设备接入

开发指南

文档版本 1.0

发布日期 2025-07-01

华为云计算技术有限公司

商标声明


和其他华为商标均为华为技术有限公司的商标。
本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为云计算技术有限公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为云计算技术有限公司对本文档内容不做任何明示或暗示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

目 录

1 开发前必读

方案概述

基于IoT平台实现一个物联网解决方案，需要完成以下操作：

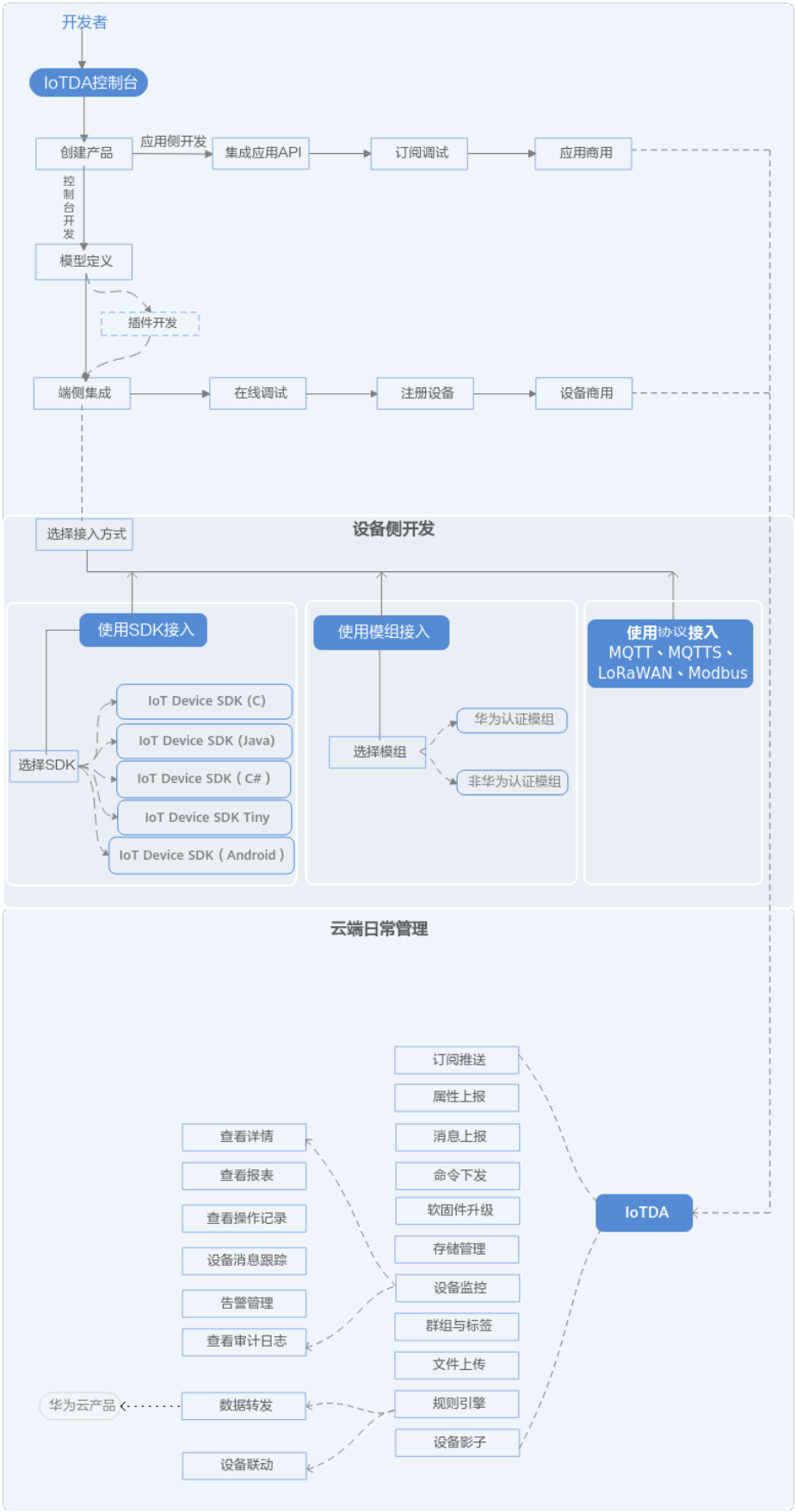
开发操作	开发说明
产品开发	主要呈现物联网平台的界面查询与操作，包括产品管理、产品模型开发、插件开发、在线调试等。
应用开发	主要为业务应用与物联网平台的集成对接开发，包括API接口调用、业务数据获取和HTTPS证书管理。
设备开发	主要为设备与物联网平台的集成对接开发，包括设备接入物联网平台、业务数据上报和对平台下发控制命令的处理。

业务概览

开通设备接入服务后，使用设备接入服务的完整流程如下图所示，主要分为产品开发、应用侧开发、设备侧开发和日常管理。

- 产品开发：开发者在进行设备接入前，基于控制台进行相应的开发工作，包括创建产品、创建设备、在线开发产品模型、在线开发插件和在线调试。
- 应用侧开发：通过API的形式对外开放物联网平台丰富的设备管理能力，应用开发人员基于API接口开发所需的行业应用，如智慧城市、智慧园区、智慧工业、车联网等行业应用，满足不同行业的需求。
- 设备侧开发：设备侧可以通过集成SDK、模组或者原生协议接入物联网平台。
- 日常管理：真实设备接入后，基于控制台或者API接口，进行日常的设备管理。

图 1-1 流程图



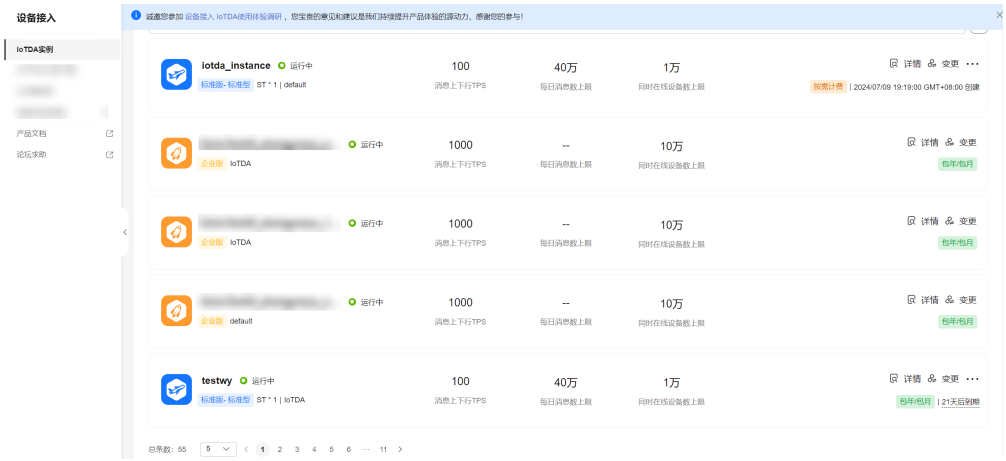
常见 FAQ

2 资源获取

平台对接信息

1. 进入IoTDA的**管理控制台**界面，选择左侧导航栏“IoTDA实例”，单击您需要的实例卡片进入实例。

图 2-1 实例管理-切换实例



2. 选择左侧导航栏“总览”页签，在选择的实例基本信息中，单击“接入信息”。

图 2-2 总览-获取接入信息

设备开发资源

物联网平台支持设备通过MQTT协议、LWM2M/CoAP协议和HTTPS协议进行接入，也可以通过IoTEdge将Modbus、OPC-UA、OPC-DA这些协议的设备接入。设备可以通过调用接口或者集成SDK的方式接入到物联网平台。

资源包名	描述	下载路径
设备侧 IoT Device Java SDK	设备可以通过集成IoT Device SDK(Java)接入物联网平台, Demo提供了调用SDK接口的样例代码。使用指导请参考 **设备侧 IoT Device Java SDK使用指南**。	**设备侧 IoT Device Java SDK**
设备侧 IoT Device C for Linux/Windows SDK	设备可以通过集成IoT Device SDK(C)接入物联网平台, Demo提供了调用SDK接口的样例代码。使用指导请参考 **设备侧 IoT Device C for Linux/Windows SDK使用指南**。	**设备侧 IoT Device C for Linux/Windows SDK**
设备侧 IoT Device C# SDK	设备可以通过集成IoT Device SDK(C#)接入物联网平台, Demo提供了调用SDK接口的样例代码。使用指导请参考 **设备侧 IoT Device C# SDK使用指南**。	**设备侧 IoT Device C# SDK**
设备侧 IoT Device Android SDK	设备可以通过集成IoT Device SDK(Android)接入物联网平台, Demo提供了调用SDK接口的样例代码。使用指导请参考 **设备侧 IoT Device Android SDK使用指南**。	**设备侧 IoT Device Android SDK**
设备侧 IoT Device Go SDK(社区版)	设备可以通过集成IoT Device SDK(Go)接入物联网平台, Demo提供了调用SDK接口的样例代码。使用指导请参考 **设备侧 IoT Device Go SDK(社区版)使用指南**。	**设备侧 IoT Device Go SDK(社区版)**

资源包名	描述	下载路径
设备侧 IoT Device Python SDK	设备可以通过集成IoT Device SDK(Python)接入物联网平台, Demo提供了调用SDK接口的样例代码。使用指导请参考**设备侧 IoT Device Python SDK使用指南**。	**设备侧 IoT Device Python SDK**
设备侧 IoT Device C for Linux/Windows SDK Tiny	设备可以通过集成IoT Device SDK Tiny (C)接入物联网平台, Demo提供了调用SDK接口的样例代码。使用指导请参考**设备侧 IoT Device C for Linux/Windows SDK Tiny使用指南**	**设备侧 IoT Device C for Linux/Windows SDK Tiny**
设备侧 IoT Device Arkts(OpenHarmony) SDK	设备可以通过集成IoT Device SDK (ArkTS)接入物联网平台, Demo提供了调用SDK接口的样例代码。使用指导请参考**设备侧 IoT Device Arkts(OpenHarmony) SDK使用指南**	**设备侧 IoT Device Arkts(OpenHarmony) SDK**
原生MQTT/MQTTS协议接入示例	设备侧可以通过原生MQTT/MQTTS协议接入到物联网平台，Demo提供了SSL加密建链和TCP不加密建链、数据上报、订阅Topic的样例代码。 接入示例参考：**Java版**、**Python版**、**Android版**、**C版**、**C#**、**NodeJS**。	**quickStart(Java)** **quickStart(Android)** **quickStart(Python)** **quickStart(C)** **quickStart(C#)** **quickStart(Node.js)**
产品模型模板	产品模型模板中包含了典型场景的产品模型样例，开发者可以在模板基础进行修改，定义自己需要的产品模型。 使用指导可以参考**离线开发产品模型**。	**产品模型开发示例**
编解码插件样例	编解码插件的代码样例工程，开发者可以基于该样例工程进行二次开发。	**编解码插件开发样例**
编解码插件检测工具	用于检测离线开发的编解码插件的编解码能力是否正常。	**编解码插件检测工具**

资源包名	描述	下载路径
NB-IoT设备模拟器	用于模拟以CoAP/LWM2M协议接入物联网平台的NB设备，实现数据上报和命令下发功能。 使用指导可以参考**基于控制台开发产品**。	**NB-IoT设备模拟器**

应用开发资源

为了降低应用的开发难度、提升开发效率，物联网平台开放了应用侧API。应用通过调用物联网平台的API，实现安全接入、设备管理、数据采集、命令下发等业务场景。

资源包名	描述	下载
应用侧开发 API Java Demo	物联网平台为应用服务器提供了**应用侧API**，能够让开发者快速验证API开放的能力，体验业务功能，熟悉业务流程。	**API Java Demo**
应用侧 Java SDK	应用侧Java SDK提供Java方法调用**应用侧API**与平台通信。使用指南可以参考**应用侧Java SDK使用指南**。	**应用侧Java SDK**
应用侧 .Net SDK	应用侧.Net SDK提供.Net方法调用**应用侧API**与平台通信。使用指南可以参考**应用侧.Net SDK使用指南**。	**应用侧.Net SDK**
应用侧 Python SDK	应用侧Python SDK提供Python方法调用**应用侧API**与平台通信。使用指南可以参考**应用侧Python SDK使用指南**。	**应用侧Python SDK**
应用侧 Go SDK	应用侧Go SDK提供Go方法调用**应用侧API**与平台通信。使用指南可以参考**应用侧Go SDK使用指南**。	**应用侧Go SDK**
应用侧 Node.js SDK	应用侧Node.js SDK提供Node.js方法调用**应用侧API**与平台通信。使用指南可以参考**应用侧Node.js SDK使用指南**。	**应用侧Node.js SDK**

资源包名	描述	下载
应用侧 PHP SDK	应用侧PHP SDK提供PHP方法调用**应用侧API**与平台通信。使用指南可以参考**应用侧PHP SDK使用指南**。	**应用侧PHP SDK**

证书资源

当设备和应用需要对IoT平台进行校验时可使用以下证书。

说明

- 此证书文件只适用于华为云物联网平台，且必须配合对应域名使用。
- CA证书具有一个过期日期，在该日期后，这些证书将无法用于验证服务器的证书；请在 CA 证书的过期日期前替换这些证书，以确保设备可以正常的连接到IoT平台。

表 2-1 证书资源

证书包名称	region&版本	证书类型	证书格式	说明	下载
certificate	中国-香港、亚太-新加坡、亚太-曼谷、亚太-雅加达、非洲-约翰内斯堡、拉美-圣地亚哥、拉美-圣保罗一、拉美-墨西哥城二、中东-利雅得	设备侧证书	pem、jks、bks	用于设备校验平台的身份。该证书必须配合当前设备侧接入域名使用。	**证书文件**
certificate	中国-香港、亚太-新加坡、亚太-曼谷、亚太-雅加达、非洲-约翰内斯堡、拉美-圣地亚哥、拉美-圣保罗一、拉美-墨西哥城二、中东-利雅得	应用侧证书	pem、jks、bks	应用接入：HTTPS/AMQPS/MQTTS 平台CA证书。	**证书文件**

3 设备侧接入开发

3.1 设备接入引导

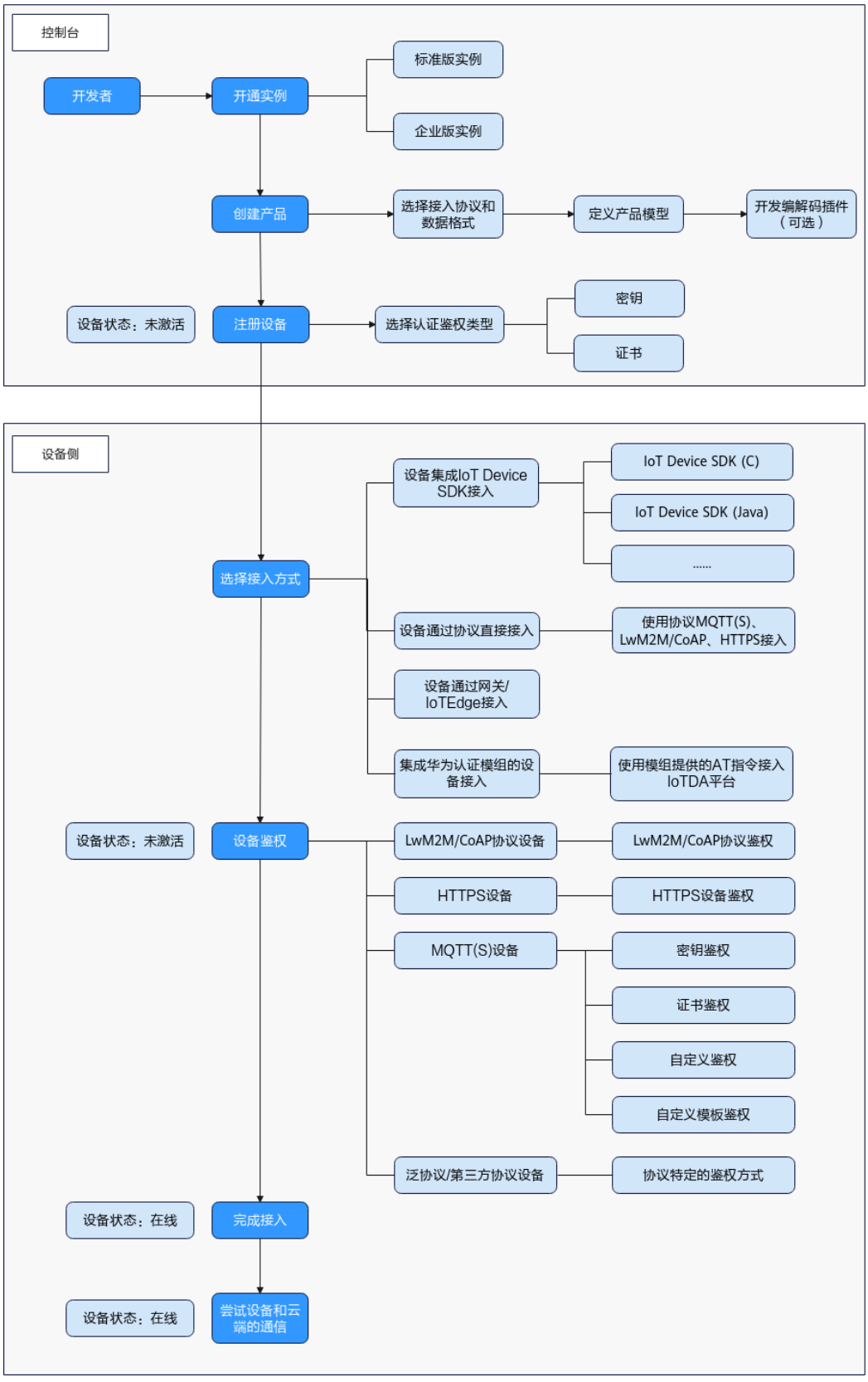
设备接入流程图

华为云IoTDA支持使用多种协议接入平台，包括：

- 常规的**原生协议直连**：MQTT(S)、HTTPS、LwM2M/CoAP(S)。
- 通过网关/IoTEdge接入的标准协议：Modbus、 OPC-UA、OPC-DA、Onvif、GB28181、LoRa等。
- 部分行业通用协议：JT808（车载终端通信协议）、SL651（水文监测数据通信协议）、HJ212（环保行业数据传输标准协议）等。
- TCP私有协议和第三方协议接入。

更多协议详情可见：**设备接入协议**。

图 3-1 设备接入开发流程图



TLS 安全传输协议

华为云IoTDA支持使用TLS进行加密通信，并支持客户端连接的安全要求。使用TLS加密接入时，客户端可以发送**SNI**（服务器名称指示），在设备建链时携带接入域名，这对于**自定义设备侧域名**、**设备自注册**、**自定义鉴权**等功能是必需的。

表 3-1 常规协议 TLS 支持类型

协议类型	支持的操作	支持的TLS版本	端口
MQTT	发布、订阅	不适用	1883
MQTTS	发布、订阅	TLSv1.1, TLSv1.2, TLSv1.3	8883
MQTT over WebSocket（wss）	发布、订阅	TLS 1.2	443
HTTPS	仅发布	TLS 1.2	443
CoAP	上报、下发	不适用	5683
CoAPS	上报、下发	DTLS 1.2	5684

设备集成设备端 SDK 接入

IoTDA提供设备侧SDK，可以通过集成SDK直接接入到平台，SDK默认实现了文件上传、下载、自动重连、OTA升级、数据上报下发、时间同步等功能，可以帮助您更简单、快捷、稳定的接入华为云平台。目前支持C、C#、Java、Android、GO、Python、ArkTS（鸿蒙生态的应用开发语言）的SDK接入，详情可见：**设备侧IoT Device SDK介绍**。

原生协议直连

设备侧接入原生支持使用MQTT(S)、HTTPS、CoAP(S)/LwM2M协议接入，对于使用这些协议接入IoTDA平台，具有以下特点，可根据您的需要确定接入方案。对于二进制格式接入的设备，需要编解码插件转换格式后接入，可以在平台部署**编解码插件**，将上报、下发数据编解码为特定数据格式，以实现在平台完成二进制数据 -> 可被识别的JSON格式的数据转换。

表 3-2 原生协议

协议类型	支持的操作	传输层	功耗	适用网络	特点	常见使用场景
MQTT(S)	上行、下行	TCP	低	不稳定/高延迟	轻量级、低功耗；使用发布/订阅模型：支持一对多通信；支持持久会话。	一般用于长连接场景，物联网行业推荐协议之一。可用于需要双向通信、设备控制或高扩展性的IoT系统（如智慧城市、车联网、能源、电力、工业4.0）。
HTTPS	仅上行	TCP	高	稳定高带宽	数据格式灵活、支持多种数据格式；单向通信，只支持客户端主动发起请求；无状态，每个请求独立。	与现有Web服务集成或需要高可读性的数据。（如APP、网页）。
CoAP(S)/LwM2M	上行、下行	UDP	极低	极低带宽/高丢包	专为受限设备设计；轻量，支持多播；低开销，使用**二进制格式**（CBOR）	资源极度受限的设备（如电池供电的传感器）或UDP-only网络。在水表、电表等资源受限的低功耗设备上应用广泛。

集成华为认证模组的设备接入

认证模组是指通过预集成**IoT Device SDK Tiny**，并且通过华为测试认证，遵循**华为指定AT命令**规范的模组。认证模组预集成了华为云SDK，可以通过AT指令一键进行数据发送接收，极大的节约设备对接工作量和设备调试周期。

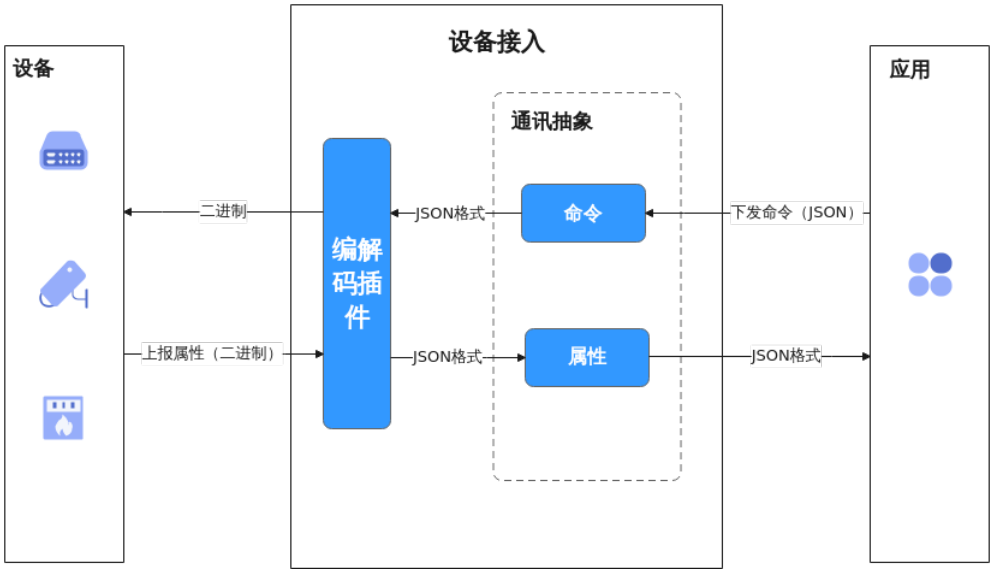
3.2 产品开发

3.2.1 产品开发指引

在物联网平台集成解决方案中，物联网平台作为承上启下的中间部分，向应用服务器开放API接口，向各种协议的设备提供API对接。为了提供更加丰富的设备管理能力，物联网平台需要理解接入设备具备的能力以及设备上报数据的格式，因此，您需要在控制台上完成产品模型和插件的开发。

- **产品模型**是用来描述设备能力的文件，通过JSON的格式定义了设备的基本属性、上报数据和下发命令的消息格式。定义产品模型，即在物联网平台构建一款设备的抽象模型，使平台理解该款设备支持的属性信息。
- **编解码插件**主要根据设备上报数据的格式来判断是否需要开发。编解码插件是供物联网平台调用，完成二进制格式和JSON格式相互转换或JSON格式之间的转换。它将设备上报的二进制数据解码为JSON格式供应用服务器“阅读”，将应用服务器下行的JSON格式命令编码为二进制或JSON格式数据供终端设备（UE）“理解执行”。以二进制与JSON转换为例，流程图如下：

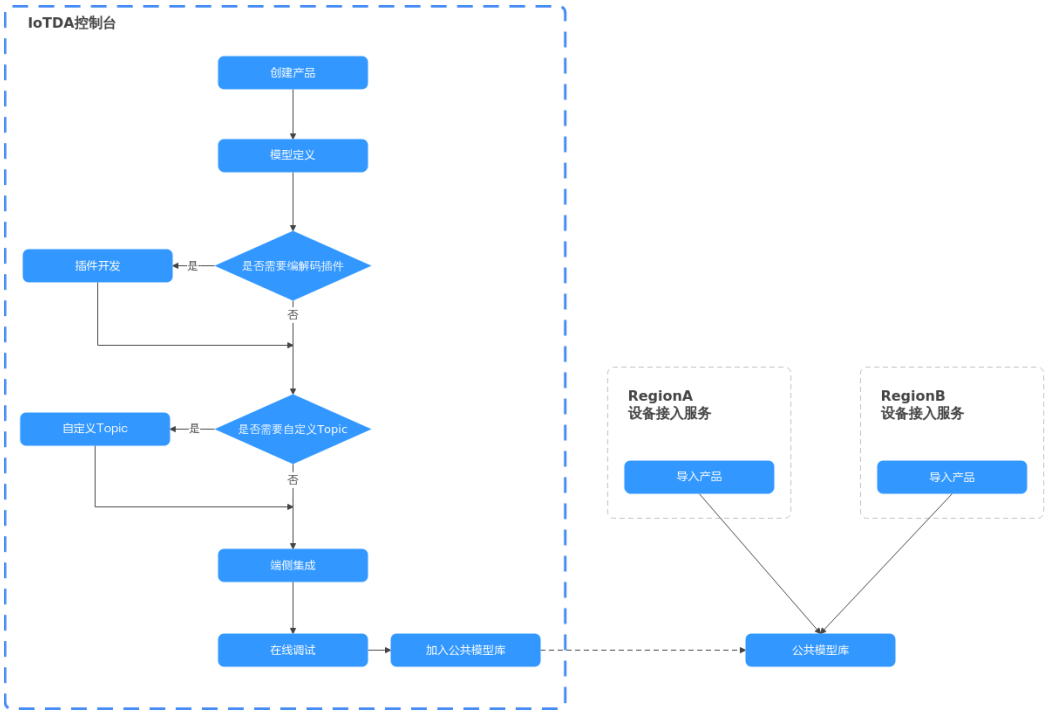
图 3-2 编解码插件流程



产品开发流程

设备接入控制台提供了可视化的界面，帮助开发者快速开发产品（产品模型、编解码插件），并进行自助测试。

图 3-3 产品开发流程图



- 创建产品：某一类具有相同能力或特征的设备的集合称为一款产品。除了设备实体，产品还包含该类设备在物联网能力建设中产生的产品信息、产品模型（Product Model）、插件等资源。

- 模型定义：即开发产品模型，产品开发最重要的是开发产品模型，产品模型用于描述设备具备的能力和特性。定义产品模型，即在物联网平台构建一款设备的抽象模型，使平台理解该款设备支持的服务、属性、命令等信息。
- 开发插件：如果设备上报的数据是二进制码流格式，就需要开发对应的插件，用于物联网平台完成二进制格式和JSON格式或JSON格式之间的转换。
- 在线调试：设备接入控制台提供了产品在线调试的功能，您可以根据自己的业务场景，在开发真实应用和真实设备之前，使用应用模拟器和设备模拟器对数据上报和命令下发等场景进行调测，也可以在真实设备开发完成后使用应用模拟器验证业务流。

说明

目前仅标准版支持MQTT协议的在线调试。

相关 FAQ

物模型相关问题

3.2.2 创建产品

在物联网平台中，某一类具有相同能力或特征的设备的合集被称为一款产品。

操作步骤

步骤1 访问设备接入服务，单击“管理控制台”进入设备接入控制台。选择您的实例，单击实例卡片进入。

步骤2 单击左侧导航栏“产品”，单击页面左侧的“创建产品”。根据页面提示填写参数，然后单击“确定”，完成产品的创建。

基本信息	
所属资源空间	下拉选择所属的资源空间。如无对应的资源空间，请先创建资源空间。
产品名称	为产品命名。产品名称在相同资源空间有唯一性。长度不超过64，只允许中文、字母、数字、以及_?'#().,&%@!-字符的组合。
协议类型	• MQTT：使用MQTT协议接入平台的设备，数据格式可以是二进制也可以是JSON格式，采用二进制时需要部署编解码插件。 • LwM2M/CoAP：使用在资源受限（包括存储、功耗等）的NB-IoT设备，数据格式是二进制，需要部署编解码插件才能与物联网平台交互。 • HTTPS：HTTPS是基于HTTP协议，通过SSL加密的一种安全通信协议。物联网平台支持HTTPS协议通信。 • Modbus：物联网平台支持使用Modbus协议接入，使用Modbus协议、通过IoT边缘节点接入平台的设备（或其它通过网关连接平台的子设备）为非直连设备。直连设备和非直连设备具体差异说明，请参考这里。 • HTTP(TLS加密)、ONVIF、OPC-UA、OPC-DA、TCP、UDP、其它协议：通过边缘接入。

基本信息	
数据格式	• JSON：平台和设备之间的通信协议采用JSON格式。 • 二进制码流：您需在控制台开发**编解码插件**，将设备上报的二进制码流数据转换为JSON格式，将平台下发的JSON格式数据解析为二进制码流格式，设备才能与平台进行通信。
编码格式	当协议类型（protocol_type）为MQTT，数据格式（data_format）为二进制时，可通过该参数配置设备上报消息的编码格式。默认为UTF-8。 • UTF-8：将二进制码流转换为Unicode编码的字符串。 • BASE64：将二进制码流转换为BASE64编码的字符串。
所属行业	请根据实际情况选择。
设备类型	请根据实际情况选择。
高级配置	
产品ID	定制ProductID，用于唯一标识一个产品。如果携带此参数，平台将产品ID设置为该参数值；如果不携带此参数，产品ID在物联网平台创建产品后由平台分配获得。
产品描述	产品描述。请根据实际情况填写。

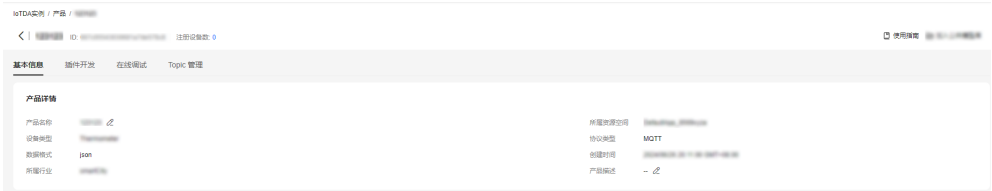
产品创建成功后，您可以单击“更多-删除”删除不再使用的产品。删除产品后，该产品下的产品模型、编解码插件等资源将被清空，请谨慎操作。

----结束

后续步骤

1. 在产品列表中，单击对应的产品，进入产品详情页。您可以查看产品ID、产品名称、设备类型、数据格式、所属资源空间、协议类型等产品基本信息。

   图 3-4 产品-产品详情

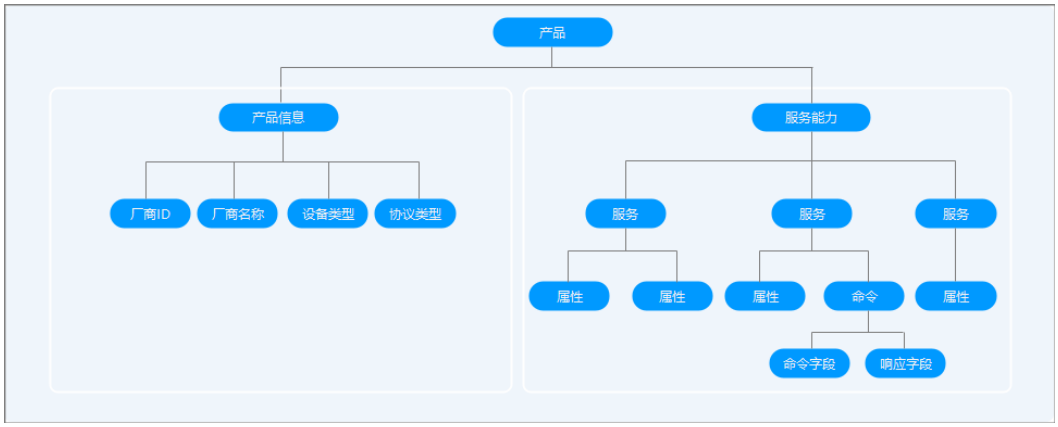
   

2. 您可以在产品详情页，**开发产品模型**、**开发编解码插件**、**在线调试**、**自定义Topic**。

3.2.3 开发产品模型

3.2.3.1 什么是产品模型

产品模型用于描述设备具备的能力和特性。开发者通过定义产品模型，在物联网平台构建一款设备的抽象模型，使平台理解该款设备支持的服务、属性、命令等信息，如开关等。当定义完一款产品模型后，在进行**注册设备**时，就可以使用在控制台上定义的产品模型。



产品模型定义了服务能力：

- **服务能力**

  描述设备具备的业务能力。将设备业务能力拆分成若干个服务后，再定义每个服务具备的属性、命令以及命令的参数。

  以水表为例，水表具有多种能力，如上报水流、告警、电量、连接等各种数据，并且能够接受服务器下发的各种命令。产品模型文件在描述水表的能力时，可以将水表的能力划分五个服务，每个服务都需要定义各自的上报属性或命令。说明如下：

服务类型	描述
基础（WaterMeterBasic）	用于定义水表上报的水流量、水温、水压等参数，如果需要命令控制或修改这些参数，还需要定义命令的参数。
告警（WaterMeterAlarm）	用于定义水表需要上报的各种告警场景的数据，必要的话需要定义命令。
电池（Battery）	定义水表的电压、电流强度等数据。
传输规则（DeliverySchedule）	定义水表的一些传输规则，必要的话需要定义命令。
连接（Connectivity）	定义水表连接参数。

> **说明**
>
> 具体定义几个服务是非常灵活的，如上面的例子可以将告警服务拆分成水压告警服务和流量告警服务，也可以将告警服务合入到水表基础服务中。物联网平台提供了多种定义产品模型的方法，您可以根据自己需求，选择对应的方法定义产品模型。

- **自定义模型（在线开发）**：从零自定义构建产品模型，详细请参考**在线开发产品模型**。
- **上传模型文件（离线开发）**：将本地写好的产品模型上传到平台，详细请参考**离线开发产品模型**。
- **Excel导入**：通过导入文件的方式快速定义产品功能。对于开发者来说，降低产品模型开发门槛，只需根据表格填写参数；对于高阶开发者和集成商来说，提升行业复杂模型开发效率。例如，楼宇自控空调模型包含的service条目超过100条，在表格中编辑开发产品模型，效率大大提升，可以随时编辑调整参数。详细请参考**Excel导入**。
- **导入库模型（平台预置产品模型）**：您可以使用平台预置的产品模型，快速完成产品开发。当前平台提供了标准模型和厂商模型。标准模型遵循行业标准的产品模型，适用行业内绝大部分厂商设备，而厂商模型针对设备类型发布的产品模型，适用于用行业内少量厂家设备。您可以根据实际需求选择相应的产品模型。

3.2.3.2 在线开发产品模型

概述

在线开发产品模型前需要**创建产品**。创建产品需要输入产品名称、协议类型、数据格式、所属行业和设备类型等信息，产品模型会使用这些信息作为设备能力字段取值。物联网平台提供了标准模型和厂商模型，这些模型涉及多个领域，模型中提供了已经编辑好的产品模型文件，您可以根据自己的需要对产品模型中的字段进行修改和增删；如果选择自定义产品模型，则需要完整定义产品模型。

本节定义包含一个服务的产品模型为示例，该产品模型包含设备上报数据、下发命令、下发命令响应等场景的服务和字段。

操作步骤

步骤1 访问**设备接入服务**，单击“管理控制台”进入“设备接入”控制台。选择您的实例，单击实例卡片进入。

步骤2 单击左侧导航栏的“产品”，在产品列表中，找到对应的产品，单击产品进入产品详情页。

步骤3 在产品详情基本信息页面，单击“自定义模型”，添加服务。

步骤4 输入“服务ID”、“服务类型”和“服务描述”，然后单击“确定”。

- “服务ID”：采用首字母大写的命名方式。比如：WaterMeter、StreetLight。
- “服务类型”：建议和服务ID保持一致。
- “服务描述”：比如路灯上报的环境光强度和路灯开关状态的属性。

添加服务后，在“添加服务”区域，对属性和命令进行定义。每个服务下，可以包含属性和命令，也可以只包含其中之一，请根据此类设备的实际情况进行配置。

步骤5 单击4新增的服务ID，在展开的页面单击“新增属性”，在弹出窗口中配置属性的各项参数，然后单击“确定”。

参数	说明
属性名称	建议采用驼峰形式，如batteryLevel、internalTemperature。
数据类型	• **int**：当上报的数据为整数时，可配置为此类型。 • **long:** 当上报的数据为长整型时，可配置为此类型。 • **decimal**：当上报的数据为小数时，可配置为此类型。配置“经纬度”属性时，数据类型建议使用“decimal”。 • **string**：当上报的数据为字符串、枚举值时，可以配置为此类型。如果为枚举值，值之间需要用英文逗号（“,”）分隔。 • **dateTime**：当上报的数据为日期时，可以配置为此类型。此类型属性上报格式推荐样例：2020-09-01T18:50:20Z或者2020-09-01T18:50:20.200Z • **jsonObject**：当上报的数据为JSON结构体时，可以配置为此类型。 • **enum:** 当上报的数据为枚举值时，可配置为此类型。搭配参数enumList格式填写，比如状态属性的enumList填写为OPEN,CLOSE，那么属性上报格式样例为"OPEN"或者"CLOSE" • **boolean:** 当上报的数据为布尔值时，可配置为此类型。此类型属性上报推荐格式样例：true/false 或者 0/1 • **stringList:** 当上报的数据为字符串数组时，可配置为此类型。 此类型属性上报推荐格式样例：["str1","str2","str3"]
访问权限	• 可读：通过接口可以查询该属性。 • 可写：通过接口可以修改该属性值。
取值范围	请根据此类设备的实际情况进行配置。
步长	
单位	

图 3-5 新增属性-batteryLevel

步骤6 单击“添加命令”，在弹出窗口中配置命令。

- “命令名称”：建议采用全大写形式，单词间用下划线连接的命名方式，如DISCOVERY，CHANGE_STATUS。
- “下发参数”：单击“新增输入参数”，在弹出窗口中配置下发命令字段的各项参数，然后“确定”。

参数	说明
参数名称	建议采用第一个单词首字母小写，其余单词的首字母大写的命名方式，比如valueChange
数据类型	请根据此类设备的实际情况进行配置。
取值范围	
步长	
单位	

图 3-6 新增命令 CHANGE_STATUS

- 如果要添加命令响应，单击“新增响应参数”，在弹出窗口中配置响应命令字段的各项参数，然后单击“确定”。

参数	说明
参数名称	建议采用第一个单词首字母小写，其余单词的首字母大写的命名方式，比如valueResult
数据类型	请根据此类设备的实际情况进行配置。
取值范围	
步长	
单位	

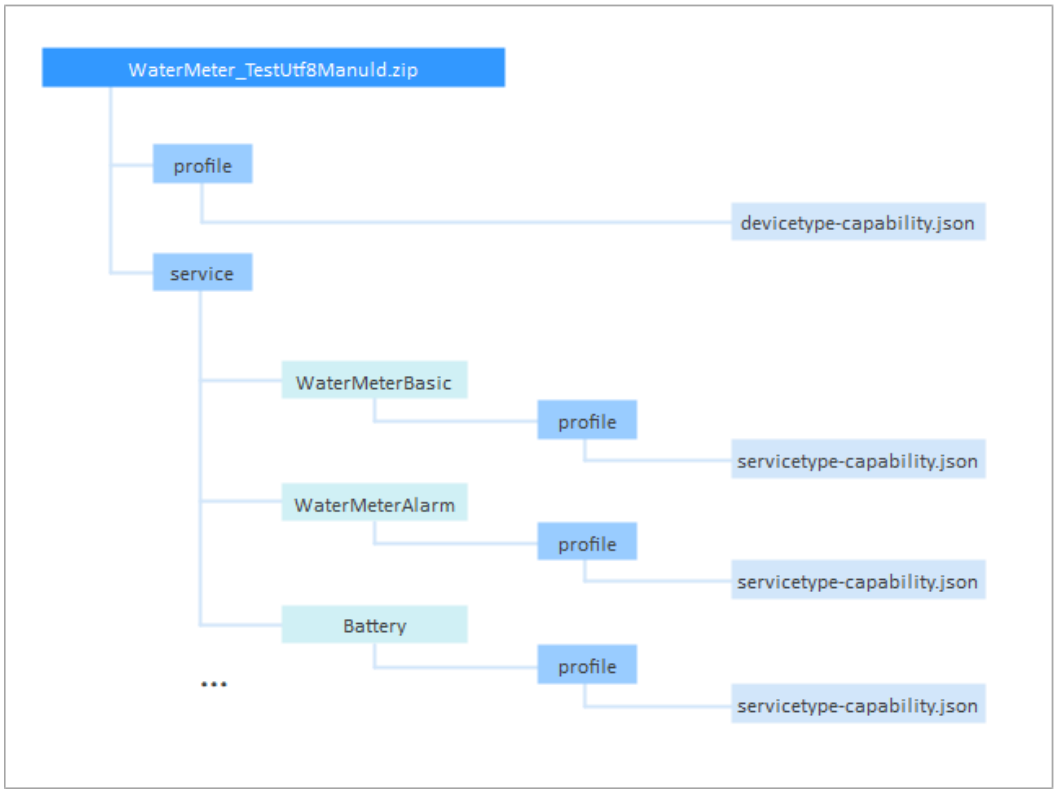
图 3-7 新增命令响应参数-valueAResult

----结束

3.2.3.3 离线开发产品模型

概述

产品模型本质上就是一个**devicetype-capability.json**文件和若干个**serviceType-capability.json**文件，其中**devicetype-capability.json**是描述产品模型包含的服务能力的文件，**serviceType-capability.json**文件是用于描述**devicetype-capability.json**文件中的service_capabilities的每一个能力的详细描述。按照如下目录打包的一个zip包。其中WaterMeter是deviceType，TestUtf8Manuld是manufactureId，WaterMeterBasic/WaterMeterAlarm/Battery是服务类型。



因为离线开发产品模型就是按照产品模型编写规则和JSON格式规范在devicetype-capability.json中定义设备能力，在servicetype-capability.json中定义服务能力，所以离线开发产品模型需要熟悉JSON的格式。

由于离线开发产品模型文件相对在线开发比较耗时，因此推荐**在线开发产品模型**。

命名规范

在产品模型的开发过程中，需要遵循如下命名规范：

- 设备类型（deviceType）、服务类型（serviceType）、服务标识（serviceId）采用单词首字母大写的命名法。例如：WaterMeter、Battery。
- 属性使用第一个单词首字母小写，其余单词的首字母大写的命名法。例如：batteryLevel、internalTemperature。
- 命令使用所有字母大写，单词间用下划线连接的格式。例如：DISCOVERY，CHANGE_COLOR。
- 设备能力描述json文件固定命名devicetype-capability.json。
- 服务能力描述json文件固定命名servicetype-capability.json。
- 厂商id在不同的产品模型文件中不能重复，且仅支持英文。
- 要注重名称的通用性，简洁性；对于服务能力描述，还要考虑其功能性。例如：对于多传感器设备，可以命名为MultiSensor；对于具有显示电量的服务，可以命名为Battery。

产品模型模板

将一款新设备接入到物联网平台，首先需要编写这款设备的产品模型。物联网平台提供了一些产品模型文件模板，如果新增接入设备的类型和功能服务已经在物联网平台

提供的设备产品模型模板中包含，则可以直接选择使用；如果在物联网平台提供的设备产品模型模板中未包含，则需要自己定义。

例如：接入一款水表 ，可以直接选择物联网平台上对应的产品模型模板，修改设备服务列表。

说明

物联网平台提供的产品模型模板会不断更新，如下表格列举设备类型和服务类型示例，仅供参考。

设备类型识别属性：

属性	产品模型JSON文件key	属性值
设备类型	deviceType	WaterMeter
厂商ID	manufacturerId	TestUtf8ManuId
厂商名称	manufacturerName	HZYB
协议类型	protocolType	CoAP

设备的服务列表

服务描述	服务标识（serviceId）	服务类型（serviceType）	选项（option）
水表的基本功能	WaterMeterBasic	Water	Mandatory
告警服务	WaterMeterAlarm	Battery	Mandatory
电池服务	Battery	Battery	Optional
数据的上报规则	DeliverySchedule	DeliverySchedule	Mandatory
水表的连通性	Connectivity	Connectivity	Mandatory

设备能力定义样例

devicetype-capability.json记录了该设备的基础信息：

```
{
   "devices": [
      {
         "manufacturerId": "TestUtf8ManuId",
         "manufacturerName": "HZYB",
         "protocolType": "CoAP",
         "deviceType": "WaterMeter",
         "omCapability":{
                     "upgradeCapability" : {
                          "supportUpgrade":true,
                          "upgradeProtocolType":"PCP"
                     },
                     "fwUpgradeCapability" : {
                          "supportUpgrade":true,
                          "upgradeProtocolType":"LWM2M"
```

```
                },
                "configCapability" : {
                    "supportConfig":true,
                    "configMethod":"file",
                    "defaultConfigFile": {
                      "waterMeterInfo" : {
                          "waterMeterPirTime" : "300"
                       }
                    }
                }
            },
            "serviceTypeCapabilities": [
              {
                "serviceId": "WaterMeterBasic",
                "serviceType": "WaterMeterBasic",
                "option": "Mandatory"
              },
              {
                "serviceId": "WaterMeterAlarm",
                "serviceType": "WaterMeterAlarm",
                "option": "Mandatory"
              },
              {
                "serviceId": "Battery",
                "serviceType": "Battery",
                "option": "Optional"
              },
              {
                "serviceId": "DeliverySchedule",
                "serviceType": "DeliverySchedule",
                "option": "Mandatory"
              },
              {
                "serviceId": "Connectivity",
                "serviceType": "Connectivity",
                "option": "Mandatory"
              }
            ]
        }
    ]
}
```

各字段的解释：

字段	子字段		可选/必选	描述
devices	-	-	必选	包含了一个设备的完整能力信息（根节点不能修改）。
-	manufacturerId	-	可选	指示设备的制造商ID。
-	manufacturerName	-	必选	指示设备的制造商名称（只允许英文）。
-	protocolType	-	必选	指示设备接入物联网平台的协议类型。如NB-IoT的设备取值为CoAP。
-	deviceType	-	必选	指示设备的类型。

字段	子字段		可选/必选	描述
-	omCapability	-	可选	定义设备的软件升级、固件升级和配置更新的能力，字段含义详情见下文中的：omCapability结构描述。 如果设备不涉及软件/固件升级，本字段可以删除。
-	serviceTypeCapabilities	-	必选	包含了设备具备的服务能力描述。
-	-	serviceId	必选	服务的Id，如果设备中同类型的服务类型只有一个则serviceId与serviceType相同， 如果有多个则增加编号，如三键开关 Switch01、Switch02、Switch03。
-	-	serviceType	必选	服务类型，与servicetype-capability.json中serviceType字段保持一致。
-	-	option	必选	标识服务字段的类型，取值范围：Master（主服务）, Mandatory（必选服务）, Optional（可选服务）。 目前本字段为非功能性字段，仅起到描述作用。

omCapability结构描述

字段	子字段	可选/必选	描述
upgradeCapability	-	可选	设备软件升级能力。
-	supportUpgrade	可选	true：设备支持软件升级。 false：设备不支持软件升级。
-	upgradeProtocolType	可选	升级使用的协议类型，此处不同于设备的protocolType，例如CoAP设备软件升级协议使用PCP。
fwUpgradeCapability	-	可选	设备固件升级能力。
-	supportUpgrade	可选	true：设备支持固件升级。 false：设备不支持固件升级。
-	upgradeProtocolType	可选	升级使用的协议类型，此处不同于设备的protocolType，当前物联网平台仅支持LWM2M固件升级。

字段	子字段	可选/必选	描述
configCapability	-	可选	设备配置更新能力。
-	supportConfig	可选	true：设备支持配置更新。 false：设备不支持配置更新。
-	configMethod	可选	file：使用文件的方式下发配置更新。
-	defaultConfigFile	可选	设备默认配置信息（Json格式），具体配置信息由设备商自定义。物联网平台只储存该信息供下发时使用，不解析处理配置字段的具体含义。

服务能力定义样例

servicetype-capability.json记录了该设备的服务信息：

```
{
   "services": [
      {
         "serviceType": "WaterMeterBasic",
         "description": "WaterMeterBasic",
         "commands": [
            {
               "commandName": "SET_PRESSURE_READ_PERIOD",
               "paras": [
                  {
                     "paraName": "value",
                     "dataType": "int",
                     "required": true,
                     "min": 1,
                     "max": 24,
                     "step": 1,
                     "maxLength": 10,
                     "unit": "hour",
                     "enumList": null
                  }
               ],
               "responses": [
                  {
                     "responseName": "SET_PRESSURE_READ_PERIOD_RSP",
                     "paras": [
                        {
                           "paraName": "result",
                           "dataType": "int",
                           "required": true,
                           "min": -1000000,
                           "max": 1000000,
                           "step": 1,
                           "maxLength": 10,
                           "unit": null,
                           "enumList": null
                        }
                     ]
                  }
               ]
            }
         ],
```

```
        "properties": [
            {
                "propertyName": "registerFlow",
                "dataType": "int",
                "required": true,
                "min": 0,
                "max": 0,
                "step": 1,
                "maxLength": 0,
                "method": "R",
                "unit": null,
                "enumList": null
            },
            {
                "propertyName": "currentReading",
                "dataType": "string",
                "required": false,
                "min": 0,
                "max": 0,
                "step": 1,
                "maxLength": 0,
                "method": "W",
                "unit": "L",
                "enumList": null
            },
            {
                "propertyName": "timeOfReading",
                "dataType": "string",
                "required": false,
                "min": 0,
                "max": 0,
                "step": 1,
                "maxLength": 0,
                "method": "W",
                "unit": null,
                "enumList": null
            },
            ......
        ]
    }
  ]
}
```

各字段的解释：

字段	子字段				必选/可选	描述
services	-	-	-	-	必选	包含了一个服务的完整信息（根节点不可修改）。
-	serviceType	-	-	-	必选	服务的类型，与devicetype-capability.json中serviceType字段保持一致。
-	description	-	-	-	必选	服务的描述信息。 非功能性字段，仅起到描述作用，可置为null。

字段	子字段				必选/可选	描述
-	commands	-	-	-	必选	设备可以执行的命令，如果本服务无命令则置null。
-	-	commandName	-	-	必选	命令的名字，命令名与参数共同构成一个完整的命令。
-	-	paras	-	-	必选	命令包含的参数。
-	-	-	paraName	-	必选	命令中参数的名字。
-	-	-	dataType	-	必选	命令参数的数据类型。 取值范围：string、int、string list、decimal、DateTime、jsonObject、enum、boolean。 上报数据时，复杂类型数据格式如下： ● string list：["str1","str2","str3"] ● DateTime：yyyyMMdd’T’HHmmss’Z’ 如:20151212T121212Z ● jsonObject：自定义json结构体，物联网平台不解析，仅进行透传
-	-	-	required	-	必选	本命令是否必选，取值为true或false，默认取值false（非必选）。 目前本字段是非功能性字段，仅起到描述作用。
-	-	-	min	-	必选	最小值。 仅当dataType为int、decimal时生效。
-	-	-	max	-	必选	最大值。 仅当dataType为int、decimal时生效。
-	-	-	step	-	必选	步长。 暂不使用，填0即可。
-	-	-	maxLength	-	必选	字符串长度。 仅当dataType为string、string list、DateTime时生效。

字段	子字段				必选/可选	描述
-	-	-	unit	-	必选	单位，需要使用英文。 取值根据参数确定，如： 温度单位：“C”或“K” 百分比单位：“%” 压强单位：“Pa”或“kPa”
-	-	-	enumList	-	必选	枚举值。 如开关状态status可有如下取值： "enumList" : ["OPEN","CLOSE"] 目前本字段是非功能性字段，仅起到描述作用，建议准确定义。
-	-	responses	-	-	必选	命令执行的响应。
-	-	-	responseName	-	必选	命名可以在该responses对应命令的commandName后面添加“_RSP”。
-	-	-	paras	-	必选	命令响应的参数。
-	-	-	-	paraName	必选	命令中参数的名字。
-	-	-	-	dataType	必选	数据类型。 取值范围：string、int、string list、decimal、DateTime、jsonObject 上报数据时，复杂类型数据格式如下： ● string list：["str1","str2","str3"] ● DateTime：yyyyMMdd’T’HHmmss’Z’ 如:20151212T121212Z ● jsonObject：自定义json结构体，物联网平台不解析，仅进行透传
-	-	-	-	required	必选	本命令响应是否必选，取值为true或false，默认取值false（非必选）。 目前本字段是非功能性字段，仅起到描述作用。

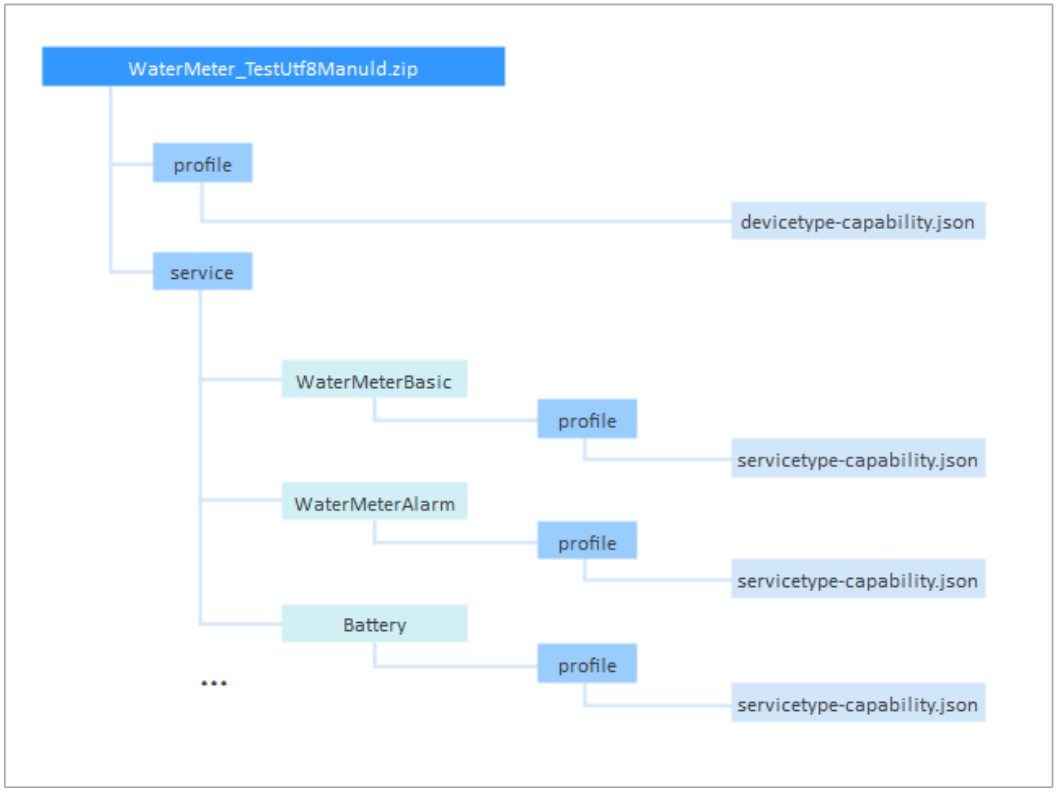
字段	子字段				必选/可选	描述
-	-	-	-	min	必选	最小值。 仅当dataType为int、decimal时生效，逻辑大于等于。
-	-	-	-	max	必选	最大值。 仅当dataType为int、decimal时生效，逻辑小于等于。
-	-	-	-	step	必选	步长。 暂不使用，填0即可。
-	-	-	-	maxLength	必选	字符串长度。 仅当dataType为string、string list、DateTime时生效。
-	-	-	-	unit	必选	单位，需要使用英文。 取值根据参数确定，如： 温度单位：“C”或“K” 百分比单位：“%” 压强单位：“Pa”或“kPa”
-	-	-	-	enumList	必选	枚举值。 如开关状态status可有如下取值： "enumList" : ["OPEN","CLOSE"] 目前本字段是非功能性字段，仅起到描述作用，建议准确定义。
-	properties	-	-	-	必选	上报数据描述，每一个子节点为一条属性。
-	-	propertyName	-	-	必选	属性名称。

字段	子字段				必选/可选	描述
-	-	dataType	-	-	必选	数据类型。 取值范围：string、int、string list、decimal、DateTime、jsonObject 上报数据时，复杂类型数据格式如下： ● string list：["str1","str2","str3"] ● DateTime：yyyyMMdd’T’HHmmss’Z’ 如:20151212T121212Z ● jsonObject：自定义json结构体，物联网平台不解析，仅进行透传
-	-	required	-	-	必选	本条属性是否必选，取值为true或false，默认取值false（非必选）。 目前本字段是非功能性字段，仅起到描述作用。
-	-	min	-	-	必选	最小值。 仅当dataType为int、decimal时生效，逻辑大于等于。
-	-	max	-	-	必选	最大值。 仅当dataType为int、decimal时生效，逻辑小于等于。
-	-	step	-	-	必选	步长。 暂不使用，填0即可。
-	-	method	-	-	必选	访问模式。 R：可读；W：可写；E：可订阅。 取值范围：R、RW、RE、RWE 、null。
-	-	unit	-	-	必选	单位，需要使用英文。 取值根据参数确定，如： 温度单位：“C”或“K” 百分比单位：“%” 压强单位：“Pa”或“kPa”
-	-	maxLength	-	-	必选	字符串长度。 仅当dataType为string、string list、DateTime时生效。

字段	子字段				必选/可选	描述
-	-	enum List	-	-	必选	枚举值。 如电池状态（batteryStatus）可有如下取值： "enumList"：[0, 1, 2, 3, 4, 5, 6] 目前本字段是非功能性字段，仅起到描述作用，建议准确定义。

产品模型打包

产品模型写作完成后，需要按如下层级结构打包：



产品模型打包需要遵循如下几点要求：

- 产品模型文件的目录层级结构必须如上图所示，不能增删。例如：第二层级只能有“profile”和“service”两个文件夹，每个服务下面必须包含“profile”文件夹等。
- 产品模型文件以zip形式压缩。
- 产品模型文件的命名必须按照deviceType_manufacturerId的格式命名，其中的deviceType、manufacturerId必须与devicetype-capability.json中对应字段的定义一致。例如：本实例中devicetype-capability.json的主要字段如下：

```
{
  "devices": [
    {
      "manufacturerId": "TestUtf8ManuId",
      "manufacturerName": "HZYB",

      "protocolType": "CoAP",
      "deviceType": "WaterMeter",
      "serviceTypeCapabilities": ****
    }
  ]
}
```

- 图中的WaterMeterBasic、WaterMeterAlarm、Battery等都是devicetype-capability.json中定义的服务。

产品模型文件中的文档格式都是JSON，在编写完成后可以在互联网上查找一些格式校验网站，检查JSON的合法性。

3.2.3.4 导出和导入产品模型

产品模型作为一种资源，可以从物联网平台导出，也可以导入到物联网平台。

- 当产品开发完成并测试验证后，需要将在线开发的产品模型移植时，则可以将产品模型导出到本地。
- 当您已经有完备的产品模型（线下开发或从其他项目/平台导出），或者使用excel编辑开发产品模型时，可以将产品模型直接导入到“物联网平台”。

导出产品模型

当产品开发完成并测试验证后，需要将在线开发的产品模型移植时，则可以将产品导出到本地。

步骤1 访问**设备接入服务**，单击“管理控制台”进入设备接入控制台。选择您的实例，单击实例卡片进入。

步骤2 单击左侧导航栏的“产品”，在产品列表中，找到对应的产品，单击产品进入产品界面。

步骤3 在产品界面，单击右边“导出”，将产品模型下载到本地。

图 3-8 产品模型-导出

----结束

导入产品模型

当您已经有完备的产品模型时（线下开发或从其他项目/平台导出），或者使用excel编辑开发产品模型时，可以将产品模型直接导入IoTDA物联网平台。

说明

通过本地导入的产品模型不含编解码插件。编解码插件是供物联网平台调用的插件，可以完成二进制格式与JSON格式相互转换、也可以完成JSON格式之间的转换，详细说明可见**什么是编解码插件**。如果设备上报采用的是二进制码流，请前往控制台进行插件开发或导入插件。

- **上传模型文件**
  a. 访问**设备接入服务**，单击“管理控制台”进入设备接入控制台。选择您的实例，单击实例卡片进入。
  b. 单击左侧导航栏的“产品”，在产品列表中，找到对应的产品，单击产品进入产品界面。
  c. 在基本信息页面，单击“上传模型文件”，在弹框中加载本地的产品模型文件，然后单击“确定”。

  图 3-9 产品-上传产品模型

- **Excel导入**
  a. 访问**设备接入服务**，单击“管理控制台”进入设备接入控制台。选择您的实例，单击实例卡片进入。
  b. 单击左侧导航栏的“产品”，在产品列表中，找到相应的产品，单击产品进入产品界面。
  c. 单击“Excel导入”，在产品模板表格中，填写“设备”页签的服务ID，以及“参数”页签的属性、命令、事件等参数。导入Excel表格后，然后单击“确定”。

图 3-10 产品-Excel 导入产品模型

3.2.4 开发编解码插件

3.2.4.1 什么是编解码插件

编解码插件是供物联网平台调用的数据格式转换插件，可以对设备上报给平台的数据、和平台要下发给设备的数据进行二进制格式与JSON格式之间的转换，或JSON格式之间的转换。

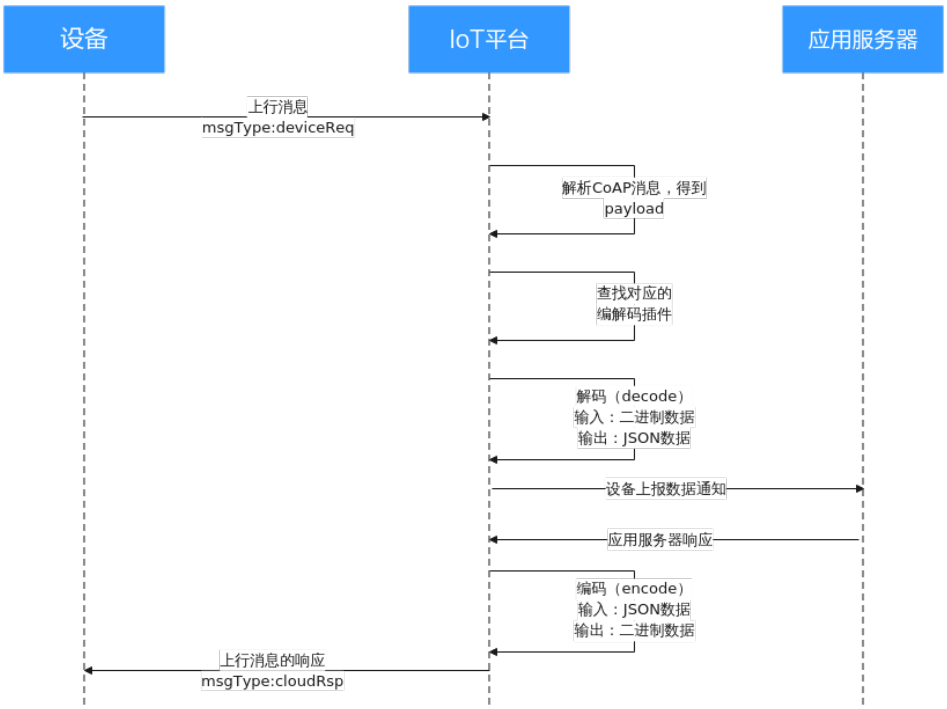
以NB-IoT场景为例，编解码插件将设备上报的二进制数据解码为JSON格式供应用服务器“阅读”，将应用服务器下行的JSON格式命令编码为二进制格式数据供终端设备（UE）“理解执行”。NB-IoT设备和物联网平台之间采用CoAP协议通讯，CoAP消息的payload为应用层数据，应用层数据的格式由设备自行定义。由于NB-IoT设备一般对省电要求较高，所以应用层数据一般不采用流行的JSON格式，而是采用二进制格式。但是，物联网平台与应用侧使用JSON格式进行通信。因此，您需要开发编码插件，供物联网平台调用，可以完成二进制格式和JSON格式的转换。

应用场景

1. 使用LwM2M/CoAP协议接入。
2. 泛协议接入，如TCP协议、JT808协议、GB32960协议等。
3. MQTT(S)/HTTP(S)协议设备接入无需进行编解码插件开发。

数据上报流程

图 3-11 数据上报编解码插件

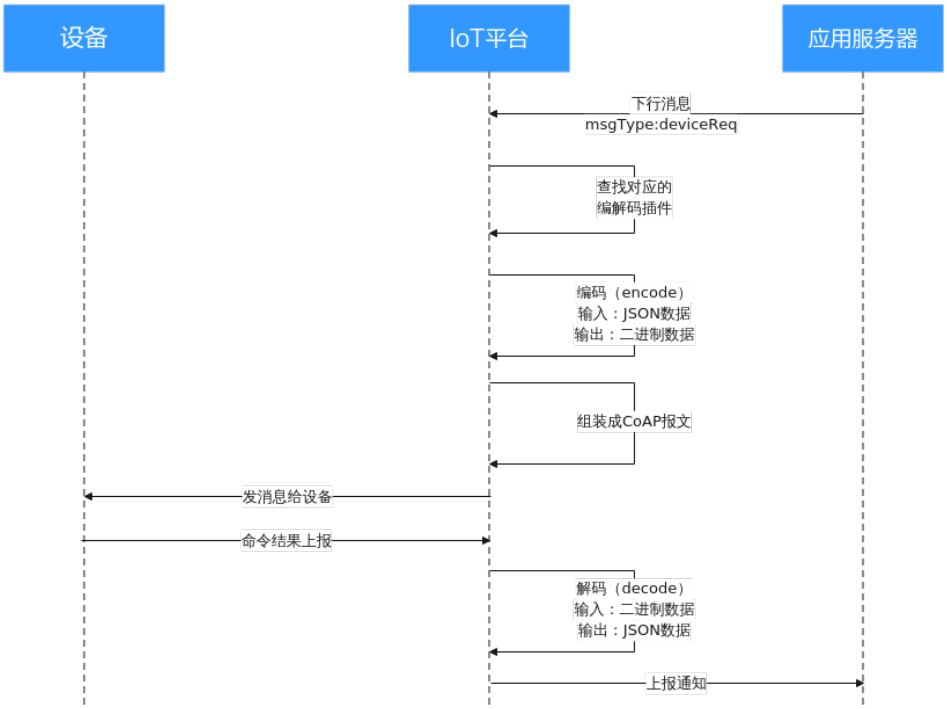


在数据上报流程中，有两处需要用到编解码插件：

- 将设备上报的二进制码流解码成JSON格式的数据，发送给应用服务器。
- 将应用服务器响应的JSON格式数据编码成二进制码流，下发给设备。

命令下发流程

图 3-12 命令下发编解码插件流程



在命令下发流程中，有两处需要用到编解码插件：

- 将应用服务器下发的JSON格式数据编码成二进制码流，下发给设备。
- 将设备响应的二进制码流解码成JSON格式的数据，上报给应用服务器。

图形化开发和脚本化开发

编解码插件的开发方式有图形化开发和脚本化开发。

- **图形化开发**是指在设备接入控制台，通过可视化的方式快速开发一款产品的编解码插件。
- **脚本化开发**是指使用JavaScript脚本实现编解码的功能。2024年12月1日后新用户不再提供JavaScript插件功能，推荐使用FunctionGraph进行JavaScript脚本编写，详细请参考**FunctionGraph开发说明**。

相关 FAQ

编解码插件相关问题

什么是NB-IoT?

相关最佳实践

模拟NB设备智慧路灯的接入与调试

3.2.4.2 图形化开发插件

当前在华为物联网平台上，使用图形化开发的编解码插件只适用于上报数据格式为二进制的设备。

在设备接入控制台，我们可以通过可视化的方式快速开发一款产品的编解码插件。

本节首先以一个NB-IoT烟感设备的例子讲解如何开发一个支持数据上报和命令下发的编解码插件，并且支持上报命令执行结果，然后再以两个场景举例说明如何完成复杂的插件开发以及调试。

- **数据上报和命令下发**
- **字符串及可变长字符串的编解码插件**
- **数组及可变长数组数据类型**

数据上报和命令下发

场景说明

有一款烟感设备，具有如下特征：

- 具有烟雾报警功能（火灾等级）和温度上报功能。
- 支持远程控制命令，可远程打开报警功能。比如火灾现场温度，远程打开烟雾报警，提醒住户疏散。
- 支持上报命令执行结果。

产品模型定义

在烟感产品的开发空间，完成**产品模型**定义。

- level：火灾级别，用于表示火灾的严重程度。
- temperature：温度，用于表示火灾现场温度。
- SET_ALARM：打开或关闭告警命令，value=0表示关闭，value=1表示打开。

图 3-13 模型定义-smokedetector

编解码插件开发

步骤1 在烟感产品的开发空间，选择“插件开发”，单击“图形化开发”。

步骤2 单击“新增消息”，新增“smokerinfo”消息。配置此步骤的主要目的是，将设备上传的二进制码流消息解码成JSON格式，以便物联网平台理解。配置示例如下：

- 消息名：smokerinfo

- 消息类型：数据上报。
- 添加响应字段：是。添加响应字段后，物联网平台在收到设备上报的数据后，会下发用户设置的响应数据到设备。
- 响应数据：AAAA0000（默认）

图 3-14 插件开发-新增消息 smokerinfo

1. 单击“添加字段”，勾选“标记为地址域”，添加messageId字段，表示消息类型。在本场景中，上报火灾等级和温度的消息类型是0x0。设备上报消息时，每条消息首个字段就是messageId。如设备上报消息为0001013A，第一个字段00就是表示此条消息是上报火灾级别和温度的消息。后续字段01和013A分别代表火灾级别和温度。如果只有一条数据上报消息和一条命令下发消息，可以不添加messageId字段。
   - “数据类型”根据数据上报消息种类的数量进行配置。messageId字段默认的数据类型为int8u。
   - “偏移值”是根据字段位置和字段的字节数的配置自动填充的。messageId为此消息的第一个字段，起始位置为0，字节长度为1，终点位置为1。所以偏移值为0-1。
   - “长度”是根据“数据类型”的配置自动填充的。
   - “默认值”可以修改，但必须为十六进制格式，且设备数据上报消息的对应字段必须和此处的默认值保持一致。

图 3-15 插件开发-添加字段 messageId

2. 添加level字段，表示火灾级别。
   - “字段名”只能输入包含字母、数字、_和$，且不能以数字开头的字符。
   - “数据类型”根据设备上报数据的实际情况进行配置，需要和产品模型相应字段的定义相匹配。产品模型中定义的火灾级别level属性的数据类型为int，最大值为9。所以选择的数据类型为int8u。
   - “偏移值”是根据字段位置和字段的字节数的配置自动填充的。“level”字段的起始位置就是前一字段的终点，前一字段“messageId”的终点位置为1，所以“level”字段的起始位置为1。“level”字段长度为1个字节，终点为2。所以“偏移值”为1-2。
   - “长度”根据“数据类型”的配置自动填充。
   - “默认值”不填。此处火灾级别level不固定，无默认值。

图 3-16 插件开发-添加字段 level

3. 添加**temperature**字段，表示温度。
   - “数据类型”，在产品模型中，temperature属性的“数据类型”为int，最大值1000，因此在插件中定义temperature字段的“数据类型”为“int16u”，以满足temperature属性的取值范围。
   - “偏移值”是根据与首字段的间隔的字符数自动配置的。“temperature”字段的起始位置就是前一字段的终点，前一字段“level”的终点位置为2，所以“temperature”字段的起始位置为2。“temperature”字段长度为2个字节，终点为4。所以“偏移值”为2-4。
   - “长度”根据数据类型的配置自动填充。
   - “默认值”不填，此处温度temperature的值不固定，无默认值。

图 3-17 插件开发-添加字段 temperature

步骤3 单击“新增消息”，新增“SET_ALARM”消息，设置火灾告警的温度阈值。例如超过60摄氏度，设备上报告警。配置此步骤的主要目的是，将平台下发的JSON格式命令消息编码成二进制数据，以便烟感设备理解。配置示例如下：

- 消息名：SET_ALARM
- 消息类型：命令下发
- 添加响应字段：是。添加响应字段后，设备在接收命令后，可以上报命令执行结果。您可以根据自己的需求，选择是否添加响应字段。

图 3-18 插件开发-新增消息 SET_ALARM

a. 单击“添加字段”，添加messageId字段，表示消息类型。例如，设置火灾告警阈值的消息类型为0x3。messageId、数据类型、长度、默认值、偏移值的说明可参考1。

图 3-19 插件开发-添加命令字段 messageId(0x3)

b. 添加mid字段。这里的mid字段是由平台生成和下发的，用于将下发的命令和命令下发响应消息关联。mid字段的数据类型默认为int16u。长度、默认值、偏移值的说明可参考**2**。

图 3-20 插件开发-添加命令字段 mid

c. 添加value字段，表示下发命令的参数值。例如，下发火灾告警的温度阈值。数据类型、长度、默认值、偏移值的说明可以参考2。

图 3-21 插件开发-添加命令字段 value

d. 单击“添加响应字段”，添加“messageId”字段，表示消息类型。命令下发响应消息为上行消息，需要通过messageId和数据上报消息进行区分。上报火灾告警温度阈值的消息类型为0x4。messageId、数据类型、长度、默认值、偏移值的说明可参考1。

图 3-22 插件开发-添加响应字段 messageId(0x4)

e. 添加mid字段。这里的mid字段需要跟平台下发的命令里的mid字段保持一致，用于将下发的命令和命令执行结果进行关联。mid字段的数据类型默认为int16u。长度、默认值、偏移值的说明可参考**2**。

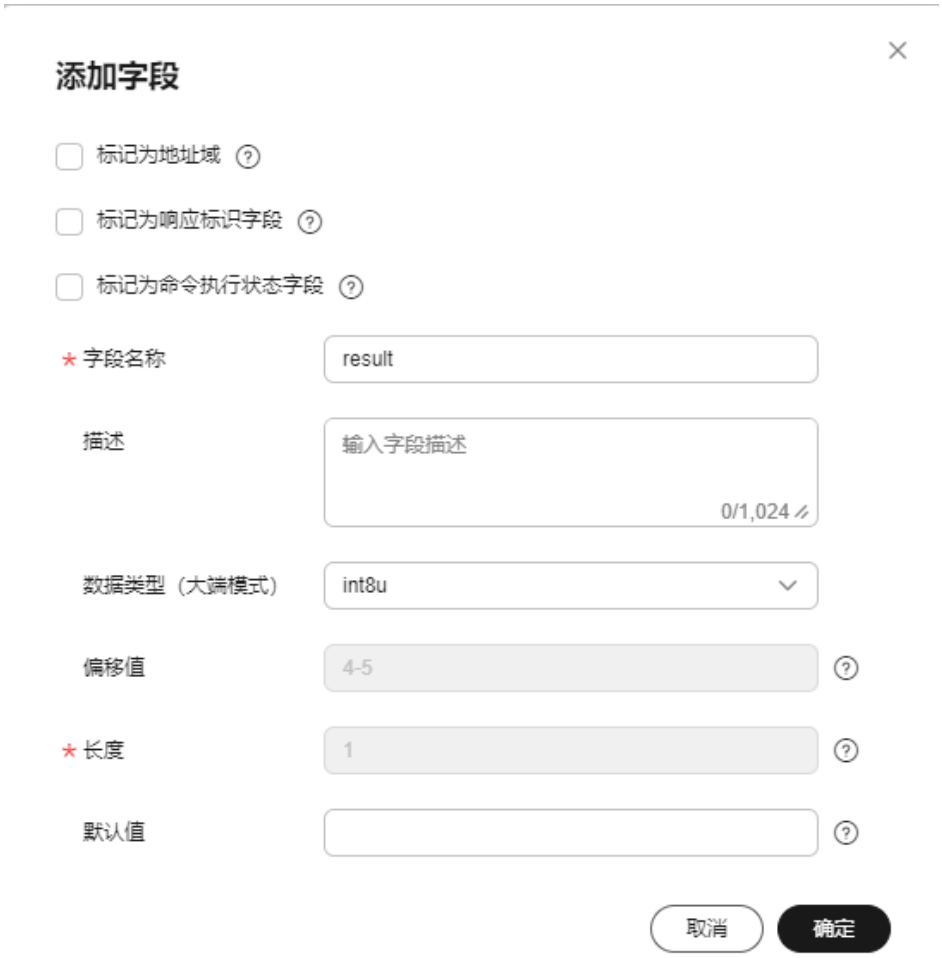
图 3-23 插件开发-添加响应字段 mid

f. 添加errcode字段，用于表示命令执行状态：00表示成功，01表示失败，如果未携带该字段，则默认命令执行成功。errcode字段的数据类型默认为int8u。长度、默认值、偏移值的说明可参考2。

图 3-24 插件开发-添加响应字段 errcode

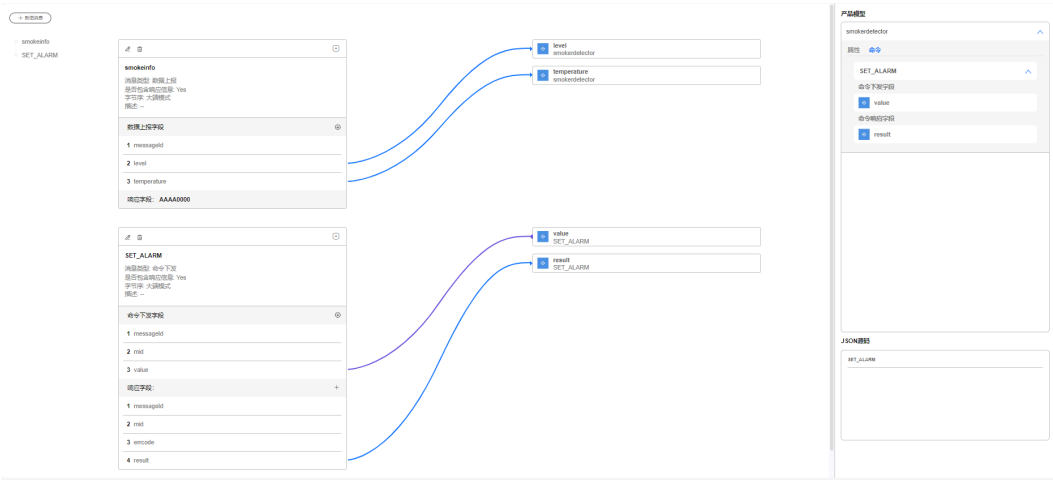
g. 添加result字段，用于表示命令执行结果。例如，设备向平台返回当前的告警阈值。

图 3-25 插件开发-添加响应字段 result



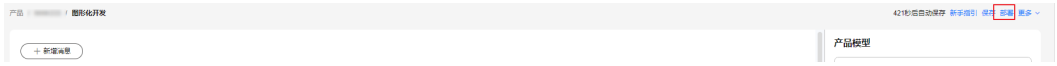
步骤4 拖动右侧“设备模型”区域的属性字段和命令字段，数据上报消息和命令下发消息的相应字段建立映射关系。

图 3-26 插件开发-在线开发插件 smokerdetector



步骤5 单击“保存”，并在插件保存成功后单击“部署”，将编解码插件部署到物联网平台。

图 3-27 插件开发-部署插件



----结束

调测编解码插件

步骤1 在烟感产品的开发空间，选择“在线调试”，并单击“**新增测试设备**”。

步骤2 用户可根据自己的业务场景，选择使用真实设备或者虚拟设备进行调测。具体调测步骤请参考**在线调试**。本文以虚拟设备为例，调测编解码插件。

在弹出的“新增测试设备”窗口，选择“虚拟设备”，单击“确定”，创建一个虚拟设备。虚拟设备名称包含 “DeviceSimulator”字样，每款产品下只能创建一个虚拟设备。

图 3-28 在线调试-创建虚拟设备

步骤3 单击“调试”，进入调试界面。

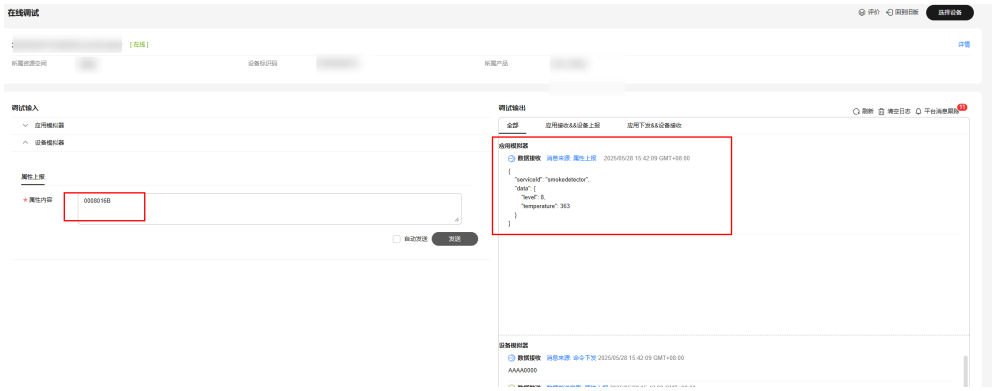
图 3-29 在线调试-进入调试

步骤4 使用设备模拟器进行数据上报。十六进制码流示例：0008016B。00为地址域messageId，08表示火灾级别level，长度为1个字节；016B表示温度，长度为2个字节。

在“应用模拟器”区域查看数据上报的结果：{level=8, temperature=363}。8为十六进制数08转换为十进制的数值；363为十六进制数016B转换为十进制的数值。

在设备模拟器区域看到平台下发的响应数据AAAA0000。

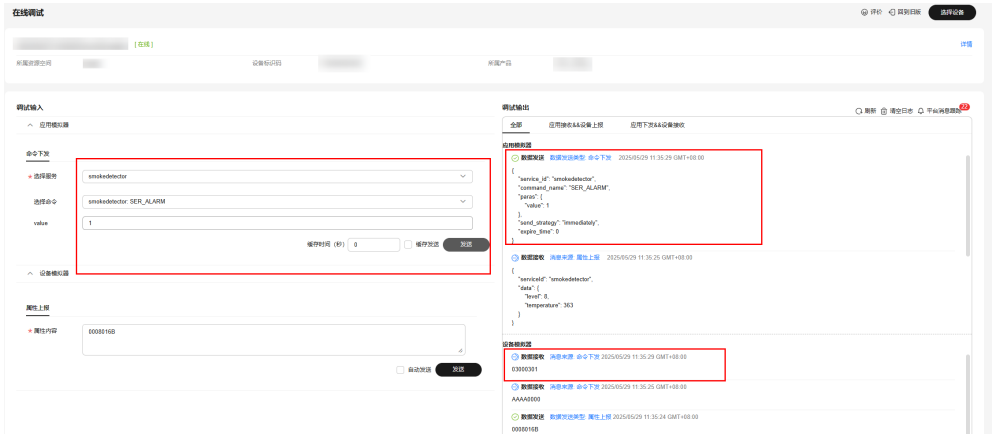
图 3-30 在线调试-模拟数据上报 smokerdetector



步骤5 使用应用模拟器进行命令下发，输入value值为1，可看到应用模拟下发命令{ "serviceId": "Smokeinfo", "method": "SET_ALARM", "paras": "{\"value\":1}" }。

在“设备模拟器”区域查看命令接收的结果：03000101。03为地址域messageId，0001为mid字段，01为十进制数1转换为十六进制的数值。

图 3-31 在线调试-模拟命令下发 smokerdetector



说明

使用CoAP的虚拟设备在线调试时，若下发命令后设备模拟器未接收到对应的命令，可以先上报一次属性后，再下发。

----结束

总结

- 如果插件需要对命令执行结果进行解析，则必须在命令和命令响应中定义mid字段。
- 命令下发的mid是2个字节，对于每个设备来说，mid从1递增到65535，对应码流为0001到FFFF。
- 设备执行完命令，命令执行结果上报中的mid要与收到命令中的mid保持一致，这样平台才能刷新对应命令的状态。

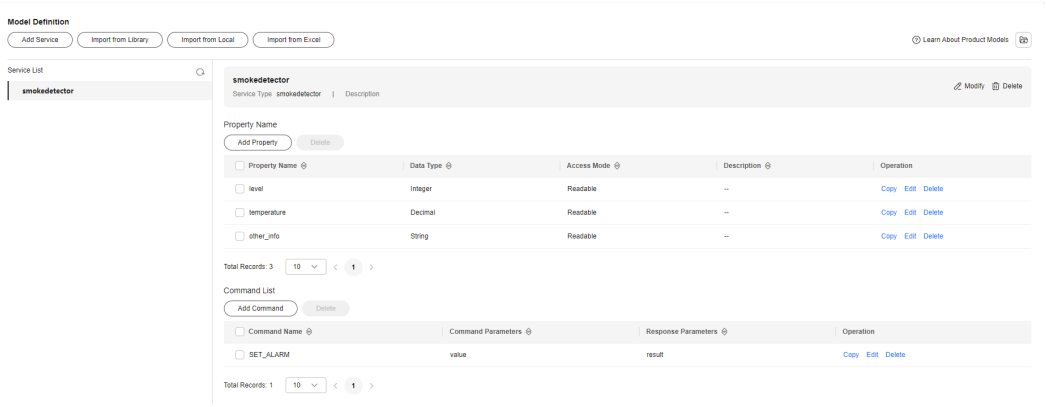
字符串及可变长字符串的编解码插件

如果该烟感设备需要支持描述信息上报功能，描述信息支持字符串和可变长度字符串两种类型，则按照以下步骤创建消息。

产品模型定义

重新创建一个烟感产品，并在烟感产品的开发空间完成产品模型定义。

图 3-32 模型定义-smokedetector 携带 other_info



编解码插件开发

步骤1 在烟感产品的开发空间，选择“插件开发”，单击“图形化开发”。

步骤2 单击“新增消息”，新增消息“other_info”，上报字符串类型的描述信息。配置此步骤的主要目的是，将设备上传的字符串二进制码流消息解码成JSON格式，以便物联网平台理解。配置示例如下：

- 消息名：other_info
- 消息类型：数据上报
- 添加响应字段：是。添加响应字段后，物联网平台在收到设备上报的数据后，会下发用户设置的响应数据到设备。
- 响应数据：AAAA0000（默认）

图 3-33 插件开发-新增消息 other_info

1. 单击“添加字段”，添加messageId字段，表示消息种类。在本场景中，0x0用于标识上报火灾等级和温度的消息，0x1用于标识只上报温度的消息，0x2用于标识上报描述信息（字符串类型）的消息。messageId、数据类型、长度、默认值、偏移值的说明可参考1。

图 3-34 插件开发-添加字段 messageId(0x2)

2. 添加**other_info**字段，表示字符串类型的描述信息。在本场景中，字符串类型的字段数据类型选择“string”，“长度”配置 6个字节。字段名、默认值、偏移值的说明可参考**2**。

图 3-35 插件开发-添加字段 other_info

步骤3 单击“新增消息”，新增“other_info2”消息名，配置数据上报消息，上报可变长度字符串类型的描述信息。配置此步骤的主要目的是，将设备上传的可变长度字符串二进制码流消息解码成JSON格式，以便物联网平台理解。配置示例如下：

- 消息名：other_info2
- 消息类型：数据上报
- 添加响应字段：是。添加响应字段后，物联网平台在收到设备上报的数据后，会下发用户设置的响应数据到设备。
- 响应数据：AAAA0000（默认）

图 3-36 插件开发-新增消息 other_info2

1. 添加messageId字段，表示消息种类。在本场景中，0x0用于标识上报火灾等级和温度的消息，0x1用于标识只上报温度的消息，0x3用于标识上报描述信息（可变长度字符串类型）的消息。messageId、数据类型、长度、默认值、偏移值的说明可参考1。

图 3-37 插件开发-添加字段 messageId(0x3)

2. 添加**length**字段，表示可变字符串长度。“数据类型”根据可变长度字符串的长度进行配置，此场景可变字符串长度在255以内，配置为“int8u”。长度、默认值、偏移值的说明可参考**2**。

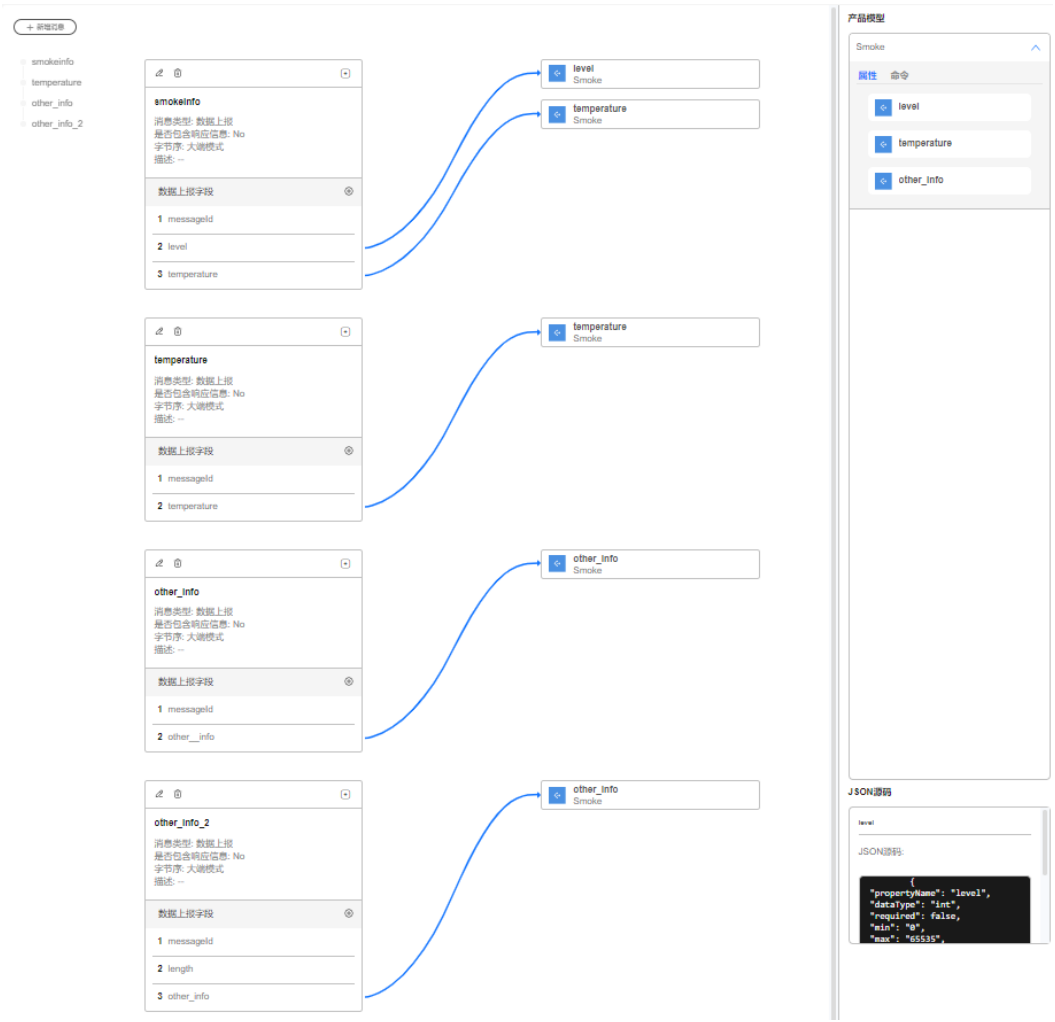
图 3-38 插件开发-添加字段 length

3. 添加**other_info**字段，数据类型选择“varstring”，表示可变长度字符串类型的描述信息。“长度关联字段”选择“length”，表示当前可变长字符串的长度由上报的length的值决定。“掩码”默认为“0xff”，用来计算该字段实际生效的长度，例如：“长度关联字段”length的值为5，其对应的二进制为：00000101，此时若掩码为0xff，对应的二进制为：11111111，那么两者进行“与“运算之后的结果为00000101，即十进制的5，那么该字段实际生效的长度为5个字节。如上报数据为03051234567890，表示当前上报的数据对应的是messageId为03的的message，可变长字符串长度为5个字节，可变长参数other_info对应的码流为1234567890。

图 3-39 插件开发-添加字段 other_info 为 varstring

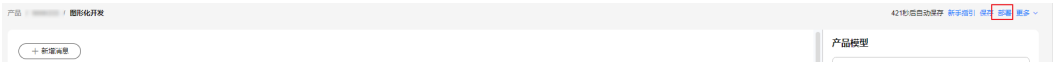
步骤4 拖动右侧“设备模型”区域的属性字段，与数据上报消息的相应字段建立映射关系。

