物联网通信

开发者手册

产品文档

腾讯云

文档目录

开发者手册
功能组件

最近更新时间：2021-09-10 10:40:19

1. SDK

详情请参见 SDK 文档。目前支持 Linux、Android 平台的设备 SDK、并支持移植到不同的硬件平台 。

2. 设备接入

设备通过 SDK 接入腾讯物联网通信平台：

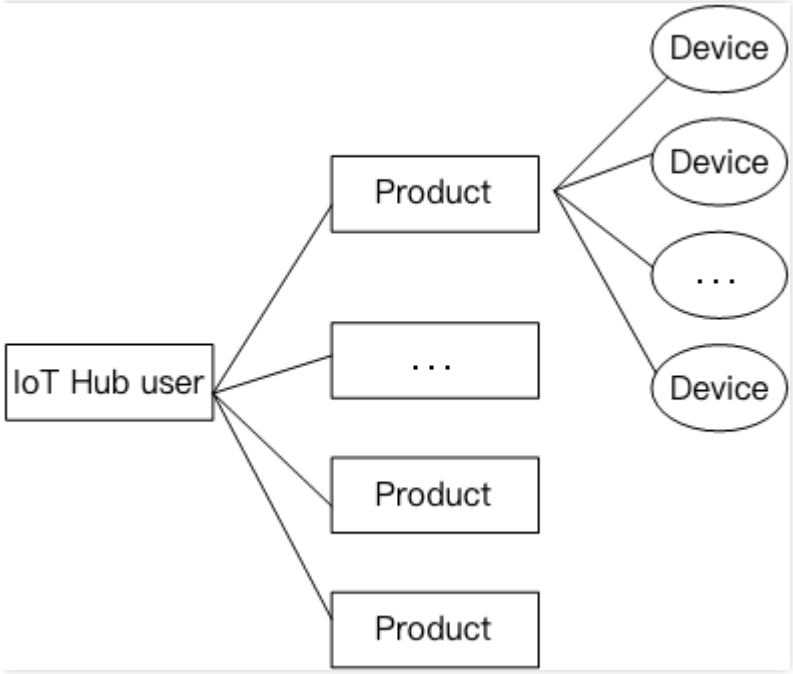
- 应用层基于 MQTT、CoAP 协议。
- 传输层基于 TCP、UDP 协议，并在此基础上引入安全网络传输协议（TLS、DTLS），实现客户端和服务器端的双向鉴权、数据加密传输。
- SDK 支持 RTOS 移植能力，跨平台移植，框架抽离硬件平台抽象层，可基于不同平台快速、轻松接入物联网通信。

设备 SDK 支持 TLS（对应 MQTT）、DTLS（对应 CoAP）的非对称和对称加密两种鉴权方式，保护设备通信安全：

- 非对称加密
  安全级别高，基于证书、非对称加密算法，适用于硬件规格较高、对功耗不是很敏感的设备。依赖设备证书、私钥，根证书等信息，在物联网通信创建设备时，会返回相关信息。
- 对称加密
  安全级别普通，基于密钥、对称加密算法，适用于资源受限、对功耗敏感的设备。依赖于设备 psKey ，在物联网通信创建设备时，会返回相关信息。

除设备 SDK 接入外，腾讯物联网通信还提供 HTTP 接入，接入协议门槛低，适用低功耗、短连接的数据上报场景。

3. 设备管理



- 一个腾讯云账号下，最多可以创建2000个产品，每个产品下最多创建100万台设备。一个设备只能隶属于一个产品。产品名和设备名在同一云账号下唯一。
- 提供设备的启用/禁用功能。设备被禁用后将不能接入物联网通信平台，将无法执行与设备有关的操作，但与设备相关联的信息依然保留，仍可查询设备相关信息。

4. 权限管理

在腾讯物联网通信，设备能够发布和订阅的 Topic 受到严格管理。一个产品下的所有设备具备相同的 Topic 类 权限，默认包括：

Topic	说明
${productId}/${deviceName}/event	发布权限，用于设备上报数据
${productId}/${deviceName}/control	订阅权限，用于设备获取后台下发的数据

上述$符包含的 productId、deviceName，针对具体创建的设备，将映射为具体的产品 Id 和设备名字。举例，一个产品名字为 pro 的产品（假设产品Id是“pro_id”）下有2个设备（假设设备名字分别为 “dev_1”、“dev_2”），那么 dev_1 可以发布的 Topic 包括 pro_id/dev_1/event，可以订阅的 topic 包括 pro_id/dev_1/control，但是不可以发布 pro_id/dev_2/event，不可以订阅 pro_id/dev_2/control。

用户可以通过控制台进行 Topic 权限的编辑修改、增删产品的 Topic 类权限。

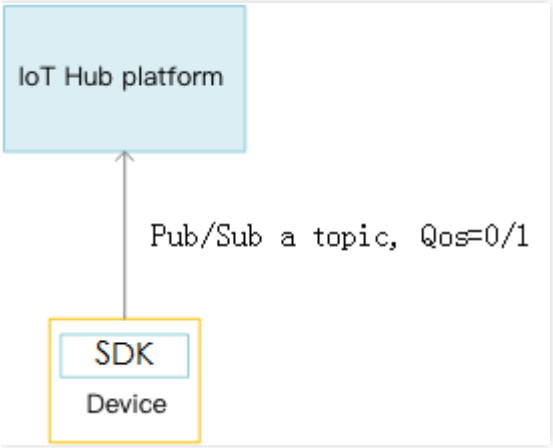
为了方便设备 SDK 订阅批量的 Topic ，设备进行订阅和取消订阅时可以使用通配符来表示多个符合的 Topic ：

通配符	描述
#	此通配符只能出现在 topic 的最后，代表本级以及所有子级 Topic ，例如，通配符 Topic 为 pro_id/dev_1/# ，这不仅可以代表 pro_id/dev_1/event ,也可以代表 pro_id/dev_1/event/subeventA
+	代表本级所有 topic ，只能出现在 deviceName 后，例如通配符 topic 为 pro_id/dev_1/event/+，可以代表 pro_id/dev_1/event/subeventA，又可以代表 pro_id/dev_1/event/subeventB，但不能代表 pro_id/dev_1/event/subeventA/close。可以出现多次，如 pro_id/dev_1/event/+/subeventA/+

通配符必须作为完整的一级，${productId}/${deviceName}/e# 和 ${productId}/${deviceName}/e+ 都是非法格式。
腾讯物联网通信定义的系统主题（$shadow，$ota，$sys）不支持通配符。

- 增加订阅时通配符的表现为：对通配符 Topic 匹配到的该产品下有订阅权限的 Topic 进行订阅，若匹配到的 Topic 列表为空也返回成功。
- 取消订阅时通配符的表现为：对通配符 Topic 匹配到的已订阅的 Topic 进行取消订阅，若匹配到的 Topic 列表为空也返回成功。${productId}/${deviceName}/# 为清除所有用户主题的订阅。

5. 消息管理



对于 MQTT 的数据传输，腾讯物联网通信支持 QoS=0 或 1，但不支持 QoS=2。基于 MQTT 协议。设备消息支持离线存储。

- QoS=0，最多只往设备发一次
  对数据传输可靠性要求一般的场景，请在 Publish、Subscribe 时选择这个 QoS。
- QoS=1，至少让设备收到一次
  对数据传输可靠性要求高的场景，请在 Publish、Subscribe 时选择这个 Qos。

其他参数见下表：

参数	说明

参数	说明
topic 名字长度	不超过64字节
MQTT 协议包大小	不超过16K字节
QoS=1 的消息存储时长（接收方离线或在线发不通）	24小时
未被设备确认的 QoS=1 消息数量	不超过150条

6. 设备影子

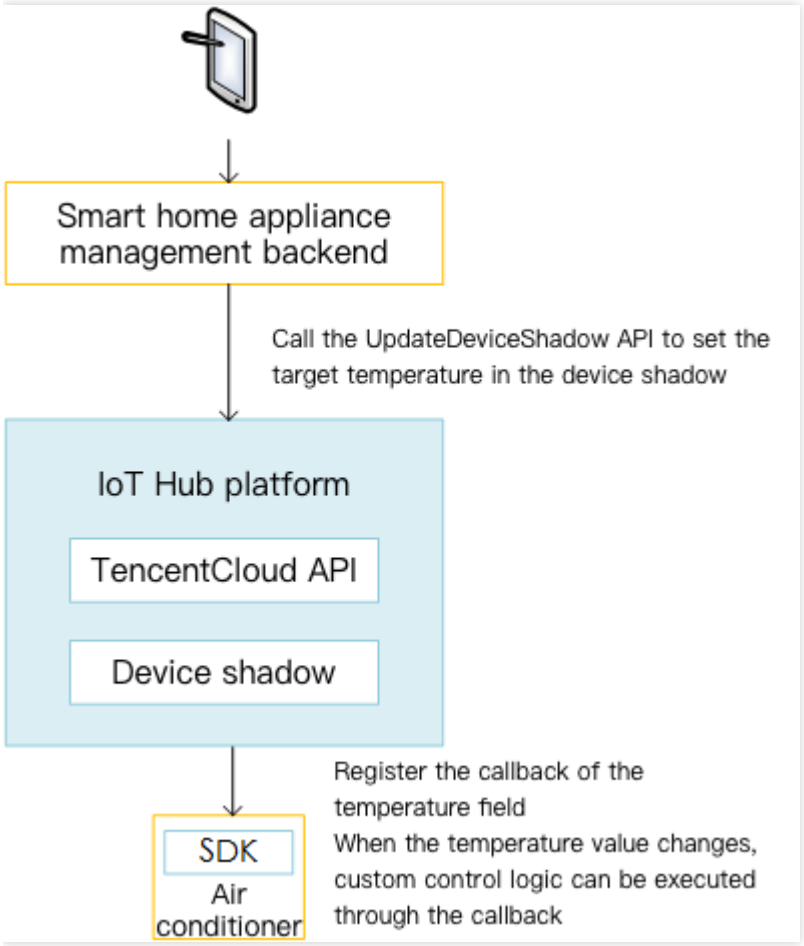
设备影子本质上是一份在服务器端缓存的设备数据（JSON 形式），主要用于保存：

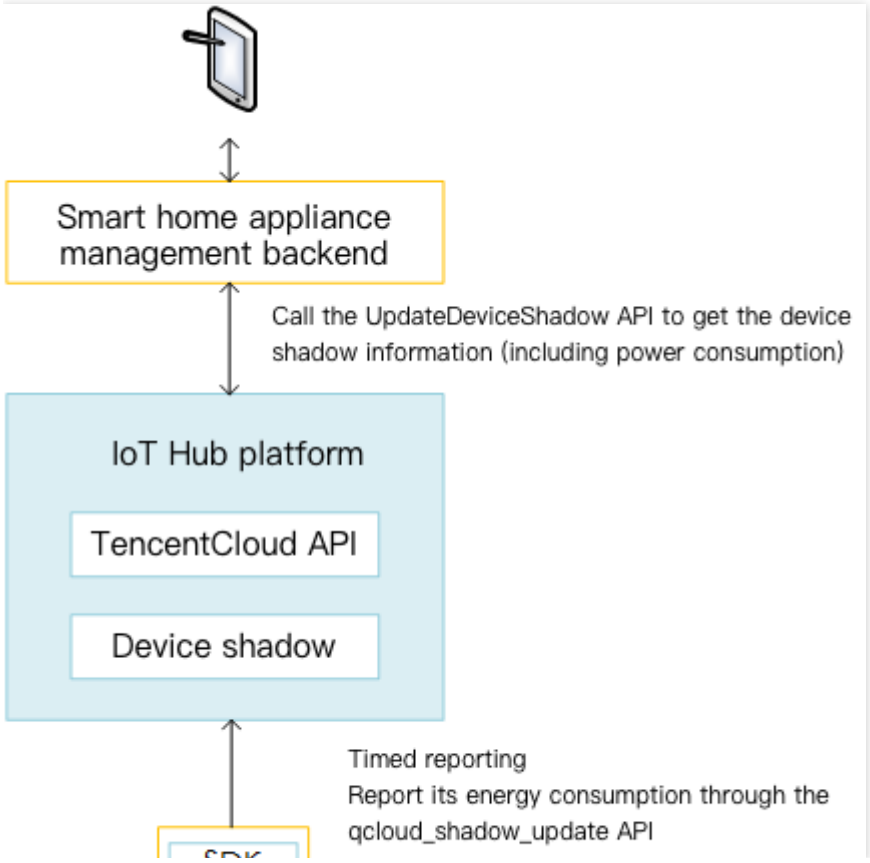
- 设备的当前配置
- 设备的当前状态

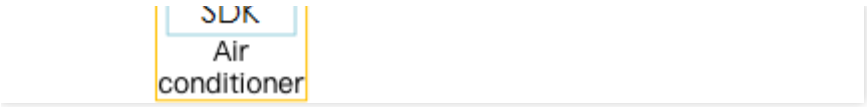
作为中介，设备影子可以有效实现设备和用户应用之间的数据双向同步：

- 对于设备配置，用户应用不需要直接修改设备，只需要修改服务器端的设备影子，由设备影子同步到设备。即使当时设备不在线，设备上线后仍能从设备影子同步到最新配置。
- 对于设备状态，设备将状态上报到设备影子，用户应用查询时，只需查询设备影子即可。这样可以有效减少设备和服务器端的网络交互，尤其是低功耗设备。

下图是“快速开始”里设备影子的应用示例：


Smart home appliance management backend
Call the UpdateDeviceShadow API to set the target temperature in the device shadow
IoT Hub platform
TencentCloud API
Device shadow
Register the callback of the temperature field
When the temperature value changes, custom control logic can be executed through the callback
SDK
Air conditioner


Smart home appliance management backend
Call the UpdateDeviceShadow API to get the device shadow information (including power consumption)
IoT Hub platform
TencentCloud API
Device shadow
Timed reporting
Report its energy consumption through the qcloud_shadow_update API



注意：
设备影子和设备消息的适用场景并不一样。从实现机制上来说，服务器端设备影子总是保存最后一份数据，而先后达到的多条消息并不会相互覆盖。

- 对于设备上报数据的场景，设备影子更适用于上报一些设备自身信息（如设备能耗），设备消息更适用于上报设备收集的数据（如测量的温度）
- 对于设备接收数据的场景，设备影子更适用于通知设备更新配置（如更改目标运行温度），设备消息更适用于设备的实时控制（如让设备向左转45度）

详情请参见 设备影子详情。

7. 规则引擎

基于规则引擎，用户可以配置规则实现以下操作：

- **语法规则**
  支持类 SQL 语法和基础语义操作，可以通过简易的语法编写，实现对设备消息的内容解析和过滤提取、重新整合，进而转发到后端服务，无缝对接腾讯云后端的多种存储组件、函数计算、大数据分析套件等。
- **设备与设备互通**
  为了实现设备的数据隔离，设备只能发布和订阅自身的 Topic 消息（请参见 权限管理）。为了实现互通，需要基于规则引擎的 repub 功能。
- **设备与用户服务器互通**
  规则提供简单的 forward 功能，可以将消息通过 HTTP 请求抄送给用户服务器。实现设备消息与用户服务的快速互通能力。
- **设备与云服务互通**
  对于用户需要对设备数据进行进一步处理的场景（如持久化存储、大数据分析），腾讯云提供相应的产品（如云数据库、大数据分析套件）。
  详情请参见 规则引擎详情。

8. 消息队列

作为设备的唯一接口，物联网通信平台支持将设备指定消息写入腾讯云 CMQ、CKafka 消息队列，第三方服务可通过 CMQ、CKafka 的 SDK 接口获取设备消息，从而打通实现与设备的异步消息通信。在此基础上完成后端的数据存储、计算分析或设备控制逻辑。

9. 控制台

控制台提供了可视化的管理界面，支持产品管理、设备管理、权限管理、规则引擎配置等功能。您可以前往 物联网通信控制台 进行体验。

10. 云 API

对于物联场景下对设备的管理流接口，提供后台快速、批量操作接口。当前支持 Python、PHP、Java、Go、NodeJS、.Net 工具包。目前腾讯物联网通信提供产品、设备、任务、消息、规则引擎、设备影子相关的 API，详细请参见云 API 概览。

11. 固件升级

支持 OTA 固件升级服务，当设备固件有安全隐患或者功能漏洞时，物联网服务端支持通过 OTA 升级，消除隐患，降低安全风险。

12. 协作管理

物联网通信平台支持通过 CAM 安全地访问、使用和管理云账号的资源。通过对子账号与协作者的身份管理和策略管理，来实现物联网通信资源的隔离与协作。

签名方法

最近更新时间：2021-08-20 16:27:32

概述

设备向平台发起 `HTTP/HTTPS` 请求时，请求报文中需包含签名信息（X-TC-Signature）以验证请求者身份。

签名步骤

设备请求报文示例：

```
curl -X POST https://ap-guangzhou.gateway.tencentdevices.com/device/register \
-H "Content-Type: application/json; charset=utf-8" \
-H "X-TC-Algorithm: hmacsha256" \
-H "X-TC-Timestamp: 155****065" \
-H "X-TC-Nonce: 5456" \
-H "X-TC-Signature: 2230eefd229f582d8b1b891af****b91597240707d778ab3738f756258d7652c" \
-d '{"ProductId":"ASJ****GX","DeviceName":"xyz"}'
```

1. 拼接签名字符串

```
StringToSign =
HTTPRequestMethod + \n +
CanonicalHost + \n +
CanonicalURI + \n +
CanonicalQueryString + \n +
Algorithm + \n +
RequestTimestamp + \n +
Nonce + \n +
HashedCanonicalRequest
```

参数名称	描述
HTTPRequestMethod	HTTP 请求方式， 支持 POST
CanonicalHost	HTTP 请求的 Host 地址

参数名称	描述
CanonicalURI	HTTP 请求 URI。例如 `https://ap-guangzhou.gateway.tencentdevices.com/device/register` 的 URI 为 `/device/register`
CanonicalQueryString	发起 HTTP 请求 URL 中的查询字符串，对于 POST 请求，固定为空字符串 `""`
Algorithm	签名方法。目前支持 HmacSha256 和 HmacSha1
RequestTimestamp	请求时间戳
Nonce	随机数
HashedCanonicalRequest	请求正文 Hash 值。HTTP 请求正文做 SHA256 哈希，然后十六进制编码，最后编码串转换成小写字母

根据以上的规则，示例中的规范签名串如下：

```
POST
ap-guangzhou.gateway.tencentdevices.com
/device/register

hmacsha256
155****065
5456
35e9c5b0e3ae67532d3c9f17ead6c902226****b1ff7f6e89887f1398934f064
```

