



开发人员指南

的托管集成 AWS IoT Device Management



的托管集成 AWS IoT Device Management: 开发人员指南

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon 的商标和商业外观不得用于任何非 Amazon 的商品或服务，也不得以任何可能引起客户混淆、贬低或诋毁 Amazon 的方式使用。所有非 Amazon 拥有的其他商标均为各自所有者的财产，这些所有者可能附属于 Amazon、与 Amazon 有关联或由 Amazon 赞助，也可能不是如此。

Table of Contents

托管集成有什么用 AWS IoT Device Management	1
支持的区域	1
您是首次使用托管集成的用户吗？	1
托管式集成概述	1
托管集成术语	2
通用托管集成术语	2
Cloud-to-cloud 术语	2
数据模型术语	2
设置托管集成	4
注册获取 AWS 账户	4
创建具有管理访问权限的用户	4
开始使用	6
设备类型	6
配置 加密密钥	7
入职技巧	7
直接连接的设备上线	7
Hub 入职	7
连接集线器的设备上线	7
Cloud-to-cloud 设备上线	7
设备预调配	8
管理设备生命周期和配置文件	10
设备	10
设备配置文件	10
数据模型	12
托管集成数据模型	12
AWS 物质数据模型的实现	14
数据模型架构	15
能力架构	15
类型定义架构	16
能力定义架构	16
类型定义架构	32
在能力架构文档中构建和使用类型定义	37
设备命令和事件	50
设备命令	50

设备事件	52
标记资源	53
标签基本知识	53
标签限制	54
标签和 IAM 策略	54
托管集成通知	57
设置 Amazon Kinesis 以接收通知	57
第 1 步：创建 Amazon Kinesis 数据流	57
步骤 2：创建权限策略	57
步骤 3：导航到 IAM 控制面板并选择角色	58
步骤 4：使用自定义信任策略	58
第 5 步：应用您的权限策略	59
步骤 6：输入角色名称	59
设置托管集成通知	59
第 1 步：向用户授予调用 CreateDestination API 的权限	60
第 2 步：调用 CreateDestination API	60
第 3 步：调用 CreateNotificationConfiguration API	61
使用托管集成监控的事件类型	61
Cloud-to-Cloud (C2C) 连接器	66
什么是 cloud-to-cloud (C2C) 连接器？	66
连接器目录	66
AWS Lambda 用作 C2C 连接器	67
托管集成连接器工作流程	67
使用 C2C (cloud-to-cloud) 连接器的指导原则	67
构建 C2C (云到云) 连接器	68
先决条件	68
C2C 连接器要求	69
OAuth 2.0 账户关联要求	70
实现 C2C 连接器接口操作	75
调用你的 C2C 连接器	93
为您的 IAM 角色添加权限	94
手动测试您的 C2C 连接器	95
使用 C2C (云到云) 连接器	95
集线器 SDK	106
中心 SDK 架构	106
设备上线	106

设备上线组件	106
设备上线流程	107
设备控制	108
设备控制流程	109
SDK 组件	109
安装并验证托管集成 Hub SDK	110
使用安装软件开发工具包 AWS IoT Greengrass	110
使用脚本部署 Hub SDK	112
使用系统部署 Hub SDK	115
登上您的集线器	119
Hub 入门子系统	119
入职设置	120
加载设备并在集线器中对其进行操作	128
设置简单，便于加载和操作设备	128
在用户指导下进行设备加载和操作的设置	134
自定义证书处理程序	142
API 定义和组件	143
示例构建	144
使用量	148
自定义协议插件	149
中心 SDK 客户端	150
获取您的托管集成 Hub SDK	150
关于 Hub SDK 工具包	150
使用 Hub SDK 客户端创建您的自定义应用程序	151
运行您的自定义应用程序	153
中心 SDK 客户端 API	153
数据类型	158
集线器控制	159
先决条件	159
终端设备 SDK 组件	160
与终端设备 SDK 集成	160
示例：构建集线器控件	163
支持的示例	164
支持的平台	164
启用 CloudWatch 日志	164
先决条件	164

设置 Hub SDK 日志配置	165
支持的 Zigbee 和 Z-Wave 设备类型	166
在树莓派上运行托管集成	168
Sonoff Zigbee 固件刷新	169
树莓派上的托管集成 Hub SDK 镜像	170
托管集成树莓派上的 Hub SDK Docker 容器	174
托管集成演示应用程序	178
场外托管集成中心	180
Hub SDK 板外流程概述	180
先决条件	181
Hub SDK 场外流程	181
下线后 Hub SDK	184
特定于协议的中间件	185
中间件架构	186
End-to-end 中间件命令流示例	186
中间件代码组织	186
将中间件与 SDK 集成	192
终端设备 SDK	195
什么是终端设备 SDK ?	195
架构和组件	195
预备人	196
置备人工作流程	197
设置环境变量	197
注册自定义终端节点	197
创建配置文件	198
创建托管事物	198
SDK 用户 Wi-Fi 配置	199
按索赔提供舰队	199
托管式事物功能	199
OTA 更新	200
OTA 架构概述	200
先决条件	200
实施 Over-the-Air (OTA) 任务	201
OTA 任务配置设置	203
将配置设置应用于 OTA 任务	204
监控 OTA 通知	205

处理工作文档	206
实施 OTA 代理	206
数据模型代码生成器	207
代码生成过程	208
环境设置	210
为设备生成代码	211
低级 C 函数 APIs	213
OnOff 集群 API	214
服务设备互动	216
处理远程命令	216
处理不请自来的事件	217
开始使用终端设备 SDK	217
移植终端设备 SDK	229
技术参考	232
安全性	235
数据保护	235
用于托管集成的静态数据加密	236
Identity and access management	242
受众	242
使用身份进行身份验证	242
使用策略管理访问	243
AWS 托管策略	245
托管集成如何与 IAM 配合使用	248
基于身份的策略示例	253
问题排查	255
使用服务关联角色	257
用 AWS Secrets Manager 于 C2C workflows 的数据保护	260
托管集成如何使用机密	260
如何创建密钥	260
授予托管集成的访问权限 AWS IoT Device Management 以检索密钥	261
合规性验证	262
使用与接口 VPC 终端节点的托管集成	262
VPC 终端节点注意事项	263
创建 VPC 端点	264
测试 VPC 终端节点	265
访问控制	266

定价	267
限制	267
连接到 AWS IoT Device Management FIPS 端点的托管集成	268
控制面板端点	268
监控	269
CloudTrail 日志	269
中的管理活动 CloudTrail	270
事件示例	271
文档历史记录	275
.....	cclxxvi

托管集成有什么用 AWS IoT Device Management ?

的托管集成可 AWS IoT Device Management 帮助物联网解决方案提供商统一对来自数百家制造商的物联网设备的控制和管理。无论设备供应商或连接协议如何，您都可以使用托管集成来实现设备设置工作流程的自动化，并支持多台设备之间的互操作性。通过托管集成，您可以使用单个用户界面和一组用户界面 APIs 来控制、管理和操作一系列设备。

主题

- [支持的区域](#)
- [您是首次使用托管集成的用户吗？](#)
- [托管式集成概述](#)
- [托管集成术语](#)

支持的区域

以下区域支持托管集成：AWS IoT Device Management

- 加拿大 (中部)
- 欧洲地区 (爱尔兰)

您是首次使用托管集成的用户吗？

如果您是首次使用托管集成的用户，我们建议您先阅读以下章节：

- [设置托管集成](#)
- [开始使用托管集成 AWS IoT Device Management](#)

托管式集成概述

下图提供了托管集成的高级概述

托管集成术语

在托管集成中，有许多概念和术语对于管理自己的设备实现至关重要。以下各节概述了这些关键概念和术语，以便更好地理解托管集成。

通用托管集成术语

与事物相比，托管集成需要理解的一个重要概念是托管 AWS IoT Core 事物。

- **AWS IoT Core 事物**：AWS IoT Core 事物是一种提供数字表示的 AWS IoT Core 结构。开发人员需要管理策略、数据存储、规则、操作、MQTT 主题以及向数据存储传输设备状态。有关什么是 AWS IoT Core 事物的更多信息，请参阅[使用管理设备 AWS IoT](#)。
- **托管集成托管事物**：通过托管事物，我们提供了一个抽象来简化设备交互，并且不需要开发人员创建规则、操作、MQTT 主题和策略等项目。

Cloud-to-cloud 术语

与托管集成集成的物理设备可能来自第三方云提供商。为了将这些设备加入托管集成并与第三方云提供商通信，以下术语涵盖了支持这些工作流程的一些关键概念：

- **Cloud-to-cloud (C2C) 连接器**：C2C 连接器在托管集成和第三方云提供商之间建立连接。
- **第三方云提供商**：对于在托管集成之外制造和管理的设备，第三方云提供商允许最终用户控制这些设备，而托管集成则与第三方云提供商就各种工作流程（例如设备命令）进行通信。

数据模型术语

托管集成使用数据模型来组织数据和设备之间的 end-to-end 通信。以下术语涵盖了理解这两种数据模型的一些关键概念：

- **设备**：代表物理设备（例如可视门铃）的实体，该实体具有多个节点协同工作以提供完整的功能集。
- **端点**：端点封装了一项独立功能（铃声、运动检测、可视门铃中的照明）。
- **功能**：一种实体，代表在端点中提供功能所需的组件（按钮或可视门铃的灯光和铃声功能）。
- **动作**：代表与设备功能的交互的实体（按铃或查看谁在门口）。
- **事件**：代表来自设备功能的事件的实体。设备可以发送事件来报告门 incident/alarm, an activity from a sensor etc. (e.g. there is knock/ring 上的)。
- **属性**：表示设备状态下特定属性的实体（铃响了，门廊灯亮了，摄像机正在录制）。

- **数据模型**：数据层对应于有助于支持应用程序功能的数据和动词元素。当有意与设备交互时，应用程序会对这些数据结构进行操作。欲了解更多信息，请参阅网站上的 [connectedhomeip](#)。GitHub
- **架构**：架构是以 JSON 格式表示数据模型。

设置托管集成

以下各节将指导您完成使用托管集成的初始设置。AWS IoT Device Management

主题

- [注册获取 AWS 账户](#)
- [创建具有管理访问权限的用户](#)

注册获取 AWS 账户

如果您没有 AWS 账户，请完成以下步骤来创建一个。

要注册 AWS 账户

1. 打开<https://portal.aws.amazon.com/billing/注册>。
2. 按照屏幕上的说明操作。

在注册时，将接到电话或收到短信，要求使用电话键盘输入一个验证码。

当您注册时 AWS 账户，就会创建AWS 账户根用户一个。根用户有权访问该账户中的所有 AWS 服务和资源。作为最佳安全实践，请为用户分配管理访问权限，并且只使用根用户来执行[需要根用户访问权限的任务](#)。

AWS 注册过程完成后会向您发送一封确认电子邮件。您可以随时前往 <https://aws.amazon.com/> 并选择“我的账户”，查看您当前的账户活动并管理您的账户。

创建具有管理访问权限的用户

注册后，请保护您的安全 AWS 账户 AWS 账户根用户 AWS IAM Identity Center，启用并创建管理用户，这样您就可以不会使用 root 用户执行日常任务。

保护你的 AWS 账户根用户

1. 选择 Root 用户并输入您的 AWS 账户 电子邮件地址，以账户所有者的身份登录。[AWS 管理控制台](#)在下一页上，输入您的密码。

要获取使用根用户登录方面的帮助，请参阅《AWS 登录 用户指南》中的 [Signing in as the root user](#)。

2. 为您的根用户启用多重身份验证 (MFA)。

有关说明，请参阅 [IAM 用户指南中的为 AWS 账户 根用户启用虚拟 MFA 设备 \(控制台 \)](#)。

创建具有管理访问权限的用户

1. 启用 IAM Identity Center。

有关说明，请参阅《AWS IAM Identity Center 用户指南》中的 [Enabling AWS IAM Identity Center](#)。

2. 在 IAM Identity Center 中，为用户授予管理访问权限。

有关使用 IAM Identity Center 目录 作为身份源的教程，请参阅《[用户指南](#)》IAM Identity Center 目录中的[使用默认设置配置AWS IAM Identity Center 用户访问权限](#)。

以具有管理访问权限的用户身份登录

- 要使用您的 IAM Identity Center 用户身份登录，请使用您在创建 IAM Identity Center 用户时发送到您的电子邮件地址的登录网址。

有关使用 IAM Identity Center 用户[登录的帮助](#)，请参阅[AWS 登录 用户指南中的登录 AWS 访问门户](#)。

将访问权限分配给其他用户

1. 在 IAM Identity Center 中，创建一个权限集，该权限集遵循应用最低权限的最佳做法。

有关说明，请参阅《AWS IAM Identity Center 用户指南》中的 [Create a permission set](#)。

2. 将用户分配到一个组，然后为该组分配单点登录访问权限。

有关说明，请参阅《AWS IAM Identity Center 用户指南》中的 [Add groups](#)。

开始使用托管集成 AWS IoT Device Management

以下各节概述了开始使用托管集成需要采取的步骤。

主题

- [设备类型](#)
- [配置 加密密钥](#)
- [入职技巧](#)

设备类型

托管集成可以管理多种类型的设备。每台设备属于以下三个类别之一：

- **直接连接的设备**：此类设备直接连接到托管集成端点。通常，这些设备由设备制造商构建和管理，其中包括用于直接连接的托管集成终端设备 SDK。
- **连接集线器的设备**：这些设备通过运行托管集成 Hub SDK 的集线器连接到托管集成，该集线器管理设备发现、入门和控制功能。最终用户可以通过按下按钮启动或扫描条形码来加载这些设备。

支持以下两个工作流程来加载与集线器连接的设备：

- 按下最终用户启动的按钮即可开始设备发现
- 基于条形码的扫描以执行设备关联
- **Cloud-to-cloud (C2C) 设备**：这些设备由供应商设计和管理，这些供应商维护自己的云基础架构和用于设备控制的特定移动应用程序。托管集成客户可以访问预先构建的 C2C 连接器目录或创建自己的连接器，以开发通过统一接口与多个第三方供应商云配合使用的物联网解决方案。

当最终用户首次开启 C2C 设备时，必须向其相应的第三方云提供商进行托管集成，以获取其设备功能和元数据。完成配置工作流程后，托管集成可以代表最终用户与云设备和第三方云提供商进行通信。

Note

集线器不是上面列出的特定设备类型。其目的是充当智能家居设备的控制器，并促进托管集成与第三方云提供商之间的连接。它既可以作为上面列出的设备类型，也可以用作集线器。

配置 加密密钥

对于在最终用户、托管集成和第三方云之间路由的数据，安全性至关重要。我们支持保护您的设备数据的方法之一是使用安全的 end-to-end 加密密钥进行加密，用于路由您的数据。

作为托管集成的客户，您可以使用以下两种方式使用加密密钥：

- 使用默认的托管集成托管加密密钥。
- 提供 AWS KMS key 您创建的。

有关 AWS KMS 服务的更多信息，请参阅[密钥管理服务 \(KMS\)](#)

通过调用《[PutDefaultEncryptionConfiguration](#) 托管集成 API 参考指南》中的 API，您可以更新要使用的加密密钥选项。默认情况下，托管集成使用默认的托管集成托管加密密钥。您可以随时使用 [PutDefaultEncryptionConfiguration](#) API 更新您的加密密钥配置。

此外，调用 [GetDefaultEncryptionConfiguration](#) API 命令会返回有关默认或指定区域中 AWS 账户的加密配置的信息。

入职技巧

下面列出了入职的类型：

直接连接的设备上线

[预备人](#)有关载入直连设备的步骤，请参阅。

Hub 入职

[将您的集线器加入托管集成](#)有关加载集线器的步骤，请参阅。

连接集线器的设备上线

[加载设备并在集线器中对其进行操作](#)有关载入连接集线器的设备的步骤，请参阅。

Cloud-to-cloud 设备上线

[使用 C2C \(云到云\) 连接器](#)有关将云设备从第三方云供应商加载到托管集成的步骤，请参阅。

设备预调配

设备配置简化了设备上线流程，监督了整个设备生命周期，并为托管集成的其他方面可以访问的设备信息建立集中存储库。托管集成提供了一个用于管理各种设备类型的统一接口，可容纳通过设备软件开发套件 (SDK) 或通过集线器设备间接链接的 commercial-off-the-shelf (COTS) 设备直接连接的第一方客户设备。

托管集成中的每台设备，无论设备类型如何，都有一个名为 `a managedThingId` 的全球唯一标识符。在整个设备生命周期中，该标识符用于设备的入门和管理。它完全由托管集成管理，并且在所有托管集成中，该特定设备都是独一无二的。AWS 区域当设备最初添加到托管集成时，系统会创建此标识符并将其附加到托管集成中的托管事物。托管事物是托管集成中物理设备的数字表示形式，用于镜像物理设备的所有设备元数据。对于第三方设备，除了 `managedThingId` 存储在代表物理设备的托管集成中的标识符外，它们可能还有自己的、针对其第三方云的独立唯一标识符。

正在配置的设备可能具有不同的状态，具体取决于它们所处的入门流程阶段。以下列表描述了每种配置状态：

- 已激活：设备已找到并且命令和控制可用。
- 已发现：设备已找到，但命令和控制尚不可用。
- 未关联：托管事物已创建，但需要进一步的操作才能发现。无法从 AWS Cloud 或 AWS IoT 托管集成控制器（集线器）访问它
- `PRE_ASSOCIATED`：托管事物已创建，开机或连接后即可自动发现。无法从 AWS Cloud 或 AWS IoT 托管集成控制器（集线器）访问它。
- `DELETE_IN_PROGRESS`：异步删除过程已启动。
- 已删除：设备已从中删除 AWS Cloud。
- 隔离：先前发现或激活的受管事物，无法再访问。例如，用于第三方云的设备，其连接器关联已全部删除。

以下入职流程用于为中心配置托管集成：

[将您的集线器加入托管集成](#)：设置核心配置器和协议特定的插件，它们可以协同工作以处理设备身份验证、通信和设置。

提供了以下入门流程，用于为集线器连接的设备配置托管集成：

- **简单设置 (SS)**：最终用户打开物联网设备的电源，并使用设备制造商的应用程序扫描其二维码。然后，设备将注册到托管集成云并连接到物联网中心。
- **零触摸设置 (ZTS)**：该设备已预先关联到供应链的上游。例如，最终用户无需扫描设备二维码，而是提前完成此步骤，以便将设备预关联到客户账户。
- **用户指导设置 (UGS)**：最终用户开启设备并按照交互式步骤将其加入托管集成。这可能包括按下 IoT 中心上的按钮、使用设备制造商的应用程序或同时按下集线器和设备上的按钮。如果简单设置失败，则可以使用此方法。

Note

托管集成中的设备配置工作流程与设备的入门要求无关。无论设备类型或设备协议如何，托管集成都提供了简化的用户界面，便于用户登录和管理设备。

设备和设备配置文件生命周期

管理设备生命周期和设备配置文件可确保您的设备群安全且高效运行。

主题

- [设备](#)
- [设备配置文件](#)

设备

在初始入职期间，托管集成会创建物理设备的数字化双胞胎，称为托管事物。Managed Thing 具有一个全球唯一标识符，用于在所有区域的托管集成中识别设备。设备在配置期间与本地集线器配对，以便与托管集成进行实时通信，或者为第三方设备配置第三方云。设备还与所有者相关联，该所有者由托管事物的公开 owner APIs 参数标识，例如 GetManagedThing。根据设备类型，设备会链接到相应的设备配置文件。

Note

如果在不同的客户下多次配置物理设备，则该设备可能有多条记录。

设备生命周期从使用 API 在托管集成中创建托管事物开始，到客户使用 CreateManagedThing API 删除托管事物时结束。DeleteManagedThing 设备生命周期由以下公众管理 APIs：

- CreateManagedThing
- ListManagedThings
- GetManagedThing
- UpdateManagedThing
- DeleteManagedThing

设备配置文件

设备配置文件代表一种特定类型的设备，例如灯泡或门铃。它与制造商相关联，包含设备的功能。设备配置文件存储托管集成的设备连接设置请求所需的身份验证材料。使用的身份验证材料是设备条形码。

在设备制造过程中，制造商可以使用托管集成注册其设备配置文件。这使制造商能够在入门和配置工作流程中从托管集成中获取设备所需的材料。设备配置文件中的元数据存储于物理设备上或打印在设备标签上。当制造商在托管集成中删除设备配置文件时，设备配置文件的生命周期即告结束。

数据模型

数据模型表示系统中数据的组织层次结构。此外，它还支持整个设备实现之间的 end-to-end 通信。对于托管集成，使用了两种数据模型。托管集成数据模型和物质数据模型的 AWS 实现。它们有相似之处，但也有细微的差异，将在以下主题中概述。

对于第三方设备，这两种数据模型都用于最终用户、托管集成和第三方云提供商之间的通信。要转换来自两个数据模型的设备命令和设备事件之类的消息，需要利用 Cloud-to-Cloud 连接器功能。

主题

- [托管集成数据模型](#)
- [AWS 物质数据模型的实现](#)
- [数据模型架构](#)

托管集成数据模型

托管集成数据模型管理最终用户与托管集成之间的所有通信。

设备层次结构

endpoint 和 capability 数据元素用于描述托管集成数据模型中的设备。

endpoint

endpoint 表示该功能提供的逻辑接口或服务。

```
{
  "endpointId": { "type": "string" },
  "capabilities": Capability[]
}
```

Capability

capability 代表设备功能。

```
{
  "$id": "string",           // Schema identifier (e.g. /schema-versions/
  capability/matter.OnOff@1.4)
  "name": "string",         // Human readable name
}
```

```

"version": "string",           // e.g. 1.0
"properties": Property[],
"actions": Action[],
"events": Event[]
}

```

对于capability数据元素，有三个项目构成该项目：propertyaction、和event。它们可用于与设备交互和监控。

- 属性：设备保持的状态，例如可调光灯的当前亮度等级属性。

- ```

{
 "name": // Property Name is outside of Property Entity
 "value": Value, // value represented in any type e.g. 4, "A", []
 "lastChangedAt": Timestamp // ISO 8601 Timestamp upto milliseconds yyyy-MM-ddTHH:mm:ss.ssssssZ
 "mutable": boolean,
 "retrievable": boolean,
 "reportable": boolean
}

```

- 操作：可以执行的任务，例如在门锁上锁门。操作可能会产生响应和结果。

- ```

{
  "name": { "$ref": "/schema-versions/definition/aws.name@1.0" }, //required
  "parameters": Map<String name, JSONNode value>,
  "responseCode": HTTPResponseCode,
  "errors": {
    "code": "string",
    "message": "string"
  }
}

```

- 事件：本质上是过去状态转换的记录。虽然事件property代表当前的状态，但却是过去的日记，包括单调递增的计数器、时间戳和优先级。它们支持捕获状态转换，以及无法轻易实现的数据建模property。

- ```

{
 "name": { "$ref": "/schema-versions/definition/aws.name@1.0" }, //
 required
 "parameters": Map<String name, JSONNode value>
}

```

# AWS 物质数据模型的实现

Matter 数据模型的 AWS 实施管理托管集成与第三方云提供商之间的所有通信。

有关更多信息，请参阅 [Matter 数据模型：开发者资源](#)。

## 设备层次结构

有两个数据元素用于描述设备：endpoint和cluster。

### endpoint

endpoint表示该功能提供的逻辑接口或服务。

```
{
 "id": { "type":"string"},
 "clusters": Cluster[]
}
```

### cluster

cluster代表设备功能。

```
{
 "id": "hexadecimalString",
 "revision": "string" // optional
 "attributes": AttributeMap<String attributeId, JSONNode>,
 "commands": CommandMap<String commandId, JSONNode>,
 "events": EventMap<String eventId, JsonNode>
}
```

对于cluster数据元素，有三个项目构成该项目：attributecommand、和event。它们可用于与设备交互和监控。

- 属性：设备保持的状态，例如可调光灯的当前亮度等级属性。

- ```
{
  "id" (hexadecimalString): (JsonNode) value
}
```

- 命令：可以执行的任务，例如在门锁上锁门。命令可能会生成响应和结果。

- ```
"id": {
```

```

 "fieldId": "fieldValue",
 ...
 "responseCode": HTTPResponseCode,
 "errors": {
 "code": "string",
 "message": "string"
 }
 }
}

```

- 事件：本质上是过去状态转换的记录。虽然事件attributes代表当前的状态，但却是过去的日记，包括单调递增的计数器、时间戳和优先级。它们支持捕获状态转换，以及无法轻易实现的数据建模attributes。

```

 "id": {
 "fieldId": "fieldValue",
 ...
 }

```

## 数据模型架构

托管集成支持两种架构类型：功能和类型定义。如果要创建自定义数据模型，则使用 JSON 架构文档来定义任一类型的架构。每个架构文档的字符数限制为 50,000。

## 能力架构

功能是代表端点内特定功能的基本构建块。借助功能，您可以使用属性、操作和事件对设备状态和行为进行建模。属性使您可以灵活地使用任何声明性数据类型对设备的状态属性进行建模。操作和事件对设备的行为进行建模，包括设备可以执行的命令和可以报告的信号。

下图显示了能力架构的高级结构。

```

Capability
|
|-- Action
|-- Event
|-- Property

```

### 操作

表示与设备功能的交互的实体。例如，按铃或查看谁在门口。

## 事件

代表来自设备功能的事件的实体。设备可以发送事件以报告事件、警报或来自传感器（例如敲门声）的活动。

## 属性

代表设备状态下特定属性的实体。例如，钟声响起或门廊灯亮起

每项功能都包括一个唯一的命名空间标识符、版本信息及其用途描述。架构文档使用语义版本控制来保持向后兼容性，同时启用新功能。

有关更多信息，请参阅 [能力定义架构](#)。

## 类型定义架构

类型定义是一种声明性的结构化数据类型，可实现可重用性和可组合性。它定义了应如何格式化和限制信息。使用类型定义在 IoT 解决方案中创建标准化数据格式。

每个类型定义包括：

- 唯一的命名空间标识符
- 标题
- 描述
- 定义数据格式和约束条件的属性

类型可以是简单的基元，例如具有定义限制的整数或字符串，也可以是复杂的结构，例如枚举或具有多个字段的自定义对象。类型定义使用 JSON 架构语法来指定约束，包括最小值和最大值、字符串长度以及允许的模式。

有关更多信息，请参阅 [类型定义架构](#)。

## 能力定义架构

一项功能是使用声明性的 JSON 文档记录的，该文档为该功能在系统中的运行方式提供了明确的协议。

对于权能，必备元素为 \$idname、extrinsicId、，extrinsicVersion 并且至少在以下一个部分中包含一个元素：

- `properties`
- `actions`
- `events`

权能中的可选元素是`$ref`、`description`、`version`、`$defs`和`extrinsicProperties`。有关权能，`$ref`必须参阅`aws.capability`。

以下各节详细介绍了用于能力定义的架构。

## `$id` ( 必填 )

`$id` 元素标识架构定义。它必须遵循以下结构：

- 从 `/schema-versions/` URI 前缀开始
- 包括`capability`架构类型
- 使用正斜杠 (`/`) 作为 URI 路径分隔符
- 包括架构标识，片段之间用句点分隔 (`.`)
- 使用`@`字符分隔架构 ID 和版本
- 以 `semver` 版本结尾，使用句点 (`.`) 分隔版本片段

架构标识必须以长度为 3-12 个字符的根命名空间开头，然后是可选的子命名空间和名称。

`semver` 版本包括主要版本 ( 最多 3 位数 )、次要版本 ( 最多 3 位数 ) 和可选的补丁版本 ( 最多 4 位数 )。

### Note

您不能使用保留的命名空间或 `aws matter`

## Example 示例 `$id`

```
/schema-version/capability/aws.Recording@1.0
```

## `$ref`

该`$ref`元素引用系统中的现有能力。它遵循与`$id`元素相同的约束。

**Note**

类型定义或功能必须与\$ref文件中提供的值相同。

**Example示例 \$ref**

```
/schema-version/definition/aws.capability@1.0
```

**姓名 ( 必填 )**

名称元素是一个字符串，表示架构文档中的实体名称。它通常包含缩写，必须遵循以下规则：

- 仅包含字母数字字符、句点 (.)、正斜杠 (/)、连字符 (-) 和空格
- 以字母开头
- 最多 64 个字符

Amazon Web Services 控制台用户界面和文档中使用名称元素。

**Example示例名称**

```
Door Lock
On/Off
Wi-Fi Network Management
PM2.5 Concentration Measurement
RTCSessionController
Energy EVSE
```

**删除实例快照**

标题元素是架构文档所代表的实体的描述性字符串。它可以包含任何字符，可在文档中使用。能力标题的最大长度为 256 个字符。

**Example标题示例**

```
Real-time Communication (RTC) Session Controller
Energy EVSE Capability
```

## description

该description元素详细解释了架构文档所代表的实体。它可以包含任何字符，可在文档中使用。能力描述的最大长度为 2048 个字符

### Example示例描述

```
Electric Vehicle Supply Equipment (EVSE) is equipment used to charge an Electric Vehicle (EV) or Plug-In Hybrid Electric Vehicle.
This capability provides an interface to the functionality of Electric Vehicle Supply Equipment (EVSE) management.
```

## version

version 元素是可选的。它是一个表示架构文档版本的字符串。它具有以下限制：

- 使用 semver 格式，以下版本片段用 . ( 句点 ) 分隔。
  - MAJOR版本，最多 3 位数
  - MINOR版本，最多 3 位数
  - PATCH版本 ( 可选 )，最多 4 位数
- 长度可以介于 3 到 12 个字符之间。

### Example示例版本

```
1.0
```

```
1.12
```

```
1.4.1
```

### 使用功能版本

功能是一个不可变的版本化实体。任何更改都将创建新版本。系统使用 MAJOR.MINOR.PATCH 格式的语义版本控制，其中：

- 更改向后不兼容的 API 时，主要版本会增加
- 以向后兼容的方式添加功能时，次要版本会增加

- 在功能中添加一些不起作用的次要添加时，补丁版本会增加。

源自 Matter 集群的功能基于 1.4 版，预计每个 Matter 版本都将导入到系统中。由于 Matter 版本同时消耗 semver 的主要和次要级别，因此托管集成只能使用 PATCH 版本。

在为 Matter 添加补丁版本时，请务必考虑到 Matter 使用顺序修订版。所有 PATCH 版本都必须符合 Matter 规范中记录的修订版，并且必须向后兼容。

要修复任何向后不兼容的问题，您必须与连接标准联盟 (CSA) 合作解决规范中的问题并发布新的修订版。

AWS-托管功能的初始版本为。1.0有了这些，就可以使用所有三个级别的版本。

### 外部版本 ( 必填 )

这是一个字符串，表示在 AWS IoT 系统之外管理的版本。对于 Matter 能力，`extrinsicVersion`映射到 `revision`

它以字符串化的整数值表示，长度可以是 1 到 10 位数字。

Example 示例版本

7

1567

### extrinsicId ( 必填 )

该 `extrinsicId` 元素表示在 Amazon Web Services 物联网系统之外管理的标识符。对于 Matter 能力 `clusterId`、`attributeId`、`commandId` 它会根据上下文映射到 `fieldId`、`、、` 或 `eventId`

`extrinsicId` 可以是字符串化的十进制整数 ( 1-10 位数 )，也可以是字符串化的十六进制整数 ( `0x` 或 `0X` 前缀，后面是 1-8 个十六进制数字 )。

#### Note

对于 AWS，供应商 ID (VID) 为 `0x1577`，对于 Matter，则为 `0`。系统确保自定义架构不会使用这些 VIDs 为功能保留的内容。

## Example 示例 extinSICID

```
0018
0x001A
0x15771002
```

## \$defs

该\$defs部分是子架构的映射，在 JSON 架构允许的情况下，可以在架构文档中引用这些子架构。在此地图中，键用于本地参考定义，值提供了 JSON 架构。

### Note

系统仅强制要求该映射\$defs是有效的映射，并且每个子架构都是有效的 JSON 架构。不强制执行其他规则。

使用定义时，请遵循以下限制：

- 在定义名称中仅使用 URI 友好字符
- 确保每个值都是一个有效的子架构
- 包括符合架构文档大小限制的任意数量的子架构

## 外在特性

该extrinsicProperties元素包含一组在外部系统中定义但保留在数据模型中的属性。对于 Matter 功能，它映射到 ZCL 集群、属性、命令或事件中不同的未建模或部分建模的元素。

外在属性必须遵循以下限制：

- 属性名称必须是字母数字，不含空格或特殊字符
- 属性值可以是任何 JSON 架构值
- 最多 20 处房产

该系统支持多种功能extrinsicProperties，包括access、apiMaturity、cliFunctionName、和其他。这些属性便于 ACL 进行数据模型转换 AWS（反之亦然）。

**Note**

功能的action、event、和struct字段元素支持外部属性property，但不支持能力或集群本身。

## 系统支持的外部属性

系统会跟踪以下未建模或部分建模的集群、属性、命令或事件属性，就像转换到 ZCL 或从 ZCL 转换extrinsicProperties期间一样：

### access

每个访问对象都包含以下内容：

- op-操作建模为enum，其值为：readwrite、或 invoke
- privilege-特权建模为enum，其值为：viewproxy\_view、operate、manage、或 administer
- role-代表操作员角色的无界字符串

### apiMaturity

代表成熟度水平的无界纯字符串。在 ZCL 中将其建模为，enum其值为：stable、provisionalinternal、或 deprecated

### side

以枚举形式建模，其值为：eitherserver、和 client

### 布尔属性

以下属性是布尔标志：

- isFabricScoped
- isFabricSensitive
- mustUseAtomicWrite
- mustUseTimedInvoke

### 字符串属性

以下属性表示为无界字符串：

- cli

- cliFunctionName
- functionName
- group
- introducedIn
- manufacturerCode
- noDefaultImplementation
- presentIf
- priority
- removedIn
- reportableChange
- reportMinInterval
- reportMaxInterval
- restriction
- storage

## 转型注意事项

对于 ZCL 变换 `extrinsicProperties`，无需处理即可存储在地图中。使用发现功能的自定义架构不会进行 ZCL 转换。但是，如果您计划将来为自定义架构实现 ZCL 转换，则必须对所有无界纯字符串类型进行建模，`extrinsicProperties` 并定义诸如枚举、模式（正则表达式）和长度之类的约束。这种准备工作可确保在转换过程中正确处理这些特性。

相比之下，对于连接器的 AWS 转换，`extrinsicProperties` 则根本不包括在内，因为连接器格式中不需要这些细节。

## 属性

属性表示该功能的设备管理状态。每个状态都被定义为一个键值对，其中键描述状态的名称，值描述状态的定义。

使用属性时，请遵循以下限制：

- 在属性名称中仅使用字母数字字符，不得使用空格或特殊字符
- 包括符合架构文档大小限制的任意数量的属性

## 使用属性

功能中的属性是一个基本元素，它代表由托管集成提供支持的设备的特定状态。它代表设备的当前状态或配置。通过标准化这些属性的定义和结构，智能家居系统可确保来自不同制造商的设备能够进行有效通信，从而创造无缝且可互操作的体验。

对于能力属性，必填元素为`extrinsicId`和`value`。能力属性中的可选元素是`descriptionretrievable`、`mutable`、`reportable`和`extrinsicProperties`。

## 值

一种无界结构，允许生成器设置任何符合 JSON 架构的约束来定义此属性的数据类型。

定义值时，请遵循以下约束：

- 对于简单类型，请使用`type`和任何其他原生 JSON 架构约束，例如`maxLength`或`maximum`
- 对于复合类型，请使用`oneOf`、`allOf`、或`anyOf`。系统不支持`not`关键字
- 要引用任何全局类型，请`$ref`使用有效的可发现引用
- 为了获得空性，请遵循 OpenAPI 类型架构定义，为可空属性提供布尔标志（`true`如果允许的值为`null`）

示例：

```
{
 "$ref": "/schema-versions/definition/matter.uint16@1.4",
 "nullable": true,
 "maximum": 4096
}
```

## 可检索

描述状态是否可读的布尔值。

状态的可读性方面将推迟到设备对功能的实现。设备决定给定状态是否可读。目前尚不支持在能力报告中报告状态的这一方面，因此不支持在系统内强制执行。

示例：`true` 或 `false`

## Mutable

描述状态是否可写的布尔值。

状态的可写性方面被推迟到设备对功能的实现。设备决定给定状态是否可写。目前尚不支持在能力报告中报告状态的这一方面，因此不支持在系统内强制执行。

示例：true 或 false

## 可报告

一个布尔值，用于描述状态发生变化时设备是否报告状态。

状态的可报告性方面将推迟到设备对功能的实现。设备决定给定状态是否可报告。目前尚不支持在能力报告中报告状态的这一方面，因此不支持在系统内强制执行。

示例：true 或 false

## 操作

操作是遵循请求-响应模型的架构管理操作。每个操作都代表一个设备实现的操作。

在实施操作时，请遵循以下限制：

- 在操作数组中仅包含唯一的动作
- 包括符合架构文档大小限制的任意数量的操作

## 使用操作

操作是与托管集成系统中的设备功能进行交互和控制的标准化方式。它代表可在设备上执行的特定命令或操作，并采用结构化格式，用于对任何必要的请求或响应参数进行建模。这些操作是用户意图和设备操作之间的桥梁，可实现对不同类型的智能设备进行一致而可靠的控制。

对于操作，必填元素为name和extrinsicId。可选元素是descriptionextrinsicProperties、request和response。

## 描述

描述的最大长度限制为 1536 个字符。

## 请求

请求部分是可选的，如果没有请求参数，则可以省略。如果省略，则系统支持仅使用名称发送没有任何有效载荷的请求Action。这用于简单的操作，例如打开或关闭灯。

复杂的动作需要额外的参数。例如，直播摄像机镜头的请求可能包含有关要使用的流媒体协议或是否将直播发送到特定显示设备的参数。

对于操作请求，必填元素为parameters。可选元素是descriptionextrinsicId、和extrinsicProperties。

### 请求描述

描述采用与第 3.5 节相同的格式，最大长度为 2048 个字符。

### 响应

在托管集成中，对于通过 [SendManagedThingCommand](#) API 发送的任何操作请求，该请求都会到达设备并期望得到异步响应。操作响应定义了该响应的结构。

对于操作请求，必填元素为parameters。可选元素是namedescription、extrinsicId、extrinsicProperties、errors和responseCode。

### 响应描述

描述采用与相同的格式[description](#)，最大长度为 2048 个字符。

### 响应名称

该名称的格式与以下内容相同[姓名 \( 必填 \)](#)，但有以下额外细节：

- 响应的常规名称是通过附加Response到操作名称得出的。
- 如果要使用其他名称，则可以在此name元素中提供该名称。如果响应中提供了 aname，则此值的优先级高于常规名称。

### 错误

如果在处理请求时出现错误，则在响应中提供的唯一消息的无限数组。

### 约束：

- 消息项声明为 JSON 对象，其中包含以下字段：
  - code: 包含字母数字字符和\_ ( 下划线 ) 的字符串，长度介于 1 到 64 个字符之间
  - message: 一个无界限的字符串值

## Example 错误消息示例

```
"errors": [
 {
 "code": "AD_001",
 "message": "Unable to receive signal from the sensor. Please check connection
with the sensor."
 }
]
```

### 响应代码

显示请求处理方式的整数代码。我们建议设备代码返回使用 HTTP 服务器响应状态码规范的代码，以实现系统内部的一致性。

约束：一个介于 100 到 599 之间的整数值。

### 请求或响应参数

参数部分定义为名称和子架构对的映射。如果可以容纳在架构文档中，则可以在请求参数中定义任意数量的参数。

参数名称只能包含字母数字字符。不允许使用空格或任何其他字符。

### 参数字段

a 中的必填元素 `parameter` 是 `extrinsicId` 和 `value`。可选元素为 `description` 和 `extrinsicProperties`。

描述元素的格式与相同 [description](#)，最大长度为 1024 个字符。

### **extrinsicId** 并 **extrinsicProperties** 覆盖

`extrinsicId` 和 `extrinsicProperties` 的格式与 [extrinsicId \(必填\)](#) 和相同 [外在特性](#)，但有以下额外细节：

- 如果在请求或响应中提供了 `extrinsicId` 则该值的优先级高于在操作级别提供的值。系统必须 `extrinsicId` 先使用 `request/response` 关卡，如果缺少则使用操作关卡 `extrinsicId`
- 如果 `extrinsicProperties` 在请求或响应中提供，则这些属性的优先级高于在操作级别提供的 `value`。系统必须采取操作级别 `extrinsicProperties` 并替换该级别提供的键值对 `request/response extrinsicProperties`

## Example extrinsicID 和 extrinsicProperties

```
{
 "name": "ToggleWithEffect",
 "extrinsicId": "0x0001",

 "extrinsicProperties": {
 "apiMaturity": "provisional",
 "introducedIn": "1.2"
 },
 "request": {
 "extrinsicProperties": {
 "apiMaturity": "stable",
 "manufacturerCode": "XYZ"
 },
 "parameters": {
 ...
 }
 },
 "response": {
 "extrinsicProperties": {
 "noDefaultImplementation": true
 },
 "parameters": {
 ...
 }
 }
}
```

在上面的示例中，操作请求的有效值为：

```
effective request
"name": "ToggleWithEffect",
"extrinsicId": "0x0001",
"extrinsicProperties": {
 "apiMaturity": "stable",
 "introducedIn": "1.2"
 "manufacturerCode": "XYZ"
},
"parameters": {
 ...
}
```

```
effective response
"name": "ToggleWithEffectResponse",
"extrinsicId": "0x0001",
"extrinsicProperties": {
 "apiMaturity": "provisional",
 "introducedIn": "1.2"
 "noDefaultImplementation": true
},
"parameters": {
 ...
}
```

## 内置动作

对于所有功能，您可以使用关键字ReadState和执行自定义操作UpdateState。这两个操作关键字将作用于数据模型中定义的功能属性。

### ReadState

向发送命令managedThing以读取其状态属性的值。ReadState用作强制更新设备状态的一种方式。

### UpdateState

发送命令以更新某些属性。

在以下情况下，强制设备状态同步可能很有用：

1. 设备处于离线状态一段时间，未发出任何事件。
2. 该设备刚刚配置完毕，尚未在云中维护任何状态。
3. 设备状态与设备的实际状态不同步。

### ReadState 例子

使用 [SendManagedThingCommandAPI](#) 检查灯是开还是关：

```
{
 "Endpoints": [
 {
 "endpointId": "1",
```

```

 "capabilities": [
 {
 "id": "aws.OnOff",
 "name": "On/Off",
 "version": "1",
 "actions": [
 {
 "name": "ReadState",
 "parameters": {
 "propertiesToRead": ["OnOff"]
 }
 }
]
 }
]
 }
}

```

读取该matter.OnOff功能的所有状态属性：

```

{
 "Endpoints": [
 {
 "endpointId": "1",
 "capabilities": [
 {
 "id": "aws.OnOff",
 "name": "On/Off",
 "version": "1",
 "actions": [
 {
 "name": "ReadState",
 "parameters": {
 "propertiesToRead": ["*"]
 // Use the wildcard operator to read ALL state properties for a
 capability
 }
 }
]
 }
]
 }
]
}

```

```
]
}
```

## UpdateState 示例

使用 [SendManagedThingCommandAPI](#) 更改OnTime为灯光：

```
{
 "Endpoints": [
 {
 "endpointId": "1",
 "capabilities": [
 {
 "id": "matter.OnOff",
 "name": "On/Off",
 "version": "1",
 "actions": [
 {
 "name": "UpdateState",
 "parameters": {
 "OnTime": 5
 }
 }
]
 }
]
 }
]
}
```

## 事件

事件是由设备实现的架构管理的单向信号。

根据以下限制实现事件：

- 在事件数组中仅包含唯一的事件
- 包括符合架构文档大小限制的任意数量的事件

## 托管集成系统中的事件

### 处理 事件

事件是一种主动了解设备或其周围环境变化的标准化方式。它表示设备将发送到云端的建模事件，以提供有关在设备上修改或在其环境中感知到的内容的信息。由于这些事件是建模的，因此客户可以在控制流中使用它们来对特定事件以及其中提供的详细信息做出反应。

对于事件，必填元素为name和extrinsicId。可选元素是descriptionextrinsicProperties、和request。

### 描述

描述采用与中所述相同的格式[description](#)，最大长度为 512 个字符。

### 请求

该request部分是可选的，如果没有请求参数，则可以省略。如果省略，则系统支持设备仅使用事件名称发送没有任何有效负载的事件请求。这用于简单的事件，例如泵上的传感器故障，或者烟雾或一氧化碳警报器上的警报被静音。

复杂的动作需要额外的参数。例如，直播摄像机镜头的请求可能包含有关要使用的流媒体协议或是否将直播发送到特定显示设备的参数。

对于活动请求，必填元素为parameters。没有可选元素。

### 响应

目前不支持事件响应。

## 类型定义架构

以下各节详细介绍了用于类型定义的架构。

### \$id

\$id 元素标识架构定义。它必须遵循以下结构：

- 从 /schema-versions/ URI 前缀开始
- 包括definition架构类型
- 使用正斜杠 (/) 作为 URI 路径分隔符

- 包括架构标识，片段之间用句点分隔 (.)
- 使用@字符分隔架构 ID 和版本
- 以 semver 版本结尾，使用句点 (.) 分隔版本片段

架构标识必须以长度为 3-12 个字符的根命名空间开头，然后是可选的子命名空间和名称。

semver 版本包括主要版本（最多 3 位数）、次要版本（最多 3 位数）和可选的补丁版本（最多 4 位数）。

#### Note

您不能使用保留的命名空间或 `aws matter`

Example 示例 \$id

```
/schema-version/capability/aws.Recording@1.0
```

## \$ref

\$ref 元素引用系统中现有的类型定义。它遵循与 \$id 元素相同的约束。

#### Note

类型定义或功能必须与 \$ref 文件中提供的值相同。

Example 示例 \$ref

```
/schema-version/definition/aws.capability@1.0
```

## 名称

名称元素是一个字符串，表示架构文档中的实体名称。它通常包含缩写，必须遵循以下规则：

- 仅包含字母数字字符、句点 (.)、正斜杠 (/)、连字符 (-) 和空格
- 以字母开头

- 最多 192 个字符

Amazon Web Services 控制台用户界面和文档中使用名称元素。

Example 示例名称

```
Door Lock
On/Off
Wi-Fi Network Management
PM2.5 Concentration Measurement
RTCSessionController
Energy EVSE
```

## 删除实例快照

标题元素是架构文档所代表的实体的描述性字符串。它可以包含任何字符，可在文档中使用。

Example 标题示例

```
Real-time Communication (RTC) Session Controller
Energy EVSE Capability
```

## description

该description元素详细解释了架构文档所代表的实体。它可以包含任何字符，可在文档中使用。

Example 示例描述

```
Electric Vehicle Supply Equipment (EVSE) is equipment used to charge an Electric
Vehicle (EV) or Plug-In Hybrid Electric Vehicle.
 This capability provides an interface to the functionality of Electric
Vehicle Supply Equipment (EVSE) management.
```

## extrinsicCID

该extrinsicId元素表示在 Amazon Web Services 物联网系统之外管理的标识符。对于 Matter 能力，它会映射到clusterIdattributeIdcommandIdeventId、fieldId、或，具体取决于上下文。

extrinsicId可以是字符串化的十进制整数（1-10 位数），也可以是字符串化的十六进制整数（0x 或 0X 前缀，后面是 1-8 个十六进制数字）。

**Note**

对于 AWS，供应商 ID (VID) 为 0x1577，对于 Matter，则为 0。系统确保自定义架构不会使用这些 VIDs 为功能保留的内容。

**Example 示例 extinSICID**

```
0018
0x001A
0x15771002
```

**外在特性**

该 `extrinsicProperties` 元素包含一组在外部系统中定义但保留在数据模型中的属性。对于 Matter 功能，它映射到 ZCL 集群、属性、命令或事件中不同的未建模或部分建模的元素。

外在属性必须遵循以下限制：

- 属性名称必须是字母数字，不含空格或特殊字符
- 属性值可以是任何 JSON 架构值
- 最多 20 处房产

该系统支持多种功能 `extrinsicProperties`，包括 `access`、`apiMaturity`、`cliFunctionName`、和其他。这些属性便于 ACL 进行数据模型转换 AWS（反之亦然）。

**Note**

功能的 `action`、`event`、和 `struct` 字段元素支持外部属性 `property`，但不支持能力或集群本身。

**系统支持的外部属性**

系统会跟踪以下未建模或部分建模的集群、属性、命令或事件属性，就像转换到 ZCL 或从 ZCL 转换 `extrinsicProperties` 期间一样：

## access

每个访问对象都包含以下内容：

- `op`-操作建模为enum，其值为：`readwrite`、或 `invoke`
- `privilege`-特权建模为enum，其值为：`viewproxy_view`、`operate`、`manage`、或 `administer`
- `role`-代表操作员角色的无界字符串

## apiMaturity

代表成熟度水平的无界纯字符串。在 ZCL 中将其建模为，enum其值为：`stable`、`provisionalinternal`、或 `deprecated`

## side

以枚举形式建模，其值为：`eitherserver`、和 `client`

## 布尔属性

以下属性是布尔标志：

- `isFabricScoped`
- `isFabricSensitive`
- `mustUseAtomicWrite`
- `mustUseTimedInvoke`

## 字符串属性

以下属性表示为无界字符串：

- `cli`
- `cliFunctionName`
- `functionName`
- `group`
- `introducedIn`
- `manufacturerCode`
- `noDefaultImplementation`
- `presentIf`
- `priority`

- `removedIn`
- `reportableChange`
- `reportMinInterval`
- `reportMaxInterval`
- `restriction`
- `storage`

## 转型注意事项

对于 ZCL 变换 `extrinsicProperties`，无需处理即可存储在地图中。使用发现功能的自定义架构不会进行 ZCL 转换。但是，如果您计划将来为自定义架构实现 ZCL 转换，则必须对所有无界纯字符串类型进行建模，`extrinsicProperties` 并定义诸如枚举、模式（正则表达式）和长度之类的约束。这种准备工作可确保在转换过程中正确处理这些特性。

