

设备接入

开发指南

文档版本 1.0
发布日期 2022-08-30



版权所有 © 华为云计算技术有限公司 2024。保留一切权利。

非经本公司书面许可，任何单位和个人不得擅自摘抄、复制本文档内容的部分或全部，并不得以任何形式传播。

商标声明



HUAWEI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	1
2 资源获取	3
3 产品开发	8
3.1 产品开发指引	8
3.2 创建产品	9
3.3 开发产品模型	11
3.3.1 什么是产品模型	11
3.3.2 在线开发产品模型	12
3.3.3 离线开发产品模型	16
3.3.4 导出和导入产品模型	27
3.4 开发编解码插件	29
3.4.1 什么是编解码插件	29
3.4.2 图形化开发插件	31
3.4.3 使用 JavaScript 开发插件	72
3.5 在线调试	86
4 设备侧开发	90
4.1 设备接入指引	90
4.2 使用 IoT Device SDK 接入	93
4.2.1 IoT Device SDK 介绍	93
4.2.2 IoT Device SDK 使用指南（Java）	97
4.2.3 IoT Device SDK 使用指南（C）	115
4.2.4 IoT Device SDK 使用指南（C#）	115
4.2.5 IoT Device SDK 使用指南（Android）	115
4.2.6 IoT Device SDK 使用指南（Go 社区版）	115
4.2.7 IoT Device SDK Tiny 使用指南（C）	115
4.2.8 IoT Device SDK 使用指南（OpenHarmony）	116
4.2.9 IoT Device SDK 使用指南（Python）	116
4.3 使用 MQTT Demo 接入	116
4.3.1 MQTT 使用指导	116
4.3.2 Java Demo 使用说明	122
4.3.3 Python Demo 使用说明	127
4.3.4 Android Demo 使用说明	135

4.3.5 C Demo 使用说明.....	146
4.3.6 C# Demo 使用说明.....	152
4.3.7 Node.js Demo 使用说明.....	161
4.4 OTA 升级设备侧适配.....	166
4.4.1 设备侧适配开发指导.....	166
4.4.2 PCP 协议介绍.....	184
5 应用侧开发.....	191
5.1 API 使用指导.....	191
5.2 使用 Postman 调测.....	195

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：

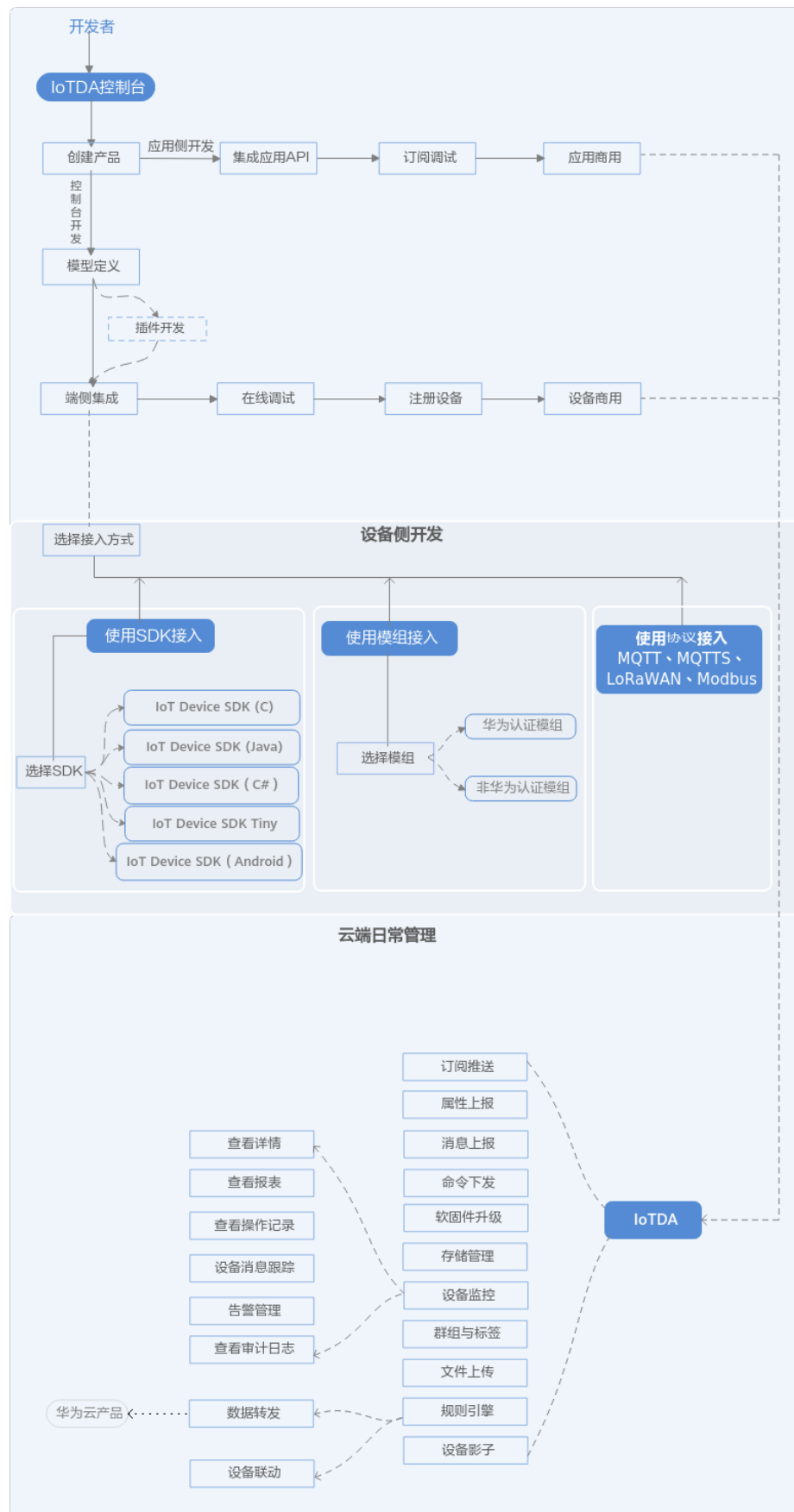
开发操作	开发说明
产品开发	主要呈现物联网平台的界面查询与操作，包括产品管理、产品模型开发、插件开发、在线调试等。
应用开发	主要为业务应用与物联网平台的集成对接开发，包括API接口调用、业务数据获取和HTTPS证书管理。
设备开发	主要为设备与物联网平台的集成对接开发，包括设备接入物联网平台、业务数据上报和对平台下发控制命令的处理。

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



2 资源获取

平台对接信息

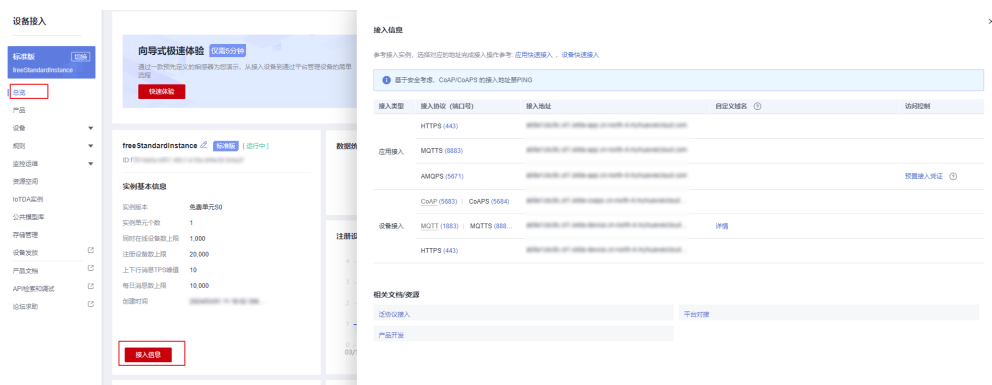
1. 进入IoTDA的**管理控制台**界面，选择左侧导航栏“IoTDA实例”，切换至需要的版本实例。

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
IoT Device SDK(Java)	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 IoT Device SDK使用指南(Java) 。	IoT Device SDK(Java)
IoT Device SDK(C)	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 IoT Device SDK(C)使用指南 。	IoT Device SDK(C)
IoT Device SDK(C#)	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 IoT Device SDK(C#)使用指南 。	IoT Device SDK(C#)
IoT Device SDK(Android)	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 IoT Device SDK(Android)使用指南 。	IoT Device SDK(Android)
IoT Device SDK(Go)	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 IoT Device SDK(Go)使用指南 。	IoT Device SDK(Go)
IoT Device SDK(Python)	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 IoT Device SDK(Python)使用指南 。	IoT Device SDK(Python)

资源包名	描述	下载路径
IoT Device SDK Tiny (C)	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 IoT Device Tiny SDK(C)使用指南	IoT Device SDK Tiny (C)
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
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
应用侧开发 Java SDK	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 Java SDK使用指南 。	Java SDK
应用侧开发 C# SDK	C# SDK提供C#方法调用 应用侧API 与平台通信。使用指南可以参考 C# SDK使用指南 。	C# SDK
应用侧开发 Python SDK	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 Python SDK使用指南 。	Python SDK
应用侧开发 Go SDK	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 Go SDK使用指南 。	Go SDK
应用侧开发 Node.js SDK	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 Node.js SDK使用指南 。	Node.js SDK
应用侧开发 PHP SDK	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 PHP SDK使用指南 。	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、非洲-约翰内斯堡	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

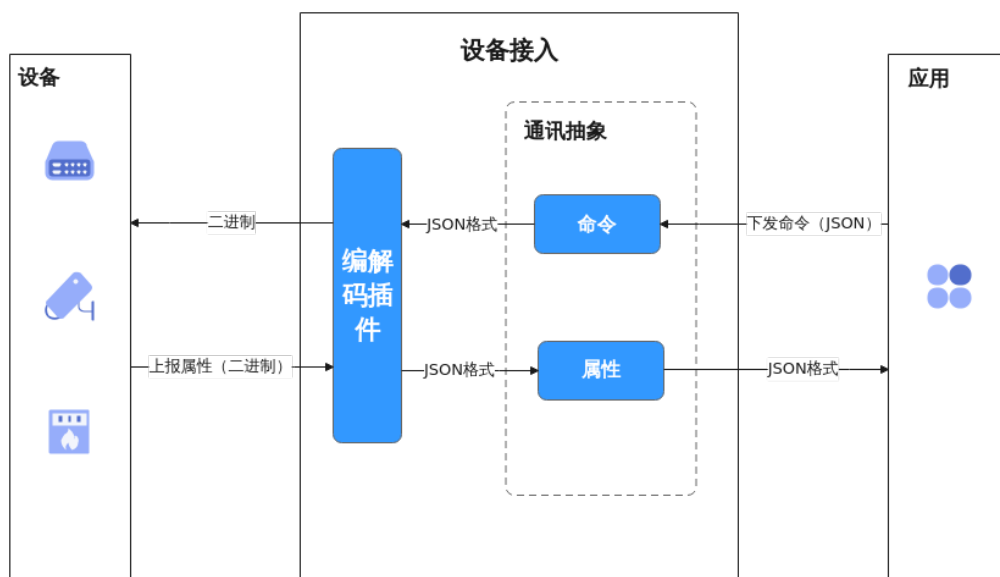
3 产品开发

3.1 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或JSON解码为JSON格式供应用服务器“阅读”，将应用服务器下发的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：

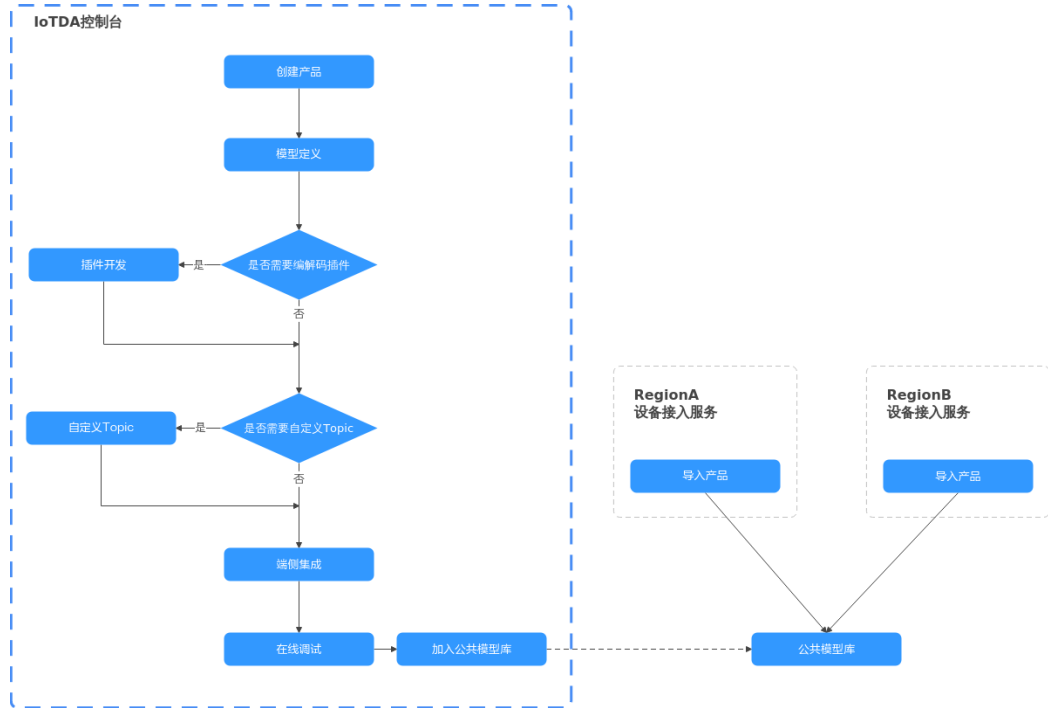
图 3-1 编解码插件流程



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 3-2 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的产品信息、产品模型（Profile）、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

目前仅基础版支持MQTT协议的在线调试。

3.2 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。

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_?'#(),.&%@!-等字符的组合。
协议类型	<ul style="list-style-type: none"> • 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协议的设备接入IoT边缘节点的方式为非直连。直连设备和非直连设备差异说明，请参考这里。 • HTTP(TLS加密)、ONVIF、OPC-UA、OPC-DA、Other：通过边缘接入。
数据格式	<ul style="list-style-type: none"> • 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。
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。

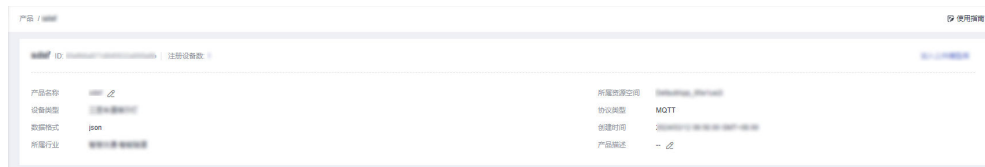
产品创建成功后，您可以单击“删除”删除不再使用的产品。删除产品后，该产品下的产品模型、编解码插件等资源将被清空，请谨慎操作。

----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 3-3 产品-产品详情

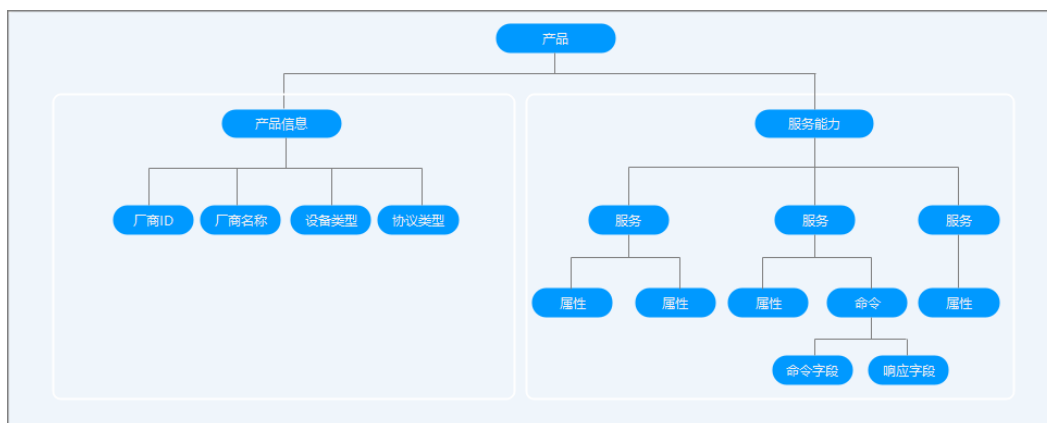


2. 您可以在产品详情页，[开发产品模型](#)、[开发编解码插件](#)、[在线调试](#)、[自定义 Topic](#)。

3.3 开发产品模型

3.3.1 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，如颜色、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。

注：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。

3.3.2 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，您可以根据需要对产品模型中的字段进行修改和删除；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。
- 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** 在产品详情模型定义页面，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“确认”。
- “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** 单击步骤4新增的服务ID，在展开的页面单击单击“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	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> 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 3-4 新增属性

✕

新增属性

* 属性名称

属性描述
0/128

* 数据类型

* 访问权限 可读 可写

* 取值范围 -

步长

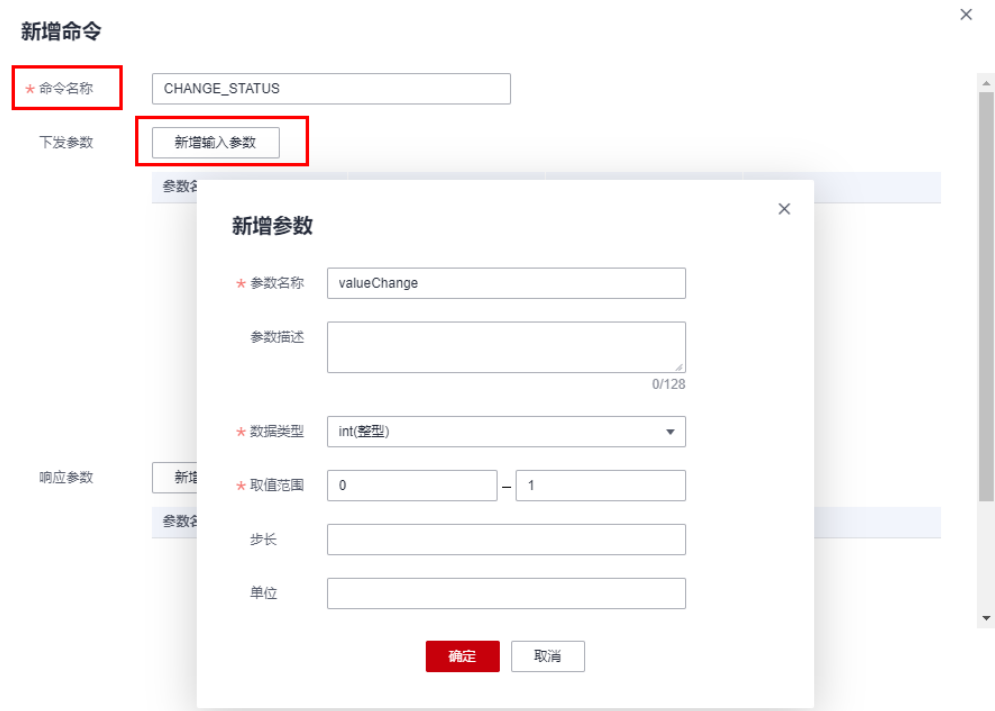
单位

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 3-5 新增命令



- 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 3-6 新增响应参数

✕

新增参数

* 参数名称	<input style="width: 90%;" type="text" value="valueAResult"/>
参数描述	<div style="border: 1px solid #ccc; height: 30px; width: 100%;"></div> <div style="text-align: right; font-size: 10px; color: #999;">0/128</div>
* 数据类型	<div style="border: 1px solid #ccc; padding: 2px;">int(整型) ▼</div>
* 取值范围	<div style="display: flex; align-items: center; gap: 5px;"><input style="width: 60px;" type="text" value="0"/> - <input style="width: 60px;" type="text" value="1"/></div>
步长	<input style="width: 100%;" type="text" value="1"/>
单位	<input style="width: 100%;" type="text"/>

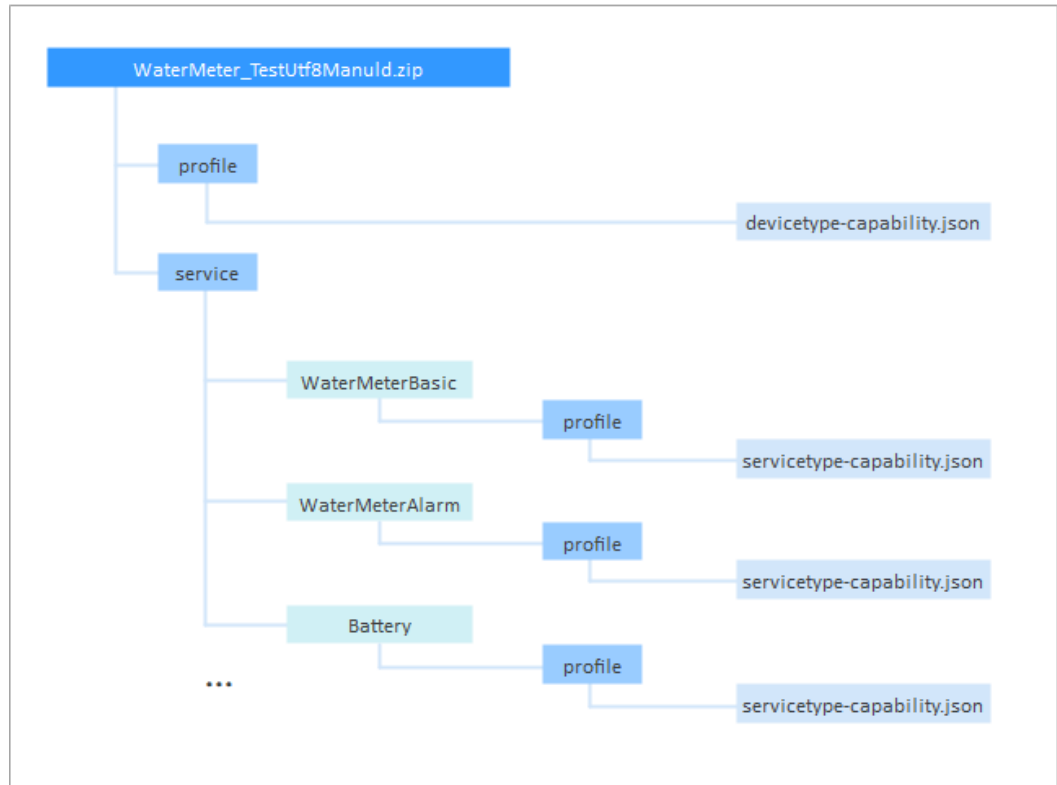
确定取消

----结束

3.3.3 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，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。



所以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。因此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，首先需要编写这款产品的产品模型。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	产品模型中key	属性值
设备类型	deviceType	WaterMeter
厂商ID	manufacturerId	TestUtf8Manuld
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": "TestUtf8Manuld",
      "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	指示服务的类型，与devicetype-capability.json中serviceType字段保持一致。
-	description	-	-	-	必选	指示服务的描述信息。 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	指示设备可以执行的命令，如果本服务无命令则置null。
-	-	commandName	-	-	必选	指示命令的名字，命令名与参数共同构成一个完整的命令。
-	-	params	-	-	必选	命令包含的参数。
-	-	-	paramName	-	必选	命令中参数的名字。
-	-	-	dataType	-	必选	指示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： <ul style="list-style-type: none"> 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	指示本命令是否必选，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	指示最小值。 仅当dataType为int、decimal时生效。
-	-	-	max	-	必选	指示最大值。 仅当dataType为int、decimal时生效。
-	-	-	step	-	必选	指示步长。 暂不使用，填0即可。
-	-	-	maxLength	-	必选	指示字符串长度。 仅当dataType为string、string list、DateTime时生效。

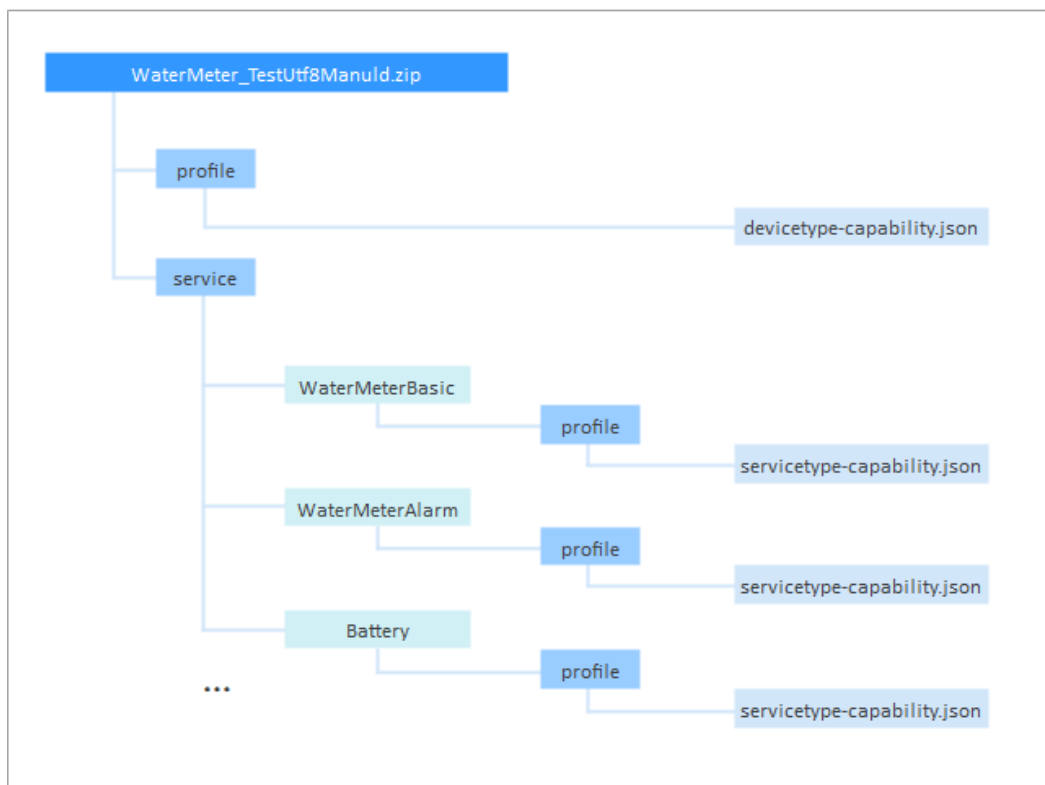
字段	子字段				必选/可选	描述
-	-	-	unit	-	必选	指示单位，英文。 取值根据参数确定，如： 温度单位：“C”或“K” 百分比单位：“%” 压强单位：“Pa”或“kPa”
-	-	-	enum List	-	必选	指示枚举值。 如开关状态status可有如下取值： "enumList" : ["OPEN","CLOSE"] 目前本字段是非功能性字段，仅起到描述作用，建议准确定义。
-	-	responses	-	-	必选	命令执行的响应。
-	-	-	responseName	-	必选	命名可以在该responses对应命令的commandName后面添加“_RSP”。
-	-	-	paras	-	必选	命令响应的参数。
-	-	-	-	parameter	必选	命令中参数的名字。
-	-	-	-	dataType	必选	指示数据类型。 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： <ul style="list-style-type: none"> 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	指示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。
-	-	-	-	min	必选	指示最小值。 仅当dataType为int、decimal时生效，逻辑大于等于。

字段	子字段				必选/可选	描述
-	-	-	-	max	必选	指示最大值。 仅当dataType为int、decimal时生效，逻辑小于等于。
-	-	-	-	step	必选	指示步长。 暂不使用，填0即可。
-	-	-	-	maxLength	必选	指示字符串长度。 仅当dataType为string、string list、DateTime时生效。
-	-	-	-	unit	必选	指示单位，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	指示枚举值。 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	指示属性名称。
-	-	dataType	-	-	必选	指示数据类型。 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： <ul style="list-style-type: none"> 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

字段	子字段				必选/可选	描述
-	-	required	-	-	必选	指示本条属性是否必选，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	指示最小值。 仅当dataType为int、decimal时生效，逻辑大于等于。
-	-	max	-	-	必选	指示最大值。 仅当dataType为int、decimal时生效，逻辑小于等于。
-	-	step	-	-	必选	指示步长。 暂不使用，填0即可。
-	-	method	-	-	必选	指示访问模式。 R：可读；W：可写；E：可订阅。 取值范围：R、RW、RE、RWE、null。
-	-	unit	-	-	必选	指示单位，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	指示字符串长度。 仅当dataType为string、string list、DateTime时生效。
-	-	enumList	-	-	必选	指示枚举值。 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": "TestUtf8Manuld",
      "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。

3.3.4 导出和导入产品模型

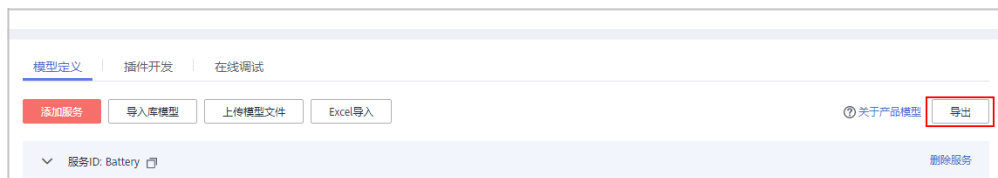
产品模型作为一种资源，可以从物联网平台导出，也可以导入到物联网平台。

- 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。
- 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。



----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“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，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**
 - 访问[设备接入服务](#)，单击“管理控制台”进入设备接入控制台。
 - 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。
 - 在模型定义页面，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“确认”。

图 3-7 上传模型文件



● Excel导入

- 访问**设备接入服务**，单击“管理控制台”进入设备接入控制台。
- 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。
- 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“确认”。



3.4 开发编解码插件

3.4.1 什么是编解码插件

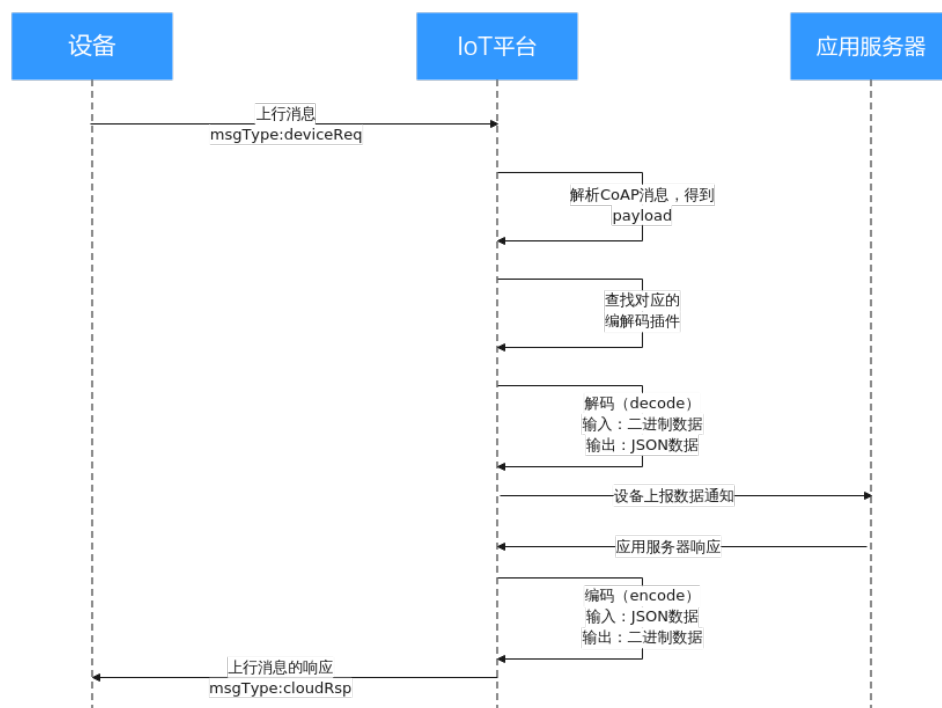
编解码插件是供物联网平台调用，可以完成二进制格式与JSON格式相互转换、也可以完成JSON格式之间的转换。

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。



数据上报流程

图 3-8 数据上报编解码插件

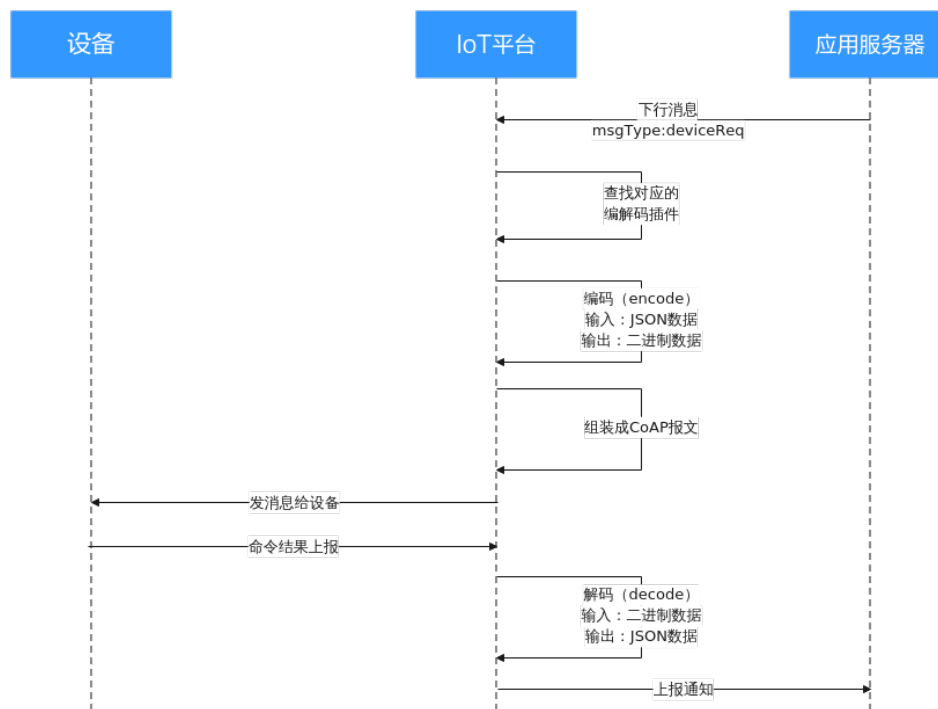


在数据上报流程中，有两处需要用到编解码插件：

- 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 3-9 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。

3.4.2 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 3-10 模型定义-smokerdetector



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 3-11 插件开发-新增消息 smokerinfo

新增消息 ×

基本信息

*消息名

*消息类型 数据上报 命令下发

添加响应字段

描述

0/1,024

字段 添加字段

偏移值	名字	描述	数据类型	长度	是否地址域	操作
<p>暂无表格数据</p>						

响应数据

确定 取消

- 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 3-12 插件开发-添加字段 messaged

×

添加字段

i 只有标记为地址域时，名字固定为messaged；其他字段名字不能设置为messaged。

标记为地址域 ?

* 字段名称	<input type="text" value="messaged"/>
描述	<input type="text" value="输入字段描述"/> 0/1,024
数据类型 (大端模式)	<input type="text" value="int8u"/>
偏移值	<input type="text" value="0-1"/> ?
* 长度	<input type="text" value="1"/> ?
默认值	<input type="text" value="0x0"/> ?

确定 取消

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，前字段“messaged”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 3-13 插件开发-添加字段 level

添加字段

×

标记为地址域 ?

* 字段名称

描述 0/1,024

数据类型 (大端模式)

偏移值 ?

* 长度 ?

默认值 ?

