物联网开发平台

开发者指南

腾讯云

【版权声明】

【商标声明】

【服务声明】

本文档意在向您介绍腾讯云全部或部分产品、服务的当时的相关概况，部分产品、服务的内容可能不时有所调整。您所购买的腾讯云产品、服务的种类、服务标准等应由您与腾讯云之间的商业合同约定，除非双方另有约定，否则，腾讯云对本文档内容不做任何明示或默示的承诺或保证。

【联系我们】

我们致力于为您提供个性化的售前购买咨询服务，及相应的技术售后服务，任何问题请联系 4009100100或95716。

文档目录

开发者指南
- 设备身份认证
  - 概述
  - 产品级密钥认证
  - 设备级密钥认证
  - 动态注册接口说明
- 设备接入协议
  - 设备基于MQTT协议接入
    - 设备基于 TCP 的 MQTT 接入
    - 设备基于 WebSocket 的 MQTT 接入
    - MQTT 持久性会话
- 物模型协议
- 固件升级协议
- 网关子设备
  - 功能概述
  - 拓扑关系管理
  - 代理子设备上下线
  - 代理子设备发布和订阅
  - 子设备固件升级
- 资源管理协议
- 文件管理协议

开发者指南
设备身份认证
概述

最近更新时间：2022-06-10 14:57:39

物联网开发平台（IoT explorer）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，平台提供了两种认证方式，以满足不同的使用场景。
物联网开发平台（IoT explorer）提供以下两种认证方式：

- 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。

两种方案在易用性、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
设备烧录信息	ProductId、Devicename、设备证书、设备私钥。	ProductId、Devicename、设备密钥。
是否需要提前创建设备	必须。	必须。
安全性	高。	一般。
使用限制	单产品下最多创建20万设备。	单产品下最多创建20万设备。
设备资源要求	较高，需要支持 X509 证书解析。	较低。

产品级密钥认证

最近更新时间：2024-03-29 16:05:31

操作场景

物联网开发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 物联网开发平台控制台，创建产品，具体操作请参考 产品定义。
2. 在产品详情页开启动态注册开关。
3. 在控制台产品下设备调试页添加设备，或通过云 API 创建设备。

4. 设备固件烧录，具体步骤如下：

4.1 下载 设备端 SDK 。

4.2 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，并在 SDK 中开启动态注册功能，具体可参见 设备接入 。

4.3 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。

4.4 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。

4.5 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。

4.6 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。

设备级密钥认证

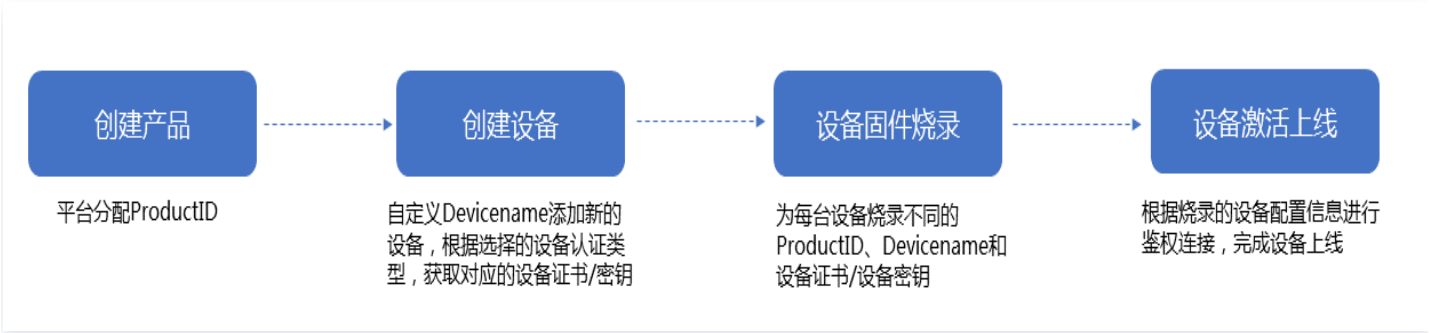
最近更新时间：2022-06-10 14:57:56

操作场景

物联网开发平台支持设备级密钥认证，在该模式下，用户需要为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

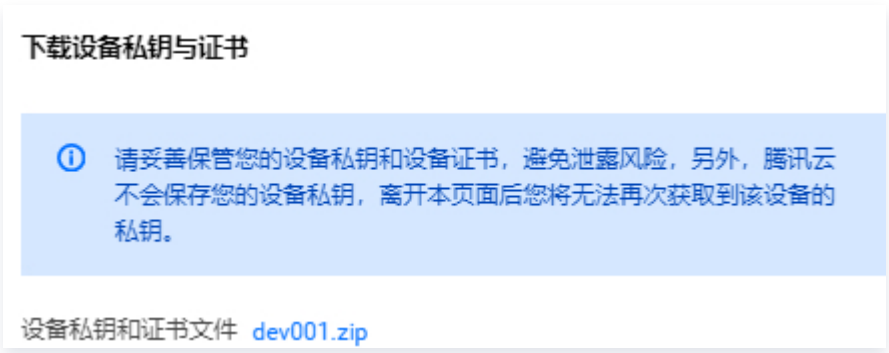
设备级密钥需要为每个设备烧录不同的固件，在产线应用中有一定实现成本，但安全性更高，推荐使用。



操作步骤

1. 登录 物联网开发平台控制台 ，创建产品与设备。具体创建步骤请参考 产品开发 。
2. 在产品详情页获取产品信息，创建设备后将会获得设备私钥和证书文件。
   - 产品信息

- 设备私钥和证书文件



3. 设备固件烧录，具体步骤如下：

3.1 下载 设备端 SDK 。

3.2 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 。

3.3 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。

3.4 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。

3.5 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。

动态注册接口说明

最近更新时间：2023-06-28 16:39:41

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，当产品认证类型为签名认证时有此参数。
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":
"031T01DWAoqFePDt71VuZXuLzkUzbIhGOnvMzpAFtNgOjagyFNHVSostNl9ztvhOuRx0d
MM/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/YYbcmSE12XpeDVmQNDcf1QgVD141RRbmmAnFwQfp1ECAu5O016mC
OvYlJJ6V59yM4OqQSiWphfTg==
```

2. Base64 解码后：