图 3-40 插件开发-数据上报字段映射关系



步骤5 单击“保存”，并在插件保存成功后单击“部署”，将编解码插件部署到物联网平台。

图 3-41 插件开发-部署插件



----结束

调测编解码插件

步骤1 在烟感产品的开发空间，选择“在线调试”，并单击“**新增测试设备**”。

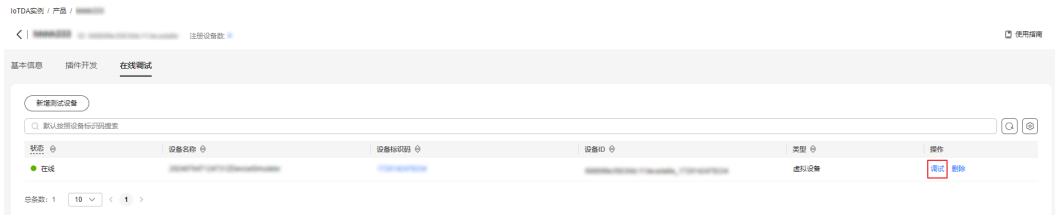
步骤2 用户可根据自己的业务场景，选择使用真实设备或者虚拟设备进行调测。具体调测步骤请参考**在线调试**。本文以虚拟设备为例，调测编解码插件。

在弹出的“新增测试设备”窗口，选择“虚拟设备”，单击“确定”，创建一个虚拟设备。虚拟设备名称包含 “DeviceSimulator”字样，每款产品下只能创建一个虚拟设备。

图 3-42 在线调试-创建虚拟设备

步骤3 单击“调试”，进入调试界面。

图 3-43 在线调试-进入调试

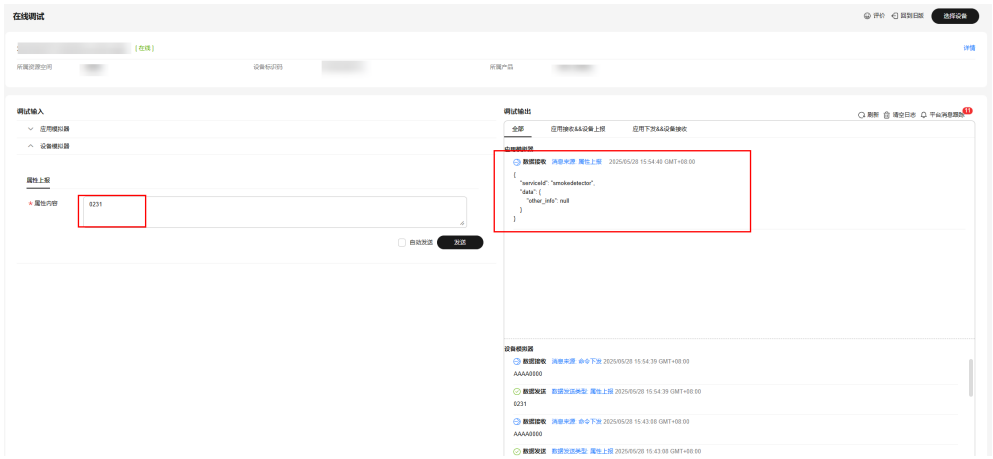


步骤4 使用设备模拟器上报字符串类型的描述信息。

十六进制码流示例：0231。02表示messageId，此消息上报字符串类型的描述信息；31表示描述信息，长度为1个字节。

在“应用模拟器”区域查看数据上报的结果：{other_info=null}。描述信息不足6个字节，编解码插件无法解析。

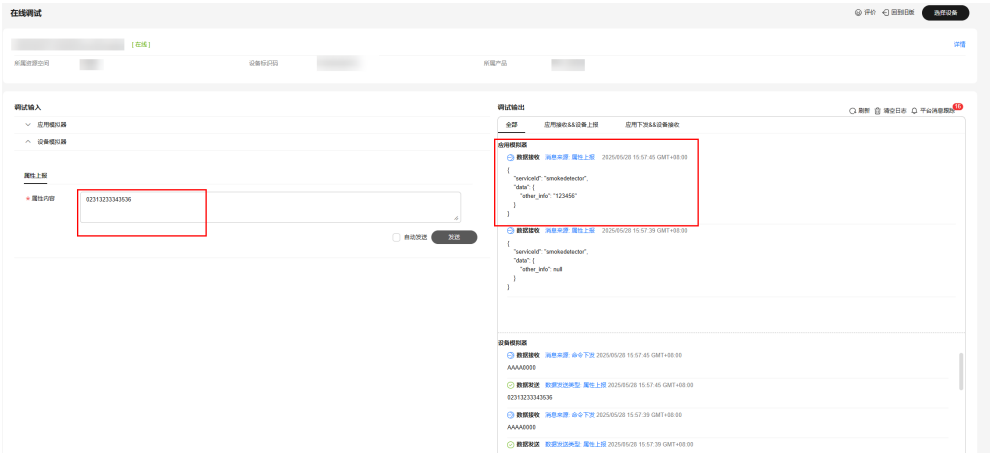
图 3-44 在线调试-模拟数据上报 other_info 长度不足



十六进制码流示例：02313233343536。02表示messageId，此消息上报字符串类型的描述信息；313233343536表示描述信息，长度为6个字节。

在“应用模拟器”区域查看数据上报的结果：{other_info=123456}。描述信息长度为6个字节，编解码插件解析成功。

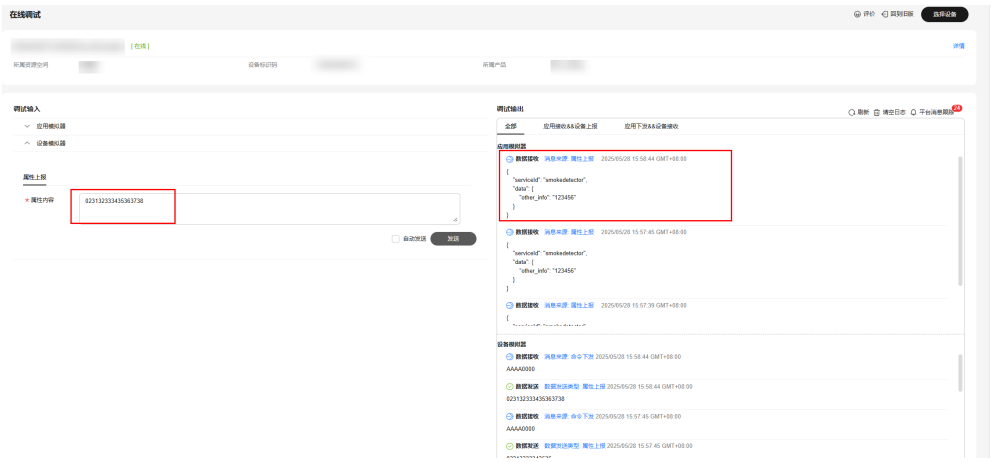
图 3-45 在线调试-模拟数据上报 other_info 长度合适



十六进制码流示例：023132333435363738。02表示messageId，此消息上报字符串类型的描述信息；3132333435363738表示描述信息，长度为8个字节。

在“应用模拟器”区域查看数据上报的结果：{other_info=123456}。描述信息长度超过6个字节，编解码插件截取前6个字节进行解析。

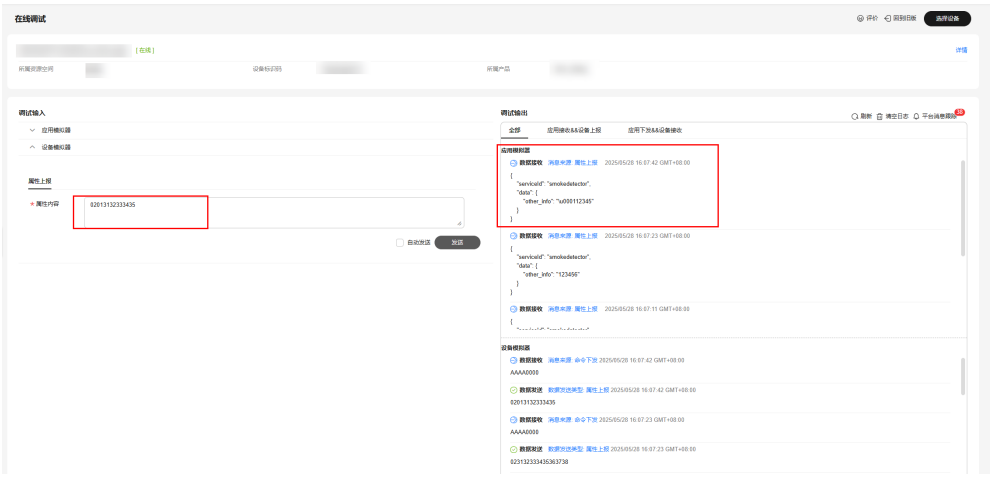
图 3-46 在线调试-模拟数据上报 other_info 长度过长



十六进制码流示例：02013132333435。02表示messageId，此消息上报字符串类型的描述信息；013132333435表示描述信息，长度为6个字节。

在“应用模拟器”区域查看数据上报的结果：{other_info=\u000112345}。01在ASCII码表里表示“标题开始”，无法用具体字符表示，因此编解码插件解析为\u0001。

图 3-47 在线调试-模拟数据上报 other_info ASCII 码

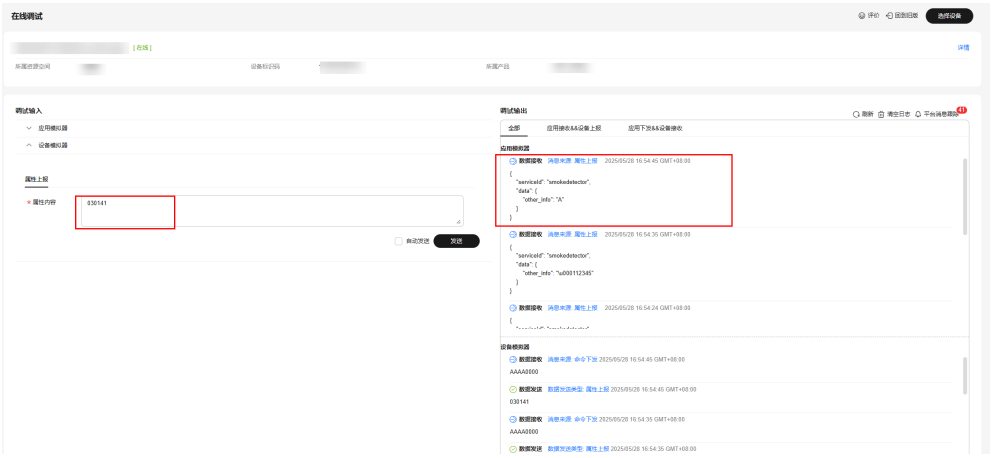


步骤5 使用设备模拟器上报可变长度字符串类型的描述信息。

十六进制码流示例：030141。03表示messageId，此消息上报可变长度字符串类型的描述信息；01表示描述信息长度；41表示描述信息，长度为1个字节。

在“应用模拟器”区域查看数据上报的结果：{other_info=A}。41是A的十六进制ASCII码。

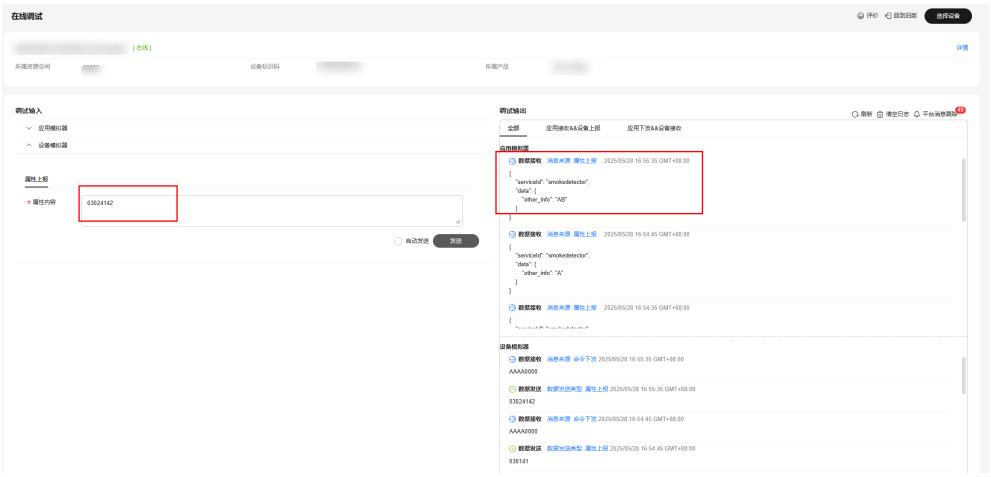
图 3-48 在线调试-模拟数据上报 other_info 可变长字符串 1



十六进制码流示例：03024142。03表示messageId，此消息上报可变长度字符串类型的描述信息；02表示描述信息的长度；4142表示描述信息，长度为2个字节。

在“应用模拟器”区域查看数据上报的结果：{other_info=AB}。4142是AB的十六进制ASCII码。

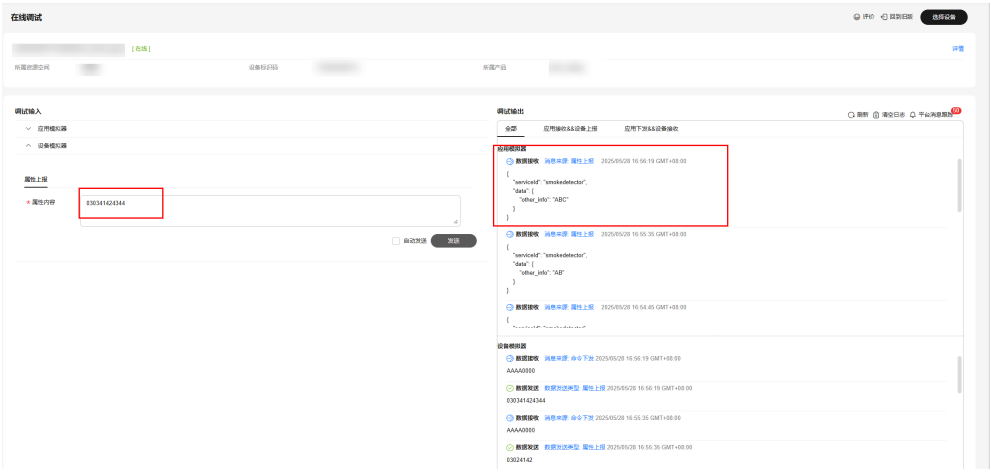
图 3-49 在线调试-模拟数据上报 other_info 可变长字符串 2



十六进制码流示例：030341424344。03表示messageId，此消息上报可变长度字符串类型的描述信息；03表示描述信息长度；41424344表示描述信息，长度为4个字节。

在“应用模拟器”区域查看数据上报的结果：{other_info=ABC}。描述信息长度超过3个字节，编解码插件截取前3个字节进行解析，414243是ABC的十六进制ASCII码。

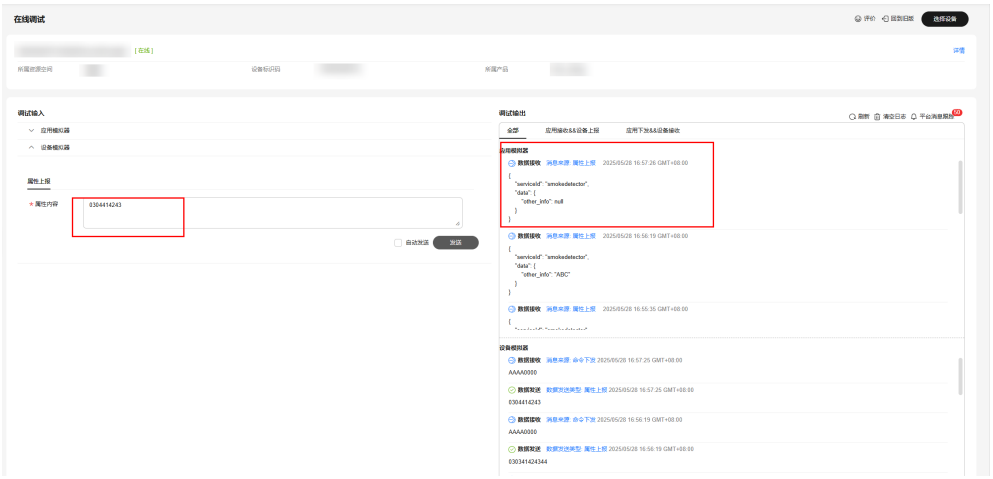
图 3-50 在线调试-模拟数据上报 other_info 可变长字符串 3



十六进制码流示例：0304414243。03表示messageId，此消息上报可变长度字符串类型的描述信息；04表示字符串长度；414243表示描述信息，长度为4个字节。

在“应用模拟器”区域查看数据上报的结果：{other_info=null}。描述信息长度不足4个字节，编解码插件解析失败。

图 3-51 在线调试-模拟数据上报 other_info 可变长字符串 4



----结束

总结

- 当数据类型为字符串或可变长度字符串时，插件是按照ASCII码进行编解码的：上报数据时，将16进制码流解码为对应字符串，比如：21解析为“!”、31解析为“1”、41解析为“A”；下发命令时，将字符串编码对应的16进制码流，比如：“！”编码为21，“1”编码为31，“A”编码为41。
- 当某字段的数据类型为可变长度字符串时，该字段需要关联长度字段，长度字段的数据类型必须为int。
- 针对可变长度字符串，命令下发和数据上报的编解码插件开发方式相同。
- 图形化开发的编解码插件使用ASCII码16进制的标准表对字符串和可变长度字符串进行编解码。解码时（数据上报），如果解析结果无法使用具体字符表示，如：标题开始、正文开始、正文结束等，则使用\u+2字节码流值表示（例如：01解析为\u0001，02解析为\u0002）；如果解析结果可以使用具体字符表示，则使用具体字符。

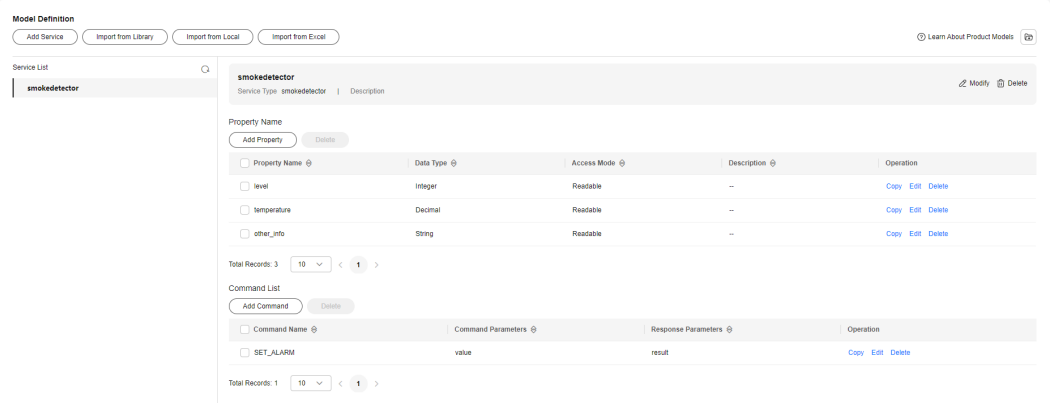
数组及可变长数组数据类型

如果该烟感设备需要支持描述信息上报功能，描述信息描述信息支持数组和可变长度数组两种类型，则按照以下步骤创建消息。

产品模型定义

在烟感产品的开发空间完成产品模型定义。

图 3-52 模型定义-smokedetector 携带 other_info



编解码插件开发

步骤1 在烟感产品的开发空间，选择“插件开发”，单击“图形化开发”。

步骤2 单击“新增消息”，新增消息“other_info”，上报数组类型的描述信息。配置此步骤的主要目的是，将设备上传的数组二进制码流消息解码成JSON格式，以便物联网平台理解。配置示例如下：

- 消息名：other_info
- 消息类型：数据上报
- 添加响应字段：是。添加响应字段后，物联网平台在收到设备上报的数据后，会下发用户设置的响应数据到设备。
- 响应数据：AAAA0000（默认）

图 3-53 插件开发-新增消息 other_info

1. 单击“添加字段”，添加messageId字段，表示消息种类。在本场景中，0x0用于标识上报火灾等级和温度的消息，0x1用于标识只上报温度的消息，0x2用于标识上报描述信息（数组类型）的消息。messageId、数据类型、长度、默认值、偏移值的说明可参考1。

图 3-54 插件开发-添加字段 messageId(0x2)

2. 添加**other_info**字段，“数据类型”选择“array”，表示数组类型的描述信息。在本场景中，“长度”配置为5个字节。字段名、默认值、偏移值的说明可参考2。

图 3-55 插件开发-添加字段 other_info 为 array

步骤3 单击“新增消息”，新增“other_info2”消息，上报可变长度数组类型的描述信息。配置此步骤的主要目的是，将设备上传的可变长度数组二进制码流消息解码成JSON格式，以便物联网平台理解。配置示例如下：

- 消息名：other_info2
- 消息类型：数据上报
- 添加响应字段：是。添加响应字段后，物联网平台在收到设备上报的数据后，会下发用户设置的响应数据到设备。
- 响应数据：AAAA0000（默认）

图 3-56 插件开发-新增消息 other_info2

1. 单击“添加字段”，添加messageId字段，表示消息种类。在本场景中，0x0用于标识上报火灾等级和温度的消息，0x1用于标识只上报温度的消息，0x3用于标识上报描述信息（可变长度数组类型）的消息。messageId、数据类型、长度、默认值、偏移值的说明可参考1。

图 3-57 插件开发-添加字段 messageId(0x3)

2. 添加length字段，表示数组长度。“数据类型”根据可变长度数组的长度进行配置，长度在255以内，配置为“int8u”。长度、默认值、偏移值的说明可参考2。

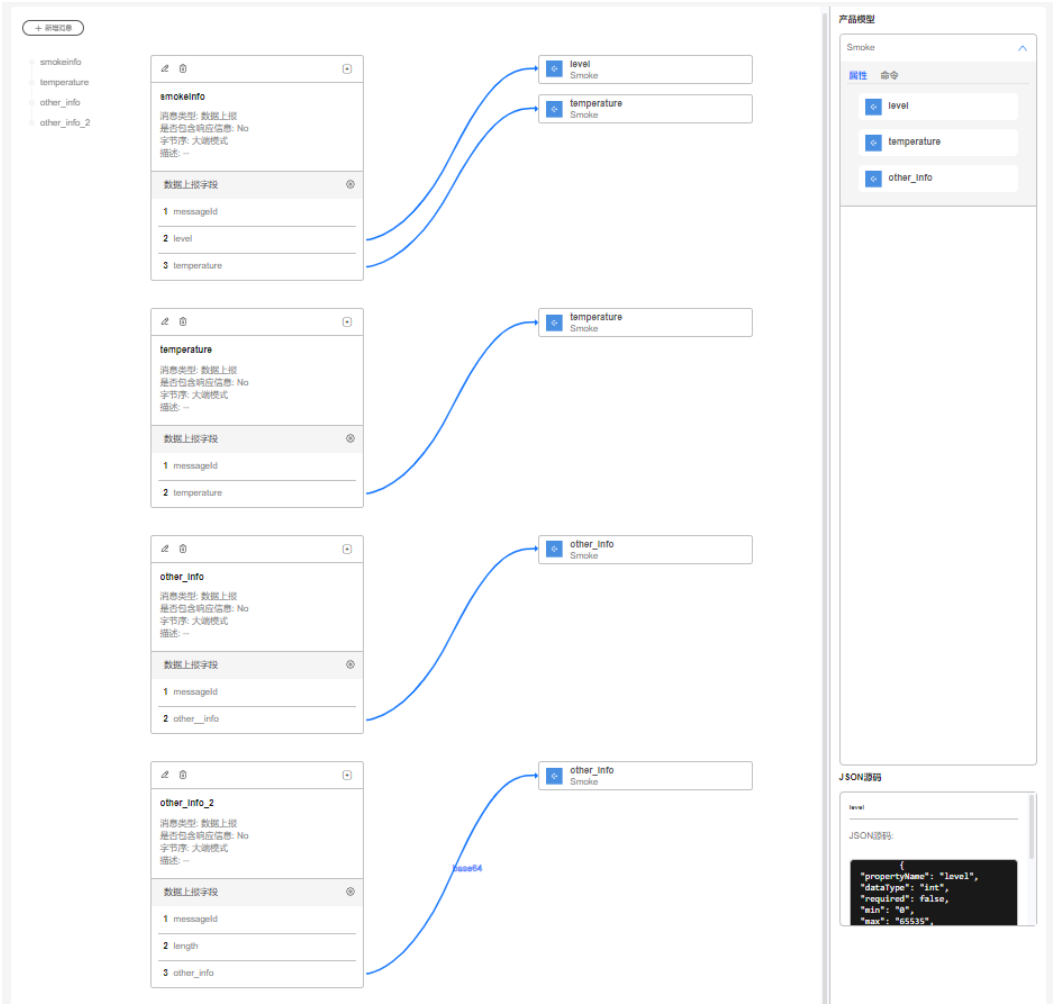
图 3-58 插件开发-添加字段 length

3. 添加**other_info**字段，数据类型选择“variant”，表示可变长度数组类型的描述信息。“长度关联字段”选择“length”，表示当前可变长数组的长度由上报的length的值决定。“掩码”默认为“0xff”，用来计算该数组实际生效的长度，例如：“长度关联字段”length的值为5，其对应的二进制为：00000101，此时若掩码为0xff，对应的二进制为：11111111，那么两者进行“与“运算之后的结果为00000101，即十进制的5，那么该数组实际生效的长度为5。如上报数据为03051234567890，表示当前上报的数据对应的是messageId为03的的message，可变长数组长度为5，可变长参数other_info对应的码流为1234567890。

图 3-59 插件开发-添加字段 other_info 为 variant

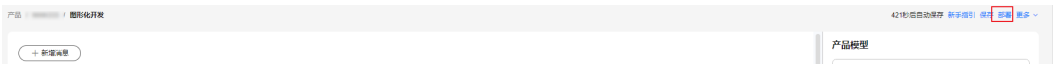
步骤4 拖动右侧“设备模型”区域的属性字段，与数据上报消息的相应字段建立映射关系。

图 3-60 插件开发-数据上报字段映射关系 other_info 为 varint



步骤5 单击“保存”，并在插件保存成功后单击“部署”，将编解码插件部署到物联网平台。

图 3-61 插件开发-部署插件



----结束

调测编解码插件

步骤1 在烟感产品的开发空间，选择“在线调试”，并单击“**新增测试设备**”。

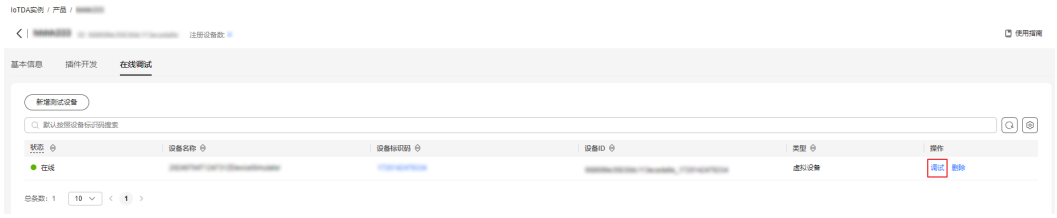
步骤2 用户可根据自己的业务场景，选择使用真实设备或者虚拟设备进行调测。具体调测步骤请参考**在线调试**。本文以虚拟设备为例，调测编解码插件。

在弹出的“新增测试设备”窗口，选择“虚拟设备”，单击“确定”，创建一个虚拟设备。虚拟设备名称包含 “DeviceSimulator”字样，每款产品下只能创建一个虚拟设备。

图 3-62 在线调试-创建虚拟设备

步骤3 单击“调试”，进入调试界面。

图 3-63 在线调试-进入调试

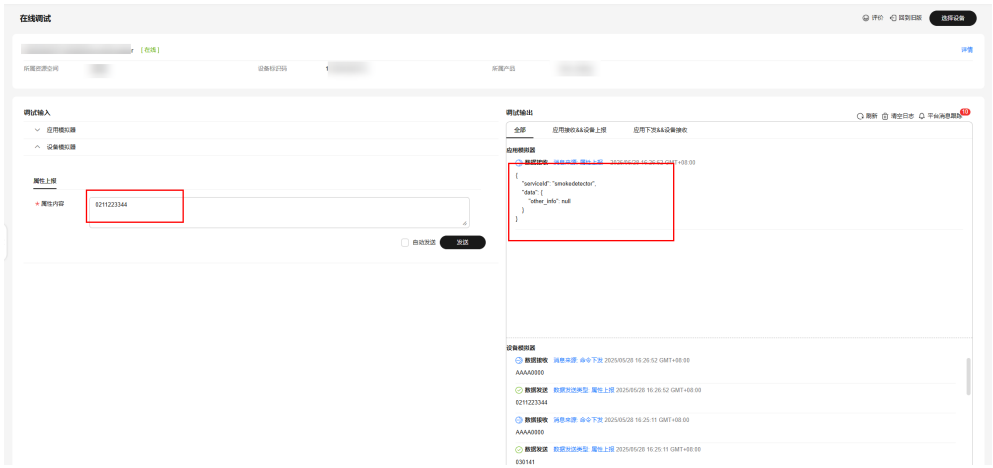


步骤4 使用设备模拟器上报数组类型的描述信息。

十六进制码流示例：0211223344。02表示messageId，此消息上报数组类型的描述信息；11223344表示描述信息，长度为4个字节。

在“应用模拟器”区域查看数据上报的结果：{other_info=null}。描述信息不足5个字节，编解码插件无法解析。

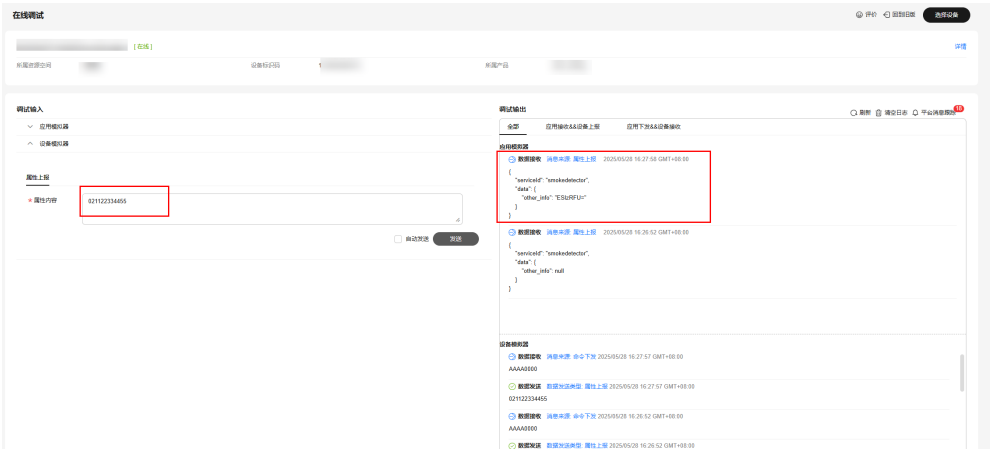
图 3-64 在线调试-模拟数据上报 other_info 数组 1



十六进制码流示例：021122334455。02表示messageId，此消息上报数组类型的描述信息；1122334455表示描述信息，长度为5个字节。

在“应用模拟器”区域查看数据上报的结果：{serviceId: smokedetector, data: {"other_info":"ESIzRFU="}}。描述信息长度为5个字节，编解码插件解析成功。

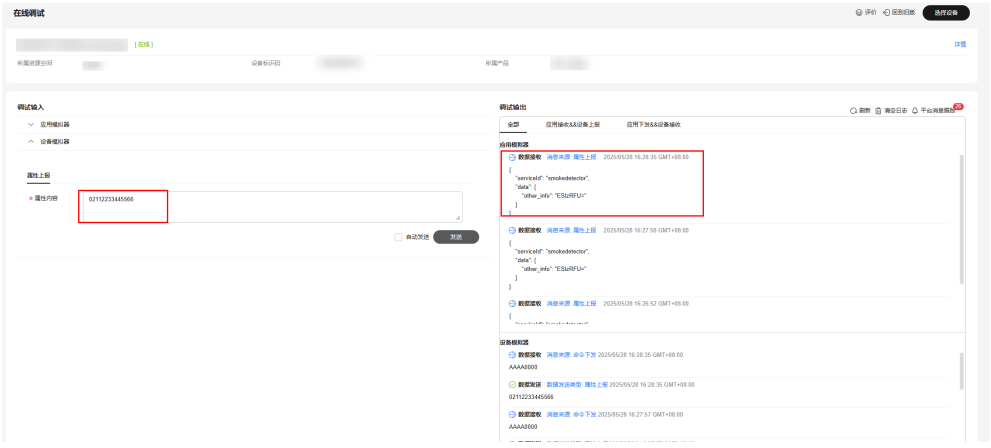
图 3-65 在线调试-模拟数据上报 other_info 数组 2



十六进制码流示例：02112233445566。02表示messageId，此消息上报数组类型的描述信息；112233445566表示描述信息，长度为6个字节。

在“应用模拟器”区域查看数据上报的结果：{serviceId: smokedetector, data: {"other_info":"ESIzRFU="}}。描述信息长度超过5个字节，编解码插件截取前5个字节进行解析。

图 3-66 在线调试-模拟数据上报 other_info 数组 3

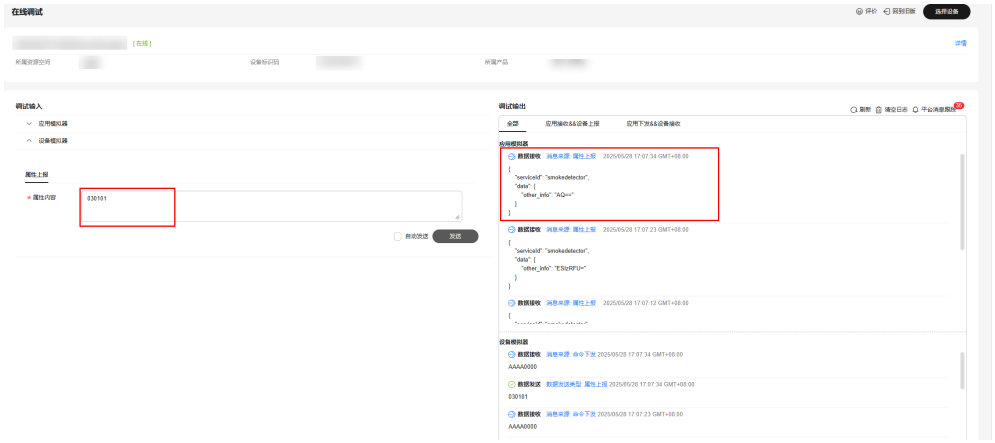


步骤5 使用设备模拟器上报可变长度数组类型的描述信息。

十六进制码流示例：030101。03表示messageId，此消息上报可变长度数组类型的描述信息；01表示描述信息长度（1个字节），长度为1个字节；01表示描述信息，长度为1个字节。

在“应用模拟器”区域查看数据上报的结果：{serviceId: smokedetector, data: {"other_info":"AQ=="}}。AQ==是01经过base64编码后的值。

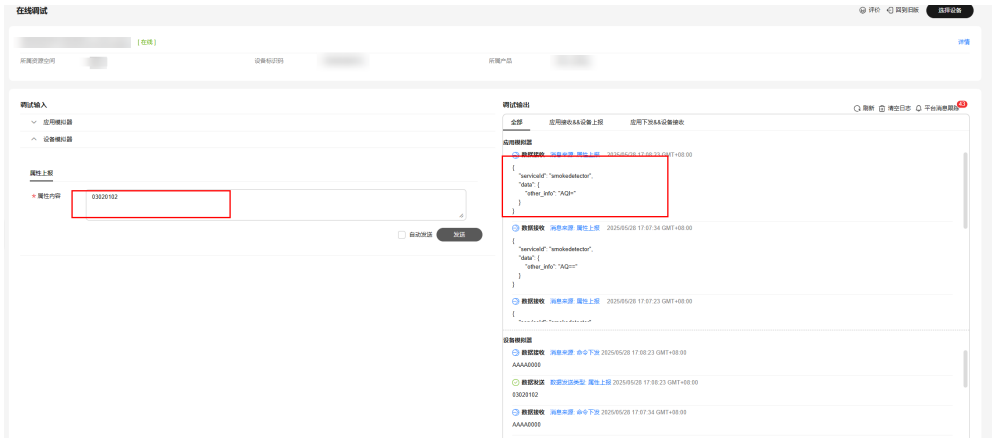
图 3-67 在线调试-模拟数据上报 other_info 可变长数组 1



十六进制码流示例：03020102。03表示messageId，此消息上报可变长度数组类型的描述信息；02表示描述信息长度（2个字节），长度为1个字节；0102表示描述信息，长度为2个字节。

在“应用模拟器”区域查看数据上报的结果：{serviceId: smokedetector, data: {"other_info":"AQI="}}。AQI=是01经过base64编码后的值。

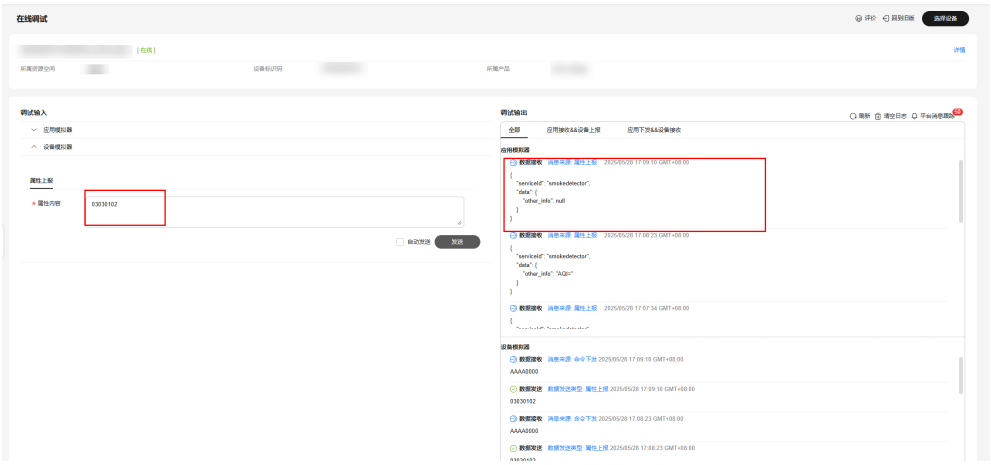
图 3-68 在线调试-模拟数据上报 other_info 可变长数组 2



十六进制码流示例：03030102。03表示messageId，此消息上报可变长度数组类型的描述信息；03表示描述信息长度（3个字节），长度为1个字节；0102表示描述信息，长度为2个字节。

在“应用模拟器”区域查看数据上报的结果：{other_info=null}。描述信息长度不足3个字节，编解码插件解析失败。

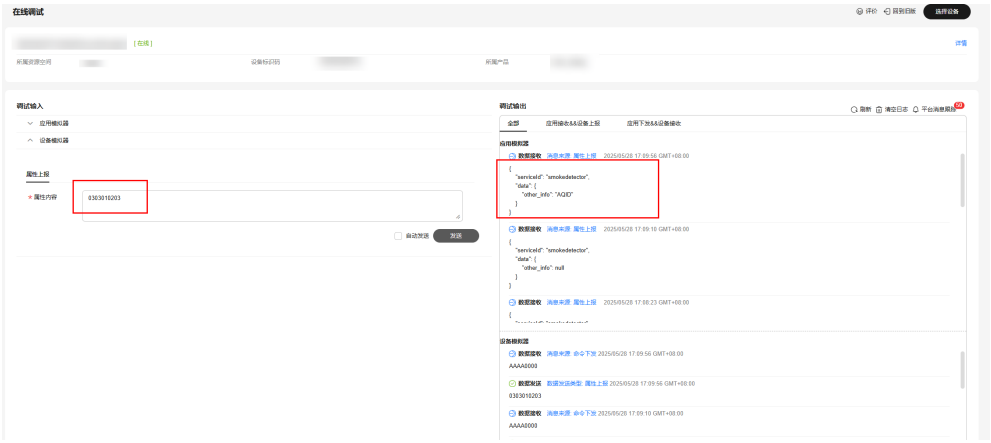
图 3-69 在线调试-模拟数据上报 other_info 可变长数组 3



十六进制码流示例：0303010203。03表示messageId，此消息上报可变长度数组类型的描述信息；03表示描述信息长度（3个字节），长度为1个字节；010203表示描述信息，长度为3个字节。

在“应用模拟器”区域查看数据上报的结果：{serviceId: smokedetector, data: {"other_info":"AQID"}}。AQID是010203经过base64编码后的值。

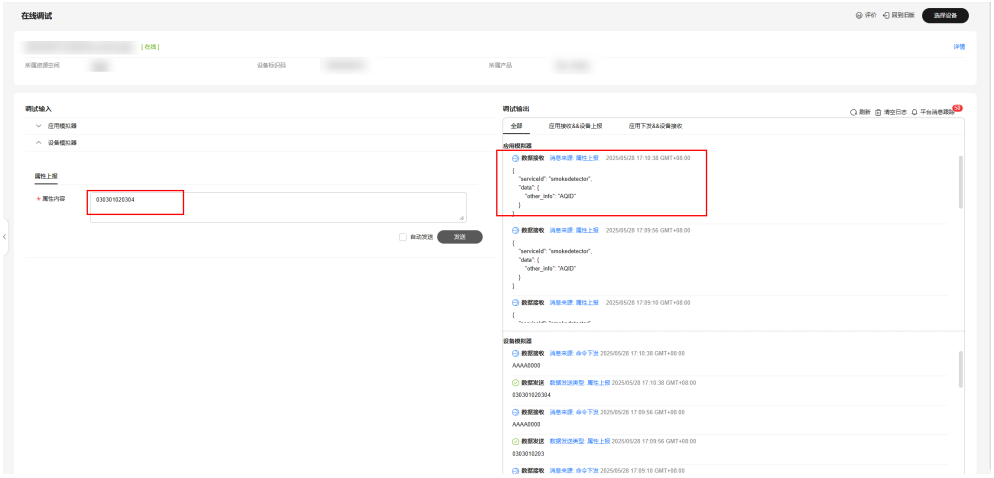
图 3-70 在线调试-模拟数据上报 other_info 可变长数组 4



十六进制码流示例：030301020304。03表示messageId，此消息上报可变长度数组类型的描述信息；03表示描述信息长度（3个字节），长度为1个字节；01020304表示描述信息，长度为4个字节。

在“应用模拟器”区域查看数据上报的结果：{other_info=AQID}。描述信息长度超过3个字节，编解码插件截取前3个字节进行解析，AQID是010203经过base64编码后的值。

图 3-71 在线调试-模拟数据上报 other_info 可变长数组 5



----结束

base64编码方式说明

base64编码方式会把3个8位字节（3*8=24）转化为4个6位字节（4*6=24），并在每个6位字节前补两个0，构成4个8位字节的形式。如果要进行编码的码流不足3个字节，则在码流后用0填充，使用0填充的字节经编码输出的字符为“=”。

base64可以将16进制码流当做字符或者数值进行编码，两种方式获得的编码结果不同。以16进制码流01为例进行说明：

- 把01当作字符，不足3个字符，补1个0，得到010。通过查询ASCII码表，将字符转换为8位二进制数，即：0转换为00110000、1转换为00110001，因此010可以转换为001100000011000100110000（3*8=24）。再转换为4个6位字节：001100、000011、000100、110000，并在每个6位字节前补两个0，得到：00001100、00000011、00000100、00110000。这4个8位字节对应的10进制数分别为12、3、4、48，通过查询base64编码表，获得M（12）、D（3）、E（4），由于3个字符中，最后一个字符通过补0获得，因此第4个8位字节使用“=”表示。最终，把01当做字符，通过base64编码得到MDE=。
- 把01当作数值（即1），不足3个字符，补两个0，得到100。将数值转换为8位2进制数，即：0转换为00000000、1转换为00000001，因此100可以转换为000000010000000000000000（3*8=24）。再转换为4个6位字节：000000、010000、000000、000000，并在每个6位字节前补两个0，得到：00000000、00010000、00000000、00000000。这4个8位字节对应的10进制数分别为：0、16、0、0，通过查询base64编码表，获得A（0）、Q（16），由于3个数值中，最后两个数值通过补0获得，因此第3、4个8位字节使用“=”表示。最终，把01当作数值，通过base64编码得到AQ==。

总结

- 当数据类型为数组或可变长度数组时，插件是按照base64进行编解码的：上报数据时，将16进制码流进行base64编码，比如：01编码为“AQ==”；命令下发时，将字符进行base64解码，比如：“AQ==”解码为01。
- 当某字段的数据类型为可变长度数组时，该字段需要关联长度字段，长度字段的数据类型必须为int。
- 针对可变长度数组，命令下发和数据上报的编解码插件开发方式相同。

- 图形化开发的编解码插件使用base64进行编码时，是将16进制码流当做**数值**进行编码。

3.2.4.3 使用 JavaScript 开发插件

物联网平台支持JavaScript脚本编解码的功能，根据您提交的脚本文件，实现设备二进制格式与JSON格式相互转换或JSON格式之间的转换。本文以烟感设备为例，介绍如何开发一个支持设备属性上报和命令下发的JavaScript编解码脚本，并介绍JavaScript脚本开发编解码插件的格式转换要求和调试方法。

说明

2024年12月1日后新用户不再提供JavaScript插件功能，推荐使用FunctionGraph进行JavaScript脚本编写，详细请参考**FunctionGraph开发说明**。

说明

- JavaScript语法规则需要遵循**ECMAScript 5.1规范**。
- 脚本编解码插件只支持es6的let和const，其他不支持，如箭头函数等。
- JavaScript脚本大小不能超过1M。
- 产品部署JavaScript脚本插件后，该产品下所有设备的上下行数据都会进行JavaScript脚本解析。开发者实现JavaScript插件时需要注意实现设备所有的上下行场景。
- 上行数据JavaScript解码后的JSON数据需要符合平台的格式要求，具体格式要求见： **数据解码格式定义** 。
- 平台下行指令的JSON格式定义见：**数据编码格式定义**，使用JavaScript编码时需要根据平台对应的JSON格式转换为对应的二进制码流或JSON。
- 脚本编辑提供自动保存功能，每次保存的间隔时间为10秒，在脚本输入框右上角勾选自动保存即可开启此功能。

烟感设备样例

场景说明

有一款烟感设备，具有如下特征：

- 具有烟雾报警功能（火灾等级）和温度上报功能。
- 支持远程控制命令，可远程打开报警功能。比如火灾现场温度，远程打开烟雾报警，提醒住户疏散。
- 该款烟感设备，设备能力比较弱，无法按照设备接口定义的JSON格式上报数据，只能上报简单的二进制数据。

产品模型定义

在烟感产品的开发空间，完成**产品模型**定义。

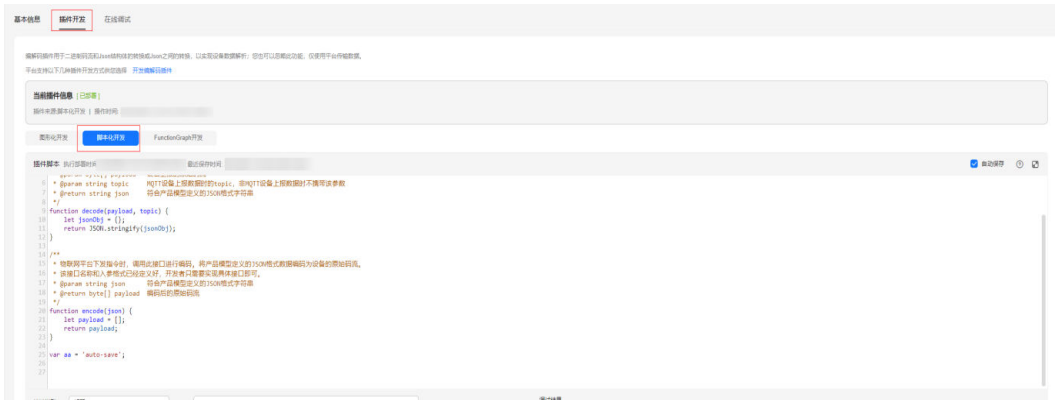
- level：火灾级别，用于表示火灾的严重程度。
- temperature：温度，用于表示火灾现场温度。
- SET_ALARM：打开或关闭告警命令，value=0表示关闭，value=1表示打开。

图 3-72 模型定义-smokedetector

编解码插件开发

步骤1 在烟感产品的详情页面，选择“插件开发”，单击“脚本化开发”。

图 3-73 插件开发-脚本化开发



步骤2 编写脚本，实现二进制数据到JSON数据的转换。脚本需要实现如下两个方法：

- decode：将设备上报的二进制数据转换为平台产品模型中定义的JSON格式。具体的JSON格式要求见： **数据解码格式定义** 。
- encode：平台有下行数据发送给设备时，将平台的JSON格式数据转换为设备支持的二进制格式。平台的JSON格式见：**数据编码格式定义**。

针对当前烟感产品实现的JavaScript样例如下：

```
//上行消息类型
var MSG_TYPE_PROPERTIES_REPORT = 'properties_report'; //设备属性上报
var MSG_TYPE_COMMAND_RSP = 'command_response'; //设备返回命令响应
var MSG_TYPE_PROPERTIES_SET_RSP = 'properties_set_response'; //设备返回属性设置响应
var MSG_TYPE_PROPERTIES_GET_RSP = 'properties_get_response'; //设备返回属性查询响应
var MSG_TYPE_MESSAGE_UP = 'message_up'; //设备消息上报
//下行消息类型
var MSG_TYPE_COMMANDS = 'commands'; //平台命令下发
var MSG_TYPE_PROPERTIES_SET = 'properties_set'; //平台下发属性设置请求
var MSG_TYPE_PROPERTIES_GET = 'properties_get'; //平台下发属性查询请求
var MSG_TYPE_MESSAGE_DOWN = 'messages'; //平台消息下发
//MQTT设备上行消息，topic同消息类型的映射表
var TOPIC_REG_EXP = {
    'properties_report': new RegExp('\\$oc/devices/(\\S+)/sys/properties/report'),
    'properties_set_response': new RegExp('\\$oc/devices/(\\S+)/sys/properties/set/response/request_id=(\\S
+)'),
    'properties_get_response': new RegExp('\\$oc/devices/(\\S+)/sys/properties/get/response/request_id=(\\S
+)'),
    'command_response': new RegExp('\\$oc/devices/(\\S+)/sys/commands/response/request_id=(\\S+)'),
```

```
    'message_up': new RegExp('\\$oc/devices/(\\S+)/sys/messages/up')
};
/*
示例：烟感设备上报属性和回复命令响应时，携带的是二进制码流，通过javascript脚本将二进制码流数据解码为符合产品模型定义的json格式数据
传入参数：
  payload:[0x00, 0x50, 0x00, 0x5a]
  topic:$oc/devices/cf40f3c4-7152-41c6-a201-a2333122054a/sys/properties/report
输出结果:
  {"msg_type":"properties_report","services":[{"service_id":"smokerdector","properties":
{"level":80,"temperature":90}}]}
传入参数:
  payload: [0x02, 0x00, 0x00, 0x01]
  topic: $oc/devices/cf40f3c4-7152-41c6-a201-a2333122054a/sys/commands/response/
request_id=bf40f0c4-4022-41c6-a201-c5133122054a
输出结果:

{"msg_type":"command_response","result_code":0,"command_name":"SET_ALARM","service_id":"smokerdect
or","paras":{"value":"1"}}
*/
function decode(payload, topic) {
    var jsonObj = {};
    var msgType = '';
    //如果有topic参数，根据topic参数解析消息类型
    if (null != topic) {
        msgType = topicParse(topic);
    }
    //将payload通过0xFF进行与操作，获取其对应的补码
    var uint8Array = new Uint8Array(payload.length);
    for (var i = 0; i < payload.length; i++) {
        uint8Array[i] = payload[i] & 0xff;
    }
    var dataView = new DataView(uint8Array.buffer, 0);
    //判断是属性上报的话，将二进制数据转换为属性上报格式
    if (msgType == MSG_TYPE_PROPERTIES_REPORT) {
        //设置serviceId参数值，该参数值对应产品模型中的服务类型smokerdector
        var serviceId = 'smokerdector';
        //从码流中获取level值
        var level = dataView.getInt16(0);
        //从码流中获取temperature值
        var temperature  = dataView.getInt16(2);
        //转换为属性上报的JSON格式
        jsonObj = {"msg_type":"properties_report","services":[{"service_id":serviceId,"properties":
{"level":level,"temperature":temperature}}]};
    }else if (msgType == MSG_TYPE_COMMAND_RSP) { //判断是命令响应的话，将二进制数据转换为命令响应格式
        //设置serviceId参数值，该参数值对应产品模型中的服务类型smokerdector
        var serviceId = 'smokerdector';
        var command = dataView.getInt8(0); //从二进制码流中获取命令名ID
        var command_name = '';
        if (2 == command) {
            command_name = 'SET_ALARM';
        }
        var result_code  = dataView.getInt16(1); //从二进制码流中获取命令执行结果
        var value = dataView.getInt8(3); //从二进制码流中获取命令执行结果返回值
        //转换为命令响应的JSON格式
        jsonObj =
{"msg_type":"command_response","result_code":result_code,"command_name":command_name,"service_id":
serviceId,"paras":{"value":value}};
    }
    //转换为JSON格式的字符串数据
    return JSON.stringify(jsonObj);
}
/*
示例数据：命令下发时，通过javascript的encode方法将平台JSON格式的数据，编码为二进制码流
传入参数 ->
    {"msg_type":"commands","command_name":"SET_ALARM","service_id":"smokerdector","paras":
{"value":1}}
输出结果 ->
```

```
    [0x01,0x00, 0x00, 0x01]
*/
function encode(json) {
    //转换为JSON对象
    var jsonObj = JSON.parse(json);
    //获取消息类型
    var msgType = jsonObj.msg_type;
    var payload = [];
    //将JSON格式数据转换为二进制数据
    if (msgType == MSG_TYPE_COMMANDS) // 命令下发
    {
        payload = payload.concat(buffer_uint8(1)); // 标识命令下发
        if (jsonObj.command_name == 'SET_ALARM') {
            payload = payload.concat(buffer_uint8(0)); // 标识命令名称
        }
        var paras_value = jsonObj.paras.value;
        payload = payload.concat(buffer_int16(paras_value)); // 设置命令属性值
    }
    //返回编码后的二进制数据
    return payload;
}
//根据topic名称解析出消息类型
function topicParse(topic) {
    for(var type in TOPIC_REG_EXP){
        var pattern = TOPIC_REG_EXP[type];
        if (pattern.test(topic)) {
            return type;
        }
    }
    return '';
}
//将8位无符号整型转换为byte数组
function buffer_uint8(value) {
    var uint8Array = new Uint8Array(1);
    var dataView = new DataView(uint8Array.buffer);
    dataView.setUint8(0, value);
    return [].slice.call(uint8Array);
}
//将16位无符号整型转换为byte数组
function buffer_int16(value) {
    var uint8Array = new Uint8Array(2);
    var dataView = new DataView(uint8Array.buffer);
    dataView.setInt16(0, value);
    return [].slice.call(uint8Array);
}
//将32位无符号整型转换为byte数组
function buffer_int32(value) {
    var uint8Array = new Uint8Array(4);
    var dataView = new DataView(uint8Array.buffer);
    dataView.setInt32(0, value);
    return [].slice.call(uint8Array);
}
```

步骤3 在线调试脚本。脚本编辑完成后，在模拟输入下，选择模拟类型，输入模拟数据在线调试脚本。

1. 模拟设备上报属性数据时将二进制码流转换为JSON数据。
   - 在topic输入框中选择设备上报的topic：$oc/devices/{device_id}/sys/properties/report。
   - 选择模拟类型为“解码”，输入以下模拟的设备数据，然后单击“调试”。
     ```
     0050005a
     ```
   - 脚本编解码引擎会根据输入的参数和您提交javascript脚本中的decode方法，将二进制码流转换为JSON格式，并将调试结果显示在输入框中。

图 3-74 脚本化开发-调试解码

- 检查调试结果是否符合预期，如果不符合预期，则修改代码后重新进行调试。

2. 模拟将应用下发的命令编码为设备能识别的二进制码流。
   - 选择模拟类型为“编码”，输入需要模拟的命令下发格式，然后单击“调试”。

```
{
    "msg_type": "commands",
    "request_id": "42aa08ea-84c1-4025-a7b2-c1f6efe547c2",
    "command_name": "SET_ALARM",
    "service_id": "smokerdector",
    "paras": {
        "value": "1"
    }
}
```

   - 脚本编解码引擎会根据输入的参数和您提交javascript脚本中的encode方法，将JSON格式数据转换为二进制码流，并将调试结果显示在输入框中。

图 3-75 脚本化开发-调试编码

   - 检查调试结果是否符合预期，如果不符合预期，则修改代码后重新进行调试。

步骤4 部署脚本。确认脚本可以正确进行编解码后，单击“部署”，将该脚本提交到物联网平台，以供数据上下行时，物联网平台调用该脚本进行编解码。

图 3-76 脚本化开发-部署

步骤5 使用真实设备调试。正式使用脚本之前，请使用真实设备与物联网平台进行上下行消息通信，以验证物联网平台能顺利调用脚本，解析上下行数据。

----结束

JavaScript 编解码插件模板

以下为JavaScript编解码插件的模板，开发者需要按照平台提供的模板，实现对应的接口。

```
/**
 * 设备上报数据到物联网平台时调用此接口进行解码, 将设备的原始数据解码为符合产品模型定义的JSON格式数
```

```
据。
 * 该接口名称和入参已经定义好，开发者只需要实现具体接口即可。
 * @param byte[] payload   设备上报的原始码流
 * @param string topic     MQTT设备上报数据时的topic，非MQTT设备上报数据时不携带该参数
 * @return string json     符合产品模型定义的JSON格式字符串
 */
function decode(payload, topic) {
    var jsonObj = {};
    return JSON.stringify(jsonObj);
}

/**
 * 物联网平台下发指令时，调用此接口进行编码, 将产品模型定义的JSON格式数据编码为设备的原始码流。
 * 该接口名称和入参格式已经定义好，开发者只需要实现具体接口即可。
 * @param string json      符合产品模型定义的JSON格式字符串
 * @return byte[] payload  编码后的原始码流
 */
function encode(json) {
    var payload = [];
    return payload;
}
```

MQTT 设备接入时 JavaScript 编解码插件样例

以下为MQTT设备JavaScript编解码插件的样例，开发者可以根据样例实现对应场景的二进制和JSON格式的转换。

```
//上行消息类型
var MSG_TYPE_PROPERTIES_REPORT = 'properties_report'; //设备属性上报
var MSG_TYPE_COMMAND_RSP = 'command_response'; //设备返回命令响应
var MSG_TYPE_PROPERTIES_SET_RSP = 'properties_set_response'; //设备返回属性设置响应
var MSG_TYPE_PROPERTIES_GET_RSP = 'properties_get_response'; //设备返回属性查询响应
var MSG_TYPE_MESSAGE_UP = 'message_up'; //设备消息上报
//下行消息类型
var MSG_TYPE_COMMANDS = 'commands'; //平台命令下发
var MSG_TYPE_PROPERTIES_SET = 'properties_set'; //平台下发属性设置请求
var MSG_TYPE_PROPERTIES_GET = 'properties_get'; //平台下发属性查询请求
var MSG_TYPE_MESSAGE_DOWN = 'messages'; //平台消息下发
//MQTT设备上行消息，topic同消息类型的映射表
var TOPIC_REG_EXP = {
    'properties_report': new RegExp('\\$oc/devices/(\\S+)/sys/properties/report'),
    'properties_set_response': new RegExp('\\$oc/devices/(\\S+)/sys/properties/set/response/request_id=(\\S
+)'),
    'properties_get_response': new RegExp('\\$oc/devices/(\\S+)/sys/properties/get/response/request_id=(\\S
+)'),
    'command_response': new RegExp('\\$oc/devices/(\\S+)/sys/commands/response/request_id=(\\S+)'),
    'message_up': new RegExp('\\$oc/devices/(\\S+)/sys/messages/up')
};
/*
示例：烟感设备上报属性和回复命令响应时，携带的是二进制码流，通过javascript脚本将二进制码流数据解码为
符合产品模型定义的json格式数据
传入参数:
  payload:[0x00, 0x50, 0x00, 0x5a]
  topic:$oc/devices/cf40f3c4-7152-41c6-a201-a2333122054a/sys/properties/report
输出结果:
  {"msg_type":"properties_report","services":[{"service_id":"smokerdector","properties":
{"level":80,"temperature":90}}]}
传入参数:
  payload: [0x02, 0x00, 0x00, 0x01]
  topic: $oc/devices/cf40f3c4-7152-41c6-a201-a2333122054a/sys/commands/response/
request_id=bf40f0c4-4022-41c6-a201-c5133122054a
输出结果:

{"msg_type":"command_response","result_code":0,"command_name":"SET_ALARM","service_id":"smokerdect
or","paras":{"value":"1"}}
*/
function decode(payload, topic) {
    var jsonObj = {};
    var msgType = '';
```

```
    //如果有topic参数，根据topic参数解析消息类型
    if (null != topic) {
        msgType = topicParse(topic);
    }
    //将payload通过0xFF进行与操作，获取其对应的补码
    var uint8Array = new Uint8Array(payload.length);
    for (var i = 0; i < payload.length; i++) {
        uint8Array[i] = payload[i] & 0xff;
    }
    var dataView = new DataView(uint8Array.buffer, 0);
    //判断是属性上报的话，将二进制数据转换为属性上报格式
    if (msgType == MSG_TYPE_PROPERTIES_REPORT) {
        //设置serviceId参数值，该参数值对应产品模型中的服务类型smokerdector
        var serviceId = 'smokerdector';
        //从码流中获取level值
        var level = dataView.getInt16(0);
        //从码流中获取temperature值
        var temperature = dataView.getInt16(2);
        //转换为属性上报的JSON格式
        jsonObj = {
            "msg_type": "properties_report",
            "services": [{"service_id": serviceId, "properties": {"level": level, "temperature": temperature}}]
        };
    } else if (msgType == MSG_TYPE_COMMAND_RSP) { //判断是命令响应的话，将二进制数据转换为命令响应
格式
        //设置serviceId参数值，该参数值对应产品模型中的服务类型smokerdector
        var serviceId = 'smokerdector';
        var command = dataView.getInt8(0); //从二进制码流中获取命令名ID
        var command_name = '';
        if (2 == command) {
            command_name = 'SET_ALARM';
        }
        var result_code = dataView.getInt16(1); //从二进制码流中获取命令执行结果
        var value = dataView.getInt8(3); //从二进制码流中获取命令执行结果返回值
        //转换为命令响应的JSON格式
        jsonObj = {
            "msg_type": "command_response",
            "result_code": result_code,
            "command_name": command_name,
            "service_id": serviceId,
            "paras": {"value": value}
        };
    } else if (msgType == MSG_TYPE_PROPERTIES_SET_RSP) {
        //转换为属性设置响应的JSON格式
        //jsonObj = {"msg_type":"properties_set_response","result_code":0,"result_desc":"success"};
    } else if (msgType == MSG_TYPE_PROPERTIES_GET_RSP) {
        //转换为属性查询响应的JSON格式
        //jsonObj = {"msg_type":"properties_get_response","services":[{"service_id":"analog","properties":
{"PhV_phsA":"1","PhV_phsB":"2"}}]};
    } else if (msgType == MSG_TYPE_MESSAGE_UP) {
        //转换为消息上报的JSON格式
        //jsonObj = {"msg_type":"message_up","content":"hello"};
    }
    //转换为JSON格式的字符串数据
    return JSON.stringify(jsonObj);
}
/*
示例数据：命令下发时，通过javascript的encode方法将平台JSON格式的数据，编码为二进制码流
传入参数 ->
    {"msg_type":"commands","command_name":"SET_ALARM","service_id":"smokerdector","paras":
{"value":1}}
输出结果 ->
    [0x01,0x00, 0x00, 0x01]
*/
function encode(json) {
    //转换为JSON对象
    var jsonObj = JSON.parse(json);
    //获取消息类型
    var msgType = jsonObj.msg_type;
```

```
    var payload = [];
    //将JSON格式数据转换为二进制数据
    if (msgType == MSG_TYPE_COMMANDS) {// 命令下发
        //命令下发格式样例：
{"msg_type":"commands","command_name":"SET_ALARM","service_id":"smokerdector","paras":{"value":1}}
        //根据命令下发的格式转换为二进制码流
        payload = payload.concat(buffer_uint8(1)); // 标识命令下发
        if (jsonObj.command_name == 'SET_ALARM') {
            payload = payload.concat(buffer_uint8(0)); // 标识命令名称
        }
        var paras_value = jsonObj.paras.value;
        payload = payload.concat(buffer_int16(paras_value)); // 设置命令属性值
    } else if (msgType == MSG_TYPE_PROPERTIES_SET) {
        //属性设置格式样例：{"msg_type":"properties_set","services":[{"service_id":"Temperature","properties":
{"value":57}}]}
        //开发者有对应属性设置场景时需要根据该JSON格式转换为对应的二进制码流
    } else if (msgType == MSG_TYPE_PROPERTIES_GET) {
        //属性查询格式样例：{"msg_type":"properties_get","service_id":"Temperature"}
        //开发者有对应属性查询场景时需要根据该JSON格式转换为对应的二进制码流
    } else if (msgType == MSG_TYPE_MESSAGE_DOWN) {
        //消息下发格式样例：{"msg_type":"messages","content":"hello"}
        //开发者对应消息下发场景时需要根据该JSON格式转换为对应的二进制码流
    }
    //返回编码后的二进制数据
    return payload;
}
//根据topic名称解析出消息类型
function topicParse(topic) {
    for (var type in TOPIC_REG_EXP) {
        var pattern = TOPIC_REG_EXP[type];
        if (pattern.test(topic)) {
            return type;
        }
    }
    return '';
}
//将8位无符号整型转换为byte数组
function buffer_uint8(value) {
    var uint8Array = new Uint8Array(1);
    var dataView = new DataView(uint8Array.buffer);
    dataView.setUint8(0, value);
    return [].slice.call(uint8Array);
}

//将16位无符号整型转换为byte数组
function buffer_int16(value) {
    var uint8Array = new Uint8Array(2);
    var dataView = new DataView(uint8Array.buffer);
    dataView.setInt16(0, value);
    return [].slice.call(uint8Array);
}
//将32位无符号整型转换为byte数组
function buffer_int32(value) {
    var uint8Array = new Uint8Array(4);
    var dataView = new DataView(uint8Array.buffer);
    dataView.setInt32(0, value);
    return [].slice.call(uint8Array);
}
```

NB-IoT 设备接入时 JavaScript 编解码插件样例

以下为NB-IoT设备JavaScript编解码插件的样例，开发者可以根据样例实现NB-IoT设备数据上报和命令下发的编解码插件开发。

```
//上行消息类型
var MSG_TYPE_PROPERTIES_REPORT = 'properties_report'; //设备属性上报
var MSG_TYPE_COMMAND_RSP = 'command_response'; //设备返回命令响应
//下行消息类型
var MSG_TYPE_COMMANDS = 'commands'; //平台命令下发
```

```
var MSG_TYPE_PROPERTIES_REPORT_REPLY = 'properties_report_reply'; //设备属性上报的响应消息
//消息类型列表
var MSG_TYPE_LIST = {
    0: MSG_TYPE_PROPERTIES_REPORT,          //码流中0字节标识为设备属性上报
    1: MSG_TYPE_PROPERTIES_REPORT_REPLY,    //码流中1字节标识为设备属性上报的响应消息
    2: MSG_TYPE_COMMANDS,                   //码流中2字节标识为平台命令下发
    3: MSG_TYPE_COMMAND_RSP                 //码流中3字节标识为设备返回命令响应
};
/*
示例：烟感设备上报属性和回复命令响应时，携带的是二进制码流，通过javascript脚本将二进制码流数据解码为符合产品模型定义的json格式数据
传入参数：
  payload:[0x00, 0x00, 0x50, 0x00, 0x5a]
输出结果:
  {"msg_type":"properties_report","services":[{"service_id":"smokerdector","properties":{"level":80,"temperature":90}}]}
传入参数:
  payload: [0x03, 0x01, 0x00, 0x00, 0x01]
输出结果:
  {"msg_type":"command_response","request_id":1,"result_code":0,"paras":{"value":"1"}}
*/
function decode(payload, topic) {
    var jsonObj = {};
    //将payload通过0xFF进行与操作，获取其对应的补码
    var uint8Array = new Uint8Array(payload.length);
    for (var i = 0; i < payload.length; i++) {
        uint8Array[i] = payload[i] & 0xff;
    }
    var dataView = new DataView(uint8Array.buffer, 0);
    //从消息码流中的第一个字节获取消息类型
    var messageId = dataView.getInt8(0);
    //判断是属性上报的话，将二进制数据转换为属性上报格式
    if (MSG_TYPE_LIST[messageId] == MSG_TYPE_PROPERTIES_REPORT) {
        //设置serviceId参数值，该参数值对应产品模型中的服务类型smokerdector
        var serviceId = 'smokerdector';
        //从码流中获取level值
        var level = dataView.getInt16(1);
        //从码流中获取temperature值
        var temperature  = dataView.getInt16(3);
        //转换为属性上报的JSON格式
        jsonObj = {"msg_type":"properties_report","services":[{"service_id":serviceId,"properties":{"level":level,"temperature":temperature}}]};
    }else if (MSG_TYPE_LIST[messageId] == MSG_TYPE_COMMAND_RSP) { //判断是命令响应的话，将二进制数据转换为命令响应格式
        var requestId = dataView.getInt8(1);
        var result_code  = dataView.getInt16(2); //从二进制码流中获取命令执行结果
        var value = dataView.getInt8(4); //从二进制码流中获取命令执行结果返回值
        //转换为命令响应的JSON格式
        jsonObj = {"msg_type":"command_response","request_id":requestId,"result_code":result_code,"paras":{"value":value}};
    }
    //转换为JSON格式的字符串数据
    return JSON.stringify(jsonObj);
}
/*
示例数据：命令下发时，通过javascript的encode方法将平台JSON格式的数据，编码为二进制码流
传入参数 ->

{"msg_type":"commands","request_id":1,"command_name":"SET_ALARM","service_id":"smokerdector","paras":{"value":1}}
输出结果 ->
    [0x02, 0x00, 0x00, 0x00, 0x01]
示例数据：设备上报属性时返回响应消息，通过javascript的encode方法将平台JSON格式的数据，编码为二进制码流
传入参数 ->
    {"msg_type":"properties_report_reply","request":"000050005a","result_code":0}
输出结果 ->
    [0x01, 0x00]
*/
```

```
function encode(json) {
    //转换为JSON对象
    var jsonObj = JSON.parse(json);
    //获取消息类型
    var msgType = jsonObj.msg_type;
    var payload = [];
    //将JSON格式数据转换为二进制数据
    if (msgType == MSG_TYPE_COMMANDS) { // 命令下发
        payload = payload.concat(buffer_uint8(2)); // 标识命令下发
        payload = payload.concat(buffer_uint8(jsonObj.request_id)); // 命令ID
        if (jsonObj.command_name == 'SET_ALARM') {
            payload = payload.concat(buffer_uint8(0)); // 标识命令名称
        }
        var paras_value = jsonObj.paras.value;
        payload = payload.concat(buffer_int16(paras_value)); // 设置命令属性值
    } else if (msgType == MSG_TYPE_PROPERTIES_REPORT_REPLY) { // 设备属性上报的响应消息
        payload = payload.concat(buffer_uint8(1)); // 标识属性上报的响应消息
        if (0 == jsonObj.result_code) {
            payload = payload.concat(buffer_uint8(0)); // 标识属性上报消息处理成功
        }
    }
    //返回编码后的二进制数据
    return payload;
}
//将8位无符号整型转换为byte数组
function buffer_uint8(value) {
    var uint8Array = new Uint8Array(1);
    var dataView = new DataView(uint8Array.buffer);
    dataView.setUint8(0, value);
    return [].slice.call(uint8Array);
}
//将16位无符号整型转换为byte数组
function buffer_int16(value) {
    var uint8Array = new Uint8Array(2);
    var dataView = new DataView(uint8Array.buffer);
    dataView.setInt16(0, value);
    return [].slice.call(uint8Array);
}
//将32位无符号整型转换为byte数组
function buffer_int32(value) {
    var uint8Array = new Uint8Array(4);
    var dataView = new DataView(uint8Array.buffer);
    dataView.setInt32(0, value);
    return [].slice.call(uint8Array);
}
```

JavaScript 编解码插件格式要求

数据解码格式定义

数据解析场景，平台收到设备侧的数据时，平台会将设备侧payload中的二进制码流，通过decode方法传到javascript脚本，脚本的decode方法需要实现数据的解码，解码为平台能识别的产品模型中定义的JSON格式，平台对解析后的JSON要求如下：

- 设备属性上报

```
{
    "msg_type": "properties_report",
    "services": [{
        "service_id": "Battery",
        "properties": {
            "batteryLevel": 57
        },
        "event_time": "20151212T121212Z"
    }]
}
```

字段名	必选/可选	类型	参数描述
msg_type	必选	String	属性上报的消息类型，固定为：properties_report
services	必选	List<ServiceProperty>	设备服务数据列表（具体结构参考下表ServiceProperty定义表）。

ServiceProperty结构定义：

字段名	必选/可选	类型	参数描述
service_id	必选	String	设备的服务ID。
properties	必选	Object	设备服务的属性列表，具体字段在设备关联的产品模型中定义。
event_time	可选	String	设备采集数据UTC时间（格式：yyyyMMddTHHmmssZ），如：20161219T114920Z。 设备上报数据不带该参数或参数格式错误时，则数据上报时间以平台时间为准。

- 平台设置设备属性的响应消息

```
{
  "msg_type": "properties_set_response",
  "request_id": "42aa08ea-84c1-4025-a7b2-c1f6efe547c2",
  "result_code": 0,
  "result_desc": "success"
}
```

字段名	必选/可选	类型	参数描述
msg_type	必选	String	属性上报的消息类型，固定为：properties_set_response
request_id	可选	String	用于唯一标识这次请求，设备侧收到的消息带该参数时，响应消息需要将该参数值返回给平台。如果解码后的消息不带该字段，则以topic中带的request_id为准。
result_code	可选	Integer	命令的执行结果，0表示成功，其他表示失败。不带默认认为成功。
result_desc	可选	String	属性设置的响应描述。

- 平台查询设备属性的响应消息

```
{
"msg_type": "properties_get_response",
```

```
"request_id": "42aa08ea-84c1-4025-a7b2-c1f6efe547c2",
   "services": [
      {
         "service_id": "analog",
         "properties": {
            "PhV_phsA": "1",
            "PhV_phsB": "2"
         },
         "event_time": "20190606T121212Z"
      }
   ]
}
```

字段名	必选/可选	类型	参数描述
msg_type	必选	String	固定为：properties_get_response
request_id	可选	String	用于唯一标识这次请求，设备侧收到的消息带该参数时，响应消息需要将该参数值返回给平台。如果解码后的消息不带该字段，则以topic中带的request_id为准。
services	必选	List<ServiceProperty>	设备服务数据列表（具体结构参考下表ServiceProperty定义表）。

ServiceProperty结构定义：

字段名	必选/可选	类型	参数描述
service_id	必选	String	设备的服务ID。
properties	必选	Object	设备服务的属性列表，具体字段在设备关联的产品模型中定义。
event_time	可选	String	设备采集数据UTC时间（格式：yyyyMMddTHHmmssZ），如：20161219T114920Z。 设备上报数据不带该参数或参数格式错误时，则数据上报时间以平台时间为准。

- 平台命令下发的响应消息

```
{
  "msg_type": "command_response",
  "request_id": "42aa08ea-84c1-4025-a7b2-c1f6efe547c2",
   "result_code": 0,
  "command_name": "ON_OFF",
  "service_id": "WaterMeter",
  "paras": {
    "value": "1"
  }
}
```

字段名	必选/可选	类型	参数描述
msg_type	必选	String	固定为：command_response
request_id	可选	String	用于唯一标识这次请求，设备侧收到的消息带该参数时，响应消息需要将该参数值返回给平台。如果解码后的消息不带该字段，则以topic中带的request_id为准。
result_code	可选	Integer	标识命令的执行结果，0表示成功，其他表示失败。不带默认认为成功。
response_name	可选	String	命令的响应名称，在设备关联的产品模型中定义。
paras	可选	Object	命令的响应参数，具体字段在设备关联的产品模型中定义。

- 设备消息上报

```
{
  "msg_type": "message_up",
  "content": "hello"
}
```

字段名	必选/可选	类型	参数描述
msg_type	必选	String	固定为：message_up
content	可选	String	消息内容。

数据编码格式定义

数据解析场景，平台指令下发时，平台会将产品模型定义的JSON格式数据（如果不是该格式的数据则可能会导致编解码失败），通过encode方法传给javascript脚本，脚本的encode方法需要实现数据的编码将JSON格式的数据，编码为设备可以识别的二进制码流，编码时平台传递到脚本的JSON格式如下：

- 设备命令下发

```
{
  "msg_type": "commands",
   "request_id": "42aa08ea-84c1-4025-a7b2-c1f6efe547c2",
  "command_name": "ON_OFF",
  "service_id": "WaterMeter",
  "paras": {
    "value": 1
  }
}
```

字段名	必选/可选	类型	参数描述
msg_type	必选	String	固定为：commands
request_id	必选	String	用于唯一标识这次请求，该标识会通过topic下发给设备。

字段名	必选/可选	类型	参数描述
service_id	可选	String	设备的服务ID。
command_name	可选	String	设备命令名称，在设备关联的产品模型中定义。
paras	可选	Object	设备命令的执行参数，具体字段在设备关联的产品模型中定义。

- 平台设置设备属性

```
{
"msg_type": "properties_set",
"request_id": "42aa08ea-84c1-4025-a7b2-c1f6efe547c2",
    "services": [{
            "service_id": "Temperature",
            "properties": {
                "value": 57
            }
        },
        {
            "service_id": "Battery",
            "properties": {
                "level": 80
            }
        }
    ]
}
```

字段名	必选/可选	类型	参数描述
msg_type	必选	String	固定为：properties_set
request_id	必选	String	用于唯一标识这次请求，设备侧收到的消息带该参数时，响应消息需要将该参数值返回给平台。
services	必选	List<ServiceProperty>	设备服务数据列表。

ServiceProperty结构定义：

字段名	必选/可选	类型	参数描述
service_id	必选	String	设备的服务ID。
properties	必选	Object	设备服务的属性列表，具体字段在产品模型里定义。

- 平台查询设备属性

```
{
  "msg_type": "properties_get",
  "request_id": "42aa08ea-84c1-4025-a7b2-c1f6efe547c2",
```

```
  "service_id": "Temperature"
}
```

字段名	必选/可选	类型	参数描述
msg_type	必选	String	固定为：properties_get
request_id	必选	String	用于唯一标识这次请求，该标识会通过topic下发给设备。
service_id	可选	String	设备的服务ID。

- 属性上报的响应消息（NB-IoT设备接入时属性上报对应的响应）

```
{
  "msg_type": "properties_report_reply",
  "request": "213355656",
  "result_code": 0
}
```

字段名	必选/可选	类型	参数描述
msg_type	必选	String	固定为：properties_report_reply
request	可选	String	属性上报的BASE64编码字符串。
result_code	可选	Integer	属性上报的执行结果。
has_more	可选	Boolean	是否存在缓存命令。

- 设备消息下发

```
{
  "msg_type": "messages",
  "content": "hello"
}
```

字段名	必选/可选	类型	参数描述
msg_type	必选	String	固定为：messages
content	可选	String	消息下发的内容。

3.2.4.4 FunctionGraph 开发编解码插件

3.2.4.4.1 FunctionGraph 开发说明

概述

物联网平台支持使用FunctionGraph编解码的功能，根据您提交的脚本文件，实现设备二进制格式与JSON格式相互转换。常在设备能力比较弱，只能上报简单的二进制数据

的设备场景下使用。FunctionGraph的函数托管计算服务支持Node.js、Python、Java、Go、C#、PHP、Cangjie和定制运行时语言，可以满足多种开发需求；具有实时查看运行日志、查看图形化监控等功能，大大的提高了开发者开发、调试效率。

说明

- FunctionGraph是一项基于事件驱动的函数托管计算服务。使用FunctionGraph函数，只需编写业务函数代码并设置运行的条件，无需配置和管理服务器等基础设施，函数以弹性、免运维、高可靠的方式运行。
- FunctionGraph的收费标准可见：**FunctionGraph函数工作流计费概述**。此外，FunctionGraph按函数实际执行资源计费，不执行不产生费用。

须知

下面将讲述数据转换格式及其原理，您可以根据您接入协议选择对应示例代码直接进行IoTDA接入实践（推荐）：

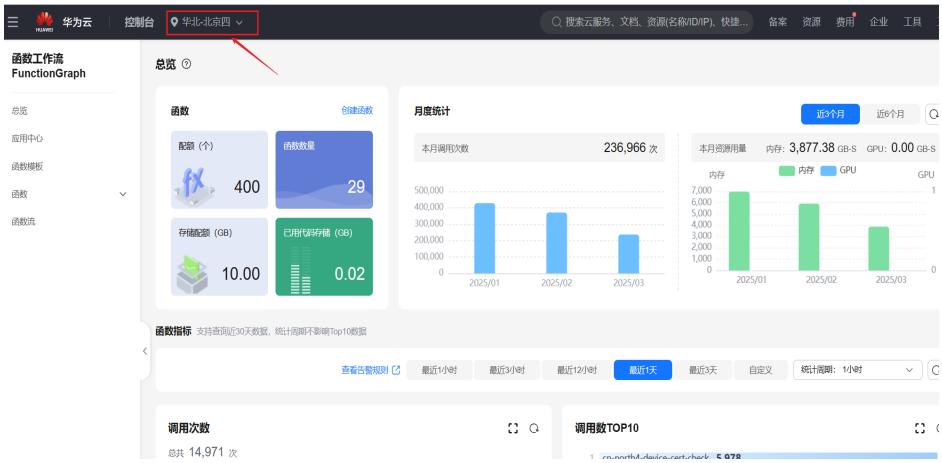
- **MQTT（S）-编解码插件样例**
- **NB-IoT（CoAP）- 编解码插件样例**

使用流程

图 3-77 FunctionGraph 使用流程

1. 创建产品：在IoTDA创建一个CoAP协议或MQTT协议的产品及设备。具体说明可见：**创建产品**。
   a. 访问**设备接入服务**，单击“管理控制台”进入设备接入控制台。选择您的实例，单击实例卡片进入。
   b. 单击左侧导航栏“产品”，单击页面左侧的“创建产品”。根据页面提示填写参数，然后单击“确定”，完成产品的创建。
2. 编写FunctionGraph插件：
   a. 创建函数方法可见：**FunctionGraph创建事件函数**。需要注意的是，创建的事件函数要与在IoTDA中创建的产品局点一致，否则无法被产品引用，局点可在控制台左上方知晓。

图 3-78 FunctionGraph 开发-局点查看



b. 编写编解码插件：FunctionGraph支持多种运行时语言，包括Python 、Node.js、Java、Go、C#、PHP、Cangjie及定制运行时等，不同语言所支持的版本有所差异。具体使用说明可见：**FunctionGraph编程语言说明及开发指导**。

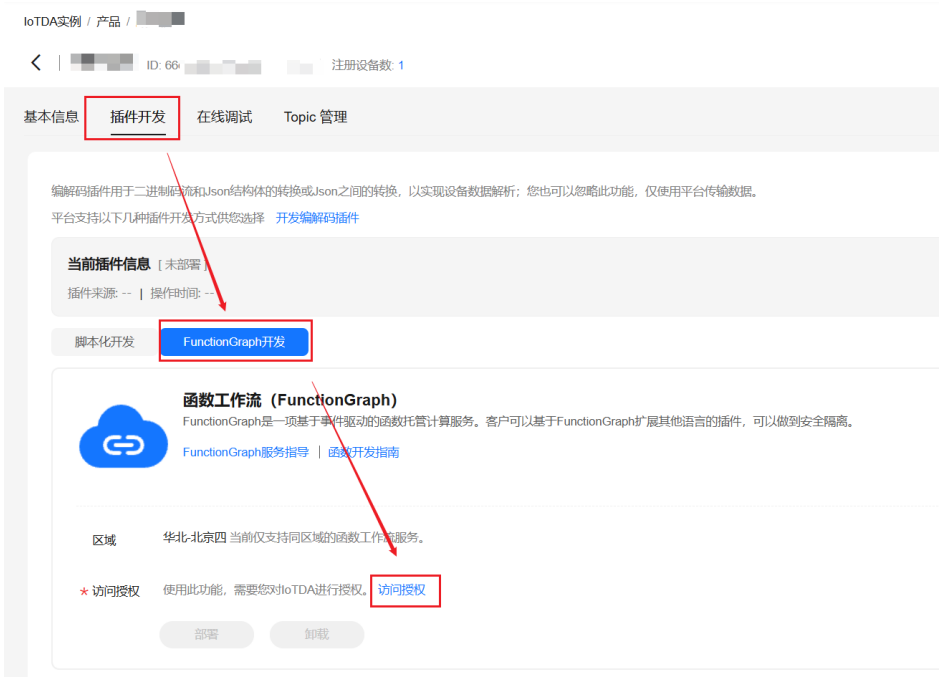
说明

FunctionGraph函数创建-快速入门可见：**使用空白模板创建并执行函数**。

3. 部署FunctionGraph插件：

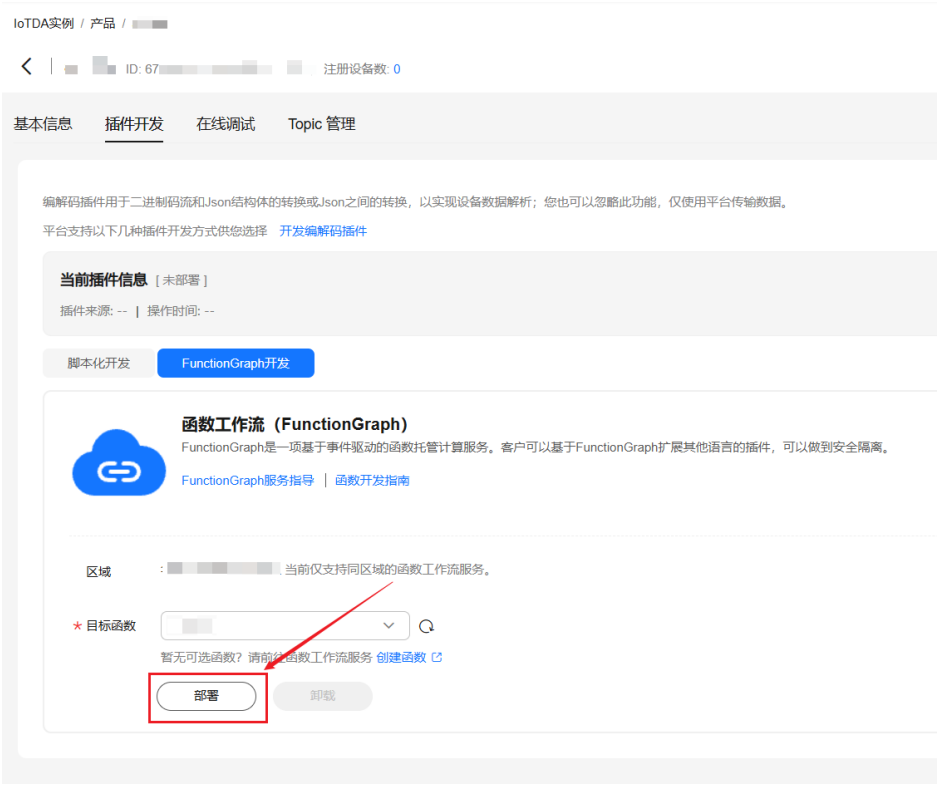
a. 回到**设备接入服务**控制台，打开产品页面，单击页面中的“插件开发 > FunctionGraph开发”。若是第一次使用，需要进行访问授权。

图 3-79 FunctionGraph 开发-插件授权



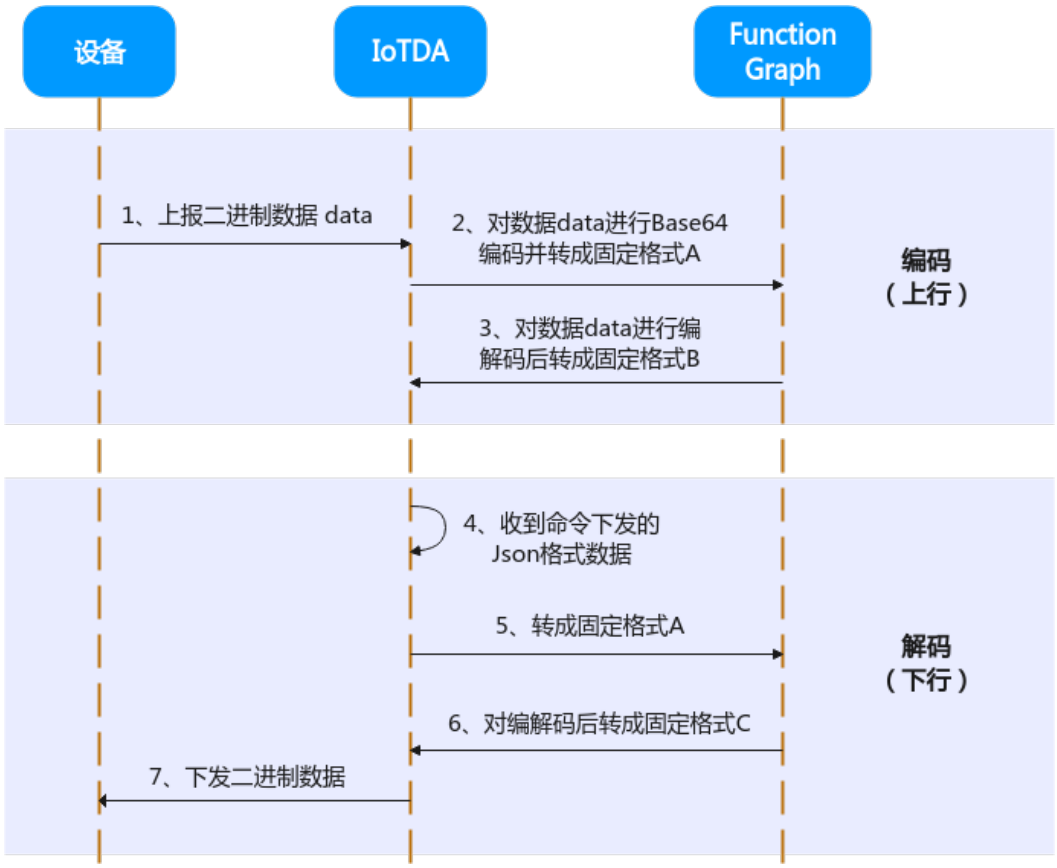
b. 授权成功后，选择您在2创建的目标函数并单击“部署”。

图 3-80 FunctionGraph 开发-部署插件



IoTDA+FunctionGraph 通信说明（API）

图 3-81 IoTDA+FunctionGraph 使用说明



1. 当上报数据为二进制数据时，IoTDA会自动将数据data进行Base64编码，在IoTDA内部以Base64编码后的数据格式存储。例如：设备上报数据[0x01, 0x02]，在IoTDA中存储的数据为“AQI=”。

   说明

   在创建产品时，若为CoAP协议，默认为二进制数据格式，发送到编解码插件会是Base64编码后的数据。若为MQTT（S）协议，将根据您选择的数据格式及编码格式进行数据转换，具体可见：**创建产品说明**。

2. IoTDA接收到数据后，若存在编解码插件，将会把数据以特定格式传输到FunctionGraph。参数及数据格式（固定格式A）为：

表 3-3 上行数据格式

字段名	必选/可选	类型	参数描述
codecType	必选	String	**参数解释：** 编解码类型。decode为上行解码（二进制流转换为JSON格式）；encode为下行编码（JSON格式转换为二进制流）

字段名	必选/可选	类型	参数描述
message	必选	String	**参数解释：** 是JSON格式转换成的字符串数据，其中包含topic和payload两个参数。 ● topic：当为MQTT协议时，会携带上报的Topic；当为CoAP协议时，值为null。 ● payload：设备上报数据的Base64编码后数据（若为MQTT协议，可在产品选择编码格式）。

IoTDA发送给FunctionGraph解码请求示例（CoAP）：

```
{
    "codecType": "decode",
    "message": "{\"topic\": null,\"payload\": \"AABQAFo=\"}"
}
```

IoTDA发送给FunctionGraph解码请求示例（MQTT）：

```
{
    "codecType": "decode",
    "message": "{\"topic\": \"$oc/devices/661f99d6da14e268414f0af6_longsj123/sys/properties/report\",\"payload\": \"AABQAFo=\"}"
}
```

3. 经过FunctionGraph解码后，返回编解码结果。其发送给平台（IoTDA）的数据格式及参数（固定格式B）为：

表 3-4 下行数据格式

字段名	必选/可选	类型	参数描述
status	必选	String	**参数解释：** 标识编解码的执行结果，200表示成功，其他表示设备执行结果为失败。
message	必选	String	**参数解释：** 用于解码数据（二进制流解码为JSON格式），为JSON格式转换成的字符串。

FunctionGraph发送给IoTDA的解码请求示例：

```
{
     "status": 200,
     "message": "{\"msg_type\":\"properties_report\",\"services\":[{\"service_id\":\"smokerdector\",\"properties\":{\"level\":258,\"temperature\":3.4}}]}"
 }
```

4. 当平台或应用侧进行数据下发时，IoTDA会下发JSON格式的数据，需要经过编解码功能把JSON格式数据转换为二进制码流再进行数据下发。
5. IoTDA下发数据前，若存在编解码插件，将会把数据以特定格式传输到FunctionGraph。传输到FunctionGraph的数据格式及参数（固定格式A）为：

表 3-5 上行数据格式

字段名	必选/可选	类型	参数描述
codecType	必选	String	**参数解释：** 编解码类型。decode为上行解码（二进制流转换为JSON格式）；encode为下行编码（JSON格式转换为二进制流）
message	必选	String	**参数解释：** 是JSON格式转换成的字符串数据。