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 3-14 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 3-15 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 3-16 插件开发-添加命令字段 messageId(0x3)

×

添加字段

i 只有标记为地址域时，名字固定为messageId；其他字段名字不能设置为messageId。

标记为地址域 ?

标记为响应标识字段 ?

* 字段名称	<input type="text" value="messageId"/>
描述	<input style="width: 90%;" type="text" value="输入字段描述"/> 0/1,024
数据类型 (大端模式)	<input style="width: 90%;" type="text" value="int8u"/>
偏移值	<input style="width: 90%;" type="text" value="0-1"/> ?
* 长度	<input style="width: 90%;" type="text" value="1"/> ?
默认值	<input style="width: 90%;" type="text" value="0x3"/> ?

确定
取消

- 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 3-17 插件开发-添加命令字段 mid

×

添加字段

i 只有标记为响应标识字段时，名字固定为mid；其他字段名字不能设置为mid。

标记为地址域 ?

标记为响应标识字段 ?

***** 字段名称

描述 0/1,024

数据类型 (大端模式)

偏移值 ?

***** 长度 ?

默认值 ?

确定 取消

- 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 3-18 插件开发-添加命令字段 value

添加字段 ×

标记为地址域 ?

标记为响应标识字段 ?

* 字段名称

描述 0/1,024 ↕

数据类型 (大端模式) ▾

偏移值 ?

* 长度 ?

默认值 ?

确定 取消

- d. 单击“添加响应字段”，添加“messageld”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messageld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。messageld、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-添加响应字段 messageId(0x4)

添加字段 ×

i 只有标记为地址域时，名字固定为messageId；其他字段名字不能设置为messageId。

标记为地址域 ?

标记为响应标识字段 ?

标记为命令执行状态字段 ?

***** 字段名称

描述 0/1,024

数据类型 (大端模式)

偏移值 ?

***** 长度 ?

默认值 ?

确定 取消

- 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 3-20 插件开发-添加响应字段 mid

×

添加字段

i 只有标记为响应标识字段时，名字固定为mid；其他字段名字不能设置为mid。

标记为地址域 ?

标记为响应标识字段 ?

标记为命令执行状态字段 ?

*** 字段名称**

描述
0/1,024

数据类型 (大端模式)

偏移值 ?

*** 长度** ?

默认值 ?

确定
取消

- 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 3-21 插件开发-添加响应字段 errcode

×

添加字段

i 只有标记为命令执行状态字段时，名字固定为errcode；其他字段名字不能设置为errcode。

标记为地址域 ?

标记为响应标识字段 ?

标记为命令执行状态字段 ?

* 字段名称

描述
0/1,024

数据类型 (大端模式)

偏移值 ?

* 长度 ?

默认值

确定 取消

- 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 3-22 插件开发-添加响应字段 result

×

添加字段

标记为地址域 ?

标记为响应标识字段 ?

标记为命令执行状态字段 ?

* 字段名称

描述 0/1,024

数据类型 (大端模式)

偏移值 ?

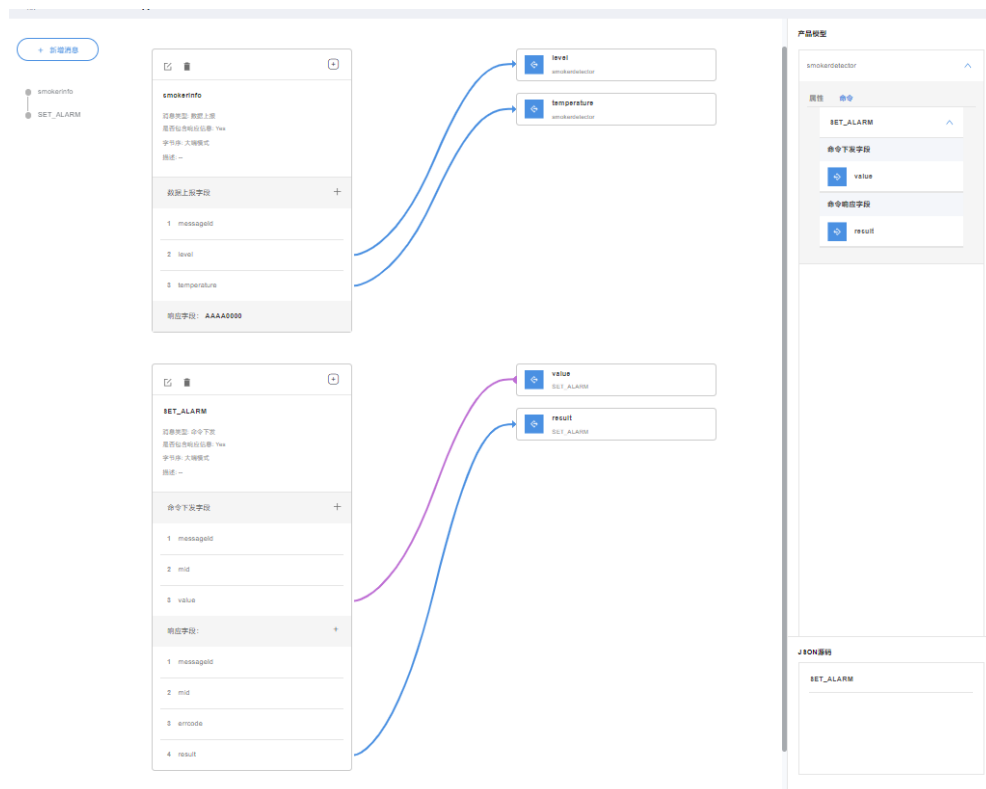
* 长度 ?

默认值 ?

确定 取消

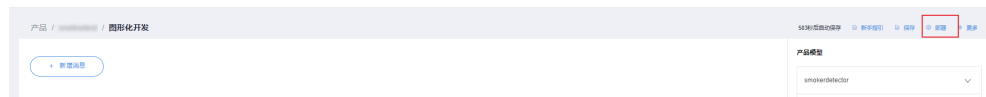
步骤4 拖动右侧“设备模型”区域的属性字段和命令字段，数据上报消息和命令下发消息的相应字段建立映射关系。

图 3-23 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 3-24 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模拟设备进行调测。具体调测步骤请参考[在线调试](#)。本文以模拟设备为例，调测编解码插件。

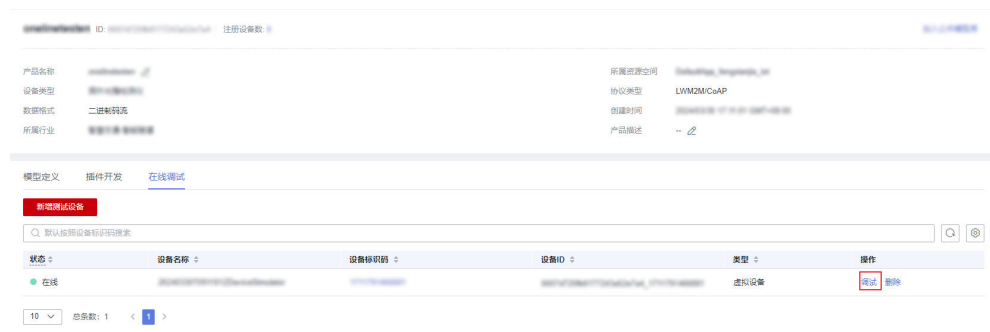
在弹出的“新增测试设备”窗口，选择“虚拟设备”，单击“确定”，创建一个虚拟设备。虚拟设备名称包含“Simulator”字样，每款产品下只能创建一个虚拟设备。

图 3-25 在线调试-创建虚拟设备



步骤3 单击“调试”，进入调试界面。

图 3-26 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。

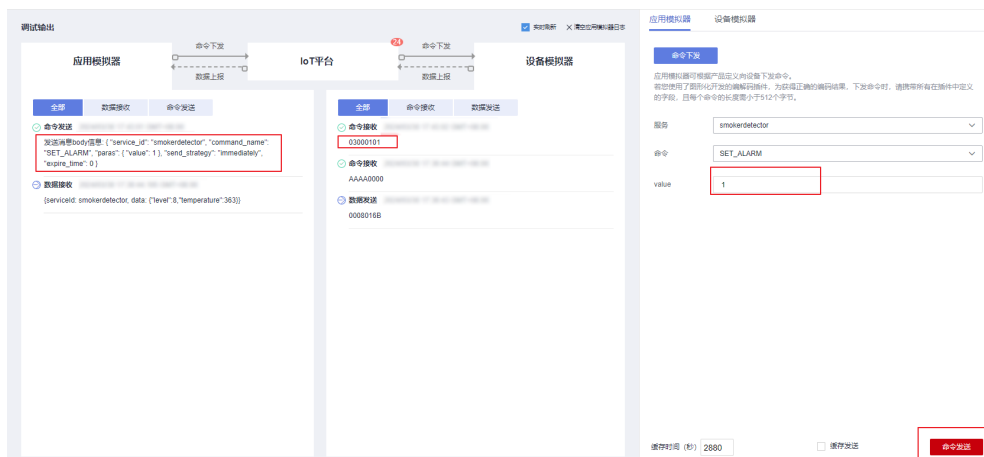
图 3-27 在线调试-模拟数据上报 smokerdetector



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 { "serviceld": "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 3-28 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 3-29 模型定义-smokerdetector 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 3-30 插件开发-新增消息 other_info

The screenshot shows the '新增消息' (Add Message) configuration window. It has a title bar with a close button. Below the title bar is a '基本信息' (Basic Information) section. It contains a text input for '消息名' (Message Name) with the value 'other_info', a radio button for '消息类型' (Message Type) with '数据上报' (Data Report) selected, and a checkbox for '添加响应字段' (Add Response Field) which is checked. To the right is a text area for '描述' (Description) with the placeholder '消息描述' (Message Description) and a character count '0/1,024'. Below this is a table for '字段' (Fields) with columns: '偏移值' (Offset), '名字' (Name), '描述' (Description), '数据类型' (Data Type), '长度' (Length), '是否地址域' (Is Address Field), and '操作' (Action). A '添加字段' (Add Field) button is located to the right of the table. Below the table is a message icon and the text '暂无表格数据' (No table data). At the bottom, there is a text input for '响应数据' (Response Data) with the value 'AAAA0000' and two buttons: '确定' (Confirm) and '取消' (Cancel).

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 3-31 插件开发-添加字段 messageId(0x2)

添加字段 [X]

i 只有标记为地址域时，名字固定为messageId；其他字段名字不能设置为messageId。

标记为地址域 [?]

* 字段名称:

描述: 0/1,024

数据类型 (大端模式):

偏移值: [?]

* 长度: [?]

默认值: [?]

确定 **取消**

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 3-32 插件开发-添加字段 other_info

×

添加字段

标记为地址域 ?

* 字段名称

描述
0/1,024

数据类型 (大端模式)

偏移值 ?

* 长度 ?

默认值 ?

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 3-33 插件开发-新增消息 other_info2

新增消息 ×

基本信息

*消息名

*消息类型 数据上报 命令下发

添加响应字段

描述

0/1,024

字段 添加字段

偏移值	名字	描述	数据类型	长度	是否地址域	操作
 暂无表格数据						

响应数据

确定 取消

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 3-34 插件开发-添加字段 messageId(0x3)

×

添加字段

i 只有标记为地址域时，名字固定为messageId；其他字段名字不能设置为messageId。

标记为地址域 ?

* 字段名称	messageId
描述	输入字段描述 0/1,024 ↴
数据类型 (大端模式)	int8u ▼
偏移值	0-1 ?
* 长度	1 ?
默认值	0x3 ?

确定
取消

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 3-35 插件开发-添加字段 length

添加字段

×

标记为地址域 ?

* 字段名称

描述 0/1,024 ↵

数据类型 (大端模式)

偏移值 ?

* 长度 ?

默认值 ?

确定
取消

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：0000101，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0000101，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 3-36 插件开发-添加字段 other_info 为 varstring

×

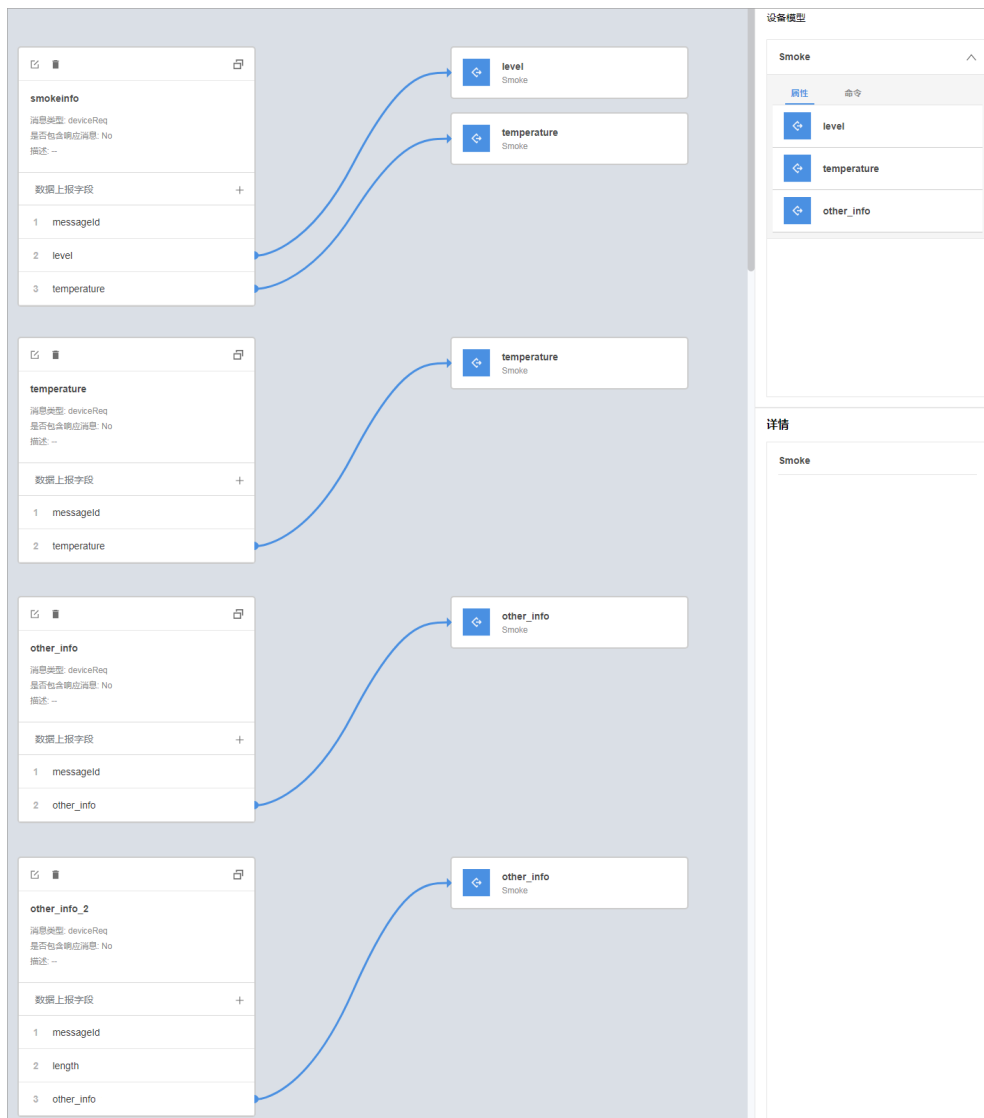
添加字段

标记为地址域 ?

* 字段名称	<input style="width: 90%;" type="text" value="other_info"/>
描述	<input style="width: 90%;" type="text" value="输入字段描述"/> 0/1,024
数据类型 (大端模式)	<input style="width: 90%;" type="text" value="varstring"/>
* 长度关联字段	<input style="width: 90%;" type="text" value="length"/> ?
* 掩码	<input style="width: 90%;" type="text" value="0xff"/> ?

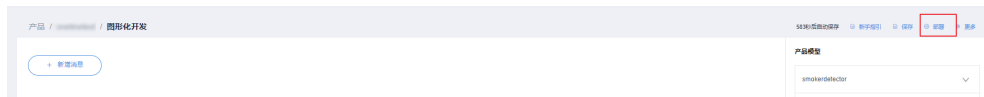
确定取消

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。



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 3-37 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模拟设备进行调测。具体调测步骤请参考[在线调试](#)。本文以模拟设备为例，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“Simulator”字样，每款产品下只能创建一个虚拟设备。

图 3-38 在线调试-创建虚拟设备



步骤3 单击“调试”，进入调试界面。

图 3-39 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 3-40 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。

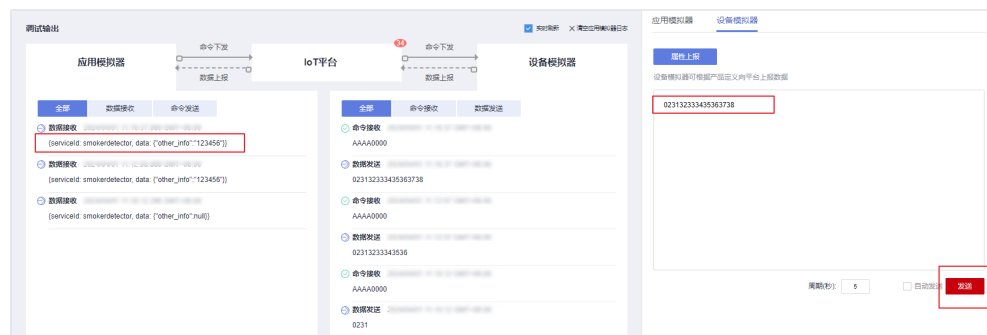
图 3-41 在线调试-模拟数据上报 other_info 长度合适



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。

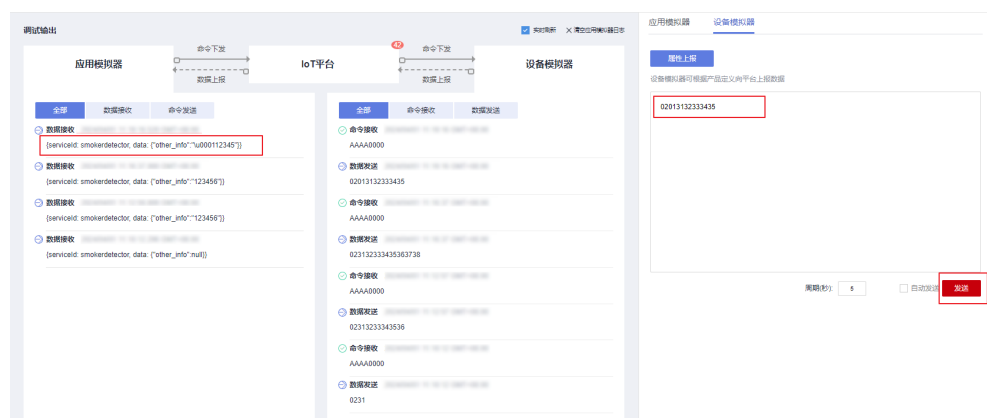
图 3-42 在线调试-模拟数据上报 other_info 长度过长



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0001。

图 3-43 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。

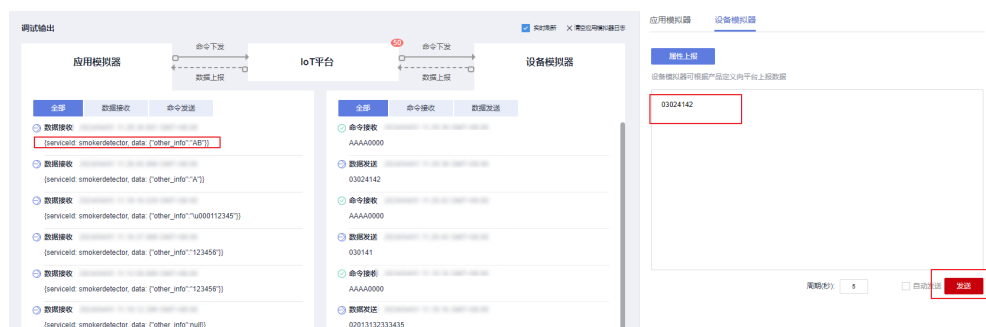
图 3-44 在线调试-模拟数据上报 other_info 可变长字符串 1



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 3-45 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 3-46 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 3-47 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 3-48 模型定义-smokerdetector 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 3-49 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 3-50 插件开发-添加字段 messageId(0x2)

添加字段 ×

i 只有标记为地址域时，名字固定为messageId；其他字段名字不能设置为messageId。

标记为地址域 ?

* 字段名称

描述 0/1,024

数据类型 (大端模式) ▼

偏移值 ?

* 长度 ?

默认值 ?

确定 取消

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 3-51 插件开发-添加字段 other_info 为 array

×

添加字段

标记为地址域 ?

* 字段名称	<input style="width: 90%;" type="text" value="other_info"/>
描述	<input style="width: 90%;" type="text" value="输入字段描述"/> 0/1,024
数据类型 (大端模式)	<input style="border-bottom: 1px solid #ccc;" type="text" value="array"/>
偏移值	<input style="background-color: #f0f0f0;" type="text" value="1-6"/> ?
* 长度	<input style="width: 90%;" type="text" value="5"/> ?
默认值	<input style="width: 90%;" type="text"/> ?

确定
取消

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 3-52 插件开发-新增消息 other_info2

新增消息

基本信息

*消息名
other_info2

描述
消息描述
0/1,024

*消息类型
 数据上报
 命令下发

添加响应字段

字段

偏移值	名字	描述	数据类型	长度	是否地址域	操作
 暂无表格数据						

添加字段

响应数据
AAAA0000

确定 取消

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 3-53 插件开发-添加字段 messageId(0x3)

×

添加字段

i 只有标记为地址域时，名字固定为messageId；其他字段名字不能设置为messageId。

标记为地址域 ?

* 字段名称	messageId
描述	输入字段描述 0/1,024
数据类型 (大端模式)	int8u v
偏移值	0-1 ?
* 长度	1 ?
默认值	0x3 ?

确定
取消

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 3-54 插件开发-添加字段 length

添加字段

×

标记为地址域 ?

* 字段名称

描述
0/1,024

数据类型 (大端模式)

偏移值 ?

* 长度 ?

默认值 ?

确定
取消

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：0000101，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0000101，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 3-55 插件开发-添加字段 other_info 为 variant

添加字段

×

标记为地址域 ?

* 字段名称

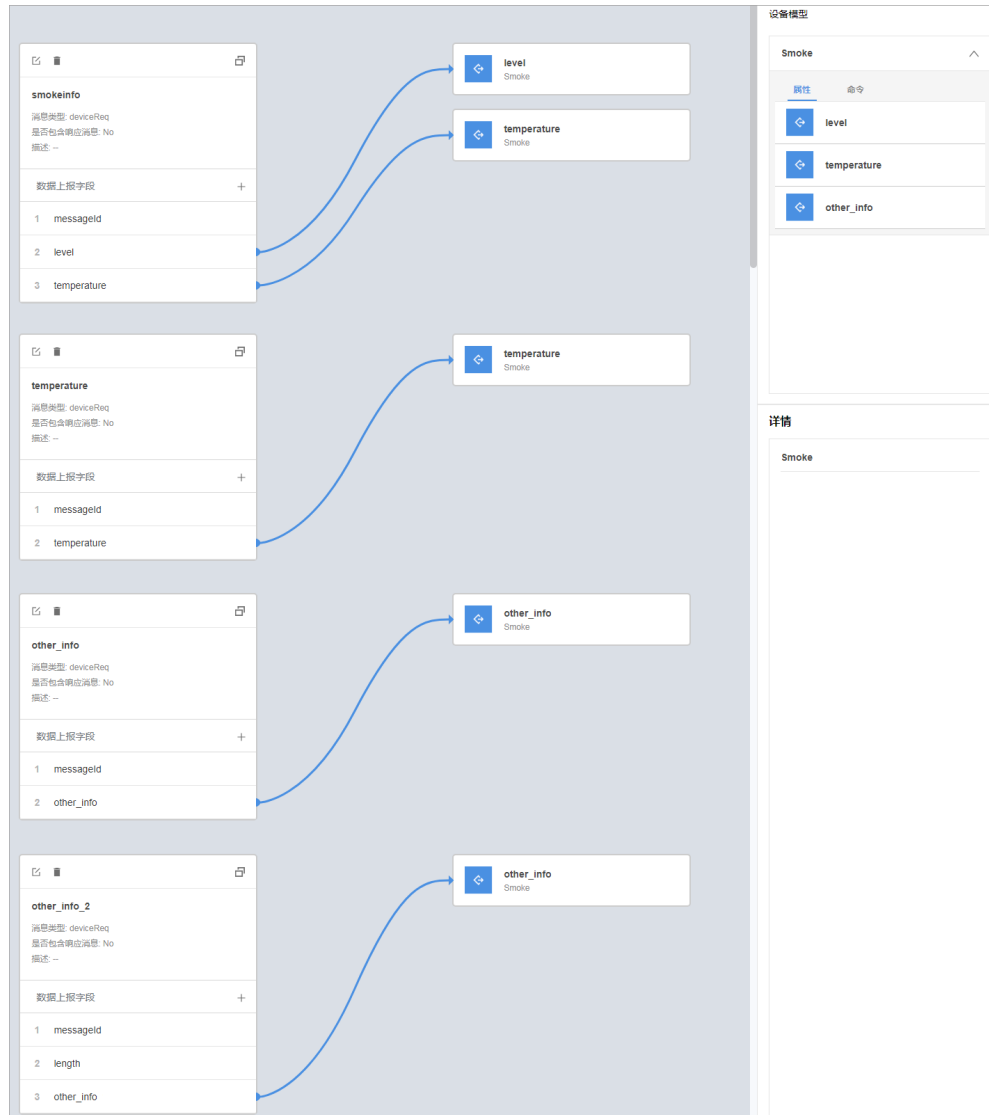
描述
0/1,024

数据类型 (大端模式)

* 长度关联字段 ?

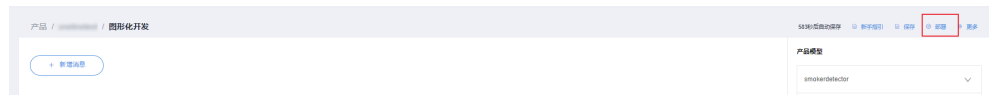
* 掩码 ?

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。



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 3-56 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模拟设备进行调测。具体调测步骤请参考[在线调试](#)。本文以模拟设备为例，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“Simulator”字样，每款产品下只能创建一个虚拟设备。

图 3-57 在线调试-创建虚拟设备



步骤3 单击“调试”，进入调试界面。

图 3-58 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 3-59 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：{serviceld: smokedetector, data: {"other_info": "ESlzRFU="}}。描述信息长度为5个字节，编解码插件解析成功。

图 3-60 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：{serviceld: smokedetector, data: {"other_info": "ESlZRFU="}}。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 3-61 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：{serviceld: smokedetector, data: {"other_info": "AQ=="}}。AQ==是01经过base64编码后的值。

图 3-62 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：`{servicId: smokedetector, data: {"other_info":"AQI="}}`。AQI是01经过base64编码后的值。

图 3-63 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 3-64 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：`{servicId: smokedetector, data: {"other_info":"AQID"}}`。AQID是010203经过base64编码后的值。

图 3-65 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 3-66 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（ $3 \times 8 = 24$ ）转化为4个6位字节（ $4 \times 6 = 24$ ），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（ $3 \times 8 = 24$ ）。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（ $3 \times 8 = 24$ ）。在转换为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。

3.4.3 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

- 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。

烟感设备样例

场景说明

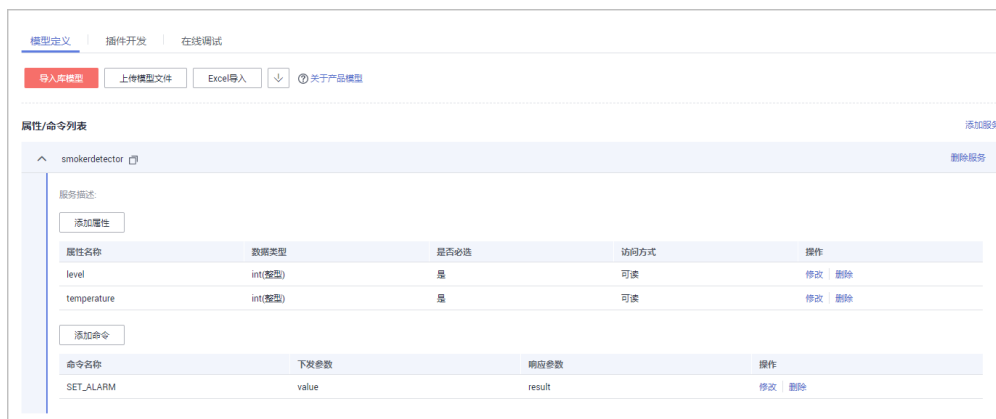
有一款烟感设备，具有如下特征：

- 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。



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”。



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":"smokerdetector","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":"smokerdector",
"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serviceld参数值, 该参数值对应产品模型中的服务类型smokerdector
        var serviceld = '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":serviceld,"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serviceld参数值, 该参数值对应产品模型中的服务类型smokerdector
        var serviceld = '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":
serviceld,"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。



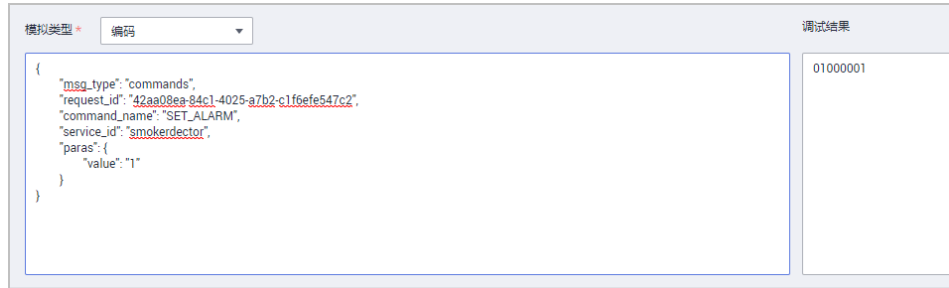
- 检查调试结果是否符合预期，如果不符合预期，则修改代码后重新进行调试。

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",

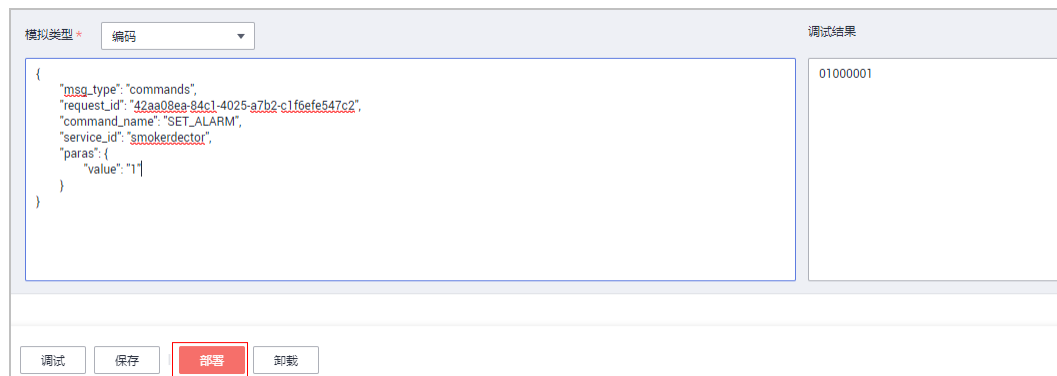

```
"command_name": "SET_ALARM",
"service_id": "smokerdetector",
"paras": {
  "value": "1"
}
}
```

- 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。



- 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。



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":"smokerdetector","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":"smokerdetector","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serviceld参数值, 该参数值对应产品模型中的服务类型smokerdetector
    var serviceld = 'smokerdetector';
    //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": serviceld, "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serviceld参数值, 该参数值对应产品模型中的服务类型smokerdetector
    var serviceld = 'smokerdetector';
    var command = dataView.getInt8(0); //从二进制码流中获取命令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": serviceld,
        "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": "smokerdetector", "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": "smokerdetector", "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":"smokerdetector","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serviceld参数值, 该参数值对应产品模型中的服务类型smokerdetector
    var serviceld = 'smokerdetector';
    //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":serviceld,"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":"smokerdetector","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：yyyyMMddTHHmmsZ），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: yyyyMMddTHHmmsZ), 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，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。
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<Service Property>	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。

3.5 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产品在线调测的功能，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。

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。模拟设备调测界面结构如下：



步骤1 在产品开发空间，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，单击右侧的“调试”，进入调试界面。

设备名称	设备标识码	设备ID	类型	操作
d2bd4d3b19ac5decnbsimulator	1594281108660	ce998692-361e-4016-8247-3859735108e0	虚拟设备	调试 删除

步骤4 在“设备模拟器”区域，输入十六进制码流或者JSON数据（以十六进制码流为例），单击“发送”，在“应用模拟器”区域查看数据上报的结果，在“消息跟踪”区域查看物联网平台处理日志。



步骤5 在“应用模拟器”区域进行命令下发，在“设备模拟器”区域查看接收到的命令（以十六进制码流为例），在“消息跟踪”区域查看物联网平台处理日志。



----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。真实设备调测界面结构如下：



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 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，单击确认。

新增测试设备

设备类型: 真实设备 虚拟设备

* 设备名称:

* 设备标识码:

设备注册方式: 不加密 加密

注：如果使用DTLS传输层安全协议接入时，设备注册方式选择“加密”，请妥善保管密钥。

📖 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。

设备名称	设备标识码	设备ID	类型	操作
streetlight	QR code	QR code	真实设备	<input checked="" type="button" value="调试"/> <input type="button" value="删除"/>

步骤4 模拟远程下发控制命令场景，在应用模拟器中，选择服务：StreetLight，命令：SWITCH_LIGHT，命令取值为：ON，单击“发送”，我们可以看到路灯被点亮。

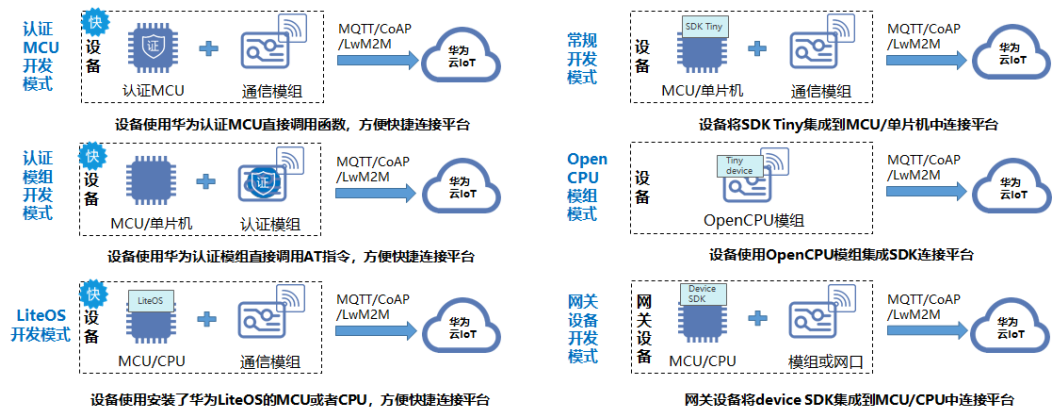
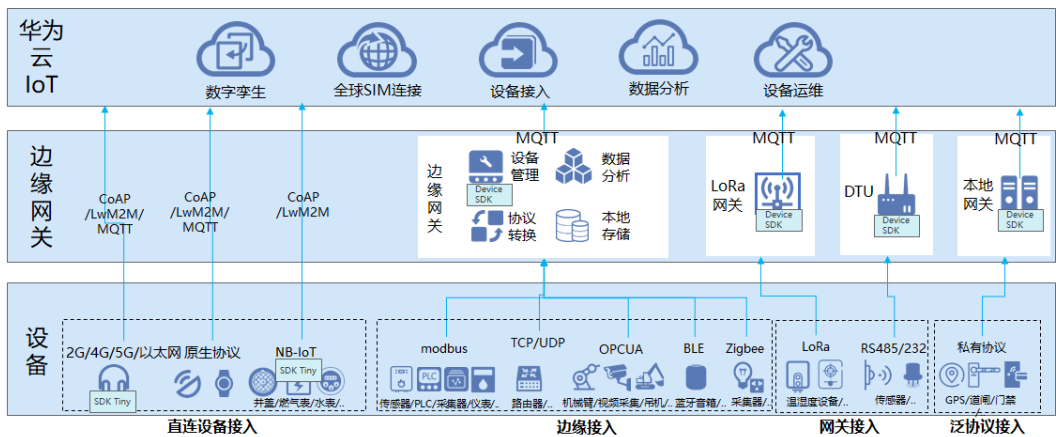
----结束

