開源等效的 VScodium），它具有很好的支持。 JavaScript

主題

- 開始使用 JavaScript
- 建立專案
- 使用本機 cdk
- 管理 AWS 建構程式庫模組
- 管理相依性 JavaScript
- AWS CDK 在成語 JavaScript
- 合成和部署
- 使用 TypeScript 範例 JavaScript
- 移轉至 TypeScript

開始使用 JavaScript

若要使用 AWS CDK，您必須擁有 AWS 帳戶和認證，並已安裝 Node.js 和工 AWS CDK 具組。請參閱開始使用 AWS CDK。

JavaScript AWS CDK 除此之外，應用程式不需要其他先決條

> **Note**
>
> 第三方語言棄用：語言版本僅在供應商或社區共享其 EOL（生命週期終止）之前受到支持，並且如有更改，恕不另行通知。

建立專案

您可以在空目錄`cdk init`中呼叫來建立新 AWS CDK 專案。使用選`--language`項並指定`javascript`：

```
mkdir my-project
cd my-project
cdk init app --language javascript
```

建立專案也會安裝`aws-cdk-lib`模組及其相依性。

`cdk init`使用專案資料夾的名稱來命名專案的各種元素，包括類別、子資料夾和檔案。資料夾名稱中的連字號會轉換為底線。但是，名稱應該遵循 JavaScript 標識符的形式；例如，它不應以數字開頭或包含空格。

使用本機 cdk

在大多數情況下，本指南假設您將 CDK Toolkit 全域 (`npm install -g aws-cdk`) 安裝，並且提供的命令示例（例如`cdk synth`）遵循此假設。這種方法可以很容易地將 CDK Toolkit 保持在最新狀態，並且由於 CDK 採取了嚴格的向後兼容性方法，因此始終使用最新版本的風險通常很小。

有些團隊更喜歡指定每個項目中的所有依賴關係，包括 CDK 工具包之類的工具。這個做法可讓您將這些元件釘選到特定版本，並確保您團隊中的所有開發人員 (以及您的 CI/CD 環境) 都完全使用這些版本。這消除了可能的變更來源，有助於使組建和部署更加一致且可重複。

CDK 在 JavaScript 項目模板中包含 CDK Toolkit 的依賴項`package.json`，因此，如果您想使用此方法，則不需要對項目進行任何更改。您需要做的就是使用稍微不同的命令來構建您的應用程序和發出`cdk`命令。

操作	使用全域 CDK 工具組	使用本地 CDK 工具包
初始化專案	`CDK ##### JavaScript`	`npx aws-cdk ##### JavaScript`
執行 CDK 工具組命令	`CDK...`	`npm ## CDK...` or `npx # aws-cdk...`

`npx aws-cdk`執行在目前專案本機安裝的 CDK Toolkit 版本 (如果有的話)，則會退回至全域安裝 (如果有的話)。如果沒有全域安裝，請`npx`下載 CDK Toolkit 的暫存副本並執行該副本。您可以使用以下`@`語法指定 CDK 工具包的任意版本：`npx aws-cdk@1.120 --version`打印`1.120.0`。

> **Tip**
>
> 設定別名，以便您可以在本機 CDK Toolkit 安裝中使用`cdk`指令。
>
> macOS/Linux
>
> ```
> alias cdk="npx aws-cdk"
> ```
>
> Windows
>
> ```
> doskey cdk=npx aws-cdk $*
> ```

管理 AWS 建構程式庫模組

使用節點 Package 件管理員 (npm) 安裝和更新「 AWS 建構程式庫」模組，以供您的應用程式使用，以及您需要的其他套件。（`npm`如果您願意，您可以使用`yarn`而不是。）`npm`還會自動為這些模塊安裝依賴關係。

大多數 AWS CDK 構造都在主 CDK 包中，命名為`aws-cdk-lib`，這是由創建的新項目中的默認依賴項。`cdk init`「實驗」 AWS 構造庫模塊，其中更高級別的構造仍在開發中，命名為類似。`aws-cdk-lib/`*`SERVICE-NAME`*`-alpha`服務名稱具有 aws- 前綴。如果您不確定模塊的名稱，請在 NPM 上搜索它。

> **Note**
>
> CDK API 參考資料也會顯示套件名稱。

例如，以下指令會安裝的實驗模組 AWS CodeStar。

```
npm install @aws-cdk/aws-codestar-alpha
```

某些服務的「建構程式庫」支援位於多個命名空間中。例如，此外aws-route53，還有三個額外的 Amazon Route 53 命名空間aws-route53-targets、aws-route53-patterns、和aws-route53resolver。

您專案的相依性會維護於package.json. 您可以編輯此檔案，將部分或全部相依性鎖定到特定版本，或允許在特定條件下將它們更新為較新的版本。要根據您在中指定的規則將項目的 NPM 依賴項更新為最新的允許版本package.json：

```
npm update
```

在中 JavaScript，您可以使用與使用 NPM 安裝模組相同的名稱將模組匯入程式碼中。我們建議您在應用程式中匯入 AWS CDK 類別和 AWS 建構程式庫模組時採取下列作法。遵循這些準則將有助於使您的代碼與其他 AWS CDK 應用程序一致，以及更容易理解。

- 使用require()，而不是 ES6 風格import的指令。舊版 Node.js 不支援 ES6 匯入，因此使用較舊的語法會更廣泛地相容。（如果您真的想使用 ES6 導入，請使用 esm 來確保您的項目與所有受支持的 Node.js 版本兼容。）
- 一般而言，從中匯入個別類別aws-cdk-lib。

  ```
  const { App, Stack } = require('aws-cdk-lib');
  ```
- 如果您需要許多類別aws-cdk-lib，您可以使用的命名空間別名，cdk而不是匯入個別類別。避免兩者都做。

  ```
  const cdk = require('aws-cdk-lib');
  ```
- 通常，使用短命名空間別名匯入 AWS 建構程式庫。

  ```
  const { s3 } = require('aws-cdk-lib/aws-s3');
  ```

管理相依性 JavaScript

在 JavaScript CDK 項目中，依賴關係在項目的主目錄中的package.json文件中指定。核心 AWS CDK 模塊位於名為的單個NPM包中aws-cdk-lib。

當您使用安裝套件時npm install，NPM 會package.json為您記錄套件。

如果您願意，您可以使用紗線代替 NPM。但是，CDK 不支持紗線 plug-and-play模式，這是紗線 2 中的默認模式。將以下內容添加到項目的.yarnrc.yml文件中以關閉此功能。

```
nodeLinker: node-modules
```

CDK 應用

以下是`cdk init --language typescript`命令所產生的範例`package.json`檔案。為生成的文件類似，只 JavaScript 是沒有相 TypeScript關條目。

```
{
  "name": "my-package",
  "version": "0.1.0",
  "bin": {
    "my-package": "bin/my-package.js"
  },
  "scripts": {
    "build": "tsc",
    "watch": "tsc -w",
    "test": "jest",
    "cdk": "cdk"
  },
  "devDependencies": {
    "@types/jest": "^26.0.10",
    "@types/node": "10.17.27",
    "jest": "^26.4.2",
    "ts-jest": "^26.2.0",
    "aws-cdk": "2.16.0",
    "ts-node": "^9.0.0",
    "typescript": "~3.9.7"
  },
  "dependencies": {
    "aws-cdk-lib": "2.16.0",
    "constructs": "^10.0.0",
    "source-map-support": "^0.5.16"
  }
}
```

對於可部署的 CDK 應用程式，`aws-cdk-lib`必須在的一`dependencies`節中指定。`package.json`您可以使用脫字符號 (^) 版本編號說明符來表示您將接受比指定版本更新的版本，只要它們位於相同的主要版本中即可。

對於實驗結構，請為 alpha 構造庫模塊指定確切的版本，這些模塊具有可能更改的 API。不要使用 ^ 或 ~，因為這些模塊的更高版本可能會帶來 API 更改，從而破壞您的應用程序。

在的devDependencies章節中指定測試應用程式所需的程式庫版本和工具 (例如，jest測試架構) package.json。或者，使用 ^ 指定可接受更新的相容版本。

第三方構造庫

如果您要開發建構程式庫，請使用peerDependencies和devDependencies區段的組合來指定其相依性，如下列範例package.json檔案所示。

```
{
  "name": "my-package",
  "version": "0.0.1",
  "peerDependencies": {
    "aws-cdk-lib": "^2.14.0",
    "@aws-cdk/aws-appsync-alpha": "2.10.0-alpha",
    "constructs": "^10.0.0"
  },
  "devDependencies": {
    "aws-cdk-lib": "2.14.0",
    "@aws-cdk/aws-appsync-alpha": "2.10.0-alpha",
    "constructs": "10.0.0",
    "jsii": "^1.50.0",
    "aws-cdk": "^2.14.0"
  }
}
```

在中peerDependencies，使用脫字符號 (^) 來指定您的程式庫可使用aws-cdk-lib的最低版本。這樣可以最大限度地提高您的庫與一系列 CDK 版本的兼容性。為 alpha 構造庫模塊指定確切版本，這些模塊具有可能更改的 API。peerDependencies使用可確保node_modules樹中只有一個所有 CDK 庫的副本。

在中devDependencies，指定測試所需的工具和程式庫，選擇性地使用 ^ 表示可接受更新的相容版本。請確實指定 (不含 ^ 或 ~) 最低版本，以aws-cdk-lib及您公告程式庫相容的其他 CDK 套件。此做法可確保您的測試會針對這些版本執行。這樣，如果您無意中使用了僅在較新版本中找到的功能，則測試可以 catch 它。

> **Warning**
>
> peerDependencies僅由 NPM 7 及更高版本自動安裝。如果您使用的是 NPM 6 或更早版本，或者如果您使用的是 Yarn，則必須在devDependencies. 否則，它們將不會被安裝，並且您將收到有關未解析的對等依賴關係的警告。

安裝和更新相依性

運行以下命令來安裝項目的依賴關係。

NPM

```
# Install the latest version of everything that matches the ranges in 'package.json'
npm install

# Install the same exact dependency versions as recorded in 'package-lock.json'
npm ci
```

Yarn

```
# Install the latest version of everything that matches the ranges in 'package.json'
yarn upgrade

# Install the same exact dependency versions as recorded in 'yarn.lock'
yarn install --frozen-lockfile
```

要更新已安裝的模塊，可以使用前面的npm install和yarn upgrade命令。其中一個命令會node_modules將中的套件更新為符合中規則的最新版本package.json。但是，它們不會package.json自行更新，您可能需要執行此操作來設置新的最低版本。如果您在裝載套件GitHub，則可以將 Dependabot 版本更新設定為自動更新。package.json或使用 npm-check-updates。

> **Important**
>
> 根據設計，當您安裝或更新相依性時，NPM 和 Yarn 會選擇符合中package.json指定需求的每個套件的最新版本。總是存在這些版本可能被破壞的風險（無論是意外還是有意）。更新項目的依賴關係後進行徹底測試。

AWS CDK 在成語 JavaScript

道具

所有的 AWS 構造庫類都使用三個參數實例化：構造被定義的範圍（它在構造樹中的父級），一個 id 和 prop，構造用於配置它創建的 AWS 資源的鍵/值對的包。其他類和方法也使用「屬性包」模式作為參數。

使用具有良好 JavaScript 自動完成功能的 IDE 或編輯器將有助於避免屬性名稱拼寫錯誤。如果一個構造期待一個`encryptionKeys`屬性，並且你拼寫它`encryptionkeys`，在實例化構造時，你還沒有傳遞你想要的值。如果需要屬性，這可能會在合成時導致錯誤，或者如果屬性是可選的，則會導致無訊息地忽略該屬性。在後一種情況下，您可能會得到您打算覆蓋的默認行為。在這裡特別小心。

當對一個 AWS 構造庫類進行子類（或者覆蓋帶有類似 props 參數的方法）時，您可能需要接受其他屬性以供自己使用。這些值將被父類或重寫的方法忽略，因為它們永遠不會在該代碼中訪問，所以你通常可以傳遞你收到的所有 prop。

的 future 版本 AWS CDK 可能會巧合地添加一個新屬性，其中包含您用於自己屬性的名稱。傳遞您接收到繼承鏈的值可能會導致非預期的行為。通過移除或設置屬性時收到的道具的淺層副本更安全`undefined`。例如：

```
super(scope, name, {...props, encryptionKeys: undefined});
```

或者，命名您的屬性，以便清楚它們屬於您的構造。這樣，它們不太可能會與 future 版本中的屬性 AWS CDK 發生衝突。如果其中有很多，請使用單個適當命名的對象來容納它們。

缺少值

物件中的缺少值 (例如`props`) 具有`undefined`中的值 JavaScript。通常的技術適用於處理這些問題。例如，用於訪問可能未定義的值的屬性的常見習慣用法如下：

```
// a may be undefined, but if it is not, it may have an attribute b
// c is undefined if a is undefined, OR if a doesn't have an attribute b
let c = a && a.b;
```

但是，`a`如果還有其他一些「假」價值`undefined`，最好使測試更加明確。在這裡，我們將利用`null`和等`undefined`於一次測試它們的事實：

```
let c = a == null ? a : a.b;
```

> **Tip**
>
> Node.js 14.0 及更新版本支援可簡化處理未定義值的新運算子。有關更多信息，請參閱可選鏈接和空合併提案。

合成和部署

您可以使用以下命令單獨或一起合成和部署在 AWS CDK 應用程序中定義的堆棧。通常，當您發出它們時，您應該位於項目的主目錄中。

- `cdk synth`：從 AWS CDK 應用程序中的一個或多個堆棧中合成 AWS CloudFormation 模板。
- `cdk deploy`：將 AWS CDK 應用程序中一個或多個堆棧定義的資源部署到 AWS。

您可以在單一命令中指定要合成或部署的多個堆疊的名稱。如果您的應用程序只定義了一個堆棧，則不需要指定它。

```
cdk synth                 # app defines single stack
cdk deploy Happy Grumpy   # app defines two or more stacks; two are deployed
```

您也可以使用萬用字元 * (任意數目的字元) 和? （任何單個字符）以按模式識別堆棧。使用萬用字元時，請將模式括在引號中。否則，shell 可能會嘗試將其擴展到當前目錄中的文件的名稱，然後再將其傳遞到 AWS CDK Toolkit。

```
cdk synth "Stack?"    # Stack1, StackA, etc.
cdk deploy "*Stack"   # PipeStack, LambdaStack, etc.
```

> **Tip**
>
> 您不需要在部署堆疊之前明確合成堆疊；請`cdk deploy`執行此步驟，以確保部署最新的程式碼。

如需`cdk`指令的完整文件，請參閱the section called “AWS CDK 工具包”。

使用 TypeScript 範例 JavaScript

TypeScript是我們用來開發的語言 AWS CDK，而且它是開發應用程式所支援的第一種語言，因此許多可用的程式 AWS CDK 碼範例都是用來編寫的 TypeScript。這些代碼示例對於 JavaScript 開發人員來說可能是一個很好的資源; 你只需要刪除代碼的 TypeScript特定部分。

TypeScript 程式碼片段通常使用較新的 ECMAScript `import` 和`export`關鍵字從其他模組匯入物件，並宣告要在目前模組外部使用的物件。Node.js 剛開始在其最新版本中支援這些關鍵字。根據您使用的 Node.js 版本（或希望支持），您可能會重寫導入和導出以使用較舊的語法。

匯入可以用對`require()`函數的呼叫來取代。

TypeScript

```
import * as cdk from 'aws-cdk-lib';
import { Bucket, BucketPolicy } from 'aws-cdk-lib/aws-s3';
```

JavaScript

```
const cdk = require('aws-cdk-lib');
const { Bucket, BucketPolicy } = require('aws-cdk-lib/aws-s3');
```

可以將匯出指定給`module.exports`物件。

TypeScript

```
export class Stack1 extends cdk.Stack {
  // ...
}

export class Stack2 extends cdk.Stack {
  // ...
}
```

JavaScript

```
class Stack1 extends cdk.Stack {
  // ...
}
```

```
class Stack2 extends cdk.Stack {
  // ...
}

module.exports = { Stack1, Stack2 }
```

> **Note**
>
> 使用舊式導入和導出的替代方法是使用該esm模塊。

一旦你已經排序了導入和導出，你可以深入研究實際的代碼。您可能會遇到以下常用 TypeScript 功能：

- 鍵入註釋
- 介面定義
- 類型轉換/轉換
- 訪問修飾符

可以為變量，類成員，函數參數和函數返回類型提供類型註釋。對於變數、參數和成員，類型是透過在識別名稱後面加上冒號和類型來指定。函數返回值跟隨函數簽名，並由冒號和類型組成。

若要將具有類型註解的程式碼轉換為 JavaScript，請移除冒號和型別。類別成員在中必須有一些值 JavaScript；undefined如果中只有類型註釋，則將其設定為 TypeScript。

TypeScript

```
var encrypted: boolean = true;

class myStack extends cdk.Stack {
    bucket: s3.Bucket;
    // ...
}

function makeEnv(account: string, region: string) : object {
    // ...
}
```

JavaScript

```
var encrypted = true;

class myStack extends cdk.Stack {
    bucket = undefined;
    // ...
}

function makeEnv(account, region) {
    // ...
}
```

在中 TypeScript，介面可用來提供必要和選用屬性的組合，以及它們的類型，一個名稱。然後，您可以使用介面名稱作為類型註釋。 TypeScript 將確保您用作函數的參數的對象具有正確類型的所需屬性。

```
interface myFuncProps {
    code: lambda.Code,
    handler?: string
}
```

JavaScript 沒有接口功能，因此一旦刪除了類型註釋，請完全刪除接口聲明。

當函數或方法傳回一般用途型別 (例如`object`)，但您想要將該值視為更特定的子類型，以存取不屬於較一般類型介面的屬性或方法時， TypeScript 可讓您使用`as`後面接著類型或介面名稱來轉換值。JavaScript 不支持（或需要）這個，所以只需刪除`as`和下面的標識符。較不常見的轉換語法是在括號中使用類型名稱`<LikeThis>`；這些轉換也必須移除。

最後， TypeScript 支援類別成員的存取修飾詞`public``protected`、和`private`。中的所有班級成員 JavaScript 都是公開的。只需將這些修飾符刪除即可。

知道如何識別和刪除這些 TypeScript 功能對於適應短 TypeScript片段很長的路要 JavaScript 走。但是，以這種方式轉換較長的 TypeScript 示例可能是不切實際的，因為它們更有可能使用其他 TypeScript 功能。對於這些情況，我們建議使用 Sucrase。例如，如果代碼使用未定義的變量，Sucrase 就不會抱怨。`tsc`如果它在語法上是有效的，那麼除了少數例外，Sucrase 可以將其翻譯為。 JavaScript這使得它對於轉換可能無法自行執行的程式碼片段特別有用。

移轉至 TypeScript

許多 JavaScript 開發人員轉移到TypeScript他們的項目變得越來越大，更複雜。 TypeScript 是 JavaScript —all JavaScript 代碼的超集合是有效的代 TypeScript 碼，因此不需要對代碼進行更改，並且它也是一種支持的語言。 AWS CDK 鍵入註釋和其他 TypeScript 功能是可選的，可以在您找到其中的價值時將其添加到您的 AWS CDK 應用程序中。 TypeScript 也可讓您在完成之前儘早存取新 JavaScript 功能，例如選擇性的鏈結和空結合，而且不需要升級 Node.js。

TypeScript的「基於形狀」的接口，它定義了對象中必需和可選屬性（及其類型）的捆綁，允許在編寫代碼時捕獲常見錯誤，並使 IDE 更容易提供強大的自動完成和其他實時編碼建議。

在中編碼 TypeScript 確實涉及一個額外的步驟：使用編譯 TypeScript 器編譯您的應用程序，`tsc`. 對於典型的 AWS CDK 應用程序，編譯最多需要幾秒鐘。

將現有 JavaScript AWS CDK 應用程序遷移到最簡單的方法 TypeScript 是使用創建一個新 TypeScript 項目`cdk init app --language typescript`，然後將源文件（以及任何其他必要的文件，例如 AWS Lambda 函數源代碼等資產）複製到新項目中。將您的 JavaScript 檔案重新命名為結尾，`.ts`然後開始在中開發 TypeScript。

AWS CDK 在 Python 中使用

Python 是完全支援的用戶端語言， AWS Cloud Development Kit (AWS CDK) 且被認為是穩定的。 AWS CDK 在 Python 中使用熟悉的工具，包括標準的 Python 實作 (CPython)、使用的虛擬環境`virtualenv`，以及 Python 套件安裝程式`pip`。包含 AWS 構造庫的模塊通過 pyp i.org 分發。 AWS CDK 甚至的 Python 版本使用 Python 樣式的標識符（例如，`snake_case`方法名稱）。

您可以使用任何編輯器或 IDE。許多開 AWS CDK 發人員使用 Visual Studio 代碼（或其開源等效的 vScodium），它具有通過官方擴展對 Python 的良好支持。Python 中包含的空閒編輯器就足以開始使用。的 Python 模組確實 AWS CDK 有類型提示，這些提示對於內嵌工具或支援型別驗證的 IDE 非常有用。

主題

- 開始使用 Python
- 建立專案
- 管理 AWS 建構程式庫模組
- 管理相依性 Python
- AWS CDK Python 中的成語
- 合成和部署

開始使用 Python

若要使用 AWS CDK，您必須擁有 AWS 帳戶和認證，並已安裝 Node.js 和工 AWS CDK 具組。請參閱開始使用 AWS CDK。

Python AWS CDK 應用程序需要 3.6 或更高版本。如果您尚未安裝它，請到 pyth on.org 下載適用於您的作業系統的相容版本。如果您運行 Linux，則您的系統可能附帶了兼容的版本，或者您可以使用發行版的軟件包管理器（yumapt等）進行安裝。Mac 用戶可能對自製軟件（macOS 的 Linux 風格的軟件包管理器）感興趣。

> **Note**
>
> 第三方語言棄用：語言版本僅在供應商或社區共享其 EOL（生命週期終止）之前受到支持，並且可能會在事先通知的情況下進行更改。

還需要 Python 軟件包安裝程序和虛擬環境管理器。`pip` `virtualenv`相容 Python 版本的視窗安裝包括這些工具。在 Linux 上，`pip`並且`virtualenv`可以在您的軟件包管理器中作為單獨的軟件包提供。或者，您可以使用以下命令安裝它們：

```
python -m ensurepip --upgrade
python -m pip install --upgrade pip
python -m pip install --upgrade virtualenv
```

如果您遇到權限錯誤，請使用`--user`旗標執行上述指令，以便將模組安裝在您的使用者目錄中，或使用`sudo`來取得在整個系統範圍內安裝模組的權限。

> **Note**
>
> 對於 Linux 發行版來說，通常使用 Python 3.x 的可執行文件名稱`python3`，並`python`參考 Python 2.x 安裝。一些發行版有一個可選的軟件包，您可以安裝，該軟件包使`python`命令引用 Python 3。否則，您可以通過在項目的主目錄`cdk.json`中編輯來調整用於運行應用程序的命令。

> **Note**
>
> 在視窗上，您可能需要使用可py執行文件（> Python 啟動器視窗pip）調用 Python（和）。除此之外，啟動器允許您輕鬆指定要使用的 Python 安裝版本。

如果python在命令行中鍵入導致有關從 Windows 應用商店安裝 Python 的消息，即使在安裝 Windows 版本的 Python 之後，打開 Windows 的管理應用程序執行別名設置面板並關閉 Python 的兩個應用程序安裝程序條目。

建立專案

您可以在空目錄`cdk init`中呼叫來建立新 AWS CDK 專案。使用選`--language`項並指定`python`：

```
mkdir my-project
cd my-project
cdk init app --language python
```

`cdk init`使用專案資料夾的名稱來命名專案的各種元素，包括類別、子資料夾和檔案。資料夾名稱中的連字號會轉換為底線。但是，名稱應該遵循 Python 標識符的形式; 例如，它不應該以數字開頭或包含空格。

若要使用新專案，請啟用其虛擬環境。如此可將專案的相依性安裝在本機專案資料夾中，而不是全域安裝。

```
source .venv/bin/activate
```

Note

您可以將其識別為用來啟用虛擬環境的 Mac/Linux 命令。Python 範本包含一個批次檔案`source.bat`，允許在 Windows 上使用相同的命令。傳統的 Windows 命令也可以使用。`.venv\Scripts\activate.bat`
如果您使用 CDK Toolkit v1.70.0 或更早版本初始化您的 AWS CDK 項目，則您的虛擬環境位於`.env`目錄中而不是。`.venv`

Important

每當您開始處理專案時，都會啟用專案的虛擬環境。否則，您將無法訪問安裝在那裡的模塊，並且您安裝的模塊將進入 Python 全局模塊目錄中（或將導致權限錯誤）。

首次啟用虛擬環境後，請安裝應用程式的標準相依性：

```
python -m pip install -r requirements.txt
```

管理 AWS 建構程式庫模組

使用 Python 套件安裝程式pip，來安裝和更新「 AWS 建構程式庫」模組以供應用程式使用，以及您需要的其他套件。 pip還會自動為這些模塊安裝依賴關係。如果您的系統不能識別pip為獨立命令，請pip作為 Python 模塊調用，如下所示：

```
python -m pip PIP-COMMAND
```

大多數 AWS CDK 建構都在aws-cdk-lib。實驗模塊位於名為 like 的單獨模塊中aws-cdk.*SERVICE-NAME*.alpha。服務名稱包括 aws 前綴。如果您不確定模塊的名稱，請在 PyPI 搜索它。例如，以下指令會安裝程式 AWS CodeStar 庫。

```
python -m pip install aws-cdk.aws-codestar-alpha
```

某些服務的結構位於多個命名空間中。例如，此外aws-cdk.aws-route53，還有三個額外的 Amazon Route 53 命名空間，命名為aws-route53-targetsaws-route53-patterns、和aws-route53resolver。

> **Note**
>
> CDK API 參考資料的 Python 版本也會顯示套件名稱。

用於將「 AWS 建構程式庫」模組匯入 Python 程式碼的名稱如下所示。

```
import aws_cdk.aws_s3 as s3
import aws_cdk.aws_lambda as lambda_
```

我們建議您在應用程式中匯入 AWS CDK 類別和 AWS 建構程式庫模組時採取下列作法。遵循這些準則將有助於使您的代碼與其他 AWS CDK 應用程序一致，以及更容易理解。

- 通常，從頂級導入單個類aws_cdk。

  ```
  from aws_cdk import App, Construct
  ```

- 如果您需要許多類別aws_cdk，您可以使用的命名空間別名，cdk而不是匯入個別類別。避免兩者都做。

```
import aws_cdk as cdk
```

- 通常，使用短命名空間別名匯入 AWS 建構程式庫。

```
import aws_cdk.aws_s3 as s3
```

安裝模塊後，更新項目的`requirements.txt`文件，其中列出了項目的依賴關係。最好手動執行此操作，而不是使用`pip freeze`。 `pip freeze`捕獲 Python 虛擬環境中安裝的所有模塊的當前版本，這在捆綁項目以在其他地方運行時非常有用。

但是，通常情況下，您`requirements.txt`應該只列出頂級依賴項（您的應用程序直接依賴的模塊），而不是這些庫的依賴關係。這種策略使更新您的依賴關係更簡單。

您可以編輯`requirements.txt`以允許升級；只需將之`==`前的版本號碼取代為以允許升級到更高的相容版本，或者完全移除版本需求，以指定模組的最新可用版本。`~=`

在適當`requirements.txt`編輯以允許升級之後，發出此命令以隨時升級項目的已安裝模塊：

```
pip install --upgrade -r requirements.txt
```

管理相依性 Python

在 Python 中，您可以通過將它們放入應用程序或`setup.py`構造庫中`requirements.txt`來指定依賴關係。然後使用 PIP 工具管理依賴關係。PIP 通過以下方式之一調用：

```
pip command options
python -m pip command options
```

python -m pip調用適用於大多數系統; pip 要求 PIP 的可執行文件位於系統路徑上。如果pip不起作用，請嘗試將其替換python -m pip。

此指cdk init --language python令會為您的新專案建立虛擬環境。這使得每個項目都有自己的依賴關係版本，也是一個基本`requirements.txt`文件。您必須在source .venv/bin/activate每次開始使用專案時執行來啟用此虛擬環境。

CDK 應用

以下是範例 `requirements.txt` 檔案。由於 PIP 沒有相依性鎖定功能，因此建議您使用 == 運算子來指定所有相依性的確切版本，如下所示。

```
aws-cdk-lib==2.14.0
aws-cdk.aws-appsync-alpha==2.10.0a0
```

使用安裝模組pip install並不會自動將其加入requirements.txt。你必須自己做。如果您要升級至相依性的較新版本，請在中編輯其版本號碼requirements.txt。

若要在建立或編輯之後安裝或更新專案的相依性requirements.txt，請執行下列指令：

```
python -m pip install -r requirements.txt
```

> **Tip**
>
> 此指pip freeze令會以可寫入文字檔的格式輸出所有已安裝相依性的版本。這可以用作需求檔案與pip install -r. 此文件很方便地將所有依賴關係（包括傳遞的依賴項）固定到您測試的確切版本。為了避免日後升級軟件包時出現問題，請使用單獨的文件來執行此操作，例如freeze.txt（不requirements.txt）。然後，在升級項目的依賴項時重新生成它。

第三方構造庫

在程式庫中，會在中指定相依性setup.py，以便在應用程式使用套件時自動下載傳遞相依性。否則，每個想要使用您的軟件包的應用程序都需要將您的依賴關係複製到它們的requirements.txt. 這裡顯示了一個例子setup.py。

```
from setuptools import setup

setup(
  name='my-package',
  version='0.0.1',
  install_requires=[
    'aws-cdk-lib==2.14.0',
  ],
  ...
)
```

要處理用於開發的軟件包，創建或激活一個虛擬環境，然後運行以下命令。

```
python -m pip install -e .
```

雖然 PIP 會自動安裝傳遞依賴關係，但是任何一個軟件包只能有一個已安裝的副本。系統會選取相依性樹狀結構中指定最高的版本；應用程式永遠會有安裝套件版本的最後一個字詞。

AWS CDK Python 中的成語

語言衝突

在 Python 中，`lambda`是一種語言關鍵字，因此您不能將其用作 AWS Lambda 構造庫模塊或 Lambda 函數的名稱。這種衝突的 Python 約定是在變量名中使用尾隨下劃線`lambda_`，如在中所示。

按照慣例， AWS CDK 構造的第二個參數被命名`id`。編寫自己的堆棧和構造時，調用參數 `id`「陰影」Python 內置函數`id()`，該函數返回對象的唯一標識符。這個函數不經常使用，但是如果您在構造中碰巧需要它，請重命名參數`construct_id`。

引數和屬性

所有的 AWS 構造庫類都使用三個參數實例化：正在定義構造的範圍（它在構造樹中的父級），一個 id 和 prop，構造用於配置它創建的資源的鍵/值對的包。其他類和方法也使用「屬性包」模式作為參數。

scope 和 id 應始終作為位置參數傳遞，而不是關鍵字參數傳遞，因為如果構造接受名為 scope 或 id 的屬性，它們的名稱會改變。

在 Python 中，道具被表示為關鍵字參數。如果一個參數包含嵌套的數據結構，這些結構是使用一個類來表示，該類在實例化時採用自己的關鍵字參數。相同的模式會套用至採用結構化引數的其他方法呼叫。

例如，在 Amazon S3 儲存貯體的`add_lifecycle_rule`方法中，該`transitions`屬性是`Transition`執行個體的清單。

```
bucket.add_lifecycle_rule(
  transitions=[
    Transition(
      storage_class=StorageClass.GLACIER,
      transition_after=Duration.days(10)
    )
  ]
)
```

當擴展一個類或覆蓋一個方法，你可能需要接受其他參數為自己的目的，這些參數不被父類理解。在這種情況下，您應該接受您不關心使用`**kwargs`成語的參數，並使用僅關鍵字參數來接受您感興趣的參

數。當調用父級的構造函數或重寫的方法時，只傳遞它期望的參數（通常只是**kwargs）。傳遞父類別或方法不期望的引數會導致錯誤。

```
class MyConstruct(Construct):
    def __init__(self, id, *, MyProperty=42, **kwargs):
        super().__init__(self, id, **kwargs)
        # ...
```

的 future 版本 AWS CDK 可能會巧合地添加一個新屬性，其中包含您用於自己屬性的名稱。這不會對您的構造或方法的用戶造成任何技術問題（因為您的屬性沒有「上鏈」傳遞，父類或重寫的方法將僅使用默認值），但可能會導致混淆。您可以通過命名屬性來避免這種潛在問題，以使它們顯然屬於您的構造。如果有許多新屬性，請將它們捆綁到適當命名的類中，並將其作為單個關鍵字參數傳遞。

缺少值

AWS CDK 用於表None示缺少或未定義的值。使用時**kwargs，如果未提供屬性，請使用字典的get()方法提供預設值。避免使用kwargs[...]，因為這會引KeyError發缺失的值。

```
encrypted = kwargs.get("encrypted")         # None if no property "encrypted" exists
encrypted = kwargs.get("encrypted", False)  # specify default of False if property is
 missing
```

某些 AWS CDK 方法（例如tryGetContext()獲取運行時上下文值）可能會返回None，您將需要明確檢查。

使用介面

Python 沒有像其他語言那樣具有接口功能，儘管它確實具有類似的抽象基類。（如果你不熟悉界面，維基百科有一個很好的介紹。）TypeScript，實現的語言確實 AWS CDK 提供了接口，並且構造和其他對 AWS CDK 象通常需要一個堅持到特定接口的對象，而不是從特定的類繼承。因此，提 AWS CDK 供了自己的接口功能作為 JSII 層的一部分。

要指示一個類實現了一個特定的接口，你可以使用@jsii.implements裝飾器：

```
from aws_cdk import IAspect, IConstruct
import jsii

@jsii.implements(IAspect)
class MyAspect():
    def visit(self, node: IConstruct) -> None:
```

```
        print("Visited", node.node.path)
```

類型陷阱

Python 使用動態類型，其中所有變量都可以引用任何類型的值。參數和返回值可以用類型註釋，但這些都是「提示」，並且不會強制執行。這意味著在 Python 中，很容易將不正確的值類型傳遞給 AWS CDK 構造。與其在構建過程中出現類型錯誤，就像從靜態類型語言那樣，當 JSII 層（在 Python 和 AWS CDK's TypeScript 核心之間進行轉換）無法處理意外類型時，您可能會遇到運行時錯誤。

根據我們的經驗，Python 程序員所犯的類型錯誤往往屬於這些類別。

- 傳遞一個值，其中一個構造需要一個容器（Python 列表或字典），反之亦然。
- 將與圖層 1 (`CfnXxxxxx`) 建構關聯的類型值傳遞給 L2 或 L3 建構，反之亦然。

AWS CDK Python 模塊確實包含類型註釋，因此您可以使用支持它們的工具來幫助類型。如果您不使用支援這些功能的 IDE，例如 PyCharm，您可能想要呼叫MyPy型別驗證程式作為建置程序中的步驟。還有一些運行時類型檢查器可以改善類型相關錯誤的錯誤消息。

合成和部署

您可以使用以下命令單獨或一起合成和部署在 AWS CDK 應用程序中定義的堆棧。通常，當您發出它們時，您應該位於項目的主目錄中。

- `cdk synth`：從 AWS CDK 應用程序中的一個或多個堆棧中合成 AWS CloudFormation 模板。
- `cdk deploy`：將 AWS CDK 應用程序中一個或多個堆棧定義的資源部署到 AWS。

您可以在單一命令中指定要合成或部署的多個堆疊的名稱。如果您的應用程序只定義了一個堆棧，則不需要指定它。

```
cdk synth                 # app defines single stack
cdk deploy Happy Grumpy   # app defines two or more stacks; two are deployed
```

您也可以使用萬用字元 * (任意數目的字元) 和？（任何單個字符）以按模式識別堆棧。使用萬用字元時，請將模式括在引號中。否則，shell 可能會嘗試將其擴展到當前目錄中的文件的名稱，然後再將其傳遞到 AWS CDK Toolkit。

```
cdk synth "Stack?"    # Stack1, StackA, etc.
cdk deploy "*Stack"   # PipeStack, LambdaStack, etc.
```

> **Tip**
>
> 您不需要在部署堆疊之前明確合成堆疊；請`cdk deploy`執行此步驟，以確保部署最新的程式碼。

如需`cdk`指令的完整文件，請參閱the section called “AWS CDK 工具包”。

AWS CDK 在 Java 中使用

Java 是完全支援的用戶端語言， AWS CDK 且被認為是穩定的。您可以開發 Java 中使用熟悉的工具，包括 JDK（甲骨文，或 OpenJDK 分佈，如 Amazon Corretto）和阿帕奇 Maven 的 AWS CDK 應用程序。

AWS CDK 支援 Java 8 及更新版本。但是，我們建議您使用最新版本，因為該語言的更新版本包括對於開發 AWS CDK 應用程序特別方便的改進。例如，Java 9 引入了該`Map.of()`方法（一種聲明將被寫為對象文字的哈希映射的便捷方 TypeScript法）。Java 10 使用`var`關鍵字引入了本地類型推斷。

> **Note**
>
> 本開發人員指南中的大多數代碼示例都適用於 Java 8。一些例子使用`Map.of()`; 這些示例包括註釋注意到它們需要 Java 9。

您可以使用任何文本編輯器或可以讀取 Maven 項目的 Java IDE 來處理您的應用 AWS CDK 程序。我們在本指南中提供了 Eclipse 提示，但是 IntelliJ IDEA 和其他 IDE 可以導入 Maven 項目 NetBeans，並且可以用於在 Java 中開發 AWS CDK 應用程序。

可以使用 Java 以外的 JVM 託管語言編寫 AWS CDK 應用程序（例如，Kotlin，Groovy，Clojure 或 Scala），但體驗可能不是特別慣用，我們無法為這些語言提供任何支持。

主題

- 建置、合成和部署

開始使用 Java

若要使用 AWS CDK，您必須擁有 AWS 帳戶和認證，並已安裝 Node.js 和工 AWS CDK 具組。請參閱開始使用 AWS CDK。

Java AWS CDK 應用程式需要 Java 8 (v1.8) 或更新版本。我們推薦 Amazon Corretto，但你可以使用任何 OpenJDK 分佈或甲骨文的 JDK。您還需要阿帕奇 Maven 3.5 或更高版本。您也可以使用工具，如搖籃，但由工具 AWS CDK 包生成的應用程序骨架是 Maven 項目。

> **Note**
>
> 第三方語言棄用：語言版本僅在供應商或社區共享其 EOL（生命週期終止）之前受到支持，並且如有更改，恕不另行通知。

建立專案

您可以在空目錄`cdk init`中呼叫來建立新 AWS CDK 專案。使用選`--language`項並指定`java`：

```
mkdir my-project
cd my-project
cdk init app --language java
```

`cdk init`使用專案資料夾的名稱來命名專案的各種元素，包括類別、子資料夾和檔案。資料夾名稱中的連字號會轉換為底線。但是，名稱應該遵循 Java 標識符的形式; 例如，它不應以數字開頭或包含空格。

生成的項目包括對 `software.amazon.awscdk` Maven 包的引用。它和它的依賴關係由 Maven 自動安裝。

如果您使用的是 IDE，您現在可以開啟或匯入專案。例如，在 Eclipse 中，選擇「檔案 > 匯入 > Maven > 現有的 Maven 專案」。請確定專案設定已設定為使用 Java 8 (1.8)。

管理 AWS 建構程式庫模組

使用 Maven 安裝 AWS 建構函式庫套件，這些套件位於群組中`software.amazon.awscdk`。大多數構造都在工件中`aws-cdk-lib`，默認情況下會添加到新的 Java 項目中。對於仍在開發更高級別

的 CDK 支持服務的模塊是在單獨的「實驗性」軟件包中，以其服務名稱的簡短版本（no AWS 或 Amazon 前綴）命名。搜索 Maven 中央存儲庫以查找所有 AWS CDK 和 AWS 構造模塊庫的名稱。

> **Note**
>
> CDK API 參考的 Java 版本也會顯示套件名稱。

某些服務的「AWS 建構程式庫」支援位於多個命名空間中。例如，Amazon 路線 53 的功能分為`software.amazon.awscdk.route53`、`route53-patterns`、`route53resolver`、和`route53-targets`。

主 AWS CDK 包在 Java 代碼中導入為`software.amazon.awscdk`. AWS 構造庫中各種服務的模塊住在下面`software.amazon.awscdk.services`，命名類似於它們的 Maven 包名稱。例如，Amazon S3 模塊的命名空間是`software.amazon.awscdk.services.s3`。

我們建議您為每個 Java 原始檔案中使用的每個「AWS 建構程式庫」類別撰寫個別的 Java `import` 陳述式，並避免匯入萬用字元。您始終可以在沒有`import`語句的情況下使用類型的完全限定名稱（包括其名稱空間）。

如果您的應用程序依賴於實驗包，請編輯項目`pom.xml`並在`<dependencies>`容器中添加新`<dependency>`元素。例如，下面的`<dependency>`元素指定了 CodeStar 實驗構造庫模塊：

```
<dependency>
    <groupId>software.amazon.awscdk</groupId>
    <artifactId>codestar-alpha</artifactId>
    <version>2.0.0-alpha.10</version>
</dependency>
```

> **Tip**
>
> 如果您使用 Java IDE，它可能具有用於管理 Maven 依賴關係的功能。但是，我們建議您`pom.xml`直接編輯，除非您絕對確定 IDE 的功能與您手動執行的操作相符。

管理相依性 Java

在 Java 中，依賴關係在中指定`pom.xml`並使用 Maven 安裝。`<dependencies>`容器包含每個封裝的`<dependency>`元素。以下是一個典型`pom.xml`的 CDK Java 應用程序的一部分。

```
<dependencies>
    <dependency>
        <groupId>software.amazon.awscdk</groupId>
        <artifactId>aws-cdk-lib</artifactId>
        <version>2.14.0</version>
    </dependency>
    <dependency>
        <groupId>software.amazon.awscdk</groupId>
        <artifactId>appsync-alpha</artifactId>
        <version>2.10.0-alpha.0</version>
    </dependency>
</dependencies>
```

> **Tip**
>
> 許多 Java IDE 都集成了 Maven 支持和可視化`pom.xml`編輯器，您可能會發現管理依賴關係方便。

Maven 不支持依賴鎖定。雖然您可以在中指定版本範圍`pom.xml`，但我們建議您始終使用確切的版本來保持組建可重複。

Maven 會自動安裝傳遞依賴關係，但每個軟件包只能有一個安裝的副本。POM 樹狀目錄中指定的最高版本會被選取；應用程式永遠會有安裝套件版本的最後一個字。

每當您構建（ ）或 package（mvn compile）項目時，Maven 都會自動安裝或更新您的依賴關係。mvn packageCDK 工具包會在每次運行時自動執行此操作，因此通常不需要手動調用 Maven。

AWS CDK 爪哇成語

道具

所有的 AWS 構造庫類都使用三個參數實例化：正在定義構造的範圍（它在構造樹中的父級），一個 id 和 prop，構造用於配置它創建的資源的鍵/值對的包。其他類和方法也使用「屬性包」模式作為參數。

在 Java 中，道具使用生成器模式表示。每個建構類型都有對應的 prop 類型；例如，`Bucket`建構 (代表 Amazon S3 儲存貯體) 將作為其道具的執行個體`BucketProps`。

`BucketProps`類（就像每個 AWS 構造庫 prop 類一樣）都有一個名為的內部類`Builder`。該`BucketProps.Builder`類型提供了設置`BucketProps`實例的各種屬性的方法。每個方法都

會傳回Builder執行個體，因此方法呼叫可以鏈結以設定多個屬性。在鏈的末尾，你調用實際生build()成的BucketProps對象。

```
Bucket bucket = new Bucket(this, "MyBucket", new BucketProps.Builder()
                           .versioned(true)
                           .encryption(BucketEncryption.KMS_MANAGED)
                           .build());
```

構造和其他類似道具的對象作為他們的最終參數的類，提供了一個快捷方式。該類有一個自己Builder的，它在一個步驟中實例化它和它的 prop 對象。這樣，您不需要明確實例化（例如）BucketProps和 a，並Bucket且不需要為 prop 類型導入。

```
Bucket bucket = Bucket.Builder.create(this, "MyBucket")
                           .versioned(true)
                           .encryption(BucketEncryption.KMS_MANAGED)
                           .build();
```

從現有建構衍生自己的建構時，您可能需要接受其他屬性。我們建議您遵循這些構建器模式。但是，這並不像子類化構造類那麼簡單。您必須自行提供兩個新Builder班級的移動部分。你可能更喜歡簡單地讓你的構造接受一個或多個額外的參數。當一個參數是可選的，你應該提供額外的構造函數。

泛型結構

在某些 API 中， AWS CDK 會使用 JavaScript 陣列或無類型物件做為方法的輸入。（例如，請參閱 AWS CodeBuild的BuildSpec.fromObject()方法。）在 Java 中，這些對象表示為java.util.Map<String, Object>。如果值都是字符串，則可以使用Map<String, String>。

Java 沒有提供像其他語言那樣為此類容器編寫文字的方法。在 Java 9 及更高版本中，您可java.util.Map.of()以使用其中一個調用方便地定義多達十個條目的映射。

```
java.util.Map.of(
    "base-directory", "dist",
    "files", "LambdaStack.template.json"
 )
```

若要建立包含十個以上項目的地圖，請使用java.util.Map.ofEntries()。

如果您使用的是 Java 8，則可以提供類似於這些方法的方法。

JavaScript 數組被表示為`List<Object>`或`List<String>`在 Java 中。該方法`java.util.Arrays.asList`對於定義短 `List` s 很方便。

```
List<String> cmds = Arrays.asList("cd lambda", "npm install", "npm install typescript")
```

缺少值

在 Java 中，缺少 AWS CDK 對象（如道具）中的值由表示`null`。您必須明確測試任何可能的值，`null`以確保它包含一個值，然後再對其執行任何操作。Java 沒有「語法糖」來幫助處理空值，就像其他語言一樣。在某些情況下，您可能會發現 Apache ObjectUtil 的defaultIfNull和firstNonNull有用的。或者，編寫自己的靜態輔助器方法，以便更容易處理潛在的空值並使代碼更具可讀性。

建置、合成和部署

在運行之前 AWS CDK 會自動編譯您的應用程序。但是，手動構建應用程序以檢查錯誤並運行測試可能很有用。您可以在 IDE 中執行此操作（例如，在 Eclipse 中按下 Control-B），或者在項目的根目錄中在命令提示符下發出`mvn compile`來。

`mvn test`在命令提示字元中執行您撰寫的任何測試。

您可以使用以下命令單獨或一起合成和部署在 AWS CDK 應用程序中定義的堆棧。通常，當您發出它們時，您應該位於項目的主目錄中。

- `cdk synth`：從 AWS CDK 應用程序中的一個或多個堆棧中合成 AWS CloudFormation 模板。
- `cdk deploy`：將 AWS CDK 應用程序中一個或多個堆棧定義的資源部署到 AWS。

您可以在單一命令中指定要合成或部署的多個堆疊的名稱。如果您的應用程序只定義了一個堆棧，則不需要指定它。

```
cdk synth                 # app defines single stack
cdk deploy Happy Grumpy   # app defines two or more stacks; two are deployed
```

您也可以使用萬用字元 * (任意數目的字元) 和？（任何單個字符）以按模式識別堆棧。使用萬用字元時，請將模式括在引號中。否則，shell 可能會嘗試將其擴展到當前目錄中的文件的名稱，然後再將其傳遞到 AWS CDK Toolkit。

```
cdk synth "Stack?"    # Stack1, StackA, etc.
cdk deploy "*Stack"   # PipeStack, LambdaStack, etc.
```

> **Tip**
>
> 您不需要在部署堆疊之前明確合成堆疊；請`cdk deploy`執行此步驟，以確保部署最新的程式碼。

如需`cdk`指令的完整文件，請參閱the section called “AWS CDK 工具包”。

AWS CDK 在 C# 中使用

.NET 是完全支援的用戶端語言， AWS CDK 且被認為是穩定的。C # 是我們提供示例和支持的主要 .NET 語言。您可以選擇以其他 .NET 語言撰寫 AWS CDK 應用程式，例如 Visual Basic 或 F#，但對於將這些語言與 CDK 搭配使用 AWS 提供有限的支援。

您可以使用熟悉的工具 (包括 Visual Studio、視覺工作室程式碼、`dotnet`命令和 NuGet 套件管理員) 在 C# 中開發 AWS CDK 應用程式。包含 AWS 構造庫的模塊通過 nuge t.org 分發。

我們建議在 Windows 上使用視窗 2019 (任何版本) 在 C# 中開發 AWS CDK 應用程式。

主題

- 開始使用 C#
- 建立專案
- 管理 AWS 建構程式庫模組
- 管理相依性 C#
- AWS CDK C # 中的成語
- 建置、合成和部署

開始使用 C#

若要使用 AWS CDK，您必須擁有 AWS 帳戶和認證，並已安裝 Node.js 和工 AWS CDK 具組。請參閱開始使用 AWS CDK。

C # AWS CDK 應用程序需要 .NET 核心 v3.1 或更高版本，可在這裡找到。

.NET 工具鏈包括`dotnet`用於建置和執行 .NET 應用程式以及管理 NuGet 套件的命令列工具。即使您主要在 Visual Studio 中工作，此命令對於批次作業和安裝 AWS 建構程式庫套件也很有用。

建立專案

您可以在空目錄`cdk init`中呼叫來建立新 AWS CDK 專案。使用選`--language`項並指定`csharp`：

```
mkdir my-project
cd my-project
cdk init app --language csharp
```

`cdk init`使用專案資料夾的名稱來命名專案的各種元素，包括類別、子資料夾和檔案。資料夾名稱中的連字號會轉換為底線。但是，該名稱應該遵循 C# 標識符的形式; 例如，它不應該以數字開頭或包含空格。

產生的專案包含`Amazon.CDK.Lib` NuGet 套件的參考。它及其依賴項會自動安裝 NuGet。

管理 AWS 建構程式庫模組

.NET 生態系統使用 NuGet 軟件包管理器。主要的 CDK 軟件包，其中包含核心類和所有穩定的服務構造，是. `Amazon.CDK.Lib` 實驗模塊，其中新功能正在積極開發中，命名為類似`Amazon.CDK.AWS.`*`SERVICE-NAME`*`.Alpha`，其中服務名稱是沒有 AWS 或 Amazon 前綴的簡短名稱。例如， AWS IoT 模組的 NuGet 套件名稱為`Amazon.CDK.AWS.IoT.Alpha`。如果您找不到想要的套件，請搜尋 Nuget t.org。

> **Note**
>
> CDK API 參考資料的 .NET 版本也會顯示套件名稱。

某些服務的「 AWS 建構程式庫」支援位於多個模組中。例如， AWS IoT 具有第二個名為的模組`Amazon.CDK.AWS.IoT.Actions.Alpha`。

在大多數 AWS CDK 應用程序中需要 AWS CDK的主模塊，在 C# 代碼中導入為`Amazon.CDK`. AWS 構造庫中各種服務的模塊住在下面`Amazon.CDK.AWS`。例如，Amazon S3 模塊的命名空間是`Amazon.CDK.AWS.S3`。