2. 计算签名

- 使用密钥签名，包括产品级密钥和设备级密钥，伪代码如下：

```
Signature = Base64_Encode(HMAC_SHA256(SignSecret, StringToSign))
```

参数名称	描述
SignSecret	签名密钥，如动态注册使用 ProductSecret，设备发布消息或上报日志则使用 psk
StringToSign	待签名的字符串

- 使用证书签名，伪代码如下：

```
Signature = Base64_Encode(RSA_SHA256(PrivateKey, StringToSign))
```

参数名称	描述
PrivateKey	证书私钥，如设备发布消息或上报日志则使用设备的 X509 私钥证书
StringToSign	待签名的字符串

3. 组装请求报文

根据上述得到的签名串，最终完整的请求如下：

```
POST https://ap-guangzhou.gateway.tencentdevices.com/devregister
Content-Type: application/json
Host: ap-guangzhou.gateway.tencentdevices.com
X-TC-Algorithm: HmacSha256
X-TC-Timestamp: 155****065
X-TC-Nonce: 5456
X-TC-Signature: 2230eefd229f582d8b1b891af71****1597240707d778ab3738f756258d7652c
{"ProductId":"ASJ****GX","DeviceName":"xyz"}
```

示例代码

Python3 示例代码如下。

```
import hashlib
import random
import time
import hmac
import base64

if __name__ == '__main__':
sign_format = '%s\n%s\n%s\n%s\n%s\n%d\n%d\n%s'
url_format = '%s://ap-guangzhou.gateway.tencentdevices.com/device/register'
request_format = "{\"ProductId\":\"%s\",\"DeviceName\":\"%s\"}"
device_name = 'dev***'
product_id = 'JCZ****KXS'
product_secret = 'X42fPqw********94cY5sQ1Y'

request_text = request_format % (product_id, device_name)
request_hash = hashlib.sha256(request_text.encode("utf-8")).hexdigest()

nonce = random.randrange(2147483647)
timestamp = int(time.time())
sign_content = sign_format % (
```

```
"POST", "ap-guangzhou.gateway.tencentdevices.com",
"/device/register", "", "hmacsha256", timestamp,
nonce, request_hash)
print("\nsign_content: \n" + sign_content)

sign_base64 = base64.b64encode(hmac.new(product_secret.encode("utf-8"),
sign_content.encode("utf-8"), hashlib.sha256).digest())

print("sign_base64: " + str(sign_base64))
```

设备身份认证
概述

最近更新时间：2021-08-20 16:27:32

物联网通信平台为每个创建的产品分配唯一标识 ProductID，用户可以自定义 Devicename 标识设备，用产品标识 + 设备标识 + 设备证书/密钥来验证设备的合法性。用户在创建产品时需要选择设备认证方式，在设备接入时需要根据指定的方式上报产品、设备信息与对应的密钥信息，认证通过后才能连接物联网通信平台。由于不同用户的设备端资源、安全等级要求都不同，平台提供了多种认证方案，以满足不同的使用场景。

物联网通信平台为您提供以下三种认证方案：

- 证书认证（设备级）：为每台设备分配证书 + 私钥，使用非对称加密认证接入，用户需要为每台设备烧录不同的配置信息。
- 密钥认证（设备级）：为每台设备分配设备密钥，使用对称加密认证接入，用户需要为每台设备烧录不同的配置信息。
- 动态注册认证（产品级）：为同一产品下的所有设备分配统一密钥，设备通过注册请求获取设备证书/密钥后认证接入，用户可以为同一批设备烧录相同的配置信息。

三种方案在易用性、安全性和对设备资源要求上各有优劣，您可以根据自己的业务场景综合评估选择。方案对比如下：

特性	证书认证	密钥认证	动态注册认证
设备烧录信息	ProductId、Devicename、设备证书、设备私钥	ProductId、Devicename、设备密钥	ProductId、Devicename、ProductSecret
是否需要提前创建设备	必须	必须	支持根据注册请求中携带的 Devicename 自动创建
安全性	高	一般	一般
使用限制	单产品下最多创建100万设备	单产品下最多创建100万设备	单产品下最多创建100万设备，用户可自定义通过注册请求自动创建的设备数上限
设备资源要求	较高，需要支持 TLS	较低	较低，支持 AES 即可

设备级密钥认证

最近更新时间：2021-11-10 11:02:21

操作场景

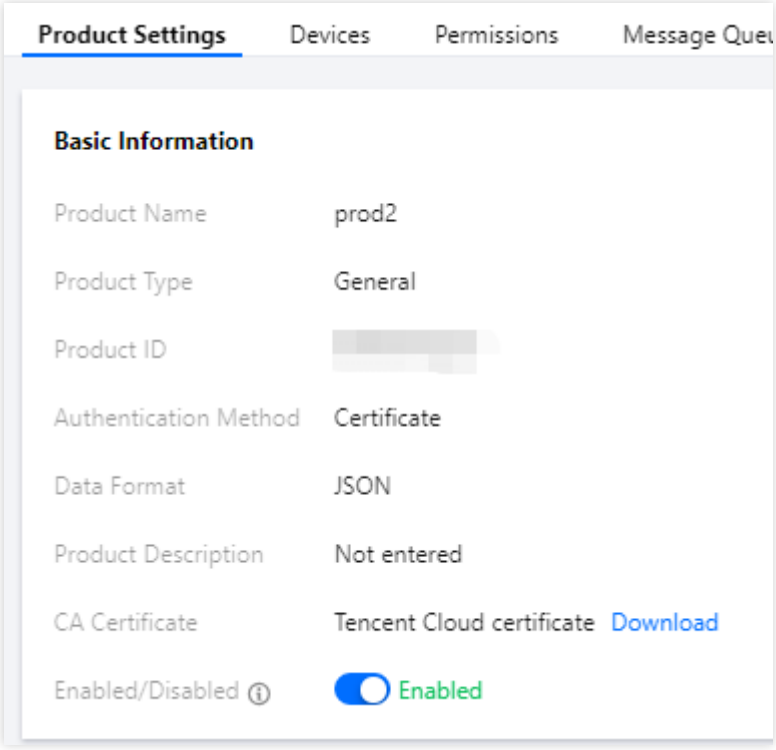
物联网通信平台支持设备级密钥认证，在该模式下，用户需要每个设备烧录不同的配置固件，平台将根据用户选择的具体认证方式（证书认证/密钥认证）进行鉴权验证，成功通过后即可与平台建立连接，进行数据通信。

流程图

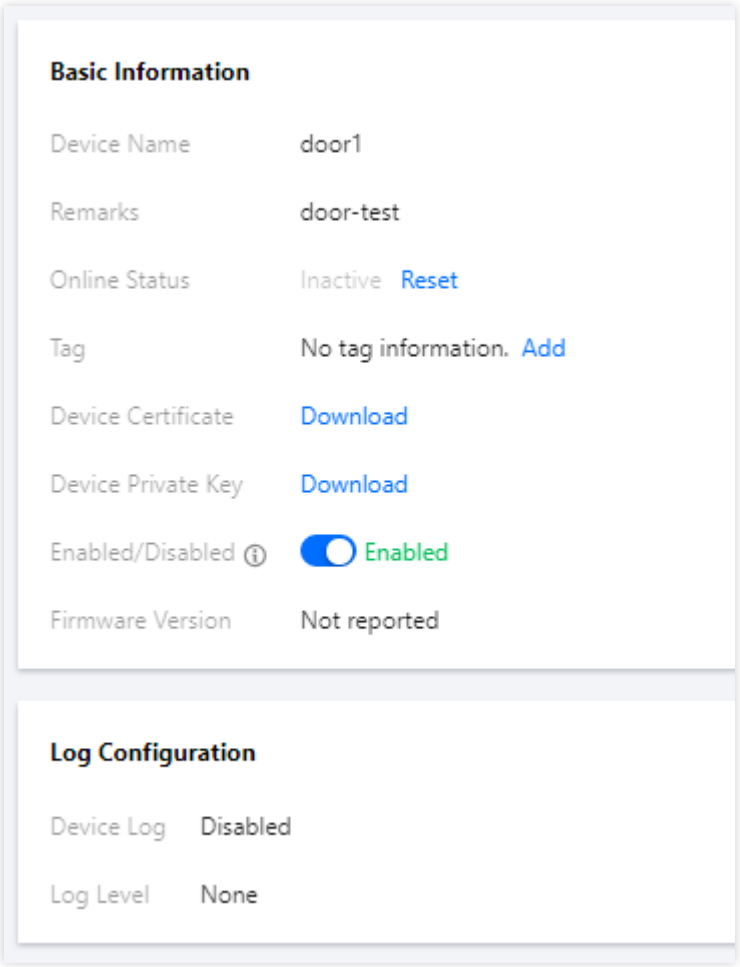
设备级密钥需要为每个设备烧录不同的固件，在产线应用中有一定实现成本，但安全性更高，推荐使用。

操作步骤

1. 登录 物联网通信控制台，创建产品与设备。具体创建步骤请参考 设备接入准备。
2. 在产品详情页获取产品信息，在设备详情页获取设备名称、设备证书/密钥。
   - 产品信息



- 设备信息



3. 设备固件烧录，具体步骤如下：
   i. 下载 设备端 SDK。
   ii. 实现 SDK 中关于产品、设备信息读写的 HAL 层函数，包括 ProductID、Devicename 与设备证书或密钥，具体可参考 设备接入。
   iii. 根据实际业务需求基于 SDK 开发设备端固件，实现设备唯一标识的读取、设备动态注册、鉴权接入、通信及 OTA 等功能。
   iv. 在生产环节，将开发测试完成的设备端固件批量烧录至设备中。
4. 设备使用烧录的设备级证书/密钥与平台发起连接，鉴权通过后完成设备激活上线，即可与云端进行数据交互，实现业务需求。

产品级密钥认证

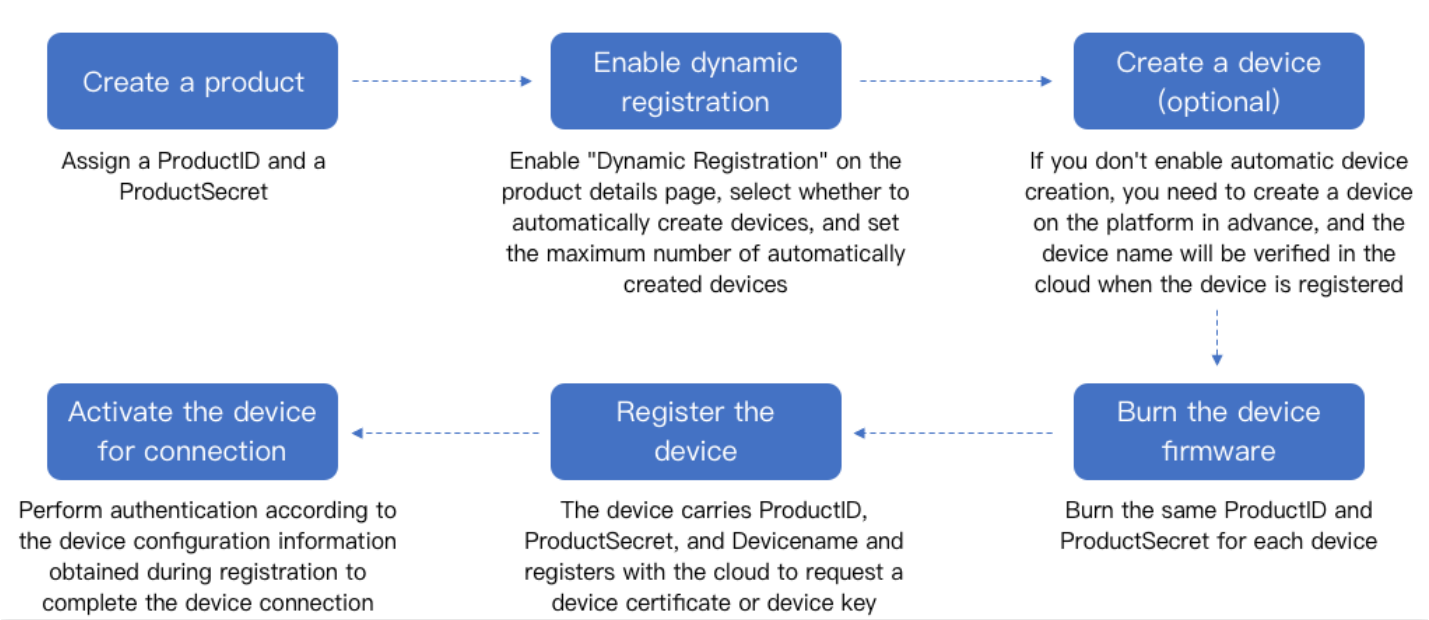
最近更新时间：2021-11-10 11:04:33

操作场景

物联网通信平台支持产品级密钥认证，在该模式下，用户只需要开启设备动态注册开关，就可以为同一产品下的所有设备烧录相同的配置固件（ProductID + ProductSecret），通过注册请求获取设备证书或密钥，再进行与平台的连接通信。

流程图

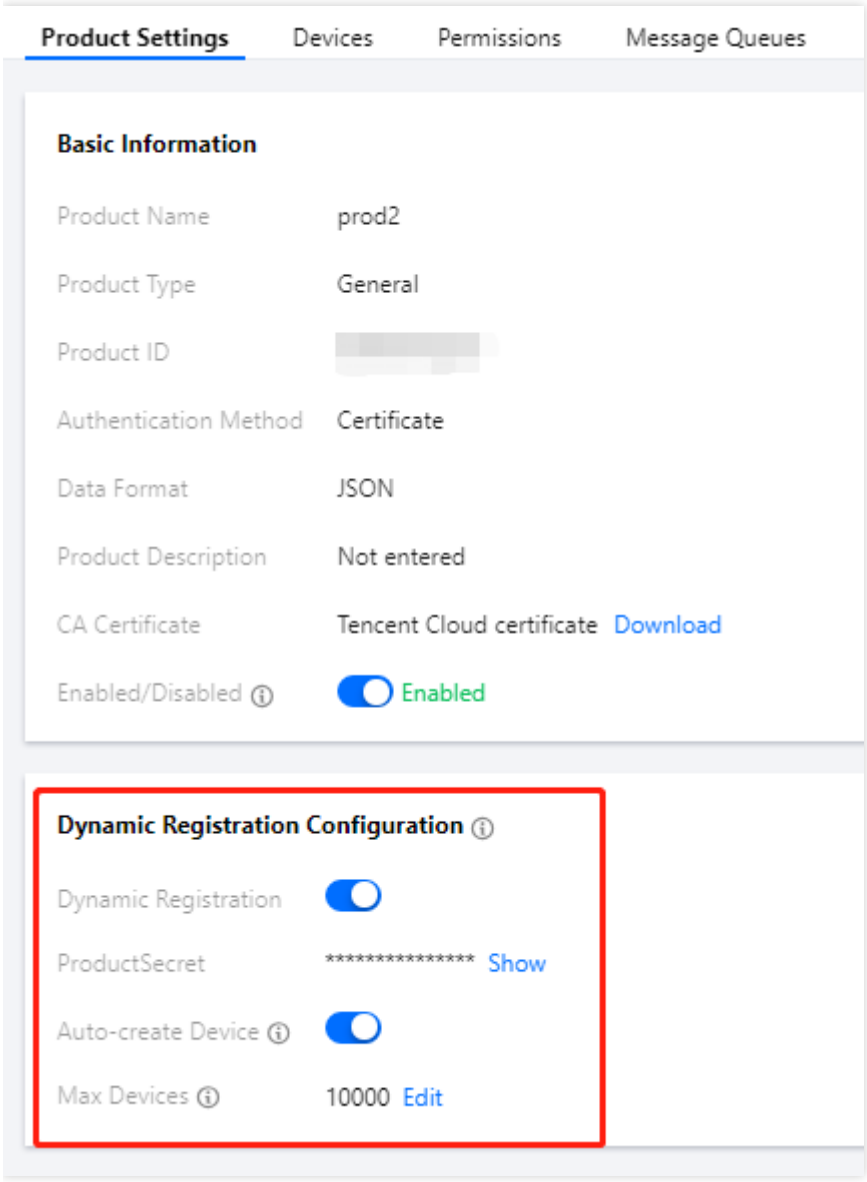
注意：
若需要使用动态注册功能，需要先在控制台产品详情页，手动开启该产品的动态注册功能。



操作步骤

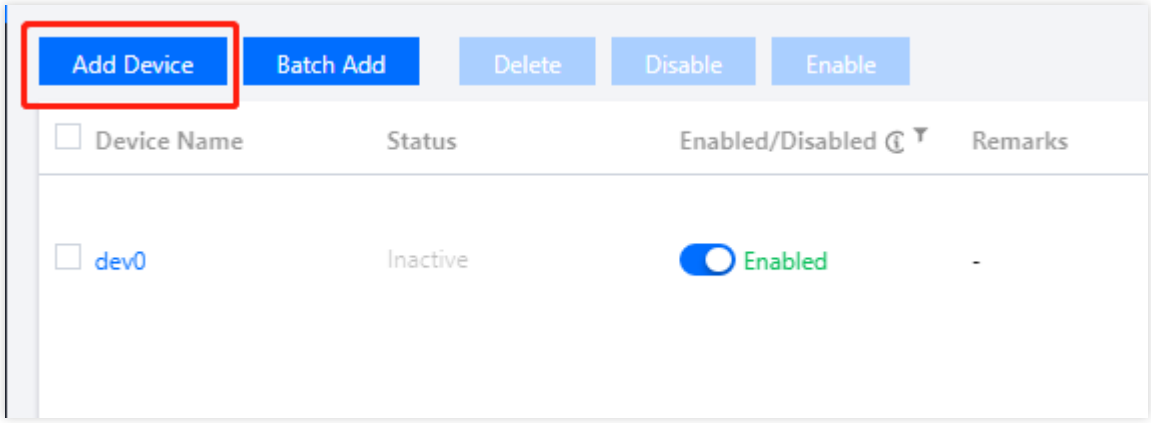
1. 登录 物联网通信控制台，创建产品，具体操作请参考 设备接入准备。
2. 在产品详情页开启动态注册开关，选择是否自动创建设备，并设置自动创建的设备上限。

