物联网开发平台

设备端开发指南

腾讯云

【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

设备端开发指南
开发者指引

最近更新时间：2023-07-13 17:41:23

腾讯云物联网开发平台提供多种语言、面向多种场景的设备端 SDK。为了让您更高效地使用正确的 SDK 接入和利用物联网开发平台，请您仔细阅读以下内容。

开发者指引

SDK 概览

SDK	说明	应用场景	参考文档
qcloud-iot-explorer-sdk-embedded-c	设备端 C 语言 SDK	面向基于 C 语言开发的平台，提供多种接入并使用物联网开发平台的适配指引。	C SDK 使用参考
qcloud-iot-explorer-5G-sdk-embedded	设备端 C 语言 SDK 5G	面向基于 C 语言的开发平台，在物联网开发平台的基础上引入 5G 和边缘计算特性。	C SDK 5G 使用参考
iot-device-java	设备端 Java 语言 SDK	面向基于 Java 语言开发的平台，提供 Android 等平台接入并使用物联网开发平台的适配指引	• Android SDK 使用参考 • Java SDK 使用参考
iot-device-python	设备端 Python 语言 SDK	面向基于Python语言开发的平台，支持运行MicroPython的微控制器或嵌入式设备接入物联网开发平台	Python SDK 使用参考
qcloud-iot-sdk-tencent-at-based	设备端 AT 模组 SDK	面向基于腾讯云定制 AT 模组开发的平台，提供 MCU+ 腾讯云定制 AT 模组接入物联网开发平台的适配指引	AT SDK 使用参考
qcloud-iot-esp-wifi	设备端 ESP8266 SDK	面向基于 ESP8266 开发的平台，提供腾讯云 ESP8266 定制固件接入流程，以及 SoftAp、SmartConfig 等多种配网协议接入腾讯连连小程序的适配指引	ESP8266 SDK 使用参考

除了使用以上 SDK 接入物联网开发平台，您还可以使用 TencentOS tiny 通过移植 C SDK 来快速接入物联网开发平台，并为您的应用引入 TencentOS tiny 低功耗、低资源占用、模块化、安全可靠等特性，详情请参见 基于 TencentOS tiny 的 SDK 使用参考 。

开发流程

开发流程分为以下三步：

1. 确定应用场景，其中应用场景包含：
   - 直连设备。
   - 网关及子设备。
   - 蓝牙设备。
   - 设备配网。
2. 根据应用场景选择相应的 SDK。
3. 参考文档以及示例实现功能。

相关指引

直连设备

直连设备接入类型分为**资源丰富类**和**资源受限类**，详情请参见 直连设备接入类型说明 。

- 资源丰富类设备

开发平台	SDK	参考文档
Linux	qcloud-iot-explorer-sdk-embedded-c	Linux 平台接入指引
Windows	qcloud-iot-explorer-sdk-embedded-c	Windows 平台接入指引
Android	iot-device-java	Android 平台接入指引
Java	iot-device-java	Java 平台接入指引
FreeRTOS+lwIP	qcloud-iot-explorer-sdk-embedded-c	FreeRTOS+lwIP 平台接入指引
其他平台	qcloud-iot-explorer-sdk-embedded-c	C SDK 移植接入指引

- 资源受限类设备

开发平台	SDK	参考文档

MCU+定制 AT 模组（蜂窝类）	qcloud-iot-sdk-tencent-at-based	MCU+ 定制 MQTT AT 模组（蜂窝类）接入指引
MCU+定制 AT 模组（Wi-Fi 类）	qcloud-iot-sdk-tencent-at-based	MCU+ 定制 MQTT AT 模组（Wi-Fi 类）接入指引
MCU+通用 AT 模组+FreeRTOS	qcloud-iot-explorer-sdk-embedded-c	MCU+ 通用 TCP AT 模组（FreeRTOS）接入指引
MCU+ 通用 AT 模组+nonOS	qcloud-iot-explorer-sdk-embedded-c	MCU+ 通用 TCP AT 模组（nonOS）接入指引

蓝牙设备

详情请参见 蓝牙设备开发 。

网关及子设备

详情请参见 网关及子设备开发 。

设备配网

配网协议	SDK	参考文档
AirKiss	qcloud-iot-esp-wifi	AirKiss 配网开发
SmartConfig	qcloud-iot-esp-wifi	SmartConfig 配网开发
softAP	qcloud-iot-esp-wifi	softAP 配网开发

最佳实践

实践项	简介
Wi-Fi 配网实践	该实践主要介绍如何将腾讯云物联 IoT C SDK 移植到乐鑫 ESP8266 RTOS 平台，并提供可运行的 Demo。同时介绍在代码级别如何使用 Wi-Fi 配网 API，并可结合腾讯连连小程序进行 SoftAP 方式进行 Wi-Fi 配网及设备绑定。
MCU+定制 AT 模组实践	该实践面向使用支持腾讯 AT 指令的模组（2/3/4/5G、NB、Wi-Fi 等）接入腾讯物联网平台的终端设备开发者，实现 mcu 侧使用腾讯 AT_SDK 的移植示例，展示了基于 STM32F103 MCU 和 FreeRTOS 的软硬件环境如何实现 HAL 层的移植。
MCU+通用 TCP 模组（I-Cube）实践	该实践基于 STM32 I-Cube 实现 STM32+esp8266+FreeRTOS STM32+esp8266+无 RTOS 的移植示例。

TencentOS-tiny	该实践基于腾讯面向物联网领域开发的实时操作系统 TencentOS-tiny 实现接入腾讯云物联网开发平台。

设备身份认证介绍

最近更新时间：2024-09-30 17:54:51

概述

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开发平台。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。
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。

> **注意：**
> 仅企业实例支持证书认证。

设备认证信息烧入方式

本地烧录

用户通过物联网开发平台控制台或云 API 预创建好设备，获取到设备认证所需的产品 ID 、设备名称 和 设备密钥（或设备证书、设备私钥）。在设备生产时，把设备认证所需的配置信息，烧录到设备Flash上长期保存，用于后续设备与物联网开发平台发起连接，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。

操作流程

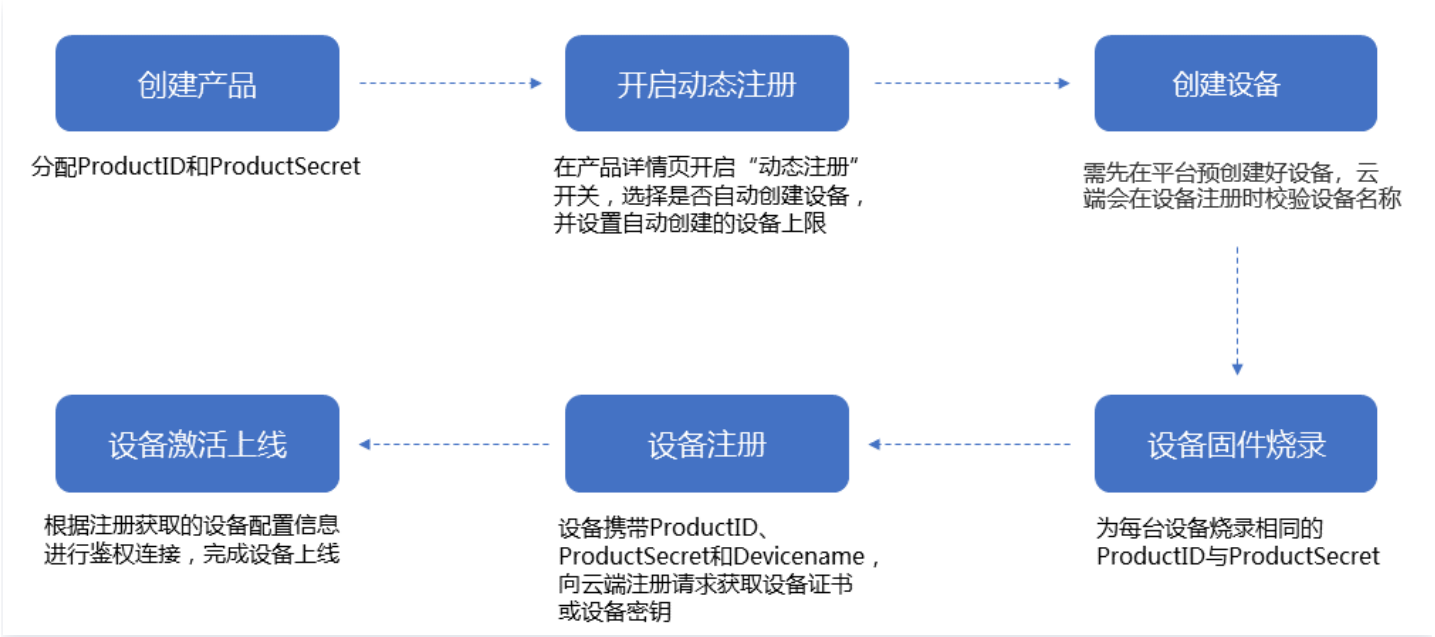
需要为每个设备烧录不同的固件，在产线应用中有一定实现成本，但安全性更高，推荐使用。

空中下发

设备可通过动态获取的方式，获取设备要烧录的设备认证信息。在该方式下，用户只需要在控制台上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）。用户在物联网开发平台控制台或云 API 预创建好设备，再通过动态注册请求获取设备认证所需的配置信息，烧录到设备Flash上长期保存，用于后续设备与物联网开发平台发起连接，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。

> **注意：**
> **若需要使用此烧录方式，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。**

操作流程



设备通信协议介绍
设备MQTT接入协议
设备基于 TCP 的 MQTT 接入

最近更新时间：2024-12-13 17:17:42

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 应该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时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：

```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 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","Y
ourPsk"));
            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：2024-12-11 11:58:42

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. 登录 物联网开发平台控制台，进入目标产品页面。在对应的设备信息部分，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. 登录 物联网开发平台控制台，进入目标产品页面。在对应的设备信息部分，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。

动态注册协议

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":
"031T01DWAoqFePDt71VuZXuLzkUzbIhGOnvMzpAFtNgOjagyFNHVSostNl9ztvhOuRx0dMM
/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/YYbcmSE12XpeDVmQNDcf1QgVD141RRbmmAnFwQfp1ECAu5O016mCOvYlJJ6V
59yM4OqQSiWphfTg==
```

2. Base64 解码后：

```
b3a141ddad4103f6186dc992135d97a5e0d599034371fd508150f5e354516e69809c5c
107e9d44080bb93b4d7a9823af625249e95e7dc8ce0ea904a25a985f4e
```

3. AES 解密。

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

物模型协议

最近更新时间：2024-12-13 17:17:42

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：2024-11-01 16:30:32

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

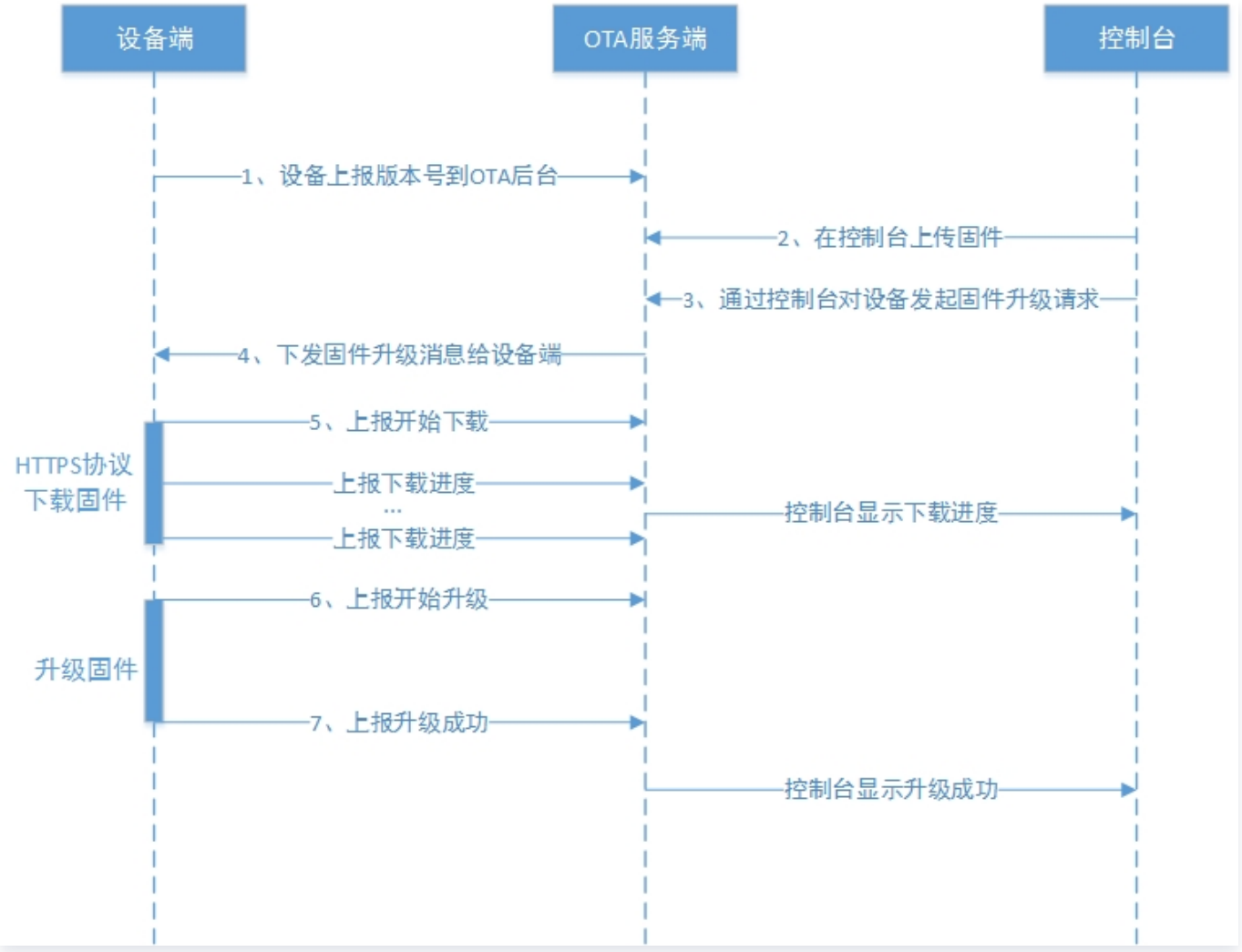
固件升级的过程中，需要设备订阅下面两个 Topic 来实现与云端的通信，如下图所示：