```
b3a141ddad4103f6186dc992135d97a5e0d599034371fd508150f5e354516e69809
c5c107e9d44080bb93b4d7a9823af625249e95e7dc8ce0ea904a25a985f4e
```

3. AES 解密。

产品密钥： `hzvf5LF9S0isvBhDSauWMaIk`

解密后数据： `{"encryptionType":2,"psk":"lDZ6Uqt+l9E0wW7rvDUs7Q=="}`

设备接入协议
设备基于MQTT协议接入
设备基于 TCP 的 MQTT 接入

最近更新时间：2024-01-09 15:06:51

MQTT 协议说明

目前物联网开发平台支持 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：

- `${productId}/${deviceName}/control` 订阅。
- `${productId}/${deviceName}/event` 发布。
- `${productId}/${deviceName}/data` 订阅和发布。
- `$shadow/operation/${productId}/${deviceName}` 发布。通过包体内部 type 来区分：update/get，分别对应设备影子文档的更新和拉取等操作。
- `$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。
- `$ota/report/${productID}/${deviceName}` 发布。设备上报版本号及下载、升级进度到云端。
- `$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 – 900s。若超过1.5倍 KeepAlive 时长物联网平台仍没收到客户端的数据，则平台将断开与客户端的连接。 • ClientId：${productId}${deviceName}，产品 ID 和设备名的组合字符串。 • UserName：`${productId}${deviceName};${sdkappid};${connid};${expiry}`，详情见下文中基于 MQTT 的签名认证接入指引 username 部分； • PassWord：密码（可赋任意值）。
密钥认证	MQTT 服务器连接地址与证书认证一致；端口：1883。	• KeepAlive：保持连接的时间，取值范围为0–900s； • ClientId：${productId}${deviceName}； • UserName：`${productId}${deviceName};${sdkappid};${connid};${expiry}`，详情见下文中基于 MQTT 的签名认证接入指引 username 部分； • PassWord：密码，详情见下文中基于 MQTT 的签名认证接入指引 password 部分。

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网平台的流程如下：

1. 登录 物联网开发平台控制台 。您可在控制台创建产品、添加设备、并获取设备密钥。
2. 按照物联网开发平台约束生成 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，unix时间戳格式如1704363215。expiry应设置为一个远超设备真实生命周期的时间或由设备侧每次获取当前系统时间加上一个较大的整数即可。若expiry 的值小于当前系统时间，MQTT身份认证将失败。
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. 按照物联网开发平台约束生成 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：

```
#!/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
range(length))
# 生成接入物联网平台需要的各参数
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：

```
package com.tencent.iot.hub.device.java.core.sign;

import org.junit.Test;

import javax.crypto.Mac;
```

```
import javax.crypto.spec.SecretKeySpec;
import java.util.*;

import static junit.framework.TestCase.fail;
import static org.junit.Assert.assertTrue;

public class SignForMqttTest {

    @Test
    public void testMqttSign() {
        try {

System.out.println(SignForMqttTest("YourProductId","YourDeviceName","YourPsk"));
            assertTrue(true);
        } catch (Exception e) {
            e.printStackTrace();
            fail();
        }
    }

    public static Map<String, String> SignForMqttTest(String productID, String
devicename, 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.getConnectId(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、根据物联网平台规则生成 password 字段
        String password = HMACSHA256.getSignature(username.getBytes(),
decoder.decode(devicePsk)) + ";hmacsha256";
        Map<String,String> map = new HashMap<>();
        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 < bytes.length; i++) {
            String hex = Integer.toHexString(0xFF & bytes[i]);
            if (hex.length() == 1) {
                sb.append('0');
            }
            sb.append(hex);
        }
        return sb.toString();
    }



    /**
     * 获取连接ID（长度为5的数字字母随机字符串）
     */
    public static String getConnectId(int length) {
        StringBuffer connectId = new StringBuffer();
        for (int i = 0; i < length; i++) {
            int flag = (int) (Math.random() * Integer.MAX_VALUE) % 3;
```

```
            int randNum = (int) (Math.random() * Integer.MAX_VALUE);
            switch (flag) {
                case 0:
                    connectId.append((char) (randNum % 26 + 'a'));
                    break;
                case 1:
                    connectId.append((char) (randNum % 26 + 'A'));
                    break;
                case 2:
                    connectId.append((char) (randNum % 10 + '0'));
                    break;
            }
        }

        return connectId.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) => {
    len = len || 32;
    var chars = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789';
    var maxPos = chars.length;
    var pwd = '';
    for (let i = 0; i < 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};${connid};${expiry}
const userName = `${clientId};12010126;${connid};${expiry}`;
//5. 对 username 进行签名，生成token、根据物联网平台规则生成 password 字段
const rawKey = crypto.enc.Base64.parse(devicePsk);   // 对设备密钥进行base64解码
const token =  crypto.HmacSHA256(userName, rawKey);
const password = token.toString(crypto.enc.Hex) + ";hmacsha256";
console.log(`userName:${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 < 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 < 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_LENGTH, &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 <= 0 || MAX_ACCESS_EXPIRE_TIMEOUT <= 0) {
        cur_timestamp = LONG_MAX;
    }

    // 20 for timestampe length & 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网平台。

设备基于 WebSocket 的 MQTT 接入

最近更新时间：2024-01-02 10:50:51

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：2024-01-02 10:50:51

物联网开发平台支持 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

设备连接物联网平台时，可将 Connect 连接报文可变报头部分的 CleanSeesion 标志位设置为0。物联网平台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。

物联网平台响应说明

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：**
> - 物联网平台发送存储的 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*7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。

- 设备连接物联网平台时，将 Connect 连接报文可变报头部分的 CleanSeesion 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。

物模型协议

最近更新时间：2023-11-07 19:21:52

简介

物模型可将物理世界中的设备功能进行数字化定义，便于应用更便利的管理设备。平台为用户提供了基于物模型的业务协议，既可以满足智慧生活场景应用，也可满足物联网各垂直行业应用需求。

- 智慧生活场景：基于物模型协议，用户将设备相关属性、事件等上报云端后，可无缝使用腾讯连连小程序或自主品牌小程序与 App，无需处理云端与小程序或 App 的通信细节，以提升用户在智慧生活场景下的应用开发效率。
- 垂直行业应用场景：基于物模型协议，无需用户解析设备数据，可使用物联网开发平台的数据分析、告警和存储服务及腾讯云相关云产品，以提升垂直行业应用的开发效率。

物模型协议

概述

产品定义物模型后，设备可以根据物模型中的定义上报属性、事件，并可对设备下发控制指令。物模型的管理详见 产品定义 。物模型协议包括了以下几部分。

- 设备属性上报：设备端将定义的属性根据设备端的业务逻辑向云端上报。
- 设备远程控制：从云端向设备端下发控制指令，即从云端设置设备的可写属性。
- 获取设备最新上报信息：获取设备最新的上报数据。
- 设备事件上报：设备可根据定义的物模型中的事件，当事件被触发，则根据设备事件上报的协议上报告警、故障等事件信息。
- 设备行为调用：云端可以通过 RPC 的方式通知设备执行某个动作行为，适用于应用需要实时获取设备的执行结果的场景。
- 设备初始信息上报：设备连接平台时上报的初始信息，便于小程序或 App 展示设备详细信息，如设备 MAC 地址、IMEI 号。
- 用户删除设备：用户在腾讯连连小程序或用户自主品牌小程序删除设备时由云端发送给设备的通知消息，便于设备重置或网关类设备清除子设备数据。

设备属性上报

1. 当设备需要向云端上报设备运行状态的变化时，以通知应用端小程序、App 实时展示或云端业务系统接收设备上报属性数据，物联网开发平台为设备设定了默认的 Topic：
   - 设备属性上行请求 Topic： `$thing/up/property/{ProductID}/{DeviceName}`
   - 设备属性下行响应 Topic： `$thing/down/property/{ProductID}/{DeviceName}`
2. 请求。
   - 设备端请求报文示例：