> 说明：
> 为了避免不可预知情况（如设备固件 Bug，产品密钥被盗取等情况）造成创建过多设备，若您勾选了自动创建设备，建议设置合理的设备上限。



3. 在产品下创建设备（可选操作）

- 用户可以在控制台设备列表添加设备，或通过云 API 创建设备。
- 若不开启自动创建设备，则云端会在设备注册时校验每一个请求的设备名是否已在云端创建完成，建议您采用设备端可读取的唯一标识作为 Devicename，如设备的 IMEI 号、SN 号、MAC 地址等，便于整个流程的顺利完成。



iv. 设备固件烧录，具体步骤如下：

1. 下载 设备端 SDK。
2. 实现 SDK 中关于产品、设备信息读写的 HAL 层函数，包括 ProductID、ProductSecret 和 Devicename 等，并 SDK 中的开启动态注册功能，具体可参见 设备接入。
3. 根据实际业务需求基于 SDK 开发设备端固件，实现设备唯一标识的读取、设备动态注册、鉴权接入、通信及 OTA 等功能。
4. 在生产环节，将开发测试完成的设备端固件批量烧录至设备中。

v. 设备注册，设备上电联网后，发起注册请求获取设备证书或密钥。

vi. 设备使用获取的设备级证书/密钥与平台发起连接，鉴权通过后完成设备激活上线，即可与云端进行数据交互，实现业务需求。

动态注册接口说明

最近更新时间：2021-08-20 16:27:32

参数说明

设备动态注册时需携带 ProductId 和 DeviceName 向平台发起 `http/https` 请求，请求接口及参数如下：

- 请求的 URL 为：
  `https://ap-guangzhou.gateway.tencentdevices.com/device/register`
  `http://ap-guangzhou.gateway.tencentdevices.com/device/register`
- 请求方式：Post

请求参数

参数名称	必选	类型	描述
ProductId	是	string	产品 Id
DeviceName	是	string	设备名称

> 说明：
> 接口只支持 application/json 格式。

签名生成

使用 HMAC-sha256 算法对请求报文进行签名，详情请参见 签名方法。

平台返回参数

参数名称	类型	描述
RequestId	String	请求 Id
Len	Int64	返回的 Payload 长度
Payload	String	返回的设备注册信息。该数据通过加密后返回，需要设备端自行解密处理

说明：
加密过程是将原始 json 格式的 Payload 转为字符串后进行 AES 加密，再进行 base64 加密。AES 加密算法为 CBC 模式，密钥长度128，取 productSecret 前16位，偏移量为长度16的字符“0”。

原始 Payload 内容说明：

key	value	描述
encryptionType	1	加密类型。 • 1表示证书认证 • 2表示密钥认证
psk	1239466501	设备密钥，当产品认证类型为密钥认证时有此参数
clientCert	-	设备证书文件字符串格式，当产品认证类型为证书认证时有此参数
clientKey	-	设备私钥文件字符串格式，当产品认证类型为证书认证时有此参数

示例代码

请求包

```
POST https://ap-guangzhou.gateway.tencentdevices.com/device/register
Content-Type: application/json
Host: ap-guangzhou.gateway.tencentdevices.com
X-TC-Algorithm: HmacSha256
X-TC-Timestamp: 1551****65
X-TC-Nonce: 5456
X-TC-Signature: 2230eefd229f582d8b1b891af7107b91597****07d778ab3738f756258d7652c
{"ProductId":"ASJ****GX","DeviceName":"xyz"}
```

返回包

```
{
"Response": {
"Len": 53,
"Payload": "031T01DWAoqFePDt71VuZXuLzkUzbIhGOnvMzpAFtNgOjagyFNHVSostNl9ztvhOuRx0
dMM/DMoWAXQCfL7jyA==",
"RequestId": "f4da4f1f-d72e-40f1-****-349fc0072ba0"
}
}
```

Payload 数据解析示例

> 说明：
> 以下数据仅提供您进行测试使用，在您正式使用时，请务必保证您的信息不被泄露。

1. Payload 原始内容为

```
s6FB3a1BA/YYbcmSE12XpeDVmQNDcf1QgVD141RRbmmAnFwQfp1ECAu5O016mCOvYlJJ6V59yM4OqQSiWphfTg==
```

2. Base64 解码后

```
b3a141ddad4103f6186dc992135d97a5e0d599034371fd508150f5e354516e69809c5c107e9d44080bb93b4d7a9823af625249e95e7dc8ce0ea904a25a985f4e
```

3. AES 解密
   - 产品密钥： `hzvf5LF9S0isvBhDSauWMaIk`
   - 解密后数据： `{"encryptionType":2,"psk":"lDZ6Uqt+I9E0wW7rvDUs7Q=="}`

设备接入协议
设备基于MQTT接入
设备基于 TCP 的 MQTT 接入

最近更新时间：2021-08-20 16:34:45

MQTT 协议说明

目前物联网通信支持 MQTT 标准协议接入(兼容3.1.1版本协议)，具体的协议请参见 MQTT 3.1.1 协议文档。

和标准 MQTT 区别

1. 支持 MQTT 的 PUB、SUB、PING、PONG、CONNECT、DISCONNECT、UNSUB 等报文。
2. 支持 cleanSession。
3. 不支持 will、retain msg。
4. 不支持 QOS2。

MQTT 通道，安全等级

支持 TLSV1，TLSV1.1，TLSV1.2 版本的协议来建立安全连接，安全级别高。

TOPIC 规范

默认情况下创建产品后，该产品下的所有设备都拥有以下 topic 类的权限：

1. `${productId}/${deviceName}/control` 订阅。
2. `${productId}/${deviceName}/event` 发布。
3. `${productId}/${deviceName}/data` 订阅和发布。
4. `$shadow/operation/${productId}/${deviceName}` 发布。通过包体内部 type 来区分：update/get，分别对应设备影子文档的更新和拉取等操作。
5. `$shadow/operation/result/${productId}/${deviceName}` 订阅。通过包体内部 type 来区分：update/get/delta，type 为 update/get 分别对应设备影子文档的更新和拉取等操作的结果；当用户通过 restAPI 修改设备影子文档后，服务端将通过该 topic 发布消息，其中 type 为 delta。
6. `$ota/report/${productID}/${deviceName}` 发布。设备上报版本号及下载、升级进度到云端。
7. `$ota/update/${productID}/${deviceName}` 订阅。设备接收云端的升级消息。

MQTT 接入

MQTT 协议支持通过设备证书和密钥签名两种方式接入物联网通信平台，您可根据自己的场景选择一种方式接入即可。接入参数如下所示：

接入认证方式	连接域名及端口	Connect报文参数
证书认证	MQTT 服务器连接地址，广州域设备填入：${productId}.iotcloud.tencentdevices.com，这里 ${productId}为变量参数，用户需填入创建产品时自动生成的产品 ID，例如 1A17RZR3XX.iotcloud.tencentdevices.com；端口：8883	• KeepAlive：保持连接的时间，取值范围为0 仍没收到客户端的数据，则平台将断开与客户端 • ClientId：${productId}${deviceName}，产品 • UserName：`${productId}${deviceName` 详情见下文中基于 MQTT 的签名认证接入指引 • PassWord：密码（可赋任意值）。
密钥认证	MQTT 服务器连接地址与证书认证一致；端口：1883	• KeepAlive：保持连接的时间，取值范围为0 • ClientId：${productId}${deviceName}； • UserName：`${productId}${deviceName` 详情见下文中基于 MQTT 的签名认证接入指引 • PassWord：密码，详情见下文中基于 MQT

说明：
采用证书认证的设备接入时不会对填写的 PassWord 部分进行验证，证书认证时 PassWord 部分可填写任意值。

证书认证设备接入指引

物联网平台采用 TLS 加密方式来保障设备传输数据时的安全性。证书设备接入时，获取到证书设备的证书、密钥与 CA 证书文件之后，设置好 KeepAlive，ClientId，UserName，PassWord 等内容（采用腾讯云设备端 SDK 方式接入的设备无需设置，SDK 可根据设备信息自动生成）。设备向证书认证对应的 URL（连接域名及端口）上传认证文件，通过之后发送 MqttConnect 消息即可完成证书设备基于 TCP 的 MQTT 接入。

密钥认证设备接入指引

物联网平台支持 HMAC-SHA256，HMAC-SHA1 等方式基于设备密钥生成摘要签名。通过签名方式接入物联云平台的流程如下：

1. 登录 物联网通信控制台。您可在控制台创建产品、添加设备、并获取设备密钥。
2. 按照物联网通信约束生成 username 字段，username 字段格式如下：

```
username 字段的格式为：
${productId}${deviceName};${sdkappid};${connid};${expiry}
注意：${} 表示变量，并非特定的拼接符号。
```

其中各字段含义如下：

- productId：产品 ID。
- deviceName： 设备名称。
- sdkappid：固定填12010126。
- connid ：一个随机字符串。
- expiry ：表示签名的有效期， 从1970年1月1日00:00:00 UTC 时间至今秒数的 UTF8 字符串。

3. 用 base64 对设备密钥进行解码得到原始密钥 raw_key。
4. 用第3步生成的 raw_key，通过 HMAC-SHA1 或者 HMAC-SHA256 算法对 username 生成一串摘要，简称 Token。
5. 按照物联网通信约束生成 password 字段，password 字段格式为：

```
password 字段格式为：
${token};hmac 签名方法
其中 hmac 签名方法字段填写第三步用到的摘要算法，可选的值有 hmacsha256 和 hmacsha1。
```

作为对照，用户生成签名的 Python、Java、Nodejs、JavaScript 和 C 代码示例如下：

Python 代码为：

```python
#!/usr/bin/python
# -*- coding: UTF-8 -*-
import base64
import hashlib
import hmac
import random
import string
import time
import sys
# 生成指定长度的随机字符串
def RandomConnid(length):
return ''.join(random.choice(string.ascii_uppercase + string.digits) for _ in ra
nge(length))
# 生成接入物联网通信平台需要的各参数
def IotHmac(productID, devicename, devicePsk):
# 1. 生成 connid 为一个随机字符串，方便后台定位问题
connid = RandomConnid(5)
# 2. 生成过期时间，表示签名的过期时间,从纪元1970年1月1日 00:00:00 UTC 时间至今秒数的 UTF8
字符串
expiry = int(time.time()) + 60 * 60
# 3. 生成 MQTT 的 clientid 部分，格式为 ${productid}${devicename}
clientid = "{}{}".format(productID, devicename)
# 4. 生成 MQTT 的 username 部分，格式为 ${clientid};${sdkappid};${connid};${expiry}
username = "{};12010126;{};{}".format(clientid, connid, expiry)
# 5. 对 username 进行签名，生成token
```

```
secret_key = devicePsk.encode('utf-8') # convert to bytes
data_to_sign = username.encode('utf-8') # convert to bytes
secret_key = base64.b64decode(secret_key) # this is still bytes
token = hmac.new(secret_key, data_to_sign, digestmod=hashlib.sha256).hexdigest()
# 6. 根据物联网通信平台规则生成 password 字段
password = "{};{}".format(token, "hmacsha256")
return {
"clientid" : clientid,
"username" : username,
"password" : password
}
if __name__ == '__main__':
print(IotHmac(sys.argv[1], sys.argv[2], sys.argv[3]))
```

将上述代码保存到 IotHmac.py，执行下面的命令即可。这里 "YOUR_PRODUCTID"、 "YOUR_DEVICENAME" 和"YOUR_PSK" 是填写您实际创建设备的产品 ID、设备名称和设备密钥。

```
python3 IotHmac.py "YOUR_PRODUCTID" "YOUR_DEVICENAME" "YOUR_PSK"
```

Java 代码为：

```
import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;
import java.util.*;
public class IotHmac {
public static void main(String[] args) throws Exception {
System.out.println(IotHmac("YOUR_PRODUCTID","YOUR_DEVICENAME","YOUR_PSK"));
}
public static Map<string, string=""> IotHmac(String productID, String devicenam
e, String
devicePsk) throws Exception {
final Base64.Decoder decoder = Base64.getDecoder();
//1. 生成 connid 为一个随机字符串，方便后台定位问题
String connid = HMACSHA256.getRandomString2(5);
//2. 生成过期时间，表示签名的过期时间,从纪元1970年1月1日 00:00:00 UTC 时间至今秒数的 UTF8
字符串
Long expiry = Calendar.getInstance().getTimeInMillis()/1000 +600;
//3. 生成 MQTT 的 clientid 部分，格式为 ${productid}${devicename}
String clientid = productID+devicename;
//4. 生成 MQTT 的 username 部分，格式为 ${clientid};${sdkappid};${connid};${expiry}
String username = clientid+";"+"12010126;"+connid+";"+expiry;
//5. 对 username 进行签名，生成token、根据物联网通信平台规则生成 password 字段
String password = HMACSHA256.getSignature(username.getBytes(), decoder.decode(de
vicePsk)) + ";hmacsha256";
Map<string,string> map = new HashMap&lt;&gt;();
map.put("clientid",clientid);
map.put("username",username);
```

```
map.put("password",password);
return map;
}
public static class HMACSHA256 {
private static final String HMAC_SHA256 = "HmacSHA256";
/**
* 生成签名数据
*
* @param data 待加密的数据
* @param key 加密使用的key
* @return 生成16进制编码的字符串
*/
public static String getSignature(byte[] data, byte[] key) {
try {
SecretKeySpec signingKey = new SecretKeySpec(key, HMAC_SHA256);
Mac mac = Mac.getInstance(HMAC_SHA256);
mac.init(signingKey);
byte[] rawHmac = mac.doFinal(data);
return bytesToHexString(rawHmac);
}catch (Exception e) {
e.printStackTrace();
}
return null;
}
/**
* byte[]数组转换为16进制的字符串
*
* @param bytes 要转换的字节数组
* @return 转换后的结果
*/
private static String bytesToHexString(byte[] bytes) {
StringBuilder sb = new StringBuilder();
for (int i = 0; i &lt; bytes.length; i++) {
String hex = Integer.toHexString(0xFF &amp; bytes[i]);
if (hex.length() == 1) {
sb.append('0');
}
sb.append(hex);
}
return sb.toString();
}
public static String getRandomString2(int length) {
Random random = new Random();
StringBuffer sb = new StringBuffer();
for (int i = 0; i &lt; length; i++) {
int number = random.nextInt(3);
long result = 0;
```

```
switch (number) {
case 0:
result = Math.round(Math.random() * 25 + 65);
sb.append(String.valueOf((char) result));
break;
case 1:
result = Math.round(Math.random() * 25 + 97);
sb.append(String.valueOf((char) result));
break;
case 2:
sb.append(String.valueOf(new Random().nextInt(10)));
break;
}
}
return sb.toString();
}
}
}
```

Nodejs 和 JavaScript 代码为：

```
// 下面为node引入方式，浏览器的话，使用对应的方式引入crypto-js库
const crypto = require('crypto-js')

// 产生随机数的函数
const randomString = (len) =&gt; {
    len = len || 32;
    var chars = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789';
    var maxPos = chars.length;
    var pwd = '';
    for (let i = 0; i &lt; len; i++) {
        pwd += chars.charAt(Math.floor(Math.random() * maxPos));
    }
    return pwd;
}
// 需要产品id，设备名和设备密钥
const productId = 'YOUR_PRODUCTID';
const deviceName = 'YOUR_DEVICENAME';
const devicePsk = 'YOUR_PSK';

// 1. 生成 connid 为一个随机字符串，方便后台定位问题
const connid = randomString(5);
// 2. 生成过期时间，表示签名的过期时间,从纪元1970年1月1日 00:00:00 UTC 时间至今秒数的 UTF8
字符串
const expiry = Math.round(new Date().getTime() / 1000) + 3600 * 24;
// 3. 生成 MQTT 的 clientid 部分，格式为 ${productid}${devicename}
```

```
const clientId = productId + deviceName;
// 4. 生成 MQTT 的 username 部分，格式为 ${clientid};${sdkappid};${connid};${expir
y}
const userName = `${clientId};12010126;${connid};${expiry}`;
//5. 对 username 进行签名，生成token、根据物联网通信平台规则生成 password 字段
const rawKey = crypto.enc.Base64.parse(devicePsk); // 对设备密钥进行base64解码
const token = crypto.HmacSHA256(userName, rawKey);
const password = token.toString(crypto.enc.Hex) + ";hmacsha256";
console.log(`userName:${userName}\npassword:${password}`);
```

C 语言代码如下：

> 说明：
> 如果您想要了解更多关于 C 语言代码内容，详情请参见 工程下载。

```
#include "limits.h"
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

#include "HAL_Platform.h"
#include "utils_base64.h"
#include "utils_hmac.h"

/* Max size of base64 encoded PSK = 64, after decode: 64/4*3 = 48*/
#define DECODE_PSK_LENGTH 48

/* MAX valid time when connect to MQTT server. 0: always valid */
/* Use this only if the device has accurate UTC time. Otherwise, set to 0 */
#define MAX_ACCESS_EXPIRE_TIMEOUT (0)

/* Max size of conn Id */
#define MAX_CONN_ID_LEN (6)

/* IoT C-SDK APPID */
#define QCLOUD_IOT_DEVICE_SDK_APPID "21****06"
#define QCLOUD_IOT_DEVICE_SDK_APPID_LEN (sizeof(QCLOUD_IOT_DEVICE_SDK_APPID) -
1)

static void HexDump(char *pData, uint16_t len)
{
int i;
```

```
for (i = 0; i &lt; len; i++) {
HAL_Printf("0x%02.2x ", (unsigned char)pData[i]);
}
HAL_Printf("\n");
}

static void get_next_conn_id(char *conn_id)
{
int i;
srand((unsigned)HAL_GetTimeMs());
for (i = 0; i &lt; MAX_CONN_ID_LEN - 1; i++) {
int flag = rand() % 3;
switch (flag) {
case 0:
conn_id[i] = (rand() % 26) + 'a';
break;
case 1:
conn_id[i] = (rand() % 26) + 'A';
break;
case 2:
conn_id[i] = (rand() % 10) + '0';
break;
}
}

conn_id[MAX_CONN_ID_LEN - 1] = '\0';
}

int main(int argc, char **argv)
{
char *product_id = NULL;
char *device_name = NULL;
char *device_secret = NULL;

char *username = NULL;
int username_len = 0;
char conn_id[MAX_CONN_ID_LEN];

char password[51] = {0};
char username_sign[41] = {0};

char psk_base64decode[DECODE_PSK_LENGTH];
size_t psk_base64decode_len = 0;

long cur_timestamp = 0;

if (argc != 4) {
```

```
HAL_Printf("please ./qcloud-mqtt-sign product_id device_name device_secret\r\n")
;
return -1;
}

product_id = argv[1];
device_name = argv[2];
device_secret = argv[3];

/* first device_secret base64 decode */
qcloud_iot_utils_base64decode((unsigned char *)psk_base64decode, DECODE_PSK_LENG
TH, &amp;psk_base64decode_len,
(unsigned char *)device_secret, strlen(device_secret));
HAL_Printf("device_secret base64 decode:");
HexDump(psk_base64decode, psk_base64decode_len);

/* second create mqtt username
* [productdevicename;appid;randomconnid;timestamp] */
cur_timestamp = HAL_Timer_current_sec() + MAX_ACCESS_EXPIRE_TIMEOUT / 1000;
if (cur_timestamp &lt;= 0 || MAX_ACCESS_EXPIRE_TIMEOUT &lt;= 0) {
cur_timestamp = LONG_MAX;
}

// 20 for timestampe length &amp; delimiter
username_len = strlen(product_id) + strlen(device_name) + QCLOUD_IOT_DEVICE_SDK_
APPID_LEN + MAX_CONN_ID_LEN + 20;
username = (char *)HAL_Malloc(username_len);
if (username == NULL) {
HAL_Printf("malloc username failed!\r\n");
return -1;
}

get_next_conn_id(conn_id);
HAL_Snprintf(username, username_len, "%s%s;%s;%s;%ld", product_id, device_name,
QCLOUD_IOT_DEVICE_SDK_APPID,
conn_id, cur_timestamp);

/* third use psk_base64decode hamc_sha1 calc mqtt username sign crate mqtt
* password */
utils_hmac_sha1(username, strlen(username), username_sign, psk_base64decode, psk
_base64decode_len);
HAL_Printf("username sign: %s\r\n", username_sign);
HAL_Snprintf(password, 51, "%s;hmacsha1", username_sign);

HAL_Printf("Client ID: %s%s\r\n", product_id, device_name);
HAL_Printf("username : %s\r\n", username);
HAL_Printf("password : %s\r\n", password);
```

```
HAL_Free(username);

return 0;
}
```