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": "36eb5951179db14a63**********22a2",
    "type": "update_firmware",
    "url": "https://ota-125*****90.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
// md5sum：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
```

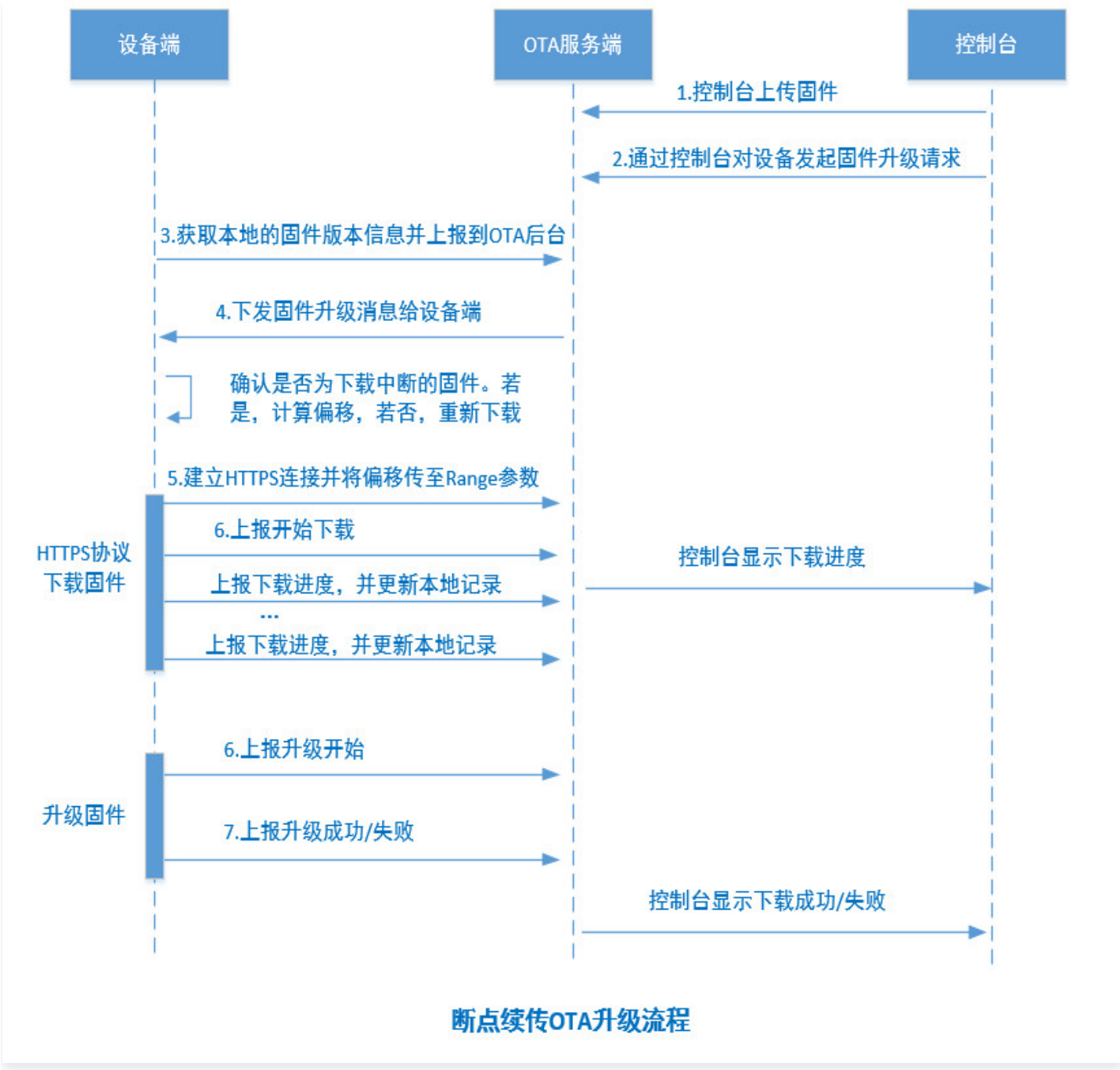
OTA 断点续传

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.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。
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：2024-10-15 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：2024-10-08 18:37:11

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

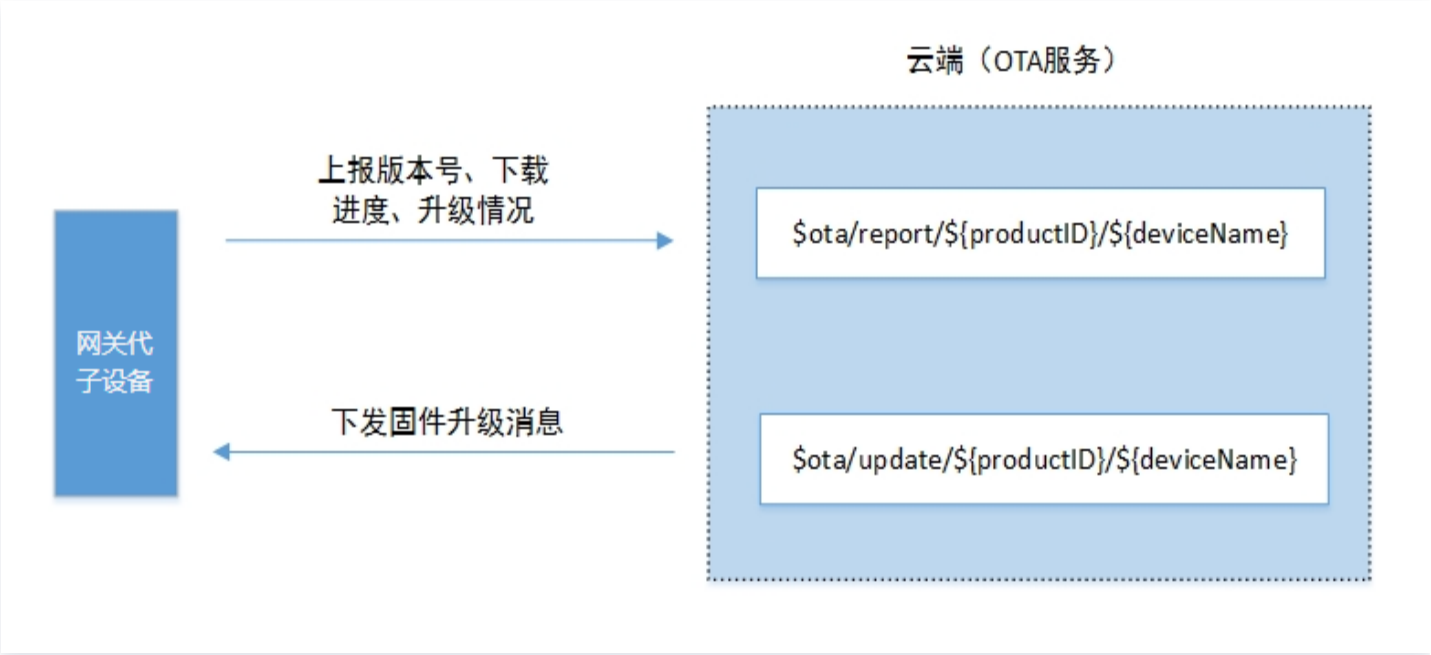
最近更新时间：2022-06-10 14:59:19

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：



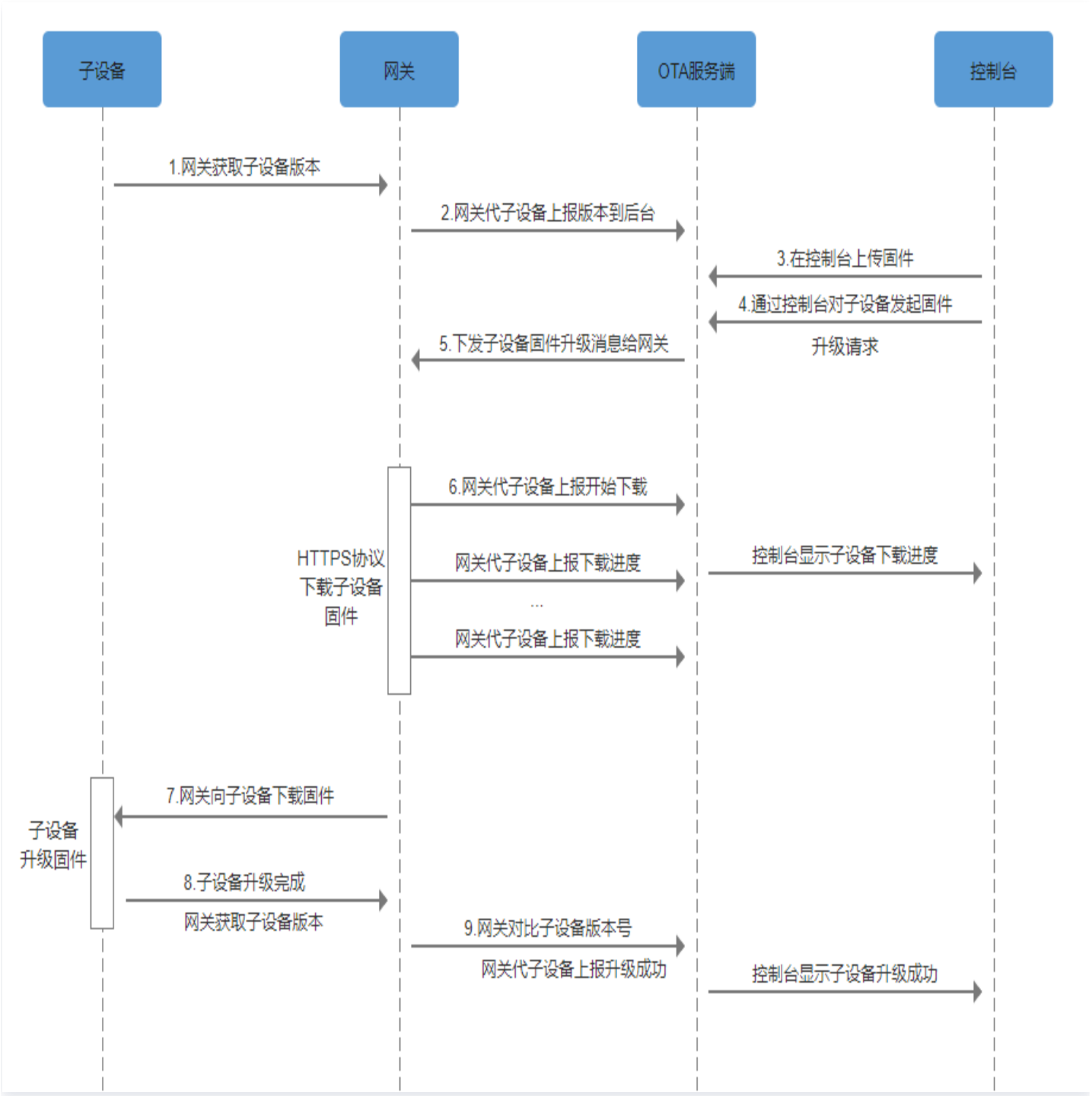
示例代码如下：

```
$ota/report/${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：2024-11-01 16:30:32

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-125*****59.cos.ap-guangzhou.myqcloud.com",
    "md5sum": "cc03e747a6afbbcbf8**********bee5",
    "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
```

```
// md5sum：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
```

微通话TWecall协议

最近更新时间：2024-07-04 11:47:31

功能概述

微通话 Twecall 是用于设备呼叫微信，进行音视频通话，主要应用于摄像头、门锁、门铃、室内屏等产品。
实现此功能需利用如下两个 Topic：

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

获取 snTicket

1. 设备端通过 MQTT 协议发布一条消息到 `$twecall/up/service/${productid}/${devicename}` ，进行获取，消息为 json 格式，内容如下：

```
{
    "method":"get_wechat_sn_ticket",
    "clientToken":"123",
    "timestamp":1628646783,
    "params":{
        "ModelId":"111",
        "miniProgramAppId":"111"
       }
}
//ModelId: 微信公众平台申请
//miniProgramAppId: 小程序的微信appid
```

2. 服务端收到获取请求上报后，服务端通过 `$twecall/down/service/${productid}/${devicename}` ，向设备端回复 snTicket 信息，消息为 json 格式，内容如下：

```
{
    "method":"get_wechat_sn_ticket_reply",
    "clientToken":"123",
    "code":0, // 0:正常，1:异常
    "status":"", //错误信息，正常为空
    "params":{
        "snTicket":"111",
        }
}
```

设备激活 TWecall 功能

1. 设备端会通过上报 `$twecall/up/service/${productid}/${devicename}` 消息，用于激活 Twecall 的 license，请求如下：

```
{
    "method":"active_device_voip_license",
    "clientToken":"123",
    "timestamp":1628646783,
    "params":{
        "pkgType":1, // 0-测试；1-家庭安防场景； 2-穿戴类场景； 3-生活娱乐场景； 4-对讲及其它场景
        "miniProgramAppId":"111",
        "modelId":"modelId1"
       }
}
//ModelId：微信公众平台申请
//miniProgramAppId: 小程序的微信appid
```

2. 服务端收到获取请求上报后，服务端通过 `$twecall/down/service/${productid}/${devicename}`，向设备端回复激活状态，消息为 json 格式，内容如下：