```
{
   "method":"report",
   "clientToken":"123",
   "timestamp":1628646783,
   "params":{
      "power_switch":1,
      "color":1,
      "brightness":32
     }
}
```

- 请求参数说明：

参数	类型	说明
method	String	report 表示设备属性上报。
clientToken	String	用于上下行消息配对标识。
timestamp	Integer	属性上报的时间，格式为 UNIX 系统时间戳，不填写该字段表示默认为当前系统时间。单位为毫秒。
params	JSON	JSON 结构内为设备上报的属性值。
params.power_switch	Boolean	布尔型属性的值一般为0或1。
params.color	Enum	枚举整型属性的值为整数值，数值类型填写错误或超过枚举项定义范围出现406返回码，表示物模型格式校验错误。
params.brightness	Integer	整数型属性的值为整数值，数值类型填写错误或超过数值范围会出现406返回码，表示物模型格式校验错误。

3. 响应。

- 云端返回设备端报文示例：

```
{
   "method":"report_reply",
   "clientToken":"123",
   "code":0,
   "status":"some message where error"
}
```

- 响应参数说明：

参数	类型	说明
method	String	report_reply 表示云端接收设备上报后的响应报文。
clientToken	String	用于上下行消息配对标识。
code	Integer	0表示云端成功收到设备上报的属性。
status	String	当 code 非0的时候，提示错误信息。

设备远程控制

1. 使用物模型协议的设备，当需要通过云端控制设备时，设备需订阅下发 Topic 接收云端指令：
   - 下发 Topic： `$thing/down/property/{ProductID}/{DeviceName}`
   - 响应 Topic： `$thing/up/property/{ProductID}/{DeviceName}`
2. 请求。
   - 远程控制请求消息格式：

```
{
  "method": "control",
  "clientToken": "123",
  "params": {
    "power_switch": 1,
    "color": 1,
    "brightness": 66
  }
}
```

   - 请求参数说明：

参数	类型	说明
method	String	control 表示云端向设备发起控制请求。
clientToken	String	用于上下行消息配对标识。
params	JSON	JSON 结构内为设备属性的设置值，可写的属性值才可控制成功。

3. 响应。
   - 设备响应远程控制请求消息格式：

```
{
```

```
    "method":"control_reply",
    "clientToken":"123",
    "code":0,
    "status":"some message where error"
}
```

- 响应参数说明：

参数	类型	说明
method	String	表示设备向云端下发的控制指令的请求响应。
clientToken	String	用于上下行消息配对标识。
code	Integer	0表示设备成功接收到云端下发的控制指令。
status	String	当 code 非0的时候，提示错误信息。

获取设备最新上报信息

1. 设备从云端接收最新消息使用的 Topic：
   - 请求 Topic： `$thing/up/property/{ProductID}/{DeviceName}`
   - 响应 Topic： `$thing/down/property/{ProductID}/{DeviceName}`
2. 请求。
   - 请求消息格式：

```
{
    "method": "get_status",
    "clientToken": "123",
    "type" : "report",
    "showmeta": 0
}
```

   - 请求参数说明：

参数	类型	说明
method	String	get_status 表示获取设备最新上报的信息。
clientToken	String	消息 Id，回复的消息将会返回该数据，用于请求响应消息的对比。
type	String	表示获取什么类型的信息。report 表示设备上报的信息。

showmeta	Integer	标识回复消息是否带 metadata，缺省为0表示不返回 metadata。

3. 响应。
   - 响应消息格式：

```
{
  "method": "get_status_reply",
  "code": 0,
  "clientToken": "123",
  "type": "report",
  "data": {
    "reported": {
      "power_switch": 1,
      "color": 1,
      "brightness": 66
    }
  }
}
```

   - 响应参数说明：

参数	类型	说明
method	String	表示获取设备最新上报信息的 reply 消息。
code	Integer	0标识云端成功收到设备上报的属性。
clientToken	String	消息 Id，回复的消息将会返回该数据，用于请求响应消息的对比。
type	String	表示获取什么类型的信息。report 表示设备上报的信息。
data	JSON	返回具体设备上报的最新数据内容。

设备事件上报

1. 当设备需要向云端上报事件时，如上报设备的故障、告警数据，平台为设备设定了默认的 Topic：
   - 设备事件上行请求 Topic：`$thing/up/event/{ProductID}/{DeviceName}`
   - 设备事件下行响应 Topic：`$thing/down/event/{ProductID}/{DeviceName}`
2. 请求。
   - 设备端请求报文示例：

```
{
   "method":"event_post",
   "clientToken":"123",
   "version":"1.0",
   "eventId":"PowerAlarm",
   "type":"fault",
   "timestamp":1212121221,
   "params":{
      "Voltage":2.8,
      "Percent":20
    }
}
```

- 请求参数说明：

参数	类型	说明
method	String	event_post 表示事件上报。
clientToken	String	消息 ID，回复的消息将会返回该数据, 用于请求响应消息的对比。
version	String	协议版本，默认为1.0。
eventId	String	事件的 Id，在物模型事件中定义。
type	String	事件类型。 • info：信息。 • alert：告警。 • fault：故障。
params	String	事件的参数，在物模型事件中定义。
timestamp	Integer	事件上报的时间，不填写该字段表示默认为当前系统时间。单位为毫秒。

3. 响应。

- 响应消息格式：

```
{
   "method": "event_reply",
   "clientToken": "123",
   "version": "1.0",
   "code": 0,
   "status": "some message where error",
```

```
    "data": {}
}
```

- 响应参数说明：

参数	类型	说明
method	String	event_reply 表示是云端返回设备端的响应。
clientToken	String	消息 Id，回复的消息将会返回该数据，用于请求响应消息的对比。
version	String	协议版本，默认为1.0。
code	Integer	事件上报结果，0表示成功。
status	String	事件上报结果描述。
data	JSON	事件上报返回的内容。

设备行为调用

1. 当应用通过云端向设备发起某个行为调用时，平台为设备行为的处理设定了默认的 Topic：
   - 应用调用设备行为 Topic： `$thing/down/action/{ProductID}/{DeviceName}`
   - 设备响应行为执行结果 Topic： `$thing/up/action/{ProductID}/{DeviceName}`
2. 请求。
   - 应用端发起设备行为调用报文示例：

```
{
    "method": "action",
    "clientToken": "20a4ccfd-d308-****-86c6-5254008a4f10",
    "actionId": "openDoor",
    "timestamp": 1212121221,
    "params": {
        "userid": "323343"
    }
}
```

   - 请求参数说明：

参数	类型	说明
method	String	action 表示是调用设备的某个行为。

clientToken	String	消息 Id，回复的消息将会返回该数据，用于请求响应消息的对比。
actionId	String	actionId 是物模型中的行为标识符，由开发者自行根据设备的应用场景定义。
timestamp	Integer	行为调用的当前时间，不填写则默认为调用行为的当前系统时间，单位为毫秒。
params	String	行为的调用参数，在物模型的行为中定义。

3. 响应。
   - 响应消息格式：