6. 最终将上面生成的参数填入对应的 MQTT connect 报文中。
7. 将 clientid 填入到 MQTT 协议的 clientid 字段。
8. 将 username 填入到 MQTT 的 username 字段。
9. 将 password 填入到 MQTT 的 password 字段，向密钥认证的域名与端口处发送 MqttConnect 信息即可接入到物联云通信平台。

设备基于 WebSocket 的 MQTT 接入

最近更新时间：2021-08-20 16:36:10

MQTT-WebSocket 概述

物联网平台支持基于 WebSocket 的 MQTT 通信，设备可以在 WebSocket 协议的基础之上使用 MQTT 协议进行消息的传输。从而使基于浏览器的应用可以实现与平台及与平台连接的设备之间的数据通信。同时 WebSocket 采用443/80端口，消息传输时可以穿过大多数防火墙。

MQTT-WebSocket 接入

由于 MQTT-WebSocket 协议与 MQTT-TCP 协议最终都是基于 MQTT 进行消息的传输，所以这两种协议在 MQTT 接入参数上是相同的，区别主要在于 MQTT 连接平台的协议及端口。密钥认证的设备采用 WS 的方式进行接入，证书认证的设备采用 WSS 的方式接入，即 WS+TLS。

证书认证设备接入指引

1. 下载证书、设备私钥等文件。
2. 连接域名：广州域设备需连接，`${ProductId}.ap-guangzhou.iothub.tencentdevices.com:443`，其中 ${ProductId} 为变量参数产品 ID。
3. MQTT 连接参数设置：
   连接参数设置与 MQTT-TCP 接入时一致，具体信息请参见 设备基于 TCP 的 MQTT 接入 文档中的 MQTT 接入章节。

```
UserName:${productid}${devicename};${sdkappid};${connid};${expiry}
PassWord:密码。（可设置任意值）
ClientId:${ProductId}${DeviceName}
KeepAlive:保持连接的时间，取值范围为0 - 900s
```

密钥认证设备接入指引

1. 获取设备密钥。
2. 连接域名：广州域设备需连接，`${ProductId}.ap-guangzhou.iothub.tencentdevices.com:80`，其中 ${ProductId} 为变量参数产品 ID。
3. MQTT 连接参数设置：
   连接参数设置与 MQTT-TCP 接入时一致，具体信息请参见 设备基于 TCP 的 MQTT 接入 文档中的密钥设备接入指引章节。

```
UserName:${productid}${devicename};${sdkappid};${connid};${expiry}
PassWord:${token};hmac 签名方法
ClientId:${ProductId}${DeviceName}
KeepAlive:保持连接的时间，取值范围为0 - 900s
```

MQTT 持久性会话

最近更新时间：2021-08-20 16:38:15

物联网通信支持 MQTT 协议 V3.1.1 版本，同时支持 QOS0 与 QOS1 的服务质量等级（不支持 QOS2）。使用 MQTT 持久性会话，可保存设备的订阅状态及设备未接收到的订阅消息。设备离线后再次上线时可恢复至之前的会话，并接收到离线时未接收到的订阅消息。

设备端创建 MQTT 持久性会话

设备连接物联网通信时，可将 Connect 连接报文可变报头部分的 CleanSession 标志位设置为0。物联网通信会根据设备连接时的客户端标识符 ClientId 对设备的会话状态进行判断，若当前没有会话则将会创建一个新的持久性会话，若存在已有会话则基于已有的会话进程进行通讯。

物联网通信响应说明

设备端发送 Connect 报文之后，物联网通信将会返回 Connack 报文，报文在连接确认标志位 SessionPresent 中，表明物联网通信是否已包含设备连接时的客户端标识符所对应的会话状态。SessionPresent 为0表示未创建持续性会话，设备端需要重新建立会话状态。SessionPresent 为1表明已创建持续性会话。

- 设备端成功连接之后，若进入已有的持久性会话，则物联网通信会将存储的 QOS1 消息和未确认的 QOS1 消息发送到设备端。
- 设备端成功连接之后，若创建了新的持久性会话，物联网通信将会保存设备的订阅状态，并在设备离线时将设备已订阅的 QOS1（不包含 QOS0）消息进行存储。设备再次上线时会将存储的 QOS1 消息和未确认的 QOS1 消息发送到设备端。

> 说明：
>
> - 物联网通信发送存储的 QOS1 消息时会按照500ms的间隔依次下发。
> - 可持久性会话中只存储 QOS1 消息，存储消息单设备最多150条，最多存储24小时。

关闭 MQTT 持久性会话

可通过以下两种方式关闭 MQTT 持久性会话。

- 设备连接物联网通信时，将 Connect 连接报文可变报头部分的 CleanSession 标志位设置为1。

- 设备断开连接的时间**超过24小时**，持久性会话会自动关闭。

> 说明：
> 设备的断开连接包含设备发送 disconnect 消息和设备通信超时导致的断开连接。

设备基于CoAP接入

最近更新时间：2021-08-20 16:39:48

目前物联网通信支持 CoAP 标准协议接入。 具体协议请参见 RFC7252 协议文档。

和标准 CoAP 区别

1. 目前只支持消息的上报，将 SDK 的信息上传到物联网通信。
2. 支持 POST 方法，不支持 GET/PUT/DELETE 方法。

CoAP 通道，安全等级

1. 支持 DTLS 的协议来建立安全连接，安全级别高。
2. 支持非对称加密。

接入参数

1. 服务器地址为：广州域设备填入，${ProductId}.iotcloud.tencentdevices.com，这里 ${ProductId} 为变量参数，用户需填入创建产品时自动生成的产品 ID。
2. 连接端口为：5684

URI 规范

CoAP 消息发送到 URI，URI 的格式为 `/${productId}/${deviceName}/xxx` ，productId 为在控制台注册的产品 ID，deviceName 为 productId 产品下的设备名称。

默认情况下创建产品后，该产品下的所有设备都拥有以下 Topic 类的权限：

1. `${productId}/${deviceName}/event` 发布。
2. `${productId}/${deviceName}/control` 订阅。
3. `${productId}/${deviceName}/data` 发布和订阅。

即 URI 与 MQTT Topic 相对应。

设备基于HTTP接入

最近更新时间：2021-08-20 16:40:59

参数说明

设备上报消息时需携带 ProductId、DeviceName 和 TopicName 向平台发起 `http/https` 请求，请求接口及说明参数如下：

- 请求的 URL 为：
  `https://ap-guangzhou.gateway.tencentdevices.com/device/publish`
  `http://ap-guangzhou.gateway.tencentdevices.com/device/publish`
- 请求方式：Post

请求参数

参数名称	必选	类型	描述
ProductId	是	String	产品 Id
DeviceName	是	String	设备名称
TopicName	是	String	发布消息的 Topic 名称
Payload	是	String	发布消息的内容
PayloadEncoding	否	String	发布消息的编码。目前只支持base64编码，不传默认发送原始的消息内容
Qos	是	Integer	消息 Qos 等级

> 说明：
> 接口只支持 application/json 格式。

签名生成

对请求报文进行签名分为两种，密钥认证使用 HMAC-sha256 算法，证书认证使用 RSA_SHA256 算法，详情请参见签名方法。

平台返回参数

参数名称	类型	描述
RequestId	String	请求 Id

示例代码

请求包

```
POST https://ap-guangzhou.gateway.tencentdevices.com/device/publish
Content-Type: application/json
Host: ap-guangzhou.gateway.tencentdevices.com
X-TC-Algorithm: HmacSha256
X-TC-Timestamp: 155****065
X-TC-Nonce: 5456
X-TC-Signature: 2230eefd229f582d8b1b891af7107b915972407****78ab3738f756258d7652c
{"DeviceName":"AAAAAA","Payload":"123","ProductId":"G8N****AHB","Qos":1,"TopicNam
e":"G8N****AHB/AAAAAA/data"}
```

返回包

```
{
"Response": {
"RequestId": "f4da4f1f-d72e-40f1-****-349fc0072ba0"
}
}
```

设备接入地域说明

最近更新时间：2022-02-22 11:36:58

目前 IoT Hub 物联网通信支持接入地域：

国内站	国际站
中国大陆	China Mainland
美国东部（弗吉尼亚）	U.S East (Virginia)
欧洲中部（法兰克福）	Central Europe (Frankfurt)
东南亚太（曼谷）	Southeast Asia Pacific (Bangkok)

MQTT 接入域名

设备接入时，用户可根据需求选择接入如下服务器地址：

地域	域名
中国大陆	${productid}.iotcloud.tencentdevices.com
美国东部（弗吉尼亚）	${productid}.us-east.iotcloud.tencentdevices.com
欧洲中部（法兰克福）	${productid}.europe.iothub.tencentdevices.com
东南亚太（曼谷）	${productid}.ap-bangkok.iothub.tencentdevices.com

CoAP 接入域名

设备接入时，用户可根据需求选择接入如下服务器地址：

地域	域名
中国大陆	${productid}.iotcloud.tencentdevices.com
美国东部（弗吉尼亚）	${productid}.us-east.iotcloud.tencentdevices.com
欧洲中部（法兰克福）	${productid}.europe.iothub.tencentdevices.com

地域	域名
东南亚太（曼谷）	${productid}.ap-bangkok.iothub.tencentdevices.com

HTTP 接入域名

设备接入时，用户可根据需求选择接入如下服务器地址：

地域	域名
中国大陆	ap-guangzhou.gateway.tencentdevices.com
美国东部（弗吉尼亚）	us-east.gateway.tencentdevices.com
欧洲中部（法兰克福）	europe.gateway.tencentdevices.com
东南亚太（曼谷）	ap-bangkok.gateway.tencentdevices.com

网关子设备
功能概述

最近更新时间：2021-09-10 10:44:38

设备分类

物联网通信平台根据设备功能性的不同将设备分为如下两类（即节点的分类）：

- **普通设备**：此类设备分为两种，一种为设备本身具备接入平台能力的设备，另一种为需借助网关设备接入平台的设备。
- **网关设备**：此类设备可直接接入物联网平台，并且可接受子设备加入局域网络。

操作场景

对于不具备直接接入以太网网络的设备，可先接入本地网关设备的网络，利用网关设备的通信功能，代理设备接入物联网通信平台。对于局域网中加入或退出网络的子设备，网关设备可对其在平台进行绑定或解绑操作，并上报与子设备的拓扑关系，实现平台对于整个局域网设备的实时监控。

接入方式

- 网关设备接入物联网通信平台的方式与普通设备一致，具体可参见 设备接入。网关设备接入之后可代理同一局域网下的子设备上/下线，管理与子设备之间的拓扑关系。
- 子设备的接入需通过网关设备来完成，子设备通过网关设备完成身份的认证之后即可成功接入云端。认证方式分为以下两种：
  - **设备级密钥方式**
    网关获取子设备的设备证书或密钥，并生成子设备绑定签名串。由网关向平台上报子设备绑定签名串信息，代理子设备完成身份的验证。
  - **产品级密钥方式**
    网关获取子设备的 ProductKey（产品密钥），并生成签名，由网关向平台发送动态注册请求。验证成功之后平台将返回子设备的 DeviceCert 或 DeviceSecret，网关设备依此生成子设备绑定签名串，并向平台上报子设备绑定签名串信息。验证成功之后即完成子设备的接入。

拓扑关系管理

最近更新时间：2021-09-10 10:44:38

功能概述

网关类型的设备，可通过与云端的数据通信，对其下的子设备进行绑定与解绑操作。实现此类功能需利用如下两个 Topic：

- 数据上行 Topic（用于发布）：`$gateway/operation/${productid}/${devicename}`
- 数据下行 Topic（用于订阅）：`$gateway/operation/result/${productid}/${devicename}`

绑定设备

网关类型的设备，可以通过数据上行 Topic 请求添加它和子设备之间的拓扑关系，实现绑定子设备。请求成功之后，云端通过数据下行 Topic 返回子设备的绑定信息。

网关绑定子设备请求数据格式：

```
{
"type": "bind",
"payload": {
"devices": [
{
"product_id": "CFCS****G7",
"device_name": "****ev",
"signature": "signature",
"random": 121213,
"timestamp": 1589786839,
"signmethod": "hmacsha256",
"authtype": "psk"
}
]
}
}
```

请求参数说明：

参数	类型	描述
type	String	网关消息类型。绑定子设备取值为：`bind`
payload.devices	Array	需要绑定的子设备列表
product_id	String	子设备产品 ID

参数	类型	描述
device_name	String	子设备名称
signature	String	子设备绑定签名串。 签名算法： 1. 签名原串，将产品 ID 设备名称，随机数，时间戳拼接： `text=${product_id}${device_name};${random};${expiration_t` 2. 使用设备 Psk 密钥，或者证书的 Sha1 摘要，进行签名： `sign = hmac_sha1(device_secret, text)`
random	Int	随机数
timestamp	Int	时间戳，单位：秒
signmethod	String	签名算法。支持 hmacsha1、hmacsha256
authtype	String	签名类型。 • psk：使用设备 psk 进行签名 • certificate：使用设备公钥证书签名

网关绑定子设备响应数据格式：

```
{
"type": "bind",
"payload": {
"devices": [
{
"product_id": "CFCS****G7",
"device_name": "****ev",
"result": -1
}
]
}
}
```

响应参数说明：

参数	类型	描述
type	String	网关消息类型。绑定子设备取值为： `bind`
payload.devices	Array	要绑定的子设备列表
product_id	String	子设备产品 ID
device_name	String	子设备名称

参数	类型	描述
result	Int	子设备绑定结果，具体错误码见下表

解绑设备

网关类型的设备， 可以通过数据上行 Topic 请求解绑它和子设备之间的拓扑关系。请求成功之后，云端通过数据下行 Topic 返回子设备的解绑信息。

网关解绑子设备请求数据格式：

```
{
"type": "unbind",
"payload": {
"devices": [
{
"product_id": "CFCS****G7",
"device_name": "****ev"
}
]
}
}
```

请求参数说明：

参数	类型	描述
type	String	网关消息类型。解绑子设备取值为：`unbind`
payload.devices	Array	需要解绑的子设备列表
product_id	String	子设备产品 ID
device_name	String	子设备名称

网关解绑子设备响应数据格式：

```
{
"type": "bind",
"payload": {
"devices": [
{
"product_id": "CFCS****G7",
"device_name": "****ev",
"result": -1
```

```
}
]
}
}
```

响应参数说明：

参数	类型	描述
type	String	网关消息类型。解绑子设备取值为：`unbind`
payload.devices	Array	需要解绑的子设备列表
product_id	String	子设备产品 ID
device_name	String	子设备名称
result	Int	子设备绑定结果，详见 错误码

查询拓扑关系

网关类型的设备， 可以通过该Topic上行请求查询子设备的拓扑关系。

数据上行Topic：`$gateway/operation/${productid}/${devicename}`

数据下行Topic：`$gateway/operation/result/${productid}/${devicename}`

网关查询子设备拓扑关系请求数据格式：

```
{
"type": "describe_sub_devices"
}
```

请求参数说明：

参数	类型	描述
type	String	网关消息类型。查询子设备取值为：`describe_sub_devices`

网关查询子设备拓扑关系响应数据格式：

```
{
"type": "describe_sub_devices",
"payload": {
"devices": [
{
"product_id": "XKFA****LX",
```

```
"device_name": "2OGDy7Ws8mG****YUe"
},
{
"product_id": "XKFA****LX",
"device_name": "5gcEHg3Yuvm****2p8"
},
{
"product_id": "XKFA****LX",
"device_name": "hmIjq0gEFcf****F5X"
},
{
"product_id": "XKFA****LX",
"device_name": "x9pVpmdRmET****mkM"
},
{
"product_id": "XKFA****LX",
"device_name": "zmHv6o6n4G3****Bgh"
}
]
}
}
```

响应参数说明：

参数	类型	描述
type	String	网关消息类型。查询子设备取值为：`describe_sub_devices`
payload.devices	Array	网关绑定的子设备列表
product_id	String	子设备产品 ID
device_name	String	子设备名称

拓扑关系变化

网关类型的设备， 可以通过该 Topic 订阅平台对子设备的拓扑关系变化。

数据下行 Topic：`$gateway/operation/result/${productid}/${devicename}`

子设备被绑定或解绑，网关将收到子设备拓扑关系变化，数据格式如下：

```
{
"type": "change",
"payload": {
"status": 0, //0-解绑 1-绑定
"devices": [
{
```

```
"product_id": "CFCS****G7",
"device_name": "****ev",
}
]
}
}
```