```
{
    "method":"active_device_voip_license_reply",
    "clientToken":"123",
    "code":0, // 0:正常，1:异常
    "status":"", //错误信息，正常为空
    "params":{
        }
}
```

查询设备 TWecall 激活详情

1. 设备端通过 MQTT 协议发布一条消息到 `$twecall/up/service/${productid}/${devicename}`，查询设备激活状态，消息为 json 格式，内容如下：

```
{
    "method":"get_voip_device_active_info",
    "clientToken":"123",
    "timestamp":1628646783,
    "params":{
        "miniProgramAppId":"111",
```

```
        "modelId":"modelId1"
      }
}
//ModelId: 微信公众平台申请
//miniProgramAppId: 小程序的微信appid
```

2. 服务端收到查询信息上报后，服务端通过 Topic `$twecall/down/service/${productid}/${devicename}` 向设备端返回设备激活状态详情，消息为 json 格式，内容如下：

```
{
    "method":"get_voip_device_active_info_reply",
    "clientToken":"123",
    "code":0, // 0:正常，1:异常
    "status":"", //错误信息，正常为空
    "params":{
        "modelId":"modelId1",
        "sn":"sn1",
        "ExpireTime":1630425600
        }
}
```

SDK 说明及下载

最近更新时间：2025-04-16 17:38:42

腾讯云物联网开发平台针对不同的设备开发场景，提供了多版本语言的设备 SDK 供客户使用，请参考 开发者指引 。

C SDK 代码托管

- 自 V3.1.0 版本开始，使用独立的 Github 托管 C 设备 SDK 代码。
- 请下载最新版 C SDK 。
- SDK 3.1.0 之前的 C SDK 版本 访问此处 。

> ⚠ **注意：**
> V3.1.0 之前的版本相较于 V3.1.0 及以后的版本，代码 GitHub 路径不一样，同时与平台交互协议有较大区别。

AT SDK 代码托管

在 IoT explorer 平台 创建产品和设备 后，选择基于 MQTT AT 定制模组开发的方式，将会自动生成 MCU 侧的 AT SDK 代码，并且在平台创建的数据模板和事件会生成对应的配置及初始化代码。

Android SDK 代码托管

自 V3.1.0 版本开始，Android SDK 使用独立的 Github 托管代码。

Java SDK 代码托管

Java SDK 使用独立的 Github 托管代码。

Python SDK 代码托管

自 V1.0.0 版本开始，Python 设备端 SDK 代码使用 Github 托管。

设备端 SDK 使用参考
C SDK 使用参考
使用概述

最近更新时间：2023-07-12 17:44:11

腾讯云物联网设备端 C SDK 依靠安全且性能强大的数据通道，为物联网领域开发人员提供设备端快速接入云端，并和云端进行双向通信的能力。

C SDK 适用范围

C SDK 采用模块化设计，分离核心协议服务与硬件抽象层，并提供灵活的配置选项和多种编译方式，适用于不同设备的开发平台和使用环境。

具备网络通讯能力并使用 Linux/Windows 操作系统的设备

- 对于具备网络通讯能力并使用标准 Linux/Windows 系统的设备。例如 PC/服务器/网关设备，及较高级的嵌入式设备，例如树莓派等，可直接在该设备上编译运行 SDK。
- 对于需要交叉编译的嵌入式 Linux 设备，如果开发环境的 toolchain 具备 glibc 或类似的库，可以提供包括 socket 通讯，select 同步 IO，动态内存分配，获取时间/休眠/随机数/打印函数，以及临界数据保护。如 Mutex 机制（仅在需要多线程时）等系统调用，则只需进行简单适配（例如，在 CMakeLists.txt 或 make.settings 里修改交叉编译器的设定）即可编译运行 SDK。

具备网络通讯能力并采用 RTOS 系统的设备

- 对于具备网络通讯能力并采用 RTOS 的物联网设备，C SDK 需要针对不同的 RTOS 做移植适配工作，目前 C SDK 已经适配了包括 FreeRTOS/RT-Thread/TencentOS tiny 等多个面向物联网的 RTOS 平台。
- 在 RTOS 设备移植 SDK 时，如果平台提供了类似 newlib 的 C 运行库和类似 lwIP 的嵌入式 TCP/IP 协议栈，则移植适配工作也可轻松完成。

MCU+ 通讯模组的设备

- 对于不具备网络通讯能力的 MCU，一般采用 MCU+ 通讯模组的方式。通讯模组（包括 Wi-Fi/2G/4G/NB-IoT）一般提供了基于串口的 AT 指令协议供 MCU 进行网络通讯。针对这种场景，C SDK 封装了 AT-socket 网络层，网络层之上的核心协议和服务层无需移植。并提供了基于 FreeRTOS 和不带操作系统（nonOS）两种方式的 HAL 实现。
- 除此之外，腾讯云物联网还提供了专用的 AT 指令集，如果通讯模组实现了该指令集，则设备接入和通讯更为简单，所需代码量更少，针对这种场景，请参考面向腾讯云定制 AT 模组专用的 MCU AT SDK 。

SDK 目录结构简介

目录结构及顶层文件简介如下：

名称	说明
CMakeLists.txt	cmake 编译描述文件。
CMakeSettings.json	visual studio下的 cmake 配置文件。
cmake_build.sh	Linux 下使用 cmake 的编译脚本。
make.settings	Linux 下使用 Makefile 直接编译的配置文件。
Makefile	Linux 下使用 Makefile 直接编译。
device_info.json	设备信息文件，当 DEBUG_DEV_INFO_USED=OFF 时，将从该文件解析出设备信息。
docs	文档目录，SDK 在不同平台下使用说明文档。
external_libs	第三方软件包组件，例如 mbedtls。
samples	应用示例。
include	提供给用户使用的外部头文件。
platform	平台相关的源码文件，目前提供了针对不同 OS（Linux/Windows/FreeRTOS/nonOS），TLS（mbedtls）以及 AT 模组下的实现。
sdk_src	SDK 核心通信协议及服务代码。
tools	SDK 配套的编译及代码生成脚本工具。

SDK 编译方式说明

C SDK 支持三种编译方式：

- cmake 方式。
- Makefile 方式。
- 代码抽取方式。

编译方式以及编译配置选项的详细说明请参考 编译配置说明 和 编译环境说明 。

SDK 示例体验

C SDK 的 samples 目录有使用各个功能的示例，关于运行示例的详细说明，请参考 SDK 文档目录下所有文档。
物联网开发平台快速体验物模型的数据交互，请参考 智能灯快速入门 。

编译配置说明

最近更新时间：2022-06-10 15:12:03

本文对 C SDK 的编译方式和编译配置选项进行说明，并介绍 Linux 和 Windows 开发环境下的编译环境搭建以及编译示例。

C SDK 编译方式说明

C SDK 支持以下编译方式。

cmake 方式

- 推荐使用 cmake 作为跨平台的编译工具，支持在 Linux 和 Windows 开发环境下进行编译。
- cmake 方式采用 CMakeLists.txt 作为编译配置选项输入文件。

Makefile 方式

- 对于不支持 cmake 的环境，使用 Makefile 直接编译的方式。
- Makefile 方式采用 make.settings 作为编译配置选项输入文件，修改完成后执行 make 即可。

代码抽取方式

- 该方式可根据需求选择功能，将相关代码抽取到一个单独的文件夹，文件夹里面的代码层次目录简洁，方便用户拷贝集成到自己的开发环境。
- 该方式需要依赖 cmake 工具，在 CMakeLists.txt 中配置相关功能模块的开关，并将 EXTRACT_SRC 设置为 ON，在 Linux 环境运行以下命令：

```
mkdir build
cd build
cmake ..
```

即可在 output/qcloud_iot_c_sdk 中找到相关代码文件，目录层次如下：

```
qcloud_iot_c_sdk
├── include
│   ├── config.h
│   ├── exports
├── platform
└── sdk_src
    └── internal_inc
```

  - include 目录为 SDK 供用户使用的 API 及可变参数，其中 config.h 为根据编译选项生成的编译宏。

- platform 目录为平台相关的代码，可根据设备的具体情况进行修改适配。
- sdk_src 为 SDK 的核心逻辑及协议相关代码，一般无需修改，其中 internal_inc 为 SDK 内部使用的头文件。

> **说明**
> 用户可将 qcloud_iot_c_sdk 拷贝到其目标平台的编译开发环境，并根据具体情况修改编译选项。

C-SDK 编译选项说明

编译配置选项

以下配置选项大部分都适用于 cmake 和 make.setting。cmake 中的 ON 值对应于 make.setting 的 y，OFF 对应于 n。

名称	cmake 值	说明
BUILD_TYPE	release/debug	• release：不启用 IOT_DEBUG 信息，编译输出到 release 目录下。 • debug：启用 IOT_DEBUG 信息，编译输出到 debug 目录下。
EXTRACT_SRC	ON/OFF	代码抽取功能开关，仅对使用 cmake 有效。
COMPILE_TOOLS	gcc	支持 gcc 和 msvc，也可以是交叉编译器。例如 arm-none-linux-gnueabi-gcc。
PLATFORM	Linux	包括 Linux/Windows/Freertos/Nonos。
FEATURE_OTA_COMM_ENABLED	ON/OFF	OTA 功能使能开关。
FEATURE_AUTH_MODE	KEY/CERT	接入认证方式。
FEATURE_AUTH_WITH_NOTLS	ON/OFF	• OFF：TLS 使能。 • ON：TLS 关闭。
FEATURE_EVENT_POST_ENABLED	ON/OFF	事件功能使能开关。
FEATURE_ACTION_ENABLED	ON/OFF	行为功能使能开关。

FEATURE_DEBUG_DEV_INFO_USED	ON/OFF	设备信息获取来源开关。
FEATURE_SYSTEM_COMM_ENABLED	ON/OFF	获取后台时间开关。
FEATURE_DEV_DYN_REG_ENABLED	ON/OFF	设备动态注册开关。
FEATURE_LOG_UPLOAD_ENABLED	ON/OFF	日志上报开关。

使用 SDK AT 框架+通用 TCP 模组配置项如下：

名称	cmake 值	说明
FEATURE_AT_TCP_ENABLED	ON/OFF	AT 模组 TCP 功能开关。
FEATURE_AT_UART_RECV_IRQ	ON/OFF	AT 模组中断接受功能开关。
FEATURE_AT_OS_USED	ON/OFF	AT 模组多线程功能开关。
FEATURE_AT_DEBUG	ON/OFF	AT 模组调试功能开关。

配置选项之间存在依赖关系，当依赖选项的值为有效值时，部分配置选项才有效，主要如下：

名称	依赖选项	有效值
FEATURE_AUTH_WITH_NOTLS	FEATURE_AUTH_MODE	KEY
FEATURE_AT_UART_RECV_IRQ	FEATURE_AT_TCP_ENABLED	ON
FEATURE_AT_OS_USED	FEATURE_AT_TCP_ENABLED	ON
FEATURE_AT_DEBUG	FEATURE_AT_TCP_ENABLED	ON

设备信息选项

在腾讯云物联网控制台创建设备之后，需要将设备信息（ProductID/DeviceName/DeviceSecret/Cert/Key 文件）配置在 SDK 中才能正确运行。在开发阶段，SDK 提供两种方式存储设备信息：

- 存放在代码中（编译选项 DEBUG_DEV_INFO_USED = ON），则在 `platform/os/xxx/HAL_Device_xxx.c` 中修改设备信息，在无文件系统的平台下可以使用这种方式。
- 存放在配置文件中（编译选项 DEBUG_DEV_INFO_USED = OFF），则在 `device_info.json` 文件修改设备信息，此方式下更改设备信息不需重新编译 SDK，在 Linux/Windows 平台下开发推荐使用这种方式。

编译环境说明

最近更新时间：2024-08-26 14:55:51

C SDK 提供多种平台下接入并使用物联网开发平台的适配指引，其中的示例 Demo 可用于在 Linux 和 Windows 上编译 SDK 进行快速体验。

编译环境

Linux 编译环境

Linux 下使用 cmake + gcc 进行 SDK 编译，其中 cmake 版本在 v3.5 以上，默认安装的 cmake 版本较低，若编译失败，请单击 下载 并参考 安装说明 进行 cmake 特定版本的下载与安装。

```
$ sudo apt-get install -y build-essential make git gcc cmake
```

Windows 编译环境

Windows 下使用 Visual Studio 2019 中的 cmake 工具进行 SDK 编译。
获取和安装 Visual Studio 2019开发环境方法如下：

1. 请访问 Visual Studio 下载网站，下载并安装 Visual Studio 2019，本文档下载安装的是 v16.2版本 Community。

2. 单击**使用 C++ 的桌面开发**，并确保勾选“用于 Windows 的 C++ CMAKE 工具”。

接入指引

- Linux 平台下编译及运行请参见 Linux 平台接入指引。

- Windows 平台下编译及运行请参见 Windows 平台接入指引 。

接口及可变参数说明

最近更新时间：2024-09-30 17:54:51

设备端 C SDK 供用户调用的 API 函数声明、常量以及可变参数定义等头文件位于 include 目录下面，本文档主要对该目录下面的可变参数以及 API 函数进行说明。

可变参数配置

C SDK 基于 MQTT 协议，可以根据具体场景需求，配置相应的参数，满足实际业务进行运行使用。可变接入参数包括：

1. MQTT 阻塞调用（包括连接，订阅，发布等）的超时时间，单位：毫秒。 建议5000毫秒。
2. MQTT 协议发送消息和接收消息的 buffer 大小默认为2048字节，最大支持16KB。
3. MQTT 心跳消息发送周期，最大值为690秒，单位：毫秒。
4. 重连最大等待时间，单位：毫秒。设备断线重连时，若失败则等待时间会翻倍，当超过该最大等待时间则退出重连。

修改 `include/qcloud_iot_export_variables.h` 文件，宏定义可以修改对应接入参数的配置，修改完需要重新编译 SDK ，示例代码如下：