IoTDA发送给FunctionGraph编码请求示例：

```
{
    "codecType": "encode",
    "message": "{\"msg_type\":\"commands\",\"service_id\": \"smokerdector\",\"paras\": {\"value\": 1},\"command_name\": \"SET_ALARM\",\"hasMore\": 0,\"request_id\": 1}"
}
```

6. 经过FunctionGraph解码后，返回编码结果。其发送给平台（IoTDA）的数据格式及参数（固定格式C）为：

表 3-6 下行数据格式

字段名	必选/可选	类型	参数描述
status	必选	String	**参数解释：** 标识编解码的执行结果，200表示成功，其他表示失败。
message	必选	String	**参数解释：** 是JSON格式转换成的字符串数据，其中包含payload参数。 • payload：FunctionGraph解码后的byte[]数据。

FunctionGraph发送给IoTDA的编码码请求示例：

```
{
    "status": 200,
    "message": "{\"payload\":[2,1,0,0,1]}"
}
```

7. 经过编解码插件后，平台将返回给设备二进制流数据。比如说：[2,1,0,0,1]。

IoTDA 物模型数据格式

表 3-7 物模型数据格式说明

编解码	数据类型	msg_type	支持协议	说明
解码（二进制码流转换成JSON格式数据）	设备属性上报	properties_report	ALL	•设备属性上报
	设备返回命令响应	command_response	ALL	•平台命令下发的响应消息
	设备返回平台设置设备属性响应	properties_set_response	MQTT/MQTTS	•平台设置设备属性的响应消息
	设备返回平台查询设备属性响应	properties_get_response	MQTT/MQTTS	•平台查询设备属性的响应消息
	设备消息上报	message_up	MQTT/MQTTS	•设备消息上报
编码（JSON格式转换成二进制码流）	平台命令下发	commands	ALL	平台命令下发
	平台响应设备属性上报	properties_report_reply	NB-IoT（CoAP）	平台响应设备属性上报（NB-IoT设备）
	平台设置设备属性	properties_set	MQTT/MQTTS	•平台设置设备属性
	平台查询设备属性	properties_get	MQTT/MQTTS	•平台查询设备属性
	平台消息下发	messages	MQTT/MQTTS	平台消息下发

数据解码格式定义

数据解析场景，平台收到设备侧的数据时，平台会将设备侧payload中的二进制码流发送到FunctionGraph；FunctionGraph中需要实现二进制码流解码，解码成平台能识别的产品模型中定义的JSON格式，平台对解析后的JSON要求：

```
{
    "status": 200,
    "message": "${解码后的JSON数据格式}"
}
```

其中${解码后的JSON数据格式}，为平台对FunctionGraph解析后要求的JSON格式，其要求的数据格式为：

- 设备属性上报

  ```
  {
     "msg_type": "properties_report",
     "services": [{
        "service_id": "Battery",
        "properties": {
           "batteryLevel": 57
        },
        "event_time": "20151212T121212Z"
  ```

```
    }]
}
```

表 3-8 设备属性上报数据格式

字段名	必选/可选	类型	参数描述
msg_type	必选	String	属性上报的消息类型，固定为：properties_report。
services	必选	List<ServiceProperty>	设备服务数据列表（具体结构参考下表ServiceProperty定义表）。

表 3-9 ServiceProperty 结构定义

字段名	必选/可选	类型	参数描述
service_id	必选	String	设备的服务ID。
properties	必选	Object	设备服务的属性列表，具体字段在设备关联的产品模型中定义。
event_time	可选	String	设备采集数据UTC时间（格式：yyyyMMddTHHmmssZ），如：20161219T114920Z。 设备上报数据不带该参数或参数格式错误时，则数据上报时间以平台时间为准。

- 平台设置设备属性的响应消息

```
{
  "msg_type": "properties_set_response",
  "request_id": "42aa08ea-84c1-4025-a7b2-c1f6efe547c2",
  "result_code": 0,
  "result_desc": "success"
}
```

表 3-10 设备属性的响应消息数据格式

字段名	必选/可选	类型	参数描述
msg_type	必选	String	属性上报的消息类型，固定为： properties_set_response
request_id	可选	String	用于唯一标识这次请求，设备侧收到的消息带该参数时，响应消息需要将该参数值返回给平台。如果解码后的消息不带该字段，则以topic中带的request_id为准。

字段名	必选/可选	类型	参数描述
result_code	可选	Integer	命令的执行结果，0表示成功，其他表示失败。不带默认认为成功。
result_desc	可选	String	属性设置的响应描述。

- 平台查询设备属性的响应消息

```
{
"msg_type": "properties_get_response",
"request_id": "42aa08ea-84c1-4025-a7b2-c1f6efe547c2",
   "services": [
      {
         "service_id": "analog",
         "properties": {
            "PhV_phsA": "1",
            "PhV_phsB": "2"
         },
         "event_time": "20190606T121212Z"
      }
   ]
}
```

表 3-11 平台查询设备属性的响应数据格式

字段名	必选/可选	类型	参数描述
msg_type	必选	String	固定为：properties_get_response
request_id	可选	String	用于唯一标识这次请求，设备侧收到的消息带该参数时，响应消息需要将该参数值返回给平台。如果解码后的消息不带该字段，则以topic中带的request_id为准。
services	必选	List<ServiceProperty>	设备服务数据列表（具体结构参考下表ServiceProperty定义表）。

表 3-12 ServiceProperty 结构定义

字段名	必选/可选	类型	参数描述
service_id	必选	String	设备的服务ID。
properties	必选	Object	设备服务的属性列表，具体字段在设备关联的产品模型中定义。

字段名	必选/可选	类型	参数描述
event_time	可选	String	设备采集数据UTC时间（格式：yyyyMMddTHHmmssZ），如：20161219T114920Z。 设备上报数据不带该参数或参数格式错误时，则数据上报时间以平台时间为准。

- 平台命令下发的响应消息

```
{
  "msg_type": "command_response",
  "request_id": "42aa08ea-84c1-4025-a7b2-c1f6efe547c2",
   "result_code": 0,
  "command_name": "ON_OFF",
  "service_id": "WaterMeter",
  "paras": {
    "value": "1"
  }
}
```

表 3-13 平台命令下发数据格式

字段名	必选/可选	类型	参数描述
msg_type	必选	String	固定为：command_response
request_id	可选	String	用于唯一标识这次请求，设备侧收到的消息带该参数时，响应消息需要将该参数值返回给平台。如果解码后的消息不带该字段，则以topic中带的request_id为准。
result_code	可选	Integer	标识命令的执行结果，0表示成功，其他表示失败。不带默认认为成功。
response_name	可选	String	命令的响应名称，在设备关联的产品模型中定义。
paras	可选	Object	命令的响应参数，具体字段在设备关联的产品模型中定义。

- 设备消息上报

```
{
  "msg_type": "message_up",
  "content": "hello"
}
```

表 3-14 设备消息上报数据格式

字段名	必选/可选	类型	参数描述
msg_type	必选	String	固定为：message_up
content	可选	String	消息内容。

数据编码格式定义

数据解析场景，平台进行数据下发时，会将产品模型定义的JSON格式数据（如果不是该格式的数据则可能会导致编解码失败），发送给FunctionGraph；FunctionGraph需要将JSON格式的数据，编码为设备可以识别的二进制码流；编码时平台传递到FunctionGraph的JSON格式如下：

```
{
    "codecType": "encode",
    "message": "${平台发送给FunctionGraph的JSON数据格式}"
}
```

其中${平台发送给FunctionGraph的JSON数据格式}，为平台发送给FunctionGraph的编码前的JSON数据，其发送的格式为：

- 平台命令下发

```
{
  "msg_type": "commands",
   "request_id": "42aa08ea-84c1-4025-a7b2-c1f6efe547c2",
  "command_name": "ON_OFF",
  "service_id": "WaterMeter",
  "paras": {
    "value": 1
  }
}
```

表 3-15 设备命令下发数据格式

字段名	必选/可选	类型	参数描述
msg_type	必选	String	固定为：commands
request_id	必选	String	用于唯一标识这次请求，该标识会通过topic下发给设备。
service_id	可选	String	设备的服务ID。
command_name	可选	String	设备命令名称，在设备关联的产品模型中定义。
paras	可选	Object	设备命令的执行参数，具体字段在设备关联的产品模型中定义。

- 平台设置设备属性

```
{
"msg_type": "properties_set",
"request_id": "42aa08ea-84c1-4025-a7b2-c1f6efe547c2",
    "services": [{
            "service_id": "Temperature",
            "properties": {
                "value": 57
            }
        },
        {
            "service_id": "Battery",
            "properties": {
                "level": 80
            }
        }
    ]
}
```

表 3-16 平台设置设备属性数据格式

字段名	必选/可选	类型	参数描述
msg_type	必选	String	固定为：properties_set
request_id	必选	String	用于唯一标识这次请求，设备侧收到的消息带该参数时，响应消息需要将该参数值返回给平台。
services	必选	List<ServiceProperty>	设备服务数据列表。

ServiceProperty结构定义：

表 3-17 ServiceProperty 结构定义

字段名	必选/可选	类型	参数描述
service_id	必选	String	设备的服务ID。
properties	必选	Object	设备服务的属性列表，具体字段在产品模型里定义。

- 平台查询设备属性

```
{
  "msg_type": "properties_get",
  "request_id": "42aa08ea-84c1-4025-a7b2-c1f6efe547c2",
  "service_id": "Temperature"
}
```

表 3-18 平台查询设备属性数据格式

字段名	必选/可选	类型	参数描述
msg_type	必选	String	固定为：properties_get
request_id	必选	String	用于唯一标识这次请求，该标识会通过topic下发给设备。
service_id	可选	String	设备的服务ID。

- 平台响应设备属性上报（NB-IoT设备）

```
{
  "msg_type": "properties_report_reply",
  "request": "213355656",
  "result_code": 0
}
```

表 3-19 平台响应设备属性上报数据格式

字段名	必选/可选	类型	参数描述
msg_type	必选	String	固定为：properties_report_reply
request	可选	String	属性上报的BASE64编码字符串。
result_code	可选	Integer	属性上报的执行结果。
has_more	可选	Boolean	是否存在缓存命令。

- 平台消息下发

```
{
  "msg_type": "messages",
  "content": "hello"
}
```

表 3-20 设备消息下发数据格式

字段名	必选/可选	类型	参数描述
msg_type	必选	String	固定为：messages
content	可选	String	消息下发的内容。

3.2.4.4.2 MQTT（S）-编解码插件样例

本文以烟感设备为例，介绍如何使用JavaScript语言开发一个支持MQTT/MQTTS协议的设备进行属性上报和命令下发编解码的FunctionGraph函数插件。并介绍从二进制数据到JSON数据格式的转换及其调试方法。

烟感设备样例

场景说明

有一款烟感设备，具有如下特征：

- 具有烟雾报警功能（火灾等级）和温度上报功能。
- 支持远程控制命令，可远程打开报警功能。比如火灾现场温度，远程打开烟雾报警，提醒住户疏散。
- 该款烟感设备，设备能力比较弱，无法按照设备接口定义的JSON格式上报数据，只能上报简单的二进制数据。

产品模型定义

在烟感产品的开发空间，完成产品模型定义。

- level：火灾级别，用于表示火灾的严重程度。
- temperature：温度，用于表示火灾现场温度。
- SET_ALARM：打开或关闭告警命令，value=0表示关闭，value=1表示打开。响应命令result用于上报修改后告警值。

图 3-82 模型定义-smokedetector

编解码插件开发

步骤1 在烟感产品的详情页面，选择“插件开发”，单击“FunctionGraph开发”，单击“创建函数”，若是第一次使用，需要进行访问授权。

图 3-83 FunctionGraph 开发-插件授权

步骤2 进入FunctionGraph控制台后，单击“创建函数”。在弹出的界面中，选择“创建空白函数”，自定义填入函数名称，运行时选择“Node.js 16.17”。

图 3-84 函数列表-创建函数

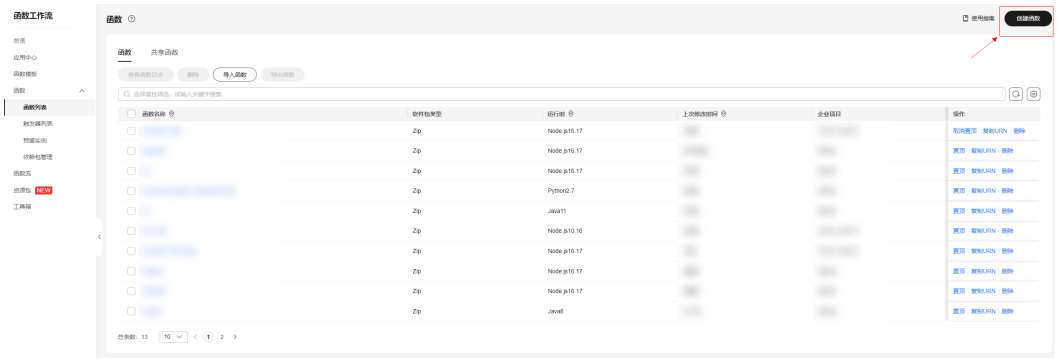
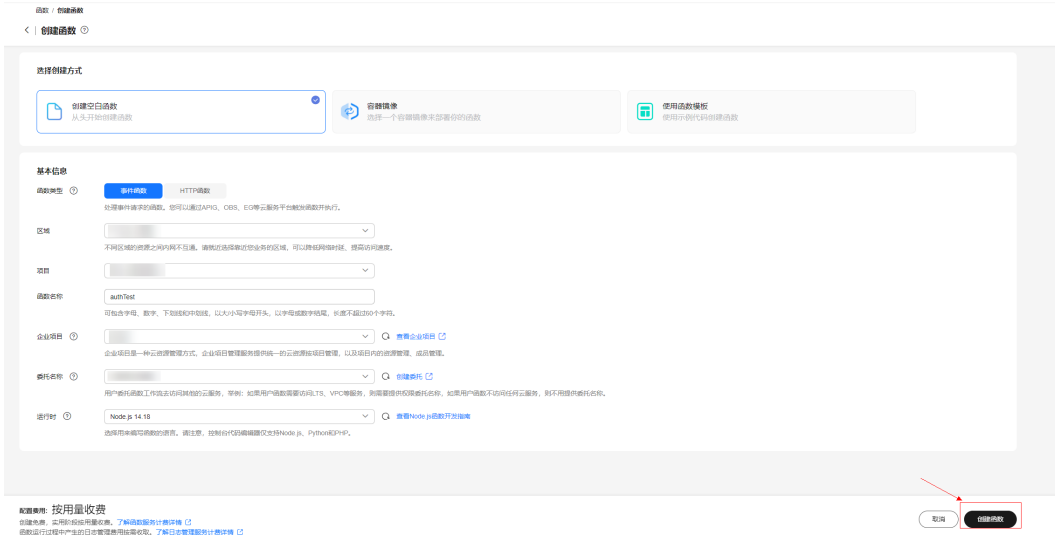


图 3-85 创建函数-参数信息



步骤3 编写脚本，实现二进制数据到JSON数据的转换。脚本需要实现如下两个方法：

- decode：将设备上报的二进制数据转换为平台产品模型中定义的JSON格式。具体的JSON格式要求见：**数据解码格式定义** 。
- encode：平台有下行数据发送给设备时，将平台的JSON格式数据转换为设备支持的二进制格式。平台的JSON格式见：**数据编码格式定义**。

针对当前烟感产品实现的JavaScript样例如下，将代码复制到Project工程中，并单击“部署代码”。

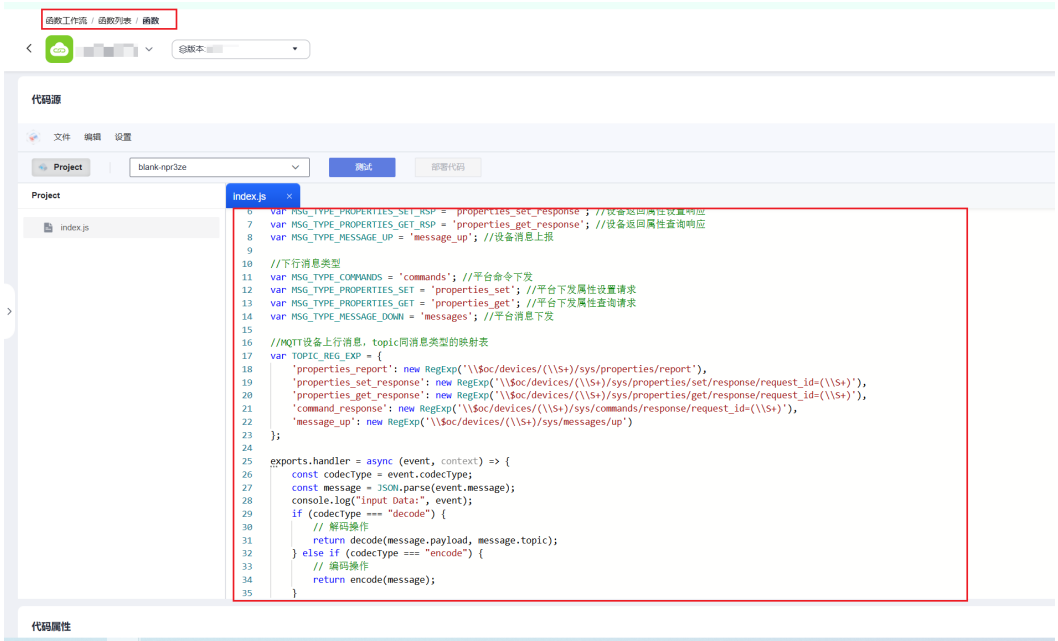
```
//上行消息类型
var MSG_TYPE_PROPERTIES_REPORT = 'properties_report'; //设备属性上报
var MSG_TYPE_COMMAND_RSP = 'command_response'; //设备返回命令响应
var MSG_TYPE_PROPERTIES_SET_RSP = 'properties_set_response'; //设备返回属性设置响应
var MSG_TYPE_PROPERTIES_GET_RSP = 'properties_get_response'; //设备返回属性查询响应
var MSG_TYPE_MESSAGE_UP = 'message_up'; //设备消息上报
//下行消息类型
var MSG_TYPE_COMMANDS = 'commands'; //平台命令下发
var MSG_TYPE_PROPERTIES_SET = 'properties_set'; //平台下发属性设置请求
var MSG_TYPE_PROPERTIES_GET = 'properties_get'; //平台下发属性查询请求
var MSG_TYPE_MESSAGE_DOWN = 'messages'; //平台消息下发

//MQTT设备上行消息，topic同消息类型的映射表
var TOPIC_REG_EXP = {
    'properties_report': new RegExp('\\$oc/devices/(\\S+)/sys/properties/report'),
    'properties_set_response': new RegExp('\\$oc/devices/(\\S+)/sys/properties/set/response/request_id=(\\S
+)'),
```

```
    'properties_get_response': new RegExp('\\$oc/devices/(\\S+)/sys/properties/get/response/request_id=(\\S
+)'),
    'command_response': new RegExp('\\$oc/devices/(\\S+)/sys/commands/response/request_id=(\\S+)'),
    'message_up': new RegExp('\\$oc/devices/(\\S+)/sys/messages/up')
};
exports.handler = async (event, context) => {
    const codecType = event.codecType;
    const message = JSON.parse(event.message);
    console.log("input Data:", event);
    if (codecType === "decode") {
        // 解码操作
        return decode(message.payload, message.topic);
    } else if (codecType === "encode") {
        // 编码操作
        return encode(message);
    }
}
/*
示例：烟感设备上报属性和回复命令响应时，携带的是二进制码流，通过javascript脚本将二进制码流数据解码为
符合产品模型定义的json格式数据
传入参数：
  // 前两个字节0x00, 0x50为属性level的值，后两个字节0x00, 0x5a为属性temperature的值。
  payload:[0x00, 0x50, 0x00, 0x5a]
  topic:$oc/devices/cf40f3c4-7152-41c6-a201-a2333122054a/sys/properties/report
输出结果:
  {"msg_type":"properties_report","services":[{"service_id":"smokerdector","properties":
{"level":80,"temperature":90}}]}
传入参数:
  // 第一个字节0x02为"command_name"是"SET_ALARM"， 第二个字节0x00为命令响应成功, 后两个字节0x00,
0x01为命令响应value的值。
  payload: [0x02, 0x00, 0x00, 0x01]
  topic: $oc/devices/cf40f3c4-7152-41c6-a201-a2333122054a/sys/commands/response/
request_id=bf40f0c4-4022-41c6-a201-c5133122054a
输出结果:

{"msg_type":"command_response","result_code":0,"command_name":"SET_ALARM","service_id":"smokerdect
or","paras":{"value":"1"}}
*/
// 解码函数
function decode(payload, topic) {
    // 在这里实现具体的解码逻辑
    var binaryString = atob(payload);
    const byteArray = new Uint8Array(binaryString.length);
    for (let i = 0; i < binaryString.length; i++) {
        byteArray[i] = binaryString.charCodeAt(i);
    }
    /*  byteArray 为解码后，设备上报的二进制数据，客户可以在此处查看上报的数据是否解析正确 */
    var returnData;
    msgType = topicParse(topic);
        if (msgType == MSG_TYPE_PROPERTIES_REPORT) {
        returnData = decodePropertiesReport(byteArray);
    } else if (msgType == MSG_TYPE_COMMAND_RSP) {
        returnData = decodeCommandRsp(byteArray);
    } else if (msgType == MSG_TYPE_PROPERTIES_SET_RSP) {
        // 转换为属性设置响应的JSON格式
        // jsonObj = {"msg_type":"properties_set_response","result_code":0,"result_desc":"success"};
        // returnData = outputData(status, jsonObj)
    } else if (msgType == MSG_TYPE_PROPERTIES_GET_RSP) {
        // 转换为属性查询响应的JSON格式
        // jsonObj = {"msg_type":"properties_get_response","services":[{"service_id":"analog","properties":
{"PhV_phsA":"1","PhV_phsB":"2"}}]};
        // returnData = outputData(status, jsonObj)
    } else if (msgType == MSG_TYPE_MESSAGE_UP) {
        // 转换为消息上报的JSON格式
        // jsonObj = {"msg_type":"message_up","content":"hello"};
        // returnData = outputData(status, jsonObj)
    }
    return returnData;
}
```

```
// 编码函数
/*
示例数据：命令下发时，通过javascript的encode方法将平台JSON格式的数据，编码为二进制码流
传入参数 ->
    {"msg_type":"commands","command_name":"SET_ALARM","service_id":"smokerdector","paras":
{"value":1}}
输出结果 ->
    // 第一个字节0x01用于标识命令下发，第二个字节0x00为command_name == 'SET_ALARM', 后两个字节
0x00, 0x01为设置的命令属性值的值。
    [0x01, 0x00, 0x00, 0x01]
*/
function encode(data) {
    var msgType = data.msg_type;
    let payload = [];
    var status = 200;
    // 命令下发
    if (msgType == MSG_TYPE_COMMANDS) {
        payload[0] = 0x02; // 标识命令下发
        if (data.command_name == 'SET_ALARM') {
            payload[1] = 0x00; // 标识命令名称
        }
        // 设置命令属性值
        payload[2] = (data.paras.value >> 8) & 0xFF;
        payload[3] = data.paras.value & 0xFF;
    } else if (msgType == MSG_TYPE_PROPERTIES_SET) {
        // 设备属性上报的响应消息
        // 属性设置格式样例：{"msg_type":"properties_set","services":[{"service_id":"Temperature","properties":
{"value":57}}]}
        // 开发者有对应属性设置场景时需要根据该JSON格式转换为对应的二进制码流
    } else if (msgType == MSG_TYPE_PROPERTIES_GET) {
        // 属性查询格式样例：{"msg_type":"properties_get","service_id":"Temperature"}
        // 开发者有对应属性查询场景时需要根据该JSON格式转换为对应的二进制码流
    } else if (msgType == MSG_TYPE_MESSAGE_DOWN) {
        // 消息下发格式样例：{"msg_type":"messages","content":"hello"}
        // 开发者对应消息下发场景时需要根据该JSON格式转换为对应的二进制码流
    }
    return outputData(status, { "payload": payload });
}
// 根据topic名称解析出消息类型
function topicParse(topic) {
    for (var type in TOPIC_REG_EXP) {
        var pattern = TOPIC_REG_EXP[type];
        if (pattern.test(topic)) {
            return type;
        }
    }
    return '';
}
// 属性上报 -- 上行
function decodePropertiesReport(byteArray) {
    // 设置serviceId参数值，该参数值对应产品模型中的服务类型smokerdector
    var serviceId = 'smokerdector';
    var level = byteArray[0] * Math.pow(2, 8) + byteArray[1];
    var status = 200;
    var jsonObj;
    if (byteArray.length < 4) {
        jsonObj = {
            "msg_type": "ERR", "message": "decodePropertiesReport byte length < 5."
        };
        status = 402;
    }
    // 获取第4位和第5位的值
    const integerPart = byteArray[2]; // 第3位
    const decimalPart = byteArray[3]; // 第4位
    // 组合成小数
    const temperature = parseFloat(integerPart + '.' + decimalPart);
    jsonObj = {
        "msg_type": MSG_TYPE_PROPERTIES_REPORT, "services":
            [{ "service_id": serviceId, "properties": { "level": level, "temperature": temperature } }]
```

```
    };
    return outputData(status, jsonObj);
}
// 命令响应 -- 上行
function decodeCommandRsp(byteArray) {
    var serviceId = 'smokerdector';
    var command = byteArray[0];
    var command_name = '';
    if (2 == command) {
        command_name = 'SET_ALARM';
    }
    var result_code = byteArray[1]; // 从二进制码流中获取命令执行结果
    var value = byteArray[2] * Math.pow(2, 8) + byteArray[3]; // 从二进制码流中获取命令执行结果返回值
    //转换为命令响应的JSON格式
    jsonObj = {
        'msg_type': MSG_TYPE_COMMAND_RSP, 'service_id': serviceId, "command_name": command_name,
        'result_code': result_code, 'paras': { 'value': value }
    };
    return outputData(200, jsonObj);
}
// 输出结果
function outputData(status, body) {
    const output =
    {
        'status': status,
        'message': JSON.stringify(body),
    }
    console.log("output Data:", output);
    return output;
}
```

图 3-86 FunctionGraph-复制代码到 Project 工程



步骤4 在线调试脚本。脚本编辑完成后，在FunctionGraph函数中单击“配置测试事件”，选择空白模板，输入模拟数据，单击“创建”。配置好测试事件后，单击“测试”，即可得到函数返回结果及其日志。

假设输入以下模拟数据：其中payload为设备侧上报的二进制：0x01, 0x02, 0x03, 0x04。"AQIDBA=="为被平台Base64编码后的值。

```
{
    "codecType": "decode",
```

```
    "message": "{\"topic\": \"$oc/devices/device_id/sys/properties/report\",\"payload\": \"AQIDBA==\"}"
}
```

图 3-87 FunctionGraph-添加测试事件

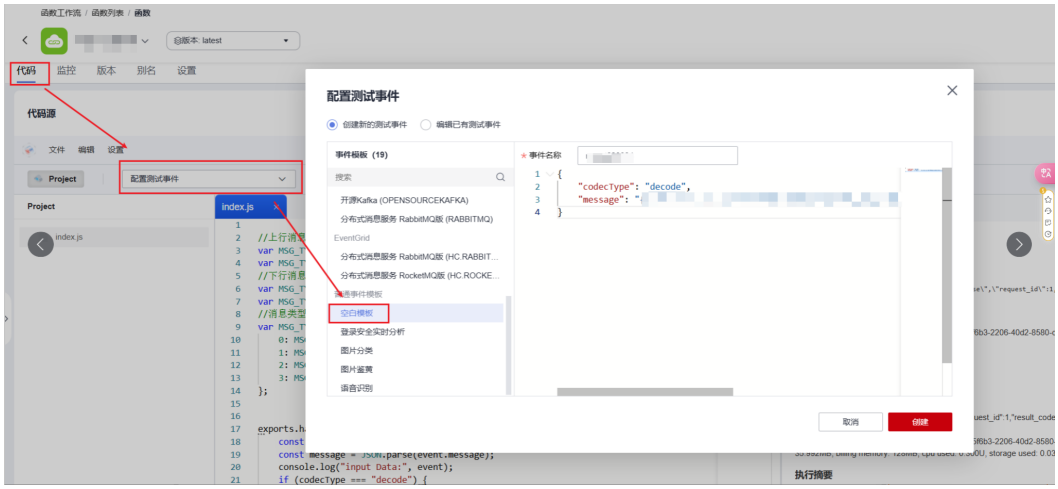
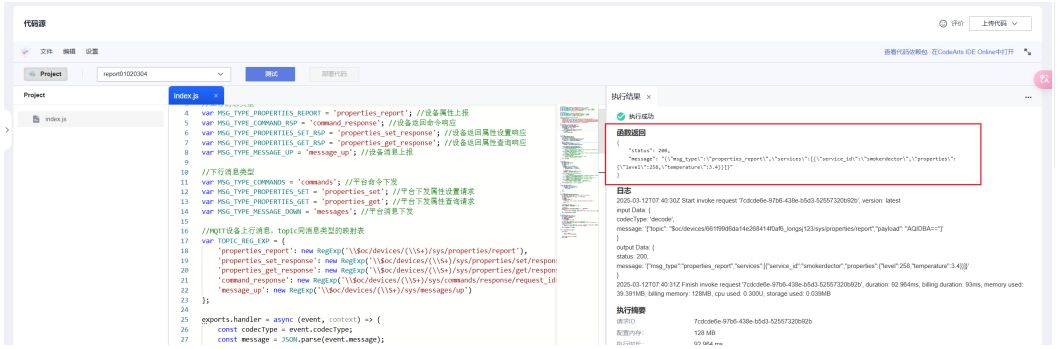
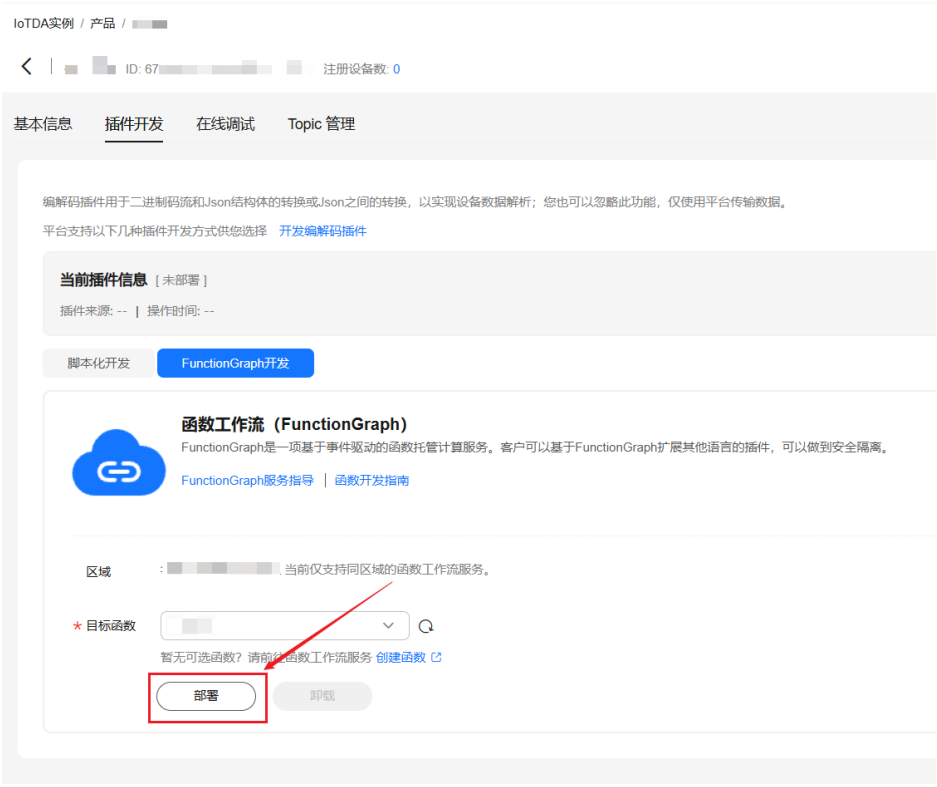


图 3-88 FunctionGraph-测试结果 MQTT



步骤5 调试成功后，在**步骤1**的“目标函数”中选择您创建的FunctionGraph函数，并单击“部署”。

图 3-89 FunctionGraph 开发-部署插件



----结束

3.2.4.4.3 NB-IoT（CoAP）- 编解码插件样例

本文以烟感设备为例，介绍如何使用JavaScript语言开发一个支持CoAP协议的设备进行属性上报和命令下发编解码的FunctionGraph函数插件。并介绍从二进制数据到JSON数据格式的转换及其调试方法。

烟感设备样例

场景说明

有一款烟感设备，具有如下特征：

- 具有烟雾报警功能（火灾等级）和温度上报功能。
- 支持远程控制命令，可远程打开报警功能。比如火灾现场温度，远程打开烟雾报警，提醒住户疏散。
- 该款烟感设备，设备能力比较弱，无法按照设备接口定义的JSON格式上报数据，只能上报简单的二进制数据。

产品模型定义

在烟感产品的开发空间，完成产品模型定义。

- level：火灾级别，用于表示火灾的严重程度。
- temperature：温度，用于表示火灾现场温度。
- SET_ALARM：打开或关闭告警命令，value=0表示关闭，value=1表示打开。响应命令result用于上报修改后告警值。

图 3-90 模型定义-smokedetector

编解码插件开发

步骤1 在烟感产品的详情页面，选择“插件开发”，单击“FunctionGraph开发”，单击“创建函数”，若是第一次使用，需要进行访问授权。

图 3-91 FunctionGraph-创建 FunctionGraph 函数

步骤2 进入FunctionGraph控制台后，单击“创建函数”。在弹出的界面中，选择“创建空白函数”，自定义填入函数名称，运行时选择“Node.js 16.17”。

图 3-92 函数列表-创建函数

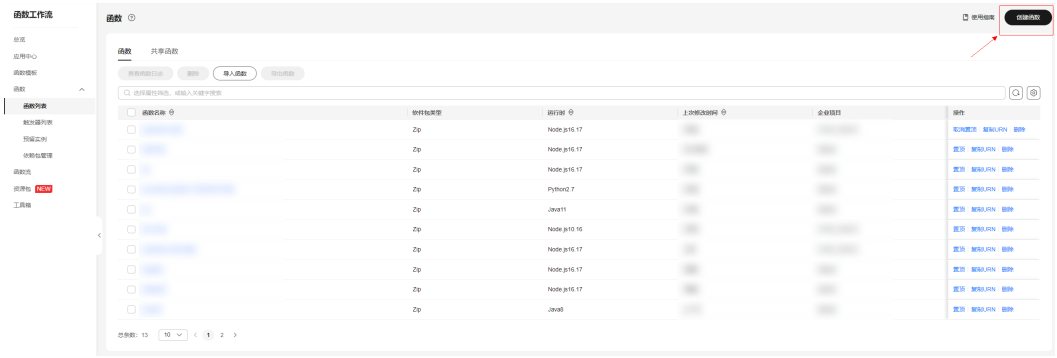
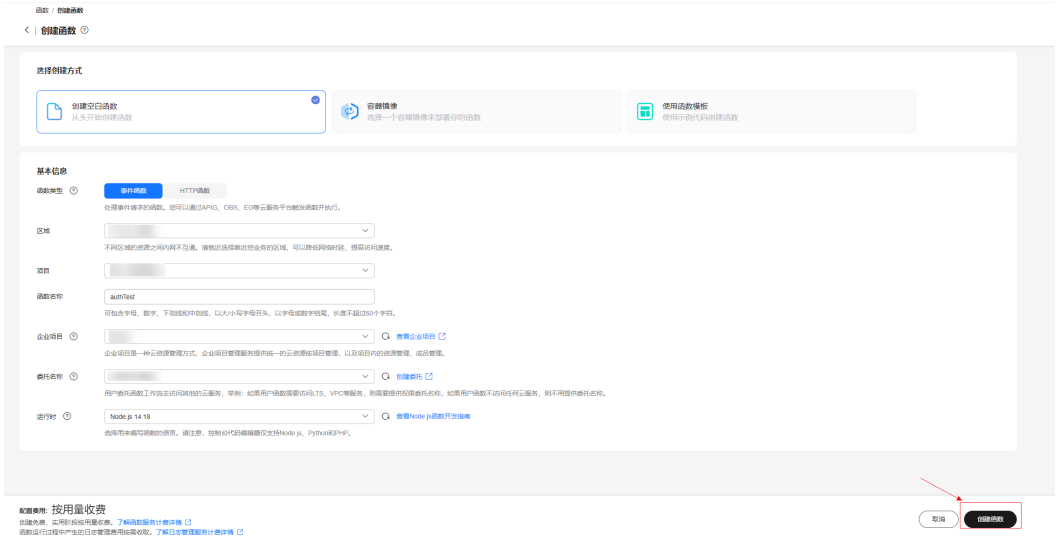


图 3-93 创建函数-参数信息



步骤3 编写脚本，实现二进制数据到JSON数据的转换。脚本需要实现如下两个方法：

- decode：将设备上报的二进制数据转换为平台产品模型中定义的JSON格式。具体的JSON格式要求见：**数据解码格式定义** 。
- encode：平台有下行数据发送给设备时，将平台的JSON格式数据转换为设备支持的二进制格式。平台的JSON格式见：**数据编码格式定义**。

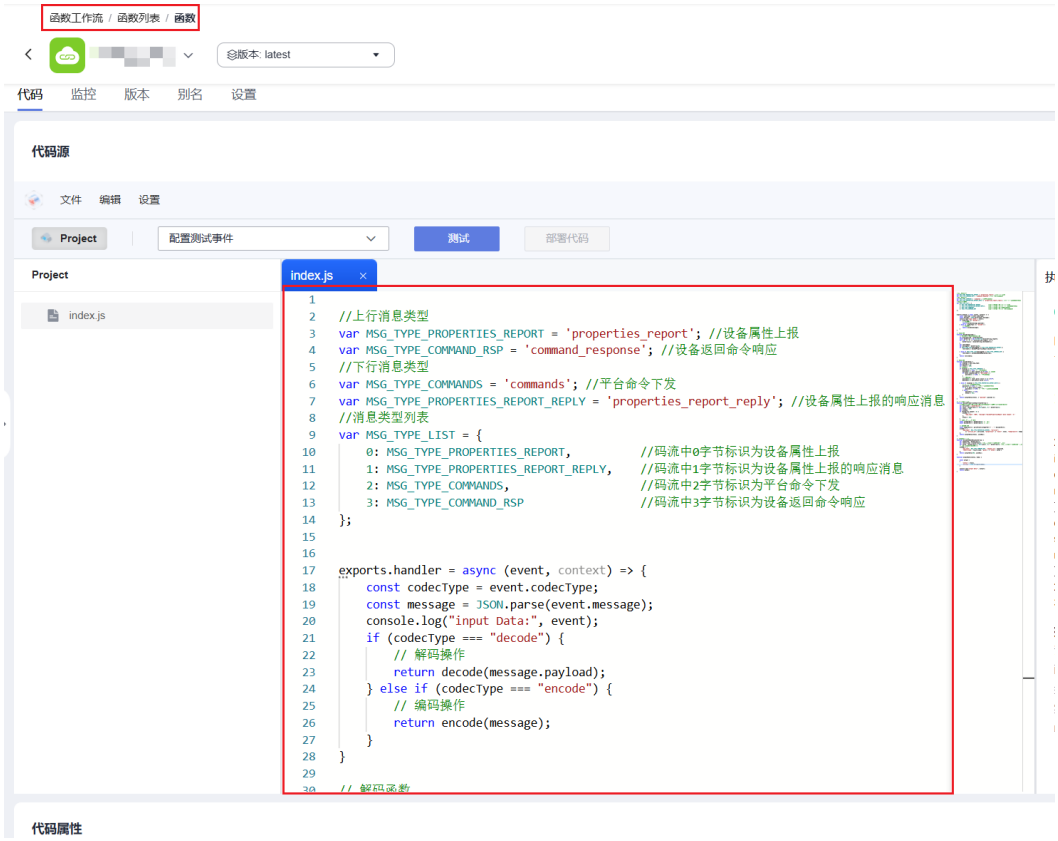
针对当前烟感产品实现的JavaScript样例如下，将代码复制到Project工程中。

```
//上行消息类型
var MSG_TYPE_PROPERTIES_REPORT = 'properties_report'; //设备属性上报
var MSG_TYPE_COMMAND_RSP = 'command_response'; //设备返回命令响应
//下行消息类型
var MSG_TYPE_COMMANDS = 'commands'; //平台命令下发
var MSG_TYPE_PROPERTIES_REPORT_REPLY = 'properties_report_reply'; //设备属性上报的响应消息
//消息类型列表
var MSG_TYPE_LIST = {
    0: MSG_TYPE_PROPERTIES_REPORT,          //码流中0字节标识为设备属性上报
    1: MSG_TYPE_PROPERTIES_REPORT_REPLY,    //码流中1字节标识为设备属性上报的响应消息
    2: MSG_TYPE_COMMANDS,                   //码流中2字节标识为平台命令下发
    3: MSG_TYPE_COMMAND_RSP                 //码流中3字节标识为设备返回命令响应
};
// FunctionGraph入口函数
exports.handler = async (event, context) => {
    const codecType = event.codecType;
    const message = JSON.parse(event.message);
    console.log("input Data:", event);
```

```
    if (codecType === "decode") {
        // 解码操作
        return decode(message.payload);
    } else if (codecType === "encode") {
        // 编码操作
        return encode(message);
    }
}
/*
示例：烟感设备上报属性和回复命令响应时，携带的是二进制码流，通过javascript脚本将二进制码流数据解码为符合产品模型定义的json格式数据
传入参数：
  payload:[0x00, 0x00, 0x50, 0x00, 0x5a]
  其中payload[0]代表数据类型，0x00代表属性上报；payload[1]与payload[2]代表属性level的值；payload[3]与payload[4]代表属性temperature的值,payload[3]为小数点前的值，payload[4]为小数点后的值。
输出结果:
  {"msg_type":"properties_report","services":[{"service_id":"smokerdector","properties":{"level":80,"temperature":90}}]}
传入参数:
  payload: [0x03, 0x01, 0x00, 0x00, 0x01]
  其中payload[0]代表数据类型，0x03代表设备返回命令响应；payload[1]表示request_id的值，用于标识是否为某一条信息；payload[2]与代表命令设置是否成功，为0则成功；payload[3]与payload[4]代表命令响应"value"的值。
输出结果:
  {"msg_type":"command_response","request_id":1,"result_code":0,"paras":{"value":1}}
*/
function decode(payload) {
    // 这里可以实现具体的解码逻辑
    var binaryString = atob(payload);
    const byteArray = new Uint8Array(binaryString.length);
    for (let i = 0; i < binaryString.length; i++) {
        byteArray[i] = binaryString.charCodeAt(i);
    }
    /* byteArray 为解码后，设备上报的二进制数据，客户可以在此处查看上报的数据是否解析正确 */
    var returnData;
    var messageId = byteArray[0];
    if (MSG_TYPE_LIST[messageId] == MSG_TYPE_PROPERTIES_REPORT) {
        returnData = decodePropertiesReport(byteArray);
    } else if (MSG_TYPE_LIST[messageId] == MSG_TYPE_COMMAND_RSP) {
        returnData = decodeCommandRsp(byteArray);
    }
    return returnData;
}
/*
示例数据：
命令下发时，通过javascript的encode方法将平台JSON格式的数据，编码为二进制码流
传入参数 ->

{"msg_type":"commands","request_id":1,"command_name":"SET_ALARM","service_id":"smokerdector","paras":{"value":1}}
输出结果 ->
    [0x02, 0x00, 0x00, 0x00, 0x01]
    其中payload[0]代表数据类型，0x02代表平台命令下发；payload[1]代表命令ID；payload[2]用于标识command_name，当command_name是SET_ALARM时，payload[2] = 0x00；payload[3]与payload[4]代表命令下发value的值。
示例数据：设备上报属性时返回响应消息，通过javascript的encode方法将平台JSON格式的数据，编码为二进制码流
传入参数 ->
    {"msg_type":"properties_report_reply","request":"000050005a","result_code":0}
输出结果 ->
    [0x01, 0x00]
    其中payload[0]代表数据类型，0x01为设备属性上报的响应消息；payload[1]代表设备响应结果，为0x00时成功。
*/
function encode(data) {
    var msgType = data.msg_type;
    let payload = [];
    var status = 200;
    // 命令下发
```

```
    if (msgType == MSG_TYPE_COMMANDS) {
        payload[0] = 0x02; // 标识命令下发
        payload[1] = data.request_id & 0xFF; // 命令ID
        if (data.command_name == 'SET_ALARM') {
            payload[2] = 0x00; // 标识命令名称
        }
        // 设置命令属性值
        payload[3] = (data.paras.value >> 8) & 0xFF;
        payload[4] = data.paras.value & 0xFF;
    } else if (msgType == MSG_TYPE_PROPERTIES_REPORT_REPLY) {
        // 设备属性上报的响应消息
        payload[0] = 0x01; // 标识属性上报的响应消息
        if (0 == data.result_code) {
            payload[1] = 0x00; // 标识属性上报消息处理成功
        } else {
            payload[1] = 0x01;
            status = 401;
        }
    }
    return outputData(status, { "payload": payload });
}
// 属性上报 -- 上行
function decodePropertiesReport(byteArray) {
    //设置serviceId参数值，该参数值对应产品模型中的服务类型smokerdector
    var serviceId = 'smokerdector';
    var level = byteArray[1] * Math.pow(2, 8) + byteArray[2];
    var status = 200;
    var jsonObj;
    if (byteArray.length < 4) {
        jsonObj = {
            "msg_type": "ERR", "message":"decodePropertiesReport byte length < 5."
        };
        status = 402;
    }
    // 获取第4位和第5位的值
    const integerPart = byteArray[3]; // 第4位
    const decimalPart = byteArray[4]; // 第5位
    // 组合成小数
    const temperature = parseFloat(integerPart + '.' + decimalPart);
    jsonObj = {
        "msg_type": MSG_TYPE_PROPERTIES_REPORT, "services":
            [{ "service_id": serviceId, "properties": { "level": level, "temperature": temperature } }]
    };
    return outputData(status, jsonObj);
}
// 命令响应 -- 上行
function decodeCommandRsp(byteArray) {
    var requestId = byteArray[1];
    var result_code = byteArray[2]; //从二进制码流中获取命令执行结果
    var value = byteArray[3] * Math.pow(2, 8) + byteArray[4]; //从二进制码流中获取命令执行结果返回值
    //转换为命令响应的JSON格式
    jsonObj = {
        'msg_type': MSG_TYPE_COMMAND_RSP, 'request_id': requestId,
        'result_code': result_code, 'paras': { 'value': value }
    };
    return outputData(200, jsonObj);
}
function outputData(status, body) {

    const output =
    {
        'status': status,
        'message': JSON.stringify(body),
    }
    console.log("output Data:", output);
    return output;
}
```

图 3-94 FunctionGraph-复制代码到 Project 工程 CoAP



步骤4 在线调试脚本。脚本编辑完成后，在FunctionGraph函数中单击“配置测试事件”，选择空白模板，输入模拟数据，单击“创建”。配置好测试事件后，单击“测试”，即可得到函数返回结果及其日志。

假设输入以下模拟数据：其中payload为设备侧上报的二进制：0x00, 0x00, 0x05, 0x00,0x5a。"AABQAFo="为被平台Base64编码后的值。

```
{
   "codecType": "decode",
   "message": "{\"topic\": null,\"payload\": \"AABQAFo=\"}"
}
```

图 3-95 FunctionGraph-添加测试事件

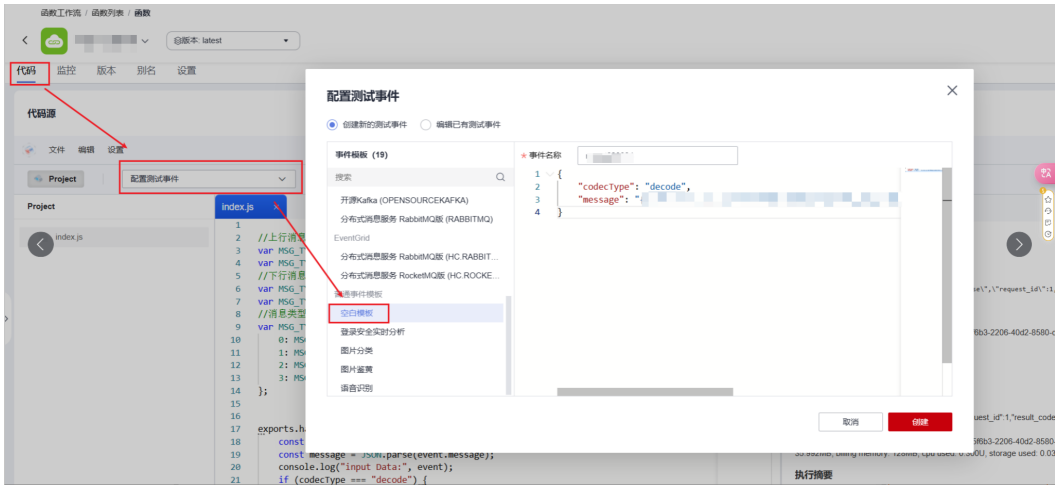
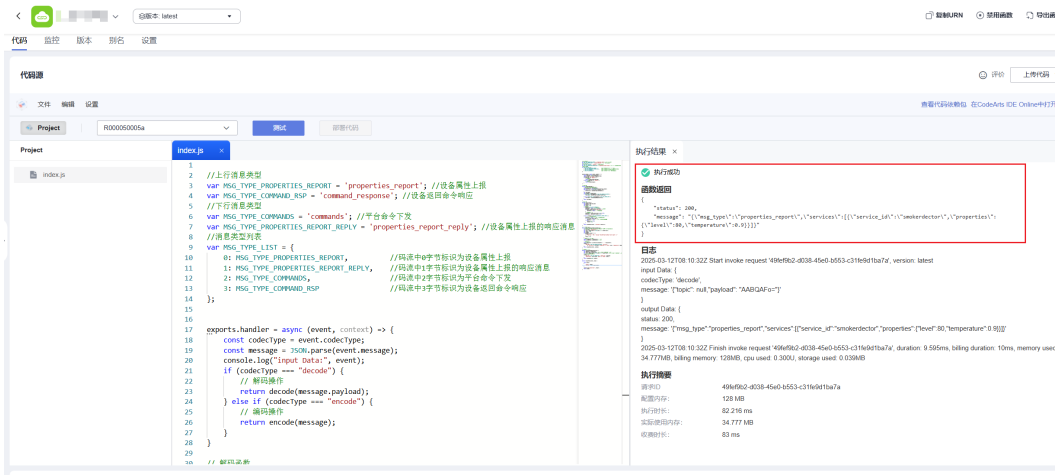


图 3-96 FunctionGraph-测试结果 CoAP



步骤5 调试成功后，在**步骤1**的“目标函数”中选择您创建的FunctionGraph函数，并单击“部署”。

图 3-97 FunctionGraph 开发-部署插件

----结束

3.2.5 在线调试

概述

当产品模型和编解码插件开发完成后，应用服务器就可以通过物联网平台接收设备上报的数据以及向设备下发命令。

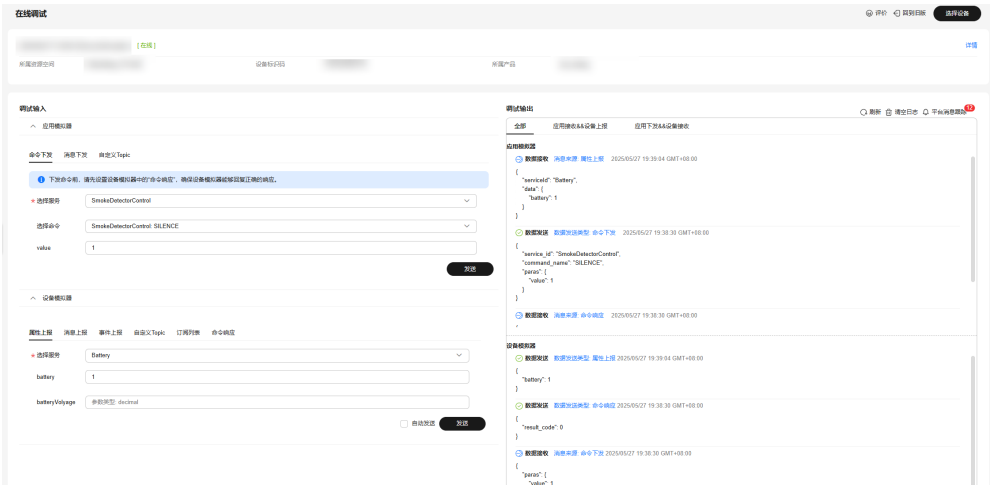
设备接入控制台提供了在线调试的功能，您可以根据自己的业务场景，在开发真实应用和真实设备之前，使用应用模拟器和设备模拟器对数据上报和命令下发等场景进行调试；也可以在真实设备开发完成后使用应用模拟器验证业务流。

使用虚拟设备调测产品

当设备侧开发和应用侧开发均未完成时，开发者可以创建虚拟设备，使用应用模拟器和设备模拟器对产品模型、插件等进行调测。在线调试界面分为以下几个部分

1. 上方设备信息区域：展示当前正在调试的设备的基本信息，包括设备名称，设备状态，设备标识码，所属资源空间和产品等。
2. 左侧上方应用模拟器区域：可模拟应用实现命令下发，消息下发和自定义Topic消息下发等功能。
3. 左侧下方设备模拟器区域：可模拟设备实现属性上报，消息上报，事件上报，自定义Topic消息上报和设置命令响应等功能。
4. 右侧上方应用模拟器展示区域：呈现应用服务器接收到和下发数据。
5. 右侧下方设备模拟器展示区域：呈现设备上报和接收到的数据。

图 3-98 在线调试-虚拟设备结构



接下来，您可以按照以下步骤进行虚拟设备的在线调试：

步骤1 在产品详情中，选择“在线调测”，并单击“新增测试设备”。

步骤2 在弹出的“新增测试设备”窗口，选择“虚拟设备”，单击“确定”，创建一个虚拟设备。虚拟设备名称包含 “DeviceSimulator”字样，每款产品下只能创建一个虚拟设备。

步骤3 在设备列表中，选择新创建的虚拟设备，

图 3-99 在线调试-创建虚拟设备

步骤4 单击右侧的“调试”，进入调试界面。

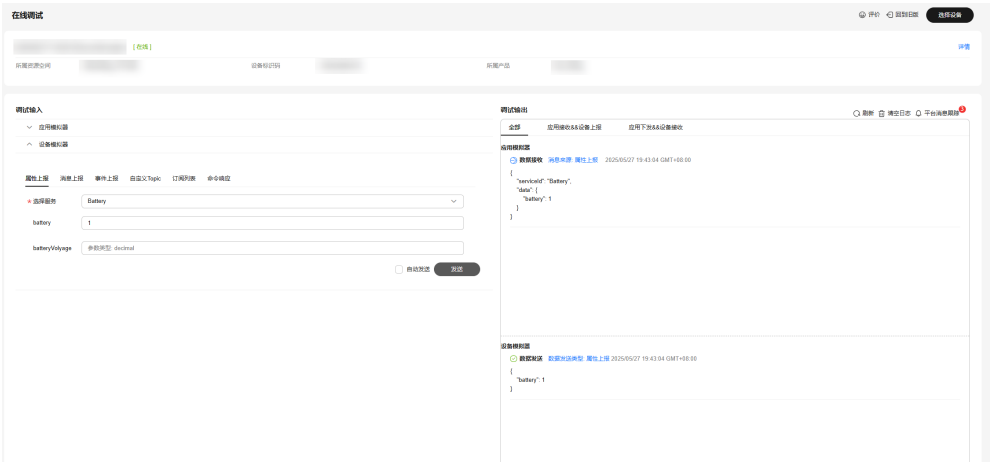
图 3-100 在线调试-进入调试

步骤5 进入在线调试页面，查看设备状态显示为”在线”。

图 3-101 在线调试-设备在线备

步骤6 在“设备模拟器”区域，针对您实际的使用场景，可以选择属性上报，消息上报，事件上报以及自定义Topic上报功能模拟设备侧进行数据的发送。以属性上报为例，切换到设备模拟器”属性上报”页签，选择对应的服务并填写需要上报属性值后，单击“发送”。在右侧设备模拟器展示区域可查看上报的属性并在应用模拟器展示区域查看应用模拟器接收到的属性值。

图 3-102 在线调试-模拟数据上报 Battery



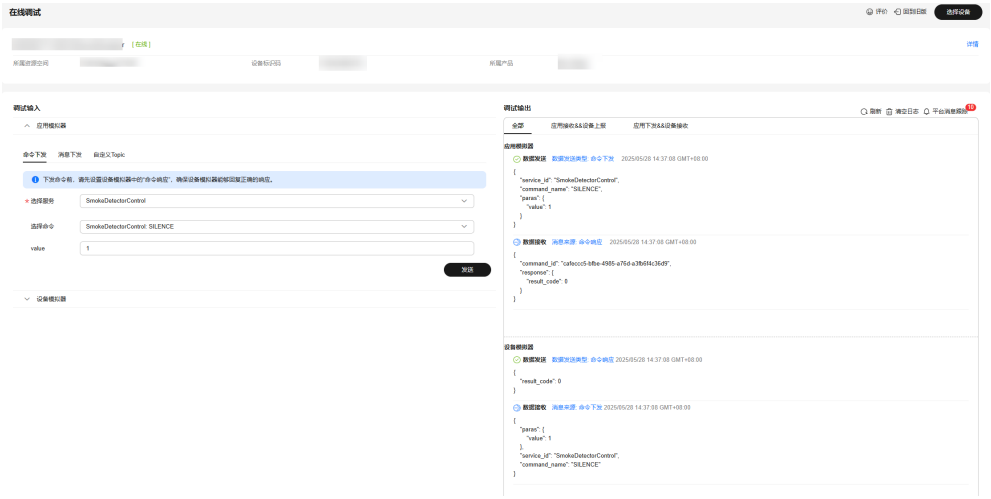
步骤7 在“应用模拟器”区域，针对您实际的使用场景，可以选择命令下发，消息下发以及自定义Topic消息下发功能模拟应用侧进行数据的发送，以命令下发为例，切换到应用模拟器”命令下发”页签，选择对应的服务和命令，并填写下发命令值后，单击“发送”，在右侧应用模拟器展示区域可查看下发的命令以及接收到的命令响应，在设备模拟器展示区域查看设备接收到的命令以及上报的命令响应。

说明

使用命令下发功能时，可在设备模拟器”命令响应”页签设置设备接收到命令后上报给平台的响应。

使用自定义Topic消息下发功能时，需要在设备模拟器”订阅列表”页签订阅对应的Topic。

图 3-103 在线调试-命令下发介绍



----结束

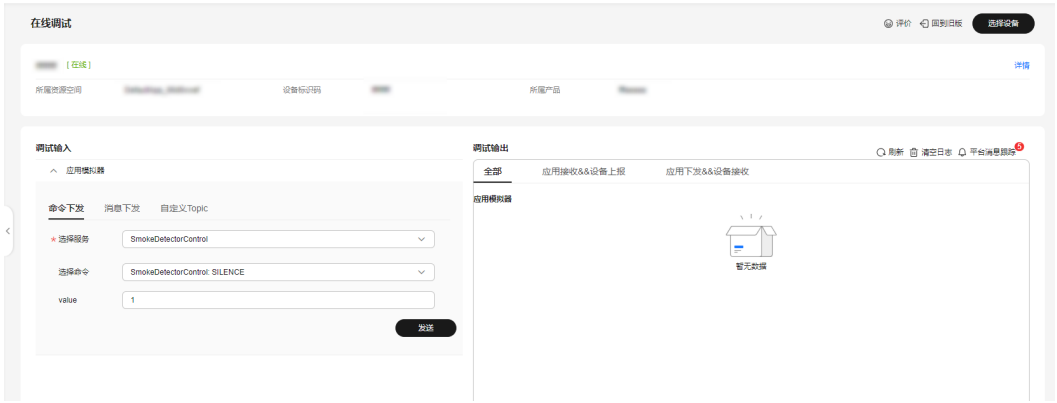
使用真实设备调测产品

当设备侧开发已经完成，但应用侧开发还未完成时，您可以创建真实设备，使用应用模拟器对设备、产品模型、插件等进行调测。真实设备调测界面分为以下几个部分：

1. 上方设备信息区域：展示当前正在调试的设备的基本信息，包括设备名称，设备状态，设备标识码，所属资源空间和产品等。

2. 左侧应用模拟器展示区域：可模拟应用实现命令下发，消息下发和自定义Topic消息下发等功能。
3. 应用模拟器展示区域：呈现应用服务器接收到和下发数据。

图 3-104 在线调试-真实设备结构



接下来，您可以创建真实设备进行在线调试。

步骤1 在产品详情中，选择“在线调测”，并单击“新增测试设备”。

步骤2 在弹出的“新增测试设备”窗口，选择“真实设备”，输入测试设备的参数，单击“确定”。

图 3-105 在线调试-新增测试设备

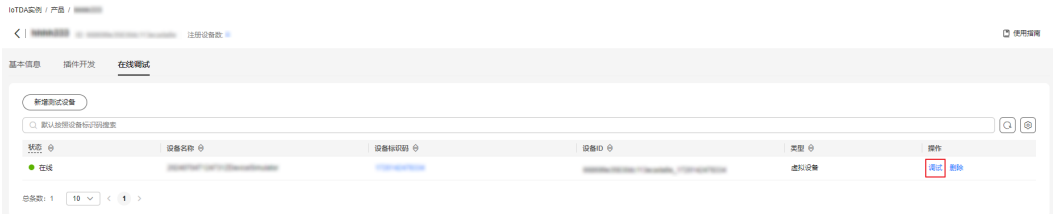
注：如果使用DTLS传输层安全协议接入时，请妥善保存密钥。

说明

新添加的设备处于未激活状态，此时不能进行在线调试，可参考**连接鉴权**，待设备接入平台后，进行调试。

步骤3 单击“调试”，进入调试界面。

图 3-106 在线调试-进入调试

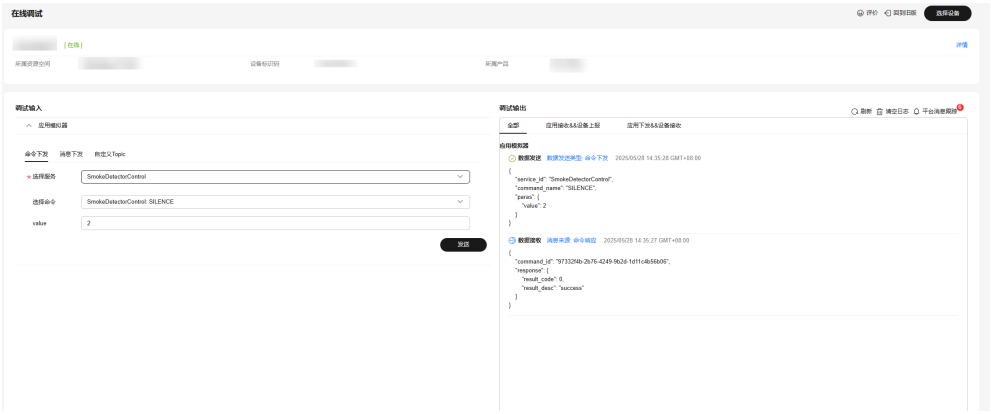


步骤4 进入在线调试页面，查看设备状态显示为”在线”。

图 3-107 在线调试-设备在线备

步骤5 在“应用模拟器”区域，针对您实际的使用场景，可以选择命令下发，消息下发以及自定义Topic消息下发功能模拟应用侧进行数据的发送，以命令下发为例，切换到应用模拟器”命令下发”页签，选择对应的服务和命令，并填写下发命令值后，单击“发送”，在右侧应用模拟器展示区域可查看下发的命令以及接收到的命令响应。在您的真实设备可以接收到下发的命令并执行相应的动作。

图 3-108 在线调试-真实设备示例



----结束

3.3 注册设备

3.3.1 注册单个设备

归属于某个产品下的设备实体，每个设备具有一个唯一的标识码。设备可以是直连物联网平台的设备，也可以是代理子设备连接物联网平台的网关。您可以在物联网平台注册您的实体设备，通过平台分配的设备ID和密钥，在集成了SDK后，您的设备可以接入到物联网平台，实现与平台的通信及交互。

物联网平台支持通过应用服务器调用**创建设备**接口，或者在控制台上注册单个设备。本文介绍如何在控制台上注册单个设备。

操作步骤

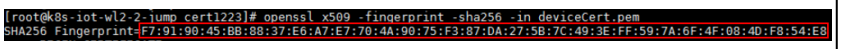
步骤1 访问**设备接入服务**，单击“管理控制台”进入设备接入控制台。选择您的实例，单击实例卡片进入。

步骤2 在左侧导航栏选择“设备 > 所有设备”，单击“注册设备”，按照如下表格填写参数后，单击“确定”。

图 3-109 设备-注册密钥设备

表 3-21 注册密钥设备

参数名称	说明
所属资源空间	选择设备所属的资源空间。

参数名称	说明
所属产品	选择设备所属的产品。 只有在**这里**创建了产品，此处才可以选择具体的产品。如没有，请先创建产品。
设备标识码	即node_id，填写为设备的IMEI、MAC地址或Serial No；若没有真实设备，填写自定义字符串，由英文字母、数字、连接号-和下划线_组成。
设备ID	设备ID，用于唯一标识一个设备。如果携带该参数，平台将设备ID设置为该参数值；如果不携带该参数，设备ID由物联网平台分配获得，生成规则为product_id + _ + node_id拼接而成。
设备名称	即device_name，可自定义。
设备描述	设备的描述信息，可自定义。
设备认证类型	• 密钥：设备通过密钥验证身份。 • X.509证书：设备使用X.509证书验证身份。
密钥	设备密钥，可自定义，不填写物联网平台会自动生成。
指纹	当“设备认证类型”选择“X.509证书”时填写，导入**设备侧预置的设备证书**对应的指纹，在OpenSSL执行**openssl x509 -fingerprint -sha256 -in deviceCert.pem**命令可查询。  填写时需要删除冒号。

设备注册成功后，请妥善保管好设备ID和密钥，用于设备接入平台认证。

图 3-110 设备-注册设备成功

说明

若密钥丢失，可以按照**设备认证凭证管理**中的步骤更新设备密钥，无法找回注册设备时生成的密钥。

用户可在设备列表删除不再使用的设备。删除设备不支持撤回，请谨慎操作。

----结束

相关 API 参考

- **查询设备列表**
- **创建设备**
- **查询设备**
- **修改设备**
- **删除设备**
- **重置设备密钥**

3.3.2 批量注册设备

物联网平台支持通过应用服务器调用**创建批量任务**接口，或者在控制台上批量注册设备。本文介绍如何在控制台上批量注册设备。

操作步骤

步骤1 访问**设备接入服务**，单击“管理控制台”进入设备接入控制台。

步骤2 在左侧导航栏选择“设备 > 所有设备”，进入“批量注册”页签，单击“批量注册”。

步骤3 弹出批量注册设备窗口，填写“任务名称”，下载并填写“批量注册设备文件模板”内容并上传文件，单击“确定”创建任务。

图 3-111 设备-批量注册设备

步骤4 批量注册执行成功，如果是原生MQTT设备注册，请单击批量任务一行，进入任务的“执行详情”，保存好设备ID和密钥，用于原生MQTT设备接入平台。

图 3-112 批量注册设备-执行详情

----结束

相关 API 参考

- 创建设备
- 查询批量任务列表
- 创建批量任务
- 查询批量任务

3.3.3 注册 X.509 证书认证的设备

X.509是一种用于通信实体鉴别的数字证书，物联网平台支持设备使用自己的X.509证书进行认证鉴权。使用X.509认证技术时，设备无法被仿冒，避免了密钥被泄露的风险。

注册X.509证书认证的设备前，您需要先在物联网平台上传设备的CA证书，然后在注册设备时将设备证书同设备进行绑定。本文介绍如何在物联网平台上传设备CA证书，以及注册X.509证书认证的设备。

限制说明

- 当前只有通过MQTT接入的设备支持使用X.509证书进行设备身份认证。
- 每个用户最多上传100个设备CA证书。

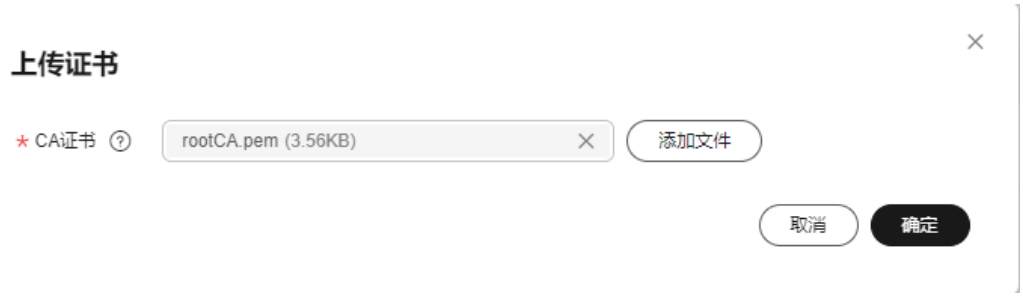
上传设备 CA 证书

步骤1 访问**设备接入服务**，单击“管理控制台”进入设备接入控制台。选择您的实例，单击实例卡片进入。

步骤2 在左侧导航栏选择“设备 > 设备证书”，进入“设备CA证书”页签，单击“上传证书”。

步骤3 在弹出的对话框中，单击“添加文件”，然后单击“确定”。

图 3-113 设备 CA 证书-上传证书



说明

设备CA证书由设备厂商提供，调测时可**自行制作调测证书**，商用时建议更换为商用证书，否则会带来安全风险。如果已购买CA证书（pem、jks等），可直接上传到平台。

----结束

制作设备 CA 调测证书

本文以Windows环境为例，介绍通过Openssl工具制作调测证书的方法，生成的证书为PEM编码格式的证书。

1. 在浏览器中访问**这里**，下载并进行安装OpenSSL工具。
2. 以管理员身份运行cmd命令行窗口。
3. 执行cd c:\openssl\bin（请替换为openssl实际安装路径），进入openssl命令视图。
4. 执行以下命令生成生成密钥对。
   ```
   openssl genrsa -out rootCA.key 2048
   ```
5. 执行以下命令，使用密钥对中的私有密钥生成 CA 证书。
   ```
   openssl req -x509 -new -nodes -key rootCA.key -sha256 -days 1024 -out rootCA.pem
   ```
   系统提示您输入如下信息，所有参数可以自定义。
   - Country Name (2 letter code) [AU]：国家，如CN。
   - State or Province Name (full name) []: 省份，如GD。
   - Locality Name (for example, city) []:城市，如SZ。
   - Organization Name (for example, company) []：组织，如Huawei。

- Organizational Unit Name (for example, section) []：组织单位，如IoT。
- Common Name (e.g. server FQDN or YOUR name) []: 名称，如zhangsan。
- Email Address []:邮箱地址，如1234567@163.com。

在openssl安装目录的bin文件夹下，获取生成的CA证书（rootCA.pem）。

上传验证证书

如果上传的是调测证书，上传后证书状态显示为“未验证”，您需要上传验证证书，来证明您拥有该CA证书。

图 3-114 设备 CA 证书-未验证证书

验证证书是由设备CA证书对应的私钥创建的，请参考如下操作制作验证证书。

步骤1 执行如下命令为私有密钥验证证书生成密钥对。

```
openssl genrsa -out verificationCert.key 2048
```

步骤2 执行如下命令为私有密钥验证证书创建CSR（Certificate Signing Request）。

```
openssl req -new -key verificationCert.key -out verificationCert.csr
```

系统提示您输入如下信息，**Common Name填写为验证证书的验证码**，其他参数自定义。

- Country Name (2 letter code) [AU]：国家，如CN。
- State or Province Name (full name) []: 省份，如GD。
- Locality Name (for example, city) []:城市，如SZ。
- Organization Name (for example, company) []：组织，如Huawei。
- Organizational Unit Name (for example, section) []：组织单位，如IoT。
- Common Name (e.g. server FQDN or YOUR name) []：验证证书的验证码，请参考**步骤5**获取 。
- Email Address []:邮箱地址，如1234567@163.com。
- Password[]：密码，如1234321。
- Optional Company Name[]：公司名称，如Huawei。

步骤3 执行以下命令使用CSR创建私有密钥验证证书。

```
openssl x509 -req -in verificationCert.csr -CA rootCA.pem -CAkey rootCA.key -CAcreateserial -out verificationCert.pem -days 500 -sha256
```

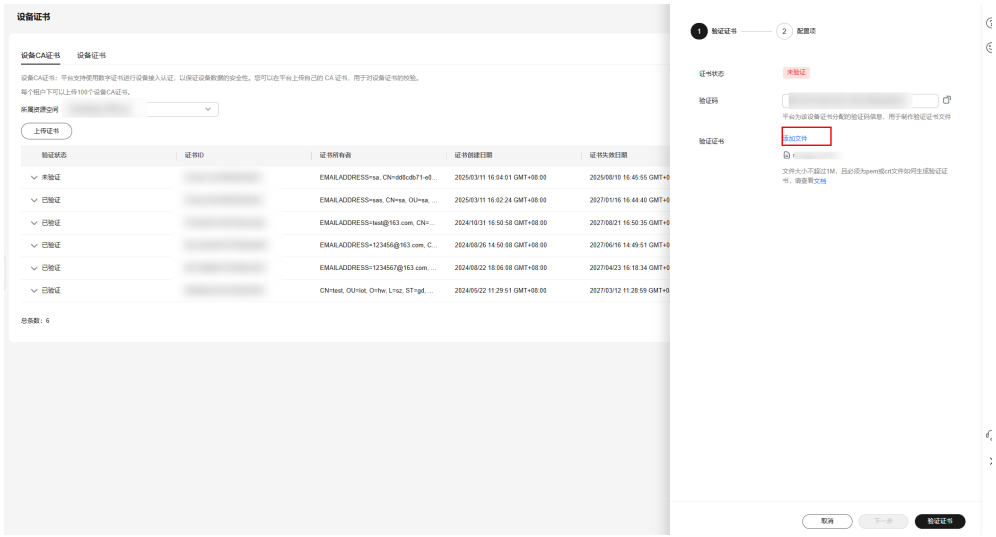
在openssl安装目录的bin文件夹下，获取生成的验证证书（verificationCert.pem）。

步骤4 选择对应证书，单击 ⌄ 然后单击“上传验证证书”。

图 3-115 设备 CA 证书-验证证书

步骤5 在弹出的对话框中已显示验证码，只需单击“添加文件”，上传验证证书后单击“确定”，上传后证书状态变为“已验证”，表明您拥有该CA证书。

图 3-116 设备 CA 证书-上传验证证书



----结束

删除设备 CA 证书

针对不再使用的设备CA证书，您可以单击右侧的“删除”按钮进行删除。

说明

删除服务设备CA证书后，使用此证书进行鉴权的设备将无法接入平台，在删除前请确保已经完成了备份，并谨慎操作。

图 3-117 设备 CA 证书-删除证书

预置 X.509 证书

在注册X.509设备之前，您需要在设备侧预置CA机构签发的X.509证书。

说明

X.509证书由CA机构签发，若没有CA机构签发的商用证书，您可以自己**制作X.509调测证书**。如果已购买相关证书，或者权威机构签发的证书（pem、jks等），可直接上传到平台。

制作X.509调测证书

1. 以管理员身份运行cmd命令行窗口，执行cd c:\openssl\bin（请替换为openssl实际安装路径），进入openssl命令视图。
2. 执行如下命令生成密钥对。
   ```
   openssl genrsa -out deviceCert.key 2048
   ```
3. 执行如下命令为设备证书创建CSR（Certificate Signing Request）。
   ```
   openssl req -new -key deviceCert.key -out deviceCert.csr
   ```
   系统提示您输入如下信息，所有参数可以自定义。
   - Country Name (2 letter code) [AU]：国家，如CN。
   - State or Province Name (full name) []: 省份，如GD。
   - Locality Name (for example, city) []:城市，如SZ。
   - Organization Name (for example, company) []：组织，如Huawei。
   - Organizational Unit Name (for example, section) []：组织单位，如IoT。
   - Common Name (e.g. server FQDN or YOUR name) []: 名称，如zhangsan。
   - Email Address []:邮箱地址，如1234567@163.com。
   - Password[]：密码，如1234321。
   - Optional Company Name[]：公司名称，如Huawei。
4. 执行以下命令使用CSR创建设备证书。
   ```
   openssl x509 -req -in deviceCert.csr -CA rootCA.pem -CAkey rootCA.key -CAcreateserial -out deviceCert.pem -days 500 -sha256
   ```
   在openssl安装目录的bin文件夹下，获取生成的设备证书（deviceCert.pem）。

注册 X.509 证书认证的设备

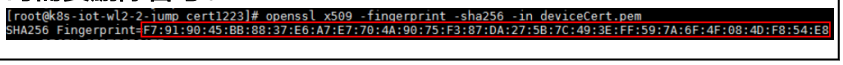
步骤1 访问**设备接入服务**，单击“管理控制台”进入设备接入控制台。选择您的实例，单击实例卡片进入。

步骤2 在左侧导航栏选择“设备 > 所有设备”，单击“注册设备”，按照如下表格填写参数后，单击“确定”。

图 3-118 设备-注册 X.509 设备

表 3-22 注册 X.509 设备

参数名称	说明
所属资源空间	选择设备所属的资源空间。
所属产品	选择设备所属的产品。 只有在**这里**创建了产品，此处才可以选择具体的产品。如没有，请先创建产品。
设备标识码	即node_id，填写为设备的IMEI、MAC地址或Serial No；若没有真实设备，填写自定义字符串，由英文字母、数字、连接号-和下划线_组成。
设备ID	设备ID，用于唯一标识一个设备。如果携带该参数，平台将设备ID设置为该参数值；如果不携带该参数，设备ID由物联网平台分配获得，生成规则为product_id + _ + node_id拼接而成。
设备名称	即device_name，可自定义。
设备描述	设备的描述信息，可自定义。
设备认证类型	X.509证书：设备使用X.509证书验证身份。

参数名称	说明
指纹	当“设备认证类型”选择“X.509证书”时填写，导入**设备侧预置的设备证书**对应的指纹，在OpenSSL执行**openssl x509 -fingerprint -sha256 -in deviceCert.pem**命令可查询。**注：填写时需要删除冒号**。 

----结束

相关 API 参考

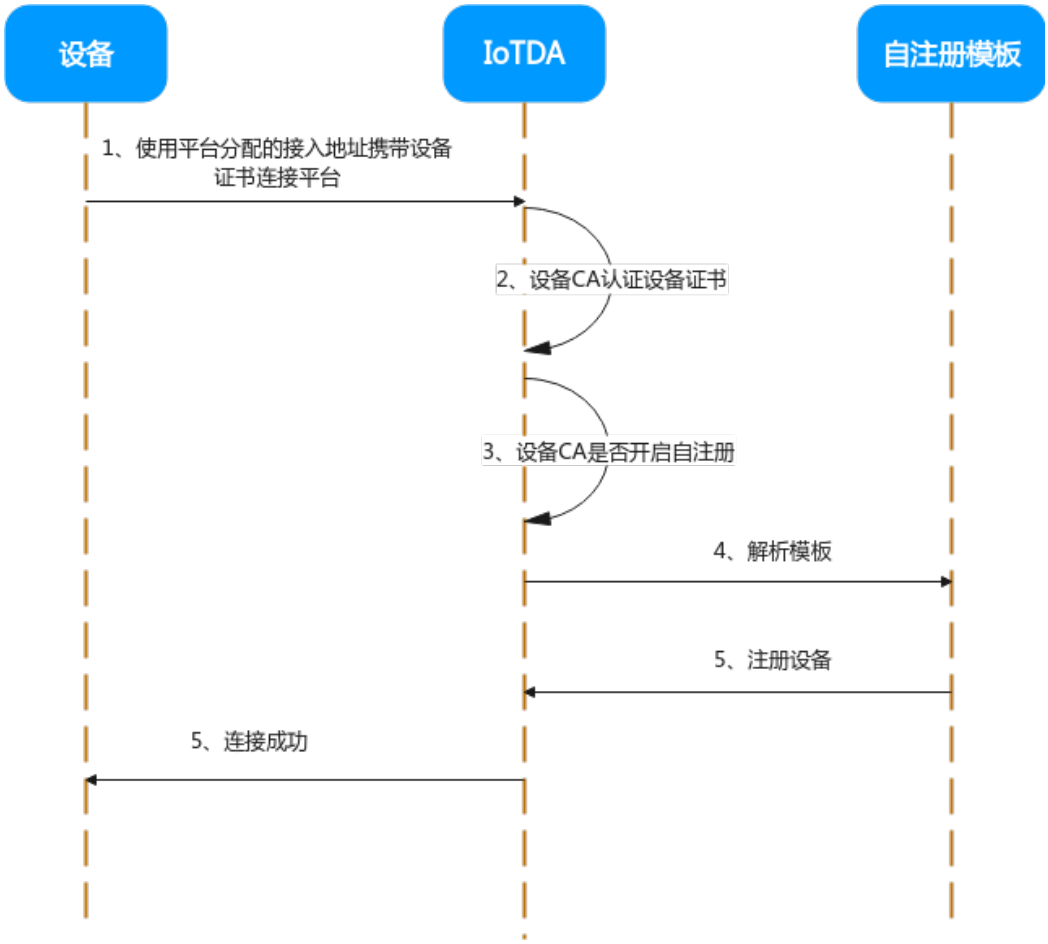
- **获取设备CA证书列表**
- **上传设备CA证书**
- **删除设备CA证书**
- **验证设备CA证书**

3.3.4 设备自注册

概述

设备自注册功能可以让用户没有在物联网平台上注册设备的前提下，在设备首次接入平台时自动完成设备的注册 ，设备自注册基于证书认证实现，通过设备证书携带设备信息。

图 3-119 设备自注册原理图



使用场景

- 自动注册设备：适用于不能提前在控制台（console）或通过调用API创建设备的场景，设备可以直接完成注册。
- 车联网场景：在车机启动时，设备连接平台会自动注册，上报数据，从而简化车端应用侧的开发。
- 多实例场景：企业客户购买多个IoTDA实例时，使用自注册功能可以无需提前在不同实例上发放注册设备，轻松实现设备跨实例管理。

使用限制

- 单实例下自注册模板最多可以创建10个。
- 使用设备自注册功能，要求设备必须使用TLS协议，并且同时开启**服务器名称指示（SNI）**扩展，SNI中需要携带平台分配的域名（参考**如何获取平台接入地址**）。
- 目前该功能仅支持MQTTS证书认证的场景。
- 上海一标准版目前暂不支持该功能。

须知

设备自注册后，设备鉴权依赖于自注册模板，修改或停用自注册模板可能会影响设备鉴权，请谨慎操作。

操作步骤

步骤1 访问**设备接入服务**，单击“管理控制台”进入设备接入控制台。选择您的实例，单击实例卡片进入。

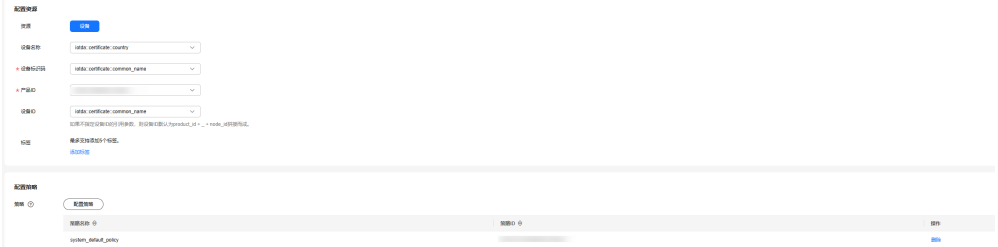
步骤2 创建产品详细步骤参考：**创建产品**。

步骤3 左侧导航栏选择“设备 > 自注册模板”，单击“创建模板”进入创建自注册模板页面，输入基本信息后输入单击“添加参数”选择对应的参数添加到模板；

图 3-120 自注册模板-添加参数

步骤4 在配置资源栏中选择设备名称、设备标识码、设备ID和产品ID。

图 3-121 自注册模板-创建模板



说明

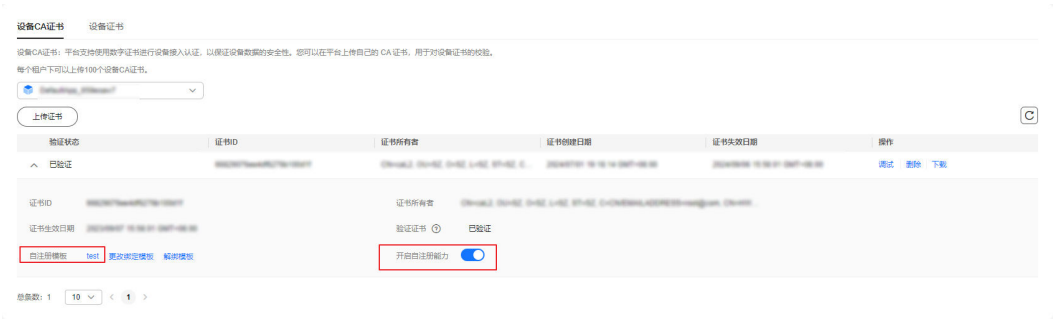
以下为平台预定义的在模板中可声明和引用的参数，证书必须包含模板中所引用的参数信息：

- iotda::certificate::country：国家
- iotda::certificate::organization：组织
- iotda::certificate::organizational_unit：部门
- iotda::certificate::distinguished_name_qualifier：可分辨名称
- iotda::certificate::state_name：省份
- iotda::certificate::common_name：通用名称
- iotda::certificate::serial_number：序列号

步骤5 在配置策略栏中添加策略，设备自注册时将会自动绑定所添加的策略，有关设备策略详细使用参考：**设备Topic策略**。

步骤6 左侧导航栏选择“设备 > 设备证书”进入设备证书页面，参考 **X.509证书认证的设备**，上传CA证书到平台并通过验证，绑定**步骤3**所创建的自注册模板并且开启自注册能力。

图 3-122 设备 CA 证书-绑定模板

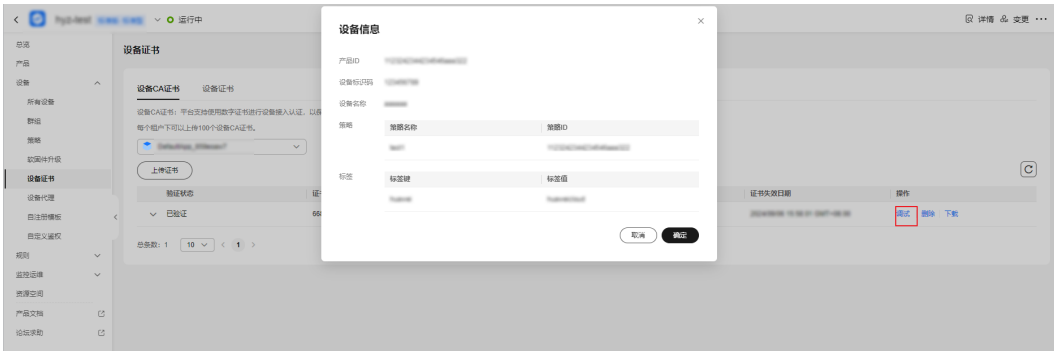


说明

注册设备所在的资源空间即为设备CA证书所在资源空间，请确保CA证书与模板中的产品ID在同一个资源空间下。

步骤7 在设备CA证书页签单击“调试”，上传**步骤6**制作的设备证书，查看预解析后的设备信息是否符合预期。

图 3-123 设备 CA 证书-调试证书



----结束

设备模拟器验证

使用MQTT.fx工具模拟设备接入平台完成设备自动注册。

步骤1 下载**MQTT.fx**（默认是64位操作系统，如果是32位操作系统，单击此处下载**MQTT.fx**），安装MQTT.fx工具。

步骤2 打开MQTT.fx软件，参考**表3-23**设置连接参数后单击“Apply”应用。

表 3-23 连接参数

参数	说明
Broker Address	平台接入地址（参考**如何获取平台接入地址**）。
Broker Port	8883
Client ID	无要求，任意字符串（建议按照**设备连接鉴权**的ClientId字段填写，按照平台规则填写的设备在模板停用后仍然可以通过平台的证书认证接入）。
User Name	无要求，任意字符串（建议按照**设备连接鉴权**的UserName字段填写，按照平台规则填写的设备在模板停用后仍然可以通过平台的证书认证接入）。
Password	空
Enable SSL/TLS	true
Self signed certificates	true
CA File	平台CA证书（参考**证书资源**）。
Client Certificate File	设备证书文件路径。
Client Key File	设备证书私钥文件路径。
Client Key Password	私钥密码，无密码则不需要填写。

图 3-124 MQTT.fx 设置

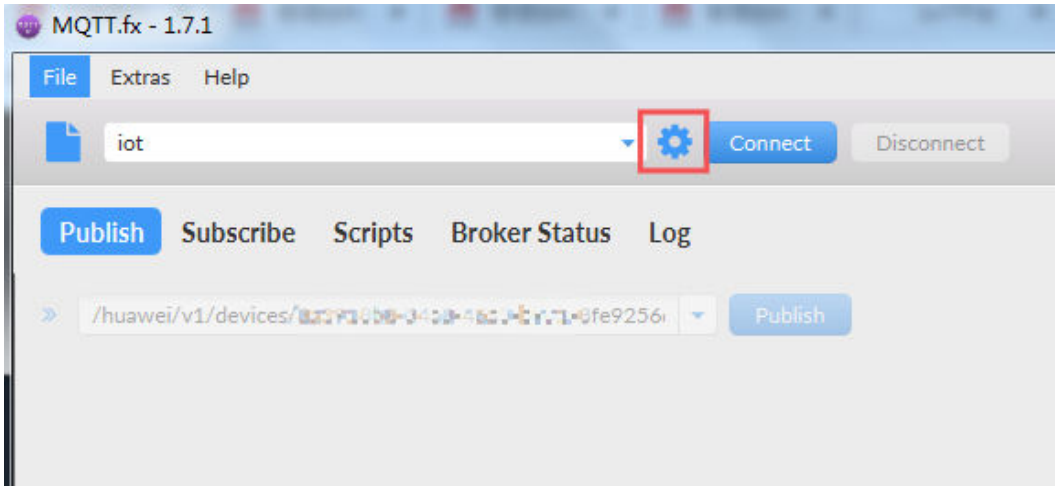


图 3-125 连接参数图

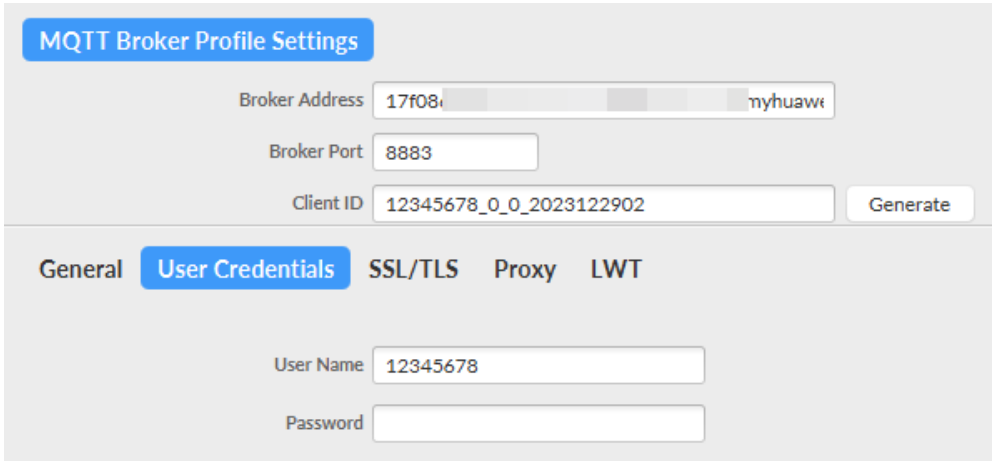
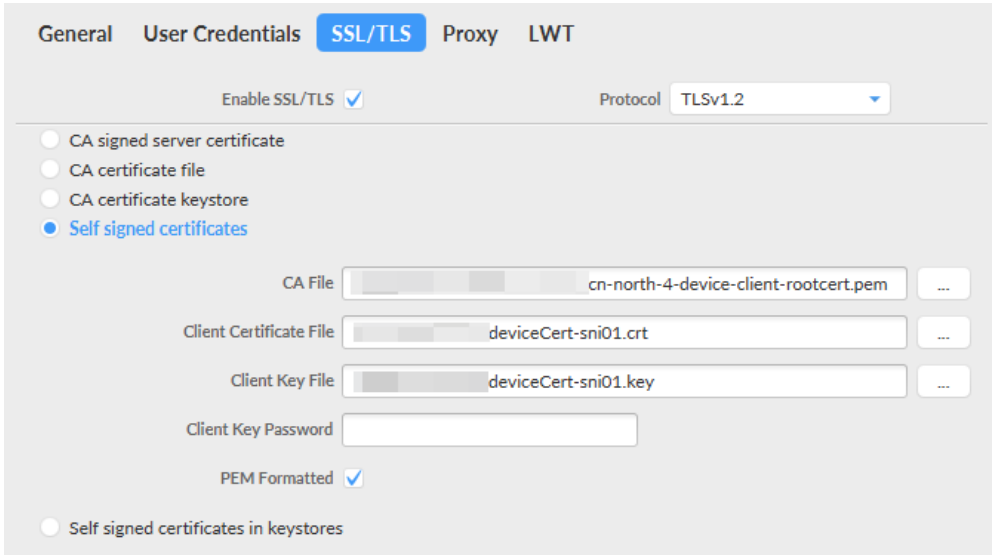
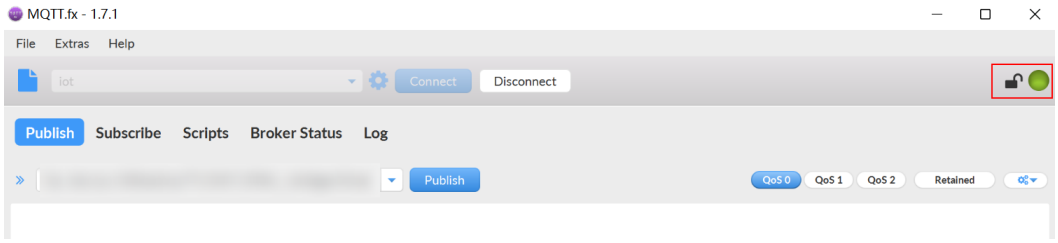