请求参数说明：

参数	类型	描述
type	String	网关消息类型。拓扑关系变化取值为：change
status	Int	拓扑关系变化状态。 • 0：解绑 • 1：绑定
payload.devices	Array	网关绑定的子设备列表
product_id	String	子设备产品 ID
device_name	String	子设备名称

网关响应，数据格式如下：

```
{
"type": "change",
"result": 0
}
```

响应参数说明：

参数	类型	描述
type	String	网关消息类型。拓扑关系变化取值为：change
result	Int	网关响应处理结果

错误码

错误码	描述
0	成功
-1	网关设备未绑定该子设备

错误码	描述
-2	系统错误，子设备上线或者下线失败
801	请求参数错误
802	设备名非法，或者设备不存在
803	签名校验失败
804	签名方法不支持
805	签名请求已过期
806	该设备已被绑定
807	非普通设备不能被绑定
808	不允许的操作
809	重复绑定
810	不支持的子设备

代理子设备上下线

最近更新时间：2021-09-18 17:01:59

功能概述

网关类型的设备，可通过与云端的数据通信，代理其下的子设备进行上线与下线操作。此类功能所用到的 Topic 与网关子设备拓扑管理的 Topic 一致：

- 数据上行 Topic（用于发布）：`$gateway/operation/${productid}/${devicename}`
- 数据下行 Topic（用于订阅）：`$gateway/operation/result/${productid}/${devicename}`

代理子设备上线

网关类型的设备，可以通过数据上行 Topic 代理子设备上线。请求成功之后，云端通过数据下行 Topic 返回子设备的上线信息。

网关代理子设备上线请求数据格式：

```
{
"type": "online",
"payload": {
"devices": [
{
"product_id": "CFCSQ5EAG7",
"device_name": "onlinedev00"
}
]
}
}
```

代理子设备上线响应数据格式：

```
{
"type": "online",
"payload": {
"devices": [
{
"product_id": "CFCSQ5EAG7",
"device_name": "onlinedev00",
"result":0
}
]
}
}
```

请求参数说明：

字段	类型	描述
type	String	网关消息类型。代理子设备上线取值为：`online`
payload.devices	Array	需上线的子设备列表
product_id	String	子设备产品 ID
device_name	String	子设备名称

响应参数说明：

字段	类型	描述
type	String	网关消息类型。代理子设备上线取值为：`online`
payload.devices	Array	需上线的子设备列表
product_id	String	子设备产品 ID
device_name	String	子设备名称
result	Int	子设备上线结果，具体错误码见下表

代理子设备下线

网关类型的设备， 可以通过数据上行 Topic 代理子设备下线。请求成功之后，云端通过数据下行 Topic 返回成功子设备的下线信息。

网关代理子设备下线请求数据格式：

```
{
"type": "offline",
"payload": {
"devices": [
{
"product_id": "CFCSQ5EAG7",
"device_name": "offlinedev00"
}
]
}
}
```

代理子设备下线响应数据格式：

```
{
"type": "offline",
"payload": {
"devices": [
{
"product_id": "CFCSQ5EAG7",
"device_name": "offlinedev00",
"result":-1
}
]
}
}
```

请求参数说明：

字段	类型	描述
type	String	网关消息类型。代理子设备下线取值为：`offline`
payload.devices	Array	需代理下线的子设备列表
product_id	String	子设备产品 ID
device_name	String	子设备名称

响应参数说明：

字段	类型	描述
type	String	网关消息类型。代理子设备下线取值为：`offline`
payload.devices	Array	需代理下线的子设备列表
product_id	String	子设备产品 ID
device_name	String	子设备名称
result	Int	子设备下线结果，具体错误码见下表

错误码

错误码	描述
0	成功
-1	网关设备未绑定该子设备

错误码	描述
-2	系统错误，子设备上线或者下线失败
801	请求参数错误
802	设备名非法，或者设备不存在
810	不支持的子设备

代理子设备发布和订阅

最近更新时间：2021-10-25 10:43:58

功能概述

网关类型的设备，可通过与云端的数据通信，代理其下的子设备发布和订阅消息。

前提条件

发布和订阅消息之前，请参见 网关产品接入 和 代理子设备上下线，进行网关设备和子设备接入上线。

发布和订阅消息

网关设备可以使用子设备的 Topic 权限，代子设备进行收发消息。

子设备固件升级

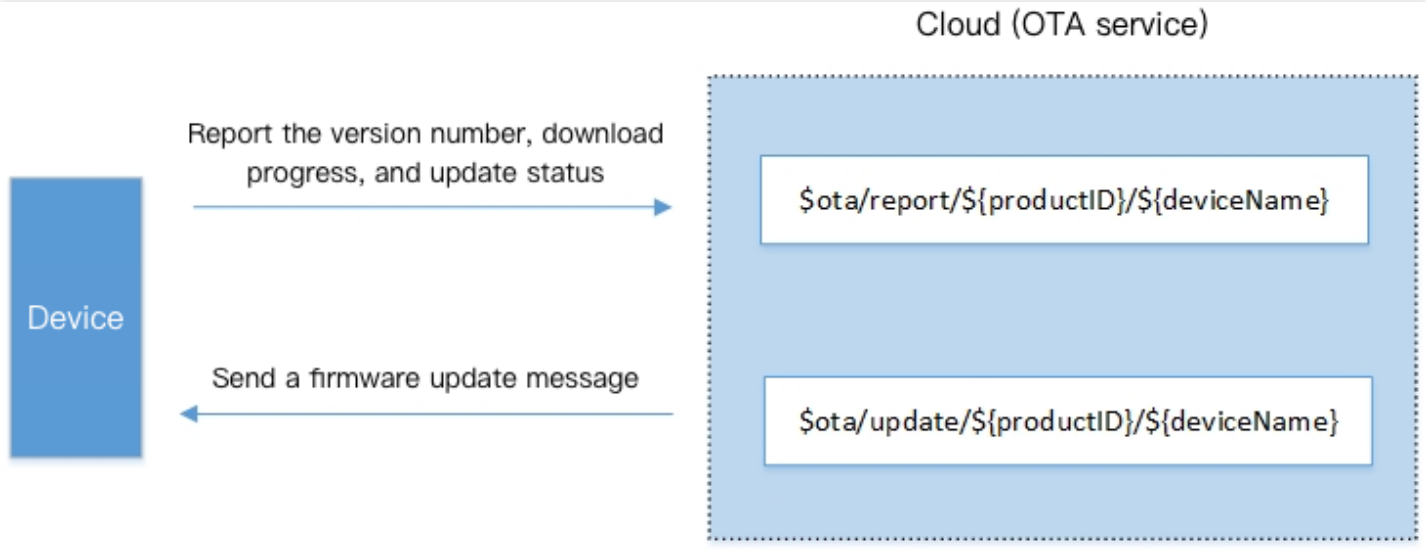
最近更新时间：2021-09-10 10:44:38

操作场景

当网关子设备有新功能或者需要修复漏洞时，子设备可以通过设备固件升级服务快速的进行固件升级。

实现原理

固件升级的过程中，需要网关代子设备使用下面两个 Topic 来实现与云端的通信，如下图所示：



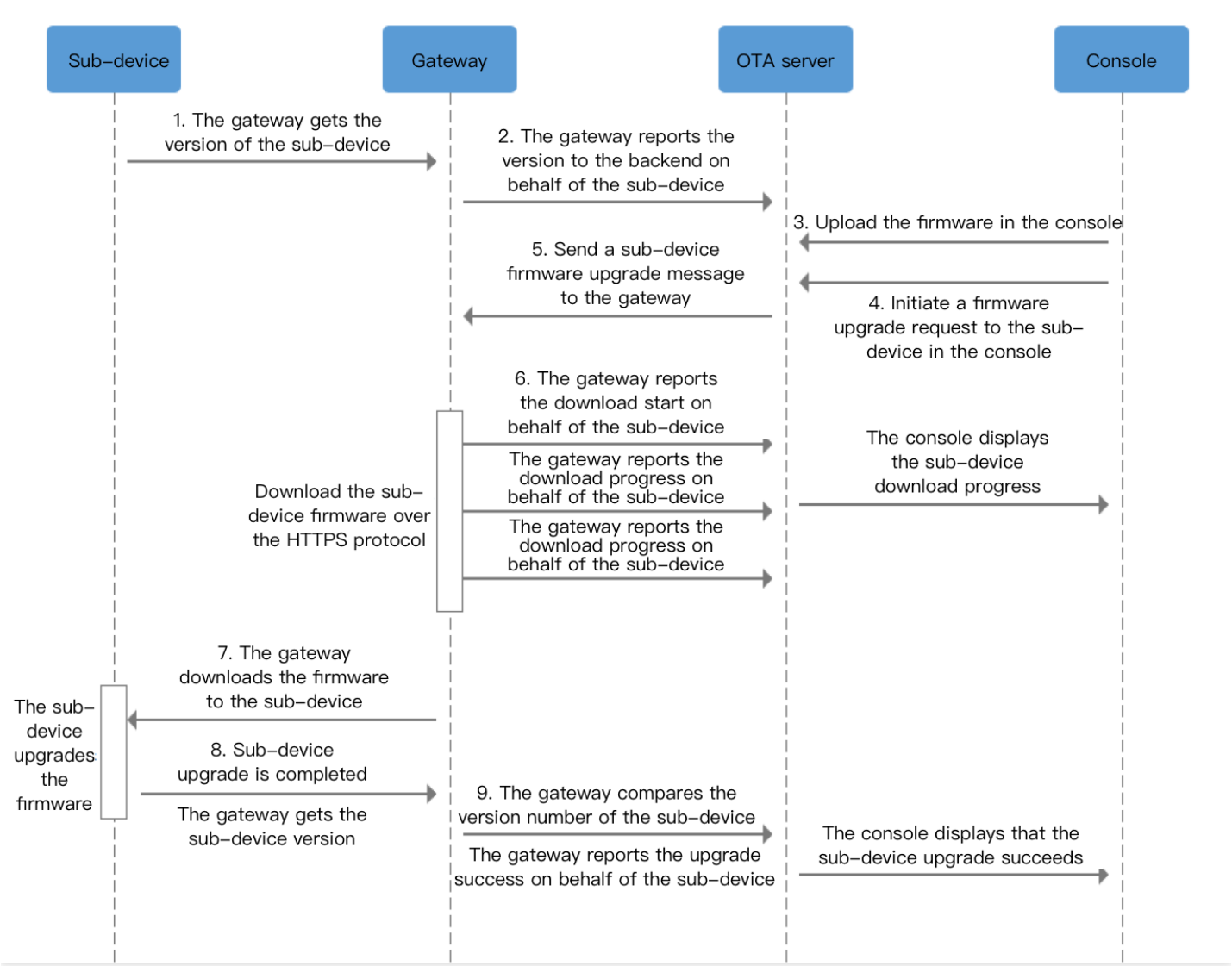
示例代码如下：

```
$ota/report/${productID}/${deviceName}
用于发布（上行）消息，设备上报版本号及下载、升级进度到云端
$ota/update/${productID}/${deviceName}
用于订阅（下行）消息，设备接收云端的升级消息
```

操作流程

以 MQTT 为例，子设备的升级流程图如下所示：

说明：
固件升级的具体操作步骤，详情请参见 设备固件升级。

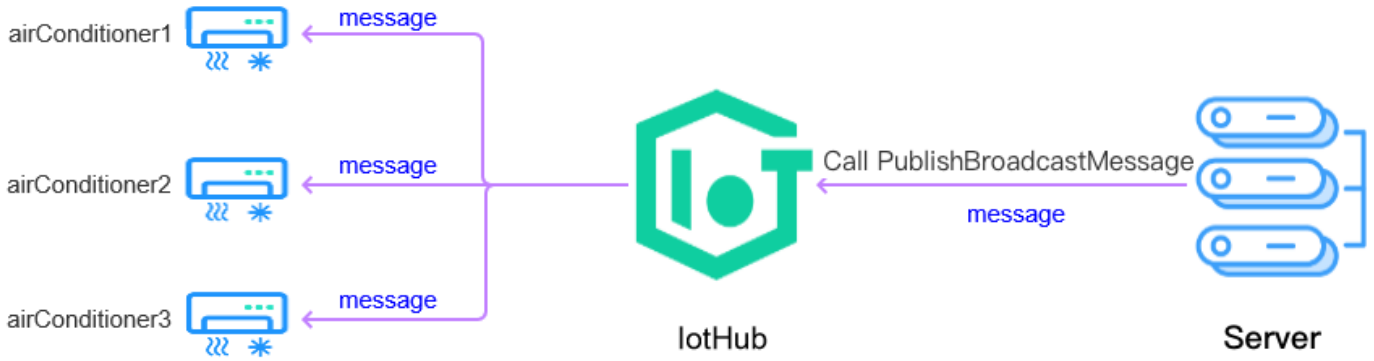


消息通信
广播通信

最近更新时间：2021-10-26 15:22:44

功能概述

物联网通信平台提供了广播通信 Topic，服务器通过调用广播通信 API 发布广播消息，同一产品下订阅了广播 Topic 的在线设备便可收到服务器通过广播 Topic 发布的广播消息。



广播 Topic

广播通信的 Topic 内容为：`$broadcast/rxd/${ProductId}/${DeviceName}` 其中 `${ProductId}`与 `${DeviceName}` 代表具体的产品 ID 和设备名称的内容。

广播通信示例

示例为基于 Linux 平台利用设备端 C-SDK 完成接入，并结合腾讯云 API Explorer 工具完成接口的调用，具体使用步骤如下。

控制台创建设备

场景说明

- 用户有多个空调设备接入物联网通信平台，则需要服务器向所有的空调设备发送一条相同的指令来关闭空调。

- 服务器调用PublishBroadcastMessage接口，指定产品 `ProductId` 和广播消息 `Payload` ，该产品订阅了所有广播 Topic 的在线设备就会收到消息内容为 `Payload` 的广播消息。

创建产品和设备

请参考 设备互通 文档创建空调产品，并依次创建 airConditioner1，airConditioner2 等多个空调设备。

编译运行示例程序（以密钥认证设备为例）

1. 编译 SDK

修改 CMakeLists.txt 确保以下选项存在：

```
set(BUILD_TYPE "release")
set(COMPILE_TOOLS "gcc")
set(PLATFORM "linux")
set(FEATURE_MQTT_COMM_ENABLED ON)
set(FEATURE_BROADCAST_ENABLED ON)
set(FEATURE_AUTH_MODE "KEY")
set(FEATURE_AUTH_WITH_NOTLS OFF)
set(FEATURE_DEBUG_DEV_INFO_USED OFF)
```

执行脚本编译：

```
./cmake_build.sh
```

示例输出 `broadcast_sample` 位于 `output/release/bin` 文件夹中。

2. 填写设备信息

将上面创建的 airConditioner1 设备的设备信息填写到一个 JSON 文件 `aircond_device_info1.json` 中：

```
{
"auth_mode":"KEY",
"productId":"KL4J2****8",
"deviceName":"airConditioner1",
"key_deviceinfo":{
"deviceSecret":"zOZXUaycuwlePt****8dBA=="
}
}
```

将 airConditioner2 的设备信息填写到一个 JSON 文件 `aircond_device_info2.json` 中：

```
{
"auth_mode":"KEY",
"productId":"KL4J2****8",
```

```
"deviceName":"airConditioner2",
"key_deviceinfo":{
"deviceSecret":"+IiVNsyKRh0AW****IE07A=="
}
}
```

依次将其他的设备信息填到对应的 JSON 文件。

3. 执行 `broadcast_sample` 示例程序

因为该场景涉及到多个 sample 同时运行，可以打开多个终端运行 `broadcast_sample` 示例，看到所有的示例都订阅了 `$broadcast/rxd/${productID}/${deviceName}` 主题，并处于等待状态。

设备 airConditioner1 输出如下：

```
./broadcast_sample -c ./aircond_device_info1.json -l 100
INF|2020-08-03 22:50:28|qcloud_iot_device.c|iot_device_info_set(50): SDK_Ver: 3.2
.0, Product_ID: KL4J2****8, Device_Name: airConditioner1
DBG|2020-08-03 22:50:28|HAL_TLS_mbedtls.c|HAL_TLS_Connect(200): Setting up the SS
L/TLS structure...
DBG|2020-08-03 22:50:28|HAL_TLS_mbedtls.c|HAL_TLS_Connect(242): Performing the SS
L/TLS handshake...
DBG|2020-08-03 22:50:28|HAL_TLS_mbedtls.c|HAL_TLS_Connect(243): Connecting to /KL
4J2****8.iotcloud.tencentdevices.com/8883...
INF|2020-08-03 22:50:28|HAL_TLS_mbedtls.c|HAL_TLS_Connect(265): connected with /K
L4J2****8.iotcloud.tencentdevices.com/8883...
INF|2020-08-03 22:50:28|mqtt_client.c|IOT_MQTT_Construct(113): mqtt connect with
id: 9**** success
INF|2020-08-03 22:50:28|broadcast_sample.c|main(197): Cloud Device Construct Succ
ess
DBG|2020-08-03 22:50:28|mqtt_client_subscribe.c|qcloud_iot_mqtt_subscribe(142): t
opicName=$broadcast/rxd/KL4J2****8/airConditioner1|packet_id=*****
INF|2020-08-03 22:50:28|broadcast_sample.c|_mqtt_event_handler(49): subscribe suc
cess, packet-id=*****
DBG|2020-08-03 22:50:28|broadcast.c|_broadcast_event_callback(37): broadcast topi
c subscribe success
```

设备 airConditioner2 输出如下：

```
./broadcast_sample -c ./aircond_device_info2.json -l 100
INF|2020-08-03 22:51:24|qcloud_iot_device.c|iot_device_info_set(50): SDK_Ver: 3.2
.0, Product_ID: KL4J2****8, Device_Name: airConditioner2
DBG|2020-08-03 22:51:24|HAL_TLS_mbedtls.c|HAL_TLS_Connect(200): Setting up the SS
L/TLS structure...
DBG|2020-08-03 22:51:24|HAL_TLS_mbedtls.c|HAL_TLS_Connect(242): Performing the SS
L/TLS handshake...
DBG|2020-08-03 22:51:24|HAL_TLS_mbedtls.c|HAL_TLS_Connect(243): Connecting to /KL
```

```
4J2****8.iotcloud.tencentdevices.com/8883...
INF|2020-08-03 22:51:24|HAL_TLS_mbedtls.c|HAL_TLS_Connect(265): connected with /K
L4J2****8.iotcloud.tencentdevices.com/8883...
INF|2020-08-03 22:51:25|mqtt_client.c|IOT_MQTT_Construct(113): mqtt connect with
id: f**** success
INF|2020-08-03 22:51:25|broadcast_sample.c|main(197): Cloud Device Construct Succ
ess
DBG|2020-08-03 22:51:25|mqtt_client_subscribe.c|qcloud_iot_mqtt_subscribe(142): t
opicName=$broadcast/rxd/KL4J2****8/airConditioner2|packet_id=*****
INF|2020-08-03 22:51:25|broadcast_sample.c|_mqtt_event_handler(49): subscribe suc
cess, packet-id=******
DBG|2020-08-03 22:51:25|broadcast.c|_broadcast_event_callback(37): broadcast topi
c subscribe success
```