```
/* default MQTT/CoAP timeout value when connect/pub/sub (unit: ms) */
#define QCLOUD_IOT_MQTT_COMMAND_TIMEOUT                                 (5 *
1000)

/* default MQTT keep alive interval (unit: ms) */
#define QCLOUD_IOT_MQTT_KEEP_ALIVE_INTERNAL                             (240
* 1000)

/* default MQTT Tx buffer size, MAX: 16*1024 */
#define QCLOUD_IOT_MQTT_TX_BUF_LEN
(2048)

/* default MQTT Rx buffer size, MAX: 16*1024 */
#define QCLOUD_IOT_MQTT_RX_BUF_LEN
(2048)

/* default COAP Tx buffer size, MAX: 1*1024 */
#define COAP_SENDMSG_MAX_BUFLEN
(512)

/* default COAP Rx buffer size, MAX: 1*1024 */
#define COAP_RECVMSG_MAX_BUFLEN
(512)
```

```
/* MAX MQTT reconnect interval (unit: ms) */
#define MAX_RECONNECT_WAIT_INTERVAL                                 (60
* 1000)
```

API 函数说明

以下是 C SDK v3.1.0 版本提供的主要功能和对应 API 接口说明，用于客户编写业务逻辑时，进行更加详细的说明，例如：接口参数及返回值可查看 SDK 代码 `include/exports/qcloud_iot_export_*.h` 等头文件中的注释。

物模型接口

物模型协议及功能介绍，请参见 物模型协议 。

序号	函数名	说明
1	IOT_Template_Construct	构造物模型客户端 Data_template_client 并连接 MQTT 云端服务。
2	IOT_Template_Destroy	关闭 Data_template MQTT 连接并销毁 Data_template Client。
3	IOT_Template_Yield	在当前线程上下文中，进行 MQTT 报文读取，消息处理，超时请求，心跳包及重连状态管理等任务。
4	IOT_Template_Publish	物模型客户端发布 MQTT 消息。
5	IOT_Template_Subscribe	物模型客户端订阅 MQTT 主题。
6	IOT_Template_Unsubscribe	物模型客户端取消订阅已订阅的 MQTT 主题。
7	IOT_Template_IsConnected	查看当前物模型客户端的 MQTT 是否已连接。

- **物模型属性接口**

序号	函数名	说明
1	IOT_Template_Register_Property	注册当前设备的物模型属性。
2	IOT_Template_UnRegister	删除已经注册过的物模型属性。

	_Property	
3	IOT_Template_Report	异步方式上报物模型属性数据。
4	IOT_Template_Report_Sync	同步方式上报物模型属性数据。
5	IOT_Template_GetStatus	异步方式获取物模型属性数据。
6	IOT_Template_GetStatus_sync	同步方式获取物模型属性数据。
7	IOT_Template_Report_SysInfo	异步方式上报系统信息。
8	IOT_Template_Report_SysInfo_Sync	同步方式上报系统信息。
9	IOT_Template_JSON_ConstructSysInfo	构造待上报的系统信息。
10	IOT_Template_ControlReply	回复收到的物模型属性控制消息。
11	IOT_Template_ClearControl	删除物模型属性控制消息，配合 IOT_Template_GetStatus 获取到 control 消息使用。

- **物模型事件接口**

序号	函数名	说明
1	IOT_Post_Event	上报物模型事件，传入事件，SDK 完成事件消息构造。
2	IOT_Post_Event_Raw	上报物模型事件，传入符合事件消息格式的数据，SDK 完成上报。
3	IOT_Event_setFlag	置位事件标志，SDK 默认支持10个事件，可以扩展增加。
4	IOT_Event_clearFlag	清除事件标志。
5	IOT_Event_getFlag	获取事件标志。

- **物模型行为接口**

序号	函数名	说明
1	IOT_ACTION_REPLY	回复物模型行为消息。

- **多线程环境使用说明**
  SDK 对于在多线程环境下的使用有如下注意事项：
  - 不允许多线程调用 IOT_Template_Yield，IOT_Template_Construct 以及 IOT_Template_Destroy。
  - IOT_Template_Yield 作为从 socket 读取并处理 MQTT 报文的函数，应保证一定的执行时间，避免被长时间挂起或抢占。

OTA 接口

关于 OTA 固件下载功能介绍，请参见 设备固件升级 。

序号	函数名	说明
1	IOT_OTA_Init	初始化 OTA 模块，客户端在调用此接口之前需要先进行 MQTT/COAP 的初始化。
2	IOT_OTA_Destroy	释放 OTA 模块相关的资源。
3	IOT_OTA_ReportVersion	向 OTA 服务器报告本地固件版本信息。
4	IOT_OTA_IsFetching	检查是否处于下载固件的状态。
5	IOT_OTA_IsFetchFinish	检查固件是否已经下载完成。
6	IOT_OTA_FetchYield	从具有特定超时值的远程服务器获取固件。
7	IOT_OTA_Ioctl	获取指定的 OTA 信息。
8	IOT_OTA_GetLastError	获取最后一个错误代码。
9	IOT_OTA_StartDownload	根据获取到的固件更新地址以及本地固件信息偏移（是否断点续传），与固件服务器建立 HTTP 连接。
10	IOT_OTA_UpdateClientMd5	断点续传前，计算本地固件的 MD5。
11	IOT_OTA_ReportUpgradeBegin	当进行固件升级前，向服务器上报即将升级的状态。
12	IOT_OTA_ReportUpgradeSuccess	当固件升级成功之后，向服务器上报升级成功的状态。

13	IOT_OTA_ReportUpgradeFail	当固件升级失败之后，向服务器上报升级失败的状态。

日志接口

设备日志上报云端功能的详细说明，请参考 SDK docs 目录下物联网通信平台文档设备日志上报功能部分。

序号	函数名	说明
1	IOT_Log_Set_Level	设置 SDK 日志的打印等级。
2	IOT_Log_Get_Level	返回 SDK 日志打印的等级。
3	IOT_Log_Set_MessageHandler	设置日志回调函数，重定向 SDK 日志于其它输出方式。
4	IOT_Log_Init_Uploader	开启 SDK 日志上报云端的功能并初始化资源。
5	IOT_Log_Fini_Uploader	停止 SDK 日志上报云端功能并释放资源。
6	IOT_Log_Upload	将 SDK 运行日志上报到云端。
7	IOT_Log_Set_Upload_Level	设置 SDK 日志的上报等级。
8	IOT_Log_Get_Upload_Level	返回 SDK 日志上报的等级。
9	Log_d/i/w/e	按级别打印添加 SDK 日志的接口。

系统时间接口

序号	函数名	说明
1	IOT_Get_SysTime	获取 IoT Hub 后台系统时间，目前仅支持 MQTT 通道对时功能。

物模型代码生成

最近更新时间：2023-05-31 18:01:11

本文为您介绍基于物联开发平台 IoT Explorer 创建的物模型如何生成物模型代码。
生成物模型代码的有以下3个步骤。

步骤1：创建物模型

创建物模型，详情请参见 物模型 文档。

步骤2：导出物模型描述文件

物模型描述文件是一个 JSON 格式的文件，描述了产品定义的属性、事件及其他信息。
单击**查看物模型JSON**，确认内容后，单击下载图标，即可导出 JSON 文件。

步骤3：生成物模型代码

1. 执行以下命令，将已下载的 JSON 文件拷贝到 tools 目录。

```
./codegen.py -c xx/config.json -d  ../targetdir/
```

2. 根据 JSON 文件在 target 目录生成所定义产品的物模型及事件的配置文件，将这个生成的配置文件拷贝到 data_template_sample.c 的同级目录。

```
./codegen.py -c light.json -d ../samples/data_template/
加载 light.json 文件成功
文件 ../samples/data_template/data_config.c 生成成功
文件 ../samples/data_template/events_config.c 生成成功
```

> **说明：**
> data_template_sample.c 阐述了通用的物模型处理框架，可以基于此框架添加业务逻辑。
> 智能灯示例 light_data_template_sample.c 即是基于此框架的场景示例。

物模型应用开发

最近更新时间：2024-08-26 14:55:51

物模型示例 `data_template_sample.c` 已实现数据、事件收发及响应的通用处理框架。可以基于此示例开发业务逻辑，上下行业务逻辑添加的入口函数分别为 `deal_up_stream_user_logic` 、`deal_down_stream_user_logic` 。详情可参考 智能灯 的场景示例 `light_data_template_sample.c` 添加业务处理逻辑。

下行业务逻辑实现

服务端下行的数据，SDK 已按物模型协议完成 JSON 数据的解析。ProductDataDefine 是第三步中根据在平台定义的产品物模型生成的模板结构体，由定义的各属性构成成员变量。入参 pData 所指向的属性数据，从服务端下行数据中，SDK 已经按物模型协议完成了属性数据的解析，用户在下行逻辑处理的函数里，可直接使用解析完成的数据添加业务逻辑即可。

- 用户根据已解析的物模型数据（pData）进行相应的业务逻辑处理。

```
/*用户需要实现的下行数据的业务逻辑,待用户实现业务逻辑*/
static void deal_down_stream_user_logic(void *client,
ProductDataDefine   * pData)
{
    Log_d("someting about your own product logic wait to be done");
}
```

- 示例代码如下：

```
/*智能灯属性数据模板*/
typedef struct _ProductDataDefine {
    TYPE_DEF_TEMPLATE_BOOL m_light_switch;
    TYPE_DEF_TEMPLATE_ENUM m_color;
    TYPE_DEF_TEMPLATE_INT  m_brightness;
    TYPE_DEF_TEMPLATE_STRING m_name[MAX_STR_NAME_LEN+1];
} ProductDataDefine;
/*示例灯光控制处理逻辑*/
static void deal_down_stream_user_logic(void *client,ProductDataDefine
*light)
{
    int i;
    const char * ansi_color = NULL;
    const char * ansi_color_name = NULL;
    char brightness_bar[]       = "||||||||||||||||||||";
    int brightness_bar_len = strlen(brightness_bar);
```

```
    /*灯光颜色*/
    switch(light->m_color) {
        case eCOLOR_RED:
            ansi_color = ANSI_COLOR_RED;
            ansi_color_name = " RED ";
            break;
        case eCOLOR_GREEN:
            ansi_color = ANSI_COLOR_GREEN;
            ansi_color_name = "GREEN";
            break;
        case eCOLOR_BLUE:
            ansi_color = ANSI_COLOR_BLUE;
            ansi_color_name = " BLUE";
            break;
        default:
            ansi_color = ANSI_COLOR_YELLOW;
            ansi_color_name = "UNKNOWN";
            break;
    }
    /* 灯光亮度显示条 */
    brightness_bar_len = (light->m_brightness >= 100)?
brightness_bar_len:(int)((light->m_brightness *
brightness_bar_len)/100);
    for (i = brightness_bar_len; i < strlen(brightness_bar); i++) {
        brightness_bar[i] = '-';
    }
    if(light->m_light_switch){
        /* 灯光开启时，按照控制参数展示 */
        HAL_Printf( "%s[  lighting  ]|[color:%s]|[brightness:%s]|
[%s]\n" ANSI_COLOR_RESET,\
                    ansi_color,ansi_color_name,brightness_bar,light-
>m_name);
    }else{
        /* 灯光关闭展示 */
        HAL_Printf( ANSI_COLOR_YELLOW"[  light is off ]|[color:%s]|
[brightness:%s]|[%s]\n" ANSI_COLOR_RESET,\
                    ansi_color_name,brightness_bar,light->m_name);
    }
#ifdef EVENT_POST_ENABLED
    if(eCHANGED == sg_DataTemplate[0].state){
        if(light->m_light_switch){
            memset(sg_message, 0, MAX_EVENT_STR_MESSAGE_LEN);
            strcpy(sg_message,"light on");
```

```
            sg_status = 1;
        }else{
            memset(sg_message, 0, MAX_EVENT_STR_MESSAGE_LEN);
            strcpy(sg_message,"light off");
            sg_status = 0;
        }
        IOT_Event_setFlag(client, FLAG_EVENT0);/*灯的开关状态发生变化时置
位事件，在eventPostCheck中会将置位的事件上报*/
    }
#endif
}
```

上行业务逻辑实现

- 设备端根据业务场景需要，对设备端数据属性采取一定策略进行监测处理。
- 用户可以在 `deal_up_stream_user_logic` 中，将需要上报的属性更新给入参 pReportDataList 属性，上报列表及需要上报的属性个数，物模型的示例处理框架，在 `IOT_Template_JSON_ConstructReportArray` 中属性数据列表将会处理为物模型的协议格式，`IOT_Template_Report` 将数据发送给服务端。

示例代码如下：

```
/*用户根据业务修改属性值，然后设置属性状态为eCHANGED*/
static void _refresh_local_property(void)
{
    //add your local property refresh logic
}
/*用户需要实现的上行数据的业务逻辑,此处仅供示例*/
static int deal_up_stream_user_logic(DeviceProperty *pReportDataList[],
int *pCount)
{
    int i, j;

        /*监测是否需要更新本地数据*/
        _refresh_local_property();

        /*将变化的属性更新到上报列表*/
     for (i = 0, j = 0; i < TOTAL_PROPERTY_COUNT; i++) {
        if(eCHANGED == sg_DataTemplate[i].state) {
            pReportDataList[j++] = &(sg_DataTemplate[i].data_property);
            sg_DataTemplate[i].state = eNOCHANGE;
        }
    }
```