```
{
   "method": "action_reply",
   "clientToken": "20a4ccfd-d308-11e9-86c6-5254008a4f10",
   "code": 0,
   "status": "some message where error",
   "response": {
      "Code":  0
    }
}
```

   - 响应参数：

参数	类型	说明
method	String	action_reply 表示是设备端执行完指定的行为向云端回复的响应。
clientToken	String	消息 Id，回复的消息将会返回该数据，用于请求响应消息的对比。
code	Integer	行为执行结果，0表示成功。
status	String	行为执行失败后的错误信息描述。
response	JSON	设备行为中定义的返回参数，设备行为执行成功后，向云端返回执行结果。

设备基础信息上报

1. 小程序或 App 展示设备详细信息时，一般会展示设备的 MAC 地址、IMEI 号、时区等基础信息。设备信息上报使用的 Topic：

- 上行请求 Topic： `$thing/up/property/{ProductID}/{DeviceName}`
- 下行响应 Topic： `$thing/down/property/{ProductID}/{DeviceName}`

2. 请求。

- 设备端请求报文示例：

```
{
  "method": "report_info",
  "clientToken": "1234567",
  "params": {
    "name": "dev001",
    "imei": "ddd",
    "module_hardinfo": "ddd",
    "mac": "ddd",
    "device_label": {
      "append_info": "ddddd"
    }
  }
}
```

- 请求参数说明：

参数	类型	说明
method	String	report_info 表示设备基础信息上报。
clientToken	String	用于上下行消息配对标识。
imei	String	设备的 IMEI 号信息，非必填项。
mac	String	设备的 MAC 信息，非必填项。
module_hardinfo	String	模组具体硬件型号。
append_info	String	设备商自定义的产品基础信息，以 KV 方式上报。

3. 响应。

- 云端返回设备端报文示例：

```
{
  "method":"report_info_reply",
  "clientToken":"1234567",
  "code":0,
  "status":"success"
```

```
}
```

- 响应参数说明：

参数	类型	说明
method	String	report_reply 表示云端接收设备上报后的响应报文。
clientToken	String	用于上下行消息配对标识。
code	Integer	0 表示云端成功收到设备上报的属性。
status	String	当 code 非0的时候，提示错误信息。

用户删除设备

1. 当用户在小程序或 App 中删除已绑定成功的设备，平台会下发用户删除设备的通知到设备，设备接收后可根据业务需求自行处理。如网关类设备接收到子设备被删除。

   下发用户删除设备 Topic： `$thing/down/service/{ProductID}/{DeviceName}`
2. 请求。
   - 应用端发起用户删除设备报文示例：

```
{
  "method": "unbind_device",
  "clientToken": "20a4ccfd-****-11e9-86c6-5254008a4f10",
  "timestamp": 1212121221
}
```

   - 请求参数说明：

参数	类型	说明
method	String	unbind_device 表示是用户在小程序或 App 中删除或解绑某个设备。
timestamp	Integer	用户删除设备的系统时间戳。

用户绑定设备通知消息

1. 当用户在小程序或 App 中成功绑定设备后，平台会下发设备已被用户绑定的通知消息到设备，设备接收后可根据业务需求自行处理。该消息用于手机应用端通知设备端，无需设备端回复。

   下发用户绑定设备通知消息 Topic： `$thing/down/service/{ProductID}/{DeviceName}`

2. 请求。
   - 应用端发起用户绑定设备通知消息报文示例：

```
{
  "method": "bind_device",
  "clientToken": "20a4ccfd-****-11e9-86c6-5254008a4f10",
  "timestamp": 1212121221
}
```

   - 请求参数说明：

参数	类型	说明
method	String	bind_device 表示是用户在小程序或 App 中绑定某个设备。
clientToken	String	用于上下行消息配对标识。
timestamp	Integer	用户绑定设备的系统时间戳。

位置服务围栏告警消息下发

1. 当用户在 控制台 位置服务功能或者小程序、App 中为设备创建并关联了地理电子围栏，设备触发围栏告警条件时，平台会下发围栏告警消息通知到设备，设备接收后可根据业务需求自行处理，如设备收到围栏告警消息，语音播报提醒设备的使用用户或者管理者注意安全。
   - 下发围栏告警消息Topic： `$thing/down/service/{ProductID}/{DeviceName}`
   - 设备响应回复 Topic： `$thing/up/service/{ProductID}/{DeviceName}`
2. 云端下发围栏告警消息。
   - 云端下发告警消息报文示例：

```
{
  "method":"alert_fence_event",
  "clientToken":"xx",
  "timestamp": 0,
  "data": {
    "alert_type":"xx", //事件，In Out InOrOut
    "alert_condition":"xx", //设备绑定围栏的触发条件 In Out InOrOut
    "alarm_time": 0, // 告警时间
    "fence_name":"xx", // 围栏名称
    "long": 0,
    "lat": 0
  }
}
```

- 请求参数说明：

参数	类型	说明
method	String	alert_fence_event 表示围栏告警事件。
clientToken	String	用于上下行消息配对标识。
timestamp	Integer	时间戳，单位为毫秒。
data.alert_type	String	告警事件类型：In 、Out 、InOrOut。
data.alert_condition	String	设备绑定围栏的触发条件In 、Out 、InOrOut。
data.alarm_time	Int	告警时间。
data.fence_name	String	围栏名称。
data.long	Float	经度。
data.lat	Float	纬度。

3. 设备端回复。

- 设备端返回云端报文示例：