4. 调用云API `PublishBroadcastMessage` 发送广播消息

打开腾讯云 API 控制台，填写个人密钥和设备参数信息，选择在线调用并发送请求。

5. 观察空调设备的消息接收

观察设备 airConditioner1 的打印输出，可以看到已经收到服务器发送的消息。

```
DBG|2020-08-03 22:55:32|broadcast.c|_broadcast_message_cb(25): topic=$broadcast/r
xd/KL4J2****8/airConditioner1
INF|2020-08-03 22:55:32|broadcast.c|_broadcast_message_cb(26): len=6, topic_msg=c
losed
INF|2020-08-03 22:55:32|broadcast_sample.c|_broadcast_message_handler(134): broad
cast message=closed
```

观察设备 airConditioner2 的打印输出，可以看到同时收到服务器发送的消息。

```
DBG|2020-08-03 22:55:32|broadcast.c|_broadcast_message_cb(25): topic=$broadcast/r
xd/KL4J2****8/airConditioner2
INF|2020-08-03 22:55:32|broadcast.c|_broadcast_message_cb(26): len=6, topic_msg=c
losed
INF|2020-08-03 22:55:32|broadcast_sample.c|_broadcast_message_handler(134): broad
cast message=closed
```

6. 空调设备关闭

接收到指令的设备解析指令进行处理。

RRPC通信

最近更新时间：2021-10-27 14:41:21

功能概述

因MQTT 协议基于发布/订阅的异步通信模式，服务器控制设备后，将无法同步感知设备的返回结果。为解决此问题，物联网通信平台利用 RRPC（Revert RPC）实现同步通信机制。

通信原理

通信 Topic

- 订阅消息 Topic： `$rrpc/rxd/${productID}/${deviceName}/+` 用于订阅云端下发（下行）的 RRPC 请求消息。
- 请求消息 Topic： `$rrpc/rxd/${productID}/${deviceName}/${processID}` 用于云端发布（下行）RRPC 请求消息。
- 应答消息 Topic： `$rrpc/txd/${productID}/${deviceName}/${processID}` 用于发布（上行）RRPC 应答消息。

> 说明：
>
> - `${productID}` ：产品 ID。
> - `${deviceName}` ：设备名称。
> - `${processID}` ： 服务器生成的唯一的消息 ID，用来标识不同 RRPC 消息。可以通过 RRPC 应答消息中携带的 processID 找到对应的 RRPC 请求消息。

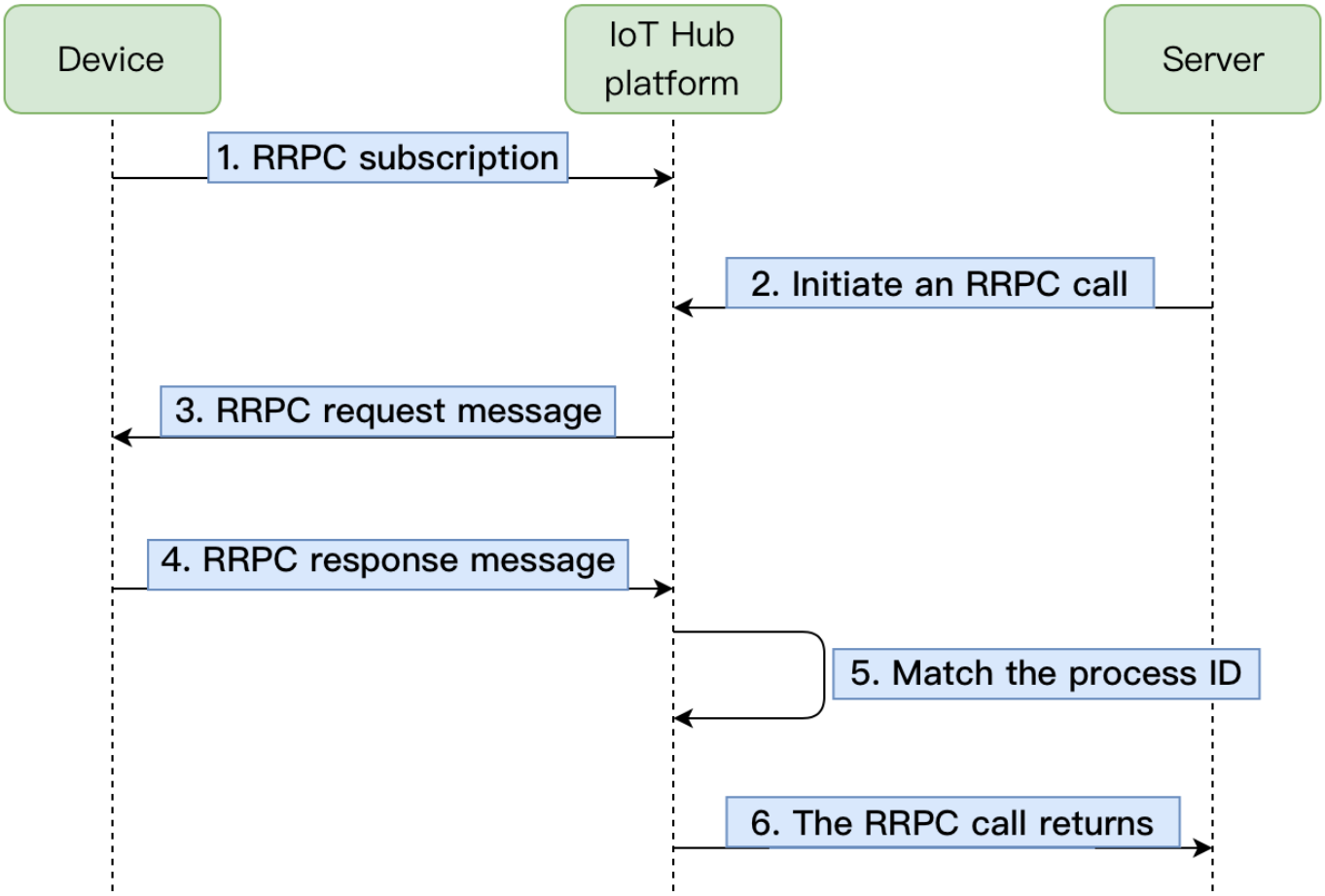
通信流程

1. 设备端订阅 RRPC 订阅消息 Topic。
2. 服务器通过调用PublishRRPCMessage接口发布 RRPC 请求消息。
3. 设备端接收到消息之后截取请求消息 Topic 中云端下发的 processID，设备将应答消息 Topic 的 processID 设置为截取的 processID，并向应答消息 Topic 发布设备的返回消息 。
4. 物联网通信平台接收到设备端返回消息之后，根据 processID 对消息进行匹配并将设备返回消息发送给服务器。

> 注意：

RRPC 请求4s超时，即4s内设备端没有应答就认为请求超时。

流程示意图如下：



RRPC 通信示例

示例为基于 Linux 平台利用设备端 C-SDK 完成接入，并结合腾讯云 API Explorer 工具完成接口的调用，具体使用步骤如下。

控制台创建设备

创建产品和设备

请参考 设备互通 创建空调产品，并创建 airConditioner1 空调设备。

编译运行示例程序（以密钥认证设备为例）

1. 编译 SDK

修改 `CMakeLists.txt` 确保以下选项存在：

```
set(BUILD_TYPE "release")
set(COMPILE_TOOLS "gcc")
set(PLATFORM "linux")
set(FEATURE_MQTT_COMM_ENABLED ON)
set(FEATURE_RRPC_ENABLED ON)
set(FEATURE_AUTH_MODE "KEY")
set(FEATURE_AUTH_WITH_NOTLS OFF)
set(FEATURE_DEBUG_DEV_INFO_USED OFF)
```

执行脚本编译：

```
./cmake_build.sh
```

示例输出 `rrpc_sample` 位于 `output/release/bin` 文件夹中。

2. 填写设备信息

将上面创建的 airConditioner1 设备的设备信息填写到 JSON 文件 `aircond_device_info1.json` 中:

```
{
"auth_mode":"KEY",
"productId":"KL4J2****8",
"deviceName":"airConditioner1",
"key_deviceinfo":{
"deviceSecret":"zOZXUaycuwleP****78dBA=="
}
}
```

3. 执行 `rrpc_sample` 示例程序

可以看到设备airConditioner1订阅了RRPC消息，然后处于等待状态。

```
./rrpc_sample -c ./aircond_device_info1.json -l 1000
INF|2020-08-03 23:57:55|qcloud_iot_device.c|iot_device_info_set(50): SDK_Ver: 3.2
.0, Product_ID: KL4J2****8, Device_Name: airConditioner1
DBG|2020-08-03 23:57:55|HAL_TLS_mbedtls.c|HAL_TLS_Connect(200): Setting up the SS
L/TLS structure...
DBG|2020-08-03 23:57:55|HAL_TLS_mbedtls.c|HAL_TLS_Connect(242): Performing the SS
L/TLS handshake...
DBG|2020-08-03 23:57:55|HAL_TLS_mbedtls.c|HAL_TLS_Connect(243): Connecting to /KL
4J2****8.iotcloud.tencentdevices.com/8883...
INF|2020-08-03 23:57:55|HAL_TLS_mbedtls.c|HAL_TLS_Connect(265): connected with /K
```

```
L4J2****8.iotcloud.tencentdevices.com/8883...
INF|2020-08-03 23:57:56|mqtt_client.c|IOT_MQTT_Construct(113): mqtt connect with
id: 2**** success
INF|2020-08-03 23:57:56|rrpc_sample.c|main(206): Cloud Device Construct Success
DBG|2020-08-03 23:57:56|mqtt_client_subscribe.c|qcloud_iot_mqtt_subscribe(142): t
opicName=$rrpc/rxd/KL4J2****8/airConditioner1/+|packet_id=****
INF|2020-08-03 23:57:56|rrpc_sample.c|_mqtt_event_handler(49): subscribe success,
packet-id=*****
DBG|2020-08-03 23:57:56|rrpc_client.c|_rrpc_event_callback(104): rrpc topic subsc
ribe success
```

4. 调用云 API `PublishRRPCMessage` 发送 RRPC 请求消息

打开腾讯云 API控制台，填写个人密钥和设备参数信息，选择在线调用并发送请求。

5. 观察 RRPC 请求消息

观察设备 airConditioner1 的打印输出，可以看到已经收到 RRPC 请求消息， `process id` 为***。

```
DBG|2020-08-04 00:07:36|rrpc_client.c|_rrpc_message_cb(85): topic=$rrpc/rxd/KL4J2
****8/airConditioner1/***
INF|2020-08-04 00:07:36|rrpc_client.c|_rrpc_message_cb(86): len=6, topic_msg=clos
ed
INF|2020-08-04 00:07:36|rrpc_client.c|_rrpc_get_process_id(76): len=3, process id
=***
INF|2020-08-04 00:07:36|rrpc_sample.c|_rrpc_message_handler(137): rrpc message=cl
osed
```

6. 观察RRPC应答消息

观察设备 airConditioner1 的打印输出，可以看到已经处理了 RRPC 请求消息，并回复了 RRPC 应答消息， `process id` 为***。

```
DBG|2020-08-04 00:07:36|mqtt_client_publish.c|qcloud_iot_mqtt_publish(340): publi
sh packetID=0|topicName=$rrpc/txd/KL4J2****8/airConditioner1/***|payload=ok
```

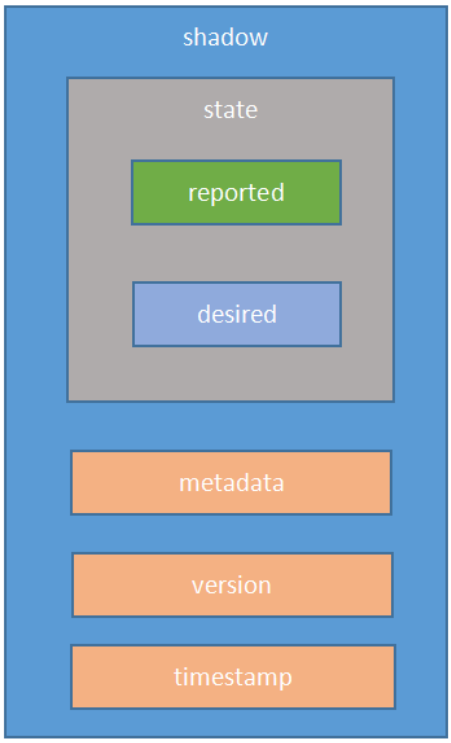
7. 观察服务器响应结果

观察服务器的响应结果，可以看到已经收到了 RRPC 应答消息。 `MessageId` 为***， `Payload` 经过 `base64` 编码后为****，其与客户端实际应答消息经过 `base64` 编码后一致。可以确认收到了应答消息。

设备影子
设备影子详情

最近更新时间：2021-08-20 16:27:33

设备影子文档是服务器端为设备缓存的一份状态和配置数据。它以 JSON 文本形式存储，由以下部分组成：



state

reported

设备自身上报的状态。设备可以向本文档部分写入数据，以报告其新状态。应用程序可以读取本文档部分，以获取设备的状态。

desired

设备预期的状态。应用程序通过 HTTP RESTful API 向本文档写入数据更新设备状态，设备 SDK 通过注册相关属性和回调，设备影子服务同步影子数据到设备。

metadata

设备影子的元数据信息，包括 state 部分每个属性项的最后更新时间等。

version

设备影子文档的版本号，每次设备影子文档更新之后，版本号都会递增。版本号由腾讯云后台维护，这可以确保设备的数据与设备影子的数据保持一致。

timestamp

设备影子文档的最后一次更新时间。设备影子文档示例如下：

```
{
"state": {
"reported": {
"attr_name1": "value1"
},
"desired": {
"attr_name2": "value2"
}
},
"metadata": {
"reported": {
"attr_name1": {
"timestamp": 123456789
}
},
"desired": {
"attr_name2": {
"timestamp": 123456789
}
}
},
"version": 1,
"timestamp": 123456789
}
```

空白部分

当设备影子文档为空时，此时获取到设备影子文档示例如下：

```
{
"state":{},
"metadata":{},
"version":0
}
```

当设备影子文档具有预期状态时，才会有 desired 部分，reported 部分可以为空，例如：

```
{
"state": {
"desired": {
"attr_name2": "value2"
}
},
"metadata": {
"desired": {
"attr_name2": {
"timestamp": 123456789
}
}
},
"version": 1,
"timestamp": 123456789
}
```

当设备更新状态成功之后，需要上报最新的状态，并将 desired 部分从文档中删除。要想将 desired 部分从文档中删除，需要将 desired 部分置为 null，例如：

```
{
"state": {
"reported": {
"attr_name1": "new_value1",
"attr_name2": "new_value2"
},
"desired": null
},
"version": 1
}
```

数组

设备影子文档支持数组，不支持针对数组某个元素进行更新，只能更新整个数组，并且数组元素不能有空值。

设备影子数据流

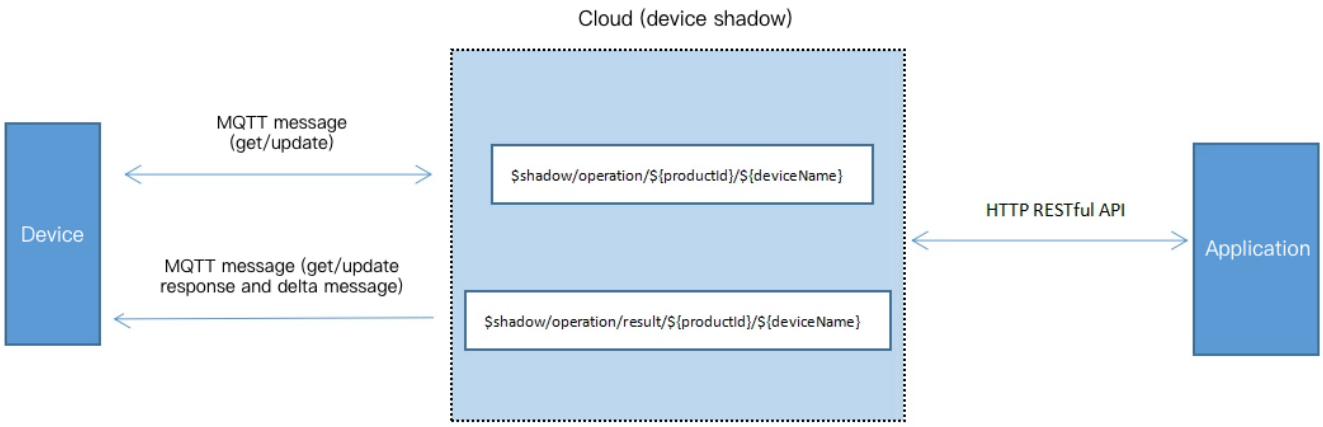
最近更新时间：2021-08-23 12:06:05

设备影子 Topic

设备影子充当中介，支持设备和用户应用程序查看和更新设备状态。设备、用户应用程序、设备影子三者之间通过两个特殊的 Topic 来实现通信：

- `$shadow/operation/${productId}/${deviceName}` ：用于发布（上行）消息，可实现对设备影子数据的 get/update 操作。
- `$shadow/operation/result/${productId}/${deviceName}` ：用于订阅（下行）消息，影子服务端通过此 Topic 发送应答和推送消息。

> 说明：
> 以上主题均为设备创建时由系统默认创建，设备 SDK 内部会自动订阅上述主题。



设备获取影子状态

如果设备想要获取设备影子最近的状态时，需要向 `$shadow/operation/${productId}/${deviceName}` 主题发布 get 消息。SDK 会提供 API 发送 get 消息，get 消息使用特定的 JSON 字符串格式：

```
{
"type": "get",
"clientToken": "clientToken"
}
```

说明：
clientToken 是用于唯一标识会话业务的 TOKEN，由请求端生成，应答端原样传回。

例如：空调设备，可以通过 SDK 提供的 API 向 `$shadow/operation/ABC1234567/AirConditioner` 发送 get 消息，获取最新的设备参数。