```
    *pCount = j;

    return (*pCount > 0)?QCLOUD_RET_SUCCESS:QCLOUD_ERR_FAILURE;
}
```

设备信息存储

最近更新时间：2023-05-24 17:12:59

腾讯云物联网开发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。并且设备端需要存储这些设备身份信息，在申请三元组或者四元组信息进行设备认证时将会使用到。C SDK 提供读写设备信息的接口及参考实现，可根据实际情况进行适配。

设备身份信息

- 证书设备要通过平台的安全认证，必须具备四元组信息：产品 ID（ProductID）、设备名（DeviceName）、设备证书文件（DeviceCert）、设备私钥文件（DevicePrivateKey），其中证书文件和私钥文件由平台生成，且一一对应。
- 密钥设备要通过平台的安全认证，必须具备三元组信息：产品 ID（ProductId）、设备名（DeviceName）、设备密钥（DeviceSecret），其中设备密钥由平台生成。

设备身份信息烧录

设备信息烧录分为预置烧录和动态烧录，两者在应用的便捷性和安全性上有些许区别，如下所述。

预置烧录

创建产品后，在 物联网开发平台控制台 或者通过 云 API 逐个创建设备，并获取对应的设备信息，将上述的四元组或者三元组信息，在设备生产的特定环节，烧录到非易失介质中，设备 SDK 运行时读取存放的设备信息，进行设备认证。

动态烧录

预置烧录需要在量产过程执行个性化生产动作，影响生产效率，为增加应用的便捷性，平台支持动态烧录的方式。

- 实现方式：产品创建后使能产品的动态注册功能，则会生成产品密钥（ProductSecret）。同一产品下的所有设备在生产过程可以烧录统一的产品信息，即产品 ID（ProductId）、产品密钥（ProductSecret）。设备出厂后，通过动态注册的方式获取设备身份信息并保存，在申请三元组或者四元组信息进行设备认证时将会使用到。
- 动态烧录设备名（DeviceName）的生成：
  - 若使能动态注册的同时使能自动设备创建，则设备名是可以由设备自己生成的，但必须保证同一产品 ID（ProductId）下的设备名不重复，一般取名为 IMEI 或者 MAC 地址。
  - 若使能动态注册的同时未使能自动设备创建，则需要把设备名预先录入平台，设备动态注册时将会校验所申请的设备名是否为合法录入的设备名，此种方式能在一定程度降低产品密钥泄露后的安全风险。

> ⚠ **注意：**
> 动态注册需要确保产品密钥（ProductSecret）的安全，否则将会产生安全隐患。

设备信息读写接口

SDK 提供了设备信息读写的 HAL 接口，必须实现。可以参考 Linux 平台 HAL_Device_Linux.c 中设备信息读写的实现。
设备信息 HAL 接口：

HAL_API	说明
HAL_SetDevInfo	设备信息写入。
HAL_GetDevInfo	设备信息读取。

开发阶段设备信息配置

创建设备之后，需要将设备信息（`ProductID/DeviceName/DeviceSecret/Cert/Key` 文件）配置在 SDK 中才能正确运行示例。在开发阶段，SDK 提供两种方式存储设备信息：

1. 存放在代码中（编译选项 DEBUG_DEV_INFO_USED = ON），则在 `platform/os/xxx/HAL_Device_xxx.c` 中修改设备信息，在无文件系统的平台下可以使用这种方式。

```
/* product Id  */
static char sg_product_id[MAX_SIZE_OF_PRODUCT_ID + 1]     =
"PRODUCT_ID";
/* device name */
static char sg_device_name[MAX_SIZE_OF_DEVICE_NAME + 1]   =
"YOUR_DEV_NAME";
#ifdef DEV_DYN_REG_ENABLED
/* product secret for device dynamic Registration  */
static char sg_product_secret[MAX_SIZE_OF_PRODUCT_SECRET + 1]   =
"YOUR_PRODUCT_SECRET";
#endif
#ifdef AUTH_MODE_CERT
/* public cert file name of certificate device */
static char sg_device_cert_file_name[MAX_SIZE_OF_DEVICE_CERT_FILE_NAME
+ 1]       = "YOUR_DEVICE_NAME_cert.crt";
/* private key file name of certificate device */
static char
sg_device_privatekey_file_name[MAX_SIZE_OF_DEVICE_SECRET_FILE_NAME +
1] = "YOUR_DEVICE_NAME_private.key";
#else
/* device secret of PSK device */
static char sg_device_secret[MAX_SIZE_OF_DEVICE_SECRET + 1] =
"YOUR_IOT_PSK";
#endif
```

2. 存放在配置文件中（编译选项 DEBUG_DEV_INFO_USED = OFF），则在 `device_info.json` 文件修改设备信息，此方式下更改设备信息不需重新编译 SDK，在 Linux/Windows 平台下开发推荐使用这种方式。

```
{
    "auth_mode":"KEY/CERT",

    "productId":"PRODUCT_ID",
    "productSecret":"YOUR_PRODUCT_SECRET",
    "deviceName":"YOUR_DEV_NAME",

    "key_deviceinfo":{
        "deviceSecret":"YOUR_IOT_PSK"
    },

    "cert_deviceinfo":{
        "devCertFile":"YOUR_DEVICE_CERT_FILE_NAME",
        "devPrivateKeyFile":"YOUR_DEVICE_PRIVATE_KEY_FILE_NAME"
    },

    "subDev":{
        "sub_productId":"YOUR_SUBDEV_PRODUCT_****",
        "sub_devName":"YOUR_SUBDEV_DEVICE_****"
    }
}
```

应用示例

- 初始化连接参数

```
static DeviceInfo sg_devInfo;
static int _setup_connect_init_params(MQTTInitParams* initParams)
{
    int ret;

    ret = HAL_GetDevInfo((void *)&sg_devInfo);
    if(QCLOUD_ERR_SUCCESS != ret){
        return ret;
    }

    initParams->device_name = sg_devInfo.device_****;
    initParams->product_id = sg_devInfo.product_****;
      ......
```