图 3-126 证书信息图



步骤3 单击“Connect”连接平台，看到MQTT.fx界面右上角圆圈转为绿色，即说明设备模拟器鉴权连接成功。

图 3-127 设备模拟器连接成功



步骤4 进入设备接入控制台左侧导航栏，选择“设备 > 所有设备”，在“设备列表”页签搜索框中通过“设备ID”或者“设备标识码”搜索设备，设备已经注册并处于在线状态。

图 3-128 设备-自注册设备详情

----结束

3.4 设备集成设备端 SDK 接入

华为云设备接入云服务（IoTDA）是提供海量设备的接入和管理能力的物联网平台，将物理设备联接到云，支撑设备数据采集上云和云端下发命令给设备进行远程控制。IoTDA配合华为云其他产品，可以帮助您快速构筑物联网解决方案。

接下来，我们将**以Java SDK为例**，体验如何使用IoT Device SDK进行代码场景适配、并将该SDK集成入IoT设备实现设备快速接入IoTDA平台。本章节以一个自定义燃气表的产品模型为例，介绍如何适配SDK代码以实现数据（消息、属性）上报，上报燃气表数据；下发命令，对燃气表进行远程配置和控制，最后将适配好的代码集成入设备中。

说明

设备侧SDK不绑定某个特定的产品模型，本章节使用自定义燃气表仅为举例，用户可根据自己实际情况适配代码。

设备侧 SDK 全表

表 3-24 设备侧 SDK 全表

资源包名	描述	下载路径
设备侧 IoT Device Java SDK	设备可以通过集成IoT Device SDK(Java)接入物联网平台, Demo提供了调用SDK接口的样例代码。使用指导请参考 **设备侧 IoT Device Java SDK使用指南**。	**设备侧 IoT Device Java SDK**
设备侧 IoT Device C for Linux/Windows SDK	设备可以通过集成IoT Device SDK(C)接入物联网平台, Demo提供了调用SDK接口的样例代码。使用指导请参考 **设备侧 IoT Device C for Linux/Windows SDK使用指南**。	**设备侧 IoT Device C for Linux/Windows SDK**
设备侧 IoT Device C# SDK	设备可以通过集成IoT Device SDK(C#)接入物联网平台, Demo提供了调用SDK接口的样例代码。使用指导请参考 **设备侧 IoT Device C# SDK使用指南**。	**设备侧 IoT Device C# SDK**
设备侧 IoT Device Android SDK	设备可以通过集成IoT Device SDK(Android)接入物联网平台, Demo提供了调用SDK接口的样例代码。使用指导请参考 **设备侧 IoT Device Android SDK使用指南**。	**设备侧 IoT Device Android SDK**
设备侧 IoT Device Go SDK(社区版)	设备可以通过集成IoT Device SDK(Go)接入物联网平台, Demo提供了调用SDK接口的样例代码。使用指导请参考 **设备侧 IoT Device Go SDK(社区版)使用指南**。	**设备侧 IoT Device Go SDK(社区版)**

资源包名	描述	下载路径
设备侧 IoT Device Python SDK	设备可以通过集成IoT Device SDK(Python)接入物联网平台, Demo提供了调用SDK接口的样例代码。使用指导请参考 **设备侧 IoT Device Python SDK使用指南**。	**设备侧 IoT Device Python SDK**
设备侧 IoT Device C for Linux/Windows SDK Tiny	设备可以通过集成IoT Device SDK Tiny (C)接入物联网平台, Demo提供了调用SDK接口的样例代码。使用指导请参考 **设备侧 IoT Device C for Linux/Windows SDK Tiny使用指南**	**设备侧 IoT Device C for Linux/Windows SDK Tiny**
设备侧 IoT Device Arkts(OpenHarmony) SDK	设备可以通过集成IoT Device SDK (ArkTS)接入物联网平台, Demo提供了调用SDK接口的样例代码。使用指导请参考 **设备侧 IoT Device Arkts(OpenHarmony) SDK使用指南**	**设备侧 IoT Device Arkts(OpenHarmony) SDK**

表 3-25 SDK 主要功能矩阵

主要功能	C	Java	C#	Android	GO	python	C Tiny	ArkTS
属性上报	√	√	√	√	√	√	√	√
消息上报、下发	√	√	√	√	√	√	√	√
事件上报、下发	√	√	√	√	√	√	√	×
命令下发、响应	√	√	√	√	√	√	√	√
设备影子	√	√	√	√	√	√	√	√
OTA升级	√	√	√	√	√	√	√	×
bootstrap	√	√	√	√	√	√	√	×
时间同步	√	√	√	√	√	√	√	×

主要功能	C	Java	C#	Android	GO	python	C Tiny	ArkTS
网关与子设备管理	√	√	√	√	√	√	√	×
端侧规则引擎	√	×	√	×	×	×	√	×
远程SSH	√	×	√	×	×	×	×	×
异常检测	√	×	√	×	×	×	×	×
端云安全通信（软总线）	√	×	√	×	×	×	×	×
M2M功能	√	×	√	×	×	×	×	×
泛协议接入	√	√	√	√	×	√	×	×

前提条件

- 开发环境要求：已安装java的集成环境（IntelliJ IDEA），并配置好Maven等相关环境。
- 本示例使用的是设备侧的MQTTS协议。
- 已注册华为云官方账号并完成实名制认证。
- 已开通设备接入服务。未开通则访问**设备接入服务**，单击“立即使用”后开通该服务。

业务流程

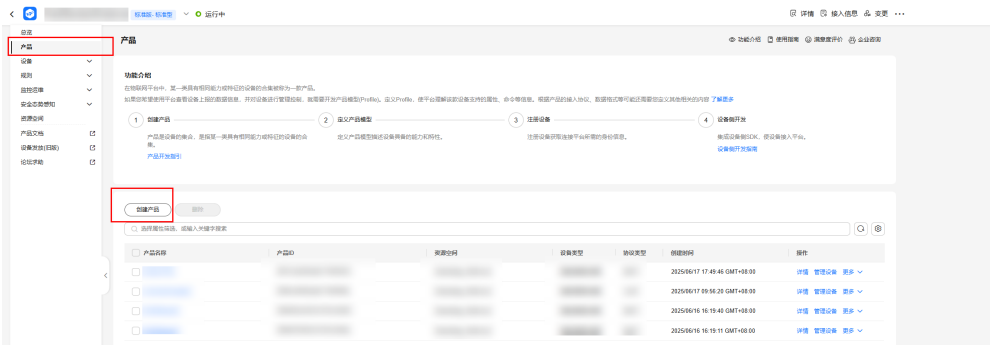
使用IoT Device SDK(Java)将设备接入华为云IoTDA平台，进行数据上报、命令下发等业务的体验。

1. **创建产品**。创建一个MQTT协议的产品。
2. **开发产品模型**。通过定义产品模型，在物联网平台构建一款燃气表设备，支持上报燃气表读数、远程给燃气表下发配置和命令。
3. **注册设备**。注册一个MQTT协议的设备。
4. **使用Java SDK接入华为云**。下载SDK、适配代码并使用Java SDK激活在物联网平台上注册的设备。
5. **消息上报**。适配代码、使用Java SDK向物联网平台上报消息。
6. **属性上报**。适配代码、使用Java SDK向物联网平台设备属性数据。
7. **命令下发**。适配代码、在管理控制台下发命令，远程设置设备属性。
8. **设备集成Java SDK并运行**。将适配好的SDK打包成可运行文件，导入IoT设备并运行，完成设备连接IoTDA平台。

创建产品

步骤1 访问**设备接入服务**，单击“管理控制台”进入设备接入控制台，选择您的实例，单击实例卡片进入。单击左侧导航栏“产品”，单击页面左侧的“创建产品”。

图 3-129 产品-创建产品



步骤2 建一个协议类型为MQTT协议、自定义设备类型为“自定义燃气表”的产品，参考下表及页面提示填写参数后，单击“确定”。

表 3-26 创建产品参数

所属资源控制	选择默认资源空间
产品名称	自定义，如“燃气表模型”
协议类型	选择“MQTT”
数据格式	选择“JSON”
设备类型选择	选择“自定义类型”
设备类型	自定义，如“自定义燃气表模型”

图 3-130 创建产品-MQTT

----结束

开发产品模型

步骤1 找到新增的产品，单击产品进入产品界面。

步骤2 在产品详情“基本信息”页面，单击“自定义模型”，配置产品的服务。

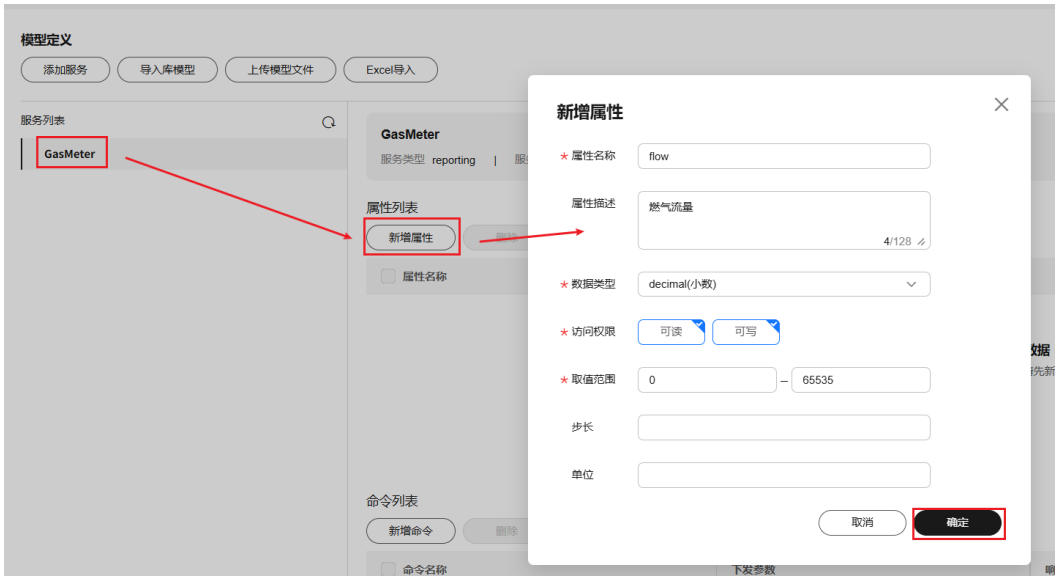
图 3-131 自定义模型-MQTT

步骤3 新增服务类型“GasMeter”。单击“自定义模型”后，会弹出“添加服务”页面，根据页面提示填写“服务ID”、“服务类型”和“服务描述”，单击“确定”。

图 3-132 添加服务-GasMeter

步骤4 新增属性物模型。单击“GasMeter”服务列表中的“新增属性”，按照下图填写相关信息后，单击“确定”。

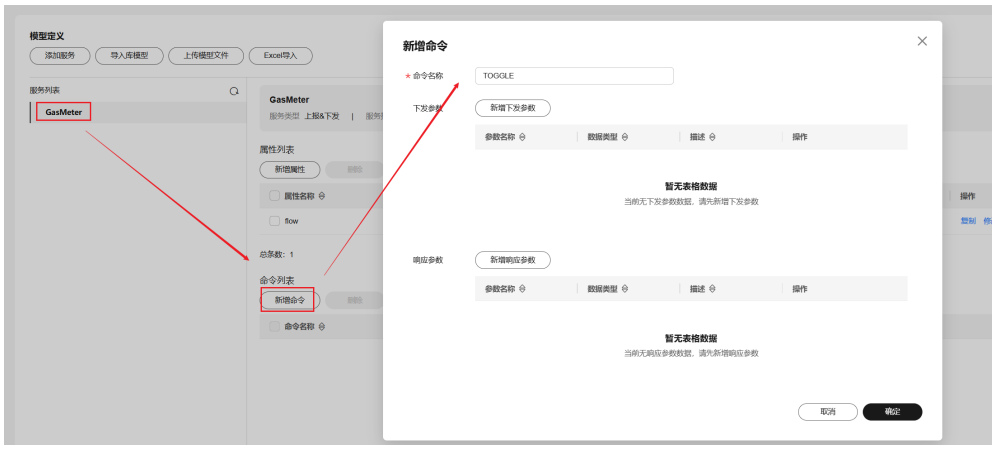
图 3-133 新增属性-flow



步骤5 新增命令模型。

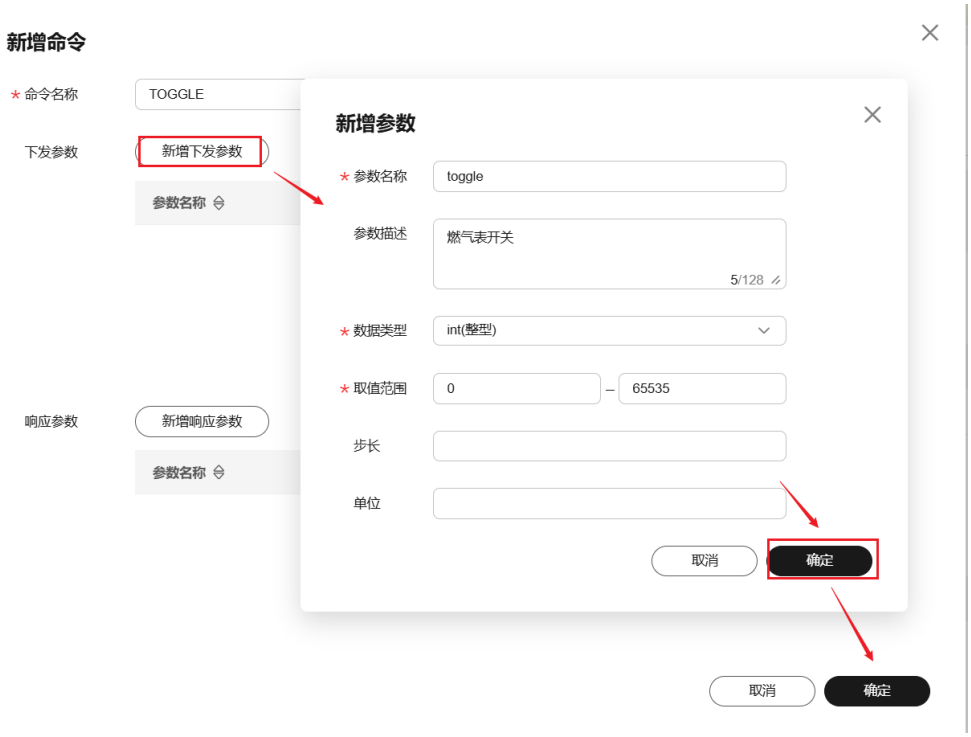
1. 单击“GasMeter”服务列表中的“新增命令”，设置命令名称为“TOGGLE”。

图 3-134 新增命令-TOGGLE



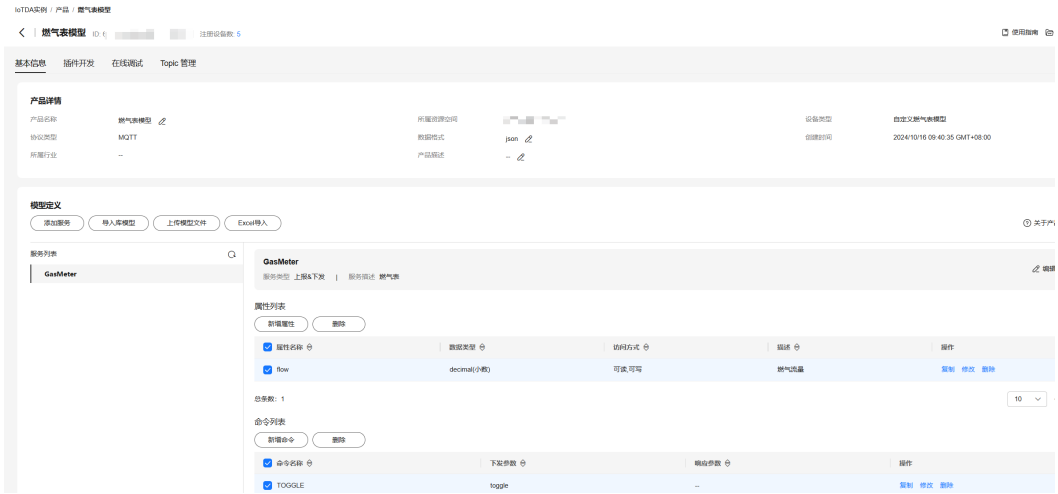
2. 在“新增命令”页面，单击“新增下发参数”，按照下图填写相关信息后，单击“确定”。

图 3-135 新增命令-toggle



步骤6 上述步骤完成后，可见物模型如下图所示：

图 3-136 物模型-燃气表模型



----结束

注册设备

步骤1 在设备接入控制台页面，选择您的实例，选择左侧导航栏“设备 > 所有设备”，单击“注册设备”。

图 3-137 所有设备-注册设备

步骤2 根据页面提示信息填写参数，然后单击“确定”。

参数名称	说明
所属资源空间	确保和所属产品归属在同一个资源空间。
所属产品	选择对应产品。
设备标识码	即nodeID，设备唯一物理标识。设备标识码长度为4至64个字符，只允许字母、数字、下划线（_）、连接符（-）的组合。可自定义由英文字母和数字组成。
设备名称	即device_name，可自定义。
设备认证类型	选择“密钥”。
密钥	此处如不填写，物联网平台会自动生成。

图 3-138 单设备注册-MQTT

步骤3 成功注册设备后，平台会自动生成设备ID和密钥，请妥善保管好设备ID（deviceId）和密钥（deviceSecret），用于设备接入。

图 3-139 设备-注册设备成功

----结束

使用 Java SDK 接入华为云

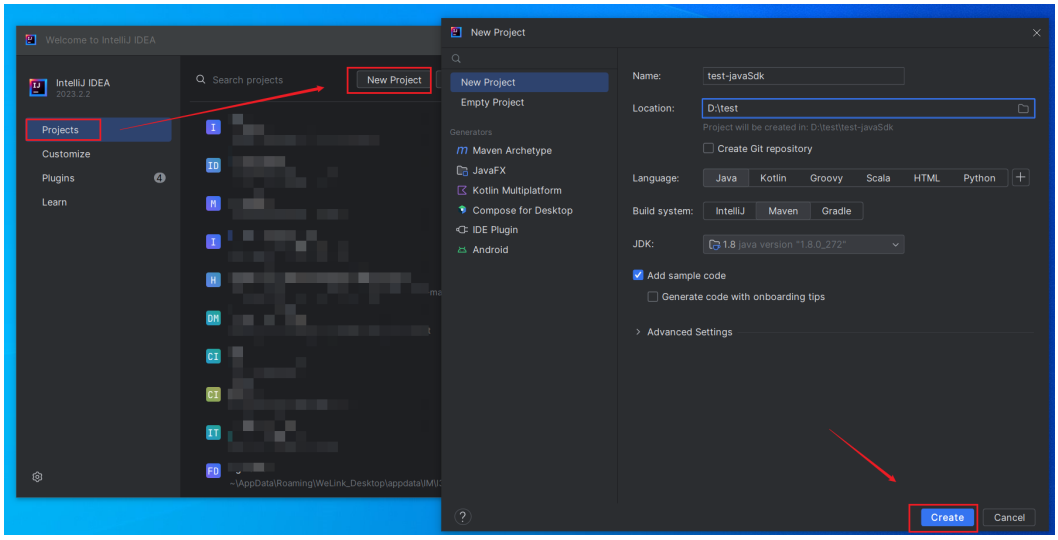
接下来将使用IntelliJ IDEA（以下截图为IntelliJ IDEA 2023社区版）进行代码编写及调试。请先确认IDEA环境完好，Maven可用。Java SDK使用说明及接口：**README.md**。

说明

推荐用户首先在电脑上完成SDK代码适配和设备激活，再将修改好的代码导入设备中完成集成。

步骤1 新建项目。打开IntelliJ IDEA，单击“New Project”，在弹出的界面填写项目名及新建项目地址，语言选择“java”，构建系统选择“Maven”，单击“Create”。

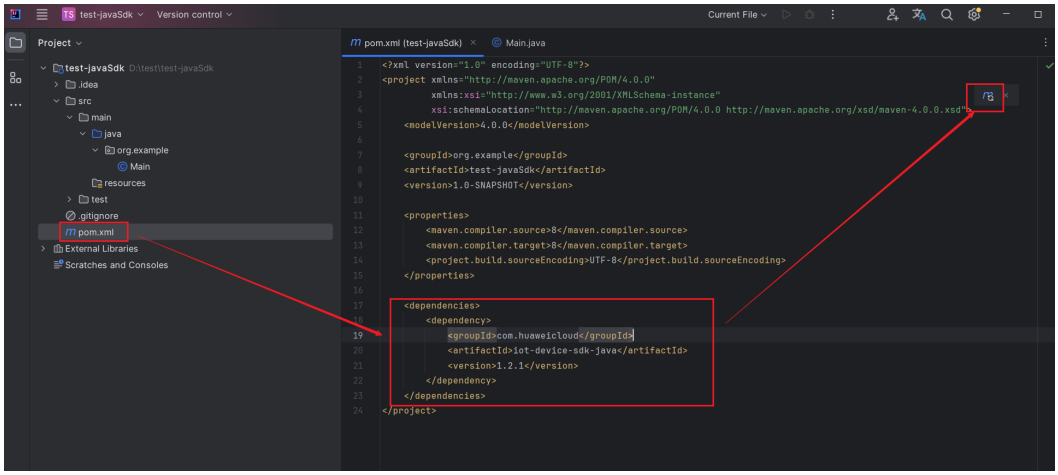
图 3-140 创建 java 项目



步骤2 添加Mvn引用。创建成功后，可在项目中看到自动生成了pom.xml及Main.java文件。打开pom.xml文件，添加Java SDK的Mvn引用；单击右上角Mvn更新图标，确认是否报错。若是报错，请确认Mvn配置。

```
<dependencies>
   <dependency>
      <groupId>com.huaweicloud</groupId>
      <artifactId>iot-device-sdk-java</artifactId>
      <version>1.2.1</version>
   </dependency>
</dependencies>
```

图 3-141 添加 Java SDK Mvn 引用



步骤3 编写引用代码，实现设备建链。

1. 打开Main.java文件，将下面代码复制到Main.java文件中。

```
import com.huaweicloud.sdk.iot.device.IoTDevice;
import java.io.File;
import java.io.IOException;
import java.io.InputStream;
import java.nio.file.Files;
import static java.nio.file.StandardCopyOption.REPLACE_EXISTING;
public class Main {
    private static final String IOT_ROOT_CA_RES_PATH = "ca.jks";
    private static final String IOT_ROOT_CA_TMP_PATH = "huaweicloud-iotda-tmp-" +
```

```
IOT_ROOT_CA_RES_PATH;
    public static void main(String[] args) throws InterruptedException, IOException {
        // 加载iot平台的ca证书，进行服务端校验
        File tmpCAFile = new File(IOT_ROOT_CA_TMP_PATH);
        try (InputStream resource =
Main.class.getClassLoader().getResourceAsStream(IOT_ROOT_CA_RES_PATH)) {
            Files.copy(resource, tmpCAFile.toPath(), REPLACE_EXISTING);
        }
        // 创建设备并初始化. 用户请替换为自己的接入地址。
        IoTDevice device = new IoTDevice("ssl://xxx.st1.iotda-device.cn-
north-4.myhuaweicloud.com:8883",
                "5e06bfee334dd4f33759f5b3_demo", "mysecret", tmpCAFile);
        if (device.init() != 0) {
            return;
        }
    }
}
```

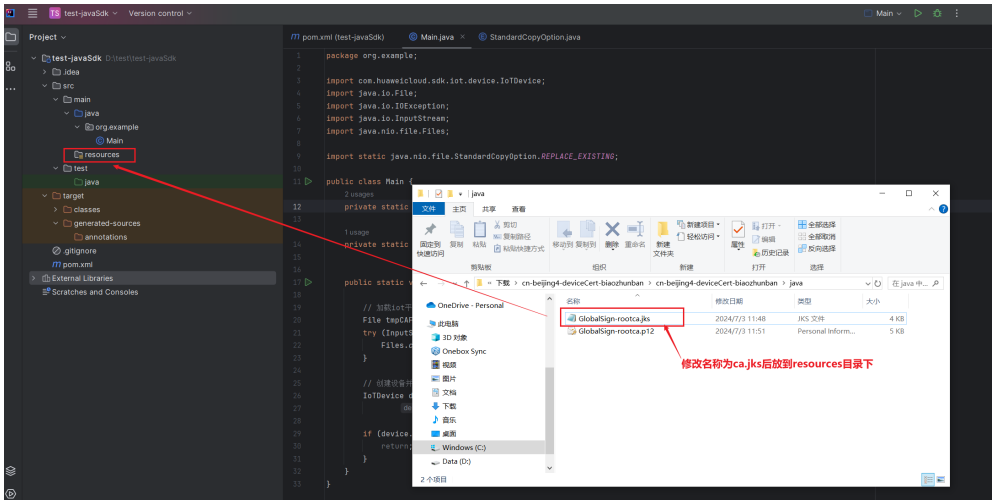
2. 添加证书文件。**获取设备侧CA证书**，请根据您的局点获取CA证书，并将证书名改为ca.jks，放到项目resources目录下。

图 3-142 获取设备侧 CA 文件

表1 证书资源

证书包名称	region&版本	证书类型	证书格式	说明	下载
certificate	北京四基础版	设备侧证书	pem、jks、bks	用于设备校验平台的身份。该证书必须配合当前设备侧接入域名使用。 注：之前的老域名(iot-acc.cn-north-4.myhuaweicloud.com)必须要配合老证书使用。	证书文件
certificate	北京四、上海一和广州标准版	设备侧证书	pem、jks、bks	用于设备校验平台的身份。该证书必须配合当前设备侧接入域名使用。	证书文件
certificate	北京四	应用侧证书	pem	用于订阅推送场景，应用侧校验平台的身份。	证书文件
certificate（设备发放）	通用	设备侧证书	pem、jks、bks	用于设备校验平台（设备发放）的身份。该证书必须配合设备发放使用。	证书文件
certificate	中国-香港、亚太-新加坡、亚太-曼谷、亚太-雅加达、非洲-约翰内斯堡、拉美-圣地亚哥、拉美-圣保罗一、拉美-墨西哥城二、中东-利雅得	设备侧证书	pem、jks、bks	用于设备校验平台的身份。该证书必须配合当前设备侧接入域名使用。	证书文件

图 3-143 添加 CA 文件到 SDK 中



3. 修改接入信息。将设备接入地址、设备ID、设备密钥，修改为**注册设备**的设备信息。

图 3-144 修改设备连接参数

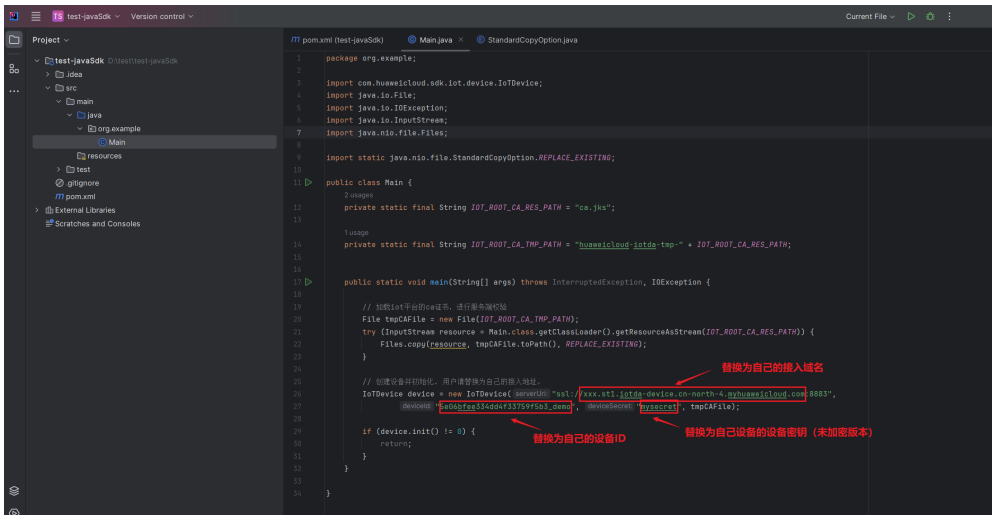
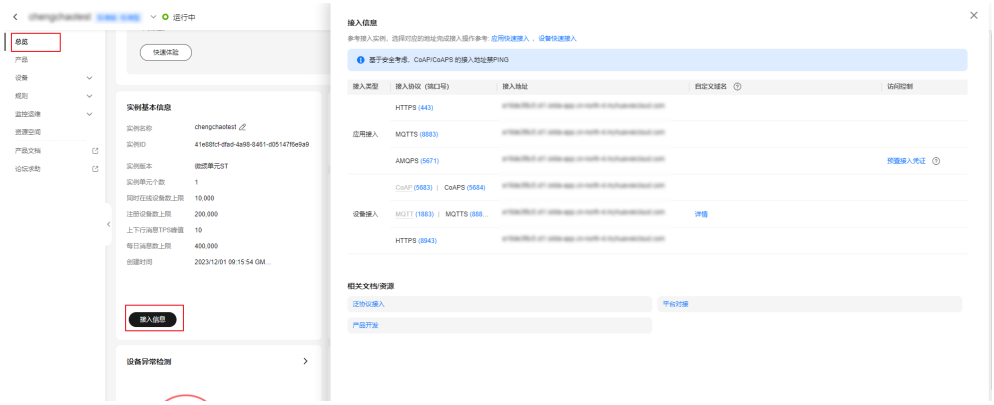


图 3-145 总览-获取接入信息



说明

- 选择设备接入的MQTTS协议接入地址，复制到代码中。若是想使用MQTT协议，可将"ssl://xxx.st1.iotda-device.cn-north-4.myhuaweicloud.com:8883"修改为："tcp://xxx.st1.iotda-device.cn-north-4.myhuaweicloud.com:1883"
- 设备ID与设备密钥在创建设备时会弹出是否保存界面，若是单击了保存，会以文件的形式保存到电脑中。

步骤4 编译及运行。单击IDEA中的运行按钮，即可在平台看到设备为在线状态。

图 3-146 IDEA 运行代码

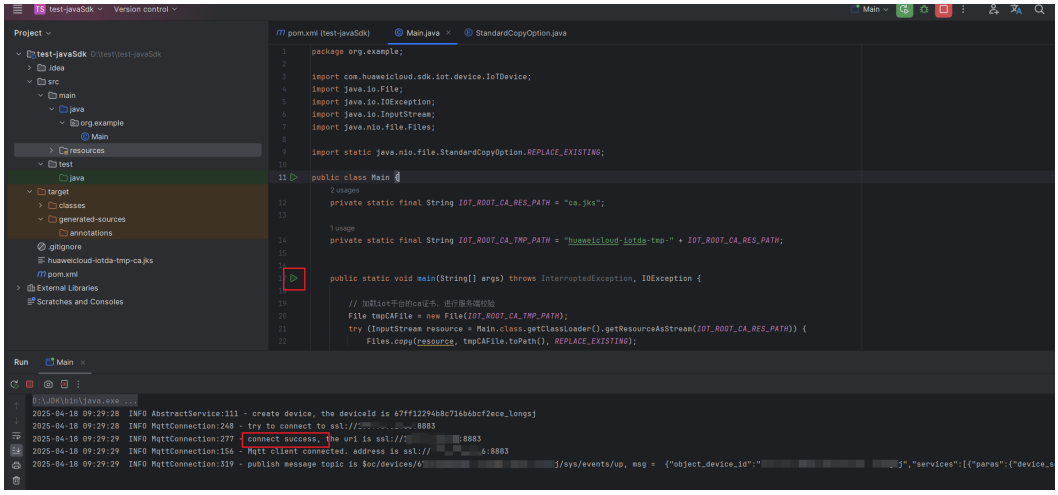


图 3-147 设备在线

----结束

说明

SDK默认具有自动重建链的功能，当设备由于网络等原因断链时，会自动重新连接。

消息上报

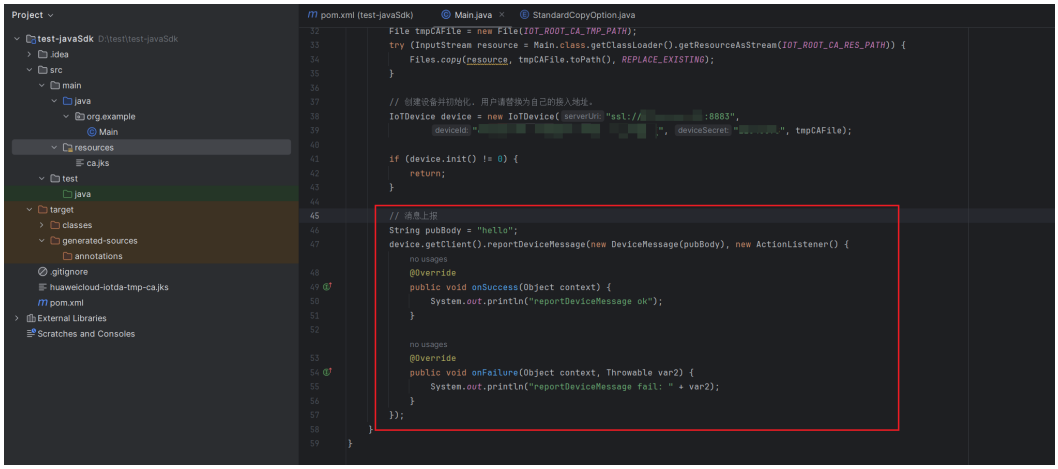
在**使用Java SDK接入华为云**的基础上，开发消息上报功能，SDK接口可见：**消息上报、下发**。

步骤1 将下列代码复制到新建项目的Main.java文件中，请复制到“if (device.init() != 0) {return;}”设备建链成功后。

```
// pubBody为要上报的消息，发送时会被编辑为标准上报数据格式
// reportDeviceMessage方法默认上报的topic为$oc/devices/{device_id}/sys/messages/up
String pubBody = "hello";
device.getClient().reportDeviceMessage(new DeviceMessage(pubBody), new ActionListener() {
    @Override
    public void onSuccess(Object context) {
        System.out.println("reportDeviceMessage ok");
    }
```

```
    @Override
    public void onFailure(Object context, Throwable var2) {
        System.out.println("reportDeviceMessage fail: " + var2);
    }
});
```

图 3-148 消息上报-IDEA



步骤2 开启设备消息跟踪，查看消息记录。在华为云管理控制台找到对应设备，进入“设备详情”，单击“消息跟踪 > 启动消息跟踪”，再次运行代码，即可看到上报消息。

图 3-149 设备列表-详情

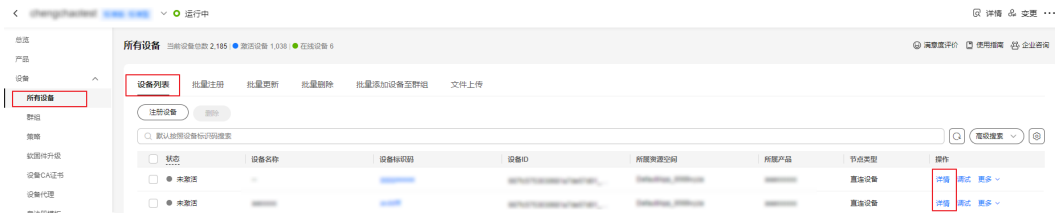
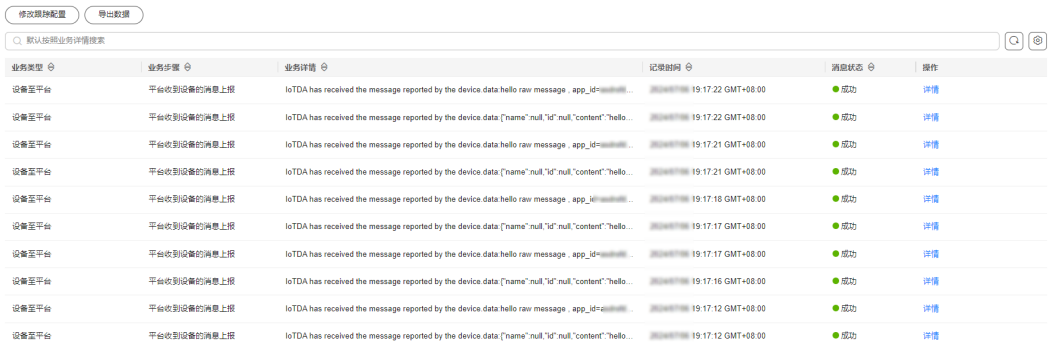


图 3-150 消息跟踪-查看 device_sdk_java 消息跟踪



----结束

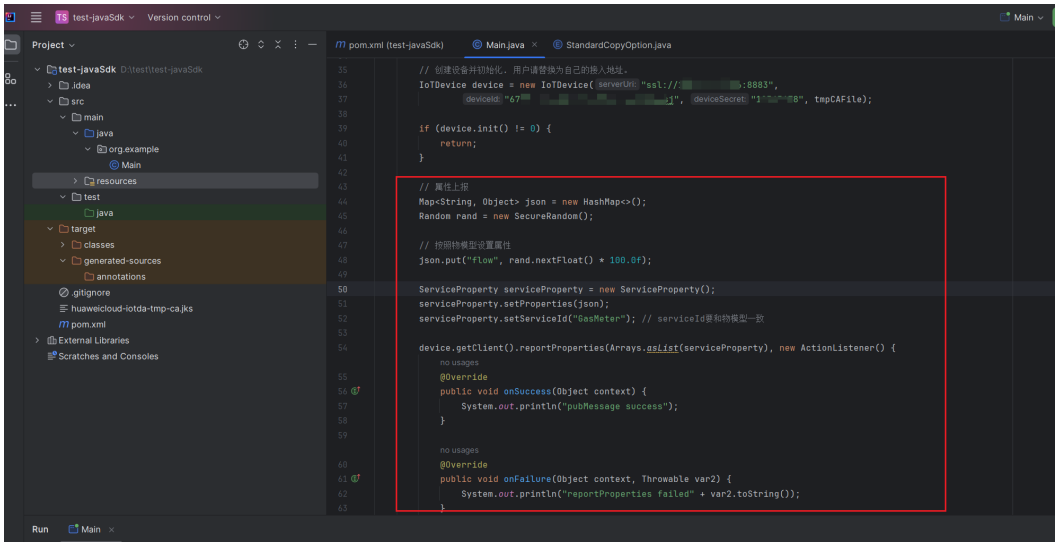
属性上报

在**使用Java SDK接入华为云**的基础上，开发属性上报功能，SDK接口可见：**属性上报**。

步骤1 将下列代码复制到Main.java中，请复制到device.init()设备建链成功后。

```
// 属性上报
Map<String, Object> json = new HashMap<>();
Random rand = new SecureRandom();
// 按照物模型设置属性
json.put("flow", rand.nextFloat() * 100.0f);
ServiceProperty serviceProperty = new ServiceProperty();
serviceProperty.setProperties(json);
serviceProperty.setServiceId("GasMeter"); // serviceId要和物模型一致

device.getClient().reportProperties(Arrays.asList(serviceProperty), new ActionListener() {
    @Override
    public void onSuccess(Object context) {
        System.out.println("pubMessage success");
    }
    @Override
    public void onFailure(Object context, Throwable var2) {
        System.out.println("reportProperties failed" + var2.toString());
    }
});
```

图 3-151 属性上报-IDEA



说明

样例代码为燃气表产品物模型的属性设置。用户可以根据自己实际场景自行进行代码适配。

步骤2 在平台查看属性上报值。在华为云控制台找到对应设备，进入“设备详情”，单击“设备信息”，再次运行代码，即可看到属性上报值。

图 3-152 设备列表-详情

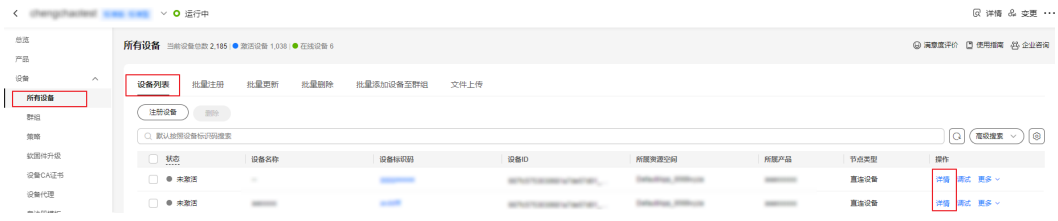
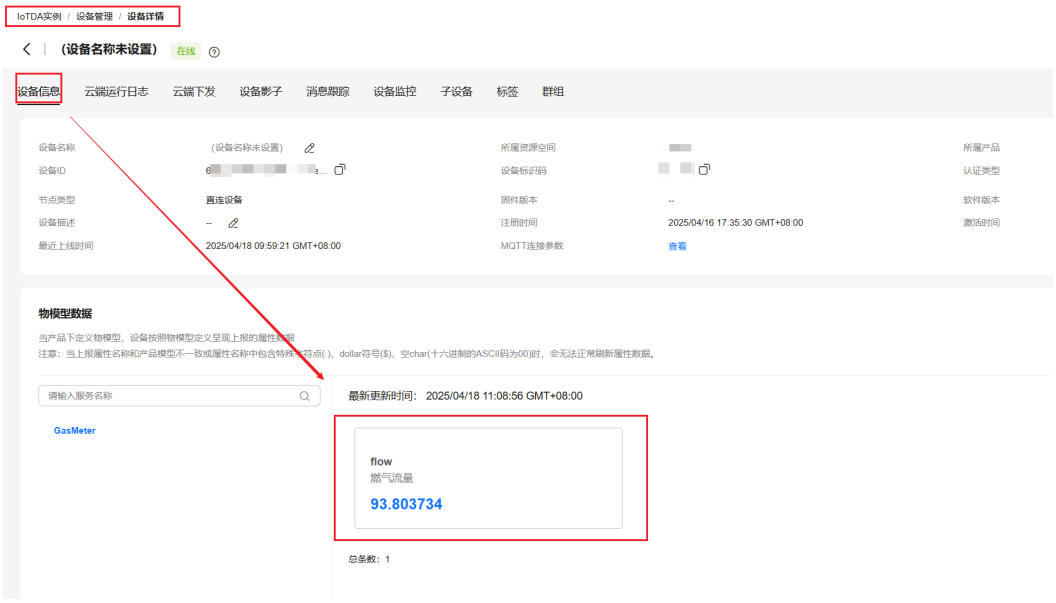


图 3-153 查看上报数据-flow



----结束

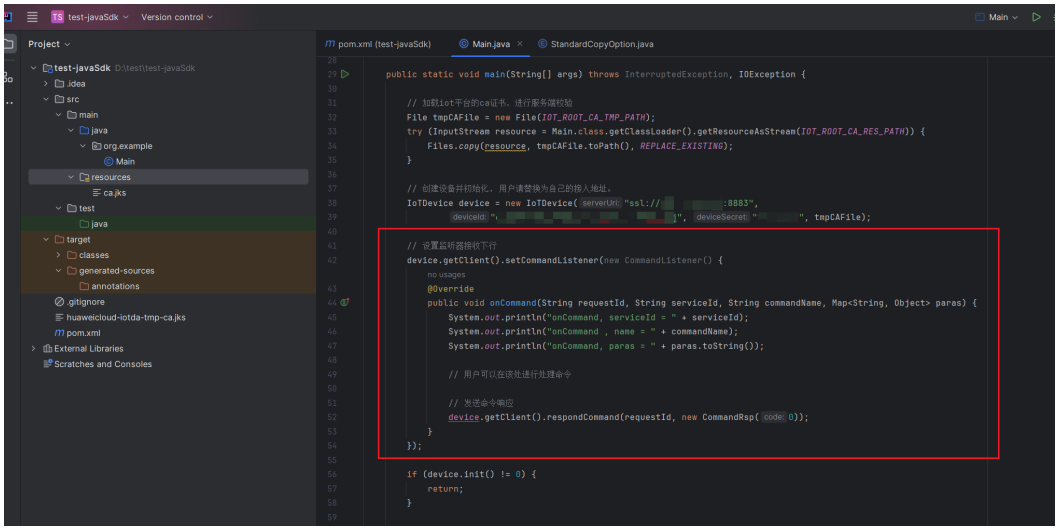
命令下发

在**使用Java SDK接入华为云**的基础上，开发下发功能，SDK接口可见：**命令下发**。

步骤1 将下列代码复制到Main.java文件中，请复制到device.init()设备建链成功前。

```
// 设置监听器接收下行
device.getClient().setCommandListener(new CommandListener() {
    @Override
    public void onCommand(String requestId, String serviceId, String commandName, Map<String, Object>
paras) {
        System.out.println("onCommand, serviceId = " + serviceId);
        System.out.println("onCommand , name = " + commandName);
        System.out.println("onCommand, paras = " + paras.toString());
        // 用户可以在该处进行处理命令

        // 发送命令响应
        device.getClient().respondCommand(requestId, new CommandRsp(0));
    }
});
```

图 3-154 命令下发-IDEA



步骤2 在平台进行命令下发。运行SDK代码，在华为云控制台找到对应设备，进入“设备详情”，单击“云端下发 > 命令下发 > 命令下发”，单击“确定”，即可在IDEA运行台中收到下发值。

图 3-155 设备列表-详情

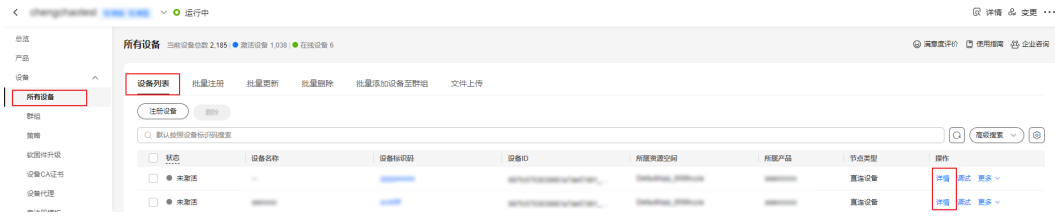


图 3-156 命令下发-TOGGLE

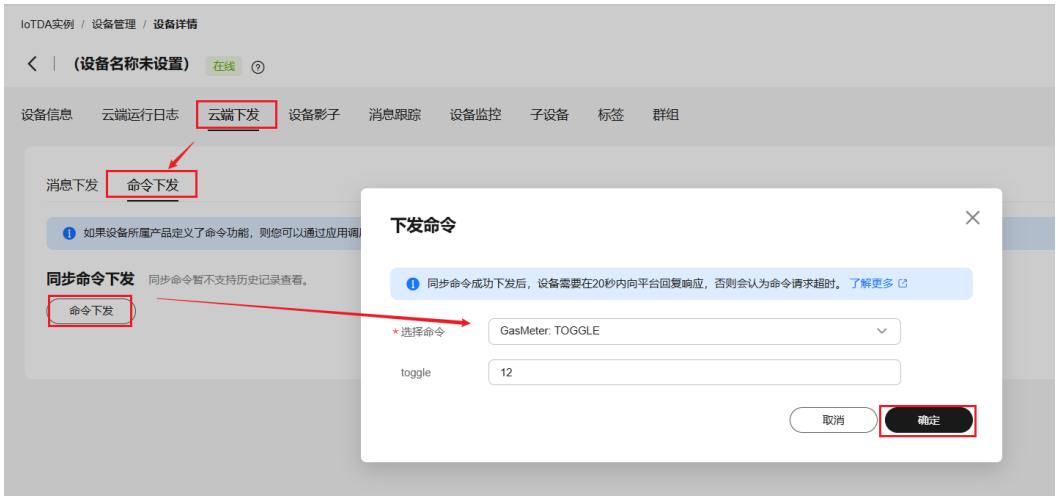
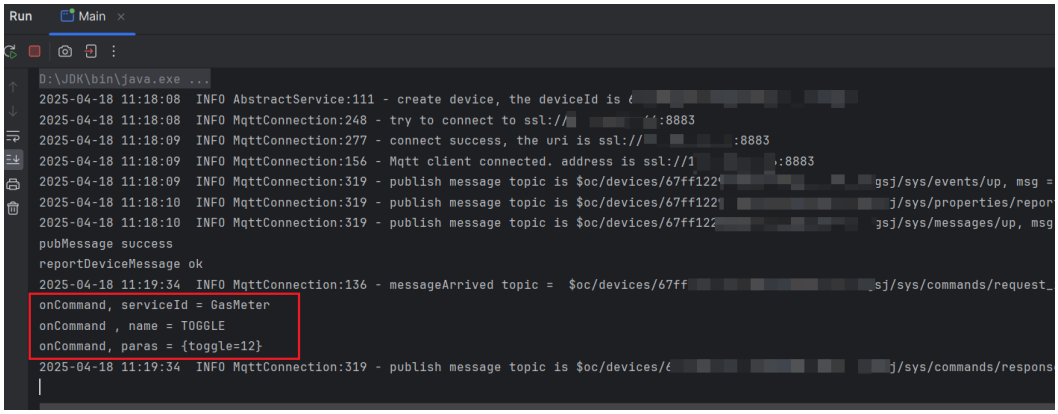


图 3-157 IDEA 命令下发结构查看



----结束

设备集成 Java SDK 并运行

华为云IoT提供的设备侧SDK可以被快速集成在用户的IoT设备中。这里以使用Linux系统的IoT设备和Java SDK为例，介绍如何将上文调试好的SDK集成进设备中，实现设备快速连接IoTDA平台。Java SDK可以通过IDEA生成jar包，传输到Linux设备中运行。

前提条件

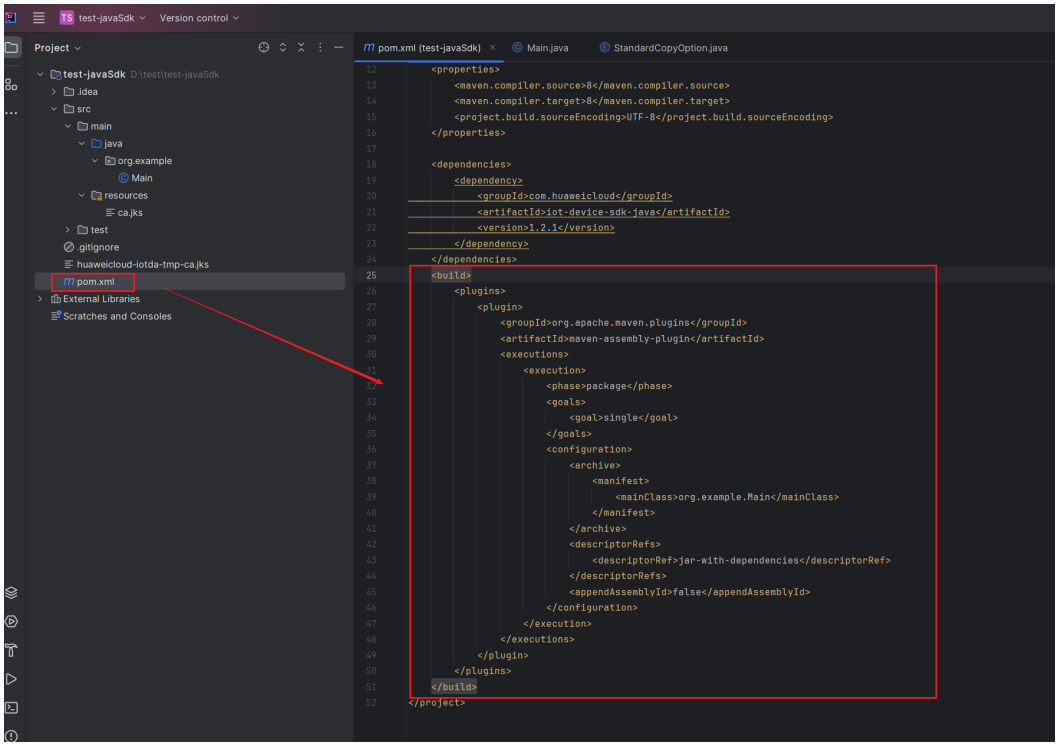
设备为Linux系统并已安装jdk。

运行步骤

步骤1 修改根目录下的pom.xml文件：复制以下配置到pom.xml文件中。

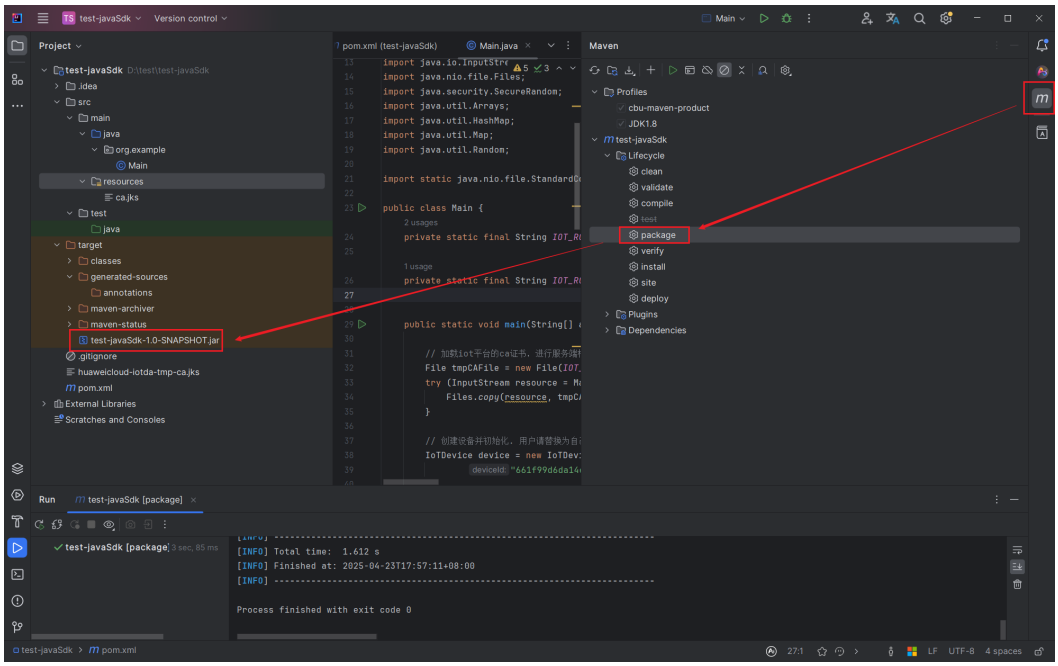
```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-assembly-plugin</artifactId>
      <executions>
        <execution>
          <phase>package</phase>
          <goals>
            <goal>single</goal>
          </goals>
          <configuration>
            <archive>
              <manifest>
                <mainClass>org.example.Main</mainClass>
              </manifest>
            </archive>
            <descriptorRefs>
              <descriptorRef>jar-with-dependencies</descriptorRef>
            </descriptorRefs>
            <appendAssemblyId>false</appendAssemblyId>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

图 3-158 IDEA-添加 package 参数



步骤2 在IDEA中打开Maven，单击package，即可在目录下生成jar包。

图 3-159 IDEA-生成 jar 包



步骤3 把jar包复制到Linux设备中，输入“java -jar test-javaSdk-1.0-SNAPSHOT”，即可看到代码运行成功，设备成功连接到IoTDA平台。

图 3-160 在 Linux 中运行 jar 包

说明

若是显示没有java命令，可输入“java -version”查看是否安装了jdk及jdk安装版本。

----结束

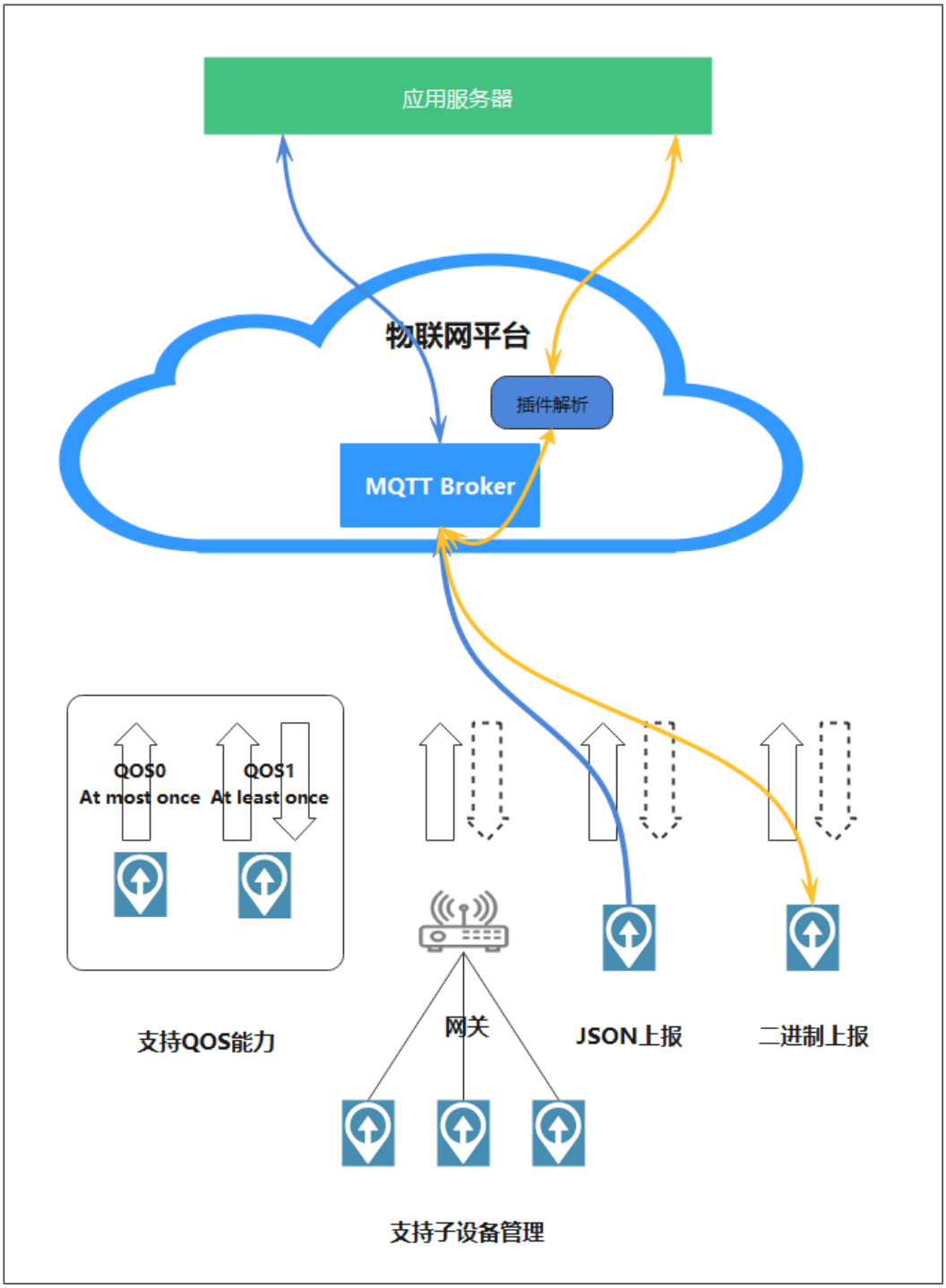
3.5 MQTT(S)协议接入

3.5.1 协议介绍

概述

MQTT（Message Queuing Telemetry Transport）是一个基于客户端-服务器的消息发布/订阅传输协议，主要应用于计算能力有限且工作在低带宽、不可靠的网络的远程传感器和控制设备，适合设备与云端保持长连接的场景，如智能路灯等。MQTT 客户端通过主题（Topic）发布或订阅消息，由MQTT Broker集中管理消息路由并根据预设的服务质量 (QoS) 确保端到端消息传递的可靠性。在此过程中，发送消息的客户端（发布者）和接收消息的客户端（订阅者）之间被解耦，发布者和订阅者之间无需建立直接连接。由于MQTT协议满足物联网对通信协议的轻量级、可靠、双向和大规模的需求，MQTT协议成为了物联网领域的最佳协议之一。更多关于MQTT协议语法及接口信息，请访问这里获取。

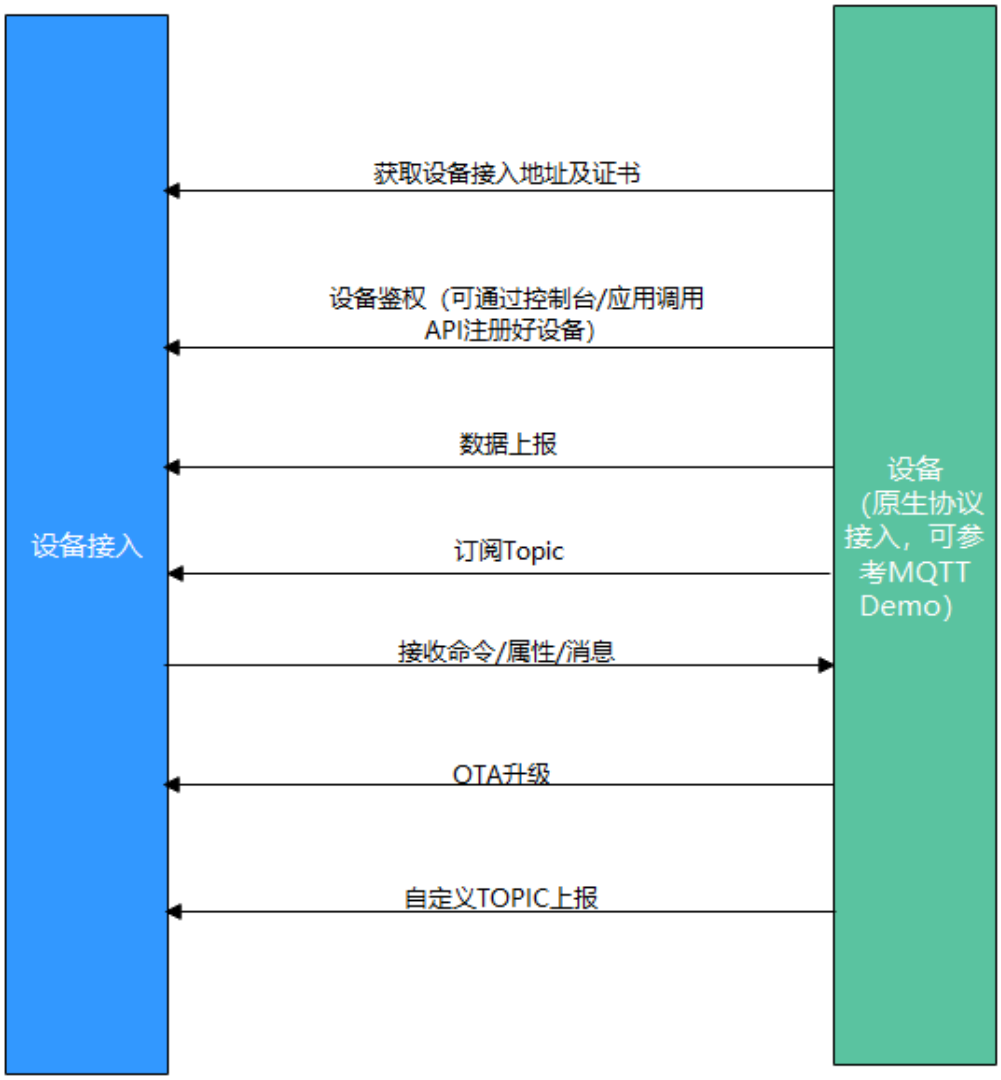
MQTTS是MQTT使用TLS加密的协议。采用MQTTS协议接入平台的设备，设备与物联网平台之间的通信过程，数据都是加密的，具有一定的安全性。



业务流程

采用MQTT协议接入物联网平台的设备，设备与物联网平台之间的通信过程，数据没有加密，建议使用MQTTS协议。

若选择MQTTS协议接入平台，建议通过**使用IoT Device SDK接入**。



1. 设备接入前，需创建产品（可通过控制台创建或者使用应用侧API**创建产品**）。
2. 产品创建完毕后，需注册设备（可通过控制台**注册单个设备**或者使用应用侧API**注册设备**创建）。
3. 设备注册完毕后，可以按照图中流程实现消息/属性上报、接收命令/属性/消息、OTA升级、自定义Topic等功能。关于平台预置Topic可参考**Topic定义**

说明

您可以通过mqtt.fx进行原生协议接入调测，可以参考**快速体验mqtt接入**。

使用限制

描述	限制
单个MQTT直连设备在同一时间的连接数	1
单账户设备侧每秒最大建链请求数量	• 基础版100 • 标准版请参考**标准版规格**

描述	限制
单账号设备侧每秒最大上行的请求数量（单消息payload平均为512字节）	• 基础版500 • 标准版请参考**标准版规格**
单个MQTT连接每秒最大上行消息数量	50/s
单个MQTT连接最大带宽（上行消息）	1MB（默认）
MQTT单条发布消息最大长度。超过此大小的发布请求将被直接拒绝。	1MB
MQTT协议规范	MQTT v5.0、MQTT v3.1.1、MQTT v3.1
与标准MQTT协议的区别	• 不支持QoS2 • 不支持will、retain msg
MQTT协议支持的安全等级	采用TCP通道基础 + TLS协议（TLSv1、 TLSv1.1、TLSv1.2和TLSv1.3版本）
MQTT连接心跳时间建议值	心跳时间限定为30至1200秒，推荐设置为120秒
MQTT协议消息发布与订阅	设备只能对自己的Topic进行消息发布与订阅
单个MQTT连接的最大订阅数量。	100个
MQTT自定义Topic支持的最大长度	128字节
MQTT自定义Topic支持每个产品添加的最大个数	10个/产品
单账号支持上传设备侧CA证书个数	100个

MQTT 设备与物联网平台通信

设备使用MQTT协议接入平台时，平台和设备通过Topic进行通信。物联网平台预置了Topic，通过这些预置的Topic，平台和设备可以实现消息、属性、命令的交互。您还可以在设备接入控制台，自定义Topic，实现设备平台通信的个性化配置。

数据类型	消息类型	说明
数据上行	设备属性上报	用于设备按产品模型中定义的格式将属性数据上报给平台。
	设备消息上报	设备无法按照产品模型中定义的属性格式进行数据上报时，将设备的自定义数据通过设备消息上报接口上报给平台，平台将设备上报的消息转发给应用服务器或华为云其他云服务上进行存储和处理。

数据类型	消息类型	说明
	网关批量属性上报	用于网关设备将多个设备的属性数据一次性上报给平台。
	设备事件上报	用于设备按产品模型中定义的格式将事件数据上报给平台。
数据下行	平台消息下发	用于平台下发自定义格式的数据给设备。
	平台设置设备属性	设备的产品模型中定义了平台可向设备设置的属性，平台/应用服务器可通过属性设置的方式修改指定设备的属性值。
	平台查询设备属性	平台/应用服务器通过属性查询的方式，实时查询指定设备的属性数据。
	平台命令下发	平台/应用服务器按产品模型中定义的命令格式下发控制命令给设备。
	平台事件下发	平台/应用服务器按产品模型中定义的事件格式下发事件给设备。

Topic接口介绍

物联网平台预置的Topic如下表所示：

Topic分类	用途	Topic	Publisher（发布者）	Subscriber（订阅者）
设备消息相关Topic	设备消息上报	$oc/devices/{device_id}/sys/messages/up	设备	平台
	平台下发消息给设备	$oc/devices/{device_id}/sys/messages/down	平台	设备
设备命令相关Topic	平台下发命令给设备	$oc/devices/{device_id}/sys/commands/request_id={request_id}	平台	设备
	设备返回命令响应	$oc/devices/{device_id}/sys/commands/response/request_id={request_id}	设备	平台
设备属性相关Topic	设备上报属性数据	$oc/devices/{device_id}/sys/properties/report	设备	平台
	网关批量上报属性数据	$oc/devices/{device_id}/sys/gateway/sub_devices/properties/report	设备	平台

Topic分类	用途	Topic	Publisher（发布者）	Subscriber（订阅者）
	平台设置设备属性	$oc/devices/{device_id}/sys/properties/set/request_id={request_id}	平台	设备
	属性设置的响应结果	$oc/devices/{device_id}/sys/properties/set/response/request_id={request_id}	设备	平台
	平台查询设备属性	$oc/devices/{device_id}/sys/properties/get/request_id={request_id}	平台	设备
	属性查询响应结果，这个结果不会对设备属性和影子产生影响	$oc/devices/{device_id}/sys/properties/get/response/request_id={request_id}	设备	平台
	设备侧主动获取平台的设备影子数据	$oc/devices/{device_id}/sys/shadow/get/request_id={request_id}	设备	平台
	设备侧主动获取平台设备影子数据的响应	$oc/devices/{device_id}/sys/shadow/get/response/request_id={request_id}	平台	设备
设备事件相关Topic	设备事件上报	$oc/devices/{device_id}/sys/events/up	设备	平台
	平台事件下发	$oc/devices/{device_id}/sys/events/down	平台	设备

另外，用户还可以通过在控制台上设置自定义Topic，上报用户个性化的数据。具体可参考自定义Topic。

MQTT 的 TLS 支持

平台推荐使用TLS来保护设备和平台的传输安全。 平台目前支持TLS1.3、1.2 、1.1版本以及GMTLS。其中TLS 1.1 计划后续不再支持，建议使用 TLS 1.3作为首选 TLS 版本。GMTLS版本仅在国密算法的企业版才支持。

基础版，标准版以及支持通用加密算法的企业版使用TLS连接时，平台支持如下加密套件：

- TLS_AES_256_GCM_SHA384
- TLS_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256

- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA

支持国密算法的企业版使用TLS连接时平台支持如下加密套件：

- ECC_SM4_GCM_SM3
- ECC_SM4_CBC_SM3
- ECDHE_SM4_GCM_SM3
- ECDHE_SM4_CBC_SM3
- TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256

说明

带CBC的加密套件存在安全风险，请谨慎使用。

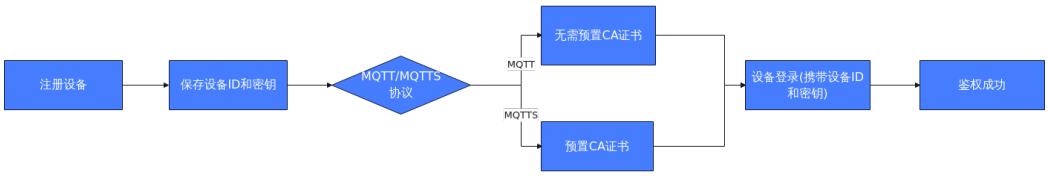
3.5.2 密钥鉴权

概述

MQTT(S)协议-密钥鉴权是指设备在接入物联网平台时，携带设备ID和密钥以完成设备的接入鉴权。对于使用MQTTS协议接入的设备，需要在设备侧预置CA证书；对于使用MQTT非安全协议接入的设备，无需在设备侧预置CA证书。

使用 MQTT(S)协议-密钥接入的鉴权流程

图 3-161 MQTT(S)协议-密钥接入鉴权流程图



1. 通过调用注册接口向物联网平台发送注册请求，或者在控制台上注册设备。

   说明

   注册时需要填写设备标识码，通常使用MAC地址，Serial No或IMEI作为nodeId。

2. 物联网平台向设备分配全局唯一的设备ID （ deviceId ）和密钥（ secret ）。

   说明

   密钥可以在注册设备时自定义，如果没有定义，平台将自动分配密钥。

3. 设备侧需集成预置CA证书**获取CA证书**（仅针对MQTTS协议接入的鉴权流程）。
4. 设备登录时，携带设备ID（ deviceId ）和密钥（ secret ）发起接入鉴权请求。
5. 平台验证通过后，返回成功响应，设备连接物联网平台成功。

密钥鉴权使用说明

本章说明以使用MQTT.fx工具激活在物联网平台上注册的设备为例，说明具体操作。

步骤1 下载MQTT.fx（默认是64位操作系统，如果是32位操作系统，单击此处下载MQTT.fx），安装MQTT.fx工具。

步骤2 进入设备信息页面，单击MQTT连接参数，查看设备的连接信息（ClientId、Username、Password）。

图 3-162 设备-连接参数

您也可以访问参数生成工具，填写注册设备后生成的设备ID（device_id）和密钥（secret），生成设备连接鉴权所需的参数（ClientId、Username、Password）。

表 3-27 参数说明

参数	是否必选	类型	描述
ClientId	是	String	**参数解释：** 一机一密的设备ClientId由4个部分组成：设备ID、设备身份标识类型、密码签名类型、时间戳，通过下划线“_”分隔。 • **设备ID**：指设备在平台成功注册后生成的唯一设备标识，通常由设备的产品ID和设备的NodeId通过分隔符“_”拼装而来。 • **设备身份标识类型**：固定值为0，表示设备ID。 • **密码签名类型**：长度1字节，当前支持2种类型： – “0”代表HMACSHA256不校验时间戳。 – “1”代表HMACSHA256校验时间戳。 • **时间戳**：为设备连接平台时的UTC时间，格式为YYYYMMDDHH，如UTC 时间2018/7/24 17:56:20则应表示为2018072417。 **取值范围：** 长度不超过256。
UserName	是	String	**参数解释：** UserName在此处即为设备ID（device_id）。 **取值范围：** 长度不超过256。

参数	是否必选	类型	描述
Password	是	String	**参数解释：** Password是使用了“HMACSHA256”算法，以secret为内容，时间戳（格式为：YYYYMMDDHH）为密钥，进行加密后获取的值（secret为注册设备时平台返回的secret）。Password = hmacsha256("secret", "时间戳")。 当设备认证类型使用密钥认证接入（SECRET）需填写“Password”，X.509证书认证接入（CERTIFICATES）不需填写“Password”。 “HMACSHA256”即使用SHA-256生成哈希值的HMAC算法，生成的哈希值通用一个长度为64位的十六进制字符串来表示。例如，设备密码为：12345678；时间戳为：2025041401；那么，HMACSHA256后的值为：“c75150e6cb841417396819e4d2ee4358a416344a03a083e3a8567074ddec820a”。 **取值范围：** 长度不超过256。

设备通过MQTT协议的connect消息进行鉴权，对于构造clientId的各个部分信息都必须包括进去，平台收到connect消息时，会判断设备的鉴权类型和密码摘要算法。

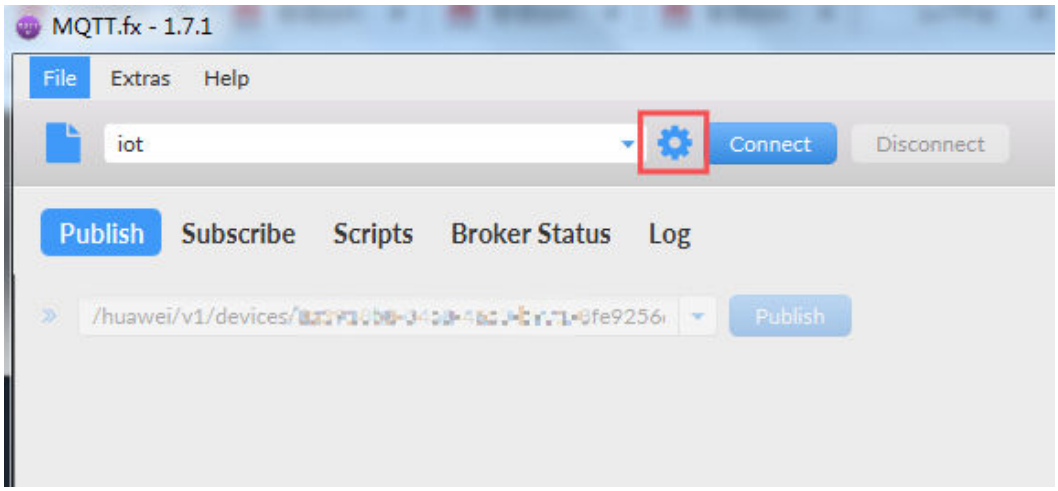
使用生成工具生成的clientId格式，默认不校验时间戳：设备ID_0_0_时间戳。

- 当采用“HMACSHA256”校验时间戳方式时，会先校验消息时间戳与平台时间是否一致，再判断密码是否正确。
- 当采用“HMACSHA256”不校验时间戳方式时，鉴权消息也必须带时间戳，但不检验时间是否准确，仅判断密码是否正确。

connect消息鉴权失败时，平台会返回错误，并自动断开MQTT链路。

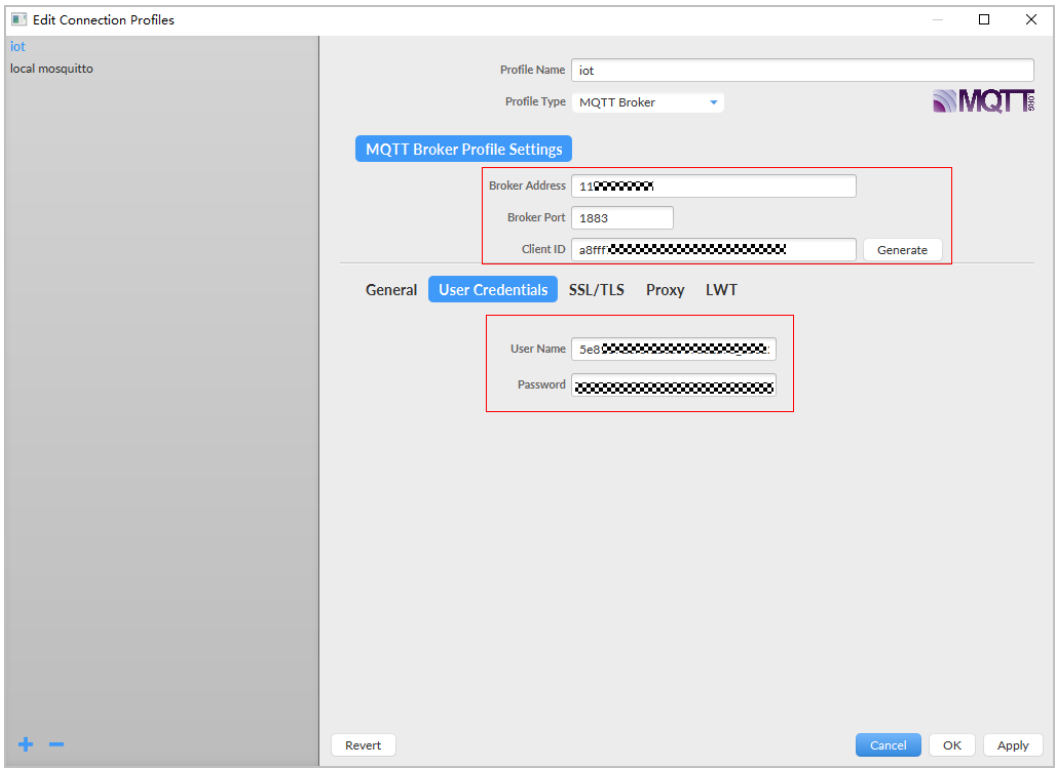
步骤3 打开MQTT.fx软件，单击设置图标。

图 3-163 MQTT.fx 软件-设置



步骤4 参考下表配置鉴权参数，然后单击“Apply”。

图 3-164 连接信息配置



参数名称	说明
Broker Address	填写从设备接入服务控制台获取的**平台对接信息**，此接入地址为域名信息。不能通过域名接入的设备，通过在cmd命令框中执行“ping 域名”获取IP地址，用IP地址接入平台。由于IP地址不固定，您需要将IP地址做成可配置项。
Broker Port	默认为1883，使用MQTT非加密协议。若用户想使用MQTTS加密协议，则需改为8883，并参考**证书资源**获取校验平台身份的证书。具体配置操作说明可参考**MQTT.fx模拟智慧路灯与平台通信**。
Client ID	设备clientID，请参考**2**中获取。
User Name	即设备ID，请参考**2**中获取。
Password	加密后的设备密钥，请参考**2**中获取。

步骤5 单击“Connect”，设备鉴权成功后，在物联网平台可以看到设备处于在线状态。

图 3-165 设备在线

----结束

相关最佳实践

在线开发MQTT协议的模拟智慧路灯

3.5.3 证书鉴权

3.5.3.1 证书鉴权使用说明

概述

MQTT(S)协议-证书鉴权是指在设备接入物联网平台前，用户通过控制台上传设备CA证书，然后应用服务调用**创建设备**接口或通过控制台在物联网平台注册设备，获取设备ID。在设备接入物联网平台时携带设备侧X.509证书（一种用于通信实体鉴别的数字证书），完成设备的接入鉴权。

约束与限制

- 当前物联网平台只支持基于MQTT(S)协议接入的设备使用X.509证书进行设备身份认证。
- 每个用户最多上传100个设备CA证书；可多个设备共用一个CA证书。

使用 MQTT(S)协议-证书接入的鉴权流程

图 3-166 MQTT(S)协议-证书接入鉴权流程图

1. 在控制台上传设备CA证书。
2. 通过调用注册接口向物联网平台发送注册请求或者在控制台上注册设备。

   说明

   注册时需要填写设备标识码，通常使用MAC地址，Serial No或IMEI作为nodeId。

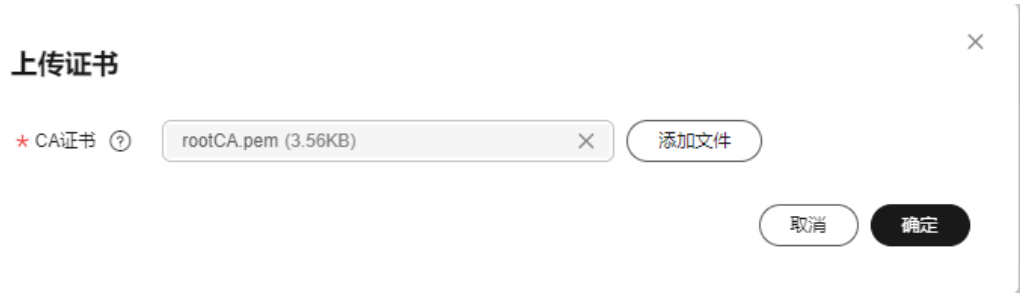
   物联网平台向设备分配全局唯一的设备ID（deviceId）。
3. 设备登录时，携带设备侧**X.509证书**发起接入鉴权请求。
4. 平台验证通过后，返回成功响应，设备连接物联网平台成功。

上传设备 CA 证书

步骤1 在左侧导航栏选择“设备 > 设备证书”，进入“设备CA证书”页签，选择资源空间，单击“上传证书”。

步骤2 在弹出的对话框中，单击“添加文件”，然后单击“确定”。

图 3-167 设备 CA 证书-上传证书



说明

- 设备CA证书由设备厂商提供，调测时可**自行制作调测证书**，商用时建议更换为商用证书，否则会带来安全风险。
- CA 证书具有一个过期日期，在该日期后，这些证书将无法用于验证服务器的证书；请在 CA 证书的过期日期前替换这些证书，以确保设备可以正常的连接到IoT平台。

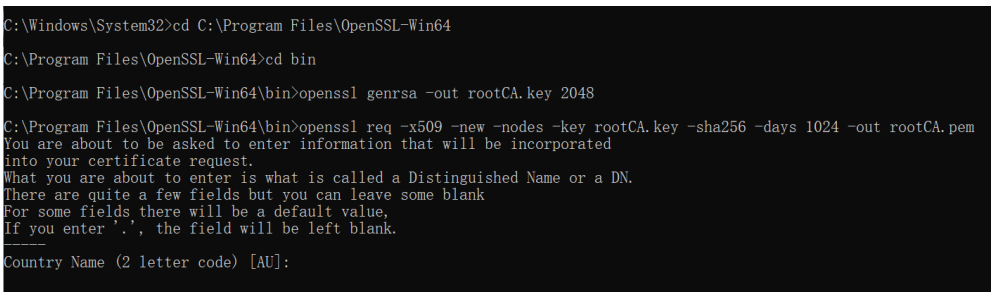
----结束

设备 CA 调测证书制作流程

本文以Windows环境为例，介绍通过Openssl工具制作调测证书的方法，生成的证书为PEM编码格式的证书。

1. 在浏览器中访问**这里**，下载并进行安装OpenSSL工具。
2. 以管理员身份运行cmd命令行窗口。
3. 执行cd c:\openssl\bin（请替换为openssl实际安装路径），进入openssl命令视图。
4. 执行以下命令生成密钥对。
   ```
   openssl genrsa -out rootCA.key 2048
   ```
5. 执行以下命令，使用密钥对中的私有密钥生成 CA 证书。
   ```
   openssl req -x509 -new -nodes -key rootCA.key -sha256 -days 1024 -out rootCA.pem
   ```

图 3-168 生成 CA 证书



系统提示您输入如下信息，所有参数可以自定义。

- Country Name (2 letter code) [AU]：国家，如CN。
- State or Province Name (full name) []: 省份，如GD。
- Locality Name (for example, city) []:城市，如SZ。
- Organization Name (for example, company) []：组织，如Huawei。
- Organizational Unit Name (for example, section) []：组织单位，如IoT。
- Common Name (e.g. server FQDN or YOUR name) []: 名称，如zhangsan。
- Email Address []:邮箱地址，如1234567@163.com。

在openssl安装目录的bin文件夹下，获取生成的CA证书（rootCA.pem）。

上传验证证书

如果上传的是调测证书，上传后证书状态显示为“未验证”，您需要上传验证证书，来证明您拥有该CA证书。

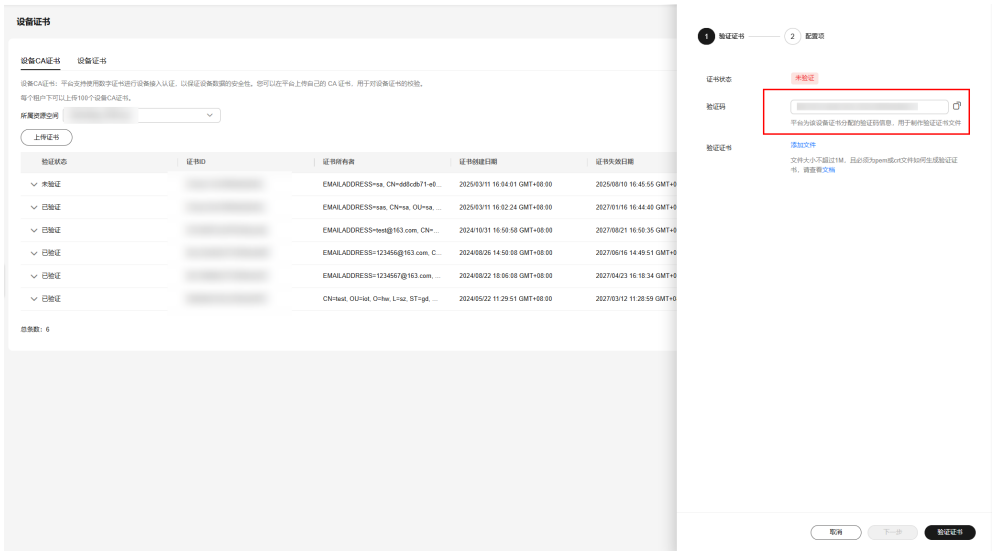
图 3-169 设备 CA 证书-未验证证书

验证证书是由设备CA证书对应的私钥创建的，请参考如下操作制作验证证书。

步骤1 获取验证证书的验证码。

图 3-170 设备 CA 证书-验证证书

图 3-171 设备 CA 证书-获取验证码



步骤2 执行如下命令为私有密钥验证证书生成密钥对。

```
openssl genrsa -out verificationCert.key 2048
```

步骤3 执行如下命令为私有密钥验证证书创建CSR（Certificate Signing Request）。

```
openssl req -new -key verificationCert.key -out verificationCert.csr
```

系统提示您输入如下信息，**Common Name填写为验证证书的验证码**，其他参数自定义。

- Country Name (2 letter code) [AU]：国家，如CN。
- State or Province Name (full name) []: 省份，如GD。
- Locality Name (for example, city) []:城市，如SZ。
- Organization Name (for example, company) []：组织，如Huawei。
- Organizational Unit Name (for example, section) []：组织单位，如IoT。
- Common Name (e.g. server FQDN or YOUR name) []：验证证书的验证码，请参考**步骤1**获取 。
- Email Address []:邮箱地址，如1234567@163.com。
- Password[]：密码。
- Optional Company Name[]：公司名称，如Huawei。

步骤4 执行以下命令使用CSR创建私有密钥验证证书。

```
openssl x509 -req -in verificationCert.csr -CA rootCA.pem -CAkey rootCA.key -CAcreateserial -out verificationCert.pem -days 500 -sha256
```

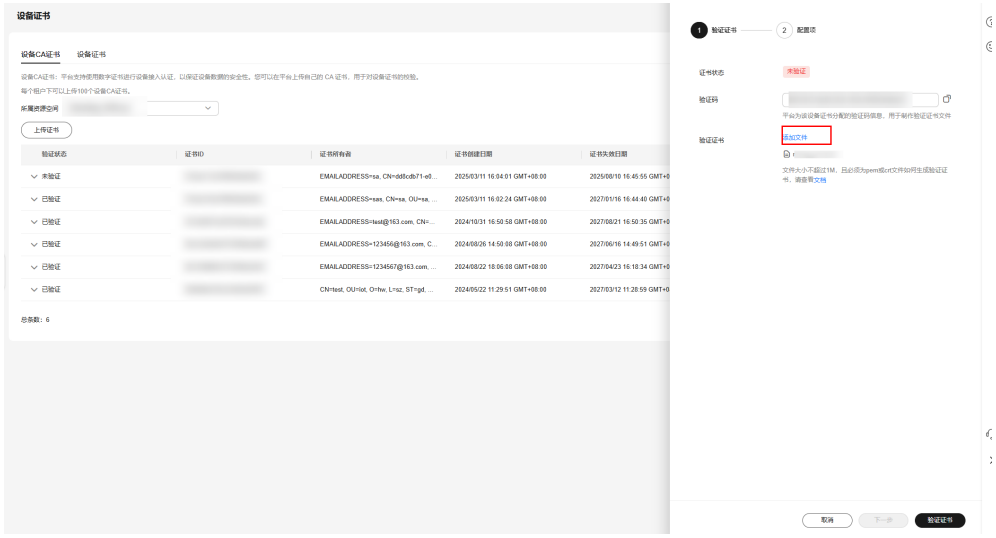
在openssl安装目录的bin文件夹下，获取生成的验证证书（verificationCert.pem）。

步骤5 选择对应证书，单击 ˅ 然后单击“上传验证证书”。

图 3-172 设备 CA 证书-验证证书

步骤6 在弹出的对话框中，单击“添加文件”，然后单击“确定”。

图 3-173 设备 CA 证书-上传验证证书



上传验证证书后，证书状态变为“已验证”，表明您拥有该CA证书。

----结束

预置 X.509 证书

在注册X.509设备之前，您需要在设备侧预置CA机构签发的X.509证书。

说明

X.509证书由CA机构签发，若没有CA机构签发的商用证书，您可以自己**制作设备CA调测证书**。

制作X.509调测证书

1. 以管理员身份运行cmd命令行窗口，执行cd c:\openssl\bin（请替换为openssl实际安装路径），进入openssl命令视图。
2. 执行如下命令生成密钥对。
   ```
   openssl genrsa -out deviceCert.key 2048
   ```
3. 执行如下命令为设备证书创建CSR（Certificate Signing Request）。
   ```
   openssl req -new -key deviceCert.key -out deviceCert.csr
   ```
   系统提示您输入如下信息，所有参数可以自定义。
   - Country Name (2 letter code) [AU]：国家，如CN。

- State or Province Name (full name) []: 省份，如GD。
- Locality Name (for example, city) []:城市，如SZ。
- Organization Name (for example, company) []：组织，如Huawei。
- Organizational Unit Name (for example, section) []：组织单位，如IoT。
- Common Name (e.g. server FQDN or YOUR name) []: 名称，如zhangsan。
- Email Address []:邮箱地址，如1234567@163.com。
- Password[]：密码。
- Optional Company Name[]：公司名称，如Huawei。

4. 执行以下命令使用CSR创建设备证书。