4 设备侧开发

4.1 设备接入指引

设备接入方式

华为物联网平台支持多种接入方式，满足各类设备和接入场景要求。请根据设备类型，选择合适的开发模式。



开发模式	特点	适用场景	难度系数
认证MCU开发模式	IoT Device SDK Tiny已经预集成在MCU (Main Control Unit) 中，可以直接调用方法连接华为云IoT。	设备商用时间短，研发成本低，无需管理子设备的直连设备。	★
认证模组开发模式	IoT Device SDK Tiny已经预集成在模组中，通过调用AT指令连接华为云IoT。	需要节省MCU资源，无需管理子设备的直连设备。	★
LiteOS开发模式	设备运行在LiteOS中，通过LiteOS对MCU进行资源管理，同时liteOS内置了IoT Device SDK Tiny，可以通过调用函数连接华为云IoT。LiteOS开发模式的设备开发耗时更短，开发难度也低。	无操作系统，无需管理子设备的直连设备	★★
常规开发模式	集成IoT Device SDK Tiny到MCU中，调用SDK中的函数连接华为云IoT，相比API接入更便捷。	设备商用时间充足，MCU的Flash与RAM资源满足IoT Device SDK Tiny集成条件的场景。	★★★
Open CPU开发模式	节省一个MCU，使用公用模组中的MCU能力，将设备的应用程序编译运行在OpenCPU上。	安全要求高，体积较小，商用时间短的设备	★★★★
网关设备开发模式	IoT Device SDK预置到CPU (Central Processing Unit) 或者MPU (Main Processing Unit) 中，通过调用函数连接到华为云IoT。	管理子设备的网关设备。	★★★

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
IoT Device SDK(Java)	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 IoT Device SDK使用指南 (Java) 。	IoT Device SDK(Java)

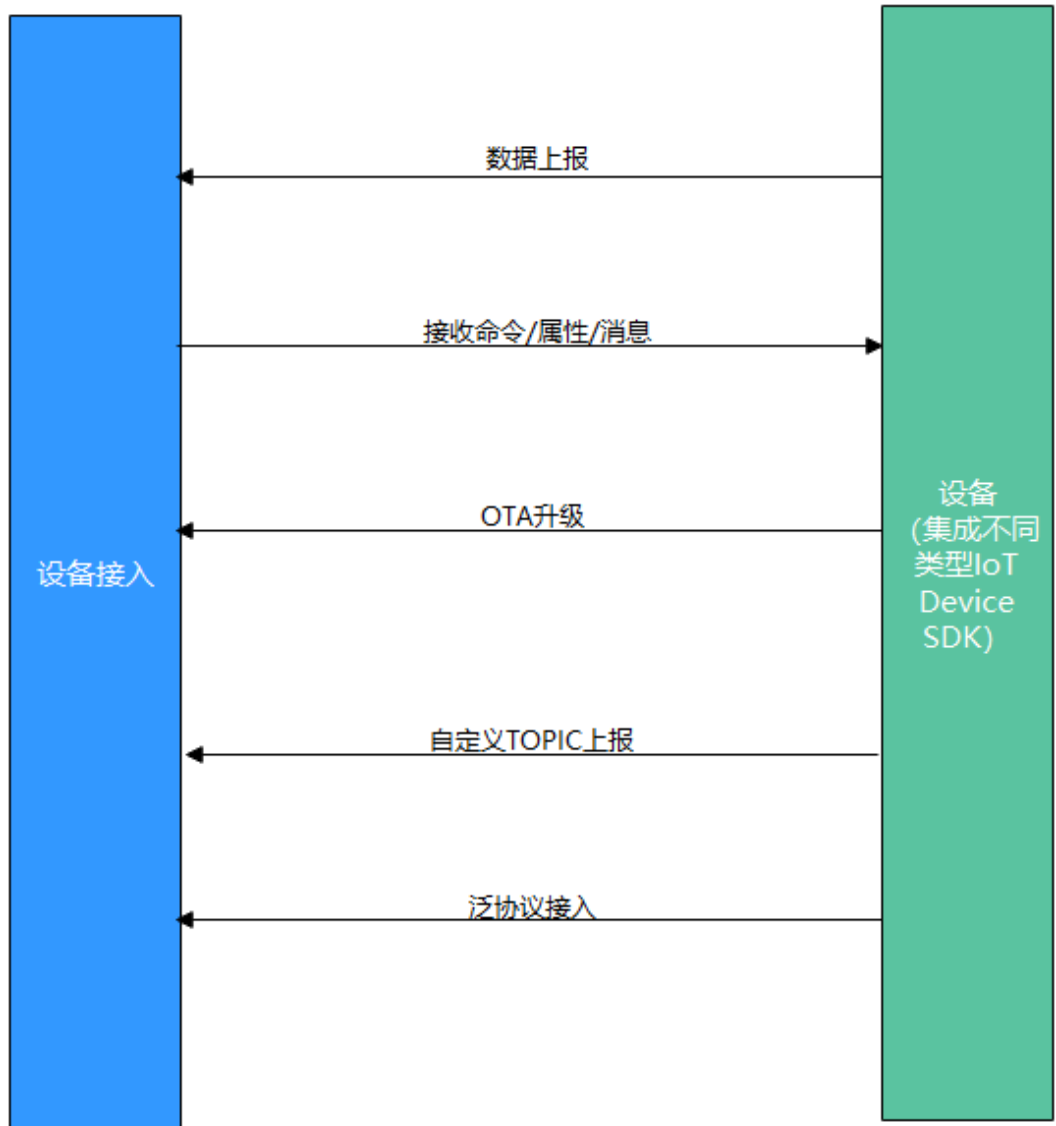
资源包名	描述	下载路径
IoT Device SDK(C)	设备可以通过集成IoT Device SDK(C)接入物联网平台, Demo提供了调用SDK接口的样例代码。使用指导请参考 IoT Device SDK(C)使用指南 。	IoT Device SDK(C)
IoT Device SDK(C#)	设备可以通过集成IoT Device SDK(C#)接入物联网平台, Demo提供了调用SDK接口的样例代码。使用指导请参考 IoT Device SDK(C#)使用指南 。	IoT Device SDK(C#)
IoT Device SDK(Android)	设备可以通过集成IoT Device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

资源包名	描述	下载路径
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
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

4.2 使用 IoT Device SDK 接入

4.2.1 IoT Device SDK 介绍

为了帮助设备快速连接到物联网平台，华为提供了IoT Device SDK。支持TCP/IP协议栈的设备集成IoT Device SDK后，可以直接与物联网平台通信。不支持TCP/IP协议栈的设备，例如蓝牙设备、ZigBee设备等需要利用网关将设备数据转发给物联网平台，此时网关需要事先集成IoT Device SDK。



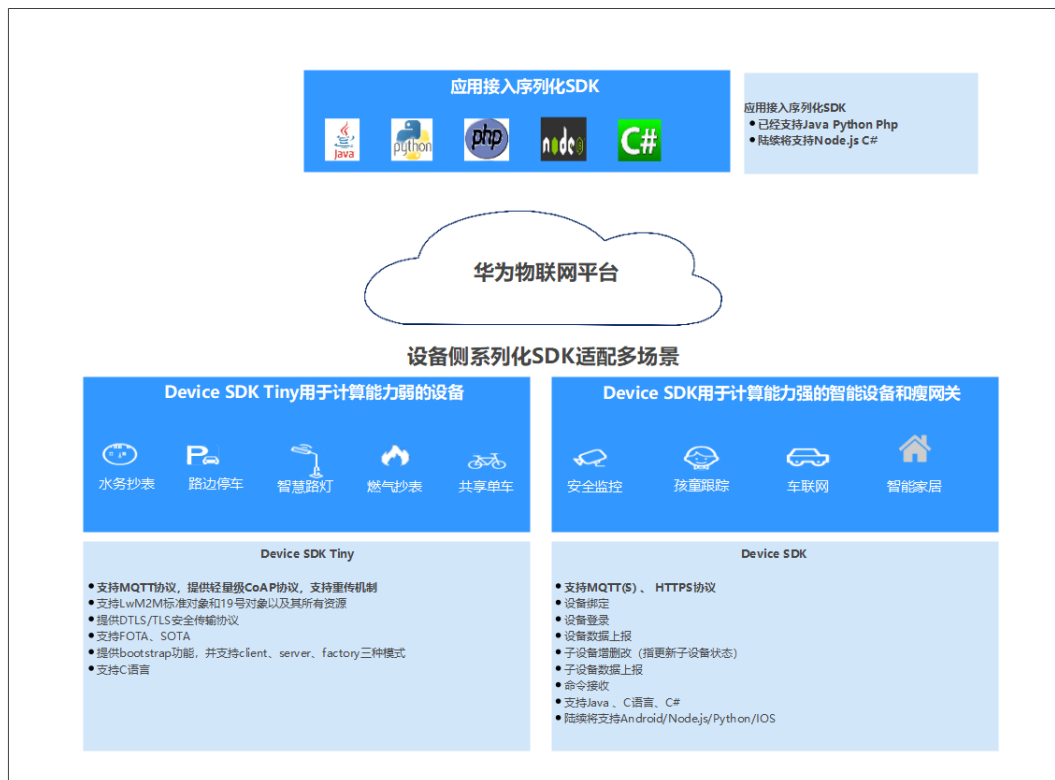
1. 设备接入前，需创建产品（可通过控制台创建或者调用应用侧API接口[创建产品](#)）。
2. 产品创建完毕后，需注册设备（可通过控制台注册单个设备或者使用应用侧API接口[注册设备](#)）。
3. 设备注册完毕后，按照图中流程实现消息/属性上报、接收命令/属性/消息、OTA升级、自定义TOPIC、泛协议接入（相关[Demo](#)）等功能。

平台提供了两种SDK，它们之间的区别如下表：

SDK种类	SDK集成场景	SDK支持的物联网通信协议
IoT Device SDK	面向运算、存储能力较强的嵌入式设备，例如网关、采集器等。	MQTT
IoT Device SDK Tiny	面向对功耗、存储、计算资源有苛刻限制的终端设备，例如单片机、模组。	LWM2M over CoAP、MQTT

对接入设备的硬件要求：

SDK名称	RAM容量	FLASH容量	CPU频率	操作系统类型	开发语言
IoT Device SDK	> 4MB	> 2MB	> 200MHZ	C版（Linux）、Java版（Linux/Windows）、C#版（Windows）、Android版（Android）、Go社区版（Linux/Windows/类unix）、OpenHarmony版（OpenHarmony）	C、Java、C#、Android、Go
IoT Device SDK Tiny	> 32KB	> 128KB	> 100MHZ	无特殊要求	C



详细SDK使用指南，请参考：

- [IoT Device SDK使用指南（C）](#)
- [IoT Device SDK使用指南（Java）](#)
- [IoT Device SDK使用指南（C#）](#)
- [IoT Device SDK使用指南（Android）](#)

- [IoT Device SDK使用指南（Go社区版）](#)
- [IoT Device SDK Tiny使用指南](#)
- [IoT Device SDK 使用指南（OpenHarmony）](#)
- [IoT Device SDK使用指南（Python）](#)

SDK主要功能矩阵，请参考：

表 4-1

主要功能	C	Java	C#	Android	GO	python	C Tiny
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	√	√	√	√	√	√	√
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）	√	×	×	×	×	×	×

主要功能	C	Java	C#	Android	GO	python	C Tiny
M2M功能	√	×	×	×	×	×	×
泛协议接入	√	√	√	√	×	√	×

4.2.2 IoT Device SDK 使用指南（Java）

准备工作

- 开发环境要求：已经安装JDK（版本1.8以上）和maven
- 访问[SDK下载页面](#)，下载SDK，整个工程包含以下子工程：

- ✔ iot-bridge-demo
- ✔ iot-bridge-sample-tcp-protocol
- ✔ iot-bridge-sdk
- ✔ iot-device-code-generator
- ✔ iot-device-demo
- ✔ iot-device-sdk-java
- ✔ iot-gateway-demo

iot-device-sdk-java：sdk代码

iot-device-demo：普通直连设备的demo代码

iot-gateway-demo：网关设备的demo代码

iot-bridge-sdk：网桥的sdk代码

iot-bridge-demo：网桥的demo代码，用来演示如何将tcp设备桥接到平台

iot-bridge-sample-tcp-protocol：子设备使用tcp协议链接网桥的样例

iot-device-code-generator：设备代码生成器，可以根据产品模型自动生成设备代码

- 编译安装：进入到SDK根目录，执行mvn install

创建产品

为了方便体验，我们提供了一个烟感的产品模型，烟感会上报烟雾值、温度、湿度、烟雾报警、还支持响铃报警命令。以烟感例，体验消息上报、属性上报等功能。

- 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，查看MQTTS设备接入域名，保存该地址。
- 步骤2** 单击左侧导航栏“产品”，单击页面左侧的“创建产品”。
- 步骤3** 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	平台自动将新创建的产品归属在默认资源空间下。如需归属在其他资源空间下，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	自定义。支持字母、数字、下划线（_）、连字符（-）的字符组合。
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	填写” smokeDetector ”
高级配置	
产品ID	不填写
产品描述	请根据实际情况填写。

----结束

上传产品模型

步骤1 单击下载烟感产品模型[smokeDetector](#)，获取产品模型文件。

步骤2 找到步骤3创建的产品，单击产品进入产品详情页。

步骤3 选择“模型定义”页签，单击“上传模型文件”，上传步骤1获取的产品模型文件。



----结束

注册设备

步骤1 选择左侧导航栏“设备 > 所有设备”，单击页面右上角的“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	确保和步骤3创建的产品归属在同一个资源空间。

参数名称	说明
所属产品	选择步骤3创建的产品。
设备标识码	即nodeID，设备唯一物理标识。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	设备密钥，可自定义。若不填写密钥，物联网平台会自动生成密钥。

设备注册成功后保存设备标识码、设备ID、密钥。

---结束

设备初始化

1. 创建设备时，需要写入在注册设备时获取的设备ID、密码，以及1中获取的设备对接信息，注意格式为 `ssl://域名信息:端口号` 或 `ssl://IP地址:端口号`

```
//例如在iot-device-demo文件 MessageSample.java中修改以下参数
IoTDevice device = new IoTDevice("ssl://域名信息8883",
    "5e06bfee334dd4f33759f5b3_demo", "mysecret", file);
```

注意

所有涉及设备ID和密码的文件均需要修改成对应的信息。

2. 建立连接。调用init接口，该接口是阻塞调用，如果建立连接成功会返回0。

```
if (device.init() != 0) {
    return;
}
```

如果连接成功就会打印：

```
2023-07-17 17:22:59 INFO MqttConnection:105 - Mqtt client connected. address :ssl://域名信息8883
```

3. 创建设备并连接成功后，可以开始使用设备进行通信。调用IoT Device 的getClient接口获取设备客户端，客户端提供了消息、属性、命令等通讯接口。

消息上报

消息上报是指设备向平台上报消息。

1. 从device中获取客户端，调用IoTDevice的getClient接口即可获取到客户端。
2. 调用客户端的reportDeviceMessage接口来上报设备消息。在MessageSample这个例子中我们周期性上报消息：

```
while (true) {
    device.getClient().reportDeviceMessage(new DeviceMessage("hello"), new ActionListener() {
        @Override
        public void onSuccess(Object context) {
            log.info("reportDeviceMessage ok");
        }
    });

    @Override
    public void onFailure(Object context, Throwable var2) {
```

```

        log.error("reportDeviceMessage fail: " + var2);
    }
});

//上报自定义topic消息，注意需要先在平台配置自定义topic
String topic = "$oc/devices/" + device.getDeviceId() + "/user/wpy";
device.getClient().publishRawMessage(new RawMessage(topic, "hello raw message "),
    new ActionListener() {
        @Override
        public void onSuccess(Object context) {
            log.info("publishRawMessage ok: ");
        }

        @Override
        public void onFailure(Object context, Throwable var2) {
            log.error("publishRawMessage fail: " + var2);
        }
    });

Thread.sleep(5000);
}

```

3. 修改MessageSample类的主函数，替换自己的设备参数后运行MessageSample类，查看日志打印看到连接成功和发送消息的打印：

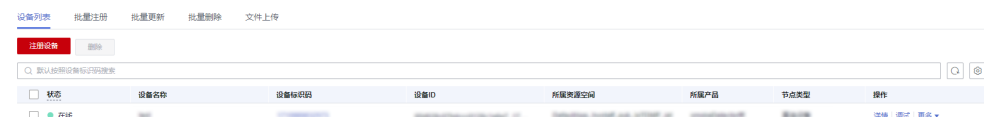
```

2024-04-16 16:43:09 INFO AbstractService:103 - create device, the deviceId is
5e06bfee334dd4f33759f5b3_demo
2024-04-16 16:43:09 INFO MqttConnection:233 - try to connect to ssl://域名信息:8883
2024-04-16 16:43:10 INFO MqttConnection:257 - connect success, the uri is ssl://域名信息:8883
2024-04-16 16:43:11 INFO MqttConnection:299 - publish message topic is $oc/devices/
5e06bfee334dd4f33759f5b3_demo/sys/events/up, msg =
{"object_device_id":"5e06bfee334dd4f33759f5b3_demo","services":[{"paras":
{"type":"DEVICE_STATUS","content":"connect
success","timestamp":"1713256990817"},"service_id":"$log","event_type":"log_report","event_time":"20
240416T084310Z","event_id":null}]}
2024-04-16 16:43:11 INFO MqttConnection:140 - Mqtt client connected. address is ssl://域名信息:8883
2024-04-16 16:43:11 INFO MqttConnection:299 - publish message topic is $oc/devices/
5e06bfee334dd4f33759f5b3_demo/sys/events/up, msg =
{"object_device_id":"5e06bfee334dd4f33759f5b3_demo","services":[{"paras":
{"device_sdk_version":"JAVA_v1.2.0","fw_version":null,"sw_version":null,"service_id":"$sdk_info","event
_type":"sdk_info_report","event_time":"20240416T084311Z","event_id":null}]}
2024-04-16 16:43:11 INFO MqttConnection:299 - publish message topic is $oc/devices/
5e06bfee334dd4f33759f5b3_demo/sys/events/up, msg =
{"object_device_id":"5e06bfee334dd4f33759f5b3_demo","services":[{"paras":
{"type":"DEVICE_STATUS","content":"connect complete, the url is ssl://域名信
息:8883","timestamp":"1713256991263"},"service_id":"$log","event_type":"log_report","event_time":"2
0240416T084311Z","event_id":null}]}
2024-04-16 16:43:11 INFO MqttConnection:299 - publish message topic is $oc/devices/
5e06bfee334dd4f33759f5b3_demo/sys/messages/up, msg =
{"name":null,"id":null,"content":"hello","object_device_id":null}
2024-04-16 16:43:11 INFO MqttConnection:299 - publish message topic is $oc/devices/
5e06bfee334dd4f33759f5b3_demo/user/wpy, msg = hello raw message
2024-04-16 16:43:11 INFO MessageSample:98 - reportDeviceMessage ok
2024-04-16 16:43:11 INFO MessageSample:113 - publishRawMessage ok:

```

4. 在设备接入控制台，选择“设备 > 所有设备”-查看设备是否在线。

图 4-1 设备列表-设备在线



5. 选择对应设备，单击“查看”，在设备详情页面启动设备消息跟踪。

图 4-2 消息跟踪-启动消息跟踪



6. 平台收到了设备的消息。

图 4-3 消息跟踪-查看 device_sdk_java 消息跟踪

业务类型	业务步骤	业务详情	记录时间	消息状态	操作
设备至平台	平台收到设备的消息上报	IoTDA has received the message reported by the device.data.hello raw message , app_id: adccc456...	19:17:22 GMT+08:00	成功	详情
设备至平台	平台收到设备的消息上报	IoTDA has received the message reported by the device.data {"name":"null","id":"null","content":"hello"}...	19:17:22 GMT+08:00	成功	详情
设备至平台	平台收到设备的消息上报	IoTDA has received the message reported by the device.data.hello raw message , app_id: adccc456...	19:17:21 GMT+08:00	成功	详情
设备至平台	平台收到设备的消息上报	IoTDA has received the message reported by the device.data {"name":"null","id":"null","content":"hello"}...	19:17:21 GMT+08:00	成功	详情
设备至平台	平台收到设备的消息上报	IoTDA has received the message reported by the device.data.hello raw message , app_id: adccc456...	19:17:18 GMT+08:00	成功	详情
设备至平台	平台收到设备的消息上报	IoTDA has received the message reported by the device.data.hello raw message , app_id: adccc456...	19:17:17 GMT+08:00	成功	详情
设备至平台	平台收到设备的消息上报	IoTDA has received the message reported by the device.data {"name":"null","id":"null","content":"hello"}...	19:17:17 GMT+08:00	成功	详情
设备至平台	平台收到设备的消息上报	IoTDA has received the message reported by the device.data {"name":"null","id":"null","content":"hello"}...	19:17:16 GMT+08:00	成功	详情
设备至平台	平台收到设备的消息上报	IoTDA has received the message reported by the device.data.hello raw message , app_id: adccc456...	19:17:12 GMT+08:00	成功	详情
设备至平台	平台收到设备的消息上报	IoTDA has received the message reported by the device.data {"name":"null","id":"null","content":"hello"}...	19:17:12 GMT+08:00	成功	详情

注：消息跟踪会有一些的延时，如果没有看到数据，请等待后刷新。

属性上报

打开PropertySample类，这个例子中会定时的上报alarm、temperature、humidity、smokeConcentration这四个属性。

```
//定时上报属性
while (true) {

    Map<String ,Object> json = new HashMap<>();
    Random rand = new Random();

    //按照物模型设置属性
    json.put("alarm", 1);
    json.put("temperature", rand.nextFloat()*100.0f);
    json.put("humidity", rand.nextFloat()*100.0f);
    json.put("smokeConcentration", rand.nextFloat() * 100.0f);

    ServiceProperty serviceProperty = new ServiceProperty();
    serviceProperty.setProperties(json);
    serviceProperty.setServiceId("smokeDetector");//serviceId要和物模型一致

    device.getClient().reportProperties(Arrays.asList(serviceProperty), new ActionListener() {
        @Override
        public void onSuccess(Object context) {
            log.info("pubMessage success" );
        }
    })
}
```

```

@Override
public void onFailure(Object context, Throwable var2) {
    log.error("reportProperties failed" + var2.toString());
}
});

Thread.sleep(10000);
}
}

```

修改PropertySample的main函数后直接运行PropertySample类，查看日志看到发送成功的打印

```

2024-04-17 15:38:37 INFO AbstractService:103 - create device, the deviceId is
5e06bf334dd4f33759f5b3_demo
2024-04-17 15:38:37 INFO MqttConnection:233 - try to connect to ssl://域名信息:8883
2024-04-17 15:38:38 INFO MqttConnection:257 - connect success, the uri is ssl://域名信息:8883
2024-04-17 15:38:38 INFO MqttConnection:299 - publish message topic is $oc/devices/
5e06bf334dd4f33759f5b3_demo/sys/events/up, msg =
{"object_device_id":"661e35467bdcc0126d1a595_feng-sdk-test3","services":[{"paras":
{"type":"DEVICE_STATUS","content":"connect
success","timestamp":"1713339518043"},"service_id":"$log","event_type":"log_report","event_time":"2024041
7T073838Z","event_id":null}]
2024-04-17 15:38:38 INFO MqttConnection:140 - Mqtt client connected. address is ssl://域名信息:8883
2024-04-17 15:38:38 INFO MqttConnection:299 - publish message topic is $oc/devices/
5e06bf334dd4f33759f5b3_demo/sys/events/up, msg =
{"object_device_id":"661e35467bdcc0126d1a595_feng-sdk-test3","services":[{"paras":
{"device_sdk_version":"JAVA_v1.2.0","fw_version":null,"sw_version":null},"service_id":"$sdk_info","event_type"
:"sdk_info_report","event_time":"20240417T073838Z","event_id":null}]
2024-04-17 15:38:38 INFO MqttConnection:299 - publish message topic is $oc/devices/
5e06bf334dd4f33759f5b3_demo/sys/events/up, msg =
{"object_device_id":"5e06bf334dd4f33759f5b3_demo","services":[{"paras":
{"type":"DEVICE_STATUS","content":"connect complete, the url is ssl://域名信
息:8883","timestamp":"1713339518464"},"service_id":"$log","event_type":"log_report","event_time":"202404
17T073838Z","event_id":null}]
2024-04-17 15:38:38 INFO MqttConnection:299 - publish message topic is $oc/devices/
5e06bf334dd4f33759f5b3_demo/sys/properties/report, msg = {"services":[{"properties":
{"alarm":1,"temperature":55.435158,"humidity":51.950867,"smokeConcentration":43.89913},"service_id":"sm
okeDetector","event_time":null}]
2024-04-17 15:38:38 INFO PropertySample:144 - pubMessage success

```

在平台设备详情页面可以看到最新上报的属性值：

图 4-4 查看上报数据-smokeDetector



属性读写

调用客户端的setPropertyListener方法来设置属性回调接口。在PropertySample这个例子中，我们实现了属性读写接口。

写属性处理：实现了alarm属性的写操作，其他属性不支持写操作。

读属性处理：将本地属性值按照接口格式进行拼装。

```

device.getClient().setPropertyListener(new PropertyListener() {

//处理写属性

```

```
@Override
public void onPropertiesSet(String requestId, List<ServiceProperty> services) {
    // 遍历service
    for (ServiceProperty serviceProperty : services) {

        log.info("OnPropertiesSet, servid is {}", serviceProperty.getServiceId());

        // 遍历属性
        for (String name : serviceProperty.getProperties().keySet()) {
            log.info("property name is {}", name);
            log.info("set property value is {}", serviceProperty.getProperties().get(name));
        }

    }
    // 修改本地的属性值
    device.getClient().respondPropsSet(requestId, lotResult.SUCCESS);
}

/**
 * 处理读属性。多数场景下，用户可以直接从平台读设备影子，此接口不用实现。
 * 但如果需要支持从设备实时读属性，则需要实现此接口。
 */
@Override
public void onPropertiesGet(String requestId, String servid) {
    log.info("OnPropertiesGet, the servid is {}", servid);
    Map<String, Object> json = new HashMap<>();
    Random rand = new SecureRandom();
    json.put("alarm", 1);
    json.put("temperature", rand.nextFloat() * 100.0f);
    json.put("humidity", rand.nextFloat() * 100.0f);
    json.put("smokeConcentration", rand.nextFloat() * 100.0f);

    ServiceProperty serviceProperty = new ServiceProperty();
    serviceProperty.setProperties(json);
    serviceProperty.setServiceId("smokeDetector");

    device.getClient().respondPropsGet(requestId, Arrays.asList(serviceProperty));
}
});
```

注：

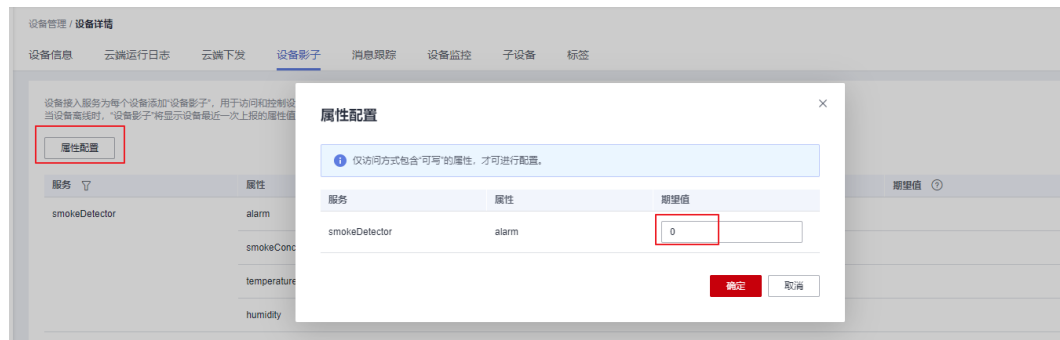
1. 属性读写接口需要调用respondPropsGet和respondPropsSet接口来上报操作结果。
 2. 如果设备不支持平台主动到设备读，onPropertiesGet接口可以空实现
- 运行PropertySample类，然后在平台上设备影子页面查看当前alarm属性值为1：

图 4-5 设备影子-查看 alarm 属性



我们把alarm属性修改为0：

图 4-6 设备影子-属性配置 alarm



查看设备侧日志，看到设备收到属性设置，alarm被修改为0：

```
2019-12-28 14:16:27 INFO MqttConnection:66 - messageArrived topic = $oc/devices/5e06bfee334dd4f33759f5b3_demo/sys/properties/set/reque
2019-12-28 14:16:27 INFO PropertySample:53 - OnPropertiesSet, serviceId = smokeDetector
2019-12-28 14:16:27 INFO PropertySample:57 - property name = alarm
2019-12-28 14:16:27 INFO PropertySample:58 - set property value = 0
```

命令下发

设置命令监听器用来接收平台下发的命令，在回调接口里，需要对命令进行处理，并上报响应。

在CommandSample例子中实现了命令的处理，收到命令后仅进行打印，然后调用respondCommand上报响应。

```
device.getClient().setCommandListener(new CommandListener() {
    @Override
    public void onCommand(String requestId, String serviceId, String commandName, Map<String,
Object> paras) {
        log.info("onCommand, serviceId = {}", serviceId);
        log.info("onCommand, name = {}", commandName);
        log.info("onCommand, paras = {}", paras.toString());

        //处理命令

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

直接运行CommandSample类，然后在平台上下发命令，命令的serviceId填smokeDetector、命令名填ringAlarm、参数携带duration为整数20。

查看日志，看到设备收到命令并上报了响应：

```
2019-12-28 15:03:36 INFO MqttConnection:66 - messageArrived topic = $oc/devices/test_testDevice/sys/commands/request_id=4, msg = {"paras":{"duration":20},"service_id":"smo
2019-12-28 15:03:36 INFO CommandSample:62 - onCommand, serviceId = smokeDetector
2019-12-28 15:03:36 INFO CommandSample:63 - onCommand, name = ringAlarm
2019-12-28 15:03:36 INFO CommandSample:64 - onCommand, paras = {duration=20}
2019-12-28 15:03:36 INFO MqttConnection:213 - publish message topic = $oc/devices/test_testDevice/sys/commands/response/request_id=4, msg = {"paras":null,"result_code":0,"
```

面向物模型编程

前面介绍了直接调用设备客户端的接口和平台进行通讯的方法，这种方式比较灵活，但用户需要妥善处理每一个接口，实现比较复杂。

SDK提供了一种更简单的方式，即面向物模型编程。面向物模型编程指基于SDK提供的物模型抽象能力，设备代码按照物模型定义设备服务，然后可以直接访问设备服务（即调用设备服务的属性读写接口），SDK就能自动和平台通讯，完成属性的同步和命令的调用。

相比直接调用客户端接口和平台进行通讯，面向物模型编程更简单，它简化了设备侧代码的复杂度，让设备代码只需要关注业务，而不用关注和平台的通讯过程。这种方式适合多数场景。

SmokeDetector例子演示了如何面向物模型编程：

1. 按照物模型定义服务类和服务的属性（如果有多个服务，则需要定义多个服务类）：

```
public static class SmokeDetectorService extends AbstractService {  
  
    //按照设备模型定义属性，注意属性的name和类型需要和模型一致，writeable表示属性知否可写，  
    name指定属性名  
    @Property(name = "alarm", writeable = true)  
    int smokeAlarm = 1;  
  
    @Property(name = "smokeConcentration", writeable = false)  
    float concentration = 0.0f;  
  
    @Property(writeable = false)  
    int humidity;  
  
    @Property(writeable = false)  
    float temperature;
```

用@Property注解来表示是一个属性，可以用name指定属性名，如果不指定则使用字段名。

属性可以加上writeable来控制权限，如果属性只读，则加上writeable = false，如果不加，默认认为可读写。

2. 定义服务的命令。设备收到平台下发的命令时，SDK会自动调用这里定义的命令。

接口入参和返回值的类型是固定的不能修改，否则会出现运行时错误。

这里定义的是一个响铃报警命令，命令名为ringAlarm，下发参数为”duration”，表示响铃报警的持续时间。

```
//定义命令，注意接口入参和返回值类型是固定的不能修改，否则会出现运行时错误  
@DeviceCommand(name = "ringAlarm")  
public CommandRsp alarm(Map<String, Object> paras) {  
    int duration = (int) paras.get("duration");  
    log.info("ringAlarm duration = " + duration);  
    return new CommandRsp(0);  
}
```

3. 定义getter和setter接口

- 当设备收到平台下发的查询属性以及设备上报属性时，会自动调用getter方法。getter方法需要读取设备的属性值，可以实时到传感器读取或者读取本地的缓存

- 当设备收到平台下发的设置属性时，会自动调用setter方法。setter方法需要更新设备本地的值。如果属性不支持写操作，setter保留空实现。

```
//setter和getter接口的命名应该符合java bean规范，sdk会自动调用这些接口  
public int getHumidity() {  
  
    //模拟从传感器读取数据  
    humidity = new Random().nextInt(100);  
    return humidity;  
}  
  
public void setHumidity(int humidity) {
```

```

//humidity是只读的，不需要实现
}

public float getTemperature() {

    //模拟从传感器读取数据
    temperature = new Random().nextInt(100);
    return temperature;
}

public void setTemperature(float temperature) {
    //只读字段不需要实现set接口
}

public float getConcentration() {

    //模拟从传感器读取数据
    concentration = new Random().nextFloat()*100.0f;
    return concentration;
}

public void setConcentration(float concentration) {
    //只读字段不需要实现set接口
}

public int getSmokeAlarm() {
    return smokeAlarm;
}

public void setSmokeAlarm(int smokeAlarm) {

    this.smokeAlarm = smokeAlarm;
    if (smokeAlarm == 0){
        log.info("alarm is cleared by app");
    }
}
}

```

4. 在main函数中创建服务实例并添加到设备。

```

//创建设备
IoTDevice device = new IoTDevice(serverUri, deviceId, secret);

//创建设备服务
SmokeDetectorService smokeDetectorService = new SmokeDetectorService();
device.addService("smokeDetector", smokeDetectorService);

if (device.init() != 0) {
    return;
}
}

```

5. 开启周期上报：

```

//启动自动周期上报
smokeDetectorService.enableAutoReport(10000);

```

备注：如果不想周期上报，也可以调用firePropertiesChanged接口手工触发上报。

直接运行SmokeDetector类，查看日志在上报属性：