我們建議您撰寫 CDK 核心結構的 C# `using` 指令，以及您在每個 C# 原始檔案中使用的每個 AWS 服務。您可能會發現為命名空間或類型使用別名來幫助解決名稱衝突很方便。您始終可以在沒有語句的情況下使用類型的完全質量名稱（包括其名稱空間）。`using`

管理相依性 C#

在 C# AWS CDK 應用程式中，您可以使用 NuGet. NuGet 有四個標準，大多是等效的接口。使用適合您的需求和工作風格的一個。您還可以使用兼容的工具，例如 Paket，MyGet甚至直接編輯`.csproj`文件。

NuGet 不允許您指定相依性的版本範圍。每個依賴關係都固定到特定版本。

更新相依性之後，Visual Studio 會在您下次建置時使用 NuGet 擷取每個套件的指定版本。如果您不是使用 Visual Studio，請使用命dotnet restore令來更新您的相依性。

直接編輯專案檔案

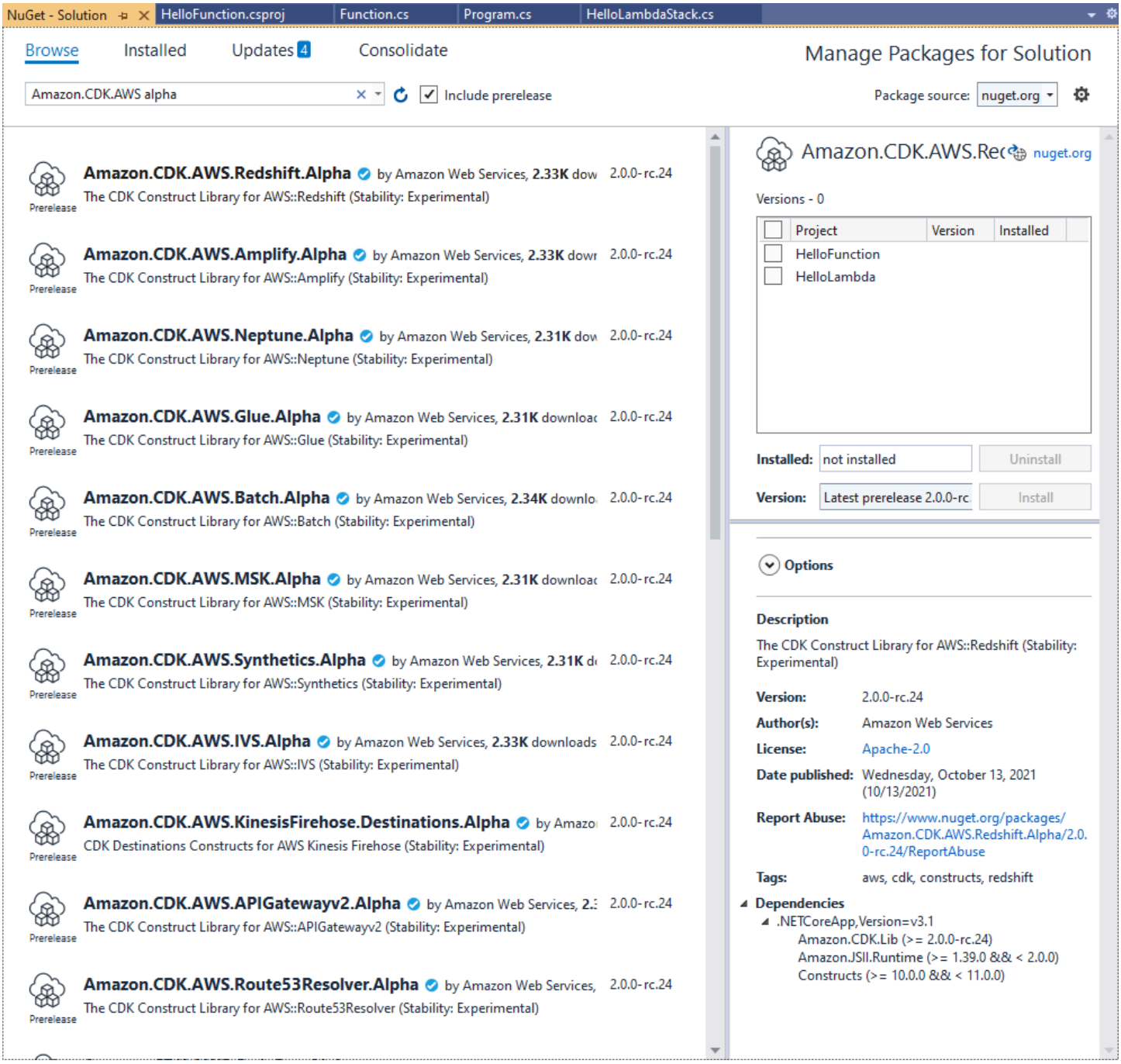
項目的`.csproj`文件包含一個`<ItemGroup>`容器，將您的依賴項列為`<PackageReference`元素。

```
<ItemGroup>
    <PackageReference Include="Amazon.CDK.Lib" Version="2.14.0" />
    <PackageReference Include="Constructs" Version="%constructs-version%" />
</ItemGroup>
```

視覺工作室 NuGet GUI

Visual Studio 的 NuGet 工具可從 [工具] > [Package 件管理員] > NuGet [管理解決方案的 NuGet 套件] 存取。使用「瀏覽」索引標籤尋找您要安裝的「 AWS 建構程式庫」套件。您可以選擇所需的版本，包括模塊的預發布版本，並將其添加到任何打開的項目中。

> **Note**
>
> 所有視為「實驗性」(請參閱the section called “版本控制”) 的「 AWS 建構程式庫」模組都會在中標記為搶鮮版， NuGet 並具有`alpha`名稱後綴。



查看更新頁面以安裝新版本的軟件包。

NuGet 控制台

主 NuGet 控台是在 Visual Studio 專案的內容中運作的 PowerShell基礎介面。 NuGet 您可以選擇「工具 > NuGet 套件管理員 > Package 件管理員主控台」，在 Visual Studio 中開啟它。如需有關使用此工具的詳細資訊，請參閱使用 Visual Studio 中的套件管理員主控台安裝及管理套件。

該dotnet命令

此命dotnet令是使用 Visual Studio C# 專案的主要命令列工具。您可以從任何 Windows 命令提示符調用它。在它的許多功能中，dotnet可以將 NuGet依賴項添加到 Visual Studio 項目。

假設您與 Visual Studio 專案 (.csproj) 檔案位於相同的目錄中，請發出如下所示的命令來安裝套件。由於創建項目時包含了主 CDK 庫，因此您只需要明確安裝實驗模塊。實驗模塊要求您指定明確的版本號。

```
dotnet add package Amazon.CDK.AWS.IoT.Alpha -v VERSION-NUMBER
```

您可以從其他目錄發出指令。若要這麼做，請在add關鍵字之後加入專案檔案或包含專案檔案的目錄路徑。下列範例假設您位於 AWS CDK 專案的主目錄中。

```
dotnet add src/PROJECT-DIR package Amazon.CDK.AWS.IoT.Alpha -v VERSION-NUMBER
```

要安裝特定版本的軟件包，請包括標-v誌和所需的版本。

若要更新套件，請發出與安裝套件相同的dotnet add指令。對於實驗模塊，再次，您必須指定一個明確的版本號。

如需有關使用dotnet命令管理套件的詳細資訊，請參閱使用 dotnet CLI 安裝及管理套件。

該nuget命令

命nuget令行工具可以安裝和更新 NuGet 軟件包。但是，它需要您的 Visual Studio 專案的設定方式與設cdk init定專案的方式不同。（技術細節：與Packages.config項目一起nuget工作，同時cdk init創建一個新風格的項PackageReference目。）

我們不建議在建立的 AWS CDK 專案中使用此nuget工具cdk init。如果您正在使用其他類型的專案，並且想要使用nuget，請參閱 NuGet CLI 參考。

AWS CDK C# 中的成語

道具

所有的 AWS 構造庫類都使用三個參數實例化：構造被定義的範圍（它在構造樹中的父級），一個 id 和 prop，構造用於配置它創建的資源的鍵/值對的包。其他類和方法也使用「屬性包」模式作為參數。

在 C# 中，道具使用道具類型表示。在慣用的 C# 方式，我們可以使用對象初始化器來設置各種屬性。在這裡，我們使用Bucket構造創建一個 Amazon S3 存儲桶; 其相應的道具類型是BucketProps。

```
var bucket = new Bucket(this, "MyBucket", new BucketProps {
    Versioned = true
});
```

> **Tip**
>
> 將該軟件包添加Amazon.JSII.Analyzers到您的項目中，以獲取在 Visual Studio 中的道具定義中檢查所需的值。

當擴展一個類或覆蓋一個方法，你可能需要接受額外的道具為自己的目的，這些道具不被父類理解。要做到這一點，子類適當的 prop 類型並添加新的屬性。

```
// extend BucketProps for use with MimeBucket
class MimeBucketProps : BucketProps {
    public string MimeType { get; set; }
}

// hypothetical bucket that enforces MIME type of objects inside it
class MimeBucket : Bucket {
     public MimeBucket( readonly Construct scope, readonly string id, readonly
 MimeBucketProps props=null) : base(scope, id, props) {
         // ...
     }
}

// instantiate our MimeBucket class
var bucket = new MimeBucket(this, "MyBucket", new MimeBucketProps {
    Versioned = true,
    MimeType = "image/jpeg"
});
```

在調用父類的初始化程序或重寫方法時，通常可以傳遞收到的 prop。新類型與其父類型相容，而且會忽略您新增的額外 prop。

的 future 版本 AWS CDK 可能會巧合地添加一個新屬性，其中包含您用於自己屬性的名稱。這不會導致使用您的構造或方法的任何技術問題（因為您的屬性沒有「上鏈」傳遞，父類或重寫的方法將只使用

默認值），但可能會導致構造的用戶混淆。您可以通過命名屬性來避免這種潛在問題，以使它們顯然屬於您的構造。如果有許多新屬性，請將它們捆綁到適當命名的類中，並將它們作為單個屬性傳遞。

泛型結構

在某些 API 中， AWS CDK 會使用 JavaScript 陣列或無類型物件做為方法的輸入。（例如，請參閱 AWS CodeBuild的`BuildSpec.fromObject()`方法。）在 C# 中，這些對象表示為`System.Collections.Generic.Dictionary<String, Object>`. 如果值是所有字符串，則可以使用`Dictionary<String, String>`。 JavaScript 數組在 C# 中表示為`object[]`或`string[]`數組類型。

> **Tip**
>
> 您可以定義短別名，以便更輕鬆地使用這些特定字典類型。
>
> ```
> using StringDict = System.Collections.Generic.Dictionary<string, string>;
> using ObjectDict = System.Collections.Generic.Dictionary<string, object>;
> ```

缺少值

在 C# 中，缺少的值，如道具的 AWS CDK 對象由表示`null`。空條件成員訪問運算符`?.`和空合併運算符很方便使用`??`這些值。

```
// mimeType is null if props is null or if props.MimeType is null
string mimeType = props?.MimeType;

// mimeType defaults to text/plain. either props or props.MimeType can be null
string MimeType = props?.MimeType ?? "text/plain";
```

建置、合成和部署

在運行之前 AWS CDK 會自動編譯您的應用程序。但是，手動構建應用程序以檢查錯誤並運行測試可能很有用。您可以通過在 Visual Studio 中按 F6 或通過`dotnet build src`從命令行發出，其中`src`是包含 Visual Studio 解決方案（`.sln`）文件的項目目錄中的目錄執行此操作。

您可以使用以下命令單獨或一起合成和部署在 AWS CDK 應用程序中定義的堆棧。通常，當您發出它們時，您應該位於項目的主目錄中。

- `cdk synth`：從 AWS CDK 應用程序中的一個或多個堆棧中合成 AWS CloudFormation 模板。

- `cdk deploy`：將 AWS CDK 應用程序中一個或多個堆棧定義的資源部署到 AWS。

您可以在單一命令中指定要合成或部署的多個堆疊的名稱。如果您的應用程序只定義了一個堆棧，則不需要指定它。

```
cdk synth                 # app defines single stack
cdk deploy Happy Grumpy   # app defines two or more stacks; two are deployed
```

您也可以使用萬用字元 * (任意數目的字元) 和? （任何單個字符）以按模式識別堆棧。使用萬用字元時，請將模式括在引號中。否則，shell 可能會嘗試將其擴展到當前目錄中的文件的名稱，然後才傳遞到 AWS CDK Toolkit。

```
cdk synth "Stack?"    # Stack1, StackA, etc.
cdk deploy "*Stack"   # PipeStack, LambdaStack, etc.
```

> **Tip**
>
> 您不需要在部署堆疊之前明確合成堆疊；請`cdk deploy`執行此步驟，以確保部署最新的程式碼。

如需`cdk`指令的完整文件，請參閱the section called “AWS CDK 工具包”。

AWS CDK 在 Go 中使用

Go 是完全支援的用戶端語言， AWS Cloud Development Kit (AWS CDK) 且被認為是穩定的。 AWS CDK 在 Go 中使用使用熟悉的工具。Go 版本的 AWS CDK 甚至使用 GO 風格的標識符。

與 CDK 支持的其他語言不同，Go 不是傳統的面向對象編程語言。Go 使用其他語言經常利用繼承的組合。我們試圖盡可能地採用慣用的 Go 方法，但有些地方 CDK 可能會有所不同。

本主題 AWS CDK 在 Go 中使用時提供指導。請參閱公告部落格文章，瞭解 AWS CDK.

主題

- AWS CDK 在圍棋成語
- 建置、合成和部署

開始使用 Go

若要使用 AWS CDK，您必須擁有 AWS 帳戶和認證，並已安裝 Node.js 和工 AWS CDK 具組。請參閱開始使用 AWS CDK。

Go 綁定 AWS CDK 使用標準 Go 工具鏈，v1.18 或更高版本。您可以使用您選擇的編輯器。

> **Note**
>
> 第三方語言棄用：語言版本僅在供應商或社區共享其 EOL（生命週期終止）之前受到支持，並且如有更改，恕不另行通知。

建立專案

您可以在空目錄`cdk init`中呼叫來建立新 AWS CDK 專案。使用選`--language`項並指定`go`：

```
mkdir my-project
cd my-project
cdk init app --language go
```

`cdk init`使用專案資料夾的名稱來命名專案的各種元素，包括類別、子資料夾和檔案。資料夾名稱中的連字號會轉換為底線。但是，名稱應該遵循 Go 標識符的形式; 例如，它不應以數字開頭或包含空格。

產生的專案包含對核心 AWS CDK Go 模組的參考`github.com/aws/aws-cdk-go/awscdk/v2`，中`go.mod`。安裝此模塊和其他必需模塊的問`go get`題。

管理 AWS 建構程式庫模組

在大多數 AWS CDK 文檔和示例中，「module」一詞通常用於指向 AWS 構造庫模塊，每個 AWS 服務一個或多個，這與術語的慣用 Go 用法不同。CDK 構造庫在一個 Go 模塊中提供，其中包含各個構造庫模塊，該模塊支持各種 AWS 服務，作為該模塊內的 Go 包提供。

某些服務的「 AWS 建構程式庫」支援位於多個「建構程式庫」模組 (Go 套件) 中。例如，Amazon Route 53 除了主awsroute53套件之外，還有三個「建構程式庫」模組 (名為awsroute53patternsawsroute53resolver、和) awsroute53targets。

在大多數 AWS CDK 應用程序中，您將需要 AWS CDK的核心軟件包在 Go 代碼中導入為github.com/aws/aws-cdk-go/awscdk/v2. AWS 構建庫中各種服務的軟件包住在下面github.com/aws/aws-cdk-go/awscdk/v2。例如，Amazon S3 模塊的命名空間是github.com/aws/aws-cdk-go/awscdk/v2/awss3。

```
import (
        "github.com/aws/aws-cdk-go/awscdk/v2/awss3"
        // ...
)
```

為要在應用程序中使用的服務導入構造庫模塊（Go 包）後，您可以使用訪問該模塊中的構造，例如，awss3.Bucket.

管理相依性 Go

在 Go 中，依賴關係版本在中定義go.mod。預設go.mod值與此處顯示的類似。

```
module my-package

go 1.16

require (
  github.com/aws/aws-cdk-go/awscdk/v2 v2.16.0
  github.com/aws/constructs-go/constructs/v10 v10.0.5
  github.com/aws/jsii-runtime-go v1.29.0
)
```

Package 名稱（模塊，在 Go 語言中）由 URL 指定，並附加了所需的版本號。Go 的模塊系統不支持版本範圍。

發出命go get令以安裝所有必需的模組並進行更新go.mod。若要查看相依性的可用更新清單，請問題go list -m -u all。

AWS CDK 在圍棋成語

欄位和方法名稱

欄位和方法名稱在 CDK 的原始語言 TypeScript中使用駱駝大小寫 (`likeThis`)。在 Go 中，這些遵循 Go 約定，帕斯卡套管（ ）也是如此。`LikeThis`

清除

在您的`main`方法中，使用`defer jsii.Close()`以確保您的 CDK 應用程序自行清理。

缺少值和指針轉換

在 Go 中， AWS CDK 對象（如屬性包）中缺少的值由表示`nil`。Go 沒有可為空的類型; 可以包含的唯一類型`nil`是一個指針。為了允許值是可選的，那麼所有 CDK 屬性，參數和返回值都是指針，即使對於原始類型也是如此。這適用於必要值以及可選值，因此，如果稍後的必要值變為可選值，則不需要中斷類型變更。

傳遞常值或運算式時，請使用下列輔助函式來建立值的指標。

- `jsii.String`
- `jsii.Number`
- `jsii.Bool`
- `jsii.Time`

為了保持一致性，我們建議您在定義自己的構造時使用類似的指針，即使例如，將構造`id`作為字符串接收而不是指向字符串的指針似乎更方便。

在處理可選 AWS CDK 值（包括原始值和複雜類型）時，您應該明確測試指針以確保它們在對它們執行任何操作`nil`之前不會出現。Go 沒有「語法糖」來幫助處理空值或缺失的值，就像其他語言一樣。但是，保證屬性包和類似結構中的所需值存在（否則構建失敗），因此不需要`nil`檢查這些值。

構造和道具

構造，它代表一個或多個 AWS 資源及其相關屬性，在 Go 中表示為接口。例如，`awss3.Bucket`是一個接口。每個構造都有一個工廠函數`awss3.NewBucket`，例如返回實現相應接口的結構。

所有的工廠函數都有三個參數：`scope`在其中定義構造（它在構造樹中的父項），一個`id`，和`props`，構造用於配置它創建的資源的鍵/值對的捆綁。「屬性包」模式也在中的其他地方使用 AWS CDK。

在 Go 中，prop 由每個構造的特定結構類型表示。例如，一個`awss3.Bucket`需要類型的 prop 引數`awss3.BucketProps`。使用結構常值來編寫 prop 參數。

```
var bucket = awss3.NewBucket(stack, jsii.String("MyBucket"), &awss3.BucketProps{
    Versioned: jsii.Bool(true),
})
```

泛型結構

在某些地方， AWS CDK 會使用 JavaScript 陣列或不具型別的物件做為方法的輸入。（例如，請參閱 AWS CodeBuild的[BuildSpec.fromObject()]方法。）在 Go 中，這些物件分別表示為切片和空白介面。

CDK 提供可變參數輔助函數，例如`jsii.Strings`用於構建包含原始類型的切片。

```
jsii.Strings("One", "Two", "Three")
```

開發自訂建構

在 Go 中，編寫一個新的構造比擴展現有構造更直接。首先，定義一個新的結構類型，如果需要類似擴展的語義，匿名嵌入一個或多個現有類型。為您要新增的任何新功能撰寫方法，以及保存所需資料所需的欄位。如果您的構造需要，請定義道具接口。最後，編寫一個工廠函數`NewMyConstruct()`來返回構造的一個實例。

如果您只是在現有構造上更改某些默認值或在實例化時添加簡單的行為，則不需要所有管道。相反，編寫一個工廠函數來調用您正在「擴展」的構造的工廠函數。例如，在其他 CDK 語言中，您可以建立一個建`TypedBucket`構，透過覆寫類型強制執行 Amazon S3 儲存貯體中的物件類型，並在新`s3.Bucket`類型的初始化設定式中新增儲存貯體政策，僅允許將指定的副檔名新增至儲存貯體。在 Go 中，簡單地編寫一個返回`NewTypedBucket`已添加適當存儲桶策略的`s3.Bucket`（實例化使用`s3.NewBucket`）更容易。沒有新的建構類型是必要的，因為這項功能已經在標準值區建構中可用；新的「建構」只是提供了一種更簡單的配置方式。

建置、合成和部署

在運行之前 AWS CDK 會自動編譯您的應用程序。但是，手動構建應用程序以檢查錯誤並運行測試可能很有用。您可以通過在項目根目錄中的命令提示符下發`go build`出來執行此操作。

`go test`在命令提示字元中執行您撰寫的任何測試。

您可以使用以下命令單獨或一起合成和部署在 AWS CDK 應用程序中定義的堆棧。通常，當您發出它們時，您應該位於項目的主目錄中。

- `cdk synth`：從 AWS CDK 應用程序中的一個或多個堆棧中合成 AWS CloudFormation 模板。
- `cdk deploy`：將 AWS CDK 應用程序中一個或多個堆棧定義的資源部署到 AWS。

您可以在單一命令中指定要合成或部署的多個堆疊的名稱。如果您的應用程序只定義了一個堆棧，則不需要指定它。

```
cdk synth                 # app defines single stack
cdk deploy Happy Grumpy   # app defines two or more stacks; two are deployed
```

您也可以使用萬用字元 * (任意數目的字元) 和? （任何單個字符）以通過模式識別堆棧。使用萬用字元時，請將模式括在引號中。否則，shell 可能會嘗試將其擴展到當前目錄中的文件的名稱，然後再將其傳遞到 AWS CDK Toolkit。

```
cdk synth "Stack?"    # Stack1, StackA, etc.
cdk deploy "*Stack"   # PipeStack, LambdaStack, etc.
```

> **Tip**
>
> 您不需要在部署堆疊之前明確合成堆疊；請`cdk deploy`執行此步驟，以確保部署最新的程式碼。

如需`cdk`指令的完整文件，請參閱the section called “AWS CDK 工具包”。

開發 AWS CDK 應用

開發 AWS Cloud Development Kit (AWS CDK) 應用程式。

主題

從建構資料庫自訂 AWS 建構

透過逸出剖面線、原始取代和自訂資源，自訂 AWS 建構資源庫中的建構。

主題

使用逃生艙口

AWS 構造庫提供了不同級別的抽象的構造。

在最高層級，您的 AWS CDK 應用程式及其中的堆疊本身就是整個雲端基礎架構的抽象概念，或其中的重要區塊。它們可以被參數化，以將它們部署在不同的環境或不同的需求。

抽象是設計和實作雲端應用程式的強大工具。您不僅可以使用其抽象來構建，還可以創建新的抽象。AWS CDK 使用現有的開放原始碼 L2 和 L3 建構做為指引，您可以建置自己的 L2 和 L3 建構，以反映您自己組織的最佳做法和意見。

沒有抽象是完美的，即使是好的抽象也不能涵蓋每個可能的用例。在開發過程中，您可能會發現幾乎符合您需求的構造，需要小型或大型定制。

因此， AWS CDK 提供了拆解建構模型的方法。這包括移動到較低級別的抽象或完全不同的模型。逃生艙口可讓您逃脫 AWS CDK 範式，並以適合您需求的方式對其進行自定義。然後，您可以將更改包裝在新構造中，以抽像出底層複雜性並為其他開發人員提供乾淨的 API。

以下是您可以使用逸出填充線的情況範例：

- AWS 服務功能可透過使用 AWS CloudFormation，但沒有 L2 建構。
- AWS 服務功能可透過使用 AWS CloudFormation，而且有 L2 結構的服務，但這些結構尚未公開此功能。由於 L2 結構是由 CDK 團隊策劃的，因此它們可能無法立即用於新功能。
- 該功能 AWS CloudFormation 根本無法通過使用。

  若要判斷功能是否可透過使用 AWS CloudFormation，請參閱AWS 資源和屬性類型參考。

開發 L1 構造的逃生艙口

如果 L2 建構不可用於服務，您可以使用自動產生的 L1 建構。這些資源可以透過名稱開頭來辨識`Cfn`，例如`CfnBucket`或`CfnRole`。您可以像使用對等 AWS CloudFormation 資源一樣實例化它們。

例如，若要在啟用分析的情況下實例化低階 Amazon S3 儲存貯體 L1，您可以撰寫類似下列內容。

TypeScript

```
new s3.CfnBucket(this, 'MyBucket', {
  analyticsConfigurations: [
    {
      id: 'Config',
      // ...
    }
  ]
});
```

JavaScript

```
new s3.CfnBucket(this, 'MyBucket', {
  analyticsConfigurations: [
    {
      id: 'Config'
      // ...
    }
  ]
});
```

Python

```
s3.CfnBucket(self, "MyBucket",
    analytics_configurations: [
        dict(id="Config",
             # ...
             )
    ]
)
```

Java

```
CfnBucket.Builder.create(this, "MyBucket")
    .analyticsConfigurations(Arrays.asList(java.util.Map.of(    // Java 9 or later
        "id", "Config", // ...
    ))).build();
```

C#

```
new CfnBucket(this, 'MyBucket', new CfnBucketProps {
    AnalyticsConfigurations = new Dictionary<string, string>
    {
        ["id"] = "Config",
        // ...
    }
});
```

在極少數情況下，您想要定義沒有對應`CfnXxx`類別的資源。這可能是尚未在資源規格中發佈的新資 AWS CloudFormation 源類型。在這種情況下，您可以`cdk.CfnResource`直接實例化並指定資源類型和屬性。如以下範例所示。

TypeScript

```
new cdk.CfnResource(this, 'MyBucket', {
  type: 'AWS::S3::Bucket',
  properties: {
    // Note the PascalCase here! These are CloudFormation identifiers.
    AnalyticsConfigurations: [
      {
        Id: 'Config',
        // ...
      }
    ]
  }
});
```

JavaScript

```
new cdk.CfnResource(this, 'MyBucket', {
  type: 'AWS::S3::Bucket',
  properties: {
    // Note the PascalCase here! These are CloudFormation identifiers.
    AnalyticsConfigurations: [
      {
        Id: 'Config'
        // ...
      }
    ]
  }
});
```

Python

```
cdk.CfnResource(self, 'MyBucket',
  type="AWS::S3::Bucket",
  properties=dict(
    # Note the PascalCase here! These are CloudFormation identifiers.
    "AnalyticsConfigurations": [
      {
        "Id": "Config",
        # ...
      }
    ]
  }
```

```
)
```

Java

```
CfnResource.Builder.create(this, "MyBucket")
        .type("AWS::S3::Bucket")
        .properties(java.util.Map.of(    // Map.of requires Java 9 or later
            // Note the PascalCase here! These are CloudFormation identifiers
            "AnalyticsConfigurations", Arrays.asList(
                    java.util.Map.of("Id", "Config", // ...
                    ))))
        .build();
```

C#

```
new CfnResource(this, "MyBucket", new CfnResourceProps
{
    Type = "AWS::S3::Bucket",
    Properties = new Dictionary<string, object>
    {   // Note the PascalCase here! These are CloudFormation identifiers
        ["AnalyticsConfigurations"] = new Dictionary<string, string>[]
        {
            new Dictionary<string, string> {
                ["Id"] = "Config"
            }
        }
    }
});
```

開發 L2 結構的逃生艙口

如果 L2 建構遺失特徵，或者您嘗試解決問題，您可以修改由 L2 建構封裝的 L1 建構。

所有 L2 建構都包含相應的 L1 建構。例如，高層`Bucket`建構會包裝低層`CfnBucket`建構。由`CfnBucket`於直接對應於 AWS CloudFormation 資源，因此它會公開透過 AWS CloudFormation提供的所有功能。

訪問 L1 構造的基本方法是使用 `construct.node.defaultChild` (Python:`default_child`)，將其轉換為正確的類型（如有必要），並修改其屬性。同樣地，讓我們以`Bucket`.

TypeScript

```
// Get the CloudFormation resource
const cfnBucket = bucket.node.defaultChild as s3.CfnBucket;

// Change its properties
cfnBucket.analyticsConfiguration = [
  {
    id: 'Config',
    // ...
  }
];
```

JavaScript

```
// Get the CloudFormation resource
const cfnBucket = bucket.node.defaultChild;

// Change its properties
cfnBucket.analyticsConfiguration = [
  {
    id: 'Config'
    // ...
  }
];
```

Python

```
# Get the CloudFormation resource
cfn_bucket = bucket.node.default_child

# Change its properties
cfn_bucket.analytics_configuration = [
    {
        "id": "Config",
        # ...
    }
]
```

Java

```
// Get the CloudFormation resource
```

```
CfnBucket cfnBucket = (CfnBucket)bucket.getNode().getDefaultChild();

cfnBucket.setAnalyticsConfigurations(
        Arrays.asList(java.util.Map.of(    // Java 9 or later
            "Id", "Config", // ...
        ));
```

C#

```
// Get the CloudFormation resource
var cfnBucket = (CfnBucket)bucket.Node.DefaultChild;

cfnBucket.AnalyticsConfigurations = new List<object> {
    new Dictionary<string, string>
    {
        ["Id"] = "Config",
        // ...
    }
};
```

您也可以使用此物件來變更 AWS CloudFormation 選項，例如`Metadata`和`UpdatePolicy`。

TypeScript

```
cfnBucket.cfnOptions.metadata = {
  MetadataKey: 'MetadataValue'
};
```

JavaScript

```
cfnBucket.cfnOptions.metadata = {
  MetadataKey: 'MetadataValue'
};
```

Python

```
cfn_bucket.cfn_options.metadata = {
    "MetadataKey": "MetadataValue"
}
```

Java

```
cfnBucket.getCfnOptions().setMetadata(java.util.Map.of(    // Java 9+
    "MetadataKey", "Metadatavalue"));
```

C#

```
cfnBucket.CfnOptions.Metadata = new Dictionary<string, object>
{
    ["MetadataKey"] = "Metadatavalue"
};
```

非逃生艙口

該 AWS CDK 還提供了上升抽象級別的能力，我們可以將其稱為「未逃脫」孵化。如果您有 L1 建構，例如`CfnBucket`，您可以建立新的 L2 建構 (`Bucket`在本例中為) 來包裝 L1 建構。

當您建立 L1 資源，但想要將其與需要 L2 資源的建構搭配使用時，這很方便。當您想要使用 L1 構造中不可用的便利方法時，`.grantXxxxx()`這也很有幫助。

您可以使用 L2 類別上的靜態方法移至較高的抽象層級，`.fromCfnXxxxx()`例如 Amazon S3 儲存貯`Bucket.fromCfnBucket()`體。L1 資源是唯一的參數。

TypeScript

```
b1 = new s3.CfnBucket(this, "buck09", { ... });
b2 = s3.Bucket.fromCfnBucket(b1);
```

JavaScript

```
b1 = new s3.CfnBucket(this, "buck09", { ...} );
b2 = s3.Bucket.fromCfnBucket(b1);
```

Python

```
b1 = s3.CfnBucket(self, "buck09", ...)
 b2 = s3.from_cfn_bucket(b1)
```

Java

```
CfnBucket b1 = CfnBucket.Builder.create(this, "buck09")
        // ....
                            .build();
IBucket b2 = Bucket.fromCfnBucket(b1);
```

C#

```
var b1 = new CfnBucket(this, "buck09", new CfnBucketProps { ... });
var v2 = Bucket.FromCfnBucket(b1);
```

從 L1 建構建立的 L2 建構是參考 L1 資源的 Proxy 物件，類似於從資源名稱、ARN 或查詢建立的物件。對這些結構的修改不會影響最終的合成 AWS CloudFormation 模板（因為您有 L1 資源，但是，您可以改為修改它）。如需代理物件的更多資訊，請參閱the section called “參考帳戶中的 AWS 資源”。

為避免混淆，請勿建立多個參考相同 L1 建構的 L2 建構。例如，如果您`CfnBucket`從上一節中的 using 技術中提取，則不應通過調`BucketBucket.fromCfnBucket()`用該`Bucket`實例來創建第二個實例`CfnBucket`。它實際上按照您的期望工作（只`AWS::S3::Bucket`有一個合成），但它使您的代碼更難維護。

原始取代

如果 L1 建構中缺少某些性質，您可以使用原始取代略過所有鍵入。這也使得可以刪除合成屬性。

使用其中一個`addOverride`方法 (Python:`add_override`) 方法，如下列範例所示。

TypeScript

```
// Get the CloudFormation resource
const cfnBucket = bucket.node.defaultChild as s3.CfnBucket;

// Use dot notation to address inside the resource template fragment
cfnBucket.addOverride('Properties.VersioningConfiguration.Status', 'NewStatus');
cfnBucket.addDeletionOverride('Properties.VersioningConfiguration.Status');

// use index (0 here) to address an element of a list
cfnBucket.addOverride('Properties.Tags.0.Value', 'NewValue');
cfnBucket.addDeletionOverride('Properties.Tags.0');
```

```
// addPropertyOverride is a convenience function for paths starting with
 "Properties."
cfnBucket.addPropertyOverride('VersioningConfiguration.Status', 'NewStatus');
cfnBucket.addPropertyDeletionOverride('VersioningConfiguration.Status');
cfnBucket.addPropertyOverride('Tags.0.Value', 'NewValue');
cfnBucket.addPropertyDeletionOverride('Tags.0');
```

JavaScript

```
// Get the CloudFormation resource
const cfnBucket = bucket.node.defaultChild ;

// Use dot notation to address inside the resource template fragment
cfnBucket.addOverride('Properties.VersioningConfiguration.Status', 'NewStatus');
cfnBucket.addDeletionOverride('Properties.VersioningConfiguration.Status');

// use index (0 here) to address an element of a list
cfnBucket.addOverride('Properties.Tags.0.Value', 'NewValue');
cfnBucket.addDeletionOverride('Properties.Tags.0');

// addPropertyOverride is a convenience function for paths starting with
 "Properties."
cfnBucket.addPropertyOverride('VersioningConfiguration.Status', 'NewStatus');
cfnBucket.addPropertyDeletionOverride('VersioningConfiguration.Status');
cfnBucket.addPropertyOverride('Tags.0.Value', 'NewValue');
cfnBucket.addPropertyDeletionOverride('Tags.0');
```

Python

```
# Get the CloudFormation resource
cfn_bucket = bucket.node.default_child

# Use dot notation to address inside the resource template fragment
cfn_bucket.add_override("Properties.VersioningConfiguration.Status", "NewStatus")
cfn_bucket.add_deletion_override("Properties.VersioningConfiguration.Status")

# use index (0 here) to address an element of a list
cfn_bucket.add_override("Properties.Tags.0.Value", "NewValue")
cfn_bucket.add_deletion_override("Properties.Tags.0")

# addPropertyOverride is a convenience function for paths starting with
 "Properties."
cfn_bucket.add_property_override("VersioningConfiguration.Status", "NewStatus")
```

```
cfn_bucket.add_property_deletion_override("VersioningConfiguration.Status")
cfn_bucket.add_property_override("Tags.0.Value", "NewValue")
cfn_bucket.add_property_deletion_override("Tags.0")
```

Java

```
// Get the CloudFormation resource
CfnBucket cfnBucket = (CfnBucket)bucket.getNode().getDefaultChild();

// Use dot notation to address inside the resource template fragment
cfnBucket.addOverride("Properties.VersioningConfiguration.Status", "NewStatus");
cfnBucket.addDeletionOverride("Properties.VersioningConfiguration.Status");

// use index (0 here) to address an element of a list
cfnBucket.addOverride("Properties.Tags.0.Value", "NewValue");
cfnBucket.addDeletionOverride("Properties.Tags.0");

// addPropertyOverride is a convenience function for paths starting with
 "Properties."
cfnBucket.addPropertyOverride("VersioningConfiguration.Status", "NewStatus");
cfnBucket.addPropertyDeletionOverride("VersioningConfiguration.Status");
cfnBucket.addPropertyOverride("Tags.0.Value", "NewValue");
cfnBucket.addPropertyDeletionOverride("Tags.0");
```

C#

```
// Get the CloudFormation resource
var cfnBucket = (CfnBucket)bucket.node.defaultChild;

// Use dot notation to address inside the resource template fragment
cfnBucket.AddOverride("Properties.VersioningConfiguration.Status", "NewStatus");
cfnBucket.AddDeletionOverride("Properties.VersioningConfiguration.Status");

// use index (0 here) to address an element of a list
cfnBucket.AddOverride("Properties.Tags.0.Value", "NewValue");
cfnBucket.AddDeletionOverride("Properties.Tags.0");

// addPropertyOverride is a convenience function for paths starting with
 "Properties."
cfnBucket.AddPropertyOverride("VersioningConfiguration.Status", "NewStatus");
cfnBucket.AddPropertyDeletionOverride("VersioningConfiguration.Status");
cfnBucket.AddPropertyOverride("Tags.0.Value", "NewValue");
cfnBucket.AddPropertyDeletionOverride("Tags.0");
```

自訂資源

如果該功能無法通過使用 AWS CloudFormation，但僅通過直接 API 調用，則必須編寫 AWS CloudFormation 自定義資源以進行所需的 API 調用。您可以使用編寫 AWS CDK 自訂資源，並將其封裝到一般建構介面中。從構造的消費者的角度來看，體驗將感覺到原生。

建置自訂資源涉及撰寫 Lambda 函數來回應資源的`CREATE`、`UPDATE`、和`DELETE`生命週期事件。如果您的自訂資源只需要進行單一 API 呼叫，請考慮使用 AwsCustomResource. 這樣可以在 AWS CloudFormation 部署期間執行任意 SDK 呼叫。否則，您應該編寫自己的 Lambda 函數來執行完成所需的工作。

主題太廣泛了，無法完全涵蓋在這裡，但是以下鏈接應該可以幫助您開始：

- 自訂資源
- 自訂資源範例
- 有關更完整的示例，請參閱 CDK 標準庫中的DnsValidatedCertificate類。這會實作為自訂資源。

從環境變數取得值

若要取得環境變數的值，請使用如下所示的程式碼。此程式碼會取得環境變數的值`MYBUCKET`。

TypeScript

```
// Sets bucket_name to undefined if environment variable not set
var bucket_name = process.env.MYBUCKET;

// Sets bucket_name to a default if env var doesn't exist
var bucket_name = process.env.MYBUCKET || "DefaultName";
```

JavaScript

```
// Sets bucket_name to undefined if environment variable not set
var bucket_name = process.env.MYBUCKET;

// Sets bucket_name to a default if env var doesn't exist
var bucket_name = process.env.MYBUCKET || "DefaultName";
```

Python

```
import os

# Raises KeyError if environment variable doesn't exist
bucket_name = os.environ["MYBUCKET"]

# Sets bucket_name to None if environment variable doesn't exist
bucket_name = os.getenv("MYBUCKET")

# Sets bucket_name to a default if env var doesn't exist
bucket_name = os.getenv("MYBUCKET", "DefaultName")
```

Java

```
// Sets bucketName to null if environment variable doesn't exist
String bucketName = System.getenv("MYBUCKET");

// Sets bucketName to a default if env var doesn't exist
String bucketName = System.getenv("MYBUCKET");
if (bucketName == null) bucketName = "DefaultName";
```

C#

```
using System;

// Sets bucket name to null if environment variable doesn't exist
string bucketName = Environment.GetEnvironmentVariable("MYBUCKET");

// Sets bucket_name to a default if env var doesn't exist
string bucketName = Environment.GetEnvironmentVariable("MYBUCKET") ?? "DefaultName";
```

使用 AWS CloudFormation 值

the section called “參數”如需將參數與配合使用的資訊，請 AWS CloudFormation 參閱 AWS CDK。

若要取得現有 AWS CloudFormation 範本中資源的參考，請參閱the section called “匯入 AWS CloudFormation 範本”。

匯入現有 AWS CloudFormation 範本

使用建構將資源轉換為 L1 `cloudformation-include.CfnInclude`建構，從 AWS CloudFormation 範本將資源匯入 AWS Cloud Development Kit (AWS CDK) 應用程式。

導入後，您可以使用應用程序中的這些資源，就像在 AWS CDK 代碼中最初定義它們的方式相同。您也可以在較高層級 AWS CDK 的建構中使用這些 L1 建構。例如，這可讓您將 L2 權限授與方法與其定義的資源搭配使用。

該`cloudformation-include.CfnInclude`構造實際上為 AWS CloudFormation 模板中的任何資源添加了一個 AWS CDK API 包裝器。使用此功能可 AWS CDK 一次將現有 AWS CloudFormation 範本匯入至片段。這樣，您可以使用 AWS CDK 建構來管理現有資源，以利用更高層級抽象的優點。您還可以使用此功能，通過提供 AWS CDK 構造 API 將 AWS CloudFormation 模板 AWS CDK 發現給開發人員。

> **Note**
>
> AWS CDK v1 還包括在內 `aws-cdk-lib.CfnInclude`，以前用於相同的一般用途。但是，它缺少的大部分功能`cloudformation-include.CfnInclude`。

主題

匯入 AWS CloudFormation 範本

以下是我們將用來提供本 AWS CloudFormation 主題範例的範例範本。複製並儲存範本，`my-template.json`以便如下所示。在完成這些範例之後，您可以使用任何現有已部署的範 AWS CloudFormation 本進一步探索。您可以從 AWS CloudFormation 控制台獲取它們。

```
{
  "Resources": {
    "MyBucket": {
```

```
      "Type": "AWS::S3::Bucket",
      "Properties": {
        "BucketName": "MyBucket",
      }
    }
  }
}
```

您可以使用 JSON 或 YAML 範本。如果可用，我們建議使用 JSON，因為 YAML 解析器在它們接受的內容上可能會略有不同。

以下是如何使用將示例模板導入您的 AWS CDK 應用程序的示例`cloudformation-include`。範本會在 CDK 堆疊的內容中匯入。

TypeScript

```
import * as cdk from 'aws-cdk-lib';
import * as cfninc from 'aws-cdk-lib/cloudformation-include';
import { Construct } from 'constructs';

export class MyStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    const template = new cfninc.CfnInclude(this, 'Template', {
      templateFile: 'my-template.json',
    });
  }
}
```

JavaScript

```
const cdk = require('aws-cdk-lib');
const cfninc = require('aws-cdk-lib/cloudformation-include');

class MyStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    const template = new cfninc.CfnInclude(this, 'Template', {
      templateFile: 'my-template.json',
    });
  }
```

```
}

module.exports = { MyStack }
```

Python

```
import aws_cdk as cdk
from aws_cdk import cloudformation_include as cfn_inc
from constructs import Construct

class MyStack(cdk.Stack):

    def __init__(self, scope: Construct, id: str, **kwargs) -> None:
        super().__init__(scope, id, **kwargs)

        template = cfn_inc.CfnInclude(self, "Template",
            template_file="my-template.json")
```

Java

```
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.cloudformation.include.CfnInclude;
import software.constructs.Construct;

public class MyStack extends Stack {
    public MyStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public MyStack(final Construct scope, final String id, final StackProps props) {
        super(scope, id, props);

        CfnInclude template = CfnInclude.Builder.create(this, "Template")
         .templateFile("my-template.json")
         .build();
    }
}
```

C#

```
using Amazon.CDK;
```

```
using Constructs;
using cfnInc = Amazon.CDK.CloudFormation.Include;

namespace MyApp
{
    public class MyStack : Stack
    {
        internal MyStack(Construct scope, string id, IStackProps props = null) :
 base(scope, id, props)
        {
            var template = new cfnInc.CfnInclude(this, "Template", new
 cfnInc.CfnIncludeProps
            {
                TemplateFile = "my-template.json"
            });
        }
    }
}
```

依預設，匯入資源會保留範本中資源的原始邏輯 ID。此行為適用於將 AWS CloudFormation 範本匯入至 AWS CDK，其中必須保留邏輯 ID。 AWS CloudFormation 需要此資訊才能將這些匯入的資源識別為 AWS CloudFormation 範本中的相同資源。

如果您正在為模板開發 AWS CDK 構造包裝器，以便其他 AWS CDK 開發人員可以使用它，請改為 AWS CDK 生成新的資源 ID。通過這樣做，該構造可以在堆棧中多次使用，而不會出現名稱衝突。若要執行此操作，請在匯入範本`false`時將`preserveLogicalIds`性質設定為。以下是範例：

TypeScript

```
const template = new cfninc.CfnInclude(this, 'MyConstruct', {
  templateFile: 'my-template.json',
  preserveLogicalIds: false
});
```

JavaScript

```
const template = new cfninc.CfnInclude(this, 'MyConstruct', {
  templateFile: 'my-template.json',
  preserveLogicalIds: false
});
```

Python

```
template = cfn_inc.CfnInclude(self, "Template",
    template_file="my-template.json",
    preserve_logical_ids=False)
```

Java

```
CfnInclude template = CfnInclude.Builder.create(this, "Template")
 .templateFile("my-template.json")
 .preserveLogicalIds(false)
 .build();
```

C#

```
var template = new cfnInc.CfnInclude(this, "Template", new cfn_inc.CfnIncludeProps
{
    TemplateFile = "my-template.json",
    PreserveLogicalIds = false
});
```

要將導入的資源放在您的 AWS CDK 應用程序的控制之下，請將堆棧添加到App：

TypeScript

```
import * as cdk from 'aws-cdk-lib';
import { MyStack } from '../lib/my-stack';

const app = new cdk.App();
new MyStack(app, 'MyStack');
```

JavaScript

```
const cdk = require('aws-cdk-lib');
const { MyStack } = require('../lib/my-stack');

const app = new cdk.App();
new MyStack(app, 'MyStack');
```

Python

```
import aws_cdk as cdk
from mystack.my_stack import MyStack

app = cdk.App()
MyStack(app, "MyStack")
```

Java

```
import software.amazon.awscdk.App;

public class MyApp {
    public static void main(final String[] args) {
        App app = new App();

        new MyStack(app, "MyStack");
    }
}
```

C#

```
using Amazon.CDK;

namespace CdkApp
{
    sealed class Program
    {
        public static void Main(string[] args)
        {
            var app = new App();
            new MyStack(app, "MyStack");
        }
    }
}
```

要驗證堆棧中的 AWS 資源不會有任何意外更改，可以執行差異。使用指 AWS CDK CLIcdk diff令並省略任何 AWS CDK特定的中繼資料。以下是範例：

```
cdk diff --no-version-reporting --no-path-metadata --no-asset-metadata
```

匯入 AWS CloudFormation 範本之後，應 AWS CDK 用程式應該會成為匯入資源的事實來源。若要變更資源，請在 AWS CDK 應用程式中修改資源，然後使用 AWS CDK CLIcdk deploy指令進行部署。

存取匯入的資源

範例程式碼template中的名稱代表匯入的 AWS CloudFormation 範本。要從中訪問資源，請使用對象的getResource()方法。要訪問返回的資源作為特定類型的資源，請將結果轉換為所需的類型。這在 Python 中不是必需的 JavaScript。以下是範例：

TypeScript

```
const cfnBucket = template.getResource('MyBucket') as s3.CfnBucket;
```

JavaScript

```
const cfnBucket = template.getResource('MyBucket');
```

Python

```
cfn_bucket = template.get_resource("MyBucket")
```

Java

```
CfnBucket cfnBucket = (CfnBucket)template.getResource("MyBucket");
```

C#

```
var cfnBucket = (CfnBucket)template.GetResource("MyBucket");
```

從這個例子中，現在cfnBucket是aws-s3.CfnBucket類的一個實例。這是代表對應 AWS CloudFormation 資源的 L1 結構。您可以將其視為其類型的任何其他資源。例如，您可以使用bucket.attrArn屬性獲取其 ARN 值。

若要改為將 L1 CfnBucket 資源包裝在 L2 aws-s3.Bucket實例中，請使用靜態方法fromBucketArn()fromBucketAttributes()、或。fromBucketName()通常情況下，該fromBucketName()方法是最方便的。以下是範例：

TypeScript

```
const bucket = s3.Bucket.fromBucketName(this, 'Bucket', cfnBucket.ref);
```

JavaScript

```
const bucket = s3.Bucket.fromBucketName(this, 'Bucket', cfnBucket.ref);
```

Python

```
bucket = s3.Bucket.from_bucket_name(self, "Bucket", cfn_bucket.ref)
```

Java

```
Bucket bucket = (Bucket)Bucket.fromBucketName(this, "Bucket", cfnBucket.getRef());
```

C#

```
var bucket = (Bucket)Bucket.FromBucketName(this, "Bucket", cfnBucket.Ref);
```

其他 L2 建構具有與從現有資源建立建構相似的方法。

當您在 L2 建構中包裝 L1 建構時，它不會建立新的資源。從我們的例子中，我們不會創建第二個 S3; 存儲桶。相反，新`Bucket`實例封裝了現有`CfnBucket`的.