```
openssl x509 -req -in deviceCert.csr -CA rootCA.pem -CAkey rootCA.key -CAcreateserial -out deviceCert.pem -days 500 -sha256
```

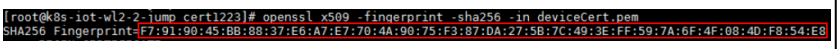
在openssl安装目录的bin文件夹下，获取生成的设备证书（deviceCert.pem）。

注册 X.509 证书认证的设备

步骤1 访问**设备接入服务**，单击**管理控制台**进入设备接入控制台。

步骤2 在左侧导航栏选择“设备 > 所有设备”，单击“注册设备”，按照如下表格填写参数后，单击“确定”。

图 3-174 设备-注册 X.509 设备

参数名称	说明
所属资源空间	选择设备所属的资源空间。
所属产品	选择设备所属的产品。
设备标识码	即node_id，填写为设备的IMEI、MAC地址或Serial No；若没有真实设备，填写自定义字符串，由英文字母和数字组成
设备名称	即device_name，可自定义。
设备认证类型	X.509证书：设备使用X.509证书验证身份。
指纹	当“设备认证类型”选择“X.509证书”时填写，导入**设备侧预置的设备证书**对应的指纹，在OpenSSL执行**openssl x509 -fingerprint -sha256 -in deviceCert.pem**命令可查询。**注：填写时需要删除冒号**。 **图 3-175** OpenSSL 执行样例 

----结束

连接鉴权

参考**连接鉴权**接口文档，使用MQTT.fx工具激活在物联网平台上注册的设备。

步骤1 下载**MQTT.fx**（默认是64位操作系统，如果是32位操作系统，单击此处下载**MQTT.fx**），安装MQTT.fx工具。

说明

- 安装最新版MQTT.fx工具，可单击此处**下载**。
- MQTT.fx 1.7.0及旧版本对带有$的主题（Topic）处理存在问题，请使用最新版本进行测试。

步骤2 进入设备信息页面，单击MQTT连接参数，查看设备的连接信息（ClientId、Username、Password）。

图 3-176 设备-连接参数

参数	必选/可选	类型	参数描述
ClientId	必选	String(256)	一机一密的设备clientId由4个部分组成：设备ID、设备身份标识类型、密码签名类型、时间戳，通过下划线“_”分隔。 • **设备ID**：指设备在平台成功注册后生成的唯一设备标识，通常由设备的产品ID和设备的NodeId通过分隔符“_”拼装而来。 • **设备身份标识类型**：固定值为0，表示设备ID。 • **密码签名类型**：长度1字节，当前支持2种类型： – “0”代表HMACSHA256不校验时间戳。 – “1”代表HMACSHA256校验时间戳。 • **时间戳**：为设备连接平台时的UTC时间，格式为YYYYMMDDHH，如UTC 时间2018/7/24 17:56:20 则应表示为2018072417。
Username	必选	String(256)	设备ID。

设备通过MQTT协议的connect消息进行鉴权，对于构造clientId的各个部分信息都必须包括进去，平台收到connect消息时，会判断设备的鉴权类型和密码摘要算法。

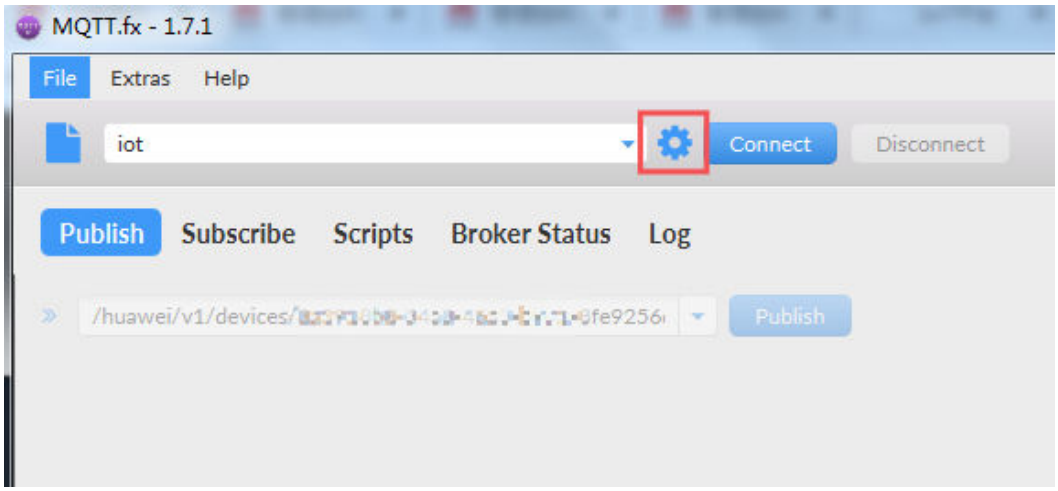
使用生成工具生成的clientId格式，默认不校验时间戳：设备ID_0_0_时间戳。

- 当采用“HMACSHA256”校验时间戳方式时，会先校验消息时间戳与平台时间是否一致，再判断密码是否正确。
- 当采用“HMACSHA256”不校验时间戳方式时，鉴权消息也必须带时间戳，但不检验时间是否准确，仅判断密码是否正确。

connect消息鉴权失败时，平台会返回错误，并自动断开MQTT链路。

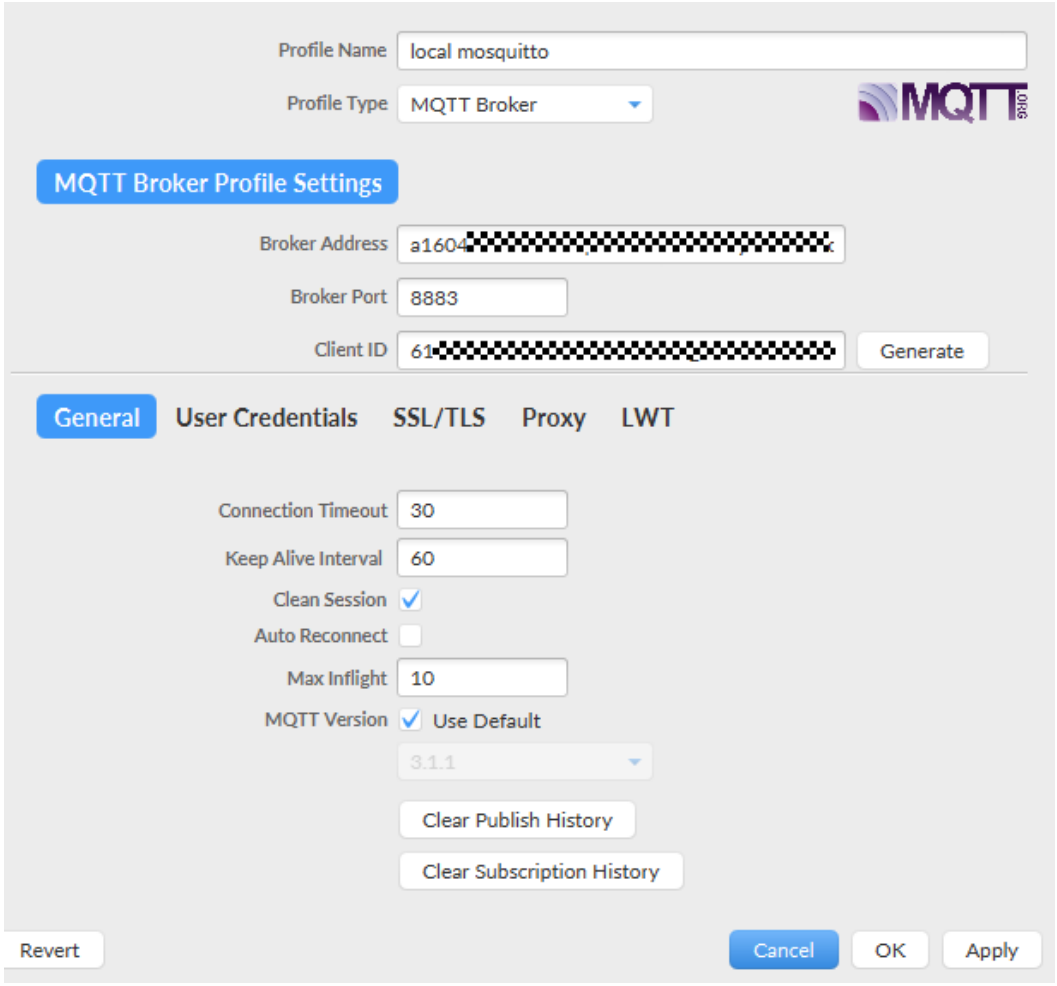
步骤3 打开MQTT.fx软件，单击“设置”图标。

图 3-177 MQTT.fx 软件-设置



步骤4 填写“Connection Profile” 相关信息。

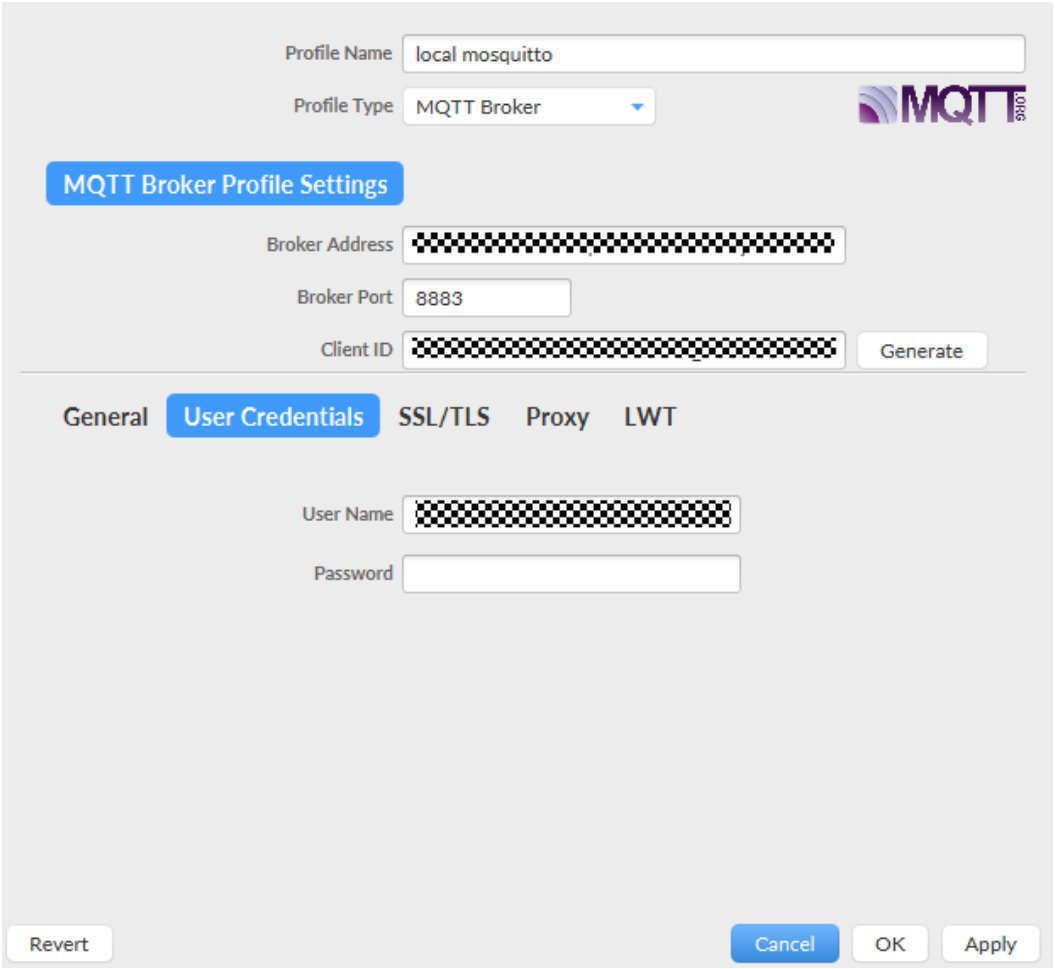
图 3-178 General-可以使用工具默认信息



参数名称	说明
Broker Address	填写从设备接入服务控制台获取的**平台对接信息**，此接入地址为域名信息。不能通过域名接入的设备，通过在cmd命令框中执行“ping 域名”获取IP地址，用IP地址接入平台。由于IP地址不固定，您需要将IP地址做成可配置项。
Broker Port	为8883。
Client ID	设备clientID，请参考**2**中获取。

步骤5 单击“User Credentials”填写“User Name”。

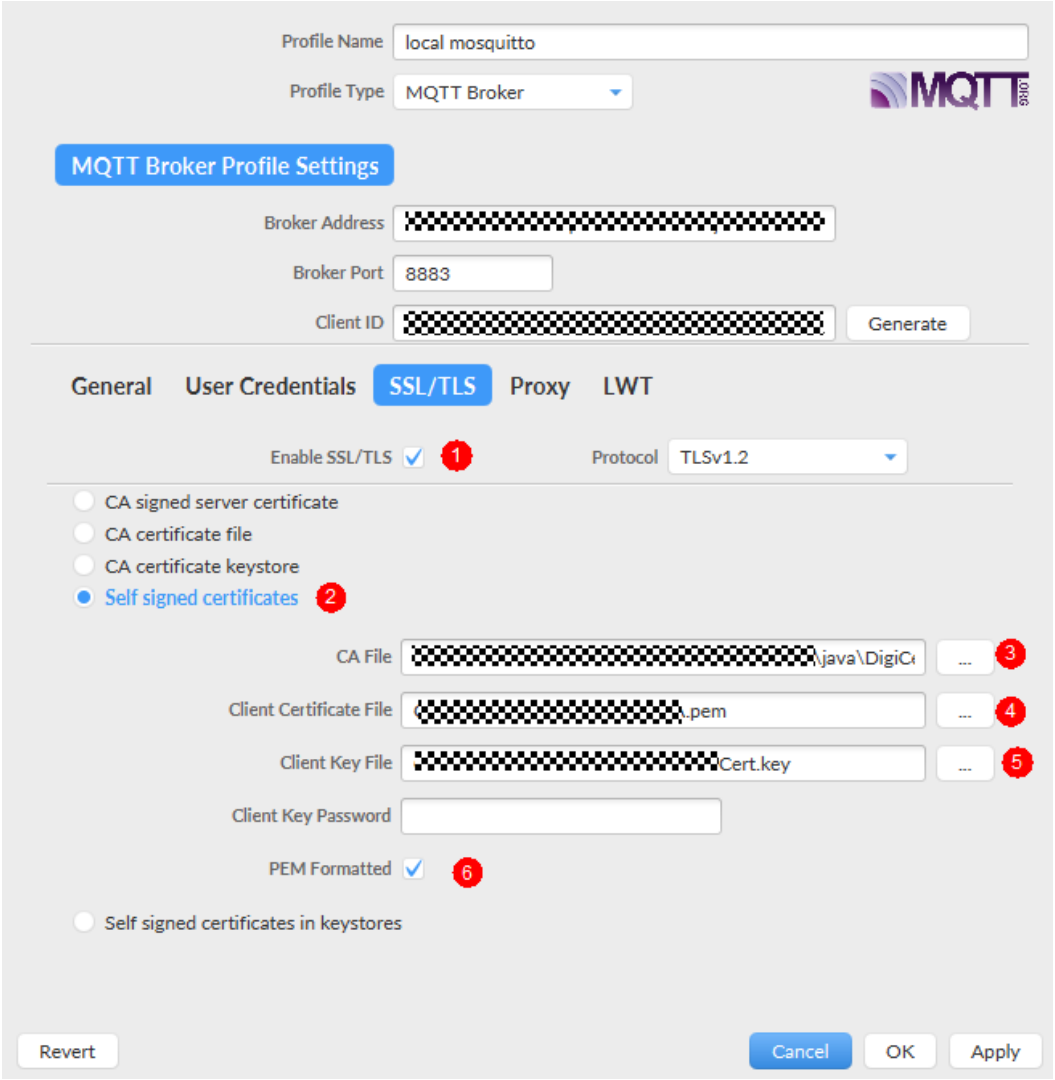
图 3-179 填写设备 ID



参数名称	说明
User Name	即设备ID，请参考2中获取。
Password	使用X.509证书认证时不需要填写。

步骤6 单击“SSL/TLS”配置鉴权参数，然后单击“Apply”。选择开启 “SSL/TLS”，勾选“ Self signed certificates”，配置相关证书内容。

图 3-180 填写 SSL/TLS 相关参数



说明

- CA File为对应的CA证书。下载并获取证书（加载pem格式的证书），获取证书请根据需要在**资源获取**里下载。
- Client Certificate File为设备的设备证书（deviceCert.pem）。
- Client Key File为设备的私钥（deviceCert.key）。

步骤7 单击“Connect”，设备鉴权成功后，在物联网平台可以看到设备处于在线状态。

图 3-181 设备列表-设备在线

----结束

相关 API 接口

- 创建设备
- 重置设备密钥
- 获取设备CA证书列表
- 上传设备CA证书
- 删除设备CA证书
- 验证设备CA证书

相关最佳实践

基于MQTT.fx的X.509证书接入

3.5.3.2 证书有效性验证（OCSP）

概述

IoTDA平台使用OCSP协议对设备侧和服务器侧证书的有效性进行校验。**OCSP**是在线证书状态协议（Online Certificate Status Protocol）的缩写，OCSP在传输安全层（TLS）握手期间检查证书吊销状态，对证书进行实时验证，与传统的证书吊销列表（CRL）相比，CRL更新频率较低，文件更大，而OCSP扩展性更强，响应更小更快，更具有实时性，更适合公开密钥基础架构（PKI）。

使用场景和名词解释

OCSP校验：本文统一指定为物联网平台对设备侧证书的有效性校验。用于证书认证的设备，平台对设备证书状态校验，检查设备证书是否已经被CA机构吊销。

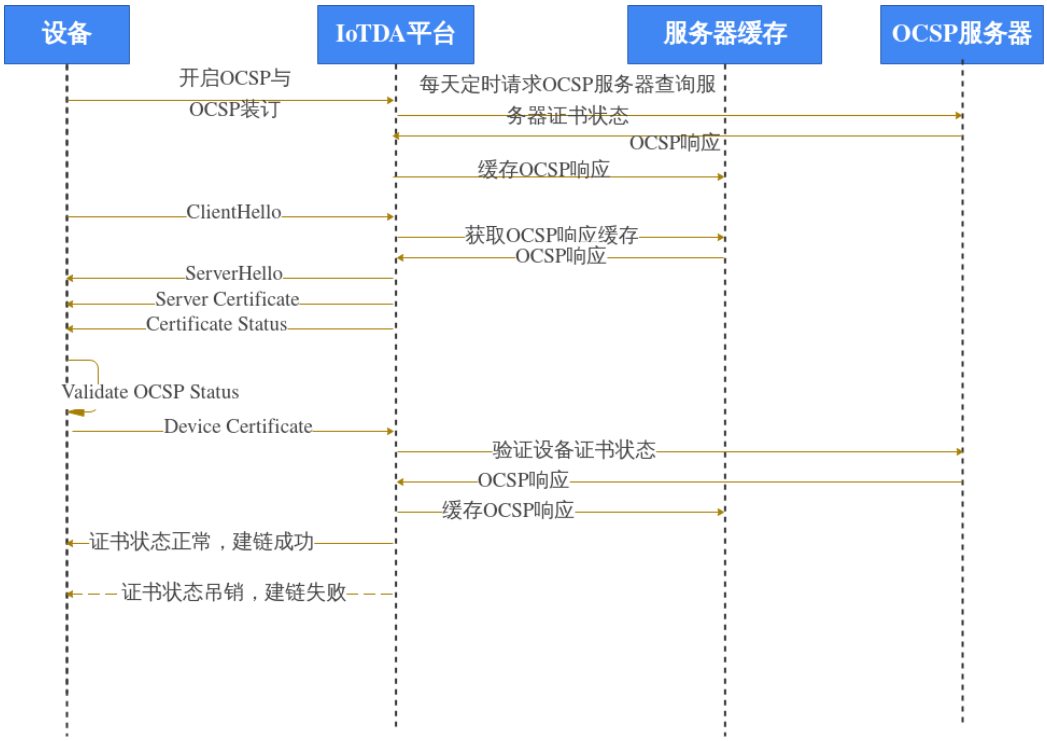
OCSP装订（OCSP Stapling）：用于设备侧对服务器证书状态校验，OCSP装订会持续检查服务器证书的吊销状态；也称服务端OCSP，是TLS证书状态查询扩展，作为在线证书状态协议的替代方法对X.509证书状态进行查询。服务器主动检查自身证书状态，在TLS握手时发送已缓存的OCSP响应，用户只需验证该响应的时效性而不用再向数字证书认证机构(CA)发送请求，可以加快握手速度。

约束与限制

- 仅企业实例支持该功能。
- 开启OCSP校验后，设备第一次连接平台由于平台要向OCSP服务器发请求，可能会导致建链时长变长，属正常现象，后续建链不受影响。
- 平台对OCSP服务器的响应缓存时长为24小时。
- 平台对OCSP服务器响应限制超时时间为5秒，响应大小不超过4k。

使用说明

图 3-182 OCSP 工作流程

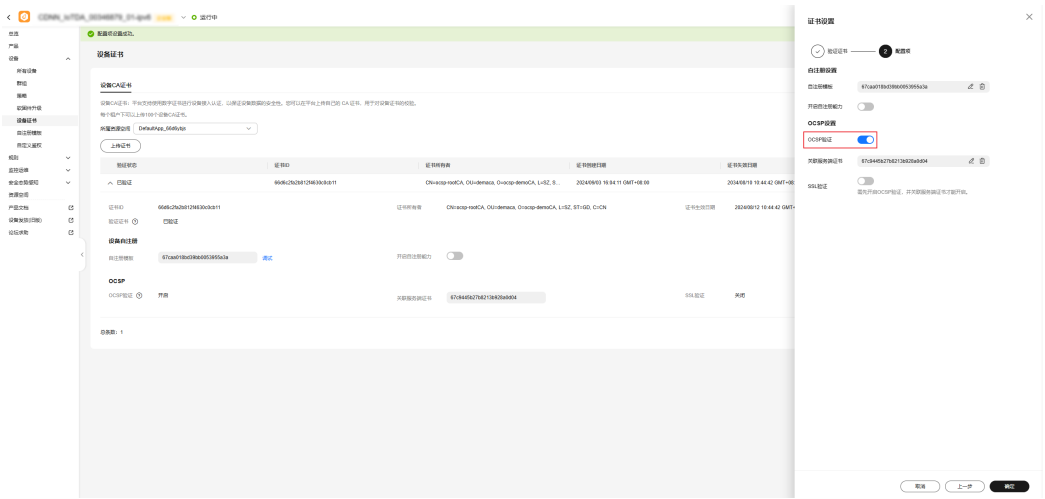


操作步骤

步骤1 访问**设备接入服务**，单击“管理控制台”进入设备接入控制台。选择您的实例，单击实例卡片进入。

步骤2 在左侧导航栏中选择“设备 > 设备证书”，进入设备证书页面选择对应CA证书，单击“证书设置”开启校验。

图 3-183 设备 CA 证书-开启 OCSP 验证



步骤3 若OCSP服务器访问方式为https协议，则在左侧导航栏中选择“规则 > 服务端证书”，单击“上传证书”，上传OCSP服务器CA证书。

返回设备接入控制台选择对应实例，单击“详情”进入实例详情页面，单击“更新证书”开启OCSP装订。若OCSP服务器访问方式为https协议，单击“SSL验证”并关联服务端证书。

查看证书是否支持OCSP以及OCSP server url，双击证书如下图所示：

图 3-184 查看 ocsp_url

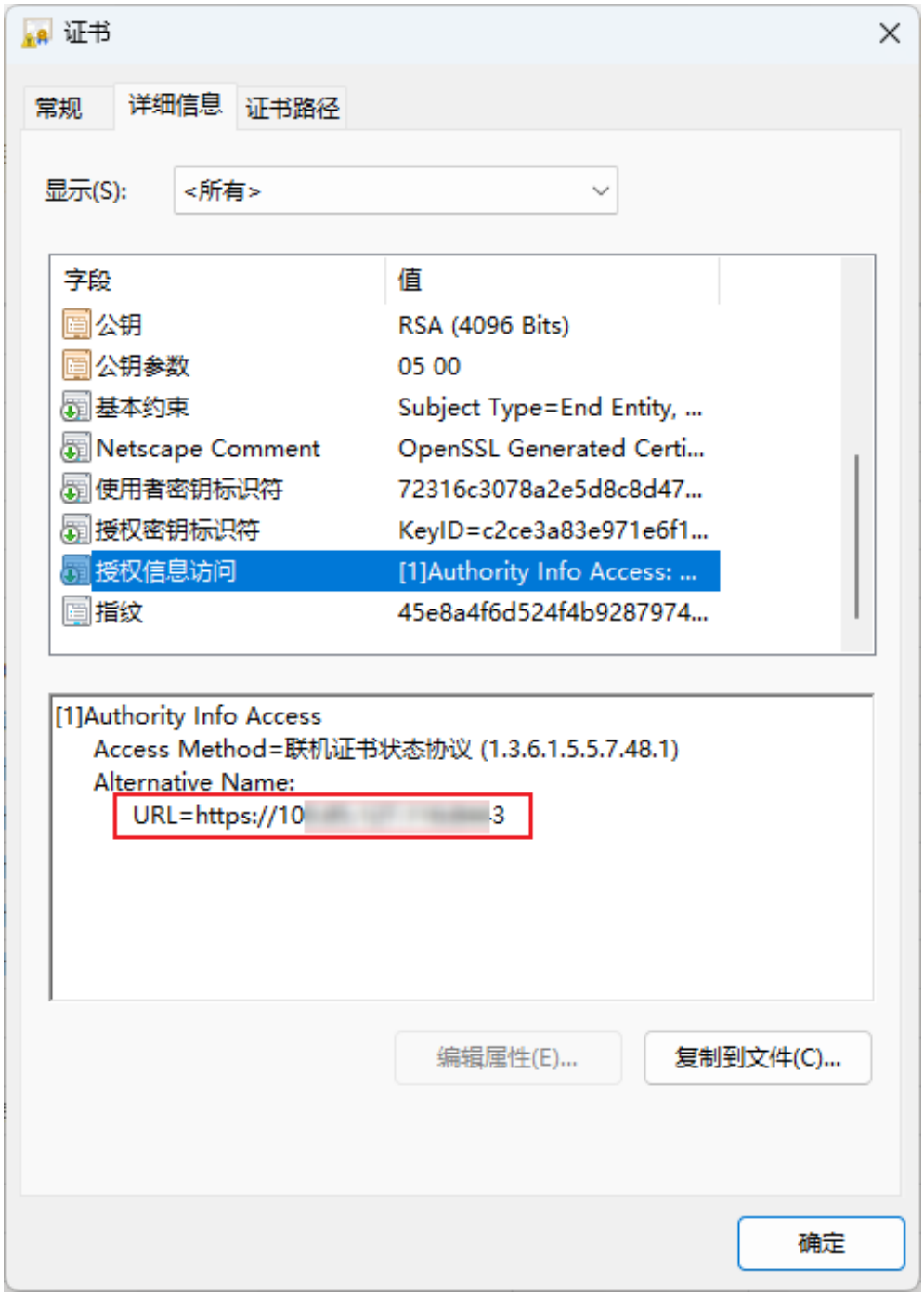
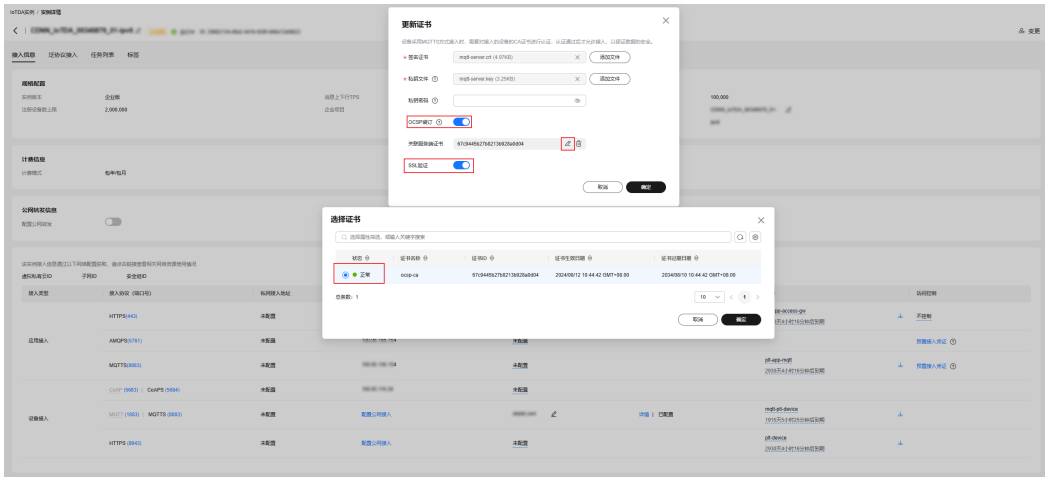


图 3-185 实例管理-开启 OCSP 装订



说明

- 开启OCSP装订签名证书链必须包含上层CA证书。
- 开启OCSP签名证书信息中必须包含ocsp url扩展字段。

步骤4 使用支持ocsp的MQTT模拟器连接平台，查看平台证书OCSP装订信息，使用抓包工具在本地抓取设备建链TLS握手报文，查看平台证书OCSP响应。证书状态共有三种类型分别是：good，revoked与unknown，设备可判断平台证书状态选择是否建立连接，比如：仅good状态下才与平台建链。

图 3-186 TLS-Certificate Status good

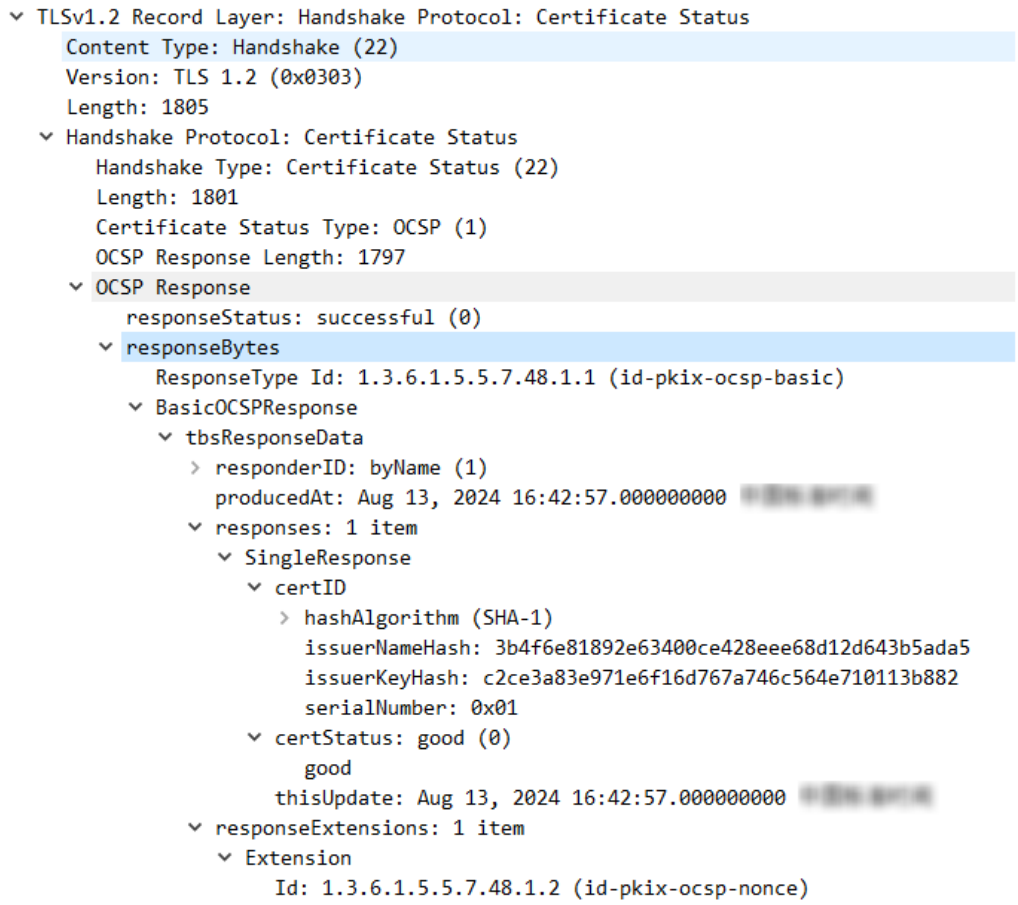


图 3-187 TLS-Certificate Status revoked

```
TLSv1.2 Record Layer: Handshake Protocol: Certificate Status
    Content Type: Handshake (22)
    Version: TLS 1.2 (0x0303)
    Length: 1851
  Handshake Protocol: Certificate Status
      Handshake Type: Certificate Status (22)
      Length: 1847
      Certificate Status Type: OCSP (1)
      OCSP Response Length: 1843
    OCSP Response
        responseStatus: successful (0)
      responseBytes
          ResponseType Id: 1.3.6.1.5.5.7.48.1.1 (id-pkix-ocsp-basic)
        BasicOCSPResponse
          tbsResponseData
            responderID: byName (1)
              byName: 0
            producedAt: May 16, 2025 17:04:04.000000000
            responses: 1 item
              SingleResponse
                certID
                  hashAlgorithm (sha256)
                      Algorithm Id: 2.16.840.1.101.3.4.2.1 (sha256)
                  issuerNameHash: 1765fd7e9a6631a50ccd267a7a02dd22af5b1594f46c9d82b0bfa4aab0ca1eee
                  issuerKeyHash: b769a9aa5f017b6edb09e1d1dbf23dbf9ca7f82e2ae9515b4c54ce0add9e42f0
                  serialNumber: 0x0d
                certStatus: revoked (1)
                  revoked
                      revocationTime: May 16, 2025 17:02:06.000000000
                thisUpdate: May 16, 2025 17:04:04.000000000
            responseExtensions: 1 item
              Extension
                  Id: 1.3.6.1.5.5.7.48.1.2 (id-pkix-ocsp-nonce)
```

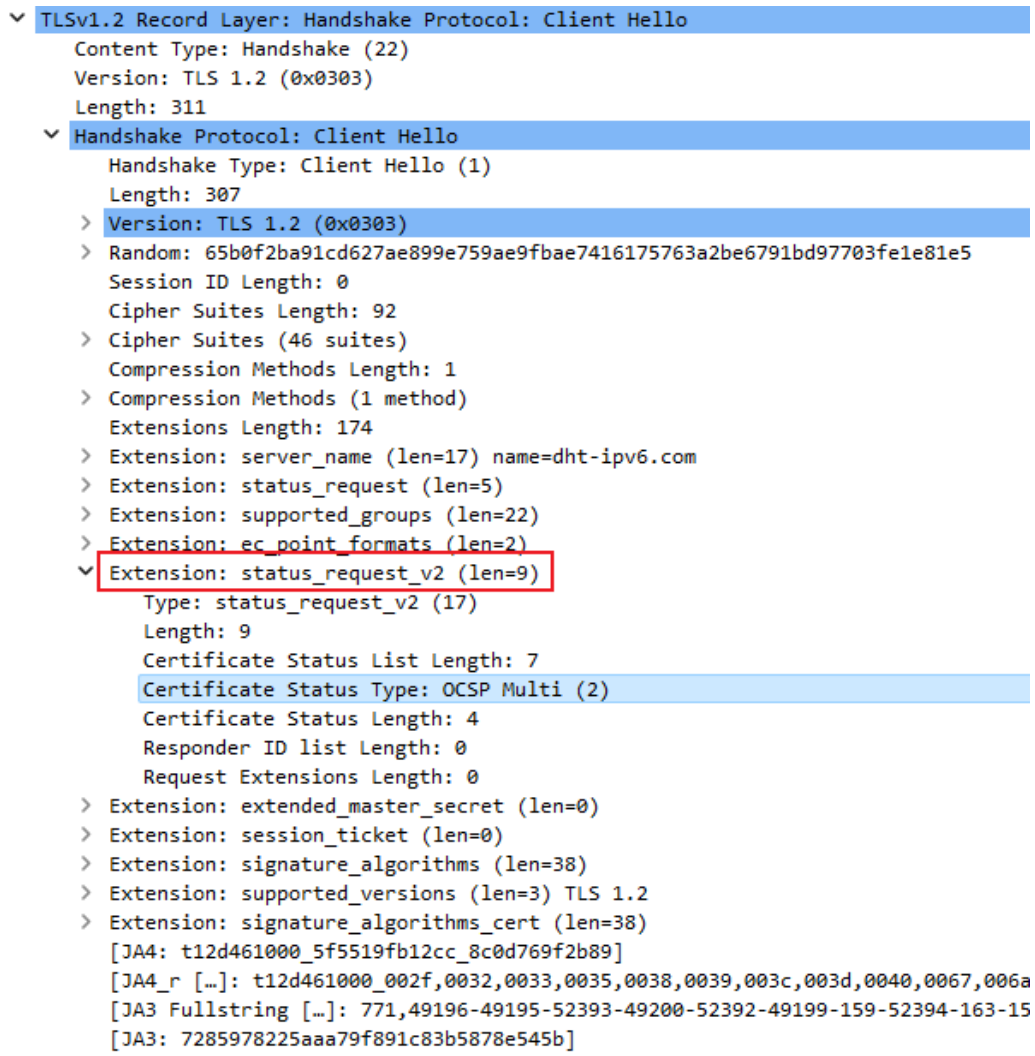
图 3-188 TLS-Certificate Status unknown

```
TLSv1.2 Record Layer: Handshake Protocol: Certificate Status
    Content Type: Handshake (22)
    Version: TLS 1.2 (0x0303)
    Length: 1834
  Handshake Protocol: Certificate Status
      Handshake Type: Certificate Status (22)
      Length: 1830
      Certificate Status Type: OCSP (1)
      OCSP Response Length: 1826
    OCSP Response
        responseStatus: successful (0)
      responseBytes
          ResponseType Id: 1.3.6.1.5.5.7.48.1.1 (id-pkix-ocsp-basic)
        BasicOCSPResponse
          tbsResponseData
            responderID: byName (1)
              byName: 0
            producedAt: May 16, 2025 17:11:09.000000000
            responses: 1 item
              SingleResponse
                certID
                  hashAlgorithm (sha256)
                      Algorithm Id: 2.16.840.1.101.3.4.2.1 (sha256)
                  issuerNameHash: e47a19c84030811d118015b4ced673b2a22b84e57e2d6e1e5f04f63d381bbf13
                  issuerKeyHash: 28f5af477864ac031960c015414f8597a324a3d56665144ed0c4f4946ff75891
                  serialNumber: 0x4e35
                certStatus: unknown (2)
                  unknown
                thisUpdate: May 16, 2025 17:11:09.000000000
            responseExtensions: 1 item
              Extension
                  Id: 1.3.6.1.5.5.7.48.1.2 (id-pkix-ocsp-nonce)
                  ReOcspNonce: 6f3dddbfdeec
          signatureAlgorithm (sha256WithRSAEncryption)
              Algorithm Id: 1.2.840.113549.1.1.11 (sha256WithRSAEncryption)
```

说明

客户端Client Hello报文中需要携带status_request扩展字段服务端才会返回Certificate Status。

图 3-189 status request



步骤5 在左侧导航栏中选择“设备 > 所有设备”，选择对应设备，单击“详情 > 消息跟踪 > 启动消息跟踪”，使用MQTT模拟器证书双向认证，设备端使用已吊销证书，查看消息跟踪错误详情。若设备证书已被吊销，建议使用新的有效证书接入，若证书泄漏建议及时吊销状态。

图 3-190 设备证书-已吊销

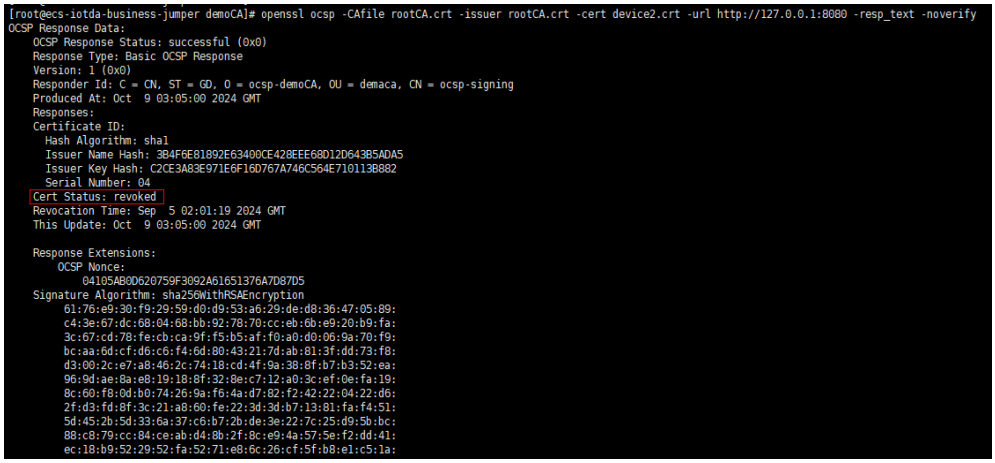


图 3-191 消息跟踪-OCSP 校验失败详情

----结束

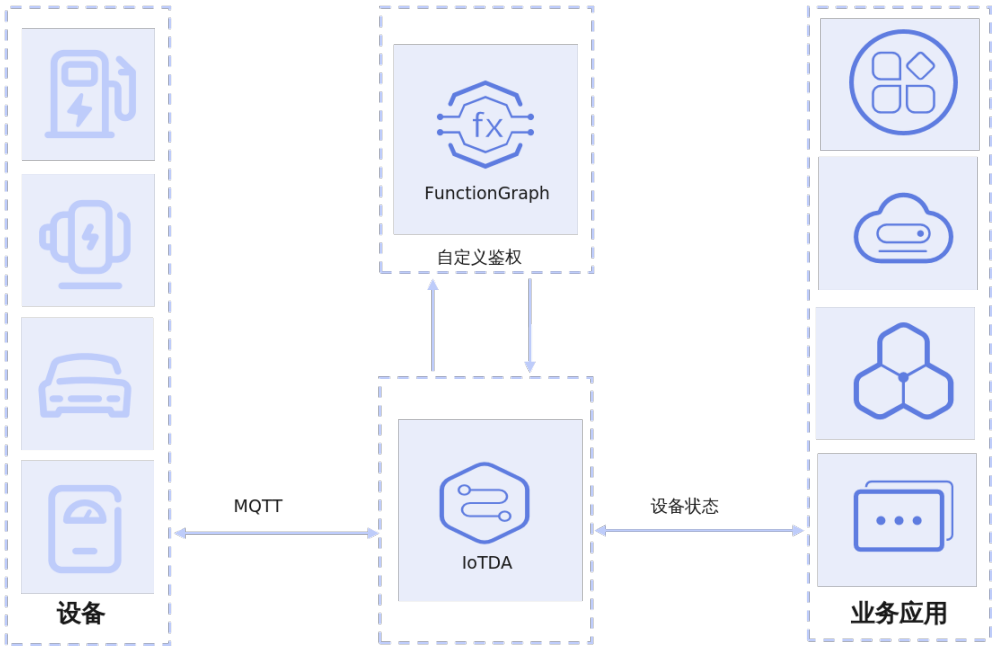
3.5.4 自定义鉴权

概述

自定义鉴权是指用户可以通过函数工作流（FunctionGraph）服务实现自定义鉴权逻辑，以对接入平台的设备进行身份认证。

在设备接入物联网平台前，用户可以通过应用服务调用控制台配置自定义鉴权信息，然后通过调用函数工作流（**FunctionGraph**）配置自定义鉴权函数。在设备接入物联网平台时，物联网平台会获取设备ID和自定义鉴权函数名称等参数，并向FunctionGraph发起发起鉴权请求，由用户实现鉴权逻辑以完成设备的接入鉴权。

图1 自定义鉴权业务架构图



使用场景

- 迁移场景：当用户的设备从第三方云平台迁移到IoTDA平台时，存量设备可以依据现有的设备鉴权方式，在平台自定义配置设备的鉴权逻辑，以实现设备鉴权方式免改动迁移。
- 原生场景：用户有自定义实现鉴权逻辑的需求（比如：用户不同的设备有其自定义的认证方式），而无需依赖于平台默认的鉴权方式。

约束与限制

- 使用自定义鉴权功能，要求设备必须使用TLS同时支持**SNI(Server Name Indication)**，SNI中需要携带平台分配的域名。
- 每个用户默认最多支持10个自定义鉴权的配置。
- 自定义鉴权的函数最大处理时间为5秒，5秒内函数没返回结果，则认为鉴权失败。
- 每个用户总鉴权请求的TPS限制参考**产品规格说明**，自定义鉴权为总鉴权TPS的50%（不包含设备自注册）。
- 若用户开启了缓存FunctionGraph的鉴权结果，则在相同参数下，函数服务的修改生效时间需在缓存超期后才能生效。
- 在所有的设备接入鉴权方式中，当满足自定义鉴权条件时（匹配到设备携带的自定义鉴权器名称或用户配置了默认自定义鉴权器），优先采用自定义鉴权方式进行设备接入。

说明

自定义认证功能是为方便用户快速接入平台，免于设备侧改造。平台将使用您提供的鉴权模板进行鉴权，请合理审视认证方式的安全程度，避免使用弱校验或者免校验。由于您自定义模板安全程度过低造成的安全问题，平台将不承担任何安全责任。

使用说明

图 3-192 自定义鉴权操作流程

操作步骤

步骤1 配置自定义鉴权函数：用户通过调用函数服务（**FunctionGraph**）创建自定义鉴权函数。单击控制台，搜索函数工作流FunctionGraph服务进入，创建函数。

图 3-193 函数列表-创建函数

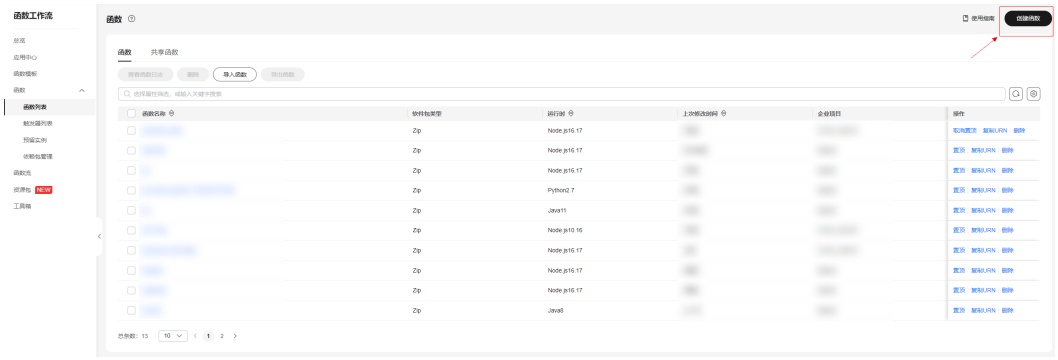
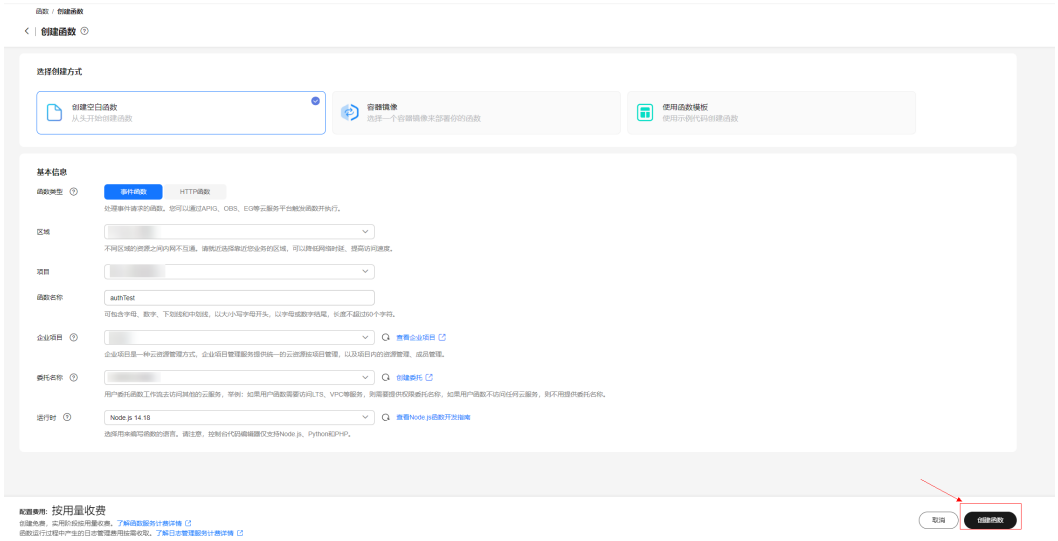


图 3-194 创建函数-参数信息



步骤2 创建自定义鉴权：用户可以通过Console配置自定义鉴权信息，IoTDA负责自定义鉴权信息存库和进行相应的管理维护。用户最多支持配置10个自定义鉴权器。

图 3-195 自定义鉴权-入口

图 3-196 自定义鉴权-创建鉴权

表 3-28 自定义鉴权参数信息

参数名称	是否必选	描述
鉴权名称	是	自定义鉴权器名称。
鉴权函数	是	自定义鉴权器对应的函数名称，从**步骤1**中FunctionGraph已经创建的函数列表中选取。
状态	是	激活停用参数，用于表示该鉴权是否为激活状态，默认为停用状态。若需使用该鉴权器，则应激活后才能生效。
签名认证	是	启动签名认证后，不满足签名要求的鉴权信息将被拒绝，以减少无效的函数调用，默认为开启。
token值	否	签名校验的token值，开启签名校验时使用，用于认证设备携带的签名信息是否正确。
公钥	否	签名校验的公钥，开启签名校验时使用，用于认证设备携带的签名信息是否正确。
是否默认自定义鉴权器	是	开启后，如果设备发起鉴权时的username没有携带authorizer_name参数，则默认使用此鉴权器。此处默认为不开启。
是否缓存	是	缓存开关，用于开启缓存FunctionGraph的鉴权结果，缓存时间为300分钟~1天，默认不开启。

步骤3 设备发起鉴权请求：设备通过MQTT协议发起的CONNECT请求需携带username参数，该参数应包含自定义鉴权的相关可选参数。

- username格式如下。参数传值时需去掉大括号“{}”，username的每个参数需采用分隔符“|”分隔，各参数具体内容不要出现“|”，否则可能导致鉴权失败。
  {**device-identifier**}|authorizer-name={**authorizer-name**}|authorizer-signature={**token-signature**}|signing-token={**token-value**}
  示例：
  659b70a0bd3f665a471e5ec9_auth|authorizer-name=Test_auth_1|authorizer-signature=***|signing-token=tokenValue

表 3-29 username 参数信息

参数名称	是否必选	描述
device-identifier	是	设备标识符，建议设定为设备id。
authorizer-name	否	自定义鉴权器名称，需要同用户配置的鉴权器一致。不携带鉴权器名称时，如用户配置了默认的自定义鉴权器，则使用自定义鉴权，否则使用原有的密钥/证书鉴权方式。
authorizer-signature	否	签名校验功能开启时，需要携带该参数。该参数由私钥和signing-token进行加密后获取。需与**步骤2**中创建自定义鉴权时填写的鉴权名称保持一致。
signing-token	否	签名校验功能开启时，需要携带该参数。该参数为签名校验的token，需与**步骤2**中创建自定义鉴权时填写的token值保持一致。

- authorizer-signature通过如下命令行获取：
  echo -n {signing-token} | openssl dgst -sha256 -sign {private key} | openssl base64

表 3-30 命令行参数解析

参数名称	描述
echo -n {signing-token}	使用echo命令将签名令牌（signing-token）的值输出，并使用-n参数去掉末尾的换行符。signing-token需与**步骤2**中创建自定义鉴权时填写的**token值**保持一致。
openssl dgst -sha256 -sign	使用 SHA256 算法对输入的数据进行哈希处理。
{private key}	使用 RSA 算法加密的私钥，可传入.pem/.key格式的私钥文件。
openssl base64	将签名结果进行Base64编码，以便于传输和存储。

步骤4 IoTDA处理鉴权请求：IoTDA收到鉴权请求时，会根据username的参数信息和自定义鉴权的配置来决策是否使用自定义鉴权方式。

1. 判断username是否携带了自定义鉴权名称，如果有携带，则根据平台的自定义鉴权名称，去匹配鉴权器处理函数。如果没有携带，则使用用户配置的默认自定义鉴权器去匹配鉴权处理函数，若无匹配到则使用原有的密钥/证书鉴权方式。

2. 检查用户是否开启了签名校验，如果开启了签名校验，则校验username携带的签名信息是否可以校验成功，校验失败则直接返回鉴权失败。
3. 匹配到处理函数后，携带设备的鉴权信息（即**步骤5**的入参event）并通过函数的URN（Uniform Resource Name，唯一标识函数。）向FunctionGraph发起鉴权请求。

步骤5 用户实现鉴权逻辑：用户在**步骤1**中通过FunctionGraph创建的处理函数中进行开发，函数的返回结果需要符合要求，函数使用示例及返回的JSON格式要求如下：

```
exports.handler = async (event, context) => {
    console.log("username=" + event.username);
    // 此处编写您的校验逻辑


    // 返回的JSON格式（格式固定不变）
    const authRes = {
        "result_code": 200,
        "result_desc": "successful",
        "refresh_seconds": 300,
        "device": {
            "device_id": "myDeviceId",
            "provision_enable": true,
            "provisioning_resource": {
                "device_name": "myDeviceName",
                "node_id": "myNodeId",
                "product_id": "myProductId",
                "app_id": "customization0000000000000000000",
                "policy_ids": ["657a4e0c2ea0cb2cd831d12a", "657a4e0c2ea0cb2cd831d12b"]
            }
        }
    }
    return JSON.stringify(authRes);
}
```

其中，函数的请求参数（即event，JSON格式）如下：

```
{
    "username": "myUserName",
    "password": "myPassword",
    "client_id": "myClientId",
    "certificate_info": {
        "common_name": "",
        "fingerprint": "123"
    }
}
```

表 3-31 请求参数信息

参数名称	参数类型	是否必选	描述
username	String	是	MQTT协议中CONNECT消息的username字段，与**步骤3**的username格式相同。
password	String	是	MQTT协议中CONNECT消息的password参数。
client_id	String	是	MQTT协议中CONNECT消息的clientId参数。
certificate_info	JsonObject	否	MQTT协议中CONNECT消息的设备证书信息。

表 3-32 certificate_info 证书信息

参数名称	参数类型	是否必选	描述
common_name	String	是	设备携带设备证书时，从设备证书中解析的commonName信息。
fingerprint	String	是	设备携带设备证书时，从设备证书中解析的指纹信息。

表 3-33 返回参数信息

参数名称	参数类型	是否必选	描述
result_code	Integer	是	鉴权结果码， 返回200标识鉴权成功。
result_desc	String	否	鉴权结果描述信息。
refresh_seconds	Integer	否	鉴权结果的缓存时间，单位(s)。
device	JsonObject	否	认证成功时的设备信息。当设备信息中的设备ID不存在时，如果用户开启自注册**设备自注册**，则平台会根据设备信息自动创建对应的设备。

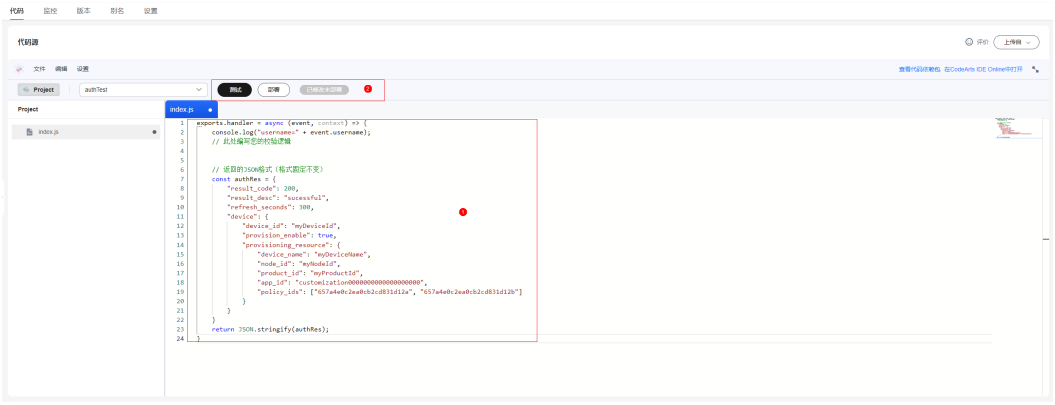
表 3-34 device 设备信息

参数名称	参数类型	是否必选	描述
device_id	String	是	**参数说明：**设备ID，自注册场景、非自注册场景都必选。全局唯一，用于唯一标识一个设备。如果携带该参数，平台将设备ID设置为该参数值；建议使用product_id + _ + node_id拼接而成。 **取值范围：**长度不超过128，只允许字母、数字、下划线（_）、连接符（-）的组合，建议不少于4个字符。
provision_enable	Boolean	否	**参数说明：**自注册开关，默认为false。
provisioning_resource	JsonObject	自注册场景必选	**参数说明：**自注册参数信息。

表 3-35 provisioning_resource 自注册参数信息

参数名称	参数类型	是否必选	描述
device_name	String	否	**参数说明：**设备名称，资源空间下唯一，用于资源空间下唯一标识一个设备。 **取值范围：**长度不超过256，只允许中文、字母、数字、以及_?'#().,&%@!-等字符的组合，建议不少于4个字符。 最小长度：**1** 最大长度：**256**
node_id	String	是	**参数说明：**设备标识码，通常使用IMEI、MAC地址或Serial No作为node_id。 设备标识码长度为1到64个字符，包含英文字母、数字、连接号-和下划线_。 注意：NB设备由于模组烧录信息后无法配置，所以NB设备会校验node_id全局唯一。 **取值范围：**长度不超过64，只允许字母、数字、下划线（_）、连接符（-）的组合，建议不少于4个字符。
product_id	String	是	**参数说明：**设备关联的产品ID，用于唯一标识一个产品模型，创建产品后获得。取值范围：长度不超过256，只允许中文、字母、数字、以及_?'#().,&%@!-等字符的组合，建议不少于4个字符。 最小长度：**1** 最大长度：**256**
app_id	String	是	**参数说明：**资源空间ID，该参数指定创建的设备归属到哪个资源空间。 **取值范围：**长度不超过36，只允许字母、数字、下划线（_）、连接符（-）的组合。
policy_ids	List<String>	否	**参数说明：**Topic策略ID。

图 3-197 函数编写-部署



步骤6 FunctionGraph返回结果后，判断是否需要支持自注册功能，如果需要则触发设备的自动注册。自注册设备全部默认为密钥认证，密钥随机生成。IoTDA收到鉴权结果后，进行接下来的业务实现流程。

----结束

3.5.5 自定义模板鉴权

3.5.5.1 自定义模板使用说明

概述

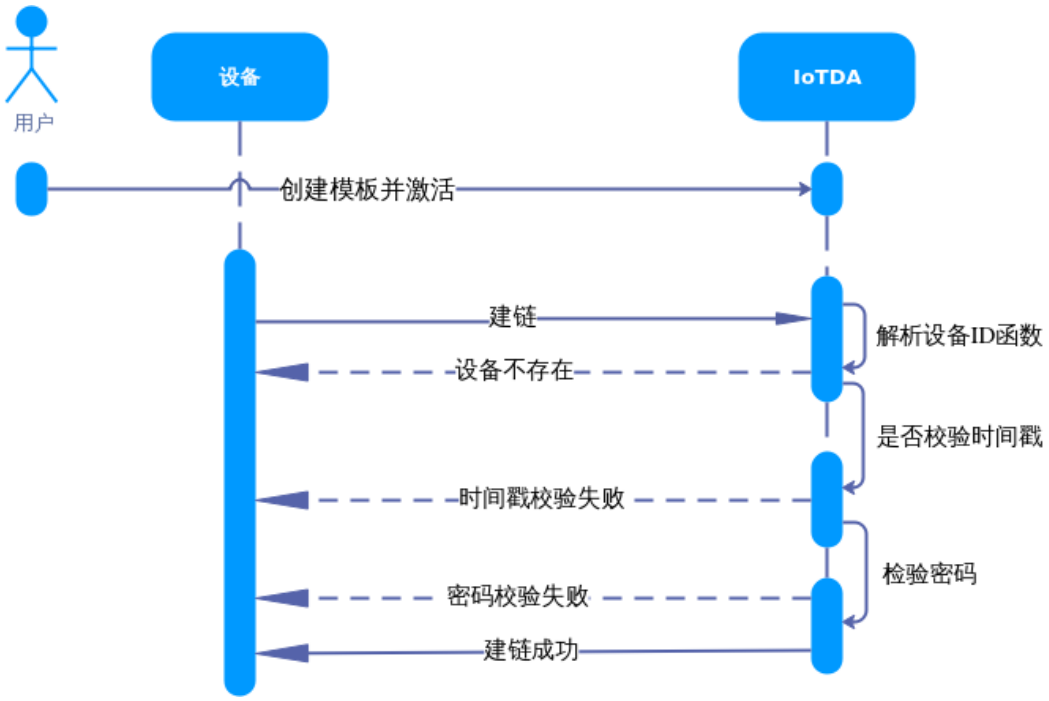
自定义模板鉴权是指用户可以使用平台提供的**内部函数**实现自定义的鉴权方式，而不需使用**平台默认鉴权方式**，对接入平台的设备进行身份认证，用户可以通过平台预置的函数灵活编排鉴权方式。

应用场景

- 迁移场景：当设备从第三方物联网平台迁移到华为云IoTDA时，通过配置平台提供的自定义鉴权模板可以兼容原来的鉴权方式，设备侧无需改动实现无缝迁移。
- 原生场景：自定义模板鉴权可以让用户扩展自己的设备鉴权方式，而不需要与平台默认鉴权绑定，极大的提高了设备可扩展性。

自定义模板鉴权流程

图 3-198 自定义模板鉴权流程图



约束与限制

1. 使用自定义鉴权功能，要求设备必须使用TLS同时支持**SNI(Server Name Indication)**，SNI中需要携带平台分配的域名。
2. 默认每个用户最多支持5个自定义鉴权模板，只能启用一个激活状态的模板。
3. 鉴权模板函数嵌套最大深度为5层。
4. 模板内容体最大长度不能超过4000字符，且不能包含中文字符。
5. 设备为密钥认证类型时，模板密码函数必须包含设备原始密钥参数(iotda::device::secret)。
6. 使用模板鉴权时，鉴权参数username不能与自定义函数鉴权username格式重叠，否则会使用自定义函数鉴权，比如：
   ```
   {deviceId}|authorizer-name={authorizer-name}|xxx
   ```
7. 自定义模板鉴权优先级高于平台默认鉴权，即激活自定义鉴权模板后设备就会使用模板鉴权，不会再使用平台默认鉴权方式。

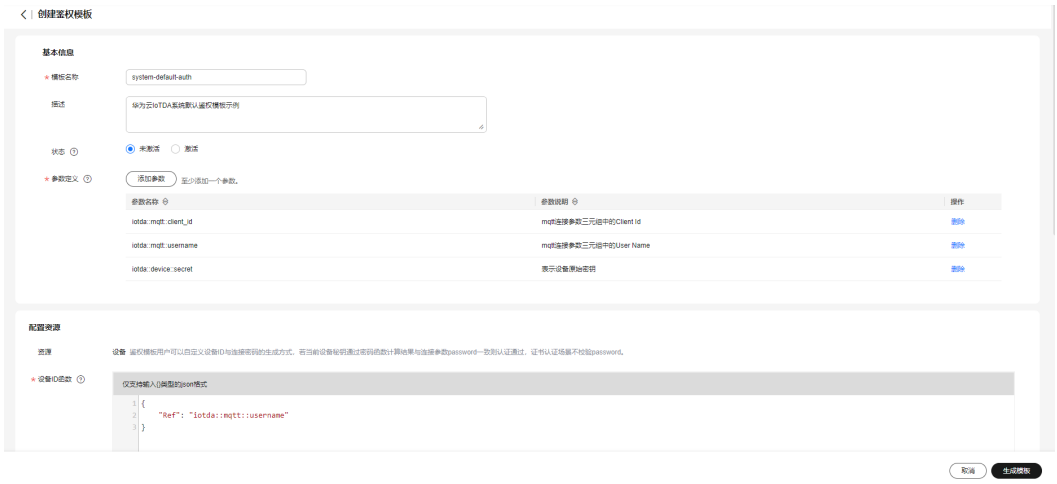
说明

自定义认证功能是为方便用户快速接入平台，免于设备侧改造，平台将使用您提供的鉴权模板进行鉴权，请合理审视认证方式的安全程度，避免使用弱校验或者免校验，由于您自定义模板安全程度过低造成的安全问题，平台将不承担任何安全责任。

操作步骤

步骤1 创建鉴权模板：进入设备接入控制台左侧导航栏，选择“设备 > 自定义鉴权”，单击“自定义模板”，单击“创建鉴权模板”。本示例演示使用的鉴权模板与**系统默认鉴权**一致。

图 3-199 自定义鉴权-创建鉴权模板



模板整体内容如下：

```
{
  "template_name": "system-default-auth",
  "description": "华为云IoTDA系统默认鉴权模板示例",
  "status": "ACTIVE",
  "template_body": {
    "parameters": {
      "iotda::mqtt::client_id": {
        "type": "String"
      },
      "iotda::mqtt::username": {
        "type": "String"
      },
      "iotda::device::secret": {
        "type": "String"
      }
    },
    "resources": {
      "device_id": {
        "Ref": "iotda::mqtt::username"
      },
      "timestamp": {
        "type": "FORMAT",
        "pattern": "yyyyMMddHH",
        "value": {
          "Fn::SubStringAfter": [
            "${iotda::mqtt::client_id}",
            "_0_1_"
          ]
        }
      },
      "password": {
        "Fn::HmacSHA256": [
          "${iotda::device::secret}",
          {
            "Fn::SubStringAfter": [
              "${iotda::mqtt::client_id}",
              "_0_1_"
            ]
          }
        ]
      }
    }
  }
}
```

表 3-36 鉴权模板参数信息

参数	参数名称	是否必填	描述
template_name	模板名称	是	鉴权模板名称，单个用户下模板名称不能重复，长度不超过128，只允许字母、数字、下划线（_）、连接符（-）的组合。
description	描述	否	鉴权模板的描述信息，长度不超过2048，只允许中文、字母、数字、以及_?'#().,&%@!-等字符的组合。
status	状态	否	是否激活该模板，默认状态为未激活，一个用户下只能有一个已激活状态的模板。
parameters	参数	是	平台预定义的MQTT连接参数列表，当设备使用密码认证时模板必须包含设备原始密钥参数（iotda::device::secret）。 平台预定义了如下参数： iotda::mqtt::client_id ： mqtt连接参数三元组中的Client Id iotda::mqtt::username ： mqtt连接参数三元组中的User Name iotda::certificate::country : 设备证书（国家/地区,C ） iotda::certificate::organization : 设备证书（组织,O ） iotda::certificate::organizational_unit : 设备证书（组织单位,OU ） iotda::certificate::distinguished_name_qualifier : 设备证书（可辨别名称限定符,dnQualifier ） iotda::certificate::state_name : 设备证书（省市,ST ） iotda::certificate::common_name : 设备证书（公用名,CN ） iotda::certificate::serial_number : 设备证书（序列号,serialNumber ） iotda::device::secret ： 表示设备原始密钥
device_id	设备ID函数	是	设备ID取值函数，JSON格式，平台通过解析该函数获取对应设备信息。
timestamp	是否开启时间戳校验	否	是否校验设备连接信息中的时间戳，如果设备连接参数（clientId、username）中包含时间戳建议开启校验。开启校验平台会对比设备携带时间戳与平台系统时间，若设备时间戳加一小时小于平台系统时间则校验失败。

参数	参数名称	是否必填	描述
type	时间戳类型	否	UNIX：表示时间戳格式为Unix时间戳，长整型，单位秒。 FORMAT：格式化时间戳类型，比如：2024-03-28 11:47:39、2024/03/28 03:49:13
pattern	时间戳格式	否	时间格式模板，时间戳类型为FORMAT时必填，具体字符含义如下： y：年 M：月 d：日 H：时 m：分 s：秒 S：毫秒 示例：yyyy-MM-dd HH:mm:ss、yyyy/MM/dd HH:mm:ss
value	时间戳取值函数	否	开启时间戳校验后必填，设备时间戳获取函数，平台通过执行该函数获取设备建链时的时间戳。
password	MQTT密码函数	否	密码函数，当设备认证类型为密钥认证时必填，且模板参数中必须包含设备原始密钥参数（iotda::device::secret），当设备为证书认证时可不填。设备认证类型参考**注册设备**。平台把设备原始密钥等参数填入函数计算结果，若函数结果与设备建链携带的password参数一致则认证通过，否则认证失败。

步骤2 选择设备调试模板：单击“调试”选择一个设备进行调试，输入mqtt连接参数后单击“调试”查看调试结果。注意：使用系统标准格式的clientId平台会校验username参数与clientId前缀一致。

图 3-200 自定义模板-调试

设备调试成功后单击“激活”启用模板，一旦激活模板所有设备鉴权将使用该模板，且已激活状态下的模板不能修改，后续修改模板建议新创建一个副本模板进行调试，确认无误后进行模板切换。

步骤3 使用mqtt.fx工具模拟设备真实建链，在控制台查看设备处于在线状态，Broker Address填**平台接入地址**，选择“总览 > 接入信息”，端口使用8883端口。

图 3-201 设备建链

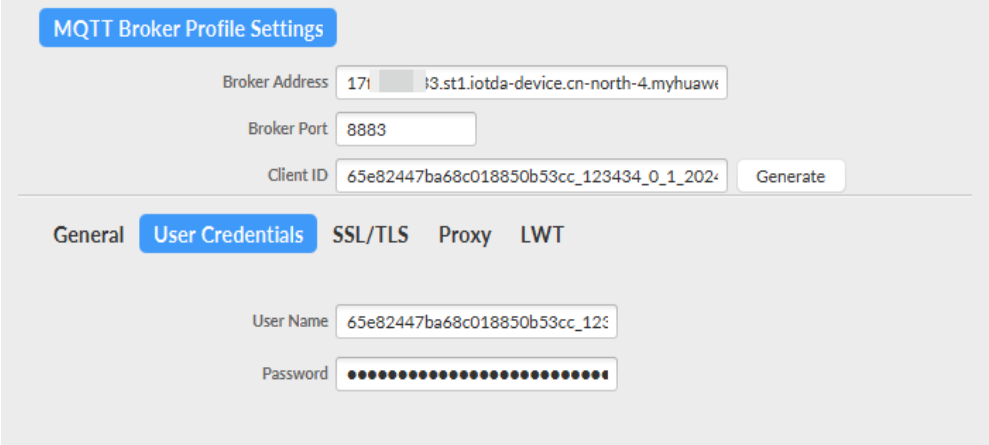


图 3-202 设备列表-设备在线

----结束

3.5.5.2 自定义模板鉴权示例

示例 1

证书认证设备，不限制UserName与ClientId参数取值，从设备证书通用名称（Common Name）中取值设备ID。

表 3-37 鉴权参数

参数	说明
Client ID	任意值
User Name	任意值
Password	空值

鉴权模板：

```
{
   "template_name": "template1",
   "description": "template1",
   "template_body": {
      "parameters": {
         "iotda::certificate::common_name": {
            "type": "String"
         }
      },
      "resources": {
```

```
        "device_id": {
            "Ref": "iotda::certificate::common_name"
        }
    }
  }
}
```

示例 2

当设备ID格式为：${ProductId}_${NodeId}，鉴权参数格式如下表所示时，自定义鉴权模板配置如下代码所示：

表 3-38 鉴权参数

参数	说明
Client ID	固定格式： **${ClientId}\|securemode=2,signmethod=hmacsha256,timestamp=${timestamp}|** ● *${ClientId}：* 固定格式 ${ProductId}.${NodeId}。 – *${NodeId}：* 设备标识码。 – *${ProductId}*：产品ID。 ● *${timestamp}：* Unix时间戳，毫秒。
User Name	由设备标识码和产品ID组成，固定格式： **${NodeId}&${ProductId}**
Password	以设备密码为密钥，将设备参数与参数值拼接后的字符串为加密串进行hmacsha256算法加密。 加密串格式： **clientId${clientId}deviceName${nodeId}productKey${productId}timestamp${timestamp}** ● *${ClientId}：* 固定格式 ${ProductId}.${NodeId}。 ● *${NodeId}：* 设备标识码。 ● *${ProductId}*：产品ID。 ● *${timestamp}：* 时间戳。

鉴权模板：

```
{
  "template_name": "template2",
  "description": "template2",
  "template_body": {
    "parameters": {
      "iotda::mqtt::client_id": {
        "type": "String"
      },
      "iotda::mqtt::username": {
        "type": "String"
      },
      "iotda::device::secret": {
        "type": "String"
      }
    },
    "resources": {
      "device_id": {
        "Fn::Join": [{
```

```
                "Fn::SplitSelect": [
                    "${iotda::mqtt::username}",
                    "&",
                    1
                ]
            }, "_", {
                "Fn::SplitSelect": [
                    "${iotda::mqtt::username}",
                    "&",
                    0
                ]
            }]
        },
        "timestamp": {
            "type": "UNIX",
            "value": {
                "Fn::MathDiv": [{
                    "Fn::ParseLong": {
                        "Fn::SplitSelect": [{
                            "Fn::SubStringAfter": [{
                                "Fn::SplitSelect": ["${iotda::mqtt::client_id}", "|", 1]
                            }, "timestamp="]
                        }, ",", 0]
                    }
                }, 1000]
            }
        },
        "password": {
            "Fn::HmacSHA256": [{
                    "Fn::Sub": [
                        "clientId${clientId}deviceName${deviceName}productKey${productKey}timestamp${timestamp}",
                        {
                            "clientId": {
                                "Fn::SplitSelect": [
                                    "${iotda::mqtt::client_id}",
                                    "|",
                                    0
                                ]
                            },
                            "deviceName": {
                                "Fn::SplitSelect": [
                                    "${iotda::mqtt::username}",
                                    "&",
                                    0
                                ]
                            },
                            "productKey": {
                                "Fn::SplitSelect": [
                                    "${iotda::mqtt::username}",
                                    "&",
                                    1
                                ]
                            },
                            "timestamp": {
                                "Fn::SplitSelect": [{
                                    "Fn::SubStringAfter": [{
                                        "Fn::SplitSelect": ["${iotda::mqtt::client_id}", "|", 1]
                                    }, "timestamp="]
                                }, ",", 0]
                            }
                        }
                    ]
                },
                "${iotda::device::secret}"
            ]
        }
    }
```

```
    }
}
```

示例 3

当设备ID格式为：${productId}${nodeId}，鉴权参数格式如下表所示时，自定义鉴权模板配置如下代码所示：

表 3-39 鉴权参数

参数	说明
Client ID	固定格式： **${productId}${nodeId}** ● *${productId}*：产品ID。 ● *${nodeId}：* 设备标识码。
User Name	固定格式： **${productId}${nodeId};12010126;${connid};${expiry}** ● *${productId}：* 产品ID。 ● *${nodeId}：* 设备标识码。 ● *${connid}：*一个随机字符串。 ● *${expiry}：Unix时间戳，单位秒。*
Password	固定格式： **${token};hmacsha256** ● *${token}：*以BASE64解码后的设备密码为密钥，对User Name字段进行hmacsha256算法加密后的值。

鉴权模板：

```
{
    "template_name": "template3",
    "description": "template3",
    "template_body": {
        "parameters": {
            "iotda::mqtt::client_id": {
                "type": "String"
            },
            "iotda::mqtt::username": {
                "type": "String"
            },
            "iotda::device::secret": {
                "type": "String"
            }
        },
        "resources": {
            "device_id": {
                "Ref": "iotda::mqtt::client_id"
            },
            "timestamp": {
                "type": "UNIX",
                "value": {
                    "Fn::ParseLong": {
                        "Fn::SplitSelect": ["${iotda::mqtt::username}", ";", 3]
                    }
                }
```

```
        },
        "password": {
            "Fn::Sub": [
                "${token};hmacsha256",
                {
                    "token": {
                        "Fn::HmacSHA256": [
                            "${iotda::mqtt::username}",
                            {
                                "Fn::Base64Decode": "${iotda::device::secret}"
                            }
                        ]
                    }
                }
            ]
        }
    }
  }
}
```

3.5.5.3 自定义模板内部函数

使用说明

华为云IoTDA提供了多个内部函数供用户在模板中使用，使用时请认真阅读每个函数的功能定义，包括入参类型，参数长度，返回值类型等。

说明

- 整个函数必须是合法的Json格式。
- 函数中可使用${}变量占位符或者"Ref"函数引用入参定义的参数值。
- 函数所使用的参数必须在模板参数中声明。
- 单一入参的函数后面直接跟参数，比如："Fn::Base64Decode": "${iotda::mqtt::username}"。
- 多个入参的函数后面接数组格式，比如："Fn::HmacSHA256": ["${iotda::mqtt::username}", "${iotda::device::secret}"]。
- 函数可以嵌套使用，即一个函数的参数可以是另一个函数，注意嵌套函数的返回值必须跟当前函数参数类型一致，比如：{"Fn::HmacSHA256": ["${iotda::mqtt::username}", {"Fn::Base64Encode": "${iotda::device::secret}"}]}
- 整个鉴权模板中hash函数（Fn::HmacSHA256）最多出现两次
- 整个鉴权模板中BASE64函数（Fn::Base64Decode Fn::Base64Encode）个数的和 不能超过2个
- 鉴权模板中密码进行HmacSHA256Function之后的结果不允许 在进行Fn::Split（字符串拆分函数）Fn::SplitSelect（字符串拆分选取函数）Fn::SubStringAfter（字符串拆分函数，截取分隔符后面字符） Fn::SubStringBefor（字符串拆分函数，截取分隔符前面字符）操作。

Fn::ArraySelect

内部函数Fn::ArraySelect返回一个字符串数组中索引为index的字符串元素。

JSON

```
{"Fn::ArraySelect": [index, [StringArray]]}
```

表 3-40 参数说明

参数名称	类型	说明
index	int	整型，数组元素索引值，从0开始计算。
StringArray	String[]	字符串数组元素。
返回值	String	索引为index的元素。

示例如下：

```
{
    "Fn::ArraySelect": [1, ["123", "456", "789"]]
}
return: "456"
```

Fn::Base64Decode

内部函数Fn::Base64Decode将一个字符串按BASE64解码成一个字节数组。

JSON

```
{ "Fn::Base64Decode" : "content" }
```

表 3-41 参数说明

参数名称	类型	说明
content	String	待解码的字符串。
返回值	byte[]	base64解码后的字节数组。

示例如下：

```
{
    "Fn::Base64Decode": "123456"]
}
return: d76df8e7 //为了方便展示，此处转化为16进制字符串
```

Fn::Base64Encode

内部函数Fn::Base64Encode将一个字符串按BASE64编码。

JSON

```
{"Fn::Base64Encode": "content"}
```

表 3-42 参数说明

参数名称	类型	说明
content	String	待编码的字符串。
返回值	String	base64编码后的字符串。

示例如下：

```
{
    "Fn::Base64Encode": "testvalue"
}
return: "dGVzdHZhbHVl"
```

Fn::GetBytes

内部函数Fn::GetBytes返回一个字符串UTF-8编码的字节数组。

JSON

```
{"Fn::GetBytes": "content"}
```

表 3-43 参数说明

参数名称	类型	说明
content	String	待编码的字符串。
返回值	byte[]	字符串UTF-8编码后的字节数组。

示例如下：

```
{
    "Fn::GetBytes": "testvalue"
}
return: "7465737476616c7565" //为了方便展示，此处转化为16进制字符串
```

Fn::HmacSHA256

内部函数Fn::HmacSHA256将一个字符串按给定密钥进行HmacSHA256算法加密。

JSON

```
{"Fn::HmacSHA256": ["content", "secret"]}
```

表 3-44 参数说明

参数名称	类型	说明
content	String	待加密的字符串。
secret	String 或 byte[]	加密密钥，可以是字符串或者字节数组类型
返回值	String	使用HmacSHA256算法加密后的值。

示例如下：

```
{
    "Fn::HmacSHA256": ["testvalue", "123456"]
}
return: "0f9fb47bd47449b6ffac1be951a5c18a7eff694940b1a075b973ff9054a08be3"
```

Fn::Join

内部函数Fn::Join可将多个字符串（数量最大值为10）拼接成一个字符串。

JSON

```
{"Fn::Join": ["element", "element"...]}
```

表 3-45 参数说明

参数名称	类型	说明
element	String	需拼接的字符串。
返回值	String	子字符串拼接在一起后的字符串。

示例如下：

```
{
   "Fn::Join": ["123", "456", "789"]
}
return: "123456789"
```

Fn::MathAdd

内部函数Fn::MathAdd将两个整数进行数学加法运算。

JSON

```
{"Fn::MathAdd": [X, Y]}
```

表 3-46 参数说明

参数名称	类型	说明
X	long	加数。
Y	long	加数。
返回值	long	和，X+Y后的值。

示例如下：

```
{
   "Fn::MathAdd": [1, 1]
}
return: 2
```

Fn::MathDiv

内部函数Fn::MathDiv将两个整数进行数学除法运算。

JSON

```
{"Fn::MathDiv": [X, Y]}
```

表 3-47 参数说明

参数名称	类型	说明
X	long	被除数。
Y	long	除数。
返回值	long	X 除Y后的值。

示例如下：

```
{
    "Fn::MathDiv": [10, 2]
}
return: 5

{
    "Fn::MathDiv": [10, 3]
}
return: 3
```

Fn::MathMod

内部函数Fn::MathMod将两个整数进行数学取余运算。

JSON

```
{"Fn::MathMod": [X, Y]}
```

表 3-48 参数说明

参数名称	类型	说明
X	long	被取余数。
Y	long	取余数。
返回值	long	X 取余 Y后的值。

示例如下：

```
{
    "Fn::MathMod": [10, 3]
}
return: 1
```

Fn::MathMultiply

内部函数Fn::MathMultiply将两个整数进行数学乘法运算。

JSON

```
{"Fn::MathMultiply": [X, Y]}
```

表 3-49 参数说明

参数名称	类型	说明
X	long	乘数。
Y	long	乘数。
返回值	long	X 乘以 Y后的值。

示例如下：

```
{
    "Fn::MathMultiply": [3, 3]
}
return: 9
```

Fn::MathSub

内部函数Fn::MathSub将两个整数进行数学减法运算。

JSON

```
{"Fn::MathSub": [X, Y]}
```

表 3-50 参数说明

参数名称	类型	说明
X	long	被减数。
Y	long	减数。
返回值	long	X 减 Y后的值。

示例如下：

```
{
    "Fn::MathSub": [9, 3]
}
return: 6
```

Fn::ParseLong

内部函数Fn::ParseLong可将一个数字字符串转化为整数。

JSON

```
{"Fn::ParseLong": "String"}
```

表 3-51 参数说明

参数名称	类型	说明
String	String	待转换的字符串。

参数名称	类型	说明
返回值	long	字符串转换为数字后的值。

示例如下：

```
{
    "Fn::ParseLong": "123"
}
return: 123
```

Fn::Split

内部函数Fn::Split将一个字符串按指定的分隔符分割成字符串数组。

JSON

```
{ "Fn::Split" : ["String", "Separator"] }
```

表 3-52 参数说明

参数名称	类型	说明
String	String	被分割的字符串。
Separator	String	分隔符。
返回值	String[]	原始参数String被分隔符Separator拆分后的字符串数组。

示例如下：

```
{
    "Fn::Split": ["a|b|c", "|"]
}
return: ["a", "b", "c"]
```

Fn::SplitSelect

内部函数Fn::SplitSelect将一个字符串按指定的分隔符分割成字符串数组，然后返回数组指定索引的元素。

JSON

```
{ "Fn::SplitSelect" : ["String", "Separator", index] }
```

表 3-53 参数说明

参数名称	类型	说明
String	String	被分割的字符串。
Separator	String	分隔符。

参数名称	类型	说明
index	int	返回元素在数组中的索引值，从0开始。
返回值	String	字符串按特定分隔符分割后指定索引的子字符串。

示例如下：

```
{
   "Fn::SplitSelect": ["a|b|c", "|", 1]
}
return: "b"
```

Fn::Sub

内部函数Fn::Sub将输入字符串中的变量替换为指定的值。在模板中您可以使用此函数来构造一个动态的字符串。

JSON

```
{ "Fn::Sub" : [ "String", { "Var1Name": Var1Value, "Var2Name": Var2Value } ] }
```

表 3-54 参数说明

参数名称	类型	说明
String	String	一个包含变量的字符串，变量使用“${}”占位符定义。
VarName	String	变量名称，必须在参数“String”中定义。
VarValue	String	变量的取值，支持函数嵌套。
返回值	String	返回原始“String”参数字符串变量替换后的值。

示例如下：

```
{
   "Fn::Sub": ["${token};hmacsha256", {
      "token": {
         "Fn::HmacSHA256": ["${iotda::mqtt::username}", {
            "Fn::Base64Decode": "${iotda::mqtt::client_id}"
         }]
      }
   }]
}
当变量
${iotda::mqtt::username}="test_device_username"
${iotda::device::client_id}="OozqTPlCWTTJjEH/5s+T6w=="
return：  "0773c4fd6c92902a1b2f4a45fdcdec416b6fc2bc6585200b496e460e2ef31c3d"
```

Fn::SubStringAfter

内部函数Fn::SubStringAfter截取字符串指定分隔符后的子字符串。

JSON

```
{ "Fn::SubStringAfter" : ["content", "separator"] }
```

表 3-55 参数说明

参数名称	类型	说明
content	String	待截取的字符串。
separator	String	分隔符。
返回值	String	字符串被指定分隔符分割后的子字符串。

示例如下：

```
{
    "Fn::SubStringAfter": ["content:123456", ":"]
]
return: "123456"
```

Fn::SubStringBefore

内部函数Fn::SubStringBefore截取字符串指定分隔符前的子字符串。

JSON

```
{ "Fn::SubStringBefore" : ["content", "separator"] }
```

表 3-56 参数说明

参数名称	类型	说明
content	String	待截取的字符串。
separator	String	分隔符。
返回值	String	字符串被指定分隔符分割前的子字符串。

示例如下：

```
{
    "Fn::SubStringBefore": ["content:123456", ":"]
]
return: "content"
```

Fn::ToLowerCase

内部函数Fn::ToLowerCase将指定字符串转的所有字符转化成小写。

JSON

```
{ "Fn::ToLowerCase" : content }
```

表 3-57 参数说明

参数名称	类型	说明
content	String	待转换的字符串。
返回值	String	字符串被转为小写后的值。

示例如下：

```
{
    "Fn::ToLowerCase": "ABC"
]
return: "abc"
```

Fn::ToUpperCase

内部函数Fn::ToUpperCase将字符串转大写函数。

JSON

```
{ "Fn::ToUpperCase" : content }
```

表 3-58 参数说明

参数名称	类型	说明
content	String	待转换的字符串。
返回值	String	字符串被转为大写后的值。

示例如下：

```
{
    "Fn::ToUpperCase": "abc"
]
return: "ABC"
```

Ref

内部函数Ref将返回指定引用参数的值，引用参数必须在模板中有声明。

JSON

```
{ "Ref" : "paramName" }
```

表 3-59 参数说明

参数名称	类型	说明
paramName	String	引用的参数名称。

参数名称	类型	说明
返回值	String	引用参数对应的值。

示例如下：