```
}
```

- 密钥设备鉴权参数生成

```
static int _serialize_connect_packet(unsigned char *buf, size_t
buf_len, MQTTConnectParams *options, uint32_t *serialized_len) {
            ......
            ......
    int username_len = strlen(options->client_id) +
strlen(QCLOUD_IOT_DEVICE_SDK_APPID) + MAX_CONN_ID_LEN +
cur_timesec_len + 4;
    options->username = (char*)HAL_Malloc(username_len);
    get_next_conn_id(options->conn_id);
    HAL_Snprintf(options->username, username_len, "%s;%s;%s;%ld",
options->client_id, QCLOUD_IOT_DEVICE_SDK_APPID, options->conn_id,
cur_timesec);
#if defined(AUTH_WITH_NOTLS) && defined(AUTH_MODE_KEY)
    if (options->device_secret != NULL && options->username != NULL) {
        char                sign[41]    = {0};
        utils_hmac_sha1(options->username, strlen(options->username),
sign, options->device_secret, options->device_secret_len);
        options->password = (char*) HAL_Malloc (51);
        if (options->password == NULL)
IOT_FUNC_EXIT_RC(QCLOUD_ERR_INVAL);
        HAL_Snprintf(options->password, 51, "%s;hmacsha1", sign);
    }
#endif
            ......
}
```

使用参考

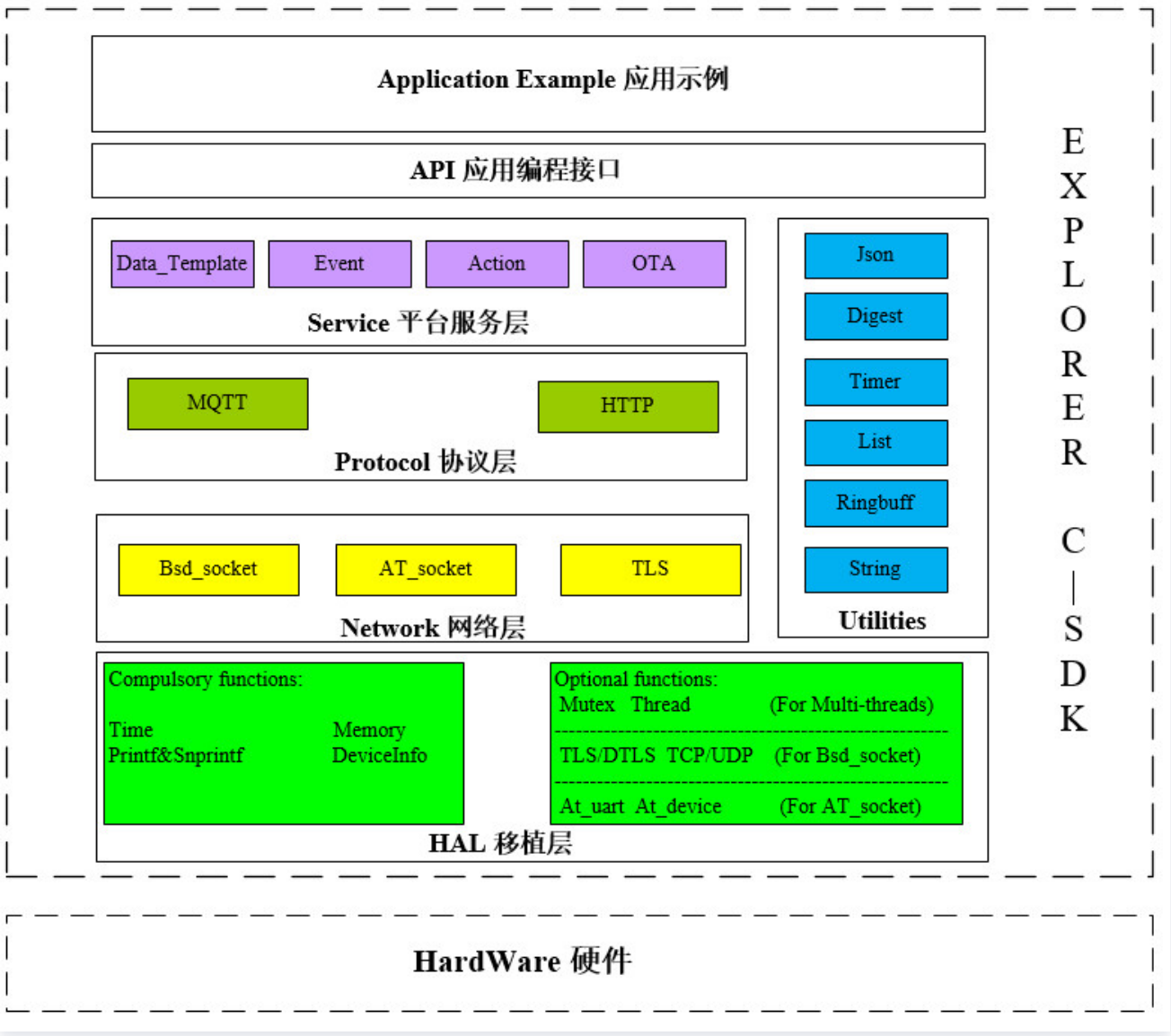
最近更新时间：2023-12-14 17:25:51

腾讯云物联网设备端 C SDK 依靠安全且性能强大的数据通道，为物联网领域开发人员提供设备端快速接入云端，并和云端进行双向通信的能力。

SDK 获取

SDK 使用 Github 托管，可访问 Github 下载最新版本设备端 C SDK 。

软件架构



SDK 分四层设计，从上至下分别为平台服务层、核心协议层、网络层、硬件抽象层。

- **服务层**
  - 在网络协议层之上，实现了包括设备接入鉴权，设备影子，网关，动态注册，日志上报和 OTA 等功能。

- 设备侧和 IoT Explorer 平台交互的核心协议为 MQTT，基于此核心协议，实现了数据模板和 OTA 功能。平台通过对 IoT 设备的通用抽象定义了 数据模板协议，云端和设备基于 MQTT 的 payload 承载的数据实现数据模板协议数据流交互。OTA 功能的升级命令、版本及固件信息通过 MQTT 协议通道交互，固件下载通过 HTTPS 协议通道交互。

- **协议层**
  设备端和 IoT 平台交互的网络协议包括 MQTT/CoAP/HTTP。
- **网络层**
  网络层的实现支持基于 bsd_socket 方式和 AT_socket 方式，对于资源丰富系统本身集成 TCP/IP 或 LwIP 网络协议栈的，可以选择 bsd_socket 的网络接口。对于部分资源受限设备，通过通信模组（蜂窝模组/Wi-Fi 模组等）和 MCU 交互而实现网络接入的可以选择 SDK 提供 at_socket 框架，SDK 未支持的通信模组参照 SDK 支持的通信模组实现 at_device 结构体 at_device_op_t 里的驱动接口即可。
- **硬件抽象层**
  硬件抽象层需要针对具体的软硬件平台开展移植，分为必须实现和可选实现两部分 HAL 层接口。必选实现的接口为时间（获取毫秒数）、打印、格式化打印、内存操作、设备信息读写。可选实现接口，使用 RTOS 的，需要实现锁、信号量、线程创建及销毁、延时睡眠。使用 AT_Socket 接入网络，需要实现 AT 串口驱动及模组驱动。SDK 已经支持 Linux、Windows、FreeRTOS、nonOS 四种典型环境的 HAL 层示例移植实现，在 Linux 和 Windows 环境可以直接编译运行相关示例。

移植指引

开发平台	文档
Linux	Linux 平台接入指引
Windows	Windows 平台接入指引
MCU+ 通用 AT 模组+FreeRTOS	MCU+ 通用 TCP AT 模组（FreeRTOS）接入指引
MCU+ 通用 AT 模组+nonOS	MCU+ 通用 TCP AT 模组（nonOS）接入指引
FreeRTOS+lwIP	FreeRTOS+lwIP 平台接入指引
其他平台	C SDK 移植接入指引

SDK 相关文档

SDK 相关文档，详情请参见 SDK 使用参考 。

AT SDK 使用参考

最近更新时间：2022-06-10 15:14:13

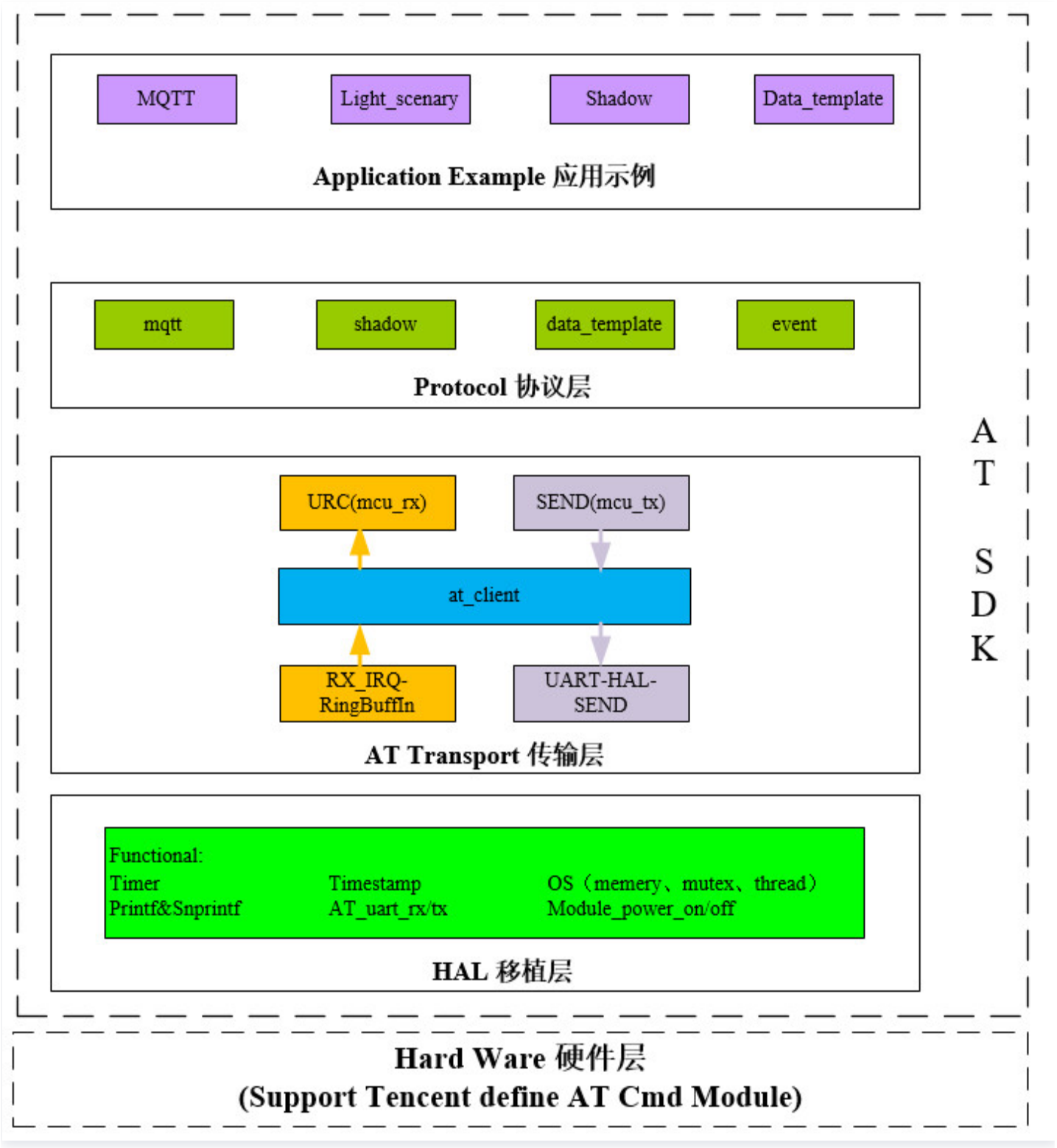
AT SDK 面向内置腾讯云物联网 AT 指令的模组，提供实现和定制模组交互的 SDK。

SDK 获取

在 IoT explorer 平台 创建产品和设备 后，选择基于 MQTT AT 定制模组开发的方式，将会自动生成 MCU 侧的 AT SDK 代码，并且把在平台创建的数据模板和事件生成了对应的配置及初始化代码。

软件架构

AT SDK 软件架构图如下：



SDK 分四层设计，从上至下分别为应用层、核心协议层、AT 传输层、硬件抽象层。

- **服务层**

  在网络协议层之上，实现了包括设备接入鉴权，设备影子，网关，动态注册，日志上报和 OTA 等功能。

- **协议层**
  设备端和 IoT 平台交互的网络协议包括 MQTT/COAP/HTTP。
- **AT 传输层**
  实现基于腾讯云物联网定制 AT 指令的网络协议栈。
- **硬件抽象层**
  实现对不同硬件平台的底层操作抽象封装，需要针对具体的软硬件平台开展移植，分为必须实现和可选实现两部分 HAL 层接口。

目录结构

名称	说明
docs	文档目录，包含腾讯 AT 指令集定义。
port	HAL 层移植目录，需要实现串口的收发接口（中断接收），延时函数，模组上下电及 OS 相关接口。
sample	应用示例，示例使用 MQTT、影子、数据模板的使用方式。
src	AT 框架及协议逻辑实现。
— event	事件功能协议封装。
— module_at	at client 抽象，实现 RX 解析，命令下行，urc 匹配，resp 异步匹配。
— shadow	基于 AT 框架的 shadow 逻辑实现。
— mqtt	基于AT 框架的 MQTT 协议实现。
— utils	json、timer、链表等应用。
— include	SDK 对外头文件及设备信息配置头文件。
usr_logic	自动生成的基于用户产品定义的业务逻辑框架代码。
— data_config.c	用户定义的数据点。
— events_config.c	用户定义的事件。
—data_template_usr_logic.c	用户业务处理逻辑框架，实现预留的上下行业务逻辑处理函数即可。
tools	代码生成脚本。
README.md	SDK 使用说明。

移植指引

开发平台	参考文档
MCU+ 定制 AT 模组（蜂窝类）	MCU+定制MQTT AT模组（蜂窝类）接入指引
MCU+ 定制 AT 模组（Wi-Fi类）	MCU+定制MQTT AT模组（Wi-Fi 类）接入指引

SDK 接口说明

关于 SDK 的更多使用方式及接口了解，请参见 qcloud_iot_api_export.h 。

Android SDK 使用参考

最近更新时间：2023-07-12 17:44:12

Android SDK 面向使用 Java 语言的平台实现接入腾讯云物联网开发平台。

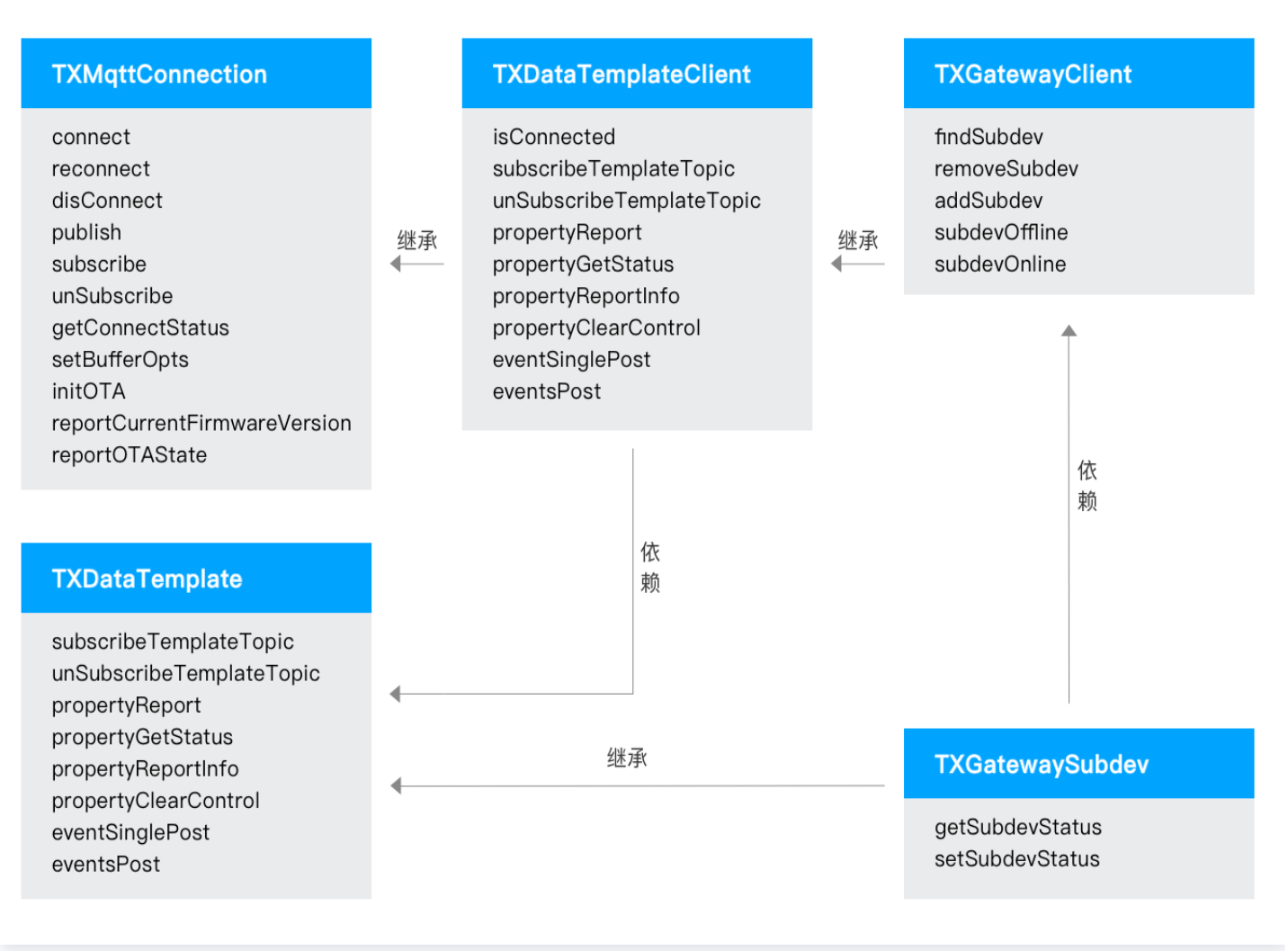
SDK 获取

SDK 使用 Github 托管，可访问 Github 下载最新版本设备端 iot-device-android 。

软件架构

类名	功能
TXMqttConnection	连接物联网开发平台。
TXDataTemplate	实现物模型基本功能。
TXDataTemplateClient	实现直连设备根据物模型连接物联网开发平台。
TXGatewayClient	实现网关设备根据物模型连接物联网开发平台。
TXGatewaySubdev	实现网关子设备根据物模型连接物联网开发平台。

腾讯云 IoT Explorer Android SDK 架构图如下：



移植指引

Android SDK移植指引，详情请参见 Android 平台接入指引 。

SDK API 说明

TXMqttConnection

方法名	说明
connect	MQTT 连接。
reconnect	MQTT 重连。
disConnect	断开 MQTT 连接。
publish	发布 MQTT 消息。

subscribe	订阅 MQTT 主题。
unSubscribe	取消订阅 MQTT 主题。
getConnectStatus	获取 MQTT 连接状态。
setBufferOpts	设置断连状态 buffer 缓冲区。
initOTA	初始化 OTA 功能。
reportCurrentFirmwareVersion	上报设备当前版本信息至后台服务器。
reportOTAState	上报设备升级状态到后台服务器。

TXDataTemplate

方法名	说明
subscribeTemplateTopic	订阅物模型相关主题。
unSubscribeTemplateTopic	取消订阅物模型相关主题。
propertyReport	上报属性。
propertyGetStatus	更新状态。
propertyReportInfo	上报设备信息。
propertyClearControl	清除控制信息。
eventSinglePost	上报单个事件。
eventsPost	上报多个事件。

TXDataTemplateClient

方法名	说明
isConnected	是否已经连接物联网开发平台。
subscribeTemplateTopic	订阅物模型相关主题。
unSubscribeTemplate	取消订阅物模型相关主题。

Topic	
propertyReport	上报属性。
propertyGetStatus	更新状态。
propertyReportInfo	上报设备信息。
propertyClearControl	清除控制信息。
eventSinglePost	上报单个事件。
eventsPost	上报多个事件。

TXGatewayClient

方法名	说明
findSubdev	查找子设备（根据产品 ID 和设备名称）。
removeSubdev	删除子设备。
addSubdev	添加子设备。
subdevOffline	上线子设备。
subdevOnline	下线子设备。

TXGatewaySubdev

方法名	说明
getSubdevStatus	获取子设备连接状态。
setSubdevStatus	设置子设备连接状态。

Java SDK 使用参考

最近更新时间：2023-07-12 17:44:12

Java SDK 面向使用 Java 语言的平台实现接入腾讯云物联网开发平台。

SDK 获取

SDK 使用 Github 托管，可访问 Github 下载最新版本设备端 explorer-device-java 。
如果您想通过 jar 引用方式进行项目开发，可在 module 目录下的 build.gradle 中添加依赖，如下依赖：

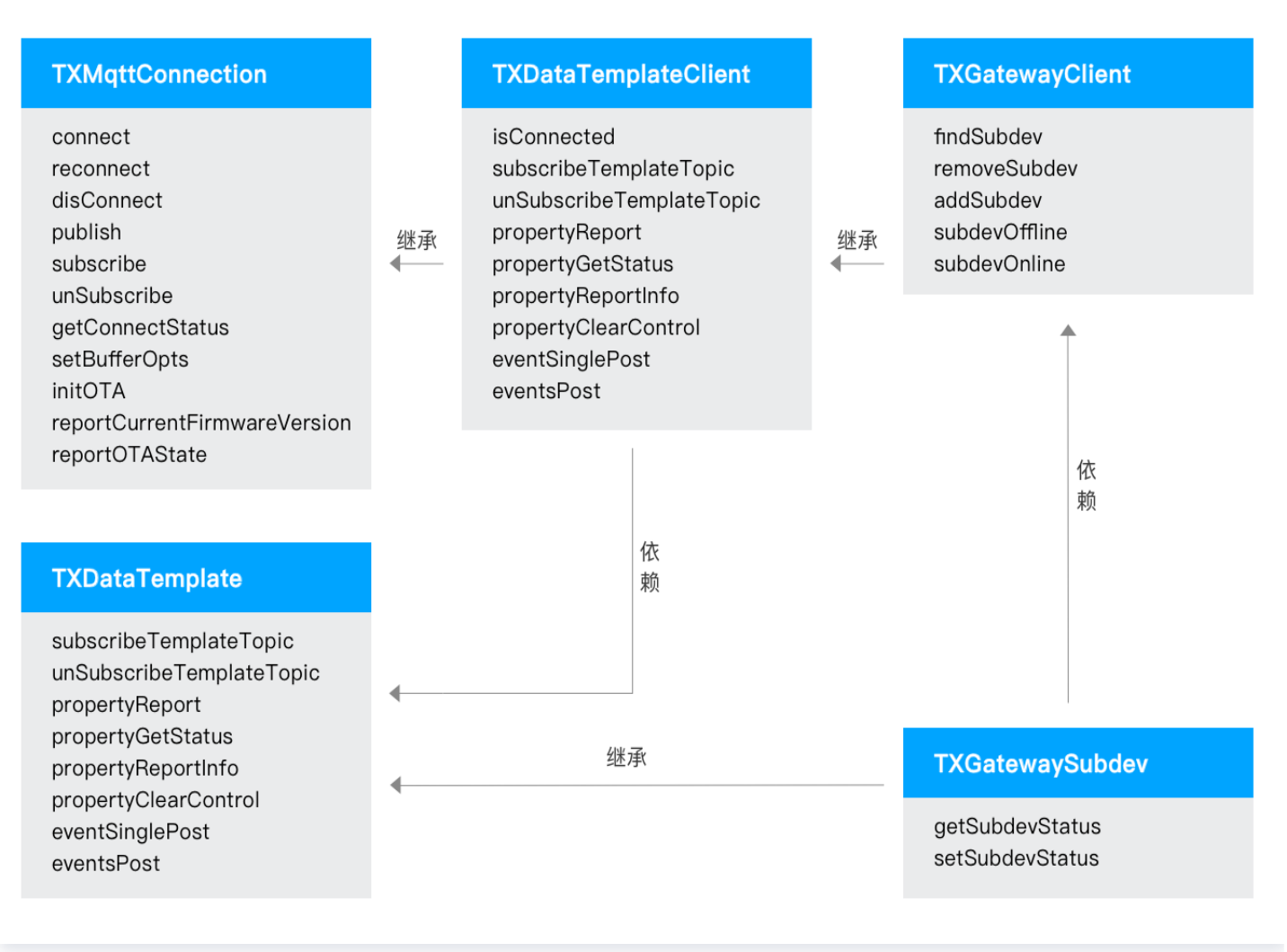
```
dependencies {
 ...
 implementation 'com.tencent.iot.explorer:explorer-device-java:x.x.x'
}
```

说明：
用户可根据 版本说明 把上述 x.x.x 设置成最新版本。

软件架构

类名	功能
TXMqttConnection	连接物联网开发平台。
TXDataTemplate	实现物模型基本功能。
TXDataTemplateClient	实现直连设备根据物模型连接物联网开发平台。
TXGatewayClient	实现网关设备根据物模型连接物联网开发平台。
TXGatewaySubdev	实现网关子设备根据物模型连接物联网开发平台。

腾讯云 IoT Explorer Java SDK 架构图如下：



移植指引

Java SDK 移植指引，详情请参见 Java 平台接入指引 。

SDK API 说明

TXMqttConnection

方法名	说明
connect	MQTT 连接。
reconnect	MQTT 重连。
disConnect	断开 MQTT 连接。
publish	发布 MQTT 消息。

subscribe	订阅 MQTT 主题。
unSubscribe	取消订阅 MQTT 主题。
getConnectStatus	获取 MQTT 连接状态。
setBufferOpts	设置断连状态 buffer 缓冲区。
initOTA	初始化 OTA 功能。
reportCurrentFirmwareVersion	上报设备当前版本信息至后台服务器。
reportOTAState	上报设备升级状态到后台服务器。

TXDataTemplate

方法名	说明
subscribeTemplateTopic	订阅物模型相关主题。
unSubscribeTemplateTopic	取消订阅物模型相关主题。
propertyReport	上报属性。
propertyGetStatus	更新状态。
propertyReportInfo	上报设备信息。
propertyClearControl	清除控制信息。
eventSinglePost	上报单个事件。
eventsPost	上报多个事件。

TXDataTemplateClient

方法名	说明
isConnected	是否已经连接物联网开发平台。
subscribeTemplateTopic	订阅物模型相关主题。
unSubscribeTemplateTopic	取消订阅物模型相关主题。
propertyReport	上报属性。
propertyGetStatus	更新状态。

propertyReportInfo	上报设备信息。
propertyClearControl	清除控制信息。
eventSinglePost	上报单个事件。
eventsPost	上报多个事件。

TXGatewayClient

方法名	说明
findSubdev	查找子设备（根据产品 ID 和设备名称）。
removeSubdev	删除子设备。
addSubdev	添加子设备。
subdevOffline	上线子设备。
subdevOnline	下线子设备。

TXGatewaySubdev

方法名	说明
getSubdevStatus	获取子设备连接状态。
setSubdevStatus	设置子设备连接状态。

Python SDK 使用参考

最近更新时间：2024-09-30 17:54:51

腾讯云物联网设备端 Python SDK 依靠安全且性能强大的数据通道，为物联网领域开发人员提供设备端快速接入云端，并和云端进行双向通信的能力。开发人员只需完成工程的相应配置即可完成设备的接入。

前提条件

已在 Explorer 平台创建好产品和设备。

引用方式

- 如果您需要通过引用方式进行项目开发，可安装 SDK，如下：