```

2019-12-28 15:26:26 INFO MqttConnection:140 - try to connect to ssl://XXXXXXXXXXXXXXXXXXXXX.myhuaweicloud.com:8883
2019-12-28 15:26:26 INFO MqttConnection:147 - connect success ssl://XXXXXXXXXXXXXXXXXXXXX.myhuaweicloud.com:8883
2019-12-28 15:26:26 INFO MqttConnection:87 - Mqtt client connected. address :ssl://XXXXXXXXXXXXXXXXXXXXX.myhuaweicloud.com:8883
connect ok
2019-12-28 15:26:26 INFO MqttConnection:213 - publish message topic = $oc/devices/5e06bfee334dd4f33759f5b3_demo/sys/properties/report, msg = {"service

```

在平台侧查看设备影子：

图 4-7 设备影子-查看 alarm 属性



在平台上修改属性alarm，查看设备日志收到属性设置：

```
2019-12-28 15:44:29 INFO MqttConnection:66 - messageArrived topic = $oc/devices/test_testDevice/sys/properties/set/request_id=2, msg = {"services":[{"pr
2019-12-28 15:44:29 INFO AbstractService:187 - write property ok:alarm
```

在平台下发ringAlarm命令：

查看设备日志看到ringAlarm命令被调用，并且上报了响应：

```
2019-12-28 15:44:29 INFO MqttConnection:66 - messageArrived topic = $oc/devices/test_testDevice/sys/commands/request_id=1, msg = {"paras":{"duration":20}}
2019-12-28 15:44:29 INFO DeviceServiceSample$SmokeDetectorService:53 - ringAlarm duration = 20
2019-12-28 15:44:29 INFO MqttConnection:213 - publish message topic = $oc/devices/test_testDevice/sys/commands/response/request_id=1, msg = {"paras":null,
```

使用代码生成器

sdk提供了设备代码生成器，用户只需要提供产品模型文件，就能自动生成设备代码框架。代码生成器可以解析设备模型文件，然后对模型里定义每个服务，生成对应的service类，然后生成一个设备主类，在main函数中创建设备并注册设备服务实例。

使用代码生成器生成设备代码的步骤：

1. 下载huaweicloud-iot-device-sdk-java工程，解压缩后进入huaweicloud-iot-device-sdk-java目录执行“mvn install”。

```
[INFO] -----
[INFO] Reactor Summary for huaweicloud iot device sdk project for java 1.2.0:
[INFO]
[INFO] huaweicloud iot device sdk project for java ..... SUCCESS [ 0.802 s]
[INFO] iot-device-sdk-java ..... SUCCESS [ 3.976 s]
[INFO] iot-device-demo ..... SUCCESS [ 4.112 s]
[INFO] iot-bridge-sdk ..... SUCCESS [ 17.187 s]
[INFO] iot-bridge-demo ..... SUCCESS [ 4.168 s]
[INFO] iot-gateway-demo ..... SUCCESS [ 2.852 s]
[INFO] iot-device-code-generator ..... SUCCESS [ 2.658 s]
[INFO] iot-bridge-sample-tcp-protocol ..... SUCCESS [ 4.122 s]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 39.978 s
[INFO] Finished at: 2023-06-16T11:25:00+08:00
[INFO] -----
```

2. 执行完成会在iot-device-code-generator的target下生成可执行jar包。

D:\git\huaweicloud-iot-device-sdk-java\iot-device-code-generator\target

Name	Date modified
apidocs	6/16/2023 11:24 AM
classes	6/16/2023 11:24 AM
generated-sources	6/16/2023 11:24 AM
javadoc-bundle-options	6/16/2023 11:24 AM
maven-archiver	6/16/2023 11:24 AM
iot-device-code-generator-1.2.0.jar	6/16/2023 11:24 AM
iot-device-code-generator-1.2.0-javadoc...	6/16/2023 11:24 AM
iot-device-code-generator-1.2.0-sources...	6/16/2023 11:24 AM
iot-device-code-generator-1.2.0-with-de...	6/16/2023 11:24 AM

3. 将产品模型文件保存到本地，比如我的模型文件“smokeDetector.zip”放到D盘。
4. 访问SDK根目录，执行“java -jar .\iot-device-code-generator\target\iot-device-code-generator-1.2.0-with-deps.jar D:\smokeDetector.zip”。

```
PS D:\git\huaweicloud-iot-device-sdk-java> java -jar .\iot-device-code-generator\target\iot-device-code-generator-1.2.0-with-deps.jar D:\smokeDetector.zip
2023-06-16 11:30:47 INFO DeviceCodeGenerator:147 - the file generation path is :D:\git\huaweicloud-iot-device-sdk-java\generated-demo\src\main\java\com\huaweicloud\sd\iot\device\demo\smokeDetectorService.java
2023-06-16 11:30:47 INFO DeviceCodeGenerator:73 - demo code generated to: D:\git\huaweicloud-iot-device-sdk-java\generated-demo
```

5. 在huaweicloud-iot-device-sdk-java目录下会生成generated-demo包。

D:\git\huaweicloud-iot-device-sdk-java\generated-demo

Name
src
target
mvnw
mvnw.cmd
pom.xml

至此，设备代码已经生成。

编译运行生成的代码：

1. 访问“huaweicloud-iot-device-sdk-java\generated-demo”，执行“mvn install”，在target下生成jar包。

```
PS D:\git\huaweicloud-iot-device-sdk-java> cd .\generated-demo\
PS D:\git\huaweicloud-iot-device-sdk-java\generated-demo> mvn install
```



```
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 4.386 s
[INFO] Finished at: 2023-06-16T11:31:47+08:00
[INFO] -----
PS D:\git\huaweicloud-iot-device-sdk-java\generated-demo> dir target

Directory: D:\git\huaweicloud-iot-device-sdk-java\generated-demo\target

Mode                LastWriteTime         Length Name
----                -
d-----            6/16/2023  11:31 AM             apidocs
d-----            6/16/2023  11:31 AM             classes
d-----            6/16/2023  11:31 AM          generated-sources
d-----            6/16/2023  11:31 AM          javadoc-bundle-options
d-----            6/16/2023  11:31 AM          maven-archiver
-a----            6/16/2023  11:31 AM          29924 iot-device-demo-ganerated-1.2.0-javado
c.jar
-a----            6/16/2023  11:31 AM          6728 iot-device-demo-ganerated-1.2.0-source
s.jar
-a----            6/16/2023  11:31 AM         11530020 iot-device-demo-ganerated-1.2.0-with-d
eps.jar
-a----            6/16/2023  11:31 AM          8031 iot-device-demo-ganerated-1.2.0.jar
```

2. 执行 `java -jar .\target\iot-device-demo-ganerated-1.2.0-with-deps.jar ssl://域名信息:8883 device_id secret`, 三个参数分别为设备接入地址、设备id和密码, 运行生成的demo。

```
D:\git\huaweicloud-iot-device-sdk-java\generated-demo>java -jar .\target\iot-device-demo-ganerated-1.2.0-with-deps.jar ssl://域名信息:8883 5e06bfee334dd4f33759f5b3_demo secret
2024-04-17 15:50:53 INFO AbstractService:73 - create device, the deviceId is 5e06bfee334dd4f33759f5b3_demo
2024-04-17 15:50:54 INFO MqttConnection:204 - try to connect to ssl://域名信息:8883
2024-04-17 15:50:55 INFO MqttConnection:228 - connect success, the uri is ssl://域名信息:8883
2024-04-17 15:50:55 INFO MqttConnection:268 - publish message topic is $oc/devices/5e06bfee334dd4f33759f5b3_demo/sys/events/up, msg =
{"object_device_id":"5e06bfee334dd4f33759f5b3_demo","services":[{"paras":
{"type":"DEVICE_STATUS","content":"connect
success","timestamp":"1713340255148"},"service_id":"$log","event_type":"log_report","event_time":"20
240417T075055Z","event_id":null}}]
2024-04-17 15:50:55 INFO MqttConnection:111 - Mqtt client connected. address is ssl://域名信息:8883
2024-04-17 15:50:55 INFO MqttConnection:268 - publish message topic is $oc/devices/5e06bfee334dd4f33759f5b3_demo/sys/events/up, msg =
{"object_device_id":"5e06bfee334dd4f33759f5b3_demo","services":[{"paras":
{"device_sdk_version":"JAVA_v1.2.0","fw_version":null,"sw_version":null,"service_id":"$sdk_info","event
_type":"sdk_info_report","event_time":"20240417T075055Z","event_id":null}}]
2024-04-17 15:50:55 INFO MqttConnection:268 - publish message topic is $oc/devices/5e06bfee334dd4f33759f5b3_demo/sys/events/up, msg =
{"object_device_id":"5e06bfee334dd4f33759f5b3_demo","services":[{"paras":
{"type":"DEVICE_STATUS","content":"connect complete, the url is ssl://域名信
息:8883","timestamp":"1713340255496"},"service_id":"$log","event_type":"log_report","event_time":"2
0240417T075055Z","event_id":null}}]
2024-04-17 15:51:03 INFO smokeDetectorService:78 - report property alarm value = 50
2024-04-17 15:51:03 INFO smokeDetectorService:104 - report property temperature value =
0.3648571367849047
2024-04-17 15:51:03 INFO smokeDetectorService:91 - report property smokeConcentration value =
0.679772877336927
2024-04-17 15:51:03 INFO smokeDetectorService:117 - report property humidity value = 15
2024-04-17 15:51:03 INFO MqttConnection:268 - publish message topic is $oc/devices/5e06bfee334dd4f33759f5b3_demo/sys/properties/report, msg = {"services":[{"properties":
{"alarm":50,"temperature":0.3648571367849047,"smokeConcentration":0.679772877336927,"humidity
":15},"service_id":"smokeDetector","event_time":"20240417T075103Z"}]}
```

修改扩展生成的代码:

生成的代码已经完成了服务的定义和注册, 用户只需要进行少量的修改即可。

1. 命令接口，需要添加具体的实现逻辑

```
/****** commands *****/  
@DeviceCommand  
public CommandRsp ringAlarm (Map<String, Object> paras) {  
    //todo 请在这里添加命令处理代码  
    return new CommandRsp(0);  
}
```

2. getter方法，生成的代码是返回随机值，需要改为从传感器读取数据。
3. setter方法，生成的代码只完成了属性的修改保存，还需要添加真实的逻辑处理，比如向传感器下发指令。

如何开发网关

网关是一个特殊的设备，除具备一般设备功能之外，还具有子设备管理、子设备消息转发的功能。SDK提供了AbstractGateway抽象类来简化网关的实现。该类提供了子设备管理功能，需要从平台获取子设备信息并保存（需要子类提供子设备持久化接口）、子设备下行消息转发功能（需要子类实现转发处理接口）、以及上报子设备列表、上报子设备属性、上报子设备状态、上报子设备消息等接口。

- **使用AbstractGateway类**

继承该类，在构造函数里提供子设备信息持久化接口，实现其下行消息转发的抽象接口：

```
public abstract void onSubdevCommand(String requestId, Command command);  
public abstract void onSubdevPropertiesSet(String requestId, PropsSet propsSet);  
public abstract void onSubdevPropertiesGet(String requestId, PropsGet propsGet);  
public abstract void onSubdevMessage(DeviceMessage message);
```

- **iot-gateway-demo代码介绍**

工程iot-gateway-demo基于AbstractGateway实现了一个简单的网关，它提供tcp设备接入能力。关键类：

SimpleGateway：继承自AbstractGateway，实现子设备管理和下行消息转发

StringTcpServer：基于netty实现一个TCP server，本例中子设备采用TCP协议，并且首条消息为鉴权消息

SubDevicesFilePersistence：子设备信息持久化，采用json文件来保存子设备信息，并在内存中做了缓存

Session：设备会话类，保存了设备id和TCP的channel的对应关系

- **SimpleGateway类**

添加或删除子设备处理

添加子设备：AbstractGateway的onAddSubDevices接口已经完成了子设备信息的保存。我们不需要再增加额外处理，因此SimpleGateway不需要重写onAddSubDevices接口

删除子设备：我们不仅需要修改持久化信息，还需要断开当前子设备的连接。所以我们重写了onDeleteSubDevices接口，增加了拆链处理，然后调用父类的onDeleteSubDevices。

```
@Override
public int onDeleteSubDevices(SubDevicesInfo subDevicesInfo) {

    for (DeviceInfo subdevice : subDevicesInfo.getDevices()) {
        Session session = nodeIdToSesseionMap.get(subdevice.getNodeId());
        if (session != null) {
            if (session.getChannel() != null) {
                session.getChannel().close();
                channelIdToSessionMap.remove(session.getChannel().id().asLongText());
                nodeIdToSesseionMap.remove(session.getNodeId());
            }
        }
    }
    return super.onDeleteSubDevices(subDevicesInfo);
}
```

- **下行消息处理**

网关收到平台下行消息时，需要转发给子设备。平台下行消息分为三种：设备消息、属性读写、命令。

- **设备消息：**这里我们需要根据deviceId获取nodeId，从而获取session，从session里获取channel，就可以往channel发送消息。在转发消息时，可以根据需要进行一定的转换处理。

```
@Override
public void onSubdevMessage(DeviceMessage message) {

    //平台接口带的都是deviceId，deviceId是由nodeId和productId拼装生成的，即
    //deviceId = productId_nodeId
    String nodeId = lotUtil.getNodeIdFromDeviceId(message.getDeviceId());
    if (nodeId == null) {
        return;
    }

    //通过nodeId获取session，进一步获取channel
    Session session = nodeIdToSesseionMap.get(nodeId);
    if (session == null) {
        log.error("subdev is not connected " + nodeId);
        return;
    }
    if (session.getChannel() == null){
        log.error("channel is null " + nodeId);
        return;
    }

    //直接把消息转发给子设备
    session.getChannel().writeAndFlush(message.getContent());
    log.info("writeAndFlush " + message);
}
```

- **属性读写：**

属性读写包括属性设置和属性查询。

属性设置：

```
@Override
public void onSubdevPropertiesSet(String requestId, PropsSet propsSet) {

    if (propsSet.getDeviceId() == null) {
        return;
    }

    String nodeId = lotUtil.getNodeIdFromDeviceId(propsSet.getDeviceId());
    if (nodeId == null) {
        return;
    }

    Session session = nodeIdToSesseionMap.get(nodeId);
    if (session == null) {
```

```
        return;
    }

    //这里我们直接把对象转成string发给子设备，实际场景中可能需要进行一定的编解码转换
    session.getChannel().writeAndFlush(JsonUtil.convertObject2String(propsSet));

    //为了简化处理，我们在这里直接响应。更合理做法是在子设备处理完后再响应
    getClient().respondPropsSet(requestId, lotResult.SUCCESS);

    log.info("writeAndFlush " + propsSet);
}
}
```

属性查询:

```
@Override
public void onSubdevPropertiesGet(String requestId, PropsGet propsGet) {

    //不建议平台直接读子设备的属性，这里直接返回失败
    log.error("not supporte onSubdevPropertiesGet");
    deviceClient.respondPropsSet(requestId, lotResult.FAIL);
}
}
```

- **命令:** 处理流程和消息类似，实际场景中可能需要不同的编解码转换。

```
@Override
public void onSubdevCommand(String requestId, Command command) {

    if (command.getDeviceId() == null) {
        return;
    }

    String nodeId = lotUtil.getNodeIdFromDeviceId(command.getDeviceId());
    if (nodeId == null) {
        return;
    }

    Session session = nodeIdToSesseionMap.get(nodeId);
    if (session == null) {
        return;
    }

    //这里我们直接把command对象转成string发给子设备，实际场景中可能需要进行一定的编解码转换
    session.getChannel().writeAndFlush(JsonUtil.convertObject2String(command));

    //为了简化处理，我们在这里直接回命令响应。更合理做法是在子设备处理完后再响应
    getClient().respondCommand(requestId, new CommandRsp(0));
    log.info("writeAndFlush " + command);
}
}
```

- **上行消息处理**

上行处理在StringTcpServer的channelRead0接口里。如果会话不存在，需要先创建会话:

如果子设备信息不存在，这里会创建会话失败，直接拒绝连接

```
@Override
protected void channelRead0(ChannelHandlerContext ctx, String s) throws Exception {
    Channel incoming = ctx.channel();
    log.info("channelRead0" + incoming.remoteAddress() + " msg :" + s);

    //如果是首条消息,创建session
    //如果是首条消息,创建session
    Session session = simpleGateway.getSessionByChannel(incoming.id().asLongText());
    if (session == null) {
        String nodeId = s;
        session = simpleGateway.createSession(nodeId, incoming);

        //创建会话失败，拒绝连接
        if (session == null) {
            log.info("close channel");
        }
    }
}
```

```

        ctx.close();
    }
}

```

如果会话存在，则进行消息转发：

```

else {
    //如果需要上报属性则调用reportSubDeviceProperties
    DeviceMessage deviceMessage = new DeviceMessage(s);
    deviceMessage.setDeviceId(session.getDeviceId());
    simpleGateway.reportSubDeviceMessage(deviceMessage, null);
}

```

到这里，网关的关键代码介绍完了，其他的部分看源代码。整个demo是开源的，用户可以根据需要进行扩展。比如修改持久化方式、转发中增加消息格式的转换、实现其他子设备接入协议。

● **iot-gateway-demo的使用**

- a. 创建子设备的产品，步骤可参考[创建产品](#)。
- b. 在创建的产品中定义模型，添加服务，服务ID为parameter。并且新增alarm和temperature两个属性，如下图所示

图 4-8 模型定义-子设备产品



- c. 修改StringTcpServer的main函数，替换构造参数，然后运行该类。

```

simpleGateway = new SimpleGateway(new SubDevicesFilePersistence(),
    "ssl://iot-acc.cn-north-4.myhuaweicloud.com:8883",
    "5e06bfec334dd4f33759f5b3_demo", "mysecret");

```

- d. 在平台上看到该网关在线后，添加子设备。

图 4-9 设备-添加子设备

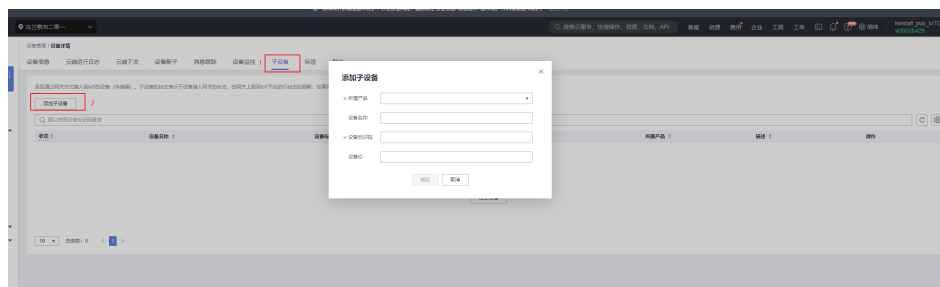


表 4-2 子设备参数

	参数描述
所属产品	子设备所属的产品，选择 步骤1 创建的产品。
设备名称	即device_name，可自定义，如subdev_name

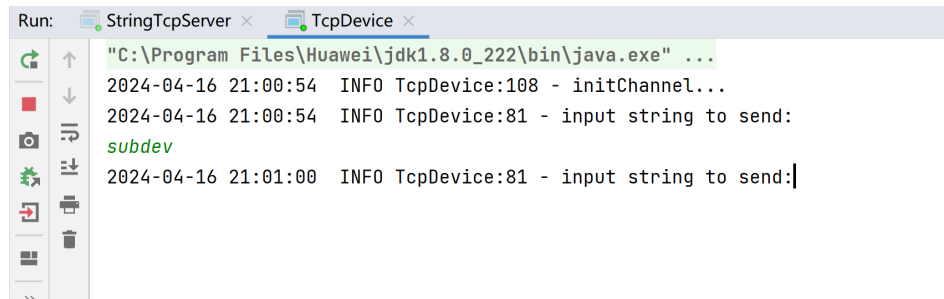
	参数描述
设备标识码	即node_id, 填写subdev。
设备ID	即device_id, 可不填写, 自动生成。

此时网关上日志打印:

```
2024-04-16 21:00:01 INFO SubDevicesFilePersistence:112 - add subdev,
the nodeId is subdev
```

- e. 运行TcpDevice类, 建立连接后, 输入步骤3中注册的子设备的nodeId, 如subdev。

图 4-10 子设备连接

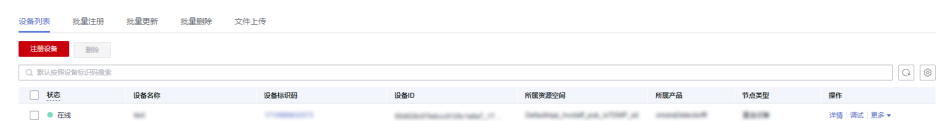


此时网关设备日志打印:

```
2024-04-16 21:00:54 INFO StringTcpServer:196 - initChannel: /127.0.0.1:21889
2024-04-16 21:01:00 INFO StringTcpServer:137 - channelRead0 is /127.0.0.1:21889, the msg is
subdev
2024-04-16 21:01:00 INFO SimpleGateway:100 - create new session ok, the session is
Session{nodeId='subdev', channel=[id: 0xf9b89f78, L:/127.0.0.1:8080 - R:/127.0.0.1:21889],
deviceId='subdev_deviceId'}
```

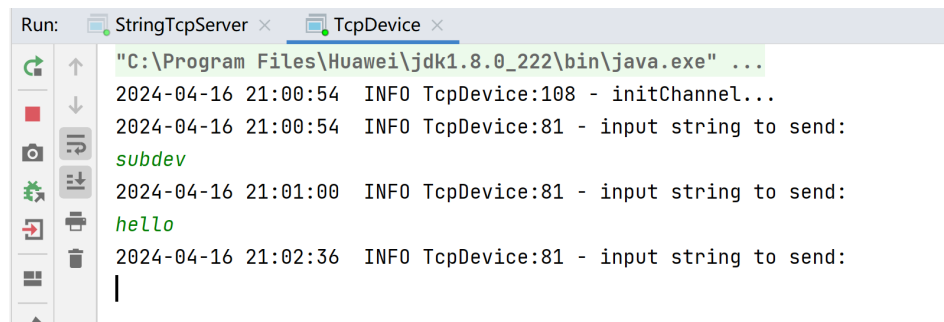
- f. 在平台上看到子设备上线。

图 4-11 设备列表-设备在线



- g. 子设备上报消息

图 4-12 子设备上报消息



查看日志看到上报成功

```
2024-04-16 21:02:36 INFO StringTcpServer:137 - channelRead0 is /127.0.0.1:21889, the msg is hello
2024-04-16 21:02:36 INFO MqttConnection:299 - publish message topic is $oc/devices/5e06bfee334dd4f33759f5b3_demo/sys/messages/up, msg = {"name":null,"id":null,"content":"hello","object_device_id":"subdev_deviceld"}
2024-04-16 21:02:36 INFO MqttConnection:299 - publish message topic is $oc/devices/5e06bfee334dd4f33759f5b3_demo/sys/gateway/sub_devices/properties/report, msg = {"devices":[{"services":[{"properties":{"temperature":2,"alarm":1},"service_id":"parameter","event_time":null},"device_id":"subdev_deviceld"}]}
```

h. 查看消息跟踪

在平台上找到网关，选择 设备详情-消息跟踪，打开消息跟踪。继续让子设备发送数据，等待片刻后看到消息跟踪：

图 4-13 消息跟踪-直连设备



4.2.3 IoT Device SDK 使用指南（C）

IoT Device SDK（C）提供设备接入华为云IoT物联网平台的C版本的SDK，提供设备和平台之间通讯能力，以及设备服务、网关服务、OTA等高级服务，并且针对各种场景提供了丰富的demo代码。相关集成指导请参考[IoT Device SDK（C）使用指南](#)。

4.2.4 IoT Device SDK 使用指南（C#）

IoT Device SDK（C#）提供设备接入华为云IoT物联网平台的C#版本的SDK，提供设备和平台之间通讯能力，以及设备服务、OTA等高级服务，并且针对各种场景提供了丰富的demo代码。相关集成指导请参考[IoT Device SDK（C#）使用指南](#)。

4.2.5 IoT Device SDK 使用指南（Android）

IoT Device SDK（Android）提供设备接入华为云IoT物联网平台的Android版本的SDK，提供设备和平台之间通讯能力，以及设备服务、OTA等高级服务，并且针对各种场景提供了丰富的demo代码。相关集成指导请参考[IoT Device SDK（Android）使用指南](#)。

4.2.6 IoT Device SDK 使用指南（Go 社区版）

Go语言版的SDK提供了跟平台基础的通信能力，由开源社区提供，如果使用有问题请在[github](#)上提issue。

4.2.7 IoT Device SDK Tiny 使用指南（C）

IoT Device SDK Tiny是部署在具备广域网能力、对功耗/存储/计算资源有苛刻限制的终端设备上的轻量级互联互通中间件，您只需调用API接口，便可实现设备快速接入到物联网平台以及数据上报和命令接收等功能。

📖 说明

IoT Device SDK Tiny可以运行于无linux操作系统的设备，也可以被模组集成，但是不提供网关服务。

4.2.8 IoT Device SDK 使用指南（OpenHarmony）

IoT Device SDK（OpenHarmony）提供设备接入华为云IoT物联网平台的OpenHarmony版本的SDK，提供设备和平台之间通讯能力，以及设备服务、OTA等高级服务，并且针对各种场景提供了丰富的demo代码。相关集成指导请参考[IoT Device SDK（OpenHarmony）使用指南](#)。

4.2.9 IoT Device SDK 使用指南（Python）

IoT Device SDK（Python）提供设备接入华为云IoT物联网平台的Python版本的SDK，提供设备和平台之间通讯能力，以及设备服务、网关服务、OTA等高级服务，并且针对各种场景提供了丰富的demo代码。相关集成指导请参考[IoT Device SDK（Python）使用指南](#)。

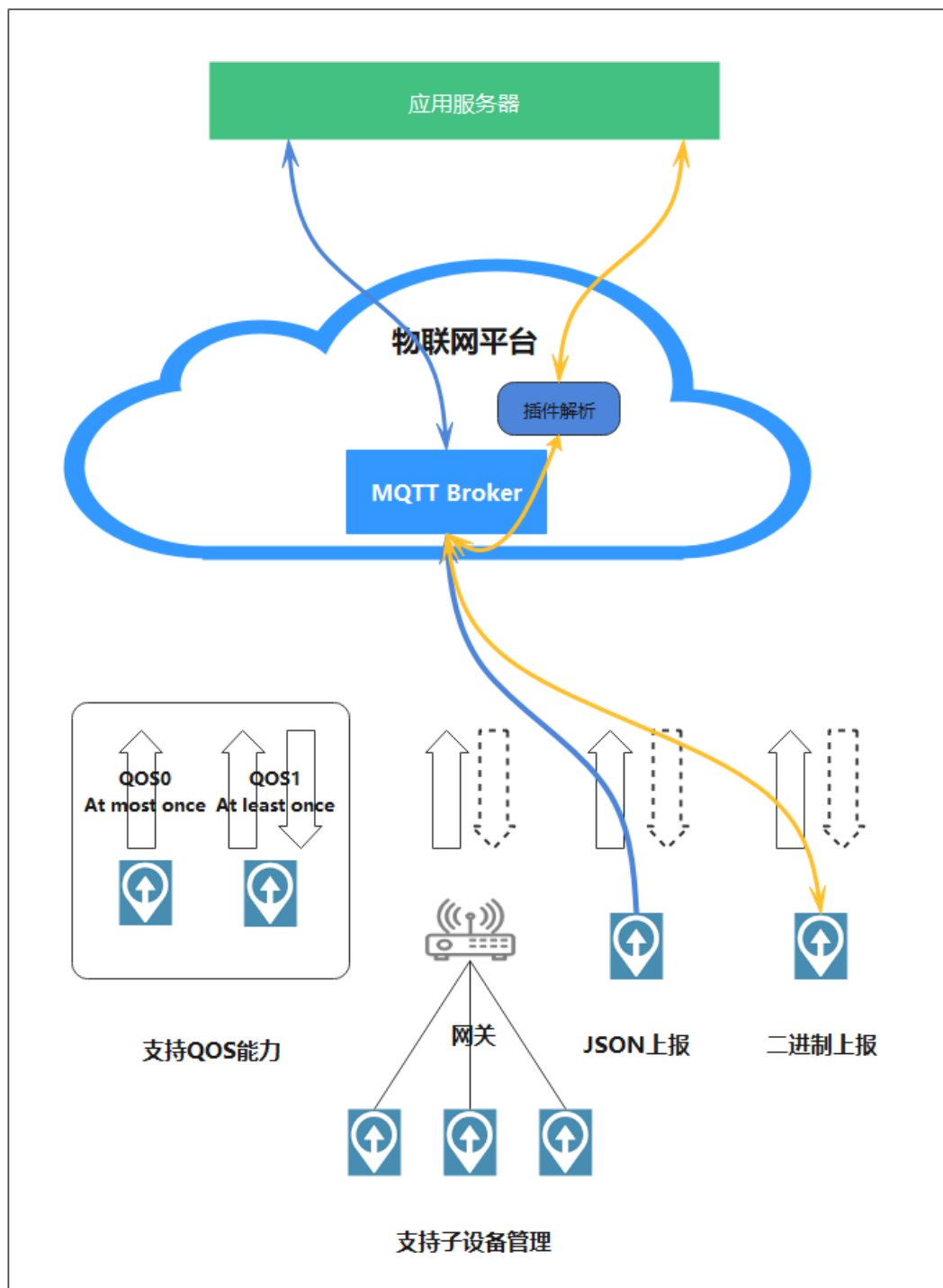
4.3 使用 MQTT Demo 接入

4.3.1 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，适合长连接的场景，如智能路灯等。更多关于MQTT协议语法及接口信息，请访问[这里](#)获取。

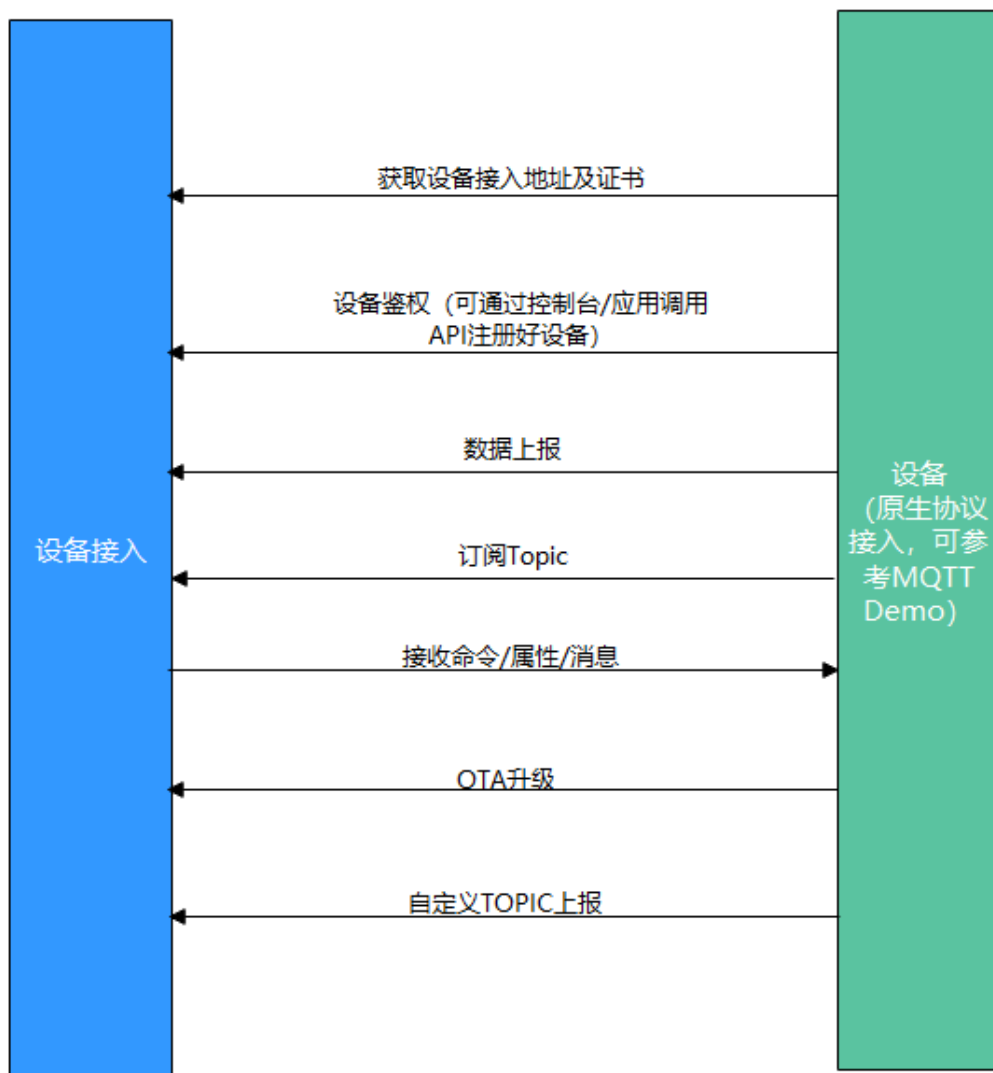
MQTTS是MQTT使用TLS加密的协议。采用MQTTS协议接入平台的设备，设备与物联网平台之间的通信过程，数据都是加密的，具有一定的安全性。



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
支持的MQTT协议版本	支持MQTT v3.1/v3.1.1/v5.0协议版本，不支持协议中的QoS2、will、retain msg。

描述	限制
与标准MQTT协议的区别	<ul style="list-style-type: none"> 支持Qos 0和Qos 1 支持Topic自定义 不支持QoS2 不支持will、retain msg
MQTTS支持的安全等级	采用TCP通道基础 + TLS协议（最高TLSv1.3版本）
单账号每秒最大MQTT连接请求数	无限制
单个设备每分钟支持的最大MQTT连接数	1
单个MQTT连接每秒的吞吐量，即带宽，包含直连设备和网关	3KB/s
MQTT单个发布消息最大长度，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
产品是否支持自定义Topic	支持
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
每个订阅请求的最大订阅数	无限制

📖 说明

平台推荐用户设备与平台通信采用加密通道（接入端口8883），保障设备通信安全。

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

Topic分类	用途	Topic	Public (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。目前支持四个版本的TLS协议，即版本1.0、1.1、1.2和1.3。TLS 1.0和1.1被视为旧版，并计划弃用，强烈建议使用TLS 1.2和TLS 1.3作为首选TLS版本。使用TLS连接时平台仅支持如下加密套件：

- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256

- TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384

4.3.2 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。

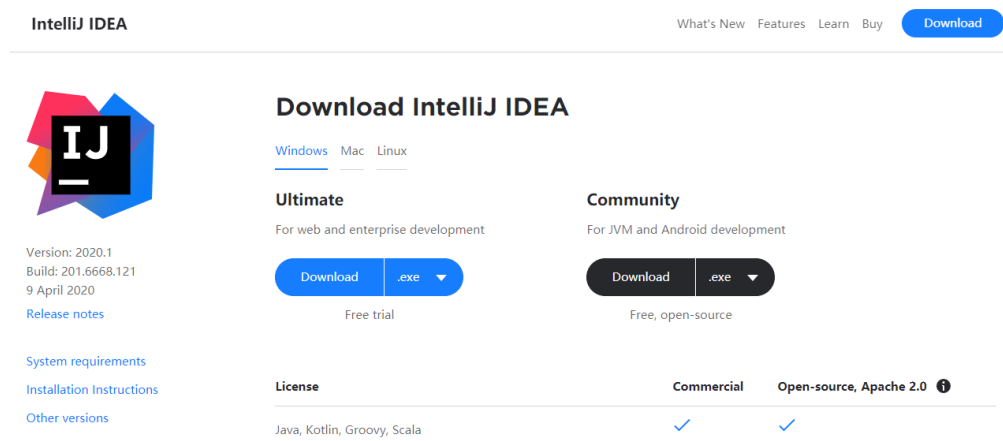
前提条件

- 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）。

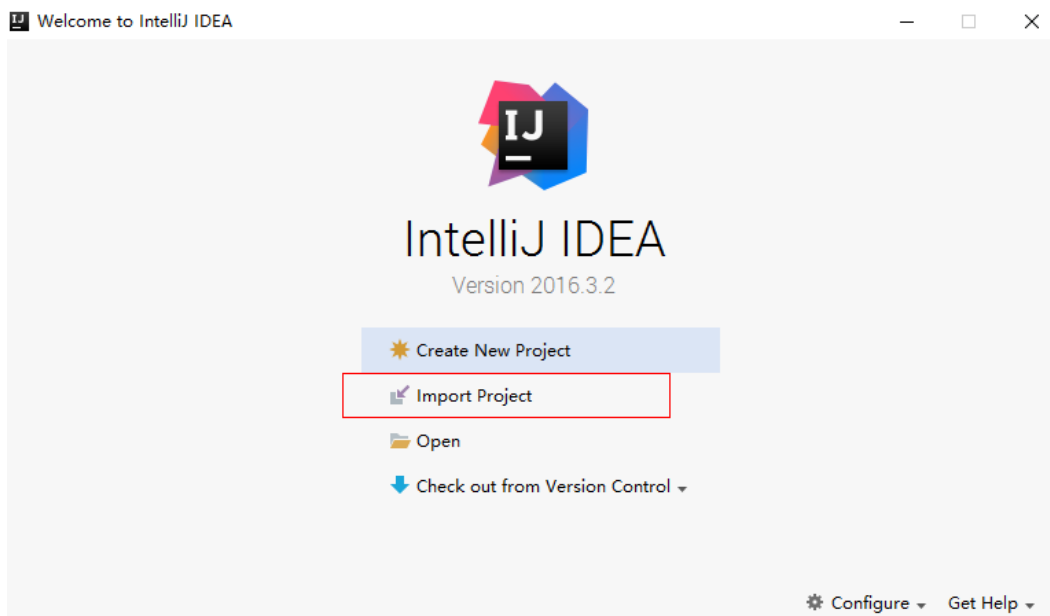


2. 下载完成后，运行安装文件，根据界面提示安装。

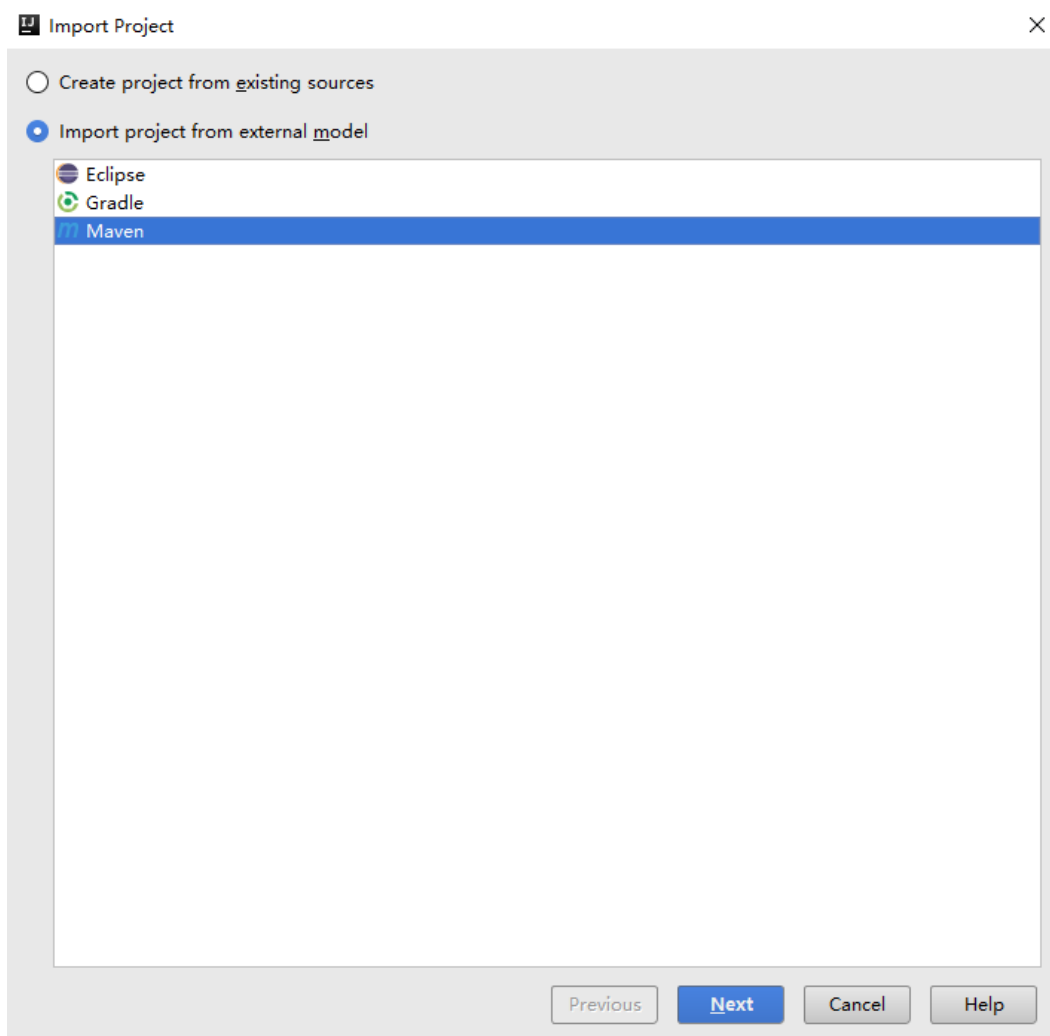
导入代码样例

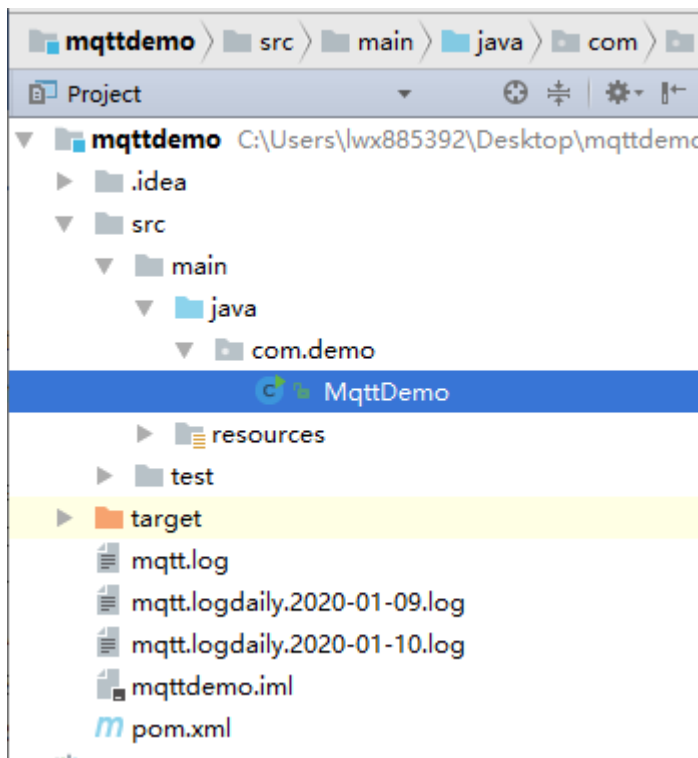
步骤1 下载[JAVA样例](#)。

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("Digi  
CertGlobalRootCA.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。

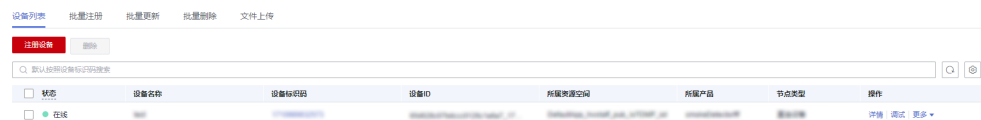
```
client.connect(options, null, new IMqttActionListener())
```

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 4-14 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](#)。





说明

如果在“设备详情”页面没有最新上报数据，请修改产品模型中服务和属性的内容，确保设备上报的服务/属性和产品模型中的服务/属性一致，或者进入“产品 > 模型定义”页面，删除所有服务。

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](#)。

4.3.3 Python Demo 使用说明

概述

本文以Python语言为例，介绍通过MQTT/ 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。

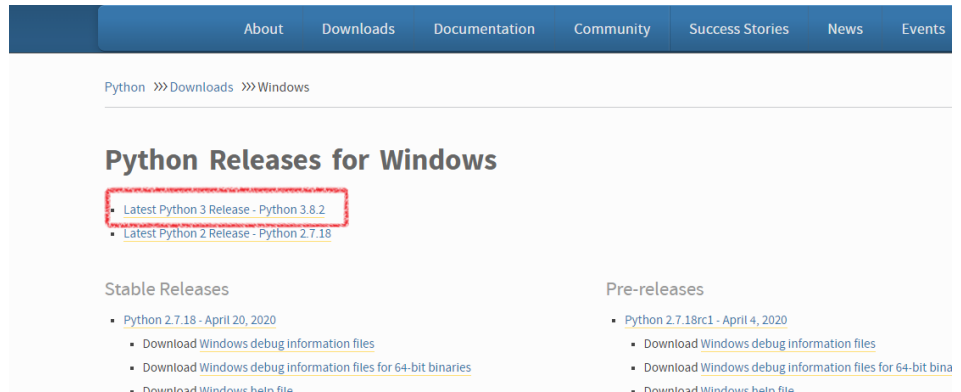
前提条件

- 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](#)。

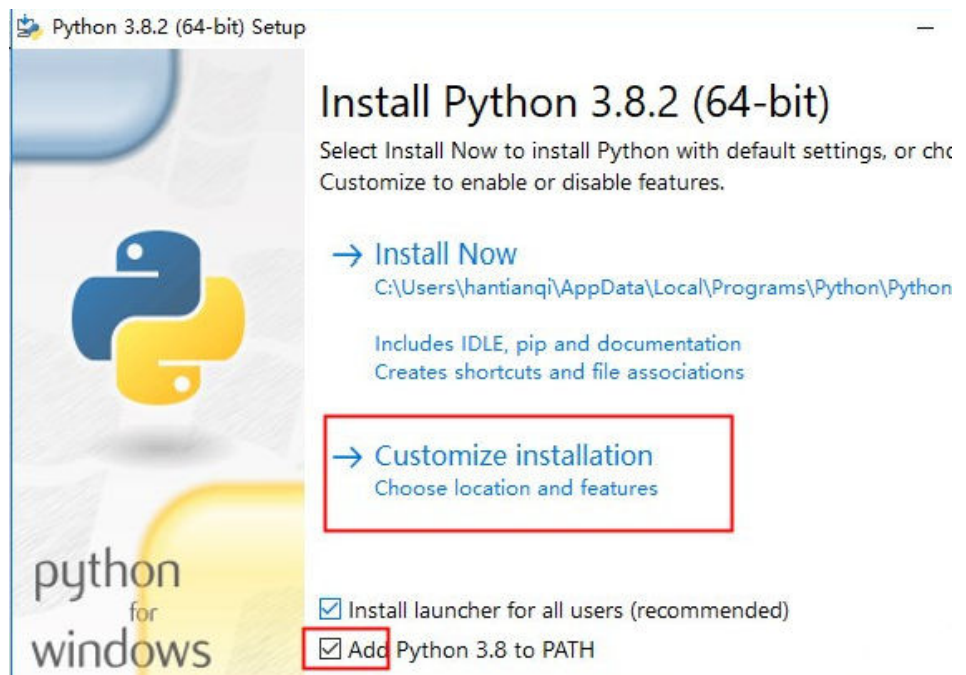
准备工作

- 安装python

- a. 访问[python官网](https://www.python.org/)，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。

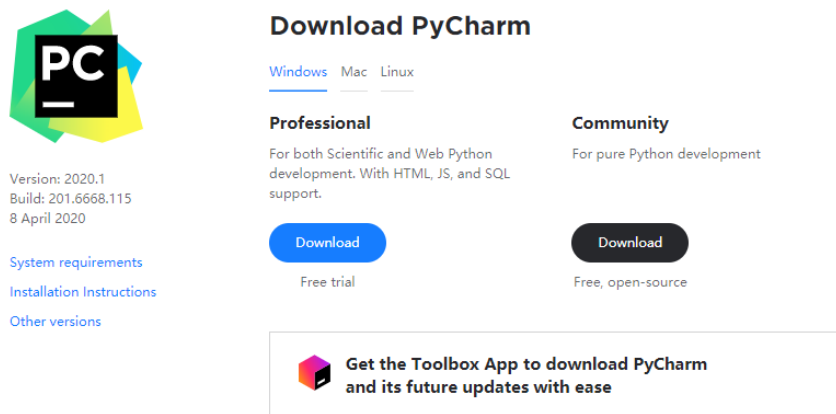