```
{
    "Ref": "iotda::mqtt::username"
}
当参数iotda::mqtt::username="device_123"
return: "device_123"
```

3.6 HTTP(S)协议接入

概述

HTTPS是基于HTTP协议，通过SSL加密的一种安全通信协议。IoTDA物联网平台支持HTTPS协议通信。该协议多使用在需要进行数据采集和分析的场景中，因为HTTP协议能够高效地传输和处理结构化数据；或应用在设备需要采用短连接且仅需单向上传数据的场景。

HTTPS协议鉴权是指设备通过调用HTTPS协议设备鉴权接口并携带设备ID和使用算法加密后的密钥，以完成设备的接入鉴权。鉴权成功后可以建立设备与平台间的连接，并且平台会返回用于业务处理的access_token。

使用限制

- 在调用属性上报、消息上报等其他HTTPS协议接口时，都需要携带access_token信息。
- 如果access_token超期，需要重新认证设备获取access_token。
- 如果access_token未超期重复获取access_token，原access_token在未超期前保留30s，30s之后失效。

表 3-60 HTTP(S)协议接入使用限制

描述	限制
支持的HTTP协议版本	支持 Hypertext Transfer Protocol — HTTP/1.0 协议 支持 Hypertext Transfer Protocol — HTTP/1.1 协议
支持HTTPS协议	物联网平台仅支持HTTPS协议，证书下载请参考**证书资源**。
支持的TLS版本	TLS 1.2
支持的body体最大长度	1MB
接口规格说明	请参考**产品规格说明**。

描述	限制
网关上报子设备属性时一次最大可上报子设备数	50
数据下行	不支持

调用说明

物联网平台的Endpoint请参见：**地区和终端节点**。

说明

使用“设备接入-> HTTPS(443)”对应的Endpoint，端口为443。

使用 HTTPS 协议接入的鉴权流程

图 3-203 HTTPS 协议接入鉴权流程图

1. 通过调用注册接口向物联网平台发送注册请求或者在控制台上注册设备。
2. 物联网平台向设备分配全局唯一的设备ID （deviceId）和密钥（secret）。

   说明

   密钥可以在注册设备时自定义，如果没有定义，平台将自动分配密钥。
3. 设备登录时，调用HTTPS协议设备鉴权接口并携带设备ID和使用“HMACSHA256”算法签名后的密钥（以时间戳为key，对平台分配的密码进行签名后的值，参考**密钥生成工具**），向平台发起接入鉴权请求。
4. 平台验证通过后，返回成功响应，设备连接物联网平台成功。

接入实践

设备使用HTTPS协议接入平台时，平台和设备通过HTTPS接口调用通信。通过这些接口，平台和设备可以实现设备鉴权、消息上报及属性上报。

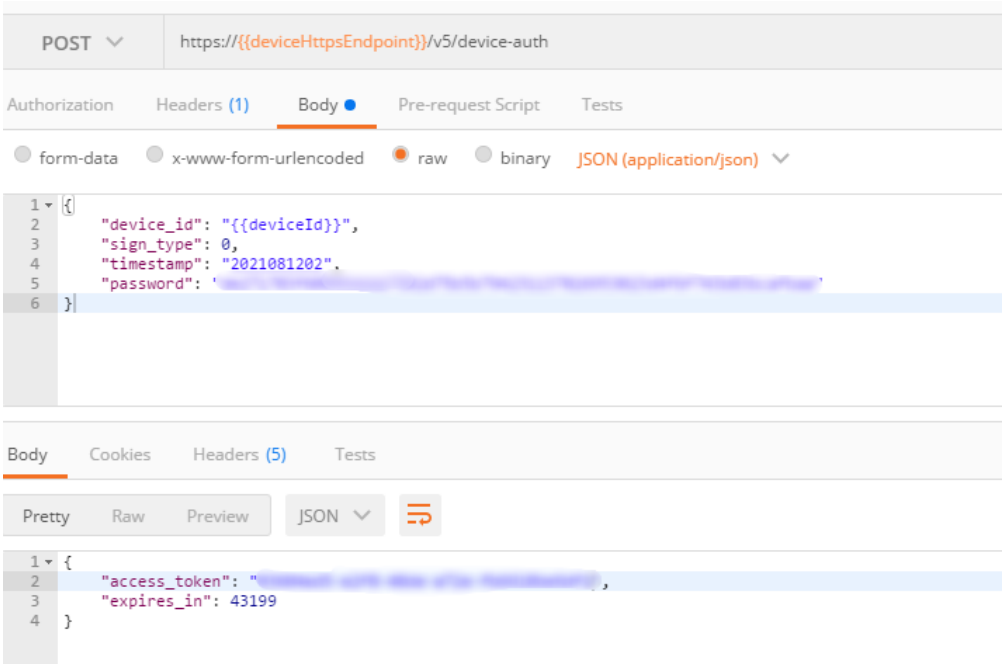
表 3-61 消息类型

消息类型	说明
设备鉴权**设备鉴权**	用于设备获取鉴权信息access_token。
设备属性上报**设备属性上报**	用于设备按产品模型中定义的格式将属性数据上报给平台。
设备消息上报**设备消息上报**	用于设备将自定义数据上报给平台，平台将设备上报的消息转发给应用服务器或华为云其他云服务上进行存储和处理。

消息类型	说明
网关批量属性上报**网关批量属性上报**	用于网关设备将多个子设备的属性数据一次性上报给平台。

1. 创建产品（可通过控制台创建或者使用应用侧API**创建产品**）。
2. 产品创建完毕后，注册设备（可通过控制台**注册单个设备**或者使用应用侧API**注册设备**创建）。
3. 设备注册完毕后，通过设备鉴权接口**设备鉴权**获取设备的access_token。

图 3-204 获取设备 access_token



4. 获取到access_token之后，可以使用消息上报**设备消息上报**/属性上报**设备属性上报**等功能。其中access_token放于消息头中，下面示例为上报属性：

图 3-205 上报属性

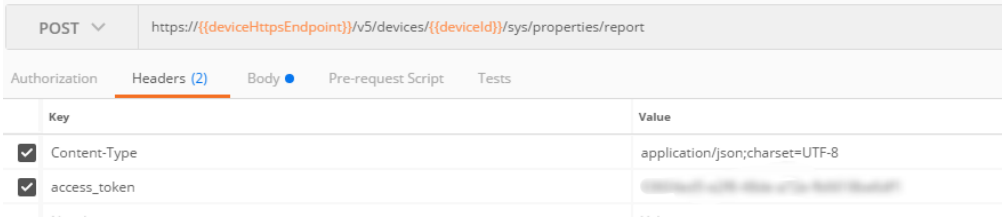
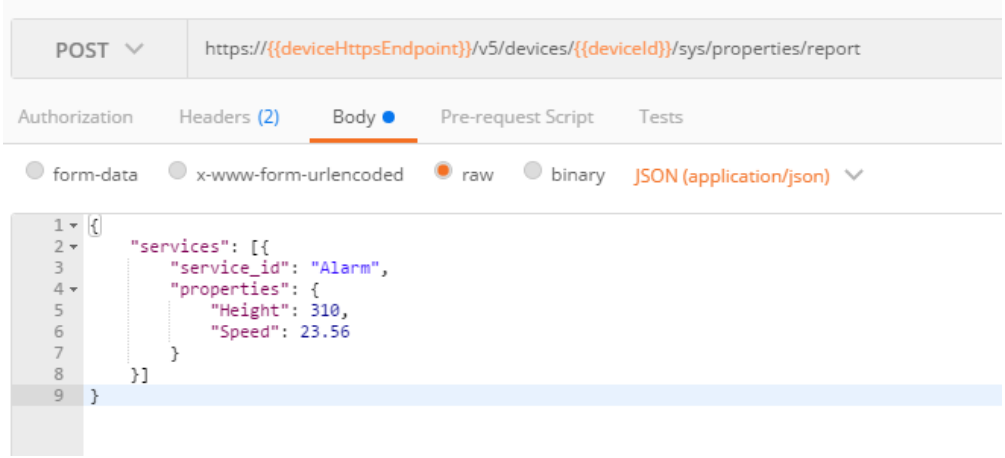


图 3-206 上报属性



3.7 LwM2M/CoAP 协议接入

概述

LwM2M（Lightweight M2M，轻量级M2M），由开发移动联盟（OMA）提出，是一种轻量级的、标准通用的物联网设备管理协议，可用于快速部署客户端/服务器模式的物联网业务。LwM2M为物联网设备的管理和应用建立了一套标准，它提供了轻便小巧的安全通信接口及高效的数据模型，以实现M2M设备管理和服务支持。

LwM2M/CoAP协议鉴权支持加密与非加密两种接入方式，若设备采用非加密方式接入时，非加密端口为5683，在设备接入物联网平台时携带设备唯一标识nodeId，完成设备的接入鉴权；当设备采用加密方式接入时，加密业务数据交互端口为5684，使用DTLS/DTLS+传输层安全协议通道接入，并携带nodeId和密钥以完成设备的接入鉴权。

物联网平台从安全角度考虑，强烈建议采用安全接入方式。

说明

LwM2M的语法和接口细节，请以此**标准规范**为准。

物联网平台支持协议规定的plain text, opaque, Core Link ,TLV , JSON编码格式。在多字段操作时（比如写多个资源），默认用TLV格式。

使用限制

表 3-62 LwM2M/CoAP 协议接入使用限制

描述	限制
支持的LwM2M协议版本	1.1
支持的DTLS版本	DTLS 1.2
支持的加密算法套件	TLS_PSK_WITH_AES_128_CCM_8， TLS_PSK_WITH_AES_128_CBC_SHA256
支持的body体最大长度	1KB

描述	限制
接口规格说明	请参考**产品规格说明**。

调用说明

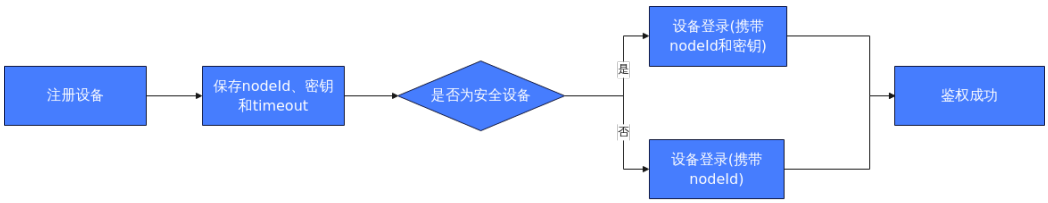
物联网平台的Endpoint请参见：**地区和终端节点**。

说明

使用“设备接入-> CoAP (5683)| CoAPS (5684)”对应的Endpoint，端口为5683（非加密接入方式）或者5684（加密接入方式）。

使用 LwM2M/CoAP 协议接入的鉴权流程

图 3-207 LwM2M/CoAP 协议接入鉴权流程图



1. 通过调用注册接口向物联网平台发送注册请求或者在控制台上注册设备。
2. 物联网平台向设备分配密钥，返回timeout。

   说明

   - 密钥可以在注册设备时自定义，如果没有定义，平台将自动分配预置密钥。
   - timeout是指超时时间，若设备在有效时间未接入物联网平台，则平台会删除该设备的注册信息。
3. 设备登录时，安全设备携带设备唯一标识码nodeId（如IMEI）和密钥发起接入鉴权请求；非安全设备携带设备唯一标识码nodeId发起接入鉴权请求。
4. 平台验证通过后，返回成功响应，设备连接物联网平台成功。

LWM2M/CoAP 协议设备接入开发流程

1. 平台侧开发：包括创建产品、开发产品模型、开发编解码插件、注册设备。详细操作指导请参考**创建产品**、**开发产品模型**、**开发编解码插件**、**注册单个设备**。
2. 设备侧开发：使用模组、设备侧Tiny SDK接入。详细操作指导请参考**IoT Device SDK Tiny使用指南（C）**

相关最佳实践

基于NB-IoT小熊派开发智慧路灯

相关 FAQ

LWM2M/CoAP协议接入热点咨询问题如下：

- 如何检测NB网络信号?
- NB模组附着网络失败如何处理?
- NB模组绑定设备失败怎么办?
- NB模组无法正常上报数据怎么办?
- NB设备接入时，出现513错误?
- 同一个NB卡，使用一个设备能上报，在另一个设备里无法上报?

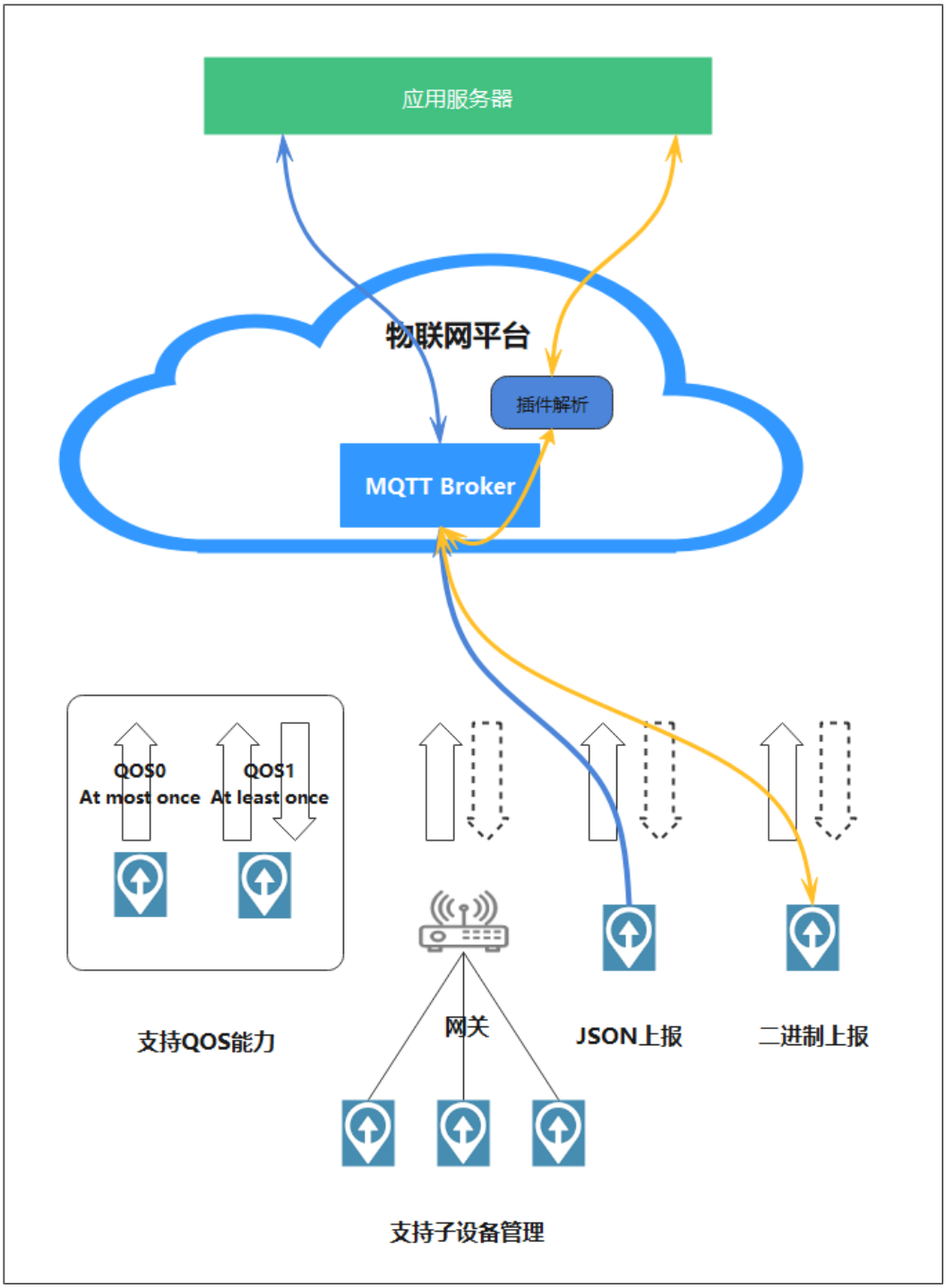
3.8 使用 MQTT Demo 接入

3.8.1 MQTT 使用指导

概述

MQTT（Message Queuing Telemetry Transport）是一个基于客户端-服务器的消息发布/订阅传输协议，主要应用于计算能力有限且工作在低带宽、不可靠的网络的远程传感器和控制设备，适合设备与云端保持长连接的场景，如智能路灯等。MQTT 客户端通过主题（Topic）发布或订阅消息，由MQTT Broker集中管理消息路由并根据预设的服务质量（QoS）确保端到端消息传递的可靠性。在此过程中，发送消息的客户端（发布者）和接收消息的客户端（订阅者）之间被解耦，发布者和订阅者之间无需建立直接连接。由于MQTT协议满足物联网对通信协议的轻量级、可靠、双向和大规模的需求，MQTT协议成为了物联网领域的最佳协议之一。更多关于MQTT协议语法及接口信息，请访问这里获取。

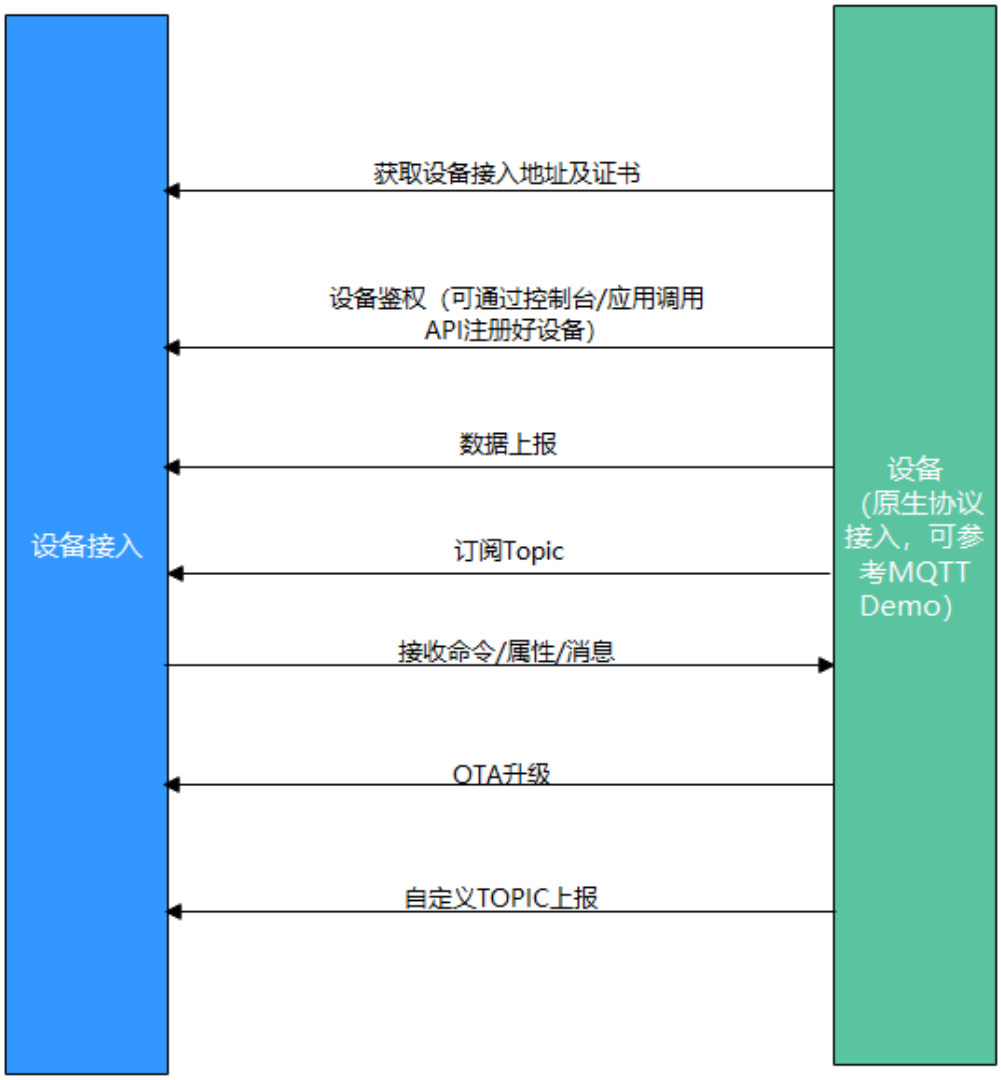
MQTTS是MQTT使用TLS加密的协议。采用MQTTS协议接入平台的设备，设备与物联网平台之间的通信过程，数据都是加密的，具有一定的安全性。



业务流程

采用MQTT协议接入物联网平台的设备，设备与物联网平台之间的通信过程，数据没有加密，建议使用MQTTS协议。

若选择MQTTS协议接入平台，建议通过**使用IoT Device SDK接入**。



1. 设备接入前，需创建产品（可通过控制台创建或者使用应用侧API**创建产品**）。
2. 产品创建完毕后，需注册设备（可通过控制台**注册单个设备**或者使用应用侧API**注册设备**创建）。
3. 设备注册完毕后，可以按照图中流程实现消息/属性上报、接收命令/属性/消息、OTA升级、自定义Topic等功能。关于平台预置Topic可参考**Topic定义**

说明

您可以通过mqtt.fx进行原生协议接入调测，可以参考**快速体验mqtt接入**。

使用限制

描述	限制
单个MQTT直连设备在同一时间的连接数	1
单账户设备侧每秒最大建链请求数量	• 基础版100 • 标准版请参考**标准版规格**

描述	限制
单账号设备侧每秒最大上行的请求数量（单消息payload平均为512字节）	• 基础版500 • 标准版请参考**标准版规格**
单个MQTT连接每秒最大上行消息数量	50/s
单个MQTT连接最大带宽（上行消息）	1MB（默认）
MQTT单条发布消息最大长度。超过此大小的发布请求将被直接拒绝。	1MB
MQTT协议规范	MQTT v5.0、MQTT v3.1.1、MQTT v3.1
与标准MQTT协议的区别	• 不支持QoS2 • 不支持will、retain msg
MQTT协议支持的安全等级	采用TCP通道基础 + TLS协议（TLSv1、 TLSv1.1、TLSv1.2和TLSv1.3版本）
MQTT连接心跳时间建议值	心跳时间限定为30至1200秒，推荐设置为120秒
MQTT协议消息发布与订阅	设备只能对自己的Topic进行消息发布与订阅
单个MQTT连接的最大订阅数量。	100个
MQTT自定义Topic支持的最大长度	128字节
MQTT自定义Topic支持每个产品添加的最大个数	10个/产品
单账号支持上传设备侧CA证书个数	100个

MQTT 设备与物联网平台通信

设备使用MQTT协议接入平台时，平台和设备通过Topic进行通信。物联网平台预置了Topic，通过这些预置的Topic，平台和设备可以实现消息、属性、命令的交互。您还可以在设备接入控制台，自定义Topic，实现设备平台通信的个性化配置。

数据类型	消息类型	说明
数据上行	设备属性上报	用于设备按产品模型中定义的格式将属性数据上报给平台。
	设备消息上报	设备无法按照产品模型中定义的属性格式进行数据上报时，将设备的自定义数据通过设备消息上报接口上报给平台，平台将设备上报的消息转发给应用服务器或华为云其他云服务上进行存储和处理。

数据类型	消息类型	说明
	网关批量属性上报	用于网关设备将多个设备的属性数据一次性上报给平台。
	设备事件上报	用于设备按产品模型中定义的格式将事件数据上报给平台。
数据下行	平台消息下发	用于平台下发自定义格式的数据给设备。
	平台设置设备属性	设备的产品模型中定义了平台可向设备设置的属性，平台/应用服务器可通过属性设置的方式修改指定设备的属性值。
	平台查询设备属性	平台/应用服务器通过属性查询的方式，实时查询指定设备的属性数据。
	平台命令下发	平台/应用服务器按产品模型中定义的命令格式下发控制命令给设备。
	平台事件下发	平台/应用服务器按产品模型中定义的事件格式下发事件给设备。

Topic接口介绍

物联网平台预置的Topic如下表所示：

Topic分类	用途	Topic	Publisher（发布者）	Subscriber（订阅者）
设备消息相关Topic	设备消息上报	$oc/devices/{device_id}/sys/messages/up	设备	平台
	平台下发消息给设备	$oc/devices/{device_id}/sys/messages/down	平台	设备
设备命令相关Topic	平台下发命令给设备	$oc/devices/{device_id}/sys/commands/request_id={request_id}	平台	设备
	设备返回命令响应	$oc/devices/{device_id}/sys/commands/response/request_id={request_id}	设备	平台
设备属性相关Topic	设备上报属性数据	$oc/devices/{device_id}/sys/properties/report	设备	平台
	网关批量上报属性数据	$oc/devices/{device_id}/sys/gateway/sub_devices/properties/report	设备	平台

Topic分类	用途	Topic	Publisher（发布者）	Subscriber（订阅者）
	平台设置设备属性	$oc/devices/{device_id}/sys/properties/set/request_id={request_id}	平台	设备
	属性设置的响应结果	$oc/devices/{device_id}/sys/properties/set/response/request_id={request_id}	设备	平台
	平台查询设备属性	$oc/devices/{device_id}/sys/properties/get/request_id={request_id}	平台	设备
	属性查询响应结果，这个结果不会对设备属性和影子产生影响	$oc/devices/{device_id}/sys/properties/get/response/request_id={request_id}	设备	平台
	设备侧主动获取平台的设备影子数据	$oc/devices/{device_id}/sys/shadow/get/request_id={request_id}	设备	平台
	设备侧主动获取平台设备影子数据的响应	$oc/devices/{device_id}/sys/shadow/get/response/request_id={request_id}	平台	设备
设备事件相关Topic	设备事件上报	$oc/devices/{device_id}/sys/events/up	设备	平台
	平台事件下发	$oc/devices/{device_id}/sys/events/down	平台	设备

另外，用户还可以通过在控制台上设置自定义Topic，上报用户个性化的数据。具体可参考自定义Topic。

MQTT 的 TLS 支持

平台推荐使用TLS来保护设备和平台的传输安全。 平台目前支持TLS1.3、1.2 、1.1版本以及GMTLS。其中TLS 1.1 计划后续不再支持，建议使用 TLS 1.3作为首选 TLS 版本。GMTLS版本仅在国密算法的企业版才支持。

基础版，标准版以及支持通用加密算法的企业版使用TLS连接时，平台支持如下加密套件：

- TLS_AES_256_GCM_SHA384
- TLS_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256

- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA

支持国密算法的企业版使用TLS连接时平台支持如下加密套件：

- ECC_SM4_GCM_SM3
- ECC_SM4_CBC_SM3
- ECDHE_SM4_GCM_SM3
- ECDHE_SM4_CBC_SM3
- TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256

说明

带CBC的加密套件存在安全风险，请谨慎使用。

相关 FAQ

MQTT接入相关问题

3.8.2 Java Demo 使用说明

概述

本文以Java语言为例，介绍通过MQTTS/MQTT协议接入平台，基于平台接口实现“属性上报”、“订阅接收命令”等功能。

说明

本文中使用的代码为样例代码，仅用于体验平台通信功能，如需进行商用，可以参考资源获取获取对应语言的IoT Device SDK进行集成。

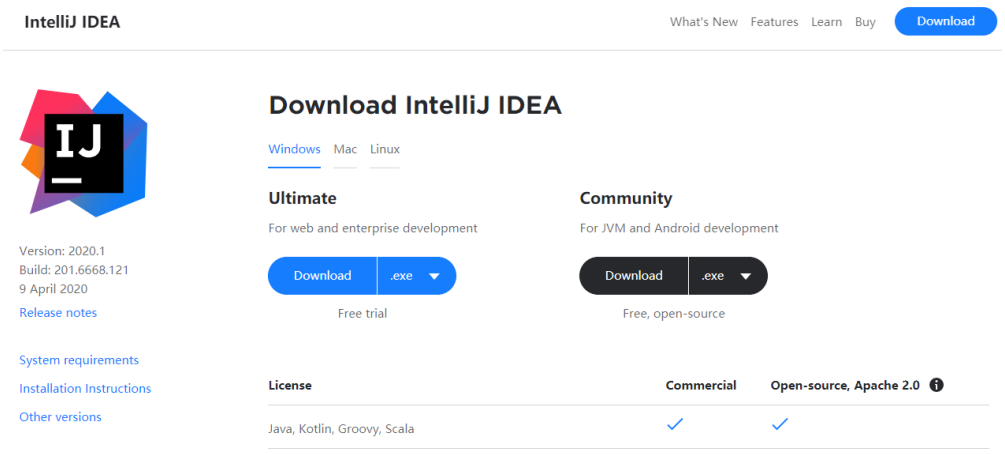
前提条件

- 已在管理控制台获取设备接入地址。获取地址的操作步骤，请参考平台对接信息。
- 已在管理控制台创建产品和设备。创建产品和设备的具体操作细节，请参考创建产品、注册单个设备或批量注册设备。

准备工作

安装**IntelliJ IDEA**

1. 访问IntelliJ IDEA官网，选择合适系统的版本下载。（本文以windows 64-bit系统IntelliJ IDEA 2019.2.3 Ultimate为例）。

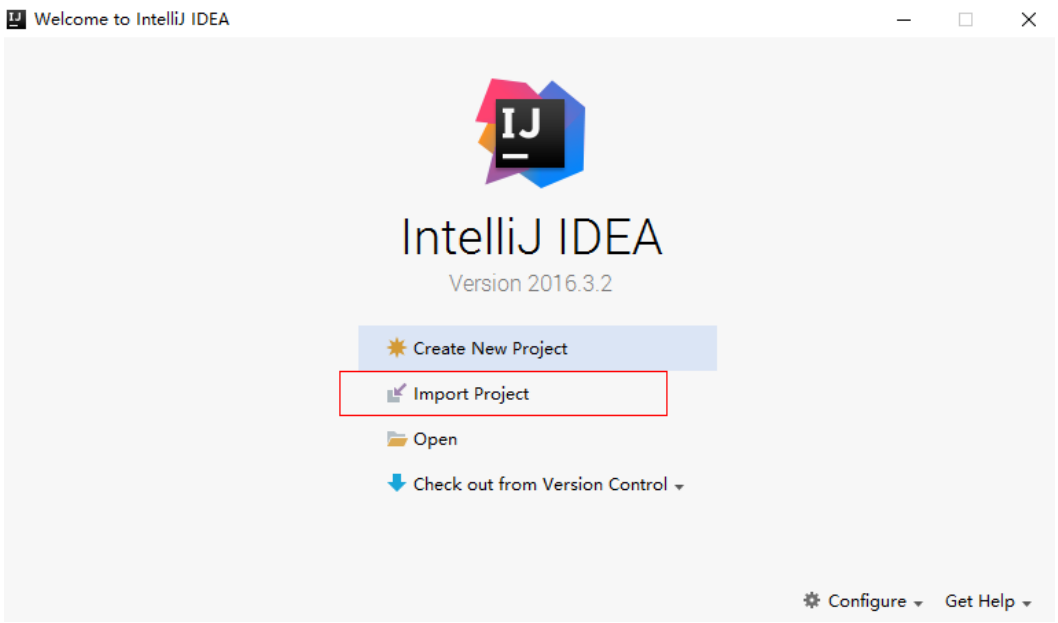


2. 下载完成后，运行安装文件，根据界面提示安装。

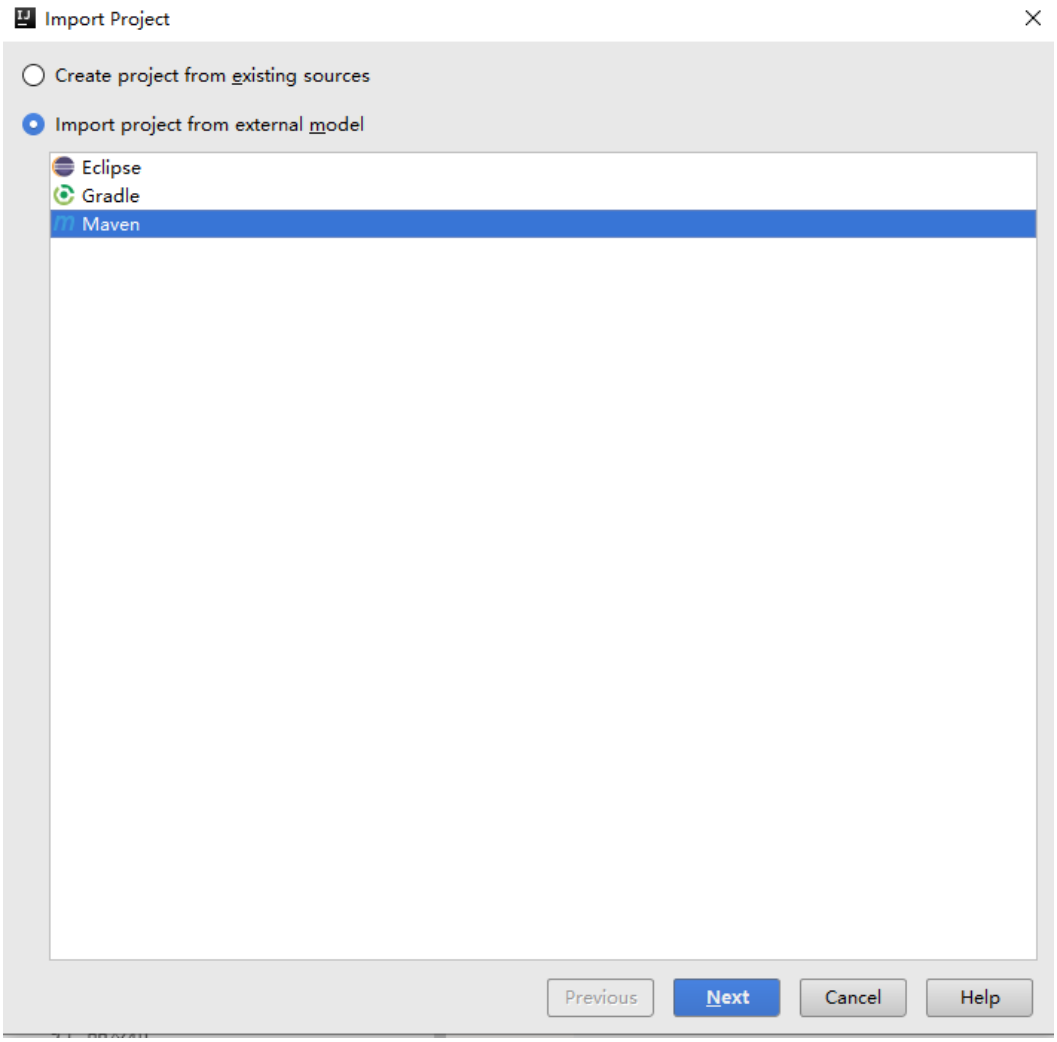
导入代码样例

步骤1 下载**JAVA样例**。

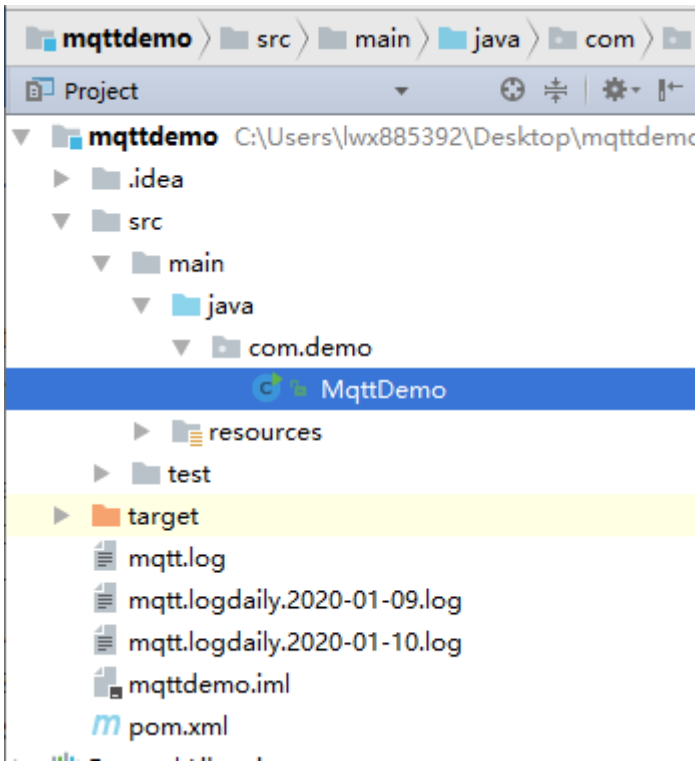
步骤2 打开IDEA开发者工具，单击“ Import Project”。



步骤3 选择步骤1中下载的样例，然后根据界面提示，单击“next”。



步骤4 完成代码导入。



----结束

建立连接

设备或网关在接入物联网平台时首先需要和平台建立连接，从而将设备或网关与平台进行关联。开发者通过传入设备信息，将设备或网关连接到物联网平台。

1. 在建立连接之前，先修改以下参数：

```
//IoT平台mqtt对接地址（要替换为设备所在的平台域名地址）
static String serverIp = "xxx.myhuaweicloud.com";
//注册设备时获得的deviceId,密钥（要替换为自己注册的设备ID与密钥）
static String deviceId = "722cb*****************";
static String secret = "******";
```

   - serverIp为物联网平台的设备对接地址，可参考**平台对接信息**获取（获取的是域名信息，可通过在cmd命令框中执行“ping 域名”，获取IP地址）。
   - deviceId和secret为设备ID和密钥，在成功**注册设备**后获取。

2. 修改完**1**中的参数后就可使用MqttClient建立连接了。mqtt连接心跳时间的建议值是120秒，有**使用限制**。

```
MqttConnectOptions options = new MqttConnectOptions();
options.setCleanSession(false);
options.setKeepAliveInterval(120);  //心跳时间限定为30至1200秒
options.setConnectionTimeout(5000);
options.setAutomaticReconnect(true);
options.setUserName(deviceId);
options.setPassword(getPassword().toCharArray());
client = new MqttAsyncClient(url, getClientId(), new MemoryPersistence());
client.setCallback(callback);
```

   1883是mqtt非安全加密接入端口，8883是mqtts安全加密接入端口（使用SSL加载证书）。

```
if (isSSL) {
    url = "ssl://" + serverIp + ":" + 8883; //mqtts连接
} else {
```

```
    url = "tcp://" + serverIp + ":" + 1883; //mqtt连接
}
```

如果建立MQTTS连接，需要加载服务器端SSL证书，需要添加SocketFactory参数。DigiCertGlobalRootCA.jks在demo的resources目录下，是设备校验平台身份的证书，用于设备侧接入物联网平台登录鉴权使用，可以在资源获取中**下载证书文件**。

```
options.setSocketFactory(getOptionSocketFactory(MqttDemo.class.getClassLoader().getResource("DigiCertGlobalRootCA.jks").getPath()));
```

3. 调用client.connect(options, null, new IMqttActionListener())发起连接。连接时，需要向函数传入MqttConnectOptions连接参数。

```
client.connect(options, null, new IMqttActionListener()
```

4. 在创建MqttConnectOptions连接参数时，调用options.setPassword()传入的密码会做一个加密。getPassword()为获取加密后的密钥。

```
public static String getPassword() {
    return sha256_mac(secret, getTimeStamp());
}
/* 调用sha256算法进行哈希 */
public static String sha256_mac(String message, String tStamp) {
    String passWord = null;
    try {
        Mac sha256_HMAC = Mac.getInstance("HmacSHA256");
        SecretKeySpec secret_key = new SecretKeySpec(tStamp.getBytes(), "HmacSHA256");
        sha256_HMAC.init(secret_key);byte[] bytes = sha256_HMAC.doFinal(message.getBytes());
        passWord = byteArrayToHexString(bytes);
    } catch (Exception e) {
        e.printStackTrace();
    }
    return passWord;
```

5. 连接成功后，设备显示在线。

图 3-208 设备列表-设备在线

注：如果连接失败，在onFailure函数中已实现退避重连，代码样例如下：

```
@Override
public void onFailure(IMqttToken iMqttToken, Throwable throwable) {
    System.out.println("Mqtt connect fail.");

    //退避重连
    int lowBound = (int) (defaultBackoff * 0.8);
    int highBound = (int) (defaultBackoff * 1.2);
    long randomBackOff = random.nextInt(highBound - lowBound);
    long backOffWithJitter = (int) (Math.pow(2.0, (double) retryTimes)) * (randomBackOff + lowBound);
    long waitTImeUntilNextRetry = (int) (minBackoff + backOffWithJitter) > maxBackoff ? maxBackoff : (minBackoff + backOffWithJitter);
    System.out.println("----  " + waitTImeUntilNextRetry);
    try {
        Thread.sleep(waitTImeUntilNextRetry);
        } catch (InterruptedException e) {
            System.out.println("sleep failed, the reason is" + e.getMessage().toString());
        }
        retryTimes++;
        MqttDemo.this.connect(true);
}
```

订阅接收命令

订阅某Topic的设备才能接收broker发布的关于该Topic的消息，关于平台预置Topic可参考**Topic定义**。详细接口信息请参考**命令下发**。

```
//订阅接收命令
client.subscribe(getCmdRequestTopic(), qosLevel, null, new IMqttActionListener();
```

getCmdRequestTopic()获取接收命令的Topic，向平台订阅该Topic的命令。

```
public static String getCmdRequestTopic() {
    return "$oc/devices/" + deviceId + "/sys/commands/#";
}
```

属性上报

属性上报是指设备主动向平台上报自己的属性，更多信息请参考**设备属性上报**。

```
//上报json数据，注意serviceId要与产品模型中的定义对应
String jsonMsg = "{\"services\": [{\"service_id\": \"Temperature\",\"properties\": {\"value\": 57}},{\"service_id
\": \"Battery\",\"properties\": {\"level\": 80}}]}";
MqttMessage message = new MqttMessage(jsonMsg.getBytes());
client.publish(getRreportTopic(), message, qosLevel, new IMqttActionListener();
```

消息体jsonMsg组装格式为JSON，其中service_id要与产品模型中的定义对应，properties是设备的属性，57为对应的属性值。event_time为可选项，为设备采集数据UTC时间，不填写默认使用系统时间。

设备或网关成功连接到物联网平台后，即可调用MqttClient.publish(String topic,MqttMessage message)向平台上报设备属性值。

getRreportTopic()即为获取上报数据的Topic。

```
public static String getRreportTopic() {
    return "$oc/devices/" + deviceId + "/sys/properties/report";
}
```

查看上报数据

运行main方法成功启动后，即可在设备设备详情页面查看上报的设备属性数据。详细接口信息请参考**设备属性上报**。

图 3-209 查看上报数据-level

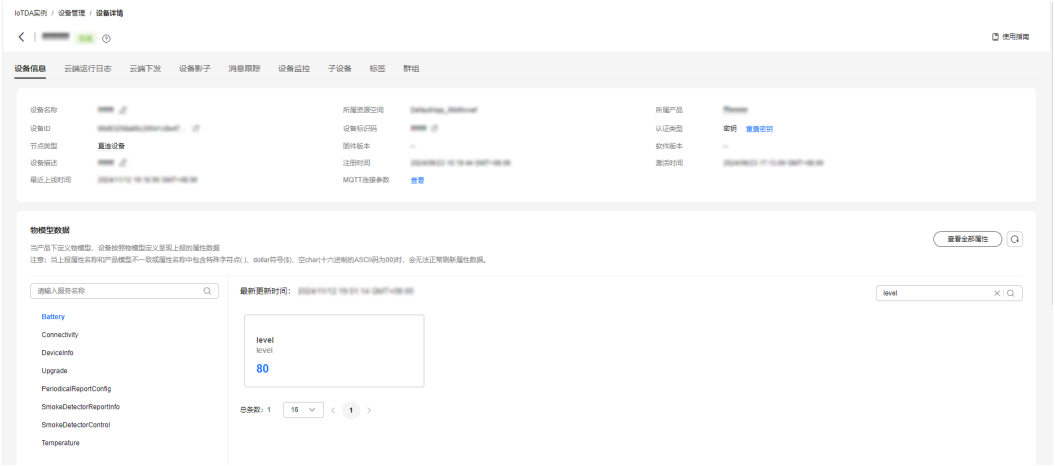


图 3-210 查看上报数据-temperature_value

说明

如果在“设备详情”页面没有最新上报数据，请确认设备上报的服务/属性和产品模型中的服务/属性一致。

相关资源

您可以参考**MQTT接口**文档，把具备MQTT通信能力的设备接入物联网平台。您也可以**在线开发MQTT协议的智慧路灯**，快速验证是否可以与物联网平台服务交互发布或订阅消息。

说明

由于是同步命令需要端侧回复响应可**参考接口**。

3.8.3 Python Demo 使用说明

概述

本文以Python语言为例，介绍通过MQTTS/MQTT协议接入平台，基于**平台接口**实现“属性上报”、“订阅接收命令”等功能。

说明

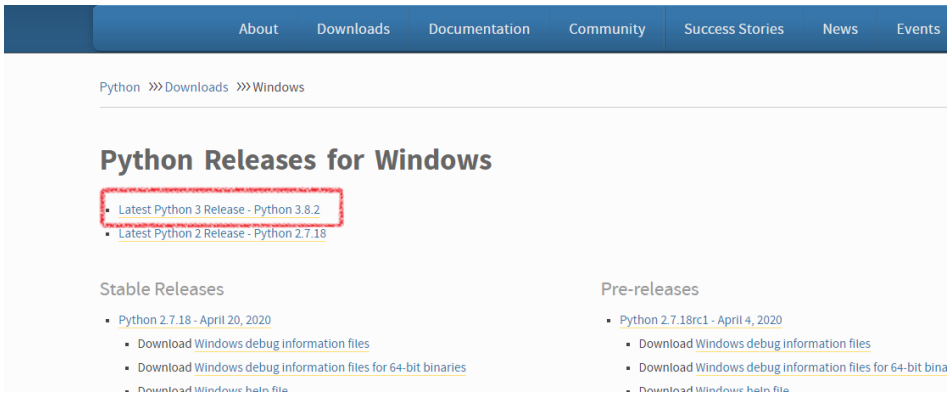
本文中使用的代码为样例代码，仅用于体验平台通信功能，如需进行商用，可以参考**资源获取**获取对应语言的IoT Device SDK进行集成。

前提条件

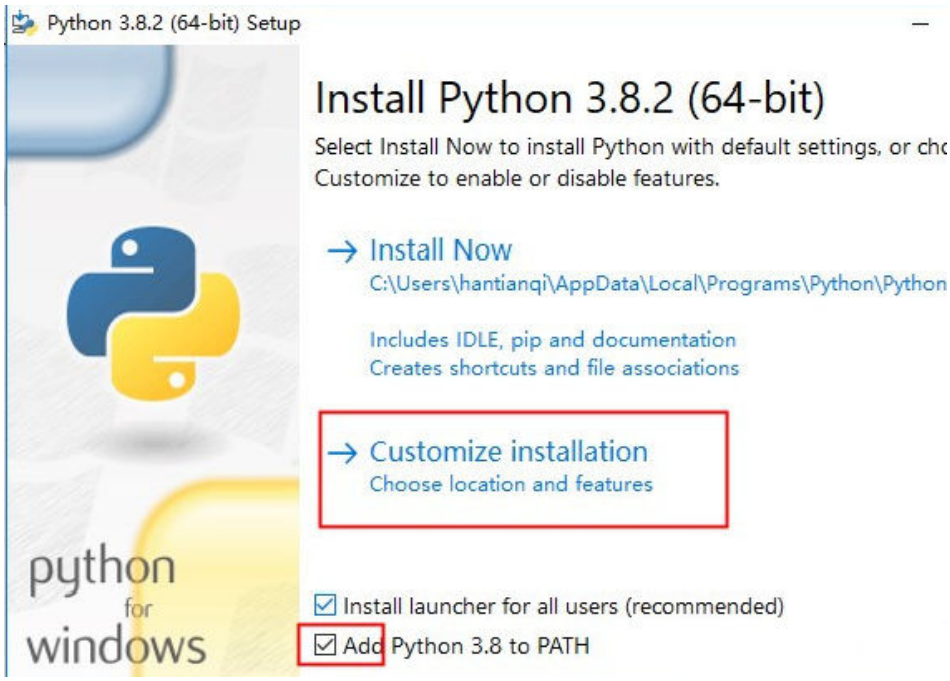
- 已安装python，若未安装请参考**安装python**。
- 已安装开发工具（本文以Pycharm为例），若未安装请参考**安装Pycharm**。
- 已在**管理控制台**获取设备接入地址。获取地址的操作步骤，请参考**平台对接信息**。
- 已在**管理控制台**创建产品和设备。创建产品和设备的具体操作细节，请参考**创建产品**、**注册单个设备**或**批量注册设备**。

准备工作

- 安装python
  a. 访问**python官网**，选择合适系统的版本下载并安装。（本文以windows系统为例，安装python3.8.2）。



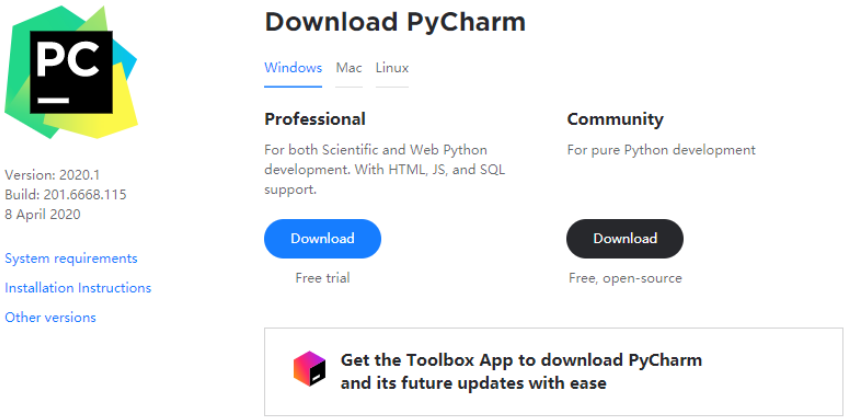
  b. 下载完成后，运行exe文件进行安装。
  c. 勾选“Add python 3.8 to PATH”（如无勾选，需手动配置环境变量），单击“Customize installation”，按照界面提示安装。



  d. 检查python是否安装成功。

     Win键 + r -->输入 cmd-->回车，进入命令行窗口，输入python –V，回车后显示python版本即表示安装成功。

- 安装Pycharm。（如已安装，请跳过此步骤）
  a. 访问**Pycharm官网**，选择合适的版本单击“Download”下载。



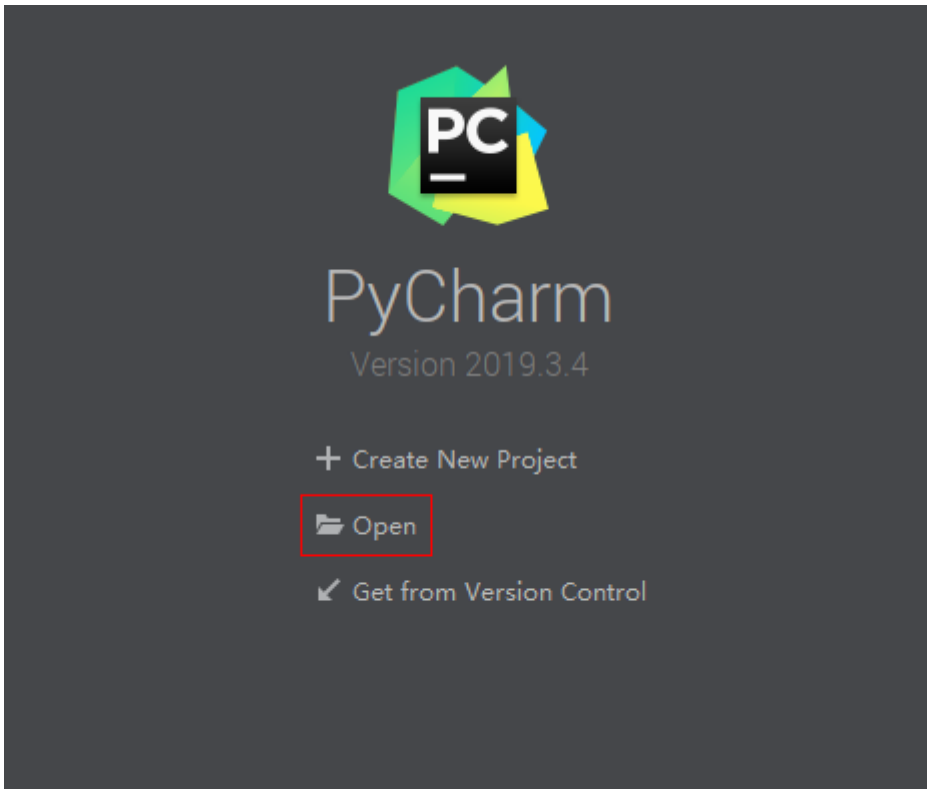
注：推荐使用专业版。

  b. 运行exe文件，按照界面提示安装。

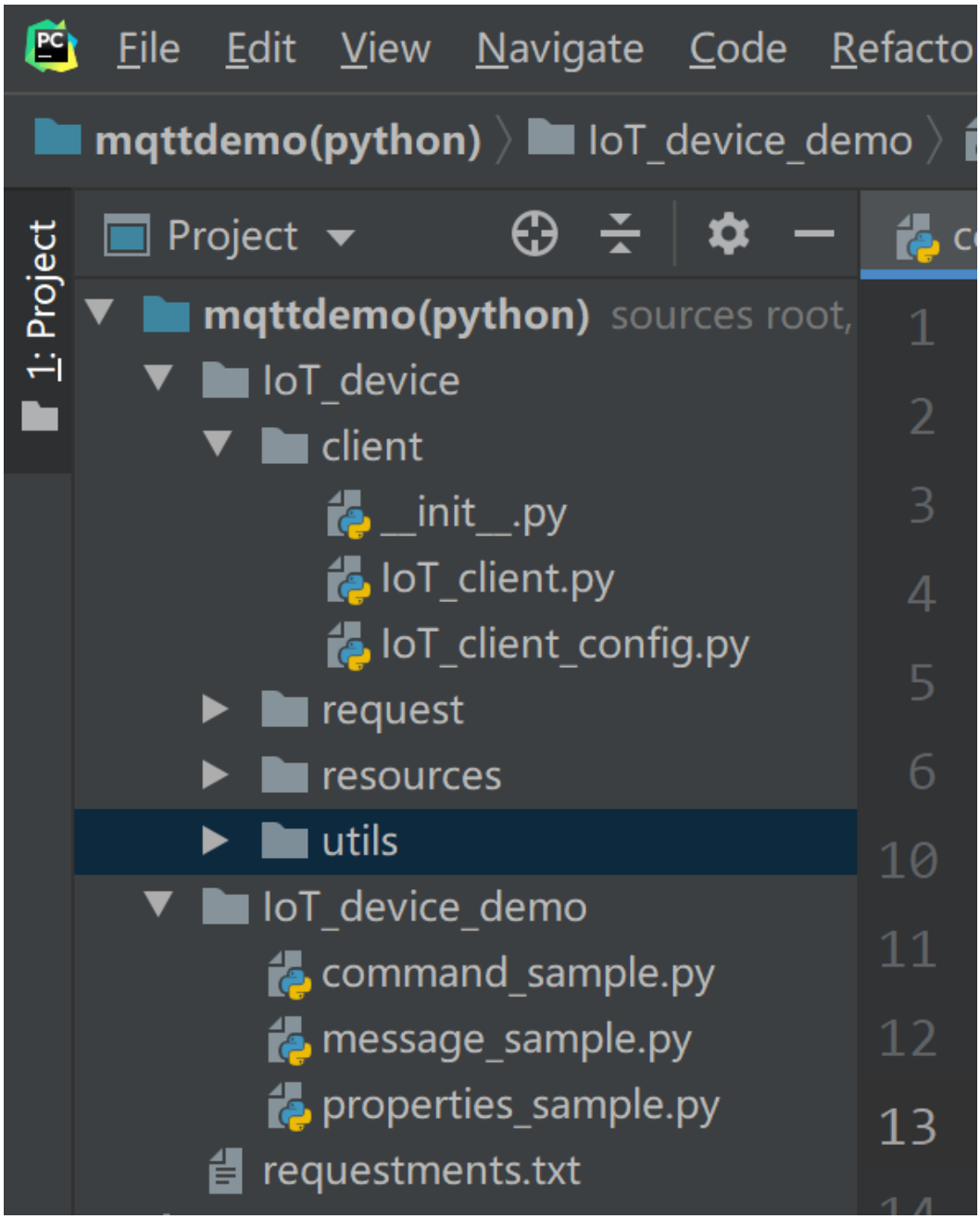
导入代码样例

步骤1 下载**QuickStart（Python）**样例。

步骤2 运行PyCharm，单击Open，选择步骤**1**中下载的样例。



步骤3 完成代码导入。



代码目录简述：

- **IoT_device_demo：**使用MQTT协议的demo文件;
  message_sample.py：设备发送消息和接收平台消息的demo；
  command_sample.py：响应平台下发命令的demo；
  properties_sample.py：属性上报等的demo；
- **IoT_device/client：**对paho-mqtt进行了封装；
  IoT_client_config.py：配置客户端信息，如设备id、密钥等；
  IoT_client.py：提供mqtt协议相关功能，如连接、订阅、发布和响应等；
- **IoT_device/Utils：**工具方法，如获取时间戳、密钥加密等；
- **IoT_device/resources：**存放证书；

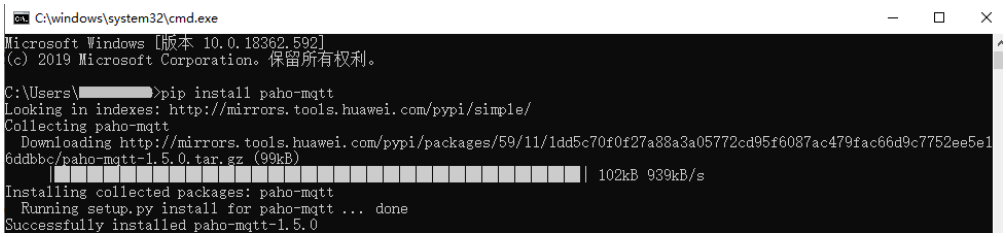
DigiCertGlobalRootCA.crt.pem：设备校验平台身份的证书，用于设备侧接入物联网平台登录鉴权使用，可以在资源获取中**下载证书文件**。

- **IoT_device/request**：对设备相关属性进行封装，如命令、消息和属性等。

步骤4 （可选）安装paho-mqtt库，paho-mqtt是python使用mqtt协议的第三方库（如已安装，可跳过）。可参考如下两种安装方式：

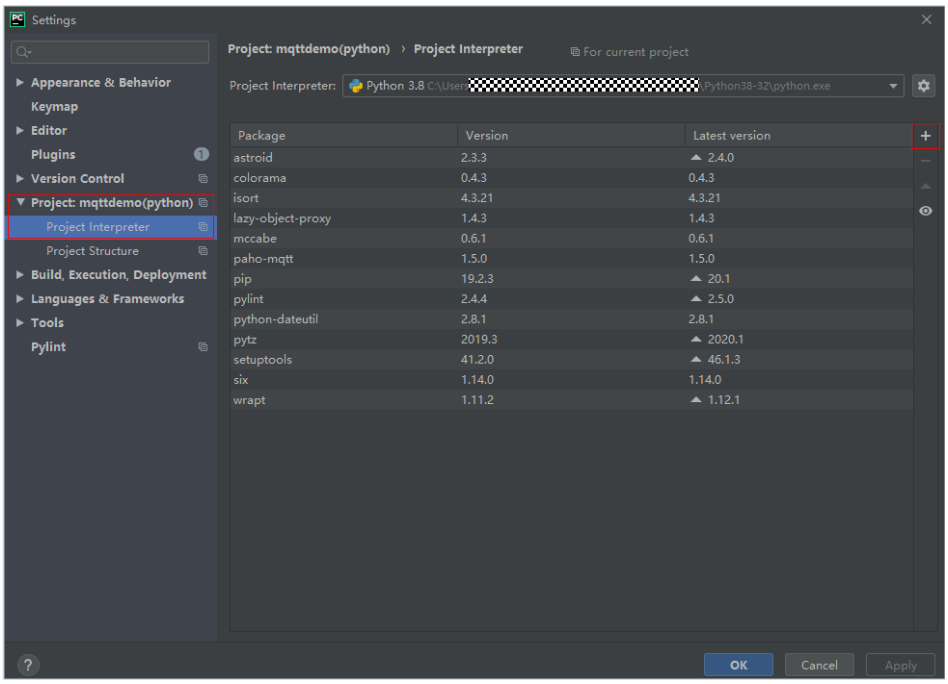
- 方法一：在命令行下采用pip工具安装（安装python时，已自带该工具）

  进入命令行界面输入命令：pip install paho-mqtt回车，提示 Successfully installed paho-mqtt 表示安装成功。（若提示pip不是内部或外部命令，请检查python环境变量的配置），如下图所示：

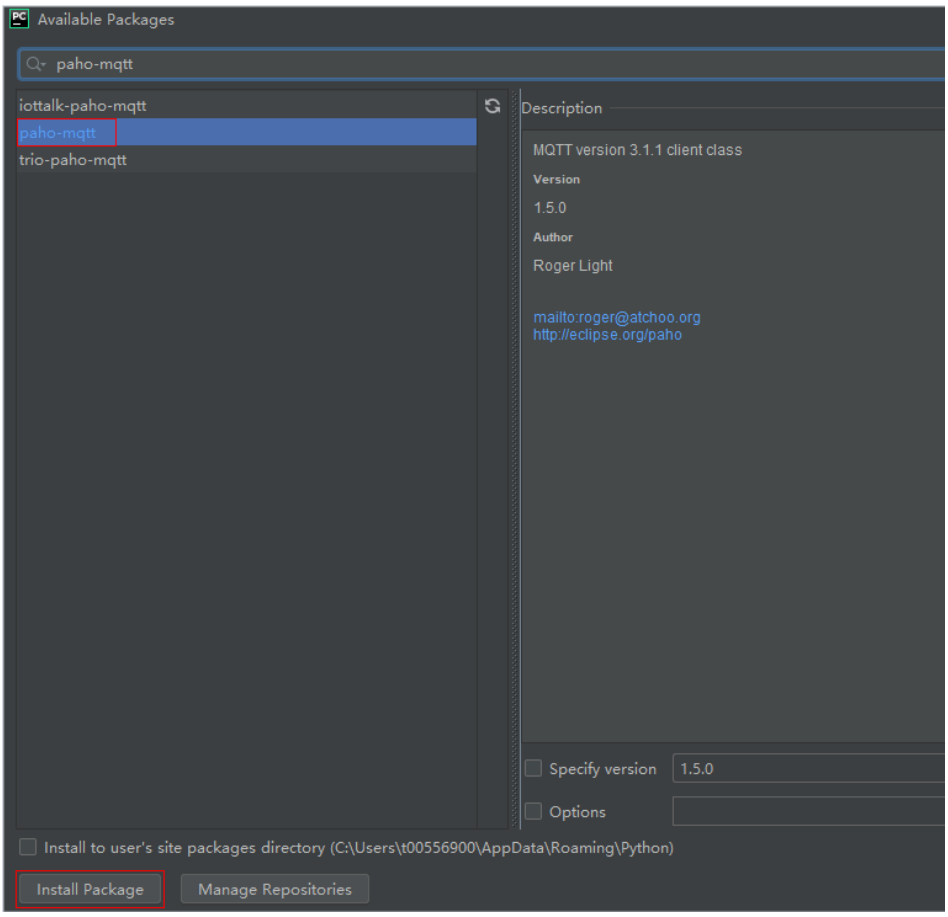


- 方法二：通过PyCharm安装

  a. 打开PyCharm，选择 “File > Setting > Project Interpreter” ，单击右侧 + 号搜索“paho-mqtt”。



  b. 单击左下角 Install Package进行安装。



----结束

建立连接

设备或网关在接入物联网平台时首先需要和平台建立连接，从而将设备或网关与平台进行关联。开发者通过传入设备信息，将设备或网关连接到物联网平台。

1. IoTClientConfig类主要提供配置客户端相关信息的功能，在建立连接之前，先修改以下参数。

```
# 客户端配置
client_cfg = IoTClientConfig(server_ip='iot-mqtts.cn-north-4.myhuaweicloud.com',
device_id='5e85a55f60b7b804c51ce15c_py123', secret='******', is_ssl=True)
# 创建设备
iot_client = IotClient(client_cfg)
```

   - **server_ip**：物联网平台的设备对接地址，可参考**平台对接信息**获取（获取的是域名信息，可通过在cmd命令框中执行“ping 域名”，获取IP地址）；
   - **device_id和secret**：在成功**注册设备**后获取；
   - **is_ssl**：设置为True 时建立MQTTS连接，False时建立MQTT连接。

2. 调用 connect 方法进行连接。

```
iot_client.connect()
```

   如果连接成功会打印：

```
----------------Connection successful !!!
```

   注：如果连接失败，在retreat_reconnection函数中已实现退避重连，代码样例如下：

```
# 退避重连
def retreat_reconnection(self):
```

```
print("---- 退避重连")
global retryTimes
minBackoff = 1
maxBackoff = 30
defaultBackoff = 1
low_bound = (int)(defaultBackoff * 0.8)
high_bound = (int)(defaultBackoff * 1.2)
random_backoff = random.randint(0, high_bound - low_bound)
backoff_with_jitter = math.pow(2.0, retryTimes) * (random_backoff + low_bound)
wait_time_until_next_retry = min(minBackoff + backoff_with_jitter, maxBackoff)
print("the next retry time is ", wait_time_until_next_retry, " seconds")
retryTimes += 1
time.sleep(wait_time_until_next_retry)
self.connect()
```

订阅 Topic

订阅某Topic的设备才能接收broker发布的关于该Topic的消息，关于平台预置Topic可参考**Topic定义**。

在message_sample.py文件中提供了订阅Topic、取消订阅Topic和设备消息上报等功能。

订阅命令下发Topic方式如下：

```
iot_client.subscribe(r'$oc/devices/' + str(self.__device_id) + r'/sys/commands/#')
```

如果订阅成功会打印（ Topic为自定义的Topic，如上Topic_1 ）：

```
------You have subscribed:  topic
```

响应命令下发

在command_sample.py文件中提供了响应平台下发命令的功能。详细接口信息请参考**命令下发接口**。

```
# 响应平台下发的命令
def command_callback(request_id, command):
    # result_code:设置为0命令下发成功，为1下发命令失败
    iot_client.respond_command(request_id, result_code=0)
iot_client.set_command_callback(command_callback)
```

属性上报

属性上报是指设备主动向平台上报自己的属性。更多接口信息请参考**设备属性上报**。

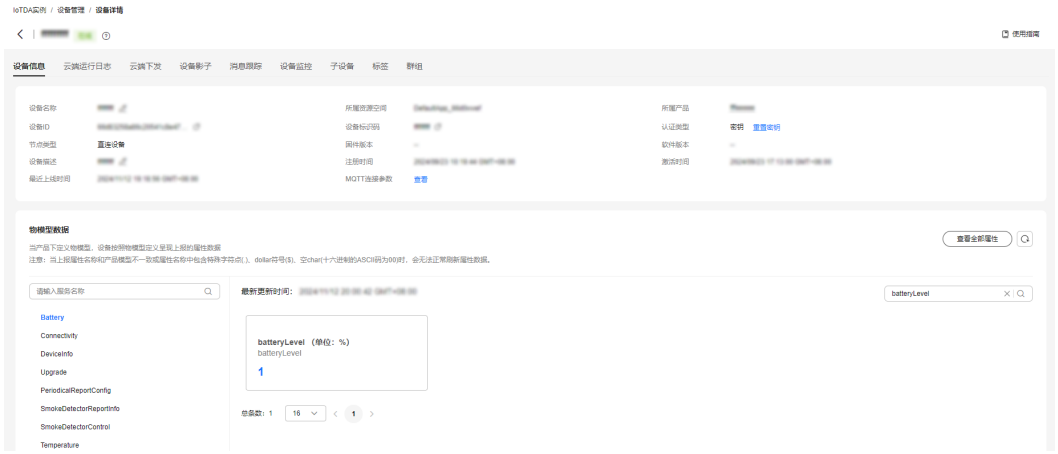
在properties_sample.py文件中实现了设备属性上报、响应平台设置与查询设备属性的功能。

代码实现了设备每隔10秒向平台上报属性的功能，service_property为设备属性对象，具体可在services_properties.py文件查看。

```
# 定时上报属性
while True:
    # 按照产品模型设置属性
    service_property = ServicesProperties()
    service_property.add_service_property(service_id="Battery", property='batteryLevel', value=1)
    iot_client.report_properties(service_properties=service_property.service_property, qos=1)
    time.sleep(10)
```

设备上报属性成功后可在设备详情页面查看到上报的属性。

图 3-211 查看上报数据-Battery_batteryLevel



说明

如果在“设备详情”页面没有最新上报数据，请确认设备上报的服务/属性和产品模型中的服务/属性一致。

消息上报

消息上报是指设备向平台上报消息。message_sample.py文件中提供了消息上报的功能。

```
# 设备向平台发送消息，系统默认topic
iot_client.publish_message('raw message: Hello Huawei cloud IoT')
```

消息上报成功后会打印：

```
Publish success---mid = 1
```

说明

由于是同步命令需要端侧回复响应可**参考接口**。

3.8.4 Android Demo 使用说明

概述

本文以Android语言为例，介绍通过MQTTS/MQTT协议接入平台，基于**平台接口**实现“属性上报”、“订阅接收命令”等功能。

说明

本文中使用的代码为样例代码，仅用于体验平台通信功能，如需进行商用，可以参考**资源获取**获取对应语言的IoT Device SDK进行集成。

前提条件

- 已安装Android，若未安装请参考**安装android studio**，还需要**安装JDK**。
- 已在**管理控制台**获取设备接入地址。获取地址的操作步骤，请参考**平台对接信息**。
- 已在**管理控制台**创建产品和设备。创建产品和设备的具体操作细节，请参考**创建产品**、**注册单个设备**或**批量注册设备**。

准备工作

- 安装android studio

  访问**android studio官网**，选择合适系统的版本下载并安装。（本文以windows 64-bit系统Android Studio 3.5为例）。

  Android Studio downloads

Platform	Android Studio package	Size	SHA-256 checksum
Windows (64-bit)	android-studio-ide-192.6392135-windows.exe Recommended	756 MB	07b6df807fda59e69f05b85ff6f6bd0c70d09e57fb151197155ef5f115f96e59
	android-studio-ide-192.6392135-windows.zip No .exe installer	770 MB	24f8f9ce467b935c25d89b90cad402d21dd45d4ba9af1ad35baeeb414609e483
Windows (32-bit)	android-studio-ide-192.6392135-windows32.zip No .exe installer	770 MB	7b24742726bbc8b40a55dab1f7cdff923ba384b233c21d35d6e96fa36320d067
Mac (64-bit)	android-studio-ide-192.6392135-mac.dmg	768 MB	c5dd347469be0d995e6b4d74ea72b3a6f2572e72b4eac37a0834b0a0984d9583
Linux (64-bit)	android-studio-ide-192.6392135-linux.tar.gz	772 MB	33ec9f61b20b71ca175cd39083b1379ebba896de78b826ea5df5d440c6adfd2a
Chrome OS	android-studio-ide-192.6392135-cros.deb	653 MB	59023aaabc7d5822fd7b1c5a71589b18e487ca8d7fd4320c3547ee0ad390e4ca

- 安装JDK（也可以使用IDE自带的JDK）

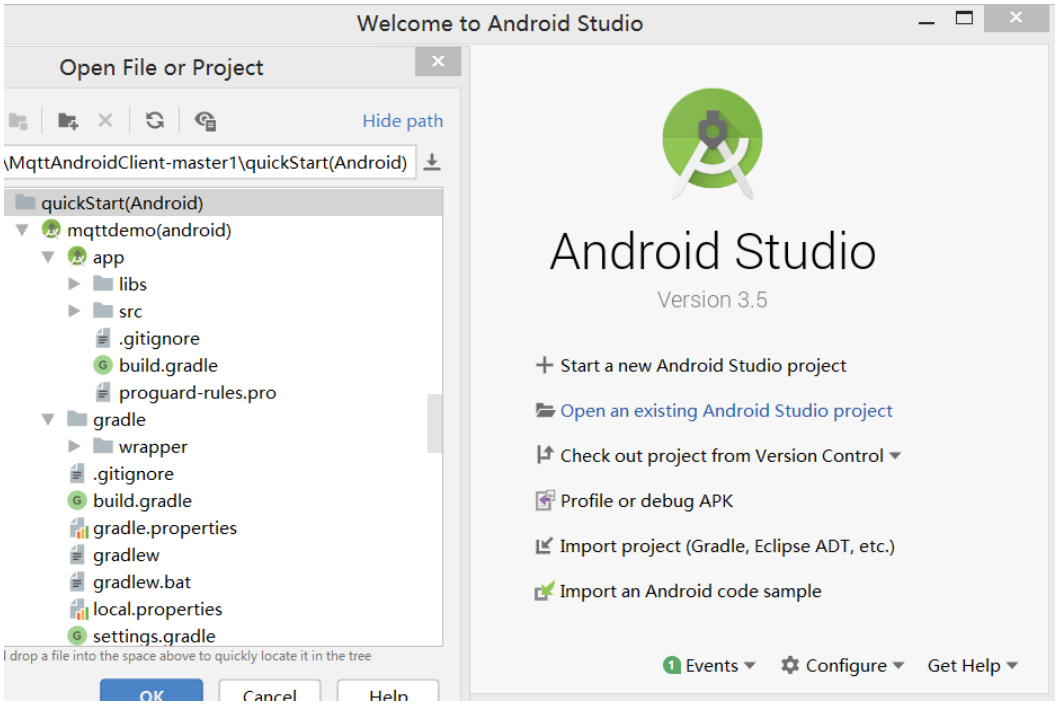
  a. 访问**Oracle官网**，选择合适的JDK版本单击“Download”下载（本文以Windows x64 JDK8为例）。

  b. 下载完成后，运行安装文件，根据界面提示安装。

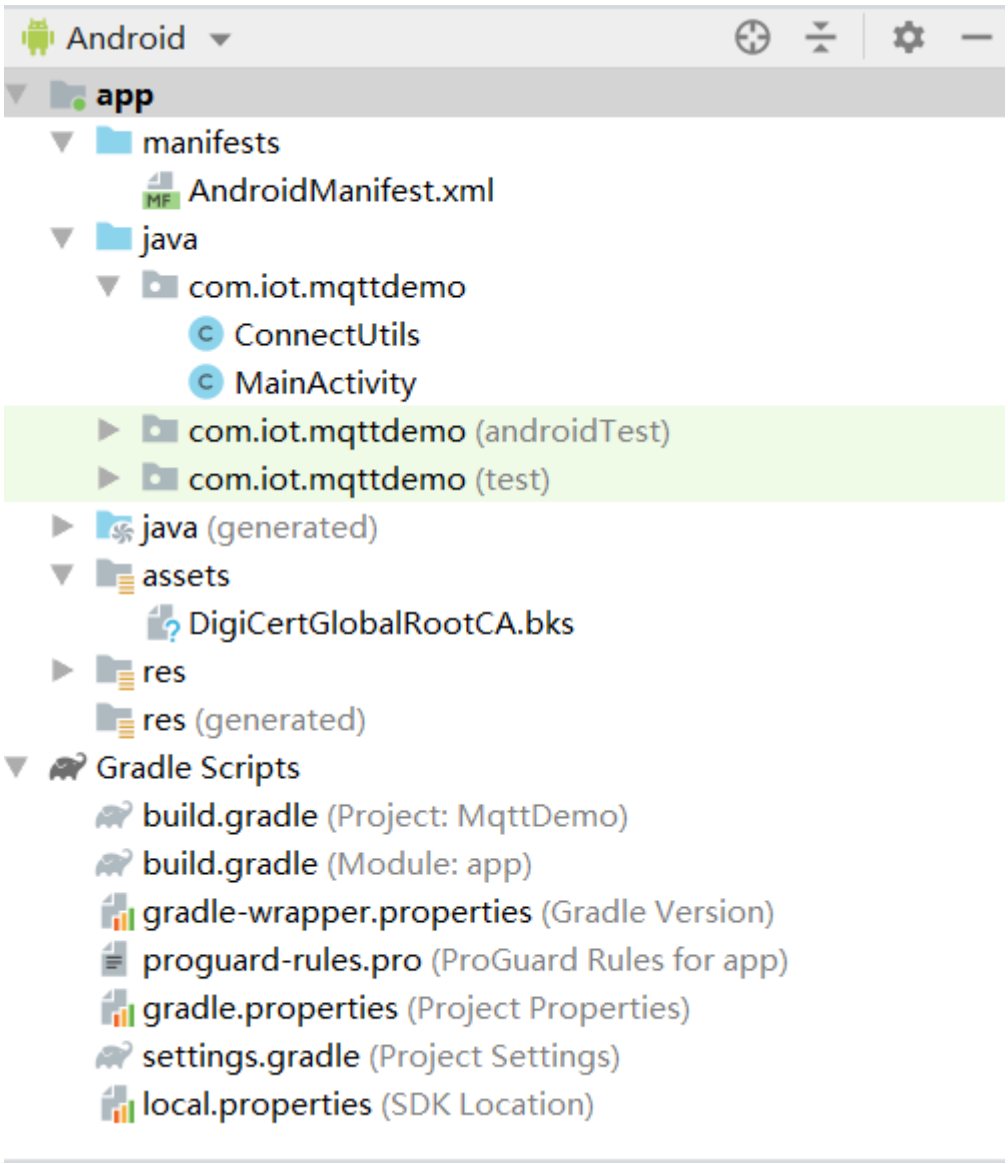
导入代码样例

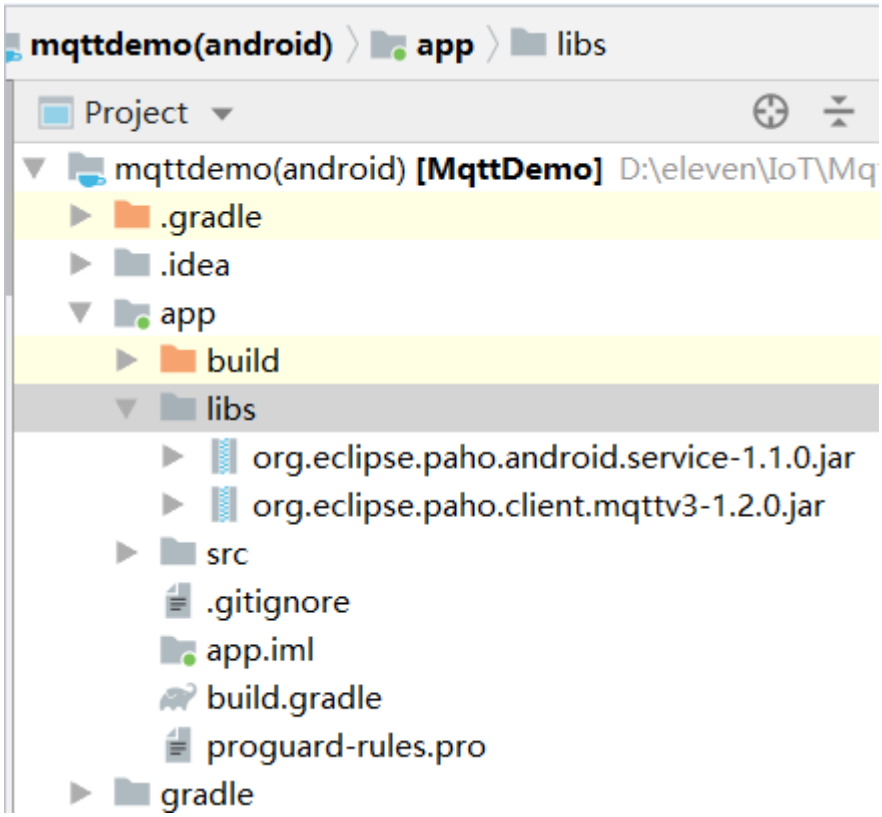
步骤1 下载**quickStart(Android)**样例。

步骤2 运行Android Studio，单击Open，选择步骤**1**中下载的样例。



步骤3 完成代码导入。





代码目录简述：

- **manifests：Android项目的配置文件**;
- **java：项目java代码；**

  MainActivity：demo界面类；

  ConnectUtils：mqtt连接辅助类；
- **asset：项目原生文件；**

  DigiCertGlobalRootCA.bks：设备校验平台身份的证书，用于设备侧接入物联网平台登录鉴权使用；
- **res：项目资源文件（图片、布局、字符串等）；**
- **gradle：项目全局的gradle构建脚本**.
- **libs：项目中使用到了第三方jar包归档目录**

  org.eclipse.paho.android.service-1.1.0.jar：Android启动后台service组件实现消息发布和订阅的组件；

  org.eclipse.paho.client.mqttv3-1.2.0.jar：mqtt java客户端组件；

步骤4 （可选）了解Demo里的关键工程配置（默认不用修改）。

- AndroidManifest.xml：需要添加，支持mqtt service。
  ```
  <service android:name="org.eclipse.paho.android.service.MqttService" />
  ```
- build.gradle：添加依赖，导入libs下的两个mqtt连接所需要的jar包。（也可以添加jar包官网引用）
  ```
  implementation files('libs/org.eclipse.paho.android.service-1.1.0.jar')
  implementation files('libs/org.eclipse.paho.client.mqttv3-1.2.0.jar')
  ```

----结束

界面展示

1. MainActivity类主要提供了界面显示，请填写设备ID和设备密钥，在物联网平台或调用接口注册设备后获取。
2. 示例中默认写了设备侧接入的域名地址（SSL加密接入时该域名要与对应的**证书文件匹配使用**）。
```
private final static String IOT_PLATFORM_URL = "iot-mqtts.cn-north-4.myhuaweicloud.com";
```
3. 用户可以选择设备侧建链时是否SSL加密/不加密，选择Qos方式是0还是1，当前不支持Qos2，可参考**使用限制**。
```
checkbox_mqtt_connet_ssl.setOnCheckedChangeListener(new
CompoundButton.OnCheckedChangeListener() {
    @Override
    public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {
        if (isChecked) {
            isSSL = true;
            checkbox_mqtt_connet_ssl.setText("SSL加密");
        } else {
            isSSL = false;
            checkbox_mqtt_connet_ssl.setText("SSL不加密");
        }
    }
})
```

建立连接

设备或网关在接入物联网平台时首先需要和平台建立连接，从而将设备或网关与平台进行关联。开发者通过传入设备信息，将设备或网关连接到物联网平台。

1. MainActivity类主要提供建立MQTT/MQTTS连接等方法，MQTT默认使用1883端口，MQTTS默认使用8883端口（需要加载证书）。
```
if (isSSL) {
    editText_mqtt_log.append("开始建立MQTTS连接" + "\n");
    serverUrl = "ssl://" + IOT_PLATFORM_URL + ":8883";
} else {
    editText_mqtt_log.append("开始建立MQTT连接" + "\n");
    serverUrl = "tcp://" + IOT_PLATFORM_URL + ":1883";
}
```
2. ConnectUtils类主要提供了SSL加载证书的getMqttsCertificate方法，如果是MQTTS建链方式，需要调用该方法加载证书。

   DigiCertGlobalRootCA.bks：设备校验平台身份的证书，用于设备侧接入物联网平台登录鉴权使用，可以在资源获取中**下载证书文件**。
```
SSLContext sslContext = SSLContext.getInstance("SSL");
KeyStore keyStore = KeyStore.getInstance("bks");
keyStore.load(context.getAssets().open("DigiCertGlobalRootCA.bks"), null);//加载libs目录下的证书
TrustManagerFactory trustManagerFactory = TrustManagerFactory.getInstance("X509");
trustManagerFactory.init(keyStore);
TrustManager[] trustManagers = trustManagerFactory.getTrustManagers();
sslContext.init(null, trustManagers, new SecureRandom());
sslSocketFactory = sslContext.getSocketFactory();
```
3. MainActivity类提供了设置初始化MqttConnectOptions的方法。mqtt连接心跳时间的建议值是120秒，有**使用限制**。
```
mqttAndroidClient = new MqttAndroidClient(mContext, serverUrl, clientId);
private MqttConnectOptions intitMqttConnectOptions(String currentDate) {
    String password =
ConnectUtils.sha256_HMAC(editText_mqtt_device_connect_password.getText().toString(),
currentDate);
    MqttConnectOptions mqttConnectOptions = new MqttConnectOptions();
    mqttConnectOptions.setAutomaticReconnect(true);
    mqttConnectOptions.setCleanSession(true);
    mqttConnectOptions.setKeepAliveInterval(120);
    mqttConnectOptions.setConnectionTimeout(30);
    mqttConnectOptions.setUserName(editText_mqtt_device_connect_deviceId.getText().toString());
    mqttConnectOptions.setPassword(password.toCharArray());
    return mqttConnectOptions;
}
```

4. MainActivity类提供了Mqtt客户端建立连接的的方法connect，并通过回调函数处理连接后的消息返回结果。

```
mqttAndroidClient.connect(mqttConnectOptions, null, new IMqttActionListener()
mqttAndroidClient.setCallback(new MqttCallBack4IoTHub());
```

注：如果连接失败，在initMqttConnects函数中的onFailure回调函数中已实现退避重连，代码样例如下：

```
@Override
public void onFailure(IMqttToken asyncActionToken, Throwable exception) {
    exception.printStackTrace();
    Log.e(TAG, "Fail to connect to: " + exception.getMessage());
    editText_mqtt_log.append("建立连接失败:" + exception.getMessage() + "\n");

    //退避重连
    int lowBound = (int) (defaultBackoff * 0.8);
    int highBound = (int) (defaultBackoff * 1.2);
    long randomBackOff = random.nextInt(highBound - lowBound);
    long backOffWithJitter = (int) (Math.pow(2.0, (double) retryTimes)) * (randomBackOff + lowBound);
    long waitTImeUntilNextRetry = (int) (minBackoff + backOffWithJitter) > maxBackoff ? maxBackoff :
(minBackoff + backOffWithJitter);
    try {
        Thread.sleep(waitTImeUntilNextRetry);
    } catch (InterruptedException e) {
        System.out.println("sleep failed, the reason is" + e.getMessage().toString());
    }
    retryTimes++;
    MainActivity.this.initMqttConnects();
}
```

订阅 Topic

订阅某Topic的设备才能接收broker发布的关于该Topic的消息，关于平台预置Topic可参考**Topic定义**。

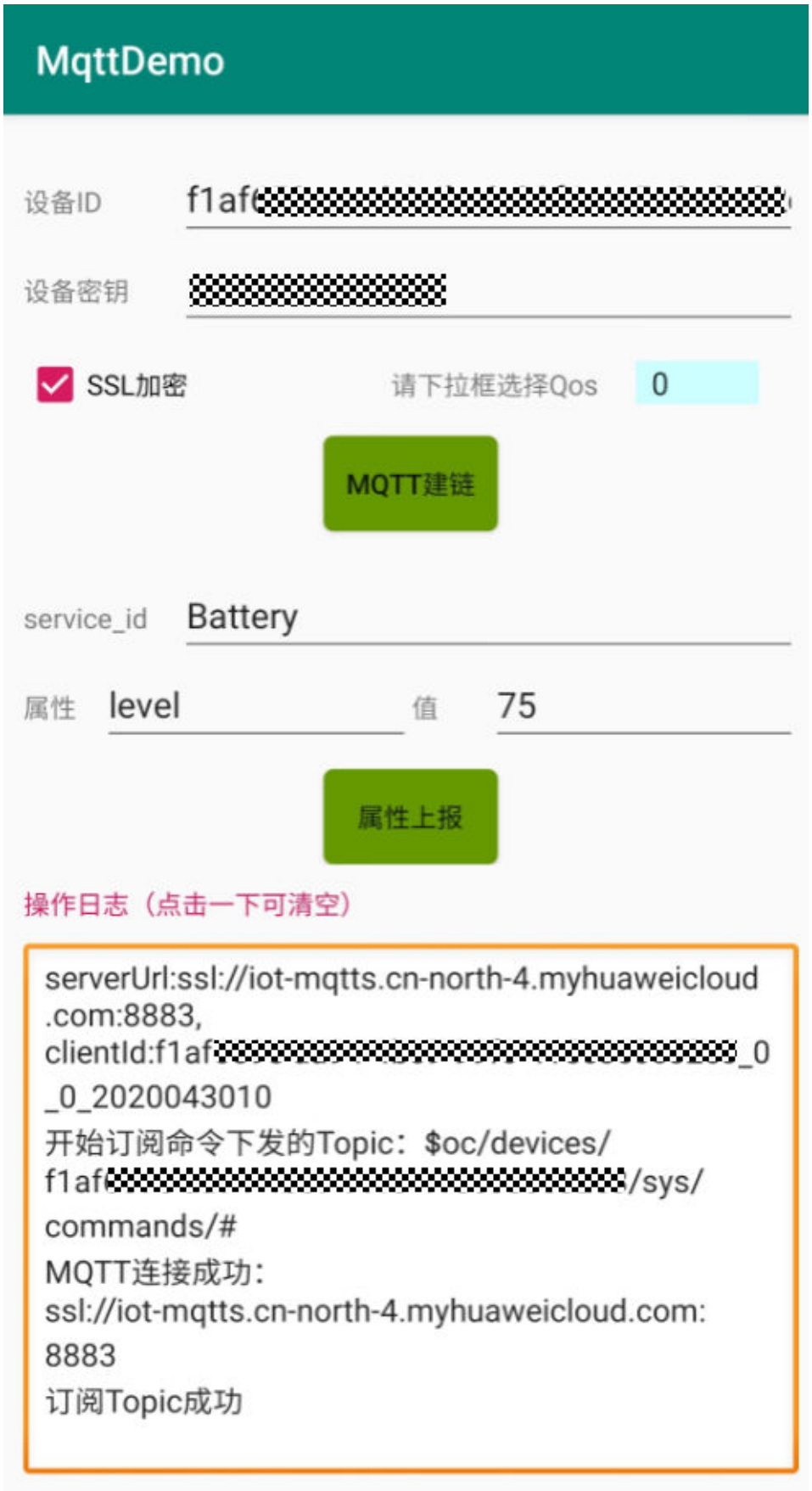
在MainActivity类中提供了订阅命令下发Topic、订阅Topic、取消订阅Topic等功能：

```
String mqtt_sub_topic_command_json = String.format("$oc/devices/%s/sys/commands/#",
editText_mqtt_device_connect_deviceId.getText().toString());
mqttAndroidClient.subscribe(getSubscriptionTopic(), qos, null, new IMqttActionListener()
mqttAndroidClient.unsubscribe(getSubscriptionTopic(), null, new IMqttActionListener()
```

如果建链成功，可以在回调函数中订阅Topic：

```
mqttAndroidClient.connect(mqttConnectOptions, null, new IMqttActionListener() {
    @Overridepublic void onSuccess(IMqttToken asyncActionToken) {
        ......
        subscribeToTopic();
}
```

建链成功后，APP界面日志栏显示如下信息：



属性上报

属性上报是指设备主动向平台上报自己的属性。更多接口信息请参考**设备属性上报**。

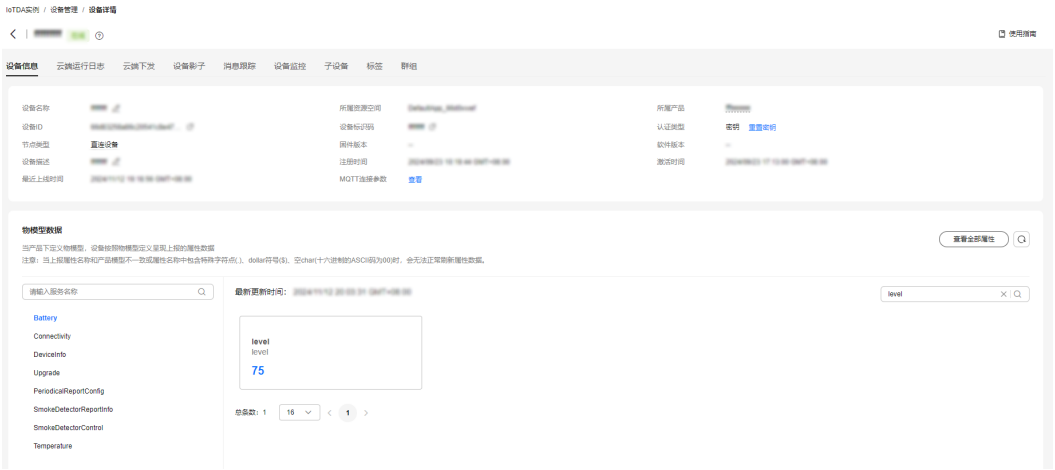
在MainActivity类中实现了属性上报Topic、属性上报功能。

```
String mqtt_report_topic_json = String.format("$oc/devices/%s/sys/properties/report",
editText_mqtt_device_connect_deviceId.getText().toString());
MqttMessage mqttMessage = new MqttMessage();
mqttMessage.setPayload(publishMessage.getBytes());
mqttAndroidClient.publish(publishTopic, mqttMessage);
```

设备上报属性成功后可在设备详情页面查看到上报的属性

图 3-212 查看上报数据-PeriodicalReportConfig

图 3-213 查看上报数据-Battery_level



说明

如果在“设备详情”页面没有最新上报数据，请确认设备上报的服务/属性和产品模型中的服务/属性一致。

接收下发命令

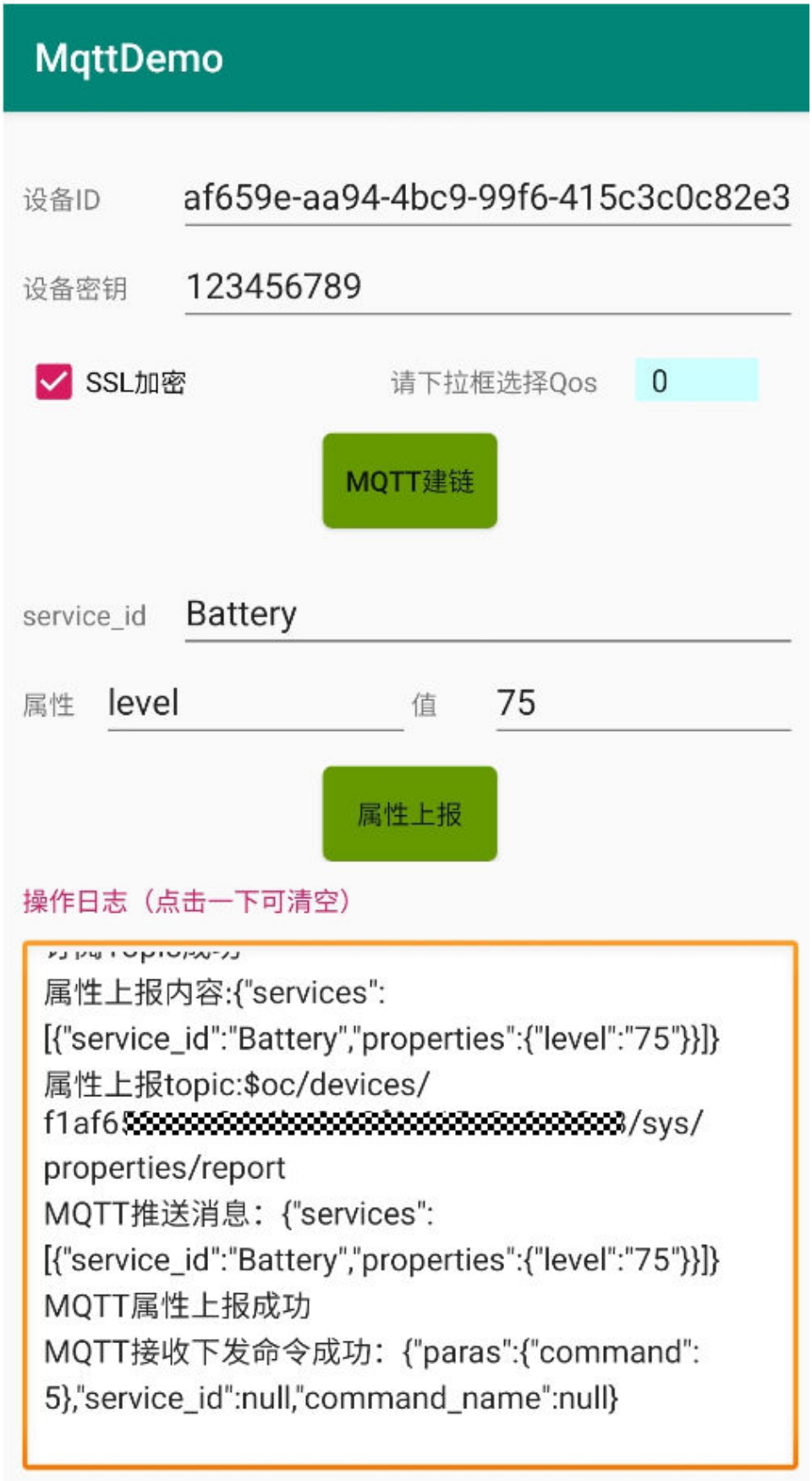
在MainActivity类中提供了接收平台下发命令的功能，在MQTT建链完成后，可以在**管理控制台**设备详情中命令下发或**使用应用侧Demo**对该设备ID进行命令下发，例如下

发参数名为command，参数值为5的命令，下发成功后，在MQTT的回调函数中接收到。

```
private final class MqttCallBack4IoTHub implements MqttCallbackExtended {
    ......
    @Overridepublic void messageArrived(String topic, MqttMessage message) throws Exception {
        Log.i(TAG, "Incoming message: " + new String(message.getPayload(), StandardCharsets.UTF_8));
        editText_mqtt_log.append("MQTT接收下发命令成功：" + message + "\n");
    }
```

在设备详情页面可以查看到命令下发状态，这里显示timeout是因为该Demo示例中仅演示接收命令，没有回复响应给平台。

属性上报和命令接收成功，APP界面显示如下：



3.8.5 C Demo 使用说明

概述

本文以C语言为例，介绍通过MQTTS/MQTT协议接入平台，基于**平台接口**实现“属性上报”、“订阅接收命令”等功能。

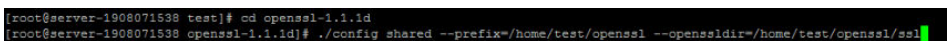
说明

本文中使用的代码为样例代码，仅用于体验平台通信功能，如需进行商用，可以参考**资源获取**获取对应语言的IoT Device SDK进行集成。

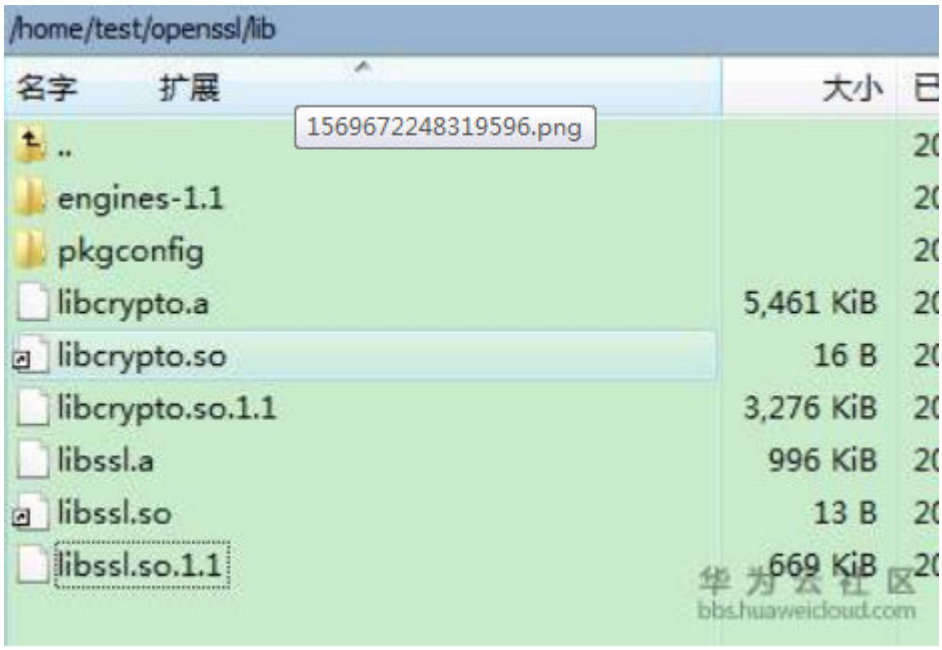
前提条件

- 环境要求：Linux操作系统上，并安装好gcc（建议4.8及以上版本）。
- 库依赖：openssl库（MQTTS需要），paho库。
- 已在**管理控制台**获取设备接入地址。获取地址的操作步骤，请参考**平台对接信息**。
- 已在**管理控制台**创建产品和设备。创建产品和设备的具体操作细节，请参考**创建产品**、**注册单个设备**或**批量注册设备**。

准备工作

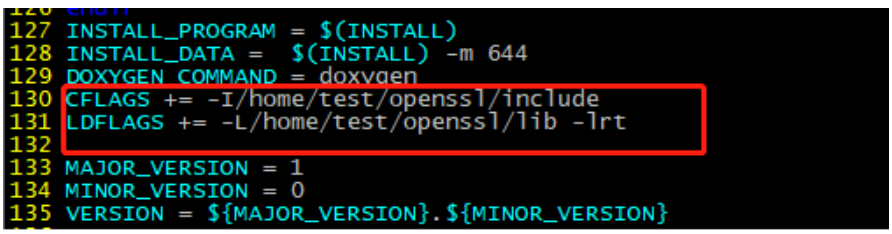
- 编译openssl库
  a. 访问openssl官网（**https://www.openssl.org/source/**）下载最新版本openssl（如openssl-1.1.1d.tar.gz），上传到linux编译机上（以上传到/home/test目录下为例），并使用如下命令解压：
  ```
  tar -zxvf openssl-1.1.1d.tar.gz
  ```
  b. 配置生成makefile文件。
  执行以下命令进入openssl源码目录
  ```
  cd openssl-1.1.1d
  ```
  运行如下配置命令：
  ```
  ./config shared --prefix=/home/test/openssl --openssldir=/home/test/openssl/ssl
  ```
  其中“prefix”是安装目录，“openssldir”是配置文件目录，“shared”作用是生成动态链接库（即.so库）。
  如果编译有问题，配置命令加上no-asm（表示不使用汇编代码）
  ./config no-asm shared --prefix=/home/test/openssl --openssldir=/home/test/openssl/ssl
  
  c. 编译出库。
  在openssl源码目录下，运行make depend命令。
  ```
  make depend
  ```
  再运行make命令进行编译。
  ```
  make
  ```
  安装openssl。
  ```
  make install
  ```
  在配置的openssl安装目录下home/test/openssl找到lib目录，有生成的库文件：

“libcrypto.so.1.1”、“libssl.so.1.1”和软链接“libcrypto.so”、“libssl.so”，请将这些文件复制到demo的lib文件夹下（同时将/home/test/openssl/include/openssl里的内容复制到demo的include/openssl下）。



注：有的编译工具是32位的，如果在64位的linux机器上使用，这时只要将Makefile中的-m64都删除，再进行编译即可。

- 编译paho库文件
  a. 访问github下载地址：**https://github.com/eclipse/paho.mqtt.c**，下载paho.mqtt.c源码。
  b. 解压后上传到linux编译机。
  c. 修改makefile
     i. 通过如下命令进行编辑Makefile
     ```
     vim Makefile
     ```
     ii. 查找字符串
     ```
     /DOXYGEN_COMMAND =
     ```
     iii. 在/DOXYGEN_COMMAND =doxygen的下一行添加下面两行（自定义的openssl的头文件和库文件）
     ```
     CFLAGS += -I/home/test/openssl/include
     LDFLAGS += -L/home/test/openssl/lib -lrt
     ```



     iv. 把如下图的CCDLAGS_SO、LDFLAGS_CS、LDFLAGS_AS、FLAGS_EXES的openssl地址都改成对应的地址