相比之下，对于连接器的 AWS 转换，`extrinsicProperties` 则根本不包括在内，因为连接器格式中不需要这些细节。

## 在能力架构文档中构建和使用类型定义

架构中的所有元素都解析为类型定义。这些类型定义要么是原始类型定义（例如布尔值、字符串、数字），要么是命名空间类型定义（为方便起见，根据原始类型定义构建的类型定义）。

定义自定义架构时，可以使用原始定义和命名空间类型定义。

### 目录

- [原始类型定义](#)
  - [布尔运算](#)
  - [整数类型支持](#)
  - [数字](#)
  - [字符串](#)
  - [null](#)
  - [数组](#)
  - [对象](#)
- [命名空间类型定义](#)
  - [matter 类型](#)

- [aws 类型](#)
  - [位图类型定义](#)
  - [枚举类型定义](#)

## 原始类型定义

原始类型定义是托管集成中定义的所有类型定义的基石。所有命名空间定义，包括自定义类型定义，都通过\$ref关键字或关键字解析为原始类型定义。type

使用关键字可以为空所有原始类型，并且可以使用nullable关键字来标识所有原始类型。type

### 布尔运算

布尔类型支持默认值。

示例定义：

```
{
 "type" : "boolean",
 "default" : "false",
 "nullable" : true
}
```

### 整数类型支持

整数类型支持以下内容：

- default 值
- maximum 值
- minimum 值
- exclusiveMaximum 值
- exclusiveMinimum 值
- multipleOf 值

如果x正在验证该值，则以下条件必须为真：

- $x \geq \text{minimum}$

- $x > \text{exclusiveMinimum}$
- $x < \text{exclusiveMaximum}$

### Note

小数部分为零的数字被视为整数，但不接受浮点数。

```
1.0 // Schema-Compliant
3.1415926 // NOT Schema-Compliant
```

虽然您可以同时指定`minimum`和`exclusiveMinimum` /或同时指定`maximum`和`exclusiveMaximum`，但我们不建议同时使用两者。

示例定义：

```
{
 "type" : "integer",
 "default" : 2,
 "nullable" : true,
 "maximum" : 10,
 "minimum" : 0,
 "multipleOf": 2
}
```

替代定义：

```
{
 "type" : "integer",
 "default" : 2,
 "nullable" : true,
 "exclusiveMaximum" : 11,
 "exclusiveMinimum" : -1,
 "multipleOf": 2
}
```

## 数字

将数字类型用于任何数值类型，包括整数和浮点数。

数字类型支持以下内容：

- default 值
- maximum 值
- minimum 值
- exclusiveMaximum 值
- exclusiveMinimum 值
- multipleOf 值。倍数可以是浮点数。

如果x正在验证该值，则以下条件必须为真：

- $x \geq \text{minimum}$
- $x > \text{exclusiveMinimum}$
- $x < \text{exclusiveMaximum}$

虽然您可以同时指定minimum和 exclusiveMinimum /或同时指定maximum和exclusiveMaximum，但我们不建议同时使用两者。

示例定义：

```
{
 "type" : "number",
 "default" : 0.4,
 "nullable" : true,
 "maximum" : 10.2,
 "minimum" : 0.2,
 "multipleOf": 0.2
}
```

替代定义：

```
{
 "type" : "number",
 "default" : 0.4,
 "nullable" : true,
 "exclusiveMaximum" : 10.2,
 "exclusiveMinimum" : 0.2,
 "multipleOf": 0.2
}
```

```
}
```

## 字符串

字符串类型支持以下内容：

- default 值
- 长度约束（必须是非负数），包括maxLength和minLength值
- pattern正则表达式的值

定义正则表达式时，如果该表达式与字符串中的任意位置匹配，则该字符串有效。例如，正则表达式p匹配任何包含p的字符串，例如“apple”，而不仅仅是字符串“p”。为清楚起见，我们建议在正则表达式周围加上`^...$`（例如，`^p$`），除非您有具体的理由不这样做。

示例定义：

```
{
 "type" : "string",
 "default" : "defaultString",
 "nullable" : true,
 "maxLength": 10,
 "minLength": 1,
 "pattern" : "^[0-9a-fA-F]{2})+$"
}
```

## null

Null 类型仅接受单个值：`null`。

示例定义：

```
{ "type": "null" }
```

## 数组

数组类型支持以下内容：

- default— 将用作默认值的列表。
- items— 对所有数组元素施加的 JSON 类型定义。

- 长度限制 ( 必须为非负数 )
  - minItems
  - maxItems
- pattern正则表达式的值
- uniqueItems— 一个布尔值，表示数组中的元素是否需要唯一
- prefixItems— 一个数组，其中每个项目都是一个架构，对应于文档数组的每个索引。也就是说，一个数组，其中第一个元素验证输入数组的第一个元素，第二个元素验证输入数组的第二个元素，依此类推。

示例定义：

```
{
 "type": "array",
 "default": ["1", "2"],
 "items" :{
 "type": "string",
 "pattern": "^[a-zA-Z0-9_ -/]+$"
 },
 "minItems" : 1,
 "maxItems": 4,
 "uniqueItems" : true,
}
```

数组验证示例：

```
//Examples:
["1", "2", "3", "4"] // Schema-Compliant
[] // NOT Schema-Compliant: minItems=1
["1", "1"] // NOT Schema-Compliant: uniqueItems=true
["{}"] // NOT Schema-Compliant: Does not match the RegEx pattern.
```

使用元组验证的替代定义：

```
{
 "type": "array",
 "prefixItems": [
 { "type": "number" },
 { "type": "string" },
 { "enum": ["Street", "Avenue", "Boulevard"] },
],
}
```

```
 { "enum": ["NW", "NE", "SW", "SE"] }
]
}

//Examples:
[1600, "Pennsylvania", "Avenue", "NW"] // Schema-Compliant

// And, by default, it's also okay to add additional items to end:
[1600, "Pennsylvania", "Avenue", "NW", "Washington"] // Schema-Compliant
```

## 对象

对象类型支持以下内容：

### • 属性限制

- `properties`— 使用关键字定义对象的属性（键值对）。`properties`的值是一个对象，其中每个键是属性的名称，每个值都是用于验证该属性的架构。任何与关键字中的任何属性名称都不匹配的属性将被该`properties`关键字忽略。
- `required`— 默认情况下，由`properties`关键字定义的属性不是必需的。但是，您可以使用`required`关键字提供所需属性的列表。该`required`关键字采用由零个或多个字符串组成的数组。这些字符串中的每一个都必须是唯一的。
- `propertyName`— 此关键字允许控制属性名称的 RegEx 模式。例如，您可能需要强制要求对象的所有属性都具有遵循特定惯例的名称。
- `patternProperties`— 这会将正则表达式映射到架构。如果属性名称与给定的正则表达式匹配，则该属性值必须根据相应的架构进行验证。例如，在给定特定类型的属性名称的情况下，使用`patternProperties`来指定值应与特定架构相匹配。
- `additionalProperties`— 此关键字控制如何处理额外属性。额外属性是指名称未在 `properties` 关键字中列出或与中的任何正则表达式匹配的属性`patternProperties`。默认情况下，允许使用其他属性。将此字段设置为`false`意味着不允许使用其他属性。
- `unevaluatedProperties`— 此关键字与类`additionalProperties`似，不同之处在于它可以识别子架构中声明的属性。`unevaluatedProperties`通过收集在处理架构时成功验证的所有属性并将其用作允许的属性列表来工作。这允许你做更复杂的事情，例如有条件地添加属性。有关更多详细信息，请参阅以下示例。
- `anyOf`— 此关键字的值必须为非空数组。数组中的每个项目都必须有效的 JSON 架构。如果实例成功针对该关键字的值定义的至少一个架构进行验证，则该实例成功地针对该关键字进行验证。
- `oneOf`— 此关键字的值必须为非空数组。数组中的每个项目都必须有效的 JSON 架构。如果实例仅针对该关键字的值定义的一个架构成功验证，则该实例成功地针对该关键字进行验证。

**必填示例：**

```
{
 "type": "object",
 "required": ["test"]
}

// Schema Compliant
{
 "test": 4
}

// NOT Schema Compliant
{}
```

**PropertyNames 示例：**

```
{
 "type": "object",
 "propertyNames": {
 "pattern": "^[A-Za-z_][A-Za-z0-9_]*$"
 }
}

// Schema Compliant
{
 "_a_valid_property_name_001": "value"
}

// NOT Schema Compliant
{
 "001 invalid": "value"
}
```

**PatternProperties 示例：**

```
{
 "type": "object",
 "patternProperties": {
 "^S_": { "type": "string" },
 "^I_": { "type": "integer" }
 }
}
```

```
}

// Schema Compliant
{ "S_25": "This is a string" }
{ "I_0": 42 }

// NOT Schema Compliant
{ "S_0": 42 } // Value must be a string
{ "I_42": "This is a string" } // Value must be an integer
```

### AdditionalProperties 示例 :

```
{
 "type": "object",
 "properties": {
 "test": {
 "type": "string"
 }
 },
 "additionalProperties": false
}

// Schema Compliant
{
 "test": "value"
}
OR
{}

// NOT Schema Compliant
{
 "notAllowed": false
}
```

### UnevaluatedProperties 示例 :

```
{
 "type": "object",
 "properties": {
 "standard_field": { "type": "string" }
 },
 "patternProperties": {
 "^@": { "type": "integer" } // Allows properties starting with '@'
 }
}
```

```
 },
 "unevaluatedProperties": false // No other properties allowed
}

// Schema Compliant
{
 "standard_field": "some value",
 "@id": 123,
 "@timestamp": 1678886400
}
// This passes because "standard_field" is evaluated by properties,
// "@id" and "@timestamp" are evaluated by patternProperties,
// and no other properties remain unevaluated.

// NOT Schema Compliant
{
 "standard_field": "some value",
 "another_field": "unallowed"
}
// This fails because "another_field" is unevaluated and doesn't match
// the @ pattern, leading to a violation of unevaluatedProperties: false
```

### AnyOf 示例 :

```
{
 "anyOf": [
 { "type": "string", "maxLength": 5 },
 { "type": "number", "minimum": 0 }
]
}

// Schema Compliant
"short"
12

// NOT Schema Compliant
"too long"
-5
```

### OneOf 示例 :

```
{
 "oneOf": [
```

```

 { "type": "number", "multipleOf": 5 },
 { "type": "number", "multipleOf": 3 }
]
}

// Schema Compliant
10
9

// NOT Schema compliant
2 // Not a multiple of either 5 or 3
15 // Multiple of both 5 and 3 is rejected.

```

## 命名空间类型定义

命名空间类型定义是根据原始类型构建的类型。这些类型必须遵循 `namespace.typeName`。托管集成在 `aws` 和 `matter` 命名空间下提供预定义类型的格式。除了保留空间 `aws` 和命名空间之外，您可以为自定义类型使用任何命名空间。

要查找可用的命名空间类型定义，请使用 `Type` 过滤器设置为 [ListSchemaVersions](#) API。 `definition`

### `matter` 类型

使用 [ListSchemaVersions](#) API 在 `matter` 命名空间下查找数据类型 `matter`，`Type` 筛选器设置为 `definition`。 `Namespace`

### `aws` 类型

使用 [ListSchemaVersions](#) API 在 `aws` 命名空间下查找数据类型 `aws`，`Type` 筛选器设置为 `definition`。 `Namespace`

## 位图类型定义

位图有两个必需的属性：

- `type` 必须是对象
- `properties` 必须是包含每个位定义的对象。每个位都是一个具有属性 `extrinsicId` 和 `value` 的对象。每个位的值必须是一个整数，最小值为 0，最大值至少为 1。

位图定义示例：

```
{
```

```
"title" : "Sample Bitmap Type",
"description" : "Type definition for SampleBitmap.",
"$ref" : "/schema-versions/definition/aws.bitmap@1.0 ",
"type" : "object",
"additionalProperties" : false,
"properties" : {
 "Bit1" : {
 "extrinsicId" : "0x0000",
 "value" : {
 "type" : "integer",
 "maximum" : 1,
 "minimum" : 0
 }
 },
 "Bit2" : {
 "extrinsicId" : "0x0001",
 "value" : {
 "type" : "integer",
 "maximum" : 1,
 "minimum" : 0
 }
 }
}
}

// Schema Compliant
{
 "Bit1": 1,
 "Bit1": 0
}

// NOT Schema Compliant
{
 "Bit1": -1,
 "Bit1": 0
}
```

## 枚举类型定义

枚举需要三个属性：

- type必须是对象
- enum必须是一个由唯一字符串组成的数组，至少包含一个项目

- `extrinsicIdMap`是一个对象，其属性是枚举值。每个属性的值都应该是与枚举值相对应的外部标识符。

枚举定义示例：

```
{
 "title" : "SampleEnum Type",
 "description" : "Type definition for SampleEnum.",
 "$ref" : "/schema-versions/definition/aws.enum@1.0",
 "type" : "string",
 "enum" : [
 "EnumValue0",
 "EnumValue1",
 "EnumValue2"
],
 "extrinsicIdMap" : {
 "EnumValue0" : "0",
 "EnumValue1" : "1",
 "EnumValue2" : "2"
 }
}

// Schema Compliant
"EnumValue0"
"EnumValue1"
"EnumValue2"

// NOT Schema Compliant
"NotAnEnumValue"
```

# 管理 IoT 设备命令和事件

设备命令提供了远程管理物理设备的功能，除了执行关键的安全、软件和硬件更新外，还可确保对设备的完全控制。对于庞大的设备群，知道设备何时执行命令可以对整个设备实现进行监督。设备命令或自动更新将触发设备状态更改，这反过来又会创建新的设备事件。此设备事件将触发自动发送到客户管理的目的地的通知。

主题

- [设备命令](#)
- [设备事件](#)

## 设备命令

命令请求是向设备发送的命令。命令请求包含一个有效负载，用于指定要执行的操作，例如打开灯泡。要发送设备命令，托管集成代表最终用户调用 `SendManagedThingCommand` API，然后将命令请求发送到设备。

对 `a` 的响应 `SendManagedThingCommand` 是 `atraceId`，你可以尽可能使用它 `traceId` 来跟踪命令交付和任何相关的命令响应工作流程。

有关 `SendManagedThingCommand` API 操作的更多信息，请参阅 [SendManagedThingCommand](#)。

### UpdateState 操作

要更新设备的状态，例如灯光亮起的时间，请在调用 `SendManagedThingCommand` API 时使用 `UpdateState` 操作。提供要更新的数据模型属性和新值 `parameters`。以下示例说明了将灯泡更新 `OnTime` 为 `SendManagedThingCommand` API 请求 5。

```
{
 "Endpoints": [
 {
 "endpointId": "1",
 "capabilities": [
 {
 "id": "matter.OnOff",
 "name": "On/Off",
 "version": "1",
 "actions": [
```

```
 {
 "name": "UpdateState",
 "parameters": {
 "OnTime": 5
 }
 }
]
}
]
```

## ReadState 操作

要获取设备的最新状态，包括所有数据模型属性的当前值，请在调用 `SendManagedThingCommand` API 时使用 `ReadState` 操作。在中 `propertiesToRead`，您可以使用以下选项：

- 提供特定的数据模型属性以获取最新值，例如 `OnOff` 确定灯是开还是关。
- 使用通配符运算符 (\*) 读取某项功能的所有设备状态属性。

以下示例说明了使用 `ReadState` 操作进行 `SendManagedThingCommand` API 请求的两种场景：

```
{
 "Endpoints": [
 {
 "endpointId": "1",
 "capabilities": [
 {
 "id": "aws.OnOff",
 "name": "On/Off",
 "version": "1",
 "actions": [
 {
 "name": "ReadState",
 "parameters": {
 "propertiesToRead": ["OnOff"]
 }
 }
]
 }
]
 }
]
}
```

```
 }
]
}
```

```
{
 "Endpoints": [
 {
 "endpointId": "1",
 "capabilities": [
 {
 "id": "aws.OnOff",
 "name": "On/Off",
 "version": "1",
 "actions": [
 {
 "name": "ReadState",
 "parameters": {
 "propertiesToRead": ["*"]
 }
 }
]
 }
]
 }
]
}
```

## 设备事件

设备事件包括设备的当前状态。这可能意味着设备已更改状态，或者即使状态未更改，也正在报告其状态。它包括在数据模型中定义的属性报告和事件。事件可能是洗衣机循环已完成，或者恒温器已达到最终用户设定的目标温度。

### 设备事件通知

最终用户可以订阅他们为更新特定设备事件而创建的特定客户管理的目的地。要创建客户管理的目的地，请调用 `CreateDestination` API。当设备向托管集成报告设备事件时，如果存在设备事件，则会通知客户管理的目的地。

# 标记您的托管集成资源

为了帮助您管理和组织资源，您可以选择以标签的形式为每个资源分配自己的元数据。本部分介绍标签并说明如何创建标签。

## 标签基本知识

您可以使用标签以不同的方式（例如，按用途、所有者或环境）对托管集成资源进行分类。这在您具有相同类型的许多资源时会很有用 - 您可以根据分配给资源的标签快速识别资源。每个标签都包含您定义的一个键和一个可选值。例如，您可以为事物类型定义一组标签来帮助您按类型跟踪设备。我们建议您为每类资源创建一组可满足您的需求的需求的标签键。使用一组连续的标签键，管理资源时会更加轻松。

您可以根据添加或应用的标签搜索和筛选资源。您还可以使用标签控制对资源的访问，如 [在 IAM 策略中使用标签](#) 中所述。

为便于使用，AWS 管理控制台中的标签编辑器提供了一种集中、统一的方式来创建和管理标签。有关更多信息，请参阅 [使用AWS 管理控制台中的使用标签编辑器](#)。

您也可以使用 AWS CLI 和托管集成 API 来处理标签。在创建标签时，您可以使用以下命令中的 Tags 字段将标签与托管事物、配置配置文件、凭证储物柜和 over-the-air (OTA) 任务相关联：

- [CreateManagedThing](#)
- [CreateProvisioningProfile](#)
- [CreateCredentialLocker](#)
- [CreateOtaTask](#)
- [CreateAccountAssociation](#)

您可以使用以下命令为支持标记的现有资源添加、修改或删除标签：

- [TagResource](#)
- [ListTagsForResource](#)
- [UntagResource](#)

您可以修改标签的键和值，还可以随时删除资源的标签。您可以将标签的值设为空的字符串，但是不能将其设为空值。如果添加的标签的键与该资源上现有标签的键相同，新值就会覆盖旧值。如果删除资源，则所有与资源相关的标签都将被删除。

## 标签限制

下面是适用于 标签的基本限制：

- 每个资源的最大标签数 - 50
- 最大密钥长度 - 127 个 Unicode 字符 ( 采用 UTF-8 格式 )
- 最大值长度 - 255 个 Unicode 字符 ( 采用 UTF-8 格式 )
- 标签键和值区分大小写。
- 请勿在标签名称或值中使用 `aws:` 前缀。它是保留供 AWS 使用的。您无法编辑或删除带此前缀的标签名称或值。具有此前缀的标签不计入每个资源的标签数限制。
- 如果在多个服务和资源中使用您的标记方案，请记住，其他服务可能对允许使用的字符有限制。允许使用的字符包括：可用 UTF-8 格式表示的字母、空格和数字以及以下特殊字符：`+ - = . _ : / @`。

## 在 IAM 策略中使用标签

您可以在用于托管集成 API 操作的 IAM 策略中应用基于标签的资源级权限。这可让您更好地控制用户可创建、修改或使用哪些资源。在 IAM 策略中将 Condition 元素 ( 也称作 Condition 块 ) 与以下条件上下文键和值结合使用来基于资源标签控制用户访问 ( 权限 )：

- 使用 `aws:ResourceTag/tag-key: tag-value` 可允许或拒绝带特定标签的资源上的用户操作。
- 使用 `aws:RequestTag/tag-key: tag-value` 可要求在发出创建或修改允许标签的资源的 API 请求时使用 ( 或不使用 ) 特定标签。
- 使用 `aws:TagKeys: [tag-key, ...]` 可要求在发出创建或修改允许标签的资源的 API 请求时使用 ( 或不使用 ) 一组特定标签键。

### Note

IAM 策略中的条件上下文密钥和值仅适用于那些托管集成操作，其中能够被标记的资源的标识符是必填参数。例如，根据条件上下文密钥和值，不允许或拒绝使用，因为此请求中未引用任何可标记的资源 ( 托管事物、配置配置文件、凭证储物柜、over-the-air任务 )。阅读[GetCustomEndpoint](#)有关可标记的托管集成资源及其支持的条件键的更多信息，请阅读[AWS IoT 托管集成的操作、资源和条件密钥](#)功能。AWS IoT Device Management

有关使用标签的更多信息，请参阅《AWS Identity and Access Management 用户指南》中的[使用标签控制访问权限](#)。该指南的[IAM JSON 策略参考](#)一部分包含 IAM 中的 JSON 策略的元素、变量和评估逻辑的详细语法、描述和示例。

以下示例策略对CreateManagedThing操作应用了两个基于标签的限制。受此策略限制的 IAM 用户：

- 无法创建带有“env=prod”标签的托管事物（在示例中，参见该行）。“aws:RequestTag/env”：“prod”
- 无法修改或访问带有现有标签“env=prod”的托管事物（在示例中，参见该行）。“aws:ResourceTag/env”：“prod”

## JSON

```
{
 "Version": "2012-10-17",
 "Statement": [
 {
 "Effect": "Deny",
 "Action": "iotmanagedintegrations:CreateManagedThing",
 "Resource": "arn:aws:iotmanagedintegrations:us-east-1:123456789012:managed-thing/*",
 "Condition": {
 "StringEquals": {
 "aws:RequestTag/env": "prod"
 }
 }
 },
 {
 "Effect": "Deny",
 "Action": [
 "iotmanagedintegrations:CreateManagedThing",
 "iotmanagedintegrations>DeleteManagedThing",
 "iotmanagedintegrations:GetManagedThing",
 "iotmanagedintegrations:UpdateManagedThing"
],
 "Resource": "arn:aws:iotmanagedintegrations:us-east-1:123456789012:managed-thing/*",
 "Condition": {
 "StringEquals": {
 "aws:ResourceTag/env": "prod"
 }
 }
 }
]
}
```

```
 }
 }
},
{
 "Effect": "Allow",
 "Action": [
 "iotmanagedintegrations:CreateManagedThing",
 "iotmanagedintegrations>DeleteManagedThing",
 "iotmanagedintegrations:GetManagedThing",
 "iotmanagedintegrations:UpdateManagedThing"
],
 "Resource": "*"
}
]
```

您还可以为给定标签键指定多个标签值，方法是将它们括在列表中，如下所示：

```
"StringEquals" : {
 "aws:ResourceTag/env" : ["dev", "test"]
}
```

#### Note

如果您基于标签允许或拒绝用户访问资源，则必须考虑显式拒绝用户对相同资源添加或删除这些标签的能力。否则，用户可能通过修改资源标签来绕过您的限制并获得资源访问权限。

# 托管集成通知

托管集成通知可提供来自设备的更新和关键见解。通知包括连接器事件、设备命令、生命周期事件、OTA ( Over-the-Air ) 更新和错误报告。这些见解提供了可操作的信息，用于创建自动化工作流程、立即采取行动或存储事件数据以进行故障排除。

目前，仅支持 Amazon Kinesis 数据流作为托管集成通知的目的地。在设置通知之前，您首先需要设置 Amazon Kinesis 数据流并允许托管集成访问该数据流。

## 设置 Amazon Kinesis 以接收通知

亚马逊 Kinesis 设置步骤

- [第 1 步：创建 Amazon Kinesis 数据流](#)
- [步骤 2：创建权限策略](#)
- [步骤 3：导航到 IAM 控制面板并选择角色](#)
- [步骤 4：使用自定义信任策略](#)
- [第 5 步：应用您的权限策略](#)
- [步骤 6：输入角色名称](#)

要将 Amazon Kinesis 设置为接收托管集成通知，请按照以下步骤操作：

### 第 1 步：创建 Amazon Kinesis 数据流

Amazon Kinesis 数据流可以实时摄取大量数据，持久存储数据，并使数据可供应用程序使用。

创建 Amazon Kinesis 数据流

- 要创建 Kinesis 数据流，请按照[创建和管理 Kinesis 数据流](#)中概述的步骤进行操作。

### 步骤 2：创建权限策略

创建权限策略，允许托管集成访问您的 Kinesis 数据流。

创建权限策略

- 要创建权限策略，请复制以下策略，然后按照[使用 JSON 编辑器创建策略中概述的](#)步骤进行操作

## JSON

```
{
 "Version": "2012-10-17",
 "Statement": [
 {
 "Action": "kinesis:PutRecord",
 "Resource": "*",
 "Effect": "Allow"
 }
]
}
```

### 步骤 3：导航到 IAM 控制面板并选择角色

打开 IAM 控制面板，然后单击“角色”。

导航到 IAM 控制面板

- 打开 IAM 控制面板，然后单击“角色”。

有关更多信息，请参阅《AWS Identity and Access Management 用户指南》中的 [IAM 角色创建](#)。

### 步骤 4：使用自定义信任策略

您可以使用自定义信任策略向托管集成授予对 Kinesis 数据流的访问权限。

使用自定义信任策略

- 创建新角色并选择自定义信任策略。单击“下一步”。

以下政策允许托管集成担任该角色，该Condition声明有助于防止混淆副手问题。

## JSON

```
{
 "Version": "2012-10-17",
 "Statement": [
```

```
{
 "Effect": "Allow",
 "Principal": {
 "Service": "iotmanagedintegrations.amazonaws.com"
 },
 "Action": "sts:AssumeRole",
 "Condition": {
 "StringEquals": {
 "aws:SourceAccount": "123456789012"
 },
 "ArnLike": {
 "aws:SourceArn": "arn:aws:iotmanagedintegrations:ca-
central-1:123456789012:*"
 }
 }
}
```

## 第 5 步：应用您的权限策略

将您在步骤 2 中创建的权限策略添加到角色中。

### 添加权限策略

- 在添加权限页面上，搜索并添加您在步骤 2 中创建的权限策略。单击“下一步”。

## 步骤 6：输入角色名称

- 输入角色名称，然后单击“创建角色”。

## 设置托管集成通知

### 通知设置步骤

- [第 1 步：向用户授予调用 CreateDestination API 的权限](#)
- [第 2 步：调用 CreateDestination API](#)
- [第 3 步：调用 CreateNotificationConfiguration API](#)

要设置托管集成通知，请按照以下步骤操作：

## 第 1 步：向用户授予调用 CreateDestination API 的权限

- 向用户授予调用 **CreateDestination** API 的权限

以下策略定义了用户调用 [CreateDestination](#) API 的要求。

[要获取托管集成的passrole权限，请参阅AWS Identity and Access Management用户指南中的授予用户将角色传递给 AWS 服务的权限。](#)

JSON

```
{
 "Version": "2012-10-17",
 "Statement": [
 {
 "Effect": "Allow",
 "Action": "iam:PassRole",
 "Resource": "arn:aws:iam::123456789012:role/ROLE_CREATED_IN_PREVIOUS_STEP",
 "Condition": {
 "StringEquals": {
 "iam:PassedToService": "iotmanagedintegrations.amazonaws.com"
 }
 }
 },
 {
 "Effect": "Allow",
 "Action": "iotmanagedintegrations:CreateDestination",
 "Resource": "*"
 }
]
}
```

## 第 2 步：调用 CreateDestination API

- 调用 **CreateDestination** API

创建 Amazon Kinesis 数据流和流访问角色后，调用 [CreateDestination](#) API 来创建通知将发送到的通知目的地。对于 `DeliveryDestinationArn` 参数，请使用您的新 Amazon Kinesis 数据流中的 `arn`

```
{
 "DeliveryDestinationArn": "Your Kinesis arn"
 "DeliveryDestinationType": "KINESIS"
 "Name": "DestinationName"
 "ClientToken": "string"
 "RoleArn": "arn:aws:iam::accountID:role/ROLE_CREATED_IN_PREVIOUS_STEP"
}
```

#### Note

`ClientToken` 是一个等性标记。如果您使用相同的客户端令牌和参数重试最初成功完成的请求，则重试尝试将成功而无需执行任何进一步的操作。

## 第 3 步：调用 `CreateNotificationConfiguration` API

- 调用 `CreateNotificationConfiguration` API

最后，使用 [CreateNotificationConfiguration](#) API 创建通知配置，将所选事件类型路由到由 Kinesis 数据流表示的目的地。在 `DestinationName` 参数中，使用与最初调用 `CreateDestination` API 时相同的目标名称。

```
{
 "EventType": "DEVICE_EVENT"
 "DestinationName" // This name has to be identical to the name in
createDestination API
 "ClientToken": "string"
}
```

## 使用托管集成监控的事件类型

以下是使用托管集成通知监控的事件类型：

- `DEVICE_COMMAND`

- [SendManagedThingCommand](#) API 命令的状态。有效值为 succeeded 或 failed。

```
{
 "version": "0",
 "messageId": "6a7e8feb-b491-4cf7-a9f1-bf3703467718",
 "messageType": "DEVICE_COMMAND",
 "source": "aws.iotmanagedintegrations",
 "customerAccountId": "123456789012",
 "timestamp": "2017-12-22T18:43:48Z",
 "region": "ca-central-1",
 "resources": [
 "arn:aws:iotmanagedintegrations:ca-
central-1:123456789012:managed-thing/6a7e8feb-b491-4cf7-a9f1-bf3703467718"
],
 "payload": {
 "traceId": "1234567890abcdef0",
 "receivedAt": "2017-12-22T18:43:48Z",
 "executedAt": "2017-12-22T18:43:48Z",
 "result": "failed"
 }
}
```

- DEVICE\_COMMAND\_REQUEST
  - 来自网络实时通信 (WebRTC) 的命令请求。

WebRTC标准允许两个对等方之间进行通信。这些对等体可以传输实时视频、音频和任意数据。托管集成支持WebRTC，以便在客户的移动应用程序和最终用户的设备之间实现这些类型的流式传输。[有关 WebRTC 标准的更多信息，请参阅 WebRTC。](#)

```
{
 "version": "0",
 "messageId": "6a7e8feb-b491-4cf7-a9f1-bf3703467718",
 "messageType": "DEVICE_COMMAND_REQUEST",
 "source": "aws.iotmanagedintegrations",
 "customerAccountId": "123456789012",
 "timestamp": "2017-12-22T18:43:48Z",
 "region": "ca-central-1",
 "resources": [
 "arn:aws:iotmanagedintegrations:ca-
central-1:123456789012:managed-thing/6a7e8feb-b491-4cf7-a9f1-bf3703467718"
],
 "payload": {
```

```

 "endpoints": [{
 "endpointId": "1",
 "capabilities": [{
 "id": "aws.DoorLock",
 "name": "Door Lock",
 "version": "1.0"
 }]
 }]
 }
}

```

- **DEVICE\_DISCOVERY\_STATUS**

- 设备的发现状态。

```

{
 "version": "0",
 "messageId": "6a7e8feb-b491-4cf7-a9f1-bf3703467718",
 "messageType": "DEVICE_DISCOVERY_STATUS",
 "source": "aws.iotmanagedintegrations",
 "customerAccountId": "123456789012",
 "timestamp": "2017-12-22T18:43:48Z",
 "region": "ca-central-1",
 "resources": [
 "arn:aws:iotmanagedintegrations:ca-central-1:123456789012:managed-thing/6a7e8feb-b491-4cf7-a9f1-bf3703467718"
],
 "payload": {
 "deviceCount": 1,
 "deviceDiscoveryId": "123",
 "status": "SUCCEEDED"
 }
}

```

- **DEVICE\_EVENT**

- 设备事件发生的通知。

```

{
 "version": "1.0",
 "messageId": "2ed545027bd347a2b855d28f94559940",
 "messageType": "DEVICE_EVENT",
 "source": "aws.iotmanagedintegrations",
 "customerAccountId": "123456789012",
 "timestamp": "1731630247280",

```

```

 "resources": [
 "/quit/1b15b39992f9460ba82c6c04595d1f4f"
],
 "payload": {
 "endpoints": [{
 "endpointId": "1",
 "capabilities": [{
 "id": "aws.DoorLock",
 "name": "Door Lock",
 "version": "1.0",
 "properties": [{
 "name": "ActuatorEnabled",
 "value": "true"
 }]
 }]
 }]
 }
 }
}

```

- **DEVICE\_LIFE\_CYCLE**
- 设备生命周期的状态。

```

{
 "version": "1.0.0",
 "messageId": "8d1e311a473f44f89d821531a0907b05",
 "messageType": "DEVICE_LIFE_CYCLE",
 "source": "aws.iotmanagedintegrations",
 "customerAccountId": "123456789012",
 "timestamp": "2024-11-14T19:55:57.568284645Z",
 "region": "ca-central-1",
 "resources": [
 "arn:aws:iotmanagedintegrations:ca-central-1:123456789012:managed-thing/d5c280b423a042f3933eed09cf408657"
],
 "payload": {
 "deviceDetails": {
 "id": "d5c280b423a042f3933eed09cf408657",
 "arn": "arn:aws:iotmanagedintegrations:ca-central-1:123456789012:managed-thing/d5c280b423a042f3933eed09cf408657",
 "createdAt": "2024-11-14T19:55:57.515841147Z",
 "updatedAt": "2024-11-14T19:55:57.515841559Z"
 },
 "status": "UNCLAIMED"
 }
}

```

```
}
}
```

- DEVICE\_OTA
  - 设备的 OTA 通知。
- DEVICE\_STATE
  - 设备状态更新时的通知。

```
{
 "messageType": "DEVICE_STATE",
 "source": "aws.iotmanagedintegrations",
 "customerAccountId": "123456789012",
 "timestamp": "1731623291671",
 "resources": [
 "arn:aws:iotmanagedintegrations:ca-central-1:123456789012:managed-
thing/61889008880012345678"
],
 "payload": {
 "addedStates": {
 "endpoints": [{
 "endpointId": "nonEndpointId",
 "capabilities": [{
 "id": "aws.OnOff",
 "name": "On/Off",
 "version": "1.0",
 "properties": [{
 "name": "OnOff",
 "value": {
 "propertyValue": "\"onoff\"",
 "lastChangedAt": "2024-06-11T01:38:09.000414Z"
 }
 }
]
 }
]
 }
}]
}]
}
```

# Cloud-to-Cloud (C2C) 连接器

cloud-to-cloud连接器允许您创建和促进第三方设备与之间的双向通信。AWS

主题

- [什么是 cloud-to-cloud \(C2C\) 连接器？](#)
- [C2C 连接器目录是什么？](#)
- [AWS Lambda 用作 C2C 连接器](#)
- [托管集成连接器工作流程](#)
- [使用 C2C \(cloud-to-cloud\) 连接器的指导原则](#)
- [构建 C2C \(云到云\) 连接器](#)
- [使用 C2C \(云到云\) 连接器](#)

## 什么是 cloud-to-cloud (C2C) 连接器？

cloud-to-cloud连接器是一种预先构建的软件包，可将安全地链接 AWS Cloud 到第三方云提供商的端点。使用 C2C 连接器，解决方案提供商可以利用 AWS IoT Device Management 的托管集成来控制连接到第三方云的设备。

托管集成包括连接器目录，AWS 客户可以在其中查看和选择他们想要集成的连接器。有关更多信息，请参阅 [C2C 连接器目录是什么？](#)。

托管集成要求将每个连接器作为一个 AWS Lambda 功能来实现。

## C2C 连接器目录是什么？

AWS IoT Device Management 的托管集成连接器目录是一系列 C2C 连接器，可促进 AWS IoT Device Management 的托管集成与第三方云提供商之间的双向通信。您可以在 AWS 管理控制台 或中查看连接器 AWS CLI。

使用控制台查看托管集成连接器目录

1. 打开[托管集成控制台](#)
2. 在左侧导航窗格中，选择托管集成
3. 在托管集成控制台的左侧导航窗格中，选择目录。

## AWS Lambda 用作 C2C 连接器

每个 C2C 连接器 Lambda 函数在托管集成和第三方平台上的相应操作之间转换和传输命令和事件。有关 Lambda 的更多信息，请参阅[什么是](#)。AWS Lambda

例如，假设最终用户拥有由第三方 OEM 制造的智能灯泡。借助 C2C 连接器，最终用户可以通过托管集成平台发出开启或关闭此灯的命令。然后，该命令将被转发到连接器中托管的 Lambda 函数，该函数会将请求转换为针对第三方平台的 API 调用，以打开或关闭设备。

当您调用 API 时，需要使用 Lambda 函数。CreateCloudConnector 部署到 Lambda 函数中的代码必须实现中提到的所有接口和功能。[构建 C2C \(云到云\) 连接器](#)

### 托管集成连接器工作流程

开发人员必须使用托管集成注册 C2C 连接器。AWS IoT Device Management 此注册过程会创建逻辑连接器资源，客户可以访问该资源以使用该连接器。

#### Note

C2C 连接器是在托管集成中为 AWS IoT Device Management 创建的一组元数据，用于描述连接器。

下图描述了 C2C 连接器在将命令从移动应用程序发送到连接云的设备时所扮演的角色。C2C 连接器充当 AWS IoT Device Management 托管集成和第三方云平台之间的转换层。

### 使用 C2C (cloud-to-cloud) 连接器的指导原则

您创建的任何 C2C 连接器都是您的内容，而您访问的由其他客户创建的任何 C2C 连接器都是第三方内容。AWS 不创建或管理任何 C2C 连接器作为托管集成的一部分。

您可以与其他托管集成客户共享您的 C2C 连接器。如果您这样做，则您授权 AWS 作为您的服务提供商在 AWS 控制台上列出这些 C2C 连接器和相关联系信息，并且您知道其他 AWS 客户可能会与您联系。您全权负责授予客户访问您的 C2C 连接器的权限，以及管理其他 AWS 客户访问您的 C2C 连接器的任何条款。

# 构建 C2C ( 云到云 ) 连接器

以下各节介绍为 AWS IoT Device Management 的托管集成构建 C2C ( 云到云 ) 连接器的步骤。

## 主题

- [先决条件](#)
- [C2C 连接器要求](#)
- [OAuth 2.0 账户关联要求](#)
- [实现 C2C 连接器接口操作](#)
- [调用你的 C2C 连接器](#)
- [为您的 IAM 角色添加权限](#)
- [手动测试您的 C2C 连接器](#)

## 先决条件

在创建 C2C ( 云到云 ) 连接器之前，您需要满足以下条件：

- AWS 账户 用于托管您的 C2C 连接器并通过托管集成进行注册。有关更多信息，请参阅[创建 AWS 账户](#)。
- 在构建连接器时，您需要某些 IAM 权限。要再次使用
- 确保连接器所针对的第三方云提供商支持 OAuth 2.0 授权。有关更多信息，请参阅[OAuth 2.0 账户关联要求](#)。

此外，要测试连接器，连接器的开发人员必须具备以下条件：

- 来自第三方云的客户端 ID，用于与 C2C 连接器关联
- 来自第三方云的客户端密钥，用于与您的 C2C 连接器关联
- OAuth 2.0 授权网址
- OAuth 2.0 代币网址
- 您的第三方 API 所需的任何 API 密钥
- 您的第三方 API 注册或托管的 OAuth 回传网址许可名单所需的任何 API 密钥。AWS 一些第三方明确 OAuth 将重定向 URL 列入许可名单，而另一些第三方则有用户可以登录和注册该 OAuth URL 的工作流程。请咨询特定的第三方，了解将托管集成 OAuth 重定向端点列入许可名单需要什么

## 所需的权限

在构建连接器时，您需要某些 IAM 权限。除了操作 `iotmanagedintegrations`: 权限外，您还需要以下权限：

- [CreateAccountAssociation](#)、[CreateConnectorDestinationGetAccountAssociation](#)、和 [StartAccountAssociationRefresh](#)，要求 `secretsmanager:GetSecretValue`
- [CreateCloudConnector](#) 需要 `lambda:Invoke`

有关 `iotmanagedintegrations`: 权限和操作的更多信息，请参阅 [AWS 托管集成定义的操作](#)

## C2C 连接器要求

您开发的 [C2C 连接器](#) 促进了 AWS IoT Device Management 的托管集成与第三方供应商云之间的双向通信。连接器必须实现托管集成的接口，AWS IoT Device Management 才能代表最终用户执行操作。这些接口提供了发现最终用户设备、启动从 AWS IoT Device Management 托管集成发送的设备命令以及基于访问令牌识别用户的功能。为了支持设备操作，连接器必须管理 AWS IoT Device Management 托管集成与相关第三方平台之间的请求和响应消息的转换。

以下是 C2C 连接器的要求：

- 第三方授权服务器必须符合 OAuth 2.0 标准以及中列出的配置 [OAuth 配置要求](#)。
- 需要使用 C2C 连接器来解释物质数据模型 AWS 实现中的标识符，并且必须发出符合物质数据模型 AWS 实现的响应和事件。有关更多信息，请参阅 [AWS 物质数据模型的实现](#)。
- C2C 连接器必须能够通过身份验证调用 AWS IoT Device Management 的托管集成。SigV4 对于 `SendConnectorEvent` 通过 API 发送的异步事件，必须使用用于注册连接器的相同 AWS 账户凭据来签署相关 `SendConnectorEvent` 请求。
- 连接器必须实现 [AWS.ActivateUser](#)、[AWS.DiscoverDevices](#)、[AWS.SendCommand](#)、和 [AWS.DeactivateUser](#) 操作。
- 当您的 C2C 连接器收到与设备命令响应或设备发现相关的第三方事件时，它必须将其转发到与 API 的托管集成。`SendConnectorEvent` 有关这些事件和 `SendConnectorEvent` API 的更多信息，请参阅 [SendConnectorEvent](#)。

### Note

`SendConnectorEvent` API 是托管集成 SDK 的一部分，用于代替手动构建和签署请求。

## OAuth 2.0 账户关联要求

每个 C2C 连接器都依赖 OAuth 2.0 授权服务器对最终用户进行身份验证。通过此服务器，最终用户将其第三方账户与客户的设备平台关联起来。账户关联是最终用户使用 C2C 连接器支持的设备所需的第一步。有关账户关联和 OAuth 2.0 中不同角色的更多信息，请参阅[账户关联角色](#)。

虽然您的 C2C 连接器不需要实现特定的业务逻辑来支持授权流程，但与 C2C 连接器关联的 OAuth2.0 授权服务器必须满足。[OAuth 配置要求](#)

### Note

AWS IoT Device Management 仅适用于的托管集成支持带有授权代码流的 OAuth 2.0。有关更多信息，请参阅[RFC 6749](#)。

账户关联是一个允许托管集成和连接器使用访问令牌访问最终用户设备的过程。此令牌为最终用户许可的 AWS IoT Device Management 提供托管集成，以便连接器可以通过 API 调用与最终用户的数据进行交互。有关更多信息，请参阅[账户关联工作流程](#)。

我们建议您不要将这些敏感令牌记录在任何日志中。但是，如果它们存储在日志中，我们建议您使用 CloudWatch 日志数据保护策略来屏蔽日志中的令牌。有关更多信息，请参阅[Help protect sensitive log data with masking](#)。

的托管集成 AWS IoT Device Management 不会直接获得访问令牌；而是通过授权码授权类型获得访问令牌。首先，AWS IoT Device Management 的托管集成必须获得授权码。然后，它将代码交换为访问令牌和刷新令牌。刷新令牌用于在旧访问令牌过期时请求新的访问令牌。如果访问令牌和刷新令牌都已过期，则必须再次执行账户关联流程。你可以通过 StartAccountAssociationRefresh API 操作来做到这一点。

### Important

发放的访问令牌的作用域必须按用户划分，但不能按 OAuth 客户机划分。该令牌不应提供对客户端下所有用户的所有设备的访问权限。

授权服务器必须执行以下操作之一：

- 发行包含可提取的最终用户（资源所有者）ID 的访问令牌，例如 JWT-Token。
- 返回每个已颁发的访问令牌的最终用户 ID。

## OAuth 配置要求

下表说明了 OAuth 授权服务器中用于托管集成 AWS IoT Device Management 以执行[账户关联](#)所需的参数：

### OAuth 服务器参数

| 字段                                | 必填 | 评论                                                                                         |
|-----------------------------------|----|--------------------------------------------------------------------------------------------|
| clientId                          | 是  | 您的应用程序的公共标识符。它用于启动身份验证流程，并且可以公开共享。                                                         |
| clientSecret                      | 是  | 用于通过授权服务器对应用程序进行身份验证的密钥，尤其是在使用授权码交换访问令牌时。应将其保密，不得公开共享。                                     |
| authorizationType                 | 是  | 此授权配置支持的授权类型。当前，“OAuth 2.0”是唯一支持的值。                                                        |
| authUrl                           | 是  | 第三方云提供商的授权 URL。                                                                            |
| tokenUrl                          | 是  | 第三方云提供商的令牌 URL。                                                                            |
| tokenEndpointAuthenticationScheme | 是  | “HTTP_BASIC”或“REQUEST_BODY_CREDENTIALS”的身份验证方案。HTTP_BASIC 表示客户端凭证包含在授权标头中，而阶梯表示它们包含在请求正文中。 |

必须对您使用的 OAuth 服务器进行配置，使访问令牌字符串值必须使用 UTF-8 字符集进行 Base64 编码。

## 账户关联角色

要创建 C2C 连接器，你需要一个 OAuth 2.0 授权服务器和账户关联。有关更多信息，请参阅 [账户关联工作流程](#)。

OAuth 2.0 在实现账户关联时定义了以下四个角色：

1. 授权服务器
2. 资源所有者 ( 最终用户 )
3. 资源服务器
4. 客户端

以下内容定义了这些 OAuth 角色中的每一个：

### 授权服务器

授权服务器是识别和验证第三方云中最终用户身份的服务器。此服务器提供的访问令牌可以将 AWS 最终用户的客户平台帐户与其第三方平台帐户关联起来。此过程称为账户关联。

授权服务器通过提供以下内容来支持账户关联：

- 显示登录页面，供最终用户登录您的系统。这通常被称为授权端点。
- 对系统中的最终用户进行身份验证。
- 生成用于识别最终用户的授权码。
- 将授权码传递给 AWS IoT Device Management 的托管集成。
- 接受 AWS IoT Device Management 托管集成的授权码，并返回访问令牌，AWS IoT Device Management 的托管集成可用于访问系统中最终用户的数据。这通常通过单独的 URI ( 称为令牌 URI 或端点 ) 完成。

#### Important

授权服务器必须支持 OAuth 2.0 授权码流程，才能与 AWS IoT Device Management Connector 的托管集成一起使用。AWS IoT Device Management 的托管集成还支持带有 [代码交换证明密钥 \(PKCE\) 的授权代码流](#)。

授权服务器必须：

- 发布包含可提取的最终用户或资源所有者 ID 的访问令牌，例如 jwt-Tokens
- 能够返回每个已发放的访问令牌的最终用户 ID

否则，您的连接器将无法支持所需的 `AWS.ActivateUser` 操作。这将防止在托管集成中使用连接器。

如果连接器开发者或所有者没有维护自己的授权服务器，则使用的授权服务器必须为连接器开发者第三方平台管理的资源提供授权。这意味着托管集成从授权服务器接收的任何令牌都必须在设备（资源）上提供有意义的安全边界。例如，最终用户令牌不允许在另一台最终用户设备上执行命令；该令牌提供的权限会映射到平台内的资源。以 Lights 公司为例。当最终用户启动与其连接器的账户关联流程时，他们将被重定向到授权服务器正面的 Lights Incorporated 登录页面。一旦他们登录并向客户端授予权限，他们就会提供一个令牌，允许连接器访问其 Lights Incorporated 账户中的资源。

### 资源所有者（最终用户）

作为资源所有者，您可以通过执行账户关联来允许 AWS IoT Device Management 客户通过托管集成访问与您的账户关联的资源。例如，以最终用户已登录 Lights Incorporated 移动应用程序的智能灯泡为例。资源所有者是指购买并登录设备的最终用户账户。在我们的示例中，资源所有者被建模为 Lights In OAuth2 incorporated .0 账户。作为资源所有者，此账户提供发出命令和管理设备的权限。

### 资源服务器

这是托管需要授权才能访问的受保护资源（设备数据）的服务器。AWS 客户需要代表最终用户访问受保护的资源，他们可以通过账户关联后的 AWS IoT Device Management 连接器的托管集成来访问受保护的资源。以之前的智能灯泡为例，资源服务器是 Lights Incorporated 拥有的一项基于云的服务，用于在灯泡上线后对其进行管理。通过资源服务器，资源所有者可以向智能灯泡发出命令，例如将其打开和关闭。受保护的资源仅向最终用户的账户以及 accounts/entities 他们可能已提供权限的其他账户提供权限。

### 客户端

在这种情况下，客户端就是您的 C2C 连接器。客户机被定义为代表最终用户授予对资源服务器内资源的访问权限的应用程序。账户关联过程代表连接器（客户端），请求访问第三方云中最终用户的资源。

尽管连接器是 OAuth 客户端，但 AWS IoT Device Management 的托管集成代表连接器执行操作。例如，AWS IoT Device Management 的托管集成向授权服务器发出获取访问令牌请求。连接器仍被视为客户端，因为它是访问资源服务器中受保护资源（设备数据）的唯一组件。

以最终用户安装的智能灯泡为例。在客户平台和 Lights Incorporated 授权服务器之间完成账户关联后，连接器本身将与资源服务器通信，以检索有关最终用户智能灯泡的信息。然后，连接器可以接

收来自最终用户的命令。这包括通过 Lights Incorporated 资源服务器代表他们打开或关闭灯。因此，我们将连接器指定为客户端。

## 账户关联工作流程

对于客户通过 C2C 连接器与 AWS IoT Device Management 平台上的终端用户设备进行交互的托管集成，它会通过以下工作流程获取访问令牌：

1. 当用户通过客户应用程序启动第三方设备的启动时，AWS IoT Device Management 的托管集成会返回授权 URI 以及 AssociationId
2. 应用程序前端存储 AssociationId 并将最终用户重定向到第三方平台的登录页面。
  - 最终用户登录。最终用户授予客户端访问其设备数据的权限。
3. 第三方平台创建授权码。最终用户将被重定向到 AWS IoT Device Management 平台回调 URI 的托管集成，包括重定向请求所附的代码。
4. 托管集成会将此代码与第三方平台令牌 URI 交换。
5. 令牌 URI 验证授权码并返回与最终用户关联的 OAuth2 .0 访问令牌和刷新令牌。
6. 托管集成调用 C2C 连接器并进行 `AWS.ActivateUser` 操作，以完成账户关联流程并获取 UserId
7. 托管集成 OAuthRedirectUrl（从连接器策略配置）将成功的身份验证页面返回到客户应用程序。