從這個範例來看，現在`bucket`是 L2 `Bucket` 建構，其行為與任何其他 L2 建構類似。例如，您可以使用值區的便利`grantWrite()`方法，授予值區的 AWS Lambda 函數寫入存取權。您不必手動定義必要的 AWS Identity and Access Management (IAM) 政策。以下是範例：

TypeScript

```
bucket.grantWrite(lambdaFunc);
```

JavaScript

```
bucket.grantWrite(lambdaFunc);
```

Python

```
bucket.grant_write(lambda_func)
```

Java

```
bucket.grantWrite(lambdaFunc);
```

C#

```
bucket.GrantWrite(lambdaFunc);
```

取代參數

如果您的 AWS CloudFormation 範本包含參數，您可以在匯入時使用屬性將其取代為建置時間`parameters`值。在下列範例中，我們將`UploadBucket`參數取代為 AWS CDK 程式碼中其他位置定義的值區的 ARN。

TypeScript

```
const template = new cfninc.CfnInclude(this, 'Template', {
  templateFile: 'my-template.json',
  parameters: {
    'UploadBucket': bucket.bucketArn,
  },
});
```

JavaScript

```
const template = new cfninc.CfnInclude(this, 'Template', {
  templateFile: 'my-template.json',
  parameters: {
    'UploadBucket': bucket.bucketArn,
  },
});
```

Python

```
template = cfn_inc.CfnInclude(self, "Template",
```

```
    template_file="my-template.json",
    parameters=dict(UploadBucket=bucket.bucket_arn)
)
```

Java

```
CfnInclude template = CfnInclude.Builder.create(this, "Template")
 .templateFile("my-template.json")
 .parameters(java.util.Map.of(    // Map.of requires Java 9+
   "UploadBucket", bucket.getBucketArn()))
 .build();
```

C#

```
var template = new cfnInc.CfnInclude(this, "Template", new cfnInc.CfnIncludeProps
{
    TemplateFile = "my-template.json",
    Parameters = new Dictionary<string, string>
    {
        { "UploadBucket", bucket.BucketArn }
    }
});
```

其他模板元素

您可以匯入任何 AWS CloudFormation 範本元素，而不只是資源。匯入的元素會成為 AWS CDK 堆疊的一部分。若要匯入這些元素，請使用`CfnInclude`物件的下列方法：

- `getCondition()`— AWS CloudFormation 條件.
- `getHook()`— 用於藍色/綠色部署的 AWS CloudFormation 掛鉤。
- `getMapping()`- AWS CloudFormation 映射。
- `getOutput()`— AWS CloudFormation 輸出。
- `getParameter()`— AWS CloudFormation 參數.
- `getRule()`— AWS Service Catalog 範本的 AWS CloudFormation 規則。

這些方法中的每一個都會傳回代表特定 AWS CloudFormation 元素類型的類別執行個體。這些對象是可變的。您對它們所做的更改將顯示在從 AWS CDK 堆棧生成的模板中。以下是從範本匯入參數並修改其預設值的範例：

TypeScript

```
const param = template.getParameter('MyParameter');
param.default = "AWS CDK"
```

JavaScript

```
const param = template.getParameter('MyParameter');
param.default = "AWS CDK"
```

Python

```
param = template.get_parameter("MyParameter")
param.default = "AWS CDK"
```

Java

```
CfnParameter param = template.getParameter("MyParameter");
param.setDefaultValue("AWS CDK")
```

C#

```
var cfnBucket = (CfnBucket)template.GetResource("MyBucket");
var param = template.GetParameter("MyParameter");
param.Default = "AWS CDK";
```

巢狀堆疊

您可以在匯入巢狀堆疊的主範本時或稍後指定巢狀堆疊來匯入巢狀堆疊。巢狀範本必須儲存在本機檔案中，但作為主範本中的NestedStack資源參考。此外，程 AWS CDK 式碼中使用的資源名稱必須與主範本中巢狀堆疊使用的名稱相符。

鑑於主模板中的這個資源定義，下面的代碼演示了如何導入引用的嵌套堆棧兩種方式。

```
"NestedStack": {
  "Type": "AWS::CloudFormation::Stack",
  "Properties": {
    "TemplateURL": "https://my-s3-template-source.s3.amazonaws.com/nested-stack.json"
  }
```

TypeScript

```
// include nested stack when importing main stack
const mainTemplate = new cfninc.CfnInclude(this, 'MainStack', {
  templateFile: 'main-template.json',
  loadNestedStacks: {
    'NestedStack': {
      templateFile: 'nested-template.json',
    },
  },
});

// or add it some time after importing the main stack
const nestedTemplate = mainTemplate.loadNestedStack('NestedTemplate', {
  templateFile: 'nested-template.json',
});
```

JavaScript

```
// include nested stack when importing main stack
const mainTemplate = new cfninc.CfnInclude(this, 'MainStack', {
  templateFile: 'main-template.json',
  loadNestedStacks: {
    'NestedStack': {
      templateFile: 'nested-template.json',
    },
  },
});

// or add it some time after importing the main stack
const nestedTemplate = mainTemplate.loadNestedStack('NestedStack', {
  templateFile: 'my-nested-template.json',
});
```

Python

```
# include nested stack when importing main stack
main_template = cfn_inc.CfnInclude(self, "MainStack",
    template_file="main-template.json",
    load_nested_stacks=dict(NestedStack=
        cfn_inc.CfnIncludeProps(template_file="nested-template.json")))

# or add it some time after importing the main stack
```

```
nested_template = main_template.load_nested_stack("NestedStack",
    template_file="nested-template.json")
```

Java

```
CfnInclude mainTemplate = CfnInclude.Builder.create(this, "MainStack")
 .templateFile("main-template.json")
 .loadNestedStacks(java.util.Map.of(    // Map.of requires Java 9+
    "NestedStack", CfnIncludeProps.builder()
    .templateFile("nested-template.json").build()))
 .build();

// or add it some time after importing the main stack
IncludedNestedStack nestedTemplate = mainTemplate.loadNestedStack("NestedTemplate",
 CfnIncludeProps.builder()
  .templateFile("nested-template.json")
  .build());
```

C#

```
// include nested stack when importing main stack
var mainTemplate = new cfnInc.CfnInclude(this, "MainStack", new
 cfnInc.CfnIncludeProps
{
    TemplateFile = "main-template.json",
    LoadNestedStacks = new Dictionary<string, cfnInc.ICfnIncludeProps>
    {
        { "NestedStack", new cfnInc.CfnIncludeProps { TemplateFile = "nested-
template.json" } }
    }
});

// or add it some time after importing the main stack
var nestedTemplate = mainTemplate.LoadNestedStack("NestedTemplate", new
 cfnInc.CfnIncludeProps {
    TemplateFile = 'nested-template.json'
});
```

您可以使用其中一種方法匯入多個巢狀堆疊。匯入主範本時，您可以在每個巢狀堆疊的資源名稱與其範本檔案之間提供對應。此對映可以包含任意數量的項目。要在初始導入後執行此操作，請為每個嵌套堆棧調用`loadNestedStack()`一次。

導入嵌套堆棧後，您可以使用主模板的getNestedStack()方法訪問它。

TypeScript

```
const nestedStack = mainTemplate.getNestedStack('NestedStack').stack;
```

JavaScript

```
const nestedStack = mainTemplate.getNestedStack('NestedStack').stack;
```

Python

```
nested_stack = main_template.get_nested_stack("NestedStack").stack
```

Java

```
NestedStack nestedStack = mainTemplate.getNestedStack("NestedStack").getStack();
```

C#

```
var nestedStack = mainTemplate.GetNestedStack("NestedStack").Stack;
```

該getNestedStack()方法返回一個IncludedNestedStack實例。在這個執行個體中，您可以透過stack屬性存取 AWS CDK NestedStack執行個體，如範例所示。您還可以通過訪問原始 AWS CloudFormation 模板對象includedTemplate，從中可以加載資源和其他 AWS CloudFormation 元素。

從 Systems Manager 參數存放區取得值

AWS Cloud Development Kit (AWS CDK) 可擷取 AWS Systems Manager 參數存放區屬性的值。在合成期間， AWS CDK 會產生 AWS CloudFormation 在部署期間解析的權杖。

AWS CDK 支援擷取普通值和安全值。您可以要求任何一種價值的特定版本。對於普通值，您可以省略請求中的版本以檢索最新版本。對於安全值，您必須在請求 secure 屬性的值時指定版本。

> **Note**
>
> 本主題說明如何從 AWS Systems Manager 參數存放區讀取屬性。您也可以從 AWS Secrets Manager (請參閱從中獲取值 AWS Secrets Manager) 讀取秘密。

主題

在部署時讀取 Systems Manager 值

若要從「Systems Manager 參數存放區」讀取值，請使用valueForString參數和valueForSecureStringParameter方法。根據您想要的屬性是純字串還是安全字串值來選擇方法。這些方法返回令牌，而不是實際值。該值在部署期 AWS CloudFormation 間解析。以下是範例：

TypeScript

```
import * as ssm from 'aws-cdk-lib/aws-ssm';

// Get latest version or specified version of plain string attribute
const latestStringToken = ssm.StringParameter.valueForStringParameter(
    this, 'my-plain-parameter-name');      // latest version
const versionOfStringToken = ssm.StringParameter.valueForStringParameter(
    this, 'my-plain-parameter-name', 1);   // version 1

// Get specified version of secure string attribute
const secureStringToken = ssm.StringParameter.valueForSecureStringParameter(
    this, 'my-secure-parameter-name', 1);   // must specify version
```

JavaScript

```
const ssm = require('aws-cdk-lib/aws-ssm');

// Get latest version or specified version of plain string attribute
const latestStringToken = ssm.StringParameter.valueForStringParameter(
    this, 'my-plain-parameter-name');      // latest version
const versionOfStringToken = ssm.StringParameter.valueForStringParameter(
```

```
    this, 'my-plain-parameter-name', 1);   // version 1

// Get specified version of secure string attribute
const secureStringToken = ssm.StringParameter.valueForSecureStringParameter(
    this, 'my-secure-parameter-name', 1);   // must specify version
```

Python

```
import aws_cdk.aws_ssm as ssm

# Get latest version or specified version of plain string attribute
latest_string_token = ssm.StringParameter.value_for_string_parameter(
    self, "my-plain-parameter-name")
latest_string_token = ssm.StringParameter.value_for_string_parameter(
    self, "my-plain-parameter-name", 1)

# Get specified version of secure string attribute
secure_string_token = ssm.StringParameter.value_for_secure_string_parameter(
    self, "my-secure-parameter-name", 1)   # must specify version
```

Java

```
import software.amazon.awscdk.services.ssm.StringParameter;

//Get latest version or specified version of plain string attribute
String latestStringToken = StringParameter.valueForStringParameter(
            this, "my-plain-parameter-name");       // latest version
String versionOfStringToken = StringParameter.valueForStringParameter(
            this, "my-plain-parameter-name", 1);    // version 1

//Get specified version of secure string attribute
String secureStringToken = StringParameter.valueForSecureStringParameter(
            this, "my-secure-parameter-name", 1);   // must specify version
```

C#

```
using Amazon.CDK.AWS.SSM;

// Get latest version or specified version of plain string attribute
var latestStringToken = StringParameter.ValueForStringParameter(
    this, "my-plain-parameter-name");      // latest version
var versionOfStringToken = StringParameter.ValueForStringParameter(
```

```
    this, "my-plain-parameter-name", 1);   // version 1

// Get specified version of secure string attribute
var secureStringToken = StringParameter.ValueForSecureStringParameter(
    this, "my-secure-parameter-name", 1);   // must specify version
```

目前支援此功能的 AWS 服務數量有限。

在合成時讀取 Systems Manager 值

有時，在合成時提供參數很有用。如此一來， AWS CloudFormation 範本將永遠使用相同的值，而不是在部署期間解析值。

若要在合成時從「Systems Manager 參數存放區」讀取值，請使用valueFromLookup方法 (Python:`value_from_lookup`)。此方法返回參數的實際值作為the section called “Context”值。如果值尚未快取`cdk.json`或在命令列上傳遞，則會從目前的 AWS 帳戶擷取該值。因此，必須使用明確的 AWS 環境資訊來合成堆疊。

以下是範例：

TypeScript

```
import * as ssm from 'aws-cdk-lib/aws-ssm';

const stringValue = ssm.StringParameter.valueFromLookup(this, 'my-plain-parameter-
name');
```

JavaScript

```
const ssm = require('aws-cdk-lib/aws-ssm');

const stringValue = ssm.StringParameter.valueFromLookup(this, 'my-plain-parameter-
name');
```

Python

```
import aws_cdk.aws_ssm as ssm

string_value = ssm.StringParameter.value_from_lookup(self, "my-plain-parameter-
name")
```

Java

```
import software.amazon.awscdk.services.ssm.StringParameter;

String stringValue = StringParameter.valueFromLookup(this, "my-plain-parameter-
name");
```

C#

```
using Amazon.CDK.AWS.SSM;

var stringValue = StringParameter.ValueFromLookup(this, "my-plain-parameter-name");
```

只有普通的 Systems Manager 字符串可以被檢索。無法擷取安全字串。將始終返回最新版本。無法要求特定版本。

> **Important**
>
> 檢索到的值將最終在您的合成 AWS CloudFormation 模板中。這可能是一個安全風險，具體取決於誰可以訪問您的 AWS CloudFormation 模板以及它是什麼樣的價值。一般而言，請勿將此功能用於您想要保密的密碼、金鑰或其他值。

將值寫入 Systems Manager

您可以使用 AWS CLI AWS Management Console、或 AWS SDK 來設定 Systems Manager 參數值。下列範例使用 ssm 放入參數 CLI 命令。

```
aws ssm put-parameter --name "parameter-name" --type "String" --value "parameter-value"
aws ssm put-parameter --name "secure-parameter-name" --type "SecureString" --value
 "secure-parameter-value"
```

當更新已存在的 SSM 值時，還要包括選`--overwrite`項。

```
aws ssm put-parameter --overwrite --name "parameter-name" --type "String" --value
 "parameter-value"
aws ssm put-parameter --overwrite --name "secure-parameter-name" --type "SecureString"
 --value "secure-parameter-value"
```

從中獲取值 AWS Secrets Manager

若要在應用程式 AWS Secrets Manager 中使 AWS CDK 用值，請使用 fromSecretAttributes() 方法。它代表從 Secrets Manager 擷取並在 AWS CloudFormation 部署時使用的值。以下是範例：

TypeScript

```
import * as sm from "aws-cdk-lib/aws-secretsmanager";

export class SecretsManagerStack extends cdk.Stack {
  constructor(scope: cdk.App, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    const secret = sm.Secret.fromSecretAttributes(this, "ImportedSecret", {
      secretCompleteArn:
        "arn:aws:secretsmanager:<region>:<account-id-number>:secret:<secret-name>-
<random-6-characters>"
      // If the secret is encrypted using a KMS-hosted CMK, either import or
 reference that key:
      // encryptionKey: ...
    });
```

JavaScript

```
const sm = require("aws-cdk-lib/aws-secretsmanager");

class SecretsManagerStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    const secret = sm.Secret.fromSecretAttributes(this, "ImportedSecret", {
      secretCompleteArn:
        "arn:aws:secretsmanager:<region>:<account-id-number>:secret:<secret-name>-
<random-6-characters>"
      // If the secret is encrypted using a KMS-hosted CMK, either import or
 reference that key:
      // encryptionKey: ...
    });
  }
}

module.exports = { SecretsManagerStack }
```

Python

```
import aws_cdk.aws_secretsmanager as sm

class SecretsManagerStack(cdk.Stack):
    def __init__(self, scope: cdk.App, id: str, **kwargs):
      super().__init__(scope, name, **kwargs)

      secret = sm.Secret.from_secret_attributes(self, "ImportedSecret",
          secret_complete_arn="arn:aws:secretsmanager:<region>:<account-id-
number>:secret:<secret-name>-<random-6-characters>",
          # If the secret is encrypted using a KMS-hosted CMK, either import or
 reference that key:
          # encryption_key=....
      )
```

Java

```
import software.amazon.awscdk.services.secretsmanager.Secret;
import software.amazon.awscdk.services.secretsmanager.SecretAttributes;

public class SecretsManagerStack extends Stack {
    public SecretsManagerStack(App scope, String id) {
        this(scope, id, null);
    }

    public SecretsManagerStack(App scope, String id, StackProps props) {
        super(scope, id, props);

        Secret secret = (Secret)Secret.fromSecretAttributes(this, "ImportedSecret",
 SecretAttributes.builder()
            .secretCompleteArn("arn:aws:secretsmanager:<region>:<account-id-
number>:secret:<secret-name>-<random-6-characters>")
             // If the secret is encrypted using a KMS-hosted CMK, either import or
 reference that key:
             // .encryptionKey(...)
             .build());
    }
}
```

C#

```
using Amazon.CDK.AWS.SecretsManager;
```

```
public class SecretsManagerStack : Stack
{
    public SecretsManagerStack(App scope, string id, StackProps props) : base(scope,
 id, props) {

        var secret = Secret.FromSecretAttributes(this, "ImportedSecret", new
 SecretAttributes {
            SecretCompleteArn = "arn:aws:secretsmanager:<region>:<account-id-
number>:secret:<secret-name>-<random-6-characters>"
            // If the secret is encrypted using a KMS-hosted CMK, either import or
 reference that key:
            // encryptionKey = ...,
        });
    }
```

Tip

使用建 AWS CLI 立機密 CLI 指令從命令列建立密碼，例如在測試時：

```
aws secretsmanager create-secret --name ImportedSecret --secret-string
 mygroovybucket
```

命令會傳回 ARN，您可以在上述範例中使用。

建立`Secret`執行個體之後，您就可以從執行個體的`secretValue`屬性取得密碼值。該值由`SecretValue`實例表示，這是一種特殊類型的the section called “代幣”。因為它是一個令牌，它只有在解決之後才有意義。您的 CDK 應用程式不需要存取其實際值。相反，應用程序可以將`SecretValue`實例（或其字符串或數字表示）傳遞給需要該值的任何 CDK 方法。

設定 CloudWatch 鬧鐘

使用 aws-cloudwatch 套件在指標上設定 Amazon CloudWatch 警示。 CloudWatch 您可以使用預先定義的指標或建立自己的指標。

主題

- 使用現有的量度
- 建立您自己的指標

- 建立鬧鐘

使用現有的量度

許多「 AWS 建構程式庫」模組可讓您在具有度量的物件執行個體上，將度量的名稱傳送至便利方法，以便在現有度量上設定警示。例如，假設 Amazon SQS 佇列，您可以ApproximateNumberOfMessagesVisible從佇列的指標 () 方法取得指標：

TypeScript

```
const metric = queue.metric("ApproximateNumberOfMessagesVisible");
```

JavaScript

```
const metric = queue.metric("ApproximateNumberOfMessagesVisible");
```

Python

```
metric = queue.metric("ApproximateNumberOfMessagesVisible")
```

Java

```
Metric metric = queue.metric("ApproximateNumberOfMessagesVisible");
```

C#

```
var metric = queue.Metric("ApproximateNumberOfMessagesVisible");
```

建立您自己的指標

如下所示建立您自己的指標，其中命名空間值應與 Amazon SQS 佇列的 AWS/SQS 類似。您還需要指定指標的名稱和維度：

TypeScript

```
const metric = new cloudwatch.Metric({
```

```
  namespace: 'MyNamespace',
  metricName: 'MyMetric',
  dimensionsMap: { MyDimension: 'MyDimensionValue' }
});
```

JavaScript

```
const metric = new cloudwatch.Metric({
  namespace: 'MyNamespace',
  metricName: 'MyMetric',
  dimensionsMap: { MyDimension: 'MyDimensionValue' }
});
```

Python

```
metric = cloudwatch.Metric(
    namespace="MyNamespace",
    metric_name="MyMetric",
    dimensionsMap=dict(MyDimension="MyDimensionValue")
)
```

Java

```
Metric metric = Metric.Builder.create()
        .namespace("MyNamespace")
        .metricName("MyMetric")
        .dimensionsMap(java.util.Map.of(    // Java 9 or later
            "MyDimension", "MyDimensionValue"))
        .build();
```

C#

```
var metric = new Metric(this, "Metric", new MetricProps
{
    Namespace = "MyNamespace",
    MetricName = "MyMetric",
    Dimensions = new Dictionary<string, object>
    {
        { "MyDimension", "MyDimensionValue" }
    }
});
```

建立鬧鐘

一旦您擁有指標 (既有指標或您定義的量度)，您就可以建立警示。在此範例中，如果在最後三個評估期間的兩個中有 100 個以上的量度，就會引發警示。您可以通過comparisonOperator屬性使用比較，例如在警報中小於。G reater-than-or-equal-To 是 AWS CDK 預設值，所以我們不需要指定它。

TypeScript

```
const alarm = new cloudwatch.Alarm(this, 'Alarm', {
  metric: metric,
  threshold: 100,
  evaluationPeriods: 3,
  datapointsToAlarm: 2,
});
```

JavaScript

```
const alarm = new cloudwatch.Alarm(this, 'Alarm', {
  metric: metric,
  threshold: 100,
  evaluationPeriods: 3,
  datapointsToAlarm: 2
});
```

Python

```
alarm = cloudwatch.Alarm(self, "Alarm",
    metric=metric,
    threshold=100,
    evaluation_periods=3,
    datapoints_to_alarm=2
)
```

Java

```
import software.amazon.awscdk.services.cloudwatch.Alarm;
import software.amazon.awscdk.services.cloudwatch.Metric;

Alarm alarm = Alarm.Builder.create(this, "Alarm")
        .metric(metric)
        .threshold(100)
```

```
        .evaluationPeriods(3)
        .datapointsToAlarm(2).build();
```

C#

```
var alarm = new Alarm(this, "Alarm", new AlarmProps
{
    Metric = metric,
    Threshold = 100,
    EvaluationPeriods = 3,
    DatapointsToAlarm = 2
});
```

另一種建立警示的方法是使用指標的 createAlarm () 方法，該方法與建構函式具有基本相同的Alarm屬性。您不需要傳遞指標，因為它已經知道。

TypeScript

```
metric.createAlarm(this, 'Alarm', {
  threshold: 100,
  evaluationPeriods: 3,
  datapointsToAlarm: 2,
});
```

JavaScript

```
metric.createAlarm(this, 'Alarm', {
  threshold: 100,
  evaluationPeriods: 3,
  datapointsToAlarm: 2,
});
```

Python

```
metric.create_alarm(self, "Alarm",
    threshold=100,
    evaluation_periods=3,
    datapoints_to_alarm=2
)
```

Java

```
metric.createAlarm(this, "Alarm", new CreateAlarmOptions.Builder()
        .threshold(100)
        .evaluationPeriods(3)
        .datapointsToAlarm(2)
        .build());
```

C#

```
metric.CreateAlarm(this, "Alarm", new CreateAlarmOptions
{
    Threshold = 100,
    EvaluationPeriods = 3,
    DatapointsToAlarm = 2
});
```

儲存和擷取上下文變數值

您可以使用 AWS Cloud Development Kit (AWS CDK) CLI或在檔案中指定上下cdk.json文變數。然後，使用該TryGetContext方法檢索值。

主題

- 指定上下文變量
- 檢索上下文變量值

指定上下文變量

您可以將上下文變數指定為指 AWS CDK CLI令的一部分，也可以在中指定cdk.json。

若要建立命令列內容變數，請使用 --context (-c) 選項，如下列範例所示。

```
cdk synth -c bucket_name=mygroovybucket
```

若要在檔案中指定相同的上下cdk.json文變數和值，請使用下列程式碼。

```
{
```

```
  "context": {
    "bucket_name": "myotherbucket"
  }
}
```

如果您同時使用 AWS CDK CLI和`cdk.json`檔案指定前後關聯變數，則 AWS CDK CLI值優先。

檢索上下文變量值

要獲取應用程序中上下文變量的值，請在構造的上下文中使用該`TryGetContext`方法。（也就是說`this`，何時或`self`在 Python 中，是某個構造的一個實例。）

在這個例子中，我們檢索`bucket_name`上下文變量的值。如果未定義請求的值，則`TryGetContext`返回`undefined`（`None`在 Python 中; `null` 在 Java 和 C# `nil` 中; 在 Go 中），而不是引發異常。

TypeScript

```
const bucket_name = this.node.tryGetContext('bucket_name');
```

JavaScript

```
const bucket_name = this.node.tryGetContext('bucket_name');
```

Python

```
bucket_name = self.node.try_get_context("bucket_name")
```

Java

```
String bucketName = (String)this.getNode().tryGetContext("bucket_name");
```

C#

```
var bucketName = this.Node.TryGetContext("bucket_name");
```

在構造的上下文之外，您可以從應用程序對像訪問上下文變量，如下所示。

TypeScript

```
const app = new cdk.App();
const bucket_name = app.node.tryGetContext('bucket_name')
```

JavaScript

```
const app = new cdk.App();
const bucket_name = app.node.tryGetContext('bucket_name');
```

Python

```
app = cdk.App()
bucket_name = app.node.try_get_context("bucket_name")
```

Java

```
App app = App();
String bucketName = (String)app.getNode().tryGetContext("bucket_name");
```

C#

```
app = App();
var bucketName = app.Node.TryGetContext("bucket_name");
```

如需使用上下文變數的詳細資訊，請參閱the section called “Context”。

使用 AWS CloudFormation 公共註冊處的資源

AWS CloudFormation 公用登錄可讓您管理公用和私有擴充功能，例如資源、模組和掛接，可在 AWS 帳戶. 您可以在 AWS Cloud Development Kit (AWS CDK) 應用程式中搭配`CfnResource`建構使用公用資源擴充功能。

若要深入瞭解 AWS CloudFormation 公用登錄，請參閱使用AWS CloudFormation 者指南中的使用 AWS CloudFormation 登錄。

所有由發佈的公開擴充功能 AWS 都可供所有區域中的所有帳戶使用，您無須採取任何動作。但是，您必須在每個帳戶和要使用的區域中激活要使用的每個第三方擴展程序。

> **Note**
>
> 當您 AWS CloudFormation 搭配第三方資源類型使用時，會產生費用。費用取決於您每月執行的處理常式作業數目和處理常式作業持續時間。詳情請參閱CloudFormation 定價。

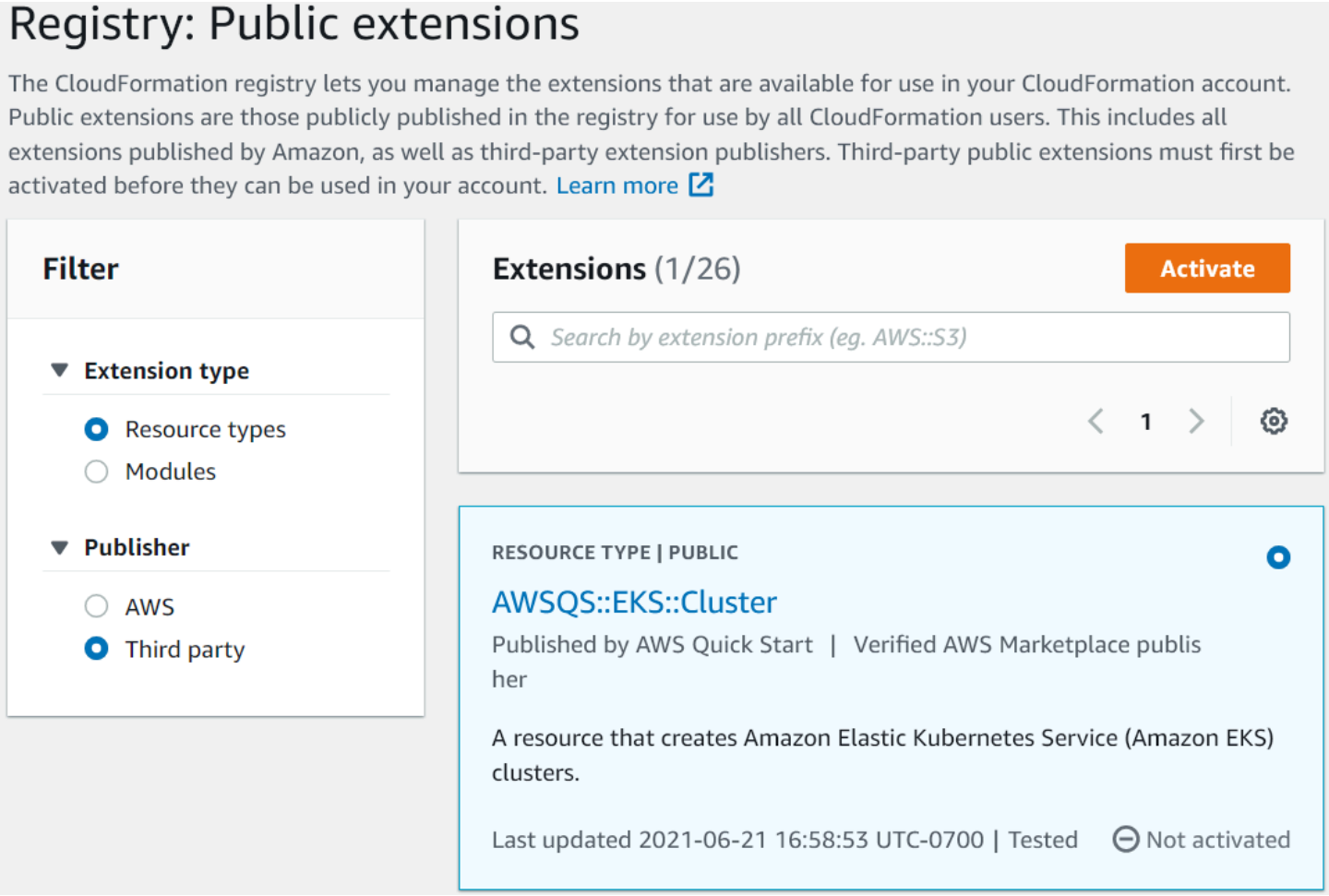
若要深入了解公用擴充功能，請參閱使用AWS CloudFormation 者指南 CloudFormation中的使用公用擴充功能

主題

在您的帳戶和區域中啟用第三方資源

發佈的擴充功能 AWS 不需要啟用。它們始終在每個帳戶和區域中可用。您可以透過 AWS Management Console、透過或部署特殊 AWS CloudFormation 資源來啟動協力廠商擴充功能。 AWS Command Line Interface

若要透過啟用第三方擴充功能， AWS Management Console 或查看有哪些資源可用



1. 登入您要使用擴充功能的 AWS 帳戶，然後切換至您要使用該擴充功能的地區。
2. 透過 [服務] 功能表導覽至 CloudFormation 主控台。
3. 選擇導覽列上的 [公開擴充功能]，然後啟用 [發行者] 下方的 [第三方] 圓鈕。可用的第三方公開擴充功能清單隨即出現。您也可AWS以選擇查看由發佈的公開擴充功能清單 AWS，不過您不需要啟用它們。)
4. 瀏覽列表並找到您要激活的擴展程序。或者，搜索它，然後激活擴展程序卡右上角的單選按鈕。
5. 選擇清單頂端的「啟用」按鈕，以啟動選取的擴充功能。隨即顯示擴充功能的「啟用」頁面
6. 在「啟動」頁面中，您可以覆寫擴充功能的預設名稱，並指定執行角色和記錄日誌組態。您也可以選擇是否在發行新版本時自動更新擴充功能。根據需要設定這些選項後，請選擇頁面底部的「啟用延伸功能」。

若要使用啟用第三方擴充功能 AWS CLI

- 使用 `activate-type` 命令。在指示的位置替換要使用的自定義類型的 ARN。

  以下是範例：

  ```
  aws cloudformation activate-type --public-type-arn public_extension_ARN --auto-update-activated
  ```

透過 CloudFormation 或 CDK 啟用第三方擴充功能

- 部署類型的資源`AWS::CloudFormation::TypeActivation`並指定下列屬性：

  a. `TypeName`-類型的名稱，例如`AWSQS::EKS::Cluster`。
  b. `MajorVersion`-您想要的擴充功能的主要版本號碼。如果您想要最新版本，請省略。
  c. `AutoUpdate`-是否在發布者發布新的次要版本時自動更新此擴展程序。（主要版本更新需要明確更改`MajorVersion`屬性。）
  d. `ExecutionRoleArn`-執行此擴充功能之 IAM 角色的 ARN。
  e. `LoggingConfig`-擴充功能的記錄組態。

  該`TypeActivation`資源可以由 CDK 使用`CfnResource`構造來部署。下一節中會針對實際擴充功能顯示此項目。

將資源從 AWS CloudFormation 公共註冊表添加到您的 CDK 應用程序

使用該`CfnResource`構造將來自 AWS CloudFormation 公共註冊表的資源包含在您的應用程序中。此構造位於 CDK 的`aws-cdk-lib`模塊中。

例如，假設您要在 AWS CDK 應用程式中`MY::S5::UltimateBucket`使用名為的公用資源。此資源採用一個屬性：值區名稱。相應的`CfnResource`實例化看起來像這樣。

TypeScript

```
const ubucket = new CfnResource(this, 'MyUltimateBucket', {
    type: 'MY::S5::UltimateBucket::MODULE',
    properties: {
           BucketName: 'UltimateBucket'
    }
```

```
});
```

JavaScript

```
const ubucket = new CfnResource(this, 'MyUltimateBucket', {
    type: 'MY::S5::UltimateBucket::MODULE',
    properties: {
           BucketName: 'UltimateBucket'
    }
});
```

Python

```
ubucket = CfnResource(self, "MyUltimateBucket",
            type="MY::S5::UltimateBucket::MODULE",
            properties=dict(
                BucketName="UltimateBucket"))
```

Java

```
CfnResource.Builder.create(this, "MyUltimateBucket")
 .type("MY::S5::UltimateBucket::MODULE")
 .properties(java.util.Map.of(    // Map.of requires Java 9+
     "BucketName", "UltimateBucket"))
 .build();
```

C#

```
new CfnResource(this, "MyUltimateBucket", new CfnResourceProps
{
    Type = "MY::S5::UltimateBucket::MODULE",
    Properties = new Dictionary<string, object>
    {
        ["BucketName"] = "UltimateBucket"
    }
});
```

部署 AWS CDK 應用

部署 AWS Cloud Development Kit (AWS CDK) 應用程式。

主題

AWS CDK 合成時間的原則驗證

主題

合成時的政策驗證

如果您或您的組織使用任何原則驗證工具 (例如AWS CloudFormation Guard或 OPA) 來定義 AWS CloudFormation 範本的限制，您可以將它們與 AWS CDK 在合成時整合。通過使用適當的策略驗證插件，您可以使 AWS CDK 應用程序在合成後立即根據策略檢查生成的 AWS CloudFormation 模板。如果有任何違規，合成將失敗，並將報告打印到控制台。

在合成階段執行的 AWS CDK 驗證會驗證控制項在部署生命週期的某個時間點，但它們不會影響在合成之外發生的動作。範例包括直接在主控台或透過服務 API 執行的動作。他們不耐合成後 AWS CloudFormation 模板的改變。其他一些更具權威性驗證相同規則集的機制應該獨立設置，如AWS CloudFormation 鉤子或. AWS Config 不過，在開發期間評估規則集的能力仍然很有用，因為它可 AWS CDK 以提高偵測速度和開發人員生產力。

AWS CDK 政策驗證的目標是盡量減少開發過程中所需的設置量，並使其盡可能簡單。

> **Note**
>
> 此功能被認為是實驗性的，並且插件 API 和驗證報告的格式 future 可能會發生變化。

主題

適用於應用程式

若要在應用程式中使用一或多個驗證外掛程式，請使用下列policyValidationBeta1屬性Stage：

```
import { CfnGuardValidator } from '@cdklabs/cdk-validator-cfnguard';
const app = new App({
  policyValidationBeta1: [
    new CfnGuardValidator()
  ],
});
// only apply to a particular stage
const prodStage = new Stage(app, 'ProdStage', {
  policyValidationBeta1: [...],
});
```

合成後，立即以這種方式註冊的所有插件將被調用，以驗證您定義的範圍內生成的所有模板。特別是，如果您在App對象中註冊模板，則所有模板都將受到驗證。

> **Warning**
>
> 除了修改雲端組件外，外掛程式還可以執行 AWS CDK 應用程式所能執行的任何操作。他們可以從文件系統讀取數據，訪問網絡等。作為插件的消費者，您有責任驗證其使用是否安全。

AWS CloudFormation Guard 插件

使用CfnGuardValidator外掛程式可讓您用AWS CloudFormation Guard來執行政策驗證。該CfnGuardValidator插件帶有一組內置的AWS Control Tower 主動控件。目前的規則集可以在專案文件中找到。如中所述合成時的政策驗證，我們建議組織使用AWS CloudFormation 掛接設定更具權威性的驗證方法。

對於AWS Control Tower客戶而言，這些相同的主動式控制可以在您的組織中部署。當您在 AWS Control Tower 環境中啟用 AWS Control Tower 主動式控制時，控制項可以停止部署不符合規範的資源，透過 AWS CloudFormation. 如需受管主動式控制及其運作方式的詳細資訊，請參閱AWS Control Tower 文件。

這些 AWS CDK 捆綁的控制項和託管的 AWS Control Tower 主動控制項最適合一起使用。在這種情況下，您可以使用在 AWS Control Tower 雲端環境中使用的相同主動式控制項來設定此驗證外掛程式。然後，您可以快速獲得 AWS CDK 應用程序將通過`cdk synth`本地運行來傳遞 AWS Control Tower 控件的信心。

驗證報告

當您合成 AWS CDK 應用程序時，將調用驗證器插件，並打印結果。下面顯示了一個示例報告。

```
Validation Report (CfnGuardValidator)
-------------------------------------
(Summary)
#####################################
# Status    # failure               #
#####################################
# Plugin    # CfnGuardValidator     #
#####################################
(Violations)
Ensure S3 Buckets are encrypted with a KMS CMK (1 occurrences)
Severity: medium
  Occurrences:

    - Construct Path: MyStack/MyCustomL3Construct/Bucket
    - Stack Template Path: ./cdk.out/MyStack.template.json
    - Creation Stack:
        ###  MyStack (MyStack)
             # Library: aws-cdk-lib.Stack
             # Library Version: 2.50.0
             # Location: Object.<anonymous> (/home/johndoe/tmp/cdk-tmp-app/src/
main.ts:25:20)
             ###  MyCustomL3Construct (MyStack/MyCustomL3Construct)
                  # Library: N/A - (Local Construct)
                  # Library Version: N/A
                  # Location: new MyStack (/home/johndoe/tmp/cdk-tmp-app/src/
main.ts:15:20)
                  ###  Bucket (MyStack/MyCustomL3Construct/Bucket)
                       # Library: aws-cdk-lib/aws-s3.Bucket
                       # Library Version: 2.50.0
                       # Location: new MyCustomL3Construct (/home/johndoe/tmp/cdk-tmp-
app/src/main.ts:9:20)
    - Resource Name: my-bucket
    - Locations:
```

```
      > BucketEncryption/ServerSideEncryptionConfiguration/0/
ServerSideEncryptionByDefault/SSEAlgorithm
  Recommendation: Missing value for key `SSEAlgorithm` - must specify `aws:kms`
  How to fix:
    > Add to construct properties for `cdk-app/MyStack/Bucket`
      `encryption: BucketEncryption.KMS`

Validation failed. See above reports for details
```

依預設，報告將以人類可讀的格式列印。如果您想要 JSON 格式的報告，請使用 CLI 或將其直接傳遞給應用程式來啟用它：`@aws-cdk/core:validationReportJson`

```
const app = new App({
  context: { '@aws-cdk/core:validationReportJson': true },
});
```

或者，您也可以使用專案目錄中的`cdk.json`或`cdk.context.json`檔案來設定此內容鍵值配對 (請參閱執行期內容)。

如果您選擇 JSON 格式，則會 AWS CDK 將政策驗證報告列印到雲端組件目錄`policy-validation-report.json`中名為的檔案。對於預設的人類可讀格式，報告將列印至標準輸出。

對於插件作者

外掛程式

AWS CDK 核心框架負責註冊和調用插件，然後顯示格式化的驗證報告。該插件的責任是充當 AWS CDK 框架和策略驗證工具之間的翻譯層。可以使用支援的任何語言建立外掛程式 AWS CDK。如果您正在建立可能會被多種語言使用的外掛程式，建議您在中建立外掛程式，以`TypeScript`便您可以使用 JSII 以每 AWS CDK 種語言發佈外掛程式。

創建插件

AWS CDK 核心模組與原則工具之間的通訊協定由`IPolicyValidationPluginBeta1`介面定義。要創建一個新的插件，你必須編寫一個實現這個接口的類。您需要實現兩件事：插件名稱（通過覆蓋`name`屬性）和方`validate()`法。

該框架將調用`validate()`，傳遞一個`IValidationContextBeta1`對象。要驗證的範本位置由下式給出`templatePaths`。外掛程式應傳回的執行個體`ValidationPluginReportBeta1`。該對象表示用戶將在合成結束時收到的報告。

```
validate(context: IPolicyValidationContextBeta1): PolicyValidationReportBeta1 {
  // First read the templates using context.templatePaths...
  // ...then perform the validation, and then compose and return the report.
  // Using hard-coded values here for better clarity:
  return {
    success: false,
    violations: [{
      ruleName: 'CKV_AWS_117',
      description: 'Ensure that AWS Lambda function is configured inside a VPC',
      fix: 'https://docs.bridgecrew.io/docs/ensure-that-aws-lambda-function-is-
configured-inside-a-vpc-1',
      violatingResources: [{
        resourceName: 'MyFunction3BAA72D1',
        templatePath: '/home/johndoe/myapp/cdk.out/MyService.template.json',
        locations: 'Properties/VpcConfig',
      }],
    }],
  };
}
```

請注意，外掛程式不允許修改雲端組件中的任何內容。任何嘗試這樣做都會導致合成失敗。

如果您的插件依賴於外部工具，請記住，某些開發人員可能尚未在其工作站中安裝該工具。為了最大程度地減少摩擦，我們強烈建議您提供一些安裝腳本以及插件包，以自動化整個過程。更好的是，作為軟件包安裝的一部分運行該腳本。例如npm，您可以使用將其新增至`package.json`檔案中的`postinstall`指令碼。

處理豁免

如果您的組織有處理豁免的機制，則可以將其作為驗證程式外掛程式的一部分來實作。

說明可能免稅機制的範例案例：

- 組織的規則不允許使用公開 Amazon S3 儲存貯體，但在特定情況下除外。
- 開發人員正在建立屬於其中一種情況的 Amazon S3 儲存貯體，並請求豁免 (例如建立票證)。
- 安全工具知道如何從註冊豁免的內部系統中讀取

在這種情況下，開發人員將在內部系統中請求異常，然後將需要某種方式「註冊」該異常。除了 guard 外掛程式範例之外，您還可以建立一個外掛程式，藉由篩選出內部票務系統中具有相符豁免的違規行為來處理豁免。

有關示例實現，請參閱現有插件。

- @cdklabs/cdk-validator-cfnguard

使用 CDK Pipelines 持續整合與交付 (CI/CD)

使用「 AWS 建構程式庫」中的 CDK Pipelines 模組來設定 AWS CDK 應用程式的持續傳遞。當您將 CDK 應用程式的原始程式碼提交到 AWS CodeCommitGitHub、或時 AWS CodeStar，CDK Pipelines 可以自動建置、測試和部署新版本。

CDK Pipelines 是自我更新的。如果您新增應用程式階段或堆疊，管線會自動重新設定以部署這些新階段或堆疊。

> **Note**
>
> CDK Pipelines 支援兩個 API。其中一個是在 CDK Pipelines 開發人員預覽版中提供的原始 API。另一個是現代 API，其中包含了在預覽階段收到的 CDK 客戶的反饋。本主題中的範例使用現代 API。有關兩個支持的 API 之間差異的詳細信息，請參閱 aws-GitHub cdk 存儲庫中的 CDK Pipelines 原始 API。

主題

引導您的 AWS 環境

在您可以使用 CDK Pipelines 之前，您必須啟動要部署堆疊的 AWS 環境。

CDK 管線至少涉及兩個環境。第一個環境是佈建管線的位置。第二個環境是您要部署應用程式堆疊或階段的位置 (階段是相關堆疊的群組)。這些環境可以是相同的，但最佳實務建議是在不同環境中將階段彼此隔離。

> **Note**
>
> 如the section called “引導”需有關透過啟動載入建立的資源種類以及如何自訂啟動程序堆疊的詳細資訊，請參閱。

使用 CDK Pipelines 持續部署需要在 CDK 工具組堆疊中包含下列項目：

- Amazon Simple Storage Service (Amazon S3) 存儲桶。
- Amazon ECR 儲存庫。
- IAM 角色可為管道的各個部分提供所需的許可。

CDK Toolkit 將升級您現有的啟動程序堆棧，或者在必要時創建一個新的。

若要啟動可佈建 AWS CDK 管線的環境，請呼叫`cdk bootstrap`如下列範例所示。如有必要，透過`npx`指令叫用 AWS CDK Toolkit 會暫時安裝它。它還將使用當前項目中安裝的 Toolkit 版本（如果存在的話）。

`--cloudformation-execution-policies`指定原則的 ARN，以便在此原則下執行 future 的 CDK Pipelines 部署。預設`AdministratorAccess`政策可確保您的管道可以部署每種類型的 AWS 資源。如果您使用此原則，請確定您信任組成應用程式的所有 AWS CDK 程式碼和相依性。

大多數組織都要求更嚴格地控制哪些類型的資源可以通過自動化部署。請洽詢組織內的適當部門，以確定您的管道應使用的原則。

如果您的預設設 AWS 定檔包含必要的驗證配置和，則可以省略`--profile`此選項 AWS 區域。

macOS/Linux

```
npx cdk bootstrap aws://ACCOUNT-NUMBER/REGION --profile ADMIN-PROFILE \
    --cloudformation-execution-policies arn:aws:iam::aws:policy/AdministratorAccess
```

Windows

```
npx cdk bootstrap aws://ACCOUNT-NUMBER/REGION --profile ADMIN-PROFILE ^
```

```
    --cloudformation-execution-policies arn:aws:iam::aws:policy/AdministratorAccess
```

若要啟動管線將要部署 AWS CDK 應用程式的其他環境，請改用下列命令。該--trust選項指出哪個其他帳戶應具有將 AWS CDK 應用程序部署到此環境的權限。對於此選項，請指定管道的 AWS 帳戶 ID。

同樣地，如果您的預設設 AWS 定檔包含必要的驗證設定和，則可以省略--profile此選項 AWS 區域。

macOS/Linux

```
npx cdk bootstrap aws://ACCOUNT-NUMBER/REGION --profile ADMIN-PROFILE \
    --cloudformation-execution-policies arn:aws:iam::aws:policy/AdministratorAccess
 \
    --trust PIPELINE-ACCOUNT-NUMBER
```

Windows

```
npx cdk bootstrap aws://ACCOUNT-NUMBER/REGION --profile ADMIN-PROFILE ^
    --cloudformation-execution-policies arn:aws:iam::aws:policy/AdministratorAccess
 ^
    --trust PIPELINE-ACCOUNT-NUMBER
```

> **Tip**
>
> 僅使用管理認證來啟動和佈建初始管道。之後，請使用管道本身 (而非本機電腦) 部署變更。

如果您要升級舊版啟動載入環境，則建立新儲存貯體時，先前的 Amazon S3 儲存貯體會變成孤立儲存貯體。使用 Amazon S3 主控台手動將其刪除。

初始化專案

建立新的空白GitHub專案，並將其複製到`my-pipeline`目錄中的工作站。（我們在本主題中使用的代碼示例 GitHub。 您也可以使用 AWS CodeStar 或 AWS CodeCommit。）

```
git clone GITHUB-CLONE-URL my-pipeline
cd my-pipeline
```

> **Note**
>
> 您可以使用應用程序主目錄以外`my-pipeline`的名稱。但是，如果這樣做，則必須在本主題稍後調整文件和類名。這是因為工 AWS CDK 具包基於主目錄的名稱的一些文件和類名。

克隆後，像往常一樣初始化項目。

TypeScript

```
cdk init app --language typescript
```

JavaScript

```
cdk init app --language javascript
```

Python

```
cdk init app --language python
```

創建應用程序後，還要輸入以下兩個命令。這些激活應用程序的 Python 虛擬環境並安裝 AWS CDK 核心依賴項。

```
source .venv/bin/activate
python -m pip install -r requirements.txt
```

Java

```
cdk init app --language java
```

如果您使用的是 IDE，您現在可以開啟或匯入專案。例如，在 Eclipse 中，選擇「檔案 > 匯入 > Maven > 現有的 Maven 專案」。請確定專案設定已設定為使用 Java 8 (1.8)。

C#

```
cdk init app --language csharp
```

如果您使用的是 Visual Studio，請在`src`目錄中開啟解決方案檔案。

Go

```
cdk init app --language go
```

建立應用程式之後，也請輸入下列指令來安裝應用程式所需的「 AWS 建構程式庫」模組。

```
go get
```

> **Important**
>
> 確保將您的`cdk.json`和`cdk.context.json`文件提交到源代碼控制。上下文信息（例如功能標誌和從您的 AWS 帳戶中檢索的緩存值）是項目狀態的一部分。其他環境中的值可能會有所不同，這可能會導致結果發生非預期的變更。如需詳細資訊，請參閱 the section called "Context"。

定義配管

您的 CDK Pipelines 應用程式將包含至少兩個堆疊：一個代表管線本身，以及一或多個代表透過其部署之應用程式的堆疊。堆疊也可以分為幾個階段，您可以使用這些階段將基礎結構堆疊的複本部署到不同的環境。現在，我們將考慮管道，然後深入研究它將部署的應用程序。

該構造`CodePipeline`是代表 CDK 管道用 AWS CodePipeline 作其部署引擎的構造。當您在堆疊`CodePipeline`中執行個體化時，您可以定義管線的來源位置 (例如 GitHub 儲存庫)。您還可以定義命令來構建應用程序。

例如，以下內容定義了其來源儲存在儲存 GitHub 庫中的管線。它還包括 TypeScript CDK 應用程序的構建步驟。在指示的地方填寫有關您的 GitHub 回購的信息。

> **Note**
>
> 根據預設，管道會 GitHub 使用儲存在 Secrets Manager 中的名稱下的個人存取權杖進行驗證。`github-token`

您還需要更新管道堆棧的實例化以指定 AWS 帳戶和區域。

TypeScript

在lib/my-pipeline-stack.ts（如果您的項目文件夾未命名，則可能會有所不同my-pipeline）：

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import { CodePipeline, CodePipelineSource, ShellStep } from 'aws-cdk-lib/pipelines';

export class MyPipelineStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    const pipeline = new CodePipeline(this, 'Pipeline', {
      pipelineName: 'MyPipeline',
      synth: new ShellStep('Synth', {
        input: CodePipelineSource.gitHub('OWNER/REPO', 'main'),
        commands: ['npm ci', 'npm run build', 'npx cdk synth']
      })
    });
  }
}
```

在bin/my-pipeline.ts（如果您的項目文件夾未命名，則可能會有所不同my-pipeline）：

```
#!/usr/bin/env node
import * as cdk from 'aws-cdk-lib';
import { MyPipelineStack } from '../lib/my-pipeline-stack';

const app = new cdk.App();
new MyPipelineStack(app, 'MyPipelineStack', {
  env: {
    account: '111111111111',
    region: 'eu-west-1',
  }
});

app.synth();
```

JavaScript

在lib/my-pipeline-stack.js（如果您的項目文件夾未命名，則可能會有所不同my-pipeline）：

```
const cdk = require('aws-cdk-lib');
const { CodePipeline, CodePipelineSource, ShellStep } = require('aws-cdk-lib/
pipelines');

 class MyPipelineStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    const pipeline = new CodePipeline(this, 'Pipeline', {
      pipelineName: 'MyPipeline',
      synth: new ShellStep('Synth', {
        input: CodePipelineSource.gitHub('OWNER/REPO', 'main'),
        commands: ['npm ci', 'npm run build', 'npx cdk synth']
      })
    });
  }
}

module.exports = { MyPipelineStack }
```

在bin/my-pipeline.js（如果您的項目文件夾未命名，則可能會有所不同my-pipeline）：

```
#!/usr/bin/env node

const cdk = require('aws-cdk-lib');
const { MyPipelineStack } = require('../lib/my-pipeline-stack');

const app = new cdk.App();
new MyPipelineStack(app, 'MyPipelineStack', {
  env: {
    account: '111111111111',
    region: 'eu-west-1',
  }
});

app.synth();
```

Python

在my-pipeline/my-pipeline-stack.py（如果您的項目文件夾未命名，則可能會有所不同my-pipeline）：

```
import aws_cdk as cdk
from constructs import Construct
from aws_cdk.pipelines import CodePipeline, CodePipelineSource, ShellStep

class MyPipelineStack(cdk.Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        pipeline =  CodePipeline(self, "Pipeline",
                        pipeline_name="MyPipeline",
                        synth=ShellStep("Synth",
                            input=CodePipelineSource.git_hub("OWNER/REPO", "main"),
                            commands=["npm install -g aws-cdk",
                                "python -m pip install -r requirements.txt",
                                "cdk synth"]
                        )
                    )
```

在 app.py 中：

```
#!/usr/bin/env python3
import aws_cdk as cdk
from my_pipeline.my_pipeline_stack import MyPipelineStack

app = cdk.App()
MyPipelineStack(app, "MyPipelineStack",
    env=cdk.Environment(account="111111111111", region="eu-west-1")
)

app.synth()
```

Java

在src/main/java/com/myorg/MyPipelineStack.java（如果您的項目文件夾未命名，則可能會有所不同my-pipeline）：

```
package com.myorg;

import java.util.Arrays;
import software.constructs.Construct;
import software.amazon.awscdk.Stack;
```