```
{
   "method":"alert_fence_event_reply",
   "clientToken":"xx",
   "timestamp": 0,
   "code":0,
   "status":"success"
}
```

- 响应参数说明：

参数	类型	说明
method	String	alert_fence_event_reply 表示围栏告警回复。
clientToken	String	用于上下行消息配对标识。
timestamp	Int	时间戳。
code	Int	0 表示已经正确处理。

status	String	当 code 非0时，提示错误信息。

错误码

code	说明
400	报文格式非 JSON 格式。
403	错误的 method 标识符或属性、事件、行为标识符与物模型定义的标识符不一致。
405	时间戳错误，当前时间和上报时间相差24小时，注意时间戳是毫秒。
406	物模型参数输入值数据类型错误或数据超出定义范围。
503	系统内部错误。

固件升级协议

最近更新时间：2022-06-10 14:58:32

操作场景

设备固件升级又称 OTA，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 OTA 服务快速的进行固件升级。

实现原理

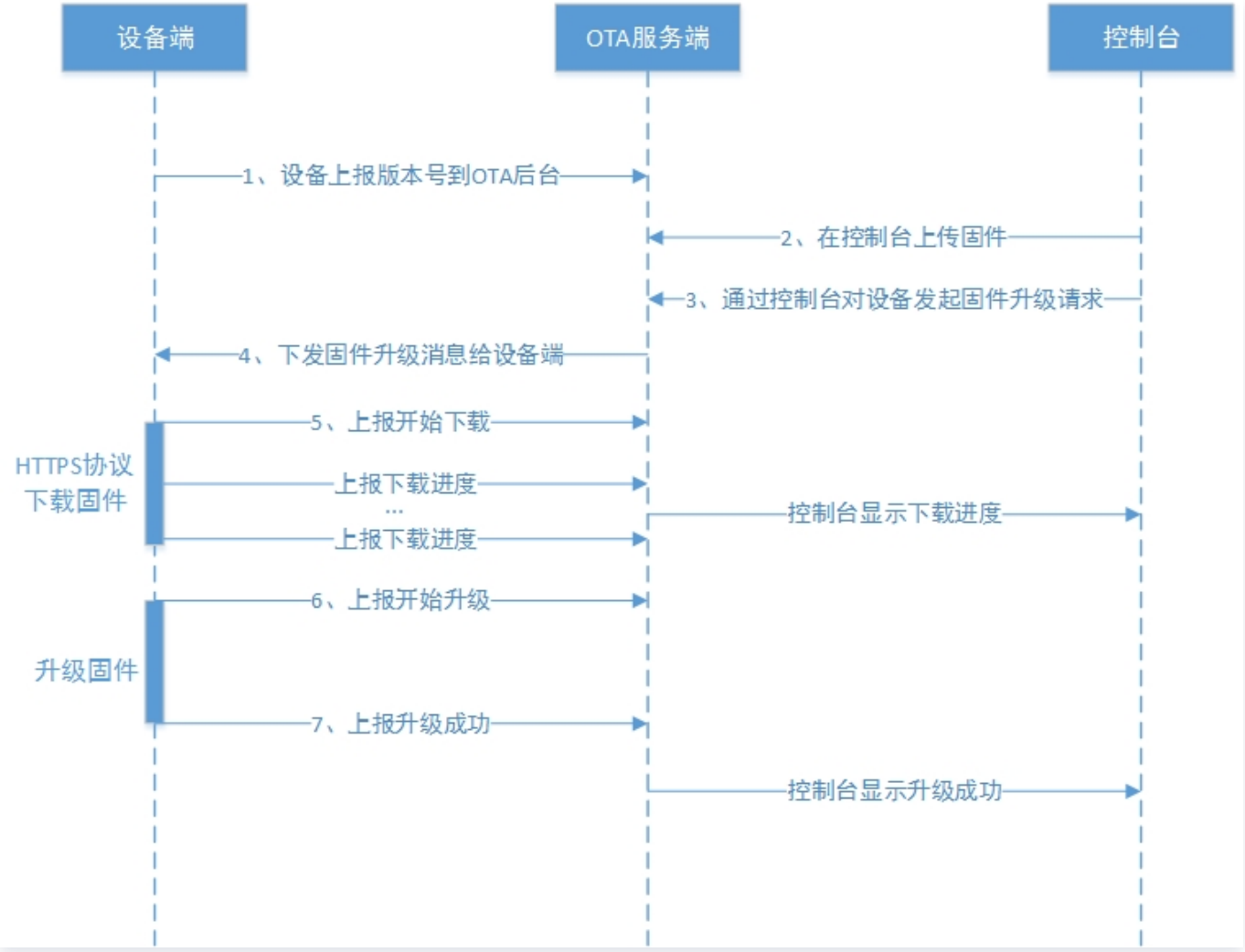
固件升级的过程中，需要设备订阅下面两个 Topic 来实现与云端的通信，如下图所示：

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

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. 然后您可以在控制台上传固件。
3. 在控制台将指定的设备升级到指定的版本。
4. 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": "36eb5951179db14a631463a37a9322a2",
   "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
```

5. 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
```

6. 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
```

7. 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
```

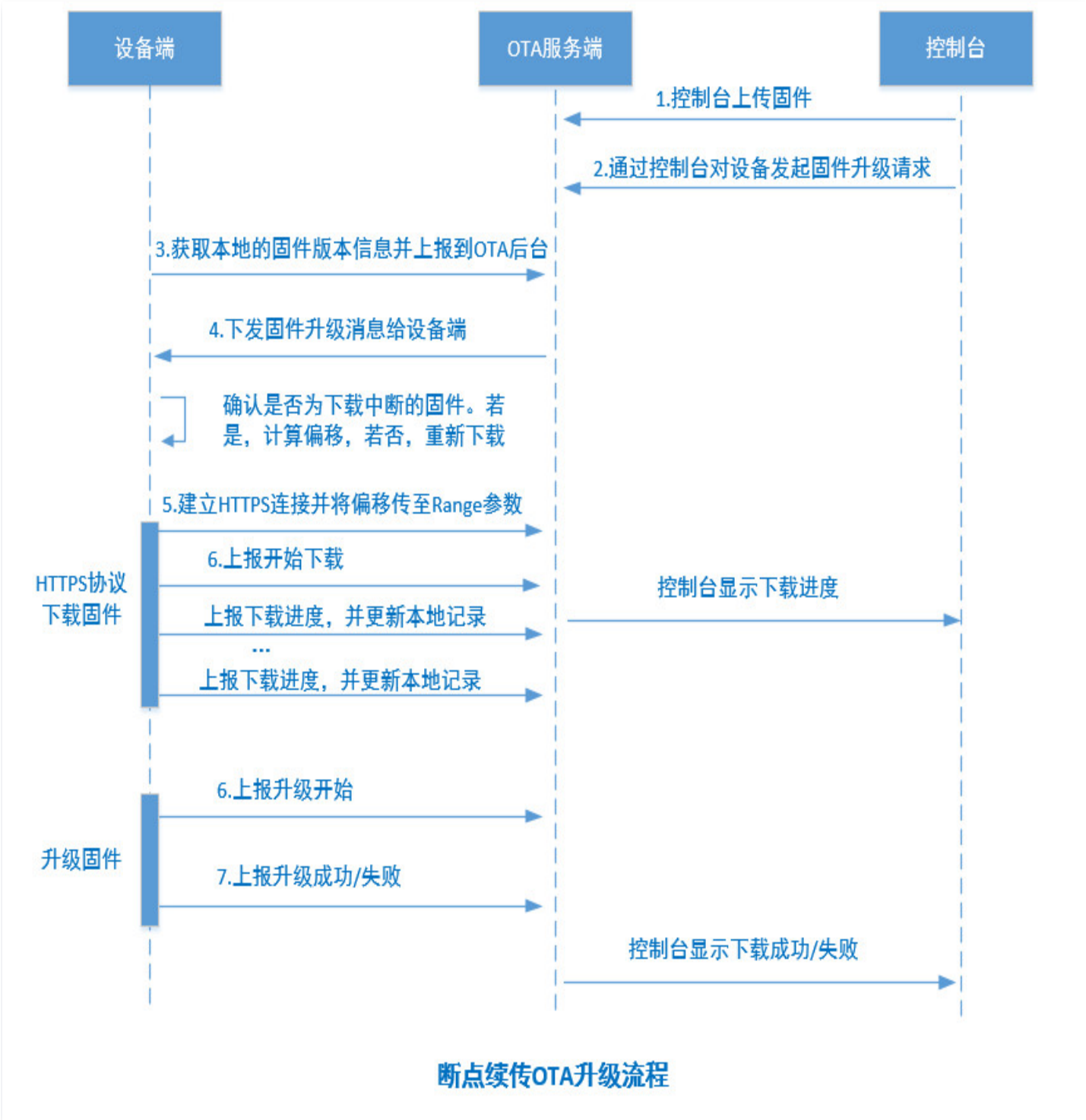
OTA 断点续传

物联网设备有部分场景处于弱网环境，在这个场景下连接会不稳定，固件下载会中断的情况出现。如果每次都从0偏移开始下载固件，则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。

- 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，设备获取到升级的目标固件的信息与本地记录的中断的固件信息比较，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：**
> 弱网环境下第3步到第6步有可能会多次执行，没有执行第7步，执行第3步，设备端都会收到第4步的消息。



断点续传OTA升级流程

网关子设备
功能概述

最近更新时间：2022-06-10 14:58:45

设备分类

物联网开发平台根据设备功能性的不同将设备分为如下三类（即节点的分类）：

- **普通设备**：此类设备可直接接入物联网开发平台且无挂载子设备。
- **网关设备**：此类设备可直接接入物联网开发平台，并且可接受子设备加入局域网络。
- **子设备**：此类设备必须依托网关设备才可与物联网开发平台进行通信，例如 Zigbee、蓝牙、RF433 等设备。

操作场景

- 对于不具备直接接入因特网的设备，可先接入本地网关设备的网络，利用网关设备与云端的通信功能，代理子设备接入物联网开发平台。
- 对于在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子设备的管理。

接入方式

- 网关设备接入物联网开发平台的方式与普通设备一致。网关设备接入之后可代理同一局域网下的子设备上/下线，代理子设备上报数据，代理子设备接收云端下发给子设备的数据，并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 ProductSecret（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：2023-07-31 15:38:42

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 返回子设备的绑定结果信息。
网关绑定子设备请求数据格式：