### Note

如果出现故障，AWS IoT Device Management 的托管集成会将错误和 `error_description` 查询参数附加到向客户应用程序提供错误详细信息的网址。

8. 客户应用程序将最终用户重定向到 OAuth RedirectUrl 此时，应用程序前端从第一步就知道 AssociationId 了关联。

通过 C2C 连接器向 AWS IoT Device Management 托管集成向第三方云平台发出的所有后续请求，例如发现设备和发送命令的命令，都将包含 OAuth2 .0 访问令牌。

下图显示了账户关联的关键组成部分之间的关系：

## 实现 C2C 连接器接口操作

的托管集成 AWS IoT Device Management 定义了您 AWS Lambda 必须处理的四个操作才有资格成为连接器。您的 C2C 连接器必须实现以下每项操作：

1. [AWS.ActivateUser](#)- AWS IoT Device Management 服务的托管集成会调用此 API 来检索与所提供的 OAuth2 .0 令牌关联的全局唯一用户标识符。可以选择使用此操作来执行账户关联过程的其他要求。
2. [AWS.DiscoverDevices](#)-AWS IoT Device Management 服务的托管集成调用此 API 到您的连接器以发现用户的设备
3. [AWS.SendCommand](#)-AWS IoT Device Management 服务的托管集成会将此 API 调用到您的连接器，以便为用户设备发送命令
4. [AWS.DeactivateUser](#)-AWS IoT Device Management 服务的托管集成会将此 API 调用到您的连接器，以停用用户的访问令牌，以便在您的授权服务器中取消链接。

的托管集成 AWS IoT Device Management 始终通过操作调用带有 JSON 字符串负载的 Lambda 函数。AWS Lambda invokeFunction 请求操作必须在每个请求负载中包含一个 operationName 字段。有关更多信息，请参阅 AWS Lambda API 参考中的[调用](#)。

每次调用超时设置为两秒，如果调用失败，则将重试五次。

您为连接器实现的 Lambda 将 operationName 从请求有效负载中解析一个，并实现相应的功能以映射到第三方云：

```
public ConnectorResponse handleRequest(final ConnectorRequest request)
 throws OperationFailedException {
 Operation operation;
 try {
 operation = Operation.valueOf(request.payload().operationName());
 } catch (IllegalArgumentException ex) {
 throw new ValidationException(
 "Unknown operation '%s'".formatted(request.payload().operationName()),
 ex
);
 }

 return switch (operation) {
 case ActivateUser -> activateUserManager.activateUser(request);
 case DiscoverDevices -> deviceDiscoveryManager.listDevices(request);
 }
}
```

```

 case SendCommand -> sendCommandManager.sendCommand(request);
 case DeactivateUser -> deactivateUser.deactivateUser(request);
 };
}

```

### Note

连接器的开发者必须实现前面示例中列出的 `activateUserManager.activateUser(request)`、`deviceDiscoveryManager.listDevices` 和 `deactivateUser.deactivateUser` 操作。

以下示例详细介绍了来自托管集成的通用连接器请求，其中包含每个必需接口的公共字段。从示例中，您可以看到既有请求标头，又有请求负载。请求标头在每个操作接口中都很常见。

```

{
 "header": {
 "auth": {
 "token": "ashriu32yr97feqy7afsaf",
 "type": "OAuth2.0"
 }
 },
 "payload": {
 "operationName": "AWS.SendCommand",
 "operationVersion": "1.0",
 "connectorId": "exampleId",
 ...
 }
}

```

## 默认请求标头

默认标题字段如下所示。

```

{
 "header": {
 "auth": {
 "token": string, // end user's Access Token
 "type": ENUM ["OAuth2.0"],
 }
 }
}

```

}

连接器托管的任何 API 都必须处理以下标头参数：

### 默认标题和字段

| 字段                             | 必填/可选 | 描述                                                           |
|--------------------------------|-------|--------------------------------------------------------------|
| <code>header:auth</code>       | 是     | C2C 连接器生成器在连接器注册期间提供的授权信息。                                   |
| <code>header:auth:token</code> | 是     | 由第三方云提供商生成并链接到的用户的授权令牌 <code>connectorAssociationID</code> 。 |
| <code>header:auth:type</code>  | 是     | 所需的授权类型。                                                     |

#### Note

对您的连接器的所有请求都将附加最终用户的访问令牌。您可以假设最终用户与托管集成客户之间已经存在账户关联。

### 请求负载

除了常用标头外，每个请求都将有一个有效负载。虽然此有效载荷对每种操作类型都有唯一的字段，但每个有效载荷都有一组默认字段，这些字段将始终存在。

请求有效载荷字段：

- `operationName`：给定请求的操作，等于以下值之一：  
— `AWS.ActivateUser`、`AWS.SendCommand`、`AWS.DiscoverDevices`、`AWS.DeactivateUser`。
- `operationVersion`：每个操作都有版本控制，以允许其随着时间的推移而演变，并为第三方连接器提供稳定的接口定义。托管集成在所有请求的有效载荷中传递一个版本字段。
- `connectorId`：已向其发送请求的连接器的 ID。

## 默认响应标头

每项ACK操作都将通过 AWS IoT Device Management 的托管集成进行响应，以确认您的 C2C 连接器已收到请求并开始处理请求。以下是上述回复的通用示例：

```
{
 "header":{
 "responseCode": 200
 },
 "payload":{
 "responseMessage": "Example response!"
 }
}
```

每个操作响应都必须具有以下通用标头：

```
{
 "header": {
 "responseCode": Integer
 }
}
```

下表列出了默认的响应标头：

### 默认响应标头和字段

| 字段                  | 必填/可选 | 评论             |
|---------------------|-------|----------------|
| header:responseCode | 是     | 表示请求执行状态的值的枚举。 |

在本文档中描述的各种连接器接口和 API 架构中，都有一个responseMessage或Message字段。这是一个可选字段，用于 C2C 连接器 Lambda 来响应有关请求及其执行的任何上下文。最好是，任何导致状态码之外的错误都200应包含描述错误的消息值。

### 使用 API 响应 C2C 连接器操作请求 SendConnectorEvent

的托管集成要求 AWS IoT Device Management 您的连接器在每个 and 操作中都以异步方式运行AWS.SendCommand。AWS.DiscoverDevices这意味着对这些操作的初始响应只是“确认”您的 C2C 连接器已收到请求。

使用 `SendConnectorEvent` API，您的连接器应将以下列表中的事件类型发送到 `for and operations`，以及主动设备事件（例如手动开启和关闭灯光）。`AWS.DiscoverDevices` `AWS.SendCommand`要阅读有关这些事件类型及其用例的详细说明，请参阅[实施 AWS。DiscoverDevices 操作](#)、[实施 AWS。SendCommand 操作](#)、和[使用 SendConnectorEvent API 发送设备事件](#)。

例如，如果您的 C2C 连接器收到 `DiscoverDevices` 请求，则 AWS IoT Device Management 的托管集成希望它与上面定义的响应格式同步响应。然后，您必须使用中[实施 AWS。DiscoverDevices 操作](#)定义的请求结构为 `DEVICE_DISCOVERY` 事件调用 `SendConnectorEvent` API。API 调用可以在任何您有权访问 C2C 连接器 AWS 账户 Lambda 凭证的地方进行。`SendConnectorEvent`在 AWS IoT Device Management 的托管集成收到此事件之前，设备发现流程才会成功。

#### Note

或者，如有必要，`SendConnectorEvent`API 调用可以在 C2C 连接器 Lambda 调用响应之前进行。但是，这种流程与软件开发的异步模型相矛盾。

- `SendConnectorEvent`-您的连接器调用 AWS IoT Device Management API 的托管集成，将设备事件发送到 AWS IoT Device Management 的托管集成。托管集成仅接受三种类型的事件：
  - “`DEVICE_DISCOVERY`” — 此事件操作应用于发送第三方云中发现的设备列表以获取特定的访问令牌。
  - “`DEVICE_COMMAND_RESPONSE`” — 此事件操作应用于发送作为命令执行结果的特定设备事件。
  - “`DEVICE_EVENT`”-此事件操作应用于源自设备且不是基于用户的命令的直接结果的任何事件。这可以作为一种常规事件类型，用于主动报告设备状态变化或通知。

## 实施 AWS。ActivateUser 操作

AWS IoT Device Management 的托管集成需要执行该 `AWS.ActivateUser` 操作，才能从最终用户的 OAuth2 .0 令牌中检索用户标识符。的托管集成 AWS IoT Device Management 将在请求标头中传递 OAuth 令牌，并期望您的连接器在响应负载中包含全局唯一的用户标识符。此操作发生在账户关联流程成功之后。

以下列表概述了连接器的要求，以促进成功的 `AWS.Activate` 用户流。

- 您的 C2C 连接器 Lambda 可以处理来自 AWS `AWS.ActivateUser` IoT Device Management 托管集成的操作请求消息。

- 您的 C2C 连接器 Lambda 可以根据 OAuth2提供的.0 令牌确定唯一的用户标识符。通常，如果是 JWT 令牌，则可以从令牌本身中提取，也可以通过令牌从授权服务器请求该令牌。

## AWS.ActivateUser 工作流

1. 的托管集成使用以下有效负载 AWS IoT Device Management 载调用您的 C2C 连接器 Lambda：

```
{
 "header": {
 "auth": {
 "token": "ashriu32yr97feqy7afsaf",
 "type": "OAuth2.0"
 }
 },
 "payload": {
 "operationName": "AWS.ActivateUser",
 "operationVersion": "1.0.0",
 "connectorId": "Your-Connector-ID",
 }
}
```

2. C2C 连接器通过令牌或通过查询您的第三方资源服务器来确定要包含在AWS.ActivateUser响应中的用户 ID。
3. C2C 连接器会响应 Lambda AWS.ActivateUser 操作调用，包括默认负载以及字段内相应的用户标识符。userId

```
{
 "header": {
 "responseCode": 200
 },
 "payload": {
 "responseMessage": "Successfully activated user with connector-id `Your-Connector-Id.",
 "userId": "123456"
 }
}
```

## 实施 AWS。DiscoverDevices 操作

设备发现会将最终用户拥有的物理设备列表与 AWS IoT Device Management 托管集成中维护的最终用户设备的数字表示形式保持一致。只有在用户与 AWS IoT Device Management 的托管集成之间完成账户关联后，AWS 客户才会在最终用户拥有的设备上执行此操作。设备发现是一个异步过程，其中 AWS IoT Device Management 的托管集成会调用连接器来启动设备发现请求。C2C 连接器异步返回已发现的最终用户设备列表，其中包含托管集成生成的参考标识符（称为 `deviceDiscoveryId`）。

下图说明了 AWS IoT Device Management 的最终用户和托管集成之间的设备发现工作流程：

### AWS。DiscoverDevices 工作流程

1. 客户代表最终用户启动设备发现流程。
2. 的托管集成 AWS IoT Device Management 会生成一个参考标识符，该标识符调 `deviceDiscoveryId` 用 AWS 客户生成的设备发现请求。
3. 的托管集成使用 `AWS.DiscoverDevices` 操作界面向 C2C 连接器 AWS IoT Device Management 发送设备发现请求，包括有效 `OAuthAccessToken` 的最终用户以及 `deviceDiscoveryId`
4. 您的连接器存储 `deviceDiscoveryId` 以包含在 `DEVICE_DISCOVERY` 活动中。此事件还将包含已发现的最终用户设备的列表，并且必须将其发送到 AWS IoT Device Management 的托管集成，并将 `SendConnectorEvent` API 作为 `DEVICE_DISCOVERY` 事件发送。
5. 您的 C2C 连接器应调用资源服务器来获取最终用户拥有的所有设备。
6. 您的 C2C 连接器 Lambda 会响应 Lambda 调用 `invokeFunction()`，并向 AWS IoT Device Management 的托管集成发出 ACK 响应，作为操作的初始响应。`AWS.DiscoverDevices` 托管集成通过 ACK 通知客户已启动的设备发现流程。
7. 您的资源服务器会向您发送最终用户拥有和操作的设备列表。
8. 您的连接器将每台最终用户设备转换为 AWS IoT Device Management 所需设备格式的托管集成 `ConnectorDeviceIdConnectorDeviceName`，包括每台设备的能力报告。
9. C2C 连接器还提供 `UserId` 已发现设备所有者的信息。它可以作为设备列表的一部分从您的资源服务器中检索，也可以根据您的资源服务器实现情况在单独的调用中检索。
10. 接下来，您的 C2C 连接器将使用 AWS 账户 凭证并将操作参数设置为 `“DEVICE_DISCOVERY”SendConnectorEvent`，通过 `sigv4` 调用 AWS IoT 设备管理 API 的托管集成。发送到 AWS IoT Device Management 托管集成的设备列表中的每台设备都将由设备特定的参数表示 `connectorDeviceId`，例如 `connectorDeviceName`、和 `a. capabilityReport`

- 根据您的资源服务器响应，您需要相应地通知 AWS IoT Device Management 的托管集成。

例如，如果您的资源服务器对最终用户发现的设备列表进行了分页响应，那么对于每次轮询，您都可以发送一个statusCode参数为的3xx单个DEVICE\_DISCOVERY操作事件。如果您的设备发现仍在进行中，请重复步骤 5、6 和 7。

11. 的托管集成会向客户 AWS IoT Device Management 发送有关已发现最终用户设备的通知。

12. 如果您的 C2C 连接器发送的DEVICE\_DISCOVERY操作事件statusCode参数更新为 200，则托管集成将通知客户设备发现工作流程已完成。

#### Important

如果需要，步骤 7 到 11 可以在步骤 6 之前进行。例如，如果您的第三方平台有一个用于列出最终用户设备的 API，则可以在 C2C 连接器 Lambda 使用典型的 ACK 响应SendConnectorEvent之前发送 DEVICE\_DISCOVERY 事件。

## 设备发现时的 C2C 连接器要求

以下列表概述了 C2C 连接器的要求，以便于成功发现设备。

- C2C 连接器 Lambda a 可以处理来自 AWS IoT Device Management 托管集成的设备发现请求消息并处理操作。AWS.DiscoverDevices
- 您的 C2C 连接器可以使用用于注册连接器的凭证 APIs 通过 Sigv4 调用 AWS IoT Device Management 的 AWS 账户 托管集成。

## 设备发现过程

以下步骤概述了使用您的 C2C 连接器和托管集成 AWS IoT Device Management 的设备发现过程。

## 设备发现过程

1. 托管集成会触发设备发现：

- 使用以下 JSON 负载DiscoverDevices向发送 POST 请求：

```
/DiscoverDevices
{
 "header": {
```

```
 "auth": {
 "token": "ashriu32yr97feqy7afsaf",
 "type": "OAuth2.0"
 },
 "payload": {
 "operationName": "AWS.DiscoverDevices",
 "operationVersion": "1.0",
 "connectorId": "Your-Connector-Id",
 "deviceDiscoveryId": "12345678"
 }
 }
}
```

## 2. 连接器确认发现：

- 连接器发送带有以下 JSON 响应的确认：

```
{
 "header": {
 "responseCode": 200
 },
 "payload": {
 "responseMessage": "Discovering devices for discovery-job-id '12345678' with connector-id `Your-Connector-Id`"
 }
}
```

## 3. 连接器发送设备发现事件：

- 使用以下 JSON 负载 `/connector-event/{your_connector_id}` 向发送 POST 请求：

```
AWS API - /SendConnectorEvent
URI - POST /connector-event/{your_connector_id}
{
 "UserId": "6109342",
 "Operation": "DEVICE_DISCOVERY",
 "OperationVersion": "1.0",
 "StatusCode": 200,
 "DeviceDiscoveryId": "12345678",
 "ConnectorId": "Your_connector_Id",
 "Message": "Device discovery for discovery-job-id '12345678' successful",
 "Devices": [
 {
```

```

 "ConnectorDeviceId": "Your_Device_Id_1",
 "ConnectorDeviceName": "Your-Device-Name",
 "CapabilityReport": {
 "nodeId": "1",
 "version": "1.0.0",
 "endpoints": [{
 "id": "1",
 "deviceTypes": ["Camera"],
 "clusters": [{
 "id": "0x0006",
 "revision": 1,
 "attributes": [{
 "id": "0x0000",
 }],
 "commands": ["0x00", "0x01"],
 "events": ["0x00"]
 }
]
 }
]
}

```

为 DISCOVER\_D CapabilityReport EVICES 事件构造一个

如上面定义的事件结构所示，在 DISCOVER\_DEVICES 事件中报告的每台设备作为对 AWS.DiscoverDevices 操作的响应，都需要 CapabilityReport 来描述相应设备的功能。`CapabilityReport` 以符合 Matter 的格式表示 AWS IoT Device Management 设备功能的托管集成。`CapabilityReport` 中必须提供以下字段：

- `nodeId`，字符串：包含以下内容的设备节点的标识符 `endpoints`
- `version`，String：此设备节点的版本，由连接器开发者设置
- `endpoints`，<Cluster>列表：此设备端点支持的案件数据模型 AWS 实现列表。
  - `id`，字符串：连接器开发者设置的端点标识符
  - `deviceTypes`，<String>列表：此端点捕获的设备类型列表，即“摄像头”。
  - `clusters`，Lis <Cluster>t：此端点支持的案件数据模型的 AWS 实现列表。
    - `id`，字符串：Matter 标准定义的集群标识符。
    - `revision`，整数：Matter 标准定义的集群修订版号。

- `attributes` , `<String, Object>` 地图：属性标识符及其对应的当前设备状态值的映射，标识符和有效值由问题标准定义。
- `id` , 字符串：案件数据模型的 AWS 实现所定义的属性 ID。
- `value` , 对象：由属性 ID 定义的属性的当前值。“值”的类型可以根据属性而变化。该`value`字段对于每个属性都是可选的，只有当您的连接器 `lambda` 可以在发现期间确定当前状态时，才应包含该字段。
- `commands` , `<String>`列表：按照 Matter 标准的定义，此集群 IDs 支持的命令列表。
- `events` , `<String>`列表：根据 Matter 标准的定义，此集群 IDs 支持的事件列表。

有关[物质数据模型支持的功能及其相应AWS 实现的](#)当前列表，请参阅最新版本的数据模型文档。

## 实施 AWS。SendCommand 操作

该`AWS.SendCommand`操作允许 AWS IoT Device Management 的托管集成通过 AWS 客户将最终用户启动的命令发送到您的资源服务器。您的资源服务器可能支持多种类型的设备，其中每种类型都有自己的响应模型。命令执行是一个异步过程，其中 AWS IoT Device Management 的托管集成发送带有“TraceID”的命令执行请求，您的连接器将包含在通过“API”`SendConnectorEvent` 发送回托管集成的命令响应中。AWS IoT Device Management 的托管集成期望资源服务器返回一个确认已收到命令的响应，但不一定表示命令已执行。

下图说明了命令执行流程，并举例说明了最终用户尝试打开房屋灯光的示例：

### 设备命令执行工作流程

1. 最终用户使用 AWS 客户的应用程序发送开灯的命令。
2. 客户将命令信息与最终用户的设备信息传递给 AWS IoT Device Management 的托管集成。
3. 托管集成会生成“TraceID”，您的连接器将在将命令响应发送回服务时使用它。
4. AWS IoT Device Management 的托管集成使用`AWS.SendCommand`操作界面向您的连接器发送命令请求。
  - 此接口定义的有效载荷包括设备标识符、以 Matter 形式制定的设备命令`endpoints/clusters/commands`、最终用户的访问令牌以及其他必需的参数。
5. 您的连接器存储`traceId`要包含在命令响应中的内容。
  - 您的连接器会将托管集成命令请求转换为资源服务器的相应格式。

6. 您的连接器 `UserId` 从提供的最终用户的访问令牌中获取，并将其与命令相关联。
  - a. 如果是 JWT 和类似的令牌，则可以使用单独的调用从您的资源服务器中检索，也可以从访问令牌中提取。 `UserId`
  - b. 实现取决于您的资源服务器和访问令牌的详细信息。
7. 您的连接器调用资源服务器以“打开”最终用户的灯。
8. 资源服务器与设备交互。
  - a. 连接器中继到资源服务器已下发命令的 AWS IoT IoT Device Management 托管集成，并以 ACK 作为初始同步命令响应进行响应。
  - b. 然后，托管集成将其中继回客户应用程序。
9. 设备开灯后，您的资源服务器会捕获该设备事件。
10. 您的资源服务器将设备事件发送到连接器。
11. 您的连接器将资源服务器生成的设备事件转换为托管集成 `DEVICE_COMMAND_RESPONSE` 事件操作类型。
12. 您的连接器调用 `SendConnectorEvent` API，操作为“设备\_命令\_响应”。
  - 它会在初始请求中附上由 AWS IoT Device Management 托管集成 `traceId` 提供的内容。
13. 托管集成会通知客户有关最终用户的设备状态变化。
14. 客户通知最终用户设备灯已亮起。

#### Note

您的资源服务器配置决定了处理失败的设备命令请求和响应消息的逻辑。这包括对命令使用相同的 `referenceId` 进行消息重试尝试。

## 执行设备命令的 C2C 连接器要求

以下列表概述了 C2C 连接器的要求，以促进设备命令的成功执行。

- C2C 连接器 Lambda 可以 `AWS.SendCommand` 处理来自 AWS IoT Device Management 托管集成的操作请求消息。
- 您的 C2C 连接器必须跟踪发送到您的资源服务器的命令，并将其映射到相应的“TraceID”。
- 您可以使用用于注册 C2C 连接器的 AWS 凭证通过 Sigv4 调用 AWS IoT Device Management 服务 API 的 AWS 账户托管集成。

## 1. 托管集成向连接器发送命令 ( 请参阅上图中的步骤 4 ) 。

```
/Send-Command
{
 "header": {
 "auth": {
 "token": "ashriu32yr97feqy7afsaf",
 "type": "OAuth2.0"
 }
 },
 "payload": {
 "operationName": "AWS.SendCommand",
 "operationVersion": "1.0",
 "connectorId": "Your-Connector-Id",
 "connectorDeviceId": "Your_Device_Id",
 "traceId": "traceId-3241u78123419",
 "endpoints": [{
 "id": "1",
 "clusters": [{
 "id": "0x0202",
 "commands": [{
 "0xff01": {
 "0x0000": "3"
 }
]
 }
]
 }
}]
}
```

## 2. C2C 连接器 ACK 命令 ( 请参阅前一图表中的步骤 7 , 其中连接器向 AWS IoT 设备管理服务的托管集成发送 ACK ) 。

```
{
 "header":{
 "responseCode":200
 },
 "payload":{
 "responseMessage": "Successfully received send-command request for
connector 'Your-Connector-Id' and connector-device-id 'Your_Device_Id'"
 }
}
```

}

### 3. 连接器发送设备命令响应事件 ( 请参阅上图中的步骤 11 ) 。

```

AWS-API: /SendConnectorEvent
URI: POST /connector-event/{Your-Connector-Id}

{
 "UserId": "End-User-Id",
 "Operation": "DEVICE_COMMAND_RESPONSE",
 "OperationVersion": "1.0",
 "StatusCode": 200,
 "Message": "Example message",
 "ConnectorDeviceId": "Your_Device_Id",
 "TraceId": "traceId-3241u78123419",
 "MatterEndpoint": {
 "id": "1",
 "clusters": [{
 "id": "0x0202",
 "attributes": [
 {
 "0x0000": "3"
 }
],
 "commands": [
 "0xff01":
 {
 "0x0000": "3"
 }
]
 }
]
}

```

#### Note

在通过 API 收到相应的 DEVICE\_COMMAND\_RESPONSE 事件之前，由于命令执行而导致的设备状态变化不会反映在 AWS IoT Device Management 的托管集成中。SendConnectorEvent 这意味着，在托管集成收到前面步骤 3 的事件之前，无论您的连接器调用响应是否表示成功，都不会更新设备状态。

## 解释 AWS 中包含的“终端节点”。SendCommand 请求

托管集成将使用设备发现期间报告的设备功能来确定设备可以接受哪些命令。每个设备功能都是通过 AWS 实现物质数据模型进行建模的；因此，所有传入的命令都将从给定集群中的“commands”字段派生。您的连接器负责解析“端点”字段，确定相应的 Matter 命令，然后对其进行翻译，以便正确的命令到达设备。通常，这意味着将 Matter 数据模型转换为相关的 API 请求。

执行命令后，您的连接器将确定由物质数据模型的 AWS 实现定义的哪些“属性”因此发生了变化。然后，这些更改将通过 API 发送的 API\_DEVICE\_COMMAND\_RESPONSE 事件报告给 AWS IoT Device Management 的托管集成。SendConnectorEvent

考虑以下示例负载中包含的“端点”字段：AWS.SendCommand

```
"endpoints": [{
 "id": "1",
 "clusters": [{
 "id": "0x0202",
 "commands": [{
 "0xff01":
 {
 "0x0000": "3"
 }
]
 }]
}]
}]
```

通过此对象，连接器可以确定以下内容：

### 1. 设置终端节点和集群信息：

- a. 将端点设置id为“1”。

#### Note

如果设备定义了多个端点，例如单个集群（例如On/Off）can control multiple capabilities (i.e. turn a light on/off as well as turning a strobe on/off)，则使用此 ID 将命令路由到正确的功能。

- b. 将集群设置id为“0x0202”（风扇控制集群）。

### 2. 设置命令信息：

- a. 将命令标识符设置为“0xff01”（更新状态命令由定义）。AWS
  - b. 使用请求中提供的值更新包含的属性标识符。
3. 更新属性：
- a. 将属性标识符设置为“0x0000”（风扇控制集FanMode 群的属性）。
  - b. 将属性值设置为“3”（高风扇速度）。

托管集成定义了两种“自定义”命令类型，这些类型不是由物质数据模型的 AWS 实现严格定义的：ReadState 和 UpdateState 命令。要获取和设置 Matter 定义的集群属性，托管集成将向您的连接器发送一个AWS.SendCommand请求，其中包含与 UpdateState (id: 0xff01) 或 ReadState (id: 0xff02) IDs 相关的命令，以及必须更新或读取的相应属性参数。对于设置为可变（可更新）或可检索（可读取）的属性，可以为任何设备类型调用这些命令，这些属性可以从相应的 Matter 数据模型 AWS 实现中调用。

## 使用 SendConnectorEvent API 发送设备事件

### 设备启动的事件概述

虽然 SendConnectorEvent API 用于异步响应AWS.SendCommand和AWS.DiscoverDevices操作，但它也用于将任何设备启动的事件通知托管集成。设备启动的事件可以定义为设备在没有用户启动命令的情况下生成的任何事件。这些设备事件可能包括但不限于设备状态变化、运动检测、电池电量等。您可以使用带操作 DEVICE\_EVENT 的 SendConnectorEvent API 将这些事件发送回托管集成。

以下部分以安装在家中的智能摄像头为例，进一步说明这些事件的工作流程：

### 设备事件工作流程

1. 您的摄像机会检测到动作，并为此生成一个发送到您的资源服务器的事件。
2. 您的资源服务器处理该事件并将其发送到您的 C2C 连接器。
3. 您的连接器会将此事件转换为 AWS IoT Device Manag DEVICE\_EVENT ement 接口的托管集成。
4. 您的 C2C 连接器使用操作设置为“DEVICE\_EVENT” SendConnectorEvent 的 API 将此设备事件发送到托管集成。
5. 托管集成可识别相关客户，并将此事件转发给客户。
6. 客户收到此事件并通过用户标识符将其显示给用户。

有关 `SendConnectorEvent` API 操作的更多信息，请参阅 `SendConnectorEvent` AWS IoT Device Management 托管集成 API 参考指南。

### 设备启动的事件要求

以下是设备启动的事件的一些要求。

- 您的 C2C 连接器资源应该能够从您的资源服务器接收异步设备事件
- 您的 C2C 连接器资源应该能够使用用于注册 C2C 连接器的 AWS 凭证通过 Sigv4 调用 AWS IoT 设备管理服务 API 的 AWS 账户 托管集成。

以下示例演示了连接器通过 API 发送源于设备的事件：`SendConnectorEvent`

```
AWS-API: /SendConnectorEvent
URI: POST /connector-event/{Your-Connector-Id}

{
 "UserId": "Your-End-User-ID",
 "Operation": "DEVICE_EVENT",
 "OperationVersion": "1.0",
 "StatusCode": 200,
 "Message": None,
 "ConnectorDeviceId": "Your_Device_Id",
 "MatterEndpoint": {
 "id": "1",
 "clusters": [{
 "id": "0x0202",
 "attributes": [
 {
 "0x0000": "3"
 }
]
 }
]
}]
}
```

从以下示例中，我们可以看到以下内容：

- 这来自 ID 等于 1 的设备端点。
- 与该事件相关的设备功能的集群 ID 为 0x0202，与风扇控制问题集群有关。
- 已更改的属性的 ID 为 0x000，与集群中的风扇模式枚举有关。它已更新为值 3，与 High 的值有关。

- 由于connectorId是云服务在创建时返回的参数，因此 Connectors 必须使用查询 GetCloudConnector 和筛选依据lambdaARN。使用 Lambda.get\_function\_url\_config API 查询 lambda 自己的ARN值。这CloudConnectorId允许在 lambda 中动态访问，而不是像之前那样进行静态配置。

## 实施 AWS。 DeactivateUser 操作

### 用户停用概述

当客户删除其客户账户，或者最终用户想要取消其系统中的账户与 AWS 客户系统的关联时，需要停用提供的用户访问令牌。AWS 在这两种用例中，托管集成都需要使用 C2C 连接器来简化此工作流程。

下图说明了如何取消最终用户帐户与系统的关联

### 用户停用工作流程

1. 用户启动 AWS 客户账户与与 C2C 连接器关联的第三方授权服务器之间的解除关联过程。
2. 客户通过 AWS IoT Device Management 的托管集成启动删除用户关联。
3. 托管集成通过使用操作界面向连接器发出请求来启动停用过程。AWS.DeactivateUser
  - /user 的访问令牌包含在请求的标头中。
4. 您的 C2C 连接器接受请求并调用您的授权服务器来撤消令牌及其提供的任何访问权限。
  - 例如，来自未关联用户账户的事件在执行后不应再发送到托管集成。AWS.DeactivateUser
5. 您的授权服务器撤消访问权限并将响应发送回您的 C2C 连接器。
6. 您的 C2C 连接器会向 AWS IoT Device Management 的托管集成发送用户访问令牌已被撤销的 ACK。
7. 托管集成会删除最终用户拥有的与您的资源服务器关联的所有资源。
8. 托管集成会向客户发送 ACK，说明与您的系统相关的所有关联都已删除。
9. 客户通知最终用户其账户已与您的平台取消关联。

### AWS。 DeactivateUser 要求

- C2C 连接器 Lambda 函数接收来自托管集成的请求消息以处理操作。AWS.DeactivateUser

- C2C 连接器必须撤销授权服务器中用户提供的 OAuth2.0 令牌和相应的刷新令牌。

以下是您的连接器将收到的示例 `AWS.DeactivateUser` 请求：

```
{
 "header": {
 "auth": {
 "token": "ashriu32yr97feqy7afsaf",
 "type": "OAuth2.0"
 }
 },
 "payload": {
 "operationName": "AWS.DeactivateUser"
 "operationVersion": "1.0"
 "connectorId": "Your-connector-Id"
 }
}
```

## 调用你的 C2C 连接器

AWS Lambda 允许基于资源的策略授权谁可以调用 Lambda。由于 AWS IoT Device Management 的托管集成是 AWS 服务，因此您必须允许托管集成通过资源策略调用 C2C 连接器 Lambda。

将至少具有以下最低权限的资源策略附加到您的 C2C 连接器 Lambda。这提供了与 Lambda 函数调用权限的托管集成。本策略包含一个 `Condition` 密钥，可帮助您将您的可用性限制 `connectorId` 为仅限目标用户。

### JSON

```
{
 "Version": "2012-10-17",
 "Id": "default",
 "Statement": [
 {
 "Sid": "Your-Desired-Policy-ID",
 "Effect": "Allow",
 "Principal": {
 "Service": "iotmanagedintegrations.amazonaws.com"
 }
 }
]
}
```

```
 },
 "Action": "lambda:InvokeFunction",
 "Resource": "arn:aws:lambda:ca-central-1:444455556666:function:connector-
lambda-name",
 "Condition": {
 "StringEquals": {
 "aws:SourceArn": "arn:aws:iotmanagedintegrations:ca-
central-1:444455556666:account-association/account-association-id"
 }
 }
 }
]
```

## 为您的 IAM 角色添加权限

所有托管集成 API 需要 AWS Sigv4 身份验证才能调用。Sigv4 正在签署协议，以使用您的 AWS 账户凭据对 AWS API 请求进行身份验证。您用于调用托管集成的 IAM 角色 API 必须具有以下权限才能成功调用：APIs

```
"Version": "2012-10-17",
"Statement": [
 {
 "Sid": "Statement1",
 "Effect": "Allow",
 "Action": [
 "iotmanagedintegrations:Your-Required-Actions"
],
 "Resource": [
 "Your-Resource"
]
 }
]
```

有关添加这些权限的更多信息，请联系支持。

### 其他资源

要注册您的 C2C 连接器，您需要满足以下条件：

- 表示您要注册的连接器的 Lambda ARN。

## 手动测试您的 C2C 连接器

要手动测试 C2C 连接器 end-to-end，必须同时模拟客户和最终用户。

您将需要以下资源：

- 表示您要 AWS Lambda 测试的连接器的 ARN。
- 来自您的云平台的测试 OAuth 2.0 用户帐户。
- 在 AWS IoT Device Management 托管集成中注册的连接器。有关更多信息，请参阅 [使用 C2C \(云到云\) 连接器](#)。

## 使用 C2C (云到云) 连接器

C2C 连接器管理请求和响应消息的翻译，并支持托管集成与第三方供应商云之间的通信。它促进了对不同设备类型、平台和协议的统一控制，从而可以加载和管理第三方设备。

以下过程列出了使用 C2C 连接器的步骤。

使用 C2C 连接器的步骤：

### 1. CreateCloudConnector

配置连接器以启用托管集成与第三方供应商云之间的双向通信。

设置连接器时，请提供以下详细信息：

- 名称：为连接器选择一个描述性名称。
- 描述：简要概述连接器的用途和功能。
- AWS Lambda ARN：指定为连接器供电的 AWS Lambda 函数的亚马逊资源名称 (ARN)。

构建和部署与第三方供应商通信的 AWS Lambda 函数 APIs 以创建连接器。接下来，在托管集成中调用 [CreateCloudConnector](#) API，并提供 AWS Lambda 函数 ARN 进行注册。确保该 AWS Lambda 函数部署在托管集成中创建连接器的 AWS 账户中。系统将为您分配一个唯一的连接器 ID 来识别集成。

CreateCloudConnector API 请求和响应示例：

Request:

```

{
 "Name": "CreateCloudConnector",
 "Description": "Testing for C2C",
 "EndpointType": "LAMBDA",
 "EndpointConfig": {
 "lambda": {
 "arn": "arn:aws:lambda:us-east-1:xxxxxx:function:TestingConnector"
 }
 },
 "ClientToken": "abc"
}

Response:

{
 "Id": "string"
}

```

创作流程：

### Note

根据需要使用 [GetCloudConnectorUpdateCloudConnector](#)、[DeleteCloudConnector](#)、和 [ListCloudConnectors](#) APIs，执行此过程。

## 2. CreateConnectorDestination

配置目标以提供连接器与第三方供应商云建立安全连接所需的设置和身份验证凭据。使用 Destinations 将您的第三方身份验证凭据注册到托管集成，例如 OAuth 2.0 授权详细信息，包括授权 URL、身份验证方案以及其中凭据的位置 AWS Secrets Manager。

先决条件

在创建之前 ConnectorDestination，您必须：

- 调用 [CreateCloudConnector](#) API 创建连接器。在 [CreateConnectorDestination](#) API 调用中使用该函数返回的 ID。
- 检索连接 tokenUrl 器的 3P 平台的。（你可以用 authCode 兑换 AccessToken）。
- 检索连接器的 3P 平台的 authURL。（最终用户可以使用其用户名和密码进行身份验证）。

- 在账户的密钥管理器中使用clientId和clientSecret (来自 3P 平台)。

CreateConnectorDestination API 请求和响应示例：

Request:

```
{
 "Name": "CreateConnectorDestination",
 "Description": "CreateConnectorDestination",
 "AuthType": "OAUTH",
 "AuthConfig": {
 "oAuth": {
 "authUrl": "https://xxxx.com/oauth2/authorize",
 "tokenUrl": "https://xxxx.com/oauth2/token",
 "scope": "testScope",
 "tokenEndpointAuthenticationScheme": "HTTP_BASIC",
 "oAuthCompleteRedirectUrl": "about:blank",
 "proactiveRefreshTokenRenewal": {
 "enabled": false,
 "DaysBeforeRenewal": 30
 }
 }
 },
 "CloudConnectorId": "<connectorId>", // The connectorID instance from response
 "SecretsManager": {
 "arn": "arn:aws:secretsmanager:*****:secret:*****",
 "versionId": "*****"
 },
 "ClientToken": "****"
}
```

Response:

```
{
 "Id": "string"
}
```

云目标创建流程：

**Note**

根据需要使用 [GetCloudConnectorUpdateCloudConnector](#)、[DeleteCloudConnector](#)、和 [ListCloudConnectors](#) APIs，执行此过程。

### 3. CreateAccountAssociation

关联表示最终用户的第三方云账户与连接器目标之间的关系。在创建关联并将最终用户与托管集成关联后，他们的设备可通过唯一的关联 ID 进行访问。这种集成支持三个关键功能：发现设备、发送命令和接收事件。

#### 先决条件

在创建之前，AccountAssociation您必须完成以下操作：

- 调用 [CreateConnectorDestination](#) API 创建目的地。该函数返回的 ID 将用于 [CreateAccountAssociation](#) API 调用。
- 调用 [CreateAccountAssociation](#) API。

CreateAccountAssociation API 请求和响应示例：

Request:

```
{
 "Name": "CreateAccountAssociation",
 "Description": "CreateAccountAssociation",
 "ConnectorDestinationId": "<destinationId>", //The destinationID from
 destination creation.
 "ClientToken": "****"
}
```

Response:

```
{
 "Id": "string"
}
```

**Note**

根据需要使用 [GetCloudConnectorUpdateCloudConnector](#)、[DeleteCloudConnector](#)、和 [ListCloudConnectors](#) APIs，执行此过程。

AccountAssociation的状态是从[GetAccountAssociation](#)和[ListAccountAssociations](#) APIs中查询的。这些 APIs 显示了协会的状况。[StartAccountAssociationRefresh](#)API 允许在刷新令牌到期时刷新AccountAssociation状态。

#### 4. 设备发现

每个托管事物都与设备特定的详细信息相关联，例如其序列号和数据模型。数据模型描述了设备的功能，表明它是灯泡、开关、恒温器还是其他类型的设备。要发现 3P 设备并为该 3P 设备创建 ManagedThing，必须按顺序执行以下步骤。

- a. 调用 [StartDeviceDiscovery](#)API 开始设备发现过程。

StartDeviceDiscovery API 请求和响应示例：

Request:

```
{
 "DiscoveryType": "CLOUD",
 "AccountAssociationId": "*****",
 "ClientToken": "abc"
}
```

Response:

```
{
 "Id": "string",
 "StartedAt": number
}
```

- b. 调用 [GetDeviceDiscovery](#)API 来检查发现过程的状态。
- c. 调用 [ListDiscoveredDevices](#)API 列出发现的设备。

ListDiscoveredDevices API 请求和响应示例：

Request:

```
//Empty body

Response:

{
 "Items": [
 {
 "Brand": "string",
 "ConnectorDeviceId": "string",
 "ConnectorDeviceName": "string",
 "DeviceTypes": ["string"],
 "DiscoveredAt": number,
 "ManagedThingId": "string",
 "Model": "string",
 "Modification": "string"
 }
],
 "NextToken": "string"
}
```

- d. 调用 [CreateManagedThing](#) API 从发现列表中选择要导入托管集成的设备。

CreateManagedThing API 请求和响应示例：

```
Request:

{
 "Role": "DEVICE",
 "AuthenticationMaterial": "CLOUD:XXXX:<connectorDeviceId1>",
 "AuthenticationMaterialType": "DISCOVERED_DEVICE",
 "Name": "sample-device-name"
 "ClientToken": "xxx"
}

Response:

{
 "Arn": "string", // This is the ARN of the managedThing
 "CreatedAt": number,
 "Id": "string"
}
```

- e. 调用 [GetManagedThing](#) API 来查看这个新创建的 managedThing。状态将是 UNASSOCIATED。
- f. 调用 [RegisterAccountAssociation](#) API managedThing 将其与特定关联 accountAssociation。成功的 [RegisterAccountAssociation](#) API 结束后，ASSOCIATED 状态会 managedThing 发生变化。

RegisterAccountAssociation API 请求和响应示例：

```
Request:
{
 "AccountAssociationId": "string",
 "DeviceDiscoveryId": "string",
 "ManagedThingId": "string"
}

Response:
{
 "AccountAssociationId": "string",
 "DeviceDiscoveryId": "string",
 "ManagedThingId": "string"
}
```

## 5. 向 3P 设备发送命令

要控制新上线的设备，请使用 [SendManagedThingCommand](#) API，使用先前创建的关联 ID 和基于设备支持的函数的控制操作。连接器使用账户关联过程中存储的凭据向第三方云进行身份验证并调用操作的相关 API 调用。

SendManagedThingCommand API 请求和响应示例：

```
Request:
{
 "AccountAssociationId": "string",
 "ConnectorAssociationId": "string",
 "Endpoints": [
 {
 "capabilities": [
 {
 "actions": [
```

```

 {
 "actionTraceId": "string",
 "name": "string",
 "parameters": JSON value,
 "ref": "string"
 }
],
 "id": "string",
 "name": "string",
 "version": "string"
}
],
"endpointId": "string"
}
]
}

Response:

{
 "TraceId": "string"
}

```

向 3P 设备流程发送命令：

## 6. 连接器向托管集成发送事件

[SendConnectorEvent](#) API 捕获从连接器到托管集成的四种类型的事件，由 Operation Type 参数的以下枚举值表示：

- `DEVICE_COMMAND_RESPONSE`：连接器为响应命令而发送的异步响应。
- `DEVICE_DISCOVERED`：为了响应设备发现过程，连接器使用 API 将发现的设备列表发送给托管集成。[SendConnectorEvent](#)
- `DEVICE_EVENT`：发送收到的设备事件。
- `DEVICE_COMMAND_REQUEST`：从设备发起的命令请求。例如，WebRTC 工作流程。

连接器还可以使用带有可选 `userId` 参数的 [SendConnectorEvent](#) API 转发设备事件。

- 对于带有以下内容的设备事件 `userId`：

## SendConnectorEvent API 请求和响应示例：

Request:

```
{
 "UserId": "*****",
 "Operation": "DEVICE_EVENT",
 "OperationVersion": "1.0",
 "StatusCode": 200,
 "ConnectorId": "*****",
 "ConnectorDeviceId": "****",
 "TraceId": "****",
 "MatterEndpoint": {
 "id": "***",
 "clusters": [{

 }]
 }
}
```

Response:

```
{
 "ConnectorId": "string"
}
```

- 对于没有userId：的设备事件

## SendConnectorEvent API 请求和响应示例：

Request:

```
{
 "Operation": "DEVICE_EVENT",
 "OperationVersion": "1.0",
 "StatusCode": 200,
 "ConnectorId": "*****",
 "ConnectorDeviceId": "*****",
 "TraceId": "*****",
 "MatterEndpoint": {
 "id": "***",
```

```
 "clusters": [{

 }]
 }
}

Response:

{
 "ConnectorId": "string"
}
```

要删除特定账户关联managedThing和账户关联之间的关联，请使用注销机制：

DeregisterAccountAssociation API 请求和响应示例：

```
Request:

{
 "AccountAssociationId": "*****",
 "ManagedThingId": "*****"
}

Response:

HTTP/1.1 200 // Empty body
```

发送事件流：

## 7. 将连接器状态更新为“已上市”，使其对其他托管集成客户可见

默认情况下，连接器是私有的，只有创建连接器的 AWS 账户才能看见。您可以选择将连接器设置为对其他托管集成客户可见。

要与其他用户共享您的连接器，请使用连接器详细信息页面 AWS 管理控制台 上的“设为可见”选项，将您的连接器 ID 提交给以 AWS 供审核。一旦获得批准，该连接器便可供所有托管集成用户使用。AWS 区域此外，您可以 IDs 通过修改连接器关联 AWS Lambda 功能的访问策略来限制对特定 AWS 账户的访问权限。为确保您的连接器可供其他客户使用，请管理您的 Lambda 函数从其他 AWS 账户到可见连接器的 IAM 访问权限。

在将连接器设置为其他托管集成客户可见之前，请查看管理连接器共享和访问权限的 AWS 服务条款和组织政策。

# 托管集成 Hub SDK

使用本节中的主题来学习如何使用托管集成 Hub SDK 加载和控制 IoT 中心设备。有关托管集成终端设备 SDK 的更多信息，请参阅。[托管集成终端设备 SDK](#)

## 中心 SDK 架构

### 设备上线

在开始使用托管集成之前，请先查看 Hub SDK 组件如何支持设备上线。本节介绍设备入门所需的基本架构组件，包括核心配置器和协议特定插件如何协同工作以处理设备身份验证、通信和用户设置。

### 用于设备加载的 Hub SDK 组件

#### SDK 组件

- [核心供应器](#)
- [特定于协议的供应器插件](#)
- [特定于协议的中间件](#)

#### 核心供应器

核心配置器是在 IoT 中心部署中协调设备启动的核心组件。它协调托管集成与您的协议特定配置器插件之间的所有通信，确保设备启动安全可靠。当您加载设备时，核心配置器会通过以下功能处理身份验证流程、管理 MQTT 消息并处理设备请求：

#### MQTT 连接

与 MQTT 代理建立连接，以便发布和订阅云主题。

#### 消息队列和处理程序

按顺序处理传入的添加和删除设备请求。

#### 协议插件接口

通过管理身份验证和无线电加入模式，与协议特定的配置器插件配合使用，用于设备入门。

## 中心 SDK 客户端 APIs

接收来自特定协议的 CDMB 插件的设备功能报告，并将其转发到托管集成。

## 特定于协议的供应器插件

特定于协议的配置器插件是用于管理不同通信协议的设备加载的库。每个插件都会将来自核心配置程序的命令转换为物联网设备的特定于协议的操作。这些插件执行以下操作：

- 特定于协议的中间件初始化
- 基于核心供应器请求的无线电加入模式配置
- 通过中间件 API 调用移除设备

## 特定于协议的中间件

特定于协议的中间件充当设备协议和托管集成之间的转换层。该组件处理双向通信——接收来自供应器插件的命令并将其发送到协议堆栈，同时还收集来自设备的响应并通过系统将其路由回去。

## 设备上线流程

查看使用 Hub SDK 加载设备时发生的操作顺序。本节显示了组件在入门过程中的交互方式，并概述了支持的入门方法。

### 入职流程

- [简单设置 \(SS\)](#)
- [零触摸设置 \(ZTS\)](#)
- [用户指导设置 \(UGS\)](#)

### 简单设置 (SS)

最终用户打开物联网设备的电源，并使用设备制造商的应用程序扫描其二维码。然后，设备将注册到托管集成云并连接到物联网中心。

### 零触摸设置 (ZTS)

零触摸设置 (ZTS) 通过在供应链上游预关联设备来简化设备上游。例如，最终用户无需扫描设备二维码，而是提前完成此步骤，以便将设备预先关联到客户账户。例如，可以在运营中心完成此步骤。

当最终用户收到设备并开机时，它会自动注册到托管集成云中并连接到物联网中心，无需任何其他设置操作。

## 用户指导设置 (UGS)

最终用户开启设备并按照交互式步骤将其加入托管集成。这可能包括按下 IoT 中心上的按钮、使用设备制造商的应用程序或同时按下集线器和设备上的按钮。如果简单设置失败，则可以使用此方法。

## 设备控制

托管集成可处理设备注册、命令执行和控制。您可以使用与供应商和协议无关的设备管理功能，在不了解设备特定协议的情况下打造最终用户体验。

通过设备控制，您可以查看和修改设备状态，例如灯泡亮度或门位置。该功能会针对状态变化发出事件，您可以将其用于分析、规则和监控。

### 主要特征

#### 修改或读取设备状态

根据设备类型查看和更改设备属性。您可以访问：

- 设备状态：当前设备属性值
- 连接状态：设备可接通性状态
- He@@ alth status：系统值，例如电池电量和信号强度 (RSSI)

#### 状态变更通知

当设备属性或连接状态发生变化时接收事件，例如灯泡亮度调整或门锁状态变化。

#### 离线模式

即使没有互联网连接，设备也能与同一物联网中心上的其他设备通信。恢复连接后，设备状态会与云同步。

#### 状态同步

跟踪来自多个来源、设备制造商应用程序和手动设备调整的状态变化。

查看通过托管集成控制设备所需的 Hub SDK 组件和流程。本主题介绍 Edge Agent、Common Data Model Bridge (CDBM) 和特定于协议的插件如何协同工作，以处理设备命令、管理设备状态和处理不同协议的响应。

## 设备控制流程

下图通过描述最终用户如何打开 ZigBee 智能插头来演示 end-to-end 设备控制流程。

## 用于设备控制的 Hub SDK 组件

Hub SDK 架构使用以下组件来处理和路由物联网实现中的设备控制命令。在将云命令转换为设备操作、管理设备状态和处理响应方面，每个组件都起着特定的作用。以下各节详细介绍了这些组件在您的部署中如何协同工作：