```
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.pipelines.CodePipeline;
import software.amazon.awscdk.pipelines.CodePipelineSource;
import software.amazon.awscdk.pipelines.ShellStep;

public class MyPipelineStack extends Stack {
    public MyPipelineStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public MyPipelineStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        CodePipeline pipeline = CodePipeline.Builder.create(this, "pipeline")
             .pipelineName("MyPipeline")
             .synth(ShellStep.Builder.create("Synth")
                .input(CodePipelineSource.gitHub("OWNER/REPO", "main"))
                .commands(Arrays.asList("npm install -g aws-cdk", "cdk synth"))
                .build())
             .build();
    }
}
```

在src/main/java/com/myorg/MyPipelineApp.java（如果您的項目文件夾未命名，則可能會有所不同my-pipeline）：

```
package com.myorg;

import software.amazon.awscdk.App;
import software.amazon.awscdk.Environment;
import software.amazon.awscdk.StackProps;

public class MyPipelineApp {
    public static void main(final String[] args) {
        App app = new App();

        new MyPipelineStack(app, "PipelineStack", StackProps.builder()
            .env(Environment.builder()
                .account("111111111111")
                .region("eu-west-1")
                .build())
            .build());
```

```
        app.synth();
    }
}
```

C#

在src/MyPipeline/MyPipelineStack.cs（如果您的項目文件夾未命名，則可能會有所不同my-pipeline）：

```
using Amazon.CDK;
using Amazon.CDK.Pipelines;

namespace MyPipeline
{
    public class MyPipelineStack : Stack
    {
        internal MyPipelineStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
            var pipeline = new CodePipeline(this, "pipeline", new CodePipelineProps
            {
                PipelineName = "MyPipeline",
                Synth = new ShellStep("Synth", new ShellStepProps
                {
                    Input = CodePipelineSource.GitHub("OWNER/REPO", "main"),
                    Commands = new string[] { "npm install -g aws-cdk", "cdk
 synth" }
                })
            });
        }
    }
}
```

在src/MyPipeline/Program.cs（如果您的項目文件夾未命名，則可能會有所不同my-pipeline）：

```
using Amazon.CDK;

namespace MyPipeline
{
    sealed class Program
```

```
    {
        public static void Main(string[] args)
        {
            var app = new App();
            new MyPipelineStack(app, "MyPipelineStack", new StackProps
            {
                Env = new Amazon.CDK.Environment {
                    Account = "111111111111", Region = "eu-west-1" }
            });

            app.Synth();
        }
    }
}
```

您必須手動部署管線一次。之後，管道會從原始程式碼儲存庫保持最新狀態。因此，請確保 repo 中的代碼是您想要部署的代碼。檢查您的更改並推送到 GitHub，然後部署：

```
git add --all
git commit -m "initial commit"
git push
cdk deploy
```

> **Tip**
>
> 現在您已完成初始部署，您的本機 AWS 帳戶不再需要系統管理存取權。這是因為對應用程序的所有更改都將通過管道部署。所有你需要能夠做的就是推動 GitHub。

申請階段

若要定義可以一次新增至管線的多堆疊 AWS 應用程式，請定義的`Stage`子類別。（這與 CDK Pipelines 模塊`CdkStage`中的不同。）

階段包含構成應用程式的堆疊。如果堆棧之間存在依賴關係，堆棧將以正確的順序自動添加到管道中。不依賴彼此的堆疊會 parallel 部署。您可以通過調用堆棧之間添加依賴關係`stack1.addDependency(stack2)`。

階段接受一個默認`env`參數，這將成為其中堆棧的默認環境。（堆棧仍然可以指定自己的環境。）。

透過呼叫addStage()的執行個體，將應用程式新增至管線Stage。階段可以實例化並多次添加到管道中，以定義 DTAP 或多區域應用程序管道的不同階段。

我們將創建一個包含一個簡單的 Lambda 函數的堆棧，並將該堆棧放置在一個階段。然後，我們將階段添加到管道，以便它可以被部署。

TypeScript

建立新檔案lib/my-pipeline-lambda-stack.ts以存放包含 Lambda 函數的應用程式堆疊。

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import { Function, InlineCode, Runtime } from 'aws-cdk-lib/aws-lambda';

export class MyLambdaStack extends cdk.Stack {
    constructor(scope: Construct, id: string, props?: cdk.StackProps) {
      super(scope, id, props);

      new Function(this, 'LambdaFunction', {
        runtime: Runtime.NODEJS_18_X,
        handler: 'index.handler',
        code: new InlineCode('exports.handler = _ => "Hello, CDK";')
      });
    }
}
```

創建新文件lib/my-pipeline-app-stage.ts以保存我們的舞台。

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from "constructs";
import { MyLambdaStack } from './my-pipeline-lambda-stack';

export class MyPipelineAppStage extends cdk.Stage {

    constructor(scope: Construct, id: string, props?: cdk.StageProps) {
      super(scope, id, props);

      const lambdaStack = new MyLambdaStack(this, 'LambdaStack');
    }
}
```

編輯lib/my-pipeline-stack.ts以將階段添加到我們的管道中。

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import { CodePipeline, CodePipelineSource, ShellStep } from 'aws-cdk-lib/pipelines';
import { MyPipelineAppStage } from './my-pipeline-app-stage';

export class MyPipelineStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    const pipeline = new CodePipeline(this, 'Pipeline', {
      pipelineName: 'MyPipeline',
      synth: new ShellStep('Synth', {
        input: CodePipelineSource.gitHub('OWNER/REPO', 'main'),
        commands: ['npm ci', 'npm run build', 'npx cdk synth']
      })
    });

    pipeline.addStage(new MyPipelineAppStage(this, "test", {
      env: { account: "111111111111", region: "eu-west-1" }
    }));
  }
}
```

JavaScript

建立新檔案lib/my-pipeline-lambda-stack.js以存放包含 Lambda 函數的應用程式堆疊。

```
const cdk = require('aws-cdk-lib');
const { Function, InlineCode, Runtime } = require('aws-cdk-lib/aws-lambda');

class MyLambdaStack extends cdk.Stack {
    constructor(scope, id, props) {
      super(scope, id, props);

      new Function(this, 'LambdaFunction', {
        runtime: Runtime.NODEJS_18_X,
        handler: 'index.handler',
        code: new InlineCode('exports.handler = _ => "Hello, CDK";')
      });
    }
}

module.exports = { MyLambdaStack }
```

創建新文件lib/my-pipeline-app-stage.js以保存我們的舞台。

```
const cdk = require('aws-cdk-lib');
const { MyLambdaStack } = require('./my-pipeline-lambda-stack');

class MyPipelineAppStage extends cdk.Stage {

    constructor(scope, id, props) {
      super(scope, id, props);

      const lambdaStack = new MyLambdaStack(this, 'LambdaStack');
    }
}

module.exports = { MyPipelineAppStage };
```

編輯lib/my-pipeline-stack.ts以將階段添加到我們的管道中。

```
const cdk = require('aws-cdk-lib');
const { CodePipeline, CodePipelineSource, ShellStep } = require('aws-cdk-lib/
pipelines');
const { MyPipelineAppStage } = require('./my-pipeline-app-stage');

 class MyPipelineStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    const pipeline = new CodePipeline(this, 'Pipeline', {
      pipelineName: 'MyPipeline',
      synth: new ShellStep('Synth', {
        input: CodePipelineSource.gitHub('OWNER/REPO', 'main'),
        commands: ['npm ci', 'npm run build', 'npx cdk synth']
      })
    });

    pipeline.addStage(new MyPipelineAppStage(this, "test", {
      env: { account: "111111111111", region: "eu-west-1" }
    }));

  }
}

module.exports = { MyPipelineStack }
```

Python

建立新檔案my_pipeline/my_pipeline_lambda_stack.py以存放包含 Lambda 函數的應用程式堆疊。

```
import aws_cdk as cdk
from constructs import Construct
from aws_cdk.aws_lambda import Function, InlineCode, Runtime

class MyLambdaStack(cdk.Stack):
    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        Function(self, "LambdaFunction",
            runtime=Runtime.NODEJS_18_X,
            handler="index.handler",
            code=InlineCode("exports.handler = _ => 'Hello, CDK';")
        )
```

創建新文件my_pipeline/my_pipeline_app_stage.py以保存我們的舞台。

```
import aws_cdk as cdk
from constructs import Construct
from my_pipeline.my_pipeline_lambda_stack import MyLambdaStack

class MyPipelineAppStage(cdk.Stage):
    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        lambdaStack = MyLambdaStack(self, "LambdaStack")
```

編輯my_pipeline/my-pipeline-stack.py以將階段添加到我們的管道中。

```
import aws_cdk as cdk
from constructs import Construct
from aws_cdk.pipelines import CodePipeline, CodePipelineSource, ShellStep
from my_pipeline.my_pipeline_app_stage import MyPipelineAppStage

class MyPipelineStack(cdk.Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)
```

```
        pipeline =  CodePipeline(self, "Pipeline",
                        pipeline_name="MyPipeline",
                        synth=ShellStep("Synth",
                            input=CodePipelineSource.git_hub("OWNER/REPO", "main"),
                            commands=["npm install -g aws-cdk",
                                "python -m pip install -r requirements.txt",
                                "cdk synth"]))

        pipeline.add_stage(MyPipelineAppStage(self, "test",
            env=cdk.Environment(account="111111111111", region="eu-west-1")))
```

Java

建立新檔案src/main/java/com.myorg/MyPipelineLambdaStack.java以存放包含 Lambda 函數的應用程式堆疊。

```
package com.myorg;

import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;

import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.lambda.Runtime;
import software.amazon.awscdk.services.lambda.InlineCode;

public class MyPipelineLambdaStack extends Stack {
    public MyPipelineLambdaStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public MyPipelineLambdaStack(final Construct scope, final String id, final
 StackProps props) {
        super(scope, id, props);

        Function.Builder.create(this, "LambdaFunction")
          .runtime(Runtime.NODEJS_18_X)
          .handler("index.handler")
          .code(new InlineCode("exports.handler = _ => 'Hello, CDK';"))
          .build();

    }
```

```
}
```

創建新文件src/main/java/com.myorg/MyPipelineAppStage.java以保存我們的舞台。

```
package com.myorg;

import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.Stage;
import software.amazon.awscdk.StageProps;

public class MyPipelineAppStage extends Stage {
    public MyPipelineAppStage(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public MyPipelineAppStage(final Construct scope, final String id, final
 StageProps props) {
        super(scope, id, props);

        Stack lambdaStack = new MyPipelineLambdaStack(this, "LambdaStack");
    }

}
```

編輯src/main/java/com.myorg/MyPipelineStack.java以將階段添加到我們的管道中。

```
package com.myorg;

import java.util.Arrays;
import software.constructs.Construct;
import software.amazon.awscdk.Environment;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.StageProps;
import software.amazon.awscdk.pipelines.CodePipeline;
import software.amazon.awscdk.pipelines.CodePipelineSource;
import software.amazon.awscdk.pipelines.ShellStep;

public class MyPipelineStack extends Stack {
    public MyPipelineStack(final Construct scope, final String id) {
        this(scope, id, null);
```

```
    }

    public MyPipelineStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        final CodePipeline pipeline = CodePipeline.Builder.create(this, "pipeline")
            .pipelineName("MyPipeline")
            .synth(ShellStep.Builder.create("Synth")
                .input(CodePipelineSource.gitHub("OWNER/REPO", "main"))
                .commands(Arrays.asList("npm install -g aws-cdk", "cdk synth"))
                .build())
            .build();

        pipeline.addStage(new MyPipelineAppStage(this, "test", StageProps.builder()
            .env(Environment.builder()
                .account("111111111111")
                .region("eu-west-1")
                .build())
            .build()));
    }
}
```

C#

建立新檔案src/MyPipeline/MyPipelineLambdaStack.cs以存放包含 Lambda 函數的應用程式堆疊。

```
using Amazon.CDK;
using Constructs;
using Amazon.CDK.AWS.Lambda;

namespace MyPipeline
{
    class MyPipelineLambdaStack : Stack
    {
        public MyPipelineLambdaStack(Construct scope, string id, StackProps
 props=null) : base(scope, id, props)
        {
            new Function(this, "LambdaFunction", new FunctionProps
            {
                Runtime = Runtime.NODEJS_18_X,
                Handler = "index.handler",
```

```
                Code = new InlineCode("exports.handler = _ => 'Hello, CDK';")
            });
        }
    }
}
```

創建新文件src/MyPipeline/MyPipelineAppStage.cs以保存我們的舞台。

```
using Amazon.CDK;
using Constructs;

namespace MyPipeline
{
    class MyPipelineAppStage : Stage
    {
        public MyPipelineAppStage(Construct scope, string id, StageProps
 props=null) : base(scope, id, props)
        {
            Stack lambdaStack = new MyPipelineLambdaStack(this, "LambdaStack");
        }
    }
}
```

編輯src/MyPipeline/MyPipelineStack.cs以將階段添加到我們的管道中。

```
using Amazon.CDK;
using Constructs;
using Amazon.CDK.Pipelines;

namespace MyPipeline
{
    public class MyPipelineStack : Stack
    {
        internal MyPipelineStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
            var pipeline = new CodePipeline(this, "pipeline", new CodePipelineProps
            {
                PipelineName = "MyPipeline",
                Synth = new ShellStep("Synth", new ShellStepProps
                {
                    Input = CodePipelineSource.GitHub("OWNER/REPO", "main"),
```

```
                    Commands = new string[] { "npm install -g aws-cdk", "cdk
 synth" }
                })
            });

            pipeline.AddStage(new MyPipelineAppStage(this, "test", new StageProps
            {
                Env = new Environment
                {
                    Account = "111111111111", Region = "eu-west-1"
                }
            }));
        }
    }
}
```

每個由新增的應用程式階段都會addStage()結果加入對應的管線階段 (以addStage()呼叫傳回的StageDeployment執行個體表示)。您可以呼叫階段或方法，將部署前addPre()或addPost()部署後動作新增至階段。

TypeScript

```
// import { ManualApprovalStep } from 'aws-cdk-lib/pipelines';

const testingStage = pipeline.addStage(new MyPipelineAppStage(this, 'testing', {
  env: { account: '111111111111', region: 'eu-west-1' }
}));

    testingStage.addPost(new ManualApprovalStep('approval'));
```

JavaScript

```
// const { ManualApprovalStep } = require('aws-cdk-lib/pipelines');

const testingStage = pipeline.addStage(new MyPipelineAppStage(this, 'testing', {
  env: { account: '111111111111', region: 'eu-west-1' }
}));

testingStage.addPost(new ManualApprovalStep('approval'));
```

Python

```
# from aws_cdk.pipelines import ManualApprovalStep

testing_stage = pipeline.add_stage(MyPipelineAppStage(self, "testing",
    env=cdk.Environment(account="111111111111", region="eu-west-1")))

testing_stage.add_post(ManualApprovalStep('approval'))
```

Java

```
// import software.amazon.awscdk.pipelines.StageDeployment;
// import software.amazon.awscdk.pipelines.ManualApprovalStep;

StageDeployment testingStage =
        pipeline.addStage(new MyPipelineAppStage(this, "test", StageProps.builder()
                .env(Environment.builder()
                        .account("111111111111")
                        .region("eu-west-1")
                        .build())
                .build()));

testingStage.addPost(new ManualApprovalStep("approval"));
```

C#

```
var testingStage = pipeline.AddStage(new MyPipelineAppStage(this, "test", new
 StageProps
{
    Env = new Environment
    {
        Account = "111111111111", Region = "eu-west-1"
    }
}));

testingStage.AddPost(new ManualApprovalStep("approval"));
```

您可以將階段新增至 Wave 以 parallel 部署階段，例如將階段部署到多個帳戶或區域時。

TypeScript

```
const wave = pipeline.addWave('wave');
wave.addStage(new MyApplicationStage(this, 'MyAppEU', {
  env: { account: '111111111111', region: 'eu-west-1' }
}));
wave.addStage(new MyApplicationStage(this, 'MyAppUS', {
  env: { account: '111111111111', region: 'us-west-1' }
}));
```

JavaScript

```
const wave = pipeline.addWave('wave');
wave.addStage(new MyApplicationStage(this, 'MyAppEU', {
  env: { account: '111111111111', region: 'eu-west-1' }
}));
wave.addStage(new MyApplicationStage(this, 'MyAppUS', {
  env: { account: '111111111111', region: 'us-west-1' }
}));
```

Python

```
wave = pipeline.add_wave("wave")
wave.add_stage(MyApplicationStage(self, "MyAppEU",
    env=cdk.Environment(account="111111111111", region="eu-west-1")))
wave.add_stage(MyApplicationStage(self, "MyAppUS",
    env=cdk.Environment(account="111111111111", region="us-west-1")))
```

Java

```
// import software.amazon.awscdk.pipelines.Wave;
final Wave wave = pipeline.addWave("wave");
wave.addStage(new MyPipelineAppStage(this, "MyAppEU", StageProps.builder()
        .env(Environment.builder()
                .account("111111111111")
                .region("eu-west-1")
                .build())
        .build()));
wave.addStage(new MyPipelineAppStage(this, "MyAppUS", StageProps.builder()
        .env(Environment.builder()
                .account("111111111111")
                .region("us-west-1")
```

```
                .build())
        .build()));
```

C#

```
var wave = pipeline.AddWave("wave");
wave.AddStage(new MyPipelineAppStage(this, "MyAppEU", new StageProps
{
    Env = new Environment
    {
        Account = "111111111111", Region = "eu-west-1"
    }
}));
wave.AddStage(new MyPipelineAppStage(this, "MyAppUS", new StageProps
{
    Env = new Environment
    {
        Account = "111111111111", Region = "us-west-1"
    }
}));
```

測試部署

您可以將步驟新增至 CDK 管道，以驗證您正在執行的部署。例如，您可以使用 CDK 管線程式庫ShellStep來執行下列工作：

- 嘗試存取由 Lambda 函數支援的新部署 Amazon API Gateway
- 透過發出 AWS CLI 指令來檢查已部署資源的設定

以最簡單的形式，添加驗證操作如下所示：

TypeScript

```
// stage was returned by pipeline.addStage

stage.addPost(new ShellStep("validate", {
  commands: ['../tests/validate.sh'],
}));
```

JavaScript

```
// stage was returned by pipeline.addStage

stage.addPost(new ShellStep("validate", {
  commands: ['../tests/validate.sh'],
}));
```

Python

```
# stage was returned by pipeline.add_stage

stage.add_post(ShellStep("validate",
  commands=[''../tests/validate.sh'']
))
```

Java

```
// stage was returned by pipeline.addStage

stage.addPost(ShellStep.Builder.create("validate")
        .commands(Arrays.asList("'../tests/validate.sh'"))
        .build());
```

C#

```
// stage was returned by pipeline.addStage

stage.AddPost(new ShellStep("validate", new ShellStepProps
{
    Commands = new string[] { "'../tests/validate.sh'" }
}));
```

許多 AWS CloudFormation 部署都會產生具有不可預知名稱的資源。因此，CDK Pipelines 提供了一種在部署後讀取 AWS CloudFormation 輸出的方法。這使得可以將（例如）負載平衡器生成的 URL 傳遞給測試操作。

要使用輸出，請公開您感興趣的`CfnOutput`對象。然後，將其傳遞到步驟的`envFromCfnOutputs`屬性中，以使其可作為該步驟中的環境變數使用。

TypeScript

```
// given a stack lbStack that exposes a load balancer construct as loadBalancer
this.loadBalancerAddress = new cdk.CfnOutput(lbStack, 'LbAddress', {
  value: `https://${lbStack.loadBalancer.loadBalancerDnsName}/`
});

// pass the load balancer address to a shell step
stage.addPost(new ShellStep("lbaddr", {
  envFromCfnOutputs: {lb_addr: lbStack.loadBalancerAddress},
  commands: ['echo $lb_addr']
}));
```

JavaScript

```
// given a stack lbStack that exposes a load balancer construct as loadBalancer
this.loadBalancerAddress = new cdk.CfnOutput(lbStack, 'LbAddress', {
  value: `https://${lbStack.loadBalancer.loadBalancerDnsName}/`
});

// pass the load balancer address to a shell step
stage.addPost(new ShellStep("lbaddr", {
  envFromCfnOutputs: {lb_addr: lbStack.loadBalancerAddress},
  commands: ['echo $lb_addr']
}));
```

Python

```
# given a stack lb_stack that exposes a load balancer construct as load_balancer
self.load_balancer_address = cdk.CfnOutput(lb_stack, "LbAddress",
    value=f"https://{lb_stack.load_balancer.load_balancer_dns_name}/")

# pass the load balancer address to a shell step
stage.add_post(ShellStep("lbaddr",
    env_from_cfn_outputs={"lb_addr": lb_stack.load_balancer_address}
    commands=["echo $lb_addr"]))
```

Java

```
// given a stack lbStack that exposes a load balancer construct as loadBalancer
loadBalancerAddress = CfnOutput.Builder.create(lbStack, "LbAddress")
                            .value(String.format("https://%s/",
```

```
                                        lbStack.loadBalancer.loadBalancerDnsName))
                                .build();

stage.addPost(ShellStep.Builder.create("lbaddr")
    .envFromCfnOutputs(     // Map.of requires Java 9 or later
        java.util.Map.of("lbAddr", loadBalancerAddress))
    .commands(Arrays.asList("echo $lbAddr"))
    .build());
```

C#

```
// given a stack lbStack that exposes a load balancer construct as loadBalancer
loadBalancerAddress = new CfnOutput(lbStack, "LbAddress", new CfnOutputProps
{
    Value = string.Format("https://{0}/", lbStack.loadBalancer.LoadBalancerDnsName)
});

stage.AddPost(new ShellStep("lbaddr", new ShellStepProps
{
    EnvFromCfnOutputs = new Dictionary<string, CfnOutput>
    {
        {  "lbAddr", loadBalancerAddress }
    },
    Commands = new string[] { "echo $lbAddr" }
}));
```

您可以直接在中編寫簡單的驗證測試`ShellStep`，但是當測試超過幾行時，這種方法會變得笨拙。對於更複雜的測試，您可以`ShellStep`通過屬性將其他文件（例如完整的 shell 腳本或其他語言的程序）帶`inputs`入。輸入可以是具有輸出的任何步驟，包括源代碼（例如 GitHub repo）或其他`ShellStep`。

如果檔案可直接在測試中使用 (例如，如果它們本身是可執行的)，則從來源儲存庫引入檔案是適當的。在此示例中，我們將我們的 GitHub repo 聲明為`source`（而不是將其實例化為內聯`CodePipeline`）。然後，我們將此文件集傳遞給管道和驗證測試。

TypeScript

```
const source = CodePipelineSource.gitHub('OWNER/REPO', 'main');

const pipeline = new CodePipeline(this, 'Pipeline', {
  pipelineName: 'MyPipeline',
```

```
  synth: new ShellStep('Synth', {
    input: source,
    commands: ['npm ci', 'npm run build', 'npx cdk synth']
  })
});

const stage = pipeline.addStage(new MyPipelineAppStage(this, 'test', {
  env: { account: '111111111111', region: 'eu-west-1' }
}));

stage.addPost(new ShellStep('validate', {
  input: source,
  commands: ['sh ../tests/validate.sh']
}));
```

JavaScript

```
const source = CodePipelineSource.gitHub('OWNER/REPO', 'main');

const pipeline = new CodePipeline(this, 'Pipeline', {
  pipelineName: 'MyPipeline',
  synth: new ShellStep('Synth', {
    input: source,
    commands: ['npm ci', 'npm run build', 'npx cdk synth']
  })
});

const stage = pipeline.addStage(new MyPipelineAppStage(this, 'test', {
  env: { account: '111111111111', region: 'eu-west-1' }
}));

stage.addPost(new ShellStep('validate', {
  input: source,
  commands: ['sh ../tests/validate.sh']
}));
```

Python

```
source   = CodePipelineSource.git_hub("OWNER/REPO", "main")

pipeline =  CodePipeline(self, "Pipeline",
                pipeline_name="MyPipeline",
                synth=ShellStep("Synth",
```

```
                    input=source,
                    commands=["npm install -g aws-cdk",
                        "python -m pip install -r requirements.txt",
                        "cdk synth"]))

stage = pipeline.add_stage(MyApplicationStage(self, "test",
            env=cdk.Environment(account="111111111111", region="eu-west-1")))

stage.add_post(ShellStep("validate", input=source,
    commands=["sh ../tests/validate.sh"],
))
```

Java

```
final CodePipelineSource source = CodePipelineSource.gitHub("OWNER/REPO", "main");

final CodePipeline pipeline = CodePipeline.Builder.create(this, "pipeline")
        .pipelineName("MyPipeline")
        .synth(ShellStep.Builder.create("Synth")
                .input(source)
                .commands(Arrays.asList("npm install -g aws-cdk", "cdk synth"))
                .build())
        .build();

final StageDeployment stage =
        pipeline.addStage(new MyPipelineAppStage(this, "test", StageProps.builder()
                .env(Environment.builder()
                        .account("111111111111")
                        .region("eu-west-1")
                        .build())
                .build()));

stage.addPost(ShellStep.Builder.create("validate")
        .input(source)
        .commands(Arrays.asList("sh ../tests/validate.sh"))
        .build());
```

C#

```
var source = CodePipelineSource.GitHub("OWNER/REPO", "main");

var pipeline = new CodePipeline(this, "pipeline", new CodePipelineProps
{
```

```
    PipelineName = "MyPipeline",
    Synth = new ShellStep("Synth", new ShellStepProps
    {
        Input = source,
        Commands = new string[] { "npm install -g aws-cdk", "cdk synth" }
    })
});

var stage = pipeline.AddStage(new MyPipelineAppStage(this, "test", new StageProps
{
    Env = new Environment
    {
        Account = "111111111111", Region = "eu-west-1"
    }
}));

stage.AddPost(new ShellStep("validate", new ShellStepProps
{
    Input = source,
    Commands = new string[] { "sh ../tests/validate.sh" }
}));
```

如果您的測試需要編譯（作為合成的一部分來完成），則從合成步驟中獲取其他文件是合適的。

TypeScript

```
const synthStep = new ShellStep('Synth', {
  input: CodePipelineSource.gitHub('OWNER/REPO', 'main'),
  commands: ['npm ci', 'npm run build', 'npx cdk synth'],
});

const pipeline = new CodePipeline(this, 'Pipeline', {
  pipelineName: 'MyPipeline',
  synth: synthStep
});

const stage = pipeline.addStage(new MyPipelineAppStage(this, 'test', {
  env: { account: '111111111111', region: 'eu-west-1' }
}));

// run a script that was transpiled from TypeScript during synthesis
stage.addPost(new ShellStep('validate', {
```

```
  input: synthStep,
  commands: ['node tests/validate.js']
}));
```

JavaScript

```
const synthStep = new ShellStep('Synth', {
  input: CodePipelineSource.gitHub('OWNER/REPO', 'main'),
  commands: ['npm ci', 'npm run build', 'npx cdk synth'],
});

const pipeline = new CodePipeline(this, 'Pipeline', {
  pipelineName: 'MyPipeline',
  synth: synthStep
});

const stage = pipeline.addStage(new MyPipelineAppStage(this, "test", {
  env: { account: "111111111111", region: "eu-west-1" }
}));

// run a script that was transpiled from TypeScript during synthesis
stage.addPost(new ShellStep('validate', {
  input: synthStep,
  commands: ['node tests/validate.js']
}));
```

Python

```
synth_step = ShellStep("Synth",
                input=CodePipelineSource.git_hub("OWNER/REPO", "main"),
                commands=["npm install -g aws-cdk",
                  "python -m pip install -r requirements.txt",
                  "cdk synth"])

pipeline   = CodePipeline(self, "Pipeline",
                pipeline_name="MyPipeline",
                synth=synth_step)

stage = pipeline.add_stage(MyApplicationStage(self, "test",
            env=cdk.Environment(account="111111111111", region="eu-west-1")))

# run a script that was compiled during synthesis
stage.add_post(ShellStep("validate",
```

```
    input=synth_step,
    commands=["node test/validate.js"],
))
```

Java

```
final ShellStep synth = ShellStep.Builder.create("Synth")
                            .input(CodePipelineSource.gitHub("OWNER/REPO", "main"))
                            .commands(Arrays.asList("npm install -g aws-cdk", "cdk
 synth"))
                            .build();

final CodePipeline pipeline = CodePipeline.Builder.create(this, "pipeline")
        .pipelineName("MyPipeline")
        .synth(synth)
        .build();

final StageDeployment stage =
        pipeline.addStage(new MyPipelineAppStage(this, "test", StageProps.builder()
                .env(Environment.builder()
                        .account("111111111111")
                        .region("eu-west-1")
                        .build())
                .build()));

stage.addPost(ShellStep.Builder.create("validate")
        .input(synth)
        .commands(Arrays.asList("node ./tests/validate.js"))
        .build());
```

C#

```
var synth = new ShellStep("Synth", new ShellStepProps
{
    Input = CodePipelineSource.GitHub("OWNER/REPO", "main"),
    Commands = new string[] { "npm install -g aws-cdk", "cdk synth" }
});

var pipeline = new CodePipeline(this, "pipeline", new CodePipelineProps
{
    PipelineName = "MyPipeline",
    Synth = synth
});
```

```
var stage = pipeline.AddStage(new MyPipelineAppStage(this, "test", new StageProps
{
    Env = new Environment
    {
        Account = "111111111111", Region = "eu-west-1"
    }
}));

stage.AddPost(new ShellStep("validate", new ShellStepProps
{
    Input = synth,
    Commands = new string[] { "node ./tests/validate.js" }
}));
```

安全注意事項

任何形式的持續交付都有固有的安全風險。在 AWS 共同責任模式下，您必須對 AWS 雲端中資訊的安全性負責。CDK Pipelines 程式庫透過整合安全的預設值和建模最佳實務，為您提供良好的開端。

但是，就其本質而言，需要高級別訪問權限才能實現其預期目的的庫無法確保完全的安全性。您的組織 AWS 和外部有許多攻擊媒介。

特別是，請記住以下幾點：

- 請注意您依賴的軟件。審核您在管道中執行的所有第三方軟體，因為它可能會變更部署的基礎結構。
- 使用相依性鎖定來防止意外升級。CDK Pipelines 尊重`package-lock.json`並`yarn.lock`確保您的依賴關係是您期望的。
- CDK Pipelines 會在您自己帳戶中建立的資源上執行，而這些資源的組態由開發人員透過管道提交程式碼來控制。因此，CDK Pipelines ine 本身無法防範試圖繞過合規性檢查的惡意開發人員。如果您的威脅模型包括編寫 CDK 代碼的開發人員，則應該具有外部合規性機制，例如 AWS CloudFormation Hook（預防性）或 AWS Config（被動），AWS CloudFormation 執行角色沒有禁用權限。
- 生產環境的認證應該是短暫的。在啟動安裝和初始佈建之後，開發人員完全不需要擁有帳戶憑據。變更可透過管線部署。首先不需要憑據，減少憑據洩漏的可能性。

故障診斷

開始使用 CDK Pipelines 時，通常會遇到下列問題。

管道：內部故障

```
CREATE_FAILED  | AWS::CodePipeline::Pipeline | Pipeline/Pipeline
Internal Failure
```

檢查您的 GitHub 訪問令牌。它可能遺失，或者可能沒有存取存放庫的權限。

索引鍵：原則包含具有一或多個無效主體的陳述式

```
CREATE_FAILED | AWS::KMS::Key | Pipeline/Pipeline/ArtifactsBucketEncryptionKey
Policy contains a statement with one or more invalid principals.
```

其中一個目標環境尚未使用新的啟動程序堆疊啟動載入。確保所有目標環境都已啟動載入。

堆棧處於回滾 _ 完成狀態，無法更新。

```
Stack STACK_NAME is in ROLLBACK_COMPLETE state and can not be updated. (Service:
AmazonCloudFormation; Status Code: 400; Error Code: ValidationError; Request
ID: ...)
```

堆疊先前的部署失敗，且處於不可重試的狀態。從 AWS CloudFormation 主控台刪除堆疊，然後重試部署。

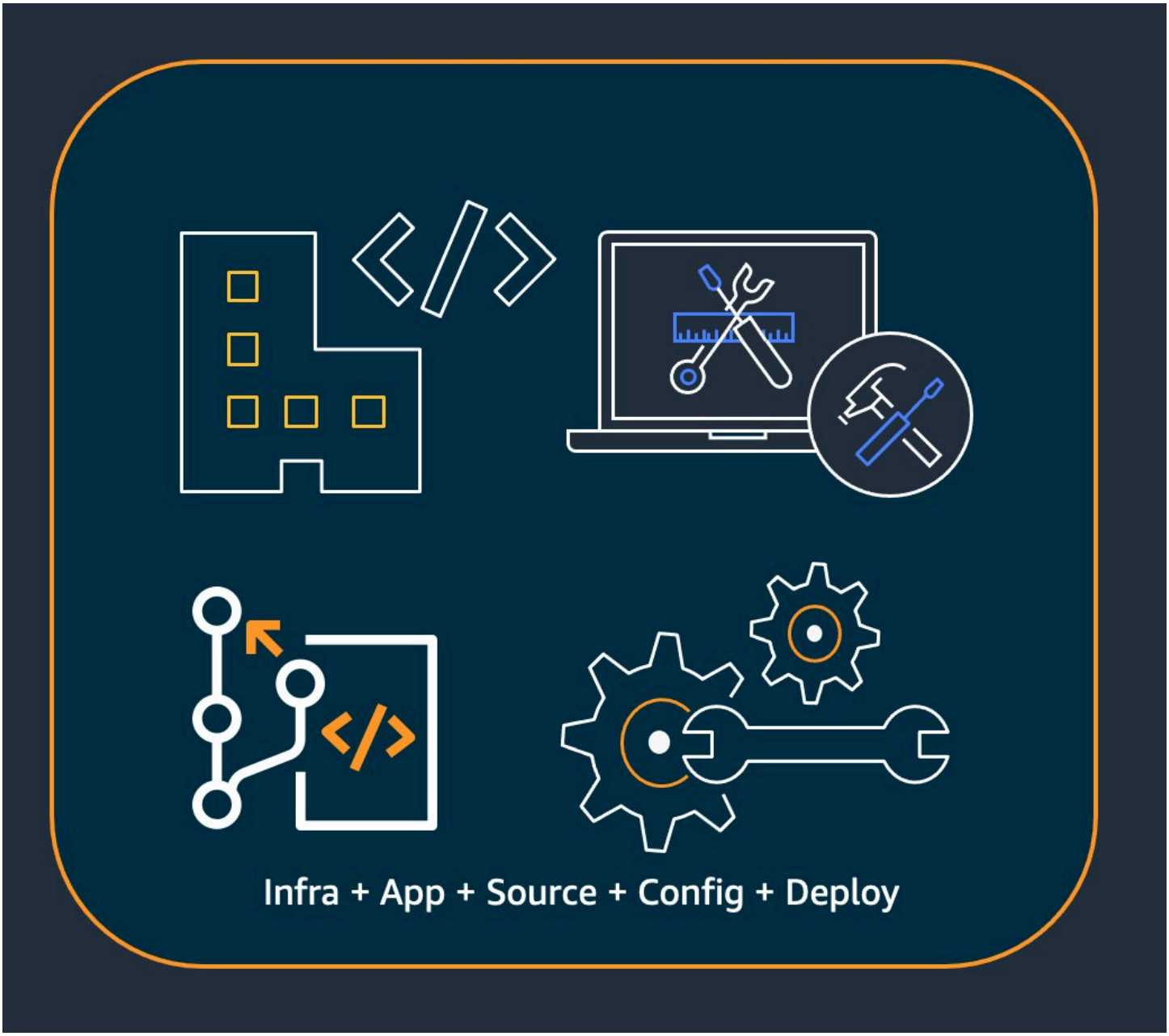
開發和部署雲端基礎架構的最佳做法 AWS CDK

開發人員或管理員可以使用支援的程式設計語言來定義他們的雲端基礎結構。 AWS CDK CDK 應用程式應組織成邏輯單元，例如 API、資料庫和監視資源，並選擇性地擁有自動化部署的管道。邏輯單元應作為構造實現，包括以下內容：

- 基礎設施 (例如 Amazon S3 儲存貯體、Amazon RDS 資料庫或 Amazon VPC 網路)
- 執行階段程式碼 (例如 AWS Lambda 函數)
- 配置代碼

堆疊定義這些邏輯單元的部署模型。有關 CDK 背後概念的更詳細介紹，請參閱開始使用。

這 AWS CDK 反映了客戶和內部團隊的需求，以及在部署和持續維護複雜雲端應用程式期間經常出現的故障模式的審慎考量。我們發現失敗通常與未完全測試之應用程式的 out-of-band "" 變更有關，例如組態變更。因此，我們開發了 AWS CDK 圍繞一個模型，其中您的整個應用程序在代碼中定義，不僅業務邏輯，而且基礎設施和配置。如此一來，可以仔細檢閱提議的變更，在類似於生產程度的環境中進行全面測試，並在出現問題時完全回復。



在部署階段， AWS CDK 會合成包含下列項目的雲端組件：

- AWS CloudFormation 在所有目標環境中描述基礎結構的範本
- 包含執行階段程式碼及其支援檔案的檔案資產

使用 CDK，應用程序主要版本控制分支中的每個提交都可以代表應用程序的完整，一致，可部署的版本。然後，只要進行變更，就可以自動部署您的應用程式。

背後的理念 AWS CDK 導致了我們建議的最佳實踐，我們分為四大類。

- the section called “組織最佳實務”
- the section called “編碼最佳實踐”
- the section called “建構最佳做法”
- the section called “應用程式最佳做”

> **Tip**
>
> 如果適用於 CDK 定義的 AWS CloudFormation基礎結構，也請考慮適用於您所使用的個別 AWS 服務的最佳做法。

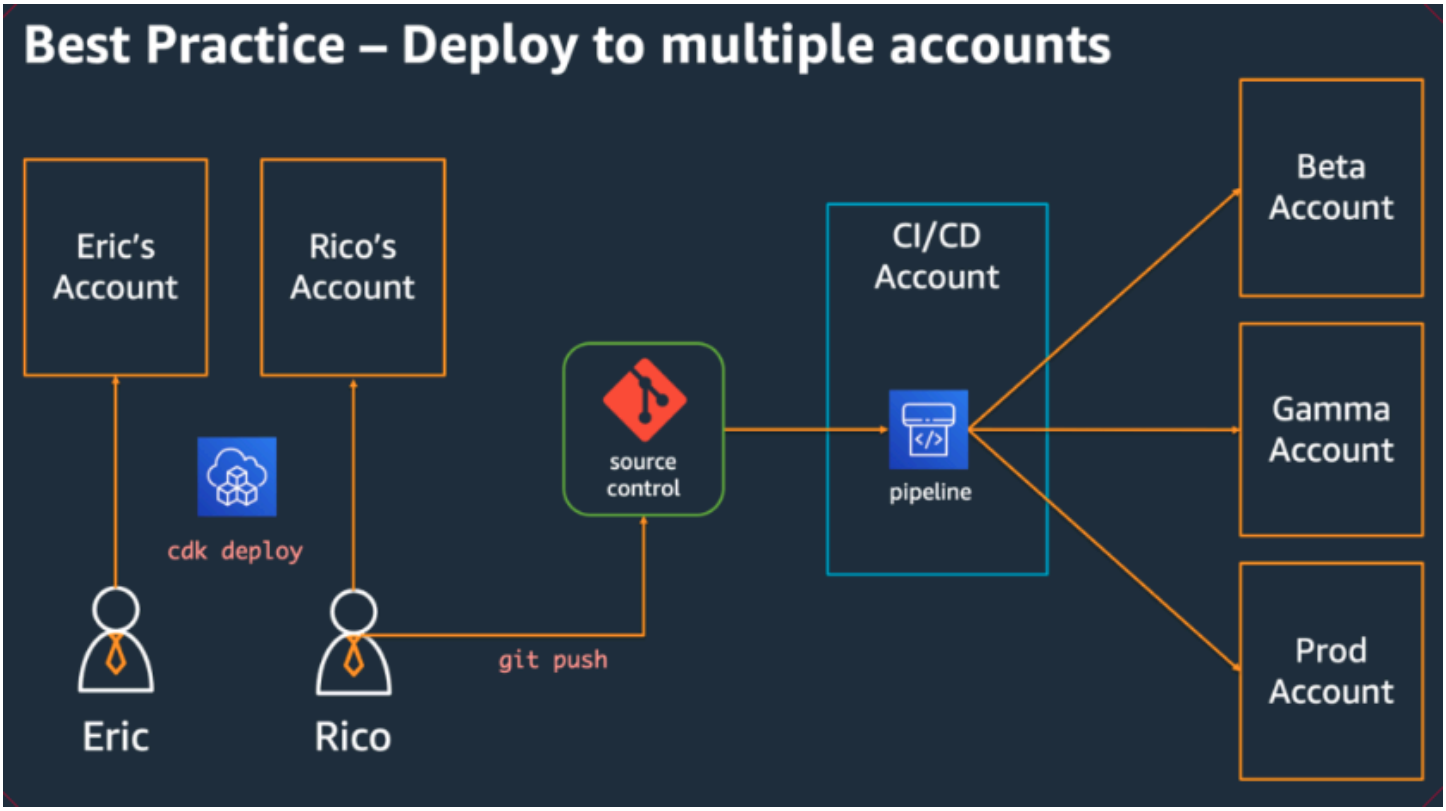
組織最佳實務

在 AWS CDK 採用的開始階段，請務必考慮如何設定組織以取得成功。擁有一支專家團隊負責培訓和指導公司其他成員採用 CDK 的最佳做法。這個團隊的規模可能會有所不同，從一個小公司的一個或兩個人到一個成熟 Cloud Center of Excellence (CCoE) 在一個大公司。該團隊負責為貴公司的雲端基礎架構設定標準和政策，以及培訓和指導開發人員。

CCoE 可能會針對雲端基礎架構應使用哪些程式設計語言提供指引。每個組織的詳細資料會有所不同，但良好的政策有助於確保開發人員能夠理解和維護公司的雲端基礎架構。

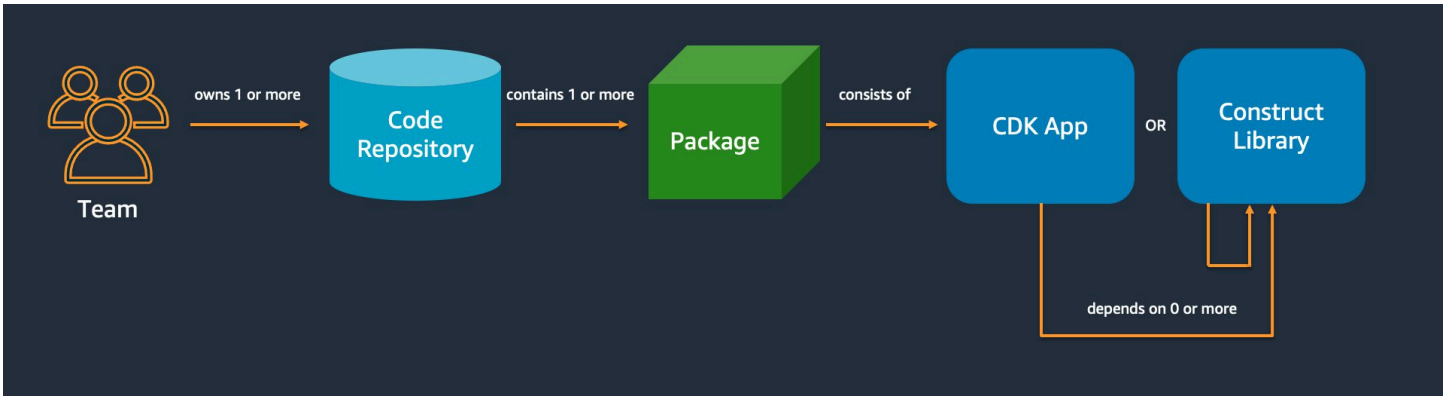
CCoE 也會建立定義其中組織單位的「landing zone」。 AWS landing zone 是根據最佳實務藍圖的預先設定、安全、可擴充的多帳戶 AWS 環境。若要將組成 landing zone 的服務結合在一起，您可以使用它 AWS Control Tower，從單一使用者介面設定和管理整個多帳戶系統。

開發團隊應該能夠根據需要使用自己的帳戶在這些帳戶中測試和部署新資源。個別開發人員可以將這些資源視為自己開發工作站的擴充功能。使用 CDK Pipelines，可以透過 CI/CD 帳戶將 AWS CDK 應用程式部署到測試、整合和生產環境 (每個環境都隔離在自己的 AWS 區域或帳戶中)。這是通過將開發人員的代碼合併到您組織的規範存儲庫中來完成的。



編碼最佳實踐

本節介紹組織 AWS CDK 程式碼的最佳作法。下圖顯示團隊與該團隊的程式碼儲存庫、套件、應用程式和建構程式庫之間的關係。



從簡單開始，只有在需要時才增加複雜性

我們大多數最佳實踐的指導原則是盡可能簡單地保持事情-但不簡單。只有在您的需求決定更複雜的解決方案時，才會增加複雜性。使用 AWS CDK，您可以視需要重構程式碼，以支援新的需求。您不必預先為所有可能的場景進行架構。

與 Well-Architected 的框 AWS 架保持一致

AWS Well-Architected 的框架將一個組件定義為代碼，配置和 AWS 資源，共同提供針對需求。組件通常是技術所有權的單位，並且與其他組件分離。工作負載一詞用於識別一組共同提供商業價值的元件。工作負載通常是企業和技術領導者溝通的詳細程度。

AWS CDK 應用程序映射到由 AWS Well-Architected 的框架定義的組件。 AWS CDK 應用程式是編碼並提供 Well-Architected 雲端應用程式最佳實務的機制。您也可以透過人工因素儲存庫，建立並共用元件做為可重複使用的程式碼庫，例如 AWS CodeArtifact.