```
pip3 install tencent-iot-device
```

如果您需查看使用的 SDK 版本，使用如下指令：

```
pip3 show --files tencent-iot-device
```

如果您需升级 SDK 版本，使用如下指令：

```
pip3 install --upgrade tencent-iot-device
```

- 如果您需要通过代码集成方式进行项目开发，可访问 Github 下载 Python SDK 源码。

MQTT 接口

MQTT 的相关接口定义在 explorer.py 类中，支持发布和订阅功能，介绍如下：

方法名	说明
connect	MQTT 连接。
disconnect	断开 MQTT 连接。
subscribe	MQTT 订阅。
unsubscribe	MQTT 取消订阅。
publish	MQTT 发布消息。
registerMqttCallback	注册 MQTT 回调函数。

registerUserCallback	注册用户回调函数。
isMqttConnected	MQTT 是否正常连接。
getConnectState	获取 MQTT 连接状态。
setReconnectInterval	设置 MQTT 重连尝试间隔。
setMessageTimout	设置消息发送超时时间。
setKeepaliveInterval	设置 MQTT 保活间隔。

MQTT 网关接口

- 对于无法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，将代理设备接入 explorer 平台。
- 对于局域网中加入或退出网络的子设备，需通过平台进行绑定或解绑操作。

> ⚠ **注意：**
> 当子设备发起过上线，后续只要网关连接成功，后台就会显示子设备在线，除非设备已发起下线操作。

MQTT 网关的相关接口定义在 gateway.py 类接口中，介绍如下：

方法名	说明
gatewayInit	网关初始化。
isSubdevStatusOnline	判断子设备是否在线。
updateSubdevStatus	更新子设备在线状态。
gatewaySubdevGetConfigList	获取配置文件中子设备列表。
gatewaySubdevOnline	代理子设备上线。
gatewaySubdevOffline	代理子设备下线。
gatewaySubdevBind	绑定子设备。
gatewaySubdevUnbind	解绑子设备。
gatewaySubdevSubscribe	子设备订阅。

物模型接口

如果需要使用物模型功能，需使用 template.py 类中的接口，介绍如下：

接口名称	接口描述
templateInit	物模型初始化。
getEventsList	获取设备 event 列表。
getActionList	获取设备 action 列表。
getPropertyList	获取设备 property 列表。
templateSetup	解析物模型。
templateEventPost	events 上报。
templateJsonConstructReportArray	构建上报的 json 结构。
templateReportSysInfo	设备信息上报。
templateControlReply	控制消息应答。
templateActionReply	action 消息应答。
templateGetStatus	获取设备最新状态。
templateReport	设备属性上报。
clearControl	清除控制。
templateDeinit	物模型销毁。

动态注册接口

如果需要使用动态注册功能，需使用 explorer.py 类中的接口，介绍如下：

方法名	说明
dynregDevice	获取设备动态注册的信息。

OTA 接口

如果需要使用 OTA 功能，需使用 explorer.py 类中的接口，介绍如下：

方法名	说明
otaInit	OTA 初始化。

otaIsFetching	判断是否正在下载。
otaIsFetchFinished	判断是否下载完成。
otaReportUpgradeSuccess	上报升级成功消息。
otaReportUpgradeFail	上报升级失败消息。
otaIoctlNumber	获取下载固件大小等 int 类型信息。
otaIoctlString	获取下载固件 md5 等 string 类型信息。
otaResetMd5	重置 md5 信息。
otaMd5Update	更新 md5 信息。
httpInit	初始化 HTTP。
otaReportVersion	上报当前固件版本信息。
otaDownloadStart	开始固件下载。
otaFetchYield	读取固件。

C# 接入参考

最近更新时间：2024-09-29 18:24:51

C# 是一种现代通用的、面向对象的编程语言，由微软在 .NET 框架中开发并推出。它结合了 C++ 的强大功能和 Java 的易用性，使得开发者能够构建各种类型的应用程序，包括 Windows 客户端应用、Web 应用、数据库应用、移动应用和游戏等。
腾讯云物联网开发平台支持使用 C# 接入。本文介绍如何在 C# 项目中，使用 MQTTnet Client 库，实现与腾讯云物联网开发平台的连接、topic 订阅、上下行消息交互等功能。

说明：
1. 本示例基于 donet 6.0。
2. 第三方类库使用版本如下：
   - MQTTnet：v3.0.11
   - MQTTnet.Extensions.ManagedClient：v3.0.11
   - Newtonsoft.Json：v12.0.3
   - NLog：v5.2.3

操作场景

一款智能灯接入到物联网开发平台，通过物联网开发平台可以远程控制灯的亮度、颜色、开关，并实时获取智能灯上报到开发平台的数据。

准备工作

1. 申请 腾讯云物联网开发平台 服务。
2. 一台 Windows 电脑并且安装了 VS2017 及以上版本。

操作步骤

创建项目

1. 登录 物联网开发平台控制台 ，选择平台默认开通的**公共实例**或用户购买的**企业实例**。
2. 单击实例后，默认进入项目列表页面，单击**新建项目**。
   - 项目名称：必填，输入“智能灯演示”或其他名称。
   - 项目描述：按照实际需求填写项目描述。

3. 项目基本信息填写完成后，单击**保存**，即可完成新建项目。
4. 项目新建成功后，即可新建产品。

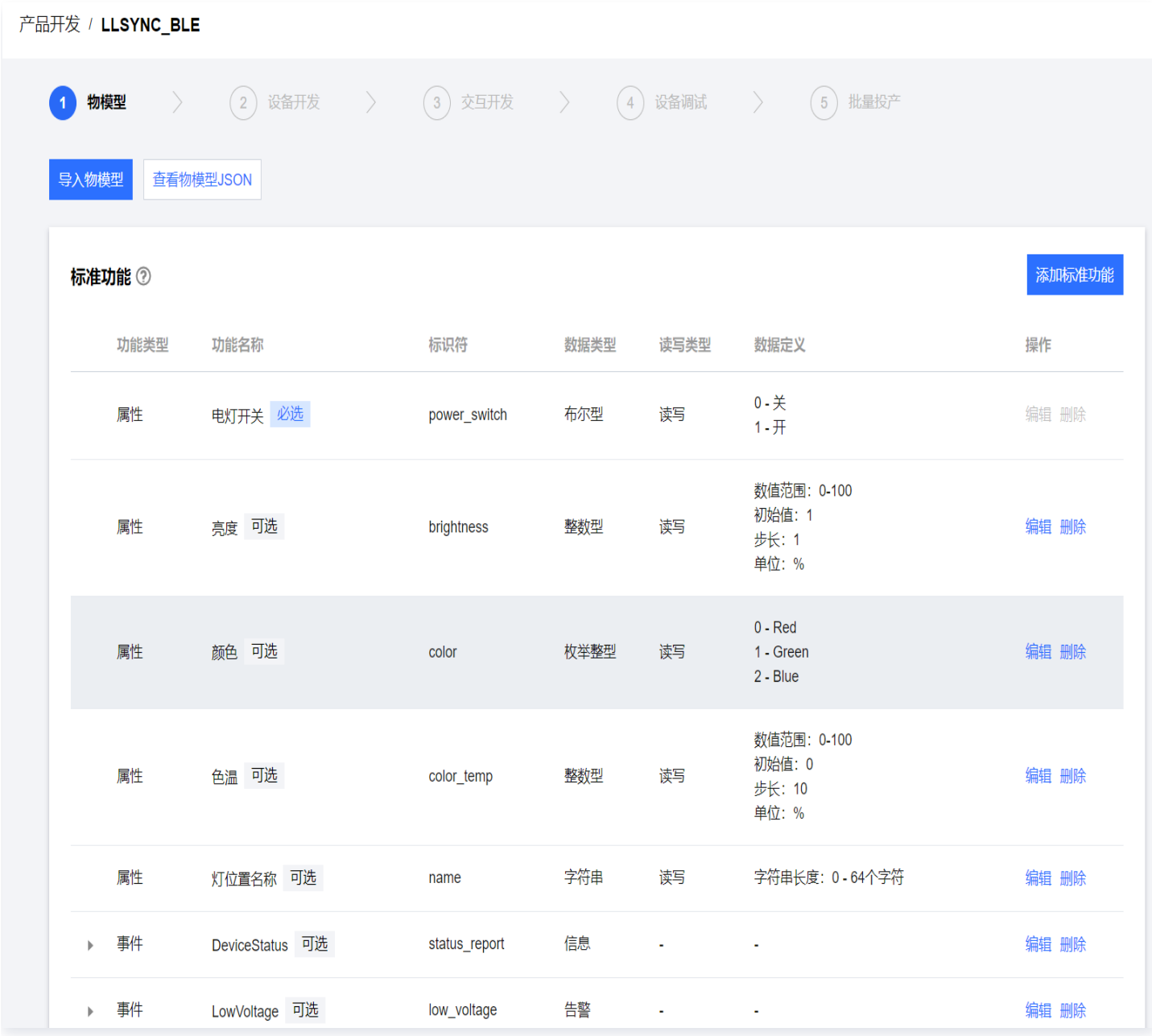
新建产品

1. 单击项目名称，进入产品列表页面，单击**新建产品**。
2. 在新建产品页面，填写产品基本信息。
   - 产品名称：必填，输入“智能灯”或其他产品名称。
   - 产品品类：选择标准品类 “智能生活” > “电工照明” > “灯” 。
   - 设备类型：选择“设备” 。
   - 认证方式：选择“密钥认证” 。
   - 通信方式：按需选择。
   - 其他都为默认选项。

3. 产品信息填写完成后，单击**保存**，即可完成新建产品。
4. 产品新建成功后，您可在产品列表页查看到“智能灯”。

定义产品物模型

选择“灯”类型后，系统会自动生成标准功能。


产品开发 / LLSYNC_BLE
1 物模型
2 设备开发
3 交互开发
4 设备调试
5 批量投产
导入物模型
查看物模型JSON
标准功能
添加标准功能
功能类型 功能名称 标识符 数据类型 读写类型 数据定义 操作
属性 电灯开关 必选 power_switch 布尔型 读写 0 - 关 1 - 开 编辑 删除
属性 亮度 可选 brightness 整数型 读写 数值范围：0-100 初始值：1 步长：1 单位：% 编辑 删除
属性 颜色 可选 color 枚举整型 读写 0 - Red 1 - Green 2 - Blue 编辑 删除
属性 色温 可选 color_temp 整数型 读写 数值范围：0-100 初始值：0 步长：10 单位：% 编辑 删除
属性 灯位置名称 可选 name 字符串 读写 字符串长度：0 - 64个字符 编辑 删除
事件 DeviceStatus 可选 status_report 信息 - - 编辑 删除
事件 LowVoltage 可选 low_voltage 告警 - - 编辑 删除

创建设备

在设备调试页面中，单击**新建设备**，设备名为 dev001。

有关物模型的协议介绍请参阅 物模型协议 。

C# MQTT 客户端使用

通过以上步骤，您已经成功拿到腾讯云物联网开发平台的设备三元组信息，接下来可以根据这个示例来使用 C# 接入。

输入三元组信息

单击创建的设备名称，获取设备三元组信息。