```
选择C:\windows\system32\cmd.exe
Microsoft Windows [版本 10.0.18362.592]
(c) 2019 Microsoft Corporation. 保留所有权利。

C:\Users\>python -V
Python 3.8.2

C:\Users\>
```

- 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。

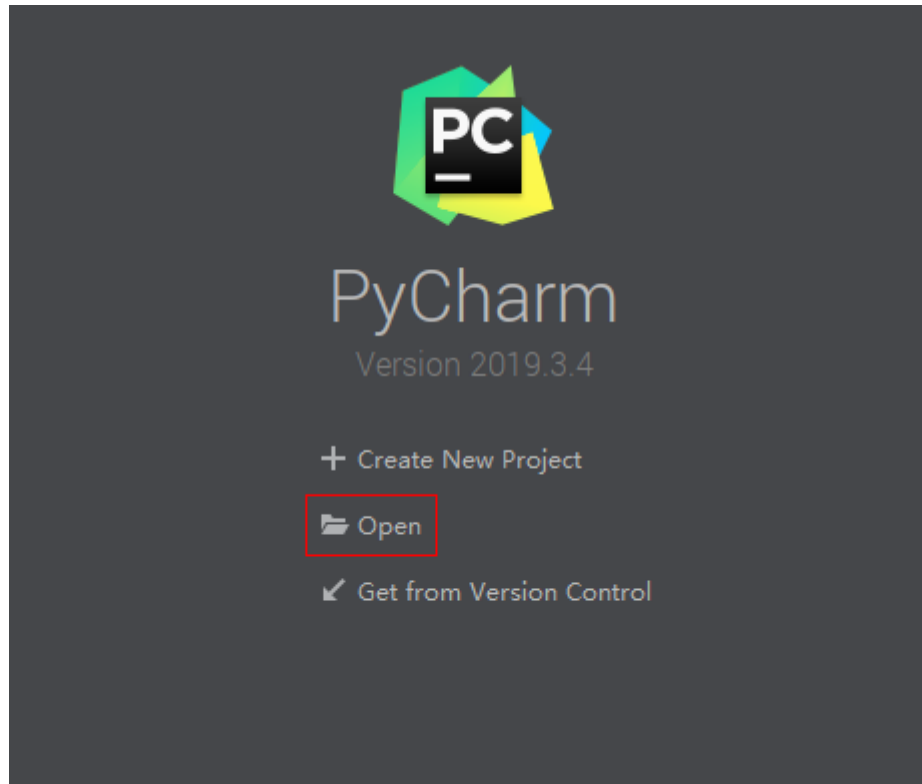


注：推荐使用专业版。

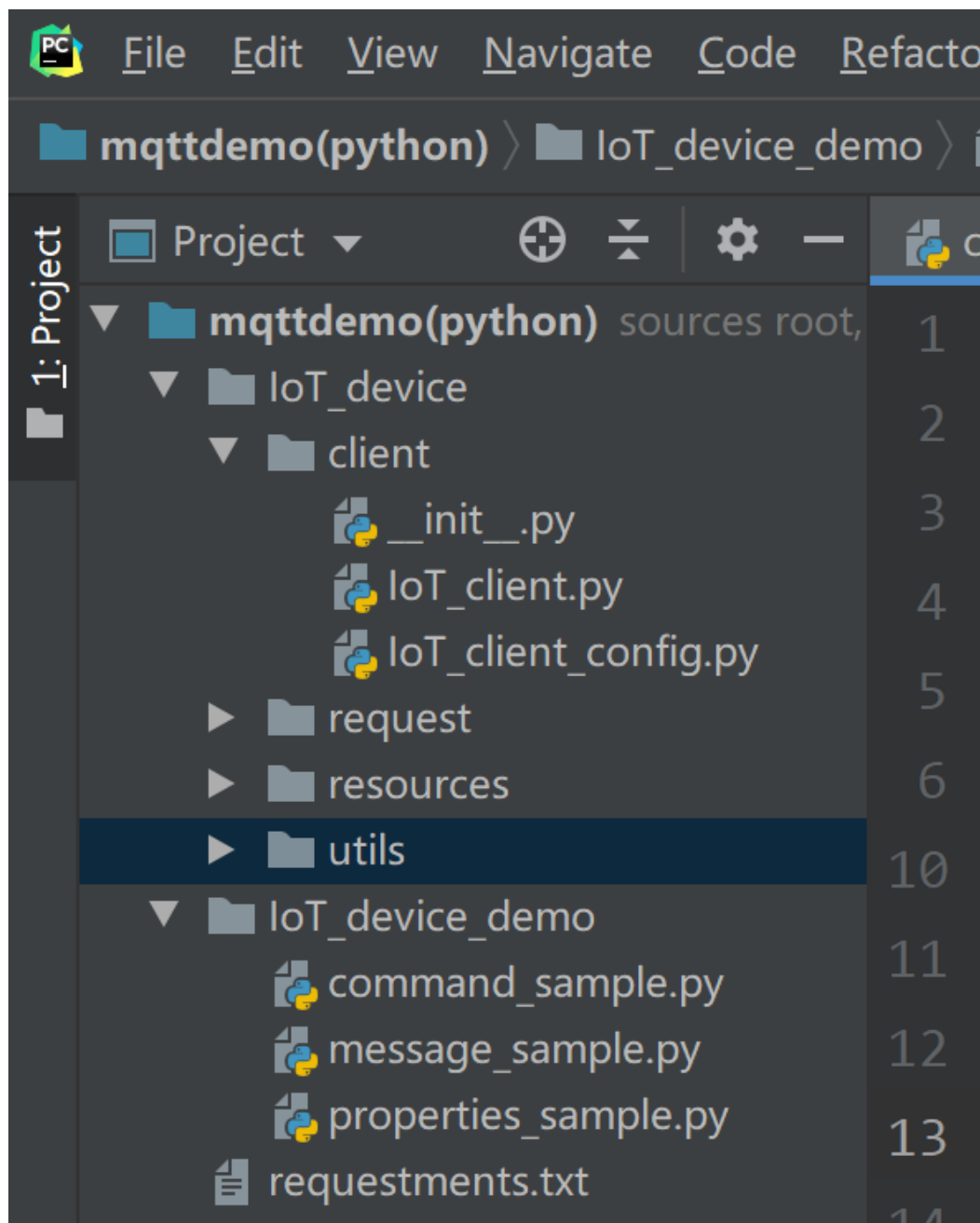
- b. 运行exe文件，按照界面提示安装。

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：

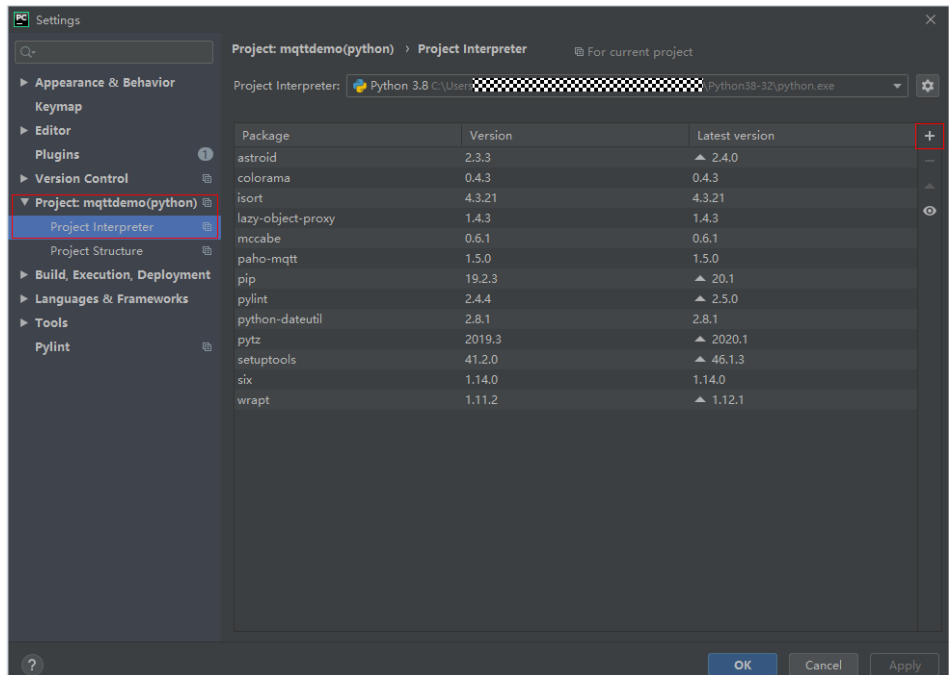
- **方法一**：在命令行下采用pip工具安装（安装python时，已自带该工具）
进入命令行界面输入命令：pip install paho-mqtt回车，提示 Succesfully installed paho-mqtt 表示安装成功。（若提示pip不是内部或外部命令，请检查python环境变量的配置），如下图所示：

```

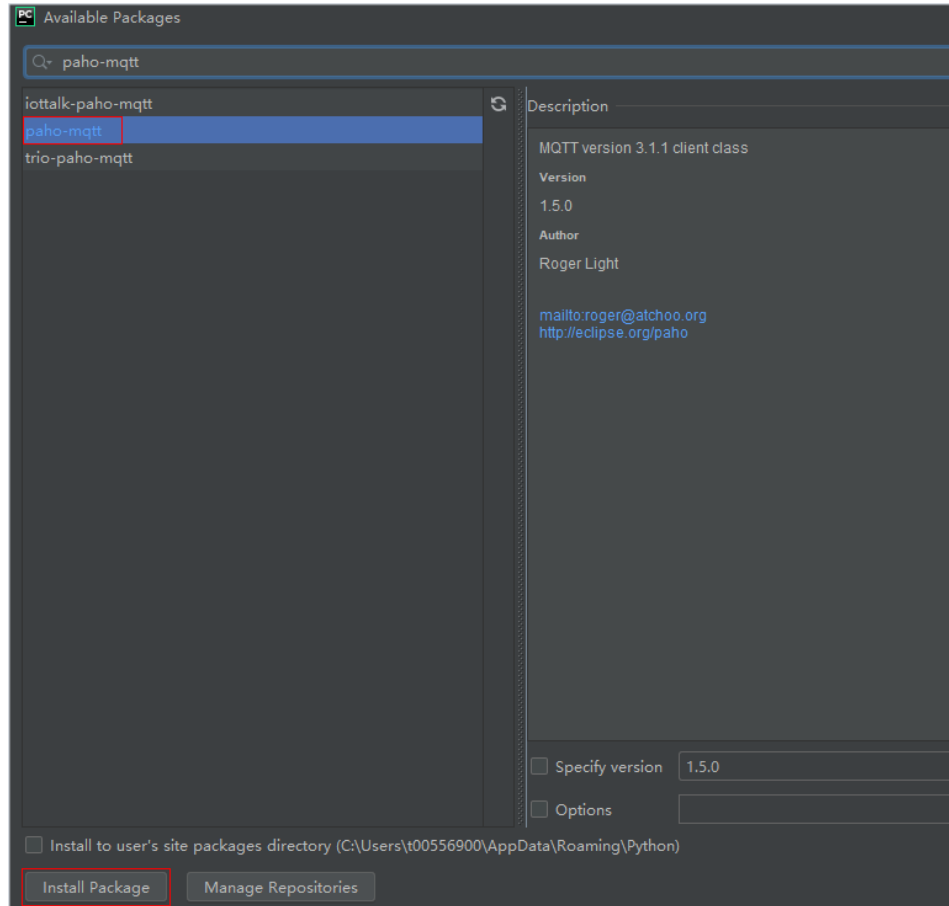
C:\windows\system32\cmd.exe
Microsoft Windows [版本 10.0.18362.592]
(c) 2019 Microsoft Corporation。保留所有权利。

C:\Users\>pip install paho-mqtt
Looking in indexes: http://mirrors.tools.huawei.com/pypi/simple/
Collecting paho-mqtt
  Downloading http://mirrors.tools.huawei.com/pypi/packages/59/11/1dd5c70f0f27a88a3a05772cd95f6087ac479fac66d9c7752ee5e16d9bb0/paho-mqtt-1.5.0.tar.gz (99kB)
    |#####| 102kB 939kB/s
Installing collected packages: paho-mqtt
  Running setup.py install for paho-mqtt ... done
Successfully installed paho-mqtt-1.5.0
    
```

- **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。

如下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_propertis.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。



说明

如果在“设备详情”页面没有最新上报数据，请修改产品模型中服务和属性的内容，确保设备上报的服务/属性和产品模型中的服务/属性一致，或者进入“产品 > 模型定义”页面，删除所有服务。

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](#)。

4.3.4 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

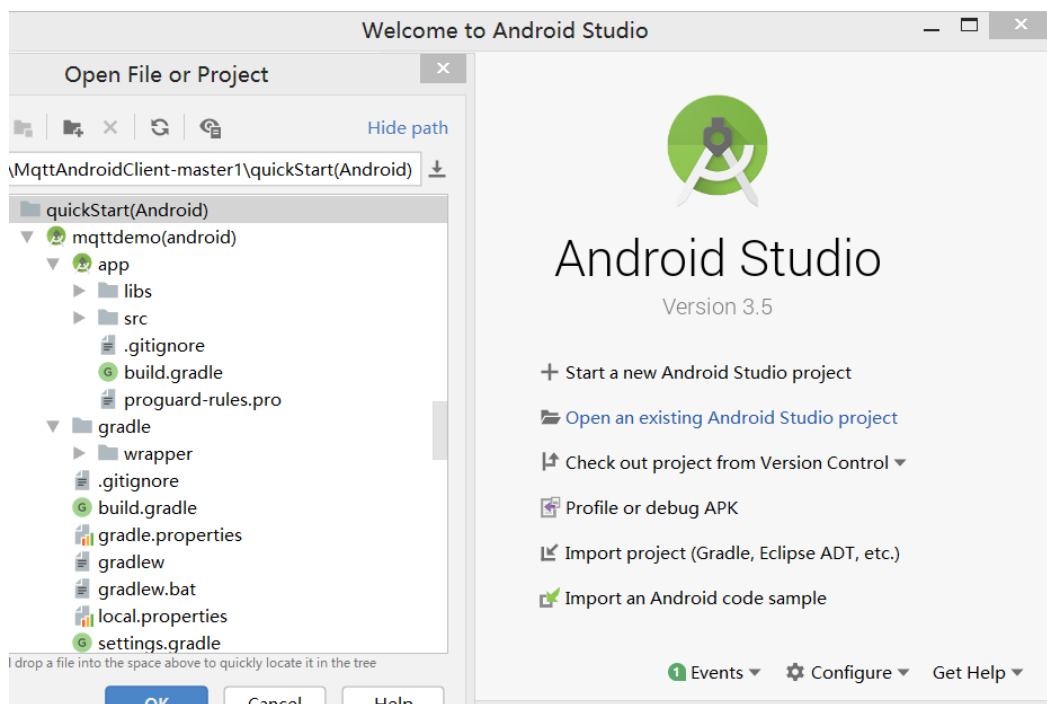
Platform	Android Studio package	Size	SHA-256 checksum
Windows (64-bit)	android-studio-ide-192.6392135-windows.exe Recommended	756 MB	07b6df807fda59e69f05b85ff66bd0c70d09e57fb151197155ef5f119f9e59
	android-studio-ide-192.6392135-windows.zip No .exe installer	770 MB	24f8f9ce467b935c25d89b90cad402d21dd45d4ba9af1ad35baeeb414609e483
Windows (32-bit)	android-studio-ide-192.6392135-windows32.zip No .exe installer	770 MB	7b24742726bbcb840a55dab1f7c0ff923ba384b233c21d35d0e96fa36320d067
Mac (64-bit)	android-studio-ide-192.6392135-mac.dmg	768 MB	c5d0347469be0d995e6b4d74ea72b3a6f2572e72b4eac37a0834b0a0984d9583
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	59023aaabc7d5822fd7b1c5a71589b18e487ca8d7f64320c3547ee0ad390e4ca

- 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。

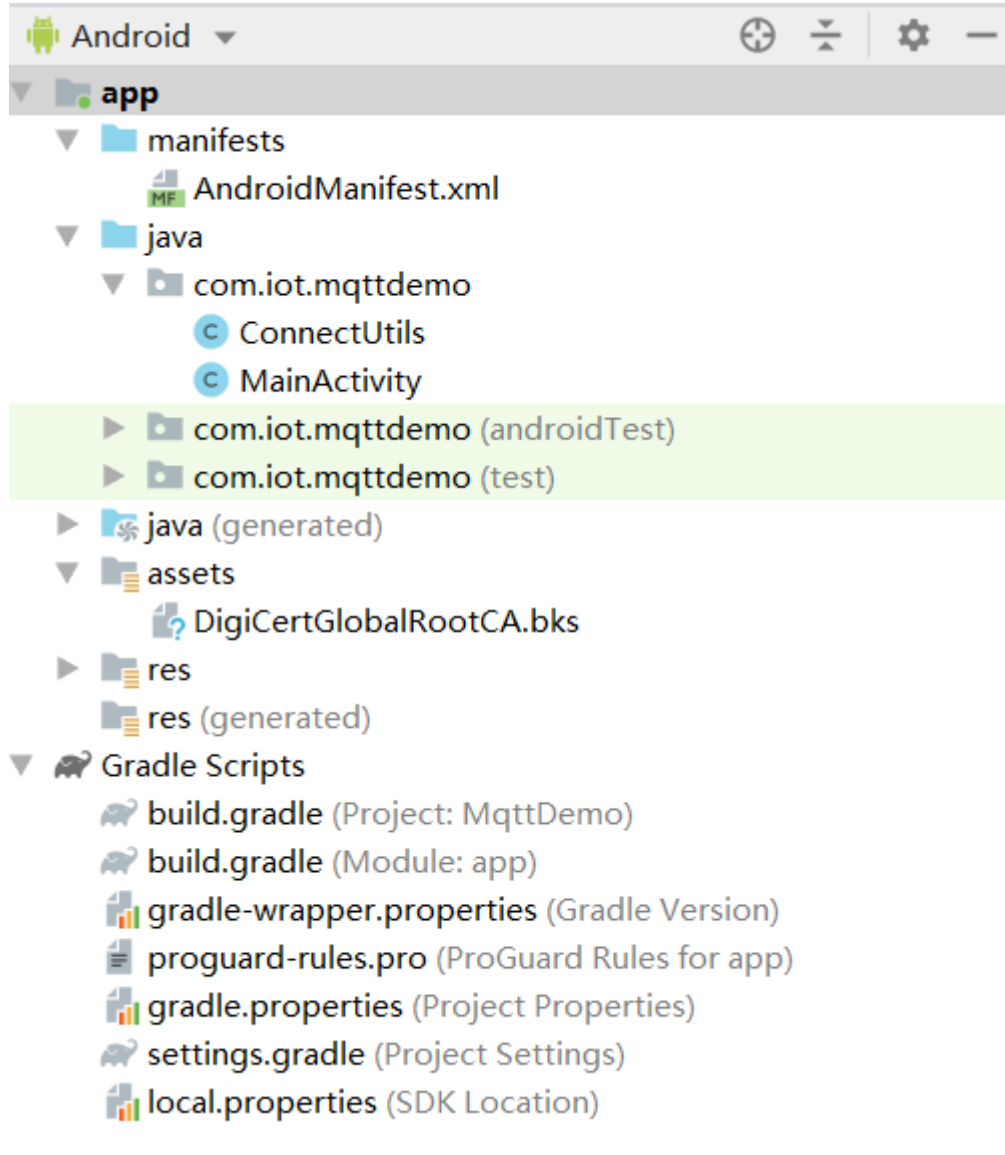
导入代码样例

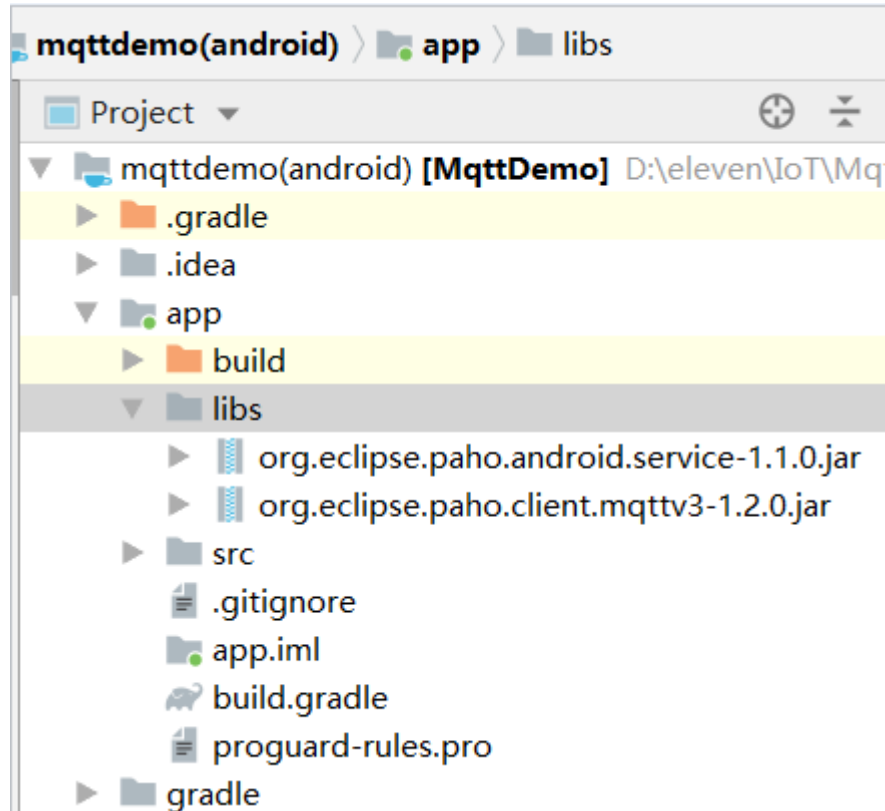
步骤1 下载[quickStart\(Android\)](#)样例。

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

MqttDemo

设备ID

设备密钥

SSL不加密 请下拉框选择Qos

service_id

属性 值

操作日志 (点击一下可清空)

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getMqttsCerificate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() {
    @Override public void onSuccess(IMqttToken asyncActionToken) {
        .....
        subscribeToTopic();
    }
}
```

建链成功后，APP界面日志栏显示如下信息：

MqttDemo

设备ID

设备密钥

SSL加密 请下拉框选择Qos

service_id

属性 值

操作日志 (点击一下可清空)

```
serverUrl:ssl://iot-mqtts.cn-north-4.myhuaweicloud.com:8883,
clientId:f1af[redacted]_0_0_2020043010
开始订阅命令下发的Topic: $oc/devices/f1af[redacted]/sys/commands/#
MQTT连接成功:
ssl://iot-mqtts.cn-north-4.myhuaweicloud.com:8883
订阅Topic成功
```