Hub SDK 由以下组件组成，便于在物联网中心上启动和控制设备。

主要组件：

### 边缘代理

充当物联网中心和托管集成之间的网关。

### 通用数据模型桥 (CDBM)

在 AWS 数据模型和本地协议数据模型（如 Z-Wave 和 Zigbee）之间进行转换。它包括一个核心 CDBM 和特定于协议的 CDBM 插件。

### 置备者

处理设备发现和上线。它包括用于特定于协议的入门任务的核心配置器和特定于协议的配置器插件。

### 次要组件

#### Hub 入职培训

为集线器配置客户端证书和密钥，以实现安全的云通信。

#### MQTT 代理

提供与托管集成云的 MQTT 连接。

## 日志记录程序

将日志写入本地或托管集成云。

## 安装并验证托管集成 Hub SDK

在以下部署方法之间进行选择，在您的设备上安装托管集成 Hub SDK，AWS IoT Greengrass 用于自动部署或手动安装脚本。本节介绍两种方法的设置和验证步骤。

### 部署方法

- [使用以下命令安装 Hub SDK AWS IoT Greengrass](#)
- [使用脚本部署 Hub SDK](#)
- [使用系统部署 Hub SDK](#)

## 使用以下命令安装 Hub SDK AWS IoT Greengrass

使用 AWS IoT Greengrass (Java 版本) 为您的设备部署托管集成 Hub SDK 组件。

### Note

您必须已经设置并了解 AWS IoT Greengrass。有关更多信息，请参阅 AWS IoT Greengrass 开发者指南文档 AWS IoT Greengrass 中的 [内容](#)。

AWS IoT Greengrass 用户必须具有修改以下目录的权限：

- /dev/aipc
- /data/aws/iotmi/config
- /data/ace/kvstorage

### 主题

- [在本地部署组件](#)
- [云部署](#)
- [验证集线器配置](#)
- [验证 CDMB 的运行情况](#)

- [验证 LPW 配置器的运行情况](#)

## 在本地部署组件

使用设备上[CreateDeployment](#) AWS IoT Greengrass 的 API 部署 Hub SDK 组件。版本号不是静态的，可能因您当时使用的版本而异。使用以下格式表示 **version** : com.amazon.io。TManagedIntegrationsDevice AceCommon= 0.2.0。

```
/greengrass/v2/bin/greengrass-cli deployment create \
--recipeDir recipes \
--artifactDir artifacts \
-m "com.amazon.IoTManagedIntegrationsDevice.AceCommon=version" \
-m "com.amazon.IoTManagedIntegrationsDevice.HubOnboarding=version" \
-m "com.amazon.IoTManagedIntegrationsDevice.AceZigbee=version" \
-m "com.amazon.IoTManagedIntegrationsDevice.LPW-Provisioner=version" \
-m "com.amazon.IoTManagedIntegrationsDevice.Agent=version" \
-m "com.amazon.IoTManagedIntegrationsDevice.MQTTProxy=version" \
-m "com.amazon.IoTManagedIntegrationsDevice.CDMB=version" \
-m "com.amazon.IoTManagedIntegrationsDevice.AceZwave=version"
```

## 云部署

按照[AWS IoT Greengrass 开发者指南](#)中的说明执行以下步骤：

1. 将项目上传到亚马逊 S3。
2. 更新配方以包含 Amazon S3 工件的位置。
3. 在设备上为新组件创建云部署。

## 验证集线器配置

通过检查您的配置文件来确认配置成功。打开 /data/aws/iotmi/config/iotmi\_config.json 文件并验证状态是否设置为 PROVISIONED。

## 验证 CDMB 的运行情况

检查日志文件中是否有 CDMB 启动消息和成功初始化。*logs file* 位置可能因安装位置 AWS IoT Greengrass 而异。

```
tail -f -n 100 /greengrass/v2/logs/com.amazon.IoTManagedIntegrationsDevice.CDMB.log
```

## 示例

```
[2024-09-06 02:31:54.413758906][IoTManagedIntegrationsDevice_CDMB][info] Successfully
subscribed to topic: south/bF|gi_044F8821D0193608C8D5BF80858E20A56E3A8490/control
[2024-09-06 02:31:54.513956059][IoTManagedIntegrationsDevice_CDMB][info] Successfully
subscribed to topic: south/bF|gi_044F8821D0193608C8D5BF80858E20A56E3A8490/setup
```

## 验证 LPW 配置器的运行情况

检查日志文件中是否有 LPW-Provisioner 启动消息和成功初始化。*logs file* 位置可能因安装位置 AWS IoT Greengrass 而异。

```
tail -f -n 100 /greengrass/v2/logs/com.amazon.IoTManagedIntegrationsDevice.LPW-
Provisioner.log
```

## 示例

```
[2024-09-06 02:33:22.068898877][LPWProvisionerCore][info] Successfully subscribed to
topic: south/bF|gi_044F8821D0193608C8D5BF80858E20A56E3A8490/setup
```

## 使用脚本部署 Hub SDK

使用安装脚本手动部署托管集成 Hub SDK 组件，然后验证部署。本节介绍脚本执行步骤和验证过程。

### 主题

- [准备好您的环境](#)
- [运行 Hub SDK 脚本](#)
- [验证集线器配置](#)
- [验证代理操作](#)
- [验证 LPW 配置器的运行情况](#)

## 准备好您的环境

在运行 SDK 安装脚本之前，请完成以下步骤：

1. 在文件夹 `middleware` 内创建一个名为 `artifacts` 的文件夹。
2. 将您的集线器中间件文件复制到该文件 `middleware` 夹。
3. 在启动 SDK 之前运行初始化命令。

**⚠ Important**

每次集线器重新启动后重复初始化命令。

```
#Get the current user
_user=$(whoami)

#Get the current group
_grp=$(id -gn)

#Display the user and group
echo "Current User: $_user"
echo "Current Group: $_grp"

sudo mkdir -p /dev/aipc/
sudo chown -R $_user:$_grp /dev/aipc
sudo mkdir -p /data/ace/kvstorage
sudo chown -R $_user:$_grp /data/ace/kvstorage
```

## 运行 Hub SDK 脚本

导航到构件目录并运行 `start_iotmi_sdk.sh` 脚本。此脚本按正确的顺序启动 Hub SDK 组件。查看以下示例日志以验证是否成功启动：

**📘 Note**

所有正在运行的组件的日志都可以在该 `artifacts/logs` 文件夹中找到。

```
hub@hub-293ea release_Oct_17$./start_iotmi_sdk.sh
-----Stopping SDK running processes---
DeviceAgent: no process found
-----Starting SDK-----
-----Creating logs directory-----
Logs directory created.
-----Verifying Middleware paths-----
All middleware libraries exist
-----Verifying Middleware pre reqs---
```

```

AIPC and KVstroage directories exist
-----Starting HubOnboarding-----
-----Starting MQTT Proxy-----
-----Starting Event Manager-----
-----Starting Zigbee Service-----
-----Starting Zwave Service-----
/data/release_Oct_17/middleware/AceZwave/bin /data/release_Oct_17
/data/release_Oct_17
-----Starting CDMB-----
-----Starting Agent-----
-----Starting Provisioner-----
-----Checking SDK status-----
hub 6199 1.7 0.7 1004952 15568 pts/2 Sl+ 21:41 0:00 ./iotmi_mqtt_proxy -
C /data/aws/iotmi/config/iotmi_config.json
Process 'iotmi_mqtt_proxy' is running.
hub 6225 0.0 0.1 301576 2056 pts/2 Sl+ 21:41 0:00 ./middleware/
AceCommon/bin/ace_eventmgr
Process 'ace_eventmgr' is running.
hub 6234 104 0.2 238560 5036 pts/2 Sl+ 21:41 0:38 ./middleware/
AceZigbee/bin/ace_zigbee_service
Process 'ace_zigbee_service' is running.
hub 6242 0.4 0.7 1569372 14236 pts/2 Sl+ 21:41 0:00 ./zwave_svc
Process 'zwave_svc' is running.
hub 6275 0.0 0.2 1212744 5380 pts/2 Sl+ 21:41 0:00 ./DeviceCdm
b
Process 'DeviceCdm
b' is running.
hub 6308 0.6 0.9 1076108 18204 pts/2 Sl+ 21:41 0:00 ./
IoTManagedIntegrationsDeviceAgent
Process 'DeviceAgent' is running.
hub 6343 0.7 0.7 1388132 13812 pts/2 Sl+ 21:42 0:00 ./
iotmi_lpw_provisioner
Process 'iotmi_lpw_provisioner' is running.
-----Successfully Started SDK-----

```

## 验证集线器配置

检查中的 `iot_provisioning_state` 字段 `/data/aws/iotmi/config/iotmi_config.json` 是否设置为 `PROVISIONED`。

## 验证代理操作

检查日志文件中是否有代理启动消息和成功初始化。

```
tail -f -n 100 logs/agent_logs.txt
```

## 示例

```
[2024-09-06 02:31:54.413758906][Device_Agent][info] Successfully subscribed to topic:
south/bF|gi_044F8821D0193608C8D5BF80858E20A56E3A8490/control
[2024-09-06 02:31:54.513956059][Device_Agent][info] Successfully subscribed to topic:
south/bF|gi_044F8821D0193608C8D5BF80858E20A56E3A8490/setup
```

### Note

检查您的artifacts目录中是否存在该iotmi.db数据库。

## 验证 LPW 配置器的运行情况

检查日志文件中是否有LPW-Provisioner启动消息和成功初始化。

```
tail -f -n 100 logs/provisioner_logs.txt
```

下面的代码显示了一个示例。

```
[2024-09-06 02:33:22.068898877][LPWProvisionerCore][info] Successfully subscribed to
topic: south/bF|gi_044F8821D0193608C8D5BF80858E20A56E3A8490/setup
```

## 使用系统部署 Hub SDK

### Important

请按readme.md照 release.tgz 文件hubSystemdSetup目录中的内容获取最新更新。

本节介绍在基于 Linux 的中心设备上部署和配置服务的脚本和过程。

## 概览

部署过程由两个主要脚本组成：

- copy\_to\_hub.sh: 在主机上运行以将必要文件复制到集线器
- setup\_hub.sh: 在集线器上运行以配置环境和部署服务

此外，还systemd/deploy\_iotshd\_services\_on\_hub.sh处理进程引导顺序和进程权限管理，并由setup\_hub.sh自动触发。

## 先决条件

成功部署需要满足列出的先决条件。

- 集线器上有 systemd 服务可用
- 通过 SSH 访问中心设备
- 集线器设备上的 Sudo 权限
- scp安装在主机上的实用程序
- sed安装在主机上的实用程序
- 主机上安装了 unzip 实用程序

## 文件结构

文件结构旨在便于组织和管理其各个组成部分，从而实现内容的高效访问和导航。

```
hubSystemdSetup/
README.md
copy_to_hub.sh
setup_hub.sh
iotshd_config.json # Sample configuration file
local_certs/ # Directory for DHA certificates
systemd/
 ### *.service.template # Systemd service templates
 ### deploy_iotshd_services_on_hub.sh
```

在 SDK 发行版 tgz 文件中，总体文件结构为：

```
IoT-managed-integrations-Hub-SDK-aarch64-v1.0.0.tgz
###package/
 ###greengrass/
 ###artifacts/
 ###recipes/
 ###hubSystemdSetup/
 ### REAME.md
 ### copy_to_hub.sh
 ### setup_hub.sh
```

```
iotshd_config.json # Sample configuration file
local_certs/ # Directory for DHA certificates
systemd/
*.service.template # Systemd service templates
deploy_iotshd_services_on_hub.sh
```

## 初始 设置

### 解压软件开发工具包包

```
tar -xzf managed-integrations-Hub-SDK-vVersion-linux-aarch64-timestamp.tgz
```

导航到解压缩的目录并准备软件包：

```
Create package.zip containing required artifacts
zip -r package.zip package/greengrass/artifacts
Move package.zip to the hubSystemdSetup directory
mv package.zip ../hubSystemdSetup/
```

### 添加设备配置文件

按照列出的两个步骤创建设备配置文件并将其复制到集线器。

1. [添加设备配置文件](#)以创建所需的设备配置文件。SDK 使用此文件来实现其功能。
2. [复制配置文件](#)以将创建的配置文件复制到集线器。

### 将文件复制到集线器

从您的主机运行部署脚本：

```
chmod +x copy_to_hub.sh
./copy_to_hub.sh hub_ip_address package_file
```

### Example 示例

```
./copy_to_hub.sh 192.168.50.223 ~/Downloads/EAR3-package.zip
```

这复制了：

- 软件包文件 ( 在集线器上重命名为 package.zip )
- 配置文件
- 证书
- 系统服务文件

## 设置集线器

复制文件后，通过 SSH 连接到集线器并运行安装脚本：

```
ssh hub@hub_ip
chmod +x setup_hub.sh
sudo ./setup_hub.sh
```

## 用户和群组配置

默认情况下，我们将用户中心和群组中心用于 SDK 组件。有多种方法可以对其进行配置：

- 使用自定义用户/群组：

```
sudo ./setup_hub.sh --user=USERNAME --group=GROUPNAME
```

- 在运行安装脚本之前手动创建它们：

```
sudo groupadd -f GROUPNAME
sudo useradd -r -g GROUPNAME USERNAME
```

- 在中添加命令 setup\_hub.sh。

## 管理服务

要重新启动所有服务，请从 hub 运行以下脚本：

```
sudo /usr/local/bin/deploy_iotshd_services_on_hub.sh
```

安装脚本将创建必要的目录、设置适当的权限并自动部署服务。如果您不使用 SSH/SCP，则必须 copy\_to\_hub.sh 针对您的特定部署方法进行修改。在部署之前，请确保所有证书文件和配置均已正确设置。

# 将您的集线器加入托管集成

通过配置所需的目录结构、证书和设备配置文件，将您的中心设备设置为与托管集成进行通信。本节介绍集线器载入子系统组件如何协同工作、证书和配置文件的存储位置、如何创建和修改设备配置文件以及完成集线器配置过程的步骤。

## Hub 入门子系统

集线器载入子系统使用以下核心组件来管理设备配置和配置：

### Hub 入门组件

通过协调中心状态、配置方法和身份验证材料来管理中心入职流程。

### 设备配置文件

在设备上存储重要的集线器配置数据，包括：

- 设备配置状态（已配置或未配置）
- 证书和密钥位置
- 身份验证信息其他 SDK 进程（例如 MQTT 代理）将引用此文件来确定集线器状态和连接设置。

### 证书处理程序接口

提供用于读取和写入设备证书和密钥的实用程序接口。你可以实现这个接口来使用：

- 文件系统存储
- 硬件安全模块 (HSM)
- 可信平台模块 (TPM)
- 定制安全存储解决方案

### MQTT 代理组件

使用以下方法管理 device-to-cloud 通信：

- 预配置的客户端证书和密钥
- 配置文件中的设备状态信息
- 与托管集成的 MQTT 连接

下图描述了集线器载入子系统架构及其组件。如果您不使用 AWS IoT Greengrass，则可以忽略图中的该组件。

## Hub 入职设置

在开始队列配置入门流程之前，请完成每台中心设备的这些设置步骤。本节介绍如何创建托管事物、设置目录结构和配置所需的证书。

### 设置步骤

- [步骤 1：注册自定义终端节点](#)
- [步骤 2：创建配置文件](#)
- [步骤 3：创建托管事物（队列配置）](#)
- [步骤 4：创建目录结构](#)
- [第 5 步：向中心设备添加身份验证材料](#)
- [步骤 6：创建设备配置文件](#)
- [第 7 步：将配置文件复制到集线器](#)

### 步骤 1：注册自定义终端节点

创建专用的通信端点，您的设备使用该端点与托管集成交换数据。此端点为所有 device-to-cloud 消息（包括设备命令、状态更新和通知）建立安全的连接点。

#### 要注册终端节点

- 使用 [RegisterCustomEndpoint](#) API 创建用于 device-to-managed 集成通信的端点。

#### RegisterCustomEndpoint 请求示例

```
aws iot-managed-integrations register-custom-endpoint
```

#### 响应：

```
{
 [ACCOUNT-PREFIX]-ats.iot.AWS-REGION.amazonaws.com
}
```

#### Note

存储端点地址。你将需要它来进行 future 设备通信。

要返回端点信息，请使用 `GetCustomEndpoint` API。

有关更多信息，请参阅托管集成 [GetCustomEndpoint](#) API 参考指南中的 API 和 [RegisterCustomEndpoint](#) API。

## 步骤 2：创建配置文件

配置文件包含您的设备连接到托管集成所需的安全凭证和配置设置。

### 创建队列配置文件

- 调用 [CreateProvisioningProfile](#) API 生成以下内容：
  - 用于定义设备连接设置的配置模板
  - 用于设备身份验证的索赔证书和私钥

#### Important

安全地存储索赔证书、私钥和模板 ID。您需要这些凭据才能将设备加入托管集成。如果您丢失了这些凭据，则必须创建新的配置文件。

## CreateProvisioningProfile 请求示例

```
aws iot-managed-integrations create-provisioning-profile \
 --provisioning-type FLEET_PROVISIONING \
 --name PROFILE_NAME
```

响应：

```
{
 "Arn": "arn:aws:iotmanagedintegrations:AWS-REGION:ACCOUNT-ID:provisioning-
profile/PROFILE-ID",
 "ClaimCertificate":
 "-----BEGIN CERTIFICATE-----
MIICiTCCAfICCQD6m7.....w3rrszlaEXAMPLE=
-----END CERTIFICATE-----",
 "ClaimCertificatePrivateKey":
```

```
"-----BEGIN RSA PRIVATE KEY-----
MIICiTCCAfICCQ...3rrszlaEXAMPLE=
-----END RSA PRIVATE KEY-----",
 "Id": "PROFILE-ID",
 "PROFILE-NAME",
 "ProvisioningType": "FLEET_PROVISIONING"
}
```

### 步骤 3：创建托管事物（队列配置）

使用 `CreateManagedThing` API 为您的中心设备创建托管事物。每个中心都需要自己的托管东西和独特的身份验证材料。有关更多信息，请参阅《托管集成 [CreateManagedThing](#) API 参考》中的 API。

创建托管事物时，请指定以下参数：

- `Role`：CONTROLLER 对于不支持命令和控制的集线器，将此值设置为，否则设置为 DEVICE。
- `AuthenticationMaterialType`：将此值设置为 WIFI\_SETUP\_QR\_BAR\_CODE。
- `AuthenticationMaterial`：包括以下字段。你可以使用其中 UPC 一个，也可以同时使用，EAN 但不能两者兼而有之。
  - SN: 此设备的唯一序列号
  - UPC: 此设备的通用产品代码
  - EAN: 此设备的国际商品编号

#### Important

每台设备的身份验证材料中必须有一个唯一的序列号 (SN)。

`CreateManagedThing` 请求示例：

```
{
 "Role": "CONTROLLER",
 "AuthenticationMaterialType": "WIFI_SETUP_QR_BAR_CODE",
 "AuthenticationMaterial": "SN:123456789524;UPC:829576019524"
}
```

有关更多信息，请参阅托管集成 API 参考 [CreateManagedThing](#) 中的。

## ( 可选 ) 获取托管事物

在ProvisioningStatus继续操作PRE\_ASSOCIATED之前，必须先完成托管事务。有关的更多信息 ProvisioningStatus，请参阅[设备配置](#)。使用 GetManagedThing API 验证您的托管事物是否存在且已准备好进行配置。有关更多信息，请参阅托管集成 API 参考[GetManagedThing](#)中的。

## 步骤 4：创建目录结构

为您的配置文件和证书创建目录。默认情况下，Hub 入职流程使用。/data/aws/iotmi/config/iotmi\_config.json

您可以在配置文件中为证书和私钥指定自定义路径。本指南使用默认路径/data/aws/iotmi/certs。

```
mkdir -p /data/aws/iotmi/config
mkdir -p /data/aws/iotmi/certs

/data/
 aws/
 iotmi/
 config/
 certs/
```

## 第 5 步：向中心设备添加身份验证材料

将证书和密钥复制到您的中心设备，然后创建设备特定的配置文件。在配置过程中，这些文件可在您的集线器和托管集成之间建立安全的通信。

### 复制索赔证书和密钥

- 将这些身份验证文件从 CreateProvisioningProfile API 响应复制到您的中心设备：
  - claim\_cert.pem: 索赔证书 ( 适用于所有设备 )
  - claim\_pk.key: 索赔证书的私钥

将两个文件都放在/data/aws/iotmi/certs目录中。

**⚠ Important**

以 PEM 格式存储证书和私钥时，请通过正确处理换行符来确保格式正确。对于 PEM 编码的文件，(\n) 必须将换行符替换为实际的行分隔符，因为仅存储转义的换行符以后将无法正确检索。

**📘 Note**

如果您使用安全存储，请将这些凭据存储在您的安全存储位置而不是文件系统中。有关更多信息，请参阅 [创建用于安全存储的自定义证书处理程序](#)。

## 步骤 6：创建设备配置文件

创建包含唯一设备标识符、证书位置和配置设置的配置文件。SDK 在集线器启动期间使用此文件对您的设备进行身份验证、管理配置状态和存储连接设置。

**📘 Note**

每台集线器设备都需要自己的配置文件，其中包含唯一的设备特定值。

使用以下过程创建或修改您的配置文件，然后将其复制到集线器。

- 创建或修改配置文件（队列配置）。

在设备配置文件中配置以下必填字段：

- 证书路径
  1. `iot_claim_cert_path`: 您的索赔证明的位置 (`claim_cert.pem`)
  2. `iot_claim_pk_path`: 您的私钥的位置 (`claim_pk.key`)
  3. 在实现安全存储证书处理程序时，这两个字段都使用 `SECURE_STORAGE`
- 连接设置
  1. `fp_template_name`: 之前的 `ProvisioningProfile` 名字。

2. `endpoint_url` : 来自 `RegisterCustomEndpoint` API 响应的托管集成终端节点 URL ( 一个区域内的所有设备都相同 )。
- 设备标识符
    1. SN: 与您的 `CreateManagedThing` API 调用匹配的设备序列号 ( 每台设备都是唯一的 )
    2. UPC来自您的 `CreateManagedThing` API 调用的通用产品代码 ( 此产品的所有设备都相同 )

```
{
 "ro": {
 "iot_provisioning_method": "FLEET_PROVISIONING",
 "iot_claim_cert_path": "<SPECIFY_THIS_FIELD>",
 "iot_claim_pk_path": "<SPECIFY_THIS_FIELD>",
 "fp_template_name": "<SPECIFY_THIS_FIELD>",
 "endpoint_url": "<SPECIFY_THIS_FIELD>",
 "SN": "<SPECIFY_THIS_FIELD>",
 "UPC": "<SPECIFY_THIS_FIELD>"
 },
 "rw": {
 "iot_provisioning_state": "NOT_PROVISIONED"
 }
}
```

## 配置文件的内容

查看 `iotmi_config.json` 文件内容。

## 内容

| 键                                    | 值                                                         | 由客户添加？ | 注意                           |
|--------------------------------------|-----------------------------------------------------------|--------|------------------------------|
| <code>iot_provisioning_method</code> | FLEET_PROVISIONING                                        | 是      | 指定要使用的配置方法。                  |
| <code>iot_claim_cert_path</code>     | 您指定的文件路径或 SECURE_STORAGE 。<br>例如， <code>/data/aws/</code> | 是      | 指定要使用的文件路径或 SECURE_STORAGE 。 |

| 键                       | 值                                                                                                  | 由客户添加？ | 注意                                                                          |
|-------------------------|----------------------------------------------------------------------------------------------------|--------|-----------------------------------------------------------------------------|
| iot_claim_pk_path       | iotmi/certs/claim_cert.pem<br>您指定的文件路径或 SECURE_STORAGE 。<br>例如， /data/aws/iotmi/certs/claim_pk.pem | 是      | 指定要使用的文件路径或 SECURE_STORAGE 。                                                |
| fp_template_name        | 队列配置模板名称应等于之前使用的名称。ProvisioningProfile                                                             | 是      | 等于之前使用的名称 ProvisioningProfile                                               |
| endpoint_url            | 托管集成的终端节点 URL。                                                                                     | 是      | 您的设备使用此 URL 连接到托管集成云。要获取此信息，请使用 <a href="#">RegisterCustomEndpointAPI</a> 。 |
| SN                      | 设备序列号。例如 AIDACKCEVSQ6C2EXAMPLE 。                                                                   | 是      | 您必须为每台设备提供此唯一信息。                                                            |
| UPC                     | 设备通用产品代码。例如 841667145075 。                                                                         | 是      | 您必须为设备提供此信息。                                                                |
| managed_tthing_id       | 托管事物的 ID。                                                                                          | 否      | 此信息将在集线器配置后通过入职流程添加。                                                        |
| iot_provisioning_state  | 供应状态。                                                                                              | 是      | 置备状态必须设置为 NOT_PROVISIONED 。                                                 |
| iot_permanent_cert_path | 物联网证书路径。例如 /data/aws/iotmi/iot_cert.pem 。                                                          | 否      | 此信息将在集线器配置后通过入职流程添加。                                                        |

| 键                                 | 值                                             | 由客户添加？ | 注意                            |
|-----------------------------------|-----------------------------------------------|--------|-------------------------------|
| iot_perma<br>nent_pk_path         | 物联网私钥文件路径。例如 /data/aws/iotmi/<br>iot_pk.pem 。 | 否      | 此信息将在集线器配置后通过入职流程添加。          |
| client_id                         | 将用于 MQTT 连接的客户端 ID。                           | 否      | 此信息稍后由集线器配置后的入门流程添加，以供其他组件使用。 |
| mqtt_keep<br>_alive_in<br>terval  | 射程为 30-1200，单位以秒为单位。默认值是 300。                 | 是      | 使用它来设置 MQTT 连接的保持连接间隔。        |
| event_man<br>ager_uppe<br>r_bound | 默认值是 500。                                     | 否      | 此信息稍后由集线器配置后的入门流程添加，以供其他组件使用。 |

## 第 7 步：将配置文件复制到集线器

将您的配置文件复制到 /data/aws/iotmi/config 或您的自定义目录路径。您将在入门过程中提供此 HubOnboarding 二进制文件路径。

### 用于舰队配置

```
/data/
 aws/
 iotmi/
 config/
 iotmi_config.json
 certs/
 claim_cert.pem
 claim_pk.key
```

# 加载设备并在集线器中对其进行操作

通过创建托管设备并将其连接到集线器，将设备设置为加载到托管集成中心。可以通过简单的设置或用户指导的设置将设备加载到集线器。

## 主题

- [设置简单，便于加载和操作设备](#)
- [在用户指导下进行设备加载和操作的设置](#)

## 设置简单，便于加载和操作设备

通过创建托管设备并将其连接到您的集线器，将您的设备设置为加载到托管集成中心。本节介绍使用简单设置完成设备上线流程的步骤。

## 先决条件

在尝试加载设备之前，请完成以下步骤：

- 将集线器设备载入托管集成中心。
- AWS CLI 从 [《托管集成 AWS CLI 命令参考》](#) 中安装最新版本的
- 订阅 [DEVICE\\_LIFE\\_CYCLE 事件通知](#)。

## 设置步骤

- [步骤 1：创建凭证柜](#)
- [第 2 步：将凭证柜添加到您的中心](#)
- [步骤 3：使用凭据创建托管事物。](#)
- [步骤 4：插入设备并检查其状态。](#)
- [步骤 5：获取设备功能](#)
- [步骤 6：向托管事物发送命令](#)
- [第 7 步：从集线器中移除托管内容](#)

## 步骤 1：创建凭证柜

为您的设备创建一个凭证柜。

## 创建凭证柜

- 使用 `create-credential-locker` 命令。执行此命令将触发所有制造资源的创建，包括 Wi-Fi 设置 key pair 和设备证书。

### create-credential-locker 示例

```
aws iot-managed-integrations create-credential-locker \
 --name "DEVICE_NAME"
```

### 响应：

```
{
 "Id": "LOCKER_ID"
 "Arn": "arn:aws:iotmanagedintegrations:AWS_REGION:AWS_ACCOUNT_ID:credential-
locker/LOCKER_ID"
 "CreatedAt": "2025-06-09T13:58:52.977000+08:00"
}
```

有关更多信息，请参阅《托管集成[create-credential-locker](#)命令参考》中的 AWS CLI 命令。

## 第 2 步：将凭证柜添加到您的中心

将凭证柜添加到您的中心。

向您的中心添加凭证柜

- 使用以下命令将凭证储物柜添加到您的集线器。

```
aws iotmi --region AWS_REGION --endpoint AWS_ENDPOINT update-managed-thing \
 --identifier "HUB_MANAGED_THING_ID" --credential-locker-id "LOCKER_ID"
```

## 步骤 3：使用凭据创建托管事物。

使用设备凭据创建托管事物。每台设备都需要自己的托管设备。

### 创建托管事物

- 使用 `create-managed-thing` 命令为您的设备创建托管事物。

## create-managed-thing 示例

```
#ZWAVE:
aws iot-managed-integrations create-managed-thing --role DEVICE \
--authentication-material '900137947003133...' \ #auth material from zwave qr code
--authentication-material-type ZWAVE_QR_BAR_CODE \
--credential-locker-id ${locker_id}

#ZIGBEE:
aws iot-managed-integrations create-managed-thing --role DEVICE \
--authentication-material 'Z:286...$I:A4DC00.' \ #auth material from zigbee qr code
--authentication-material-type ZIGBEE_QR_BAR_CODE \
--credential-locker-id ${locker_id}
```

### Note

Z-Wave 和 Zigbee 设备有不同的命令。

响应：

```
{
 "Id": "DEVICE_MANAGED_THING_ID"
 "Arn": "arn:aws:iotmanagedintegrations:AWS_REGION:AWS_ACCOUNT_ID:managed-thing/DEVICE_MANAGED_THING_ID"
 "CreatedAt": "2025-06-09T13:58:52.977000+08:00"
}
```

有关更多信息，请参阅《托管集成[create-managed-thing](#)命令参考》中的 AWS CLI 命令。

## 步骤 4：插入设备并检查其状态。

接通设备电源并检查其状态。

- 使用 `get-managed-thing` 命令检查设备的状态。必须激活您的托管事物。ProvisioningStatus 有关的更多信息 ProvisioningStatus，请参阅[设备配置](#)。

## get-managed-thing 示例

```
#KINESIS NOTIFICATION:
{
 "version": "1.0.0",
 "messageId": "4ac684bb7f4c41adbb2eccc1e7991xxx",
 "messageType": "DEVICE_LIFE_CYCLE",
 "source": "aws.iotmanagedintegrations",
 "customerAccountId": "12345678901",
 "timestamp": "2025-06-10T05:30:59.852659650Z",
 "region": "us-east-1",
 "resources": ["XXX"],
 "payload": {
 "deviceDetails": {
 "id": "1e84f61fa79a41219534b6fd57052XXX",
 "arn": "XXX",
 "createdAt": "2025-06-09T06:24:34.336120179Z",
 "updatedAt": "2025-06-10T05:30:59.784157019Z"
 },
 "status": "ACTIVATED"
 }
}
aws iot-managed-integrations get-managed-thing \
--identifier "DEVICE_MANAGED_THING_ID"
```

响应：

```
{
 "Id": "DEVICE_MANAGED_THING_ID"
 "Arn": "arn:aws:iotmanagedintegrations:AWS_REGION:AWS_ACCOUNT_ID:managed-thing/MANAGED_THING_ID"
 "CreatedAt": "2025-06-09T13:58:52.977000+08:00"
}
```

有关更多信息，请参阅《托管集成[get-managed-thing](#)命令参考》中的 AWS CLI 命令。

## 步骤 5：获取设备功能

使用 `get-managed-thing-capabilities` 命令获取您的终端节点 ID 并查看设备可能执行的操作列表。

## 获取设备的功能

- 使用 `get-managed-thing-capabilities` 命令并记下端点 ID。

### `get-managed-thing-capabilities` 示例

```
aws iotmi get-managed-thing-capabilities \
--identifier "DEVICE_MANAGED_THING_ID"
```

响应：

```
{
 "ManagedThingId": "1e84f61fa79a41219534b6fd57052cbc",
 "CapabilityReport": {
 "version": "1.0.0",
 "nodeId": "zw.FCB10009+06",
 "endpoints": [
 {
 "id": "ENDPOINT_ID"
 "deviceTypes": [
 "On/Off Switch"
],
 "capabilities": [
 {
 "id": "matter.OnOff@1.4",
 "name": "On/Off",
 "version": "6",
 "properties": [
 "OnOff"
],
 "actions": [
 "Off",
 "On"
],
 "events": []
 }
 ...
]
 }
]
 }
}
```

有关更多信息，请参阅《托管集成[get-managed-thing-capabilities](#)命令参考》中的 AWS CLI 命令。

## 步骤 6：向托管事物发送命令

使用该 `send-managed-thing-command` 命令向您的托管事物发送切换操作命令。

向你的托管事物发送命令

- 使用 `send-managed-thing-command` 命令向您的托管事物发送命令。

`send-managed-thing-command` 示例

```
json=$(jq -cr '.|@json') <<EOF
[
 {
 "endpointId": "1",
 "capabilities": [
 {
 "id": "matter.OnOff@1.4",
 "name": "On/Off",
 "version": "1",
 "actions": [
 {
 "name": "Toggle",
 "parameters": {}
 }
]
 }
]
 }
]
EOF
aws iot-managed-integrations send-managed-thing-command \
--managed-thing-id "DEVICE_MANAGED_THING_ID" --endpoints "ENDPOINT_ID"
```

### Note

这个例子使用了 `jq cli`，但你也可以传递整个字符串 `endpointId`

响应：

```
{
```

```
"TraceId": "TRACE_ID"
}
```

有关更多信息，请参阅《托管集成[send-managed-thing-command](#)命令参考》中的 AWS CLI 命令。

## 第 7 步：从集线器中移除托管内容

通过移除托管的东西来清理集线器。

### 删除托管事物

- 使用 `delete-managed-thing` 命令从设备中心移除托管内容。

`delete-managed-thing` 示例

```
aws iot-managed-integrations delete-managed-thing \
--identifier "DEVICE_MANAGED_THING_ID"
```

有关更多信息，请参阅《托管集成[delete-managed-thing](#)命令参考》中的 AWS CLI 命令。

#### Note

如果设备处于某种 `DELETE_IN_PROGRESS` 状态，请将该 `--force` 标志附加到 `delete-managed-thing` command

#### Note

对于 Z-Wave 设备，您需要在执行命令后将设备置于配对模式。

## 在用户指导下进行设备加载和操作的设置

通过创建托管设备并将其连接到集线器，将设备设置为加载到托管集成中心。本节介绍使用用户指导设置完成设备上线流程的步骤。

## 先决条件

在尝试加载设备之前，请完成以下步骤：

- 将集线器设备载入托管集成中心。
- AWS CLI 从 [《托管集成 AWS CLI 命令参考》](#) 中安装最新版本的
- 订阅 [设备发现状态事件通知](#)。

### 用户指导的设置步骤

- [先决条件：在 Z Wave 设备上启用配对模式](#)
- [步骤 1：开始设备发现](#)
- [步骤 2：查询发现任务 ID](#)
- [第 3 步：为您的设备创建托管内容](#)
- [步骤 4：查询托管事物](#)
- [第 5 步：获取托管事物功能](#)
- [步骤 6：向托管事物发送命令](#)
- [第 7 步：检查托管事物的状态](#)
- [第 8 步：从集线器中移除托管内容](#)

### 先决条件：在 Z Wave 设备上启用配对模式

在 Z-Wave 设备上启用配对模式。每台 Z-Wave 设备的配对模式可能有所不同，因此请参阅设备的说明以正确设置配对模式。它通常是用户必须按下的按钮。

### 步骤 1：开始设备发现

为您的集线器启动设备发现，以获取用于加载设备的发现任务 ID。

#### 开始设备发现

- 使用 [start-device-discovery](#) 命令获取发现任务 ID。

start-device-discovery 示例

```
#For Zigbee
aws iot-managed-integrations start-device-discovery --discovery-type ZIGBEE \
```

```
--controller-identifier HUB_MANAGED_THING_ID

#For Zwave
aws iot-managed-integrations start-device-discovery --discovery-type ZWAVE \
--controller-identifier HUB_MANAGED_THING \
--authentication-material-type ZWAVE_INSTALL_CODE \
--authentication-material 13333

#For Cloud
aws iot-managed-integrations start-device-discovery --discovery-type CLOUD \
--account-association-id C2C_ASSOCIATION_ID \

#For Custom
aws iot-managed-thing start-device-discovery --discovery-type CUSTOM \
--controller-identifier HUB_MANAGED_THING_ID \
--custom-protocol-detail NAME : NON_EMPTY_STRING \
```

响应：

```
{
 "Id": DISCOVERY_JOB_ID,
 "StartedAt": "2025-06-03T14:43:12.726000-07:00"
}
```

#### Note

Z-Wave 和 Zigbee 设备有不同的命令。

有关更多信息，请参阅《托管集成 AWS CLI 命令参考》中的 [start-device-discovery](#) API。

## 步骤 2：查询发现任务 ID

使用 `list-discovered-devices` 命令获取设备的身份验证材料。

查询您的发现任务 ID

- 使用发现任务 ID 和 `list-discovered-devices` 命令获取设备的身份验证材料。

```
aws iot-managed-integrations list-discovered-devices --identifier DISCOVERY_JOB_ID
```

响应：

```
"Items": [
 {
 "DeviceTypes": [],
 "DiscoveredAt": "2025-06-03T14:43:37.619000-07:00",
 "AuthenticationMaterial": AUTHENTICATION_MATERIAL
 }
]
```

### 第 3 步：为您的设备创建托管内容

使用 `create-managed-thing` 命令为您的设备创建托管事物。每台设备都需要自己的托管设备。

创建托管事物

- 使用 `create-managed-thing` 命令为您的设备创建托管事物。

`create-managed-thing` 示例

```
aws iot-managed-integrations create-managed-thing \
 --role DEVICE --authentication-material-type DISCOVERED_DEVICE \
 --authentication-material "AUTHENTICATION_MATERIAL"
```

响应：

```
{
 "Id": "DEVICE_MANAGED_THING_ID"
 "Arn": "arn:aws:iotmanagedintegrations:AWS_REGION:AWS_ACCOUNT_ID:managed-
 thing/DEVICE_MANAGED_THING_ID"
 "CreatedAt": "2025-06-09T13:58:52.977000+08:00"
}
```

有关更多信息，请参阅《托管集成 [create-managed-thing](#) 命令参考》中的 AWS CLI 命令。

### 步骤 4：查询托管事物

您可以使用 `get-managed-thing` 命令检查托管事物是否已激活。

## 查询托管事物

- 使用 `get-managed-thing` 命令检查托管事物的配置状态是否设置为 `ACTIVATED`。有关配置状态的更多信息，请参阅 [设备配置](#)。

### `get-managed-thing` 示例

```
aws iot-managed-integrations get-managed-thing \
 --identifier "DEVICE_MANAGED_THING_ID"
```

响应：

```
{
 "Id": "DEVICE_MANAGED_THING_ID",
 "Arn": "arn:aws:iotmanagedintegrations:AWS_REGION:AWS_ACCOUNT_ID:managed-
thing/DEVICE_MANAGED_THING_ID",
 "Role": "DEVICE",
 "ProvisioningStatus": "ACTIVATED",
 "MacAddress": "MAC_ADDRESS",
 "ParentControllerId": "PARENT_CONTROLLER_ID",
 "CreatedAt": "2025-06-03T14:46:35.149000-07:00",
 "UpdatedAt": "2025-06-03T14:46:37.500000-07:00",
 "Tags": {}
}
```

有关更多信息，请参阅《托管集成[get-managed-thing](#)命令参考》中的 AWS CLI 命令。

## 第 5 步：获取托管事物功能

您可以使用查看托管事物的可用操作列表 `get-managed-thing-capabilities`。

### 获取设备的功能

- 使用 `get-managed-thing-capabilities` 命令获取端点 ID。另请注意可能的操作列表。

### `get-managed-thing-capabilities` 示例

```
aws iot-managed-integrations get-managed-thing-capabilities \
 --identifier "DEVICE_MANAGED_THING_ID"
```

响应：

```
{
 "ManagedThingId": "DEVICE_MANAGED_THING_ID",
 "CapabilityReport": {
 "version": "1.0.0",
 "nodeId": "zb.539D+4A1D",
 "endpoints": [
 {
 "id": "1",
 "deviceTypes": [
 "Unknown Device"
],
 "capabilities": [
 {
 "id": "matter.OnOff@1.4",
 "name": "On/Off",
 "version": "6",
 "properties": [
 "OnOff",
 "OnOff",
 "OnTime",
 "OffWaitTime"
],
 "actions": [
 "Off",
 "On",
 "Toggle",
 "OffWithEffect",
 "OnWithRecallGlobalScene",
 "OnWithTimedOff"
],
 ...
 }
]
 }
]
 }
}
```

有关更多信息，请参阅《托管集成[get-managed-thing-capabilities](#)命令参考》中的 AWS CLI 命令。

## 步骤 6：向托管事物发送命令

您可以使用该 `send-managed-thing-command` 命令向您的托管事物发送切换操作命令。

使用切换操作向托管事物发送命令。

- 使用该 `send-managed-thing-command` 命令发送切换操作命令。

#### send-managed-thing-command 示例

```
json=$(jq -cr '.*|@json') <<EOF
[
 {
 "endpointId": "1",
 "capabilities": [
 {
 "id": "matter.OnOff@1.4",
 "name": "On/Off",
 "version": "1",
 "actions": [
 {
 "name": "Toggle",
 "parameters": {}
 }
]
 }
]
 }
]
EOF
aws iot-managed-integrations send-managed-thing-command \
--managed-thing-id ${device_managed_thing_id} --endpoints ENDPOINT_ID
```

#### Note

这个例子使用了 jq cli，但你也可以传递整个字符串 `endpointId`

响应：

```
{
 "TraceId": TRACE_ID
}
```

有关更多信息，请参阅《托管集成[send-managed-thing-command](#)命令参考》中的 AWS CLI 命令。

## 第 7 步：检查托管事物的状态

检查托管事物的状态以验证切换操作是否成功。

检查托管事物的设备状态

- 使用 `get-managed-thing-state` 命令验证切换操作是否成功。

`get-managed-thing-state` 示例

```
aws iot-managed-integrations get-managed-thing-state --managed-thing-id DEVICE_MANAGED_THING_ID
```

响应：

```
{
 "Endpoints": [
 {
 "endpointId": "1",
 "capabilities": [
 {
 "id": "matter.OnOff@1.4",
 "name": "On/Off",
 "version": "1.4",
 "properties": [
 {
 "name": "OnOff",
 "value": {
 "propertyValue": true,
 "lastChangedAt": "2025-06-03T21:50:39.886Z"
 }
 }
]
 }
]
 }
]
}
```

```
}
```

有关更多信息，请参阅《托管集成[get-managed-thing-state](#)命令参考》中的 AWS CLI 命令。

## 第 8 步：从集线器中移除托管内容

通过移除托管的东西来清理集线器。

### 删除托管事物

- 使用[delete-managed-thing](#)命令移除托管事物。

delete-managed-thing 示例

```
aws iot-managed-integrations delete-managed-thing \
 --identifier MANAGED_THING_ID
```

有关更多信息，请参阅《托管集成[delete-managed-thing](#)命令参考》中的 AWS CLI 命令。

#### Note

如果设备停滞在DELETE\_IN\_PROGRESS状态，delete-managed-thing请在命令后面附加--force标志。

#### Note

对于 Z-Wave 设备，您需要在执行命令后将设备置于配对模式。

## 创建用于安全存储的自定义证书处理程序

在加入托管集成中心时，设备证书管理至关重要。虽然默认情况下证书存储在文件系统中，但您可以创建自定义证书处理程序以增强安全性和灵活的凭据管理。

托管集成 End device SDK 为安全存储接口提供了证书处理程序，您可以将其实现为共享对象 (.so) 库。构建安全存储实现以读取和写入证书，然后在运行时将库文件链接到 HubOnboarding 进程。

## API 定义和组件

查看以下 `secure_storage_cert_handler_interface.hpp` 文件，了解您的实现的 API 组件和要求

主题

- [API 定义](#)
- [关键组件](#)

### API 定义

`secure_storage_cert_handler_interface.hpp` 的内容

```
/*
 * Copyright 2024 Amazon.com, Inc. or its affiliates. All rights reserved.
 *
 * AMAZON PROPRIETARY/CONFIDENTIAL
 *
 * You may not use this file except in compliance with the terms and
 * conditions set forth in the accompanying LICENSE.txt file.
 *
 * THESE MATERIALS ARE PROVIDED ON AN "AS IS" BASIS. AMAZON SPECIFICALLY
 * DISCLAIMS, WITH RESPECT TO THESE MATERIALS, ALL WARRANTIES, EXPRESS,
 * IMPLIED, OR STATUTORY, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT.
 */
#ifndef SECURE_STORAGE_CERT_HANDLER_INTERFACE_HPP
#define SECURE_STORAGE_CERT_HANDLER_INTERFACE_HPP

#include <iostream>
#include <memory>

namespace IoTManagedIntegrationsDevice {
namespace CertHandler {
/**
 * @enum CERT_TYPE_T
 * @brief enumeration defining certificate types.
 */
typedef enum { CLAIM = 0, DHA = 1, PERMANENT = 2 } CERT_TYPE_T;
class SecureStorageCertHandlerInterface {
public:
```

```
/**
 * @brief Read certificate and private key value of a particular certificate
 * type from secure storage.
 */
virtual bool read_cert_and_private_key(const CERT_TYPE_T cert_type,
 std::string &cert_value,
 std::string &private_key_value) = 0;

/**
 * @brief Write permanent certificate and private key value to secure storage.
 */
virtual bool write_permanent_cert_and_private_key(
 std::string_view cert_value, std::string_view private_key_value) = 0;
};

std::shared_ptr<SecureStorageCertHandlerInterface>
createSecureStorageCertHandler();
} //namespace CertHandler
} //namespace IoTManagedIntegrationsDevice

#endif //SECURE_STORAGE_CERT_HANDLER_INTERFACE_HPP
```

## 关键组件

- CERT\_TYPE\_T-集线器上不同类型的证书。
  - CLAIM-最初在集线器上的索赔证书将兑换成永久证书。
  - DHA-暂时未使用。
  - 永久-用于连接托管集成端点的永久证书。
- read\_cert\_and\_private\_key- ( 函数待实现 ) 将证书和密钥值读入参考输入。此函数必须能够读取 CLAIM 和永久证书，并根据上述证书类型进行区分。
- write\_permanent\_cert\_and\_private\_key- ( 函数待实现 ) 将永久证书和密钥值写入所需的位置。

## 示例构建

将内部实现标头与公共接口 (secure\_storage\_cert\_handler\_interface.hpp) 分开，以保持干净的项目结构。通过这种分离，您可以在构建证书处理程序的同时管理公用和私有组件。

### Note

宣布secure\_storage\_cert\_handler\_interface.hpp为公开。

## 主题

- [项目结构](#)
- [继承接口](#)
- [实施](#)
- [CMakeList.txt](#)

## 项目结构

## 继承接口

创建一个继承接口的具体类。将此头文件和其他文件隐藏在单独的目录下，以便在构建时可以轻松区分私有和公共标头。

```
#ifndef IOTMANAGEDINTEGRATIONSDEVICE_SDK_STUB_SECURE_STORAGE_CERT_HANDLER_HPP
#define IOTMANAGEDINTEGRATIONSDEVICE_SDK_STUB_SECURE_STORAGE_CERT_HANDLER_HPP

#include "secure_storage_cert_handler_interface.hpp"

namespace IoTManagedIntegrationsDevice::CertHandler {
class StubSecureStorageCertHandler : public SecureStorageCertHandlerInterface {
public:
 StubSecureStorageCertHandler() = default;

 bool read_cert_and_private_key(const CERT_TYPE_T cert_type,
 std::string &cert_value,
 std::string &private_key_value) override;

 bool write_permanent_cert_and_private_key(
 std::string_view cert_value, std::string_view private_key_value) override;
 /*
 * any other resource for function you might need
 */

};
}
#endif //IOTMANAGEDINTEGRATIONSDEVICE_SDK_STUB_SECURE_STORAGE_CERT_HANDLER_HPP
```

## 实施

实现上面定义的存储类，src/stub\_secure\_storage\_cert\_handler.cpp。

```
/*
 * Copyright 2024 Amazon.com, Inc. or its affiliates. All rights reserved.
 *
 * AMAZON PROPRIETARY/CONFIDENTIAL
 *
 * You may not use this file except in compliance with the terms and
 * conditions set forth in the accompanying LICENSE.txt file.
 *
 * THESE MATERIALS ARE PROVIDED ON AN "AS IS" BASIS. AMAZON SPECIFICALLY
 * DISCLAIMS, WITH RESPECT TO THESE MATERIALS, ALL WARRANTIES, EXPRESS,
 * IMPLIED, OR STATUTORY, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT.
 */

#include "stub_secure_storage_cert_handler.hpp"

using namespace IoTManagedIntegrationsDevice::CertHandler;

bool StubSecureStorageCertHandler::write_permanent_cert_and_private_key(
 std::string_view cert_value, std::string_view private_key_value) {
 // TODO: implement write function
 return true;
}

bool StubSecureStorageCertHandler::read_cert_and_private_key(const CERT_TYPE_T
cert_type,
 std::string &cert_value,
 std::string
&private_key_value) {
 std::cout<<"Using Stub Secure Storage Cert Handler, returning dummy values";
 cert_value = "StubCertVal";
 private_key_value = "StubKeyVal";
 // TODO: implement read function
 return true;
}
```

实现接口中定义的工厂函数src/secure\_storage\_cert\_handler.cpp。

```
#include "stub_secure_storage_cert_handler.hpp"

std::shared_ptr<IoTManagedIntegrationsDevice::CertHandler::SecureStorageCertHandlerInterface>
 IoTManagedIntegrationsDevice::CertHandler::createSecureStorageCertHandler() {
 // TODO: replace with your implementation
 return
std::make_shared<IoTManagedIntegrationsDevice::CertHandler::StubSecureStorageCertHandler>();
}
```