```
{
 "type": "bind",
 "payload": {
  "devices": [
   {
    "product_id": "CFC******AG7",
    "device_name": "subdeviceaaaa”,
    "signature": "signature",
    "random": 121213,
    "timestamp": 1589786839,
    "signmethod": "hmacsha256",
    "authtype": "psk"
   }
  ]
 }
}
```

网关绑定子设备响应数据格式：

```
{
 "type": "bind",
 "payload": {
  "devices": [
   {
    "product_id": "CFC******AG7",
```

```
      "device_name": "subaaa",
      "result": -1
    }
  ]
 }
}
```

请求参数说明：

字段	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。
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};${random};${timestamp} 2. 使用设备 Psk 密钥，或者证书的 Sha1 摘要，进行签名：`base64_encode(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。

响应参数说明：

字段	类型	描述
type	String	网关消息类型。绑定子设备取值为：`bind`。
payload.devi	Array	要绑定的子设备列表。

ces		
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，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": "CFC******AG7",
    "device_name": "subaaa"
   }
  ]
 }
}
```

网关解绑子设备响应数据格式：

```
{
 "type": "unbind",
 "payload": {
  "devices": [
   {
    "product_id": "CFC******AG7",
    "device_name": "subaaa",
    "result": -1
   }
  ]
 }
}
```

请求参数说明：

字段	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。

响应参数说明：

字段	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，具体 错误码 见下表。

通知网关开启搜索状态

在应用端小程序或app需要进入某个子设备绑定流程内，平台会通知网关开启和关闭搜索子设备功能，协议如下：

- 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}`

平台下发数据格式：

```
{
  "type": "search_devices",
  "payload": {
  "status": 0 //0-关闭 1-开启
  }
}
```

请求参数说明：

字段	类型	描述
type	String	网关消息类型。通知网关开启搜索状态取值为：search_devices。

status	Int	网关搜索状态： • 0：关闭。 • 1：开启。

网关回复数据格式：

```
{
  "type": "search_devices",
  "payload": {
  "status": 0, //0-关闭 1-开启
  "result": 0
  }
}
```

字段	类型	描述
type	String	网关消息类型。通知网关开启搜索状态取值为：search_devices。
status	Int	网关搜索状态： • 0：关闭。 • 1：开启。
result	Int	网关响应处理结果： • 0：成功。 • 1：失败。

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。
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， 可以通过该数据下行 Topic 订阅平台对子设备的拓扑关系变化。
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。
-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：2024-03-15 14:26:21

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 返回子设备的上线结果信息。
网关代理子设备上线请求数据格式：

```
{
 "type": "online",
 "payload": {
  "devices": [
   {
    "product_id": "CFC******AG7",
    "device_name": "subdeviceaaaa"
   }
  ]
 }
}
```

代理子设备上线响应数据格式：

```
{
 "type": "online",
 "payload": {
  "devices": [
   {
    "product_id": "CFC******AG7",
    "device_name": "subdeviceaaaa",
    "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": "CFC******AG7",
    "device_name": "subdeviceaaaa"
   }
  ]
 }
}
```