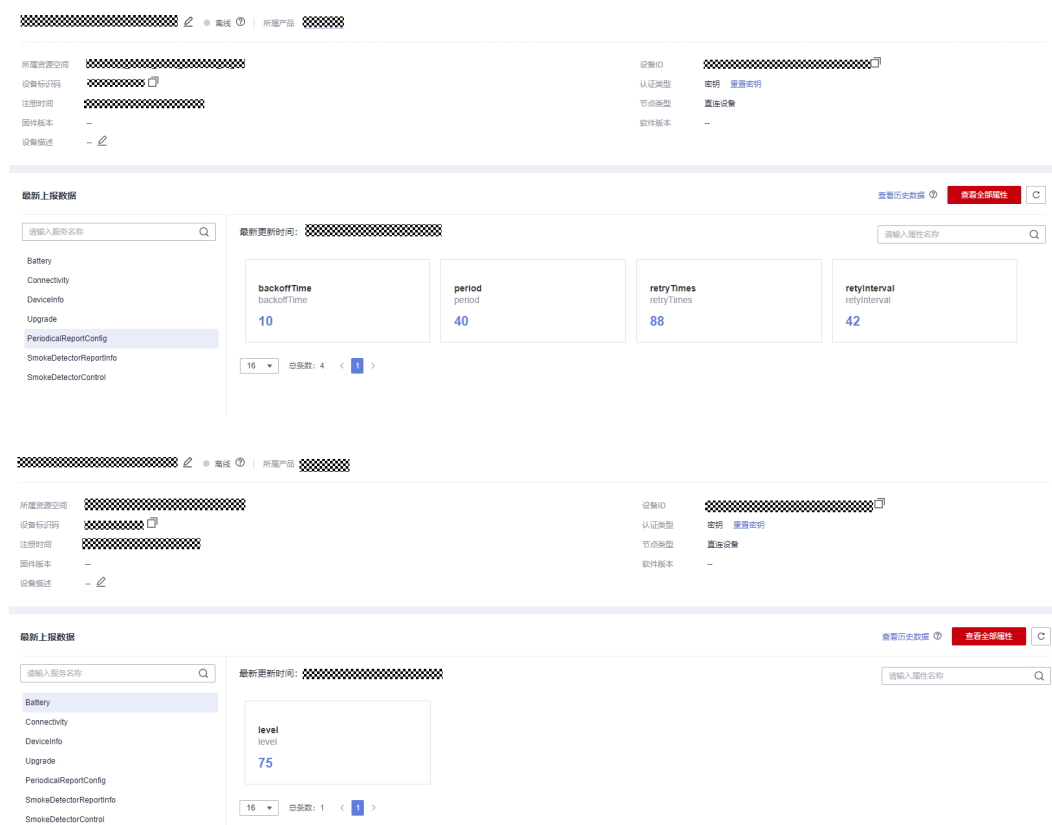
属性上报

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_deviceld.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



说明

如果在“设备详情”页面没有最新上报数据，请修改产品模型中服务和属性的内容，确保设备上报的服务/属性和产品模型中的服务/属性一致，或者进入“产品 > 模型定义”页面，删除所有服务。

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 {
    .....
    @Override public 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：

MqttDemo

设备ID af659e-aa94-4bc9-99f6-415c3c0c82e3

设备密钥 123456789

SSL加密 请下拉框选择Qos

MQTT建链

service_id Battery

属性 level 值 75

属性上报

操作日志 (点击一下可清空)

```
属性上报内容:{"services":  
[{"service_id":"Battery","properties":{"level":"75"}}]}  
属性上报topic:$oc/devices/  
f1af6[REDACTED]/sys/  
properties/report  
MQTT推送消息: {"services":  
[{"service_id":"Battery","properties":{"level":"75"}}]}  
MQTT属性上报成功  
MQTT接收下发命令成功: {"paras":{"command":  
5},"service_id":null,"command_name":null}
```

4.3.5 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
```

```
[root@server-1908071538 test]# cd openssl-1.1.1d
[root@server-1908071538 openssl-1.1.1d]# ./config 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”，请将这些文件拷贝到demo的lib文件夹下（同时将/home/test/openssl/include/openssl里的内容拷贝到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

- 访问github下载地址：<https://github.com/eclipse/paho.mqtt.c>，下载paho.mqtt.c源码。
- 解压后上传到linux编译机。
- 修改makefile

- 通过如下命令进行编辑Makefile

```
vim Makefile
```

- 查找字符串

```
/DOXYGEN_COMMAND =
```

- 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
```

```
LDLFLAGS += -L/home/test/openssl/lib -lrt
```

```
127 INSTALL_PROGRAM = $(INSTALL)
128 INSTALL_DATA = $(INSTALL) -m 644
129 DOXYGEN_COMMAND = doxygen
130 CFLAGS += -I/home/test/openssl/include
131 LDLFLAGS += -L/home/test/openssl/lib -lrt
132
133 MAJOR_VERSION = 1
134 MINOR_VERSION = 0
135 VERSION = ${MAJOR_VERSION}.${MINOR_VERSION}
```

- 把如下图的CCDLFLAGS_SO、LDLFLAGS_CS、LDLFLAGS_AS、FLAGS_EXES的openssl地址都改成对应的地址


```

194
195 CFLAGS_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。

/home/test/paho.mqtt.c/build/output			
名字	扩展	大小	已改变
↑			2019/10/23 15:34:01
samples			2019/10/23 15:34:14
test			2019/10/23 15:34:16
libpaho-mqtt3a.so		19 B	2019/10/23 15:34:10
libpaho-mqtt3a.so.1		21 B	2019/10/23 15:34:10
libpaho-mqtt3a.so.1.0		477 KiB	2019/10/23 15:34:10
libpaho-mqtt3as.so		20 B	2019/10/23 15:34:13
libpaho-mqtt3as.so.1		22 B	2019/10/23 15:34:13
libpaho-mqtt3as.so.1.0		529 KiB	2019/10/23 15:34:13
libpaho-mqtt3c.so		19 B	2019/10/23 15:34:03
libpaho-mqtt3c.so.1		21 B	2019/10/23 15:34:03
libpaho-mqtt3c.so.1.0		446 KiB	2019/10/23 15:34:03
libpaho-mqtt3cs.so		20 B	2019/10/23 15:34:07
libpaho-mqtt3cs.so.1		22 B	2019/10/23 15:34:07
libpaho-mqtt3cs.so.1.0		498 KiB	2019/10/23 15:34:07
paho_c_version		13,768 B	2019/10/23 15:34:14

f. 拷贝paho库文件。

当前SDK仅用到了libpaho-mqtt3as，请将“libpaho-mqtt3as.so”和“libpaho-mqtt3as.so.1”文件拷贝到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拷贝到demo的include/base文件夹下）。

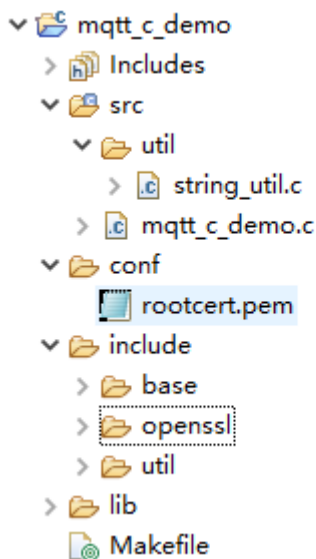
注意

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 将代码拷贝到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文件内容拷贝到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 = "5ebac693352cfb02c567ec88_test2345"; //deviceId  
char *password = "602d6cc77d87271be8f462f52d27d818";
```

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

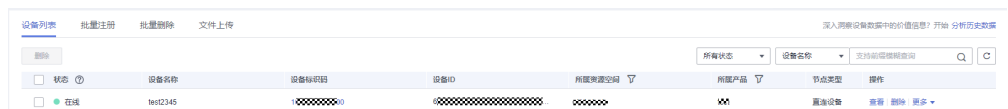
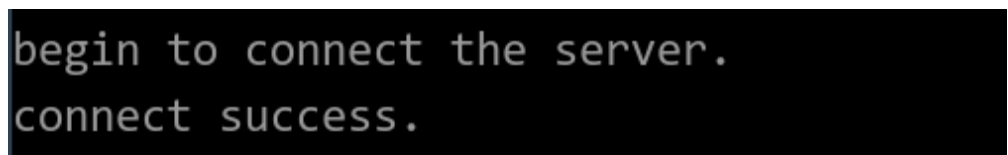
置为120秒，用户如果想修改，可以修改代码中的“keepAliveInterval”参数，心跳时间范围具体可参考[使用限制](#)。

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。



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\",\"lmbA_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 4-15 设备详情



说明

如果在“设备详情”页面没有最新上报数据，请修改产品模型中服务和属性的内容，确保设备上报的服务/属性和产品模型中的服务/属性一致，或者进入“产品 > 模型定义”页面，删除所有服务。

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=b5fb4352-43cb-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](#)。

4.3.6 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。

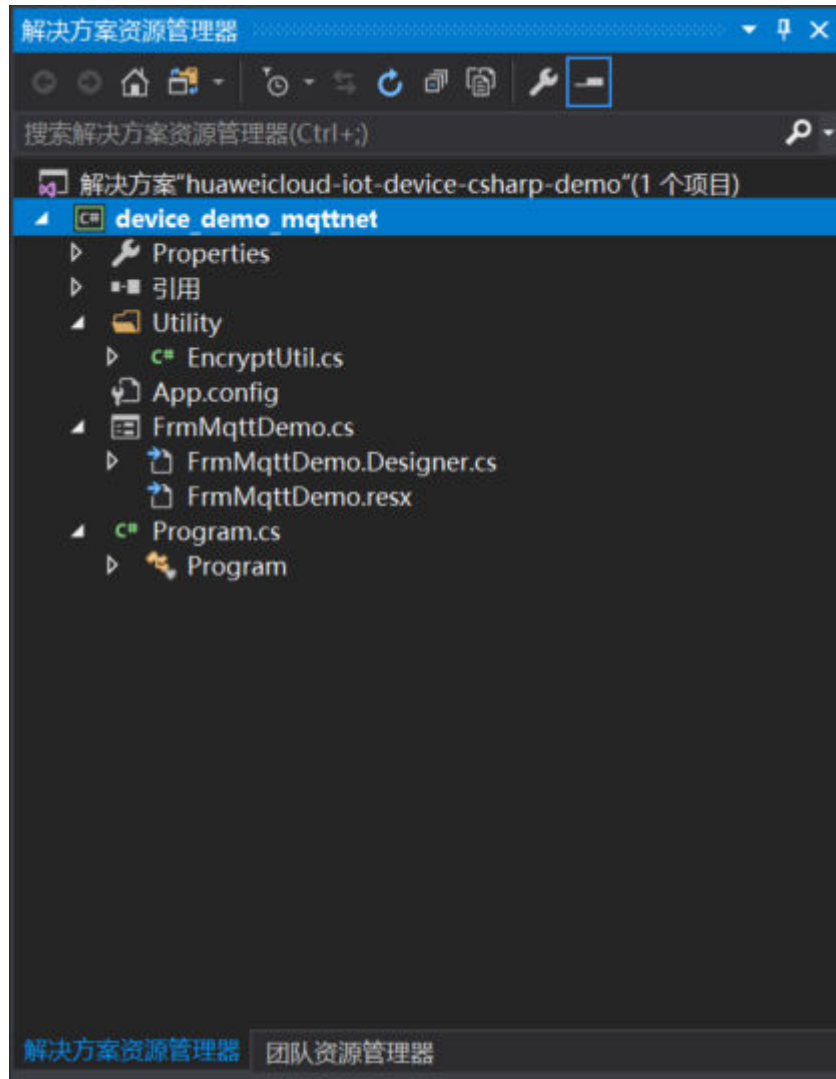
导入代码样例

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-mqtt.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();
    }));
}

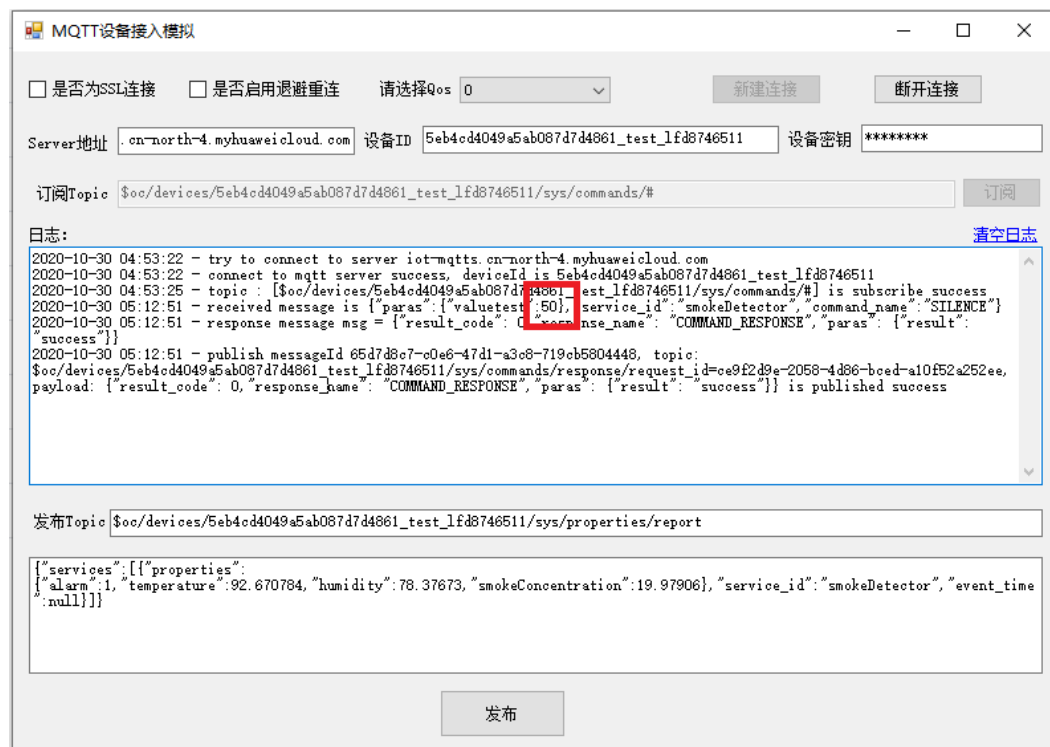
```

例如下发参数名为SmokeDetectorControl: SILENCE，参数值为50的命令。

图 4-16 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。

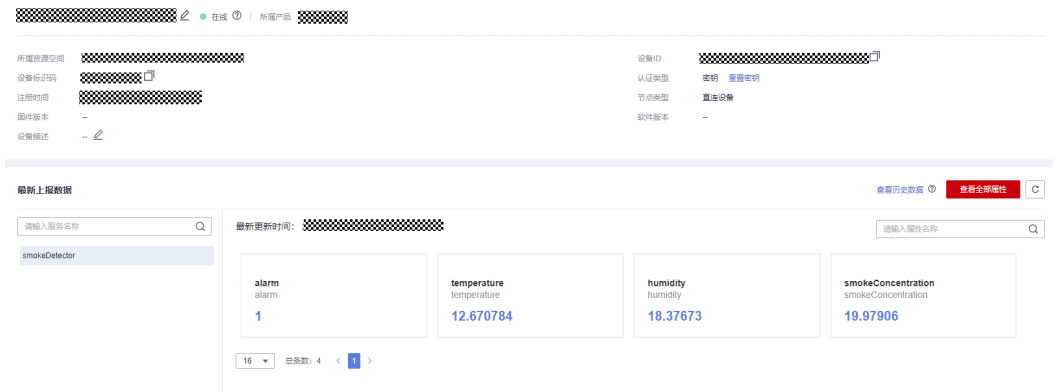
```
var appMsg = new MqttApplicationMessage();
appMsg.Payload = Encoding.UTF8.GetBytes(inputString);
appMsg.Topic = topic;
appMsg.QualityOfServiceLevel = int.Parse(cbOosSelect.Selected.Value.ToString()) == 0 ?
MqttQualityOfServiceLevel.AtMostOnce : MqttQualityOfServiceLevel.AtLeastOnce;
appMsg.Retain = 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：



 说明

如果在“设备详情”页面没有最新上报数据，请修改产品模型中服务和属性的内容，确保设备上报的服务/属性和产品模型中的服务/属性一致，或者进入“产品 > 模型定义”页面，删除所有服务。

 说明

由于是同步命令需要端侧回复响应可[参考接口](#)。

4.3.7 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）。

Downloads

Latest LTS Version: 12.18.0 (includes npm 6.14.4)

Download the Node.js source code or a pre-built installer for your platform, and start developing today.

LTS Recommended For Most Users		Current Latest Features	
 Windows Installer node-v12.18.0-x64.msi	 macOS Installer node-v12.18.0.pkg	 Source Code node-v12.18.0.tar.gz	

- Windows Installer (.msi)
- Windows Binary (.zip)
- macOS Installer (.pkg)
- macOS Binary (.tar.gz)
- Linux Binaries (x64)
- Linux Binaries (ARM)
- Source Code

32-bit	64-bit
32-bit	64-bit
64-bit	
64-bit	
64-bit	
ARMv7	ARMv8
node-v12.18.0.tar.gz	

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。

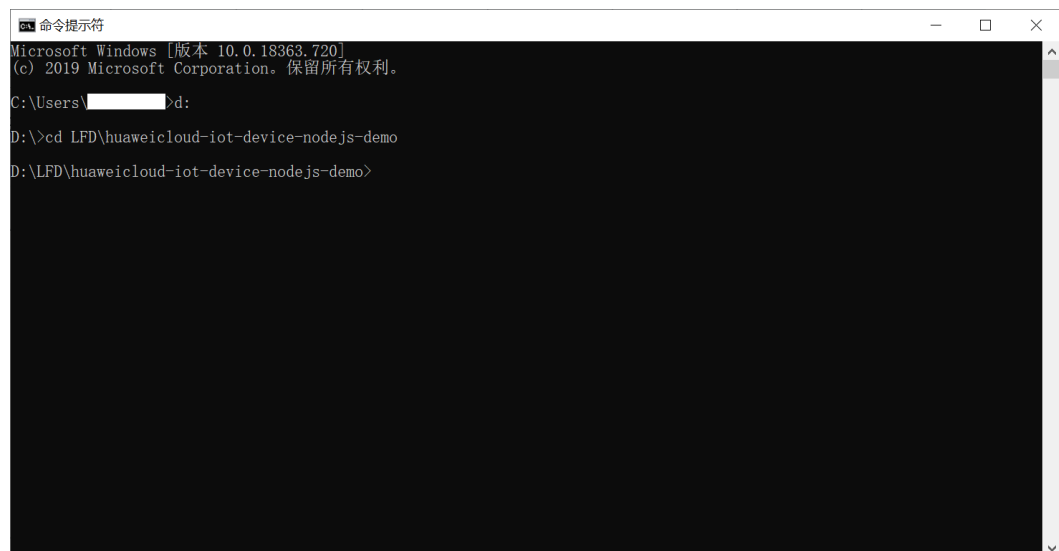
步骤2 运行Win键 + r -->输入 cmd-->回车，进入命令行窗口，安装全局模块，执行如下命令：

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
var serverUrl = "*****";//请填写设备所在平台的接入地址
var deviceId = "*****";//请填写在平台注册的设备ID
var secret = "*****";//请填写在平台注册的设备密钥
```

```
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: 'mqtt',
  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: 'mqtt'
  //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);

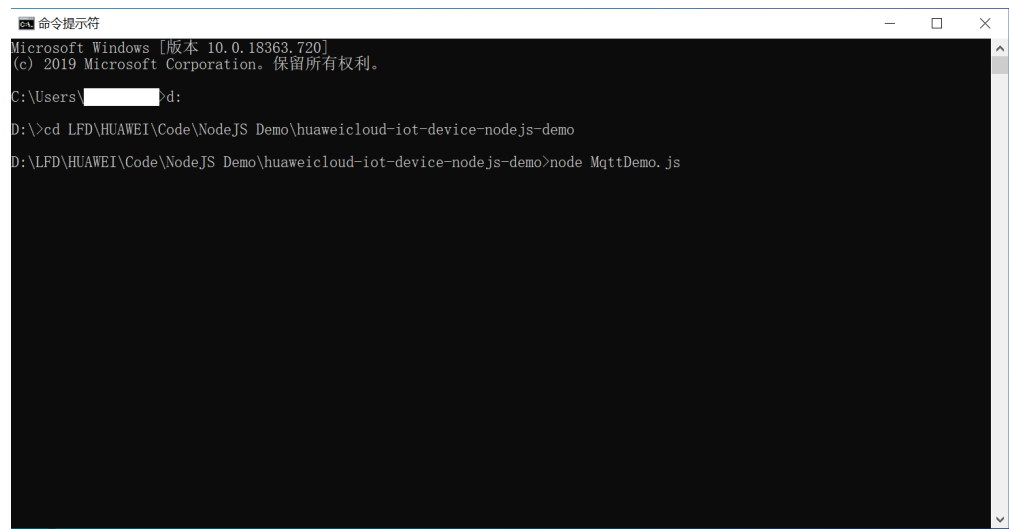
client.on('connect', function () {
  log("connect to mqtt server success, deviceId is " + deviceId);
  //订阅Topic
  subscribeTopic();
  //发布消息
  publishMessage();
})

//命令下发响应
client.on('message', function (topic, message) {
  log('received message is ' + message.toString());

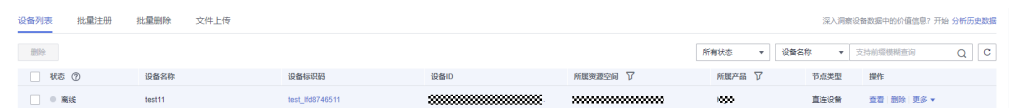
  var jsonMsg = responseReq;
  client.publish(getResponseTopic(topic.toString().split("=")[1]), jsonMsg);
```

```
log('responded 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。



启动程序后，设备状态变为在线



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':'smokeDetector','event_time':null}]};

```

命令行主界面如下：

```

Microsoft Windows [版本 10.0.18363.720]
(c) 2019 Microsoft Corporation. 保留所有权利。

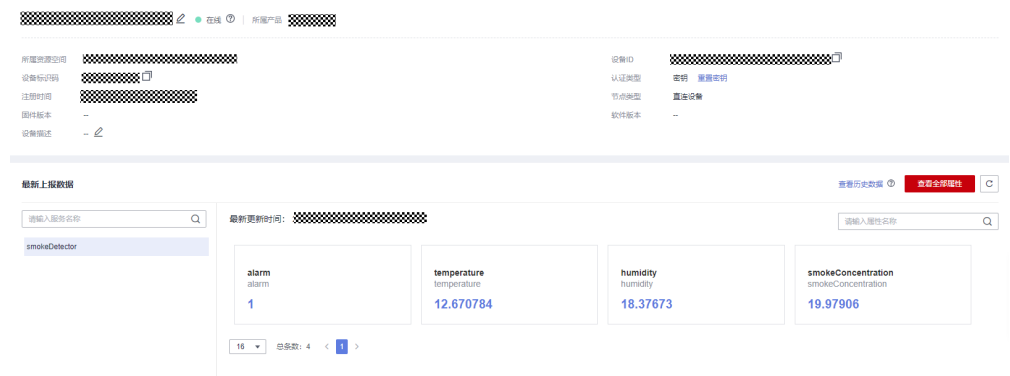
C:\Users\>cd:

D:\>cd LFD\HUAWEI\Code\NodeJS Demo\huaweicloud-iot-device-nodejs-demo

D:\LFD\HUAWEI\Code\NodeJS Demo\huaweicloud-iot-device-nodejs-demo>node MqttDemo.js
2020-06-12 11:47:15 - connect to mqtt server success, deviceId is 5eb4cd4049a5ab087d7d4861_test_lfd8746511
2020-06-12 11:47:15 - publish message topic is $oc/devices/5eb4cd4049a5ab087d7d4861_test_lfd8746511/sys/properties/report
2020-06-12 11:47:15 - publish message is {"services":[{"properties":{"alarm":1,"temperature":12.670784,"humidity":18.37673,"smokeConcentration":19.97906},"service_id":"smokeDetector","event_time":null}]}
2020-06-12 11:47:15 - publish message successful
2020-06-12 11:47:15 - topic : $oc/devices/5eb4cd4049a5ab087d7d4861_test_lfd8746511/sys/commands/# is subscribed success

```

属性上报成功，平台界面如下：



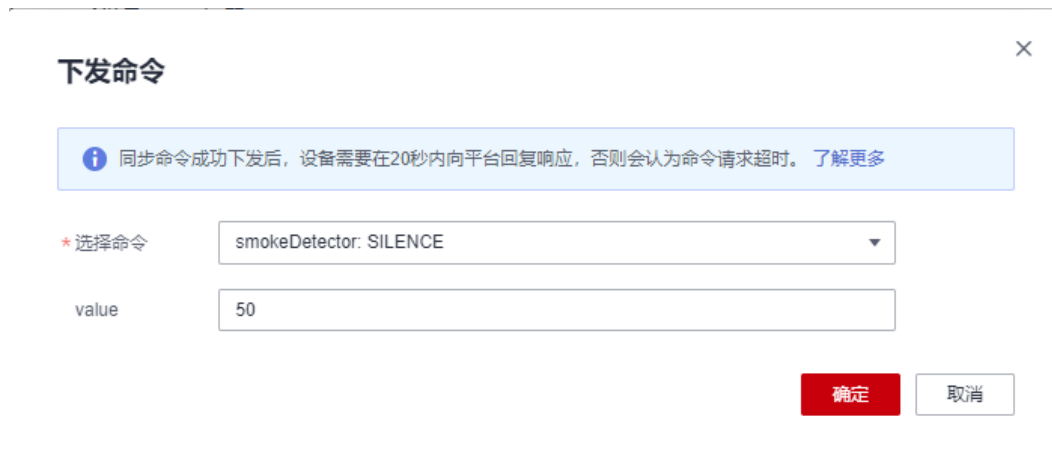
📖 说明

如果在“设备详情”页面没有最新上报数据，请修改产品模型中服务和属性的内容，确保设备上报的服务/属性和产品模型中的服务/属性一致，或者进入“产品 > 模型定义”页面，删除所有服务。

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。



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：

```
选择命令提示符 - node MqttDemo.js
Microsoft Windows [版本 10.0.18363.720]
(c) 2019 Microsoft Corporation. 保留所有权利。

C:\Users\>
D:\>cd LFD\HUAWEI\Code\NodeJS Demo\huaweicloud-iot-device-nodejs-demo
D:\LFD\HUAWEI\Code\NodeJS Demo\huaweicloud-iot-device-nodejs-demo>node MqttDemo.js
2020-06-12 11:56:18 - connect to mqtt server success, deviceId is 5eb4cd4049a5ab087d7d4861_test_lfd8746511
2020-06-12 11:56:18 - publish message topic is $oc/devices/5eb4cd4049a5ab087d7d4861_test_lfd8746511/sys/properties/report
2020-06-12 11:56:18 - publish message is [{"services":[{"properties":{"alarm":1,"temperature":12.670784,"humidity":18.37673,"smokeConcentration":19.97906},"service_id":"smokeDetector","event_time":null}]}]
2020-06-12 11:56:18 - publish message successful
2020-06-12 11:56:18 - topic : $oc/devices/5eb4cd4049a5ab087d7d4861_test_lfd8746511/sys/commands/# is subscribed success
2020-06-12 11:56:28 - received message is {"paras":{"value":50,"service_id":"smokeDetector","command_name":"SILENCE"}}
2020-06-12 11:56:28 - responded message is {"result_code":0,"response_name":"COMMAND_RESPONSE","paras":{"result":"success"}}
```

说明

由于是同步命令需要端侧回复响应可[参考接口](#)。

4.4 OTA 升级设备侧适配

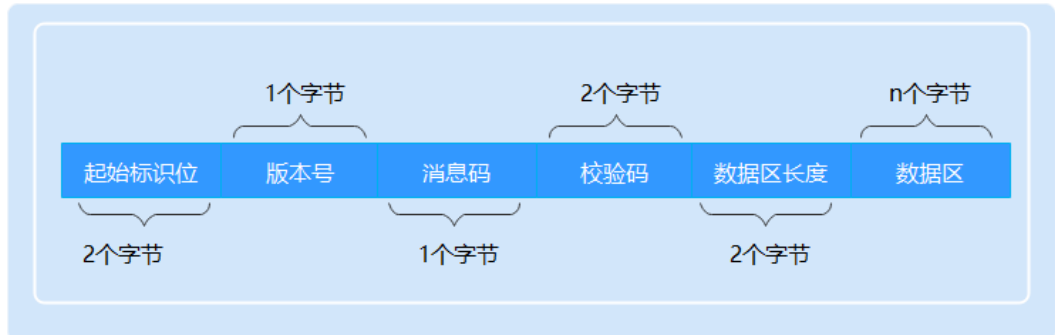
4.4.1 设备侧适配开发指导

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：



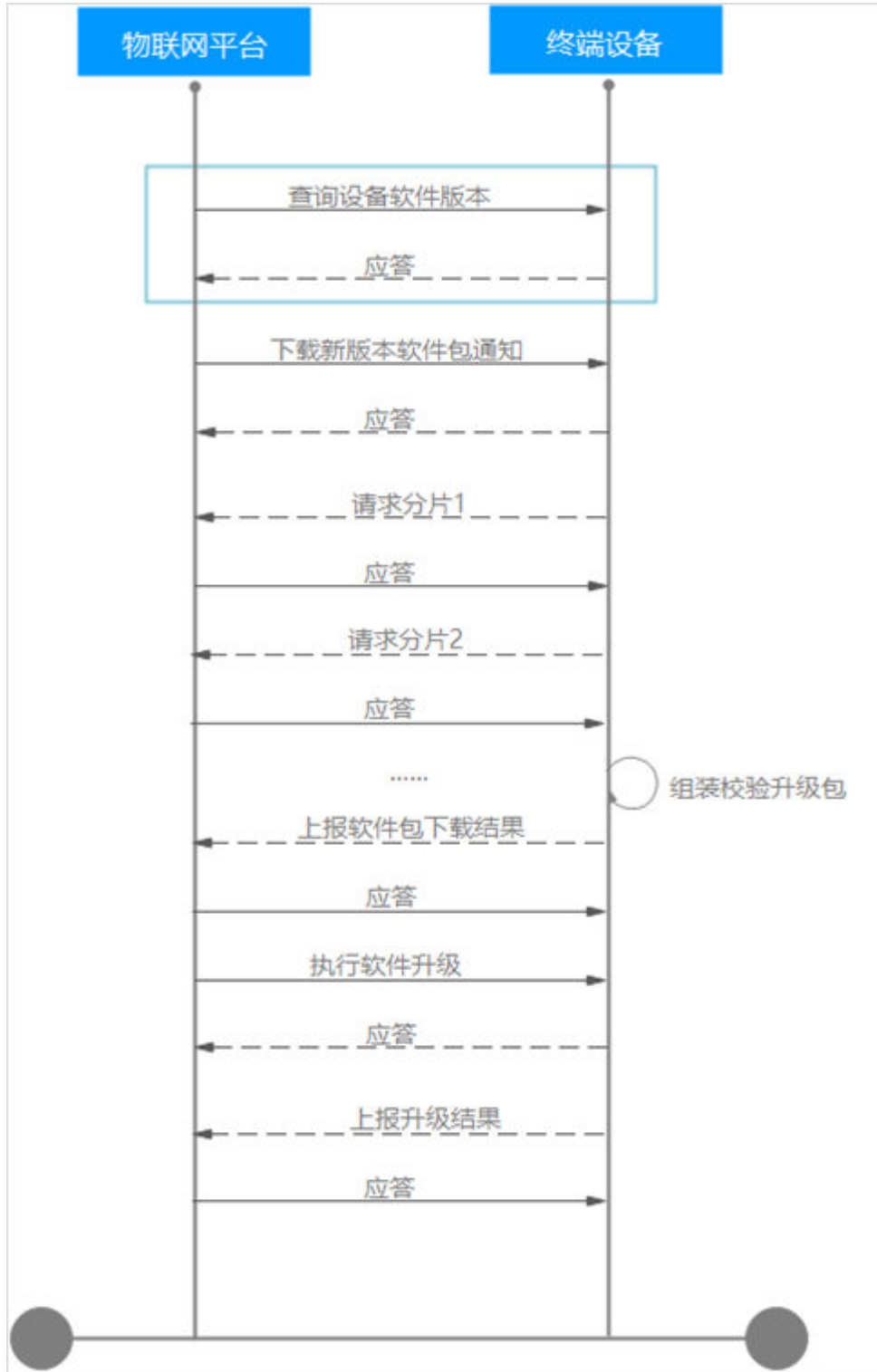
PCP协议消息由：起始标识位、版本号、消息码、校验码、数据区长度和数据区组成，各字段的的要求和描述如下表所示。

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。
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： <ul style="list-style-type: none"> ● 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。

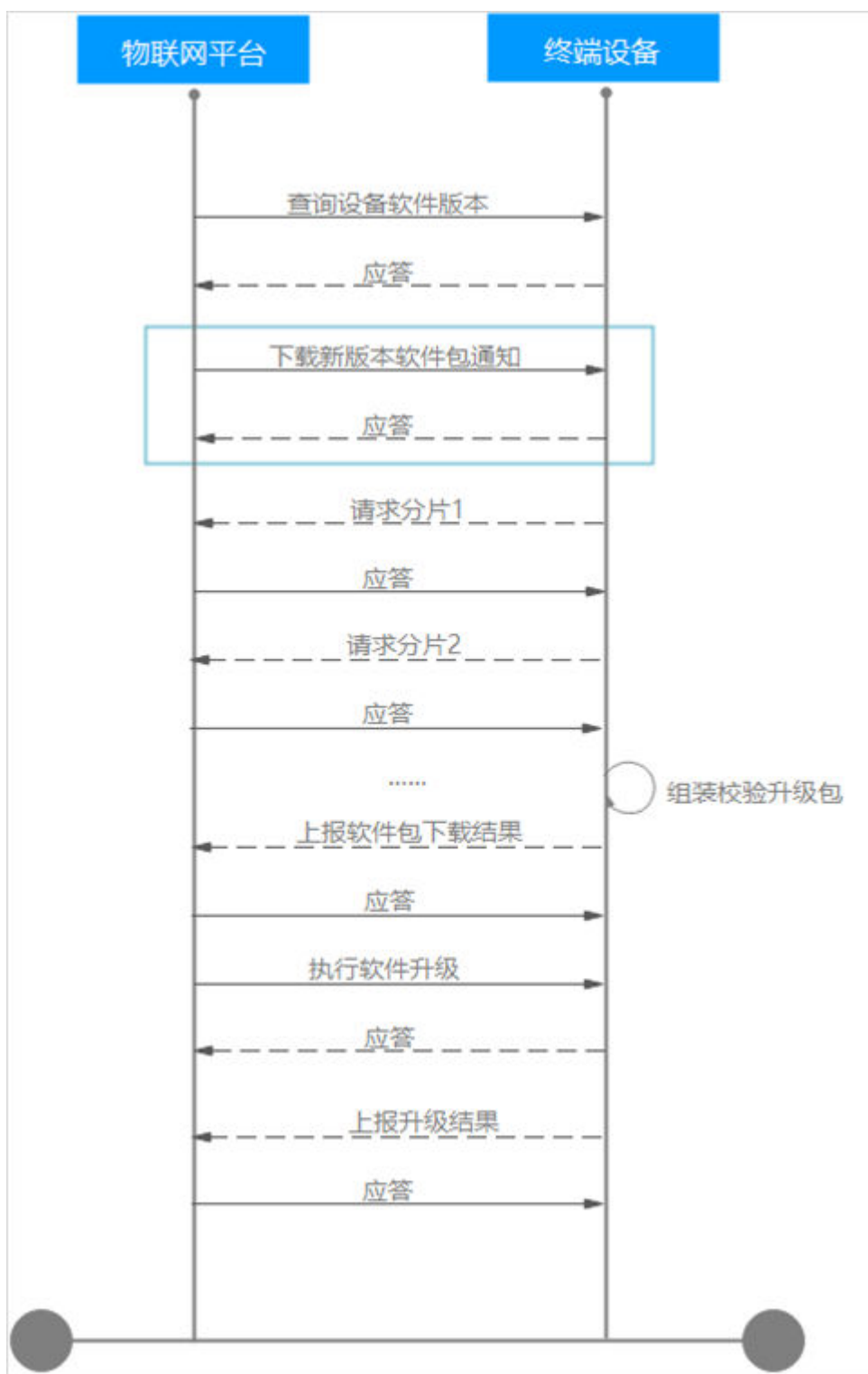
- 起始标识固定为：FFFE。
- 版本号固定为：01。
- 消息码：与请求的消息码一致，为13。
- 校验码：CRC16计算前先用0000替代。
- 数据区长度：根据数据区的字段的数据类型得出数据区长度为17个字节，转换为十六进制为：0011。
- 数据区：根据数据区的定义可知，处理成功的结果码为00，版本号信息假设为V0.9，将V0.9进行ASCII转码得到56302E39，由于版本号的数据类型为BYTE[16]，即16个字节，当前只有4个字节，因此需要在版本号数据后面补0，得到56302E39000000000000000000000000。因此，数据区合并后为0056302E39000000000000000000000000。