## CMakeList.txt

```
#project name must stay the same
project(SecureStorageCertHandler)

Public Header files. The interface definition must be in top level with exactly
the same name
#ie. Not in anotherDir/secure_storage_cert_handler_interface.hpp
set(PUBLIC_HEADERS
 ${PROJECT_SOURCE_DIR}/include
)

private implementation headers.
set(PRIVATE_HEADERS
 ${PROJECT_SOURCE_DIR}/internal/stub
)

#set all sources
set(SOURCES
 ${PROJECT_SOURCE_DIR}/src/secure_storage_cert_handler.cpp
 ${PROJECT_SOURCE_DIR}/src/stub_secure_storage_cert_handler.cpp
)

Create the shared library
add_library(${PROJECT_NAME} SHARED ${SOURCES})
target_include_directories(
 ${PROJECT_NAME}
 PUBLIC
 ${PUBLIC_HEADERS}
 PRIVATE
 ${PRIVATE_HEADERS}
```

```
)

Set the library output location. Location can be customized but version must
stay the same
set_target_properties(${PROJECT_NAME} PROPERTIES
 LIBRARY_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/../lib
 VERSION 1.0
 SOVERSION 1
)

Install rules
install(TARGETS ${PROJECT_NAME}
 LIBRARY DESTINATION lib
 ARCHIVE DESTINATION lib
)

install(FILES ${HEADERS}
 DESTINATION include/SecureStorageCertHandler
)
```

## 使用量

编译完成后，您将拥有一个libSecureStorageCertHandler.so共享的对象库文件及其关联的符号链接。将库文件和符号链接复制到 HubOnboarding 二进制文件所需的库位置。

### 主题

- [重要注意事项](#)
- [使用安全存储](#)

### 重要注意事项

- 验证您的用户帐户是否具有 HubOnboarding 二进制文件和libSecureStorageCertHandler.so库的读写权限。
- 保留secure\_storage\_cert\_handler\_interface.hpp为唯一的公共头文件。所有其他头文件都应保留在您的私有实现中。

- 验证您的共享对象库名称。在构建时libSecureStorageCertHandler.so , HubOnboarding 可能需要在文件名中使用特定的版本，例如libSecureStorageCertHandler.so.1.0。使用ldd命令检查库依赖关系并根据需要创建符号链接。
- 如果共享库的实现具有外部依赖关系，请将其存储在 HubOnboarding 可以访问的目录中，例如/usr/lib or the iotmi\_common目录。

## 使用安全存储

通过将iot\_claim\_cert\_path和iot\_claim\_pk\_path都设置为来更新您的iotmi\_config.json文件**SECURE\_STORAGE**。

```
{
 "ro": {
 "iot_provisioning_method": "FLEET_PROVISIONING",
 "iot_claim_cert_path": "SECURE_STORAGE",
 "iot_claim_pk_path": "SECURE_STORAGE",
 "fp_template_name": "device-integration-example",
 "iot_endpoint_url": "[ACCOUNT-PREFIX]-ats.iot.AWS-REGION.amazonaws.com",
 "SN": "1234567890",
 "UPC": "1234567890"
 },
 "rw": {
 "iot_provisioning_state": "NOT_PROVISIONED"
 }
}
```

## 自定义协议插件

您可以使用自定义协议插件将您的专有物联网协议集成到 AWS IoT Device Management 生态系统的托管集成中。通过定义明确的 SDK 接口，您可以加载设备、定义功能和处理实时控制流程，同时保持与托管集成和集线器 SDK 组件的完全兼容性。

以下列表讨论了自定义协议插件的主要功能。

### 数据模型自定义

定义您自己的 AWS 数据模型架构，并在配置流程中将其上传到托管集成。您可以稍后在工作流程中使用这些架构。

## 灵活的插件实现

- 使用构建自己的插件组件[中心 SDK 客户端](#)。
- 为不同的功能（例如配置和控制）实施单独的插件，或者为两者创建统一的客户端。
- 在托管集成资产和您自己的资产（例如中间件堆栈）之间保持明确的界限，以实现解耦且便于开发的代码逻辑。

## 向后兼容

对于现有客户，可以顺利加入新的自定义协议，同时保持现有无线电类型按原样工作。

下图说明了自定义协议插件架构。

## 中心 SDK 客户端

Hub SDK 客户端库充当托管集成 Hub SDK 和在同一集线器上运行的您自己的协议堆栈之间的接口。它公开了一组公共信息，APIs 以方便您的协议栈与 Device Hub SDK 组件的交互。用例包括自定义插件控件、自定义插件供应器和本地控制器。

### 主题

- [获取您的托管集成 Hub SDK](#)
- [关于 Hub SDK 工具包](#)
- [使用 Hub SDK 客户端创建您的自定义应用程序](#)
- [运行您的自定义应用程序](#)
- [Hub 软件开发工具包客户端 API](#)
- [数据类型](#)

## 获取您的托管集成 Hub SDK

Hub SDK 客户端附带托管集成 SDK。请通过[托管集成控制台](#)联系我们，访问 Hub SDK。

## 关于 Hub SDK 工具包

下载后，您将看到一个IotMI-DeviceSDK-Toolkit文件夹，其中包含所有公共头.so文件和可以在应用程序中使用的文件。托管集成团队还提供了一个main.cpp用于演示目的的示例，以及您可以直接执行的演示应用程序二进制文件。bin/您可以选择将其用作应用程序的起点。

## 使用 Hub SDK 客户端创建您的自定义应用程序

使用以下步骤创建您的自定义应用程序。

1. 在应用程序中包含头文件 (.h) 和共享目标文件 (.so)。

您必须在应用程序中包含公共头文件 (.h) 和共享目标文件 (.so)。对于 .so 文件，您可以将它们放在 lib 文件夹中。最终布局将如下所示：

```
include
iotmi_device_sdk_client
iotmi_device_sdk_client_common_types.h
iotmi_device_sdk_client.h
iotshd_status.h
lib
libiotmi_devicesdk_client_module.so
libiotmi_log_c.so
```

2. 在主应用程序中创建 Hub SDK 客户端。
  - a. 在主应用程序中，必须先初始化 Hub SDK 客户端，然后才能使用它来处理请求。你可以简单地使用来构造客户端 `clientId`。
  - b. 拥有客户端后，您可以将其连接到托管集成设备 SDK。

以下是创建 Hub SDK 客户端以及如何连接的示例。

```
#include <cstdlib>
#include <string>
#include "iotshd_status.h"
#include "iotmi_device_sdk_client.h"

auto client = std::make_unique<DeviceSDKClient>(your_own_clientId);
iotmi_statusCode_t status = client->connect();
```

### Note

`your_own_clientId` 必须与您在用户指导设置 [start-device-discovery](#) 中或简单设置配置流程 [create-managed-thing](#) 中指定的设置相同。

3. 通过执行以下步骤进行发布和订阅。

- a. 建立连接后，您现在可以从托管集成 Hub SDK 订阅传入的任务。传入的任务可以是控制任务或预配任务。您还需要根据收到的任务定义自己的回调函数，并定义自己的自定义上下文以用于自己的跟踪目的。

```
// subscribe to provisioning tasks
iotmi_statusCode_t status = client->iotmi_provision_subscribe_to_tasks(
 example_subscriber_callback, custom_context);

// subscribe to control tasks
iotmi_statusCode_t status = client->iotmi_control_subscribe_to_tasks(
 example_subscriber_callback, custom_context);
```

- b. 建立连接后，您现在可以将来自应用程序的请求发布到托管集成 Hub SDK。您可以定义自己的任务消息类型，为不同的业务目的使用不同的负载。这些请求可以包括控制请求和配置请求，类似于订阅流程。最后，你可以为你分配一个地址，让你rspPayload以同步的方式从托管集成 Hub SDK 获得响应。

```
// publish control request
iotmi_client_request_t api_payload = {
 .messageType = C2MIMessageType::C2MI_CONTROL_EVENT,
 .reqPayload = (uint8_t *)"define_your_req_payload",
 .rspPayload = (uint8_t *)calloc(1000, sizeof(uint8_t))
};

status = client->iotmi_control_publish_request(&api_payload);

// publish provision request
iotmi_client_request_t api_payload = {
 .messageType = C2MIMessageType::C2MI_DEVICE_ONBOARDED,
 .reqPayload = (uint8_t *)"define_your_req_payload",
 .rspPayload = (uint8_t *)calloc(1000, sizeof(uint8_t))
};

status = client->iotmi_provision_publish_request(&api_payload);
```

4. 自己构建CMakeLists.txt，然后从那里构建应用程序。最终输出可能是可执行的二进制文件，例如 MyFirstApplication

## 运行您的自定义应用程序

在运行自定义应用程序之前，请完成以下步骤来设置中心并启动托管集成 Hub SDK：

- 请按照中的入职说明进行操作。[将您的集线器加入托管集成](#)
- 完成中记录的安装过程[安装并验证托管集成 Hub SDK](#)。

满足先决条件后，您就可以运行您的自定义应用程序了。例如：

```
./MyFirstApplication
```

### Important

必须[使用脚本部署 Hub SDK](#)使用脚本手动更新中列出的启动脚本才能启动自己的应用程序。顺序很重要，不要更改顺序。

更新以下内容。更改

```
./IotMI-DeviceSDK-Toolkit/bin/DeviceSDKClientDemo >> $LOGS_DIR/
logDeviceSDKClientDemo_logs.txt &
```

到

```
./MyFirstApplication >> $LOGS_DIR/MyFirstApplication_logstxt &
```

## Hub 软件开发工具包客户端 API

Hub SDK 客户端 (设备SDKClient类) 为您的自定义应用程序提供了一个接口，用于与托管集成设备 SDK 进行交互。使用此客户端，您可以执行以下操作：

- 从托管集成组件中订阅与配置和控制相关的任务。
- 向托管集成组件发布与配置和控制相关的请求。

有关托管集成的信息 AWS IoT Device Management APIs，请参阅[什么是 AWS Lambda](#)。

主题

- [客户端初始化](#)

- [预配任务订阅](#)
- [置备任务发布](#)
- [控制任务订阅](#)
- [控制任务发布](#)
- [日志功能](#)
- [其他 APIs](#)

## 客户端初始化

要开始使用DeviceSDKClient，请使用客户端 ID 对其进行初始化。

```
iotmi_statusCode_t DeviceSDKClient(const std::string& clientId)
```

这将使用指定的创建一个新DeviceSDKClient实例clientId。clientId必须与您在托管集成中注册的相匹配。

### 参数

clientId ( 字符串 ) -此实例的客户端 ID。

```
connect()
```

将DeviceSDKClient实例连接到托管集成。

### 返回值

- IOTMI\_STATUS\_OK-连接成功。
- IOTMI\_STATUS\_CUSTOM\_PLUGIN\_CONNECTION\_ERROR-连接到托管集成时出错。

## 预配任务订阅

使用这些方法从托管集成组件订阅与配置相关的任务。

```
iotmi_statusCode_t
iotmi_provision_subscribe_to_tasks(DeviceSDKClient_SubscriberCallback callback, char*
context)
```

从托管集成组件订阅与配置相关的任务，例如设备启动和取消配置。

### 参数

- `callback(Device SDKClient_SubscriberCallback)`-收到任务时执行的回调函数。
- `context(char*)`-传递给回调函数的自定义上下文。

### 返回值

- `IOTMI_STATUS_OK`-订阅成功。
- `IOTMI_STATUS_CUSTOM_PLUGIN_CLIENT_NOT_CONNECTED`-设备SDKClient 实例未连接到托管集成。
- `IOTMI_STATUS_CUSTOM_PLUGIN_SUBSCRIBE_ERROR`-订阅任务时出错。

## 置备任务发布

使用这些方法向托管集成组件发布与配置相关的请求。

```
iotmi_statusCode_t iotmi_provision_publish_request(DataModel::iotmi_client_request_t request)
```

向托管集成组件发布与配置相关的请求。例如，设备已载入事件或取消配置状态

### 参数

`request(DataModel:: iotmi_client_request_t)`-指向包含详细信息的请求结构的指针。

### 返回值

- `IOTMI_STATUS_OK`-请求已成功发布。
- `IOTMI_STATUS_CUSTOM_PLUGIN_CLIENT_NOT_CONNECTED`-该DeviceSDKClient实例未连接到托管集成。
- `IOTMI_STATUS_INVALID_PARAMETER`-请求中的一个或多个参数无效。
- `IOTMI_STATUS_INVALID_JSON_OBJECT`-请求负载不是有效的 JSON 对象。
- `IOTMI_STATUS_NO_MEMORY`-发生内存分配错误。

## 控制任务订阅

使用这些方法订阅托管集成组件中与控制相关的任务。

```
iotmi_statusCode_t iotmi_control_subscribe_to_tasks(DeviceSDKClient_SubscriberCallback
callback, char context)
```

从托管集成组件订阅与控制相关的任务（例如设备控制请求）。

### 参数

- `callback(Device SDKClient_SubscriberCallback)`-收到任务时执行的回调函数。
- `context(char)`-传递给回调函数的自定义上下文。

### 返回值

- `IOTMI_STATUS_OK`-订阅成功。
- `IOTMI_STATUS_CUSTOM_PLUGIN_CLIENT_NOT_CONNECTED`-该DeviceSDKClient实例未连接到托管集成。
- `IOTMI_STATUS_CUSTOM_PLUGIN_SUBSCRIBE_ERROR`-订阅任务时出错。

## 控制任务发布

使用这些方法向托管集成组件发布与控制相关的请求。

```
iotmi_statusCode_t iotmi_control_publish_request(DataModel::iotmi_client_request_t
request)
```

向托管集成组件发布与控制相关的请求。例如，未经请求的事件、命令请求或设备状态查询。

### 参数

`request(DataModel:: iotmi_client_request_t)`-指向包含详细信息的请求结构的指针。

### 返回值

- `IOTMI_STATUS_OK`-请求已成功发布。
- `IOTMI_STATUS_CUSTOM_PLUGIN_CLIENT_NOT_CONNECTED`-设备SDKClient实例未连接到托管集成。
- `IOTMI_STATUS_INVALID_PARAMETER`-请求中的一个或多个参数无效。
- `IOTMI_STATUS_INVALID_JSON_OBJECT`-请求负载不是有效的JSON对象。
- `IOTMI_STATUS_NO_MEMORY`-发生内存分配错误。

## 日志功能

使用这些方法来实现托管集成提供的日志功能。

### 记录器初始化

```
void iotmi_devicesdk_log_init(const char* logger_name)
```

在使用任何日志记录功能之前，必须初始化记录器。

### 参数

`logger_name`-您指定的记录器名称。默认值为：`MyApplication`

### 记录宏

```
LOGGER_LOGD(...)
```

在应用程序中使用此宏进行调试级别的日志记录。

```
LOGGER_LOGI(...)
```

在应用程序中使用此宏进行信息级别的日志记录。

```
LOGGER_LOGW(...)
```

在应用程序中使用此宏进行警告级别的日志记录。

```
LOGGER_LOGE(...)
```

在应用程序中使用此宏进行错误级别的日志记录。

#### Note

有关日志功能的更多信息，请参阅 [Hub 日志记录文档](#)。自定义协议插件完全支持托管集成提供的所有日志功能。

## 其他 APIs

```
std::string get_client_id()
```

返回与设备SDKClient 实例关联的客户端 ID。

返回值

客户端 ID。

## 数据类型

本节定义了用于自定义协议插件的数据类型。

iotmi\_client\_request\_t

表示要发布到托管集成组件的请求。

消息类型

消息的类型 (CommonTypes:: C2 MIMessage 类型)。以下列表显示了有效值。

- C2MI\_DEVICE\_ONBOARDED: 表示带有相关负载的设备启动消息。
- C2MI\_DE\_PROVISIONING\_PRE\_ASSOCIATED\_COMPLETE : 表示预关联设备的取消配置任务完成通知。
- C2MI\_DE\_PROVISIONING\_ACTIVATED\_COMPLETE : 表示已激活设备的取消配置任务完成通知。
- C2MI\_DE\_PROVISIONING\_COMPLETE\_RESPONSE : 表示取消置备任务已完成响应。
- C2MI\_CONTROL\_EVENT : 表示可能发生设备状态变化的控制事件。
- C2MI\_CONTROL\_SEND\_COMMAND: 表示来自本地控制器的控制命令。
- C2MI\_CONTROL\_SEND\_DEVICE\_STATE\_QUERY: 表示来自本地控制器的控制设备状态查询。

reqPayl

请求负载，通常是 JSON 格式的字符串。

RSPPayload

由托管集成组件填充的响应有效负载。

iotmi\_client\_event\_t

表示从托管集成组件收到的事件。

事件\_id

事件的唯一标识符。

## length

事件数据的长度。

## data

指向事件数据的指针，包括messageType。以下列表显示了可能的值。

- C2MI\_PROVISION\_UGS\_TASK：表示 UGS 流程的配置任务。
- C2MI\_PROVISION\_SS\_TASK：表示 SimpleSetup 流程的置备任务。
- C2MI\_DE\_PROVISION\_PRE\_ASSOCIATED\_TASK：表示预关联设备的取消配置任务。
- C2MI\_DE\_PROVISION\_ACTIVATED\_TASK：表示已激活设备的取消配置任务。
- C2MI\_DEVICE\_ONBOARDED\_RESPONSE：表示设备加载响应。
- C2MI\_CONTROL\_TASK：表示控制任务。
- C2MI\_CONTROL\_EVENT\_NOTIFICATION：表示本地控制器的控制事件通知。

## ctx

与事件关联的自定义上下文。

## 集线器控制

集线器控件是托管集成终端设备 SDK 的扩展，允许它与 Hub SDK 中的MQTTProxy组件交互。借助集线器控制，您可以使用终端设备 SDK 实现代码，并通过托管集成云作为单独的设备控制您的集线器。集线器控制 SDK 将作为单独的软件包在 Hub SDK 中提供，标记为iot-managed-integrations-hub-control-x.x.x。

### 主题

- [先决条件](#)
- [终端设备 SDK 组件](#)
- [与终端设备 SDK 集成](#)
- [示例：构建集线器控件](#)
- [支持的示例](#)
- [支持的平台](#)

## 先决条件

要设置集线器控制，您需要满足以下条件：

- 已载入 Hub [SDK 的集线器](#)，版本 0.4.0 或更高版本。
- 从下载最新版本的[终端设备 SDK](#) AWS 管理控制台。
- 在集线器上运行的 [MQTT 代理](#)组件，版本 0.5.0 或更高版本。

## 终端设备 SDK 组件

使用[终端设备 SDK](#) 中的以下组件：

- 数据模型的代码生成器
- 数据模型处理器

由于 Hub SDK 已经具有入门流程和云端连接，因此您不需要以下组件：

- 预备人
- PKCS 接口
- 作业处理者
- MQTT 代理

## 与终端设备 SDK 集成

1. 按照[数据模型代码生成器](#)中的说明生成低级 C 代码。
2. 按照[集成终端设备 SDK](#) 中的说明执行以下操作：
  - a. 设置构建环境

作为开发主机，在亚马逊 Linux 2023/x86\_64 上构建代码。安装必要的编译依赖项：

```
dnf install make gcc gcc-c++ cmake
```

- b. 开发硬件回调函数

在实现硬件回调函数之前，请先了解 API 的工作原理。此示例使用 On/Off 群集和 OnOff 属性来控制设备功能。有关 API 的详细信息，请参阅[低级 C 函数 APIs](#)。

```
struct DeviceState
{
 struct iotmiDev_Agent *agent;
```

```

struct iotmiDev_Endpoint *endpointLight;
/* This simulates the HW state of OnOff */
bool hwState;
};

/* This implementation for OnOff getter just reads
the state from the DeviceState */
iotmiDev_DMStatus exampleGetOnOff(bool *value, void *user)
{
 struct DeviceState *state = (struct DeviceState *) (user);
 *value = state->hwState;
 return iotmiDev_DMStatusOk;
}

```

### c. 设置端点并挂接硬件回调函数

实现函数后，创建端点并注册您的回调。完成以下任务：

- i. 创建设备代理
- ii. 为要支持的每个集群结构填充回调函数点
- iii. 设置终端节点并注册支持的集群

```

struct DeviceState
{
 struct iotmiDev_Agent * agent;
 struct iotmiDev_Endpoint *endpoint1;

 /* OnOff cluster states*/
 bool hwState;
};

/* This implementation for OnOff getter just reads
the state from the DeviceState */
iotmiDev_DMStatus exampleGetOnOff(bool * value, void * user)
{
 struct DeviceState * state = (struct DeviceState *) (user);
 *value = state->hwState;
 printf("%s(): state->hwState: %d\n", __func__, state->hwState);
 return iotmiDev_DMStatusOk;
}

```

```
iotmiDev_DMStatus exampleGetOnTime(uint16_t * value, void * user)
{
 *value = 0;
 printf("%s(): OnTime is %u\n", __func__, *value);
 return iotmiDev_DMStatusOk;
}

iotmiDev_DMStatus exampleGetStartupOnOff(iotmiDev_OnOff_StartUpOnOffEnum *
value, void * user)
{
 *value = iotmiDev_OnOff_StartUpOnOffEnum_Off;
 printf("%s(): StartupOnOff is %d\n", __func__, *value);
 return iotmiDev_DMStatusOk;
}

void setupOnOff(struct DeviceState *state)
{
 struct iotmiDev_clusterOnOff clusterOnOff = {
 .getOnOff = exampleGetOnOff,
 .getOnTime = exampleGetOnTime,
 .getStartupOnOff = exampleGetStartupOnOff,
 };
 iotmiDev_OnOffRegisterCluster(state->endpoint1,
 &clusterOnOff,
 (void *) state);
}

/* Here is the sample setting up an endpoint 1 with OnOff
cluster. Note all error handling code is omitted. */
void setupAgent(struct DeviceState *state)
{
 struct iotmiDev_Agent_Config config = {
 .thingId = IOTMI_DEVICE_MANAGED_THING_ID,
 .clientId = IOTMI_DEVICE_CLIENT_ID,
 };
 iotmiDev_Agent_InitDefaultConfig(&config);

 /* Create a device agent before calling other SDK APIs */
 state->agent = iotmiDev_Agent_new(&config);

 /* Create endpoint#1 */
 state->endpoint1 = iotmiDev_Agent_addEndpoint(state->agent,
 1,
```

```
Device",
 "Data Model Handler Test"
},
{ "Camera" },
 (const char*[])
 1);
 setupOn0ff(state);
}
```

## 示例：构建集线器控件

集线器控件作为 Hub SDK 包的一部分提供。集线器控制子包标有与未经修改的设备 SDK 不同的库，`iot-managed-integrations-hub-control-x.x.x`并且包含不同的库。

1. 将代码生成的文件移到文件`example`夹：

```
cp codegen/out/* example/dm
```

2. 要构建集线器控件，请运行以下命令：

```
cd <hub-control-root-folder>
```

```
mkdir build
```

```
cd build
```

```
cmake -DBUILD_EXAMPLE_WITH_MQTT_PROXY=ON -
DIOTMI_USE_MANAGED_INTEGRATIONS_DEVICE_LOG=ON ..
```

```
cmake -build .
```

3. 使用集线器上的MQTTProxy组件运行示例，并运行HubOnboarding和MQTTProxy组件。

```
./examples/iotmi_device_sample_camera/iotmi_device_sample_camera
```

[托管集成数据模型](#)有关数据模型，请参阅。按照中的步骤 5 [开始使用终端设备 SDK](#) 设置终端并管理最终用户与之间的通信 `iot-managed-integrations`。

## 支持的示例

已构建并测试了以下示例：

- `iotmi_device_dm_air_purifier_demo`
- `iotmi` 设备基本诊断
- `iotmi_device_dm_camera_demo`

## 支持的平台

下表显示了支持的集线器控制平台。

| 架构      | 操作系统  | 海湾合作委员会版本 | Binutils 版本 |
|---------|-------|-----------|-------------|
| X86_64  | Linux | 10.5.0    | 2.37        |
| aarch64 | Linux | 10.5.0    | 2.37        |

## 启用 CloudWatch 日志

Hub SDK 提供全面的日志记录功能。默认情况下，Hub SDK 会将日志写入本地文件系统。但是，您可以利用云 API 将日志流配置为 CloudWatch 日志，它提供：

- **监控设备性能**：捕获详细的运行时日志，进行主动设备管理。在您的设备群中启用高级日志分析和监控
- **故障排除**：生成精细的日志条目以进行快速诊断分析。记录系统和应用程序级事件以进行深入调查。
- **灵活而集中的日志记录**：无需直接访问设备即可进行远程日志管理。将来自多个设备的日志聚合到一个可搜索的存储库中。

## 先决条件

- 将受管设备载入云端。有关详细信息，请参阅[Hub 入职设置](#)。
- 验证 Hub 代理已启动并成功初始化。有关详细信息，请参阅[安装并验证托管集成 Hub SDK](#)。

**Note**

要创建日志配置，详情请参阅 [PutRuntimeLogConfiguration API](#)。

**Warning**

启用日志计入分层配额计量。增加日志级别将导致更高的消息量和额外的成本。

## 设置 Hub SDK 日志配置

通过调用 API 来设置运行时日志配置，配置 Hub SDK 日志设置。

Example API 请求示例

```
aws iot-managed-integrations put-runtime-log-configuration \
 --managed-thing-id MANAGED_THING_ID \
 --runtime-log-configurations LogLevel=DEBUG,UploadLog=TRUE
```

### RuntimeLogConfigurations 属性

以下属性是可选的，可以在 RuntimeLogConfigurations API 中进行配置。

#### LogLevel

设置运行时跟踪的最低严重性级别。值：DEBUG, ERROR, INFO, WARN

默认：WARN ( 已发布版本 )

#### LogFlushLevel

确定立即将数据刷新到本地存储的严重性级别。值：DEBUG, ERROR, INFO, WARN

默认值：DISABLED

#### LocalStoreLocation

指定运行时跟踪的存储位置。默认值：/var/log/awsiotmi

- 活动日志：/var/log/awsiotmi/ManagedIntegrationsDeviceSdkHub.log
- 轮换日志：/var/log/awsiotmi/ManagedIntegrationsDeviceSdkHub.N.log ( N 表示轮换顺序 )

## LocalStoreFileRotationMaxBytes

当当前文件超过指定大小时触发文件轮换。

### Important

为了获得最佳效率，请将文件大小保持在 125 KB 以下。将自动限制大于 125 KB 的值。

## LocalStoreFileRotationMaxFiles,

设置日志守护程序允许的最大轮换文件数。

## UploadLog

控制将运行时跟踪传输到云端。日志存储在 /aws/iotmanagedintegration CloudWatch 日志组中。

默认值：false。

## UploadPeriodMinutes

定义运行时跟踪上传的频率。默认值：5

## DeleteLocalStoreAfterUpload

控制上传后的文件删除。默认值：true

### Note

如果设置为 false，则上传的文件将重命名为：`/var/log/awsiotmi/ManagedIntegrationsDeviceSdkHub.uploaded.{uploaded_timestamp}`

## 示例日志文件

参见下面的 CloudWatch 日志文件示例：

## 支持的 Zigbee 和 Z-Wave 设备类型

本页列出了已通过托管集成测试并受支持的与集线器连接的设备类型。托管集成同时支持[简单设置 \(SS\)](#)和[用户指导设置 \(UGS\)](#)适用于这些设备。

此表列出了支持的 ZigBee 设备。

| Zigbee 设备类型     | 支持的功能                                                       |
|-----------------|-------------------------------------------------------------|
| 智能灯泡/可调光灯/RGB 灯 | OnOff, LevelControl, ColorControl                           |
| 智能插头            | OnOff                                                       |
| 智能开关            | OnOff                                                       |
| LED 灯条          | OnOff, LevelControl, ColorControl                           |
| 水阀              | OnOff                                                       |
| 散热器阀            | 恒温器, 计时 OnOff器                                              |
| 恒温器             | 恒温器、FanControl、定时 OnOff器                                    |
| 车库门开启器          | WindowCovering, OnOff, LevelControl                         |
| 烟雾报警器           | BooleanState, OnOff TemperatureMeasurement, 计时器, 烟雾 COAlarm |
| 运动传感器           | BooleanState                                                |
| 占用/人体存在传感器      | BooleanState, OccupancySensing                              |
| 门窗传感器           | BooleanState                                                |
| 漏水传感器           | BooleanState                                                |
| 振动传感器           | BooleanState                                                |
| 温度和湿度传感器        | TemperatureMeasurement, RelativeHumidityMeasurement         |

下表列出了支持的 Z-Wave 设备。

| Z-Wave 设备类型 | 支持的功能                                                   |
|-------------|---------------------------------------------------------|
| 智能灯泡/可调光灯   | OnOff, LevelControl                                     |
| 智能插头        | OnOff                                                   |
| 车库门控制器      | OnOff, LevelControl                                     |
| 能量计         | ElectricalEnergyMeasurement, ElectricalPowerMeasurement |
| 电池          | LevelControl                                            |
| 警笛          | LevelControl                                            |
| 运动传感器       | BooleanState                                            |
| 门窗传感器       | BooleanState                                            |
| 漏水传感器       | BooleanState                                            |
| 温度传感器       | TemperatureMeasurement                                  |
| 一氧化碳传感器     | 抽烟 COAlarm                                              |
| 烟雾传感器       | 抽烟 COAlarm                                              |

## 在树莓派上运行托管集成

### Note

Raspberry Pi 上的 AWS IoT Hub SDK 的实现是一个演示项目，仅用于学习和测试目的，不打算在生产环境中使用。在本演示中，为了便于开发，请设置以下配置：

**AWS 凭证存储：**仅出于演示目的，凭证和证书存储在可访问的位置，便于测试和开发。生产环境必须使用安全的存储解决方案 AWS Secrets Manager，例如或 Systems Manager Parameter Store。他们必须实现静态加密，并遵循 AWS IoT 安全准则。

**容器权限：**该演示以提升的权限运行，允许不受限制地访问主机资源并简化开发工作流程。在生产环境中，容器应以最低要求的权限运行。

**网桥配置：**该演示使用网络桥接配置，该配置可公开内部网络流量，便于调试和监控。在生产环境中，实施适当的网络隔离和分段，以防止未经授权访问内部网络流量。

**USB 设备权限：**启用不受限制的 USB 设备访问权限，便于轻松连接开发外围设备和测试设备。在生产环境中，实施严格的 USB 设备控制和验证，以防止设备欺骗攻击。

这些配置支持直接测试，不得在生产环境中使用。部署到生产环境时，请遵循安全最佳实践，以防止主机系统受损和未经授权访问凭证。

作为先决条件，在设置 Raspberry Pi 之前，你必须设置 Sonoff Zigbee USB 加密狗。

## 将固件刷入 Sonoff Zigbee USB 加密狗

### 先决条件

- [Sonoff Zigbee USB 加密狗](#)
- Windows：安装 [CP210x 通用 Windows 驱动程序](#)

### 刷新固件

1. 下载 [Zigbee 加密狗固件版本 7.4.1.0](#)。
2. 打开 [Silabs 固件闪存器](#)。
3. 将 Sonoff Zigbee USB Dongle 连接到你的电脑。
4. 滚动并找到 ZBDongle-E。
5. 选择连接。
6. 等待设备连接。
7. 选择“更改固件”。
8. 选择“上传自己的固件”。
9. 找到 [Zigbee Dongle Firmware Build 7.4.1.0 下载](#)的位置并将其选中。
10. 单击 Install (安装)。
11. 等待固件安装。
12. 安装完成后选择“继续”。

解密器现已准备就绪，可以使用。

在下面列出的选项中进行选择，在 Raspberry Pi 上运行托管集成 Hub SDK。下面列出了两种方法的设置和验证步骤。

## 主题

- [树莓派上的托管集成 Hub SDK 镜像](#)
- [托管集成树莓派上的 Hub SDK Docker 容器](#)
- [托管集成演示应用程序](#)

## 树莓派上的托管集成 Hub SDK 镜像

### Note

Raspberry Pi 上的 AWS IoT Hub SDK 的实现是一个演示项目，仅用于学习和测试目的，不打算在生产环境中使用。在本演示中，为了便于开发，请设置以下配置：

**AWS 凭证存储：**仅出于演示目的，凭证和证书存储在可访问的位置，便于测试和开发。生产环境必须使用安全的存储解决方案 AWS Secrets Manager，例如或 Systems Manager Parameter Store。他们必须实现静态加密，并遵循 AWS IoT 安全准则。

**容器权限：**该演示以提升的权限运行，允许不受限制地访问主机资源并简化开发工作流程。在生产环境中，容器应以最低要求的权限运行。

**网桥配置：**该演示使用网络桥接配置，该配置可公开内部网络流量，便于调试和监控。在生产环境中，实施适当的网络隔离和分段，以防止未经授权访问内部网络流量。

**USB 设备权限：**启用不受限制的 USB 设备访问权限，便于轻松连接开发外围设备和测试设备。在生产环境中，实施严格的 USB 设备控制和验证，以防止设备欺骗攻击。

这些配置支持直接测试，不得在生产环境中使用。部署到生产环境时，请遵循安全最佳实践，以防止主机系统受损和未经授权访问凭证。

## 先决条件

在部署 Raspberry Pi 镜像之前，请完成以下要求：

- 下载并安装 [Raspberry Pi 成像器](#)。
- 获取 [SD 卡](#)。

- 设置[配备 2.4Ghz 64 位四核 CPU \( 8GB RAM \) 的 Raspberry Pi 5](#)。
- 连接 [Sonoff Zigbee USB 加密狗](#)。
- 将固件刷入 [Sonoff Zigbee USB 加密狗](#)。
- 连接 [Silicon Labs SLUSB001A 加密狗](#)。
- [注册一个 AWS 账户](#)。
- [AWS CLI 从《托管集成 AWS CLI 命令参考》中安装最新版本的](#)。

## 在新的 SD 卡上刷一张 Raspberry Pi 镜像

按照以下步骤将托管集成映像刷到您的 SD 卡：

1. 下载[托管集成 Raspberry Pi Hub SDK 图片](#)。
2. 在桌面上启动 Raspberry Pi Imager。
3. 将 SD 卡插入计算机的内置 SD 卡读卡器或外置 USB 读卡器。
4. 选择选择设备 → 树莓派 5。
5. 选择“选择操作系统” → “使用自定义” → “查找 lotMI-HubSDK-RPi-Image-v1.0.0.img.gz 文件” → “打开”。
6. 选择选择存储 → 选择您的 SD 读卡器。
7. 验证您的配置是否与以下屏幕相匹配：
8. 单击下一步。
9. 配置操作系统自定义设置：
  - 主机名：选择 r aspberrypi。
  - 用户名和密码：
    - 启用设置用户名和密码：
    - 在用户名:中，输入hub123456。
    - 在“密码:”中，输入sh123456。
  - 无线局域网：
    - 启用配置无线局域网。
    - 输入路由器的 SSID 和密码。

设置示例：

- SSID : `iotmi-tplink`
- 密码 : `*****` (至少 8 个字符)
- 将“国家:”设置为US。
- 设置区域设置 :
  - 将“时区:”设置为America/Los Angeles。
  - 将“键盘布局:”设置为US。
- SSH :
  - 选择服务选项卡。
  - 选中启用 SSH。
  - 选择使用密码身份验证。

10. 确认所有弹出窗口以进行操作系统自定义和数据删除。

11. 等待写入过程完成。

12. 使用以下屏幕验证成功完成 :

13. 单击继续。

14. 取出 SD 卡并将其插入 Raspberry Pi。

## 在树莓派上运行 Hub SDK

在你配置的 Raspberry Pi 上启动 Hub SDK 服务 :

1. 将准备好的 SD 卡插入树莓派 5 设备。
2. 将 Sonoff Zigbee USB 加密狗和 Silicon Labs SLUSB001A 加密狗连接到 Raspberry Pi。
3. 开启 Raspberry Pi。
4. 确保 Raspberry Pi 和你的计算机 ( 通过 SSH 从中获得 SSH ) 在同一个网络上。
5. 使用您在映像部署期间设置的凭据 SSH 到 Raspberry Pi。

```
ssh username@hostname
```

6. 导航到 hub SDK 目录 :

```
cd /data/aws/iotmi
```

7. 完成 [Hub 入门设置](#) 以添加身份验证和配置材料。

**Note**

您必须处于YUL或DUB所在地区才能执行此步骤。

**8. 运行 Hub SDK :**

```
cd /data/aws/iotmi
bash start_hub_sdk.sh
```

如果成功启动 Hub SDK , 系统会显示以下响应 :

```
-----Stopping SDK running processes---
-----Starting Hub SDK-----
-----Creating logs directory-----
Logs directory created.
-----Verifying Middleware paths-----
All middleware libraries exist
-----Verifying Middleware pre reqs---
AIPC and KVstroage directories exist
-----Starting HubOnboarding-----
-----Starting MQTT Proxy-----
-----Staring Log Daemon---
-----Starting Event Manager-----
-----Starting Zigbee Service-----
--Checking Zigbee network information--
-----Starting Zwave Service-----
/data/aws/iotmi/middleware/AceZwave/bin /data/aws/iotmi
/data/aws/iotmi
-----Starting CDMB-----
-----Starting Agent-----
-----Starting Provisioner-----
-----Checking SDK status-----
hub1234+ 1780 0.2 0.1 1093936 16368 pts/1 Sl+ 16:34 0:00 ./iotmi_mqtt_proxy -
C /data/aws/iotmi/config/iotmi_config.json
Process 'iotmi_mqtt_proxy' is running.
hub1234+ 1884 0.0 0.0 236272 2624 pts/1 Sl+ 16:34 0:00 ./middleware/
AceCommon/bin/ace_eventmgr
Process 'ace_eventmgr' is running.
hub1234+ 1892 9.1 0.1 393040 8352 pts/1 Sl+ 16:34 0:04 ./middleware/
AceZigbee/bin/ace_zigbee_service
Process 'ace_zigbee_service' is running.
```

```
hub1234+ 1923 0.0 0.1 1570736 12736 pts/1 Sl+ 16:34 0:00 ./zwave_svc
Process 'zwave_svc' is running.
hub1234+ 1958 0.0 0.0 1067632 5776 pts/1 Sl+ 16:34 0:00 ./iotmi_cdmb
Process 'iotmi_cdmb' is running.
hub1234+ 2001 0.2 0.2 2017712 21264 pts/1 Sl+ 16:35 0:00 ./iotmi_device_agent
Process 'iotmi_device_agent' is running.
hub1234+ 2045 0.0 0.1 1457824 12624 pts/1 Sl+ 16:35 0:00 ./
iotmi_lpw_provisioner
Process 'iotmi_lpw_provisioner' is running.
hub1234+ 1813 0.0 0.0 875152 6848 pts/1 Sl+ 16:34 0:00 ./iotmi_log_daemon
Process 'iotmi_log_daemon' is running.
-----Successfully Started Hub SDK-----
```

## 后续步骤

成功启动 Hub SDK 后，请通过继续进行设备加载和管理。[在用户指导下进行设备加载和操作的设置](#)

## 托管集成树莓派上的 Hub SDK Docker 容器

### Note

Raspberry Pi 上的 AWS IoT Hub SDK 的实现是一个演示项目，仅用于学习和测试目的，不打算在生产环境中使用。在本演示中，为了便于开发，请设置以下配置：

**AWS 凭证存储：**仅出于演示目的，凭证和证书存储在可访问的位置，便于测试和开发。生产环境必须使用安全的存储解决方案 AWS Secrets Manager，例如或 Systems Manager Parameter Store。他们必须实现静态加密，并遵循 AWS IoT 安全准则。

**容器权限：**该演示以提升的权限运行，允许不受限制地访问主机资源并简化开发工作流程。在生产环境中，容器应以最低要求的权限运行。

**网桥配置：**该演示使用网络桥接配置，该配置可公开内部网络流量，便于调试和监控。在生产环境中，实施适当的网络隔离和分段，以防止未经授权访问内部网络流量。

**USB 设备权限：**启用不受限制的 USB 设备访问权限，便于轻松连接开发外围设备和测试设备。在生产环境中，实施严格的 USB 设备控制和验证，以防止设备欺骗攻击。

这些配置支持直接测试，不得在生产环境中使用。部署到生产环境时，请遵循安全最佳实践，以防止主机系统受损和未经授权访问凭证。

## 先决条件

docker 容器需要满足以下先决条件。

- 下载并安装 [Raspberry Pi 成像器](#)。
- 获取 [S D 卡](#)。
- 设置 [配备 2.4Ghz 64 位四核 CPU \( 8GB RAM \) 的 Raspberry Pi 5](#)。
- 连接 [Sonoff Zigbee USB 加密狗](#)。
- [将固件刷入 Sonoff Zigbee USB 加密狗](#)。
- 连接 [Silicon Labs SLUSB001A 加密狗](#)。
- [注册一个 AWS 账户](#)。
- 安装 [托管集成 AWS CLI 命令参考 AWS CLI 中的](#) 最新版本的。
- 使用 IP 地址或主机名通过 SSH 访问树莓派。

## 在 Raspberry Pi 上使用托管集成中心 SDK Docker 容器

1. 下载 [管理集成 Raspberry Pi Hub SDK Docker](#)
2. 使用 SCP 将文件复制到 Raspberry Pi :

```
scp ~/path/to/IotMI-HubSDK-Docker-v1.0.0.tar.gz [username]@raspberrypi.local:~
```

3. 通过 SSH 连接到 Raspberry Pi :

```
ssh hub123456@raspberrypi.local
```

4. 如果没有，请安装 Docker :

```
Install Docker
cd
curl -fsSL https://get.docker.com | sudo sh

Add your user to docker group
sudo usermod -aG docker $USER
exit # exit ssh

Log in again
```

5. 如果没有，请安装 Docker Compose :

```
Install Docker Compose
sudo apt-get update
```





成功启动 Hub SDK 后，请通过继续进行设备加载和管理。[在用户指导下进行设备加载和操作的设置](#)

### Note

- 要访问 Docker 容器 bash 外壳，请运行以下命令：

```
docker compose exec hubsdk bash
```

- 要在重启后重新启动容器，请运行以下命令：

```
docker compose up -d
```

- 要更新 Hub SDK，请替换以下文件夹中的二进制文件：

```
hub-docker/iotmi
```

- 要在保留数据的同时安全地重启容器，请执行以下操作：

```
docker compose down
docker compose up -d
docker compose logs -f
```

## 托管集成演示应用程序

### Note

Raspberry Pi 上的 AWS IoT Hub SDK 的实现是一个演示项目，仅用于学习和测试目的，不打算在生产环境中使用。在本演示中，为了便于开发，请设置以下配置：

**AWS 凭证存储：**仅出于演示目的，凭证和证书存储在可访问的位置，便于测试和开发。生产环境必须使用安全的存储解决方案 AWS Secrets Manager，例如或 Systems Manager Parameter Store。他们必须实现静态加密，并遵循 AWS IoT 安全准则。

**容器权限：**该演示以提升的权限运行，允许不受限制地访问主机资源并简化开发工作流程。在生产环境中，容器应以最低要求的权限运行。

**网桥配置：**该演示使用网络桥接配置，该配置可公开内部网络流量，便于调试和监控。在生产环境中，实施适当的网络隔离和分段，以防止未经授权访问内部网络流量。

**USB 设备权限：**启用不受限制的 USB 设备访问权限，便于轻松连接开发外围设备和测试设备。在生产环境中，实施严格的 USB 设备控制和验证，以防止设备欺骗攻击。

这些配置支持直接测试，不得在生产环境中使用。部署到生产环境时，请遵循安全最佳实践，以防止主机系统受损和未经授权访问凭证。

该演示应用程序是一个基于 React 的演示应用程序，展示了用于智能家居设备管理的托管集成功能。该应用程序通过现代 Web 界面演示 Z-Wave 和 Zigbee 设备的设备启动、控制和监控。

## 先决条件

- [注册一个 AWS 账户](#)。
- [创建一个凭证柜](#)，然后将[凭证储物柜](#)添加到您的中心。
- 完成 [Hub 新手入门设置](#)。
- [Node.js 18+ 还有 npm](#)。
- [AWS CLI 从《托管集成 AWS CLI 命令参考》中安装最新版本的](#)。
- 现代网络浏览器 ( Chrome、火狐浏览器、Safari、Edge )

## 安装和配置应用程序

1. 下载[托管集成演示应用程序](#)。
2. 解压缩软件包：

```
cd ~/Downloads
tar -xzf IotMI-HubSDK-DemoApp-v1.0.0.tar.gz
cd IotManagedIntegrations-DemoApp
```

3. 安装依赖项：

```
npm install
```

4. 在根目录下创建一个 .env 文件：

```
AWS Configuration
REACT_APP_AWS_REGION=your_region
REACT_APP_AWS_ACCESS_KEY_ID=your_access_key
REACT_APP_AWS_SECRET_ACCESS_KEY=your_secret_key
REACT_APP_AWS_SESSION_TOKEN=your_session_token

IoT Managed Integrations Endpoint
```

```
REACT_APP_IOT_ENDPOINT=https://your-iot-endpoint.amazonaws.com

Hub Configuration
REACT_APP_HUB_MANAGED_THING_ID=your_hub_id
REACT_APP_CREDENTIAL_LOCKER_ID=your_credential_locker_id
```

5. 生成并启动应用程序：

```
npm start
```

6. 访问该应用程序，网址为：

```
http://localhost:3000
```

有关定价信息，请参阅[AWS IoT 设备管理定价页面上的托管集成部分](#)。

## 场外托管集成中心

### Hub SDK 板外流程概述

集线器离线过程会将集线器从 AWS Cloud 管理系统中移除。当云端发送 [DeleteManagedThing](#) 请求时，该过程可以实现两个主要目标：

设备端操作：

- 重置集线器的内部状态
- 删除所有本地保存的数据
- 为设备做好准备，以备将来可能重新上线

云端操作：

- 移除与中心关联的所有云资源
- 完全断开与先前账户的连接

客户通常在以下情况下启动集线器下线：

- 更改中心的关联账户
- 用新设备替换现有集线器

该过程可确保在集线器配置之间实现干净、安全的过渡，从而实现无缝的设备管理和帐户灵活性。

## 先决条件

- 你必须有一个已载入的集线器。有关说明，请参阅 [Hub 入门设置](#)。
- 在位于/data/aws/iotmi/config/iotmi\_config.json的文件中，验证是否iot\_provisioning\_state显示PROVISIONED。
- 确认中引用的永久证书和密钥iotmi\_config.json存在于其指定路径中。
- 确保代理 HubOnboarding、置备器和 MQTT 代理已正确配置并正在运行。
- 确认集线器没有子设备。在继续操作之前，请使用 [DeleteManagedThing](#) API 移除所有子设备。

## Hub SDK 场外流程

请按照以下步骤退出集线器：

### 检索 hub\_managed\_thing ID

该iotmi\_config.json文件用于存储托管集成中心的托管事物 ID。此标识符是允许集线器与 AWS IoT 托管集成服务通信的关键信息。托管事物 ID 存储在 JSON 文件的 rw（读写）部分的字段下managed\_thing\_id。在以下示例配置中可以看到这一点：

```
{
 "ro": {
 "iot_provisioning_method": "FLEET_PROVISIONING",
 "iot_claim_cert_path": "PATH",
 "iot_claim_pk_path": "PATH",
 "UPC": "UPC",
 "sh_endpoint_url": "ENDPOINT_URL",
 "SN": "SN",
 "fp_template_name": "TEMPLATENAME"
 },
 "rw": {
 "iot_provisioning_state": "PROVISIONED",
 "client_id": "ID",
 "managed_thing_id": "ID",
 "iot_permanent_cert_path": "CERT_PATH",
 "iot_permanent_pk_path": "KEY",
 "metadata": {
 "last_updated_epoch_time": 1747766125
 }
 }
}
```

```
 }
 }
}
```

## 向机外集线器发送命令

使用您的账户凭证，并使用上一节中managed\_thing\_id检索到的凭据运行命令：

```
aws iot-managed-integrations delete-managed-thing \
 --identifier HUB_MANAGED_THING_ID
```

## 验证集线器已下线

使用您的账户凭证，并使用上一节中managed\_thing\_id检索到的凭据运行命令：

```
aws iot-managed-integrations get-managed-thing \
 --identifier HUB_MANAGED_THING_ID
```

## 成功和失败场景

### 成功场景

如果成功执行了移出集线器的命令，则预计会出现以下示例响应：

```
{
 "Message" : "Managed Thing resource not found."
}
```

此外，如果集线器离线命令成功，则iotmi\_config.json会观察到以下示例。验证 rw 部分是否仅包含可选iot\_provisioning\_state的元数据。缺少元数据是可以接受的。iot\_provisioning\_state必须是 NOT\_PROVISIONED。

```
{
 "ro": {
 "iot_provisioning_method": "FLEET_PROVISIONING",
 "iot_claim_cert_path": "PATH",
 "iot_claim_pk_path": "PATH",
 "UPC": "1234567890101",
 "sh_endpoint_url": "ENDPOINT_URL",
 "SN": "1234567890101",
 "fp_template_name": "test-template" }}
```

```

 },
 "rw": {
 "iot_provisioning_state": "NOT_PROVISIONED",
 "metadata": {
 "last_updated_epoch_time": 1747766125
 }
 }
 }
}

```

## 失败场景

如果下线集线器的命令失败，则预计会出现以下示例响应：

```

{
 "Arn" : "ARN",
 "CreatedAt" : 1.748968266655E9,
 "Id" : "ID",
 "ProvisioningStatus" : "DELETE_IN_PROGRESS",
 "Role" : "CONTROLLER",
 "SerialNumber" : "SERIAL_NO",
 "Tags" : { },
 "UniversalProductCode" : "UPC",
 "UpdatedAt" : 1.748968272107E9
}