设备影子服务端通过向 `$shadow/operation/result/${productId}/${deviceName}` 主题发布消息进行响应，通过 JSON 数据返回设备影子所有数据内容，SDK 会通过相应的回调函数通知业务层。

影子服务端通过向 `$shadow/operation/result/ABC1234567/AirConditioner` 发送下列数据来响应空调设备的 get 请求。示例代码如下：

```
{
"type": "get",
"result": 0,
"timestamp": 1514967088,
"clientToken": "clientToken",
"payload": {
"state": {
"reported": {
"temperature": 27
},
"desired": {
"temperature": 25
},
"delta": {
"temperature": 25
}
},
"metadata": {
"reported": {
"temperature": {
"timestamp": 1514967066
}
},
"desired": {
"temperature": {
```

```
"timestamp": 1514967076
}
},
"delta": {
"temperature": {
"timestamp": 1514967076
}
}
},
"version": 1,
"timestamp": 1514967076
}
}
```

如果设备影子文档中有 desired 部分，则设备影子服务会自动生成相应的 delta 部分。如果没有 desired 部分，则没有 desired 和 delta 部分的内容。

> 说明：
> 设备影子服务 **不存储** delta 消息。

设备更新影子

设备通过向 `$shadow/operation/${productId}/${deviceName}` 主题发送 update 消息，告知设备影子服务端其当前状态。SDK 提供相应的 API 发送 update 消息，业务层只需指定 reported 字段的内容。消息内容使用特定的 JSON 字符串格式。

空调设备向 `$shadow/operation/ABC1234567/AirConditioner` 发送 update 消息以报告设备当前的设备状态。示例代码如下：

```
{
"type": "update",
"state": {
"reported": {
"temperature": 27
}
},
"version": 1,
"clientToken": "clientToken"
}
```

当设备影子服务端收到此消息时，首先判断消息中的 version 是否与设备影子服务端中的 version 一致。如果一致，则设备影子服务端执行更新设备影子流程。

影子服务端向空调设备应答消息。示例代码如下：

```
{
"type": "update",
"result": 0,
"timestamp": 1514967066,
"clientToken": "clientToken",
"payload": {
"state": {
"reported": {
"temperature": 27
}
},
"metadata": {
"reported": {
"temperature": {
"timestamp": 1514967066
}
}
},
"version": 2,
"timestamp": 1514967066
}
}
```

如果消息中的 version 与设备影子服务端的 version 不一致，则设备影子服务向 `$shadow/operation/result/ABC1234567/AirConditioner` 发送以下消息进行回应。

```
{
"type": "update",
"result": 5005,
"timestamp": 1514967066,
"clientToken": "clientToken",
"payload": {
"state": {
"reported": {
"temperature": 27,
"mode": "cool"
}
},
"metadata": {
"reported": {
"temperature": {
```

```
"timestamp": 1514967066
},
"mode": {
"timestamp": 1514967050
}
}
},
"version": 2,
"timestamp": 1514967066
}
}
```

此时 payload 中的内容将返回完整的设备影子文档内容。

应用程序更新影子

用户应用程序通过 HTTP RESTful API 修改设备影子 desired 字段。

用户应用程序通过 HTTP RESTful API 修改空调运行参数。示例代码如下：

```
{
"type": "update",
"state": {
"desired": {
"temperature": 25
}
},
"version": 2,
"clientToken": "clientToken"
}
```

当设备影子服务端收到此消息后，首先判断消息中的 version 是否与设备影子服务端中的 version 一致。如果一致，则执行更新设备影子流程，并通过 HTTP RESTful API 给应用程序应答 JSON 消息。

```
{
"type": "update",
"result": 0,
"timestamp": 1514967076,
"clientToken": "clientToken",
"payload": {
"state": {
"desired": {
"temperature": 25
}
```

```
},
"metadata": {
"desired": {
"temperature": {
"timestamp": 1514967076
}
}
},
"version": 3,
"timestamp": 1514967076
}
}
```

另外，影子服务端通过向 `$shadow/operation/result/ABC1234567/AirConditioner` 发送 delta 消息。

```
{
"type": "delta",
"timestamp": 1514967076,
"payload": {
"state": {
"temperature": 25
},
"metadata": {
"temperature": {
"timestamp": 1514967076
}
},
"version": 3,
"timestamp": 1514967076
}
}
```

SDK 通过相应的回调函数通知业务层消息已收到。

设备应答 delta 消息

当设备收到 delta 消息后，业务层可以将 desired 字段内容置空并发送给设备影子服务端，表示设备已经 **响应** 本次 delta 消息，方式是通过向 `$shadow/operation/${productId}/${deviceName}` 主题发送消息。

例如：空调调整温度后，向 `$shadow/operation/ABC1234567/AirConditioner` 发送消息：

```
{
"type": "update",
"state": {
```

```
"desired": null
},
"version": 3,
"clientToken": "clientToken"
}
```

SDK 提供相应的 API 发送上述消息。当设备影子服务端收到该消息后，会将 desired 字段内容清除，防止由于 reported 与 desired 中属性值不同引起的重复下发。

影子服务端收到消息后，通过向: `$shadow/operation/result/${productId}/${deviceName}` 发送应答消息。

例如：影子服务端收到空调的 `"desired":null` 消息后，通过向 `$shadow/operation/result/ABC1234567/AirConditioner` 发送更新设备影子文档成功的消息。

```
{
"type": "update",
"result": 0,
"timestamp": 1514967086,
"clientToken": "clientToken",
"payload": {
"state": {
"reported": {
"temperature": 25
},
"desired": null
},
"metadata": {
"reported": {
"temperature": {
"timestamp": 1514967086
}
},
"desired": {
"temperature": {
"timestamp": 1514967086
}
}
},
"version": 4,
"timestamp": 1514967086
}
}
```

如果设备 reported 某些属性字段为 null，则代表要将设备影子中相应的字段删除。update 成功时，返回的 payload 内容中，字段仅包含对此次更新字段的相关内容。

如果设备 update 时携带上的 version 值小于服务端上保存的 version 值，则代表设备上的数据是旧的。此时服务端将发送失败消息，其中的返回码（ result 字段）会明确告知 SDK 本次 update 失败，原因是 version 版本过低，同时在 payload 中携带最新的内容一同下发给设备端。

设备固件升级

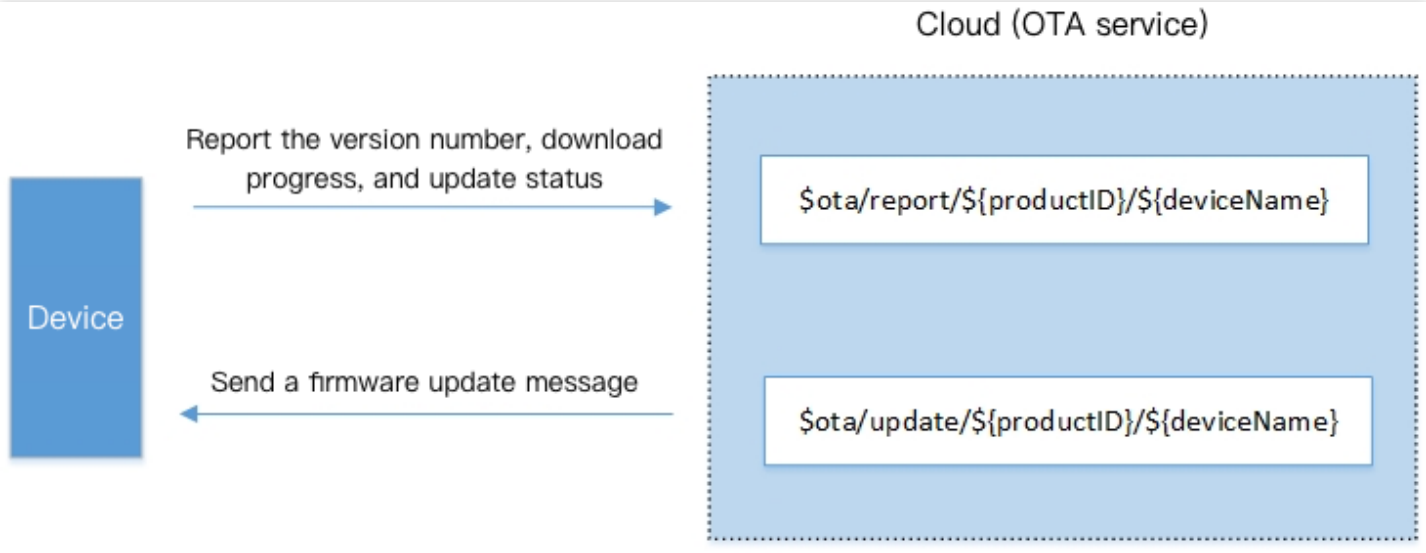
最近更新时间：2021-09-10 10:37:16

操作场景

设备固件升级，是物联网通信服务的重要组成部分。当物联设备有新功能或者需要修复漏洞时，设备可以通过设备固件升级服务快速的进行固件升级。

实现原理

固件升级的过程中，需要设备订阅下面两个 Topic 来实现与云端的通信，如下图所示：

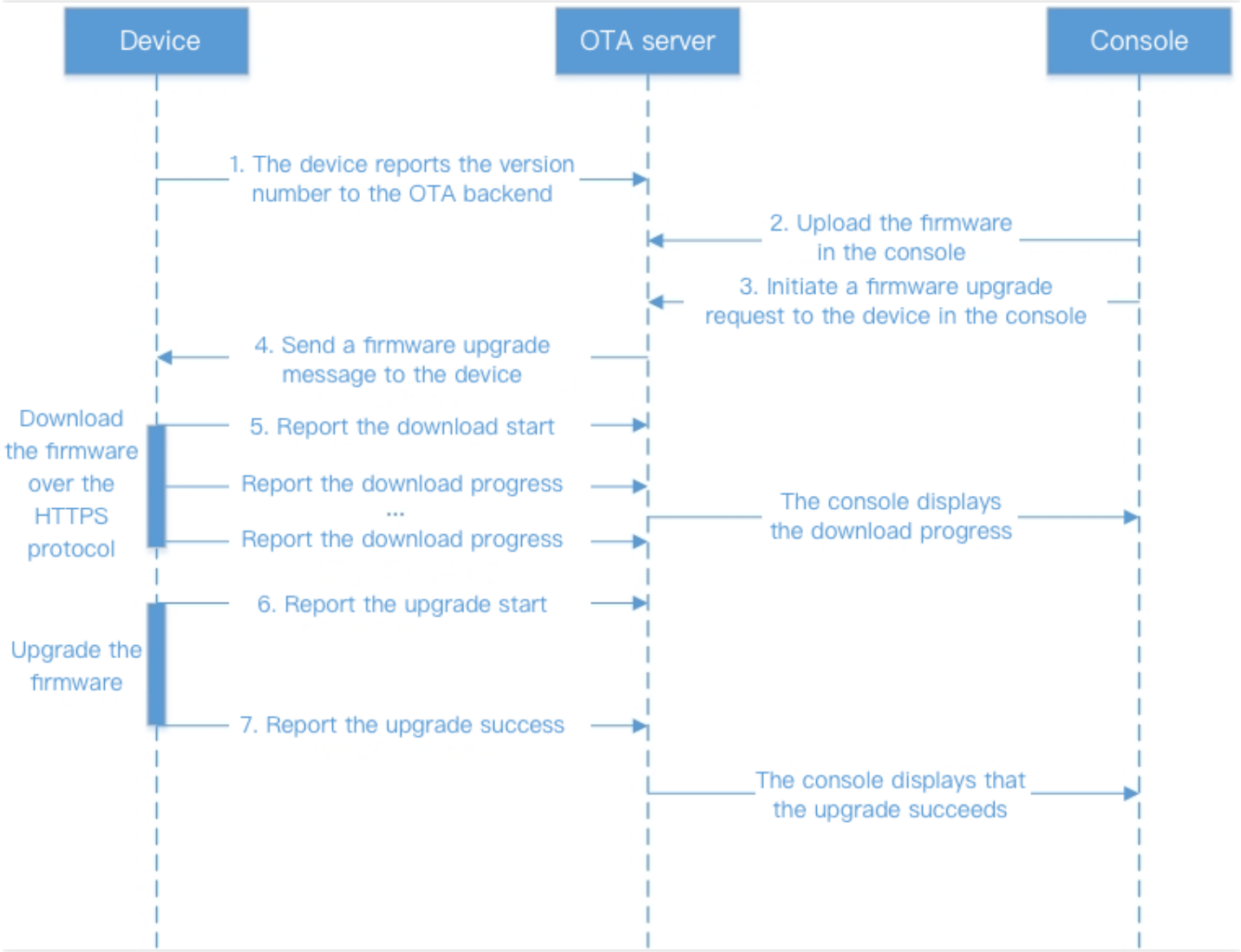


示例如下：

```
$ota/report/${productID}/${deviceName}
用于发布（上行）消息，设备上报版本号及下载、升级进度到云端
$ota/update/${productID}/${deviceName}
用于订阅（下行）消息，设备接收云端的升级消息
```

操作流程

以 MQTT 为例，设备的升级流程如下所示：



1. 设备上报当前版本号。设备端通过 MQTT 协议发布一条消息到 Topic `$ota/report/${productID/${deviceName}`，进行版本号的上报，消息为 json 格式，内容如下：

```
{
"type": "report_version",
"report":{
"version": "0.1"
}
}
// type：消息类型
// version：上报的版本号
```

2. 登录 物联网通信控制台，进行固件上传，并将指定的设备升级到指定的版本。

3. 触发固件升级操作后，设备端会通过订阅的 Topic `$ota/update/${productID}/${deviceName}` 收到固件升级的消息，内容如下：

```
{
"file_size": 708482,
"md5sum": "36eb5951179db14a63****a37a9322a2",
"type": "update_firmware",
"url": "https://ota-1255858890.cos.ap-guangzhou.myqcloud.com",
"version": "0.2"
}
// type：消息类型为update_firmware
// version：升级版本
// url：下载固件的url
// md5asum：固件的MD5值
// file_size：固件大小，单位为字节
```

4. 设备在收到固件升级的消息后，根据 URL 下载固件，下载的过程中设备 SDK 会通过 Topic `$ota/report/${productID}/${deviceName}` 不断的上报下载进度，上报的内容如下：

```
{
"type": "report_progress",
"report":{
"progress":{
"state":"downloading",
"percent":"10",
"result_code":"0",
"result_msg":""
},
"version": "0.2"
}
}
// type：消息类型
// state：状态为正在下载中
// percent：当前下载进度，百分比
```

5. 当设备下载完固件，设备需要通过 Topic `$ota/report/${productID}/${deviceName}` 上报一条开始升级的消息，内容如下：

```
{
"type": "report_progress",
"report":{
"progress":{
"state":"burning",
```

```
"result_code":"0",
"result_msg":""
},
"version": "0.2"
}
}
// type：消息类型
// state：状态为烧制中
```

6. 设备固件升级完成后，再向 Topic `$ota/report/${productID}/${deviceName}` 上报升级成功消息，内容如下：

```
{
"type": "report_progress",
"report":{
"progress":{
"state":"done",
"result_code":"0",
"result_msg":""
},
"version": "0.2"
}
}
// type：消息类型
// state：状态为已完成
```

在下载固件或升级固件的过程中，如果失败，则通过 Topic `$ota/report/${productID}/${deviceName}` 上报升级失败消息，内容如下：

```
{
"type": "report_progress",
"report":{
"progress":{
"state":"fail",
"result_code":"-1",
"result_msg":"time_out"
},
"version": "0.2"
}
}
// state：状态为失败
// result_code：错误码，-1：下载超时；-2：文件不存在；-3：签名过期；-4:MD5不匹配；-5：更新
```

```
固件失败
// result_msg：错误消息
```

OTA 断点续传

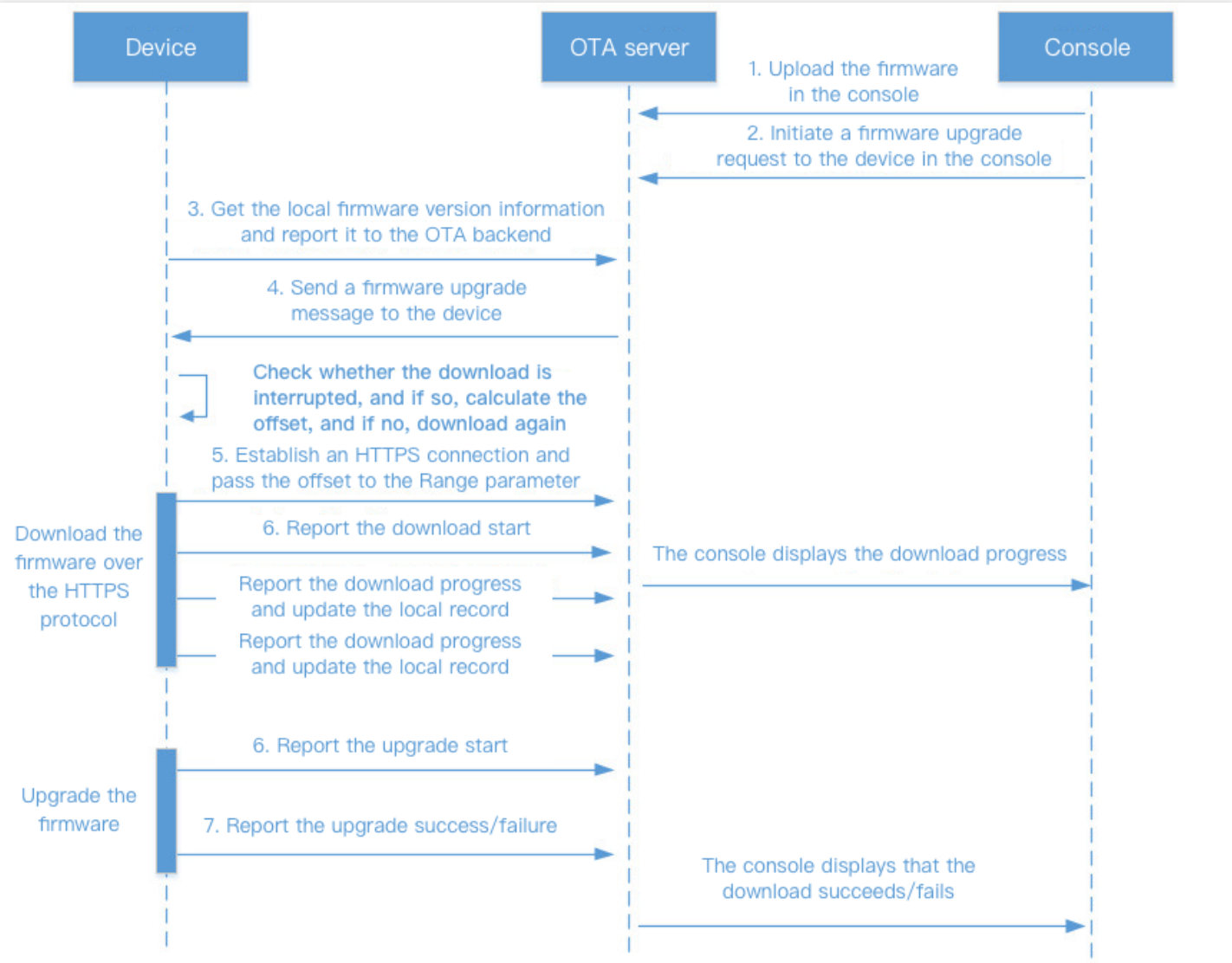
物联网设备会出现在部分场景处于弱网环境，在此场景下连接将会不稳定，固件下载会出现中断的情况。如果每次都从0偏移开始下载固件，则在弱网环境有可能一直无法完成全部固件下载，因此固件的断点续传功能特别必要，关于断点续传的具体说明如下。