网关代理子设备下线响应数据格式：

```
{
 "type": "offline",
 "payload": {
  "devices": [
   {
    "product_id": "CFC******AG7",
    "device_name": "subdeviceaaaa",
    "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。
-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：2022-06-10 14:59:11

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 网关设备接入 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

网关产品与子产品建立绑定，获取子设备的 Topic 权限后，网关设备可以使用子设备Topic代理进行收发消息，同时可以在设备调试-设备日志中查看通信信息。

子设备固件升级

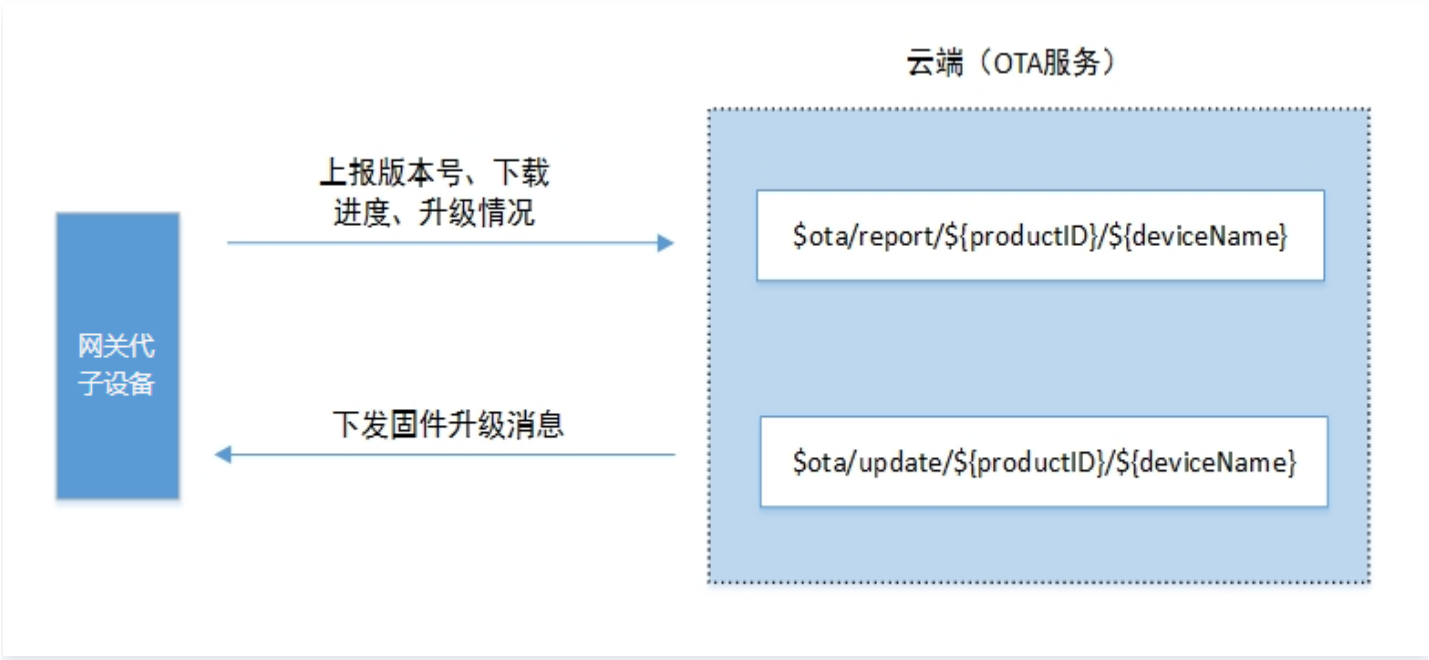
最近更新时间：2022-06-10 14:59:19

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：



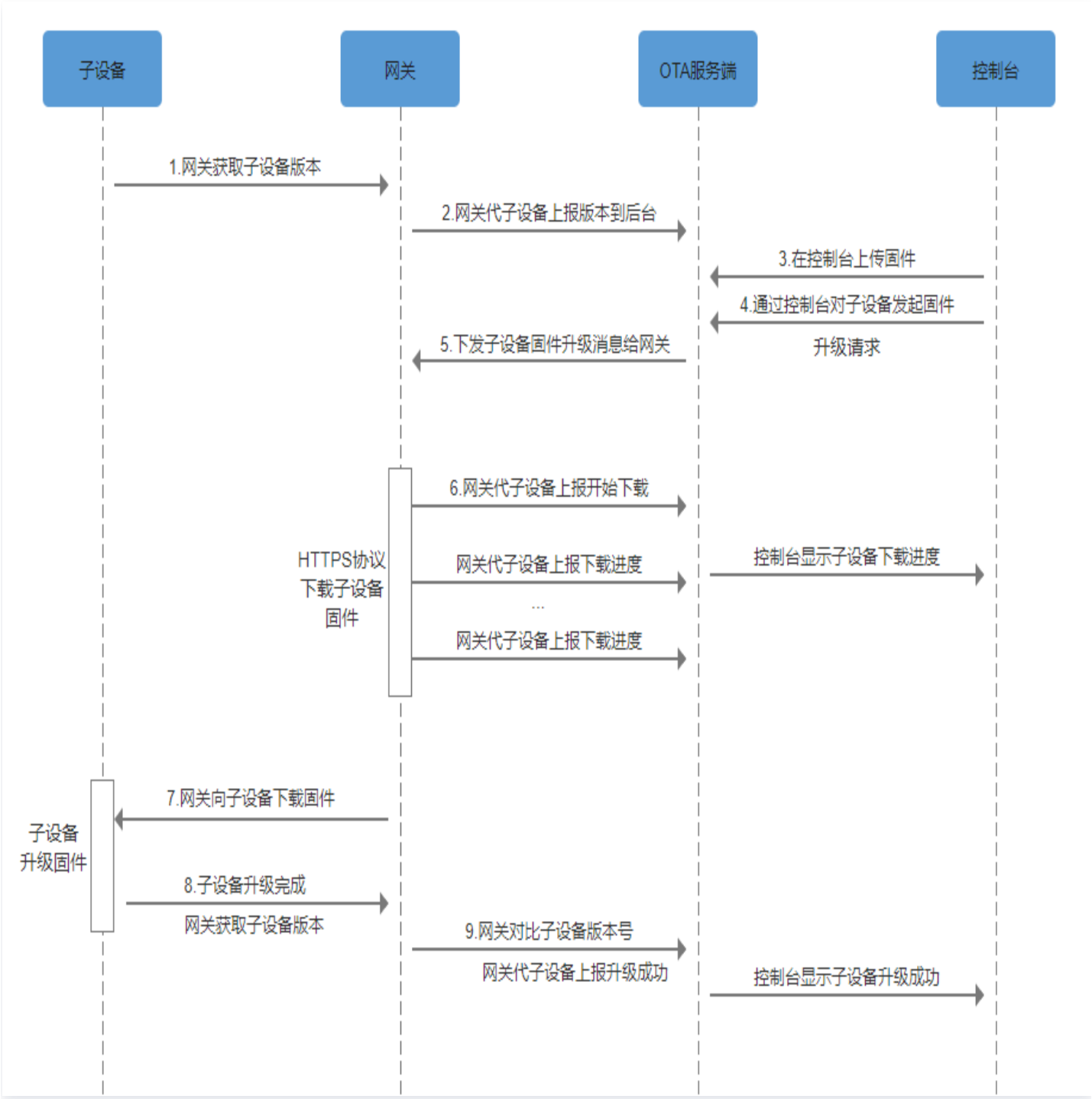
示例代码如下：

```
$ota/report/${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 。


子设备
网关
OTA服务端
控制台
1.网关获取子设备版本
2.网关代子设备上报版本到后台
3.在控制台上传固件
4.通过控制台对子设备发起固件
升级请求
5.下发子设备固件升级消息给网关
6.网关代子设备上报开始下载
HTTPS协议
下载子设备
固件
网关代子设备上报下载进度
控制台显示子设备下载进度
...
网关代子设备上报下载进度
7.网关向子设备下载固件
子设备
升级固件
8.子设备升级完成
网关获取子设备版本
9.网关对比子设备版本号
网关代子设备上报升级成功
控制台显示子设备升级成功

资源管理协议

最近更新时间：2022-06-10 14:59:28

功能概述

资源管理主要是用于开发者向设备端下发人脸识别库、图片库、音乐库等标准的设备资源，实现平台与设备间资源内容的上传及下载。
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":"file",
    "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":"file",
    "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":"file",
    "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":"file",
    "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"
}
```

步骤3：上报资源下载成功

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":"file",
    "progress":{
        "state":"done",
        "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"
}
```