```

- 如果ProvisioningStatus是DELETE\_IN\_PROGRESS，请按照 [Hub 恢复](#) 中的说明进行操作。
- 如果不ProvisioningStatus是DELETE\_IN\_PROGRESS，则在托管集成云中关闭集线器的命令要么失败，要么未被托管集成云接收。按照 [Hub 恢复](#) 中的说明进行操作。
- 如果离线失败，则您的iotmi\_config.json文件将类似于下面的示例文件。

```

{
 "ro": {
 "iot_provisioning_method": "FLEET_PROVISIONING",
 "iot_claim_cert_path": "PATH",
 "iot_claim_pk_path": "PATH",
 "UPC": "123456789101",
 "sh_endpoint_url": "ENDPOINT_URL",
 "SN": "123456789101",
 "fp_template_name": "test-template"
 },
 "rw": {

```

```
 "iot_provisioning_state": "PROVISIONED",
 "client_id": "ID",
 "managed_thing_id": "ID",
 "iot_permanent_cert_path": "PATH",
 "iot_permanent_pk_path": "PATH",
 "metadata": {
 "last_updated_epoch_time": 1747766125
 }
 }
}
```

## ( 可选 ) 下线后 Hub SDK

### Important

以下场景列出了在离线 Hub SDK 失败后要采取的可选操作，或者您是否想在离线后重新加入集线器时要采取的操作。

## 重新登机

如果成功下线，请按照[第 3 步：创建托管事物（队列配置）](#)以及其余的板载流程进行加载 Hub SDK。

## 集线器恢复

设备中心成功下线和云端下线失败

如果 [GetManagedThing](#) API 调用未返回 Managed Thing resource not found 消息，但文件 `iotmi_config.json` 已被移除。有关示例 json 文件，请参阅[成功场景](#)。

要从这种情况中恢复，请参阅[强制删除](#)。

设备中心下线失败

这种情况是指文件 `iotmi_config.json` 未正确卸载。有关示例 json 文件，请参阅[失败场景](#)。

要从这种情况中恢复，请参阅[强制删除](#)。如果仍未脱机，`iotmi_config.json` 则必须将集线器恢复出厂设置。

设备中心离线和云端离线失败

在这种情况下，仍 `iotmi_config.json` 未脱机，集线器状态为 `ACTIVATED`、`DISCOVERED` 或 `DISCOVERED`。

要从这种情况中恢复，请参阅[强制删除](#)。如果强制删除失败或仍未脱机，`iotmi_config.json`则必须将集线器恢复出厂设置。

集线器处于离线状态且集线器状态为 `DELETE_IN_PROGRESS`

在这种情况下，集线器处于离线状态，云端收到离线命令。

要从这种情况中恢复，请参阅[强制删除](#)。

## 强制删除

要在设备中心未成功下线的情况下删除云资源，请按照以下步骤操作。此操作可能会导致云端和设备状态不一致，从而可能导致将来的操作出现问题。

使用集线器 `managed_thing_id` 和 `force` 参数调用 [DeleteManagedThing](#) API：

```
aws iot-managed-integrations delete-managed-thing \
 --identifier HUB_MANAGED_THING_ID \
 --force
```

接下来，调用 [GetManagedThing](#) API 并验证它是否返回 `Managed Thing resource not found`。这确认云资源已被删除。

### Note

不建议使用这种方法，因为它可能导致云和设备状态不一致。通常，在尝试删除云资源之前，最好确保设备中心成功下线。

## 特定于协议的中间件

### Important

此处提供的文档和代码描述了中间件的参考实现。它不是作为 SDK 的一部分提供给您。

特定于协议的中间件在与底层协议栈交互方面起着至关重要的作用。托管集成 Hub SDK 的设备入门和设备控制组件都使用它来与终端设备进行交互。

中间件执行以下功能。

- 通过提供一组通用的 APIs，从不同供应商的设备协议堆栈中抽出来。 APIs
- 提供软件执行管理，例如线程调度器、事件队列管理和数据缓存。

## 中间件架构

下面的方框图代表了 Zigbee 中间件的架构。其他协议（如 Z-Wave）的中间件的架构也类似。

特定于协议的中间件有三个主要组件。

- ACS Zigbee DPK : Zigbee 设备移植套件 (DPK) 用于提供对底层硬件和操作系统的抽象，从而实现可移植性。基本上，这可以被视为硬件抽象层 (HAL)，它提供了一组通用集 APIs 来控制来自不同供应商的 Zigbee 无线电并与之通信。Zigbee 中间件包含 Silicon Labs Zigbee 应用程序框架的 DPK API 实现。
- ACS Zigbee 服务 : Zigbee 服务作为专用守护程序运行。它包括一个 API 处理程序，通过 IPC 通道为来自客户端应用程序的 API 调用提供服务。AIPC 用作 Zigbee 适配器和 Zigbee 服务之间的 IPC 通道。它还提供其他功能，例如处理这两个 async/sync 命令、处理来自 HAL 的事件以及使用 ACS 事件管理器进行事件注册/发布。
- ACS Zigbee 适配器 : Zigbee 适配器是在应用程序进程中运行的库（在本例中，应用程序是 CDMB 插件）。Zigbee 适配器提供了一组供客户端应用程序（例如 CDMB/Provisioner 协议插件）使用，用于控制终端设备并与之通信。 APIs

## End-to-end 中间件命令流示例

以下是通过 ZigBee 中间件的命令流示例。

以下是通过 Z-Wave 中间件执行命令流的示例。

## 特定于协议的中间件代码组织

本节包含有关 IotManagedIntegrationsDeviceSDK-Middleware 存储库中每个组件的代码位置的信息。以下是此存储库中文件夹结构的示例。

```
./IotManagedIntegrationsDeviceSDK-Middleware
|- greengrass
|- example-iot-ace-dpk
```

```
|– example-iot-ace-general
|– example-iot-ace-project
|– example-iot-ace-z3-gateway
|– example-iot-ace-zware
|– example-iot-ace-zwave-mw
```

## 主题

- [Zigbee 中间件代码组织](#)
- [Z-Wave 中间件代码组织](#)

## Zigbee 中间件代码组织

以下显示了 ZigBee 参考中间件代码组织。

## 主题

- [ACS Zigbee DPK](#)
- [硅实验室 Zigbee SDK](#)
- [ACS Zigbee 服务](#)
- [ACS Zigbee 适配器](#)

## ACS Zigbee DPK

Zigbee DPK 的代码位于以下示例中列出的目录中：

```
./IotManagedIntegrationsDeviceSDK-Middleware/example-iot-ace-dpk/example/dpk/ace_hal/
|– common
|– |– fxnDbusClient
|– |– include
|– kvs
|– log
|– wifi
|– |– include
|– |– src
|– |– wifid
|– |– fxnWifiClient
|– |– include
|– zibgee
|– |– include
```

```

|- |- src
|- |- zigbeed
|- |- ember
|- |- include
|- zwave
|- |- include
|- |- src
|- |- zwaved
|- |- fxnZwaveClient
|- |- include
|- |- zware

```

## 硅实验室 Zigbee SDK

Silicon Labs SDK 显示在 IotManagedIntegrationsDeviceSDK-Middleware/*example*-iot-ace-z3-gateway 文件夹中。这个 ACS Zigbee DPK 层是为这个 Silicon Labs SDK 实现的。

```

./IotManagedIntegrationsDeviceSDK-Middleware/example-iot-ace-zz3-gateway/
|- autogen
|- config
|- gecko_sdk_4.3.2
|- |- platform
|- |- protocol
|- |- util

```

## ACS Zigbee 服务

ZigBee 服务的代码位于该文件夹内。IotManagedIntegrationsDeviceSDK-Middleware/*example*-iot-ace-general/middleware/zigbee/ 此位置的 src 和 include 子文件夹包含与 ACS Zigbee 服务相关的所有文件。

```

IotManagedIntegrationsDeviceSDK-Middleware/example-iot-ace-general/middleware/zigbee/
src/
|- zb_alloc.c
|- zb_callbacks.c
|- zb_database.c
|- zb_discovery.c
|- zb_log.c
|- zb_main.c
|- zb_region_info.c
|- zb_server.c
|- zb_svc.c

```

```
|– zb_svc_pwr.c
|– zb_timer.c
|– zb_util.c
|– zb_zdo.c
|– zb_zts.c
IotManagedIntegrationsDeviceSDK-Middleware/example-iot-ace-general/middleware/zigbee/
include/
|– init.zigbeeservice.rc
|– zb_ace_log_uhl.h
|– zb_alloc.h
|– zb_callbacks.h
|– zb_client_aipc.h
|– zb_client_event_handler.h
|– zb_database.h
|– zb_discovery.h
|– zb_log.h
|– zb_region_info.h
|– zb_server.h
|– zb_svc.h
|– zb_svc_pwr.h
|– zb_timer.h
|– zb_util.h
|– zb_zdo.h
|– zb_zts.h
```

## ACS Zigbee 适配器

ACS Zigbee 适配器的代码位于文件夹内。IotManagedIntegrationsDeviceSDK-Middleware/*example*-iot-ace-general/middleware/zigbee/api此位置的src和include子文件夹包含与 ACS Zigbee Adaptor 库相关的所有文件。

```
IotManagedIntegrationsDeviceSDK-Middleware/example-iot-ace-general/middleware/zigbee/
api/src/
|– zb_client_aipc.c
|– zb_client_api.c
|– zb_client_event_handler.c
|– zb_client_zcl.c
IotManagedIntegrationsDeviceSDK-Middleware/example-iot-ace-general/middleware/zigbee/
api/include/
|– ace
|– |– zb_adapter.h
|– |– zb_command.h
|– |– zb_network.h
```

```
|- |- zb_types.h
|- |- zb_zcl.h
|- |- zb_zcl_cmd.h
|- |- zb_zcl_color_control.h
|- |- zb_zcl_hvac.h
|- |- zb_zcl_id.h
|- |- zb_zcl_identify.h
|- |- zb_zcl_level.h
|- |- zb_zcl_measure_and_sensing.h
|- |- zb_zcl_onoff.h
|- |- zb_zcl_power.h
```

## Z-Wave 中间件代码组织

下图显示了 Z-Wave 参考中间件代码组织。

### 主题

- [ACS Z-Wave DPK](#)
- [芯科实验室 ZWare 和 Zip 网关](#)
- [ACS Z-Wave 服务](#)
- [ACS Z-Wave 适配器](#)

### ACS Z-Wave DPK

Z-Wave DPK 的代码位于该文件夹内。IotManagedIntegrationsDeviceSDK-Middleware/*example*-iot-ace-dpk/*example*/dpk/ace\_hal/zwave

```
./IotManagedIntegrationsDeviceSDK-Middleware/example-iot-ace-dpk/example/dpk/ace_hal/
|- common
|- |- fxnDBusClient
|- |- include
|- kvs
|- log
|- wifi
|- |- include
|- |- src
|- |- wifid
|- |- fxnWifiClient
|- |- include
|- zibgee
```

```
|- |- include
|- |- src
|- |- zigbeed
|- |- ember
|- |- include
|- zwave
|- |- include
|- |- src
|- |- zwaved
|- |- fxnZwaveClient
|- |- include
|- |- zware
```

## 芯科实验室 ZWave 和 Zip 网关

Silicon Labs ZWave 和 Zip Gateway 的代码位于 IotManagedIntegrationsDeviceSDK-Middleware/*example*-iot-ace-z3-gateway 文件夹内。这个 ACS Z-Wave DPK 层是为 Z-Wave C APIs 和 Zip 网关实现的。

```
./IotManagedIntegrationsDeviceSDK-Middleware/example-iot-ace-z3-gateway/
|- autogen
|- config
|- gecko_sdk_4.3.2
|- |- platform
|- |- protocol
|- |- util
```

## ACS Z-Wave 服务

Z-Wave 服务的代码位于该文件夹中列出的文件夹内。IotManagedIntegrationsMiddlewares/*example*iot-ace-zwave-mw/此位置的 src 和 include 文件夹包含与 ACS Z-Wave 服务相关的所有文件。

```
IotManagedIntegrationsDeviceSDK-Middleware/example-iot-ace-zwave-mw/src/
|- zwave_mgr.c
|- zwave_mgr_cc.c
|- zwave_mgr_ipc_aipc.c
|- zwave_svc.c
|- zwave_svc_dispatcher.c
|- zwave_svc_hsm.c
|- zwave_svc_ipc_aipc.c
|- zwave_svc_main.c
```

```
|– zwave_svc_publish.c
IotManagedIntegrationsDeviceSDK-Middleware/example-iot-ace-zwave-mw/include/
|– ace
|– |– zwave_common_cc.h
|– |– zwave_common_cc_battery.h
|– |– zwave_common_cc_doorlock.h
|– |– zwave_common_cc_firmware.h
|– |– zwave_common_cc_meter.h
|– |– zwave_common_cc_notification.h
|– |– zwave_common_cc_sensor.h
|– |– zwave_common_cc_switch.h
|– |– zwave_common_cc_thermostat.h
|– |– zwave_common_cc_version.h
|– |– zwave_common_types.h
|– |– zwave_mgr.h
|– |– zwave_mgr_cc.h
|– zwave_log.h
|– zwave_mgr_internal.h
|– zwave_mgr_ipc.h
|– zwave_svc_hsm.h
|– zwave_svc_internal.h
|– zwave_utils.h
```

## ACS Z-Wave 适配器

ACS Zigbee 适配器的代码位于文件夹内。IotManagedIntegrationsDeviceSDK-Middleware/*example*-iot-ace-zwave-mw/cli/此位置的src和include文件夹包含与 ACS Z-Wave Adaptor 库相关的所有文件。

```
IotManagedIntegrationsDeviceSDK-Middleware/example-iot-ace-zwave-mw/cli/
|– include
|– |– zwave_cli.h
|– src
|– |– zwave_cli.yaml
|– |– zwave_cli_cc.c
|– |– zwave_cli_event_monitor.c
|– |– zwave_cli_main.c
|– |– zwave_cli_net.c
```

## 将中间件与 SDK 集成

以下各节将讨论新集线器上的中间件集成。

## 主题

- [设备移植套件 \(DPK\) API 集成](#)
- [参考实现和代码组织](#)

## 设备移植套件 (DPK) API 集成

为了将任何芯片组供应商的 SDK 与中间件集成，中间的 DPK (设备移植套件) 层提供了标准的 API 接口。托管集成服务提供商或 ODMs 需要 APIs 根据其物联网中心上使用的 Zigbee/Z-wave/Wi Wi-Fi 芯片组支持的供应商 SDK 来实现这些服务。

## 参考实现和代码组织

除中间件外，所有其他设备 SDK 组件，例如托管集成 Device Agent 和通用数据模型桥 (CDBM)，无需任何修改即可使用，只需要交叉编译即可。

中间件的实现基于适用于 Zigbee 和 Z-Wave 的 Silicon Labs SDK。如果中间件中的 Silicon Labs SDK 支持新集线器中使用的 Z-Wave 和 Zigbee 芯片组，则无需任何修改即可使用参考中间件。你只需要交叉编译中间件，然后它就可以在新的集线器上运行。

Zigbee 的 DPK (设备移植套件) APIs 可以在中找到 `acehal_zigbee.c`，DPK 的参考实现 APIs 位于该文件夹中。zigbee

```
IotManagedIntegrationsDeviceSDK-Middleware/example-iot-ace-dpk/example/dpk/ace_hal/
zigbee/
|- CMakeLists.txt
|- include
|- |- zigbee_log.h
|- src
|- |- acehal_zigbee.c
|- zigbeed
|- |- CMakeLists.txt
|- |- ember
|- |- ace_ember_common.c
|- |- ace_ember_ctrl.c
|- |- ace_ember_hal_callbacks.c
|- |- ace_ember_network_creator.c
|- |- ace_ember_power_settings.c
|- |- ace_ember_zts.c
|- |- include
|- |- |- zbd_api.h
|- |- |- zbd_callbacks.h
```

```

|- |- |- zbd_common.h
|- |- |- zbd_network_creator.h
|- |- |- zbd_power_settings.h
|- |- |- zbd_zts.h

```

Z-Wave APIs 的 DPK 可以在中找到 `acehal_zwave.c`，文件夹中有 DPK APIs 的参考实现。zwaved

```

IotManagedIntegrationsDeviceSDK-Middleware/example-iot-ace-dpk/example/dpk/ace_hal/
zwave/
|- CMakeLists.txt
|- include
|- |- zwave_log.h
|- src
|- |- acehal_zwave.c
|- zwaved
|- |- CMakeLists.txt
|- |- fxnZwaveClient
|- |- |- zwave_client.c
|- |- |- zwave_client.h
|- |- include
|- |- |- zwaved_cc_intf_api.h
|- |- |- zwaved_common_utils.h
|- |- |- zwaved_ctrl_api.h
|- |- zware
|- |- |- ace_zware_cc_intf.c
|- |- |- ace_zware_common_utils.c
|- |- |- ace_zware_ctrl.c
|- |- |- ace_zware_debug.c
|- |- |- ace_zware_debug.h
|- |- |- ace_zware_internal.h

```

作为为不同供应商 SDK 实现 DPK 层的起点，可以使用和修改参考实现。要支持不同的供应商 SDK，需要进行以下两项修改：

1. 将当前供应商 SDK 替换为存储库中的新供应商 SDK。
2. APIs 根据新的供应商 SDK 实现中间件 DPK（设备移植套件）。

# 托管集成终端设备 SDK

构建一个物联网平台，将智能设备连接到托管集成，并通过统一的控制界面处理命令。终端设备 SDK 可与您的设备固件集成，并通过 SDK 边缘组件提供简化的设置，AWS IoT Core 以及与 AWS IoT 设备管理的安全连接。从下载最新版本的终端设备 SDK AWS 管理控制台

本指南介绍如何在固件中实现终端设备 SDK。查看架构、组件和集成步骤，开始构建您的实现。

## 主题

- [什么是终端设备 SDK？](#)
- [终端设备 SDK 架构和组件](#)
- [预备人](#)
- [Over-the-Air 更新](#)
- [数据模型代码生成器](#)
- [低级 C 函数 APIs](#)
- [托管集成中的功能和设备交互](#)
- [开始使用终端设备 SDK](#)

## 什么是终端设备 SDK？

### 什么是终端设备 SDK？

终端设备 SDK 是由提供的源代码、库和工具的集合 AWS IoT。该软件开发工具包专为资源有限的环境而构建，支持内存低至 512 KB 和 4 MB 闪存的设备，例如在嵌入式 Linux 和实时操作系统 (RTOS) 上运行的摄像头和空气净化器。从[AWS IoT 管理控制台](#)下载最新版本的终端设备 SDK。

### 核心组件

SDK 结合了用于云通信的 MQTT 代理、用于任务管理的作业处理程序和托管集成（数据模型处理器）。这些组件协同工作，可在您的设备和托管集成之间提供安全的连接和自动数据转换。

有关详细的技术要求，请参阅[技术参考](#)。

## 终端设备 SDK 架构和组件

本节介绍终端设备 SDK 架构及其组件如何与低级 C 函数交互。下图说明了 SDK 框架中的核心组件及其关系。

## 终端设备 SDK 组件

终端设备 SDK 架构包含以下用于托管集成功能集成的组件：

### 预备人

在托管集成云中创建设备资源，包括用于安全 MQTT 通信的设备证书和私钥。这些凭证可在您的设备与托管集成之间建立可信连接。

### MQTT 代理

通过线程安全 C 客户端库管理 MQTT 连接。此后台进程处理多线程环境中的命令队列，可为内存受限的设备配置队列大小。消息通过托管集成进行处理。

### 作业处理者

处理设备固件、安全补丁和文件传送的 over-the-air (OTA) 更新。此内置服务管理所有已注册设备的软件更新。

### 数据模型处理器

使用 AWS“物质数据模型”的实现，在托管集成和低级 C 函数之间转换操作。有关更多信息，请参阅上的 [Matter 文档GitHub](#)。

### 密钥和证书

[通过 PKCS #11 API 管理加密操作，同时支持硬件安全模块和核心等软件实现。](#) PKCS11 此 API 在 TLS 连接期间处理 Provisionee 和 MQTT 代理等组件的证书操作。

## 预备人

provisionee 是托管集成的组成部分，支持按声明进行队列配置。使用配置者，您可以安全地配置您的设备。SDK 为设备配置创建了必要的资源，其中包括从托管集成云中获取的设备证书和私钥。当您想要配置设备时，或者如果有任何更改可能需要您重新配置设备，则可以使用预备者。

### 主题

- [置备人工作流程](#)
- [设置环境变量](#)
- [注册自定义终端节点](#)
- [创建配置文件](#)

- [创建托管事物](#)
- [SDK 用户 Wi-Fi 配置](#)
- [按索赔提供舰队](#)
- [托管式事物功能](#)

## 置备人工作流程

该过程需要在云端和设备端进行设置。客户配置云需求，例如自定义端点、配置文件和托管事物。设备首次开机时，供应者：

1. 使用声明证书连接到托管集成端点
2. 通过队列配置挂钩验证设备参数
3. 在设备上获取并存储永久证书和私钥
4. 设备使用永久证书重新连接
5. 发现设备功能并将其上传到托管集成

成功配置后，设备将直接与托管集成进行通信。预配者仅在重新配置任务时激活。

## 设置环境变量

在您的云环境中设置以下 AWS 凭据：

```
$ export AWS_ACCESS_KEY_ID=YOUR-ACCOUNT-ACCESS-KEY-ID
$ export AWS_SECRET_ACCESS_KEY=YOUR-ACCOUNT-SECRET-ACCESS-KEY
$ export AWS_DEFAULT_REGION=YOUR-DEFAULT-REGION
```

## 注册自定义终端节点

在您的云环境中使用 [RegisterCustomEndpoint](#) API 命令创建用于 device-to-cloud 通信的自定义终端节点。

```
aws iot-managed-integrations register-custom-endpoint
```

### 响应示例

```
{ "EndpointAddress": "[ACCOUNT-PREFIX]-ats.iot.AWS-REGION.amazonaws.com" }
```

**Note**

存储用于配置配置参数的端点地址。使用 `GetCustomEndpoint` API 返回端点信息。有关更多信息，请参阅 [GetCustomEndpoint](#) 《托管集成 [RegisterCustomEndpoint](#) API 参考指南》中的 API 和 API。

## 创建配置文件

创建用于定义您的队列配置方法的配置文件。在您的云环境中运行 [CreateProvisioningProfile](#) API 以返回用于设备身份验证的声明证书和私钥：

```
aws iot-managed-integrations create-provisioning-profile \
--provisioning-type "FLEET_PROVISIONING" \
--name "PROVISIONING-PROFILE-NAME"
```

### 响应示例

```
{ "Arn": "arn:aws:iot-managed-integrations:AWS-REGION:YOUR-ACCOUNT-ID:provisioning-
profile/PROFILE_NAME",
 "ClaimCertificate": "string",
 "ClaimCertificatePrivateKey": "string",
 "Name": "ProfileName",
 "ProvisioningType": "FLEET_PROVISIONING" }
```

您可以实现核心 PKCS11 平台抽象库 (PAL)，使核心 PKCS11 库与您的设备配合使用。核心 PKCS11 PAL 端口必须提供存储索赔证书和私钥的位置。使用此功能，您可以安全地存储设备的私钥和证书。您可以将私钥和证书存储在硬件安全模块 (HSM) 或可信平台模块 (TPM) 上。

## 创建托管事物

使用 [CreateManagedThing](#) API 将您的设备注册到托管集成云。包括设备的序列号 (SN) 和通用产品代码 (UPC)：

```
aws iot-managed-integrations create-managed-thing --role DEVICE \
--authentication-material-type WIFI_SETUP_QR_BAR_CODE \
--authentication-material "SN:DEVICE-SN;UPC:DEVICE-UPC;"
```

以下显示了 API 响应示例。

```
{
 "Arn": "arn:aws:iot-managed-integrations:AWS-REGION:ACCOUNT-ID:managed-
thing/59d3c90c55c4491192d841879192d33f",
 "CreatedAt": 1.730960226491E9,
 "Id": "59d3c90c55c4491192d841879192d33f"
}
```

API 返回可用于配置验证的托管事物 ID。您需要提供设备序列号 (SN) 和通用产品代码 (UPC)，这些序列号和通用产品代码 (UPC) 在置备交易期间与批准的托管事物相匹配。该交易返回的结果类似于以下内容：

```
/**
 * @brief Device info structure.
 */
typedef struct iotmiDev_DeviceInfo
{
 char serialNumber[IOTMI_DEVICE_MAX_SERIAL_NUMBER_LENGTH + 1U];
 char universalProductCode[IOTMI_DEVICE_MAX_UPC_LENGTH + 1U];
 char internationalArticleNumber[IOTMI_DEVICE_MAX_EAN_LENGTH + 1U];
} iotmiDev_DeviceInfo_t;
```

## SDK 用户 Wi-Fi 配置

设备制造商和解决方案提供商拥有自己的专有 Wi-Fi 配置服务，用于接收和配置 Wi-Fi 凭证。Wi-Fi 配置服务包括使用专用的移动应用程序、低功耗蓝牙 (BLE) 连接和其他专有协议，在初始设置过程中安全地传输 Wi-Fi 凭证。

终端设备 SDK 的使用者必须实现 Wi-Fi 配置服务，设备才能连接到 Wi-Fi 网络。

## 按索赔提供舰队

使用 provisionee，最终用户可以配置唯一的证书，并使用按声明配置将其注册到托管集成。

客户端 ID 可以从配置模板响应或设备证书中获取 <common name>“\_”<serial number>

## 托管式事物功能

预配者发现托管事物功能，然后将这些功能上传到托管集成。它使应用程序和其他服务可以使用这些功能进行访问。设备、其他 Web 客户端和服务可以使用 MQTT 和保留的 MQTT 主题更新功能，也可以使用 REST API 使用 HTTP 来更新功能。

# Over-the-Air 更新

## OTA 架构概述

Over-the-Air ( OTA ) 更新过程涉及多个组件协同工作，为您的设备提供固件更新。下图说明了如何通过终端设备 SDK、Hub SDK 和该功能之间的交互来处理 OTA 更新请求。

OTA 更新架构由以下组件组成：

- 客户：将任务文档上传到 S3 存储桶并通过 API 启动更新
- OTA 服务：处理任务创建、验证和管理
- AWS IoT 作业：管理任务执行和向设备传送
- 设备：使用 Harmony SDK 接收和应用更新

## 先决条件

在创建 OTA 任务之前，必须配置以下先决条件：

### 配置 Amazon S3 访问权限

要启用 OTA 更新，您必须将任务文档上传到 Amazon S3 存储桶并配置相应的访问权限：

1. 将您的 OTA 任务文档上传到 S3 存储桶
2. 添加 Amazon S3 存储桶策略，授予托管集成访问您的任务文档的权限：

### JSON

```
{
 "Version": "2012-10-17",
 "Statement": [
 {
 "Sid": "PolicyForS3JobDocument",
 "Effect": "Allow",
 "Principal": {
 "Service": "iotmanagedintegrations.amazonaws.com"
 },
 "Action": "s3:GetObject",
```

```
"Resource": [
 "arn:aws:s3:::YOUR_BUCKET/*",
 "arn:aws:s3:::YOUR_BUCKET/ota_job_document.json",
 "arn:aws:s3:::YOUR_BUCKET"
]
}
]
}
```

## 实施 Over-the-Air (OTA) 任务

您可以通过两种方式创建 OTA 任务，具体取决于您的更新要求和设备定位策略：

### 一次性 OTA 任务更新

一次性 OTA 任务包含用于执行 OTA 更新的静态目标列表 (ManagedThings)。您一次最多可以添加 100 个目标。该工作流程使用带队列索引的 AWS IoT 作业，同时维护托管集成抽象层。

使用以下示例创建一次性 OTA 任务：

```
aws iotmanagedintegrations create-ota-task \
 --description "One-time OTA update" \
 --s3-url "s3://test-job-document-bucket/ota-job-document.json" \
 --protocol HTTP \
 --target ["arn:aws:iotmanagedintegrations:region:account id:managed-thing/managed
thing id"] \
 --ota-mechanism PUSH \
 --ota-type ONE_TIME \
 --client-token "foo" \
 --tags '{"key1":"foo","key2":"foo"}'
```

### 持续更新 OTA 任务

OTA ( Over-the-Air ) 分组工作流程使您可以根据特定属性将固件更新部署到设备组，使用带有队列索引的 AWS IoT 作业，同时维护托管集成抽象层。持续的 OTA 任务使用查询字符串而不是特定目标。所有符合查询条件的设备都会进行 OTA 更新，并且会不断重新评估查询条件。匹配的目标将有工作部署。

#### 配置先决条件

在创建连续的 OTA 任务之前，请完成以下先决条件：

1. 通过调用 [CreateManagedThing](#) API 创建托管事物并执行队列配置。
2. 在您的托管内容中添加元数据属性以进行查询定位。

ManagedThing 使用 [UpdateManagedThing](#) API 向其中添加属性和元数据：

```
aws iotmanagedintegrations update-managed-thing \
 --managed-thing-id "YOUR_MANAGED_THING_ID" \
 --meta-data '{"owner":"managedintegrations","version":"1.0"}'
```

使用以下示例创建连续的 OTA 任务：

```
aws iotmanagedintegrations create-ota-task \
 --description "Continuous OTA update" \
 --s3-url "s3://test-job-document-bucket/ota-job-document.json" \
 --protocol HTTP \
 --ota-mechanism PUSH \
 --ota-type CONTINUOUS \
 --client-token "foo" \
 --ota-target-query-string "attributes.owner=managedintegrations" \
 --tags '{"key1":"foo","key2":"foo"}'
```

## 了解持续的 OTA 工作流程

持续 OTA 更新工作流程遵循以下步骤：

1. 您可以使用 [UpdateManagedThing](#) API 更新带有属性的托管事物。
2. 使用针对特定设备属性的查询字符串创建 OTA 作业。
3. OTA 服务 AWS IoT Core 根据查询属性在中创建动态事物组
4. IoT 任务在匹配的设备上执行更新
5. 您可以通过 [ListOtaTaskExecutions](#) API 监控进度，也可以通过 Kinesis 直播（如果已启用）通过 OTA 通知监控进度。

## 托管集成 OTA 和物联网作业之间的区别

托管集成 OTA 和 IoT Jobs 之间的根本区别在于服务编排和自动化。托管集成 OTA 提供了一种单一服务解决方案，可消除多服务协调的复杂性。

托管集成 OTA 自动执行的操作：

- 动态事物组创建：根据您的查询条件自动生成 AWS IoT Core 事物组。
- 目标分辨率：将查询字符串（示例：attributes.owner=managedintegrations）转换为实际的设备目标。
- 服务集成：在物联网任务和舰队索引服务之间 AWS IoT Core 进行无缝协调。
- 生命周期管理：处理从创建到执行监控的整个 OTA 工作流程。

MI OTA 消除了什么：

- 在中创建事物组 AWS IoT Core。
- 向群组添加内容。
- 创建物联网任务。

托管集成 OTA 会根据您的查询字符串在内部处理所有三项操作，自动发现符合您条件的设备，在幕后创建 IoT 任务，并协调完整的 OTA 工作流程，而无需您直接与多项 AWS 服务交互。

## OTA 任务配置设置

您可以为 OTA 更新创建配置，以控制向设备推出更新的方式、设置中止条件和配置超时。

例如：CreateOtaTaskConfiguration

使用以下示例创建 OTA 任务配置：

```
aws iotmanagedintegrations create-ota-task-configuration \
 --description "OTA configuration" \
 --name "MyOtaConfig" \
 --push-config '{
 "AbortConfig": {
 "AbortConfigCriteriaList": [
 {
 "Action": "CANCEL",
 "FailureType": "FAILED",
 "MinNumberOfExecutedThings": 1,
 "ThresholdPercentage": 90.0
 }
]
 },
 "RolloutConfig": {
```

```

 "ExponentialRolloutRate": {
 "BaseRatePerMinute": 1,
 "IncrementFactor": 3.0,
 "RateIncreaseCriteria": {
 "numberOfNotifiedThings": 1
 }
 },
 "MaximumPerMinute": 1
 },
 "TimeoutConfig": {
 "InProgressTimeoutInMinutes": 100
 }
}' \
--client-token "foo"

```

## 将配置设置应用于 OTA 任务

创建配置后，您将收到添加到CreateOtaTask请求中的配置以及其他配置：`taskConfigurationId`

```

aws iotmanagedintegrations create-ota-task \
 --description "OTA with configuration" \
 --s3-url "s3://test-job-document-bucket/ota-job-document.json" \
 --protocol HTTP \
 --target ["arn:aws:iotmanagedintegrations:region:account id:managed-thing/managed thing id"] \
 --ota-mechanism PUSH \
 --ota-type ONE_TIME \
 --client-token "foo" \
 --task-configuration-id "ae4f49352c5443369f43ad6c3a7f1580" \
 --ota-scheduling-config '{
 "EndBehavior": "STOP_ROLLOUT",
 "EndTime": "2024-10-23T17:00",
 "StartTime": "2024-10-20T17:00"
 }' \
 --ota-task-execution-retry-config '{
 "RetryConfigCriteria": [
 {
 "FailureType": "FAILED",
 "MinNumberOfRetries": 1
 }
]
 }' \

```

```
--tags '{"key1":"foo","key2":"foo"}'
```

## 监控 OTA 通知

您可以使用两种不同的方法监控 OTA 更新：

### 通过 Kinesis Data Streams 推送通知

启用 OTA 通知后，更新状态事件会自动推送到您的 Kinesis 直播中。这样可以实时查看跨设备的固件更新进度。

### 使用 ListOtaTaskExecutions API 进行监控

您可以使用 [ListOtaTaskExecutions](#) API 手动检查托管事物的 OTA 更新状态：

```
aws iotmanagedintegrations list-ota-task-executions \
 --task-id "task-123456789" \
 --max-results 25
```

该响应提供了每个托管事物的详细执行状态：

```
{
 "taskExecutionSummaries": [
 {
 "taskExecutionSummary": {
 "executionNumber": 1,
 "lastUpdatedAt": 1634567890,
 "queuedAt": 1634567800,
 "startedAt": 1634567830,
 "status": "SUCCEEDED",
 "retryAttempt": 0
 },
 "managedThingId": "device-001"
 },
 {
 "taskExecutionSummary": {
 "executionNumber": 1,
 "lastUpdatedAt": 1634567920,
 "queuedAt": 1634567800,
 "startedAt": 1634567840,
 "status": "IN_PROGRESS",
```

```
 "retryAttempt": 0
 },
 "managedThingId": "device-002"
}
],
"nextToken": "NEXT_TOKEN"
}
```

此 API 允许您检索特定 OTA 任务所针对的每个托管事物的详细执行状态，包括时间戳和当前状态。

## 处理工作文档

创建 OTA 任务时，任务处理程序会在您的设备上运行以下步骤。当有更新可用时，它会通过 MQTT 请求任务文档。

1. 订阅 MQTT 通知主题。
2. 为待处理的任务调用 [StartNextPendingJobExecution](#) API。
3. 接收可用的工作文件。
4. 根据您的指定的超时时间处理更新。

使用作业处理程序，应用程序可以决定是立即采取行动还是等到指定的超时时间。

## 实施 OTA 代理

当您收到来自托管集成的任务文档时，必须实现自己的 OTA 代理，该代理可以处理任务文档、下载更新和执行任何安装操作。OTA 代理需要执行以下步骤：

1. 解析固件 Amazon S3 URLs 的任务文档。
2. 通过 HTTP 下载固件更新。
3. 验证数字签名。
4. 安装经过验证的更新。
5. `iotmi\_JobsHandler\_updateJobStatus` 使用 `SUCCESS` 或 `FAILED` 状态拨打电话。

当您的设备成功完成 OTA 操作后，它必须调用状态为的 `iotmi\_JobsHandler\_updateJobStatus` API `JobSucceeded` 才能报告成功的作业。

```
/**
```

```
* @brief Enumeration of possible job statuses.
*/
typedef enum{
 JobQueued, /** The job is in the queue, waiting to be processed. */
 JobInProgress, /** The job is currently being processed. */
 JobFailed, /** The job processing failed. */
 JobSucceeded, /** The job processing succeeded. */
 JobRejected /** The job was rejected, possibly due to an error or invalid
request. */
} iotmi_JobCurrentStatus_t;

/**
 * @brief Update the status of a job with optional status details.
 *
 * @param[in] pJobId Pointer to the job ID string.
 * @param[in] jobIdLength Length of the job ID string.
 * @param[in] status The new status of the job.
 * @param[in] statusDetails Pointer to a string containing additional details about the
job status.
 *
 * This can be a JSON-formatted string or NULL if no details
are needed.
 * @param[in] statusDetailsLength Length of the status details string. Set to 0 if
`statusDetails` is NULL.
 *
 * @return 0 on success, non-zero on failure.
 */
int iotmi_JobsHandler_updateJobStatus(const char * pJobId,
 size_t jobIdLength,
 iotmi_JobCurrentStatus_t status,
 const char * statusDetails,
 size_t statusDetailsLength);
```

## 数据模型代码生成器

学习如何使用数据模型的代码生成器。生成的代码可用于序列化和反序列化在云端和设备之间交换的数据模型。

项目存储库包含用于创建 C 代码数据模型处理程序的代码生成工具。以下主题描述了代码生成器和工作流程。

### 主题

- [代码生成过程](#)

- [环境设置](#)
- [为设备生成代码](#)

## 代码生成过程

代码生成器根据三个主要输入创建 C 源文件：AWS“来自 Zigbee 集群库 (ZCL) 高级平台的物质数据模型 (.matter 文件) 的实现、处理预处理的 Python 插件和定义代码结构的 Jinja2 模板。在生成过程中，Python 插件通过添加全局类型定义、根据数据类型的依赖关系组织数据类型以及格式化模板渲染信息来处理您的 .matter 文件。

下图描述了创建 C 源文件的代码生成器。

终端设备 SDK 包括可在项目 [codegen.py](#) 中使用的 Python 插件和 Jinja2 模板。[connectedhomeip](#) 这种组合会根据您的 .matter 文件输入为每个集群生成多个 C 文件。

以下子主题描述了这些文件。

- [Python 插件](#)
- [Jinja2 模板](#)
- [\( 可选 \) 自定义架构](#)

## Python 插件

代码生成器解析 .matter 文件，并将这些信息作为 Python 对象发送到插件。codegen.py 插件文件会对这些数据 `iotmi_data_model.py` 进行预处理，并使用提供的模板呈现源文件。预处理包括：

1. 添加不可用的信息 `codegen.py`，例如全局类型
2. 对数据类型执行拓扑排序以建立正确的定义顺序

### Note

拓扑排序可确保依赖类型在依赖关系之后定义，无论其原始顺序如何。

## Jinja2 模板

终端设备 SDK 提供专为数据模型处理程序和低级 C 函数量身定制的 Jinja2 模板。

## Jinja2 模板

| 模板                                                  | 生成的来源                                                    | 备注                                  |
|-----------------------------------------------------|----------------------------------------------------------|-------------------------------------|
| <code>cluster.h.jinja</code>                        | <code>iotmi_device_&lt;cluster&gt;.h</code>              | 创建低级 C 函数头文件。                       |
| <code>cluster.c.jinja</code>                        | <code>iotmi_device_&lt;cluster&gt;.c</code>              | 使用数据模型处理程序实现和注册回调函数指针。              |
| <code>cluster_type_helpers.h.jinja</code>           | <code>iotmi_device_type_helpers_&lt;cluster&gt;.h</code> | 定义数据类型的函数原型。                        |
| <code>cluster_type_helpers.c.jinja</code>           | <code>iotmi_device_type_helpers_&lt;cluster&gt;.c</code> | 为特定于集群的枚举、位图、列表和结构生成数据类型函数原型。       |
| <code>iot_device_dm_types.h.jinja</code>            | <code>iotmi_device_dm_types.h</code>                     | 为全局数据类型定义 C 数据类型。                   |
| <code>iot_device_type_helpers_global.h.jinja</code> | <code>iotmi_device_type_helpers_global.h</code>          | 为全局操作定义 C 数据类型。                     |
| <code>iot_device_type_helpers_global.c.jinja</code> | <code>iotmi_device_type_helpers_global.c</code>          | 声明标准数据类型，包括布尔值、整数、浮点数、字符串、位图、列表和结构。 |

## ( 可选 ) 自定义架构

终端设备 SDK 将标准化代码生成过程与自定义架构相结合。这可以为您的设备和设备软件扩展 Matter 数据模型。自定义架构可以帮助描述设备的 device-to-cloud 通信能力。

有关托管集成数据模型的详细信息，包括格式、结构和要求，请参阅[托管集成数据模型](#)。

使用 `codegen.py` 工具为自定义架构生成 C 源文件，如下所示：

**Note**

对于以下三个文件，每个自定义集群都需要相同的集群 ID。

- 创建自定义架构，其JSON格式应为能力报告提供集群的表示形式，以便在云中创建新的自定义集群。示例文件位于codegen/custom\_schemas/custom.SimpleLighting@1.0。
- 以包含与自定义架构相同信息的XML格式创建 ZCL ( Zigbee 集群库 ) 定义文件。使用 ZAP 工具从 ZCL XML 生成你的 Matter IDL 文件。示例文件位于codegen/zcl/custom.SimpleLighting.xml。
- ZAP 工具的输出是，它定义了Matter IDL File (.matter)与您的自定义架构相对应的 Matter 集群。这是为终端设备 SDK 生成 C 源文件的codegen.py工具的输入。示例文件位于codegen/matter\_files/custom-light.matter。

有关如何将自定义托管集成数据模型集成到代码生成工作流程中的详细说明，请参阅[为设备生成代码](#)。

## 环境设置

了解如何配置您的环境以使用codegen.py代码生成器。

### 主题

- [先决条件](#)
- [配置环境](#)

### 先决条件

在配置环境之前，请安装以下项目：

- Git
- Python 3.10 或更高版本
- 诗歌 1.2.0 或更高版本

### 配置环境

使用以下过程将您的环境配置为使用 codegen.py 代码生成器。

1. 从下载最新版本的[终端设备 SDK AWS 管理控制台](#)。

## 2. 设置 Python 环境。代码生成项目基于 python，使用 Poetry 进行依赖关系管理。

- 在codegen目录中使用 poetry 安装项目依赖项：

```
poetry run poetry install --no-root
```

## 3. 设置您的存储库。

- 克隆connectedhomeip存储库。它使用位于connectedhomeip/scripts/文件夹中的codegen.py脚本生成代码。欲了解更多信息，请参阅上的 [connectedhomeip](#)。GitHub

```
git clone -b v1.4.0.0 https://github.com/project-chip/connectedhomeip.git
```

- 将其克隆到与IoT-managed-integrations-End-Device-SDK 根文件夹相同的级别。您的文件夹结构应与以下内容相匹配：

```
| -connectedhomeip
| -IoT-managed-integrations-End-Device-SDK
```

### Note

你不需要递归克隆子模块。

## 为设备生成代码

使用托管集成代码生成工具为您的设备创建自定义 C 代码。本节介绍如何从 SDK 附带的示例文件或您自己的规范中生成代码。学习如何使用生成脚本、了解工作流程以及如何创建符合设备要求的代码。

### 主题

- [先决条件](#)
- [为自定义.matter 文件生成代码](#)
- [代码生成工作流程](#)

### 先决条件

- Python 3.10 或更高版本。

2. 从.matter 文件开始生成代码。终端设备 SDK 在以下文件中提供了两个示例文件codgen/matter\_files folder :

- custom-air-purifier.matter
- aws\_camera.matter

**Note**

这些示例文件为演示应用程序集群生成代码。

## 生成代码

运行以下命令在 out 文件夹中生成代码：

```
bash ./gen-data-model-api.sh
```

## 为自定义.matter 文件生成代码

要为特定.matter文件生成代码或提供您自己的.matter文件，请执行以下任务。

为自定义.matter 文件生成代码

1. 准备好你的.matter 文件
2. 运行生成命令：

```
./codegen.sh [--format] configs/dm_basic.json path-to-matter-file output-directory
```

( 可选 ) 使用自定义架构生成代码

1. 按JSON格式准备自定义架构
2. 运行生成命令：

```
./codegen.sh [--format] configs/dm_basic.json path-to-matter-file output-directory
--custom-schemas-dir path-to-custom-schema-directory
```

上面的命令使用多个组件将您的.matter文件转换为C代码：

- `codegen.py`来自ConnectedHome知识产权项目
- Python 插件位于 `codegen/py_scripts/iotmi_data_model.py`
- 文件夹中的 Jinja2 模板 `codegen/py_scripts/templates`

该插件定义了要传递给 Jinja2 模板的变量，然后使用这些变量生成最终的 C 代码输出。添加该 `--format` 标志会将 Clang 格式应用于生成的代码。

## 代码生成工作流程

代码生成过程使用实用函数和拓扑排序来组织您的 `.matter` 文件数据结构。`topsort.py`这样可以确保数据类型及其依赖关系的正确排序。

然后，该脚本将您的 `.matter` 文件规范与 Python 插件处理相结合，以提取和格式化必要的信息。最后，它应用 Jinja2 模板格式来创建最终的 C 代码输出。

此工作流程可确保将 `.matter` 文件中的设备特定要求准确地转换为与托管集成系统集成的功能性 C 代码。

## 低级 C 函数 APIs

使用提供的低级 C-Function 将您的设备专用代码与托管集成集成。APIs本节介绍 AWS 数据模型中每个集群可用的 API 操作，以实现设备到云端的高效交互。了解如何实现回调函数、发出事件、通知属性更改以及为设备终端节点注册集群。

关键的 API 组件包括：

1. 属性和命令的回调函数指针结构
2. 事件发射函数
3. 属性变更通知功能
4. 集群注册功能

通过实现这些功能 APIs，您可以在设备的物理操作和托管集成云功能之间架起一座桥梁，从而确保无缝通信和控制。

以下部分说明了 [OnOff集群](#) API。

## OnOff 集群 API

集 [OnOff.xml](#) 群支持以下属性和命令:

- 属性：
  - OnOff (boolean)
  - GlobalSceneControl (boolean)
  - OnTime (int16u)
  - OffWaitTime (int16u)
  - StartUpOnOff (StartUpOnOffEnum)
- 命令：
  - Off : () -> Status
  - On : () -> Status
  - Toggle : () -> Status
  - OffWithEffect : (EffectIdentifier: EffectIdentifierEnum, EffectVariant: enum8) -> Status
  - OnWithRecallGlobalScene : () -> Status
  - OnWithTimedOff : (OnOffControl: OnOffControlBitmap, OnTime: int16u, OffWaitTime: int16u) -> Status

对于每个命令，我们都提供了 1:1 映射的函数指针，你可以用它来挂钩你的实现。

属性和命令的所有回调都是在以集群命名的 C 结构中定义的。

### 示例 C 结构

```
struct iotmiDev_clusterOnOff
{
 /*
 - Each attribute has a getter callback if it's readable
 - Each attribute has a setter callback if it's writable
 - The type of `value` are derived according to the data type of
 the attribute.
 - `user` is the pointer passed during an endpoint setup
```

```

- The callback should return iotmiDev_DMStatus to report success or not.

- For unsupported attributes, just leave them as NULL.
*/
iotmiDev_DMStatus (*getOnTime)(uint16_t *value, void *user);
iotmiDev_DMStatus (*setOnTime)(uint16_t value, void *user);
/*
- Each command has a command callback

- If a command takes parameters, the parameters will be defined in a struct
 such as `iotmiDev_OnOff_OnWithTimedOffRequest` below.

- `user` is the pointer passed during an endpoint setup

- The callback should return iotmiDev_DMStatus to report success or not.

- For unsupported commands, just leave them as NULL.
*/
iotmiDev_DMStatus (*cmdOff)(void *user);
iotmiDev_DMStatus (*cmdOnWithTimedOff)(const iotmiDev_OnOff_OnWithTimedOffRequest
*request, void *user);
};

```

除了 C 结构外，还为所有属性定义了属性变更报告函数。

```

/* Each attribute has a report function for the customer to report
 an attribute change. An attribute report function is thread-safe.
*/
void iotmiDev_OnOff_OnTime_report_attr(struct iotmiDev_Endpoint *endpoint, uint16_t
newValue, bool immediate);

```

事件报告功能是为所有特定于集群的事件定义的。由于集OnOff群未定义任何事件，因此以下是该CameraAvStreamManagement集群的示例。

```

/* Each event has a report function for the customer to report
 an event. An event report function is thread-safe.
 The iotmiDev_CameraAvStreamManagement_VideoStreamChangedEvent struct is
 derived from the event definition in the cluster.
*/

```

```
void iotmiDev_CameraAvStreamManagement_VideoStreamChanged_report_event(struct
 iotmiDev_Endpoint *endpoint, const
 iotmiDev_CameraAvStreamManagement_VideoStreamChangedEvent *event, bool immediate);
```

每个集群还具有寄存器功能。

```
iotmiDev_DMStatus iotmiDev_OnOffRegisterCluster(struct iotmiDev_Endpoint *endpoint,
 const struct iotmiDev_clusterOnOff *cluster, void *user);
```

传递给寄存器函数的用户指针将传递给回调函数。

## 托管集成中的功能和设备交互

本节介绍了 C-Function 实现的作用以及设备与托管集成设备功能之间的交互。

主题

- [处理远程命令](#)
- [处理不请自来的事件](#)

### 处理远程命令

远程命令由终端设备 SDK 与该功能之间的交互来处理。以下操作描述了如何使用此交互打开灯泡的示例。

MQTT 客户端接收有效负载并传递给数据模型处理器

当您发送远程命令时，MQTT 客户端会接收 JSON 格式的托管集成消息。然后，它将有效载荷传递给数据模型处理程序。例如，假设你想使用托管集成来打开灯泡。灯泡有一个支持 OnOff 集群的终端节点 #1。在这种情况下，当您发送打开灯泡的命令时，托管集成会通过 MQTT 向设备发送请求，表示它要在端点 #1 上调用 On 命令。

数据模型处理程序检查回调函数并调用它们

数据模型处理程序解析 JSON 请求。如果请求包含属性或操作，则数据模型处理程序会找到端点并按顺序调用相应的回调函数。例如，对于灯泡，当数据模型处理程序收到 MQTT 消息时，它会检查与 OnOff 集群中定义的 On 命令相对应的回调函数是否已注册到终端节点 #1 上。