```
194
195 CCFLAGS_SO += -Wno-deprecated-declarations -DOSX -I /home/test/openssl/include
196 LDFLAGS_C += -Wl,-install_name,lib${MQTTLIB_C}.so.${MAJOR_VERSION}
197 LDFLAGS_CS += -Wl,-install_name,lib${MQTTLIB_CS}.so.${MAJOR_VERSION} -L /home/test/openssl/lib
198 LDFLAGS_A += -Wl,-install_name,lib${MQTTLIB_A}.so.${MAJOR_VERSION}
199 LDFLAGS_AS += -Wl,-install_name,lib${MQTTLIB_AS}.so.${MAJOR_VERSION} -L /home/test/openssl/lib
200 FLAGS_EXE += -DOSX
201 FLAGS_EXES += -L /home/test/openssl/lib
202
203 LDCONFIG = echo
204
205 endif
```

d. 编译

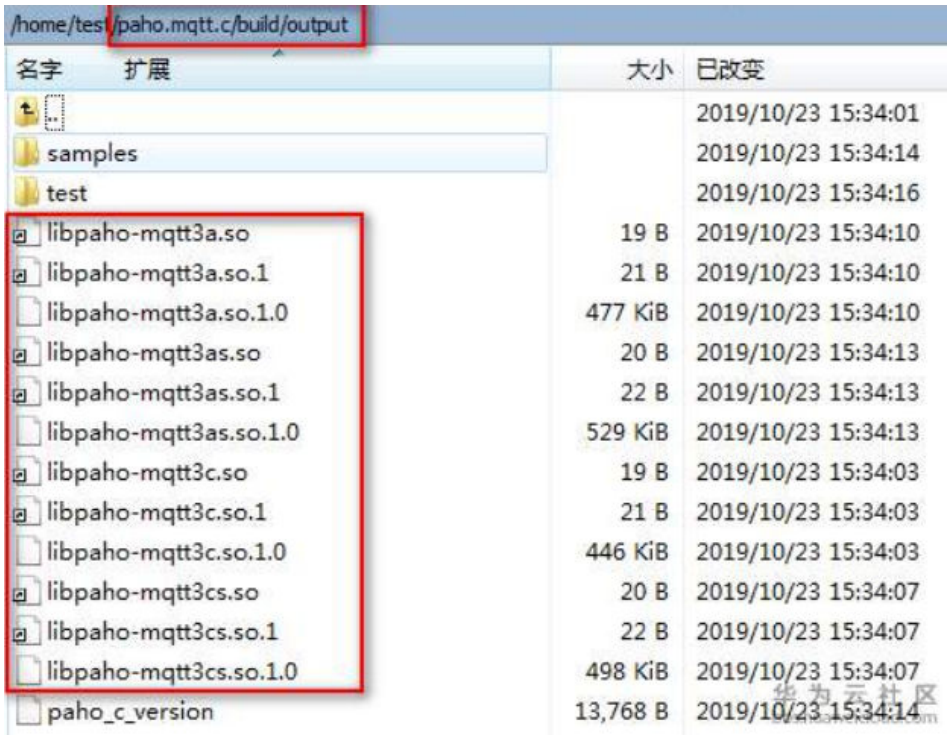
i. 执行清空命令

```
make clean
```

ii. 执行编译命令

```
make
```

e. 编译完成后，可以在build/output目录下看到编译成功的库。



f. 复制paho库文件。

当前SDK仅用到了libpaho-mqtt3as，请将“libpaho-mqtt3as.so”和“libpaho-mqtt3as.so.1”文件复制到demo的lib文件夹下。（同时回到paho源代码路径，进入src目录，将MQTTAsync.h、MQTTClient.h、MQTTClientPersistence.h、MQTTProperties.h、MQTTReasonCodes.h、MQTTSubscribeOpts.h复制到demo的include/base文件夹下）。

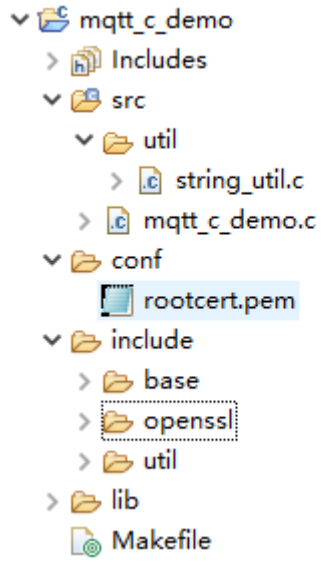
说明

有的paho版本会有MQTTExportDeclarations.h头文件，建议可以将MQTT相关的头文件都添加进去。

导入代码样例

步骤1 下载**quickStart(C)**样例。

步骤2 将代码复制到linux运行环境中。可以看到代码文件层级如下图。



代码目录简述：

- **src：源码目录**

  mqtt_c_demo：demo核心源码；

  util/string_util.c：工具资源文件；

- **conf：证书目录**

  rootcert.pem：设备校验平台身份的证书，用于设备侧接入物联网平台登录鉴权使用；如果对接的IoTDA版本非基础版，请将该**证书文件**中c/ap-southeast-1-device-client-rootcert.pem文件内容复制到conf/rootcert.pem文件中。

- **include: 头文件目录**

  base目录：存放依赖的paho头文件

  openssl目录：存放依赖的openssl头文件

  util目录：存放依赖的工具资源头文件

- **lib：依赖库文件**

  libcrypto.so*/libssl.so*: openssl库文件

  libpaho-mqtt3as.so*: paho库文件

- **Makefile：Makefile文件**

----结束

建立连接

设备或网关在接入物联网平台时首先需要和平台建立连接，从而将设备或网关与平台进行关联。开发者通过传入设备信息，将设备或网关连接到物联网平台。

1. 设置参数。

```
char *uri = "ssl://iot-mqtts.cn-north-4.myhuaweicloud.com:8883";
int port = 8883;
char *username = "********"; //deviceId
char *password = "********";
```

注意：MQTTS为8883端口接入，如果使用MQTT协议接入，url为：tcp://域名空间:1883， port为1883，其中域名空间参考**平台对接信息**获取。心跳时间默认设

置为120秒，用户如果想修改，可以修改代码中的“keepAliveInterval ”参数，心跳时间范围具体可参考**使用限制**。

2. 连接。
   - 在Makefile的第15行最后添加 -lm ，执行**make**进行编译。如果是32位的操作系统，请删除Makefile中的"-m64"。
   - 执行**export LD_LIBRARY_PATH=./lib/**加载库文件。
   - 运行**./MQTT_Demo.o**。

```
//connect
int ret = mqtt_connect();
if (ret != 0) {
    printf("connect failed, result %d\n", ret);
}
```

3. 连接成功后，打印“connect success”，同时在控制台可看到设备已在线。

```
begin to connect the server.
connect success.
```

图 3-214 设备列表-设备在线

注：如果连接失败，在mqtt_connect_failure函数中已实现退避重连，代码样例如下：

```
void mqtt_connect_failure(void *context, MQTTAsync_failureData *response) {
    retryTimes++;
    printf("connect failed: messageId %d, code %d, message %s\n", response->token, response->code, response->message);
    //退避重连
    int lowBound =  defaultBackoff * 0.8;
    int highBound = defaultBackoff * 1.2;
    int randomBackOff = rand() % (highBound - lowBound + 1);
    long backOffWithJitter = (int)(pow(2.0, (double)retryTimes) - 1) * (randomBackOff + lowBound);
    long waitTImeUntilNextRetry = (int)(minBackoff + backOffWithJitter) > maxBackoff ? (minBackoff + backOffWithJitter) : maxBackoff;

    TimeSleep(waitTImeUntilNextRetry);

    //connect
    int ret = mqtt_connect();
    if (ret != 0) {
        printf("connect failed, result %d\n", ret);
    }
}
```

订阅 Topic

订阅某Topic的设备才能接收broker发布的关于该Topic的消息，关于平台预置Topic可参考**Topic定义**。

订阅Topic：

```
//subscribe
char *cmd_topic = combine_strings(3, "$oc/devices/", username, "/sys/commands/#");
```

```
ret = mqtt_subscribe(cmd_topic);
free(cmd_topic);
cmd_topic = NULL;
if (ret < 0) {
    printf("subscribe topic error, result %d\n", ret);
}
```

订阅成功后，demo中会打印“subscribe success”字样。

属性上报

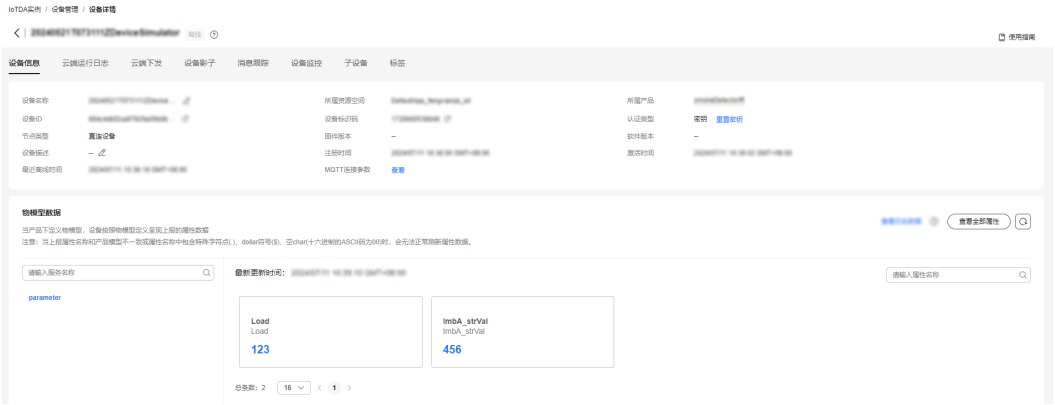
属性上报是指设备主动向平台上报自己的属性，更多信息请参考**设备属性上报**。

```
//publish data
char *payload = "{\"services\":[{\"service_id\":\"parameter\",\"properties\":{\"Load\":\"123\",\"ImbA_strVal
\":\"456\"}}]}";
char *report_topic = combine_strings(3, "$oc/devices/", username, "/sys/properties/report");
ret = mqtt_publish(report_topic, payload);
free(report_topic);
report_topic = NULL;
if (ret < 0) {
    printf("publish data error, result %d\n", ret);
}
```

设备上报属性成功后，demo中会打印“publish success”字样。

同时在“设备详情”页面查看到上报的属性：

图 3-215 查看上报数据-parameter



说明

如果在“设备详情”页面没有最新上报数据，请确认设备上报的服务/属性和产品模型中的服务/属性一致。

接收下发命令

订阅了命令Topic后，可以在控制台下发同步命令。详情请参考**MQTT设备同步命令下发**。

命令下发后，demo中接收到命令：

```
mqtt_message_arrive() success, the topic is $oc/devices/5ebac693352cfb02c567ec88_test2345/sys/commands/request_id=b5fb4352-43
cb-43d7-9ab0-802c435e9ec8, the payload is {"paras":{"timeRead":"1"},"service_id":"command","command_name":"timeRead"}
```

demo中接收命令的代码为：

```
//receive message from the server
int mqtt_message_arrive(void *context, char *topicName, int topicLen, MQTTAsync_message *message) {
    printf( "mqtt_message_arrive() success, the topic is %s, the payload is %s \n", topicName, message->payload);
    return 1; //can not return 0 here, otherwise the message won't update or something wrong would happen
}
```

说明

由于是同步命令需要端侧回复响应可**参考接口**。

3.8.6 C# Demo 使用说明

概述

本文以C#语言为例，介绍通过MQTTS/MQTT协议接入平台，基于**平台接口**实现“属性上报”、“订阅接收命令”等功能。

说明

本文中使用的代码为样例代码，仅用于体验平台通信功能，如需进行商用，可以参考**资源获取**获取对应语言的IoT Device SDK进行集成。

前提条件

- 已安装Microsoft Visual Studio，若未安装请参考**安装Microsoft Visual Studio**。
- 已在**管理控制台**获取设备接入地址。获取地址的操作步骤，请参考**平台对接信息**。
- 已在**管理控制台**创建产品和设备。创建产品和设备的具体操作细节，请参考**创建产品**、**注册单个设备**或**批量注册设备**。

准备工作

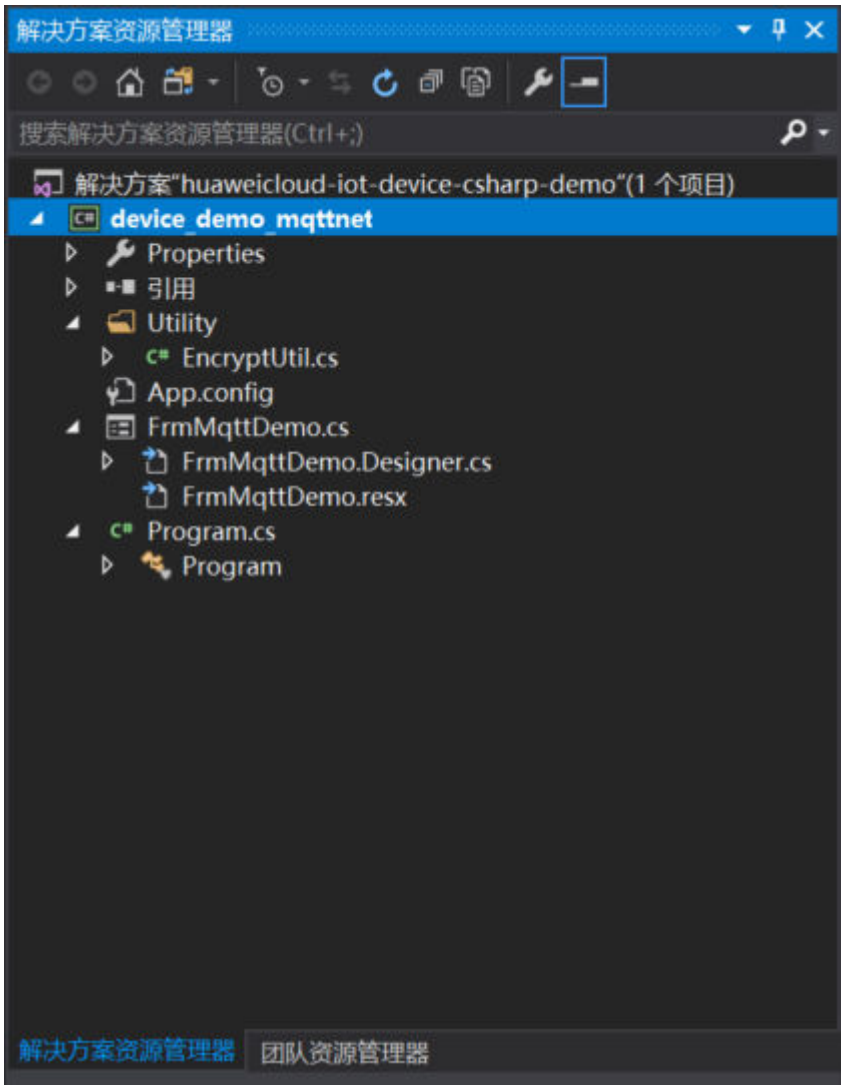
- 访问**Microsoft官网**，选择合适系统的版本下载Microsoft Visual Studio。（本文以windows 64-bit系统，Microsoft Visual Studio 2017和.NET Framework 4.5.1为例）。
- 下载完成后，运行安装文件，根据界面提示安装。

导入代码样例

步骤1 下载**quickStart(C#)**样例。

步骤2 运行Microsoft Visual Studio 2017，单击“打开项目/解决方案”，选择步骤**1**中下载的样例。

步骤3 完成代码导入。



代码目录简述：

- **App.config：Server地址和设备信息配置文件**;
- **C#：项目C#代码；**

  EncryptUtil.cs：设备密钥加密辅助类；

  FrmMqttDemo.cs：窗体界面；

  Program.cs：Demo程序启动入口。
- **dll：项目中使用到了第三方库**

  MQTTnet：v3.0.11，是一个基于 MQTT 通信的高性能 .NET 开源库，它同时支持 MQTT 服务器端和客户端，引用库文件包含MQTTnet.dll。

  MQTTnet.Extensions.ManagedClient：v3.0.11，这是一个扩展库，它使用 MQTTnet为托管MQTT客户机提供附加功能。

步骤4 Demo里的工程配置参数。

- App.config：需要配置服务器地址、设备ID和设备密钥，用于启动Demo程序的时候，程序将此信息自动写到Demo主界面。

  <add key="serverUri" value="serveruri"/>

  <add key="deviceId" value="deviceid"/>

```
<add key="deviceSecret" value="secret"/>
<add key="PortIsSsl" value="8883"/>
<add key="PortNotSsl" value="1883"/>
```

----结束

界面展示

1. FrmMqttDemo主要提供了界面显示，默认启动后自动从App.config中获取Server地址、设备ID、设备密钥。请根据实际创建的设备信息填写。
   - Server地址：即域名，参考**平台对接信息**获取；
   - 设备ID和设备密钥：在物联网平台**注册设备**或调用**创建设备**接口后获取。
2. 示例中App.config默认写了设备侧接入的Server地址（SSL加密接入时该Server地址要与对应的**证书文件匹配使用**）。
   ```
   <add key="serverUri" value="iot-mqtts.cn-north-4.myhuaweicloud.com"/>;
   ```
3. 用户可以选择设备侧建链时是否为SSL加密，选择Qos方式是0还是1，当前不支持Qos2，可参考**使用限制**。

新建连接

设备或网关在接入物联网平台时首先需要和平台建立连接，从而将设备或网关与平台进行关联。开发者通过传入设备信息，将设备或网关连接到物联网平台。

1. FrmMqttDemo类主要提供建立MQTT/MQTTS连接等方法，MQTT默认使用1883端口，MQTTS默认使用8883端口（需要加载设备校验平台身份的证书DigiCertGlobalRootCA.crt.pem，用于设备侧接入物联网平台登录鉴权使用，可以在资源获取中**下载证书文件**），ManagedMqttClientOptionsBuilder中提供了设置初始化KeepAlivePeriod的属性。mqtt连接心跳时间的建议值是120秒，有**使用限制**。

```
int portIsSsl = int.Parse(ConfigurationManager.AppSettings["PortIsSsl"]);
int portNotSsl = int.Parse(ConfigurationManager.AppSettings["PortNotSsl"]);

if (client == null)
{
    client = new MqttFactory().CreateManagedMqttClient();
}

string timestamp = DateTime.Now.ToString("yyyyMMddHH");
string clientID = txtDeviceId.Text + "_0_0_" + timestamp;

// 对密码进行HmacSHA256加密
string secret = string.Empty;
if (!string.IsNullOrEmpty(txtDeviceSecret.Text))
{
    secret = EncryptUtil.HmacSHA256(txtDeviceSecret.Text, timestamp);
}

// 判断是否为安全连接
if (!cbSSLConnect.Checked)
{
    options = new ManagedMqttClientOptionsBuilder()
    .WithAutoReconnectDelay(TimeSpan.FromSeconds(RECONNECT_TIME))
    .WithClientOptions(new MqttClientOptionsBuilder()
        .WithTcpServer(txtServerUri.Text, portNotSsl)
        .WithCommunicationTimeout(TimeSpan.FromSeconds(DEFAULT_CONNECT_TIMEOUT))
        .WithCredentials(txtDeviceId.Text, secret)
        .WithClientId(clientID)
        .WithKeepAlivePeriod(TimeSpan.FromSeconds(DEFAULT_KEEPLIVE))
        .WithCleanSession(false)
        .WithProtocolVersion(MqttProtocolVersion.V311)
        .Build())
    .Build();
}
else
{
    string caCertPath = Environment.CurrentDirectory + @"\certificate\rootcert.pem";
    X509Certificate2 crt = new X509Certificate2(caCertPath);

    options = new ManagedMqttClientOptionsBuilder()
    .WithAutoReconnectDelay(TimeSpan.FromSeconds(RECONNECT_TIME))
    .WithClientOptions(new MqttClientOptionsBuilder()
        .WithTcpServer(txtServerUri.Text, portIsSsl)
        .WithCommunicationTimeout(TimeSpan.FromSeconds(DEFAULT_CONNECT_TIMEOUT))
        .WithCredentials(txtDeviceId.Text, secret)
        .WithClientId(clientID)
        .WithKeepAlivePeriod(TimeSpan.FromSeconds(DEFAULT_KEEPLIVE))
        .WithCleanSession(false)
        .WithTls(new MqttClientOptionsBuilderTlsParameters()
        {
            AllowUntrustedCertificates = true,
            UseTls = true,
            Certificates = new List<X509Certificate> { crt },
            CertificateValidationHandler = delegate { return true; },
            IgnoreCertificateChainErrors = false,
            IgnoreCertificateRevocationErrors = false
        })
        .WithProtocolVersion(MqttProtocolVersion.V311)
        .Build())
    .Build();
}
```

2. FrmMqttDemo类提供了Mqtt客户端建立连接的的方法StartAsync，连接成功后会通过回调函数OnMqttClientConnected打印连接成功日志。

```
Invoke((new Action(() =>
{
    ShowLogs($"{"try to connect to server " + txtServerUri.Text}{Environment.NewLine}");
})));

if (client.IsStarted)
```

```
{
    await client.StopAsync();
}

// 注册事件
client.ApplicationMessageProcessedHandler = new
ApplicationMessageProcessedHandlerDelegate(new
Action<ApplicationMessageProcessedEventArgs>(ApplicationMessageProcessedHandlerMethod)); // 消息发布回调

client.ApplicationMessageReceivedHandler = new
MqttApplicationMessageReceivedHandlerDelegate(new
Action<MqttApplicationMessageReceivedEventArgs>(MqttApplicationMessageReceived)); // 命令下发回调

client.ConnectedHandler = new MqttClientConnectedHandlerDelegate(new
Action<MqttClientConnectedEventArgs>(OnMqttClientConnected)); // 连接成功回调

client.DisconnectedHandler = new MqttClientDisconnectedHandlerDelegate(new
Action<MqttClientDisconnectedEventArgs>(OnMqttClientDisconnected)); // 连接断开回调

// 连接平台设备
await client.StartAsync(options);
```

注：如果连接失败，在OnMqttClientDisconnected函数中已实现退避重连，代码样例如下：

```
private void OnMqttClientDisconnected(MqttClientDisconnectedEventArgs e)
{
    try {
        Invoke((new Action(() =>
        {
            ShowLogs("mqtt server is disconnected" + Environment.NewLine);

            txtSubTopic.Enabled = true;
            btnConnect.Enabled = true;
            btnDisconnect.Enabled = false;
            btnPublish.Enabled = false;
            btnSubscribe.Enabled = false;
        })));

        if (cbReconnect.Checked)
        {
            Invoke((new Action(() =>
            {
                ShowLogs("reconnect is starting" + Environment.NewLine);
            })));

            //退避重连
            int lowBound = (int)(defaultBackoff * 0.8);
            int highBound = (int)(defaultBackoff * 1.2);
            long randomBackOff = random.Next(highBound - lowBound);
            long backOffWithJitter = (int)(Math.Pow(2.0, retryTimes)) * (randomBackOff + lowBound);
            long waitTImeUtilNextRetry = (int)(minBackoff + backOffWithJitter) > maxBackoff ?
maxBackoff : (minBackoff + backOffWithJitter);

            Invoke((new Action(() =>
            {
                ShowLogs("next retry time: " + waitTImeUtilNextRetry + Environment.NewLine);
            })));

            Thread.Sleep((int)waitTImeUtilNextRetry);

            retryTimes++;

            Task.Run(async () => { await ConnectMqttServerAsync(); });
        }
    }
    catch (Exception ex)
    {
```

```
            Invoke((new Action(() =>
            {
                ShowLogs("mqtt demo error: " + ex.Message + Environment.NewLine);
            })));
        }
    }
```

订阅 Topic

订阅某Topic的设备才能接收broker发布的关于该Topic的消息，关于平台预置Topic可参考**Topic定义**。

在FrmMqttDemo类中提供了订阅命令下发Topic的功能：

```
List<MqttTopicFilter> listTopic = new List<MqttTopicFilter>();

var topicFilterBulderPreTopic = new MqttTopicFilterBuilder().WithTopic(topic).Build();
listTopic.Add(topicFilterBulderPreTopic);

// 订阅Topic
client.SubscribeAsync(listTopic.ToArray()).Wait();
```

建链后，如果成功订阅Topic，主界面日志栏显示如下信息：

接收下发命令

在FrmMqttDemo类中提供了接收平台下发命令的功能，在MQTT建链完成并成功订阅Topic后，可以在**管理控制台**设备详情中命令下发或**使用应用侧Demo**对该设备ID进行命令下发。下发成功后，在MQTT的回调函数中接收到平台下发给设备的命令。

```
private void MqttApplicationMessageReceived(MqttApplicationMessageReceivedEventArgs e)
{
    Invoke((new Action(() =>
    {
        ShowLogs($"received message is {Encoding.UTF8.GetString(e.ApplicationMessage.Payload)}
{Environment.NewLine}");
```

```
        string msg = "{\"result_code\": 0,\"response_name\": \"COMMAND_RESPONSE\",\"paras\": {\"result\": \"success\"}}";

        string topic = "$oc/devices/" + txtDeviceId.Text + "/sys/commands/response/request_id=" + e.ApplicationMessage.Topic.Split('=')[1];

        ShowLogs($"{"response message msg = " + msg}{Environment.NewLine}");

        var appMsg = new MqttApplicationMessage();
        appMsg.Payload = Encoding.UTF8.GetBytes(msg);
        appMsg.Topic = topic;
        appMsg.QualityOfServiceLevel = int.Parse(cbOosSelect.SelectedValue.ToString()) == 0 ? MqttQualityOfServiceLevel.AtMostOnce : MqttQualityOfServiceLevel.AtLeastOnce;
        appMsg.Retain = false;

        // 上行响应
        client.PublishAsync(appMsg).Wait();
    }));
}
```

例如下发参数名为SmokeDetectorControl: SILENCE，参数值为50的命令。

图 3-216 命令下发-同步命令下发

命令下发成功后，Demo界面显示如下：

发布 Topic

发布Topic是指设备主动向平台上报自己的属性或消息，详细见**设备属性上报接口**文档。

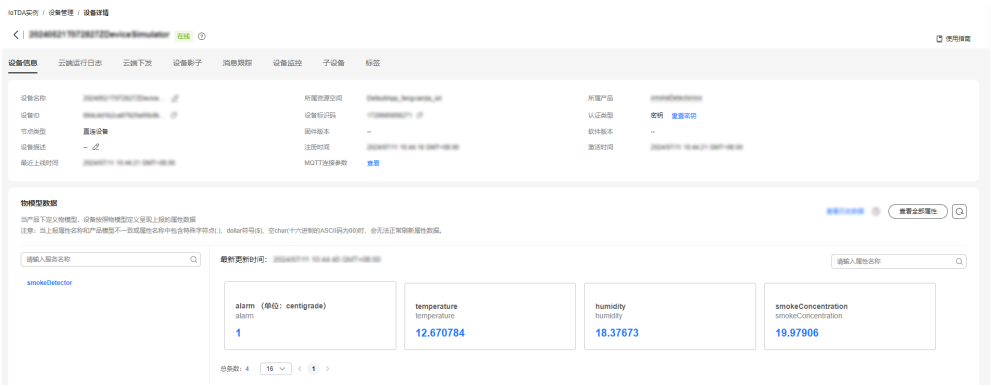
在FrmMqttDemo中实现了上报Topic、属性上报功能。

```
var appMsg = new MqttApplicationMessage();
appMsg.Payload = Encoding.UTF8.GetBytes(inputString);
appMsg.Topic = topic;
appMsg.QualityOfServiceLevel = int.Parse(cbOosSelect.SelectedValue.ToString()) == 0 ?
MqttQualityOfServiceLevel.AtMostOnce : MqttQualityOfServiceLevel.AtLeastOnce;
appMsg.Retain = false;

// 上行响应
client.PublishAsync(appMsg).Wait();
```

发布Topic后，Demo界面显示如下：

设备上报属性成功后可在设备详情页面查看到上报的属性：

图 3-217 查看上报数据-Demo_smokeDetector



说明

如果在“设备详情”页面没有最新上报数据，请确认设备上报的服务/属性和产品模型中的服务/属性一致。

说明

由于是同步命令需要端侧回复响应可**参考接口**。

3.8.7 Node.js Demo 使用说明

概述

本文以Node.js为例，介绍通过MQTTS/MQTT协议接入平台，基于**平台接口**实现“属性上报”、“订阅接收命令”等功能。

说明

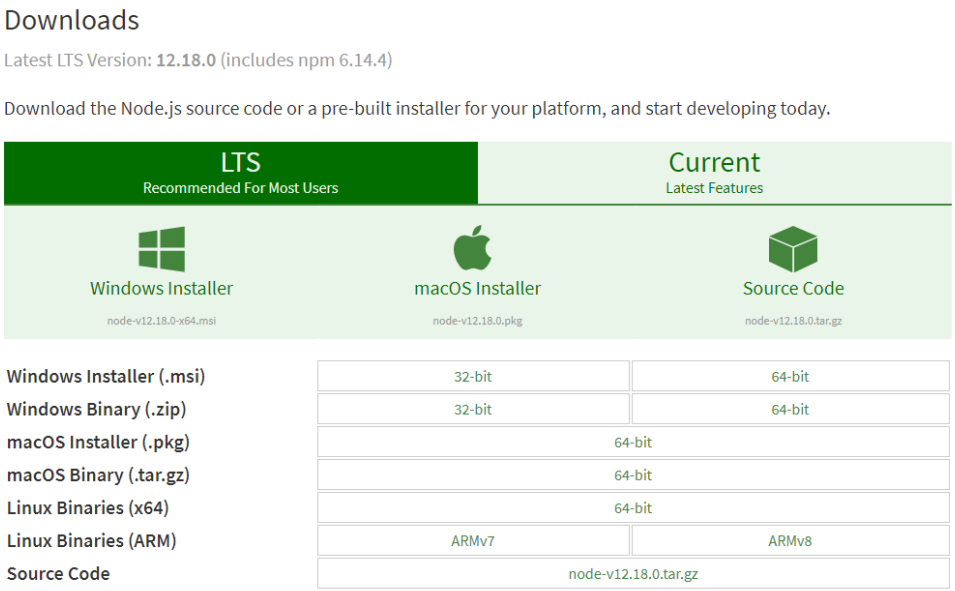
本文中使用的代码为样例代码，仅用于体验平台通信功能，如需进行商用，可以参考**资源获取**获取对应语言的IoT Device SDK进行集成。

前提条件

- 已安装Node.js，若未安装请参考**安装Node.js**。
- 已在**管理控制台**获取设备接入地址。获取地址的操作步骤，请参考**平台对接信息**。
- 已在**管理控制台**创建产品和设备。创建产品和设备的具体操作细节，请参考**创建产品**、**注册单个设备**或**批量注册设备**。

准备工作

1. 安装Node.js访问**Node.js官网**，选择合适系统的版本下载。（本文以windows 64-bit系统，Node.js版本v12.18.0（npm 6.14.4）为例）。



2. 下载完成后，运行安装文件，根据界面提示安装。
3. 检查Node.js是否安装成功。

   Win键 + r -->输入 cmd-->回车，进入命令行窗口。

   输入node –v，回车后显示Node.js版本，输入npm -v显示版本信息，即表示安装成功。

导入代码样例

步骤1 下载**quickStart(Node.js)**样例，并解压。

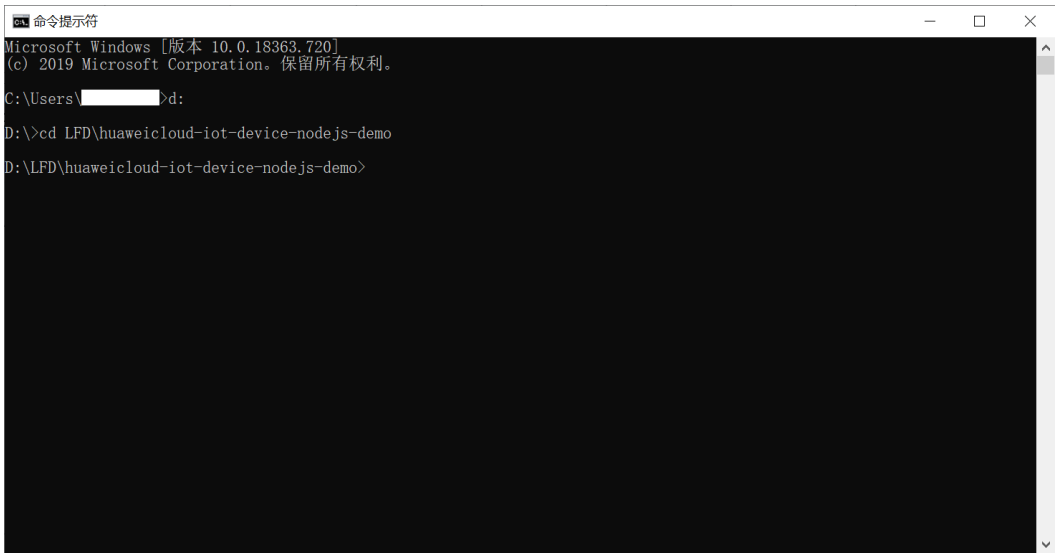
步骤2 运行Win键 + r -->输入 cmd-->回车，进入命令行窗口，安装全局模块，执行如下命令：

npm install mqtt -g：mqtt协议模块；

npm install crypto-js -g：设备密钥加密算法模块；

npm install fs -g：用于加载平台证书；

步骤3 找到文件解压的对应目录，如下图所示。



代码目录介绍如下：

- **DigiCertGlobalRootCA.crt.pem**：平台证书文件;
- **MqttDemo.js**：Node.js源码，包含MQTT/MQTTS连接到平台，并进行属性上报，命令下发；

步骤4 Demo里的关键工程配置参数。其中MqttDemo.js需要配置Server地址、设备ID和设备密钥，用于启动Demo时，连接控制台上注册的设备。

- Server地址：即域名，参考**平台对接信息**获取，SSL加密接入时该Server地址要与对应的**证书文件匹配使用**；
- 设备ID和设备密钥：在物联网平台**注册设备**或调用**创建设备**接口后获取。

```
var TRUSTED_CA = fs.readFileSync("DigiCertGlobalRootCA.crt.pem");//获取证书

//IoT平台mqtt对接地址（要替换为设备所在的平台域名地址）
var serverUrl = "xxx.myhuaweicloud.com"; //请填写设备所在平台的接入地址

//注册设备时获得的deviceId,密钥（要替换为自己注册的设备ID与密钥）
var deviceId = "722cb***************";
var secret = "****";
var timestamp = dateFormat("YYYYmmddHH", new Date());

var propertiesReportJson = {'services':[{'properties':
{'alarm':1,'temperature':12.670784,'humidity':18.37673,'smokeConcentration':19.97906},'service_id':'smokeDet
ector','event_time':null}]};
var responseReqJson = {'result_code': 0,'response_name': 'COMMAND_RESPONSE','paras': {'result': 'success'}};
```

步骤5 用户可通过mqtt.connect(options)中选择不同的options，确定设备侧建链时是否进行SSL加密，建议使用默认的MQTTS安全连接。

```
//MQTTS安全连接
var options = {
    host: serverUrl,
    port: 8883,
    clientId: getClientId(deviceId),
    username: deviceId,
    password:HmacSHA256(secret, timestamp).toString(),
    ca: TRUSTED_CA,
    protocol: 'mqtts',
    rejectUnauthorized: false,
```

```
    keepalive: 120,
    reconnectPeriod: 10000,
    connectTimeout: 30000
}

//MQTT非安全连接，不建议使用
var option = {
    host: serverUrl,
    port: 1883,
    clientId: getClientId(deviceId),
    username: deviceId,
    password: HmacSHA256(secret, timestamp).toString(),
    keepalive: 120,
    reconnectPeriod: 10000,
    connectTimeout: 30000
    //protocol: 'mqtts'
    //rejectUnauthorized: false
}

//此处默认使用options为安全连接
var client = mqtt.connect(options);
```

----结束

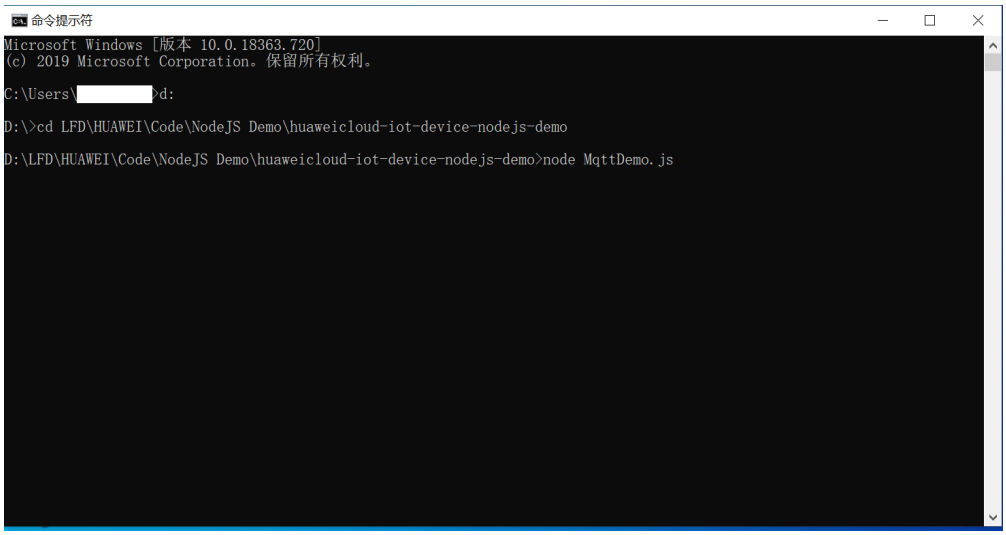
程序启动

设备或网关在接入物联网平台时首先需要和平台建立连接，从而将设备或网关与平台进行关联。开发者通过传入设备信息，将设备或网关连接到物联网平台。

1. 此Demo主要提供建立MQTT/MQTTS连接等方法，MQTT使用1883端口，MQTTS使用8883端口（需要加载设备校验平台身份的证书，用于设备侧接入物联网平台登录鉴权使用），并提供了Mqtt客户端建立连接的方法mqtt.connect(options)。

   ```
   var client = mqtt.connect(options);

   client.on('connect', function () {
       log("connect to mqtt server success, deviceId is " + deviceId);
       //订阅Topic
       subScribeTopic();
       //发布消息
       publishMessage();
   })

   //命令下发响应
   client.on('message', function (topic, message) {
       log('received message is ' + message.toString());

       var jsonMsg = responseReq;
       client.publish(getResponseTopic(topic.toString().split("=")[1]), jsonMsg);
       log('response message is ' + jsonMsg);
   })
   ```

   找到Node.js Demo源码目录，修改**关键工程配置参数**后启动程序，如下图：



启动程序前，设备状态是离线。

图 3-218 设备列表-设备离线

启动程序后，设备状态变为在线

图 3-219 设备列表-设备在线

注：如果连接失败，在重连回调函数中已实现退避重连，代码样例如下：

```
client.on('reconnect', () => {

    log("reconnect is starting");

    //退避重连
    var lowBound = Number(defaultBackoff)*Number(0.8);
    var highBound = Number(defaultBackoff)*Number(1.2);

    var randomBackOff = parseInt(Math.random()*(highBound-lowBound+1),10);

    var backOffWithJitter = (Math.pow(2.0, retryTimes)) * (randomBackOff + lowBound);

    var waitTImeUtilNextRetry = (minBackoff + backOffWithJitter) > maxBackoff ? maxBackoff :
(minBackoff + backOffWithJitter);

    client.options.reconnectPeriod = waitTImeUtilNextRetry;

    log("next retry time: " + waitTImeUtilNextRetry);

    retryTimes++;
})
```

2. 订阅某Topic的设备才能接收broker发布的关于该Topic的消息，关于平台预置Topic可参考**Topic定义**。此Demo调用subScribeTopic方法进行订阅Topic，订阅成功后等待平台命令下发：

```
//订阅接收命令topic
function subScribeTopic() {
    client.subscribe(getCmdRequestTopic(), function (err) {
        if (err) {
            log("subscribe error:" + err);
        } else {
            log("topic : " + getCmdRequestTopic() + " is subscribed success");
        }
    })
}
```

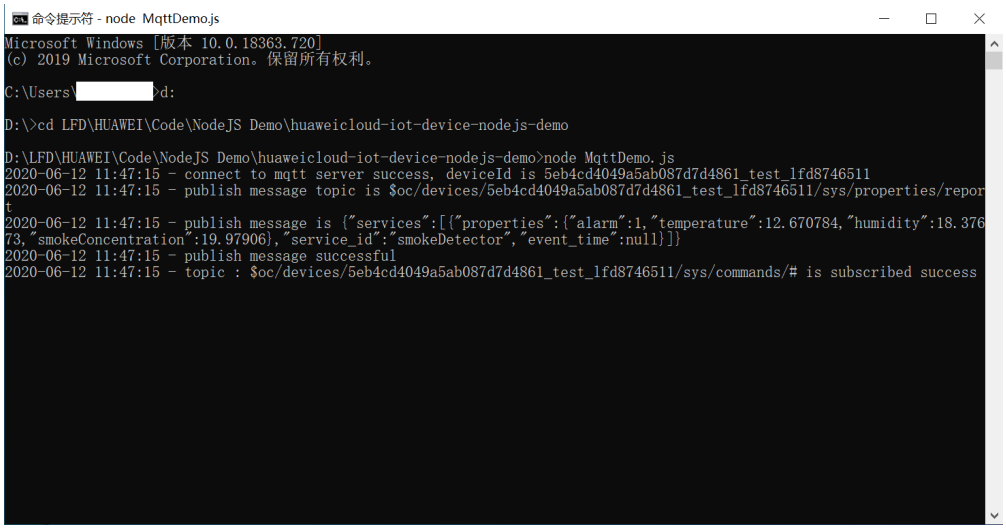
3. 发布Topic是指设备主动向平台上报自己的属性或消息，详细见**设备属性上报接口**文档。连接成功后，调用publishMessage方法进行属性上报：

```
//上报json数据，注意serviceId要与Profile中的定义对应
function publishMessage() {
    var jsonMsg = propertiesReport;
    log("publish message topic is " + getReportTopic());
    log("publish message is " + jsonMsg);
    client.publish(getReportTopic(), jsonMsg);
    log("publish message successful");
}
```

上报属性的json：

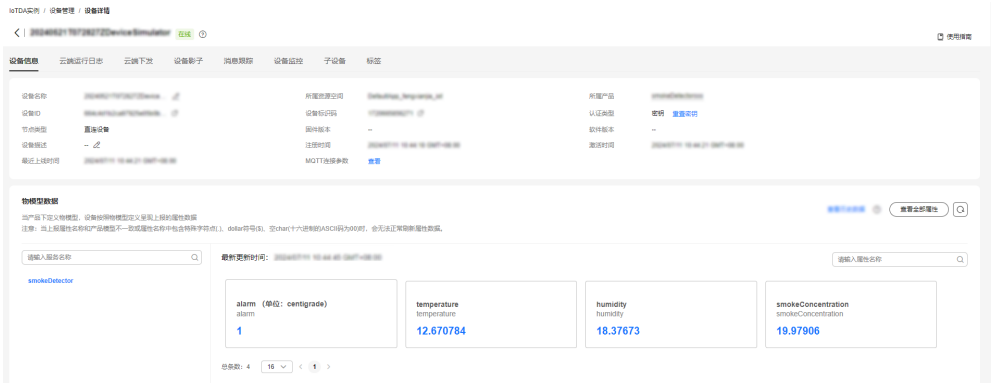
```
var propertiesReportJson = {'services':[{'properties':
{'alarm':1,'temperature':12.670784,'humidity':18.37673,'smokeConcentration':19.97906},'service_id':'smo
keDetector','event_time':null}]};
```

命令行主界面如下：



属性上报成功，平台界面如下：

图 3-220 查看上报数据-Demo_smokeDetector



说明

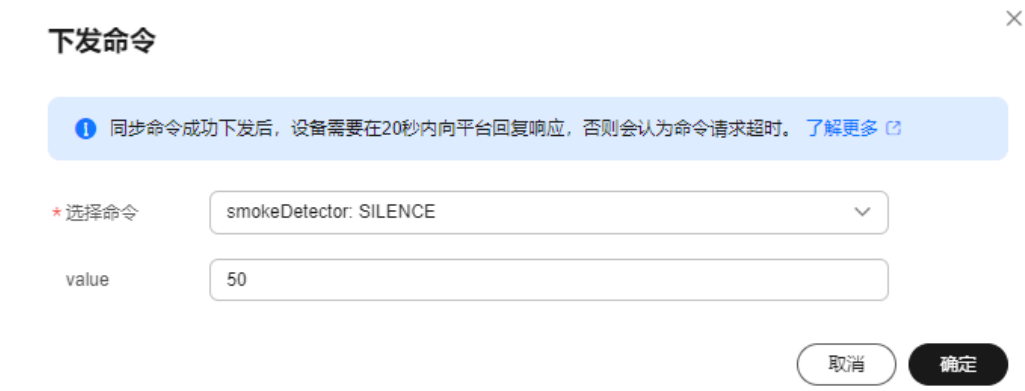
如果在“设备详情”页面没有最新上报数据，请确认设备上报的服务/属性和产品模型中的服务/属性一致。

接收下发命令

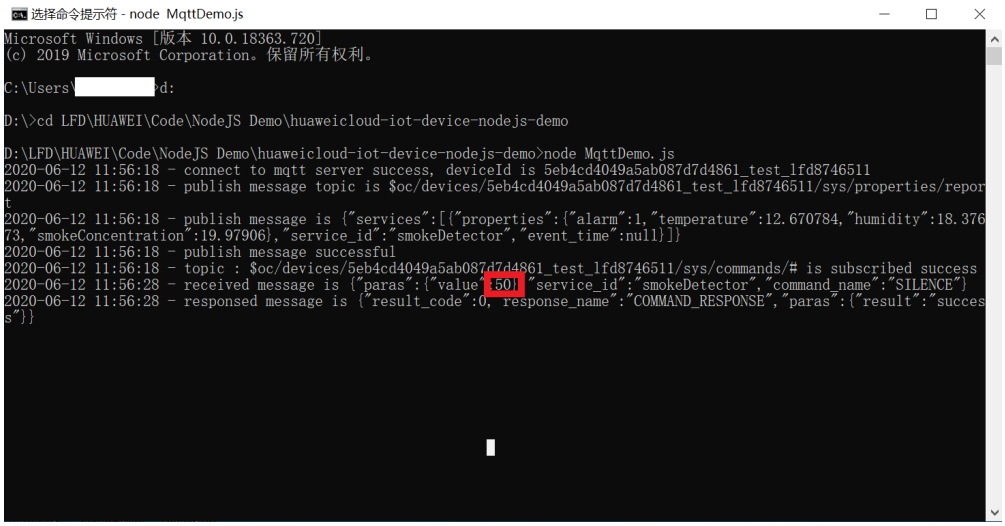
在Demo中提供了接收平台下发命令的功能，在MQTT建链完成并成功订阅Topic后，可以在**管理控制台**设备详情中命令下发或**使用应用侧Demo**对该设备ID进行命令下发。下发成功后，在Demo中接收到平台下发给设备的命令。

例如下发参数名为smokeDetector: SILENCE，参数值为50的命令。

图 3-221 命令下发-SILENCE



命令下发成功后，Demo收到的消息是50，命令运行主界面显示如下：



说明

由于是同步命令需要端侧回复响应可**参考接口**。

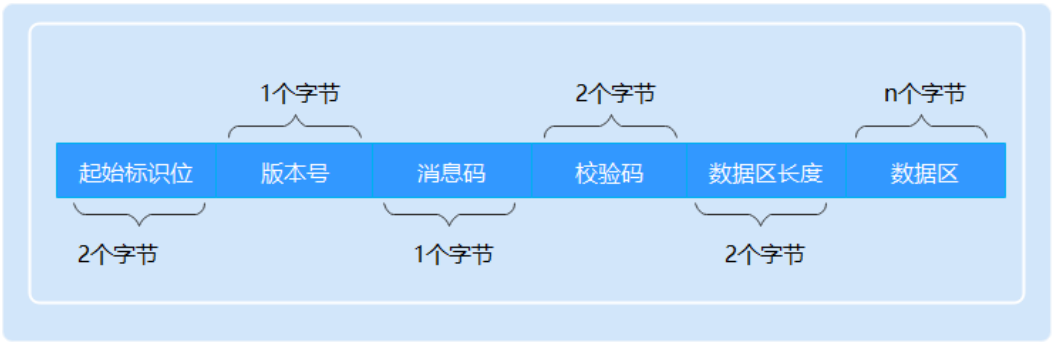
3.9 OTA 升级设备侧适配

3.9.1 设备侧适配开发指导

概述

设备的OTA软件升级是基于华为定义的**PCP协议**进行的，设备侧需根据PCP协议定义的交互流程进行适配开发。下面我们将结合物联网平台与设备的软件升级交互流程，介绍终端设备将如何基于PCP协议构建交互过程中的请求消息和应答消息，帮助您更好的根据PCP协议进行终端侧的软件升级功能开发。

下面我们先了解下PCP消息的结构，PCP协议的请求消息和应答消息都遵循相同的消息结构，主要由这几部分组成：



PCP协议消息由：起始标识位、版本号、消息码、校验码、数据区长度和数据区组成，各字段的要求和描述如下表所示。

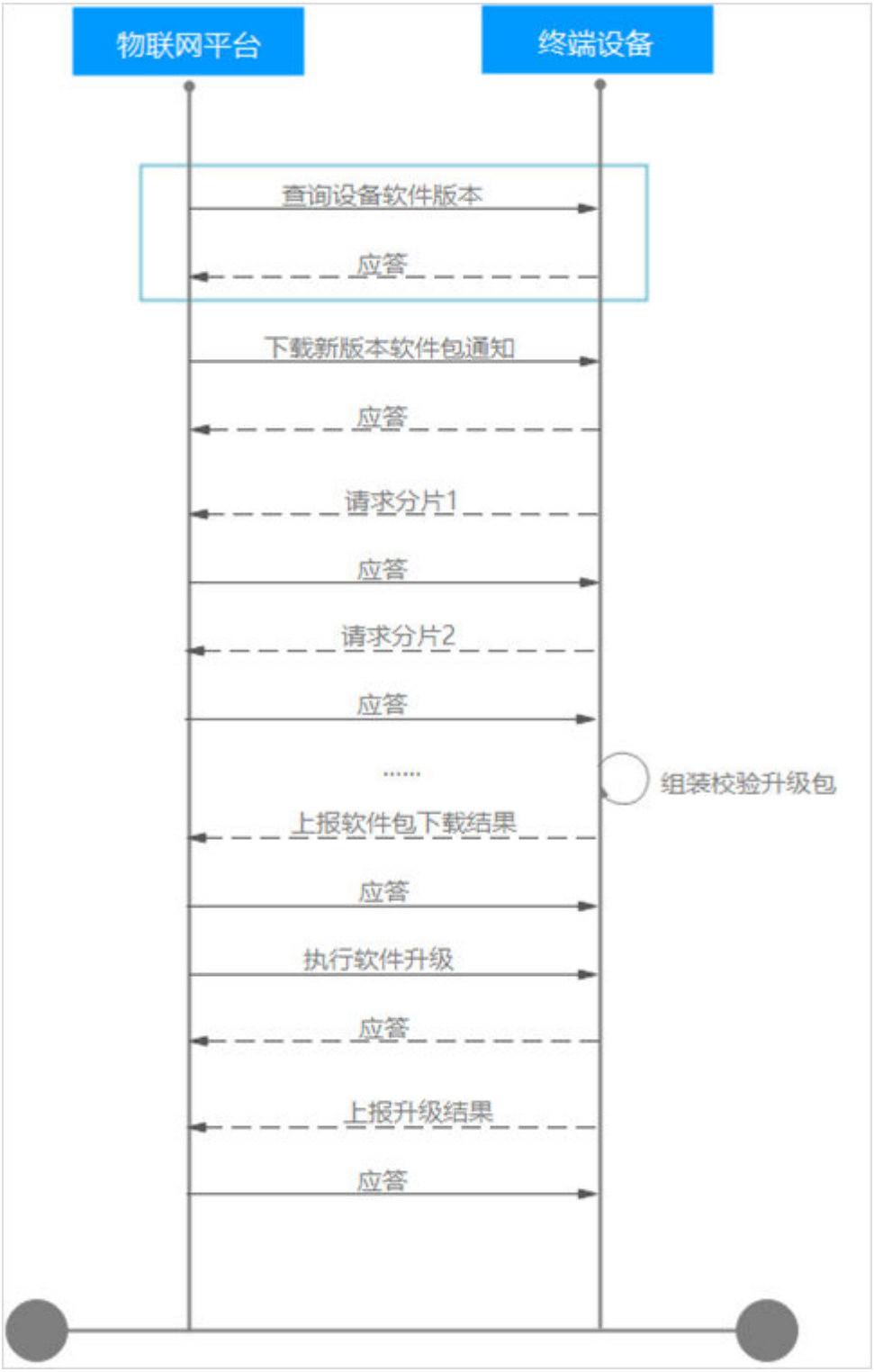
字段名	字段类型	描述和要求
起始标识	WORD	起始标识，固定为0XFFFE。

字段名	字段类型	描述和要求
版本号	BYTE	高四位预留；低四位表示协议版本号，当前为1。
消息码	BYTE	标识物联网平台与设备之间的请求消息类型，应答消息的消息码和请求消息相同。消息码的定义为： ● 0-18：预留消息码，暂未使用。 ● 19：查询设备版本。 ● 20：下载新版本软件包通知。 ● 21：请求下载升级包。 ● 22：上报升级包下载结果。 ● 23：执行软件升级。 ● 24：上报升级结果。 ● 25-127：预留消息码，暂未使用。
校验码	WORD	从起始标识到数据区的最后一个字节的CRC16校验值，计算前先把校验码字段置为0，计算完成后把结果写到校验码字段。 **说明** CRC16算法：CRC16/CCITT x16+x12+x5+1
数据区长度	WORD	数据区的长度。
数据区	BYTE[n]	可变长度，具体由各个指令定义，可参考下面介绍的各个指令对应的请求消息和应答消息定义。

数据类型	描述
BYTE	无符号一字节整数
WORD	无符号二字节整数
DWORD	无符号四字节整数
BYTE[n]	n字节的十六进制数
STRING	字符串

查询设备版本号

从设备的软件升级流程（本流程只描述物联网平台与设备基于PCP协议交互的流程）可以看到，首先物联网平台向设备下发查询版本号信息，设备进行应答。



物联网平台发送消息

根据**PCP消息结构的定义**可以得出，物联网平台向设备下发查询版本号时，各消息字段的填写如下：

- 起始标识：固定为消息流的前2个字节，固定为FFFE。
- 版本号：数据类型为1个字节整数，且固定为1，即在消息流中为01。

- 消息码：数据类型为1个字节整数，查询设备版本的消息码为19，转换为十六进制为13。
- 校验码：数据类型为2个字节整数，先将校验码置为0000，然后将完整的消息码流进行CRC16的算法计算得到校验码，再将得到的校验码替换原消息中的0000。
- 数据区长度：数据类型为2个字节整数，代表数据区的消息长度，根据数据区的数据结构可以得出该条消息无数据区，即数据区长度为0000。
- 数据区：数据区代表要真正发送给设备的数据，根据查询版本信息的数据区定义，该条消息是没有实际要传送的数据的，即无需数据区字段。

字段	数据类型	描述及要求
无数据区		

因此将查询版本消息的码流组合起来得到：FFFE 01 13 0000 0000。前面的校验码时讲了，需要将组合后的消息码流进行CRC16算法（物联网平台提供了基于JAVA和C语言的**CRC16算法代码样例**，您可以直接使用）得到校验码4C9A，然后将该校验码替换原码流中的0000后得到FFFE01134C9A0000，该消息码流即为物联网平台发送给设备的查询版本信息的消息码流。

设备返回的应答消息

设备收到物联网平台要查询设备的软件版本号消息，设备要向物联网平台反馈查询的结果，各消息字段的填写如下。

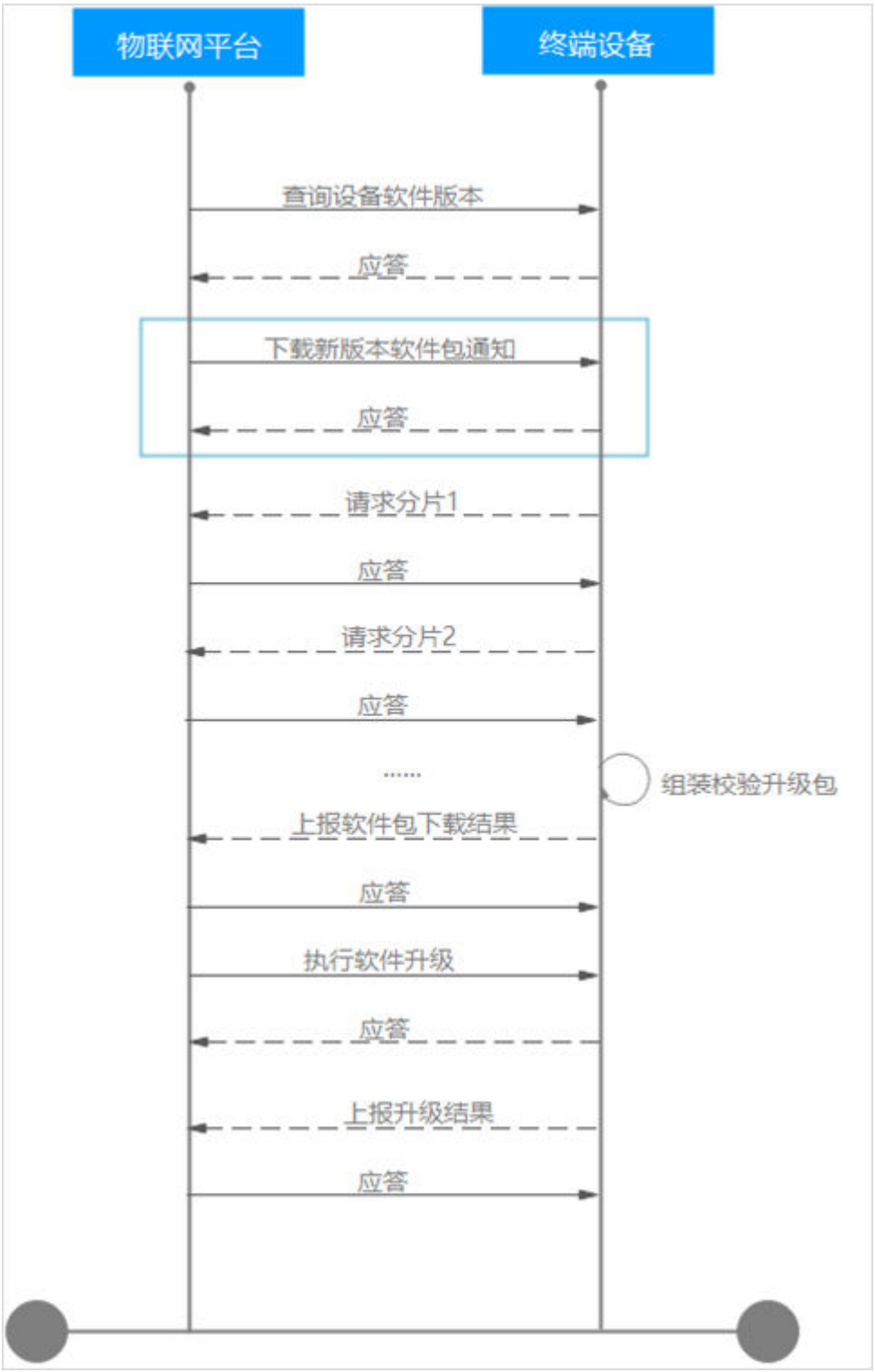
- 起始标识固定为：FFFE。
- 版本号固定为：01。
- 消息码：与请求的消息码一致，为13。
- 校验码：CRC16计算前先用0000替代。
- 数据区长度：根据数据区的字段的数据类型得出数据区长度为17个字节，转换为十六进制为：0011。
- 数据区：根据数据区的定义可知，处理成功的结果码为00，版本号信息假设为V0.9，将V0.9进行ASCII转码得到56302E39，由于版本号的数据类型为BYTE[16]，即16个字节，当前只有4个字节，因此需要在版本号数据后面补0，得到56302E39000000000000000000000000。因此，数据区合并后为0056302E39000000000000000000000000。

字段	数据类型	描述及要求
结果码	BYTE	“0X00”处理成功
当前版本号	BYTE[16]	当前版本号，由ASCII字符组成，位数不足时，后补“0X00”。

将查询版本信息的消息流组合起来得到：FFFE 01 13 0000 0011 0056302E39000000000000000000000000。前面讲到，还要将消息流进行CRC16算法计算得到校验码为8DE3。因此，物联网平台向设备查询版本号信息，设备向平台返回的消息流为FFFE01138DE300110056302E39000000000000000000000000。

下载新版本软件包通知

根据PCP协议约定的交互流程，查询完版本号后，物联网平台下发指令让设备下载新版本的软件包。



物联网平台发送消息

根据**PCP消息结构的定义**可以得出，物联网平台向设备下发下载新版本软件包通知时，各消息字段的填写如下。

- 起始标识固定为：FFFE。
- 版本号固定为：01。
- 消息码：此处为新版本通知，查询**消息码**表可以知道新版本通知为20，转换为十六进制为14。
- 校验码：CRC16计算前先用0000替代。
- 数据区长度：根据数据区的消息字段可以得出，数据区长度为22个字节，转换为十六进制为：0016。
- 数据区：根据数据区的定义可知。
  - 目标版本号：由16个字节组成，假设升级的目标版本号为v1.0版本，转换为十六进制并在后面14个字节补充0后得到：56312E300000000000000000000000。
  - 升级包分片大小：由2个字节组成，单位为byte，用户上传软件包时可以手动输入升级包分片大小，如果不设置默认为500byte，大小为32~500之间。假设为500byte，转换为十六进制后为：01F4。
  - 升级包分片总数：由2个字节组成，由软件包大小除以每个分片的大小并向上取整获得。假设软件包大小为500byte，则分片数量为1，转换为十六进制后为：0001。
  - 检验码：由2个字节组成，目前已废弃，固定为：0000。

字段	数据类型	描述及要求
目的版本号	BYTE[16]	目的版本号，由ASCII字符组成，位数不足时，后补“0X00”。
升级包分片大小	WORD	每个分片的大小
升级包分片总数	WORD	升级包分片总数
升级包校验码	WORD	固定为：0000

将下载新版本软件包通知的消息流组合起来得到：FFFE 01 14 0000 0016 56312E300000000000000000000000 01F4 0001 0000。前面说了，还要将消息流进行CRC16算法计算得到校验码为02F7。因此，物联网平台向设备通知下载新版本软件包的信息，物联网平台向设备发送的消息流为FFFE011402F7001656312E300000000000000000000000001F400010000。

设备返回的应答消息

设备收到下载新版本软件包通知后，设备向物联网平台返回应答消息，是否允许设备进行升级，各消息字段的填写如下。

- 起始标识固定为：FFFE。
- 版本号固定为：01。
- 消息码：与请求的消息码一致，为14。
- 校验码：CRC16计算前先用0000替代。
- 数据区长度：根据数据区的字段的数据类型得出数据区长度为1个字节，转换为十六进制为：0001。

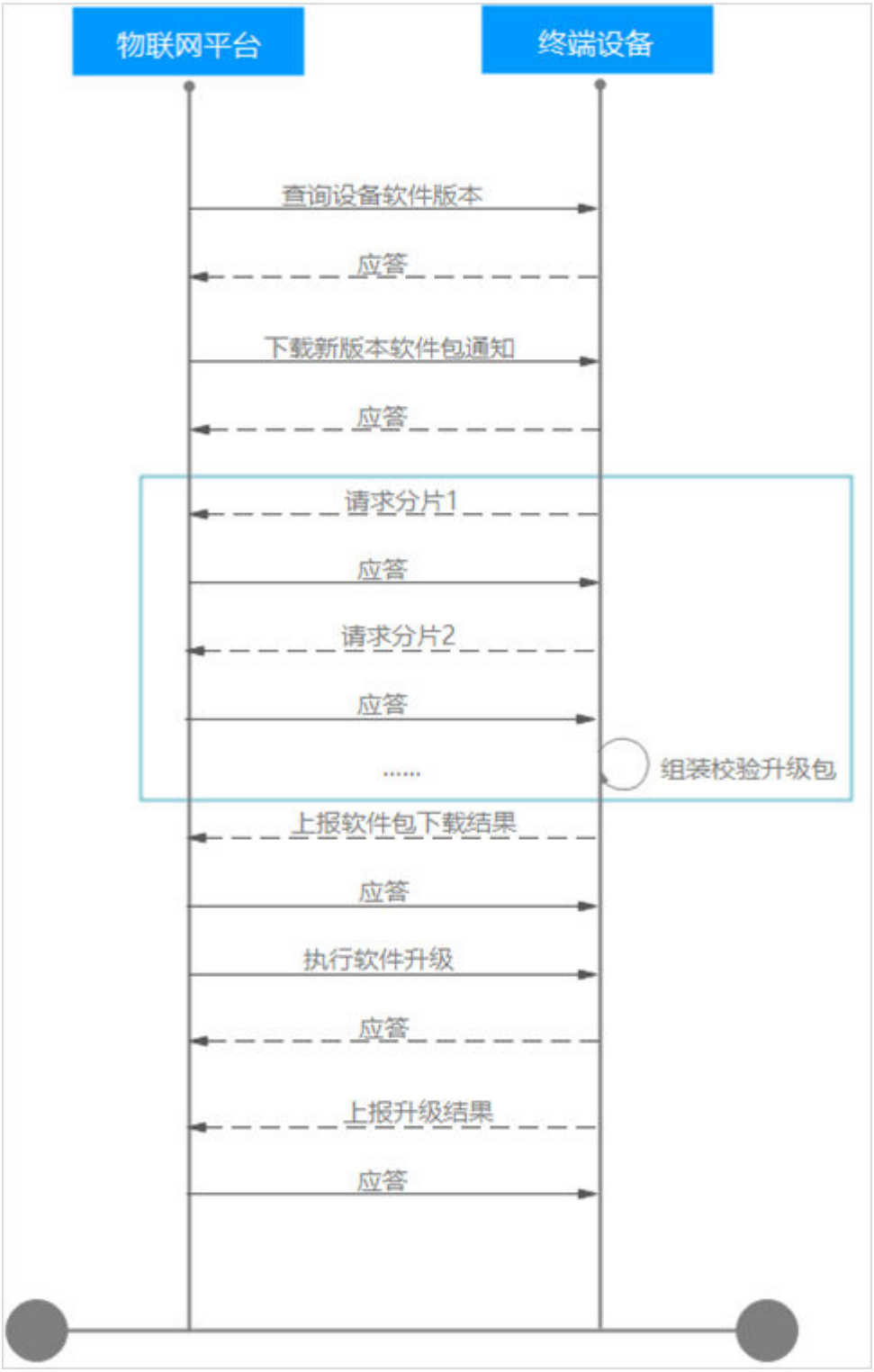
- 数据区：设备根据自身的情况对平台下发的新版本通知进行响应，本示例以设备应答“允许升级”为例进行介绍，得出数据区为：00。其它应答消息请根据应答消息字段进行适配。

字段	数据类型	描述及要求
结果码	BYTE	“0X00”允许升级 “0X01”设备使用中 “0X02”信号质量差 “0X03”已经是最新版本 “0X04”电量不足 “0X05”剩余空间不足 “0X09”内存不足 “0X7F”内部异常

将设备给物联网平台的应答消息流组合起来得到：FFFE 01 14 0000 0001 00。前面说了，还要将消息流进行CRC16算法计算得到校验码为D768。因此，设备向平台返回的应答消息流为FFFE0114D768000100。

下载升级包

根据PCP协议约定的交互流程，物联网平台通知设备有新的软件版本时，设备向物联网平台请求下载软件包，按照分片的序号进行下载。



设备发送的请求消息

根据**PCP消息结构的定义**可以得出，设备向物联网平台发送的请求软件包分片的第一条消息，各消息字段的填写如下。

- 起始标识固定为：FFFE。
- 版本号固定为：01。

- 消息码：查询**消息码**表可知请求升级包的消息码为21，转换为十六进制为：15。
- 校验码：CRC16计算前先用0000替代。
- 数据区长度：根据数据区的字段的数据类型得出数据区长度为18个字节，转换为十六进制为：0012。
- 数据区：目标版本号为平台下发的新版本通知版本号，即v1.0，转换为十六进制为56312E30000000000000000000000000，分片序号为第0个分片，即0000。

字段	数据类型	描述及要求
目的版本号	BYTE[16]	目的版本号，由ASCII字符组成，位数不足时，后补“0X00”。
分片序号	WORD	表示请求获取的分片序号，从0开始计算，分片的总数为软件包大小除以每个分片的大小并向上取整获得。设备可以保存已经收到的分片，下次直接从缺失的分片开始请求，达到断点续传的效果。

设备向物联网平台发送请求软件包分片的第一条消息为：FFFE 01 15 0000 0012 56312E30000000000000000000000000 0000（CRC16校验前），经CRC16计算得到校验码为：5618。则替换校验码后设备发送的第一条请求分片消息为：FFFE01155618001256312E30000000000000000000000000000000。

其它分片请求的消息流只需要替换分片序号后，重新计算并替换CRC16校验码即可，此处就不再展开。

物联网平台的应答消息

物联网平台收到设备的请求软件包分片消息后，将会给设备下发分片的数据。物联网平台向设备响应的第一条请求分片的消息，各消息字段的填写如下。

- 起始标识固定为：FFFE。
- 版本号固定为：01。
- 消息码：与请求的消息码一致：15。
- 校验码：CRC16计算前先用0000替代。
- 数据区：先讲数据区再讲数据区长度。结果码：00，分片序号：第0个分片：0000，分片数据：跟软件包定义的内容有关，我们假设软件包内容为HELLO, IoT SOTA!，经ASCII码转换为十六进制为：48454C4C4F2C20496F5420534F544121，共16字节。用户上传软件包时手动输入升级包分片大小为500byte，即最大长度为500字节。这种情况下，无需在数据的后面补充0。
- 数据区长度：根据数据区的字段定义得出该数据长度为19，转换为十六进制为：0013。

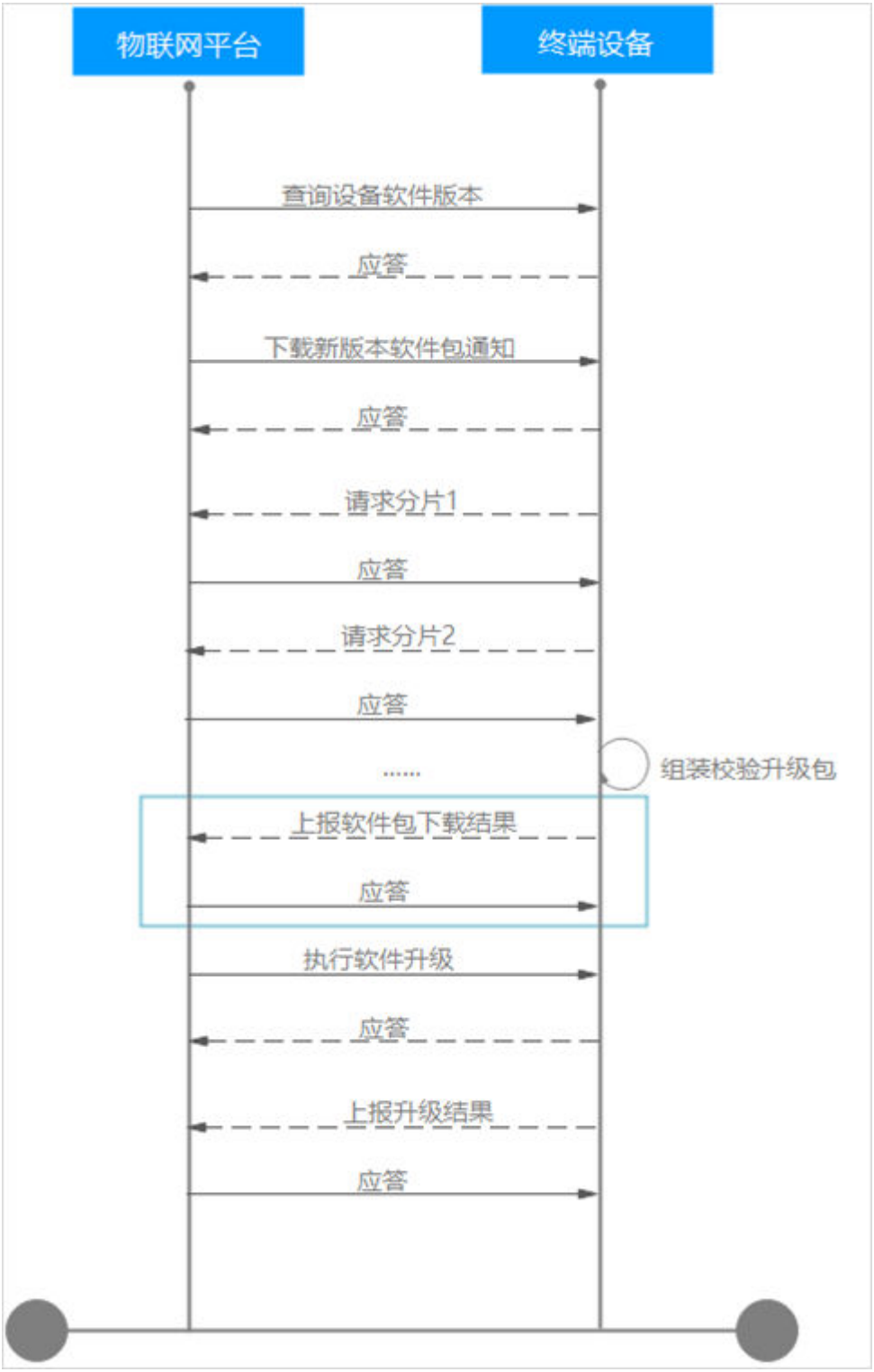
字段	数据类型	描述及要求
结果码	BYTE	0X00处理成功。 0X80升级任务不存在。 0X81指定的分片不存在。
分片序号	WORD	表示返回的分片序号。
分片数据	BYTE[n]	分片的内容，n为实际的分片大小。如果结果码不为0，则不带此字段。

物联网平台向设备发送的第一个软件包分片消息为：FFFE 01 15 0000 0013 00 0000 48454C4C4F2C20496F5420534F544121（CRC16校验前），经CRC16计算得到校验码为：E107。则替换校验码后物联网平台向设备发送的第一个软件包分片消息为：FFFE0115E107001300000048454C4C4F2C20496F5420534F544121。

其它软件包分片的消息流只需要替换分片序号和分片数据后，重新计算并替换CRC16校验码即可，此处就不再展开。

上报下载结果

根据PCP协议约定的交互流程，设备接收完所有分片数据并组装完软件包后，需要向物联网平台上报软件包的下载结果。



设备发送的请求消息

根据**PCP消息结构的定义**可以得出，设备向物联网平台发送的上报软件包下载结果消息，各个消息字段的填写如下：

- 起始标识固定为：FFFE。
- 版本号固定为：01。

- 消息码：与请求的消息码一致，为16。
- 校验码：CRC16计算前先用0000替代。
- 数据区长度：根据数据区的字段的数据类型得出数据区长度为1个字节，转换为十六进制为：0001。
- 数据区：上报软件包的下载结果，比如下载成功，设备侧上报00。

字段	数据类型	描述及要求
下载状态	BYTE	0X00下载成功。 0X05剩余空间不足。 0X06下载超时。 0X07升级包校验失败。 0X08升级包类型不支持。

设备向物联网平台发送升级包下载结果的消息为：FFFE 01 16 0000 0001 00（CRC16校验前），经CRC16计算得到校验码为：850E。则替换校验码后设备发送的升级包下载结果的消息为：FFFE0116850E000100。

物联网平台的应答消息

物联网平台收到设备上报的软件包下载结果后，将会向设备返回应答消息，各个消息字段的填写如下。

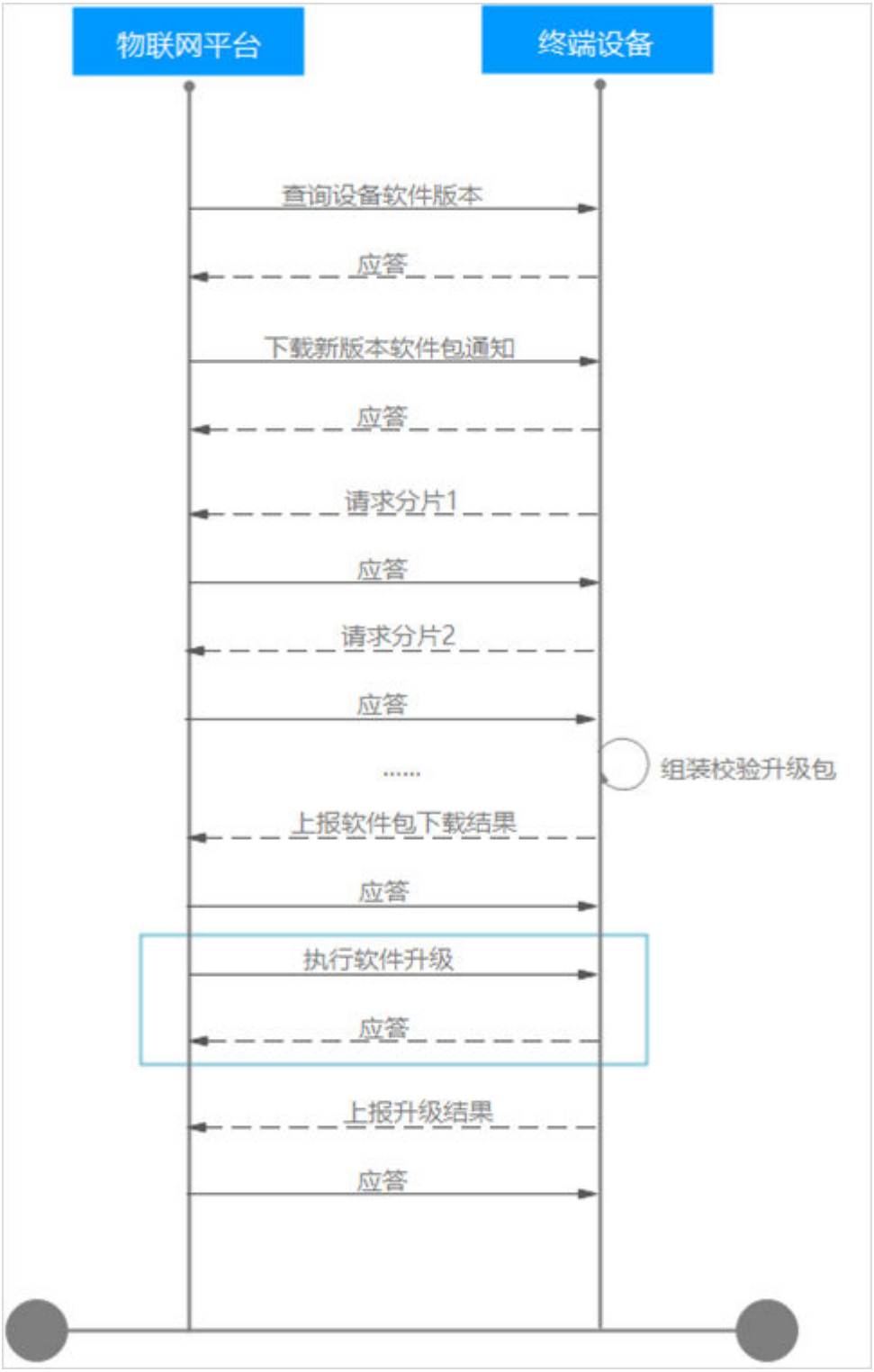
- 起始标识固定为：FFFE。
- 版本号固定为：01。
- 消息码：与请求的消息码一致：16。
- 校验码：CRC16计算前先用0000替代。
- 数据区长度：根据数据区的字段定义得出该数据长度为1个字节，转换为十六进制为：0001。
- 数据区：处理成功，则返回00，处理失败返回80。本示例以返回00处理成功为例进行说明。

字段	数据类型	描述及要求
结果码	BYTE	0X00处理成功。 0X80升级任务不存在。

物联网平台向设备应答的消息为：FFFE 01 16 0000 0001 00 （CRC16校验前），经CRC16计算得到校验码为：850E。则替换校验码后物联网平台向设备应答的消息为：FFFE0116850E000100。

执行软件升级

根据PCP协议约定的交互流程，物联网平台收到设备发送的软件包下载结果通知后，需要通知设备进行升级操作。



物联网平台发送的请求消息

根据**PCP消息结构的定义**可以得出，物联网平台向设备发送执行软件升级消息，各个消息字段的填写如下：

- 起始标识固定为：FFFE。
- 版本号固定为：01。

- 消息码：与请求的消息码一致，为17。
- 校验码：CRC16计算前先用0000替代。
- 数据区长度：根据数据区的字段的数据类型得出无数据区，即为0字节，转换为十六进制为：0000。
- 数据区：无数据区，无需携带该字段。

字段	数据类型	描述及要求
无数据区		

物联网平台向设备下发的执行软件升级的消息为：FFFE 01 17 0000 0000（CRC16校验前），经CRC16计算得到校验码为：CF90。则替换校验码后物联网平台向设备发送的消息为：FFFE0117CF900000。

设备发送的应答消息

设备收到物联网平台下发的执行升级消息后，将对收到消息后的执行动作进行应答，各消息字段的填写如下。

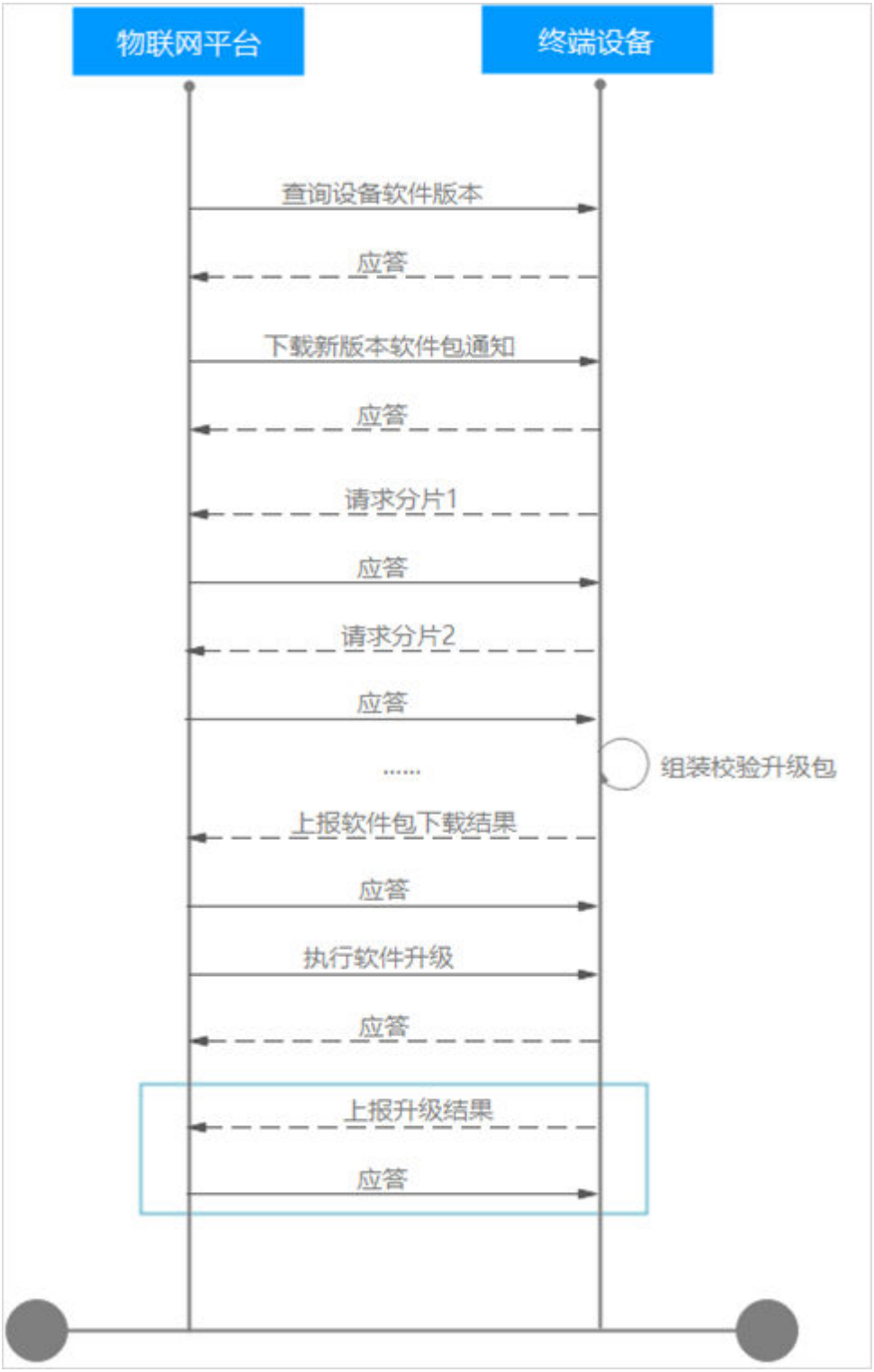
- 起始标识固定为：FFFE。
- 版本号固定为：01。
- 消息码：与请求的消息码一致：17。
- 校验码：CRC16计算前先用0000替代。
- 数据区长度：根据数据区的字段定义得出该数据长度为1个字节，转换为十六进制为：0001。
- 数据区：处理成功，则返回00，其它处理结果请参考数据区定义。本示例以返回00处理成功为例进行说明。

字段	数据类型	描述及要求
结果码	BYTE	0X00处理成功。 0X01设备使用中。 0X04电量不足。 0X05剩余空间不足。 0X09内存不足。

设备向物联网平台应答的消息为：FFFE 01 17 0000 0001 00 （CRC16校验前），经CRC16计算得到校验码为：B725。则替换校验码后设备返回的响应消息为：FFFE0117B725000100 。

上报升级结果

根据PCP协议约定的交互流程，设备在执行完软件升级后，将会向物联网平台上报升级的结果。



设备发送的请求消息

根据**PCP消息结构的定义**可以得出，设备向物联网平台上报升级结果，各个消息字段的填写如下：

- 起始标识固定为：FFFE。
- 版本号固定为：01。

- 消息码：与请求的消息码一致，为18。
- 校验码：CRC16计算前先用0000替代。
- 数据区长度：根据数据区的字段的数据类型得出数据区长度为17字节，转换为十六进制为：0011。
- 数据区：结果码，以上报升级成功为例，结果码为00。当前版本号：升级完成后的版本号，与物联网平台下发的软件版本一致，即v1.0，转换为十六进制为：56312E300000000000000000000000000。

字段	数据类型	描述及要求
结果码	BYTE	0X00升级成功。 0X01设备使用中。 0X04电量不足。 0X05剩余空间不足。 0X09内存不足。 0X0A安装升级包失败。 0X7F内部异常。
当前版本号	BYTE[16]	设备当前版本号。

设备向物联网平台上报升级结果的消息为：FFFE 01 18 0000 0011 0056312E300000000000000000000000000（CRC16校验前），经CRC16计算得到校验码为：C7D2。则替换校验码后设备向物联网平台上报升级结果码流为：FFFE0118C7D200110056312E300000000000000000000000000。

物联网平台发送的应答消息

物联网平台收到设备上报的升级结果消息后，将对设备进行应答，各个消息字段的填写如下。

- 起始标识固定为：FFFE。
- 版本号固定为：01。
- 消息码：与请求的消息码一致：18。
- 校验码：CRC16计算前先用0000替代。
- 数据区长度：根据数据区的字段定义得出该数据长度为1个字节，转换为十六进制为：0001。
- 数据区：处理成功，则返回00，升级任务不存在80。本示例以返回00处理成功为例进行说明。

字段	数据类型	描述及要求
结果码	BYTE	0X00处理成功。 0X80升级任务不存在。

物联网平台向设备的应答消息为：FFFE 01 18 0000 0001 00 （CRC16校验前），经CRC16计算得到校验码为：AFA1。则替换校验码后物联网平台返回的应答消息为：FFFE0118AFA1000100 。

CRC16 算法代码样例

基于JAVA的CRC16算法样例：

```
public class CRC16 {

    /*
     * CCITT标准CRC16(1021)余数表 CRC16-CCITT ISO HDLC, ITU X.25, x16+x12+x5+1 多项式
     * 高位在先时生成多项式 Gm=0x11021 低位在先时生成多项式，Gm=0x8408 本例采用高位在先
     */
    private static int[] crc16_ccitt_table = { 0x0000, 0x1021, 0x2042, 0x3063, 0x4084, 0x50a5, 0x60c6, 0x70e7,
            0x8108, 0x9129, 0xa14a, 0xb16b, 0xc18c, 0xd1ad, 0xe1ce, 0xf1ef, 0x1231, 0x0210, 0x3273, 0x2252,
            0x52b5, 0x4294, 0x72f7, 0x62d6, 0x9339, 0x8318, 0xb37b, 0xa35a, 0xd3bd, 0xc39c, 0xf3ff, 0xe3de,
            0x2462, 0x3443, 0x0420, 0x1401, 0x64e6, 0x74c7, 0x44a4, 0x5485, 0xa56a, 0xb54b, 0x8528, 0x9509,
            0xe5ee, 0xf5cf, 0xc5ac, 0xd58d, 0x3653, 0x2672, 0x1611, 0x0630, 0x76d7, 0x66f6, 0x5695, 0x46b4,
            0xb75b, 0xa77a, 0x9719, 0x8738, 0xf7df, 0xe7fe, 0xd79d, 0xc7bc, 0x48c4, 0x58e5, 0x6886, 0x78a7,
            0x0840, 0x1861, 0x2802, 0x3823, 0xc9cc, 0xd9ed, 0xe98e, 0xf9af, 0x8948, 0x9969, 0xa90a, 0xb92b,
            0x5af5, 0x4ad4, 0x7ab7, 0x6a96, 0x1a71, 0x0a50, 0x3a33, 0x2a12, 0xdbfd, 0xcbdc, 0xfbbf, 0xeb9e,
            0x9b79, 0x8b58, 0xbb3b, 0xab1a, 0x6ca6, 0x7c87, 0x4ce4, 0x5cc5, 0x2c22, 0x3c03, 0x0c60, 0x1c41,
            0xedae, 0xfd8f, 0xcdec, 0xddcd, 0xad2a, 0xbd0b, 0x8d68, 0x9d49, 0x7e97, 0x6eb6, 0x5ed5, 0x4ef4,
            0x3e13, 0x2e32, 0x1e51, 0x0e70, 0xff9f, 0xefbe, 0xdfdd, 0xcffc, 0xbf1b, 0xaf3a, 0x9f59, 0x8f78,
            0x9188, 0x81a9, 0xb1ca, 0xa1eb, 0xd10c, 0xc12d, 0xf14e, 0xe16f, 0x1080, 0x00a1, 0x30c2, 0x20e3,
            0x5004, 0x4025, 0x7046, 0x6067, 0x83b9, 0x9398, 0xa3fb, 0xb3da, 0xc33d, 0xd31c, 0xe37f, 0xf35e,
            0x02b1, 0x1290, 0x22f3, 0x32d2, 0x4235, 0x5214, 0x6277, 0x7256, 0xb5ea, 0xa5cb, 0x95a8, 0x8589,
            0xf56e, 0xe54f, 0xd52c, 0xc50d, 0x34e2, 0x24c3, 0x14a0, 0x0481, 0x7466, 0x6447, 0x5424, 0x4405,
            0xa7db, 0xb7fa, 0x8799, 0x97b8, 0xe75f, 0xf77e, 0xc71d, 0xd73c, 0x26d3, 0x36f2, 0x0691, 0x16b0,
            0x6657, 0x7676, 0x4615, 0x5634, 0xd94c, 0xc96d, 0xf90e, 0xe92f, 0x99c8, 0x89e9, 0xb98a, 0xa9ab,
            0x5844, 0x4865, 0x7806, 0x6827, 0x18c0, 0x08e1, 0x3882, 0x28a3, 0xcb7d, 0xdb5c, 0xeb3f, 0xfb1e,
            0x8bf9, 0x9bd8, 0xabbb, 0xbb9a, 0x4a75, 0x5a54, 0x6a37, 0x7a16, 0x0af1, 0x1ad0, 0x2ab3, 0x3a92,
            0xfd2e, 0xed0f, 0xdd6c, 0xcd4d, 0xbdaa, 0xad8b, 0x9de8, 0x8dc9, 0x7c26, 0x6c07, 0x5c64, 0x4c45,
            0x3ca2, 0x2c83, 0x1ce0, 0x0cc1, 0xef1f, 0xff3e, 0xcf5d, 0xdf7c, 0xaf9b, 0xbfba, 0x8fd9, 0x9ff8,
            0x6e17, 0x7e36, 0x4e55, 0x5e74, 0x2e93, 0x3eb2, 0x0ed1, 0x1ef0 };