文件管理协议

最近更新时间：2023-11-03 10:39:41

功能概述

文件管理功能是用于厂商去完成设备端、平台侧与应用端间的文件互传，用于存储设备量产后用户在使用设备过程中所需文件，由于该部分文件权限不归属于设备厂商查看管理，因此由服务端文件管理 topic 消息通道进行实现。
应用场景举例：

- 用户在小程序端录音，需下发到设备端进行播报。
- 门禁摄像头在有人来访时，拍取来访人照片，在小程序端进行查看。

实现此类功能需利用如下两个 Topic：

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

设备文件版本信息上报

1. 设备上报当前文件版本信息，设备端通过 MQTT 协议发布一条消息到 `$thing/up/service/${productid}/${devicename}` ，进行版本号的上报，消息为 json 格式，内容如下：

```
{
   "method": "report_version",
   "request_id": "12345678",
   "report":{"resource_name": "123.wav", "version": "1.0.0", "resource_type":
"FILE"}
}
//method：消息类型
//resource_name: 文件名称
//version：文件版本号
//resource_type： 文件类型，固件（fw）、文件（file）
//后台逻辑: 接收消息，并将文件的版本信息更新到对应产品/设备的
```

特别的，若上报的文件列表为空，则云端会回复云端记录的设备的所有文件列表，设备端可基于此特性，实现设备端文件列表的异常恢复操作。

```
{
   "method": "report_version",
   "report":{
      "resource_list":[]
    }
}
```

2. 服务端收到文件版本信息上报后，服务端通过 `$thing/down/service/${productid}/${devicename}` ，向设备端回复收到的版本信息，消息为 json 格式，内容如下：

```
{
   "method": "report_version_rsp",
   "result_code":0,
   "result_msg":"success",
   "resource_list":[
      {"resource_name": "audio_woman_mandarin", "version":
"1.0.0", "resource_type": "FILE"},
      {"resource_name": "audio_woman_sichuanhua", "version":
"2.0.0", "resource_type": "FILE"}
   ]
}
//method：消息类型
//result_code: 版本上报结果
//result_msg：版本上报结果信息
//resource_list: 将收到的版本信息回送过来
//若设备端上报的resource_list为空，则服务端回应已记录的资源列表
```

设备文件下载

1. 用户在小程序端使用时调用应用端 API 上传文件，创建文件下载任务。
2. 设备端会通过订阅的 `$thing/down/service/${productid}/${devicename}` 接收文件的更新消息，文件更新消息内容如下：

```
{
   "method": "update_resource",
   "resource_name": "audio_woman_sichuanhua",
   "resource_type": "FILE",
   "version": "1.0.0",
   "url": "https://ota-1254092559.cos.ap-guangzhou.myqcloud.com",
   "md5sum": "cc03e747a6afbbcbf8be7668acfebee5",
   "file_size": 31242
}
// method：消息类型
// resource_name：文件名称
// resource_type：固件（fw）、文件（file），控制台下拉选择
// version：升级版本
// url：下载文件的url
// md5asum：文件的MD5值
// file_size：文件大小，单位为字节
```

3. 设备在收到文件更新的消息后，根据 URL 下载资源，下载的过程中设备 SDK 会通过 `$thing/up/service/${productid}/${devicename}` 不断的上报下载进度，上报的内容如下：

```
{
   "method": "report_progress",
   "report":{
      "progress": {
         "resource_name": "audio_woman_sichuanhua",
         "state":"downloading",
         "percent":"10",
         "result_code":"0",
         "result_msg":""
      },
    "version": "1.0.0"
   }
}
// method：消息类型
//resource_name:正在下载的文件名称
// state：状态为正在下载中
// percent：当前下载进度，百分比
```

4. 当设备完成文件下载，设备需要通过 `$thing/up/service/${productid}/${devicename}` 上报一条下载的结果，内容如下：

```
//下载成功
{
   "method": "report_result",
   "report":{
      "progress":{
         "resource_name": "audio_woman_sichuanhua",
         "state":"done",
         "result_code":"0",
         "result_msg":"success"
      },
    "version": "1.0.0"
   }
}
// method：消息类型
// state：状态下载结束
// result_code: 下载结果，0成功，非0失败
// result_msg: 失败情况的具体描述信息
```

设备文件上传

1. 设备请求文件上传的 URL，设备端通过 MQTT 协议发布一条消息到 `$thing/up/service/${productid}/${devicename}` ，请求文件上传的URL，消息为 json 格式，内容如下：

```
{
   "method": "request_url",
   "request_id": "12345678",
   "report":{"resource_name": "123.wav", "version": "1.0.0", "resource_type":
"AUDIO"}
}
//method：消息类型
//resource_name: 文件名称
//version：文件版本号
//resource_type： 文件类型
```

2. 服务端收到文件版本信息上报后，服务端通过 Topic `$thing/down/service/${productid}/${devicename}` 向设备端返回已完成预签名的 cos url，消息为 json 格式，内容如下：

```
{
   "method": "request_url_resp",
   "result_code":0,
   "result_msg":"success",
   "resource_url": "presigned_url_xxx",
   "resource_token": "123456abcdef",
   "request_id": "12345678"
}
//method：消息类型
//result_code: 版本上报结果
//result_msg：版本上报结果信息
//resource_url: cos 预签名 url
//resource_token：文件token，后续可以根据token映射资源url
```

3. 设备将资源 put 到对应的 cos url，上传结束后，上报上传结果，消息为 json 格式，内容如下：

```
{
   "method": "report_post_result",
   "report":{
      "progress":{
         "resource_token": "123456abcdef",
         "state":"done",
         "result_code":"0",
         "result_msg":"success"
```

```
        },
    }
}
//method：消息类型
//resource_name: 文件名称
//state：上传结果
//result_code: 上传结果错误码，0成功
//result_msg: 上传结果信息
//version：文件版本
```

文件删除

1. 设备端会通过订阅的 Topic $thing/down/service/${productID/${deviceName} 接收文件的删除消息，文件删除消息内容如下：

```
{
    "method": "del_resource",
    "resource_name": "audio_woman_sichuanhua",
    "resource_type": "FILE",
    "version": "1.0.0"
}
// method：消息类型为
// resource_name：文件名称
```

2. 当设备完成文件下载，设备需要通过 Topic `$thing/up/service/${productID/${deviceName}` 上报一条删除的结果，特别的，若待删除的文件在设备端不存在(设备端刷机导致丢失等)，建议回复删除成功，否则此文件在设备端和云端的记录一直不一致，内容如下：

```
{
    "method": "del_result",
    "report":{
        "progress":{
            "resource_name": "audio_woman_sichuanhua",
            "state":"done",
            "result_code":"0",
            "result_msg":"success"
        },
    "version": "1.0.0"
    }
}
// method：消息类型
// state：删除结束
```

```
// result_code: 删除结果，0成功，非0失败
// result_msg: 失败情况的具体描述信息
```