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 0056302E390000000000000000000000000000。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FFFE01138DE300110056302E39000000000000000000000000000000。

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：
56312E30000000000000000000000000。
 - 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 56312E30000000000000000000000000 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 FFFE011402F7001656312E3000000000000000000000000000000001F400010000。

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。

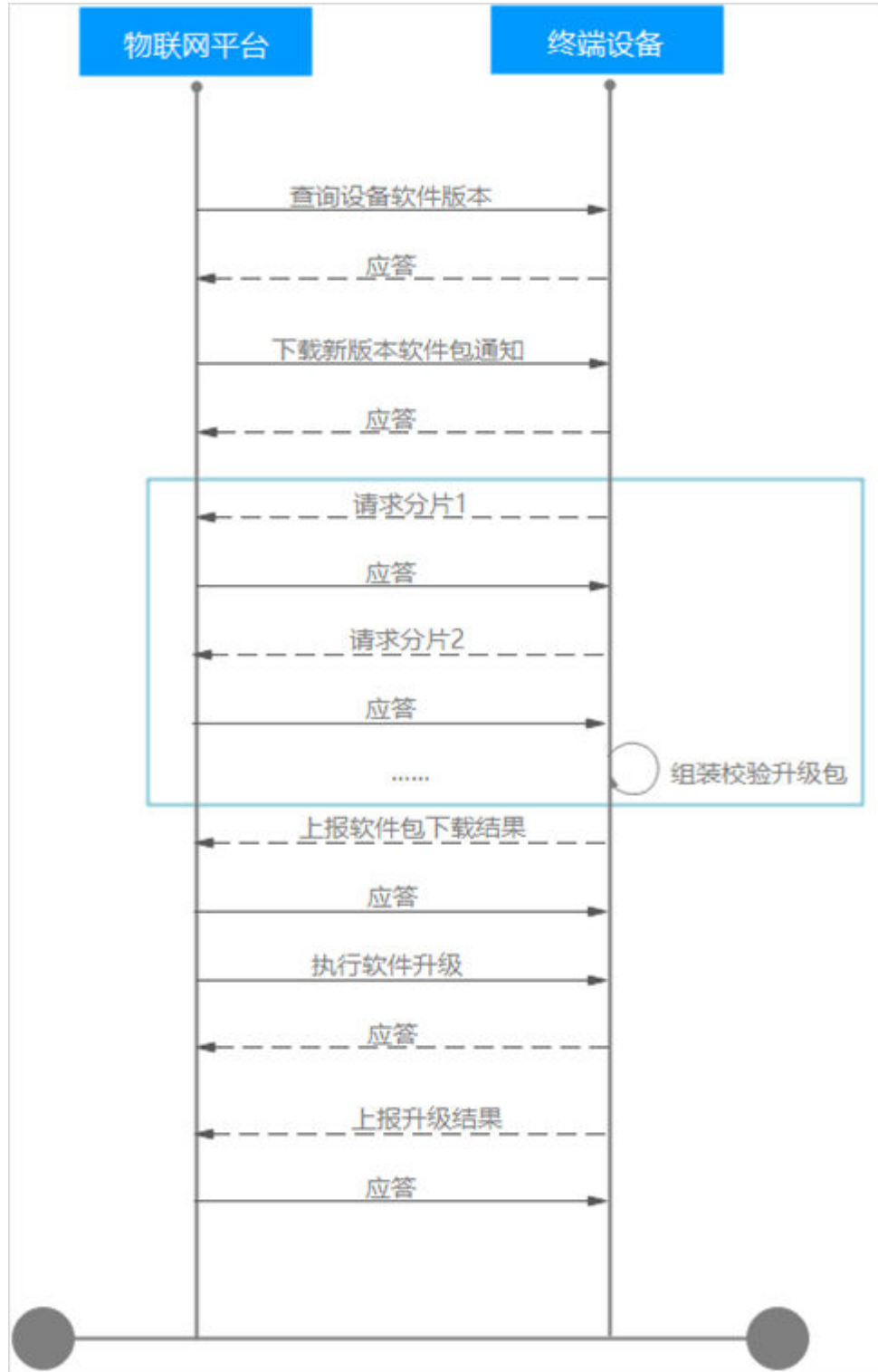
- 数据区：设备根据自身的情况对平台下发的新版本通知进行响应，本示例以设备应答“允许升级”为例进行介绍，得出数据区为：00。其它应答消息请根据应答消息字段进行适配。

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：FFFE01155618001256312E3000。

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。

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。

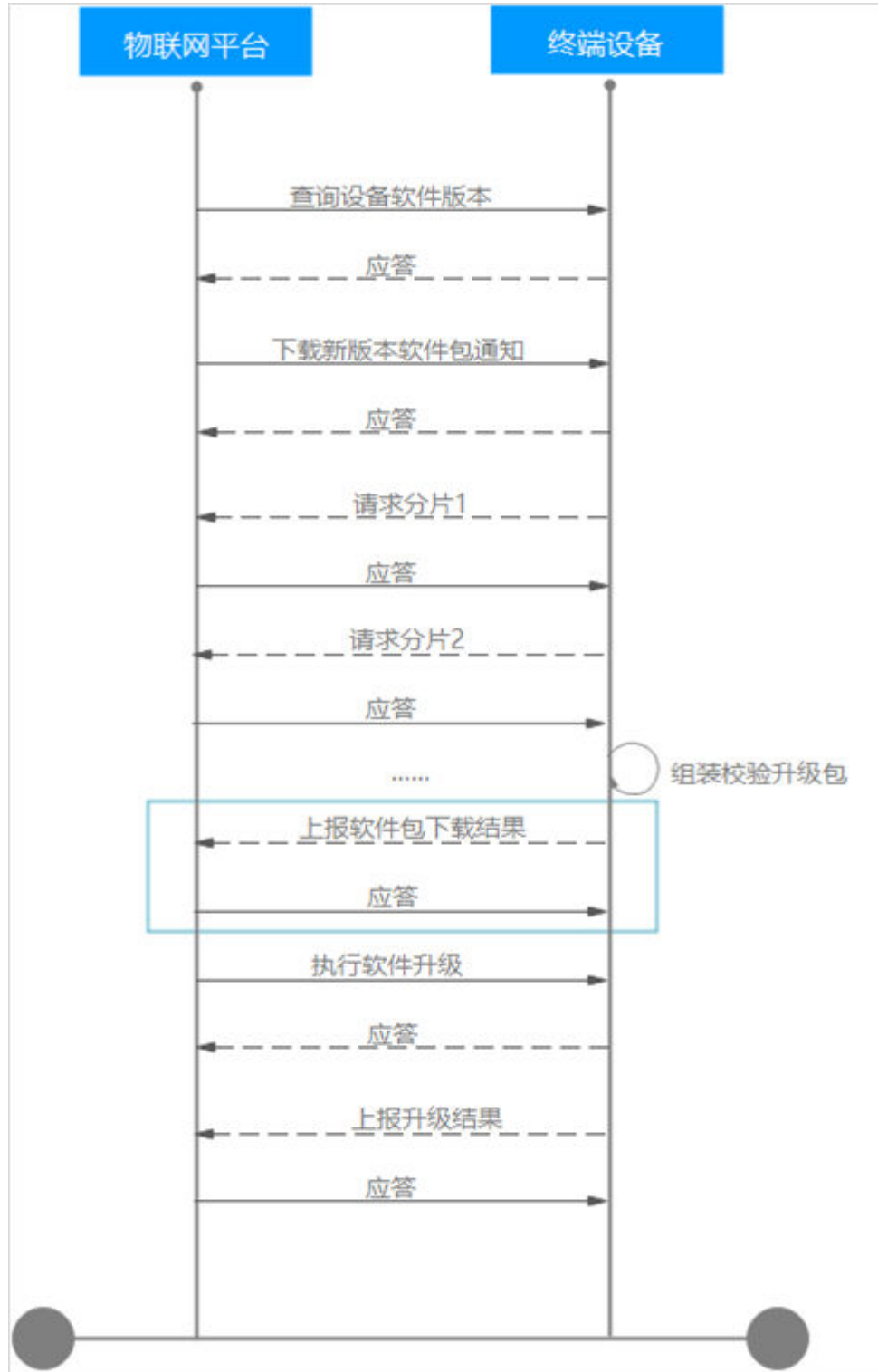
- 数据区长度：根据数据区的字段定义得出该数据长度为19，转换为十六进制为：0013。

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。

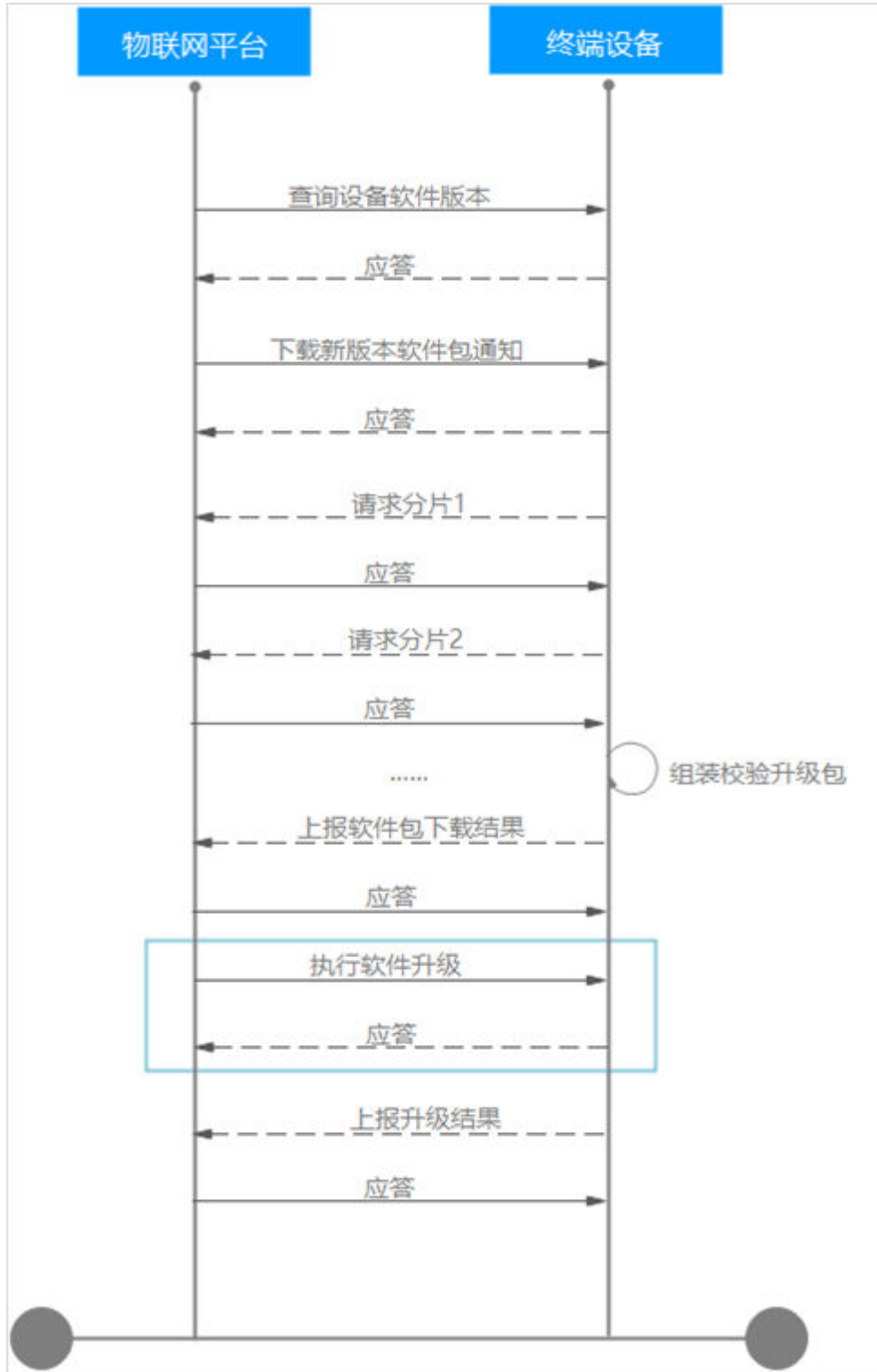
- 起始标识固定为：FFFE。
- 版本号固定为：01。
- 消息码：与请求的消息码一致：16。
- 校验码：CRC16计算前先用0000替代。
- 数据区长度：根据数据区的字段定义得出该数据长度为1个字节，转换为十六进制为：0001。
- 数据区：处理成功，则返回00，处理失败返回80。本示例以返回00处理成功为例进行说明。

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。

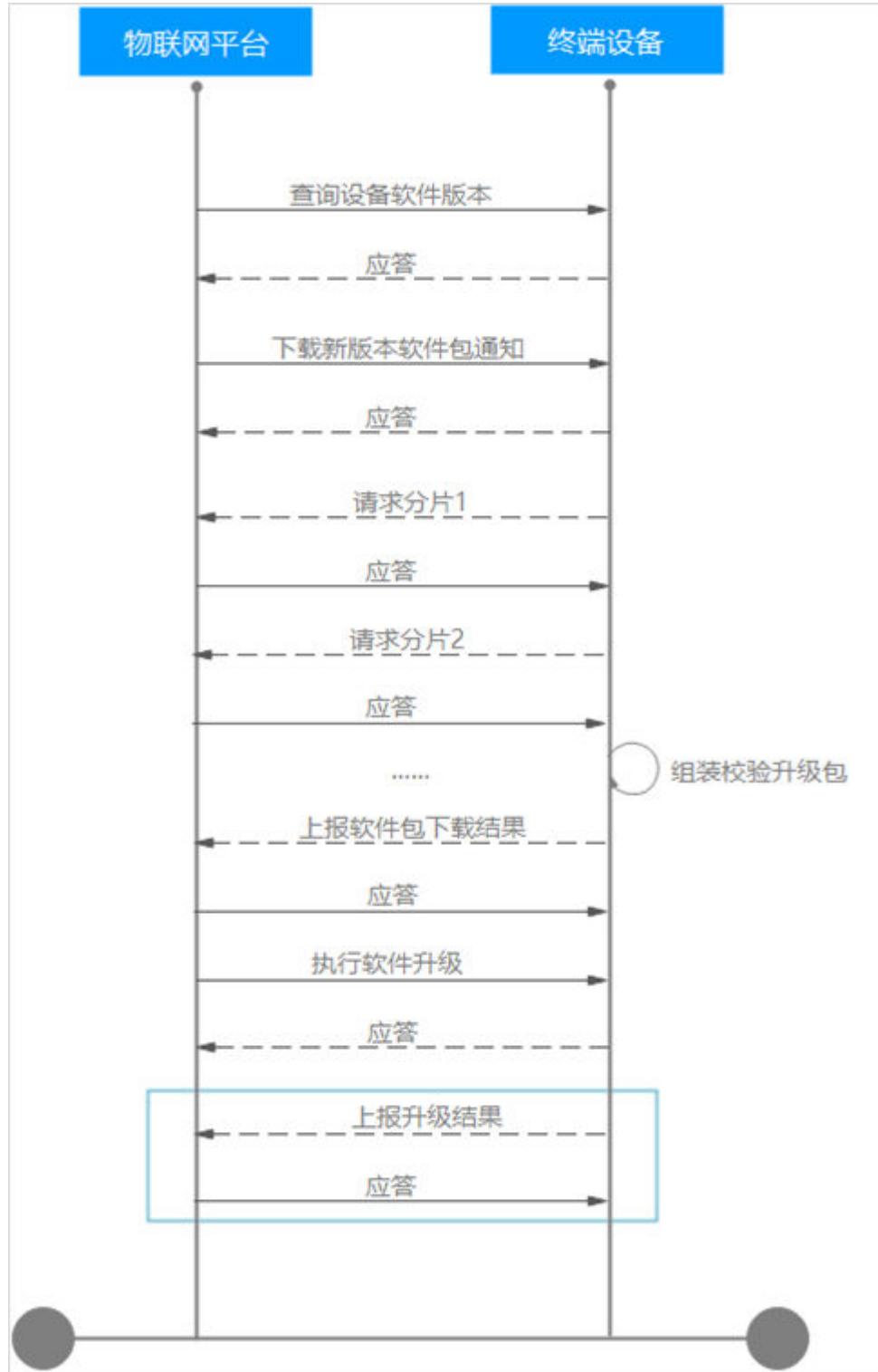
- 起始标识固定为：FFFE。
- 版本号固定为：01。
- 消息码：与请求的消息码一致：17。
- 校验码：CRC16计算前先用0000替代。
- 数据区长度：根据数据区的字段定义得出该数据长度为1个字节，转换为十六进制为：0001。
- 数据区：处理成功，则返回00，其它处理结果请参考数据区定义。本示例以返回00处理成功为例进行说明。

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：56312E30000000000000000000000000。

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 0056312E300000000000000000000000000000 (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：FFFE0118C7D200110056312E300000000000000000000000000000。

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。

至此，设备的软件升级功能适配就完成了，赶紧动手试一试吧。

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, 0xa8c4, 0x98e5, 0x6886, 0x78a7,  
        0x0840, 0x1861, 0x2802, 0x3823, 0xc9cc, 0xd9ed, 0xe98e, 0xf9af, 0x8948, 0x9969, 0xa90a, 0xb92b,  
        0x5af5, 0x4ad4, 0x7ab7, 0x6a96, 0x1a71, 0x0a50, 0x3a33, 0x2a12, 0xdbfd, 0xcdbc, 0xfbbf, 0xeb9e,  
        0x9b79, 0x8b58, 0xbb3b, 0xab1a, 0x6ca6, 0x7c87, 0x4ce4, 0x5cc5, 0x2c22, 0x3c03, 0x0c60, 0x1c41,  
        0xedaе, 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, 0xcdbc, 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);
}
```

4.4.2 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： <ul style="list-style-type: none">● 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

数据类型	描述
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

字段	数据类型	描述及要求
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”。
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。

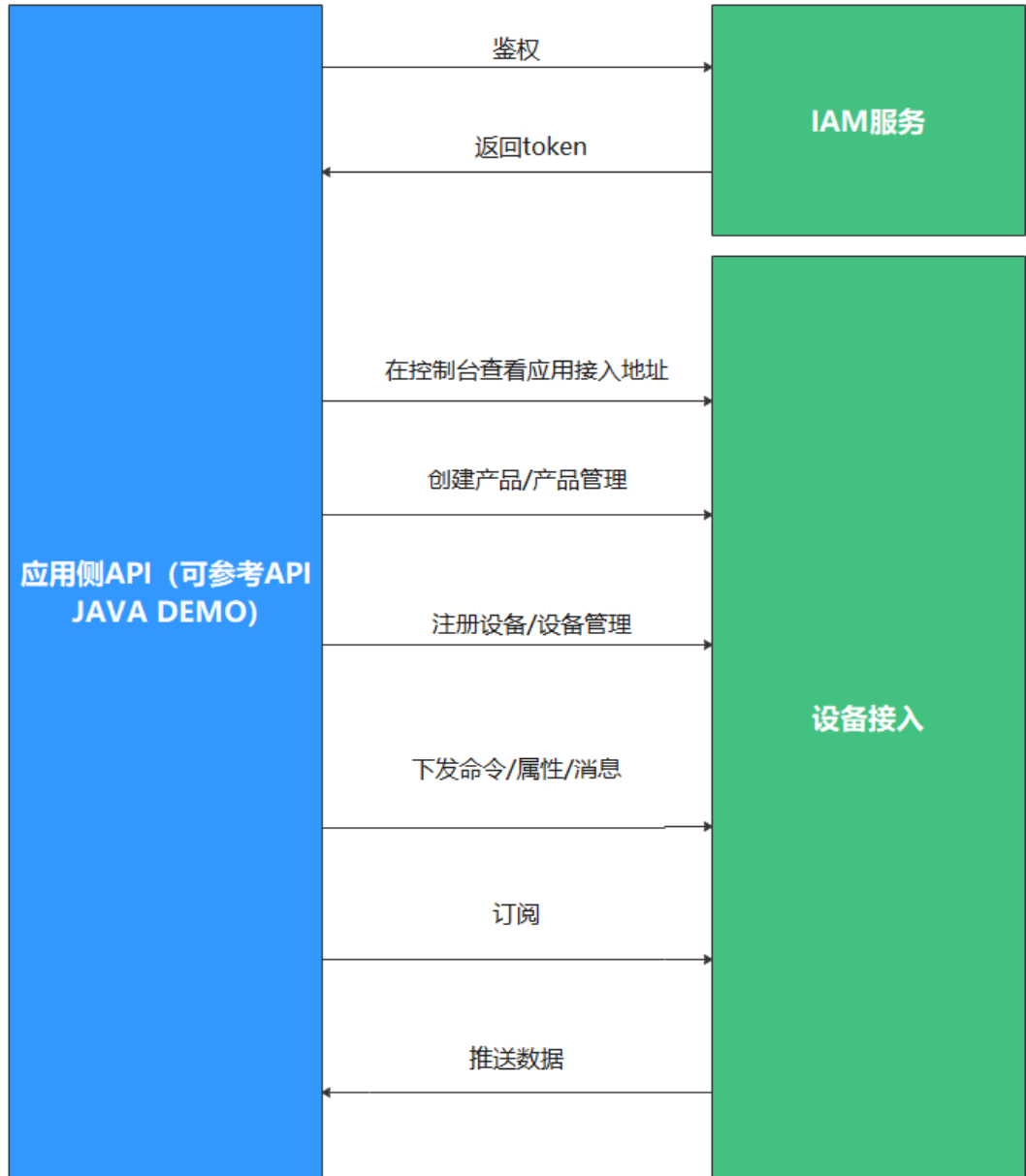
5 应用侧开发

5.1 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。

须知

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
应用侧开发 Java SDK	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 Java SDK使用指南 。	Java SDK
应用侧开发 C# SDK	C# SDK提供C#方法调用 应用侧API 与平台通信。使用指南可以参考 C# SDK使用指南 。	C# SDK
应用侧开发 Python SDK	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 Python SDK使用指南 。	Python SDK
应用侧开发 Go SDK	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 Go SDK使用指南 。	Go SDK
应用侧开发 Node.js SDK	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 Node.js SDK使用指南 。	Node.js SDK
应用侧开发 PHP SDK	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 PHP SDK使用指南 。	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。
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。

API分组	应用场景
接入凭证管理	接入凭证是用于AMQP、MQTTS等协议建立长链接时认证使用。
数据转发、设备联动	<p>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。</p> <ul style="list-style-type: none"> 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	<p>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。</p> <ul style="list-style-type: none"> 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	<p>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。</p> <p>当前仅设备支持标签。</p>
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	<p>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。</p> <ul style="list-style-type: none"> 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。
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。

API分组	应用场景
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。

5.2 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。


Choose your platform:

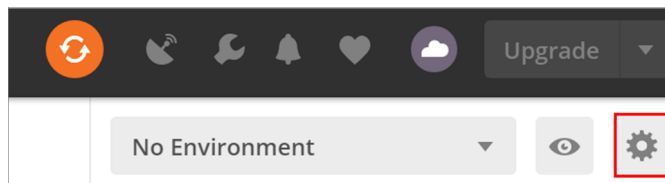


📖 说明

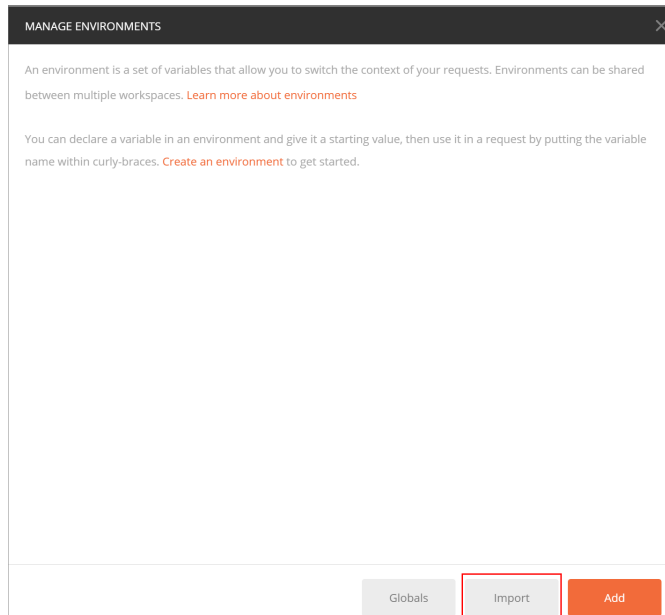
- 安装Postman依赖.NET Framework 4.5组件。
 - 如需下载Windows 32位Postman最新版本，访问[此处](#)下载。
2. 填写邮箱、用户名和密码注册Postman。

步骤2 导入Postman环境变量。

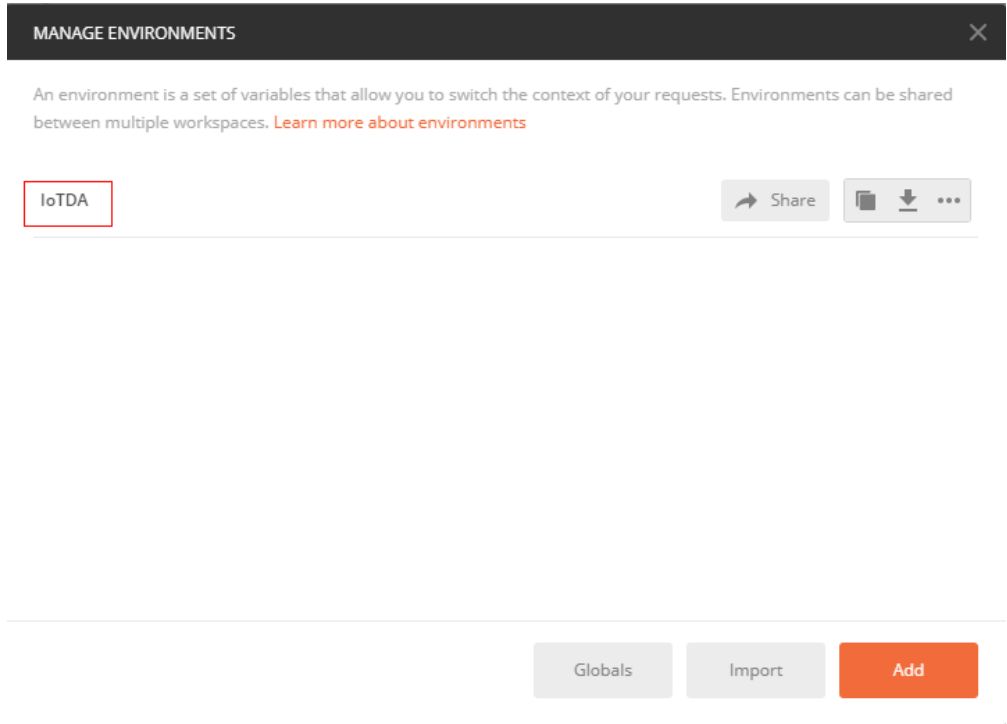
1. 单击右上角的  图标，打开“MANAGE ENVIRONMENTS”窗口。



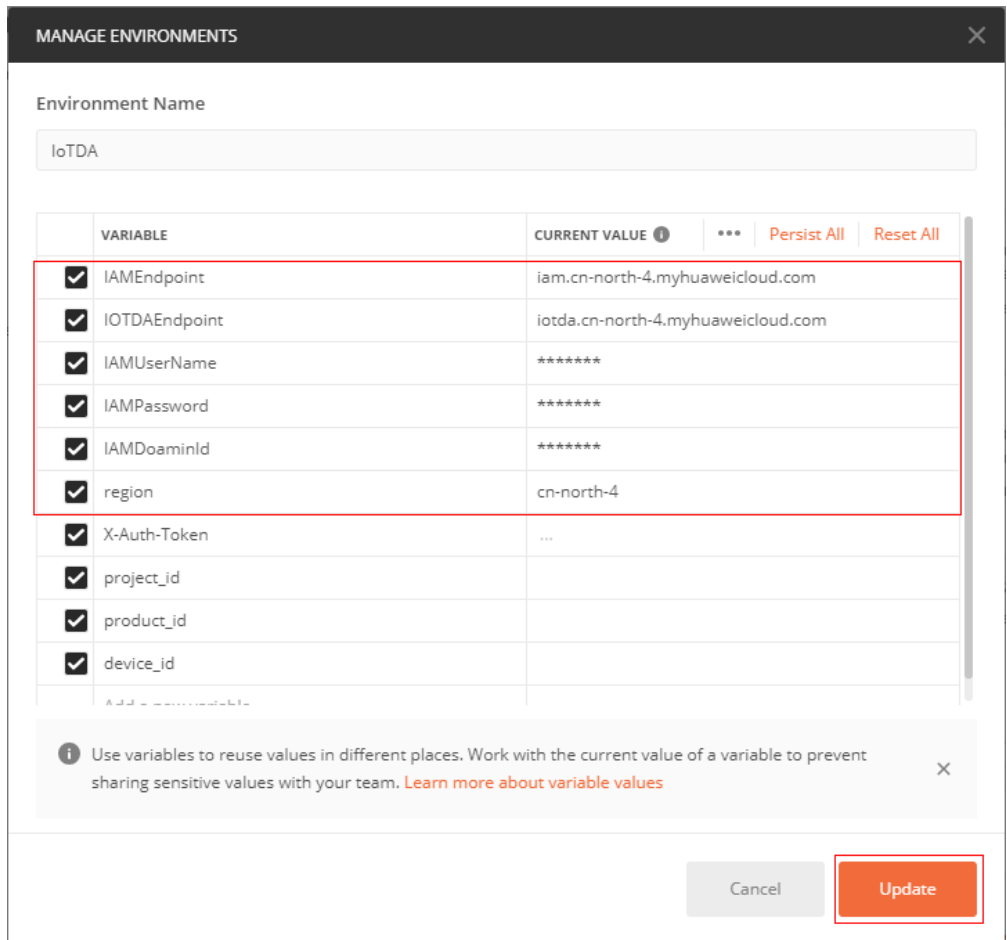
2. 单击“Import”，在弹出的页面中，单击“选择文件”，导入 **IoTDA.postman_environment.json**文件（下载[Collection](#)解压后获取）。



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。

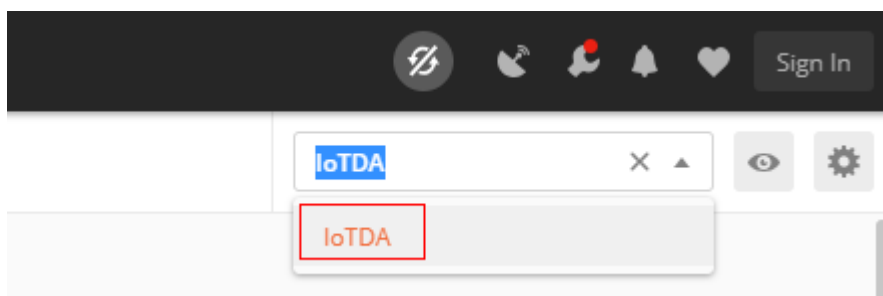
5. IoTDAEndpoint参考

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

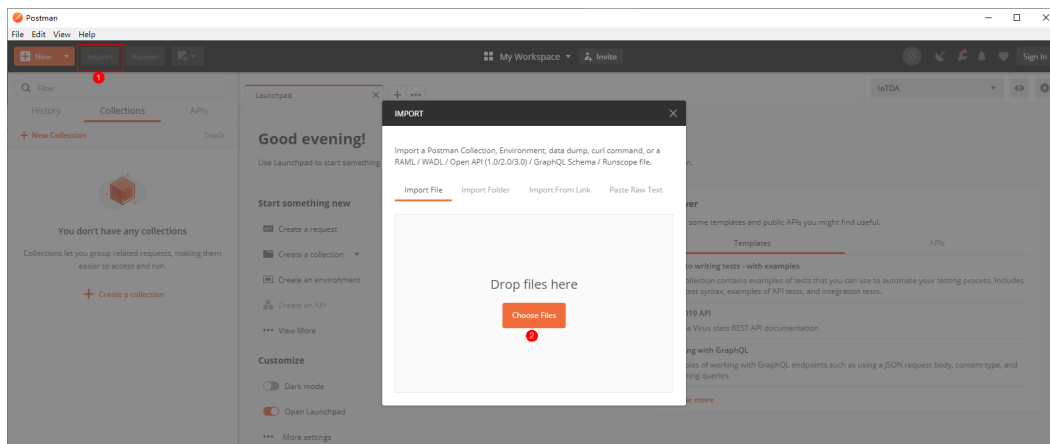
图 5-1 接入信息



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。

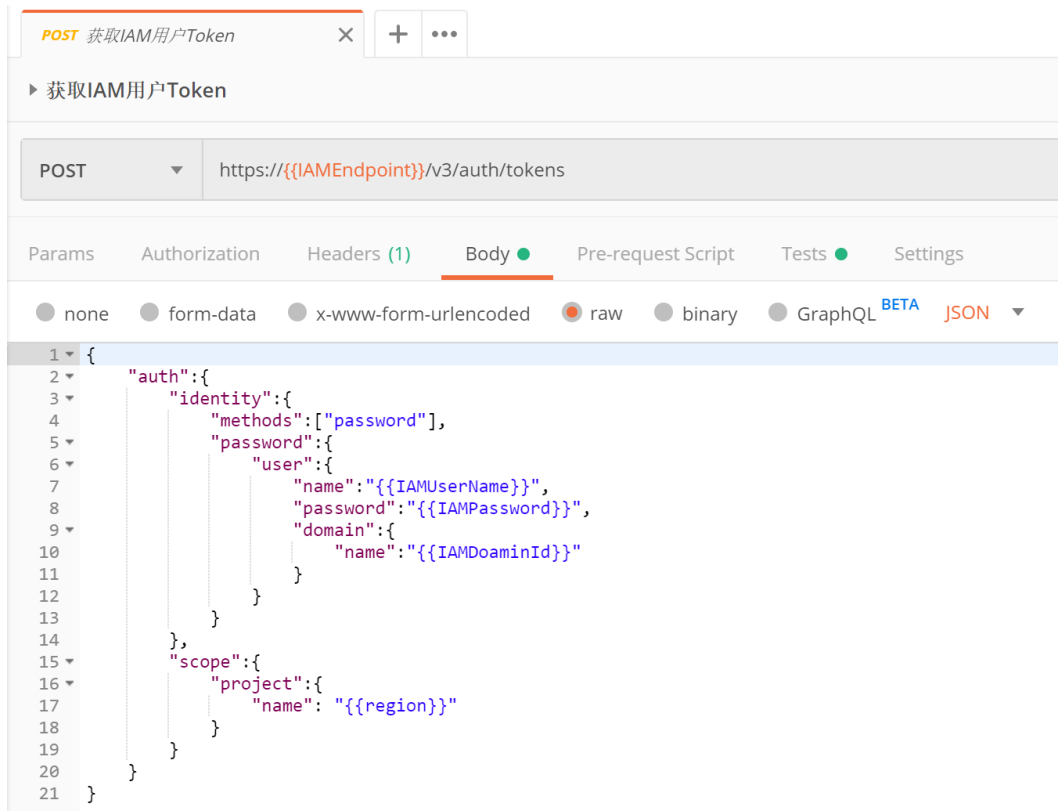
步骤1 配置“获取IAM用户Token”接口的HTTP方法、URL和Headers。

The screenshot shows an API client interface with the following configuration:

- Method:** POST
- URL:** https://{{IAMEndpoint}}/v3/auth/tokens
- Headers (1):**

KEY	VALUE
<input checked="" type="checkbox"/> Content-Type	application/json; charset=utf-8
Key	Value

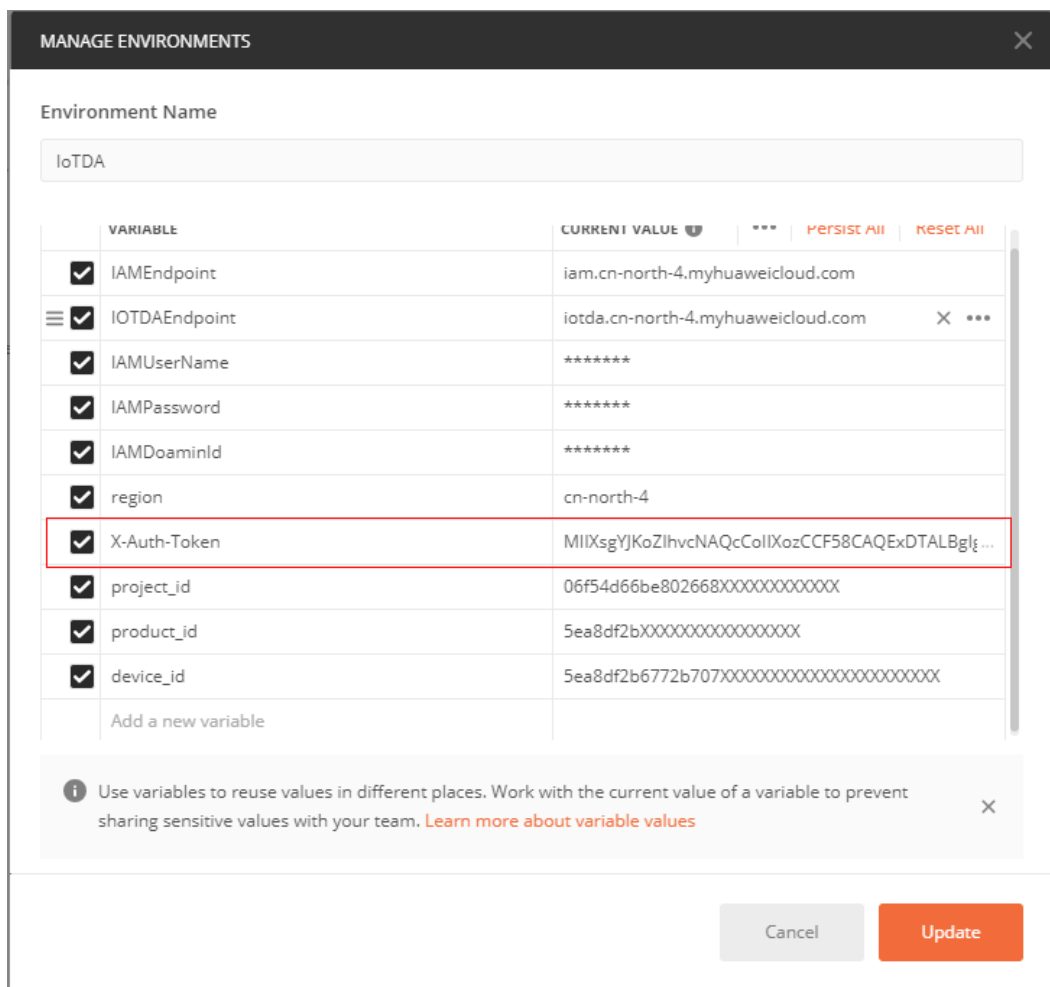
步骤2 配置“获取IAM用户Token”接口的Body。



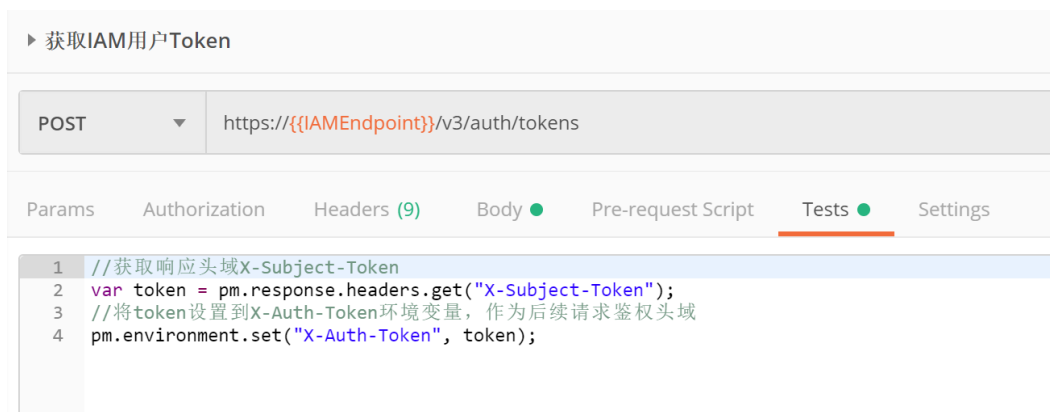
步骤3 单击“Send”，在下方查看返回码和响应消息内容。

KEY	VALUE
Date	Wed, 04 Mar 2020 01:00:53 GMT
Content-Type	application/json; charset=UTF-8
Content-Length	18468
Connection	keep-alive
X-IAM-Trace-Id	token_cn-north-4_null_5e627fb3ddfc776374456e059c3666a8
Cache-Control	no-cache, no-store, must-revalidate
Pragma	no-cache
Expires	Thu, 01 Jan 1970 00:00:00 GMT
X-Subject-Token	MllbZAYJKoZlhcNAQcCollbVTCGG1ECAQExDTALBglghkgBZQMEAgEwghI2BgkqhkiG9w0BBwGggh...
X-Request-Id	c93c1b0311803c589f61b89ef900b48d
Server	api-gateway
Strict-Transport-Security	max-age=31536000; includeSubdomains;
X-Frame-Options	SAMEORIGIN
X-Content-Type-Options	nosniff
X-Download-Options	noopen
X-XSS-Protection	1; mode=block;

步骤4 请将返回头域中的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。

► 查询IAM用户可以访问的项目列表

GET <https://{{IAMEndpoint}}/v3/auth/projects>

Params Authorization Headers (9) Body Pre-request Script Tests ● Settings

▼ Headers (2)

	KEY	VALUE
<input checked="" type="checkbox"/>	Content-Type	application/json
<input checked="" type="checkbox"/>	X-Auth-Token	{{X-Auth-Token}}

步骤2 单击“Send”，在下方查看返回码和响应消息内容。

Body Cookies Headers (15) Test Results Status: 200 OK

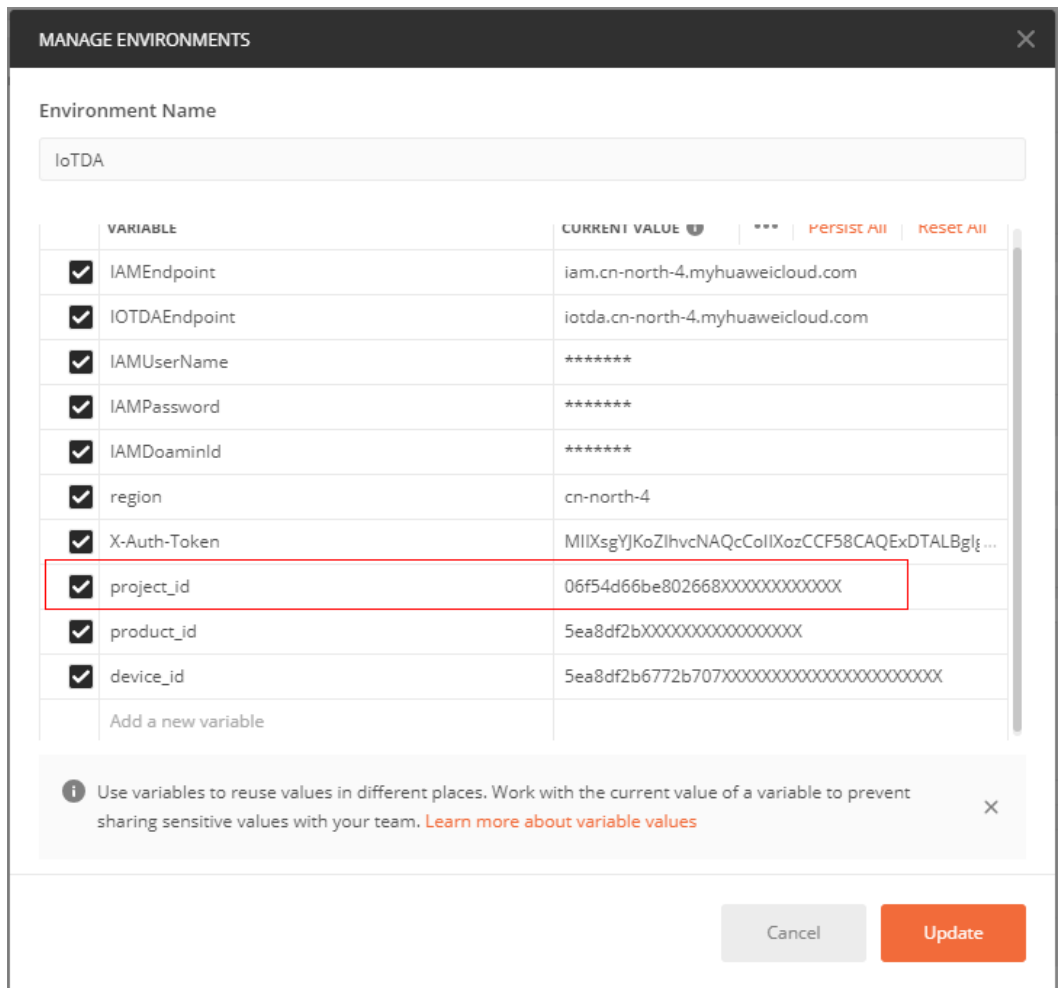
Pretty Raw Preview Visualize BETA JSON ↻