## 处理程序和 C 函数实现执行命令

数据模型处理程序调用它找到的相应回调函数并调用它们。然后，C-Function 实现调用相应的硬件函数来控制物理硬件并返回执行结果。例如，对于灯泡，数据模型处理程序调用回调函数并存储执行结果。然后，回调函数会打开灯泡。

### 数据模型处理程序返回执行结果

调用所有回调函数后，数据模型处理程序会合并所有结果。然后，它以 JSON 格式打包响应，并使用 MQTT 客户端将结果发布到托管集成云中。对于灯泡，响应中的 MQTT 消息将包含回调函数打开灯泡的结果。

## 处理不请自来的事件

终端设备 SDK 与该功能之间的交互也会处理未经请求的事件。以下操作描述了操作方法。

### 设备向数据模型处理器发送通知

当发生属性更改或事件时，例如在设备上按下物理按钮时，C-Function 实现会生成未经请求的事件通知，并调用相应的通知函数将通知发送给数据模型处理程序。

### 数据模型处理程序翻译通知

数据模型处理程序处理收到的通知并将其转换为 AWS 数据模型。

### 数据模型处理程序向云端发布通知

然后，数据模型处理程序使用 MQTT 客户端将未经请求的事件发布到托管集成云中。

## 开始使用终端设备 SDK

按照以下步骤在 Linux 设备上运行终端设备 SDK。本节将指导您完成环境设置、网络配置、硬件功能实现和端点配置。

### Important

examples 目录中的演示应用程序及其中的平台抽象层 (PAL) 实现 platform/posix 仅供参考。请勿在生产环境中使用它们。

仔细查看以下步骤的每个步骤，确保设备与托管集成的正确集成。

## 集成终端设备 SDK

### 1. 设置亚马逊 EC2 实例

登录 AWS 管理控制台 并使用亚马逊 Linux AMI 启动亚马逊 EC2 实例。请参阅 [《亚马逊弹性容器注册表用户指南》](#) EC2 中的“亚马逊入门”。

### 2. 设置构建环境

作为开发主机，在亚马逊 Linux 2023/x86\_64 上构建代码。安装必要的编译依赖项：

```
dnf install make gcc gcc-c++ cmake
```

### 3. ( 可选 ) 设置网络

终端设备 SDK 最好与物理硬件配合使用。如果使用 Amazon EC2，请不要执行此步骤。

如果您在使用示例应用程序 EC2 之前未使用 Amazon，请初始化网络并将您的设备连接到可用的 Wi-Fi 网络。在设备配置之前完成网络设置：

```
/* Provisioning the device PKCS11 with claim credential. */
status = deviceCredentialProvisioning();
```

### 4. 配置配置参数

#### Note

在继续操作之前，请按照 [Provisionee](#) 获取索赔证书和私钥。

example/project\_name/device\_config.sh 使用以下配置参数修改配置文件：

#### 配置参数

| 宏观参数                   | 说明         | 如何获取此信息                                                             |
|------------------------|------------|---------------------------------------------------------------------|
| IOTMI_R00<br>T_CA_PATH | 根 CA 证书文件。 | 您可以从 AWS IoT Core 开发者指南的 <a href="#">下载 Amazon 根 CA 证书</a> 部分下载此文件。 |

| 宏观参数                                | 说明             | 如何获取此信息                                                                                               |
|-------------------------------------|----------------|-------------------------------------------------------------------------------------------------------|
| IOTMI_CLAIM_CERTIFICATE_PATH        | 索赔证书文件的路径。     | 要获取声明证书和私钥，请使用 <a href="#">CreateProvisioningProfile</a> API 创建配置文件。有关说明，请参阅 <a href="#">创建配置文件</a> 。 |
| IOTMI_CLAIM_PRIVATE_KEY_PATH        | 声明私钥文件的路径。     |                                                                                                       |
| IOTMI_MANAGED_INTEGRATIONS_ENDPOINT | 托管集成的终端节点 URL。 | 要获取托管集成端点，请使用 <a href="#">RegisterCustomEndpoint</a> API。有关说明，请参阅 <a href="#">注册自定义终端节点</a> 。         |
| IOTMI 托管集成_端点_端口                    | 托管集成端点的端口号     | 默认情况下，端口 8883 用于 MQTT 发布和订阅操作。端口 443 设置为设备使用的应用层协议协商 (ALPN) TLS 扩展。                                   |

## 5. 构建并运行演示应用程序

本节演示了两个 Linux 演示应用程序：一个简单的安全摄像头和一个空气净化器，两者都 CMake 用作构建系统。

### a. 简单的安全摄像头应用程序

要生成并运行应用程序，请执行以下命令：

```
>cd <path-to-code-drop>
If you didn't generate cluster code earlier
>(cd codegen && poetry run poetry install --no-root && ./gen-data-model-api.sh)
>mkdir build
>cd build
>cmake ..
>cmake -build .
>./examples/iotmi_device_sample_camera/iotmi_device_sample_camera
```

此演示为带有 RTC 会话控制器和录制集群的模拟摄像机实现了低级 C 函数。在运行 [置备人工工作流程](#) 之前完成中提到的流程。

## 演示应用程序的输出示例：

```
[2406832727][MAIN][INFO] ===== Device initialization and WIFI provisioning
=====
[2406832728][MAIN][INFO] fleetProvisioningTemplateName: XXXXXXXXXXXX
[2406832728][MAIN][INFO] managedintegrationsEndpoint: XXXXXXXXXXXX.account-prefix-
ats.iot.region.amazonaws.com
[2406832728][MAIN][INFO] pDeviceSerialNumber: XXXXXXXXXXXX
[2406832728][MAIN][INFO] universalProductCode: XXXXXXXXXXXX
[2406832728][MAIN][INFO] rootCertificatePath: XXXXXXXXXX
[2406832728][MAIN][INFO] pClaimCertificatePath: XXXXXXXXXX
[2406832728][MAIN][INFO] pClaimKeyPath: XXXXXXXXXXXXXXXXXXXX
[2406832728][MAIN][INFO] deviceInfo.serialNumber XXXXXXXXXXXX
[2406832728][MAIN][INFO] deviceInfo.universalProductCode XXXXXXXXXXXXXXXXXXXX
[2406832728][PKCS11][INFO] PKCS #11 successfully initialized.
[2406832728][MAIN][INFO] ===== Start certificate provisioning
=====
[2406832728][PKCS11][INFO] ===== Loading Root CA and claim credentials
through PKCS#11 interface =====
[2406832728][PKCS11][INFO] Writing certificate into label "Root Cert".
[2406832728][PKCS11][INFO] Creating a 0x1 type object.
[2406832728][PKCS11][INFO] Writing certificate into label "Claim Cert".
[2406832728][PKCS11][INFO] Creating a 0x1 type object.
[2406832728][PKCS11][INFO] Creating a 0x3 type object.
[2406832728][MAIN][INFO] ===== Fleet-provisioning-by-Claim =====
[2025-01-02 01:43:11.404995144][iotmi_device_sdkLog][INFO] [2406832728]
[MQTT_AGENT][INFO]
[2025-01-02 01:43:11.405106991][iotmi_device_sdkLog][INFO] Establishing a TLS
session to XXXXXXXXXXXXXXXXXXXX.account-prefix-ats.iot.region.amazonaws.com
[2025-01-02 01:43:11.405119166][iotmi_device_sdkLog][INFO]
[2025-01-02 01:43:11.844812513][iotmi_device_sdkLog][INFO] [2406833168]
[MQTT_AGENT][INFO]
[2025-01-02 01:43:11.844842576][iotmi_device_sdkLog][INFO] TLS session
connected
[2025-01-02 01:43:11.844852105][iotmi_device_sdkLog][INFO]
[2025-01-02 01:43:12.296421687][iotmi_device_sdkLog][INFO] [2406833620]
[MQTT_AGENT][INFO]
[2025-01-02 01:43:12.296449663][iotmi_device_sdkLog][INFO] Session present: 0.
[2025-01-02 01:43:12.296458997][iotmi_device_sdkLog][INFO]
[2025-01-02 01:43:12.296467793][iotmi_device_sdkLog][INFO] [2406833620]
[MQTT_AGENT][INFO]
[2025-01-02 01:43:12.296476275][iotmi_device_sdkLog][INFO] MQTT connect with
clean session.
```

```
[2025-01-02 01:43:12.296484350][iotmi_device_sdkLog][INFO]
[2025-01-02 01:43:13.171056119][iotmi_device_sdkLog][INFO] [2406834494]
[FLEET_PROVISIONING][INFO]
[2025-01-02 01:43:13.171082442][iotmi_device_sdkLog][INFO] Received accepted
response from Fleet Provisioning CreateKeysAndCertificate API.
[2025-01-02 01:43:13.171092740][iotmi_device_sdkLog][INFO]
[2025-01-02 01:43:13.171122834][iotmi_device_sdkLog][INFO] [2406834494]
[FLEET_PROVISIONING][INFO]
[2025-01-02 01:43:13.171132400][iotmi_device_sdkLog][INFO] Received privatekey
and certificate with Id: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
[2025-01-02 01:43:13.171141107][iotmi_device_sdkLog][INFO]
[2406834494][PKCS11][INFO] Creating a 0x3 type object.
[2406834494][PKCS11][INFO] Writing certificate into label "Device Cert".
[2406834494][PKCS11][INFO] Creating a 0x1 type object.
[2025-01-02 01:43:18.584615126][iotmi_device_sdkLog][INFO] [2406839908]
[FLEET_PROVISIONING][INFO]
[2025-01-02 01:43:18.584662031][iotmi_device_sdkLog][INFO] Received accepted
response from Fleet Provisioning RegisterThing API.
[2025-01-02 01:43:18.584671912][iotmi_device_sdkLog][INFO]
[2025-01-02 01:43:19.100030237][iotmi_device_sdkLog][INFO] [2406840423]
[FLEET_PROVISIONING][INFO]
[2025-01-02 01:43:19.100061720][iotmi_device_sdkLog][INFO] Fleet-provisioning
iteration 1 is successful.
[2025-01-02 01:43:19.100072401][iotmi_device_sdkLog][INFO]
[2406840423][MQTT][ERROR] MQTT Connection Disconnected Successfully
[2025-01-02 01:43:19.216938181][iotmi_device_sdkLog][INFO] [2406840540]
[MQTT_AGENT][INFO]
[2025-01-02 01:43:19.216963713][iotmi_device_sdkLog][INFO] MQTT agent thread
leaves thread loop for iotmiDev_MQTTAgentStop.
[2025-01-02 01:43:19.216973740][iotmi_device_sdkLog][INFO]
[2406840540][MAIN][INFO] iotmiDev_MQTTAgentStop is called to break thread loop
function.
[2406840540][MAIN][INFO] Successfully provision the device.
[2406840540][MAIN][INFO] Client ID :
XXXXXXXXXXXXXXXXXXXXX_XXXXXXXXXXXXXXXXXXXXX
[2406840540][MAIN][INFO] Managed thing ID : XXXXXXXXXXXXXXXXXXXXXXXX
[2406840540][MAIN][INFO] ===== application loop
=====
[2025-01-02 01:43:19.217094828][iotmi_device_sdkLog][INFO] [2406840540]
[MQTT_AGENT][INFO]
[2025-01-02 01:43:19.217124600][iotmi_device_sdkLog][INFO] Establishing a TLS
session to XXXXXXXXX.account-prefix-ats.iot.region.amazonaws.com:8883
[2025-01-02 01:43:19.217138724][iotmi_device_sdkLog][INFO]
[2406840540][Cluster On0ff][INFO] exampleOn0ffInitCluster() for endpoint#1
```

```
[2406840540][MAIN][INFO] Press Ctrl+C when you finish testing...
[2406840540][Cluster ActivatedCarbonFilterMonitoring][INFO]
 exampleActivatedCarbonFilterMonitoringInitCluster() for endpoint#1
[2406840540][Cluster AirQuality][INFO] exampleAirQualityInitCluster() for
 endpoint#1
[2406840540][Cluster CarbonDioxideConcentrationMeasurement][INFO]
 exampleCarbonDioxideConcentrationMeasurementInitCluster() for endpoint#1
[2406840540][Cluster FanControl][INFO] exampleFanControlInitCluster() for
 endpoint#1
[2406840540][Cluster HepaFilterMonitoring][INFO]
 exampleHepaFilterMonitoringInitCluster() for endpoint#1
[2406840540][Cluster Pm1ConcentrationMeasurement][INFO]
 examplePm1ConcentrationMeasurementInitCluster() for endpoint#1
[2406840540][Cluster Pm25ConcentrationMeasurement][INFO]
 examplePm25ConcentrationMeasurementInitCluster() for endpoint#1
[2406840540][Cluster TotalVolatileOrganicCompoundsConcentrationMeasurement]
[INFO]
 exampleTotalVolatileOrganicCompoundsConcentrationMeasurementInitCluster() for
 endpoint#1
[2025-01-02 01:43:19.648185488][iotmi_device_sdkLog][INFO] [2406840971]
[MQTT_AGENT][INFO]
[2025-01-02 01:43:19.648211988][iotmi_device_sdkLog][INFO] TLS session
 connected
[2025-01-02 01:43:19.648225583][iotmi_device_sdkLog][INFO]

[2025-01-02 01:43:19.938281231][iotmi_device_sdkLog][INFO] [2406841261]
[MQTT_AGENT][INFO]
[2025-01-02 01:43:19.938304799][iotmi_device_sdkLog][INFO] Session present: 0.
[2025-01-02 01:43:19.938317404][iotmi_device_sdkLog][INFO]
```

## b. 简单的空气净化器应用

要生成并运行应用程序，请运行以下命令：

```
>cd <path-to-code-drop>
If you didn't generate cluster code earlier
>(cd codegen && poetry run poetry install --no-root && ./gen-data-model-api.sh)
>mkdir build
>cd build
>cmake ..
>cmake --build .
>./examples/iotmi_device_dm_air_purifier/iotmi_device_dm_air_purifier_demo
```

此演示为具有两个端点和以下支持的集群的模拟空气净化器实现了低级 C 函数：

### 空气净化器端点支持的集群

| 端点             | 集群            |
|----------------|---------------|
| 终点 #1: 空气净化器   | OnOff         |
|                | 风扇控制          |
|                | HEPA 过滤器监测    |
|                | 活性炭过滤器监测      |
| 终点 #2: 空气质量传感器 | 空气质量          |
|                | 二氧化碳浓度测量      |
|                | 甲醛浓度测量        |
|                | Pm25 浓度测量     |
|                | Pm1 浓度测量      |
|                | 总挥发性有机化合物浓度测量 |

输出与相机演示应用程序类似，但支持的集群不同。

## 6. 后续步骤：

托管集成终端设备软件开发工具包和演示应用程序现已在您的 Amazon EC2 实例上运行。这使您能够在自己的物理硬件上开发和测试应用程序。通过此设置，您可以利用托管集成服务来控制您的 AWS IoT 设备。

### a. 开发硬件回调函数

在实现硬件回调函数之前，请先了解 API 的工作原理。此示例使用 On/Off 群集和 OnOff 属性来控制设备功能。有关 API 的详细信息，请参阅[低级 C 函数 APIs](#)。

```
struct DeviceState
{
```

```
struct iotmiDev_Agent *agent;
struct iotmiDev_Endpoint *endpointLight;
/* This simulates the HW state of OnOff */
bool hwState;
};

/* This implementation for OnOff getter just reads
the state from the DeviceState */
iotmiDev_DMStatus exampleGetOnOff(bool *value, void *user)
{
 struct DeviceState *state = (struct DeviceState *) (user);
 *value = state->hwState;
 return iotmiDev_DMStatusOk;
}
```

## b. 设置端点并挂接硬件回调函数

实现函数后，创建端点并注册您的回调。完成以下任务：

### i. 创建设备代理。

- A. 在调用任何其他 SDK 函数 `iotmiDev_Agent_new()` 之前，使用创建设备代理。
- B. 您的配置必须至少包含 `thingID` 和 `clientID` 参数。
- C. 使用该 `iotmiDev_Agent_initDefaultConfig()` 函数为队列大小和最大端点等参数设置合理的默认值。
- D. 使用完资源后，使用该 `iotmiDev_Agent_free()` 函数将其释放。这样可以防止内存泄漏并确保在应用程序中进行适当的资源管理。

### ii. 为要支持的每个集群结构填充回调函数指针。

### iii. 设置终端节点并注册支持的集群。

使用创建终端节点 `iotmiDev_Agent_addEndpoint()`，这需要：

- A. 唯一的终端节点 ID。
- B. 描述性端点名称
- C. 一种或多种符合 AWS 数据模型定义的设备类型。
- D. 创建终端节点后，使用相应的集群特定注册功能注册集群。
- E. 每个集群注册都需要使用属性和命令的回调函数。系统会将您的用户上下文指针传递给回调，以在两次调用之间保持状态。

```
struct DeviceState
{
 struct iotmiDev_Agent * agent;
 struct iotmiDev_Endpoint *endpoint1;

 /* OnOff cluster states*/
 bool hwState;
};

/* This implementation for OnOff getter just reads
the state from the DeviceState */
iotmiDev_DMStatus exampleGetOnOff(bool * value, void * user)
{
 struct DeviceState * state = (struct DeviceState *) (user);
 *value = state->hwState;
 printf("%s(): state->hwState: %d\n", __func__, state->hwState);
 return iotmiDev_DMStatusOk;
}

iotmiDev_DMStatus exampleGetOnTime(uint16_t * value, void * user)
{
 *value = 0;
 printf("%s(): OnTime is %u\n", __func__, *value);
 return iotmiDev_DMStatusOk;
}

iotmiDev_DMStatus exampleGetStartupOnOff(iotmiDev_OnOff_StartUpOnOffEnum *
value, void * user)
{
 *value = iotmiDev_OnOff_StartUpOnOffEnum_Off;
 printf("%s(): StartupOnOff is %d\n", __func__, *value);
 return iotmiDev_DMStatusOk;
}

void setupOnOff(struct DeviceState *state)
{
 struct iotmiDev_clusterOnOff clusterOnOff = {
 .getOnOff = exampleGetOnOff,
 .getOnTime = exampleGetOnTime,
 .getStartupOnOff = exampleGetStartupOnOff,
 };
};
```

```
iotmiDev_OnOffRegisterCluster(state->endpoint1,
 &clusterOnOff,
 (void *) state);
}

/* Here is the sample setting up an endpoint 1 with OnOff
 cluster. Note all error handling code is omitted. */
void setupAgent(struct DeviceState *state)
{
 struct iotmiDev_Agent_Config config = {
 .thingId = IOTMI_DEVICE_MANAGED_THING_ID,
 .clientId = IOTMI_DEVICE_CLIENT_ID,
 };
 iotmiDev_Agent_InitDefaultConfig(&config);

 /* Create a device agent before calling other SDK APIs */
 state->agent = iotmiDev_Agent_new(&config);

 /* Create endpoint#1 */
 state->endpoint1 = iotmiDev_Agent_addEndpoint(state->agent,
 1,
 "Data Model Handler Test
Device",
 (const char*[])
{ "Camera" },
 1);
 setupOnOff(state);
}
```

c. 使用作业处理程序获取作业文档

i. 发起对您的 OTA 应用程序的调用：

```
static iotmi_JobCurrentStatus_t processOTA(iotmi_JobData_t * pJobData)
{
 iotmi_JobCurrentStatus_t jobCurrentStatus = JobSucceeded;

 ...
 // This function should create OTA tasks
 jobCurrentStatus = YOUR_OTA_FUNCTION(iotmi_JobData_t * pJobData);
 ...

 return jobCurrentStatus;
}
```

```
}
```

- ii. 调用 `iotmi_JobsHandler_start` 以初始化作业处理程序。
- iii. `iotmi_JobsHandler_getJobDocument` 致电从托管集成中检索任务文档。
- iv. 成功获取任务文档后，在 `processOTA` 函数中写入您的自定义 OTA 操作并返回 `JobSucceeded` 状态。

```
static void prvJobsHandlerThread(void * pParam)
{
 JobsHandlerStatus_t status = JobsHandlerSuccess;
 iotmi_JobData_t jobDocument;
 iotmiDev_DeviceRecord_t * pThreadParams = (iotmiDev_DeviceRecord_t *)
pParam;
 iotmi_JobsHandler_config_t config = { .pManagedThingID = pThreadParams-
>pManagedThingID, .jobsQueueSize = 10 };

 status = iotmi_JobsHandler_start(&config);

 if(status != JobsHandlerSuccess)
 {
 LogError(("Failed to start Jobs Handler."));
 return;
 }

 while(!bExit)
 {
 status = iotmi_JobsHandler_getJobDocument(&jobDocument, 30000);

 switch(status)
 {
 case JobsHandlerSuccess:
 {
 LogInfo(("Job document received."));
 LogInfo(("Job ID: %.*s", (int) jobDocument.jobIdLength,
jobDocument.pJobId));
 LogInfo(("Job document: %.*s", (int)
jobDocument.jobDocumentLength, jobDocument.pJobDocument));

 /* Process the job document */
 iotmi_JobCurrentStatus_t jobStatus =
processOTA(&jobDocument);
 }
 }
 }
}
```

```
 iotmi_JobsHandler_updateJobStatus(jobDocument.pJobId,
jobDocument.jobIdLength, jobStatus, NULL, 0);

 iotmiJobsHandler_destroyJobDocument(&jobDocument);

 break;
 }
 case JobsHandlerTimeout:
 {
 LogInfo(("No job document available. Polling for job
document."));

 iotmi_JobsHandler_pollJobDocument();

 break;
 }
 default:
 {
 LogError(("Failed to get job document."));
 break;
 }
}
}

while(iotmi_JobsHandler_getJobDocument(&jobDocument, 0) ==
JobsHandlerSuccess)
{
 /* Before stopping the Jobs Handler, process all the remaining
jobs. */

 LogInfo(("Job document received before stopping."));
 LogInfo(("Job ID: %.*s", (int) jobDocument.jobIdLength,
jobDocument.pJobId));
 LogInfo(("Job document: %.*s", (int)
jobDocument.jobDocumentLength, jobDocument.pJobDocument));

 storeJobs(&jobDocument);

 iotmiJobsHandler_destroyJobDocument(&jobDocument);
}

iotmi_JobsHandler_stop();

LogInfo(("Job handler thread end."));
```

```
}
```

## 将终端设备 SDK 移植到您的设备上

将终端设备 SDK 移植到您的设备平台。请按照以下步骤将您的设备与 AWS IoT 设备管理连接起来。

### 下载并验证终端设备 SDK

1. 从[托管集成控制台](#)下载最新版本的终端设备 SDK。
2. 验证您的平台是否在支持的平台列表中[参考：支持的平台](#)。

#### Note

终端设备 SDK 已在指定平台上进行了测试。其他平台可能有效，但尚未经过测试。

3. 将 SDK 文件提取（解压缩）到您的工作区。
4. 使用以下设置配置您的构建环境：
  - 源文件路径
  - 头文件目录
  - 所需的库
  - 编译器和链接器标志
5. 在移植平台抽象层 (PAL) 之前，请确保平台的基本功能已初始化。功能包括：
  - 操作系统任务
  - 外围设备
  - 网络接口
  - 特定于平台的要求

### 将 PAL 移植到您的设备上

1. 在现有平台目录中为特定于平台的实现创建一个新目录。例如，如果您使用 FreeRTOS，请在上创建一个目录。platform/freertos

## Example SDK 目录结构

```
<SDK_ROOT_FOLDER>
CMakeLists.txt
LICENSE.txt
cmake
commonDependencies
components
docs
examples
include
lib
platform
test
tools
```

2. 将 POSIX 参考实现文件 (.c 和.h) 从 posix 文件夹复制到新的平台目录中。这些文件为您需要实现的函数提供了一个模板。
  - 凭据存储的闪存管理
  - PKCS #11 的实现
  - 网络传输接口
  - 时间同步
  - 系统重启和重置功能
  - 日志记录机制
  - 特定于设备的配置
3. 使用 TLS 设置传输层安全 (TLS) 身份验证。Mbed
  - 如果您已经有与平台上的 SDK 版本相匹配的 Mbed TLS 版本，请使用提供的 POSIX 实现。
  - 使用不同的 TLS 版本，您可以使用堆栈为 TLS 堆栈实现传输挂钩。TCP/IP
4. 将您平台的 mbedTLS 配置与中的软件开发工具包要求进行比较。platform/posix/mbedtls/mbedtls\_config.h 确保所有必需的选项都已启用。
5. 该软件开发工具包依赖 CoreMQTT 与云端进行交互。因此，您必须实现使用以下结构的网络传输层：

```
typedef struct TransportInterface
{
 TransportRecv_t recv;
 TransportSend_t send;
 NetworkContext_t * pNetworkContext;
} TransportInterface_t;
```

有关更多信息，请参阅 FreeRTOS 网站上的[传输接口文档](#)。

6. (可选) SDK 使用 PKCS #11 API 来处理证书操作。CorePKCS 是用于原型设计的非硬件特定的 PKCS #11 实现。我们建议您在生产环境中使用安全的加密处理器，例如可信平台模块 (TPM)、硬件安全模块 (HSM) 或安全元素：

- 查看使用 Linux 文件系统进行凭据管理的 PKCS #11 实现示例，网址为。platform/posix/corePKCS11-mbedtls
- 在以下位置实现 PKCS #11 PAL 层。commonDependencies/core\_pkcs11/corePKCS11/source/include/core\_pkcs11.h
- 在上实现 Linux 文件系统platform/posix/corePKCS11-mbedtls/source/iotmi\_pal\_Pkcs11operations.c。
- 在上实现您的存储类型的存储和加载功能platform/include/iotmi\_pal\_Nvm.h。
- 在中实现标准文件访问权限platform/posix/source/iotmi\_pal\_Nvm.c。

有关详细的移植说明，请参阅 FreeRTOS 用户指南中的[移植核心PKCS11库](#)。

7. 将 SDK 静态库添加到您的构建环境中：

- 设置库路径以解决任何链接器问题或符号冲突
- 验证所有依赖关系是否正确关联

## 测试你的端口

您可以使用现有的示例应用程序来测试您的端口。编译完成时必须没有任何错误或警告。

### Note

我们建议您从尽可能简单的多任务处理应用程序开始。示例应用程序提供了等效的多任务处理功能。

1. 在中查找示例应用程序examples/[device\_type\_sample]。
2. 将main.c文件转换为您的项目，然后添加一个条目来调用现有的 main () 函数。
3. 确认您可以成功编译演示应用程序。

## 技术参考

### 主题

- [参考：支持的平台](#)
- [参考：技术要求](#)
- [参考：常用 API](#)

### 参考：支持的平台

下表显示了 SDK 支持的平台。

#### 支持的平台

| 平台           | 架构               | 操作系统     |
|--------------|------------------|----------|
| Linux x86_64 | x86_64           | Linux    |
| Ambarella    | Armv8 () AArch64 | Linux    |
| ameBad       | armv8-M 32 位     | FreeRTOS |
| ESP32S3      | Xtensa 32 位 LX7  | FreeRTOS |

### 参考：技术要求

下表显示了 SDK 的技术要求，包括 RAM 空间。使用相同的配置时，终端设备 SDK 本身需要大约 5 到 10 MB 的 ROM 空间。

#### 内存空间

| 软件开发工具包和组件  | 空间要求 (已用字节) |
|-------------|-------------|
| 终端设备 SDK 本身 | 180 KB      |

| 软件开发工具包和组件     | 空间要求 ( 已用字节 )   |
|----------------|-----------------|
| 默认 MQTT 代理命令队列 | 480 字节 ( 可以配置 ) |
| 默认 MQTT 代理传入队列 | 320 字节 ( 可以配置 ) |

## 参考：常用 API

本部分列出了非特定于集群的 API 操作。

```
/* return code for data model related API */
enum iotmiDev_DMStatus
{
 /* The operation succeeded */
 iotmiDev_DMStatusOk = 0,
 /* The operation failed without additional information */
 iotmiDev_DMStatusFail = 1,
 /* The operation has not been implemented yet. */
 iotmiDev_DMStatusNotImplement = 2,
 /* The operation is to create a resource, but the resource already exists. */
 iotmiDev_DMStatusExist = 3,
}

/* The opaque type to represent a instance of device agent. */
struct iotmiDev_Agent;

/* The opaque type to represent an endpoint. */
struct iotmiDev_Endpoint;

/* A device agent should be created before calling other API */
struct iotmiDev_Agent* iotmiDev_create_agent();

/* Destroy the agent and free all occupied resources */
void iotmiDev_destroy_agent(struct iotmiDev_Agent *agent);

/* Add an endpoint, which starts with empty capabilities */
struct iotmiDev_Endpoint* iotmiDev_addEndpoint(struct iotmiDev_Agent *handle, uint16
 id, const char *name);

/* Test all clusters registered within an endpoint.
 Note: this API might exist only for early drop. */
```

```
void iotmiDev_testEndpoint(struct iotmiDev_Endpoint *endpoint);
```

# 托管集成中的安全性 AWS IoT Device Management

云安全 AWS 是重中之重。作为 AWS 客户，您可以受益于专为满足大多数安全敏感型组织的要求而构建的数据中心和网络架构。

安全是双方共同承担 AWS 的责任。[责任共担模式](#)将其描述为云的安全性和云中的安全性：

- 云安全 — AWS 负责保护在云中运行 AWS 服务的基础架构 AWS Cloud。AWS 还为您提供可以安全使用的服务。作为[AWS 合规计划](#)的一部分，第三方审计师定期测试和验证我们安全的有效性。要了解适用于托管集成的合规性计划，请参阅按合规计划划分的[范围内的AWS 服务按合规计划](#)。
- 云端安全-您的责任由您使用的 AWS 服务决定。您还需要对其他因素负责，包括您的数据的敏感性、您的公司的要求以及适用的法律法规。

本文档可帮助您了解在使用托管集成时如何应用责任共担模型。以下主题向您介绍如何配置托管集成以满足您的安全和合规性目标。您还将学习如何使用其他 AWS 服务来帮助您监控和保护托管集成资源。

## 主题

- [托管集成中的数据保护](#)
- [托管集成的身份和访问管理](#)
- [用 AWS Secrets Manager 于 C2C 工作流程的数据保护](#)
- [托管集成的合规性验证](#)
- [使用与接口 VPC 终端节点的托管集成](#)
- [连接到 AWS IoT Device Management FIPS 端点的托管集成](#)

## 托管集成中的数据保护

分 AWS [担责任模型](#)适用于托管集成中的数据保护。AWS IoT Device Management 如本模型所述 AWS，负责保护运行所有内容的全球基础架构 AWS Cloud。您负责维护对托管在此基础结构上的内容的控制。您还负责您所使用的 AWS 服务的安全配置和管理任务。有关数据隐私的更多信息，请参阅[数据隐私常见问题](#)。有关欧洲数据保护的信息，请参阅 AWS Security Blog 上的 [AWS Shared Responsibility Model and GDPR](#) 博客文章。

出于数据保护目的，我们建议您保护 AWS 账户凭证并使用 AWS IAM Identity Center 或 AWS Identity and Access Management (IAM) 设置个人用户。这样，每个用户只获得履行其工作职责所需的权限。还建议您通过以下方式保护数据：

- 对每个账户使用多重身份验证 ( MFA )。
- 用于 SSL/TLS 与 AWS 资源通信。我们要求使用 TLS 1.2，建议使用 TLS 1.3。
- 使用设置 API 和用户活动日志 AWS CloudTrail。有关使用 CloudTrail 跟踪捕获 AWS 活动的信息，请参阅《AWS CloudTrail 用户指南》中的[使用跟 CloudTrail 踪](#)。
- 使用 AWS 加密解决方案以及其中的所有默认安全控件 AWS 服务。
- 使用高级托管安全服务（例如 Amazon Macie），它有助于发现和保护存储在 Amazon S3 中的敏感数据。
- 如果您在 AWS 通过命令行界面或 API 进行访问时需要经过 FIPS 140-3 验证的加密模块，请使用 FIPS 端点。有关可用的 FIPS 端点的更多信息，请参阅[《美国联邦信息处理标准 \( FIPS \) 第 140-3 版》](#)。

强烈建议您切勿将机密信息或敏感信息（如您客户的电子邮件地址）放入标签或自由格式文本字段（如名称字段）。这包括您 AWS 服务使用控制台、API AWS IoT Device Management 或为其他人处理托管集成时。AWS CLI AWS SDKs 在用于名称的标签或自由格式文本字段中输入的任何数据都可能用于计费或诊断日志。如果您向外部服务器提供网址，强烈建议您不要在网址中包含凭证信息来验证对该服务器的请求。

## 用于托管集成的静态数据加密

默认情况下，托管集成使用 AWS IoT Device Management 加密密钥对敏感的静态客户数据进行加密。

有两种类型的加密密钥用于保护托管集成客户的敏感数据：

### 客户管理的密钥 (CMK)

托管集成支持使用您可以创建、拥有和管理的对称客户托管密钥。您可以完全控制这些 KMS 密钥，包括建立和维护其密钥策略、IAM policy 和授权、启用和禁用它们、轮换其加密材料、添加标签、创建别名（引用了 KMS 密钥）以及计划删除 KMS 密钥。

### AWS 拥有的密钥

默认情况下，托管集成使用这些密钥来自动加密敏感的客户数据。您无法查看、管理或审核其使用情况。您无需采取任何措施或更改任何程序即可保护加密数据的密钥。默认情况下，静态数据加密有助于

降低保护敏感数据的操作开销和复杂性。同时，它还支持构建符合严格加密合规性和监管要求的安全应用程序。

使用的默认加密密钥是 AWS 自有密钥。或者，更新加密密钥的可选 API 是 [PutDefaultEncryptionConfiguration](#)。

有关 AWS KMS 加密密钥类型的更多信息，请参阅 [AWS KMS 密钥](#)。

## AWS KMS 用于托管集成

托管集成使用信封加密来加密和解密所有客户数据。这种类型的加密将获取您的纯文本数据，并使用数据密钥对其进行加密。接下来，称为包装密钥的加密密钥将对用于加密纯文本数据的原始数据密钥进行加密。在信封加密中，可以使用其他包装密钥来加密与原始数据密钥分离程度更接近的现有包装密钥。由于原始数据密钥由单独存储的包装密钥加密，因此您可以将原始数据密钥和加密的纯文本数据存储在同一位置。密钥环用于生成、加密和解密数据密钥，此外还使用包装密钥来加密和解密数据密钥。

### Note

AWS 数据库加密 SDK 为您的客户端加密实现提供信封加密。有关 AWS 数据库加密 SDK 的更多信息，请参阅 [什么是 AWS 数据库加密 SDK ?](#)

有关信封加密、数据密钥、包装密钥和密钥环的更多信息，请参阅 [信封加密](#)、[数据密钥](#)、[包装密钥](#) 和 [密钥环](#)。

托管集成要求服务使用您的客户托管密钥进行以下内部操作：

- 向发送 DescribeKey 请求，AWS KMS 以验证轮换数据密钥时提供的对称客户托管密钥 ID。
- 向发送 GenerateDataKeyWithoutPlaintext 请求 AWS KMS 以生成由您的客户托管密钥加密的数据密钥。
- 向发送 ReEncrypt\* 请求，AWS KMS 要求使用您的客户托管密钥重新加密数据密钥。
- 向发送 Decrypt 请求，要求 AWS KMS 使用您的客户托管密钥解密数据。

## 使用加密密钥加密的数据类型

托管集成使用加密密钥来加密静态存储的多种类型的数据。以下列表概述了使用加密密钥进行静态加密的数据的类型：

- 云到云 (C2C) 连接器事件，例如设备发现和设备状态更新。

- 创建代表物理设备的托管事物和包含特定设备类型功能的设备配置文件。有关设备和设备配置文件的更多信息，请参阅[设备](#)和[设备](#)。
- 有关设备实现各个方面的托管集成通知。有关托管集成通知的更多信息，请参阅[设置托管集成通知](#)。
- 最终用户的个人身份信息 (PII)，例如设备身份验证材料、设备序列号、最终用户姓名、设备标识符和设备 Amazon 资源名称 (arn)。

## 托管集成如何使用关键策略 AWS KMS

对于分支密钥轮换和异步调用，托管集成需要密钥策略才能使用您的加密密钥。使用密钥策略的原因如下：

- 以编程方式授权其他 AWS 委托人使用加密密钥。

有关在托管集成中用于管理加密密钥访问权限的密钥策略示例，请参阅 [创建加密密钥](#)

### Note

对于 AWS 拥有的密钥，不需要密钥策略，因为 AWS 拥有的密钥归其所有 AWS，您无法查看、管理或使用它。默认情况下，托管集成使用 AWS 自有的密钥来自动加密您的敏感客户数据。

除了使用密钥策略来管理带有 AWS KMS 密钥的加密配置外，托管集成还使用 IAM 策略。有关 IAM 策略的更多信息，请参阅[中的策略和权限 AWS Identity and Access Management](#)。

## 创建加密密钥

您可以使用 AWS 管理控制台 或创建加密密钥 AWS KMS APIs。

### 创建加密密钥

按照 AWS Key Management Service 开发人员指南中[创建 KMS 密钥](#)的步骤进行操作。

### 密钥策略

密钥策略声明控制对 AWS KMS 密钥的访问权限。每个 AWS KMS 密钥将仅包含一个密钥策略。该密钥策略决定了哪些 AWS 委托人可以使用密钥以及他们如何使用密钥。有关使用密钥策略声明管理 AWS KMS 密钥访问和使用的更多信息，请参阅[使用策略管理访问权限](#)。

以下是密钥政策声明的示例，您可以使用它来管理托管集成中存储的 AWS KMS 密钥 AWS 账户 的访问和使用情况：

```
{
 "Statement" : [
 {
 "Sid" : "Allow access to principals authorized to use managed integrations",
 "Effect" : "Allow",
 "Principal" : {
 //Note: Both role and user are acceptable.
 "AWS" : "arn:aws:iam::111122223333:user/username",
 "AWS" : "arn:aws:iam::111122223333:role/roleName"
 },
 "Action" : [
 "kms:GenerateDataKeyWithoutPlaintext",
 "kms:Decrypt",
 "kms:ReEncrypt*"
],
 "Resource" : "arn:aws:kms:region:111122223333:key/key_ID",
 "Condition" : {
 "StringEquals" : {
 "kms:ViaService" : "iotmanagedintegrations.amazonaws.com"
 },
 "ForAnyValue:StringEquals": {
 "kms:EncryptionContext:aws-crypto-ec:iotmanagedintegrations": "111122223333"
 },
 "ArnLike": {
 "aws:SourceArn": [
 "arn:aws:iotmanagedintegrations:<region>:<accountId>:managed-thing/
<managedThingId>",
 "arn:aws:iotmanagedintegrations:<region>:<accountId>:credential-locker/
<credentialLockerId>",
 "arn:aws:iotmanagedintegrations:<region>:<accountId>:provisioning-profile/
<provisioningProfileId>",
 "arn:aws:iotmanagedintegrations:<region>:<accountId>:ota-task/<otaTaskId>"
]
 }
 }
 },
 {
 "Sid" : "Allow access to principals authorized to use managed integrations for
async flow",
 "Effect" : "Allow",
```

```

 "Principal" : {
 "Service": "iotmanagedintegrations.amazonaws.com"
 },
 "Action" : [
 "kms:GenerateDataKeyWithoutPlaintext",
 "kms:Decrypt",
 "kms:ReEncrypt*"
],
 "Resource" : "arn:aws:kms:region:111122223333:key/key_ID",
 "Condition" : {
 "ForAnyValue:StringEquals": {
 "kms:EncryptionContext:aws-crypto-ec:iotmanagedintegrations": "111122223333"
 },
 "ArnLike": {
 "aws:SourceArn": [
 "arn:aws:iotmanagedintegrations:<region>:<accountId>:managed-thing/
<managedThingId>",
 "arn:aws:iotmanagedintegrations:<region>:<accountId>:credential-locker/
<credentialLockerId>",
 "arn:aws:iotmanagedintegrations:<region>:<accountId>:provisioning-profile/
<provisioningProfileId>",
 "arn:aws:iotmanagedintegrations:<region>:<accountId>:ota-task/<otaTaskId>"
]
 }
 }
 },
 {
 "Sid" : "Allow access to principals authorized to use managed integrations for
describe key",
 "Effect" : "Allow",
 "Principal" : {
 "AWS": "arn:aws:iam::111122223333:user/username"
 },
 "Action" : [
 "kms:DescribeKey",
],
 "Resource" : "arn:aws:kms:region:111122223333:key/key_ID",
 "Condition" : {
 "StringEquals" : {
 "kms:ViaService" : "iotmanagedintegrations.amazonaws.com"
 }
 }
 },
 {

```

```
"Sid": "Allow access for key administrators",
"Effect": "Allow",
"Principal": {
 "AWS": "arn:aws:iam::111122223333:root"
},
"Action" : [
 "kms:*"
],
"Resource": "*"
}
]
}
```

有关密钥库的更多信息，请参阅[密钥库](#)。

## 更新加密配置

无缝更新加密配置的能力对于管理托管集成的数据加密实施至关重要。当您最初使用托管集成时，系统将提示您选择加密配置。您可以选择默认 AWS 拥有的密钥或创建自己的密 AWS KMS 钥。

### AWS 管理控制台

要在中更新您的加密配置 AWS 管理控制台，请打开 AWS IoT 服务主页，然后导航到统一控制的托管集成 > 设置 > 加密。在加密设置窗口中，您可以通过选择新的密 AWS KMS 钥来更新您的加密配置，以获得额外的加密保护。选择“自定义加密设置（高级）”以选择现有 AWS KMS 密钥，也可以选择“创建 AWS KMS 密钥”来创建自己的客户托管密钥。

### API 命令

在托管集成中，有两种方法 APIs 用于管理 AWS KMS 密钥的加密配置：PutDefaultEncryptionConfiguration 和 GetDefaultEncryptionConfiguration。

要更新默认加密配置，请调用 PutDefaultEncryptionConfiguration。有关 PutDefaultEncryptionConfiguration 的更多信息，请参阅[PutDefaultEncryptionConfiguration](#)。

要查看默认加密配置，请致电 GetDefaultEncryptionConfiguration。有关 GetDefaultEncryptionConfiguration 的更多信息，请参阅[GetDefaultEncryptionConfiguration](#)。

# 托管集成的身份和访问管理

AWS Identity and Access Management (IAM) AWS 服务 可帮助管理员安全地控制对 AWS 资源的访问权限。IAM 管理员控制谁可以进行身份验证 ( 登录 ) 和授权 ( 有权限 ) 使用托管集成资源。您可以使用 IAM AWS 服务 ，无需支付额外费用。

## 主题

- [受众](#)
- [使用身份进行身份验证](#)
- [使用策略管理访问](#)
- [AWS 托管集成的托管策略](#)
- [托管集成如何与 IAM 配合使用](#)
- [托管集成的基于身份的策略示例](#)
- [对托管集成、身份和访问进行故障排除](#)
- [使用服务相关角色进行托管集成](#)

## 受众

您的使用方式 AWS Identity and Access Management (IAM) 因您的角色而异：

- 服务用户：如果您无法访问功能，请向管理员申请权限 ( 请参阅[对托管集成、身份和访问进行故障排除](#) )
- 服务管理员 - 确定用户访问权限并提交权限请求 ( 请参阅 [托管集成如何与 IAM 配合使用](#) )
- IAM 管理员 - 编写用于管理访问权限的策略 ( 请参阅 [托管集成的基于身份的策略示例](#) )

## 使用身份进行身份验证

身份验证是您 AWS 使用身份凭证登录的方式。您必须以 IAM 用户身份进行身份验证 AWS 账户根用户，或者通过担任 IAM 角色进行身份验证。

您可以使用来自身份源的证书 AWS IAM Identity Center ( 例如 ( IAM Identity Center ) )、单点登录身份验证或 Google/Facebook 证书，以联合身份登录。有关登录的更多信息，请参阅《AWS 登录 用户指南》中的[如何登录您的 AWS 账户](#)。

对于编程访问，AWS 提供 SDK 和 CLI 来对请求进行加密签名。有关更多信息，请参阅《IAM 用户指南》中的[适用于 API 请求的AWS 签名版本 4](#)。

## AWS 账户 root 用户

创建时 AWS 账户，首先会有一个名为 AWS 账户 root 用户的登录身份，该身份可以完全访问所有资源 AWS 服务和资源。我们强烈建议不要使用根用户进行日常任务。有关需要根用户凭证的任务，请参阅《IAM 用户指南》中的[需要根用户凭证的任务](#)。

## 联合身份

作为最佳实践，要求人类用户使用与身份提供商的联合身份验证才能 AWS 服务 使用临时证书进行访问。

联合身份是指来自您的企业目录、Web 身份提供商的用户 Directory Service ，或者 AWS 服务 使用来自身份源的凭据进行访问的用户。联合身份代入可提供临时凭证的角色。

要集中管理访问权限，建议使用。AWS IAM Identity Center 有关更多信息，请参阅《AWS IAM Identity Center 用户指南》中的[什么是 IAM Identity Center ?](#)。

## IAM 用户和群组

[IAM 用户](#)是对单个人员或应用程序具有特定权限的一个身份。建议使用临时凭证，而非具有长期凭证的 IAM 用户。有关更多信息，请参阅 IAM 用户指南中的[要求人类用户使用身份提供商的联合身份验证才能 AWS 使用临时证书进行访问](#)。

[IAM 组](#)指定一组 IAM 用户，便于更轻松地对大量用户进行权限管理。有关更多信息，请参阅《IAM 用户指南》中的[IAM 用户的使用案例](#)。

## IAM 角色

[IAM 角色](#)是具有特定权限的身份，可提供临时凭证。您可以通过[从用户切换到 IAM 角色 \(控制台\)](#) 或调用 AWS CLI 或 AWS API 操作来代入角色。有关更多信息，请参阅《IAM 用户指南》中的[担任角色的方法](#)。

IAM 角色对于联合用户访问、临时 IAM 用户权限、跨账户访问、跨服务访问以及在 Amazon 上运行的应用程序非常有用。EC2 有关更多信息，请参阅《IAM 用户指南》中的[IAM 中的跨账户资源访问](#)。

## 使用策略管理访问

您可以 AWS 通过创建策略并将其附加到 AWS 身份或资源来控制中的访问权限。策略定义了与身份或资源关联时的权限。AWS 在委托人提出请求时评估这些政策。大多数策略都以 JSON 文档的 AWS 形式存储在中。有关 JSON 策略文档的更多信息，请参阅《IAM 用户指南》中的[JSON 策略概述](#)。

管理员使用策略，通过定义哪个主体可以对什么资源以及在什么条件下执行操作，来指定谁有权访问什么内容。

默认情况下，用户和角色没有权限。IAM 管理员创建 IAM 策略并将其添加到角色中，然后用户可以代入这些角色。IAM 策略定义权限，而不考虑您使用哪种方法来执行操作。

## 基于身份的策略

基于身份的策略是您附加到身份（用户、组或角色）的 JSON 权限策略文档。这些策略控制身份可在何种条件下对哪些资源执行什么操作。要了解如何创建基于身份的策略，请参阅《IAM 用户指南》中的[使用客户管理型策略定义自定义 IAM 权限](#)。

基于身份的策略可以是内联策略（直接嵌入到单个身份中）或托管策略（附加到多个身份的独立策略）。要了解如何在托管策略和内联策略之间进行选择，请参阅《IAM 用户指南》中的[在托管策略与内联策略之间进行选择](#)。

## 基于资源的策略

基于资源的策略是附加到资源的 JSON 策略文档。示例包括 IAM 角色信任策略和 Amazon S3 存储桶策略。在支持基于资源的策略的服务中，服务管理员可以使用它们来控制对特定资源的访问。您必须在基于资源的策略中[指定主体](#)。

基于资源的策略是位于该服务中的内联策略。您不能在基于资源的策略中使用 IAM 中的 AWS 托管策略。

## 其他策略类型

AWS 支持其他策略类型，这些策略类型可以设置更常见的策略类型授予的最大权限：

- 权限边界 – 设置基于身份的策略可以授予 IAM 实体的最大权限。有关更多信息，请参阅《IAM 用户指南》中的[IAM 实体的权限边界](#)。
- 服务控制策略 (SCPs)-在中指定组织或组织单位的最大权限 AWS Organizations。有关更多信息，请参阅《AWS Organizations 用户指南》中的[服务控制策略](#)。
- 资源控制策略 (RCPs)-设置账户中资源的最大可用权限。有关更多信息，请参阅《AWS Organizations 用户指南》中的[资源控制策略 \(RCPs\)](#)。
- 会话策略 – 在为角色或联合用户创建临时会话时，作为参数传递的高级策略。有关更多信息，请参阅《IAM 用户指南》中的[会话策略](#)。

## 多个策略类型

当多个类型的策略应用于一个请求时，生成的权限更加复杂和难以理解。要了解在涉及多种策略类型时如何 AWS 确定是否允许请求，请参阅 IAM 用户指南中的[策略评估逻辑](#)。

## AWS 托管集成的托管策略

要向用户、群组和角色添加权限，使用 AWS 托管策略比自己编写策略要容易得多。创建仅为团队提供所需权限的 [IAM 客户管理型策略](#) 需要时间和专业知识。要快速入门，您可以使用我们的 AWS 托管策略。这些策略涵盖常见使用案例，可在您的 AWS 账户中使用。有关 AWS 托管策略的更多信息，请参阅 IAM 用户指南中的[AWS 托管策略](#)。

AWS 服务维护和更新 AWS 托管策略。您无法更改 AWS 托管策略中的权限。服务偶尔会向 AWS 托管策略添加额外权限以支持新特征。此类更新会影响附加策略的所有身份（用户、组和角色）。当启动新特征或新操作可用时，服务最有可能更新 AWS 托管策略。服务不会从 AWS 托管策略中移除权限，因此策略更新不会破坏您的现有权限。

此外，还 AWS 支持跨多个服务的工作职能的托管策略。例如，ReadOnlyAccess AWS 托管策略提供对所有 AWS 服务和资源的只读访问权限。当服务启动一项新功能时，AWS 会为新操作和资源添加只读权限。有关工作职能策略的列表和说明，请参阅 IAM 用户指南中的[适用于工作职能的 AWS 托管策略](#)。