每個應用程序都從單個存儲庫中的單個軟件包開始

單個軟件包是應 AWS CDK 用程序的進入點。在這裡，您可以定義如何以及在何處部署應用程序的不同邏輯單元。您也可以定義 CI/CD 管線來部署應用程式。該應用程序的構造定義了解決方案的邏輯單元。

針對您在多個應用程式中使用的建構使用其他套件。（共享構造也應該有自己的生命週期和測試策略。）同一個存儲庫中的軟件包之間的依賴關係由您的存儲庫的構建工具管理。

雖然這是可能的，但我們不建議將多個應用程式放在同一個儲存庫中，尤其是在使用自動化部署管線時。這樣做會增加部署期間變更的「爆炸半徑」。當儲存庫中有多個應用程式時，對一個應用程式的變更會觸發其他應用程式的部署 (即使其他應用程式尚未變更)。此外，在一個應用程式中斷會防止其他應用程式被部署。

根據程式碼生命週期或團隊擁有權將程式碼移至儲存

當套件開始在多個應用程式中使用時，請將它們移至自己的儲存庫。如此一來，使用這些套件的應用程式建置系統就可以參考這些套件，也可以在不受應用程式生命週期影響的節奏上進行更新。但是，首先將所有共享結構放在一個存儲庫中可能是有意義的。

此外，當不同的團隊正在處理套件時，將套件移至其自己的儲存庫。這有助於強制執行存取控制。

若要跨儲存庫邊界使用套件，您需要一個私有套件儲存庫 — 類似於 NPM PyPi、或 Maven Central，但是組織內部。您也需要一個發行程序來建置、測試套裝程式，並將其發佈至私有套裝程式儲存區域。CodeArtifact可以託管大多數流行編程語言的軟件包。

套件儲存庫中套件的相依性是由您語言的套件管理員所管理，例如 NPM for TypeScript or JavaScript 應用程式。您的軟件包管理器有助於確保構建是可重複的。它通過記錄應用程序所依賴的每個軟件包的特定版本來實現此目的。它還允許您以受控的方式升級這些依賴關係。

共用套件需要不同的測試策略。對於單個應用程序，將應用程序部署到測試環境並確認它仍然可以正常工作可能足夠好。但是，共享軟件包必須獨立於消費應用程序進行測試，就好像它們被發布給公眾一樣。(您的組織可能會選擇實際發行一些共用套件給大眾。)

請記住，構造可以是任意簡單或複雜的。A `Bucket` 是一個構造，但也`CameraShopWebsite`可以是一個構造。

基礎結構和運行時代碼存在於同一個包中

除了產生用於部署基礎設施的 AWS CloudFormation 範本之外， AWS CDK 還會將 Lambda 函數和 Docker 映像等執行階段資產捆綁在一起，並將它們與基礎設施一起部署。這樣就可以將定義基礎結構的程式碼和實作執行階段邏輯的程式碼結合成單一建構。這是執行此操作的最佳做法。這兩種程式碼不需要位於單獨的儲存庫中，甚至不需要位於單獨的套件中。

若要同時發展這兩種程式碼，您可以使用完整描述功能 (包括其基礎結構和邏輯) 的獨立建構。使用獨立建構，您可以隔離測試這兩種程式碼、跨專案共用和重複使用程式碼，以及同步化所有程式碼的版本。

建構最佳做法

本節包含開發建構的最佳作法。建構是封裝資源的可重複使用、可組合的模組。它們是 AWS CDK 應用程式的建置區塊。

具有構造的模型，使用堆棧部署

堆棧是部署的單元：堆棧中的所有內容都一起部署。因此，當從多個 AWS 資源構建應用程序的更高級別的邏輯單元時，將每個邏輯單元表示為 a `Construct`，而不是. `Stack` 只使用堆疊來描述如何針對各種部署案例組成和連接建構。

例如，如果其中一個邏輯單元是網站，則組成它的建構 (例如 Amazon S3 儲存貯體、API Gateway、Lambda 函數或 Amazon RDS 表) 應該組成為單一高層建構。然後，該構造應該在一個或多個堆棧中實例化以進行部署。

透過使用建構來建置和部署的堆疊，您可以提高基礎結構的重複使用潛力，並在部署方式上提供更大的彈性。

使用屬性和方法進行配置，而不是環境變量

構造和堆棧中的環境變量查找是常見的反模式。構造和堆棧都應該接受一個屬性對象，以便完全在代碼中完全可配置。否則會引入執行程式碼之機器的相依性，這會建立您必須追蹤和管理的更多設定資訊。

一般而言，環境變數查詢應限制在應 AWS CDK 用程式的最上層。它們也應該用來傳遞在開發環境中執行所需的資訊。如需詳細資訊，請參閱 the section called “環境”。

單元測試您的基礎結

要在所有環境中始終如一地在構建時運行完整的單元測試套件，請避免在合成期間進行網絡查找，並在代碼中對所有生產階段進行模型。（這些最佳做法將在稍後介紹。）如果任何單次提交總是導致相同的生成模板，則可以信任您編寫的單元測試，以確認生成的模板看起來像您期望的那樣。如需詳細資訊，請參閱 測試結構。

不要變更可設定狀態資源的邏輯 ID

變更資源的邏輯 ID 會在下次部署時將資源取代為新的邏輯 ID。對於資料庫和 S3 儲存貯體等可設定狀態資源，或 Amazon VPC 等持續性基礎設施，這很少是您想要的。請注意任何可能導致 ID 更改的 AWS CDK 代碼重構。編寫單元測試，聲明有狀態資源的邏輯 ID 保持靜態。邏輯 ID 衍生自`id`您在實例化建構時所指定的，以及建構在建構樹中的位置。如需詳細資訊，請參閱 the section called “邏輯識別碼”。

構造是不夠的合規

許多企業客戶為 L2 構造編寫自己的包裝函數（「策劃」構造，它們代表具有內置理智默認值和最佳實踐的個別 AWS 資源）。這些包裝函式會強制執行安全性最佳做法，例如靜態加密和特定 IAM 政策。例如，您可以建立一個`MyCompanyBucket`，然後在應用程式中使用它來取代一般的 Amazon S3 建`Bucket`構。這種模式對於在軟體開發生命週期的早期呈現安全性指引非常有用，但不要依賴它作為執行的唯一手段。

而是使用服務控制原則和權限界限等 AWS 功能，在組織層級強制執行安全防護。在部署之前，使用the section called “面向”或諸如 CloudFormation Guard 之類的工具，對基礎結構元素的安全性屬性進行宣告。用 AWS CDK 於它最擅長的東西。

最後，請記住，撰寫您自己的「L2+」結構可能會阻止您的開發人員利用 AWS CDK 套件，例如AWS 解決方案結構或來自 Construc t Hub 的協力廠商建構。這些套件通常建立在標準 AWS CDK 結構上，而且無法使用您的包裝函式結構。

應用程式最佳做

在本節中，我們將討論如何編寫 AWS CDK 應用程序，結合構造來定義資 AWS 源的連接方式。

在合成時間做出決定

雖然 AWS CloudFormation 可讓您在部署階段做出決策 (使用Conditions{ Fn::If }、和Parameters)，並且可 AWS CDK 讓您存取這些機制，但我們建議您不要使用它們。與通用程式設計語言中可用的值類型相比，您可以使用的值類型以及可對其執行的操作類型受到限制。

相反地，請嘗試使用程式設計語言的if陳述式和其他功能，在 AWS CDK 應用程式中做出所有決定，例如要實體化的建構。例如，一個常見的 CDK 習慣用法，迭代列表並使用列表中每個項目的值實例化構造，根本不可能使用表達式。 AWS CloudFormation

視 AWS CloudFormation 為 AWS CDK 用於強大雲端部署的實作詳細資料，而不是作為語言目標。您不是在 Python 中 TypeScript 編寫 AWS CloudFormation 模板，而是在編寫碰巧用 CloudFormation 於部署的 CDK 代碼。

使用產生的資源名稱，而非實體名稱

名字是寶貴的資源。每個名稱只能使用一次。因此，如果您將資料表名稱或值區名稱硬式編碼到基礎結構和應用程式中，則無法在同一帳戶中部署兩次該基礎結構。（我們在這裡討論的名稱是由 Amazon S3 存儲桶構造上的bucketName屬性指定的名稱。）

更糟糕的是，您無法對需要更換它的資源進行更改。如果只能在資源建立時設定屬性 (例如 Amazon DynamoDB 表格)，則該屬性不可變。KeySchema變更此屬性需要新的資源。但是，新資源必須具有相同的名稱，才能成為真正的替代品。但是當現有資源仍在使用該名稱時，它不能具有相同的名稱。

更好的方法是指定盡可能少的名稱。如果您省略資源名稱， AWS CDK 將以不會導致問題的方式為您生成它們。假設你有一個表作為資源。然後，您可以將生成的表名作為環境變量傳遞到 AWS Lambda 函數中。在您的 AWS CDK 應用程序中，您可以將表名稱引用為table.tableName。或者，您可以在啟動時在 Amazon EC2 執行個體上產生組態檔，或將實際的表名稱寫入 AWS Systems Manager 參數存放區，以便您的應用程式可以從該處讀取該檔案。

如果你需要它的地方是另一個 AWS CDK 堆棧，那就更直接了。假設一個堆棧定義了資源，另一個堆棧需要使用它，以下情況適用：

- 如果兩個堆棧位於同一個 AWS CDK 應用程序中，請在兩個堆棧之間傳遞引用。例如，將對資源構造的引用保存為定義的 stack (this.stack.uploadBucket = myBucket) 的屬性。然後，將該屬性傳遞給需要資源的堆棧的構造函數。
- 當兩個堆棧位於不同的 AWS CDK 應用程序中時，請使用靜態from方法根據其 ARN，名稱或其他屬性使用外部定義的資源。(例如，用Table.fromArn()於動 DynamoDB 料表)。使

用`CfnOutput`建構在的輸出中列印 ARN 或其他所需值`cdk deploy`，或在中 AWS Management Console查看。或者，第二個應用程序可以讀取第一個應用程序生成的 CloudFormation模板，並從該`Outputs`部分中檢索該值。

定義移除原則和記錄保留

預設為保留您所建立之所有項目的原則，以防止您遺失資料的 AWS CDK 嘗試。例如，包含資料 (例如 Amazon S3 儲存貯體和資料庫表格) 的資源的預設移除政策不會在資源從堆疊中移除資源時刪除該資源。相反，該資源是從堆棧中孤立的。同樣地，CDK 的預設值是永久保留所有記錄檔。在生產環境中，這些預設值可能會快速儲存您實際上不需要的大量資料，以及對應的 AWS 帳單。

請仔細考慮您希望這些原則適用於每個生產資源的原則，並相應地指定它們。用the section called “面向”於驗證堆疊中的移除和記錄原則。

根據部署需求，將您的應用程式分成多個堆疊

您的應用程序需要多少堆棧沒有硬性和快速的規則。您通常最終會根據部署模式做出決定。請記住以下準則：

- 將盡可能多的資源盡可能多地保留在同一堆棧中通常更直接，因此除非您知道要將它們分開，否則請將它們保持在一起。
- 考慮將有狀態資源（如數據庫）與無狀態資源保存在單獨的堆棧中。然後，您可以在可設定狀態堆疊上開啟終止保護。如此一來，您就可以自由銷毀或建立無狀態堆疊的多個副本，而不會造成資料遺失的風險。
- 有狀態資源對於建構重新命名更敏感 — 重新命名會導致資源取代。因此，請勿將有狀態資源嵌套在可能移動或重新命名的建構內 (除非遺失時可重建狀態，例如快取)。這是將有狀態資源放在自己的堆棧中的另一個很好的理由。

提交`cdk.context.json`以避免非確定性行為

決定性是成功 AWS CDK 部署的關鍵。每當 AWS CDK 應用程序部署到給定的環境時，應該具有基本相同的結果。

由於您的 AWS CDK 應用程序是以通用編程語言編寫的，因此它可以執行任意代碼，使用任意庫並進行任意網絡調用。例如，您可以在合成應用程序時使用 AWS SDK 從您的 AWS 帳戶中檢索某些信息。認識到這樣做會導致額外的認證設置要求，增加延遲，並且每次運行時都有很小的機會失敗`cdk synth`。

在合成過程中，切勿修改您的 AWS 帳戶或資源。合成應用程序不應該有副作用。只有在產生 AWS CloudFormation 範本之後，才會在部署階段變更基礎結構。這樣，如果有問題， AWS CloudFormation 可以自動回滾更改。要進行無法在 AWS CDK 框架中輕鬆進行的更改，請使用自定義資源在部署時執行任意代碼。

即使嚴格只讀調用也不一定安全。考慮如果網路呼叫傳回的值變更，會發生什麼情況。您的基礎架構的哪一部分會影響？ 已部署的資源會發生什麼情況？ 以下是其中值突然改變可能會導致一個問題兩個例子的情況。

- 如果您將 Amazon VPC 佈建到指定區域中的所有可用區域，且部署日的 AZ 數量為兩個，則您的 IP 空間會分成兩半。如果第二天 AWS 啟動新的可用區域，則在此之後的下一個部署會嘗試將 IP 空間分割成三分之一，需要重新建立所有子網路。這可能是不可能的，因為您的 Amazon EC2 實例仍在運行，您必須手動清理它。
- 如果您查詢最新的 Amazon Linux 機器映像檔並部署 Amazon EC2 執行個體，並在第二天發行新映像，則後續部署會選取新的 AMI 並取代您的所有執行個體。這可能不是您所期望發生的事情。

這些情況可能是有害的，因為 AWS-side 變更可能會在成功部署數月或數年後發生。突然間，您的部署「無緣無故」失敗，您很久以前就忘記了自己做了什麼以及為什麼。

幸運的是， AWS CDK 包括一種稱為上下文提供者的機制，用於記錄非確定性值的快照。這可讓 future 的合成作業產生與第一次部署時完全相同的範本。新範本中唯一的變更是您在程式碼中所做的變更。當你使用一個構造的`.fromLookup()`方法，調用的結果被緩存在`cdk.context.json`。您應該將其與代碼的其餘部分一起提交給版本控制，以確保 future 的 CDK 應用程序執行使用相同的值。CDK Toolkit 包含用於管理內容快取的命令，因此您可以在需要時重新整理特定項目。如需詳細資訊，請參閱 the section called “Context”。

如果您需要一些沒有原生 CDK 內容提供者的值（來自 AWS 或其他地方），我們建議您編寫單獨的腳本。指令碼應擷取值並將其寫入檔案，然後在 CDK 應用程式中讀取該檔案。只有當您想要重新整理儲存的值時，才執行指令碼，而不是做為一般建置程序的一部分。

讓 AWS CDK 管理角色和安全組

使用 AWS CDK 建構程式庫的`grant()`便利方法，您可以建立 AWS Identity and Access Management 角色，使用最小範圍的許可授與另一個資源的存取權。例如，請考慮如下所示的一行：

```
myBucket.grantRead(myLambda)
```

這一行將政策添加到 Lambda 函數的角色（也是為您創建的）。這個角色及其政策是十幾行 CloudFormation ，你不必寫。僅 AWS CDK 授予函數從值區讀取所需的最低權限。

如果您要求開發人員始終使用由安全團隊創建的預定義角色，則 AWS CDK 編碼變得更加複雜。您的團隊在設計應用程式方面可能會失去很多彈性。更好的替代方法是使用服務控制政策和權限界限，以確保開發人員保持在護欄內。

在代碼中為所有生產階段建模

在傳統 AWS CloudFormation 案例中，您的目標是產生一個參數化的單一成品，以便在套用特定於這些環境的組態值之後，將其部署到各種目標環境中。在 CDK 中，您可以並且應該將該配置構建到源代碼中。為您的生產環境建立堆疊，並為其他每個階段建立單獨的堆疊。然後，將每個堆棧的配置值放在代碼中。使用 Secrets Manager 和 Systems Manager 參數存放區等服務來取得您不想簽入原始檔控制的敏感值，使用這些資源的名稱或 ARN。

當您合成應用程式時，在`cdk.out`資料夾中建立的雲端組件會包含每個環境的個別範本。您的整個構建是確定性的。您的應用程序沒有 out-of-band更改，任何給定的提交始終產生完全相同的 AWS CloudFormation 模板和隨附的資產。這使得單元測試更加可靠。

衡量一切

實現完全持續部署的目標，無需人工干預，需要高水平的自動化。只有大量監控才能實現自動化。若要衡量已部署資源的所有層面，請建立指標、警示和儀表板。不要停止測量 CPU 使用率和磁盤空間之類的東西。同時記錄您的業務指標，並使用這些測量來自動化部署決策，例如復原。中的大部分 L2 建構 AWS CDK 都有方便的方法來協助您建立度量，例如 Dynamo DB.Table 類別上的`metricUserErrors()`方法。

AWS CDK 參考

本節包含的參考資訊 AWS Cloud Development Kit (AWS CDK)。

主題

- API 參考
- AWS CDK 版本化

API 參考

API 參考包含有關 AWS 建構程式庫和其他 API 的資訊 AWS Cloud Development Kit (AWS CDK)。大多數 AWS 構造庫都包含在一個TypeScript名為：的單個包中`aws-cdk-lib`。實際的軟件包名稱因語言而異。針對每種受支援的程式設計語言，都會提供不同的 API 參考版本。

CDK API 參考被組織成子模塊。每個子模塊都有一個或多個 AWS 服務子模塊。

每個子模塊都有一個概述，其中包括有關如何使用其 API 的信息。例如，S3 概觀示範如何在 Amazon Simple Storage Service (Amazon S3) 儲存貯體上設定預設加密。

AWS CDK 版本化

本主題提供有關如何 AWS Cloud Development Kit (AWS CDK) 處理版本化的參考資訊。

版本號由三個數字版本部分組成：主要。 未成年人。 補丁，並嚴格遵守語義版本模型。這意味著對穩定 API 的重大更改僅限於主要版本。

次要版本和修補程式版本與回溯相容。以相同主要版本的舊版本撰寫的程式碼可以升級至相同主要版本中的較新版本。它也將繼續構建和運行，產生相同的輸出。

主題

- AWS CDKCLI兼容性
- AWS 建構程式庫版本
- 語言綁定穩定性

AWS CDKCLI兼容性

始終與語義上較低或相等版本號的構造庫兼容。 AWS CDK CLI因此，在同一主要版本 AWS CDK CLI 中升級始終是安全的。

並不總 AWS CDK CLI是與語義上更高版本的構造庫兼容。相容性取決於兩個元件是否使用相同的雲端組件結構描述版本。該 AWS CDK 框架在合成過程中生成一個雲程序集， AWS CDK CLI並將其用於部署。嚴格指定定義雲端組合格式的結構描述並進行版本化。

AWS 使用指定的雲端組合資料架構版 AWS CDK CLI本建構資源庫與使用該資料架構版本或更新版本的版本相容。這可能包括比給定建構程式庫版本還早的發行版本。 AWS CDK CLI

當建構資源庫所需的雲端組件版本與支援的版本不相容時 AWS CDK CLI，您會收到類似下列的錯誤訊息：

```
Cloud assembly schema version mismatch: Maximum schema version supported is 3.0.0, but
 found 4.0.0.
    Please upgrade your CLI in order to interact with this app.
```

若要解決此錯誤，請更新 AWS CDK CLI為與所需雲端組件版本相容的版本，或更新至最新的可用版本。通常不建議使用替代方法（降級您的應用程序使用的構造庫模塊）。

> **Note**
>
> 若要取得有關雲端組合資料架構的更多詳細資訊，請參閱〈雲端組

AWS 建構程式庫版本

在 AWS 構造庫中的模塊移動通過各個階段，因為它們是從概念開發到成熟的 API。不同階段在後續版本中提供不同程度的 API 穩定性 AWS CDK。

主 AWS CDK 庫中的 API 是穩定的`aws-cdk-lib`，並且庫在語義上是完全版本化的。此套件包含 AWS CloudFormation 適用於所有 AWS 服務和所有穩定較高層級 (L2 和 L3) 模組的 (L1) 建構。（它還包括核心 CDK 類，如`App`和`Stack`）。在 CDK 的下一個主要發行版本之前，API 不會從此套件中移除 (儘管它們可能已被取代)。任何個別 API 都不會有突破性的變化。當需要突破性更改時，將添加一個全新的 API。

針對已納入的服務開發中的新 API `aws-cdk-lib` 會使用Beta*N*尾碼來識別，其中從 1 N 開始，並隨著新 API 的每次重大變更而遞增。 Beta*N*API 永遠不會移除，只會取代，因此您現有的應用程式可繼續使用`aws-cdk-lib`. 當 API 被視為穩定時，將添加一個沒有後Beta*N*綴的新 API。

當較高層級 (L2 或 L3) API 開始針對先前只有 L1 API 的 AWS 服務開發時，這些 API 一開始會分散在單獨的套件中。這種軟件包的名稱具有「Alpha」後綴，其版本與第一個版本兼容，具有`alpha`子版本。`aws-cdk-lib`當模組支援預期的使用案例時，會將其 API 新增至`aws-cdk-lib`。

語言綁定穩定性

隨著時間的推移，我們可能會增加 AWS CDK 對其他程式設計語言的支援。儘管所有語言中描述的 API 都是相同的，但 API 的表達方式會因語言而異，並且可能會隨著語言支持的發展而改變。出於這個原因，語言綁定被視為一段時間的實驗性，直到它們被認為是準備好用於生產使用。

Language	Stability
TypeScript	Stable
JavaScript	Stable
Python	Stable
Java	Stable
C#/.NET	Stable
Go	Stable

AWS CDK 教程

本節包含的自學課程 AWS Cloud Development Kit (AWS CDK)。

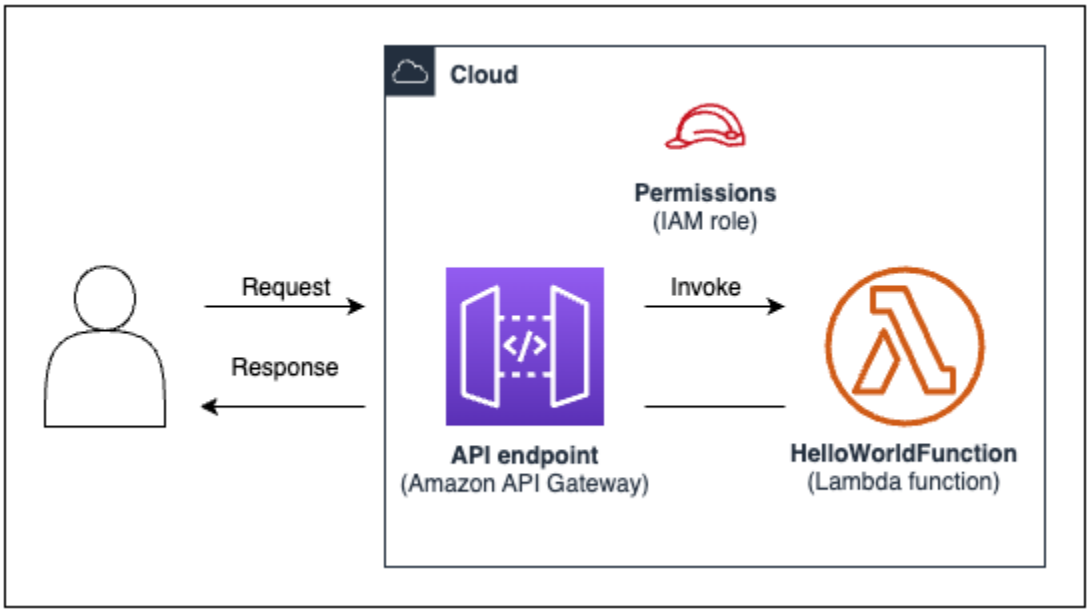
主題

建立無伺服器的 Hello World 應用程式

在本教學課程中，您可以透過建立下列項目 AWS Cloud Development Kit (AWS CDK) 來建立簡單的無伺服器Hello World應用程式，以實作基本 API 後端：

- Amazon API Gateway REST API — 提供用於透過HTTP GET請求叫用函數的 HTTP 端點。
- AWS Lambda function — 使用HTTP端點叫用時傳回`Hello World!`訊息的函數。
- 整合和許可 — 讓您的資源彼此互動並執行動作 (例如將日誌寫入 Amazon) 的組態詳細資訊和許可 CloudWatch。

下圖顯示此應用程式的組件：



對於本教學課程，您將完成以下操作：

1. 建立 AWS CDK 專案。

2. 使用建構程式庫中的 L2 AWS 建構來定義 Lambda 函數和 API Gateway REST API。
3. 將您的應用程式部署到 AWS 雲端.
4. 與中的應用程式互動 AWS 雲端。
5. 從中刪除範例應用程式 AWS 雲端。

必要條件

開始本教學課程之前，請完成下列動作：

- 創建一個 AWS 帳戶 並安裝和配置 AWS Command Line Interface （ AWS CLI ）。
- 安裝Node.js和npm。
- 在全球範圍內安裝 CDK 工具組，使用`npm install -g aws-cdk`.

如需詳細資訊，請參閱 開始使用 AWS CDK。

我們還建議對以下內容有基本的了解：

- 什麼是 AWS CDK？有關的基本介紹 AWS CDK.
- AWS CDK 概念以取得的核心概念概念的概觀 AWS CDK。

步驟 1：建立 CDK 專案

在此步驟中，您要使用 AWS CDK CLI`cdk init`指令建立新 CDK 專案。

若要建立 CDK 專案

1. 從您選擇的起始目錄中，創建並導航到計算機`cdk-hello-world`上名為的項目目錄：

   ```
   $ mkdir cdk-hello-world && cd cdk-hello-world
   ```

2. 使用指`cdk init`令以您偏好的程式設計語言建立新專案：

   TypeScript

   ```
   $ cdk init --language typescript
   ```

   安裝 AWS CDK 程式庫：

```
$ npm install aws-cdk-lib constructs
```

JavaScript

```
$ cdk init --language javascript
```

安裝 AWS CDK 程式庫：

```
$ npm install aws-cdk-lib constructs
```

Python

```
$ cdk init --language python
```

啟用虛擬環境：

```
$ source .venv/bin/activate
```

安裝 AWS CDK 庫和項目依賴關係：

```
(.venv)$ python3 -m pip install -r requirements.txt
```

Java

```
$ cdk init --language java
```

安裝 AWS CDK 庫和項目依賴關係：

```
$ mvn package
```

C#

```
$ cdk init --language csharp
```

安裝 AWS CDK 庫和項目依賴關係：

```
$ dotnet restore src
```

Go

```
$ cdk init --language go
```

安裝項目依賴關係：

```
$ go mod tidy
```

CDK CLI 創建具有以下結構的項目：

TypeScript

```
cdk-hello-world
### .git
### .gitignore
### .npmignore
### README.md
### bin
#   ### cdk-hello-world.ts
### cdk.json
### jest.config.js
### lib
#   ### cdk-hello-world-stack.ts
### node_modules
### package-lock.json
### package.json
### test
#   ### cdk-hello-world.test.ts
### tsconfig.json
```

JavaScript

```
cdk-hello-world
### .git
### .gitignore
### .npmignore
### README.md
```

```
### bin
#   ### cdk-hello-world.js
### cdk.json
### jest.config.js
### lib
#   ### cdk-hello-world-stack.js
### node_modules
### package-lock.json
### package.json
### test
    ### cdk-hello-world.test.js
```

Python

```
cdk-hello-world
### .git
### .gitignore
### .venv
### README.md
### app.py
### cdk.json
### cdk_hello_world
#   ### __init__.py
#   ### cdk_hello_world_stack.py
### requirements-dev.txt
### requirements.txt
### source.bat
### tests
```

Java

```
cdk-hello-world
### .git
### .gitignore
### README.md
### cdk.json
### pom.xml
### src
#   ### main
#   #   ### java
#   #       ### com
#   #           ### myorg
#   #               ### CdkHelloWorldApp.java
```

```
#   #               ### CdkHelloWorldStack.java
### target
```

C#

```
cdk-hello-world
### .git
### .gitignore
### README.md
### cdk.json
### src
    ### CdkHelloWorld
    #   ### CdkHelloWorld.csproj
    #   ### CdkHelloWorldStack.cs
    #   ### GlobalSuppressions.cs
    #   ### Program.cs
    ### CdkHelloWorld.sln
```

Go

```
cdk-hello-world
### .git
### .gitignore
### README.md
### cdk-hello-world.go
### cdk-hello-world_test.go
### cdk.json
### go.mod
```

CDK CLI 會自動建立包含單一堆疊的 CDK 應用程式。CDK 應用程序實例是從App類創建的。以下是 CDK 應用程式檔案的一部分：

TypeScript

位於`bin/cdk-hello-world.ts`：

```
#!/usr/bin/env node
import 'source-map-support/register';
import * as cdk from 'aws-cdk-lib';
import { CdkHelloWorldStack } from '../lib/cdk-hello-world-stack';
```

```
const app = new cdk.App();
new CdkHelloWorldStack(app, 'CdkHelloWorldStack', {
});
```

JavaScript

位於`bin/cdk-hello-world.js`：

```
#!/usr/bin/env node
const cdk = require('aws-cdk-lib');
const { CdkHelloWorldStack } = require('../lib/cdk-hello-world-stack');
const app = new cdk.App();
new CdkHelloWorldStack(app, 'CdkHelloWorldStack', {
});
```

Python

位於`app.py`：

```
#!/usr/bin/env python3
import os
import aws_cdk as cdk
from cdk_hello_world.cdk_hello_world_stack import CdkHelloWorldStack

app = cdk.App()
CdkHelloWorldStack(app, "CdkHelloWorldStack",)
app.synth()
```

Java

位於`src/main/java/.../CdkHelloWorldApp.java`：

```
package com.myorg;

import software.amazon.awscdk.App;
import software.amazon.awscdk.Environment;
import software.amazon.awscdk.StackProps;

import java.util.Arrays;

public class JavaApp {
    public static void main(final String[] args) {
        App app = new App();
```

```
        new JavaStack(app, "JavaStack", StackProps.builder()
                .build());

        app.synth();
    }
}
```

C#

位於src/CdkHelloWorld/Program.cs：

```
using Amazon.CDK;
using System;
using System.Collections.Generic;
using System.Linq;

namespace CdkHelloWorld
{
    sealed class Program
    {
        public static void Main(string[] args)
        {
            var app = new App();
            new CdkHelloWorldStack(app, "CdkHelloWorldStack", new StackProps
            {

            });
            app.Synth();
        }
    }
}
```

Go

位於cdk-hello-world.go：

```
package main
import (
    "github.com/aws/aws-cdk-go/awscdk/v2"
    "github.com/aws/constructs-go/constructs/v10"
    "github.com/aws/jsii-runtime-go"
)
```

```
// ...

func main() {
    defer jsii.Close()
    app := awscdk.NewApp(nil)
    NewCdkHelloWorldStack(app, "CdkHelloWorldStack", &CdkHelloWorldStackProps{
        awscdk.StackProps{
            Env: env(),
        },
    })
    app.Synth(nil)
}

func env() *awscdk.Environment {
    return nil
}
```

步驟 2：建立您的 Lambda 函數

在 CDK 專案中，建立包含新`hello.js`檔案的`lambda`目錄。以下是範例：

TypeScript

從項目的根目錄中，運行以下命令：

```
$ mkdir lambda && cd lambda
$ touch hello.js
```

現在應該將以下內容添加到您的 CDK 項目中：

```
cdk-hello-world
### lambda
    ### hello.js
```

JavaScript

從項目的根目錄中，運行以下命令：

```
$ mkdir lambda && cd lambda
$ touch hello.js
```

現在應該將以下內容添加到您的 CDK 項目中：

```
cdk-hello-world
### lambda
    ### hello.js
```

Python

從項目的根目錄中，運行以下命令：

```
$ mkdir lambda && cd lambda
$ touch hello.js
```

現在應該將以下內容添加到您的 CDK 項目中：

```
cdk-hello-world
### lambda
    ### hello.js
```

Java

從項目的根目錄中，運行以下命令：

```
$ mkdir -p src/main/resources/lambda
$ cd src/main/resources/lambda
$ touch hello.js
```

現在應該將以下內容添加到您的 CDK 項目中：

```
cdk-hello-world
### src
    ### main
        ###resources
            ###lambda
                ###hello.js
```

C#

從項目的根目錄中，運行以下命令：

```
$ mkdir lambda && cd lambda
```

```
$ touch hello.js
```

現在應該將以下內容添加到您的 CDK 項目中：

```
cdk-hello-world
### lambda
    ### hello.js
```

Go

從項目的根目錄中，運行以下命令：

```
$ mkdir lambda && cd lambda
$ touch hello.js
```

現在應該將以下內容添加到您的 CDK 項目中：

```
cdk-hello-world
### lambda
    ### hello.js
```

> **Note**
>
> 為了簡化本教程，我們對所有 CDK 編程語言使用 JavaScript Lambda 函數。

將下列項目新增至新建立的檔案，以定義 Lambda 函數：

```
exports.handler = async (event) => {
    return {
        statusCode: 200,
        headers: { "Content-Type": "text/plain" },
        body: JSON.stringify({ message: "Hello, World!" }),
    };
};
```

步驟 3：定義您的建構

在此步驟中，您將使用 AWS CDK L2 建構來定義 Lambda 和 API Gateway 資源。

開啟定義 CDK 堆疊的專案檔案。您將修改此檔案以定義您的建構。以下是起始堆棧文件的示例：

TypeScript

位於`lib/cdk-hello-world-stack.ts`：

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';

export class CdkHelloWorldStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Your constructs will go here

  }
}
```

JavaScript

位於`lib/cdk-hello-world-stack.js`：

```
const { Stack, Duration } = require('aws-cdk-lib');
const lambda = require('aws-cdk-lib/aws-lambda');
const apigateway = require('aws-cdk-lib/aws-apigateway');

class CdkHelloWorldStack extends Stack {

  constructor(scope, id, props) {
    super(scope, id, props);

    // Your constructs will go here

  }
}

module.exports = { CdkHelloWorldStack }
```

Python

位於`cdk_hello_world/cdk_hello_world_stack.py`：

```
from aws_cdk import Stack
```

```
from constructs import Construct

class CdkHelloWorldStack(Stack):
    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

          // Your constructs will go here
```

Java

位於src/main/java/.../CdkHelloWorldStack.java：

```
package com.myorg;

import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;

public class CdkHelloWorldStack extends Stack {
    public CdkHelloWorldStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public CdkHelloWorldStack(final Construct scope, final String id, final
 StackProps props) {
        super(scope, id, props);

        // Your constructs will go here
    }
}
```

C#

位於src/CdkHelloWorld/CdkHelloWorldStack.cs：

```
using Amazon.CDK;
using Constructs;

namespace CdkHelloWorld
{
    public class CdkHelloWorldStack : Stack
    {
```

```
        internal CdkHelloWorldStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
            // Your constructs will go here
        }
    }
}
```

Go

位於cdk-hello-world.go：

```
package main
import (
    "github.com/aws/aws-cdk-go/awscdk/v2"
    "github.com/aws/constructs-go/constructs/v10"
    "github.com/aws/jsii-runtime-go"
)
type CdkHelloWorldStackProps struct {
    awscdk.StackProps
}
func NewCdkHelloWorldStack(scope constructs.Construct, id string, props
 *CdkHelloWorldStackProps) awscdk.Stack {
    var sprops awscdk.StackProps
    if props != nil {
        sprops = props.StackProps
    }
    stack := awscdk.NewStack(scope, &id, &sprops)
    // Your constructs will go here
    return stack
}
func main() {
    // ...
}

func env() *awscdk.Environment {
    return nil
}
```

在這個文件中，AWS CDK 正在執行以下操作：

- 您的 CDK 堆棧實例是從類中實例化的Stack。

- Constructs基類被導入並作為堆棧實例的作用域或父級提供。

定義您的 Lambda 函數資源

若要定義 Lambda 函數資源，您可以從「建構程式庫」匯入並使用 aws-lambda L2 AWS 建構。

修改堆棧文件，如下所示：

TypeScript

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
// Import Lambda L2 construct
import * as lambda from 'aws-cdk-lib/aws-lambda';

export class CdkHelloWorldStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Define the Lambda function resource
    const helloWorldFunction = new lambda.Function(this, 'HelloWorldFunction', {
      runtime: lambda.Runtime.NODEJS_20_X, // Choose any supported Node.js runtime
      code: lambda.Code.fromAsset('lambda'), // Points to the lambda directory
      handler: 'hello.handler', // Points to the 'hello' file in the lambda
 directory
    });
  }
}
```

JavaScript

```
const { Stack, Duration } = require('aws-cdk-lib');
// Import Lambda L2 construct
const lambda = require('aws-cdk-lib/aws-lambda');


class CdkHelloWorldStack extends Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    // Define the Lambda function resource
    const helloWorldFunction = new lambda.Function(this, 'HelloWorldFunction', {
```

```
      runtime: lambda.Runtime.NODEJS_20_X, // Choose any supported Node.js runtime
      code: lambda.Code.fromAsset('lambda'), // Points to the lambda directory
      handler: 'hello.handler', // Points to the 'hello' file in the lambda
 directory
    });
  }
}

module.exports = { CdkHelloWorldStack }
```

Python

```
from aws_cdk import (
    Stack,
    # Import Lambda L2 construct
    aws_lambda as _lambda,
)
# ...

class CdkHelloWorldStack(Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        # Define the Lambda function resource
        hello_world_function = _lambda.Function(
            self,
            "HelloWorldFunction",
            runtime = _lambda.Runtime.NODEJS_20_X, # Choose any supported Node.js
 runtime
            code = _lambda.Code.from_asset("lambda"), # Points to the lambda
 directory
            handler = "hello.handler", # Points to the 'hello' file in the lambda
 directory
        )
```

> **Note**
>
> 我們導入aws_lambda模塊，_lambda因為它lambda是中Python的內置標識符。

Java

```
// ...
// Import Lambda L2 construct
import software.amazon.awscdk.services.lambda.Code;
import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.lambda.Runtime;

public class CdkHelloWorldStack extends Stack {
    public CdkHelloWorldStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public CdkHelloWorldStack(final Construct scope, final String id, final
 StackProps props) {
        super(scope, id, props);

        // Define the Lambda function resource
        Function helloWorldFunction = Function.Builder.create(this,
 "HelloWorldFunction")
                .runtime(Runtime.NODEJS_20_X)  // Choose any supported Node.js
 runtime
                .code(Code.fromAsset("src/main/resources/lambda")) // Points to the
 lambda directory
                .handler("hello.handler")  // Points to the 'hello' file in the
 lambda directory
                .build();
    }
}
```

C#

```
// ...
// Import Lambda L2 construct
using Amazon.CDK.AWS.Lambda;

namespace CdkHelloWorld
{
    public class CdkHelloWorldStack : Stack
    {
        internal CdkHelloWorldStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
```

```
            // Define the Lambda function resource
            var helloWorldFunction = new Function(this, "HelloWorldFunction", new
 FunctionProps
            {
                Runtime = Runtime.NODEJS_20_X, // Choose any supported Node.js
 runtime
                Code = Code.FromAsset("lambda"), // Points to the lambda directory
                Handler = "hello.handler" // Points to the 'hello' file in the
 lambda directory
            });
        }
    }
}
```

Go

```
package main

import (
    // ...
    // Import Lambda L2 construct
    "github.com/aws/aws-cdk-go/awscdk/v2/awslambda"
    // ...
)

// ...

func NewCdkHelloWorldStack(scope constructs.Construct, id string, props
 *CdkHelloWorldStackProps) awscdk.Stack {
    var sprops awscdk.StackProps
    if props != nil {
        sprops = props.StackProps
    }
    stack := awscdk.NewStack(scope, &id, &sprops)

    // Define the Lambda function resource
    helloWorldFunction := awslambda.NewFunction(stack,
 jsii.String("HelloWorldFunction"), &awslambda.FunctionProps{
        Runtime: awslambda.Runtime_NODEJS_20_X(), // Choose any supported Node.js
 runtime
        Code:    awslambda.Code_FromAsset(jsii.String("lambda")), // Points to the
 lambda directory
```

```
        Handler: jsii.String("hello"), // Points to the 'hello' file in the lambda
 directory
    })

    return stack
}

// ...
```

在這裡，您可以建立 Lambda 函數資源並定義下列屬性：

- `runtime`— 函數執行的環境。在這裡，我們使用Node.js版本20.x。
- `code`— 本機電腦上函數程式碼的路徑。
- `handler`— 包含函數程式碼的特定檔案名稱。

定義您的 API Gateway REST API 資源

若要定義您的 API Gateway REST API 資源，請從「建構程式庫」匯入並使用 `aws-apigateway` L2 AWS 建構。

修改堆棧文件，如下所示：

TypeScript

```
// ...
//Import API Gateway L2 construct
import * as apigateway from 'aws-cdk-lib/aws-apigateway';


export class CdkHelloWorldStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // ...

    // Define the API Gateway resource
    const api = new apigateway.LambdaRestApi(this, 'HelloWorldApi', {
      handler: helloWorldFunction,
      proxy: false,
    });
```

```
    // Define the '/hello' resource with a GET method
    const helloResource = api.root.addResource('hello');
    helloResource.addMethod('GET');
  }
}
```

JavaScript

```
// ...
// Import API Gateway L2 construct
const apigateway = require('aws-cdk-lib/aws-apigateway');


class CdkHelloWorldStack extends Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    // ...

    // Define the API Gateway resource
    const api = new apigateway.LambdaRestApi(this, 'HelloWorldApi', {
      handler: helloWorldFunction,
      proxy: false,
    });

    // Define the '/hello' resource with a GET method
    const helloResource = api.root.addResource('hello');
    helloResource.addMethod('GET');
  };
};

// ...
```

Python

```
from aws_cdk import (
    # ...
    # Import API Gateway L2 construct
    aws_apigateway as apigateway,
)
from constructs import Construct

class CdkHelloWorldStack(Stack):
```

```
    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        # ...

        # Define the API Gateway resource
        api = apigateway.LambdaRestApi(
            self,
            "HelloWorldApi",
            handler = hello_world_function,
            proxy = False,
        )

        # Define the '/hello' resource with a GET method
        hello_resource = api.root.add_resource("hello")
        hello_resource.add_method("GET")
```

Java

```
// ...
// Import API Gateway L2 construct
import software.amazon.awscdk.services.apigateway.LambdaRestApi;
import software.amazon.awscdk.services.apigateway.Resource;

public class CdkHelloWorldStack extends Stack {
    public CdkHelloWorldStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public CdkHelloWorldStack(final Construct scope, final String id, final
 StackProps props) {
        super(scope, id, props);

        // ...

        // Define the API Gateway resource
        LambdaRestApi api = LambdaRestApi.Builder.create(this, "HelloWorldApi")
                .handler(helloWorldFunction)
                .proxy(false) // Turn off default proxy integration
                .build();

        // Define the '/hello' resource and its GET method
```

```
        Resource helloResource = api.getRoot().addResource("hello");
        helloResource.addMethod("GET");
    }
}
```

C#

```
// ...
// Import API Gateway L2 construct
using Amazon.CDK.AWS.APIGateway;

namespace CdkHelloWorld
{
    public class CdkHelloWorldStack : Stack
    {
        internal CdkHelloWorldStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
           // ...

            // Define the API Gateway resource
            var api = new LambdaRestApi(this, "HelloWorldApi", new
 LambdaRestApiProps
            {
                Handler = helloWorldFunction,
                Proxy = false
            });

            // Add a '/hello' resource with a GET method
            var helloResource = api.Root.AddResource("hello");
            helloResource.AddMethod("GET");
        }
    }
}
```

Go

```
// ...

import (
    // ...
    // Import Api Gateway L2 construct
```

```
    "github.com/aws/aws-cdk-go/awscdk/v2/awsapigateway"
    // ...
)

// ...

func NewCdkHelloWorldStack(scope constructs.Construct, id string, props
 *CdkHelloWorldStackProps) awscdk.Stack {
    var sprops awscdk.StackProps
    if props != nil {
        sprops = props.StackProps
    }
    stack := awscdk.NewStack(scope, &id, &sprops)

    // Define the Lambda function resource
    // ...

    // Define the API Gatweay resource
    api := awsapigateway.NewLambdaRestApi(stack, jsii.String("HelloWorldApi"),
 &awsapigateway.LambdaRestApiProps{
        Handler: helloWorldFunction,
        Proxy: jsii.Bool(false),
    })

    // Add a '/hello' resource with a GET method
    helloResource := api.Root().AddResource(jsii.String("hello"))
    helloResource.AddMethod(jsii.String("GET"))

    return stack
}

// ...
```

您可以在此建立 API Gateway REST API 資源，以及下列項目：

- REST API與您的 Lambda 函數之間的整合，可讓 API 叫用您的函數。這包括建立 Lambda 權限資源。
- 名為的新資源或路徑`hello`，會新增至 API 端點的根目錄。這將創建一個新的端點，並添加`/hello`到您的基礎URL。
- `hello`資源的 GET 方法。當 GET 要求傳送至`/hello`端點時，會叫用 Lambda 函數，並傳回其回應。

步驟 4：準備應用程式以進行部署

在此步驟中，您可以視需要建置並使用 AWS CDK CLIcdk synth命令執行基本驗證來準備應用程式以進行部署。

如有必要，建立您的應用程式：

TypeScript

從項目的根目錄中，運行以下命令：

```
$ npm run build
```

JavaScript

不需要建築物。

Python

不需要建築物。

Java

從項目的根目錄中，運行以下命令：

```
$ mvn package
```

C#

從項目的根目錄中，運行以下命令：

```
$ dotnet build src
```

Go

從項目的根目錄中，運行以下命令：

```
$ go build
```

運行cdk synth以從 CDK 代 AWS CloudFormation 碼合成模板。透過使用 L2 建構，許多組態詳細資料REST API都是為了促進 Lambda 函數之間的互動所需的，並由. AWS CloudFormation AWS CDK

從項目的根目錄中，運行以下命令：

```
$ cdk synth
```

> **Note**
>
> 如果您收到類似下列的錯誤訊息，請確認您位於cdk-hello-world目錄中，然後再試一次：
>
> ```
> --app is required either in command-line, in cdk.json or in ~/.cdk.json
> ```

如果成功， AWS CDK CLI將在命令提示符下以YAML格式輸出 AWS CloudFormation 模板。JSON格式化的範本也會儲存在目cdk.out錄中。

以下是 AWS CloudFormation 模板的示例輸出：

AWS CloudFormation 範本

```
Resources:
  HelloWorldFunctionServiceRoleunique-identifier:
    Type: AWS::IAM::Role
    Properties:
      AssumeRolePolicyDocument:
        Statement:
          - Action: sts:AssumeRole
            Effect: Allow
            Principal:
              Service: lambda.amazonaws.com
        Version: "2012-10-17"
      ManagedPolicyArns:
        - Fn::Join:
            - ""
            - - "arn:"
              - Ref: AWS::Partition
              - :iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/HelloWorldFunction/ServiceRole/Resource
  HelloWorldFunctionunique-identifier:
    Type: AWS::Lambda::Function
    Properties:
      Code:
        S3Bucket:
          Fn::Sub: cdk-unique-identifier-assets-${AWS::AccountId}-${AWS::Region}
```

```
        S3Key: unique-identifier.zip
      Handler: hello.handler
      Role:
        Fn::GetAtt:
          - HelloWorldFunctionServiceRoleunique-identifier
          - Arn
      Runtime: nodejs20.x
    DependsOn:
      - HelloWorldFunctionServiceRoleunique-identifier
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/HelloWorldFunction/Resource
      aws:asset:path: asset.unique-identifier
      aws:asset:is-bundled: false
      aws:asset:property: Code
  HelloWorldApiunique-identifier:
    Type: AWS::ApiGateway::RestApi
    Properties:
      Name: HelloWorldApi
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/HelloWorldApi/Resource
  HelloWorldApiDeploymentunique-identifier:
    Type: AWS::ApiGateway::Deployment
    Properties:
      Description: Automatically created by the RestApi construct
      RestApiId:
        Ref: HelloWorldApiunique-identifier
    DependsOn:
      - HelloWorldApihelloGETunique-identifier
      - HelloWorldApihellounique-identifier
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/HelloWorldApi/Deployment/Resource
  HelloWorldApiDeploymentStageprod012345ABC:
    Type: AWS::ApiGateway::Stage
    Properties:
      DeploymentId:
        Ref: HelloWorldApiDeploymentunique-identifier
      RestApiId:
        Ref: HelloWorldApiunique-identifier
      StageName: prod
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/HelloWorldApi/DeploymentStage.prod/Resource
  HelloWorldApihellounique-identifier:
    Type: AWS::ApiGateway::Resource
    Properties:
```

```
      ParentId:
        Fn::GetAtt:
          - HelloWorldApiunique-identifier
          - RootResourceId
      PathPart: hello
      RestApiId:
        Ref: HelloWorldApiunique-identifier
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/HelloWorldApi/Default/hello/Resource
  HelloWorldApihelloGETApiPermissionCdkHelloWorldStackHelloWorldApiunique-identifier:
    Type: AWS::Lambda::Permission
    Properties:
      Action: lambda:InvokeFunction
      FunctionName:
        Fn::GetAtt:
          - HelloWorldFunctionunique-identifier
          - Arn
      Principal: apigateway.amazonaws.com
      SourceArn:
        Fn::Join:
          - ""
          - - "arn:"
            - Ref: AWS::Partition
            - ":execute-api:"
            - Ref: AWS::Region
            - ":"
            - Ref: AWS::AccountId
            - ":"
            - Ref: HelloWorldApi9E278160
            - /
            - Ref: HelloWorldApiDeploymentStageprodunique-identifier
            - /GET/hello
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/HelloWorldApi/Default/hello/GET/
ApiPermission.CdkHelloWorldStackHelloWorldApiunique-identifier.GET..hello
  HelloWorldApihelloGETApiPermissionTestCdkHelloWorldStackHelloWorldApiunique-
identifier:
    Type: AWS::Lambda::Permission
    Properties:
      Action: lambda:InvokeFunction
      FunctionName:
        Fn::GetAtt:
          - HelloWorldFunctionunique-identifier
          - Arn
```

```
      Principal: apigateway.amazonaws.com
      SourceArn:
        Fn::Join:
          - ""
          - - "arn:"
            - Ref: AWS::Partition
            - ":execute-api:"
            - Ref: AWS::Region
            - ":"
            - Ref: AWS::AccountId
            - ":"
            - Ref: HelloWorldApiunique-identifier
            - /test-invoke-stage/GET/hello
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/HelloWorldApi/Default/hello/GET/
ApiPermission.Test.CdkHelloWorldStackHelloWorldApiunique-identifier.GET..hello
  HelloWorldApihelloGETunique-identifier:
    Type: AWS::ApiGateway::Method
    Properties:
      AuthorizationType: NONE
      HttpMethod: GET
      Integration:
        IntegrationHttpMethod: POST
        Type: AWS_PROXY
        Uri:
          Fn::Join:
            - ""
            - - "arn:"
              - Ref: AWS::Partition
              - ":apigateway:"
              - Ref: AWS::Region
              - :lambda:path/2015-03-31/functions/
              - Fn::GetAtt:
                  - HelloWorldFunctionunique-identifier
                  - Arn
              - /invocations
      ResourceId:
        Ref: HelloWorldApihellounique-identifier
      RestApiId:
        Ref: HelloWorldApiunique-identifier
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/HelloWorldApi/Default/hello/GET/Resource
  CDKMetadata:
    Type: AWS::CDK::Metadata
```

```
    Properties:
      Analytics: v2:deflate64:unique-identifier
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/CDKMetadata/Default
    Condition: CDKMetadataAvailable
Outputs:
  HelloWorldApiEndpointunique-identifier:
    Value:
      Fn::Join:
        - ""
        - - https://
          - Ref: HelloWorldApiunique-identifier
          - .execute-api.
          - Ref: AWS::Region
          - "."
          - Ref: AWS::URLSuffix
          - /
          - Ref: HelloWorldApiDeploymentStageprodunique-identifier
          - /
Conditions:
  CDKMetadataAvailable:
    Fn::Or:
      - Fn::Or:
          - Fn::Equals:
              - Ref: AWS::Region
              - af-south-1
          - Fn::Equals:
              - Ref: AWS::Region
              - ap-east-1
          - Fn::Equals:
              - Ref: AWS::Region
              - ap-northeast-1
          - Fn::Equals:
              - Ref: AWS::Region
              - ap-northeast-2
          - Fn::Equals:
              - Ref: AWS::Region
              - ap-south-1
          - Fn::Equals:
              - Ref: AWS::Region
              - ap-southeast-1
          - Fn::Equals:
              - Ref: AWS::Region
              - ap-southeast-2
```

```
          - Fn::Equals:
              - Ref: AWS::Region
              - ca-central-1
          - Fn::Equals:
              - Ref: AWS::Region
              - cn-north-1
          - Fn::Equals:
              - Ref: AWS::Region
              - cn-northwest-1
      - Fn::Or:
          - Fn::Equals:
              - Ref: AWS::Region
              - eu-central-1
          - Fn::Equals:
              - Ref: AWS::Region
              - eu-north-1
          - Fn::Equals:
              - Ref: AWS::Region
              - eu-south-1
          - Fn::Equals:
              - Ref: AWS::Region
              - eu-west-1
          - Fn::Equals:
              - Ref: AWS::Region
              - eu-west-2
          - Fn::Equals:
              - Ref: AWS::Region
              - eu-west-3
          - Fn::Equals:
              - Ref: AWS::Region
              - il-central-1
          - Fn::Equals:
              - Ref: AWS::Region
              - me-central-1
          - Fn::Equals:
              - Ref: AWS::Region
              - me-south-1
          - Fn::Equals:
              - Ref: AWS::Region
              - sa-east-1
      - Fn::Or:
          - Fn::Equals:
              - Ref: AWS::Region
              - us-east-1
```

```
          - Fn::Equals:
              - Ref: AWS::Region
              - us-east-2
          - Fn::Equals:
              - Ref: AWS::Region
              - us-west-1
          - Fn::Equals:
              - Ref: AWS::Region
              - us-west-2
Parameters:
  BootstrapVersion:
    Type: AWS::SSM::Parameter::Value<String>
    Default: /cdk-bootstrap/hnb659fds/version
    Description: Version of the CDK Bootstrap resources in this environment,
 automatically retrieved from SSM Parameter Store. [cdk:skip]
Rules:
  CheckBootstrapVersion:
    Assertions:
      - Assert:
          Fn::Not:
            - Fn::Contains:
                - - "1"
                  - "2"
                  - "3"
                  - "4"
                  - "5"
                - Ref: BootstrapVersion
        AssertDescription: CDK bootstrap stack version 6 required. Please run 'cdk
 bootstrap' with a recent version of the CDK CLI.
```

透過使用 L2 建構，您可以定義一些屬性來設定資源，並使用輔助程式方法將它們整合在一起。 AWS CDK 設定佈建應用程式所需的大部分 AWS CloudFormation 資源和屬性。

步驟 5：部署應用程式

在此步驟中，您可以使用 AWS CDK CLIcdk deploy命令來部署應用程式。該服務 AWS CDK 與 AWS CloudFormation 服務一起提供您的資源。

> **Important**
>
> 在部署之前，您必須執行 AWS 環境的一次性啟動載入。如需指示，請參閱啟動您的環境。

從項目的根目錄，運行以下命令。出現提示時確認變更：

```
$ cdk deploy

#  Synthesis time: 2.44s

...

Do you wish to deploy these changes (y/n)? y
```

部署完成後， AWS CDK CLI將輸出您的端點 URL。複製此 URL 以進行下一步。以下是範例：

```
...
#  HelloWorldStack

#  Deployment time: 45.37s

Outputs:
HelloWorldStack.HelloWorldApiEndpointunique-identifier = https://<api-id>.execute-
api.<region>.amazonaws.com/prod/
Stack ARN:
arn:aws:cloudformation:region:account-id:stack/HelloWorldStack/unique-identifier
...
```

步驟 6：與您的應用程式互動

在此步驟中，您會向 API 端點啟動 GET 要求，並接收您的 Lambda 函數回應。

從上一個步驟中找到端點 URL，然後新增/hello路徑。然後，使用瀏覽器或命令提示符，向端點發送 GET 請求。以下是範例：

```
$ curl https://<api-id>.execute-api.<region>.amazonaws.com/prod/hello
{"message":"Hello World!"}%
```

恭喜您，您已成功建立、部署您的應用程式，並使用 AWS CDK!