```

1  {
2    "projects": [
3      {
4        "domain_id": "ba21fb12cfc440569954a2ac9a99323a",
5        "is_domain": false,
6        "parent_id": "ba21fb12cfc440569954a2ac9a99323a",
7        "name": "ap-southeast-1",
8        "description": "",
9        "links": {
10         "self": "https://iam.myhuaweicloud.com/v3/projects/072a8dbc980100d2f0ec0146f237196"
11       },
12       "id": "072a8dbc980100d2f0ec0146f237196",
13       "enabled": true
14     },
15     {
16       "domain_id": "ba21fb12cfc440569954a2ac9a99323a",
17       "is_domain": false,
18       "parent_id": "ba21fb12cfc440569954a2ac9a99323a",
19       "name": "MOS",
20       "description": "",
21       "links": {
22         "self": "https://iam.myhuaweicloud.com/v3/projects/b6c7508ff62e4beb91cee1c1ce49ecd9"
23       },
24       "id": "b6c7508ff62e4beb91cee1c1ce49ecd9",
25       "enabled": true
26     }
27   ]
28 }
```

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。

```
Body Cookies Headers (15) Test Results Status: 200 OK
Pretty Raw Preview Visualize BETA JSON
95 },
96   "id": "072a8dcdbd08026542f00c014ee62ff50",
97   "enabled": true
98 },
99 {
100   "domain_id": "ba21fb12cfc440569954a2ac9a99323a",
101   "is_domain": false,
102   "parent_id": "ba21fb12cfc440569954a2ac9a99323a",
103   "name": "cn-north-4",
104   "description": "",
105   "links": {
106     "self": "https://iam.myhuaweicloud.com/v3/projects/06f54d66be8026682f21c014815a69ba"
107   },
108   "id": "06f54d66be8026682f21c014815a69ba",
109   "enabled": true
110 },
111 {
112   "domain_id": "ba21fb12cfc440569954a2ac9a99323a",
113   "is_domain": false,
114   "parent_id": "ba21fb12cfc440569954a2ac9a99323a",
115   "name": "ap-southeast-3",
116   "description": "",
117   "links": {
118     "self": "https://iam.myhuaweicloud.com/v3/projects/072a8dcdbd08026502fb1c014ead6fc7a"
119   },
120   "id": "072a8dcdbd08026502fb1c014ead6fc7a",
121   "enabled": true
122 },
},
```



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。

► 查询IAM用户可以访问的项目列表

GET https://{{IAMEndpoint}}/v3/auth/projects

Params Authorization Headers (9) Body Pre-request Script Tests ● Settings

```

1 var region = pm.environment.get("region");
2 var jsonData = pm.response.json();
3 var projects = jsonData.projects;
4 for (i = 0; i < projects.length; i++) {
5     if (projects[i].name == region) {
6         pm.environment.set("project_id", projects[i].id);
7     }
8 }

```

---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",

```

```

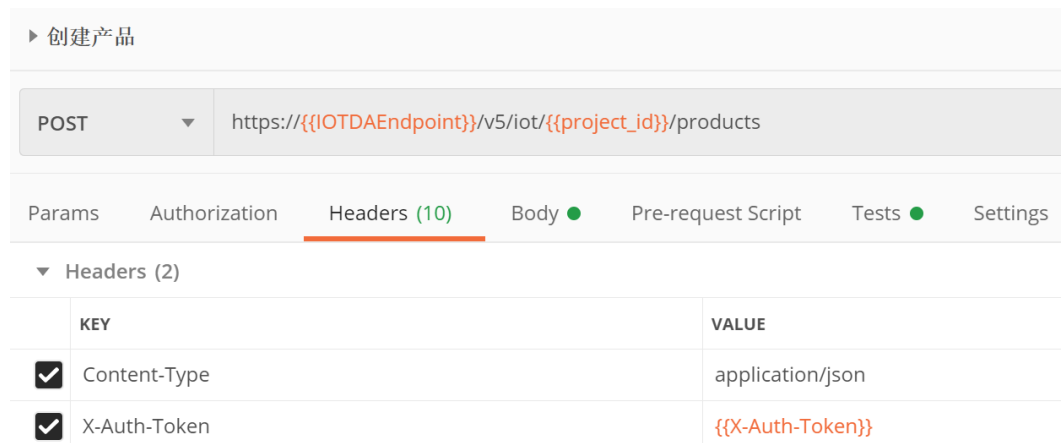
    "description": "force",
    "step": 0.1,
    "enum_list": [ "string" ],
    "required": false,
    "max_length": 100
  } ]
},
"paras": [ {
  "unit": "km/h",
  "min": "1",
  "max": "100",
  "para_name": "force",
  "data_type": "string",
  "description": "force",
  "step": 0.1,
  "enum_list": [ "string" ],
  "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。



▶ 创建产品

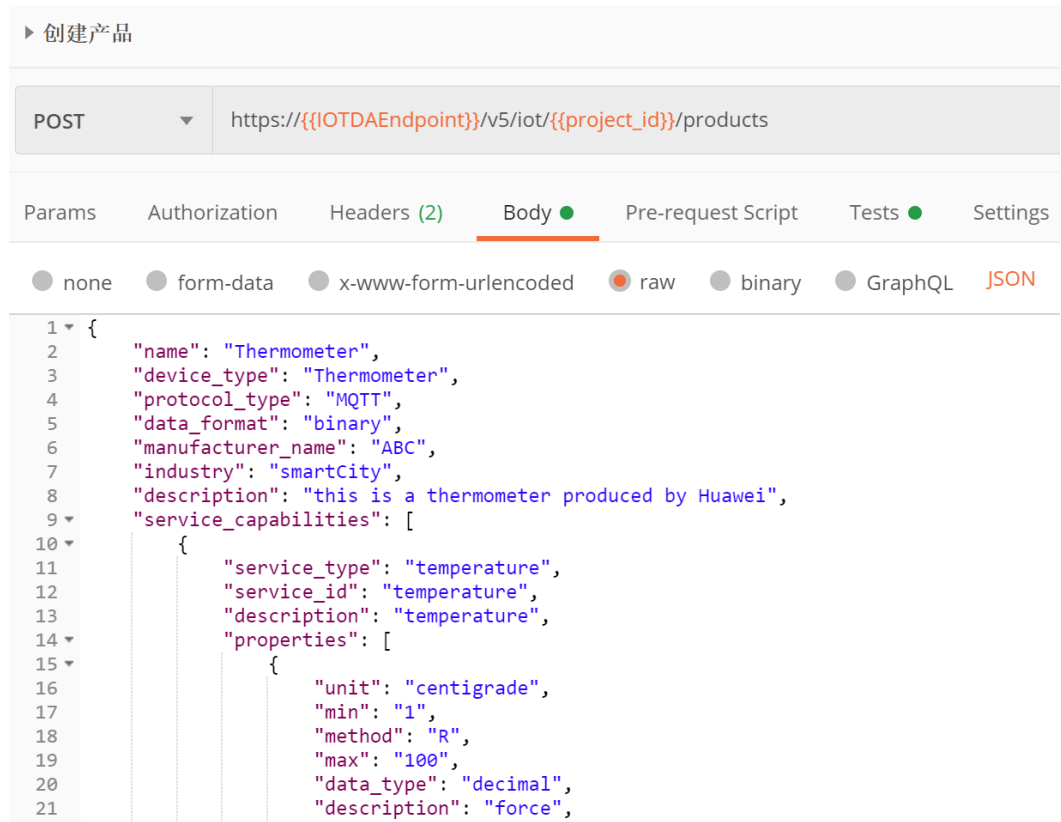
POST https://{{IOTDAEndpoint}}/v5/iot/{{project_id}}/products

Params Authorization **Headers (10)** Body ● Pre-request Script Tests ● Settings

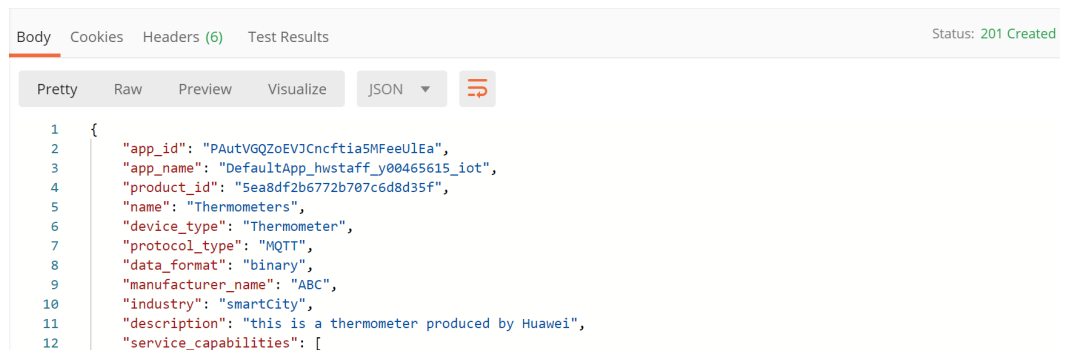
▼ Headers (2)

	KEY	VALUE
<input checked="" type="checkbox"/>	Content-Type	application/json
<input checked="" type="checkbox"/>	X-Auth-Token	{{X-Auth-Token}}

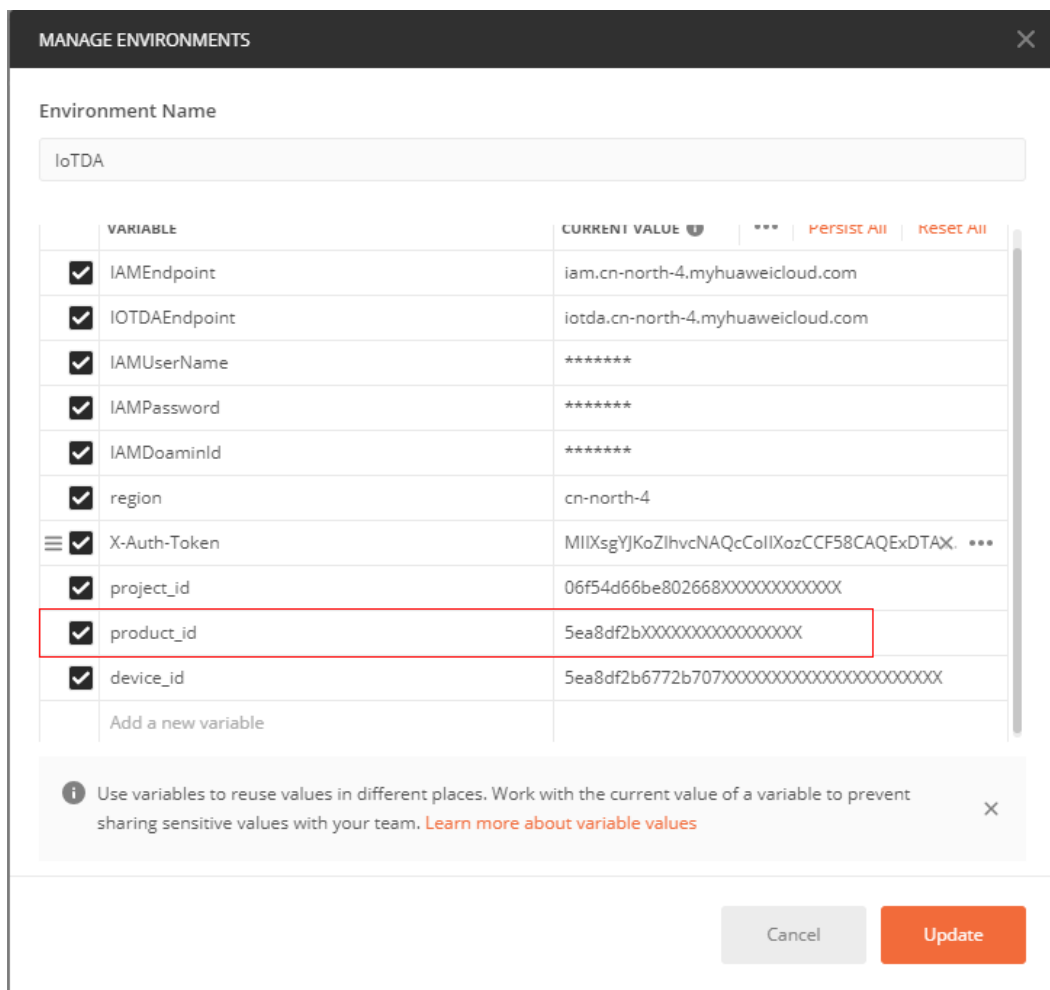
步骤2 配置“创建产品”接口的BODY。



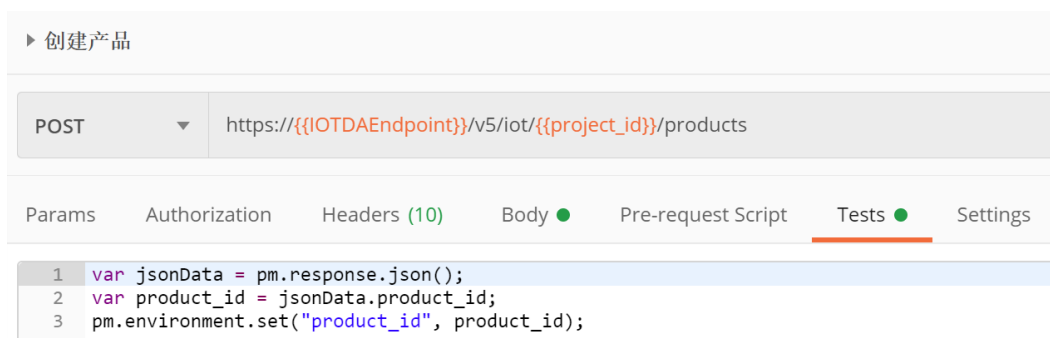
步骤3 单击“Send”，在下方查看返回码和响应消息内容。



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。

▶ 查询产品

GET ▼ https://{{IOTDAEndpoint}}/v5/iot/{{project_id}}/products/{{product_id}}

Params Authorization Headers (9) Body Pre-request Script Tests Settings

▼ Headers (2)

	KEY	VALUE
<input checked="" type="checkbox"/>	Content-Type	application/json
<input checked="" type="checkbox"/>	X-Auth-Token	{{X-Auth-Token}}

步骤2 单击“Send”，在下方查看返回码和响应消息内容。

Body Cookies Headers (6) Test Results Status: 200 OK

Pretty Raw Preview Visualize JSON ↻

```

1  {
2    "app_id": "PAutVG0ZoEVJCncftia5MFeeU1Ea",
3    "app_name": "DefaultApp_hwstaff_y00465615_iot",
4    "product_id": "5ea8df2b6772b707c6d8d35f",
5    "name": "Thermometers",
6    "device_type": "Thermometer",
7    "protocol_type": "MQTT",
8    "data_format": "binary",
9    "manufacturer_name": "ABC",
10   "industry": "smartCity",
11   "description": "this is a thermometer produced by Huawei",
12   "service_capabilities": [
13     {
14       "service_id": "temperature",
15       "service_type": "temperature",
16       "properties": [
17         {
18           "property_name": "temperature",
19           "required": true,
20           "data_type": "decimal",
21           "enum_list": [
22             "string"
23           ],
24           "min": "1",
25           "max": "100",
26           "max_length": 100,
27           "step": 0.1,
28           "unit": "centigrade",
29           "method": "R",
30           "description": "force",
31           "default_value": null
32         }
33       ]
14     }
15   ]
16 }

```

---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": "dc0f1016f495157344ac5f1296335cff725ef22f",
    "secret": "3b935a250c50dc2c6d481d048cefdc3c",
    "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。

注册设备

POST ▼ https://{{IOTDAEndpoint}}/v5/iot/{{project_id}}/devices

Params Authorization Headers (10) Body ● Pre-request Script Tests ● Settings

▼ Headers (2)

	KEY	VALUE
<input checked="" type="checkbox"/>	Content-Type	application/json
<input checked="" type="checkbox"/>	X-Auth-Token	{{X-Auth-Token}}

步骤2 配置“创建设备”接口的Body。

注册设备

POST ▼ https://{{IOTDAEndpoint}}/v5/iot/{{project_id}}/devices

Params Authorization Headers (10) Body ● Pre-request Script Tests ● Settings

● none ● form-data ● x-www-form-urlencoded ● raw ● binary ● GraphQL JSON ▼

```

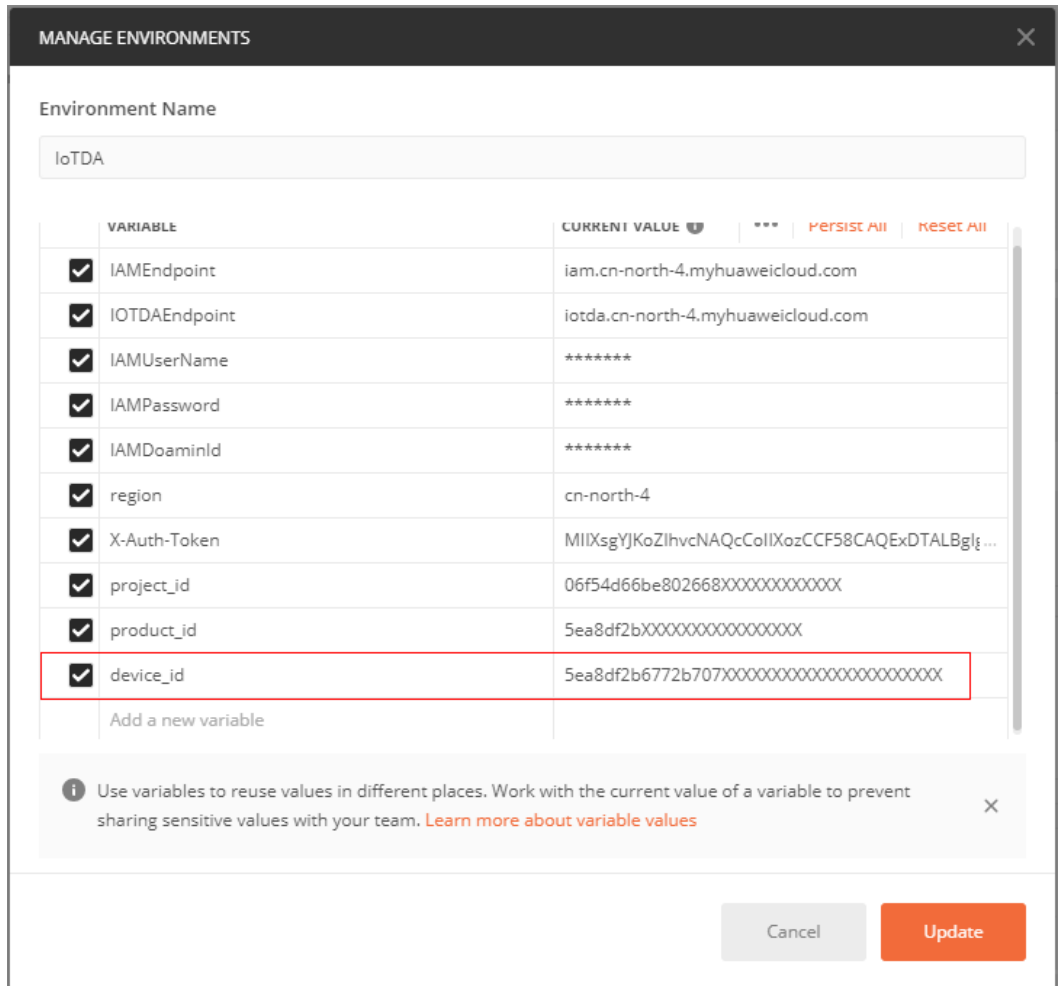
1 {
2   "node_id": "ABC123456789",
3   "device_name": "testdevice",
4   "product_id": "{{product_id}}",
5   "auth_info": {
6     "auth_type": "SECRET",
7     "secure_access": true,
8     "fingerprint": "dc0f1016f495157344ac5f1296335cff725ef22f",
9     "secret": "3b935a250c50dc2c6d481d048cefdc3c",
10    "timeout": 300
11  },
12  "description": "watermeter device"
13 }
```

步骤3 单击“Send”，在下方查看返回码和响应消息内容。

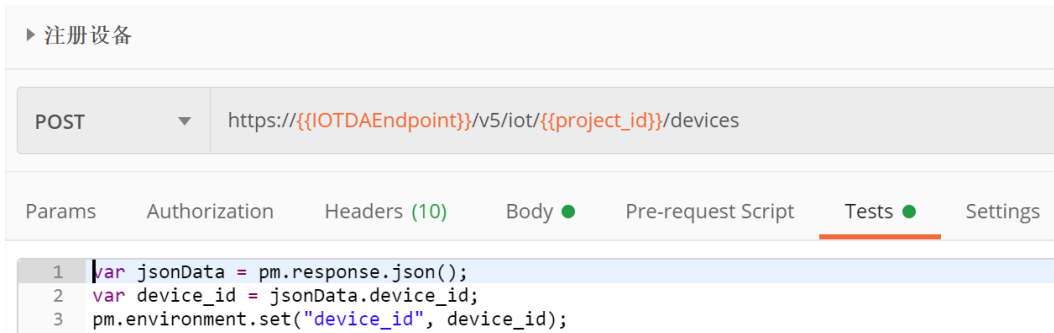
```

1  {
2    "app_id": "PAutVGQZoEVJCncftia5MFeeU1Ea",
3    "device_id": "5e5efefc9071cb07289e7733_ABC123456789",
4    "node_id": "ABC123456789",
5    "gateway_id": "5e5efefc9071cb07289e7733_ABC123456789",
6    "device_name": "dianadevice",
7    "node_type": "GATEWAY",
8    "description": "watermeter device",
9    "fw_version": null,
10   "sw_version": null,
11   "auth_info": {
12     "auth_type": "SECRET",
13     "secret": "3b935a250c50dc2c6d481d048cefdc3c",
14     "fingerprint": null,
15     "secure_access": true,
16     "timeout": 300
17   },
18   "product_id": "5e5efefc9071cb07289e7733",
19   "status": "INACTIVE",
20   "create_time": "20200304T010621Z",
21   "tags": []
22 }
    
```

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。

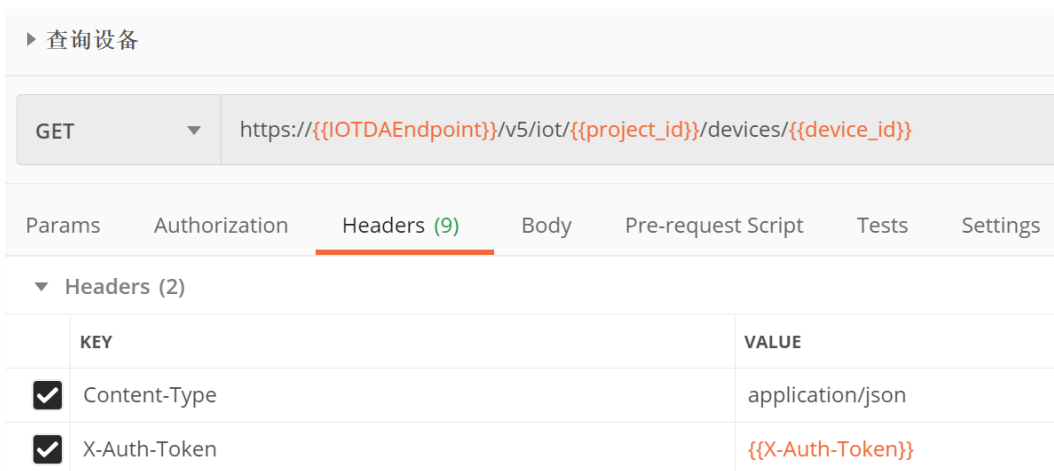
应用服务器需要构造一个HTTP请求，请求示例如下：

```
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。


```
Body Cookies Headers (14) Test Results Status: 200 OK
Pretty Raw Preview Visualize BETA JSON
1 {
2   "app_id": "PAutVGQZoEVJCncftia5MFeeU1Ea",
3   "device_id": "5e5efefc9071cb07289e7733_ABC123456789",
4   "node_id": "ABC123456789",
5   "gateway_id": "5e5efefc9071cb07289e7733_ABC123456789",
6   "device_name": "dianadevice",
7   "node_type": "GATEWAY",
8   "description": "watermeter device",
9   "fw_version": null,
10  "sw_version": null,
11  "auth_info": {
12    "auth_type": "SECRET",
13    "secret": "*****",
14    "fingerprint": null,
15    "secure_access": true,
16    "timeout": 0
17  },
18  "product_id": "5e5efefc9071cb07289e7733",
19  "status": "INACTIVE",
20  "create_time": "20200304T010621Z",
21  "tags": []
22 }
```

----结束