- 断点续传指从文件上次中断的地方开始重新下载或上传，要实现断点续传的功能，需要设备端记录固件下载的中断位置，同时记录下载固件的 MD5、文件大小、版本信息。
- 平台针对 OTA 中断的场景，设备侧 report 设备的版本，如果上报的版本号与要升级的目标版本号不一致，则平台会再次下发固件升级消息，设备将获取到要升级的目标固件信息与本地记录的中断的固件信息进行比较，确定为同一固件后，基于断点继续下载。

带断点续传的 OTA 升级流程如下：

说明：

- 弱网环境下步骤3-6有可能会多次执行，执行成功后再执行第7步。
- 执行步骤3后，设备端都会收到需要执行步骤4的消息。



Checkpoint restart process for OTA upgrade

设备远程配置

最近更新时间：2021-09-26 17:17:35

功能概述

设备使用场景中，对于需要更新系统参数（如：设备的 IP、端口号和串口参数等）的设备，可采用远程配置功能对设备系统参数进行更新。

功能详情

设备远程配置分为物联网平台主动下发和设备端主动请求两种配置更新方式。对于同一产品下所有设备均需更新配置的场景，可采用物联网平台主动下发的形式，将配置信息通过远程配置 Topic 下发到同一产品下的所有设备中。对于部分设备更新配置信息的场景，可采用设备端主动请求远程配置 Topic 的方式来完成。

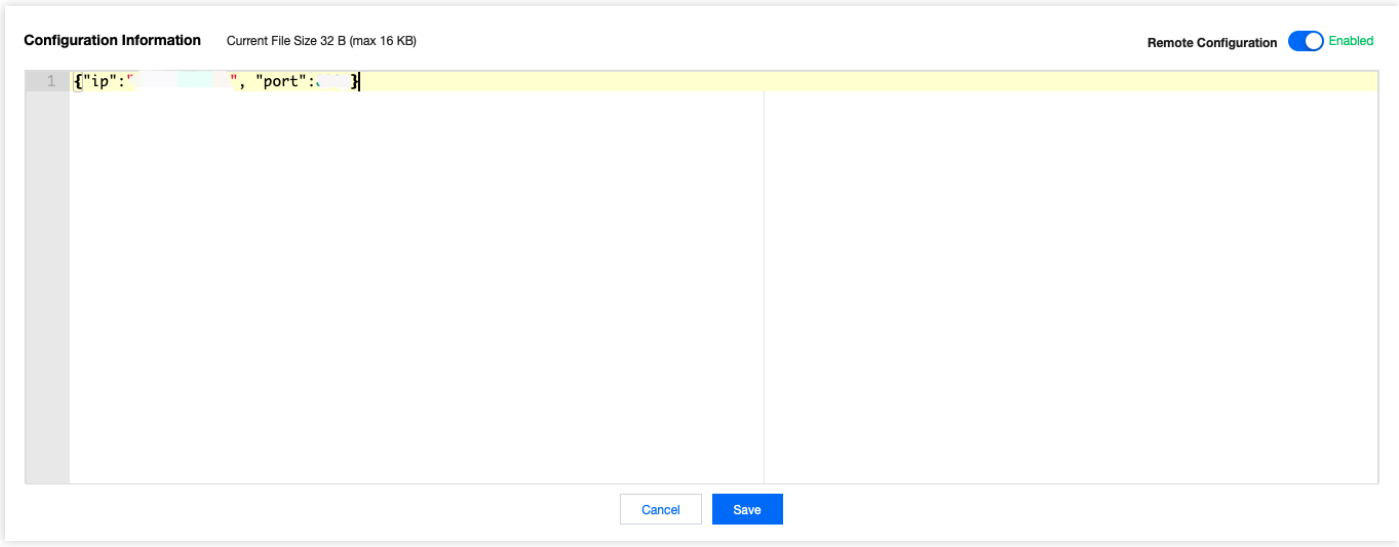
- 远程配置请求 Topic： `$config/operation/${productid}/${devicename}`
- 远程配置订阅 Topic： `$config/operation/result/${productid}/${devicename}`

> 说明：
>
> - `${productID}` ：产品 ID。
> - `${deviceName}` ：设备名称。

物联网平台主动下发

1. 设备端订阅远程配置 Topic。

2. 在 物联网通信控制台 配置界面开启远程配置开关，并输入 JSON 格式的配置信息。



3. 单击【批量下发】，即可将配置信息通过远程配置订阅 Topic 批量下发到该产品下的所有设备中。
   云端通过远程配置订阅 Topic 下发的消息内容格式如下：

```
{"type":"push",
"result":0,
"payload":{yourConfigurationMessage}
}
```

   参数说明：

字段	类型	含义
type	string	物联网平台主动下发时取值 push。 ◦ push：物联网平台主动下发 ◦ reply：设备端主动请求
result	int	错误码。 ◦ 0：成功 ◦ 1001：配置已禁用
payload	string	配置信息内容详情

设备端成功接收到物联网通信下发的配置信息之后，设备端通过调用 SDK 中提供的回调函数获取到配置信息，并将信息更新到设备的系统参数中。此部分更新配置参数逻辑需用户自定义。

设备端主动请求

1. 在控制台配置界面开启远程配置开关，并输入 JSON 格式的配置信息。
2. 设备端订阅远程配置 Topic，并通过 Topic 发送远程配置请求。

3. 云端成功接收到设备请求远程配置信息之后,通过远程配置订阅 Topic 将配置界面的设备配置信息下发到设备端。
   - 设备发送配置请求信息的内容固定如下：

```
{"type":"get"}
```

参数说明：

字段	类型	含义
type	string	设备端主动请求时取值 get

   - 云端通过远程配置订阅 Topic 下发的消息内容格式如下：

```
{"type":"reply",
"result":0,
"payload":{yourConfigurationMessage}
}
```

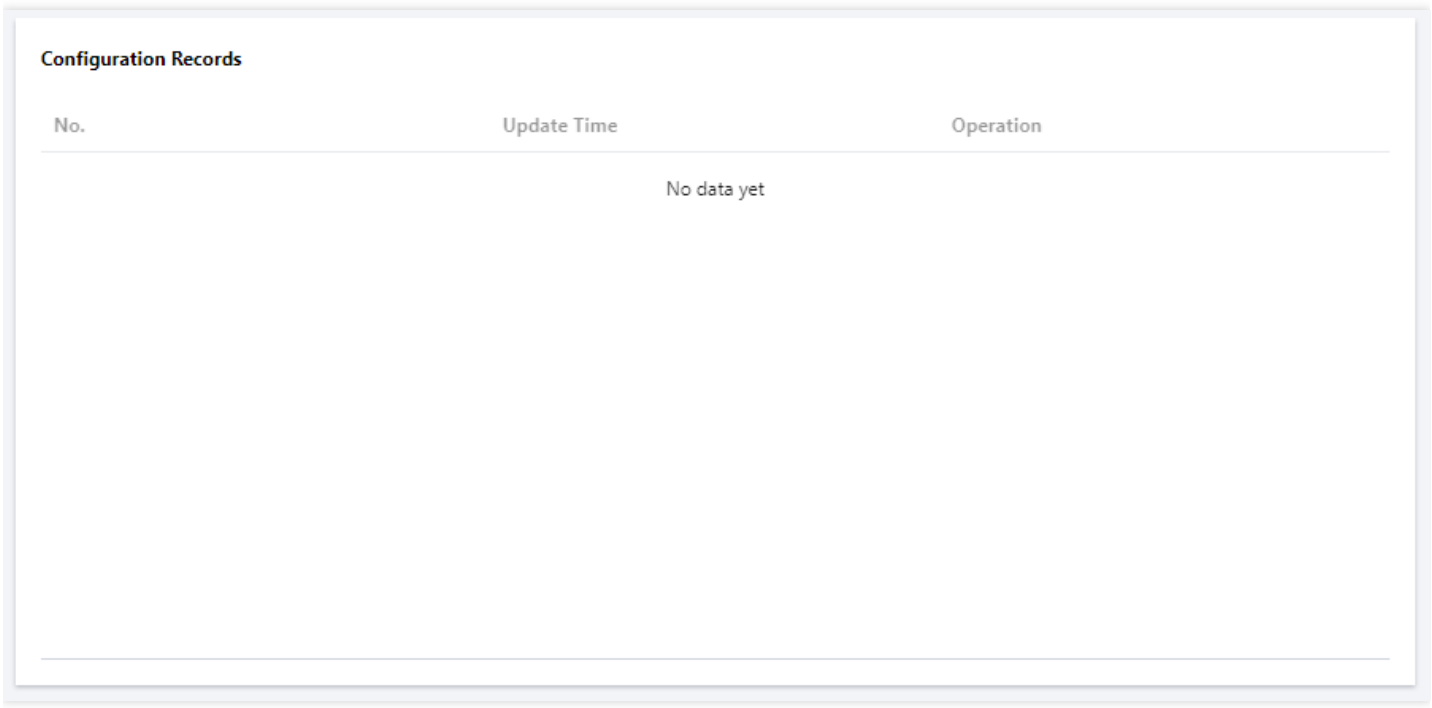
参数说明：

字段	类型	含义
type	string	设备端主动请求时取值 reply。 • push：物联网平台主动下发 • reply：设备端主动请求
result	int	错误码。 • 0：成功 • 1001：配置已禁用
payload	string	配置信息内容详情

4. 设备端接收到数据之后的操作步骤与云端主动下发部分的步骤一致。

配置信息管理

物联网平台提供配置信息管理功能，用户可在控制台查询近五次的配置信息记录。重新编辑并保存配置信息后，上一次的配置信息将显示在配置信息记录中。您可以查看编号、更新时间和配置内容，便于管理配置信息。


Configuration Records
No.
Update Time
Operation
No data yet

资源管理

最近更新时间：2021-08-20 16:27:33

功能概述

资源管理主要是用于设备与平台之间，进行资源互传。实现此类功能需利用如下两个 Topic：

- 数据上行 Topic（用于发布）：`$resource/up/service/${productid}/${devicename}`。
- 数据下行 Topic（用于订阅）：`$resource/down/service/${productid}/${devicename}`。

设备资源上传

步骤1：设备端创建资源上传任务

1. 设备端通过 MQTT 协议发布一条消息到 `$resource/up/service/${productid}/${devicename}`，进行创建设备资源上传任务，消息为 json 格式，内容如下：

```
{
"type":"create_upload_task",
"size":100,
"name":"zxc",
"md5sum":"************",
}
```

2. 创建成功，后台通过 `$resource/down/service/${productid}/${devicename}` 返回资源上传的链接，消息为 json 格式，内容如下：

```
{
"type":"create_upload_task_rsp",
"size":100,
"name":"zxc",
"md5sum":"************",
"url":"https://iothub.cos.ap-guangzhou.myqcloud.com/********"
}
```

步骤2：上报资源上传进度

1. 资源上传使用 HTTP PUT 请求，所以 header 需要添加 MD5 值（base64 编码）。资源上传过程中，设备端通过 `$resource/up/service/${productid}/${devicename}` 上报资源上传进度，消息为 json 格式，内容如下：

```
{
"type":"report_upload_progress",
"name":"zxc",
"progress":{
"state":"uploading",
"percent":89,
"result_code":0,
"result_msg":""
}
}
```

2. 进度上报响应，通过 `$resource/down/service/${productid}/${devicename}` 下发给设备，消息为 json 格式，内容如下：

```
{
"type":"report_upload_progress_rsp",
"result_code":0,
"result_msg":"ok"
}
```

平台资源下发

步骤1：查询资源下载链接

1. 设备端通过 `$resource/up/service/${productid}/${devicename}` 上报消息，查询下载任务，消息为 json 格式，内容如下：

```
{
"type":"get_download_task"
}
```

2. 如果存在下载任务，则通过 `$resource/down/service/${productid}/${devicename}` 下发结果，消息为 json 格式，内容如下：

```
{
"type":"get_download_task_rsp",
"size":372338,
"name":"AAAA",
"md5sum":"a567907174*****3bb9a2bb20716fd97",
"url":"https://iothub.cos.ap-guangzhou.myqcloud.com/********"
}
```

步骤2：上报资源下载进度

1. 资源下载进度通过 `$resource/up/service/${productid}/${devicename}` 进行上报，消息为 json 格式，内容如下：

```
{
"type":"report_download_progress",
"name":"zxc",
"progress":{
"state":"downloading",
"percent":89,
"result_code":0,
"result_msg":""
}
}
```

2. 进度上报响应，通过 `$resource/down/service/${productid}/${devicename}` 下发给设备，消息为 json 格式，内容如下：

```
{
"type":"report_download_progress_rsp",
"result_code":0,
"result_msg":"ok"
}
```

设备日志上报

最近更新时间：2021-08-20 16:27:33

功能概述

设备日志主要用于平台远程查看设备运行日志，平台可通过下发消息，通知设备进行日志上报，日志级别包括错误、警告、信息和调试。实现此功能需利用如下两个 Topic：

- 数据上行 Topic（用于发布）：`$log/operation/${productid}/${devicename}`。
- 数据下行 Topic（用于订阅）：`$log/operation/result/${productid}/${devicename}`。

查询日志级别

1. 设备端通过 MQTT 协议发布一条消息到 `$log/operation/${productid}/${devicename}`，进行查询是否需上传日志，及上传日志级别，消息为 json 格式，内容如下：

```
{
"type": "get_log_level",
"clientToken": "PPXLSKBUPZ-**"
}
```

2. 设备主动查询是否需上报日志，或平台远程通知设备开启日志上报，后台通过 `$log/operation/result/${productid}/${devicename}` 向设备发送是否开启日志上报，及上报的日志级别，消息为 json 格式，内容如下：

```
{
"type": "get_log_level",
"clientToken": "PPXLSKBUPZ-**",
"log_level": 4,
"result": 0,
"timestamp": 1619599073
}
//log_level：0-不上报日志 1-上报错误日志 2-上报警告日志 3-上报信息日志 4-上报调试日志
```

日志上传

参数说明

设备日志上传时需携带 ProductId 和 DeviceName 向平台发起 `http/https` 请求，请求接口及参数如下：

- 请求的 URL 为：
  `http://ap-guangzhou.gateway.tencentdevices.com/device/reportlog`
  `https://ap-guangzhou.gateway.tencentdevices.com/device/reportlog`
- 请求方式：Post

请求参数

参数名称	是否必选	类型	描述
ProductId	是	String	产品 Id
DeviceName	是	String	设备名称
Message	是	Array	上报的日志内容。 字符串数组，每条日志内容前面需要加上日志等级，目前支持 DBG、INF、ERR、WRN

> 说明：
> 接口只支持 application/json 格式。

签名生成

对请求报文进行签名有两种方式，密钥认证使用 HMAC-sha256 算法，证书认证使用 RSA_SHA256 算法，详情请参见 签名方法。

平台返回参数

参数名称	类型	描述
RequestId	String	请求 Id

示例代码

请求包

```
POST https://ap-guangzhou.gateway.tencentdevices.com/device/reportlog
Content-Type: application/json
Host: ap-guangzhou.gateway.tencentdevices.com
```

```
X-TC-Algorithm: HmacSha256
X-TC-Timestamp: 1551****65
X-TC-Nonce: 5456
X-TC-Signature: 2230eefd229f582d8b1b891af7107b91597240707d7****3738f756258d7652c
{"DeviceName":"AAAAAA","Message":["INFmqtt connect success."],"ProductId":"G8N9****HB"}
```

返回包

```
{
"Response": {
"RequestId": "f4da4f1f-d72e-40f1-****-349fc0072ba0"
}
}
```

NTP服务

最近更新时间：2021-08-20 17:04:58

功能概述

NTP 服务主要是解决资源受限的设备，系统不包含 NTP 服务，没有精确时间戳的问题。实现此类功能需利用以下两个 Topic：

- 请求 Topic（用于发布）：`$sys/operation/${ProductId}/${DeviceName}`。
- 响应 Topic（用于订阅）：`$sys/operation/result/${ProductId}/${DeviceName}`。

实现原理

物联网通信平台借鉴 NTP 协议原理，将平台作为 NTP 服务器。设备端向平台请求时，平台返回的 NTP 时间。设备端收到返回后，再结合请求时间和接收时间，一起计算出当前精确时间。

操作步骤

1. 设备端通过 MQTT 协议发布一条消息到 `$sys/operation/${ProductId}/${DeviceName}`，请求平台下发 NTP 时间，同时设备端记录请求时间 deviceSendtime，请求消息为 json 格式，内容如下：

```
{
"type": "get",
"resource": [
"time"
]
}
```

2. 平台通过 `$sys/operation/result/${ProductId}/${DeviceName}` 返回 NTP 时间，同时设备端记录接收时间 deviceRecvtime，返回消息为 json 格式，内容如下：

```
{
"type": "get",
"time": 1621562342,
"ntptime1": 1621562342773,
"ntptime2": 1621562342773
}
```

3. 通过设备端收到的 NTP 时间（${ntptime1} + ${ntptime2}）、接收时间（${deviceRecvtime}）和请求时间（${deviceSendtime}），一起计算精确时间，方法如下：

精确时间 =(${ntptime1} + ${ntptime2} + ${deviceRecvtime} - ${deviceSendtime}) / 2