    /**
     *
     * @param reg_init
     *            CRC校验时初值
     * @param message
     *            校验值
     * @return
     */
    private static int do_crc(int reg_init, byte[] message) {
        int crc_reg = reg_init;
        for (int i = 0; i < message.length; i++) {
            crc_reg = (crc_reg >> 8) ^ crc16_ccitt_table[(crc_reg ^ message[i]) & 0xff];
        }
        return crc_reg;
    }

    /**
     * 根据数据生成CRC校验码
     *
     * @param message
     *            byte数据
     *
     * @return int 返校验码
     */
    public static int do_crc(byte[] message) {
        // 计算CRC校验时初值从0x0000开始。
        int crc_reg = 0x0000;
        return do_crc(crc_reg, message);
    }
}
```

基于C的CRC16算法样例：

```
/**
* CCITT标准CRC16(1021)余数表 CRC16-CCITT ISO HDLC, ITU X.25, x16+x12+x5+1 多项式
* 高位在先时生成多项式 Gm=0x11021 低位在先时生成多项式，Gm=0x8408 本例采用高位在先
```

```
*/
const unsigned short crc16_table[256] = {
    0x0000, 0x1021, 0x2042, 0x3063, 0x4084, 0x50a5, 0x60c6, 0x70e7,
    0x8108, 0x9129, 0xa14a, 0xb16b, 0xc18c, 0xd1ad, 0xe1ce, 0xf1ef,
    0x1231, 0x0210, 0x3273, 0x2252, 0x52b5, 0x4294, 0x72f7, 0x62d6,
    0x9339, 0x8318, 0xb37b, 0xa35a, 0xd3bd, 0xc39c, 0xf3ff, 0xe3de,
    0x2462, 0x3443, 0x0420, 0x1401, 0x64e6, 0x74c7, 0x44a4, 0x5485,
    0xa56a, 0xb54b, 0x8528, 0x9509, 0xe5ee, 0xf5cf, 0xc5ac, 0xd58d,
    0x3653, 0x2672, 0x1611, 0x0630, 0x76d7, 0x66f6, 0x5695, 0x46b4,
    0xb75b, 0xa77a, 0x9719, 0x8738, 0xf7df, 0xe7fe, 0xd79d, 0xc7bc,
    0x48c4, 0x58e5, 0x6886, 0x78a7, 0x0840, 0x1861, 0x2802, 0x3823,
    0xc9cc, 0xd9ed, 0xe98e, 0xf9af, 0x8948, 0x9969, 0xa90a, 0xb92b,
    0x5af5, 0x4ad4, 0x7ab7, 0x6a96, 0x1a71, 0x0a50, 0x3a33, 0x2a12,
    0xdbfd, 0xcbdc, 0xfbbf, 0xeb9e, 0x9b79, 0x8b58, 0xbb3b, 0xab1a,
    0x6ca6, 0x7c87, 0x4ce4, 0x5cc5, 0x2c22, 0x3c03, 0x0c60, 0x1c41,
    0xedae, 0xfd8f, 0xcdec, 0xddcd, 0xad2a, 0xbd0b, 0x8d68, 0x9d49,
    0x7e97, 0x6eb6, 0x5ed5, 0x4ef4, 0x3e13, 0x2e32, 0x1e51, 0x0e70,
    0xff9f, 0xefbe, 0xdfdd, 0xcffc, 0xbf1b, 0xaf3a, 0x9f59, 0x8f78,
    0x9188, 0x81a9, 0xb1ca, 0xa1eb, 0xd10c, 0xc12d, 0xf14e, 0xe16f,
    0x1080, 0x00a1, 0x30c2, 0x20e3, 0x5004, 0x4025, 0x7046, 0x6067,
    0x83b9, 0x9398, 0xa3fb, 0xb3da, 0xc33d, 0xd31c, 0xe37f, 0xf35e,
    0x02b1, 0x1290, 0x22f3, 0x32d2, 0x4235, 0x5214, 0x6277, 0x7256,
    0xb5ea, 0xa5cb, 0x95a8, 0x8589, 0xf56e, 0xe54f, 0xd52c, 0xc50d,
    0x34e2, 0x24c3, 0x14a0, 0x0481, 0x7466, 0x6447, 0x5424, 0x4405,
    0xa7db, 0xb7fa, 0x8799, 0x97b8, 0xe75f, 0xf77e, 0xc71d, 0xd73c,
    0x26d3, 0x36f2, 0x0691, 0x16b0, 0x6657, 0x7676, 0x4615, 0x5634,
    0xd94c, 0xc96d, 0xf90e, 0xe92f, 0x99c8, 0x89e9, 0xb98a, 0xa9ab,
    0x5844, 0x4865, 0x7806, 0x6827, 0x18c0, 0x08e1, 0x3882, 0x28a3,
    0xcb7d, 0xdb5c, 0xeb3f, 0xfb1e, 0x8bf9, 0x9bd8, 0xabbb, 0xbb9a,
    0x4a75, 0x5a54, 0x6a37, 0x7a16, 0x0af1, 0x1ad0, 0x2ab3, 0x3a92,
    0xfd2e, 0xed0f, 0xdd6c, 0xcd4d, 0xbdaa, 0xad8b, 0x9de8, 0x8dc9,
    0x7c26, 0x6c07, 0x5c64, 0x4c45, 0x3ca2, 0x2c83, 0x1ce0, 0x0cc1,
    0xef1f, 0xff3e, 0xcf5d, 0xdf7c, 0xaf9b, 0xbfba, 0x8fd9, 0x9ff8,
    0x6e17, 0x7e36, 0x4e55, 0x5e74, 0x2e93, 0x3eb2, 0x0ed1, 0x1ef0
};

int do_crc(int reg_init, byte* data, int length)
{
    int cnt;
    int crc_reg = reg_init;
    for (cnt = 0; cnt < length; cnt++)
    {
        crc_reg = (crc_reg >> 8) ^ crc16_table[(crc_reg ^ *(data++)) & 0xFF];
    }
    return crc_reg;
}

int main(int argc, char **argv)
{
    //FFFE011300000000用byte数组表示：
    byte message[8] = {0xFF,0xFE,0x01,0x13,0x00,0x00,0x00,0x00};
    // 计算CRC校验时初值从0x0000开始
    int a = do_crc(0x0000, message, 8);
    printf("a ==> %x\n", a);
}
```

相关 FAQ

OTA升级相关问题

相关最佳实践

MQTT协议设备OTA固件升级

3.9.2 PCP 协议介绍

平台升级协议（PCP协议）规定了设备和平台之间升级的通信内容与格式。

本协议规定设备和IoT平台（以下简称“平台”）之间的应用层升级协议（简称“PCP协议”），用于实现设备的升级。

通讯方式

1. PCP协议运行在应用层，底层可以是LwM2M/CoAP/MQTT或者其他非流式协议。
2. 由于PCP协议消息没有使用单独的端口号，并且不依赖于底层协议，为了和设备业务消息区分，PCP协议固定以0XFFFE作为起始字节。因此要求设备的业务消息的前两个字节不能是0XFFFE，更多细节参考附录**PCP消息识别**。
3. 本协议消息采用一问一答模式，所有请求消息都有一个响应消息。

消息结构

字段名	字段类型	描述和要求
起始标识	WORD	起始标识，固定为0XFFFE。
版本号	BYTE	高四位预留；低四位表示协议版本号，当前为1。
消息码	BYTE	标识物联网平台与设备之间的请求消息类型，应答消息的消息码和请求消息相同。消息码的定义为： ● 0-18：预留消息码，暂未使用。 ● 19：查询设备版本。 ● 20：下载新版本软件包通知。 ● 21：请求下载升级包。 ● 22：上报升级包下载结果。 ● 23：执行软件升级。 ● 24：上报升级结果。 ● 25-127：预留消息码，暂未使用。
校验码	WORD	从起始标识到数据区的最后一个字节的CRC16校验值，计算前先把校验码字段置为0，计算完成后把结果写到校验码字段。 **说明** CRC16算法：CRC16/CCITT x16+x12+x5+1
数据区长度	WORD	数据区的长度。
数据区	BYTE[n]	可变长度，具体由各个指令定义，可参考下面介绍的各个指令对应的请求消息和应答消息定义。

数据类型

数据类型	描述
BYTE	无符号一字节整数
WORD	无符号二字节整数
DWORD	无符号四字节整数
BYTE[n]	n字节的十六进制数
STRING	字符串

说明

本协议采用网络序来传输WORD和DWORD。

查询设备版本消息

请求消息：

方向：平台->设备

字段	数据类型	描述及要求
无数据区		

响应消息：

方向：设备->平台

字段	数据类型	描述及要求
结果码	BYTE	“0X00”处理成功
当前版本号	BYTE[16]	当前版本号，由ASCII字符组成，位数不足时，后补“0X00”。

说明

- 正常处理：平台根据版本号判断设备是否需要升级，如果需要，下发请求升级。
- 异常处理：如果响应超时，平台中止升级任务。

新版本通知消息

请求消息：

方向：平台->设备

字段	数据类型	描述及要求
目的版本号	BYTE[16]	目的版本号，由ASCII字符组成，位数不足时，后补“0X00”。
升级包分片大小	WORD	每个分片的大小
升级包分片总数	WORD	升级包分片总数
升级包校验码	WORD	固定为：0000

应答消息：

方向：设备->平台

字段	数据类型	描述及要求
结果码	BYTE	“0X00”允许升级 “0X01”设备使用中 “0X02”信号质量差 “0X03”已经是最新版本 “0X04”电量不足 “0X05”剩余空间不足 “0X09”内存不足 “0X7F”内部异常

说明

- 正常处理：如果设备不允许升级，平台中止升级任务。
- 异常处理：如果响应超时，而且没收到请求升级包消息，平台中止升级任务。

请求消息包消息

请求消息：

方向：设备->平台

字段	数据类型	描述及要求
目的版本号	BYTE[16]	目的版本号，由ASCII字符组成，位数不足时，后补“0X00”。

字段	数据类型	描述及要求
分片序号	WORD	表示请求获取的分片序号，从0开始计算，分片的总数为软件包大小除以每个分片的大小并向上取整获得。设备可以保存已经收到的分片，下次直接从缺失的分片开始请求，达到断点续传的效果。

响应消息：

方向：平台->设备

字段	数据类型	描述及要求
结果码	BYTE	0X00处理成功。 0X80升级任务不存在。 0X81指定的分片不存在。
分片序号	WORD	表示返回的分片序号。
分片数据	BYTE[n]	分片的内容，n为实际的分片大小。如果结果码不为0，则不带此字段。

上报升级包下载状态消息

请求消息：

方向：设备->平台

字段	数据类型	描述及要求
下载状态	BYTE	0X00下载成功。 0X05剩余空间不足。 0X06下载超时。 0X07升级包校验失败。 0X08升级包类型不支持。

响应消息：

方向：平台->设备

字段	数据类型	描述及要求
结果码	BYTE	0X00处理成功。 0X80升级任务不存在。

执行升级消息

请求消息：

方向：平台->设备

字段	数据类型	描述及要求
无数据区		

响应消息：

方向：设备->平台

字段	数据类型	描述及要求
结果码	BYTE	0X00处理成功。 0X01设备使用中。 0X04电量不足。 0X05剩余空间不足。 0X09内存不足。

上报升级结果消息

请求消息：

方向：设备->平台

字段	数据类型	描述及要求
结果码	BYTE	0X00升级成功。 0X01设备使用中。 0X04电量不足。 0X05剩余空间不足。 0X09内存不足。 0X0A安装升级包失败。 0X7F内部异常。
当前版本号	BYTE[16]	设备当前版本号。

响应消息：

方向：平台->设备

字段	数据类型	描述及要求
结果码	BYTE	0X00处理成功。 0X80升级任务不存在。

PCP 消息识别

由于PCP协议消息和设备业务消息共用一个端口和URL通讯，平台收到设备的消息时，按照如下步骤判断是PCP协议消息还是业务消息：

1. 检查设备是否支持软件升级（根据设备profile的omCapability.upgradeCapability定义），如果不支持，则认为是业务消息。
2. 检查设备软件升级协议是否是PCP，如果不是，则认为是业务消息。
3. 检查消息前两个字节是否为0XFFFE，如果不是，则认为是业务消息。
4. 检查版本号是否合法，如果不合法，则认为是业务消息。
5. 检查消息码是否合法，如果不合法，则认为是业务消息。
6. 检查校验码是否正确，如果不正确，则认为是业务消息。
7. 检查数据区长度是否正确，如果不正确，则认为是业务消息。
8. 如果以上检查都通过，认为是PCP协议消息。

说明

对设备的要求：需要设备保证业务消息的起始字节不是0XFFFE。

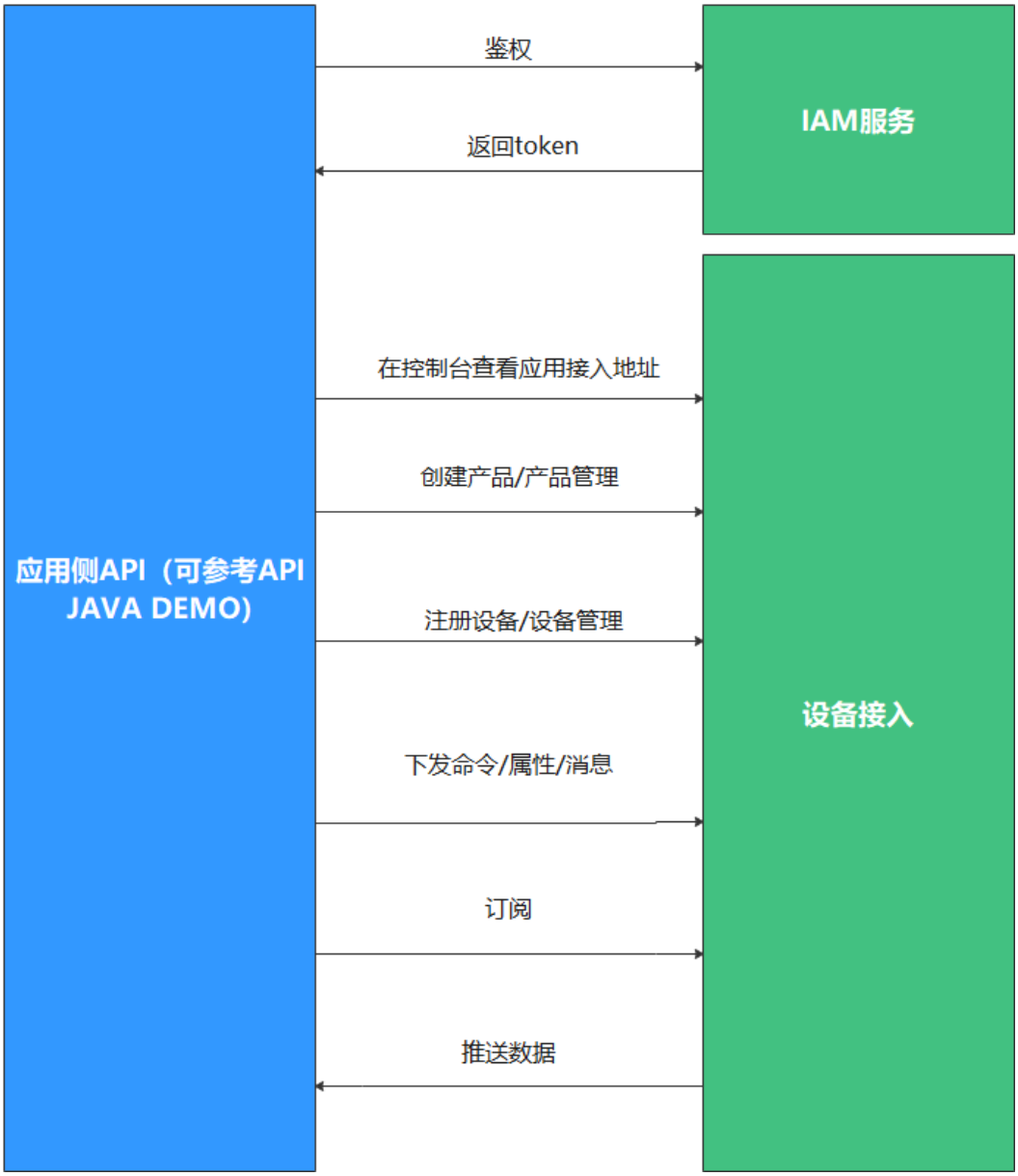
4 应用侧集成开发

4.1 API 使用指导

为了降低应用侧的开发难度、提升应用侧开发效率，物联网平台向应用侧开放了API（Application Programming Interface）。您可以调用开放的API，快速集成物联网平台的功能，如产品管理、设备管理、订阅管理、设备命令、规则管理等功能。

说明

应用侧需要通过IAM服务鉴权，获取token，详细步骤可参考**调测“获取IAM用户Token”接口**。



应用开发资源

为了降低应用的开发难度、提升开发效率，物联网平台开放了应用侧API。应用通过调用物联网平台的API，实现安全接入、设备管理、数据采集、命令下发等业务场景。

资源包名	描述	下载
应用侧开发 API Java Demo	物联网平台为应用服务器提供了**应用侧API**，能够让开发者快速验证API开放的能力，体验业务功能，熟悉业务流程。	**API Java Demo**

资源包名	描述	下载
应用侧 Java SDK	应用侧Java SDK提供Java方法调用**应用侧API**与平台通信。使用指南可以参考**应用侧Java SDK使用指南**。	**应用侧Java SDK**
应用侧 .Net SDK	应用侧.Net SDK提供.Net方法调用**应用侧API**与平台通信。使用指南可以参考**应用侧.Net SDK使用指南**。	**应用侧.Net SDK**
应用侧 Python SDK	应用侧Python SDK提供Python方法调用**应用侧API**与平台通信。使用指南可以参考**应用侧Python SDK使用指南**。	**应用侧Python SDK**
应用侧 Go SDK	应用侧Go SDK提供Go方法调用**应用侧API**与平台通信。使用指南可以参考**应用侧Go SDK使用指南**。	**应用侧Go SDK**
应用侧 Node.js SDK	应用侧Node.js SDK提供Node.js方法调用**应用侧API**与平台通信。使用指南可以参考**应用侧Node.js SDK使用指南**。	**应用侧Node.js SDK**
应用侧 PHP SDK	应用侧PHP SDK提供PHP方法调用**应用侧API**与平台通信。使用指南可以参考**应用侧PHP SDK使用指南**。	**应用侧PHP SDK**

接口介绍

API分组	应用场景
产品管理	产品模型定义了该产品下所有设备具备的能力或特征，产品管理为应用服务器提供对已导入物联网平台中产品模型的操作管理功能。
设备管理	设备管理为应用服务器提供对设备的操作管理功能，包括对设备基本信息和设备数据的操作。
设备消息	设备消息为应用服务器提供向设备透传消息的功能。
设备命令	设备的产品模型中定义了物联网平台可向设备下发的命令，设备命令为应用服务器提供向设备下发命令的功能，实现对设备的控制操作。

API分组	应用场景
设备属性	设备的产品模型中定义了物联网平台可向设备下发的属性，设备属性为应用服务器提供向设备下发属性的功能。
AMQP队列管理	AMQP队列管理为客户创建、删除、查看队列。AMQP队列可通过规则订阅后通过AMQP客户端接收消息数据。
接入凭证管理	接入凭证是用于AMQP、MQTTS等协议建立长连接时认证使用。
数据转发、设备联动	规则管理为应用服务器提供物联网平台的规则引擎功能，通过设置规则实现业务的联动变化或将数据转发至其他华为云服务。包含设备联动和数据转发两种类型。 • 设备联动：包含触发条件和执行动作两部分。当满足设置的触发条件后，触发相应动作，如“下发命令”、“发送通知”、“上报告警”、“恢复告警”。 • 数据转发：包含设置转发数据、设置转发目标和启动规则三部分。支持转发至“数据接入服务DIS”、“分布式消息服务Kafka”、“对象存储服务 OBS”、“应用与数据集成平台ROMA Connect”、“第三方应用服务（HTTP推送）”、“AMQP推送消息队列”。
订阅管理	订阅管理为应用服务器提供对物联网平台资源的订阅功能，若订阅的资源发生变化，平台会通知应用服务器。
设备影子	设备影子是一个用于存储和检索设备当前状态信息的文件，设备影子为应用服务器提供对设备影子的操作管理功能。 • 每个设备有且只有一个设备影子，由设备ID唯一标识。 • 设备影子仅保存最近一次设备的上报数据和用户设置的预期数据。 • 无论该设备是否在线，都可以通过该影子查询和设置设备的状态。
设备组管理	设备组管理为应用服务器提供对设备组的管理操作功能，包括对设备组信息和设备组设备的操作。
标签管理	标签可用于对资源进行分类，标签管理为应用服务器提供对各类资源绑定和解绑标签功能。 当前仅设备支持标签。
资源空间管理	资源空间管理为应用服务器提供对资源空间的管理能力，包括资源空间的增删改查。
批量任务	批量任务为应用服务器提供批量处理功能，对接入物联网平台的设备进行批量操作。 • 目前提供批量软、固件升级，批量创建/删除/更新设备，批量冻结/解冻设备，批量创建同步/异步命令，批量创建消息和批量配置设备影子的能力。 • 当前单用户单一任务类型的未完成的任务最大为10，超过则无法创建新的任务。

API分组	应用场景
设备CA证书管理	设备CA证书管理为应用服务器提供对设备CA证书进行操作管理功能，包括对设备CA证书进行上传、验证、查询等操作。物联网平台支持使用证书进行设备接入认证。
OTA升级包管理	OTA升级包管理为应用服务器提供对升级包进行操作管理功能，包括对升级包的创建、查询、删除等操作。
广播消息	广播消息为应用服务器向订阅了指定Topic的所有在线设备发布消息。
设备隧道管理	设备隧道可用于应用服务器与设备进行数据传输。
数据流转积压策略管理	数据流转积压策略管理为应用服务器提供了对积压策略的管理操作功能，包括对数据流转积压策略的创建，查询，修改删除等操作功能。
数据流转流控策略管理	数据流转流控策略管理为应用服务器提供了对流控策略的管理操作功能，包括对数据流转流控策略的创建，查询，修改删除等操作功能。

相关 FAQ

应用集成相关问题

消息通信相关问题

订阅推送相关问题

应用服务器如何通过IoTDA获取设备数据?

IAM用户访问API提示没有权限?

4.2 使用 Postman 调测

概述

Postman是网页调试与辅助接口调用的工具，具有界面简洁清晰、操作方便快捷的特性，可以处理用户发送的HTTP请求，例如：GET，PUT、POST，DELETE等，支持用户修改HTTP请求中的参数并返回响应数据。

为充分了解接口，建议提前获取**应用侧API参考**查阅。我们已经写好了Postman的collection，在Collection中接口的请求结构体已经完成可以直接使用。

本文档以Postman为例，模拟应用服务器以HTTPS协议接入物联网平台，调测以下API接口：

- “获取IAM用户Token”接口
- “查询IAM用户可以访问的项目列表”接口
- “创建产品”接口

- “查询产品”接口
- “创建设备”接口
- “查询设备”接口

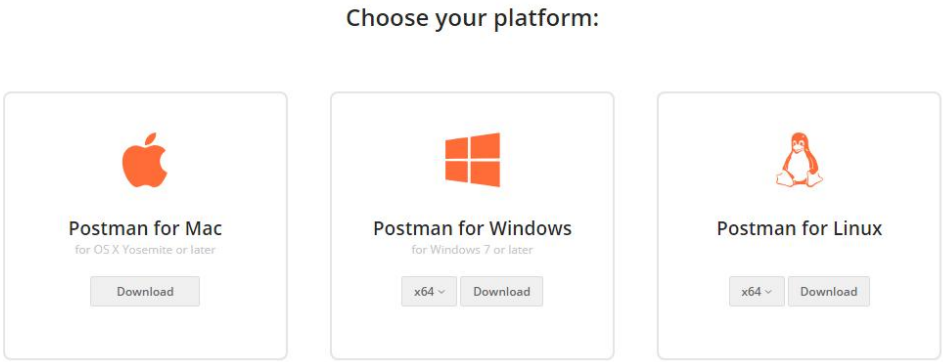
前置条件

- 下载并安装Postman。若未安装，请参考安装Postman进行安装。
- 下载Collection。
- 已在管理控制台完成产品模型和编解码插件的开发。

安装并配置 Postman

步骤1 安装Postman。

1. 访问Postman官网，下载并安装Windows 64位Postman最新版本。



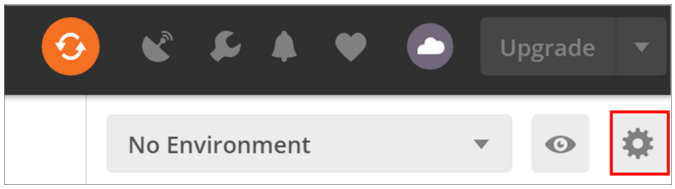
说明

- 安装Postman依赖.NET Framework 4.5组件。
- 如需下载Windows 32位Postman最新版本，访问此处下载。

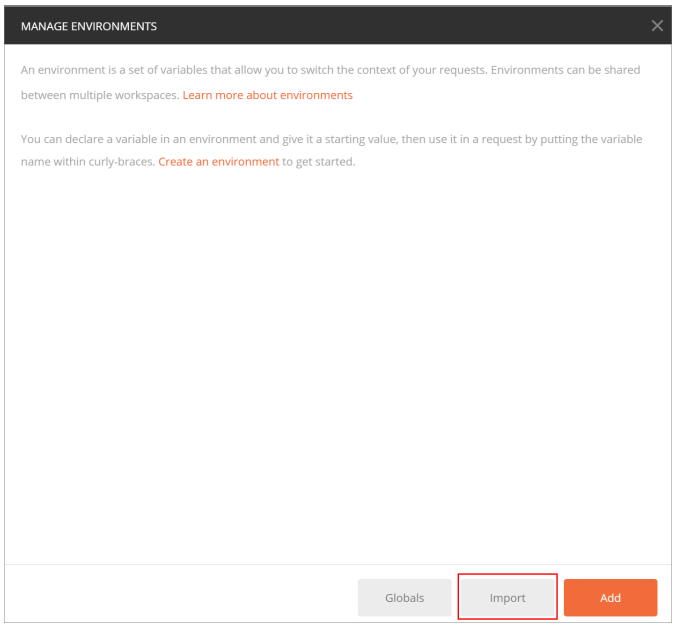
2. 填写邮箱、用户名和密码注册Postman。

步骤2 导入Postman环境变量。

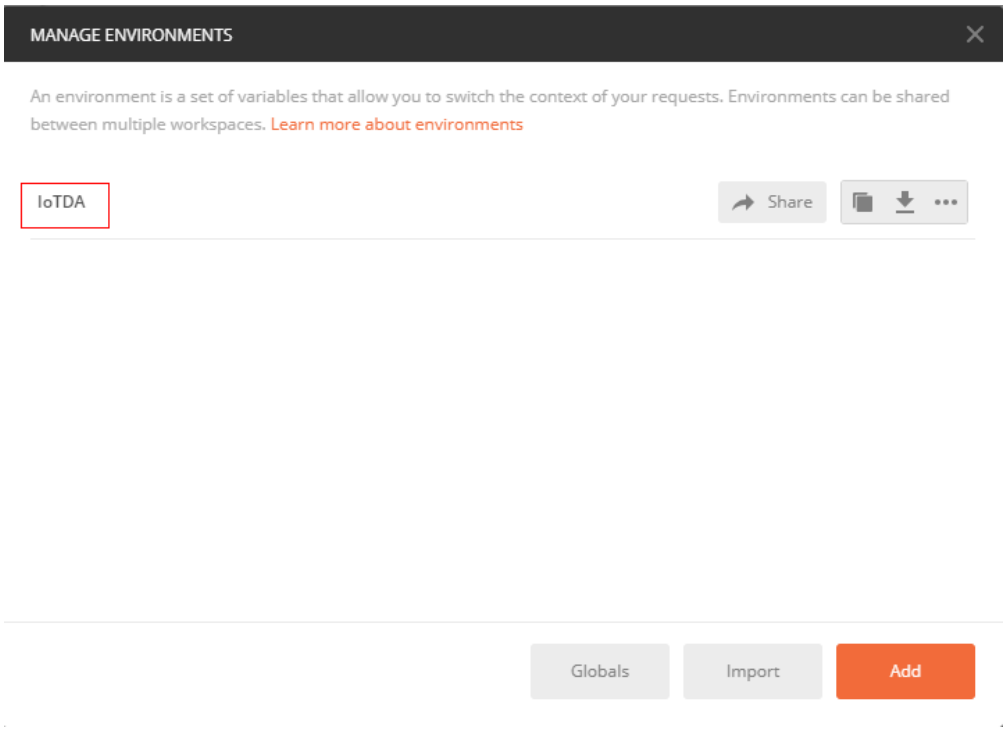
1. 单击右上角的 图标，打开“MANAGE ENVIRONMENTS”窗口。



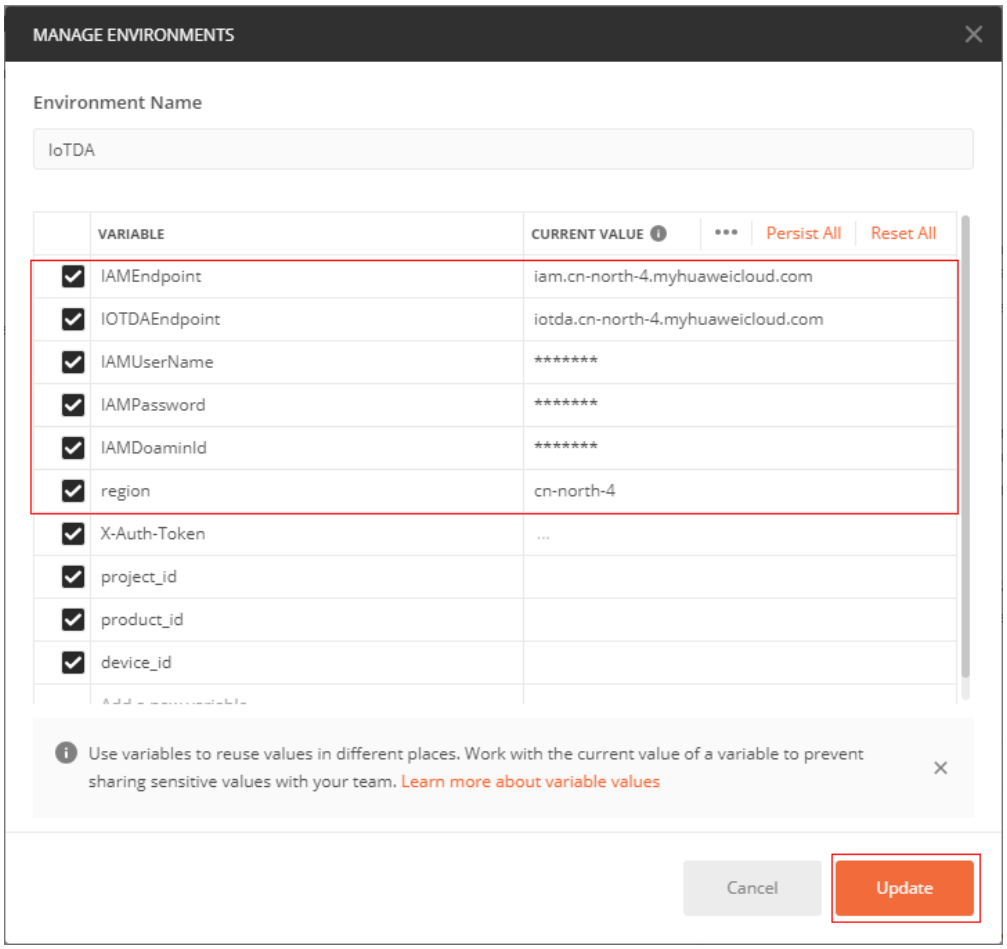
2. 单击“Import”，在弹出的页面中，单击“选择文件”，导入**IoTDA.postman_environment.json**文件（下载Collection解压后获取）。



3. 单击导入的“**IoTDA**”环境。



4. 参考下表修改以下参数。

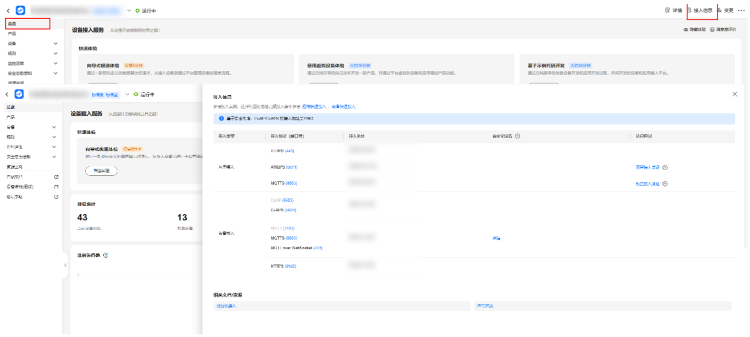


参数名	参数说明
IAMEndpoint	IAM终端节点，参考**IAM地区和终端节点**获取。
IoTDAEndpoint	物联网平台终端节点，参考**步骤2.5**。
IAMUserName	IAM用户名，参考**我的凭证**获取。
IAMPassword	登录华为云的密码。
IAMDoaminId	账号名，参考**我的凭证**获取。
region	开通设备接入服务的区域。

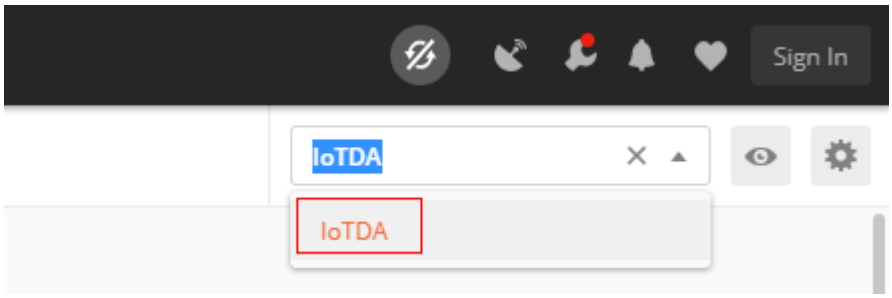
5. IoTDAEndpoint参考。

   进入控制台，选择左侧导航栏“总览”，单击“实例基本信息-接入信息”，根据相应的接入类型和协议选择对应的接入地址。

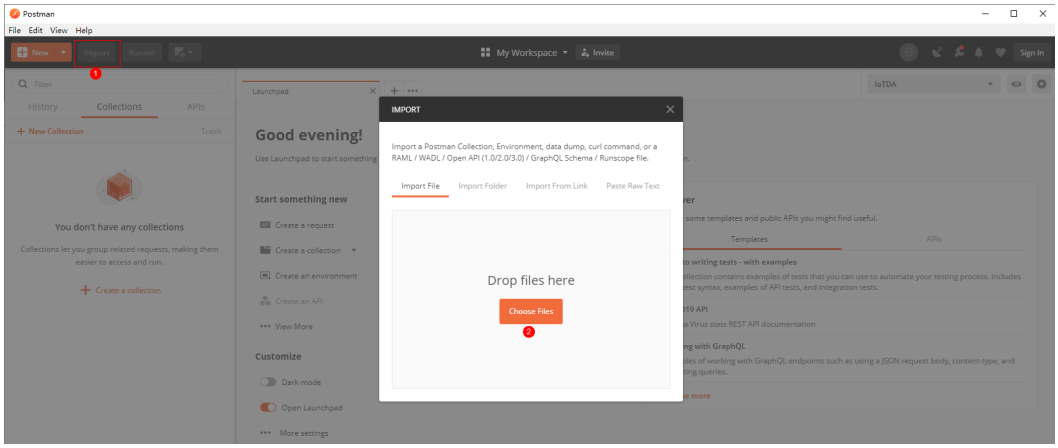
图 4-1 总览-获取接入信息



6. 返回主页，选择环境变量为刚导入的“**IoTDA**”。



步骤3 单击左上角的“Import”，单击“Choose Files”导入“**应用侧API调用（V5版本）.postman_collection.json**”。



导入成功后，显示如下。

----结束

调测“获取 IAM 用户 Token”接口

在访问物联网平台业务接口前，应用服务器需要调用“获取IAM用户Token”接口鉴权，华为云认证通过后向应用服务器返回鉴权令牌**X-Subject-Token**。

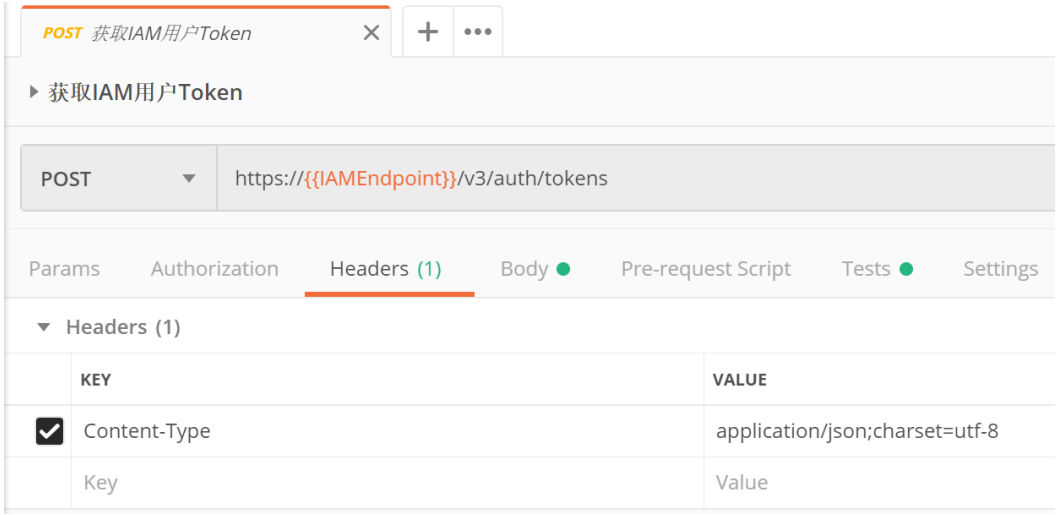
应用服务器需要构造一个HTTP请求，请求示例如下：

```
POST https://iam.cn-north-4.myhuaweicloud.com/v3/auth/tokens
Content-Type: application/json

{
    "auth": {
        "identity": {
            "methods": [
                "password"
            ],
            "password": {
                "user": {
                    "name": "username",
```

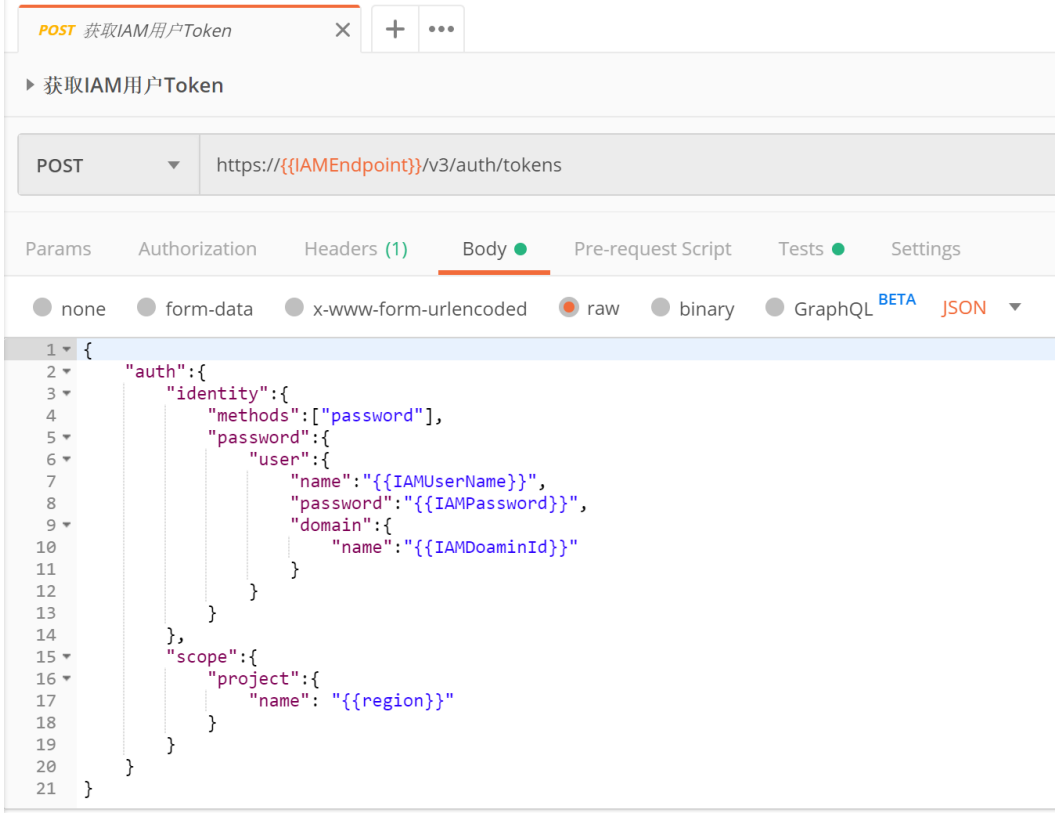
```
                "password": "********",
                "domain": {
                    "name": "domainname"
                }
            }
        }
    },
    "scope": {
        "project": {
            "name": "xxxxxxxx"
        }
    }
  }
}
```

参考API文档，调测**获取IAM用户Token**接口。

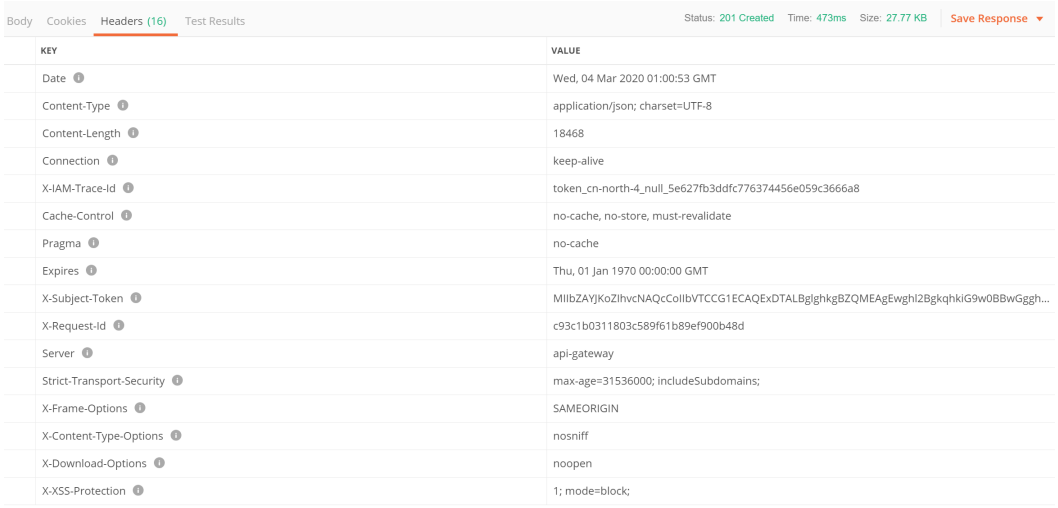
步骤1 配置“获取IAM用户Token”接口的**HTTP**方法、**URL**和**Headers**。



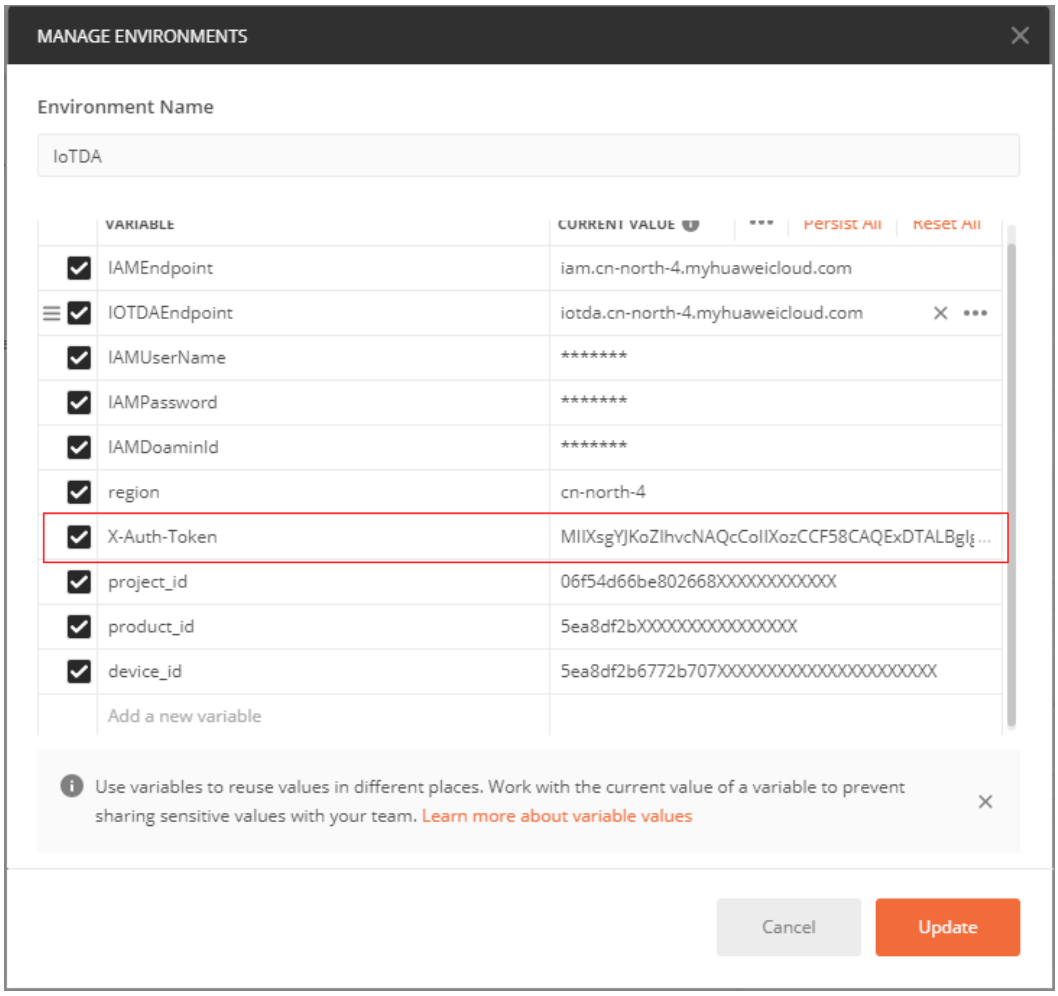
步骤2 配置“获取IAM用户Token”接口的**Body**。



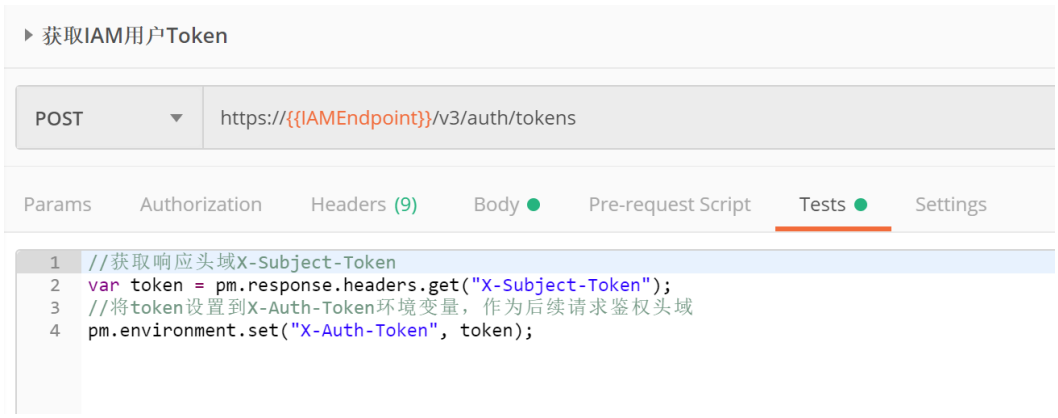
步骤3 单击“Send”，在下方查看返回码和响应消息内容。



步骤4 请将返回头域中的的**X-Subject-Token**更新到“**IoTDA**”环境的“**X-Auth-Token**”参数中，以便于在调用其它接口时使用。若超过令牌有效时间，需要重新调用鉴权接口。



这里我们已经在postman中自动更新了“**X-Auth-Token**”参数，使用时无需手动操作。



----结束

调测“查询 IAM 用户可以访问的项目列表”接口

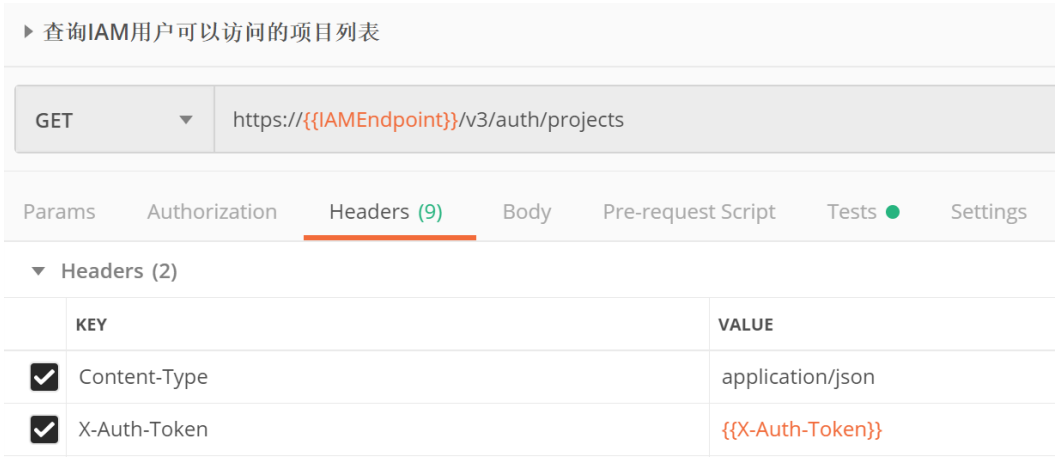
在访问物联网平台业务接口前，应用服务器需要调用“查询IAM用户可以访问的项目列表”接口获取用户的项目ID，用于后续访问物联网平台业务接口。

应用服务器需要构造一个HTTP请求，请求示例如下：

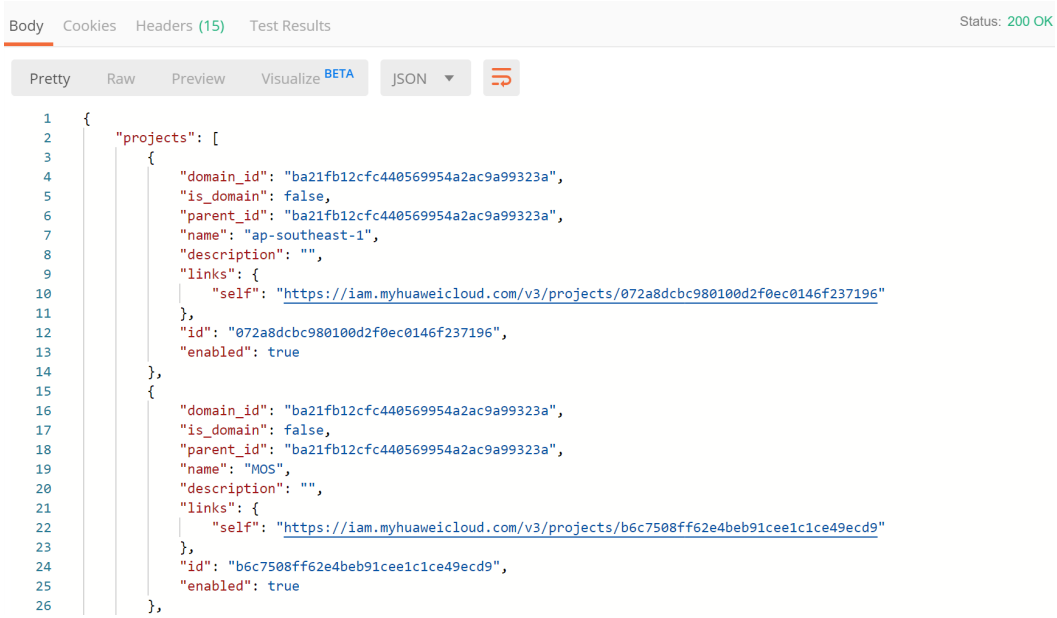
```
GET https://iam.cn-north-4.myhuaweicloud.com/v3/auth/projects
Content-Type: application/json
X-Auth-Token: ********
```

参考API文档，调测**查询IAM用户可以访问的项目列表**接口。

步骤1 配置“查询IAM用户可以访问的项目列表”接口的**HTTP**方法、**URL**和**Headers**。

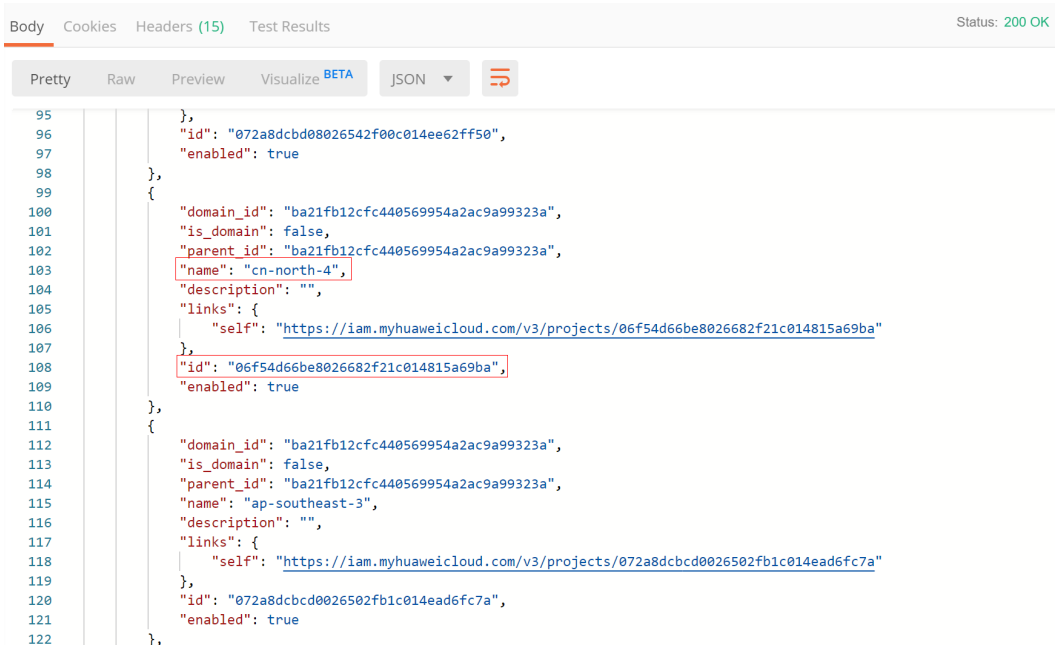


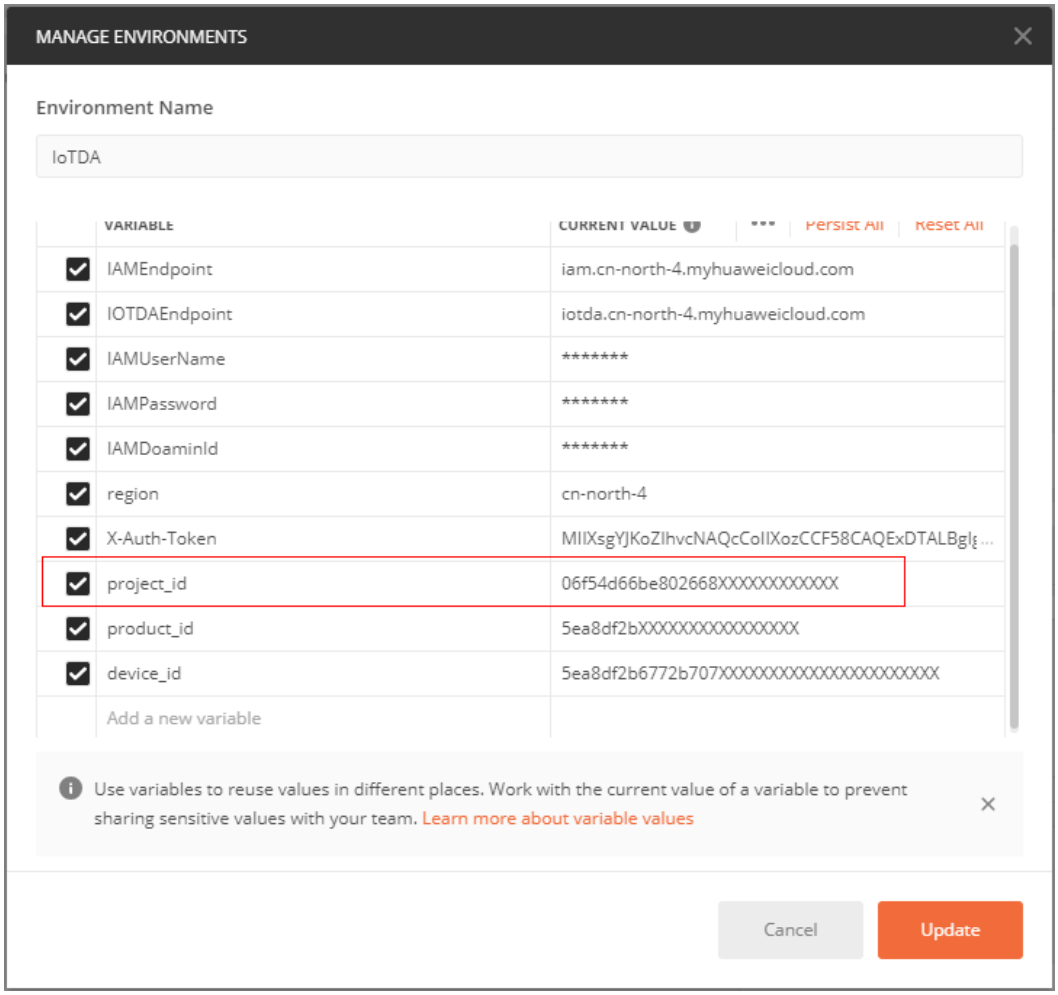
步骤2 单击“Send”，在下方查看返回码和响应消息内容。



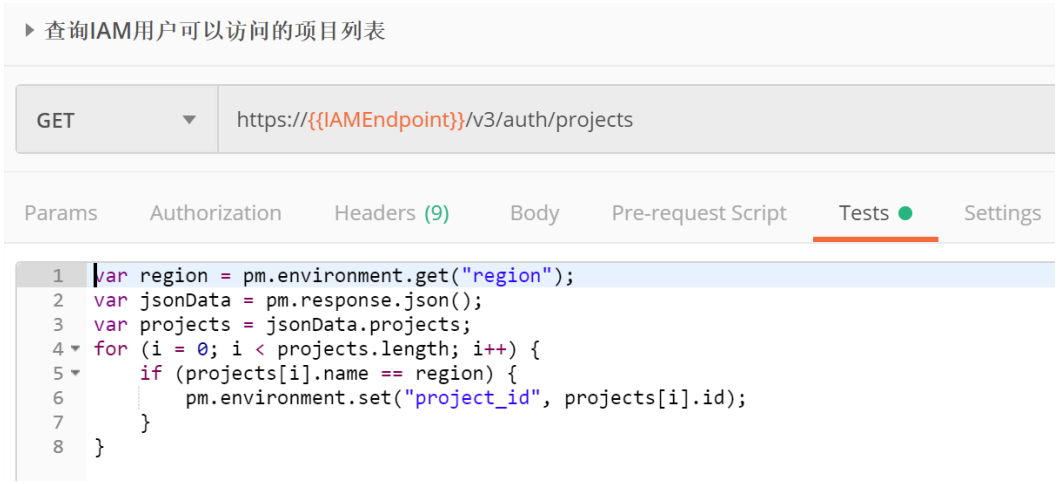

```
{
    "projects": [
        {
            "domain_id": "ba21fb12cfc440569954a2ac9a99323a",
            "is_domain": false,
            "parent_id": "ba21fb12cfc440569954a2ac9a99323a",
            "name": "ap-southeast-1",
            "description": "",
            "links": {
                "self": "https://iam.myhuaweicloud.com/v3/projects/072a8dcbc980100d2f0ec0146f237196"
            },
            "id": "072a8dcbc980100d2f0ec0146f237196",
            "enabled": true
        },
        {
            "domain_id": "ba21fb12cfc440569954a2ac9a99323a",
            "is_domain": false,
            "parent_id": "ba21fb12cfc440569954a2ac9a99323a",
            "name": "MOS",
            "description": "",
            "links": {
                "self": "https://iam.myhuaweicloud.com/v3/projects/b6c7508ff62e4beb91cee1c1ce49ecd9"
            },
            "id": "b6c7508ff62e4beb91cee1c1ce49ecd9",
            "enabled": true
        },
```

步骤3 返回**body**中包含一个**projects**列表，查找其中“**name**”参数值与“**IoTDA**”环境中“**region**”参数值相同的条目，取其“**id**”参数值更新到“**IoTDA**”环境中“**project_id**”参数，以便于在调用其它接口时使用。





这里我们已经在postman中自动更新了“**project_id**”参数，使用时无需手动操作。



----结束

调测“创建产品”接口

在设备接入物联网平台前，应用服务器需要调用此接口创建产品，后续注册设备时需要使用这里创建的产品。

应用服务器需要构造一个请求，请求示例如下：

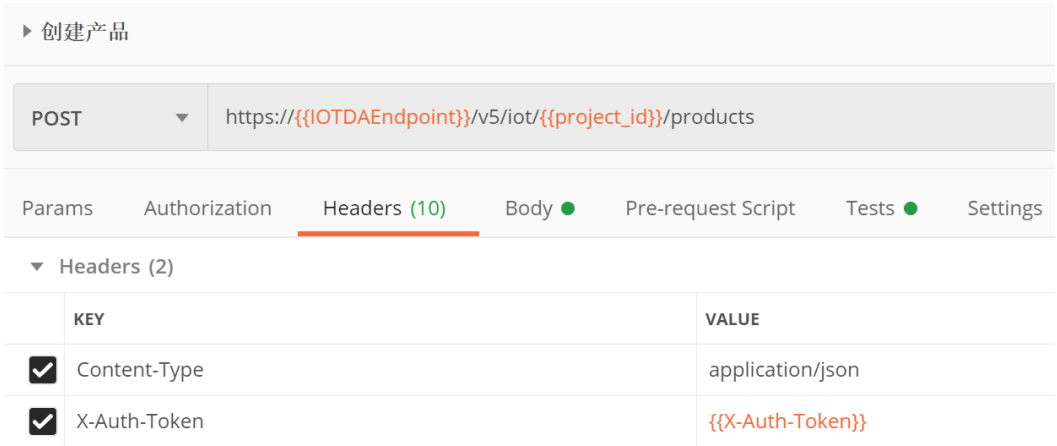
```
POST https://iotda.cn-north-4.myhuaweicloud.com/v5/iot/{project_id}/products
Content-Type: application/json
X-Auth-Token: ********

{
  "name" : "Thermometer",
  "device_type" : "Thermometer",
  "protocol_type" : "MQTT",
  "data_format" : "binary",
  "manufacturer_name" : "ABC",
  "industry" : "smartCity",
  "description" : "this is a thermometer produced by Huawei",
  "service_capabilities" : [ {
    "service_type" : "temperature",
    "service_id" : "temperature",
    "description" : "temperature",
    "properties" : [ {
      "unit" : "centigrade",
      "min" : "1",
      "method" : "R",
      "max" : "100",
      "data_type" : "decimal",
      "description" : "force",
      "step" : 0.1,
      "enum_list" : [ "string" ],
      "required" : true,
      "property_name" : "temperature",
      "max_length" : 100
    } ],
    "commands" : [ {
      "command_name" : "reboot",
      "responses" : [ {
        "response_name" : "ACK",
        "paras" : [ {
          "unit" : "km/h",
          "min" : "1",
          "max" : "100",
          "para_name" : "force",
          "data_type" : "string",
```

```
        "description" : "force",
        "step" : 0.1,
        "enum_list" : [ "string" ],
        "required" : false,
        "max_length" : 100
      } ]
    } ],
    "paras" : [ {
      "unit" : "km/h",
      "min" : "1",
      "max" : "100",
      "para_name" : "force",
      "data_type" : "string",
      "description" : "force",
      "step" : 0.1,
      "enum_list" : [ "string" ],
      "required" : false,
      "max_length" : 100
    } ]
  } ],
  "option" : "Mandatory"
 } ],
 "app_id" : "jeQDJQZltU8iKgFFoW060F5SGZka"
}
```

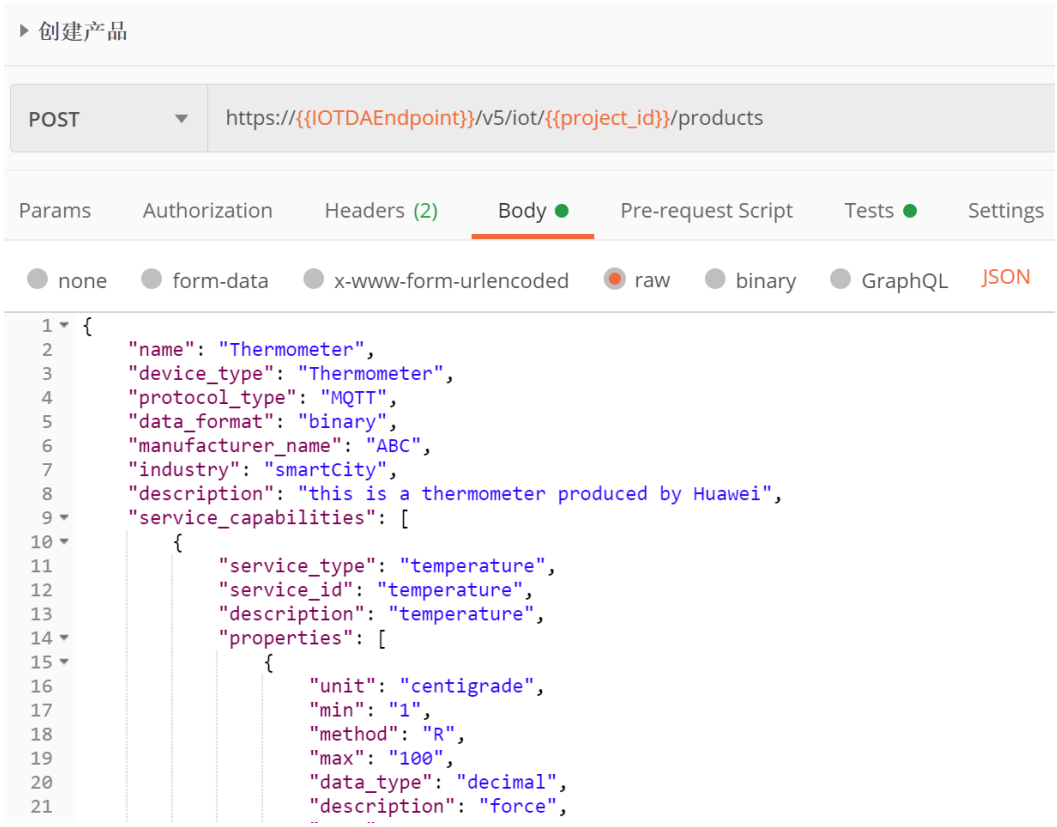

参考API文档，调测物联网平台**创建产品**接口。

注：在以下步骤中，只呈现样例调测用到的参数。

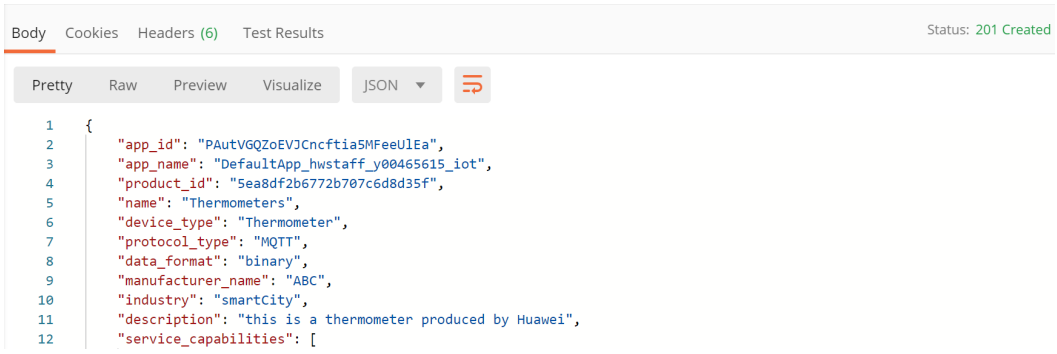
步骤1 配置“创建产品”接口的**HTTP方法**、**URL**和**Headers**。



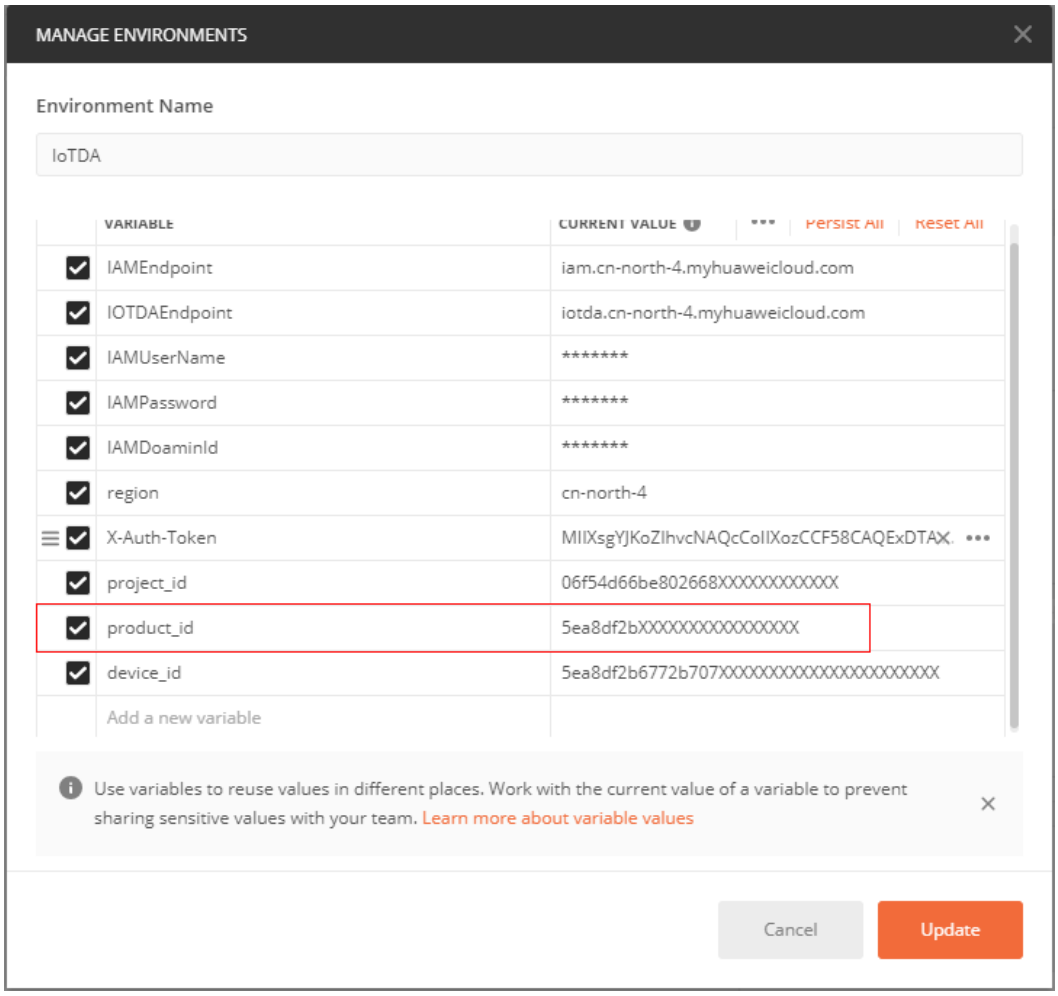
步骤2 配置“创建产品”接口的**BODY**。



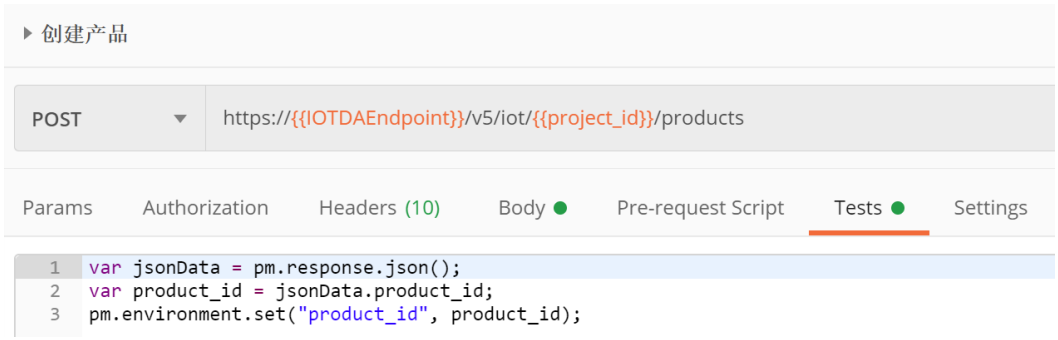
步骤3 单击“Send”，在下方查看返回码和响应消息内容。



步骤4 将返回的“**product_id**”更新到“**IoTDA**”环境中的“**product_id**”参数中，用于后续其它接口使用。



注：在postman中自动更新了“**product_id**”参数，使用时无需手动操作。



----结束

调测“查询产品”接口

应用服务器如果需要查询之前创建的产品详情，可以调用此接口。

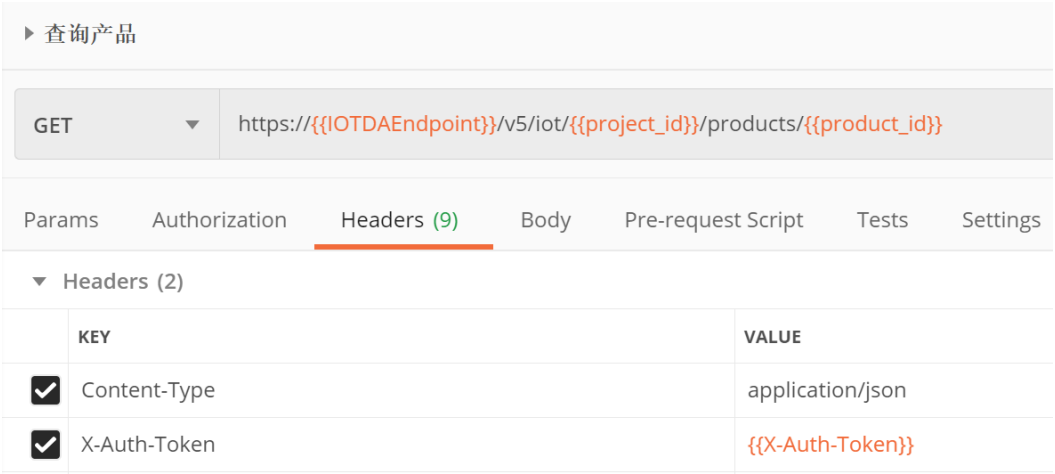
应用服务器需要构造一个请求，请求示例如下：

```
GET https://iotda.cn-north-4.myhuaweicloud.com/v5/iot/{project_id}/products/{product_id}
Content-Type: application/json
X-Auth-Token: ********
```

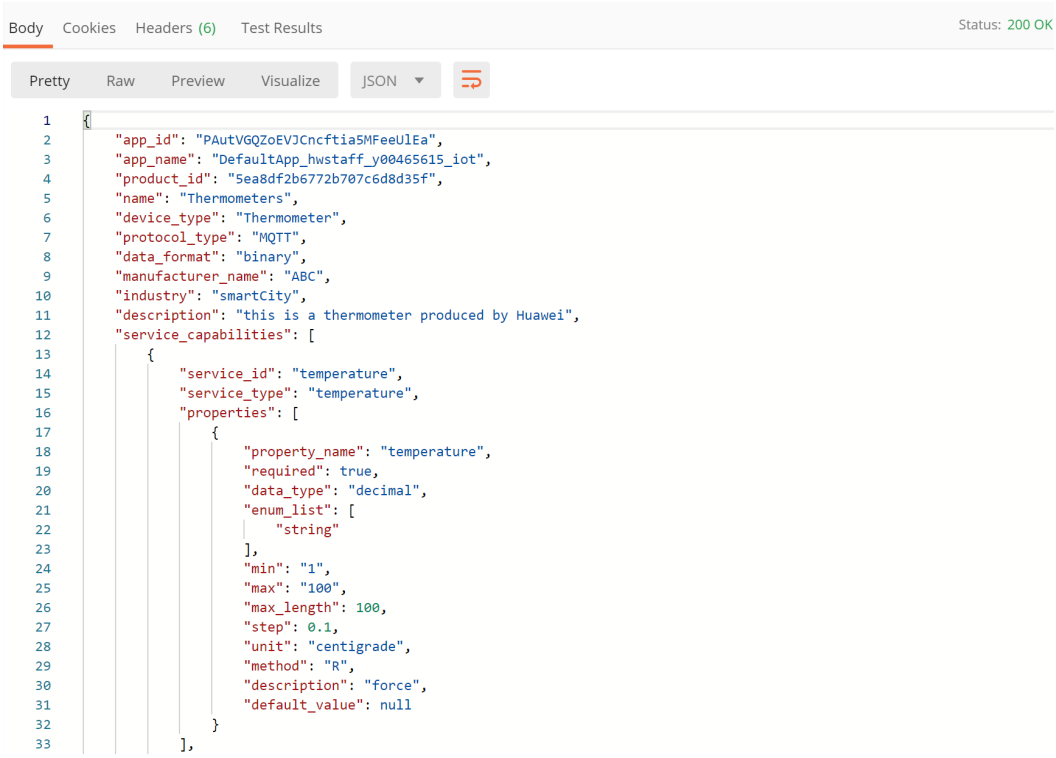
接下来参考API文档，调测物联网平台**查询产品**接口。

注：在以下步骤中，只呈现样例调测用到的参数。

步骤1 配置“查询产品”接口的**HTTP方法**、**URL**和**Headers**。



步骤2 单击“Send”，在下方查看返回码和响应消息内容。



----结束

调测“创建设备”接口

在设备接入物联网平台前，应用服务器需要调用此接口在物联网平台创建设备。在设备接入物联网平台时携带设备唯一标识，完成设备的接入认证。

应用服务器需要构造一个HTTP请求，请求示例如下：

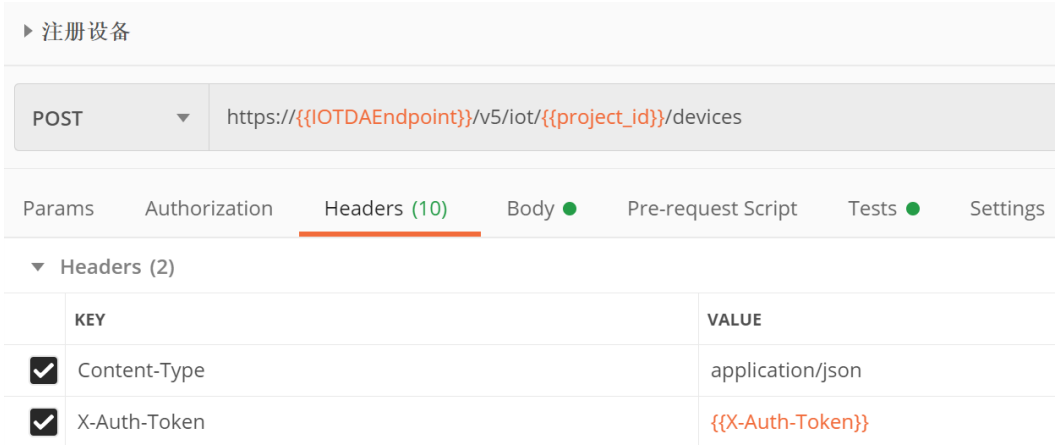
```
POST https://iotda.cn-north-4.myhuaweicloud.com/v5/iot/{project_id}/devices
Content-Type: application/json
```

```
X-Auth-Token: ********

{
  "node_id" : "ABC123456789",
  "device_name" : "dianadevice",
  "product_id" : "b640f4c203b7910fc3cbd446ed437cbd",
  "auth_info" : {
    "auth_type" : "SECRET",
    "secure_access" : true,
    "fingerprint" : "********",
    "secret" : "********",
    "timeout" : 300
  },
  "description" : "watermeter device"
}
```

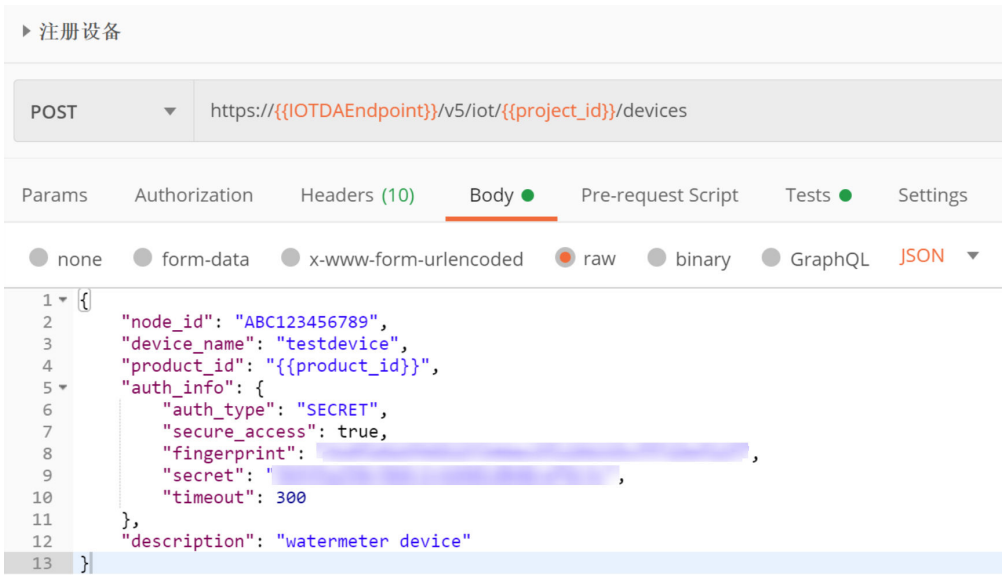
参考API文档，调测物联网平台**创建设备**接口。

注：在以下步骤中，只呈现样例调测用到的参数。

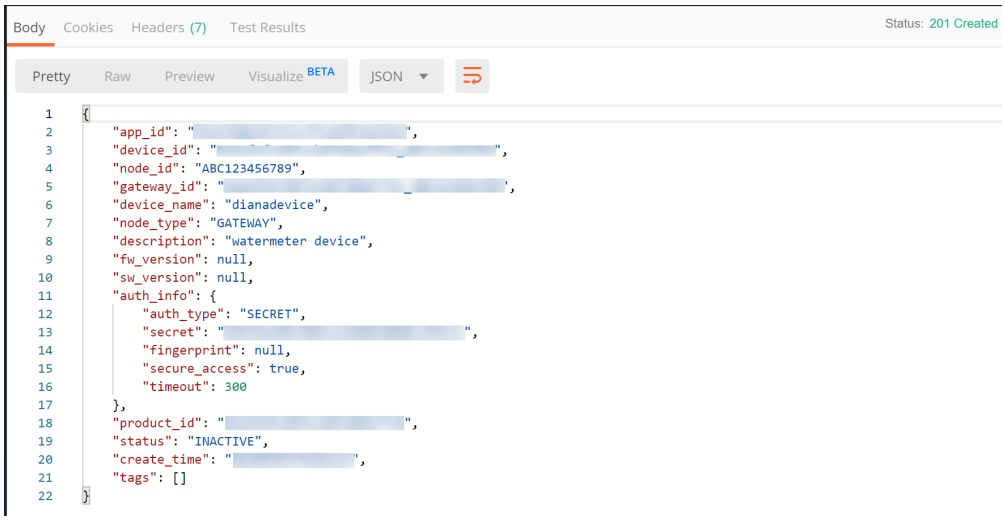
步骤1 配置“创建设备”接口的**HTTP方法**、**URL**和**Headers**。



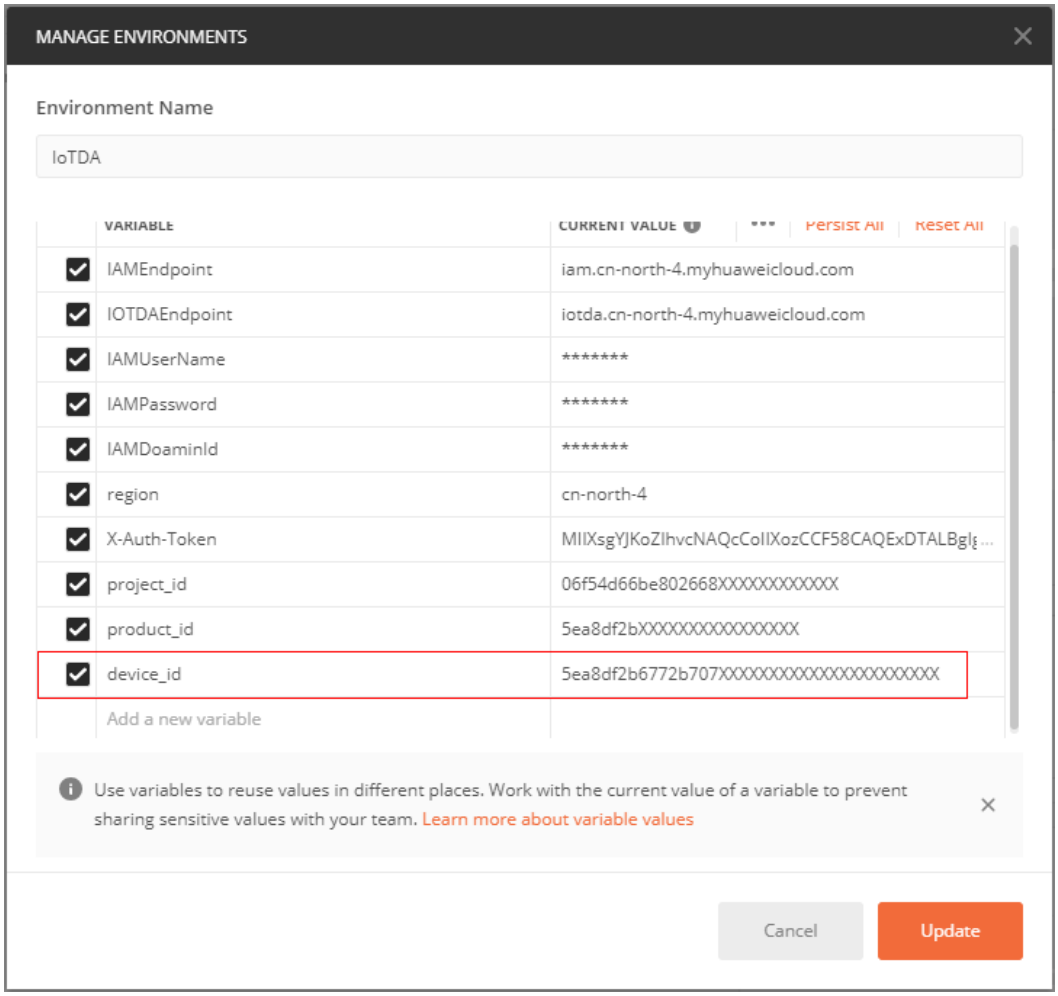
步骤2 配置“创建设备”接口的**Body**。



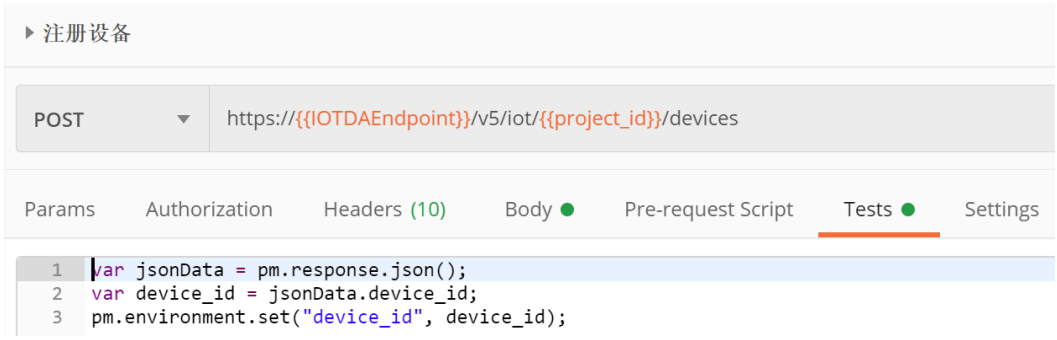
步骤3 单击“Send”，在下方查看返回码和响应消息内容。



步骤4 请将返回的“**device_id**”更新到“**IoTDA**”环境中的“**device_id**”参数中，用于后续其它接口使用。



注意：在postman中自动更新了“**device_id**”参数，使用时无需手动操作。



----结束

调测“查询设备”接口

应用服务器需要查询在物联网平台创建的设备详情时，可以调用此接口。

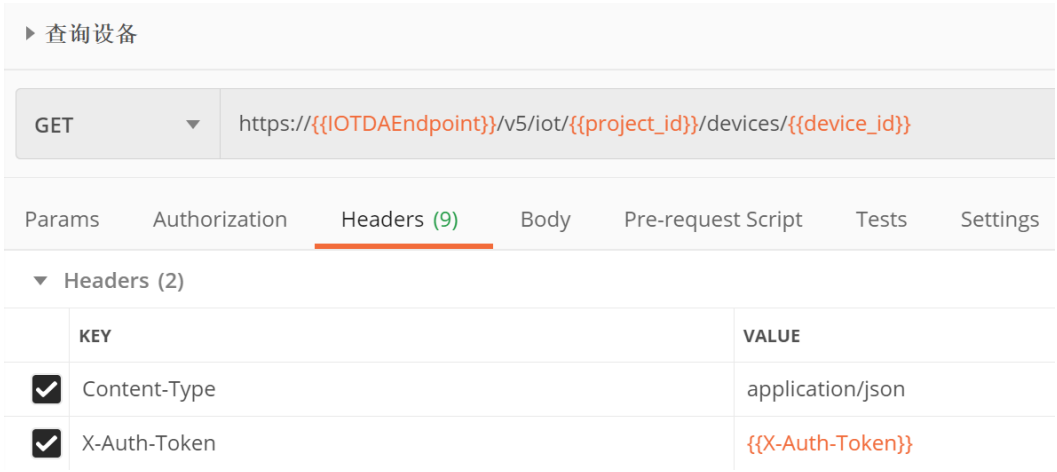
应用服务器需要构造一个HTTP请求，请求示例如下：

```
GET https://iotda.cn-north-4.myhuaweicloud.com/v5/iot/{project_id}/devices/{device_id}
Content-Type: application/json
X-Auth-Token: ********
```

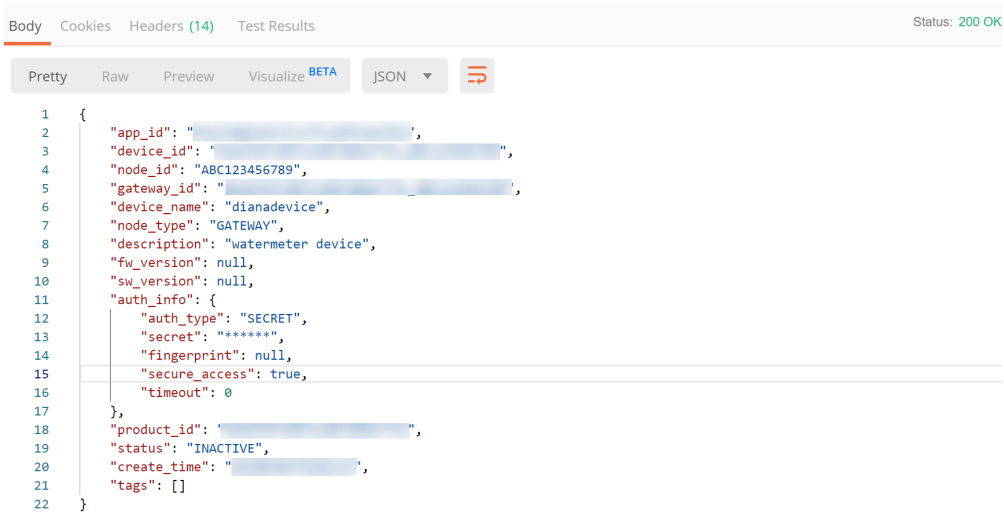
参考API文档，调测物联网平台**查询设备**接口。

注：在以下步骤中，只呈现样例调测用到的参数。

步骤1 配置“查询设备”接口的**HTTP方法**、**URL**和**Headers**。



步骤2 单击“Send”，在下方查看返回码和响应消息内容。



----结束