步驟 7：刪除您的應用程式

在此步驟中，您可以使 AWS CDK CLI用從中刪除您的應用程式 AWS 雲端。

若要刪除您的應用程式，請執行cdk destroy。出現提示時，請確認刪除應用程式的要求：

```
$ cdk destroy
Are you sure you want to delete: CdkHelloWorldStack (y/n)? y
CdkHelloWorldStack: destroying... [1/1]
...
 #  CdkHelloWorldStack: destroyed
```

故障診斷

錯誤:{「訊息」:「內部伺服器錯誤」}%

呼叫已部署的 Lambda 函數時，您會收到此錯誤訊息。發生此錯誤的原因有多種。

進一步疑難排解

使用 AWS CLI 來叫用您的 Lambda 函數。

1. 修改堆疊檔案以擷取已部署 Lambda 函數名稱的輸出值。以下是範例：

```
...

class CdkHelloWorldStack extends Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    // Define the Lambda function resource
    // ...

    new CfnOutput(this, 'HelloWorldFunctionName', {
      value: helloWorldFunction.functionName,
      description: 'JavaScript Lambda function'
    });

    // Define the API Gateway resource
    // ...
```

2. 再次部署您的應用程式。 AWS CDK CLI將輸出您部署的 Lambda 函數名稱的值：

```
$ cdk deploy

#  Synthesis time: 0.29s
...
 #  CdkHelloWorldStack
```

```
#  Deployment time: 20.36s

Outputs:
...
CdkHelloWorldStack.HelloWorldFunctionName = CdkHelloWorldStack-
HelloWorldFunctionunique-identifier
...
```

3. 使 AWS CLI 用在中叫用 Lambda 函數， AWS 雲端 並將回應輸出至文字檔案：

```
$ aws lambda invoke --function-name CdkHelloWorldStack-HelloWorldFunctionunique-
identifier output.txt
```

4. 檢查output.txt以查看結果。

可能的原因：API Gateway 資源在堆疊檔案中定義不正確。

如果output.txt顯示成功的 Lambda 函數回應，則問題可能出在於您如何定義 API Gateway REST API。會直接 AWS CLI 呼叫您的 Lambda，而不是透過您的端點呼叫。檢查您的代碼，以確保它與本教程匹配。然後，再次部署。

可能的原因：堆疊檔案中的 Lambda 資源定義不正確。

如果output.txt返回錯誤，則問題可能出在於您如何定義 Lambda 函數。檢查您的代碼，以確保它與本教程匹配。然後再次部署。

建立含有多個堆疊的應用程式

您可以建立包含多個堆疊的 AWS Cloud Development Kit (AWS CDK) 應用程式。當您部署 AWS CDK 應用程式時，每個堆疊都會變成其自己的 AWS CloudFormation 範本。您也可以使用 AWS CDK CLIcdk deploy命令個別合成和部署每個堆疊。

本教學課程涵蓋以下內容：

- 如何擴展Stack類以接受新的屬性或參數。
- 如何使用屬性來確定堆棧包含哪些資源及其配置。
- 如何從這個類實例化多個堆棧。

本主題中的範例使用名為 `encryptBucket` (Python:`encrypt_bucket`) 的布林屬性。它會指出是否應加密 Amazon S3 儲存貯體。如果是這樣，堆疊會使用由 AWS Key Management Service (AWS KMS) 管理的金鑰啟用加密。該應用程序創建此堆棧的兩個實例，一個帶有加密，一個沒有。

主題

開始之前

首先，安裝 Node.js 和命 AWS CDK 令行工具，如果你還沒有。如需詳細資訊，請參閱 開始使用 AWS CDK。

接下來，透過在指令行中輸入以下指令來建立 AWS CDK 專案。

TypeScript

```
mkdir multistack
cd multistack
cdk init --language=typescript
```

JavaScript

```
mkdir multistack
cd multistack
cdk init --language=javascript
```

Python

```
mkdir multistack
cd multistack
cdk init --language=python
source .venv/bin/activate
pip install -r requirements.txt
```

Java

```
mkdir multistack
cd multistack
cdk init --language=java
```

您可以將生成的 Maven 項目導入到您的 Java IDE 中。

C#

```
mkdir multistack
cd multistack
cdk init --language=csharp
```

您可以`src/Pipeline.sln`在視覺工作室中開啟檔案。

加入可選參數

`Stack`構造函`props`數的參數完成了接口`StackProps`。在此範例中，我們希望堆疊接受其他屬性，告訴我們是否要加密 Amazon S3 儲存貯體。我們應該創建一個包含該屬性的接口或類。這允許編譯器確保屬性具有布林值，並在 IDE 中為其啟用自動完成功能。

因此，在 IDE 或編輯器中打開指示的源文件，然後添加新的接口，類或參數。更改後代碼應該如下所示。我們添加的行以粗體顯示。

TypeScript

檔案：`lib/multistack-stack.ts`

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';

interface MultiStackProps extends cdk.StackProps {
  encryptBucket?: boolean;
}

export class MultistackStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: MultiStackProps) {
    super(scope, id, props);

    // The code that defines your stack goes here
```

```
  }
}
```

JavaScript

檔案：lib/multistack-stack.js

JavaScript 沒有界面功能; 我們不需要添加任何代碼。

```
const cdk = require('aws-cdk-stack');

class MultistackStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    // The code that defines your stack goes here
  }
}

module.exports = { MultistackStack }
```

Python

檔案：multistack/multistack_stack.py

Python 沒有接口功能，因此我們將擴展堆棧以通過添加關鍵字參數來接受新的屬性。

```
import aws_cdk as cdk
from constructs import Construct

class MultistackStack(cdk.Stack):

    # The Stack class doesn't know about our encrypt_bucket parameter,
    # so accept it separately and pass along any other keyword arguments.
    def __init__(self, scope: Construct, id: str, *, encrypt_bucket=False,
                 **kwargs) -> None:
        super().__init__(scope, id, **kwargs)

        # The code that defines your stack goes here
```

Java

檔案：src/main/java/com/myorg/MultistackStack.java

這比我們真正想要在 Java 中擴展道具類型更複雜。相反，編寫堆棧的構造函數以接受可選的布爾參數。因為`props`是一個可選的參數，我們將編寫一個額外的構造函數，讓你跳過它。它將默認為`false`。

```
package com.myorg;

import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.constructs.Construct;

import software.amazon.awscdk.services.s3.Bucket;

public class MultistackStack extends Stack {
    // additional constructors to allow props and/or encryptBucket to be omitted
    public MultistackStack(final Construct scope, final String id, boolean
 encryptBucket) {
        this(scope, id, null, encryptBucket);
    }

    public MultistackStack(final Construct scope, final String id) {
        this(scope, id, null, false);
    }

    public MultistackStack(final Construct scope, final String id, final StackProps
 props,
            final boolean encryptBucket) {
        super(scope, id, props);

        // The code that defines your stack goes here
    }
}
```

C#

檔案：`src/Multistack/MultistackStack.cs`

```
using Amazon.CDK;
using constructs;

namespace Multistack
{
```

```
    public class MultiStackProps : StackProps
    {
        public bool? EncryptBucket { get; set; }
    }


    public class MultistackStack : Stack
    {
        public MultistackStack(Construct scope, string id, MultiStackProps props) :
 base(scope, id, props)
        {
            // The code that defines your stack goes here
        }
    }
}
```

新屬性是可選的。如果 encryptBucket (Python:encrypt_bucket) 不存在，它的值是undefined，或者是本地等價物。儲存貯體預設為未加密。

定義堆棧類

現在，讓我們定義我們的堆棧類，使用我們的新屬性。使代碼如下所示。您需要新增或變更的程式碼會以粗體顯示。

TypeScript

檔案：lib/multistack-stack.ts

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from constructs;
import * as s3 from 'aws-cdk-lib/aws-s3';

interface MultistackProps extends cdk.StackProps {
  encryptBucket?: boolean;
}

export class MultistackStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: MultistackProps) {
    super(scope, id, props);

    // Add a Boolean property "encryptBucket" to the stack constructor.
    // If true, creates an encrypted bucket. Otherwise, the bucket is unencrypted.
```

```
    // Encrypted bucket uses KMS-managed keys (SSE-KMS).
    if (props && props.encryptBucket) {
      new s3.Bucket(this, "MyGroovyBucket", {
        encryption: s3.BucketEncryption.KMS_MANAGED,
        removalPolicy: cdk.RemovalPolicy.DESTROY
      });
    } else {
      new s3.Bucket(this, "MyGroovyBucket", {
        removalPolicy: cdk.RemovalPolicy.DESTROY});
    }
  }
}
```

JavaScript

檔案：lib/multistack-stack.js

```
const cdk = require('aws-cdk-lib');
const s3 = require('aws-cdk-lib/aws-s3');

class MultistackStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    // Add a Boolean property "encryptBucket" to the stack constructor.
    // If true, creates an encrypted bucket. Otherwise, the bucket is unencrypted.
    // Encrypted bucket uses KMS-managed keys (SSE-KMS).
    if ( props && props.encryptBucket) {
      new s3.Bucket(this, "MyGroovyBucket", {
        encryption: s3.BucketEncryption.KMS_MANAGED,
        removalPolicy: cdk.RemovalPolicy.DESTROY
      });
    } else {
      new s3.Bucket(this, "MyGroovyBucket", {
        removalPolicy: cdk.RemovalPolicy.DESTROY});
    }
  }
}

module.exports = { MultistackStack }
```

Python

檔案：multistack/multistack_stack.py

```
import aws_cdk as cdk
from constructs import Construct
from aws_cdk import aws_s3 as s3

class MultistackStack(cdk.Stack):

    # The Stack class doesn't know about our encrypt_bucket parameter,
    # so accept it separately and pass along any other keyword arguments.
    def __init__(self, scope: Construct, id: str, *, encrypt_bucket=False,
                 **kwargs) -> None:
        super().__init__(scope, id, **kwargs)

        # Add a Boolean property "encryptBucket" to the stack constructor.
        # If true, creates an encrypted bucket. Otherwise, the bucket is
 unencrypted.
        # Encrypted bucket uses KMS-managed keys (SSE-KMS).
        if encrypt_bucket:
            s3.Bucket(self, "MyGroovyBucket",
                      encryption=s3.BucketEncryption.KMS_MANAGED,
                      removal_policy=cdk.RemovalPolicy.DESTROY)
        else:
            s3.Bucket(self, "MyGroovyBucket",
                     removal_policy=cdk.RemovalPolicy.DESTROY)
```

Java

檔案：src/main/java/com/myorg/MultistackStack.java

```
package com.myorg;

import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.constructs.Construct;
import software.amazon.awscdk.RemovalPolicy;

import software.amazon.awscdk.services.s3.Bucket;
import software.amazon.awscdk.services.s3.BucketEncryption;

public class MultistackStack extends Stack {
    // additional constructors to allow props and/or encryptBucket to be omitted
    public MultistackStack(final Construct scope, final String id,
            boolean encryptBucket) {
        this(scope, id, null, encryptBucket);
```

```
    }

    public MultistackStack(final Construct scope, final String id) {
        this(scope, id, null, false);
    }

    // main constructor
    public MultistackStack(final Construct scope, final String id,
            final StackProps props, final boolean encryptBucket) {
        super(scope, id, props);

        // Add a Boolean property "encryptBucket" to the stack constructor.
        // If true, creates an encrypted bucket. Otherwise, the bucket is
        // unencrypted. Encrypted bucket uses KMS-managed keys (SSE-KMS).
        if (encryptBucket) {
            Bucket.Builder.create(this, "MyGroovyBucket")
                    .encryption(BucketEncryption.KMS_MANAGED)
                    .removalPolicy(RemovalPolicy.DESTROY).build();
        } else {
            Bucket.Builder.create(this, "MyGroovyBucket")
                    .removalPolicy(RemovalPolicy.DESTROY).build();
        }
    }
}
```

C#

檔案：src/Multistack/MultistackStack.cs

```
using Amazon.CDK;
using Amazon.CDK.AWS.S3;

namespace Multistack
{

    public class MultiStackProps : StackProps
    {
        public bool? EncryptBucket { get; set; }
    }

    public class MultistackStack : Stack
    {
        public MultistackStack(Construct scope, string id, IMultiStackProps props =
 null) : base(scope, id, props)
```

```
        {
            // Add a Boolean property "EncryptBucket" to the stack constructor.
            // If true, creates an encrypted bucket. Otherwise, the bucket is
 unencrypted.
            // Encrypted bucket uses KMS-managed keys (SSE-KMS).
            if (props?.EncryptBucket ?? false)
            {
                new Bucket(this, "MyGroovyBucket", new BucketProps
                {
                    Encryption = BucketEncryption.KMS_MANAGED,
                    RemovalPolicy = RemovalPolicy.DESTROY
                });
            }
            else
            {
                new Bucket(this, "MyGroovyBucket", new BucketProps
                {
                    RemovalPolicy = RemovalPolicy.DESTROY
                });
            }
        }
    }
}
```

創建兩個堆棧實例

現在，我們將添加代碼以實例化兩個單獨的堆棧。和以前一樣，以粗體顯示的代碼行是您需要添加的代碼行。刪除現有的MultistackStack定義。

TypeScript

檔案：bin/multistack.ts

```
#!/usr/bin/env node
import 'source-map-support/register';
import * as cdk from 'aws-cdk-lib';
import { MultistackStack } from '../lib/multistack-stack';

const app = new cdk.App();

new MultistackStack(app, "MyWestCdkStack", {
    env: {region: "us-west-1"},
```

```
    encryptBucket: false
});

new MultistackStack(app, "MyEastCdkStack", {
    env: {region: "us-east-1"},
    encryptBucket: true
});

app.synth();
```

JavaScript

檔案：bin/multistack.js

```
#!/usr/bin/env node
const cdk = require('aws-cdk-lib');
const { MultistackStack } = require('../lib/multistack-stack');

const app = new cdk.App();

new MultistackStack(app, "MyWestCdkStack", {
    env: {region: "us-west-1"},
    encryptBucket: false
});

new MultistackStack(app, "MyEastCdkStack", {
    env: {region: "us-east-1"},
    encryptBucket: true
});

app.synth();
```

Python

檔案：./app.py

```
#!/usr/bin/env python3

import aws_cdk as cdk

from multistack.multistack_stack import MultistackStack

app = cdk.App()
```

```
MultistackStack(app, "MyWestCdkStack",
                env=cdk.Environment(region="us-west-1"),
                encrypt_bucket=False)

MultistackStack(app, "MyEastCdkStack",
                env=cdk.Environment(region="us-east-1"),
                encrypt_bucket=True)

app.synth()
```

Java

檔案：src/main/java/com/myorg/MultistackApp.java

```
package com.myorg;

import software.amazon.awscdk.App;
import software.amazon.awscdk.Environment;
import software.amazon.awscdk.StackProps;

public class MultistackApp {
    public static void main(final String argv[]) {
        App app = new App();

        new MultistackStack(app, "MyWestCdkStack", StackProps.builder()
                .env(Environment.builder()
                        .region("us-west-1")
                        .build())
                .build(), false);

        new MultistackStack(app, "MyEastCdkStack", StackProps.builder()
                .env(Environment.builder()
                        .region("us-east-1")
                        .build())
                .build(), true);

        app.synth();
    }
}
```

C#

檔案:src/Multistack/Program.cs

```
using Amazon.CDK;

namespace Multistack
{
    class Program
    {
        static void Main(string[] args)
        {
            var app = new App();

            new MultistackStack(app, "MyWestCdkStack", new MultiStackProps
            {
                Env = new Environment { Region = "us-west-1" },
                EncryptBucket = false
            });

            new MultistackStack(app, "MyEastCdkStack", new MultiStackProps
            {
                Env = new Environment { Region = "us-east-1" },
                EncryptBucket = true
            });

            app.Synth();
        }
    }
}
```

此程式碼會使用MultistackStack類別上的 new encryptBucket (Python:encrypt_bucket) 屬性來實體化下列項目：

- 在該us-east-1 AWS 區域中具有一個加密 Amazon S3 儲存貯體的一個堆疊。
- 在該us-west-1 AWS 區域中具有未加密 Amazon S3 儲存貯體的一個堆疊。

合成並部署堆疊

現在，您可以從應用程序部署堆棧。首先，合成一個 AWS CloudFormation 模板 MyEastCdkStack-堆棧中us-east-1。這是含有加密 S3 儲存貯體的堆疊。

```
$ cdk synth MyEastCdkStack
```

若要將此堆疊部署到您的 AWS 帳戶，請發出下列其中一個指令。第一個命令使用您的預設設 AWS 定檔取得認證來部署堆疊。第二個使用您指定的縱斷面。對於 *PROFILE_NAME*，請取代包含部署至「區域」之適當證明資料的 AWS CLI 設定檔名稱。us-east-1 AWS

```
cdk deploy MyEastCdkStack
```

```
cdk deploy MyEastCdkStack --profile=PROFILE_NAME
```

清除

若要避免您部署的資源收費，請使用下列命令銷毀堆疊。

```
cdk destroy MyEastCdkStack
```

如果堆棧的存儲桶中存儲了任何內容，則銷毀操作失敗。如果您只按照本主題中的說明進行操作，則不應該存在。但是，如果您確實在存儲桶中放入了某些內容，則必須在銷毀堆棧之前刪除存儲桶內容。(請勿刪除值區本身。) 使用 AWS Management Console 或刪 AWS CLI 除值區內容。

範例

本主題包含下列範例：

- 建立無伺服器的 Hello World 應用程式使用 Lambda、API Gateway 和 Amazon S3 建立無伺服器應用程式。
- 使用建立 AWS Fargate 服務 AWS CDK從上的圖像創建一個 Amazon ECS Fargate 服務。DockerHub

使用建立 AWS Fargate 服務 AWS CDK

本範例將引導您如何從 Amazon ECR 上的映像建立在 Amazon Elastic Container Service (Amazon ECS) 叢集上執行的 AWS Fargate 服務，該叢集由網際網路對向 Application Load Balancer 器前方執行的 Fargate 服務。

Amazon ECS 是具高可擴展性且快速的容器管理服務，可以在叢集上輕鬆執行、停用及管理 Docker 容器。您可以透過使用 Fargate 啟動類型啟動服務或任務，在由 Amazon ECS 管理的無伺服器基礎設施上託管叢集。如需更多控制權，您可以在使用 Amazon EC2 啟動類型管理的 Elastic Compute Cloud (Amazon EC2) 執行個體叢集上託管任務。

本教學課程說明如何使用 Fargate 啟動類型來啟動某些服務。如果您已使 AWS Management Console 用建立 Fargate 服務，您知道完成該工作需要遵循許多步驟。 AWS 有數個自學課程和文件主題，可引導您逐步建立 Fargate 服務，包括：

- 如何部署碼頭容器- AWS
- 使用 Amazon ECS 進行設置
- 使用 Fargate 開始使用 Amazon ECS

此範例會在 AWS CDK 程式碼中建立類似的 Fargate 服務。

本教學中使用的 Amazon ECS 建構可提供下列優點，協助您使用 AWS 服務：

- 自動設定負載平衡器。
- 自動為負載平衡器開啟安全群組。這可讓負載平衡器與執行個體通訊，而無需您明確建立安全群組。
- 自動排序服務與連接至目標群組的負載平衡器之間的相依性，在建立 AWS CDK 執行個體之前，會強制執行建立接聽程式的正確順序。

- 自動設定自動調整群組的使用者資料。這會建立正確的組態，將叢集與 AMI 產生關聯。
- 提早驗證參數組合。這會先前公開 AWS CloudFormation 問題，因此可節省您的部署時間。例如，根據任務的不同，很容易錯誤配置內存設置。以前，在部署應用程序之前，您不會遇到錯誤。但是現在，當您合成應用程序時， AWS CDK 可以檢測到配置錯誤並發出錯誤。
- 如果您使用來自 Amazon ECR 的映像，則會自動為 Amazon Elastic Container Registry (Amazon ECR) 添加許可。
- 自動縮放。該方法提 AWS CDK 供了一種方法，以便您可以在使用 Amazon EC2 叢集時自動擴展執行個體。當您在 Fargate 叢集中使用執行個體時，會自動發生這種情況。

  此外，當自動調度資源設定嘗試終 AWS CDK 止執行個體，但工作正在執行或排定在該執行個體上時，也可防止執行個體遭到刪除。

  之前，您必須建立 Lambda 函數才能擁有此功能。
- 提供資產支援，讓您只需一個步驟即可將來源從機器部署到 Amazon ECS。之前，若要使用應用程式來源，您必須執行數個手動步驟，例如上傳到 Amazon ECR 和建立 Docker 映像。

如需詳細資訊，請參閱 ECS。

> **Important**
>
> 我們將使用的`ApplicationLoadBalancedFargateService`結構包含許多 AWS 組件，其中一些組件如果保留在您的 AWS 帳戶中佈建，即使您不使用它們，也會產生非平凡的成本。完成此範例後，請務必清理 (cdk destroy)。

建立目錄並初始化 AWS CDK

讓我們先創建一個目錄來保存 AWS CDK 代碼，然後在該目錄中創建一個 AWS CDK 應用程序。

TypeScript

```
mkdir MyEcsConstruct
cd MyEcsConstruct
cdk init --language typescript
```

JavaScript

```
mkdir MyEcsConstruct
```

```
cd MyEcsConstruct
cdk init --language javascript
```

Python

```
mkdir MyEcsConstruct
cd MyEcsConstruct
cdk init --language python
source .venv/bin/activate
pip install -r requirements.txt
```

Java

```
mkdir MyEcsConstruct
cd MyEcsConstruct
cdk init --language java
```

您現在可以將 Maven 項目導入到 IDE 中。

C#

```
mkdir MyEcsConstruct
cd MyEcsConstruct
cdk init --language csharp
```

您現在可以在視覺工作室`src/MyEcsConstruct.sln`中打開。

運行應用程序並確認它創建了一個空堆棧。

```
cdk synth
```

建立 Fargate 服務

使用 Amazon ECS 執行容器任務有兩種不同的方式：

- 使用`Fargate`啟動類型，Amazon ECS 會為您管理執行容器的實體機器。
- 使用`EC2`啟動類型，您可以在其中進行管理，例如指定自動調整比例。

在此範例中，我們將建立在面向網際網路的 Application Load Balancer 前方的 ECS 叢集上執行的 Fargate 服務。

將以下「 AWS 建構程式庫」模組匯入新增至指定的檔案。

TypeScript

檔案：lib/my_ecs_construct-stack.ts

```
import * as ec2 from "aws-cdk-lib/aws-ec2";
import * as ecs from "aws-cdk-lib/aws-ecs";
import * as ecs_patterns from "aws-cdk-lib/aws-ecs-patterns";
```

JavaScript

檔案：lib/my_ecs_construct-stack.js

```
const ec2 = require("aws-cdk-lib/aws-ec2");
const ecs = require("aws-cdk-lib/aws-ecs");
const ecs_patterns = require("aws-cdk-lib/aws-ecs-patterns");
```

Python

檔案：my_ecs_construct/my_ecs_construct_stack.py

```
from aws_cdk import (aws_ec2 as ec2, aws_ecs as ecs,
                     aws_ecs_patterns as ecs_patterns)
```

Java

檔案：src/main/java/com/myorg/MyEcsConstructStack.java

```
import software.amazon.awscdk.services.ec2.*;
import software.amazon.awscdk.services.ecs.*;
import software.amazon.awscdk.services.ecs.patterns.*;
```

C#

檔案：src/MyEcsConstruct/MyEcsConstructStack.cs

```
using Amazon.CDK.AWS.EC2;
using Amazon.CDK.AWS.ECS;
using Amazon.CDK.AWS.ECS.Patterns;
```

將建構函式結尾的註解取代為下列程式碼。

TypeScript

```
    const vpc = new ec2.Vpc(this, "MyVpc", {
      maxAzs: 3 // Default is all AZs in region
    });

    const cluster = new ecs.Cluster(this, "MyCluster", {
      vpc: vpc
    });

    // Create a load-balanced Fargate service and make it public
    new ecs_patterns.ApplicationLoadBalancedFargateService(this, "MyFargateService",
 {
      cluster: cluster, // Required
      cpu: 512, // Default is 256
      desiredCount: 6, // Default is 1
      taskImageOptions: { image: ecs.ContainerImage.fromRegistry("amazon/amazon-ecs-
sample") },
      memoryLimitMiB: 2048, // Default is 512
      publicLoadBalancer: true // Default is true
    });
```

JavaScript

```
    const vpc = new ec2.Vpc(this, "MyVpc", {
      maxAzs: 3 // Default is all AZs in region
    });

    const cluster = new ecs.Cluster(this, "MyCluster", {
      vpc: vpc
    });

    // Create a load-balanced Fargate service and make it public
    new ecs_patterns.ApplicationLoadBalancedFargateService(this, "MyFargateService",
 {
      cluster: cluster, // Required
      cpu: 512, // Default is 256
      desiredCount: 6, // Default is 1
      taskImageOptions: { image: ecs.ContainerImage.fromRegistry("amazon/amazon-ecs-
sample") },
      memoryLimitMiB: 2048, // Default is 512
      publicLoadBalancer: true // Default is true
    });
```

Python

```
        vpc = ec2.Vpc(self, "MyVpc", max_azs=3)     # default is all AZs in region

        cluster = ecs.Cluster(self, "MyCluster", vpc=vpc)

        ecs_patterns.ApplicationLoadBalancedFargateService(self, "MyFargateService",
            cluster=cluster,            # Required
            cpu=512,                    # Default is 256
            desired_count=6,            # Default is 1
            task_image_options=ecs_patterns.ApplicationLoadBalancedTaskImageOptions(
                image=ecs.ContainerImage.from_registry("amazon/amazon-ecs-sample")),
            memory_limit_mib=2048,      # Default is 512
            public_load_balancer=True)  # Default is True
```

Java

```
        Vpc vpc = Vpc.Builder.create(this, "MyVpc")
                             .maxAzs(3)  // Default is all AZs in region
                             .build();

        Cluster cluster = Cluster.Builder.create(this, "MyCluster")
                             .vpc(vpc).build();

        // Create a load-balanced Fargate service and make it public
        ApplicationLoadBalancedFargateService.Builder.create(this,
 "MyFargateService")
                        .cluster(cluster)           // Required
                        .cpu(512)                   // Default is 256
                         .desiredCount(6)            // Default is 1
                         .taskImageOptions(
                                 ApplicationLoadBalancedTaskImageOptions.builder()
                                         .image(ContainerImage.fromRegistry("amazon/
amazon-ecs-sample"))
                                         .build())
                         .memoryLimitMiB(2048)       // Default is 512
                         .publicLoadBalancer(true)   // Default is true
                         .build();
```

C#

```
            var vpc = new Vpc(this, "MyVpc", new VpcProps
            {
```

```
                MaxAzs = 3 // Default is all AZs in region
            });

            var cluster = new Cluster(this, "MyCluster", new ClusterProps
            {
                Vpc = vpc
            });

            // Create a load-balanced Fargate service and make it public
            new ApplicationLoadBalancedFargateService(this, "MyFargateService",
                new ApplicationLoadBalancedFargateServiceProps
                {
                    Cluster = cluster,          // Required
                    DesiredCount = 6,           // Default is 1
                    TaskImageOptions = new ApplicationLoadBalancedTaskImageOptions
                    {
                        Image = ContainerImage.FromRegistry("amazon/amazon-ecs-
sample")
                    },
                    MemoryLimitMiB = 2048,      // Default is 256
                    PublicLoadBalancer = true   // Default is true
                }
            );
```

保存它並確保它運行並創建一個堆棧。

```
cdk synth
```

堆棧是數百行，所以我們不會在這裡顯示它。堆疊應包含一個預設執行個體、一個私有子網路和三個可用區域的公用子網路，以及一個安全群組。

部署堆疊。

```
cdk deploy
```

AWS CloudFormation 顯示部署應用程式時所採取的數十個步驟的相關資訊。

這就是創建一個 Fargate 端驅動的 Amazon ECS 服務來運行 Docker 映像是多麼容易。

清除

為避免意外 AWS 收費，請在完成此練習後銷毀 AWS CDK 堆疊。

```
cdk destroy
```

AWS CDK 例子

如需您最愛支援的程式設計語言的 AWS CDK 堆疊和應用程式的更多範例，請參閱上的AWS CDK 範例存放庫 GitHub。

AWS CDK 工具

本節包含下列 AWS CDK 工具的相關資訊。

主題

AWS CDK 工具包（**cdk**命令）

CLI 命令cdk工具 AWS CDK 包是與您的 AWS CDK 應用程序進行交互的主要工具。它會執行您的應用程式、詢問您定義的應用程式模型，以及產生和部 AWS CloudFormation 署由. AWS CDK它還提供了其他有用於創建和使用 AWS CDK 項目的功能。本主題包含 CDK 工具組常見使用案例的相關資訊。

該 AWS CDK 工具包與節點 Package 管理器一起安裝。在大多數情況下，我們建議在全球範圍內安裝。

```
npm install -g aws-cdk             # install latest version
npm install -g aws-cdk@X.YY.Z      # install specific version
```

> **Tip**
>
> 如果您經常使用的多個版本 AWS CDK，請考慮在個別 CDK 專案中安裝相符版本的 AWS CDK Toolkit。若要執行這項操作，請省略`-g`npm install指令。然後用npx aws-cdk來調用它。如果存在本地版本，這將運行本地版本，如果不存在，則返回到全局版本。

工具包命令

所有 CDK 工具組命令都以開頭cdk，後面接著一個子指令 (list、synthesizedeploy、等等)。一些子命令有一個較短的版本（等等）lssynth，這是等價的。選項和引數會以任何順序跟隨子命令。這裡概述了可用的命令。

Command	函式
`cdk list` (`ls`)	列出應用程式中的堆疊
`cdk synthesize` (`synth`)	合成並打印一個或多個指定堆棧的 CloudFormation 模板
`cdk bootstrap`	部署 CDK 工具組暫存堆疊；請參閱 the section called “引導”
`cdk deploy`	部署一或多個指定的堆疊
`cdk destroy`	銷毀一個或多個指定的堆棧
`cdk diff`	將指定的堆疊及其相依性與已部署的堆疊或本機CloudFormation範本進行比較
`cdk import`	使用 CloudFormation 資源匯入將現有資源匯入 CDK 管理的堆疊
`cdk metadata`	顯示有關指定堆疊的中繼資料
`cdk init`	從指定的模板在當前目錄中創建一個新的 CDK 項目
`cdk context`	管理緩存的上下文值
`cdk docs` (`doc`)	在瀏覽器中開啟 CDK API 參考
`cdk doctor`	檢查您的 CDK 專案是否存在潛在問題

如需每個指令的可用選項，請參閱the section called “內建說明”。

指定選項及其值

指令行選項以兩個連字號 (`--`) 開頭。某些常用選項具有以單一連字號開頭的單一字母同義字 (例如，`--app`具有同`-a`義字)。 AWS CDK Toolkit 命令中的選項順序並不重要。

所有選項都接受一個值，該值必須在選項名稱之後。該值可以通過空格或等號與名稱分隔=。以下兩個選項是相同的。

```
--toolkit-stack-name MyBootstrapStack
--toolkit-stack-name=MyBootstrapStack
```

一些選項是標誌（布爾值）。您可以指定true或false作為它們的值。如果您不提供值，則會將該值視為true。您也可以在選項名稱前綴no-以暗示false。

```
# sets staging flag to true
--staging
--staging=true
--staging true

# sets staging flag to false
--no-staging
--staging=false
--staging false
```

可以多次指定一些選項 --context --parameters --plugin --tags--trust，即、、、和來指定多個值。這些被註明為在 CDK 工具包幫助中具有[array]類型。例如：

```
cdk bootstrap --tags costCenter=0123 --tags responsibleParty=jdoe
```

內建說明

該 AWS CDK 工具包集成了幫助。您可以通過發出以下命令來查看有關該實用程序的一般幫助以及提供的子命令列表：

```
cdk --help
```

例deploy如，若要查看特定子指令的說明，請在--help旗標前指定該指令。

```
cdk deploy --help
```

顯cdk version示 AWS CDK 工具包版本的問題。請求支援時提供此資訊。

版本報告

若要深入瞭解如何使 AWS CDK 用，系統會使用識別為AWS::CDK::Metadata的資源來收集和報告 AWS CDK 應用程式所使用的建構。此資源已新增至 AWS CloudFormation 範本，而且可以輕鬆檢

閱。此資訊也可用於使用具有已知安全性或可靠性問題的建構 AWS 來識別堆疊。它也可以用來聯繫他們的用戶與重要信息。

> **Note**
>
> 在版本 1.93.0 之前， AWS CDK 報告了在合成過程中加載的模塊的名稱和版本，而不是堆棧中使用的構造。

根據預設，會 AWS CDK 報告在堆疊中使用的下列 NPM 模組中建構的使用：

- AWS CDK 核心模組
- AWS 建構程式庫模組
- AWS 解決方案建構模組
- AWS 彩現農場部署套件模組

資`AWS::CDK::Metadata`源看起來像下面的內容。

```
CDKMetadata:
  Type: "AWS::CDK::Metadata"
  Properties:
    Analytics:
 "v2:deflate64:H4sIAND9SGAAAzXKSw5AMBAA0L1b2PdzBYnEAdio3RglglY60zQi7u6TWL/
XKmNUlxeQSOKwaPTBqrNhwEWU3hGHiCzK0dWWfAxoL/Fd8mvk+QkS/0X6BdjnCdgmOOQKWz
+AqqLDt2Y3YMnLYWwAAAA="
```

該`Analytics`屬性是堆棧中構造的 gzip，base64 編碼的前綴編碼列表。

若要選擇退出版本報告，請使用下列其中一種方法：

- 使用指cdk令搭配引--no-version-reporting數來選擇退出單一指令。

  ```
  cdk --no-version-reporting synth
  ```

  請記住， AWS CDK 工具包在部署之前會合成新的模板，因此您也應該添加`--no-version-reporting`到`cdk deploy`命令中。
- 在`./cdk.json`或中設定versionReporting為假`~/.cdk.json`。除非您透過在個別指令`--version-reporting`上指定來選擇加入，否則此選項會選擇退出。

```
{
  "app": "...",
  "versionReporting": false
}
```

使用驗證 AWS

您可以透過不同的方式設定 AWS 資源的程式設計存取，具體取決於環境和您可用的 AWS 存取權限。

要選擇您的身份驗證方法並為 CDK Toolkit 進行配置，請參閱 AWS SDK 和工具參考指南中的身份驗證和訪問。

對於在本地開發的新用戶，誰沒有給他們的雇主認證的方法，推薦的方法是設置 AWS IAM Identity Center。此方法包括安裝以便 AWS CLI 於設定，以及定期登入 AWS 存取入口網站。如果選擇此方法，則在 AWS SDK 和工具參考指南中完成 IAM 身分中心身份驗證的程序後，您的環境應包含以下元素：

- 您可以在 AWS CLI執行應用程式之前啟動 AWS 存取入口網站工作階段。
- 具有設定AWSconfig檔的共用檔案，其中包含可從中參考的一組組態值 AWS CDK。[default]若要尋找此檔案的位置，請參閱 AWS SDK 和工具參考指南中的共用檔案位置。
- 共用config檔案會region設定設定。這會設定 AWS 要求使用 AWS 區域 的預設值 AWS CDK 和 CDK 工具組。
- CDK Toolkit 會使用設定檔的 SSO 權杖提供者組態，在傳送要求之前取得認證。 AWS這個sso_role_name值是連接到 IAM 身分中心權限集的 IAM 角色，應該允許存取應用程式中 AWS 服務 使用的角色。

  下列範例config檔案顯示使用 SSO 權杖提供者組態設定的預設設定檔。設定檔的sso_session設定是指已命名的sso-session區段。此sso-session區段包含用來啟動 AWS 存取入口網站工作階段的設定。

```
[default]
sso_session = my-sso
sso_account_id = 111122223333
sso_role_name = SampleRole
region = us-east-1
output = json
```

```
[sso-session my-sso]
sso_region = us-east-1
sso_start_url = https://provided-domain.awsapps.com/start
sso_registration_scopes = sso:account:access
```

啟動 AWS 存取入口網站會話

在存取之前 AWS 服務，您需要 CDK Toolkit 的使用中 AWS 存取入口網站工作階段，才能使用 IAM 身分中心身分驗證來解析登入資料。根據您設定的工作階段長度，您的存取最終會過期，CDK Toolkit 會遇到驗證錯誤。在中執行下列命令 AWS CLI 以登入 AWS 存取入口網站。

```
aws sso login
```

如果您的 SSO 權杖提供者組態使用具名的設定檔而非預設設定檔，則命令為`aws sso login --profile NAME`。使用--profile選項或AWS_PROFILE環境變數發出指cdk令時，也請指定此設定檔。

若要測試您是否已有作用中的工作階段，請執行下列 AWS CLI 命令。

```
aws sts get-caller-identity
```

對此命令的回應，應報告共用 `config` 檔案中設定的 IAM Identity Center 帳戶和許可集合。

> **Note**
>
> 如果您已經擁有作用中的 AWS 存取入口網站工作階段並執行`aws sso login`，則不需要提供認證。
> 登入程序可能會提示您允許 AWS CLI 存取您的資料。由 AWS CLI 於建置在適用於 Python 的 SDK 之上，因此權限訊息可能包含`botocore`名稱的變體。

指定區域和其他組態

CDK 工具組需要知道您要部署的 AWS 區域以及如何進行驗證。 AWS這是部署作業和在合成期間擷取內容值所需的。您的帳戶和區域共同構成了環境。

區域可以使用環境變量或在配置文件中指定。這些與其他 AWS 工具 (例如和各種 AWS SDK) 所使用的變數 AWS CLI 和檔案相同。CDK 工具組會依下列順序尋找此資訊。

- 環AWS_DEFAULT_REGION境變數。
- 在標準 AWS config檔案中定義並使用指cdk令上的--profile選項指定的具名紀要。
- 標準 AWS config檔案的[default]區段。

除了在區[default]段中指定 AWS 驗證和區域之外，您還可以新增一或多個區[profile *NAME*]段，其中 *NAME* 是設定檔的名稱。有關命名配置文件的更多信息，請參閱 AWS SDK 和工具參考指南中的共享配置和憑據文件。

標準 AWS config檔案位於 ~/.aws/config (MacOS /Linux) 或 %USERPROFILE%\.aws\config (視窗)。有關詳細信息和替代位置，請參閱 AWS SDK 和工具參考指南中的共享配置和憑據文件的位置

您在應用 AWS CDK 程序中通過使用堆棧的env屬性指定的環境將在合成過程中使用。它是用來產生環境特定的 AWS CloudFormation 範本，而在部署期間，它會覆寫上述方法之一所指定的帳戶或區域。如需詳細資訊，請參閱 the section called “環境”。

> **Note**
>
> AWS CDK 使用與其他 AWS 工具和 SDK 相同來源檔案的認證，包括. AWS Command Line Interface 但是，其行為 AWS CDK 可能與這些工具有所不同。它使用引擎蓋 AWS SDK for JavaScript 下。如需設定認證的完整詳細資訊 AWS SDK for JavaScript，請參閱設定認證。

您可以選擇性地使用 --role-arn (或-r) 選項來指定應用於部署的 IAM 角色的 ARN。此角色必須由正在使用的 AWS 帳戶確定。

指定應用程式命令

CDK Toolkit 的許多功能都需要合成一個或多個 AWS CloudFormation 模板，這反過來又需要運行您的應用程序。 AWS CDK 支持以各種語言編寫的程序。因此，它使用配置選項來指定運行應用程序所需的確切命令。您可以透過兩種方式指定此選項。

首先，也是最常見的，它可以使用文件中的app密鑰來指定cdk.json。這是在你的 AWS CDK 項目的主目錄。使cdk init用建立新專案時，CDK 工具組會提供適當的指令。例如，這是cdk.json來自一個新 TypeScript 項目。

```
{
  "app": "npx ts-node bin/hello-cdk.ts"
```

```
}
```

當嘗試運行您的應用程序時，CDK 工具包在當前工作目錄cdk.json中查找。因此，您可能會在項目的主目錄中保持 shell 打開，以發出 CDK Toolkit 命令。

CDK 工具包還會在中查找應用程序密鑰~/.cdk.json（即在您的主目錄中），如果它無法在中./cdk.json找到它。如果您通常使用相同語言的 CDK 代碼，則在此添加應用程序命令會很有用。

如果您位於其他目錄中，或是要使用中指令以外的指令執行應用程式cdk.json，請使用 --app (或-a) 選項來指定它。

```
cdk --app "npx ts-node bin/hello-cdk.ts" ls
```

部署時，您也可以指定包含合成雲端組件的目錄，例如cdk.out，作為的--app值。指定的堆疊會從此目錄部署；應用程式不會合成。

指定堆疊

許多 CDK Toolkit 命令（例如cdk deploy）都可以在您的應用程序中定義的堆棧上工作。如果您的應用程序只包含一個堆棧，則 CDK Toolkit 假定您的意思是，如果您沒有明確指定堆棧。

否則，您必須指定要使用的一個或多個堆疊。您可以通過在命令行上單獨指定 ID 所需的堆棧來完成此操作。回想一下，ID 是當你實例化堆棧的第二個參數指定的值。

```
cdk synth PipelineStack LambdaStack
```

您也可以使用萬用字元來指定符合模式的 ID。

- ?符合任何單一字元
- *匹配任意數量的字符（*單獨匹配所有堆棧）
- **符合階層中的所有項目

您也可以使用此選--all項來指定所有堆疊。

如果您的應用程序使用 CDK Pipelines，則 CDK 工具包將您的堆棧和階段理解為層次結構。此外，--all選項和*萬用字元只會與頂層堆疊相符。要匹配所有堆棧，請使用**。也可**以用來指示特定層次結構下的所有堆疊。

使用萬用字元時，請將模式括在引號中，或使用逸出萬用字元。\如果不這樣做，您的 shell 可能會嘗試將模式擴展到當前目錄中的文件名。充其量，這不會做到你所期望的; 最壞的情況下，你可以部署你不打算的堆棧。這在 Windows 上並不是絕對必要的，因為cmd.exe不擴展通配符，但仍然是很好的做法。

```
cdk synth "*Stack"    # PipelineStack, LambdaStack, etc.
cdk synth 'Stack?'    # StackA, StackB, Stack1, etc.
cdk synth \*          # All stacks in the app, or all top-level stacks in a CDK
 Pipelines app
cdk synth '**'        # All stacks in a CDK Pipelines app
cdk synth 'PipelineStack/Prod/**'   # All stacks in Prod stage in a CDK Pipelines app
```

> **Note**
>
> 您指定堆疊的順序不一定是處理堆疊的順序。在決定處理堆疊的順序時， AWS CDK Toolkit 會考慮堆疊之間的相依性。例如，假設一個堆棧使用另一個堆棧生成的值（例如第二個堆棧中定義的資源的 ARN）。在這種情況下，由於這種依賴關係，第二個堆棧之前合成第一個堆棧。您可以使用堆棧的addDependency()方法手動添加堆棧之間的依賴關係。

引導您的環境 AWS

使用 CDK 部署堆疊需要特殊的專用 AWS CDK 資源才能佈建。該cdk bootstrap命令為您創建必要的資源。只有在部署需要這些專用資源的堆疊時，才需要啟動程序。如需詳細資訊，請參閱 the section called “引導”。

```
cdk bootstrap
```

如果沒有發布任何參數，如下所示，該cdk bootstrap命令將合成當前應用程序並啟動其堆棧將部署到的環境。如果應用程序包含與環境無關的堆棧（未明確指定環境），則默認帳戶和 Region 將被引導，或者使用指定的環境。--profile

在應用程式之外，您必須明確指定要啟動載入的環境。您也可以這樣做來引導未在您的應用程序或本地 AWS 配置文件中指定的環境。必須為指定的帳戶和區域配置認證（例如，在中~/.aws/credentials）。您可以指定包含所需憑證的設定檔。

```
cdk bootstrap ACCOUNT-NUMBER/REGION # e.g.
cdk bootstrap 1111111111/us-east-1
```

```
cdk bootstrap --profile test 1111111111/us-east-1
```

Important

您部署此類堆疊的每個環境 (帳戶/區域組合) 都必須個別啟動載入。

您可能會對啟動載入資源中的 AWS CDK 商店產生 AWS 費用。此外，如果您使用`-bootstrap-customer-key`，則會建立 AWS KMS 金鑰，這也會針對每個環境產生費用。

Note

舊版的啟動程序範本預設會建立 KMS 金鑰。為了避免收費，請重新引導使用`--no-bootstrap-customer-key`.