### AWS 托管策略：AWSIoTManagedIntegrationsFullAccess

您可以将 AWSIoTManagedIntegrationsFullAccess 策略附加到 IAM 身份。

此策略授予对托管集成和相关服务的完全访问权限。要在中查看此政策 AWS 管理控制台，请参阅[AWSIoTManagedIntegrationsFullAccess](#)。

#### 权限详细信息

该策略包含以下权限：

- `iotmanagedintegrations`— 向您添加此策略的 IAM 用户、群组和角色提供对托管集成和相关服务的完全访问权限。

- iam— 允许分配的 IAM 用户、群组和角色在中创建 AWS 账户服务相关角色。

## JSON

```
{
 "Version": "2012-10-17",
 "Statement": [
 {
 "Effect": "Allow",
 "Action": "iotmanagedintegrations:*",
 "Resource": "*"
 },
 {
 "Effect": "Allow",
 "Action": "iam:CreateServiceLinkedRole",
 "Resource": "arn:aws:iam::*:role/aws-service-role/iotmanagedintegrations.amazonaws.com/AWSServiceRoleForIoTManagedIntegrations",
 "Condition": {
 "StringEquals": {
 "iam:AWSServiceName": "iotmanagedintegrations.amazonaws.com"
 }
 }
 }
]
}
```

## AWS 托管策略：AWS IoTManaged IntegrationsRolePolicy

您可以将 AWS IoTManagedIntegrationsRolePolicy 策略附加到 IAM 身份。

该策略授予托管集成代表您发布 Amazon CloudWatch 日志和指标的权限。

要在中查看此政策 AWS 管理控制台，请参阅[AWSIoTManagedIntegrationsRolePolicy](#)。

### 权限详细信息

该策略包含以下权限。

- logs— 提供创建 Amazon CloudWatch 日志组并将日志流式传输到这些组的功能。
- cloudwatch— 提供发布 Amazon CloudWatch 指标的功能。有关亚马逊 CloudWatch 指标的更多信息，请参阅 [Amazon 中的指标 CloudWatch](#)。

## JSON

```
{
 "Version": "2012-10-17",
 "Statement": [
 {
 "Sid": "CloudWatchLogs",
 "Effect": "Allow",
 "Action": [
 "logs:CreateLogGroup"
],
 "Resource": [
 "arn:aws:logs:*:*:log-group:/aws/iotmanagedintegrations/*"
],
 "Condition": {
 "StringEquals": {
 "aws:PrincipalAccount": "${aws:ResourceAccount}"
 }
 }
 },
 {
 "Sid": "CloudWatchStreams",
 "Effect": "Allow",
 "Action": [
 "logs:CreateLogStream",
 "logs:PutLogEvents"
],
 "Resource": [
 "arn:aws:logs:*:*:log-group:/aws/iotmanagedintegrations/*:log-stream:*"
],
 "Condition": {
 "StringEquals": {
 "aws:PrincipalAccount": "${aws:ResourceAccount}"
 }
 }
 },
 {
 "Sid": "CloudWatchMetrics",
 "Effect": "Allow",
 "Action": [
 "cloudwatch:PutMetricData"
],
 "Resource": "*",
 }
]
}
```

```

 "Condition": {
 "StringEquals": {
 "cloudwatch:namespace": [
 "AWS/IoTManagedIntegrations",
 "AWS/Usage"
]
 }
 }
]
}

```

## 托管集成对托 AWS 管策略的更新

查看自该服务开始跟踪 AWS 托管集成的托管策略更新以来这些变更的详细信息。要获得有关此页面变更的自动提醒，请在托管集成文档历史记录页面上订阅 RSS feed。

| 更改          | 描述                     | 日期             |
|-------------|------------------------|----------------|
| 托管集成已开始跟踪更改 | 托管集成开始跟踪其 AWS 托管策略的更改。 | 2025 年 3 月 3 日 |

## 托管集成如何与 IAM 配合使用

在使用 IAM 管理托管集成的访问权限之前，请先了解托管集成可以使用哪些 IAM 功能。

可在托管集成中使用的 IAM 功能

| IAM 功能                  | 托管集成支持 |
|-------------------------|--------|
| <a href="#">基于身份的策略</a> | 是      |
| <a href="#">基于资源的策略</a> | 否      |
| <a href="#">策略操作</a>    | 是      |
| <a href="#">策略资源</a>    | 是      |
| <a href="#">策略条件键</a>   | 是      |

| IAM 功能                        | 托管集成支持 |
|-------------------------------|--------|
| <a href="#">ACLs</a>          | 否      |
| <a href="#">ABAC (策略中的标签)</a> | 否      |
| <a href="#">临时凭证</a>          | 是      |
| <a href="#">主体权限</a>          | 是      |
| <a href="#">服务角色</a>          | 是      |
| <a href="#">服务关联角色</a>        | 是      |

要全面了解托管集成和其他 AWS 服务如何与大多数 IAM 功能配合使用，请参阅 IAM 用户指南中的与 IAM 配合使用的 AWS [服务](#)。

## 用于托管集成的基于身份的策略

支持基于身份的策略：是

基于身份的策略是可附加到身份（如 IAM 用户、用户组或角色）的 JSON 权限策略文档。这些策略控制用户和角色可在何种条件下对哪些资源执行哪些操作。要了解如何创建基于身份的策略，请参阅《IAM 用户指南》中的[使用客户管理型策略定义自定义 IAM 权限](#)。

通过使用 IAM 基于身份的策略，您可以指定允许或拒绝的操作和资源以及允许或拒绝操作的条件。要了解可在 JSON 策略中使用的所有元素，请参阅《IAM 用户指南》中的[IAM JSON 策略元素引用](#)。

托管集成的基于身份的策略示例

要查看托管集成基于身份的策略的示例，请参阅。[托管集成的基于身份的策略示例](#)

## 托管集成中基于资源的策略

支持基于资源的策略：否

基于资源的策略是附加到资源的 JSON 策略文档。基于资源的策略的示例包括 IAM 角色信任策略和 Amazon S3 存储桶策略。在支持基于资源的策略的服务中，服务管理员可以使用它们来控制对特定资源的访问。对于在其中附加策略的资源，策略定义指定主体可以对该资源执行哪些操作以及在什么条件下执行。您必须在基于资源的策略中[指定主体](#)。委托人可以包括账户、用户、角色、联合用户或 AWS 服务。

要启用跨账户访问，您可以将整个账户或其他账户中的 IAM 实体指定为基于资源的策略中的主体。有关更多信息，请参阅《IAM 用户指南》中的 [IAM 中的跨账户资源访问](#)。

## 托管集成的政策措施

支持策略操作：是

管理员可以使用 AWS JSON 策略来指定谁有权访问什么。也就是说，哪个主体可以对什么资源执行操作，以及在什么条件下执行。

JSON 策略的 Action 元素描述可用于在策略中允许或拒绝访问的操作。在策略中包含操作以授予执行关联操作的权限。

要查看托管集成操作列表，请参阅《[服务授权参考](#)》中的[托管集成定义的操作](#)。

托管集成中的策略操作在操作前使用以下前缀：

```
iot-mi
```

要在单个语句中指定多项操作，请使用逗号将它们隔开。

```
"Action": [
 "iot-mi:action1",
 "iot-mi:action2"
]
```

要查看托管集成基于身份的策略的示例，请参阅。[托管集成的基于身份的策略示例](#)

## 托管集成的策略资源

支持策略资源：是

管理员可以使用 AWS JSON 策略来指定谁有权访问什么。也就是说，哪个主体可以对什么资源执行操作，以及在什么条件下执行。

Resource JSON 策略元素指定要向其应用操作的一个或多个对象。作为最佳实践，请使用其 [Amazon 资源名称 \(ARN\)](#) 指定资源。对于不支持资源级权限的操作，请使用通配符 (\*) 指示语句应用于所有资源。

```
"Resource": "*"
```

要查看托管集成资源类型及其列表 ARNs，请参阅《[服务授权参考](#)》中的“[托管集成定义的资源](#)”。要了解您可以使用哪些操作来指定每种资源的 ARN，请参阅[托管集成定义的操作](#)。

要查看托管集成基于身份的策略的示例，请参阅。[托管集成的基于身份的策略示例](#)

## 托管集成的策略条件密钥

支持特定于服务的策略条件键：是

管理员可以使用 AWS JSON 策略来指定谁有权访问什么。也就是说，哪个主体可以对什么资源执行操作，以及在什么条件下执行。

Condition 元素指定语句何时根据定义的标准执行。您可以创建使用[条件运算符](#)（例如，等于或小于）的条件表达式，以使策略中的条件与请求中的值相匹配。要查看所有 AWS 全局条件键，请参阅 IAM 用户指南中的[AWS 全局条件上下文密钥](#)。

要查看托管集成条件密钥列表，请参阅《[服务授权参考](#)》中的[托管集成的条件密钥](#)。要了解可以使用条件键的操作和资源，请参阅[托管集成定义的操作](#)。

要查看托管集成基于身份的策略的示例，请参阅。[托管集成的基于身份的策略示例](#)

## ACLs 在托管集成中

支持 ACLs：否

访问控制列表 (ACLs) 控制哪些委托人（账户成员、用户或角色）有权访问资源。ACLs 与基于资源的策略类似，尽管它们不使用 JSON 策略文档格式。

## 带有托管集成的 ABAC

支持 ABAC（策略中的标签）：部分支持

基于属性的访问权限控制（ABAC）是一种授权策略，该策略基于称为标签的属性来定义权限。您可以将标签附加到 IAM 实体和 AWS 资源，然后设计 ABAC 策略以允许在委托人的标签与资源上的标签匹配时进行操作。

要基于标签控制访问，您需要使用 `aws:ResourceTag/key-name`、`aws:RequestTag/key-name` 或 `aws:TagKeys` 条件键在策略的[条件元素](#)中提供标签信息。

如果某个服务对于每种资源类型都支持所有这三个条件键，则对于该服务，该值为是。如果某个服务仅对于部分资源类型支持所有这三个条件键，则该值为部分。

有关 ABAC 的更多信息，请参阅《IAM 用户指南》中的[使用 ABAC 授权定义权限](#)。要查看设置 ABAC 步骤的教程，请参阅《IAM 用户指南》中的[使用基于属性的访问权限控制 \( ABAC \)](#)。

## 在托管集成中使用临时证书

支持临时凭证：是

临时证书提供对 AWS 资源的短期访问权限，并且是在您使用联合身份或切换角色时自动创建的。AWS 建议您动态生成临时证书，而不是使用长期访问密钥。有关更多信息，请参阅《IAM 用户指南》中的[IAM 中的临时安全凭证](#)和[使用 IAM 的 AWS 服务](#)

## 托管集成的跨服务主体权限

支持转发访问会话 ( FAS )：是

转发访问会话 (FAS) 使用调用主体的权限 AWS 服务，再加上 AWS 服务 向下游服务发出请求的请求。有关发出 FAS 请求时的策略详情，请参阅[转发访问会话](#)。

## 托管集成的服务角色

支持服务角色：是

服务角色是由一项服务担任、代表您执行操作的 [IAM 角色](#)。IAM 管理员可以在 IAM 中创建、修改和删除服务角色。有关更多信息，请参阅《IAM 用户指南》中的[创建向 AWS 服务委派权限的角色](#)。

### Warning

更改服务角色的权限可能会破坏托管集成功能。只有当托管集成提供了相关指导时，才可以编辑服务角色。

## 托管集成的服务相关角色

支持服务关联角色：是

服务相关角色是一种与服务相关联的 AWS 服务服务角色。服务可以代入代表您执行操作的角色。服务相关角色出现在您的中 AWS 账户，并且归服务所有。IAM 管理员可以查看但不能编辑服务关联角色的权限。

有关创建或管理服务相关角色的详细信息，请参阅[能够与 IAM 搭配使用的 AWS 服务](#)。在表中查找服务相关角色列中包含 Yes 的表。选择是链接以查看该服务的服务相关角色文档。

## 托管集成的基于身份的策略示例

默认情况下，用户和角色无权创建或修改托管集成资源。要授予用户对所需资源执行操作的权限，IAM 管理员可以创建 IAM 策略。

要了解如何使用这些示例 JSON 策略文档创建基于 IAM 身份的策略，请参阅《IAM 用户指南》中的[创建 IAM 策略 \(控制台\)](#)。

有关托管集成定义的操作和资源类型（包括每种资源类型的格式）的详细信息，请参阅《服务授权参考》中的[“托管集成的操作、资源和条件密钥”](#)。ARNs

### 主题

- [策略最佳实践](#)
- [使用托管集成控制台](#)
- [允许用户查看他们自己的权限](#)

## 策略最佳实践

基于身份的策略决定了某人是否可以在您的账户中创建、访问或删除托管集成资源。这些操作可能会使 AWS 账户产生成本。创建或编辑基于身份的策略时，请遵循以下指南和建议：

- 开始使用 AWS 托管策略并转向最低权限权限 — 要开始向用户和工作负载授予权限，请使用为许多常见用例授予权限的 AWS 托管策略。它们在你的版本中可用 AWS 账户。我们建议您通过定义针对您的用例的 AWS 客户托管策略来进一步减少权限。有关更多信息，请参阅《IAM 用户指南》中的[AWS 托管式策略](#)或[工作职能的 AWS 托管式策略](#)。
- 应用最低权限：在使用 IAM 策略设置权限时，请仅授予执行任务所需的权限。为此，您可以定义在特定条件下可以对特定资源执行的操作，也称为最低权限许可。有关使用 IAM 应用权限的更多信息，请参阅《IAM 用户指南》中的[IAM 中的策略和权限](#)。
- 使用 IAM 策略中的条件进一步限制访问权限：您可以向策略添加条件来限制对操作和资源的访问。例如，您可以编写策略条件来指定必须使用 SSL 发送所有请求。如果服务操作是通过特定的方式使用的，则也可以使用条件来授予对服务操作的访问权限 AWS 服务，例如 CloudFormation。有关更多信息，请参阅《IAM 用户指南》中的[IAM JSON 策略元素：条件](#)。
- 使用 IAM Access Analyzer 验证您的 IAM 策略，以确保权限的安全性和功能性：IAM Access Analyzer 会验证新策略和现有策略，以确保策略符合 IAM 策略语言（JSON）和 IAM 最佳实

践。IAM Access Analyzer 提供 100 多项策略检查和可操作的建议，以帮助您制定安全且功能性强的策略。有关更多信息，请参阅《IAM 用户指南》中的[使用 IAM Access Analyzer 验证策略](#)。

- 需要多重身份验证 (MFA)-如果 AWS 账户您的场景需要 IAM 用户或根用户，请启用 MFA 以提高安全性。若要在调用 API 操作时需要 MFA，请将 MFA 条件添加到您的策略中。有关更多信息，请参阅《IAM 用户指南》中的[使用 MFA 保护 API 访问](#)。

有关 IAM 中的最佳实操的更多信息，请参阅《IAM 用户指南》中的[IAM 中的安全最佳实践](#)。

## 使用托管集成控制台

要访问托管集成控制台，您必须拥有一组最低权限。这些权限必须允许您列出和查看有关您的 AWS 账户托管集成资源的详细信息。如果创建比必需的最低权限更为严格的基于身份的策略，对于附加了该策略的实体（用户或角色），控制台将无法按预期正常运行。

对于仅调用 AWS CLI 或 AWS API 的用户，您无需为其设置最低控制台权限。相反，只允许访问与其尝试执行的 API 操作相匹配的操作。

为确保用户和角色仍然可以使用托管集成控制台，还需要将托管集成 *ConsoleAccess* 或 *ReadOnly* AWS 管策略附加到实体。有关更多信息，请参阅《IAM 用户指南》中的[为用户添加权限](#)。

## 允许用户查看他们自己的权限

该示例说明了您如何创建策略，以允许 IAM 用户查看附加到其用户身份的内联和托管式策略。此策略包括在控制台上或使用 AWS CLI 或 AWS API 以编程方式完成此操作的权限。

```
{
 "Version": "2012-10-17",
 "Statement": [
 {
 "Sid": "ViewOwnUserInfo",
 "Effect": "Allow",
 "Action": [
 "iam:GetUserPolicy",
 "iam:ListGroupsWithUser",
 "iam:ListAttachedUserPolicies",
 "iam:ListUserPolicies",
 "iam:GetUser"
],
 "Resource": ["arn:aws:iam::*:user/${aws:username}"]
 },
],
}
```

```
{
 "Sid": "NavigateInConsole",
 "Effect": "Allow",
 "Action": [
 "iam:GetGroupPolicy",
 "iam:GetPolicyVersion",
 "iam:GetPolicy",
 "iam:ListAttachedGroupPolicies",
 "iam:ListGroupPolicies",
 "iam:ListPolicyVersions",
 "iam:ListPolicies",
 "iam:ListUsers"
],
 "Resource": "*"
}
```

## 对托管集成、身份和访问进行故障排除

使用以下信息来帮助您诊断和修复在使用托管集成和 IAM 时可能遇到的常见问题。

### 主题

- [我无权在托管集成中执行任何操作](#)
- [我无权执行 iam : PassRole](#)
- [我想允许我以外的人 AWS 账户 访问我的托管集成资源](#)

### 我无权在托管集成中执行任何操作

如果您收到错误提示，指明您无权执行某个操作，则必须更新策略以允许执行该操作。

当 mateojackson IAM 用户尝试使用控制台查看有关虚构 *my-example-widget* 资源的详细信息，但不拥有虚构 *iot-mi:GetWidget* 权限时，会发生以下示例错误。

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform: iot-mi:GetWidget on resource: my-example-widget
```

在此情况下，必须更新 mateojackson 用户的策略，以允许使用 `iot-mi:GetWidget` 操作访问 `my-example-widget` 资源。

如果您需要帮助，请联系您的 AWS 管理员。您的管理员是提供登录凭证的人。

## 我无权执行 iam : PassRole

如果您收到错误消息，指出您无权执行该 `iam:PassRole` 操作，则必须更新您的策略以允许您将角色传递给托管集成。

有些 AWS 服务 允许您将现有角色传递给该服务，而不是创建新的服务角色或服务相关角色。为此，您必须具有将角色传递到服务的权限。

当名为的 IAM 用户 `marymajor` 尝试使用控制台在托管集成中执行操作时，会出现以下示例错误。但是，服务必须具有服务角色所授予的权限才可执行此操作。Mary 不具有将角色传递到服务的权限。

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
iam:PassRole
```

在这种情况下，必须更新 Mary 的策略以允许她执行 `iam:PassRole` 操作。

如果您需要帮助，请联系您的 AWS 管理员。您的管理员是提供登录凭证的人。

## 我想允许我以外的人 AWS 账户 访问我的托管集成资源

您可以创建一个角色，以便其他账户中的用户或您组织外的人员可以使用该角色来访问您的资源。您可以指定谁值得信赖，可以代入角色。对于支持基于资源的策略或访问控制列表 (ACLs) 的服务，您可以使用这些策略向人们授予访问您的资源的权限。

要了解更多信息，请参阅以下内容：

- 要了解托管集成是否支持这些功能，请参阅[托管集成如何与 IAM 配合使用](#)。
- 要了解如何提供对您拥有的资源的访问权限 AWS 账户，请参阅[IAM 用户指南中的向您拥有 AWS 账户的另一个 IAM 用户提供访问权限](#)。
- 要了解如何向第三方提供对您的资源的访问权限 AWS 账户，请参阅[IAM 用户指南中的向第三方提供访问权限](#)。AWS 账户
- 要了解如何通过身份联合验证提供访问权限，请参阅《IAM 用户指南》中的[为经过外部身份验证的用户（身份联合验证）提供访问权限](#)。
- 要了解使用角色和基于资源的策略进行跨账户访问之间的差别，请参阅《IAM 用户指南》中的[IAM 中的跨账户资源访问](#)。

## 使用服务相关角色进行托管集成

AWS IoT 设备管理用户 AWS Identity and Access Management (IAM) [服务相关](#)角色的托管集成。服务相关角色是一种独特的 IAM 角色，直接链接到托管集成。服务相关角色由托管集成预定义，包括该服务代表您调用其他 AWS 服务所需的所有权限。

服务相关角色可以更轻松地设置托管集成，因为您不必手动添加必要的权限。AWS IoT 设备管理的托管集成定义了其服务相关角色的权限，除非另有定义，否则只有托管集成才能扮演其角色。定义的权限包括信任策略和权限策略，以及不能附加到任何其他 IAM 实体的权限策略。

只有在首先删除相关资源后，您才能删除服务关联角色。这可以保护您的托管集成资源，因为您不会无意中移除对资源的访问权限。

有关支持服务相关角色的其他服务的信息，请参阅与 [IAM 配合使用的 AWS 服务](#)，并在服务相关角色列表中查找标有“是”的服务。选择是和链接，查看该服务的服务关联角色文档。

### 托管集成的服务相关角色权限

AWS IoT 设备管理托管集成使用名为 `Inte AWSServiceRoleForIoTManagedgrat ions` 的服务相关角色——为 AWS IoT 设备管理提供托管集成，允许您发布日志和指标。

`AWSServiceRoleForIoTManaged`集成服务相关角色信任以下服务来代替该角色：

- `iotmanagedintegrations.amazonaws.com`

名为的角色权限策略 `AWSIoTManagedIntegrationsServiceRolePolicy` 允许托管集成对指定资源完成以下操作：

- 操作：`on all of your managed integrations resources.` 上的  
`logs:CreateLogGroup`, `logs:DescribeLogGroups`, `logs:CreateLogStream`,  
`logs:PutLogEvents`, `logs:DescribeLogStreams`, `cloudwatch:PutMetricData`

### JSON

```
{
 "Version": "2012-10-17",
 "Statement": [
 {
 "Sid": "CloudWatchLogs",
 "Effect": "Allow",
```

```
"Action" : [
 "logs:CreateLogGroup",
 "logs:DescribeLogGroups"
],
"Resource" : [
 "arn:aws:logs:*:*:log-group:/aws/iotmanagedintegrations/*"
]
},
{
 "Sid" : "CloudWatchStreams",
 "Effect" : "Allow",
 "Action" : [
 "logs:CreateLogStream",
 "logs:PutLogEvents",
 "logs:DescribeLogStreams"
],
 "Resource" : [
 "arn:aws:logs:*:*:log-group:/aws/iotmanagedintegrations/*:log-stream:*"
]
},
{
 "Sid" : "CloudWatchMetrics",
 "Effect" : "Allow",
 "Action" : [
 "cloudwatch:PutMetricData"
],
 "Resource" : "*",
 "Condition" : {
 "StringEquals" : {
 "cloudwatch:namespace" : [
 "AWS/IoTManagedIntegrations",
 "AWS/Usage"
]
 }
 }
}
]
}
```

您必须配置使用户、组或角色能够创建、编辑或删除服务相关角色的权限。有关更多信息，请参阅《IAM 用户指南》中的[服务相关角色权限](#)。

## 为托管集成创建服务相关角色

您无需手动创建服务关联角色。当您创建事件类型（例如在PutRuntimeLogConfiguration、或 API 中调用CreateEventLogConfiguration AWS 管理控制台、或 RegisterCustomEndpoint AP AWS I 命令）时，托管集成会为您创建服务相关角色。AWS CLI有关PutRuntimeLogConfiguration、CreateEventLogConfiguration或的更多信息RegisterCustomEndpoint，请参阅[PutRuntimeLogConfigurationCreateEventLogConfiguration](#)、或[RegisterCustomEndpoint](#)。

如果您删除该服务关联角色，然后需要再次创建，您可以使用相同流程在账户中重新创建此角色。当您创建事件类型（例如调用PutRuntimeLogConfigurationCreateEventLogConfiguration、或 RegisterCustomEndpoint API 命令）时，托管集成会再次为您创建服务相关角色。或者，您可以通过以下方式联系 AWS 客户支持 AWS Support Center Console。有关 AWS 支持计划的更多信息，请参阅[比较 AWS Support 计划](#)。

您还可以使用 IAM 控制台通过 IoT ManagedIntegrations -托管角色用例创建服务相关角色。在 AWS CLI 或 AWS API 中，使用服务名称创建服务相关角色。iotmanagedintegrations.amazonaws.com有关更多信息，请参阅 IAM 用户指南 中的[创建服务相关角色](#)。如果您删除了此服务相关角色，可以使用同样的过程再次创建角色。

## 编辑托管集成的服务相关角色

托管集成不允许您编辑 AWSServiceRoleForIoTManaged集成服务相关角色。创建服务关联角色后，您将无法更改角色的名称，因为可能有多种实体引用该角色。但是可以使用 IAM 编辑角色描述。有关更多信息，请参阅《IAM 用户指南》中的[编辑服务关联角色](#)。

## 删除托管集成的服务相关角色

如果不再需要使用某个需要服务关联角色的功能或服务，我们建议您删除该角色。这样就没有未被主动监控或维护的未使用实体。但是，必须先清除服务相关角色的资源，然后才能手动删除它。

### Note

如果您尝试删除资源时，托管集成正在使用该角色，则删除可能会失败。如果发生这种情况，请等待几分钟后重试。

## 使用 IAM 手动删除服务关联角色

使用 IAM 控制台 AWS CLI、或 AWS API 删除 AWSServiceRoleForIoTManaged 集成服务相关角色。有关更多信息，请参阅《IAM 用户指南》中的[删除服务相关角色](#)。

## 托管集成服务相关角色支持的区域

AWS IoT 设备管理托管集成支持在提供服务的所有区域中使用服务相关角色。有关更多信息，请参阅[AWS 区域和端点](#)。

## 用 AWS Secrets Manager 于 C2C workflows 的数据保护

AWS Secrets Manager 是一项密钥存储服务，可用于保护数据库凭据、API 密钥和其他机密信息。然后，您可以将硬编码凭证改为对 Secrets Manager 进行 API 调用。这有助于确保别人在检查您的代码时不会泄露密钥，因为代码中没有密钥。有关概述，请参阅《AWS Secrets Manager 用户指南》<https://docs.aws.amazon.com/secretsmanager/latest/userguide/intro.html>。

Secrets Manager 使用密 AWS Key Management Service 钥对机密进行加密。有关更多信息，请参阅[Secret encryption and decryption in AWS Key Management Service](#)。

的托管 AWS IoT Device Management 集成与集成，AWS Secrets Manager 因此您可以将数据存储在 Secrets Manager 中，并在配置中使用密钥 ID。

## 托管集成如何使用机密

Open Authorization (OAuth) 是委托访问授权的开放标准，允许用户在不共享密码的情况下授予网站或应用程序访问其在其他网站上的信息的权限。这是第三方应用程序代表用户访问用户数据的安全方式，为共享密码提供了一种更安全的替代方案。

在中 OAuth，客户端 ID 和客户端密钥是在客户端应用程序请求访问令牌时对其进行识别和身份验证的凭证。

托管集成，用于 OAuth 与 AWS IoT Device Management 使用 C2C workflows 的客户进行沟通。客户需要提供客户端 ID 和客户端密钥才能进行通信。托管集成客户将在其 AWS 账户中存储客户端 ID 和客户端密钥，托管集成会读取我们客户账户中的客户端 ID 和客户端机密。

## 如何创建密钥

要创建密钥，请按照《AWS Secrets Manager 用户指南》中[创建 AWS Secrets Manager 密钥](#)中的步骤进行操作。

您必须使用客户管理的密钥创建您的密 AWS KMS 钥，以便进行托管集成，才能读取密钥值。有关更多信息，请参阅[《AWS Secrets Manager 用户指南》中的 AWS KMS 密钥权限](#)。

您还必须使用下一节中的 IAM 策略。

## 授予托管集成的访问权限 AWS IoT Device Management 以检索密钥

要允许托管集成从 Secrets Manager 检索密钥值，请在创建密钥时在密钥的资源策略中加入以下权限。

JSON

```
{
 "Version": "2012-10-17",
 "Statement": [{
 "Effect": "Allow",
 "Principal": {
 "Service": "iotmanagedintegrations.amazonaws.com"
 },
 "Action": ["secretsmanager:GetSecretValue"],
 "Resource": "*",
 "Condition": {
 "StringEquals": {
 "aws:SourceArn": "arn:aws:iotmanagedintegrations:AWS Region:account-
id:account-association:account-association-id"
 }
 }
 }]
}
```

将以下声明添加到您的客户管理 AWS KMS 密钥的策略中。

JSON

```
{
 "Version": "2012-10-17",
 "Statement": [{
 "Effect": "Allow",
 "Action": [
 "kms:Decrypt",
 "kms:DescribeKey"
],
 }]
}
```

```
 "Principal": {
 "Service": [
 "iotmanagedintegrations.amazonaws.com"
]
 },
 "Resource": [
 "arn:aws:kms:us-east-1:123456789012:key/*"
]
 }
}
```

## 托管集成的合规性验证

要了解是否属于特定合规计划的范围，请参阅AWS 服务 [“按合规计划划分的范围”](#)，然后选择您感兴趣的合规计划。AWS 服务 有关一般信息，请参阅[AWS 合规计划AWS](#)。

您可以使用下载第三方审计报告 AWS Artifact。有关更多信息，请参阅中的 [“下载报告”](#) 中的 [“AWS Artifact”](#)。

您在使用 AWS 服务 时的合规责任取决于您的数据的敏感性、贵公司的合规目标以及适用的法律和法规。有关您在使用时的合规责任的更多信息 AWS 服务，请参阅[AWS 安全文档](#)。

## 使用与接口 VPC 终端节点的托管集成

您可以通过创建接口 Amazon VPC 终端节点在您的 Amazon VPC 和 AWS IoT 托管集成之间建立私有连接。接口端点由一项技术提供支持 AWS PrivateLink，该技术使您能够使用私有 IP 地址私密访问服务。AW PrivateLink S 将您的 VPC 和物联网托管集成之间的所有网络流量限制到亚马逊网络。您不需要互联网网关、NAT 设备或 VPN 连接。

您无需使用 AWS PrivateLink，但建议使用。有关 AWS PrivateLink 和 VPC 终端节点的更多信息，请参阅[AWS PrivateLink 指南 AWS PrivateLink](#)中的[通过访问 AWS 服务](#)。

### 主题

- [AWS IoT 托管集成 VPC 终端节点的注意事项](#)
- [为 AWS IoT 托管集成创建接口 VPC 终端节点](#)
- [测试您的 VPC 终端节点](#)
- [控制通过 VPC 终端节点访问服务](#)

- [定价](#)
- [限制](#)

## AWS IoT 托管集成 VPC 终端节点的注意事项

在为 AWS IoT 托管集成设置接口 VPC 终端节点之前，请查看 AWS PrivateLink 指南中的[接口终端节点属性和限制](#)。

AWS IoT 托管集成支持通过接口 VPC 终端节点从您的 VPC 调用其所有 API 操作。

### 支持的终端节点

AWS IoT 托管集成支持以下服务接口的 VPC 终端节点：

- 控制平面 API：`com.amazonaws.region.iotmanagedintegrations.api`

### 不支持的终端节点

以下 AWS IoT 托管集成终端节点不支持 VPC 终端节点：

- MQTT 端点：MQTT 设备通常部署在终端用户环境中，而不是部署在终端用户环境中 AWS VPCs，因此无需 AWS PrivateLink 集成。
- OAuth 回调端点：许多第三方平台不在 AWS 基础设施内运行，这降低了 AWS PrivateLink 支持 OAuth 流程的好处。

### 可用性

AWS IoT 托管集成 VPC 终端节点可在以下 AWS 区域使用：

- 加拿大 (中部) – `ca-central-1`
- 欧洲 (爱尔兰) – `eu-west-1`

随着 AWS IoT 托管集成的可用性不断扩大，将支持其他区域。

### 双栈支持

AWS IoT 托管集成 VPC 终端节点同时支持 IPv4 和 IPv6 流量。您可以使用以下 IP 地址类型创建 VPC 终端节点：

- IPv4: 为端点网络接口分配 IPv4 地址
- IPv6 : 为端点网络接口分配 IPv6 地址 ( IPv6仅需要子网 )
- Dualstack : 为端点网络接口分配 IPv4 和 IPv6 地址

## 为 AWS IoT 托管集成创建接口 VPC 终端节点

您可以使用 Amazon VPC 控制台或 AWS CLI (AWS CLI) 为 AWS IoT 托管集成服务创建 VPC 终端节点。

### 为 AWS IoT 托管集成创建接口 VPC 终端节点 ( 控制台 )

1. 在亚马逊 VPC 控制台上打开[亚马逊 VPC 控制台](#)。
2. 在导航窗格中，选择端点。
3. 选择 创建端点。
4. 对于服务类别，选择 AWS 服务。
5. 在服务名称中，选择与您的 AWS 地区对应的服务名称。例如：
  - `com.amazonaws.ca-central-1.iotmanagedintegrations.api`
  - `com.amazonaws.eu-west-1.iotmanagedintegrations.api`
6. 对于 VPC，请选择您要从中访问 AWS IoT 托管集成的 VPC。
7. 对于其他设置，默认情况下选中“启用 DNS 名称”。我们建议您保留此设置。这样可以确保对 AWS IoT 托管集成公共服务终端节点的请求解析到您的 Amazon VPC 终端节点。
8. 对于子网，选择要在其中创建端点网络接口的子网。您可为每个可用区选择一个子网。
9. 对于 IP address type ( IP 地址类型 )，可从以下选项中进行选择：
  - IPv4: 为端点网络接口分配 IPv4 地址
  - IPv6 : 为端点网络接口分配 IPv6 地址 ( 仅当所有选定的子网均为 IPv6-only 时才支持 )
  - Dualstack : 为端点网络接口分配 IPv4 和 IPv6 地址
10. 对于 Security groups ( 安全组 )，选择要与端点网络接口关联的安全组。安全组规则必须允许终端节点网络接口与您的 VPC 中与服务通信的资源之间进行通信。
11. 对于策略，选择完全访问权限以允许所有委托人通过接口端点对所有资源进行所有操作。要限制访问，请选择“自定义”并指定策略。
12. ( 可选 ) 若要添加标签，请选择 Add new tag ( 添加新标签 )，然后输入该标签的键和值。
13. 选择创建端点。

## 为物联网托管集成 (AWS CLI) 创建接口 VPC 终端节点

使用 `create-vpc-endpoint` 命令并指定 VPC ID、VPC 终端节点类型 (接口)、服务名称、将使用终端节点的子网以及与终端节点网络接口关联的安全组。

```
aws ec2 create-vpc-endpoint \
 --vpc-id vpc-12345678 \
 --route-table-ids rtb-12345678 \
 --service-name com.amazonaws.ca-central-1.iotmanagedintegrations.api \
 --vpc-endpoint-type Interface \
 --subnet-ids subnet-12345678 subnet-87654321 \
 --security-group-ids sg-12345678
```

## 测试您的 VPC 终端节点

创建 VPC 终端节点后，您可以通过从 VPC 中的 EC2 实例对 AWS IoT 托管集成发出 API 调用来测试连接。

### 先决条件

- 您的 VPC 内私有子网中的 EC2 实例
- AWS IoT 托管集成操作的相应 IAM 权限
- 允许 HTTPS 流量 (端口 443) 到达 VPC 终端节点的安全组规则

### 测试 连接

1. 在私有子网中连接到您的 Amazon EC2 实例。
2. 验证私有 DNS 名称的 DNS 解析：

```
dig api.iotmanagedintegrations.region.api.aws
```

3. 测试 HTTPS 连接：

```
curl -v https://api.iotmanagedintegrations.region.api.aws
```

4. 调用 AWS IoT 托管集成 API：

```
aws iot-managed-integrations list-destinations \
 --region region \
 --region region \
 --region region
```

```
--endpoint-url https://api.iotmanagedintegrations.region.api.aws
```

region 替换为您 AWS 所在的地区 ( 例如 , ca-central-1 ) 。

## 控制通过 VPC 终端节点访问服务

VPC 终端节点策略是您在创建或修改接口 VPC 终端节点时附加到接口 VPC 终端节点的 IAM 资源策略。如果在创建端点时未附加策略，我们将为您附加默认策略以允许对服务进行完全访问。端点策略不会覆盖或替换 IAM 用户策略或服务特定的策略。这是一个单独的策略，用于控制从端点中对指定服务进行的访问。

端点策略必须采用 JSON 格式编写。有关更多信息，请参阅 Amazon VPC 用户指南中的[使用 VPC 终端节点控制对服务的访问](#)。

### 示例：AWS IoT 托管集成操作的 VPC 终端节点策略

以下是 AWS IoT 托管集成的端点策略示例。此策略允许通过 VPC 终端节点连接到 AWS IoT 托管集成的用户访问目标，但拒绝访问凭证柜。

```
{
 "Statement": [
 {
 "Effect": "Allow",
 "Principal": "*",
 "Action": [
 "iotmanagedintegrations:ListDestinations",
 "iotmanagedintegrations:GetDestination",
 "iotmanagedintegrations:CreateDestination",
 "iotmanagedintegrations:UpdateDestination",
 "iotmanagedintegrations>DeleteDestination"
],
 "Resource": "*"
 },
 {
 "Effect": "Deny",
 "Principal": "*",
 "Action": [
 "iotmanagedintegrations:ListCredentialLockers",
 "iotmanagedintegrations:GetCredentialLocker",
 "iotmanagedintegrations:CreateCredentialLocker",
 "iotmanagedintegrations:UpdateCredentialLocker",

```

```

 "iotmanagedintegrations:DeleteCredentialLocker"
],
 "Resource": "*"
}
]
}

```

## 示例：限制访问特定 IAM 角色的 VPC 终端节点策略

以下 VPC 终端节点策略仅允许在其信任链中具有指定 IAM 角色的 IAM 委托人访问 AWS IoT 托管集成。所有其他 IAM 委托人均被拒绝访问。

```

{
 "Statement": [
 {
 "Effect": "Allow",
 "Principal": "*",
 "Action": "*",
 "Resource": "*",
 "Condition": {
 "StringEquals": {
 "aws:PrincipalArn": "arn:aws:iam::123456789012:role/IoTManagedIntegrationsVPCRole"
 }
 }
 }
]
}

```

## 定价

创建和使用带有 AWS IoT 托管集成的接口 VPC 终端节点需要按标准费率付费。有关更多信息，请参阅[AWS PrivateLink 定价](#)。

## 限制

- 该 [CreateAccountAssociation](#) API 旨在 OAuth 与第三方云服务一起运行，这需要请求离开 Amazon 网络。这对于使用 AWS PrivateLink 在 VPC 内控制流量的客户来说非常重要，因为 AWS PrivateLink 无法完全 end-to-end 控制此 API 调用。
- AWS IoT 托管集成的 VPC 终端节点在中不可用。AWS GovCloud (US) Regions

有关 VPC 终端节点的一般限制，请参阅 Amazon VPC 用户指南中的[接口终端节点属性和限制](#)。

## 连接到 AWS IoT Device Management FIPS 端点的托管集成

AWS IoT 提供了支持[联邦信息处理标准 \(FIPS\) 140-2](#) 的控制平面端点。符合 FIPS 标准的端点与标准 AWS 端点不同。要以符合 FIPS 的方式与托管集成进行 AWS IoT Device Management 交互，您必须将下述端点与兼容 FIPS 的客户端一起使用。该 AWS IoT 主机不符合 FIPS 标准。

以下各节介绍如何使用 REST API、软件开发工具包或访问符合 FIPS 标准的 AWS IoT 终端节点。

### AWS CLI

### 控制面板端点

支持托管集成操作的 FIPS 兼容控制平面端点及其相关 AWS CLI 命令列在按服务划分的[FIPS 端点](#)中。在[FIPS 按服务划分的终端节点](#)中，找到 AWS IoT Device Management 托管集成服务，然后查找您的终端节点。AWS 区域

要在访问托管集成操作时使用符合 FIPS 标准的终端节点，请使用适合您的端点的 AWS SDK 或 REST API。AWS 区域

要在运行托管集成 CLI 命令时使用符合 FIPS 的终端节点，请在命令中添加带有相应终端节点的--endpoint 参数。AWS 区域

# 监控托管集成

监控是维护托管集成和您的其他 AWS 解决方案的可靠性、可用性和性能的重要组成部分。AWS 提供以下监控工具，用于监视托管集成，在出现问题时进行报告，并在适当时自动采取措施：

- AWS CloudTrail捕获由您的账户或代表您的 AWS 账户进行的 API 调用和相关事件，并将日志文件传输到您指定的 Amazon S3 存储桶。您可以识别哪些用户和帐户拨打了电话 AWS、发出呼叫的源 IP 地址以及呼叫发生的时间。有关更多信息，请参阅 [AWS CloudTrail 用户指南](#)。

## 使用记录托管集成 API 调用 AWS CloudTrail

托管集成与[AWS CloudTrail](#)一项服务集成，该服务提供用户、角色或角色所执行操作的 AWS 服务记录。CloudTrail 将托管集成的所有 API 调用捕获为事件。捕获的调用包括来自托管集成控制台的调用以及对托管集成 API 操作的代码调用。使用收集的信息 CloudTrail，您可以确定向托管集成发出的请求、发出请求的 IP 地址、发出请求的时间以及其他详细信息。

每个事件或日志条目都包含有关生成请求的人员信息。身份信息有助于您确定以下内容：

- 请求是使用根用户凭证还是用户凭证发出的。
- 请求是否代表 IAM Identity Center 用户发出。
- 请求是使用角色还是联合用户的临时安全凭证发出的。
- 请求是否由其他 AWS 服务发出。

CloudTrail 在您创建账户 AWS 账户时在您的账户中处于活动状态，并且您自动可以访问 CloudTrail 活动历史记录。CloudTrail 事件历史记录提供了过去 90 天中记录的管理事件的可查看、可搜索、可下载且不可变的记录。AWS 区域有关更多信息，请参阅《AWS CloudTrail 用户指南》中的“[使用 CloudTrail 事件历史记录](#)”。查看活动历史记录不 CloudTrail收取任何费用。

要持续记录 AWS 账户过去 90 天内的事件，请创建跟踪或 [CloudTrailLake](#) 事件数据存储。

### CloudTrail 步道

跟踪允许 CloudTrail 将日志文件传输到 Amazon S3 存储桶。使用创建的所有跟踪 AWS 管理控制台都是多区域的。您可以通过使用 AWS CLI 创建单区域或多区域跟踪。建议创建多区域跟踪，因为您可以捕获账户 AWS 区域中的所有活动。如果您创建单区域跟踪，则只能查看跟踪的 AWS 区域中记录的事件。有关跟踪的更多信息，请参阅《AWS CloudTrail 用户指南》中的[为您的 AWS 账户创建跟踪](#)和[为组织创建跟踪](#)。

通过创建跟踪，您可以免费将正在进行的管理事件的一份副本传送到您的 Amazon S3 存储桶，但是，会收取 Amazon S3 存储费用。CloudTrail 有关 CloudTrail 定价的更多信息，请参阅[AWS CloudTrail 定价](#)。有关 Amazon S3 定价的信息，请参阅 [Amazon S3 定价](#)。

## CloudTrail 湖泊事件数据存储

CloudTrail Lake 允许你对自己的事件运行基于 SQL 的查询。CloudTrail Lake 将基于行的 JSON 格式的现有事件转换为 [Apache ORC](#) 格式。ORC 是一种针对快速检索数据进行优化的列式存储格式。事件将被聚合到事件数据存储中，它是基于您通过应用[高级事件选择器](#)选择的条件的不可变的事件集合。应用于事件数据存储的选择器用于控制哪些事件持续存在并可供您查询。有关 CloudTrail Lake 的更多信息，请参阅《AWS CloudTrail 用户指南》中的“[使用 AWS CloudTrail Lake](#)”。

CloudTrail 湖泊事件数据存储和查询会产生费用。创建事件数据存储时，您可以选择要用于事件数据存储的[定价选项](#)。定价选项决定了摄取和存储事件的成本，以及事件数据存储的默认和最长保留期。有关 CloudTrail 定价的更多信息，请参阅[AWS CloudTrail 定价](#)。

## 中的管理活动 CloudTrail

[管理事件](#)提供有关对中的资源执行的管理操作的信息 AWS 账户。这些也称为控制面板操作。默认情况下，CloudTrail 记录管理事件。

托管集成将以下托管集成控制平面操作记录 CloudTrail 为管理事件。

- CreateCloudConnector
- UpdateCloudConnector
- GetCloudConnector
- DeleteCloudConnector
- ListCloudConnectors
- CreateConnectorDestination
- UpdateConnectorDestination
- GetConnectorDestination
- DeleteConnectorDestination
- ListConnectorDestinations
- CreateAccountAssociation
- UpdateAccountAssociation

- `GetAccountAssociation`
- `DeleteAccountAssociation`
- `ListAccountAssociations`
- `StartAccountAssociationRefresh`
- `ListManagedThingAccountAssociations`
- `RegisterAccountAssociation`
- `DeregisterAccountAssociation`
- `SendConnectorEvent`
- `ListDeviceDiscoveries`
- `ListDiscoveredDevices`

## 事件示例

事件代表来自任何来源的单个请求，包括有关所请求的 API 操作、操作的日期和时间、请求参数等的信息。CloudTrail 日志文件不是公共 API 调用的有序堆栈跟踪，因此事件不会按任何特定顺序出现。

以下示例显示了一个演示 `CreateCloudConnector` API 操作成功 CloudTrail 的事件。

**CreateCloudConnector** API 操作成功 CloudTrail 事件。

```
{
 "eventVersion": "1.09",
 "userIdentity": {
 "type": "AssumedRole",
 "principalId": "EXAMPLE",
 "arn": "arn:aws:sts::111122223333:assumed-role/Admin/EXAMPLE",
 "accountId": "111122223333",
 "accessKeyId": "EXAMPLEKYSBQSCGRIC",
 "sessionContext": {
 "sessionIssuer": {
 "type": "Role",
 "principalId": "AR0AZOZQFKYSFZVB2J2GN",
 "arn": "arn:aws:iam::111122223333:role/Admin",
 "accountId": "111122223333",
 "userName": "Admin"
 },
 "attributes": {
 "creationDate": "2025-06-05T18:26:16Z",
 "mfaAuthenticated": "false"
 }
 }
 }
}
```

```
 }
 },
 "eventTime": "2025-06-05T18:30:40Z",
 "eventSource": "iotmanagedintegrations.amazonaws.com",
 "eventName": "CreateCloudConnector",
 "awsRegion": "us-east-1",
 "sourceIPAddress": "192.0.2.0",
 "userAgent": "PostmanRuntime/7.44.0",
 "requestParameters": {
 "EndpointType": "LAMBDA",
 "Description": "Manual testing for C2C CT Validation",
 "ClientToken": "abc7460",
 "EndpointConfig": {
 "lambda": {
 "arn": "arn:aws:lambda:us-
east-1:111122223333:function:LightweightMockConnector7460"
 }
 },
 "Name": "EdenManualTestCloudConnector"
 },
 "responseElements": {
 "X-Frame-Options": "DENY",
 "Access-Control-Expose-Headers": "Content-Length,Content-Type,X-Amzn-
Errortype,X-Amzn-Requestid",
 "Strict-Transport-Security": "max-age:47304000; includeSubDomains",
 "Cache-Control": "no-store, no-cache",
 "X-Content-Type-Options": "nosniff",
 "Content-Security-Policy": "upgrade-insecure-requests; default-src 'none';
object-src 'none'; frame-ancestors 'none'; base-uri 'none'",
 "Pragma": "no-cache",
 "Id": "f7e633e719404c4a933596b4d0cc276e",
 "Arn": "arn:aws:iotmanagedintegrations:us-east-1:111122223333:cloud-connector/
EXAMPLE404c4a933596b4d0cc276e"
 },
 "requestID": "c0071fd1-b8e0-400a-bcc0-EXAMPLE9e4",
 "eventID": "95b318ea-2f63-4183-9c22-EXAMPLE3e",
 "readOnly": false,
 "eventType": "AwsApiCall",
 "managementEvent": true,
 "recipientAccountId": "111122223333",
 "eventCategory": "Management"
}
```

以下示例显示了一个演示 ListDiscoveredDevices API 操作成功 CloudTrail 的事件。

**ListDiscoveredDevices** API 操作成功 CloudTrail 事件。

```
{
 "eventVersion": "1.09",
 "userIdentity": {
 "type": "AssumedRole",
 "principalId": "EZAMPLE",
 "arn": "arn:aws:sts::444455556666:assumed-role/Admin/EXAMPLE",
 "accountId": "444455556666",
 "accessKeyId": "EXAMPLERJ26PYMH",
 "sessionContext": {
 "sessionIssuer": {
 "type": "Role",
 "principalId": "EXAMPLE",
 "arn": "arn:aws:iam::444455556666:role/Admin",
 "accountId": "444455556666",
 "userName": "Admin"
 },
 "attributes": {
 "creationDate": "2025-06-10T23:37:31Z",
 "mfaAuthenticated": "false"
 }
 }
 },
 "eventTime": "2025-06-10T23:38:07Z",
 "eventSource": "iotmanagedintegrations.amazonaws.com",
 "eventName": "ListDiscoveredDevices",
 "awsRegion": "us-east-1",
 "sourceIPAddress": "192.0.2.0",
 "userAgent": "EXAMPLE-runtime/2.4.0",
 "requestParameters": {
 "Identifier": "EXAMPLE4f268483a17d8060f014"
 },
 "responseElements": null,
 "requestID": "27ae1f61-e2e6-43e4-bf17-EXAMPLEa568",
 "eventID": "34734e81-76a8-49a4-9641-EXAMPLE28ed",
 "readOnly": true,
 "eventType": "AwsApiCall",
 "managementEvent": true,
 "recipientAccountId": "444455556666",
 "eventCategory": "Management"
}
```

```
}
```

有关 CloudTrail 录音内容的信息，请参阅《AWS CloudTrail 用户指南》中的[CloudTrail 录制内容](#)。

## 托管集成的文档历史记录《开发者指南》

下表描述了托管集成的文档版本。

| 变更                     | 说明              | 日期              |
|------------------------|-----------------|-----------------|
| <a href="#">公开发布版本</a> | 托管集成开发者指南正式发布   | 2025 年 6 月 25 日 |
| <a href="#">初始预览版</a>  | 托管集成开发者指南的初始预览版 | 2025 年 3 月 3 日  |

本文属于机器翻译版本。若本译文内容与英语原文存在差异，则一律以英文原文为准。