Note

CDK 工具包 v2 不支持原始引導模板，稱為傳統模板，默認情況下與 CDK v1 一起使用。

Important

現代啟動程序範本有效地授予`--trust`清單中任`--cloudformation-execution-policies`何 AWS 帳戶所隱含的權限。默認情況下，這將擴展讀取和寫入啟動載入帳戶中的任何資源的權限。確保使用您熟悉的策略和受信任帳戶配置啟動載入堆棧。

建立新的應用程式

要創建一個新的應用程序，為它創建一個目錄，然後，在目錄內，問題`cdk init`。

```
mkdir my-cdk-app
cd my-cdk-app
cdk init TEMPLATE --language LANGUAGE
```

支持的語言（*##*）是：

代碼	語言
typescript	TypeScript
javascript	JavaScript
python	Python
java	Java
csharp	C#

##是一個可選的模板。如果所需的模板是 app，默認情況下，您可以省略它。可用的範本包括：

範本	描述
app(預設值)	創建一個空的 AWS CDK 應用程序。
sample-app	使 AWS CDK 用包含 Amazon SQS 佇列和 Amazon SNS 主題的堆疊建立應用程式。

模板使用項目文件夾的名稱來生成新應用程序中的文件和類的名稱。

列出堆疊

若要查看應用程式中堆疊的 ID 清單， AWS CDK 請輸入下列其中一個對等命令：

```
cdk list
cdk ls
```

如果您的應用程式包含 CDK Pipelines 堆疊，CDK Toolkit 會根據其在管線階層中的位置，將堆疊名稱顯示為路徑。(例如，PipelineStackPipelineStack/Prod、和PipelineStack/Prod/MyService。)

如果您的應用程式包含許多堆疊，您可以指定要列出之堆疊的完整或部分堆疊 ID。如需詳細資訊，請參閱 the section called “指定堆疊”。

新增--long旗標以查看堆疊的詳細資訊，包括堆疊名稱及其環境 (AWS 帳戶和區域)。

合成堆疊

該cdk synthesize命令（幾乎總是縮寫synth）將應用程序中定義的堆棧合成為 CloudFormation 模板。

```
cdk synth         # if app contains only one stack
cdk synth MyStack
cdk synth Stack1 Stack2
cdk synth "*"     # all stacks in app
```

> **Note**
>
> CDK Toolkit 實際上運行您的應用程序，並在大多數操作之前（例如部署或比較堆棧時）合成新模板。依預設，這些範本會儲存在目cdk.out錄中。該cdk synth命令只是為一個或多個指定的堆棧打印生成的模板。

請參閱cdk synth --help以取得所有可用選項。下一節涵蓋了一些最常用的選項。

指定上下文值

使用--context或選-c項將執行階段內容值傳遞至您的 CDK 應用程式。

```
# specify a single context value
cdk synth --context key=value MyStack

# specify multiple context values (any number)
cdk synth --context key1=value1 --context key2=value2 MyStack
```

部署多個堆疊時，通常會將指定的內容值傳遞給所有堆疊。如果需要，您可以通過將堆棧名稱前綴為上下文值來為每個堆棧指定不同的值。

```
# different context values for each stack
cdk synth --context Stack1:key=value Stack2:key=value Stack1 Stack2
```

指定顯示格式

依預設，合成的範本會以 YAML 格式顯示。添加--json標誌以 JSON 格式顯示它。

```
cdk synth --json MyStack
```

指定輸出目錄

添加--output（ -o ）選項，將合成模板寫入目錄以外cdk.out的目錄。

```
cdk synth --output=~/templates
```

部署堆疊

cdk deploy子命令會將一個或多個指定的堆疊部署到您的 AWS 帳戶。

```
cdk deploy        # if app contains only one stack
cdk deploy MyStack
cdk deploy Stack1 Stack2
cdk deploy "*"    # all stacks in app
```

> **Note**
>
> CDK 工具包在部署任何內容之前運行您的應用程序並合成新 AWS CloudFormation 模板。因此，您可以搭配使用的大多數指令行選項 cdk synth (例如，--context) 也可以搭配使用cdk deploy。

請參閱cdk deploy --help以取得所有可用選項。下一節介紹了一些最有用的選項。

跳过合成

該cdk deploy命令通常會在部署之前合成應用程序的堆棧，以確保部署反映應用程序的最新版本。如果您知道自上次以來尚未變更程式碼cdk synth，則可以在部署時隱藏多餘的合成步驟。為此，請在選項中指定--app項cdk.out目的目錄。

```
cdk deploy --app cdk.out StackOne StackTwo
```

停用復原

AWS CloudFormation 具有回滾更改的能力，以便部署是原子的。這意味著他們要么成功或失敗作為一個整體. 會 AWS CDK 繼承此功能，因為它會合成並部署 AWS CloudFormation 範本。

回復可確保您的資源始終處於一致的狀態，這對於生產堆疊至關重要。不過，當您仍在開發基礎結構時，有些失敗是不可避免的，而復原失敗的部署可能會拖慢您的速度。

基於這個原因，CDK 工具組可讓您透過新增--no-rollback至您的cdk deploy命令來停用復原。使用此旗標時，不會復原失敗的部署。相反地，在失敗資源之前部署的資源會保留在原位，而下一次部署會從失敗的資源開始。您將花費更少的時間等待部署和更多的時間開發您的基礎架構。

熱插拔

使用--hotswap旗標與可cdk deploy嘗試直接更新您的 AWS 資源，而不是產生 AWS CloudFormation 變更集並部署它。如果無法進行熱交換， AWS CloudFormation 部署會退回部署。

目前熱交換支援 Lambda 函數、Step Functions 狀態機器和 Amazon ECS 容器映像。該--hotswap標誌還禁用回滾（即暗示--no-rollback）。

> **Important**
>
> 對於生產部署，不建議使用熱交換。

手錶模式

CDK Toolkit 的監視模式（cdk deploy --watch或cdk watch簡稱）持續監視 CDK 應用程序的源文件和資產以進行更改。當偵測到變更時，它會立即執行指定堆疊的部署。

根據預設，這些部署會使用--hotswap旗標來快速追蹤 Lambda 函數變更的部署。 AWS CloudFormation 如果您已變更基礎結構組態，它也會回到部署。若要cdk watch始終執行完整 AWS CloudFormation 部署，請將--no-hotswap旗標新增至cdk watch。

在執行部署時cdk watch所做的任何變更都會合併到單一部署中，並在進行中的部署完成後立即開始。

監視模式會使用專案中的"watch"金鑰cdk.json來決定要監視哪些檔案。根據預設，這些檔案是您的應用程式檔案和資產，但這可以透過修改"watch"金鑰中的"include"和"exclude"項目來變更。下列cdk.json檔案顯示這些項目的範例。

```
{
  "app": "mvn -e -q compile exec:java",
  "watch": {
    "include": "src/main/**",
    "exclude": "target/*"
  }
}
```

cdk watch從執行"build"命令cdk.json以在合成之前構建您的應用程序。如果您的部署需要任何命令來建置或封裝 Lambda 程式碼 (或其他不在您的 CDK 應用程式中)，請在此處新增。

Git 樣式萬用字元 (*和**) 都可以在和金鑰中使用。"watch" "build"每個路徑都會相對於的父目錄進行解譯cdk.json。的預設值include是**/*，表示專案根目錄中的所有檔案和目錄。 exclude是可選的。

> **Important**
>
> 不建議在生產部署中使用手錶模式。

指定 AWS CloudFormation 參數

「工 AWS CDK 具組」支援在部署時指定 AWS CloudFormation 參數。您可以在--parameters標誌後面的命令行上提供這些內容。

```
cdk deploy MyStack --parameters uploadBucketName=UploadBucket
```

若要定義多個參數，請使用多個--parameters旗標。

```
cdk deploy MyStack --parameters uploadBucketName=UpBucket --parameters
 downloadBucketName=DownBucket
```

如果您要部署多個堆疊，則可以為每個堆疊指定不同的參數值。若要這麼做，請在參數名稱前加上堆疊名稱和冒號。否則，相同的值將傳遞給所有堆棧。

```
cdk deploy MyStack YourStack --parameters MyStack:uploadBucketName=UploadBucket --
parameters YourStack:uploadBucketName=UpBucket
```

依預設，會 AWS CDK 保留先前部署的參數值，如果未明確指定，則會在稍後的部署中使用它們。使用--no-previous-parameters旗標可要求指定所有參數。

指定輸出檔案

如果你的堆棧聲明 AWS CloudFormation 輸出，這些通常在部署結束時顯示在屏幕上。若要將它們寫入 JSON 格式的檔案，請使用--outputs-file旗標。

```
cdk deploy --outputs-file outputs.json MyStack
```

安全性相關變更

為了保護您免受會影響安全性狀態的意外變更， AWS CDK Toolkit 會提示您核准與安全性相關的變更，然後再部署這些變更。您可以指定需要核准的變更層次：

```
cdk deploy --require-approval LEVEL
```

LEVEL 可以是下列其中一項：

術語	意義
never	永遠不需要核准
any-change	需要任何 IAM 或 security-group-related 變更的核准
broadening (預設值)	新增 IAM 陳述式或流量規則時需要核准；移除不需要核准

此設定也可以在cdk.json檔案中進行配置。

```
{
  "app": "...",
  "requireApproval": "never"
}
```

比較堆疊

該cdk diff命令將應用程序中定義的堆棧的當前版本（及其依賴項）與已部署的版本或保存的 AWS CloudFormation 模板進行比較，並顯示更改列表。

```
Stack HelloCdkStack
IAM Statement Changes
###############################################################################################
#   # Resource                    # Effect # Action                      # Principal
                  # Condition #
###############################################################################################
# + # ${Custom::S3AutoDeleteObject # Allow  # sts:AssumeRole              #
 Service:lambda.amazonaws.com #           #
```

```
#   # sCustomResourceProvider/Role #        #                                #
                    #           #
#   # .Arn}                          #        #                                #
                    #           #
#################################################################################################
# + # ${MyFirstBucket.Arn}          # Allow  # s3:DeleteObject*              # AWS:
${Custom::S3AutoDeleteOb #           #
#   # ${MyFirstBucket.Arn}/*        #        # s3:GetBucket*                 #
 jectsCustomResourceProvider/ #           #
#   #                              #        # s3:GetObject*                 # Role.Arn}
                    #           #
#   #                              #        # s3:List*                      #
                    #           #
#################################################################################################
IAM Policy Changes
#################################################################################################
#   # Resource                                                  # Managed Policy ARN
                              #
#################################################################################################
# + # ${Custom::S3AutoDeleteObjectsCustomResourceProvider/Ro # {"Fn::Sub":"arn:
${AWS::Partition}:iam::aws:policy/serv #
#   # le}                                                       # ice-role/
AWSLambdaBasicExecutionRole"}                #
#################################################################################################
(NOTE: There may be security-related changes not in this list. See https://github.com/
aws/aws-cdk/issues/1299)

Parameters
[+] Parameter
 AssetParameters/4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392/
S3Bucket
 AssetParameters4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392S3BucketBF7A7F3
 {"Type":"String","Description":"S3 bucket for asset
 \"4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392\""}
[+] Parameter
 AssetParameters/4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392/
S3VersionKey
 AssetParameters4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392S3VersionKeyFAF
 {"Type":"String","Description":"S3 key for asset version
 \"4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392\""}
[+] Parameter
 AssetParameters/4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392/
ArtifactHash
 AssetParameters4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392ArtifactHashE56
```

```
 {"Type":"String","Description":"Artifact hash for asset
 \"4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392\""}

Resources
[+] AWS::S3::BucketPolicy MyFirstBucket/Policy MyFirstBucketPolicy3243DEFD
[+] Custom::S3AutoDeleteObjects MyFirstBucket/AutoDeleteObjectsCustomResource
 MyFirstBucketAutoDeleteObjectsCustomResourceC52FCF6E
[+] AWS::IAM::Role Custom::S3AutoDeleteObjectsCustomResourceProvider/Role
 CustomS3AutoDeleteObjectsCustomResourceProviderRole3B1BD092
[+] AWS::Lambda::Function Custom::S3AutoDeleteObjectsCustomResourceProvider/Handler
 CustomS3AutoDeleteObjectsCustomResourceProviderHandler9D90184F
[~] AWS::S3::Bucket MyFirstBucket MyFirstBucketB8884501
 ## [~] DeletionPolicy
 #   ## [-] Retain
 #   ## [+] Delete
 ## [~] UpdateReplacePolicy
     ## [-] Retain
     ## [+] Delete
```

要將應用程序的堆棧與現有部署進行比較：

```
cdk diff MyStack
```

若要將應用程式的堆疊與儲存的 CloudFormation 範本進行比較：

```
cdk diff --template ~/stacks/MyStack.old MyStack
```

將現有資源匯入堆疊

您可以使用`cdk import`命令將資源置於特定 AWS CDK 堆疊 CloudFormation 的管理之下。如果您要遷移到堆棧 AWS CDK，或在堆棧之間移動資源或更改其邏輯 ID，這非常有`cdk import`用。使用 CloudFormation 資源導入。請參閱此處可匯入的資源清單。

若要將現有資源匯入 AWS CDK 堆疊，請依照下列步驟執行：

- 確保資源當前未由任何其他 CloudFormation 堆棧管理。如果是，請先在資源目前所在的堆疊`RemovalPolicy.RETAIN`中將移除原則設定為，然後執行部署。然後，從堆疊中移除資源並執行另一個部署。此過程將確保資源不再受管理， CloudFormation 但不會刪除它。
- 執行 a `cdk diff` 以確定您要匯入資源的 AWS CDK 堆疊沒有擱置的變更。「導入」操作中允許的唯一更改是添加要導入的新資源。

- 為要導入到堆棧的資源添加構造。例如，如果您想導入 Amazon S3 存儲桶，請添加類似的內容`new s3.Bucket(this, 'ImportedS3Bucket', {});`。請勿對任何其他資源進行任何修改。

  您還必須確保將資源當前具有的狀態精確建立到定義中的模型。對於值區的範例，請務必包含 AWS KMS 金鑰、生命週期原則，以及任何與值區相關的其他項目。如果不這樣做，後續的更新作業可能無法執行您預期的作業。

  您可以選擇是否要包含實體值區名稱。我們通常建議不要將資源名稱包含在資 AWS CDK 源定義中，以便更輕鬆地多次部署資源。
- 執行 `cdk import` *`STACKNAME`*。
- 如果資源名稱不在您的模型中，CLI 將提示您傳入正在匯入的資源的實際名稱。在此之後，導入開始。
- 當`cdk import`報告成功時，資源現在由 AWS CDK 和管理 CloudFormation。您對應用程序中的資源屬性所做的任何後續更改構造配置都將 AWS CDK 應用於下一次部署。
- 要確認 AWS CDK 應用程序中的資源定義與資源的當前狀態匹配，您可以啟動CloudFormation 漂移檢測操作。

此功能目前不支援將資源匯入巢狀堆疊。

配置（`cdk.json`）

許多 CDK Toolkit 命令列旗標的預設值可以儲存在專`cdk.json`案的檔案或使用者目錄中的`.cdk.json`檔案中。以下是支援的組態設定的字母參照。

金鑰	備註	CDK 工具包選項
app	執行 CDK 應用程式的指令。	--app
`assetMetadata`	如果`false`，CDK 不會將中繼資料新增至使用資產的資源。	--no-asset-metadata
`bootstrapKmsKeyId`	覆寫用於加密 Amazon S3 部署儲存貯體的 AWS KMS 金鑰識別碼。	--bootstrap-kms-key-id

金鑰	備註	CDK 工具包選項
build	在合成之前編譯或構建 CDK 應用程序的命令。不允許在~/.cdk.json 。	--build
browser	用於啟動子指令之 Web 瀏覽器的cdk docs指令。	--browser
context	請參閱the section called “Context”。組態檔案中的前後關聯值不會被清除cdk context --clear。CDK 工具組會將快取的內容值放入cdk.context.json .)	--context
debug	如果true，CDK 工具包會發出更詳細的信息，有用於調試。	--debug
language	用於初始化新項目的語言。	--language
lookups	如果false，則不允許任何前後關聯查詢。如果需要執行任何上下文查找，合成將失敗。	--no-lookups
notices	如果false，則會隱藏有關安全性弱點、回歸和不支援版本的訊息顯示。	--no-notices
output	將合成雲端組件發射到其中的目錄名稱 (預設值"cdk.out")。	--output
outputsFile	將從已部署堆疊的 AWS CloudFormation 輸出寫入的檔案 (以 JSON 格式)。	--outputs-file

金鑰	備註	CDK 工具包選項
`pathMetadata`	如果`false`，則不會將 CDK 路徑中繼資料新增至合成範本。	--no-path-metadata
`plugin`	JSON 陣列，指定擴展 CDK 的軟件包名稱或本地路徑	--plugin
`profile`	用於指定區域和帳戶認證的預設設定 AWS 檔名稱。	--profile
`progress`	如果設定為`"events"`，CDK Toolkit 會在部署期間顯示所有 AWS CloudFormation 事件，而不是進度列。	--progress
`requireApproval`	安全性變更的預設核准層級。請參閱the section called “安全性相關變更”	--require-approval
`rollback`	如果`false`，則不會復原失敗的部署。	--no-rollback
`staging`	如果`false`，資源不會複製到輸出目錄（用於源文件的本地調試 AWS SAM）。	--no-staging
`tags`	包含堆棧的標籤（鍵值對）的 JSON 對象。	--tags
`toolkitBucketName`	用於部署 Lambda 函數和容器映像等資產的 Amazon S3 儲存貯體的名稱 (請參閱the section called “引導您的環境 AWS”。	--toolkit-bucket-name
`toolkitStackName`	啟動程序堆疊的名稱 (請參閱the section called “引導您的環境 AWS”。	--toolkit-stack-name

金鑰	備註	CDK 工具包選項
`versionReporting`	如果`false`選擇退出版本報告。	--no-version-reporting
`watch`	JSON 物件包含以`"include"` 及指示哪些檔案應該 (或不應該) 在變更時觸發專案重建的`"exclude"` 金鑰。請參閱the section called “手錶模式”。	--watch

cdk migrate指令參考

AWS Cloud Development Kit (AWS CDK) 指令行介面 (CLI) 指`cdk migrate`令的參考。如需使用的詳細資訊`cdk migrate`，請參閱將現有資源和 AWS CloudFormation 範本移轉至 AWS CDK。

此命`cdk migrate`令會將已部署的 AWS 資源、 AWS CloudFormation 堆疊和本機 AWS CloudFormation 範本移轉至 AWS CDK。

主題

- 用量
- 選項

用量

```
$ cdk migrate <options>
```

選項

必要的選項

`--stack-name` *STRING*

移轉後將在 CDK 應用程式中建立的 AWS CloudFormation 堆疊名稱。

必要：是

條件式選項

`--from-path` *PATH*

要移轉之 AWS CloudFormation 範本的路徑。提供此選項以指定本端樣板。

必要：有條件限制。如果從本端 AWS CloudFormation 樣板移轉，則需要此選項。

`--from-scan` *STRING*

從 AWS 環境移轉已部署的資源時，請使用此選項指定是否應啟動新掃描，或是否 AWS CDK CLI 應使用上次成功掃描。

必要：有條件限制。從部署的 AWS 資源移轉時需要此選項。

接受的值：`most-recent`，`new`

`--from-stack`

提供此選項以從已部署的 AWS CloudFormation 堆疊移轉。用`--stack-name`於指定已部署 AWS CloudFormation 堆疊的名稱。

必要：有條件限制。如果從已部署的 AWS CloudFormation 堆疊移轉，則為必要項

選擇性選項

`--account` *STRING*

要從中擷取 AWS CloudFormation 堆疊範本的帳戶。

必要：否

預設值：從預設來源 AWS CDK CLI取得科目資訊。

`--compress`

提供此選項可將產生的 CDK 專案壓縮為ZIP檔案。

必要：否

`--filter` *ARRAY*

從帳號和移轉已部署的 AWS 資源時使用 AWS 區域。此選項指定篩選器，以決定要移轉哪些已部署的資源。

此選項接受鍵-值對，其中鍵表示過濾器類型和值表示要過濾的值的數組。

以下是可接受的金鑰：

- `resource-identifier`— 資源的識別元。值可以是資源邏輯或實體 ID。例如 `resource-identifier="ClusterName"`。
- `resource-type-prefix`— AWS CloudFormation 資源類型前置詞。例如，指定篩`resource-type-prefix="AWS::DynamoDB::"`選所有 Amazon DynamoDB 資源。
- `tag-key`— 資源標籤的索引鍵。例如 `tag-key="myTagKey"`。
- `tag-value`— 資源標籤的值。例如 `tag-value="myTagValue"`。

為AND條件式邏輯提供多個鍵值對。下列範例會篩選任何標記myTagKey為標籤鍵的 DynamoDB 資源：。`--filter resource-type-prefix="AWS::DynamoDB::", tag-key="myTagKey"`

在OR條件式邏輯的單一命令中多次提供`--filter`選項。下列範例會篩選任何屬於 DynamoDB 資源或標記myTagKey為標籤鍵的資源：`--filter resource-type-prefix="AWS::DynamoDB::" --filter tag-key="myTagKey"`

必要：否

`--language` *`STRING`*

移轉期間所建立之 CDK 專案所使用的程式設計語言。

必要：否

接受的值：typescriptpython、java、、csharp、go。

預設：typescript

`--output-path` *`PATH`*

移轉的 CDK 專案的輸出路徑。

必要：否

預設值：依預設， AWS CDK CLI將使用您目前的工作目錄。

`--region` *`STRING`*

AWS 區域 要從中擷取 AWS CloudFormation 堆疊範本的。

必要：否

預設值：從預設來源 AWS CDK CLI取得 AWS 區域 資訊。

AWS Toolkit for Visual Studio 代碼的工具包

Visual Studio 程式碼的AWS 工具組是適用於 Visual Studio 程式碼的開放原始碼外掛程式，可讓您更輕鬆地在上建立、偵錯和部署應用程式 AWS。此工具組提供開發 AWS CDK 應用程式的整合體驗。它包括資 AWS CDK 源管理器功能列出您的 AWS CDK 項目並瀏覽 CDK 應用程序的各種組件。安裝 AWS 工具包並了解有關使用 AWS CDK 資源管理器的更多信息。

AWS SAM 整合

AWS CDK 和 AWS Serverless Application Model (AWS SAM) 可以共同運作，讓您在本機建置和測試 CDK 中定義的無伺服器應用程式。如需完整資訊，請參閱AWS Cloud Development Kit (AWS CDK)開 AWS SAM 發人員指南中的。若要安裝 SAM CLI，請參閱安裝 AWS SAM CLI。

測試結構

有了 AWS CDK，您的基礎結構可以像您撰寫的任何其他程式碼一樣進行測試。測試 AWS CDK 應用程序的標準方法使用 AWS CDK的斷言模塊和流行的測試框架，如 Jest for TypeScript 和 JavaScript 或 Pytest for Python。

您可以為 AWS CDK 應用程序編寫兩類測試。

- 細粒度斷言會測試生成的 AWS CloudFormation 模板的特定方面，例如「此資源具有此值的此屬性」。這些測試可以檢測回歸。當您使用測試驅動開發開發來開發新功能時，它們也很有用。（您可以先編寫測試，然後通過編寫正確的實現來通過它。）細粒度斷言是最常用的測試。
- 快照測試會根據先前儲存的基準 AWS CloudFormation 範本來測試合成的範本。快照測試使您可以自由重構，因為您可以確保重構代碼的工作方式與原始代碼完全相同。如果變更是有意為之，您可以接受未來測試的新基準。不過，CDK 升級也可能導致合成範本發生變更，因此您不能只依賴快照來確保您的實作正確無誤。

> **Note**
>
> 本主題中作為範例使用的 TypeScript、Python 和 Java 應用程式的完整版本可在上取得 GitHub。

開始使用

為了說明如何編寫這些測試，我們將創建一個包含 AWS Step Functions 狀態機和 AWS Lambda 函數的堆棧。Lambda 函數已訂閱 Amazon SNS 主題，只需將訊息轉寄到狀態機器即可。

首先，使用 CDK Toolkit 創建一個空的 CDK 應用程序項目並安裝我們需要的庫。我們將使用的構造都在主 CDK 包中，這是使用 CDK Toolkit 創建的項目中的默認依賴項。但是，您必須安裝測試框架。

TypeScript

```
mkdir state-machine && cd state-machine
cdk init --language=typescript
npm install --save-dev jest @types/jest
```

為您的測試創建一個目錄。

```
mkdir test
```

編輯該項目package.json以告訴 NPM 如何運行 Jest，並告訴 Jest 要收集哪些類型的文件。必要的變更如下。

- 將新的test金鑰新增至scripts區段
- 將開玩笑及其類型添加到該devDependencies部分
- 使用moduleFileExtensions聲明添加新的jest頂級密鑰

這些變更顯示在下列大綱中。將新文字置於中所示的位置package.json。「...」預留位置表示檔案中不應變更的現有部分。

```
{
  ...
  "scripts": {
    ...
    "test": "jest"
  },
  "devDependencies": {
    ...
    "@types/jest": "^24.0.18",
    "jest": "^24.9.0"
  },
  "jest": {
    "moduleFileExtensions": ["js"]
  }
}
```

JavaScript

```
mkdir state-machine && cd state-machine
cdk init --language=javascript
npm install --save-dev jest
```

為您的測試創建一個目錄。

```
mkdir test
```

編輯該項目package.json以告訴 NPM 如何運行 Jest，並告訴 Jest 要收集哪些類型的文件。必要的變更如下。

- 將新的test金鑰新增至scripts區段
- 將開玩笑添加到該部分 devDependencies
- 使用moduleFileExtensions聲明添加新的jest頂級密鑰

這些變更顯示在下列大綱中。將新文字置於中所示的位置package.json。「...」佔位符表示文件中不應該更改的現有部分。

```
{
  ...
  "scripts": {
    ...
    "test": "jest"
  },
  "devDependencies": {
    ...
    "jest": "^24.9.0"
  },
  "jest": {
    "moduleFileExtensions": ["js"]
  }
}
```

Python

```
mkdir state-machine && cd state-machine
cdk init --language=python
source .venv/bin/activate
python -m pip install -r requirements.txt
python -m pip install -r requirements-dev.txt
```

Java

```
mkdir state-machine && cd-state-machine
cdk init --language=java
```

在您偏好的 Java IDE 中開啟專案。（在 Eclipse 中，使用文件 > 導入 > 現有的 Maven 項目。）

C#

```
mkdir state-machine && cd-state-machine
cdk init --language=csharp
```

src\StateMachine.sln在視覺工作室中開啟。

以滑鼠右鍵按一下 [方案總管] 中的方案，然後選擇 [新增] 搜索 MSTest C # 並為 C # 添加一個 MSTest 測試項目。(預設名稱TestProject1沒問題。)

按一下滑鼠右鍵TestProject1並選擇「新增」>「專案參考」，然後將StateMachine專案新增為參考。

示例堆棧

以下是將在本主題中測試的堆疊。如前所述，它包含 Lambda 函數和 Step Functions 數狀態機器，並接受一或多個 Amazon SNS 主題。Lambda 函數已訂閱 Amazon SNS 主題，並將其轉寄至狀態機器。

您不必做任何特別的事情即可使應用程序可測試。實際上，此 CDK 堆棧與本指南中的其他示例堆棧在任何重要的方式上都沒有區別。

TypeScript

請將下列程式碼放入lib/state-machine-stack.ts：

```
import * as cdk from "aws-cdk-lib";
import * as sns from "aws-cdk-lib/aws-sns";
import * as sns_subscriptions from "aws-cdk-lib/aws-sns-subscriptions";
import * as lambda from "aws-cdk-lib/aws-lambda";
import * as sfn from "aws-cdk-lib/aws-stepfunctions";
import { Construct } from "constructs";

export interface StateMachineStackProps extends cdk.StackProps {
  readonly topics: sns.Topic[];
}

export class StateMachineStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props: StateMachineStackProps) {
    super(scope, id, props);

    // In the future this state machine will do some work...
```

```
    const stateMachine = new sfn.StateMachine(this, "StateMachine", {
      definition: new sfn.Pass(this, "StartState"),
    });

    // This Lambda function starts the state machine.
    const func = new lambda.Function(this, "LambdaFunction", {
      runtime: lambda.Runtime.NODEJS_18_X,
      handler: "handler",
      code: lambda.Code.fromAsset("./start-state-machine"),
      environment: {
        STATE_MACHINE_ARN: stateMachine.stateMachineArn,
      },
    });
    stateMachine.grantStartExecution(func);

    const subscription = new sns_subscriptions.LambdaSubscription(func);
    for (const topic of props.topics) {
      topic.addSubscription(subscription);
    }
  }
}
```

JavaScript

請將下列程式碼放入`lib/state-machine-stack.js`：

```
const cdk = require("aws-cdk-lib");
const sns = require("aws-cdk-lib/aws-sns");
const sns_subscriptions = require("aws-cdk-lib/aws-sns-subscriptions");
const lambda = require("aws-cdk-lib/aws-lambda");
const sfn = require("aws-cdk-lib/aws-stepfunctions");

class StateMachineStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    // In the future this state machine will do some work...
    const stateMachine = new sfn.StateMachine(this, "StateMachine", {
      definition: new sfn.Pass(this, "StartState"),
    });

    // This Lambda function starts the state machine.
    const func = new lambda.Function(this, "LambdaFunction", {
      runtime: lambda.Runtime.NODEJS_18_X,
```

```
      handler: "handler",
      code: lambda.Code.fromAsset("./start-state-machine"),
      environment: {
        STATE_MACHINE_ARN: stateMachine.stateMachineArn,
      },
    });
    stateMachine.grantStartExecution(func);

    const subscription = new sns_subscriptions.LambdaSubscription(func);
    for (const topic of props.topics) {
      topic.addSubscription(subscription);
    }
  }
}

module.exports = { StateMachineStack }
```

Python

請將下列程式碼放入`state_machine/state_machine_stack.py`：

```
from typing import List

import aws_cdk.aws_lambda as lambda_
import aws_cdk.aws_sns as sns
import aws_cdk.aws_sns_subscriptions as sns_subscriptions
import aws_cdk.aws_stepfunctions as sfn
import aws_cdk as cdk

class StateMachineStack(cdk.Stack):
    def __init__(
        self,
        scope: cdk.Construct,
        construct_id: str,
        *,
        topics: List[sns.Topic],
        **kwargs
    ) -> None:
        super().__init__(scope, construct_id, **kwargs)

        # In the future this state machine will do some work...
        state_machine = sfn.StateMachine(
            self, "StateMachine", definition=sfn.Pass(self, "StartState")
        )
```

```
        # This Lambda function starts the state machine.
        func = lambda_.Function(
            self,
            "LambdaFunction",
            runtime=lambda_.Runtime.NODEJS_18_X,
            handler="handler",
            code=lambda_.Code.from_asset("./start-state-machine"),
            environment={
                "STATE_MACHINE_ARN": state_machine.state_machine_arn,
            },
        )
        state_machine.grant_start_execution(func)

        subscription = sns_subscriptions.LambdaSubscription(func)
        for topic in topics:
            topic.add_subscription(subscription)
```

Java

```
package software.amazon.samples.awscdkassertionssamples;

import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.services.lambda.Code;
import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.lambda.Runtime;
import software.amazon.awscdk.services.sns.ITopicSubscription;
import software.amazon.awscdk.services.sns.Topic;
import software.amazon.awscdk.services.sns.subscriptions.LambdaSubscription;
import software.amazon.awscdk.services.stepfunctions.Pass;
import software.amazon.awscdk.services.stepfunctions.StateMachine;

import java.util.Collections;
import java.util.List;

public class StateMachineStack extends Stack {
    public StateMachineStack(final Construct scope, final String id, final
 List<Topic> topics) {
        this(scope, id, null, topics);
    }
```

```
    public StateMachineStack(final Construct scope, final String id, final
 StackProps props, final List<Topic> topics) {
        super(scope, id, props);

        // In the future this state machine will do some work...
        final StateMachine stateMachine = StateMachine.Builder.create(this,
 "StateMachine")
                .definition(new Pass(this, "StartState"))
                .build();

        // This Lambda function starts the state machine.
        final Function func = Function.Builder.create(this, "LambdaFunction")
                .runtime(Runtime.NODEJS_18_X)
                .handler("handler")
                .code(Code.fromAsset("./start-state-machine"))
                .environment(Collections.singletonMap("STATE_MACHINE_ARN",
 stateMachine.getStateMachineArn()))
                .build();
        stateMachine.grantStartExecution(func);

        final ITopicSubscription subscription = new LambdaSubscription(func);
        for (final Topic topic : topics) {
            topic.addSubscription(subscription);
        }
    }
}
```

C#

```
using Amazon.CDK;
using Amazon.CDK.AWS.Lambda;
using Amazon.CDK.AWS.StepFunctions;
using Amazon.CDK.AWS.SNS;
using Amazon.CDK.AWS.SNS.Subscriptions;
using Constructs;

using System.Collections.Generic;

namespace AwsCdkAssertionSamples
{
    public class StateMachineStackProps : StackProps
    {
        public Topic[] Topics;
```

```
    }

    public class StateMachineStack : Stack
    {

        internal StateMachineStack(Construct scope, string id,
 StateMachineStackProps props = null) : base(scope, id, props)
        {
            // In the future this state machine will do some work...
            var stateMachine = new StateMachine(this, "StateMachine", new
 StateMachineProps
            {
                Definition = new Pass(this, "StartState")
            });

            // This Lambda function starts the state machine.
            var func = new Function(this, "LambdaFunction", new FunctionProps
            {
                Runtime = Runtime.NODEJS_18_X,
                Handler = "handler",
                Code = Code.FromAsset("./start-state-machine"),
                Environment = new Dictionary<string, string>
                {
                    { "STATE_MACHINE_ARN", stateMachine.StateMachineArn }
                }
            });
            stateMachine.GrantStartExecution(func);

            foreach (Topic topic in props?.Topics ?? new Topic[0])
            {
                var subscription = new LambdaSubscription(func);
            }

        }
    }
}
```

我們將修改應用程序的主入口點，以便我們不實際實例化我們的堆棧。我們不想不小心部署它。我們的測試將創建一個應用程序和一個用於測試的堆棧實例。與測試驅動開發結合使用時，這是一種有用的策略：在啟用部署之前確保堆棧通過所有測試。

TypeScript

在 bin/state-machine.ts 中：

```
#!/usr/bin/env node
import * as cdk from "aws-cdk-lib";

const app = new cdk.App();

// Stacks are intentionally not created here -- this application isn't meant to
// be deployed.
```

JavaScript

在 bin/state-machine.js 中：

```
#!/usr/bin/env node
const cdk = require("aws-cdk-lib");

const app = new cdk.App();

// Stacks are intentionally not created here -- this application isn't meant to
// be deployed.
```

Python

在 app.py 中：

```
#!/usr/bin/env python3
import os

import aws_cdk  as cdk

app = cdk.App()

# Stacks are intentionally not created here -- this application isn't meant to
# be deployed.

app.synth()
```

Java

```
package software.amazon.samples.awscdkassertionssamples;
```

```
import software.amazon.awscdk.App;

public class SampleApp {
    public static void main(final String[] args) {
        App app = new App();

        // Stacks are intentionally not created here -- this application isn't meant
 to be deployed.

        app.synth();
    }
}
```

C#

```
using Amazon.CDK;

namespace AwsCdkAssertionSamples
{
    sealed class Program
    {
        public static void Main(string[] args)
        {
            var app = new App();

            // Stacks are intentionally not created here -- this application isn't
 meant to be deployed.

            app.Synth();
        }
    }
}
```

Lambda 函數

我們的範例堆疊包含啟動狀態機器的 Lambda 函數。我們必須提供此函數的原始程式碼，以便 CDK 可以將其捆綁並部署為建立 Lambda 函數資源的一部分。

- `start-state-machine`在應用程序的主目錄中創建文件夾。

- 在此資料夾中，至少建立一個檔案。例如，您可以在中儲存下列程式碼`start-state-machines/index.js`。

```
exports.handler = async function (event, context) {
 return 'hello world';
};
```

  但是，任何文件都可以工作，因為我們實際上不會部署堆棧。

執行測試

作為參考，以下是您用於在應用 AWS CDK 程序中運行測試的命令。這些命令與您在使用相同測試框架的任何項目中用於運行測試的命令相同。對於需要構建步驟的語言，請包括以確保您的測試已編譯。

TypeScript

```
tsc && npm test
```

JavaScript

```
npm test
```

Python

```
python -m pytest
```

Java

```
mvn compile && mvn test
```

C#

構建您的解決方案（F6）以發現測試，然後運行測試（測試 > 運行所有測試）。要選擇要運行的測試，請打開測試資源管理器（測試 > 測試資源管理器）。

或者：

```
dotnet test src
```

細粒度斷言

使用細粒度斷言測試堆棧的第一步是合成堆棧，因為我們正在針對生成的模板編寫斷言。 AWS CloudFormation

我們StateMachineStackStack要求我們將其傳遞給 Amazon SNS 主題，以轉發到狀態機。因此，在我們的測試中，我們將創建一個單獨的堆棧來包含該主題。

通常，在編寫 CDK 應用程序時，您可以在堆棧的構造函數中對 Amazon SNS 主題進行子類別Stack和實例化。在我們的測試中，我們Stack直接實例化，然後將此堆棧作為範圍傳遞，將其附加到堆棧中。Topic這在功能上是等價的，而且不太冗長。它還有助於使僅在測試中使用的堆棧與您打算部署的堆棧「看起來不同」。

TypeScript

```
import { Capture, Match, Template } from "aws-cdk-lib/assertions";
import * as cdk from "aws-cdk-lib";
import * as sns from "aws-cdk-lib/aws-sns";
import { StateMachineStack } from "../lib/state-machine-stack";

describe("StateMachineStack", () => {
  test("synthesizes the way we expect", () => {
    const app = new cdk.App();

    // Since the StateMachineStack consumes resources from a separate stack
    // (cross-stack references), we create a stack for our SNS topics to live
    // in here. These topics can then be passed to the StateMachineStack later,
    // creating a cross-stack reference.
    const topicsStack = new cdk.Stack(app, "TopicsStack");

    // Create the topic the stack we're testing will reference.
    const topics = [new sns.Topic(topicsStack, "Topic1", {})];

    // Create the StateMachineStack.
    const stateMachineStack = new StateMachineStack(app, "StateMachineStack", {
      topics: topics, // Cross-stack reference
    });

    // Prepare the stack for assertions.
    const template = Template.fromStack(stateMachineStack);
```

```
}
```

JavaScript

```
const { Capture, Match, Template } = require("aws-cdk-lib/assertions");
const cdk = require("aws-cdk-lib");
const sns = require("aws-cdk-lib/aws-sns");
const { StateMachineStack } = require("../lib/state-machine-stack");

describe("StateMachineStack", () => {
  test("synthesizes the way we expect", () => {
    const app = new cdk.App();

    // Since the StateMachineStack consumes resources from a separate stack
    // (cross-stack references), we create a stack for our SNS topics to live
    // in here. These topics can then be passed to the StateMachineStack later,
    // creating a cross-stack reference.
    const topicsStack = new cdk.Stack(app, "TopicsStack");

    // Create the topic the stack we're testing will reference.
    const topics = [new sns.Topic(topicsStack, "Topic1", {})];

    // Create the StateMachineStack.
    const StateMachineStack = new StateMachineStack(app, "StateMachineStack", {
      topics: topics, // Cross-stack reference
    });

    // Prepare the stack for assertions.
    const template = Template.fromStack(stateMachineStack);
```

Python

```
from aws_cdk import aws_sns as sns
import aws_cdk as cdk
from aws_cdk.assertions import Template

from app.state_machine_stack import StateMachineStack

def test_synthesizes_properly():
    app = cdk.App()

    # Since the StateMachineStack consumes resources from a separate stack
    # (cross-stack references), we create a stack for our SNS topics to live
```

```
    # in here. These topics can then be passed to the StateMachineStack later,
    # creating a cross-stack reference.
    topics_stack = cdk.Stack(app, "TopicsStack")

    # Create the topic the stack we're testing will reference.
    topics = [sns.Topic(topics_stack, "Topic1")]

    # Create the StateMachineStack.
    state_machine_stack = StateMachineStack(
        app, "StateMachineStack", topics=topics  # Cross-stack reference
    )

    # Prepare the stack for assertions.
    template = Template.from_stack(state_machine_stack)
```

Java

```
package software.amazon.samples.awscdkassertionssamples;

import org.junit.jupiter.api.Test;
import software.amazon.awscdk.assertions.Capture;
import software.amazon.awscdk.assertions.Match;
import software.amazon.awscdk.assertions.Template;
import software.amazon.awscdk.App;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.services.sns.Topic;

import java.util.*;

import static org.assertj.core.api.Assertions.assertThat;

public class StateMachineStackTest {
    @Test
    public void testSynthesizesProperly() {
        final App app = new App();

        // Since the StateMachineStack consumes resources from a separate stack
 (cross-stack references), we create a stack
        // for our SNS topics to live in here. These topics can then be passed to
 the StateMachineStack later, creating a
        // cross-stack reference.
        final Stack topicsStack = new Stack(app, "TopicsStack");
```

```
        // Create the topic the stack we're testing will reference.
        final List<Topic> topics =
 Collections.singletonList(Topic.Builder.create(topicsStack, "Topic1").build());

        // Create the StateMachineStack.
        final StateMachineStack stateMachineStack = new StateMachineStack(
                app,
                "StateMachineStack",
                topics // Cross-stack reference
        );

        // Prepare the stack for assertions.
        final Template template = Template.fromStack(stateMachineStack)
```

C#

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

using Amazon.CDK;
using Amazon.CDK.AWS.SNS;
using Amazon.CDK.Assertions;
using AwsCdkAssertionSamples;

using ObjectDict = System.Collections.Generic.Dictionary<string, object>;
using StringDict = System.Collections.Generic.Dictionary<string, string>;

namespace TestProject1
{
    [TestClass]
    public class StateMachineStackTest
    {
        [TestMethod]
        public void TestMethod1()
        {
            var app = new App();

            // Since the StateMachineStack consumes resources from a separate stack
 (cross-stack references), we create a stack
            // for our SNS topics to live in here. These topics can then be passed
 to the StateMachineStack later, creating a
            // cross-stack reference.
            var topicsStack = new Stack(app, "TopicsStack");
```

```
            // Create the topic the stack we're testing will reference.
            var topics = new Topic[] { new Topic(topicsStack, "Topic1") };

            // Create the StateMachineStack.
            var StateMachineStack = new StateMachineStack(app, "StateMachineStack",
 new StateMachineStackProps
            {
                Topics = topics
            });

            // Prepare the stack for assertions.
            var template = Template.FromStack(stateMachineStack);

            // test will go here
        }
    }
}
```

現在，我們可以斷言 Lambda 函數和 Amazon SNS 訂閱已經建立。

TypeScript

```
    // Assert it creates the function with the correct properties...
    template.hasResourceProperties("AWS::Lambda::Function", {
      Handler: "handler",
      Runtime: "nodejs14.x",
    });

    // Creates the subscription...
    template.resourceCountIs("AWS::SNS::Subscription", 1);
```

JavaScript

```
    // Assert it creates the function with the correct properties...
    template.hasResourceProperties("AWS::Lambda::Function", {
      Handler: "handler",
      Runtime: "nodejs14.x",
    });

    // Creates the subscription...
    template.resourceCountIs("AWS::SNS::Subscription", 1);
```

Python

```
# Assert that we have created the function with the correct properties
    template.has_resource_properties(
        "AWS::Lambda::Function",
        {
            "Handler": "handler",
            "Runtime": "nodejs14.x",
        },
    )

    # Assert that we have created a subscription
    template.resource_count_is("AWS::SNS::Subscription", 1)
```

Java

```
        // Assert it creates the function with the correct properties...
        template.hasResourceProperties("AWS::Lambda::Function", Map.of(
                "Handler", "handler",
                "Runtime", "nodejs14.x"
        ));

         // Creates the subscription...
        template.resourceCountIs("AWS::SNS::Subscription", 1);
```

C#

```
            // Prepare the stack for assertions.
            var template = Template.FromStack(stateMachineStack);

            // Assert it creates the function with the correct properties...
            template.HasResourceProperties("AWS::Lambda::Function", new StringDict {
                { "Handler", "handler"},
                { "Runtime", "nodejs14x" }
            });

            // Creates the subscription...
            template.ResourceCountIs("AWS::SNS::Subscription", 1);
```

我們的 Lambda 函數測試聲明函數資源的兩個特定屬性具有特定的值。默認情況下，該hasResourceProperties方法對合成 CloudFormation 模板中給出的資源屬性執行部分匹配。此測試要求提供的屬性存在並具有指定的值，但資源也可以具有未測試的其他屬性。

我們的 Amazon SNS 聲明聲稱合成範本包含訂閱，但沒有與訂閱本身有關的內容。我們包括這個斷言主要是為了說明如何斷言資源計數。此Template類別提供更具體的方法，可針對 CloudFormation 範本的ResourcesOutputs、和Mapping區段撰寫宣告。

匹配器

的預設部分相符行為hasResourceProperties可以使用Match類別中的匹配器來變更。

匹配器範圍從寬鬆（Match.anyValue）到嚴格（Match.objectEquals）。它們可以嵌套，以將不同的匹配方法應用於資源屬性的不同部分。例如，使用Match.objectEquals並Match.anyValue一起使用，我們可以更完整地測試狀態機器的 IAM 角色，而不需要可能變更的屬性的特定值。

TypeScript

```
// Fully assert on the state machine's IAM role with matchers.
template.hasResourceProperties(
  "AWS::IAM::Role",
  Match.objectEquals({
    AssumeRolePolicyDocument: {
      Version: "2012-10-17",
      Statement: [
        {
          Action: "sts:AssumeRole",
          Effect: "Allow",
          Principal: {
            Service: {
              "Fn::Join": [
                "",
                ["states.", Match.anyValue(), ".amazonaws.com"],
              ],
            },
          },
        },
      ],
    },
  })
);
```

JavaScript

```
    // Fully assert on the state machine's IAM role with matchers.
    template.hasResourceProperties(
      "AWS::IAM::Role",
      Match.objectEquals({
        AssumeRolePolicyDocument: {
          Version: "2012-10-17",
          Statement: [
            {
              Action: "sts:AssumeRole",
              Effect: "Allow",
              Principal: {
                Service: {
                  "Fn::Join": [
                    "",
                    ["states.", Match.anyValue(), ".amazonaws.com"],
                  ],
                },
              },
            },
          ],
        },
      })
    );
```

Python

```
from aws_cdk.assertions import Match

    # Fully assert on the state machine's IAM role with matchers.
    template.has_resource_properties(
        "AWS::IAM::Role",
        Match.object_equals(
            {
                "AssumeRolePolicyDocument": {
                    "Version": "2012-10-17",
                    "Statement": [
                        {
                            "Action": "sts:AssumeRole",
                            "Effect": "Allow",
                            "Principal": {
                                "Service": {
```

```
                                "Fn::Join": [
                                    "",
                                    [
                                        "states.",
                                        Match.any_value(),
                                        ".amazonaws.com",
                                    ],
                                ],
                            },
                        },
                    },
                ],
            },
        }
    ),
)
```

Java

```
        // Fully assert on the state machine's IAM role with matchers.
        template.hasResourceProperties("AWS::IAM::Role", Match.objectEquals(
                Collections.singletonMap("AssumeRolePolicyDocument", Map.of(
                        "Version", "2012-10-17",
                        "Statement", Collections.singletonList(Map.of(
                                "Action", "sts:AssumeRole",
                                "Effect", "Allow",
                                "Principal", Collections.singletonMap(
                                        "Service", Collections.singletonMap(
                                                "Fn::Join", Arrays.asList(
                                                        "",
                                                        Arrays.asList("states.",
Match.anyValue(), ".amazonaws.com")
                                                )
                                        )
                                )
                        ))
                ))
        ));
```

C#

```
            // Fully assert on the state machine's IAM role with matchers.
            template.HasResource("AWS::IAM::Role", Match.ObjectEquals(new ObjectDict
```

```
            {
                { "AssumeRolePolicyDocument", new ObjectDict
                    {
                        { "Version", "2012-10-17" },
                        { "Action", "sts:AssumeRole" },
                        { "Principal", new ObjectDict
                            {
                                { "Version", "2012-10-17" },
                                { "Statement", new object[]
                                    {
                                        new ObjectDict {
                                            { "Action", "sts:AssumeRole" },
                                            { "Effect", "Allow" },
                                            { "Principal", new ObjectDict
                                                {
                                                    { "Service", new ObjectDict
                                                        {
                                                            { "", new object[]
                                                                { "states",
Match.AnyValue(), ".amazonaws.com" }
                                                            }
                                                        }
                                                    }
                                                }
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }));
```

許多 CloudFormation 資源包括表示為字符串的序列化 JSON 對象。`Match.serializedJson()`匹配器可用於匹配此 JSON 內的屬性。

例如，Step Functions 狀態機是使用以 JSON 為基礎的 Amazon 州語言中的字串定義的。我們將用`Match.serializedJson()`來確保我們的初始狀態是唯一的步驟。同樣，我們將使用嵌套匹配器將不同種類的匹配應用於對象的不同部分。

TypeScript

```
// Assert on the state machine's definition with the Match.serializedJson()
// matcher.
template.hasResourceProperties("AWS::StepFunctions::StateMachine", {
  DefinitionString: Match.serializedJson(
    // Match.objectEquals() is used implicitly, but we use it explicitly
    // here for extra clarity.
    Match.objectEquals({
      StartAt: "StartState",
      States: {
        StartState: {
          Type: "Pass",
          End: true,
          // Make sure this state doesn't provide a next state -- we can't
          // provide both Next and set End to true.
          Next: Match.absent(),
        },
      },
    })
  ),
});
```

JavaScript

```
// Assert on the state machine's definition with the Match.serializedJson()
// matcher.
template.hasResourceProperties("AWS::StepFunctions::StateMachine", {
  DefinitionString: Match.serializedJson(
    // Match.objectEquals() is used implicitly, but we use it explicitly
    // here for extra clarity.
    Match.objectEquals({
      StartAt: "StartState",
      States: {
        StartState: {
          Type: "Pass",
          End: true,
          // Make sure this state doesn't provide a next state -- we can't
          // provide both Next and set End to true.
          Next: Match.absent(),
        },
      },
    })
```

```
      ),
    });
```

Python

```
    # Assert on the state machine's definition with the serialized_json matcher.
    template.has_resource_properties(
        "AWS::StepFunctions::StateMachine",
        {
            "DefinitionString": Match.serialized_json(
                # Match.object_equals() is the default, but specify it here for
clarity
                Match.object_equals(
                    {
                        "StartAt": "StartState",
                        "States": {
                            "StartState": {
                                "Type": "Pass",
                                "End": True,
                                # Make sure this state doesn't provide a next state
--
                                # we can't provide both Next and set End to true.
                                "Next": Match.absent(),
                            },
                        },
                    }
                )
            ),
        },
    )
```

Java

```
        // Assert on the state machine's definition with the Match.serializedJson()
matcher.
        template.hasResourceProperties("AWS::StepFunctions::StateMachine",
Collections.singletonMap(
                "DefinitionString", Match.serializedJson(
                        // Match.objectEquals() is used implicitly, but we use it
explicitly here for extra clarity.
                        Match.objectEquals(Map.of(
                                "StartAt", "StartState",
                                "States", Collections.singletonMap(
```

```
                                        "StartState", Map.of(
                                                "Type", "Pass",
                                                "End", true,
                                                // Make sure this state doesn't
provide a next state -- we can't provide
                                                // both Next and set End to true.
                                                "Next", Match.absent()
                                        )
                                )
                        ))
                )
        ));
```

C#

```
            // Assert on the state machine's definition with the
Match.serializedJson() matcher
            template.HasResourceProperties("AWS::StepFunctions::StateMachine", new
ObjectDict
            {
                { "DefinitionString", Match.SerializedJson(
                    // Match.objectEquals() is used implicitly, but we use it
explicitly here for extra clarity.
                    Match.ObjectEquals(new ObjectDict {
                        { "StartAt", "StartState" },
                        { "States", new ObjectDict
                        {
                            { "StartState", new ObjectDict {
                                { "Type", "Pass" },
                                { "End", "True" },
                                // Make sure this state doesn't provide a next state
-- we can't provide
                                // both Next and set End to true.
                                { "Next", Match.Absent() }
                            }}
                        }}
                    })
                )}});
```

捕捉

測試屬性通常很有用，以確保它們遵循特定的格式，或與另一個屬性具有相同的值，而不需要提前知道它們的確切值。該assertions模塊在其Capture類中提供了此功能。

透過指定Capture實體來取代中的值hasResourceProperties，該值會保留在Capture物件中。實際捕獲的值可以使用對象的as方法來檢索asNumber()，包括asString()，和asObject，並進行測試。與匹Capture配器搭配使用，指定要在資源屬性中捕獲的值的確切位置，包括序列化的 JSON 屬性。

下面的示例測試，以確保我們的狀態機的啟動狀態有一個名稱開頭為Start。它還測試此狀態是否存在於機器中的狀態列表中。

TypeScript

```
// Capture some data from the state machine's definition.
const startAtCapture = new Capture();
const statesCapture = new Capture();
template.hasResourceProperties("AWS::StepFunctions::StateMachine", {
  DefinitionString: Match.serializedJson(
    Match.objectLike({
      StartAt: startAtCapture,
      States: statesCapture,
    })
  ),
});

// Assert that the start state starts with "Start".
expect(startAtCapture.asString()).toEqual(expect.stringMatching(/^Start/));

// Assert that the start state actually exists in the states object of the
// state machine definition.
expect(statesCapture.asObject()).toHaveProperty(startAtCapture.asString());
```

JavaScript

```
// Capture some data from the state machine's definition.
const startAtCapture = new Capture();
const statesCapture = new Capture();
template.hasResourceProperties("AWS::StepFunctions::StateMachine", {
  DefinitionString: Match.serializedJson(
    Match.objectLike({
```

```
          StartAt: startAtCapture,
          States: statesCapture,
        })
      ),
    });

    // Assert that the start state starts with "Start".
    expect(startAtCapture.asString()).toEqual(expect.stringMatching(/^Start/));

    // Assert that the start state actually exists in the states object of the
    // state machine definition.
    expect(statesCapture.asObject()).toHaveProperty(startAtCapture.asString());
```

Python

```
import re

    from aws_cdk.assertions import Capture

    # ...

    # Capture some data from the state machine's definition.
    start_at_capture = Capture()
    states_capture = Capture()
    template.has_resource_properties(
        "AWS::StepFunctions::StateMachine",
        {
            "DefinitionString": Match.serialized_json(
                Match.object_like(
                    {
                        "StartAt": start_at_capture,
                        "States": states_capture,
                    }
                )
            ),
        },
    )

    # Assert that the start state starts with "Start".
    assert re.match("^Start", start_at_capture.as_string())

    # Assert that the start state actually exists in the states object of the
    # state machine definition.
```

```
    assert start_at_capture.as_string() in states_capture.as_object()
```

Java

```
        // Capture some data from the state machine's definition.
        final Capture startAtCapture = new Capture();
        final Capture statesCapture = new Capture();
        template.hasResourceProperties("AWS::StepFunctions::StateMachine",
Collections.singletonMap(
                "DefinitionString", Match.serializedJson(
                        Match.objectLike(Map.of(
                                "StartAt", startAtCapture,
                                "States", statesCapture
                        ))
                )
        ));

        // Assert that the start state starts with "Start".
        assertThat(startAtCapture.asString()).matches("^Start.+");

        // Assert that the start state actually exists in the states object of the
state machine definition.
        assertThat(statesCapture.asObject()).containsKey(startAtCapture.asString());
```

C#

```
            // Capture some data from the state machine's definition.
            var startAtCapture = new Capture();
            var statesCapture = new Capture();
            template.HasResourceProperties("AWS::StepFunctions::StateMachine", new
ObjectDict
            {
                { "DefinitionString", Match.SerializedJson(
                    new ObjectDict
                    {
                        { "StartAt", startAtCapture },
                        { "States", statesCapture }
                    }
                )}
            });

            Assert.IsTrue(startAtCapture.ToString().StartsWith("Start"));
```

```
 Assert.IsTrue(statesCapture.AsObject().ContainsKey(startAtCapture.ToString()));
```

快照測試

在快照測試中，您可以將整個合成 CloudFormation 範本與先前儲存的基準 (通常稱為「主要」) 範本進行比較。與細粒度斷言不同，快照測試在捕獲回歸方面沒有用。這是因為快照集測試適用於整個範本，除了程式碼變更之外，其他項目可能會導致合成結果的小 (或 not-so-small) 差異。這些變更甚至可能不會影響您的部署，但仍會導致快照測試失敗。

例如，您可以更新 CDK 建構以納入新的最佳實務，這可能會導致合成資源的變更或組織方式。或者，您可以將 CDK Toolkit 更新為報告其他中繼資料的版本。對前後關聯值的變更也會影響合成範本。

但是，只要您保持不變，可能會影響合成模板的所有其他因素，快照測試在重構方面有很大的幫助。如果您所做的變更意外變更了範本，您將立即知道。如果是故意的變更，只要接受新範本作為基準線即可。

例如，如果我們有這樣的`DeadLetterQueue`結構：

TypeScript

```
export class DeadLetterQueue extends sqs.Queue {
  public readonly messagesInQueueAlarm: cloudwatch.IAlarm;

  constructor(scope: Construct, id: string) {
    super(scope, id);

    // Add the alarm
    this.messagesInQueueAlarm = new cloudwatch.Alarm(this, 'Alarm', {
      alarmDescription: 'There are messages in the Dead Letter Queue',
      evaluationPeriods: 1,
      threshold: 1,
      metric: this.metricApproximateNumberOfMessagesVisible(),
    });
  }
}
```

JavaScript

```
class DeadLetterQueue extends sqs.Queue {
```

```
  constructor(scope, id) {
    super(scope, id);

    // Add the alarm
    this.messagesInQueueAlarm = new cloudwatch.Alarm(this, 'Alarm', {
      alarmDescription: 'There are messages in the Dead Letter Queue',
      evaluationPeriods: 1,
      threshold: 1,
      metric: this.metricApproximateNumberOfMessagesVisible(),
    });
  }
}

module.exports = { DeadLetterQueue }
```

Python

```
class DeadLetterQueue(sqs.Queue):
    def __init__(self, scope: Construct, id: str):
        super().__init__(scope, id)

        self.messages_in_queue_alarm = cloudwatch.Alarm(
            self,
            "Alarm",
            alarm_description="There are messages in the Dead Letter Queue.",
            evaluation_periods=1,
            threshold=1,
            metric=self.metric_approximate_number_of_messages_visible(),
        )
```

Java

```
public class DeadLetterQueue extends Queue {
    private final IAlarm messagesInQueueAlarm;

    public DeadLetterQueue(@NotNull Construct scope, @NotNull String id) {
        super(scope, id);

        this.messagesInQueueAlarm = Alarm.Builder.create(this, "Alarm")
                .alarmDescription("There are messages in the Dead Letter Queue.")
                .evaluationPeriods(1)
                .threshold(1)
```

```
                .metric(this.metricApproximateNumberOfMessagesVisible())
                .build();
    }

    public IAlarm getMessagesInQueueAlarm() {
        return messagesInQueueAlarm;
    }
}
```

C#

```
namespace AwsCdkAssertionSamples
{
    public class DeadLetterQueue : Queue
    {
        public IAlarm messagesInQueueAlarm;

        public DeadLetterQueue(Construct scope, string id) : base(scope, id)
        {
            messagesInQueueAlarm = new Alarm(this, "Alarm", new AlarmProps
            {
                AlarmDescription = "There are messages in the Dead Letter Queue.",
                EvaluationPeriods = 1,
                Threshold = 1,
                Metric = this.MetricApproximateNumberOfMessagesVisible()
            });
        }
    }
}
```

我們可以像這樣測試它：

TypeScript

```
import { Match, Template } from "aws-cdk-lib/assertions";
import * as cdk from "aws-cdk-lib";
import { DeadLetterQueue } from "../lib/dead-letter-queue";

describe("DeadLetterQueue", () => {
  test("matches the snapshot", () => {
    const stack = new cdk.Stack();
    new DeadLetterQueue(stack, "DeadLetterQueue");
```

```
    const template = Template.fromStack(stack);
    expect(template.toJSON()).toMatchSnapshot();
  });
});
```

JavaScript

```
const { Match, Template } = require("aws-cdk-lib/assertions");
const cdk = require("aws-cdk-lib");
const { DeadLetterQueue } = require("../lib/dead-letter-queue");

describe("DeadLetterQueue", () => {
  test("matches the snapshot", () => {
    const stack = new cdk.Stack();
    new DeadLetterQueue(stack, "DeadLetterQueue");

    const template = Template.fromStack(stack);
    expect(template.toJSON()).toMatchSnapshot();
  });
});
```

Python

```
import aws_cdk_lib as cdk
from aws_cdk_lib.assertions import Match, Template

from app.dead_letter_queue import DeadLetterQueue

def snapshot_test():
    stack = cdk.Stack()
    DeadLetterQueue(stack, "DeadLetterQueue")

    template = Template.from_stack(stack)
    assert template.to_json() == snapshot
```

Java

```
package software.amazon.samples.awscdkassertionssamples;

import org.junit.jupiter.api.Test;
```

```
import au.com.origin.snapshots.Expect;
import software.amazon.awscdk.assertions.Match;
import software.amazon.awscdk.assertions.Template;
import software.amazon.awscdk.Stack;

import java.util.Collections;
import java.util.Map;

public class DeadLetterQueueTest {
    @Test
    public void snapshotTest() {
        final Stack stack = new Stack();
        new DeadLetterQueue(stack, "DeadLetterQueue");

        final Template template = Template.fromStack(stack);
        expect.toMatchSnapshot(template.toJSON());
    }
}
```

C#

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

using Amazon.CDK;
using Amazon.CDK.Assertions;
using AwsCdkAssertionSamples;

using ObjectDict = System.Collections.Generic.Dictionary<string, object>;
using StringDict = System.Collections.Generic.Dictionary<string, string>;

namespace TestProject1
{
    [TestClass]
    public class StateMachineStackTest

    [TestClass]
    public class DeadLetterQueueTest
    {
    [TestMethod]
        public void SnapshotTest()
        {
            var stack = new Stack();
            new DeadLetterQueue(stack, "DeadLetterQueue");
```

```
            var template = Template.FromStack(stack);

            return Verifier.Verify(template.ToJSON());
        }
    }
}
```

測試提示

請記住，您的測試將只要他們測試的代碼一樣長，並且它們將一樣經常被讀取和修改。因此，花點時間考慮如何最好地編寫它們是值得的。

不要複製和粘貼設置行或常見斷言。相反，將此邏輯重構為夾具或輔助函數。使用反映每個測試實際測試的好名稱。

不要試圖在一次測試中做太多。最好是，測試應該測試一個且只能測試一種行為。如果您不小心破壞了該行為，則應該只有一個測試失敗，並且測試的名稱應該告訴您失敗的內容。然而，這是一個更理想的努力; 有時你會不可避免地（或無意中）編寫測試多個行為的測試。由於我們已經描述的原因，快照測試尤其容易出現此問題，因此請謹慎使用它們。

安全性 AWS Cloud Development Kit (AWS CDK)

雲端安全是 Amazon Web Services (AWS) 最重視的一環。身為 AWS 客戶，您可以從資料中心和網路架構中獲益，該架構專為滿足對安全性最敏感的組織的需求而打造。安全是 AWS 與您之間共同承擔的責任。共同責任模型 將此描述為雲端本身的安全和雲端內部的安全。

雲的安全性 — AWS 負責保護運行 AWS 雲中提供的所有服務的基礎設施，並為您提供可以安全使用的服務。我們的安全責任是我們的首要任務 AWS，並且我們的安全性有效性是由第三方審計師定期測試和驗證，作為AWS 合規計劃的一部分。

雲端安全性 — 您的責任取決於您使用的 AWS 服務，以及其他因素，包括資料的敏感性、組織的需求，以及適用的法律和法規。

AWS CDK 遵循共同的責任模型，透過其支援的特定 Amazon Web Services 務 (AWS) 服務。如需 AWS 服務安全性資訊，請參閱AWS 服務安全性說明文件頁面和符合性計劃 AWS 遵循工作範圍的 AWS 服務。

主題

- 身分識別與存取管理 AWS Cloud Development Kit (AWS CDK)
- 符合性驗證 AWS Cloud Development Kit (AWS CDK)
- 適應的韌性 AWS Cloud Development Kit (AWS CDK)
- 的基礎架構安全性 AWS Cloud Development Kit (AWS CDK)

身分識別與存取管理 AWS Cloud Development Kit (AWS CDK)

AWS Identity and Access Management (IAM) 可協助系統管理員安全地控制 AWS 資源存取權。 AWS 服務 IAM 管理員控制哪些人可以通過身份驗證 (登入) 和授權 (具有權限) 來使用 AWS 資源。IAM 是您可以使用的 AWS 服務 ，無需額外付費。

物件

您使用 AWS Identity and Access Management (IAM) 的方式會有所不同，具體取決於您在進行的工作 AWS。

服務使用者 — 如果您 AWS 服務 用於執行工作，則管理員會為您提供所需的認證和權限。當您使用更多 AWS 功能來完成工作時，您可能需要其他權限。了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。

服務管理員 — 如果您負責公司的 AWS 資源，您可能擁有完整的 AWS 資源存取權。決定您的服務使用者應該存取哪些 AWS 服務 資源是您的工作。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。

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 AWS 服務存取權的詳細資訊。

使用身分驗證

驗證是您 AWS 使用身分認證登入的方式。您必須以 IAM 使用者身分或假設 IAM 角色進行驗證 (登入 AWS)。 AWS 帳戶根使用者

您可以使用透過 AWS 身分識別來源提供的認證，以聯合身分識別身分登入。 AWS IAM Identity Center (IAM 身分中心) 使用者、貴公司的單一登入身分驗證，以及您的 Google 或 Facebook 登入資料都是聯合身分識別的範例。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。當您使 AWS 用同盟存取時，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。如需有關登入的詳細資訊 AWS，請參閱《AWS 登入 使用指南》 AWS 帳戶中的如何登入您的。

若要 AWS 以程式設計方式存取 AWS CDK，請 AWS 提供軟體開發套件 (SDK) 和命令列介面 (CLI)，以便使用您的認證加密簽署要求。如果您不使用 AWS 工具，則必須自行簽署要求。如需使用建議方法來自行簽署請求的詳細資訊，請參閱 AWS 一般參考 中的第 4 版簽署程序。

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) 來增加帳戶的安全性。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

當您建立時 AWS 帳戶，您會從一個登入身分開始，該身分可完整存取該帳戶中的所有資源 AWS 服務和資源。此身分稱為 AWS 帳戶 root 使用者，可透過使用您用來建立帳戶的電子郵件地址和密碼登入來存取。強烈建議您不要以根使用者處理日常任務。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

最佳作法是要求人類使用者 (包括需要系統管理員存取權的使用者) 使用與身分識別提供者的同盟，才能使用臨時認證 AWS 服務 來存取。

聯合身分識別是來自企業使用者目錄的使用者、Web 身分識別提供者、Identi ty Center 目錄，或使用透過身分識別來源提供的認證進行存取 AWS 服務 的任何使用者。 AWS Directory Service同盟身分存取時 AWS 帳戶，他們會假設角色，而角色則提供臨時認證。

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識別來源中的一組使用者和群組，以便在所有應用程式 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 使用者是您內部的身分，具 AWS 帳戶 有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。如需更多資訊，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。您無法以群組身分簽署。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程序變得更為容易。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。如需進一步了解，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 內部具有特定許可的身分。它類似 IAM 使用者，但未與特定的人員建立關聯。您可以切換角色，在中暫時擔任 IAM 角色。 AWS Management Console 您可以透過呼叫 AWS CLI 或 AWS API 作業或使用自訂 URL 來擔任角色。如需使用角色的方法更多相關資訊，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 – 若要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，並獲授予由角色定義的許可。如需有關聯合角色的相關資訊，請參閱 IAM 使用者指南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，對於某些策略 AWS 服務，您可以將策略直接附加到資源（而不是使用角色作為代理）。若要了解跨帳戶存取權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。
- 跨服務訪問 — 有些 AWS 服務 使用其他 AWS 服務功能。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。
  - 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。如需更多資訊，請參閱 IAM 使用者指南中的建立角色以委派許可給 AWS 服務。
  - 服務連結角色 — 服務連結角色是連結至. AWS 服務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中， AWS 帳戶 且屬於服務所有。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 — 您可以使用 IAM 角色來管理在 EC2 執行個體上執行的應用程式以及發出 AWS CLI 或 AWS API 請求的臨時登入資料。這是在 EC2 執行個體內儲存存取金鑰的較好方式。若要將 AWS 角色指派給 EC2 執行個體並提供給其所有應用程式，請建立連接至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。如需更多資訊，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。

若要了解是否要使用 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。

符合性驗證 AWS Cloud Development Kit (AWS CDK)

AWS CDK 遵循共同的責任模型，透過其支援的特定 Amazon Web Services 務 (AWS) 服務。如需 AWS 服務安全性資訊，請參閱AWS 服務安全性說明文件頁面和符合性計劃 AWS 遵循工作範圍的 AWS 服務。

AWS 服務的安全性和合規性是由第三方審計師評估為多個 AWS 合規計劃的一部分。這些措施包括 SOC、PCI、FedRAMP、HIPAA 等。 AWS 在符合性計劃的 AWS 服務範圍中，提供特定合規方案範圍內經常更新的AWS 服務清單。

協力廠商稽核報告可供您使用 AWS Artifact 下載。如需詳細資訊，請參閱下載中的報告 AWS Artifact。

如需 AWS 規範遵循方案的詳細資訊，請參閱AWS 合規性方案。

使用存取 AWS 服務時，您的 AWS CDK 合規責任取決於資料的敏感度、組織的合規目標以及適用的法律和法規。如果您使用某項 AWS 服務時必須遵守 HIPAA、PCI 或 FedRAMP 等標準，則會 AWS 提供資源以協助：

- 安全性與合規性快速入門指南 — 部署指南，討論架構考量，並提供步驟，以便在上部署以安全性為重點和遵循法規遵循的基準環境。 AWS
- AWS 合規性資源 — 可能適用於您的產業和位置的工作簿和指南集合。
- AWS Config：此服務可評定資源組態與內部實務、業界準則和法規的合規狀態。
- AWS Security Hub— 全面檢視您的安全狀態，可協助 AWS 您檢查您是否符合安全性產業標準和最佳做法。

適應的韌性 AWS Cloud Development Kit (AWS CDK)

Amazon Web Services (AWS) 全球基礎設施是以區域和可用區域為基礎建置的。 AWS

AWS 區域提供多個實體分離和隔離的可用區域，這些區域透過低延遲、高輸送量和高度備援的網路連線。

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。

如需區域和可用區域的相關 AWS 資訊，請參閱AWS 全域基礎結構。

AWS CDK 遵循共同的責任模型，透過其支援的特定 Amazon Web Services 務 (AWS) 服務。如需 AWS 服務安全性資訊，請參閱AWS 服務安全性說明文件頁面和符合性計劃 AWS 遵循工作範圍的 AWS 服務。

的基礎架構安全性 AWS Cloud Development Kit (AWS CDK)

AWS CDK 遵循共同的責任模型，透過其支援的特定 Amazon Web Services 務 (AWS) 服務。如需 AWS 服務安全性資訊，請參閱AWS 服務安全性說明文件頁面和符合性計劃 AWS 遵循工作範圍的 AWS 服務。

常見 AWS CDK 問題的疑難

本主題說明如何疑難排解下列問題 AWS CDK。

更新之後 AWS CDK， AWS CDK 工具組 (CLI) 會報告與 AWS 建構程式庫不相符

AWS CDK Toolkit (提供cdk指令) 的版本必須至少等於主要「 AWS 建構程式庫」模組的版本aws-cdk-lib。該工具包的目的是向後兼容。該工具包的最新 2.x 版本可以與任何 1.x 或 2.x 版本的庫一起使用。因此，我們建議您在全域安裝此元件並保持最新狀態。

```
npm update -g aws-cdk
```

如果您需要使用多個版本的 AWS CDK Toolkit，請在專案資料夾本機中安裝特定版本的工具組。

如果您正在使用 TypeScript 或 JavaScript，則您的專案目錄已包含 CDK Toolkit 的版本化本機副本。

如果您正在使用其他語言，請npm使用安裝 AWS CDK Toolkit，省略標-g誌並指定所需的版本。例如：

```
npm install aws-cdk@2.0
```

若要執行本機安裝的 AWS CDK Toolkit，請使用命令npx aws-cdk而非僅使用cdk。例如：

```
npx aws-cdk deploy MyStack
```

npx aws-cdk執行「 AWS CDK 工具組」的本機版本 (如果存在)。當項目沒有本地安裝時，它會退回到全局版本。您可能會發現設置 shell 別名以確保cdk始終以這種方式調用它很方便。

macOS/Linux

```
alias cdk="npx aws-cdk"
```

Windows

```
doskey cdk=npx aws-cdk $*
```

(返回列表)

部署 AWS CDK 堆疊時，我收到**NoSuchBucket**錯誤訊息

您的 AWS 環境尚未啟動載入，因此在部署期間沒有 Amazon S3 儲存貯體來保留資源。您可以使用下列指令建立暫存貯體和其他必要資源：

```
cdk bootstrap aws://ACCOUNT-NUMBER/REGION
```

為了避免產生非預期的 AWS 費用， AWS CDK 不會自動引導任何環境。您必須明確地啟動要部署的每個環境。

根據預設，啟動程序資源是在當前 AWS CDK 應用程序中由堆棧使用的地區中創建的。或者，它們是使用該配置文件的帳戶在您本地 AWS 配置文件中指定的區域（由設置`aws configure`）中創建的。您可以在命令行上指定不同的帳戶和區域，如下所示。（如果您不在應用程序的目錄中，則必須指定帳戶和地區。）

```
cdk bootstrap aws://ACCOUNT-NUMBER/REGION
```

如需詳細資訊，請參閱 the section called “引導”。

(返回列表)

部署 AWS CDK 堆疊時，我收到一**forbidden: null**則訊息

您正在部署需要啟動程序資源的堆疊，但使用的是缺少寫入權限的 IAM 角色或帳戶。（當部署包含資產的堆棧或合成大於 50K 的 AWS CloudFormation 模板時，會使用臨時存儲桶。）使用具有對錯誤訊息中提到的值區執`s3:*`行動作之權限的帳號或角色。

(返回列表)

合成 AWS CDK 堆棧時，我收到消息 **--app is required either in command-line, in cdk.json or in ~/.cdk.json**

此消息通常意味著您在發出問題時不在 AWS CDK 項目的主目錄中`cdk synth`。由`cdk init`命令創建的此目錄`cdk.json`中的文件包含運行（並從而合成）AWS CDK 應用程序所需的命令行。例如，對於 TypeScript 應用程序，默認值`cdk.json`看起來像這樣：

```
{
  "app": "npx ts-node bin/my-cdk-app.ts"
}
```

我們建議您只在專案的主目錄中發出`cdk`指令，以便 AWS CDK 工具組可以`cdk.json`在該處找到並成功執行您的應用程式。

如果由於某種原因這不實際，則 AWS CDK 工具包會在其他兩個位置查找應用程序的命令行：

- `cdk.json`在您的主目錄
- 在`cdk synth`命令本身上使用該`-a`選項

例如，您可以按照以下方式從 TypeScript 應用程序合成堆棧。

```
cdk synth --app "npx ts-node my-cdk-app.ts" MyStack
```

(返回列表)

當合成 AWS CDK 堆棧時，我收到一個錯誤，因為 AWS CloudFormation 模板包含太多資源

會 AWS CDK 產生並部署 AWS CloudFormation 範本。 AWS CloudFormation 對堆棧可以包含的資源數量有硬性限制。使用時 AWS CDK，您可以比預期的更快速地執行此限制。

> **Note**
>
> 在撰寫本文時， AWS CloudFormation 資源限制為 500。請參閱AWS CloudFormation 配額以瞭解目前的資源限制。

Con AWS struct Library 的較高層級的意向型結構會自動佈建記錄、金鑰管理、授權及其他用途所需的任何輔助資源。例如，授與一個資源存取權給另一個資源會產生相關服務通訊所需的任何 IAM 物件。

根據我們的經驗，實際使用以意向為基礎的建構會導致每個建構產生 1—5 個 AWS CloudFormation 資源，儘管這可能會有所不同。對於無伺服器應用程式，每個 API 端點為 5 到 8 個 AWS 資源。

模式代表較高層級的抽象，可讓您以更少的程式碼定義更多 AWS 資源。例如the section called "ECS"，中的 AWS CDK 代碼生成 50 多個 AWS CloudFormation 資源，而只定義三個構造！

超過資 AWS CloudFormation 源限制是 AWS CloudFormation 合成過程中的錯誤。如果您的堆棧超過限制的 80%，則會 AWS CDK 發出警告。您可以通過在堆棧上設置`maxResources`屬性來使用不同的限制，也可以通過將其設置`maxResources`為 0 來禁用驗證。

> **Tip**
>
> 您可以使用以下實用程序腳本獲得合成輸出中資源的精確計數。（由於每個 AWS CDK 開發人員都需要 Node.js，因此腳本是用 JavaScript.）
>
> ```
> // rescount.js - count the resources defined in a stack
> // invoke with: node rescount.js <path-to-stack-json>
> // e.g. node rescount.js cdk.out/MyStack.template.json
>
> import * as fs from 'fs';
> const path = process.argv[2];
>
> if (path) fs.readFile(path, 'utf8', function(err, contents) {
>   console.log(err ? `${err}` :
>   `${Object.keys(JSON.parse(contents).Resources).length} resources defined in
>  ${path}`);
> }); else console.log("Please specify the path to the stack's output .json
>  file");
> ```

當堆疊的資源計數接近限制時，請考慮重新架構以減少堆疊包含的資源數量：例如，結合某些 Lambda 函數，或將堆疊分解為多個堆疊。CDK 支持堆棧之間的引用，因此您可以以對您最有意義的任何方式將應用程序的功能分成不同的堆棧。

> **Note**
>
> AWS CloudFormation 專家經常建議使用嵌套堆棧作為資源限制的解決方案。通過`NestedStack`構造 AWS CDK 支持這種方法。

(返回列表)

我為 Auto Scaling 群組或 VPC 指定了三個 (或更多) 可用區域，但僅部署在兩個區域

若要取得您要求的可用區域數目，請在堆疊的內env容中指定帳戶和區域。如果您未同時指定兩者，依預設 AWS CDK，會將堆疊合成為與環境無關的情況。然後，您可以使用將堆疊部署到特定區域 AWS CloudFormation。由於某些區域只有兩個可用區域，因此不受環境限制的範本使用不超過兩個。

> **Note**
>
> 在過去，區域偶爾會啟動只有一個可用區域。與環境無關的 AWS CDK 堆疊無法部署到此類區域。然而，在撰寫本文時，所有 AWS 區域至少都有兩個 AZ。

您可以通過覆蓋堆棧的 `availablilityZones`（Python:`availability_zones`）屬性來顯式指定要使用的區域來更改此行為。

如需在合成時指定堆疊帳戶和區域的詳細資訊，同時保留部署到任何區域的彈性，請參閱the section called “環境”。

(返回列表)

當我發出問題時，我的 S3 儲存貯體、DynamoDB 資料表或其他資源並未刪除 **`cdk destroy`**

默認情況下，可以包含用戶數據的資源具有`removalPolicy`（Python:`removal_policy`）屬性`RETAIN`，並且在堆棧銷毀時不會刪除該資源。相反，該資源是從堆棧中孤立的。然後，您必須在堆棧銷毀後手動刪除資源。在您這麼做之前，重新部署堆疊會失敗。這是因為部署期間所建立的新資源名稱與孤立資源的名稱衝突。

如果您將資源的移除政策設定為`DESTROY`，則當堆疊銷毀時，該資源將會被刪除。

TypeScript

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as s3 from 'aws-cdk-lib/aws-s3';

export class CdkTestStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    const bucket = new s3.Bucket(this, 'Bucket', {
      removalPolicy: cdk.RemovalPolicy.DESTROY,
    });
  }
```

```
}
```

JavaScript

```
const cdk = require('aws-cdk-lib');
const s3 = require('aws-cdk-lib/aws-s3');

class CdkTestStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    const bucket = new s3.Bucket(this, 'Bucket', {
      removalPolicy: cdk.RemovalPolicy.DESTROY
    });
  }
}

module.exports = { CdkTestStack }
```

Python

```
import aws_cdk as cdk
from constructs import Construct
import aws_cdk.aws_s3 as s3

class CdkTestStack(cdk.stack):
    def __init__(self, scope: Construct, id: str, **kwargs):
        super().__init__(scope, id, **kwargs)

        bucket = s3.Bucket(self, "Bucket",
            removal_policy=cdk.RemovalPolicy.DESTROY)
```

Java

```
software.amazon.awscdk.*;
import software.amazon.awscdk.services.s3.*;
import software.constructs;

public class CdkTestStack extends Stack {
    public CdkTestStack(final Construct scope, final String id) {
        this(scope, id, null);
    }
```

```
    public CdkTestStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        Bucket.Builder.create(this, "Bucket")
                .removalPolicy(RemovalPolicy.DESTROY).build();
    }
}
```

C#

```
using Amazon.CDK;
using Amazon.CDK.AWS.S3;

public CdkTestStack(Construct scope, string id, IStackProps props) : base(scope, id,
 props)
{
    new Bucket(this, "Bucket", new BucketProps {
        RemovalPolicy = RemovalPolicy.DESTROY
    });
}
```

> **Note**
>
> AWS CloudFormation 無法刪除非空的 Amazon S3 儲存貯體。如果您將 Amazon S3 儲存貯體的移除政策設為DESTROY，且其中包含資料，則嘗試銷毀堆疊將會失敗，因為無法刪除儲存貯體。您可以先 AWS CDK 刪除值區中的物件，然後再嘗試將值區的 `autoDeleteObjects` prop 設定為將其銷毀`true`。

(返回列表)

和 jsii 的 AWS CDK 開啟 PGP 金鑰

本主題包含和 jsii 的 AWS CDK 目前和歷史 OpenPGP 金鑰。

目前的按鍵

這些金鑰應該用來驗證 AWS CDK 和 jsii 的目前發行版本。

AWS CDK 開啟 PGP 金鑰

金鑰識別碼：	0X42B9CF2286CD987A
類型：	RSA
尺寸:	4096/4096
已建立：	2022-07-05
到期：	2026-07-04
使用者識別碼：	AWS Cloud Development Kit < aws-cdk@amazon.com >
按鍵指紋：	69B5 2D5B A295 1D11 法 65 413B 42B9 CF22 86CD 987A

選擇「複製」圖示以複製下列 OpenPGP 金鑰：

```
-----BEGIN PGP PUBLIC KEY BLOCK-----

mQINBGLEgOsBEADCoAMwvnszMLybJ+AD9cHhVyX6+rYIUEXYSgVnfkl6Z7qawIwv
wgd/a5fEs9Kiz2XJmfwS9Rxb4d+0+Y1ls1A+gnpw9FMLcZlqkC9KLnS2MqvuXWLB
t3z4kjZaL9fQ+58PoD4gy/M2hDg6gZrYqR3gtJuw8FcFpb/1KlkzRQUM8eAMFxf2
TyfjP0V0tSHwcB+84oushX7fUXVMyc3+OHsCPOe/WBFMIlWgKA+n33JKIQlUUC8f
kCWBAsAFupil0lCveT6mZu5slNRlc1I3iBLjUZ3/MtLygfqAMKwUVXeawtDvRIZe
PrAFc2NyODEhly2JG6K0FW7eIcvBqR3rg8U49t9Y74ELTM0kKnfd+flvq35xWqQC
0zghnk3kDppRTN4zWBgTKiCMxBcsHXGOoGn57t4B9VY9Zy3vkeySigeiwl/Tw9nJ
PE0SRnwEc/HnjTTfX+GTG1aQVE0xSVyZ4m5ymRNCu6+rNH8lKwo5FujlXJ+GXPkp
```

```
qT+Lx6Ix/Ny7PaoweWxwtZUkLRS4pWUsg0yotZrGyIbS+X3yMEG8WBTFI9hf6HTq
0ryfi5/TsBrdrGKqWB99EC9xYEGgtHp4fKO5X0ynOagVOhf0jSe8t1uyuJPGb2Gc
MQagSys5xMhdG/ZnEY4Cb+JDtH/4jc3tca0+4Z5RQ7kF9IhCncFtrbjJbwARAQAB
tC5BV1MgQ2xvdWQgRGV2ZWxvcG1lbnQgS2l0IDxhd3MtY2RrQGFtYXpvbi5jb20+
iQI/BBMBAgApBQJixIDrAhsvBQkHhM4ABwsJCAcDAgEGFQgCCQoLBBYCAwECHgEC
F4AACgkQQrnPIobNmHo2qg//Zt9p/kN1DevflzxWKouUX0AS7UmUtRYXu5k/EEbu
wkYNHpUr7+lZ+Me5YyjcIpt6UwuG9cW4SvwuxIfXucyKAWiwEbydCQauvnrYDxDa
J6Yr/ntk7Sii6An9re99qic3IsvX+xlUXh+qJ/34ooP/1PHziCMqykvW/DwAIyhx
2qvTXy+9+OlOWSUbhkCnNz5XKb4XQGq73DqalZX1nH4dG6fckZmYRX+dpw2njfTw
ZLdZ7bkrfiL84FI4A21RfSbEU4s4ngiV17lZ9ivilBKTbDv3da7+yc919M7C5N4J
yrlxvtyYNDoqKAD2WYZAnpEbG/shu3f56RyOJd56tXGwl9nKPh+F9y+379XthSwA
xZTURFtjWf7wWHaDZadU0DKi+Oeeszjg2f/VJaGmmS8PIg7q6GiSHHpqHqNvACHm
ZXMw12QFd3qt3xu0JMmE11ZC5VBgblwpkQTrO04Sq1rOpJwXI9ODMS/ZEhAIoYmT
OR7ouknlAx6mj9fwpavWDAAJHLdVUMYBZTXiQYFzDvx51ivvTRWkB1zTJcFdqShY
B37+Jz2jLDNdMrcHk2yfVp/VvfbxKcexg8wEwrrtQUslTUenl5jBZJouoz/wW81s
Y4U1nCPCdTK5/C7JCKzR2gVnCpe6uaxAWkkM2feQhjqJZkTC4cFVgBT+4M6WcT1r
yq4=
=ahbs
-----END PGP PUBLIC KEY BLOCK-----
```

使用開啟 PGP 金鑰

金鑰識別碼：	0056C4E15DA3D8
類型：	RSA
尺寸:	4096/4096
已建立：	2022-07-05
到期：	2026-07-04
使用者識別碼：	AWS 集成工程團隊 < aws-jsii@amazon.com >
按鍵指紋：	1E07 31 日 4 57E5 二月 87 號 530A 056C 4E15 守護者

選擇「複製」圖示以複製下列 OpenPGP 金鑰：

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
```

```
mQINBGLEgOkBEAD27EPVG9g2mHQ3+M6tF6le+tfhARJ2EV7m7NKIrTdSlCZATLWn
AVLlxG1unW34NlkKZbcbR86gAxRnnAhuEhPuloU/S5wAqPGbRiFl58YjYZDNJw6U
1SSMpE4O1sfjxv9yAbiRihLYtvksyHHZmaDhYner2aK1PdeWu+BKq/tjfm3Yzsd2
uuVEduJ72YoQk/29dEiGOHfT+2kUKxUX+0tJSJ9MGlEf4NtQE4WLzrT6Xqb2SG4+
alIiIVxIEi0XKDn7n8ZLjFwfJwOYxVYLtEUkqFWM8e8vgoc9/nYc+vDXZVED2g3Z
FWrwSnDSXbQpnMa2cLhD4xLpDHUS3i2p7r3dkJQGLo/5JGOopLibrOAbYZ72izhu
H/TuPFogSz0mNFPglrWdnLF04UIjIq420+06V4WQZC9n55Zjcbki/OhnC3B9pAdU
tiy8zg070bWq45dPGf5STkPPn7G8A2zmKefyO51iLIi26ZzW78siB+FvcGRhdg25
39sHJ1cmrTeC+B+k4KeV5sQ/m3UucimrZnk1xdaiVp8mWzRqWb8bB6Rs8K9RMrMV
tFBOKOBAT2QxOQtRGAantVgm193E1T1cmNpD0FKAKkDdPs64rKBEwFiHxccXHbah
eMd1weVwn3AKFD6uAm8ZRMV+dyssfcQxqpo/kfT1XpA6cQeOmGDOcKBfdwARAQAB
tCNBV1MgSlNJSSBUZWFtIDxhd3MtanNpaUBhbWF6b24uY29tPokCPwQTAQIAKQUC
YsSA6QIbLwUJB4TOAAcLCQgHAwIBBhUIAgkKCwQWAgMBAh4BAheAAAoJEAVsThXa
49jZjU4QANoyqOJUT4gRrXshE3N0mW5Ad4i8Ke09GA62HyvTtfbsA+2nkNVGJpXm
sFMzdaFO95Q65RkLS9vW4nhhjXBEc2XYNCt2AnARudA/41ykjDPwU112z9ZTB9he
y4ItIeNGpHvMWr51fihl0y2nkpODOBeiv44jscLbHyOmZfki1f5fuIu2U2IbUGK3
5FtYyeHcgRHnpYkzLuzK4PfayOywqQPJ7M9DWrHf+v5Cu4ZCZDOIKfzF+ew7MWwc
6KaoWHCYbFpX8jxFppbGsSFOQ8Sl2quoP0TLz9Wsq70Khi6C2P8JI6lm0HRLO+1M
jFbQxNOwAcN3k4HSwunAjXBlmT/6oc1RsdBdpXBaZ2AWseIXwSYZqNXp+5L179uZ
vSiD3DSSUqLJbdQRVOsJi3/87V5QU59byq2dToHveRjtSbVnK0TkTx9Zlgkcpjv M
BwHNqWhratV6af2Upjq2YQ0fdSB42f3pgopInxNJPMvlAb+cCfr0Pfwu7ge7UooQ
WHTxbpCvwtn/HNctMGpWscO02WsWgoYVjnVFay/XphE77pQ9rRUkhMe6VKXfxj/n
OCZJKrydluIIwR8vvONNqO+QwZ1xDEhO7MaSZlOm1AuUZIXFPgaWQkPZHKiiwFA/
QWnL/+shuRtMH2geTjkev198Jgb5HyXFm4SyYtZferQROyliEhik
=BuGv
-----END PGP PUBLIC KEY BLOCK-----
```

歷史密鑰

這些金鑰可用來驗證在 2022-07-05 之前發行的 AWS CDK 和 jsii 的發行版本。

> **Important**
>
> 新的金鑰會在先前的金鑰到期之前建立。因此，在任何給定的時刻，多個鍵可能是有效的。金鑰是用來簽署人工因素的建立日開始，因此請在金鑰有效性重疊的情況下使用最近發出較多的金鑰。

AWS CDK 開放式全球通用金鑰 (2022-04-07)

> **Note**
>
> 在 2022-07-05 之後，此金鑰並未用來簽署 AWS CDK 文物。

金鑰識別碼：	015584281F44A3C3
類型：	RSA
尺寸:	4096/4096
已建立：	2022-04-07
到期：	2026-04-06
使用者識別碼：	AWS Cloud Development Kit < aws-cdk@amazon.com >
按鍵指紋：	EAE1 1A24 82B0 AA86 456E 6C67 0155 8428 1F44 A3C3

選擇「複製」圖示以複製下列 OpenPGP 金鑰：

```
-----BEGIN PGP PUBLIC KEY BLOCK-----

mQINBGJPLgUBEADtlR5jQtxtBmROQvmWlPOViqqnJNhk0dULc3tXnq8NS/l6X81r
wHk+/CHG5kBunwvM0qaqLFRC6z9NnnNDxEHcTi47n+OAjWyDM6unxxWOPz8Dfaps
Uq/ZWa4by292ZeqRC9Ir2wdrizb69JbRjeshBwlJDAS/qtqCAqBRH/f7Zw7QSD6/
XTxyIy+KOVjZwFPFNHMRQ/NmgUc/Rfxsa0pUjk1YAj/AkvQlwwD8DEnASoBhO0DP
QonZxouLqIpgp4LsGo8TZdQv30ocIj0C9DuYUiUXWlCPlYPgDj6IWf3rgpMQ6nB9
wC9lx4t/L3Zg1HUD52y8aymndmbdHVn90mzlNg4XWyc58rioYrEk57YwbDnea/Kk
Hv4kVHZRfJ4/OFPyqs5ex1X3X6rb07VvA1tfLgPywO9XF2Xws8YWOWcEobaWTcnb
AzyVC6wKya8rEQzxkYJ6UkJlhDB6g6bZwIpsI2zlimG+kSBsyFvE2oRYMS0cXPqU
o+tX0+4TvxEyW3RrUQzQHIpqXrb0X1Q8Z2idPn5dwsipDEa4gsFXtrSXmbB/0Cee
eJVvKWQAsxol3+NE9L/yozq3cz5PWh0SSbmCLRcs78lMJ23MmzbMWV7BWC9DXdY+
TywY5IkDUPjGCKlD8VlrI3TgC222bH6qaua6LYCiTtRtvpDYuJNAlUjhawARAQAB
tC5BV1MgQ2xvdWQgRGV2ZWxvcG1lbnQgS2l0IDxhd3MtY2RrQGFtYXpvbi5jb20+
iQI/BBMBAgApBQJiTy4FAhsvBQkHhM4ABwsJCAcDAgEGFQgCCQoLBBYCAwECHgEC
```

```
F4AACgkQAVWEKB9Eo8NpbxAAiBF0kR/lVw3vuam60mk4l0iGMVsP8Xq6g/buzbEO
2MEB4Ftk04qOnoa+93S0ZiLR9PqxrwsGSp4ADDX3Vtc4uxwzUlKUi1ywEhQ1cwyL
YHQI3Hd75K1J81ozMEu6qJH+yF0TtTDZMeZHtH/XvuIYJW3Lx4o5ZFlsEegFPAgX
YCCpUS+k9qC6M8g2VjcltQJpyjGswsKm6FWaKHW+B9dfjdOHlImB9E2jaknJ8eoY
zb9zHgFANluMzpZ6rYVSiCuXiEgYmazQWCvlPcMOP7nX+1hq1z11LMqeSnfE09gX
H+OYho9cMEJkb1dzx1H9MRpylFIn9tL+2iCp4UPJjnqi6uawWyLZ2tp4G11haqQq
1yAh69u233I8GZKFUySzjHwH5qWGRgBTjrZ6FdcjSS2w/wMkVKuCPkWtdvo/TJrm
msCd1Reye8SEKYqrs0ujTwmlvWmUZmO06AdUjo1kWiBKeslTJrWEuG7Yk4pFOoA4
dsaq83gxpOJNVCh6M3y4DLNrvl7dhF95NwTWMROPj2otw7NIjF4/cdzve2+P7YNN
pVAtyCtTJdD3eZbQPVaL3T8cf1VGqt6++pnLGnWJ0+X3TyvfmTohdJvN3TE+tq7A
7cprDX/q9c56HaXdJzVpxEzuf/YC+JuYKeHwsX3QouDhyRg3PsigdZES/02Wr8so
l6U=
=MQI4
-----END PGP PUBLIC KEY BLOCK-----
```

使用開放式全球通用證券金鑰 (2022-04-07)

> **Note**
>
> 在 2022-07 年 5 月 5 日之後，這個機碼並未用來簽署 JSII 文物。

金鑰識別碼：	0X985F5B974B79356
類型：	RSA
尺寸:	4096/4096
已建立：	2022-04-07
到期：	2026-04-06
使用者識別碼：	AWS 集成工程團隊 < aws-jsii@amazon.com >
按鍵指紋：	35A7 1785 8FA6 282D C5AC CD95 985F 5BC9 74B7 9356

選擇「複製」圖示以複製下列 OpenPGP 金鑰：

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
```

```
mQINBGJPLewBEADHH4TXup/gOlHrKDZRbj8MvsMTdM6eDteA6/c32UYV/YsK9rDA
jN8Jv/xlfosOebcHrfnFpHF9VTkmjuOpN695XdwMrW/NvlEPISTGEJf21x6ZTQ2r
1xWFYzC3sl3FZmvj9XAXTmygdv+XM3TqsFgZeCaBkZVdiLbQf+FhYrovUlgotb5D
YiCQI3ofV5QTE+141jhO5Pkd3ZIoBG+P826LaT8NXhwS0o1XqVk39DCZNoFshNmR
WFZpkVCTHyv5ZhVey1NWXnD8opO375htGNV4AeSmSIH9YkURD1g5F+2t7RiosKFo
kJrfPmUjhHn8IFpReGc8qmMMZX0WaV3t+VAWfOHGGyrXDfQ4xz1VCot75C2+qypM
+qhwOAOOP0zA7CfI96ULZzSH/j8HuQk3O0DsUCybpMuKEazEMxP3tgGtRerwDaFG
jQvAlK8Rbq3v8buBI6YJuXTwSzJE8KLjleUiTFumE6WP4rsAvlP/5rBvubeMfa3n
NIMm5Rkl36Z+jt3e2Z2ZqWDPpBRta8m7QHccrZhkvqu3YC3Gl6kdnm4Vio3Xfpg2
qtWhIQutQ6DmItewV+weQHas3hl88RPJtSrfWWIIMkpbF7Y4vbX9xcnsYCLlp2Mz
tWbbnU+EWATNSsufml/Kdnu9iEEuLmeovE11I69nwjNOq9P+GJ3r/FUb2wARAQAB
tCNBV1MgSlNJSSBUZWFtIDxhd3MtanNpaUBhbWF6b24uY29tPokCPwQTAQIAKQUC
Yk8t7AIbLwUJB4TOAAcLCQgHAwIBBhUIAgkKCwQWAgMBAh4BAheAAAoJEJhfW8l0
t5NWo64P/2y7gcMRylLLW/wbrCjton2O4+YRocwQxKm1cBml9FVDUR5967YczNuu
EwEOfH/Pu3UAlrBfKAfxPNhKchLwYiOBNh2Wk5UUxRcldNHTLb5jn5gxCeWNAsl/
Tc46qY+ObdBMdOf2Vu33UCOg83WLbg1bfBoA8Bm1cd0XObtLGucu606EBt1dBrKq
9UTcbJfuGivY2Xjy5r4kEiMHBoLKcFrSo2Mm7VtYlE4Mabjyj9+orqUio7qxOl60
aa7Psa6rMvs1Ip9IOrAdG7o5Y29tQpeINH0R1/u47BrlTEAgG63Dfy49w2h/1g0G
c9KPXVuN55OWRIu0hsiySDMk/2ERsF348TU3NURZltnCOxp6pHlbPJIxRVTNa9Cn
f8tbLB3y3HfA80516g+qwNYIYiqksDdV2bz+VbvmCWcO+FellDZli831gyMGa5JJ
rq7d0lEr6nqjcnKiVwItTQXyFYmKTAXweQtVC72g1sd3oZIyqa7T8pvhWpKXxoJV
WP+OPBhGg/JEVC9sguhuv53tzVwayrNwb54JxJsD2nemfhQm1Wyvb2bPTEaJ3mrv
mhPUvXZj/I9rgsEq3L/sm2Xjy09nra4o3oe3bhEL8nOj11wkIodi17VaGP0y+H3s
I5zB5UztS6dy+cH+J7DoRaxzVzq7qtH/ZY2quClt30wwqDHUX1ef
=+iYX
-----END PGP PUBLIC KEY BLOCK-----
```

AWS CDK 開放式全球通用金鑰 (2018-06-19)

金鑰識別碼：	0x0566A784E17F3870
類型：	RSA
尺寸:	4096/4096
已建立：	2018-06-19
到期：	2022-06-18
使用者識別碼：	AWS CDK 團隊 < aws-cdk@amazon.com >

按鍵指紋：	E88B E3B6 F0B1 E350 9E36 4F96 0566 A784 E17F 3870

選擇「複製」圖示以複製下列 OpenPGP 金鑰：

```
-----BEGIN PGP PUBLIC KEY BLOCK-----

mQINBFsovE8BEADEFVCHeAVPvoQgsjVu9FPUczxy9P+2zGIT/MLI3/vPLiULQwRy
IN2oxyBNDtcDToNa/fTkW3Ev0NTP4V1h+uBoKDZD/p+dTmSDRfByECMI0sGZ3UsG
Ohhyl2Of44s0sL8gdLtDnqSRLf+ZrfT3gpgUnplW7VitkwLxr78jDpW4QD8p8dZ9
WNm3JgB55jyPgaJKqA1Ln4Vduni/1XkrG42nxrrU71uUdZPvPZ2ELLJa6n0/raG8
jq3le+xQh45gAIs6PGaAgy7jAsfbwkGTBHjjujITAY1DwvQH5iS31OaCM9n4JNpc
xGZeJAVYTLilznf2QtS/a50t+ZOmpq67Ssp2j6qYpiumm0Lo9q3K/R4/yF0FZ8SL
1TuNX0ecXEptiMVUfTiqrLsANg18EPtLZZOYW+ZkbcVytkDpiqj7bMwA7mI7zGCJ
1gjaTbcEmOmVdQYS1G6ZptwbTtvrgA6AfnZxX1HUxLRQ7tT/wvRtABfbQKAh85Ff
a3U9W4oC3c1MP5IyhNV1Wo8Zm0flZiZc0iZnojTtSG6UbcxNNL4Q8e08FWjhungj
yxSsIBnQO1Aeo1N4BbzlI+n9iaXVDUN7Kz1QEyS4PNpjvUyrUiQ+a9C5sRA7WP+x
IEOaBBGpoAXB3oLsdTNO6AcwcDd9+r2NlXlhWC4/uH2YHQUIegPqHmPWxwARAQAB
tCFBV1MgQ0RLIFRlYW0gPGF3cy1jZGtAYW1hem9uLmNvbT6JAj8EEwEIACkFAlso
vE8CGy8FCQeEzgAHCwkIBwMCAQYVCAIJCgsEFgIDAQIeAQIXgAAKCRAFZqeE4X84
cLGxD/0XHnhoR2xvz38GM8HQlwlZy9W1wVhQKmNDQUavw8Zx7+iRR3m7nq3xM7Qq
BDbcbKSg1lVLSBQ6H2V6vRpysOhkPSH1nN2dO8DtvSKIPcxK48+1x7lmO+ksSs/+
oo1UvOmTDaRzOitYh3kOGXHHXk/l11GtF2FGQzYssX5iM4PHcjBsK1unThs56IMh
OJeZezEYzBaskTu/ytRJ236bPP2kZIEXfzAvhmTytuXWUXEftxOxc6fIAcYiKTha
aofG7WyR+Fvb1j5gNLcbY552QMxa23NZd5cSZH7468WEW1SGJ3AdLA7k5xvsPPOC
2YvQFD+vUOZ1JJuu6B5rHkiEMhRTLklkvqXEShTxuXiCp7iTOo6TBCmrWAT4eQr7
htLmqlXrgKi8qPkWmRdXXG+MQBzI/UyZq2q8KC6cx2md1PhANmeefhiM7FZZfeNM
WLonWfh8gVCsNH5h8WJ9fxsQCADd3Xxx3NelS2zDYBPRoaqZEEBbgUP6LnWFprA2
EkSlc/RoDqZCpBGgcoy1FFWvV/ZLgNU6OTQlYH6oYOWiylSJnaTDyurrktsxJI6d
4gdsFb6tqwTGecuUPvvZaEuvhWExLxAebhu780FdAPXgVTX+YCLI2zf+dWQvkFQf
80RE7ayn7BsiaLzFBVux/zz/WgvudsZX18r8tDiVQBL51ORmqw==
=0wuQ
-----END PGP PUBLIC KEY BLOCK-----
```

開放式全球通用證券金鑰 (2018-08-06)

金鑰識別碼：	0x1C7ACE4CB2A1B93A
類型：	RSA

尺寸:	4096/4096
已建立：	2018-08-06
到期：	2022-08-05
使用者識別碼：	AWS 集成工程團隊 < aws-jsii@amazon.com >
按鍵指紋：	85 EF 6522 4CE2 1E8C 72 分貝 28 分貝 1C7A CE4C B93A

選擇「複製」圖示以複製下列 OpenPGP 金鑰：

```
-----BEGIN PGP PUBLIC KEY BLOCK-----

mQINBFtoSs0BEAD6WweLD0B26h0F7Jo9iR6tVQ4PgQBK1Va5H/eP+A2Iqw79UyxZ
WNzHYhzQ5MjYYI1SgcPavXy5/LV1N8HJ7QzyKszybnLYpNTLPYArWE8ZM9ZmjvIR
p1GzwnVBGQfoOlxyeutE9T5ZkAn45dTS5jlno4unji4gHjnwXKf2nP1APU2CZfdK
8vDpLOgj9LeeGlerYNbx+7xtY/I+csFIQvK09FPLSNMJQLlkBhY0r6Rt9ZQG+653
tJn+AUjyM237w0UIX1IqyYc5IONXu8HklPGu0NYuX9AY/63Ak2Cyfj0w/PZlvueQ
noQNM3j0nkOEsTOEXCyaLQw9iBKpxvLnm5RjMSODDCkj8c9uu0LHr7J4EOtgt2S1
pem7Y/c/N+/Z+Ksg9fP8fVTfYwRPvdI1x2sCiRDfLoQSG9tdrN5VwPFi4sGV04sI
x7Al8Vf/OBjAGZrDaJgM/gVvb9SKAQUA6t3ofeP14gDrS0eYodEXZ+lamnxFglxF
Sn8NRC4JFNmkXSUaTNGUdFf//F0D69PRNT8CnFfmniGj0CphN5037PCA2LC/Buq2
3+K6mTPkCcCHYPC/SwItp/xIDAQsGuDc1i1SfDYXrjsK7uOuwC5jLA9X6wZ/jgXQ
4umRRJBAV1aW8b1+yfaYYCO2AfXXO6caObv8IvH7Pc4leC2DoqylD3KklQARAQAB
tCNBV1MgSlNJSSBUZWFtIDxhd3MtanNpaUBhbWF6b24uY29tPokCPwQTAQgAKQUC
W2hKzQIbLwUJB4TOAAcLCQgHAwIBBhUIAgkKCwQWAgMBAh4BAheAAAoJEBx6zkyy
obk6B34P/iNb5QjKyhT0glZiq1wK7tuDDRpR6fC/sp6Jd/GhaNjO4BzlDbUPSjW5
950VT+qwaHXbIma/QVP7EIRztfwWy7m8eOodjpiu7JyJprhwG9nocXiNsLADcMoH
BvabkDRWXWIWSurq2wbcFMlTVwxjHPIQs6kt2oojpzP985CDS/KTzyjow6/gfMim
DLdhSSbDUM34STEgew79L2sQzL7cvM/N59k+AGyEMHZDXHkEw/Bge5Ovz50YOnsp
lisH4BzPRIw7uWqPlkVPzJKwMuo2WvMjDfgbYLbyjfvs5mqDxT2GTwAx/rd2taU6
iSqP0QmLM54BtTVVdoVXZSmJyTmXAAGlITq8ECZ/coUW9K2pUSgVuWyu63lktFP6
MyCQYRmXPh9aSd4+ielteXM9Y39snlyLgEJBhMxioZXVO2oszwluPuhPoAp4ekwj
/umVsBf6As6PoAchg7Qzr+lRZGmV9YTJOgDn2Z7jf/7tOes0g/mdiXTQMSGtp/Fp
ggnifTBx3iXkrQhqHlwtam8XTHGHy3MvX17ZslNuB8Pjh+07hhCxv0VUVZPUHJqJ
ZsLa398LMteQ8UMxwJ3t06jwDWAd7mbr2tatIilLHtWWBFoCwBh1XLe/03ENCpDp
njZ7OsBsBK2nVVcN0H2v5ey0T1yE93o6r7xOwCwBiVp5skTCRUob
=2Tag
-----END PGP PUBLIC KEY BLOCK-----
```

AWS CDK 開發者指南歷史

如需有關版本的資訊，請參閱 AWS CDK 版本 大約每週更新一次。 AWS CDK 維護版本可能會在每週發行版本之間發行，以解決重大問題。每個版本都包含一個匹配的 AWS CDK 工具包（CDK CLI），AWS 構建庫和 API 參考。本指南的更新通常不會與 AWS CDK 發行版本同步。

> **Note**
>
> 下表代表重要的文件里程碑。我們會持續修正錯誤並改善內容。

變更	描述	日期
新增 CDK 移轉功能的說明文件	使用命 AWS CDK CLI`cdk migrate`令將已部署的 AWS 資源、已部署的 AWS CloudFormation 堆疊和本機 AWS CloudFormation 範本移轉至 AWS CDK。如需詳細資訊，請參閱移轉至 AWS CDK。	2024年2月2日
IAM 最佳實務更新	更新了指南以符合 IAM 最佳實務。如需更多詳細資訊，請參閱 IAM 中的安全最佳實務。	2023 年 3 月 23 日
文件 `cdk.json`	新增`cdk.json`組態值的文件。	2022 年 4 月 20 日
相依性管理	使用新增有關管理相依性的主題 AWS CDK。	2022 年 4 月 7 日
從 Java 實例中刪除雙大括號	`Map.of`在整個過程中用 Java 9 替換此反模式。	2022 年 3 月 9 日

AWS CDK 第 2 版本	AWS CDK 開發人員指南的第 2 版已發行。CDK 第 1 版的文件歷史記錄。	2021年12月4 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。