```
// 云端生成的三元组信息
static string PRODUCT_ID = "YOUR_PRODUCT_ID";  // 产品id
static string DEVICE_NAME = "YOUR_DEVICE_NAME"; // 设备名称
static string DEVICE_SECRET = "IOT_PSK";  // 设备密钥
```

生成 MQTT client 连接参数

通过刚才填入的三元组信息，生成 MQTT 的 `clientId` 、 `userName` 、 `password` 。

```
// 生成MQTT连接参数
byte[] decodeBytes = Convert.FromBase64String(DEVICE_SECRET);
string clientId = PRODUCT_ID + DEVICE_NAME;
string usrNmae = clientId + ";21010406;" + GetNextConnId() +";"+
0x7fffffff.ToString();
string password = ComputeHmacSha1(usrNmae, decodeBytes) + ";hmacsha1";
```

创建 MQTT client 并连接云平台

```
IManagedMqttClient mqttClient = new
MqttFactory().CreateManagedMqttClient();
```

```
var mqttOptions = new ManagedMqttClientOptionsBuilder()
    .WithAutoReconnectDelay(TimeSpan.FromSeconds(10))
    .WithClientOptions(new MqttClientOptionsBuilder()
        .WithClientId(clientId)
        .WithCredentials(usrNmae, password)
        .WithTcpServer(url, 1883) // 非tls模式
        .WithCleanSession()
        .Build())
    .Build();
```

如果使用 TLS 加密接入，则按照如下方式配置：

```
IManagedMqttClient mqttClient = new
MqttFactory().CreateManagedMqttClient();

var mqttOptions = new ManagedMqttClientOptionsBuilder()
    .WithAutoReconnectDelay(TimeSpan.FromSeconds(10))
    .WithClientOptions(new MqttClientOptionsBuilder()
        .WithClientId(clientId)
        .WithCredentials(usrNmae, password)
        .WithTcpServer(url, 8883) // tls 接入
        .WithTls(new MqttClientOptionsBuilderTlsParameters {
            UseTls = true,
            IgnoreCertificateChainErrors = true,
            IgnoreCertificateRevocationErrors = true,
            AllowUntrustedCertificates = true,
        })
        .WithCleanSession()
        .Build())
    .Build();
```

监听 MQTT 连接或断连或接收事件

```
// 监听连接事件
mqttClient.UseConnectedHandler(e =>
{
    log.Info("mqtt connect success with " + deviceId);
    return Task.CompletedTask;
});

// 监听断开事件
```

```
mqttClient.UseDisconnectedHandler(e =>
{
    log.Error("mqtt disconnect with " + deviceId);
    return Task.CompletedTask;
});

// 监听收到的消息
mqttClient.UseApplicationMessageReceivedHandler(e =>
{

    // 处理物模型数据
    string topic = e.ApplicationMessage.Topic;
    string payload =
Encoding.UTF8.GetString(e.ApplicationMessage.Payload);
    log.Debug($"down message topic: {topic} paylaod : {payload}");
    if (topic.Contains("/property/"))
    {
        // 属性处理

    }else if (topic.Contains("/event/"))
    {
        // 事件处理

    }else if (topic.Contains("/action/"))
    {
        // 行为处理

    }
    return Task.CompletedTask;
});
```

连接云平台

```
// 连接到平台
await mqttClient.StartAsync(mqttOptions);
```

订阅物模型 topic

```
 // 订阅物模型topic
var topicFilters = new[]
{
```

```
    new
MqttTopicFilterBuilder().WithTopic("$thing/down/property/"+deviceId).Bui
ld(),
    new
MqttTopicFilterBuilder().WithTopic("$thing/down/event/"+deviceId).Build(
),
    new
MqttTopicFilterBuilder().WithTopic("$thing/down/action/"+deviceId).Build
()
 };
await mqttClient.SubscribeAsync(topicFilters);
```

上报亮度示例

```
  int brightness = 25;
  var payload_json = new JObject
  {
      ["method"] = "report",
      ["clientToken"] =
DateTimeOffset.Now.ToUnixTimeSeconds().ToString(),
      ["params"] = new JObject
      {
          ["brightness"] = brightness
      }
  };
  string payload = JsonConvert.SerializeObject(payload_json).ToString();
  var message = new MqttApplicationMessageBuilder()
      .WithTopic("$thing/up/property/" + deviceId)
      .WithPayload(payload)
      .WithQualityOfServiceLevel(0)
      .WithRetainFlag(false)
      .Build();
  await mqttClient.PublishAsync(message);
```

完整的示例代码如下：

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Security.Cryptography;
using NLog;
using MQTTnet;
```

```
using MQTTnet.Client.Options;
using MQTTnet.Extensions.ManagedClient;
using Newtonsoft.Json;
using Newtonsoft.Json.Linq;


namespace IoT.Explorer.Test
{
    class DatatemplateTest
    {
        private static Logger log = LogManager.GetCurrentClassLogger();
        // 云端生成的三元组信息
        static string PRODUCT_ID = "F2F43QKKA4";
        static string DEVICE_NAME = "5629fbfa12f4";
        static string DEVICE_SECRET = "a91V4htL41oILv80lgCeLA==";

        public static string GetNextConnId()
        {
            char[] connId = new char[6];
            Random random = new Random();
            for (int i = 0; i < 6 - 1; i++)
            {
                int flag = random.Next(3);
                switch (flag)
                {
                    case 0:
                        connId[i] = (char)(random.Next(26) + 'a');
                        break;
                    case 1:
                        connId[i] = (char)(random.Next(26) + 'A');
                        break;
                    case 2:
                        connId[i] = (char)(random.Next(10) + '0');
                        break;
                }
            }

            connId[6 - 1] = '\0';
            return new string(connId);
        }

        public static string ComputeHmacSha1(string inputString, byte[]
keyBytes)
```

```
        {
            byte[] inputBytes = Encoding.UTF8.GetBytes(inputString);

            using (HMACSHA1 hmac = new HMACSHA1(keyBytes))
            {
                byte[] hashBytes = hmac.ComputeHash(inputBytes);
                string hashString =
BitConverter.ToString(hashBytes).Replace("-", "").ToLower();
                return hashString;
            }
        }

        static async Task Main(string[] args)
        {
            // 生成MQTT连接参数
            byte[] decodeBytes =
Convert.FromBase64String(DEVICE_SECRET);
            string clientId = PRODUCT_ID + DEVICE_NAME;
            string usrNmae = clientId + ";21010406;" + GetNextConnId()
+";"+ 0x7fffffff.ToString();
            string password = ComputeHmacSha1(usrNmae, decodeBytes) +
";hmacsha1";
            string url = PRODUCT_ID + ".iotcloud.tencentdevices.com";
            string deviceId = PRODUCT_ID + "/" + DEVICE_NAME;

            try
            {
                IManagedMqttClient mqttClient = new
MqttFactory().CreateManagedMqttClient();

                var mqttOptions = new ManagedMqttClientOptionsBuilder()
                    .WithAutoReconnectDelay(TimeSpan.FromSeconds(10))
                    .WithClientOptions(new MqttClientOptionsBuilder()
                        .WithClientId(clientId)
                        .WithCredentials(usrNmae, password)
                        .WithTcpServer(url, 8883)
                        .WithTls(new
MqttClientOptionsBuilderTlsParameters {
                            UseTls = true,
                            IgnoreCertificateChainErrors = true,
                            IgnoreCertificateRevocationErrors = true,
                            AllowUntrustedCertificates = true,
                        })
```

```
                    .WithCleanSession()
                    .Build())
                .Build();

            // 监听连接事件
            mqttClient.UseConnectedHandler(e =>
            {
                log.Info("mqtt connect success with " + deviceId);
                return Task.CompletedTask;
            });

            // 监听断开事件
            mqttClient.UseDisconnectedHandler(e =>
            {
                log.Error("mqtt disconnect with " + deviceId);
                return Task.CompletedTask;
            });

            // 监听收到的消息
            mqttClient.UseApplicationMessageReceivedHandler(e =>
            {

                // 处理物模型数据
                string topic = e.ApplicationMessage.Topic;
                string payload =
Encoding.UTF8.GetString(e.ApplicationMessage.Payload);
                log.Debug($"down message topic: {topic} paylaod :
{payload}");
                if (topic.Contains("/property/"))
                {
                    // 属性处理

                }else if (topic.Contains("/event/"))
                {
                    // 事件处理

                }else if (topic.Contains("/action/"))
                {
                    // 行为处理

                }
                return Task.CompletedTask;
            });
```

```
                // 连接到平台
                await mqttClient.StartAsync(mqttOptions);
                Thread.Sleep(2000);

                // 订阅物模型topic
                var topicFilters = new[]
                {
                    new
MqttTopicFilterBuilder().WithTopic("$thing/down/property/"+deviceId).Bui
ld(),
                    new
MqttTopicFilterBuilder().WithTopic("$thing/down/event/"+deviceId).Build(
),
                    new
MqttTopicFilterBuilder().WithTopic("$thing/down/action/"+deviceId).Build
()
                 };
                await mqttClient.SubscribeAsync(topicFilters);
                int brightness = 0;
                while (true) {
                    // 周期上报亮度
                    var payload_json = new JObject
                    {
                        ["method"] = "report",
                        ["clientToken"] =
DateTimeOffset.Now.ToUnixTimeSeconds().ToString(),
                        ["params"] = new JObject
                        {
                            ["brightness"] = brightness
                        }
                    };
                    string payload =
JsonConvert.SerializeObject(payload_json).ToString();
                    var message = new MqttApplicationMessageBuilder()
                        .WithTopic("$thing/up/property/" + deviceId)
                        .WithPayload(payload)
                        .WithQualityOfServiceLevel(0)
                        .WithRetainFlag(false)
                        .Build();
                    await mqttClient.PublishAsync(message);
                    log.Debug("publish message :" + payload);
                    brightness++;
```

```
                    if (!mqttClient.IsConnected)
                    {
                        break;
                    }
                    Thread.Sleep(10000);
                }
                // disconnect
                await mqttClient.StopAsync();

            }
            catch (Exception ex)
            {
                var name = ex.GetType().FullName;
                log.Error("异常: "+ex.Message);
            }

        }
    }
}
```

运行日志如下：

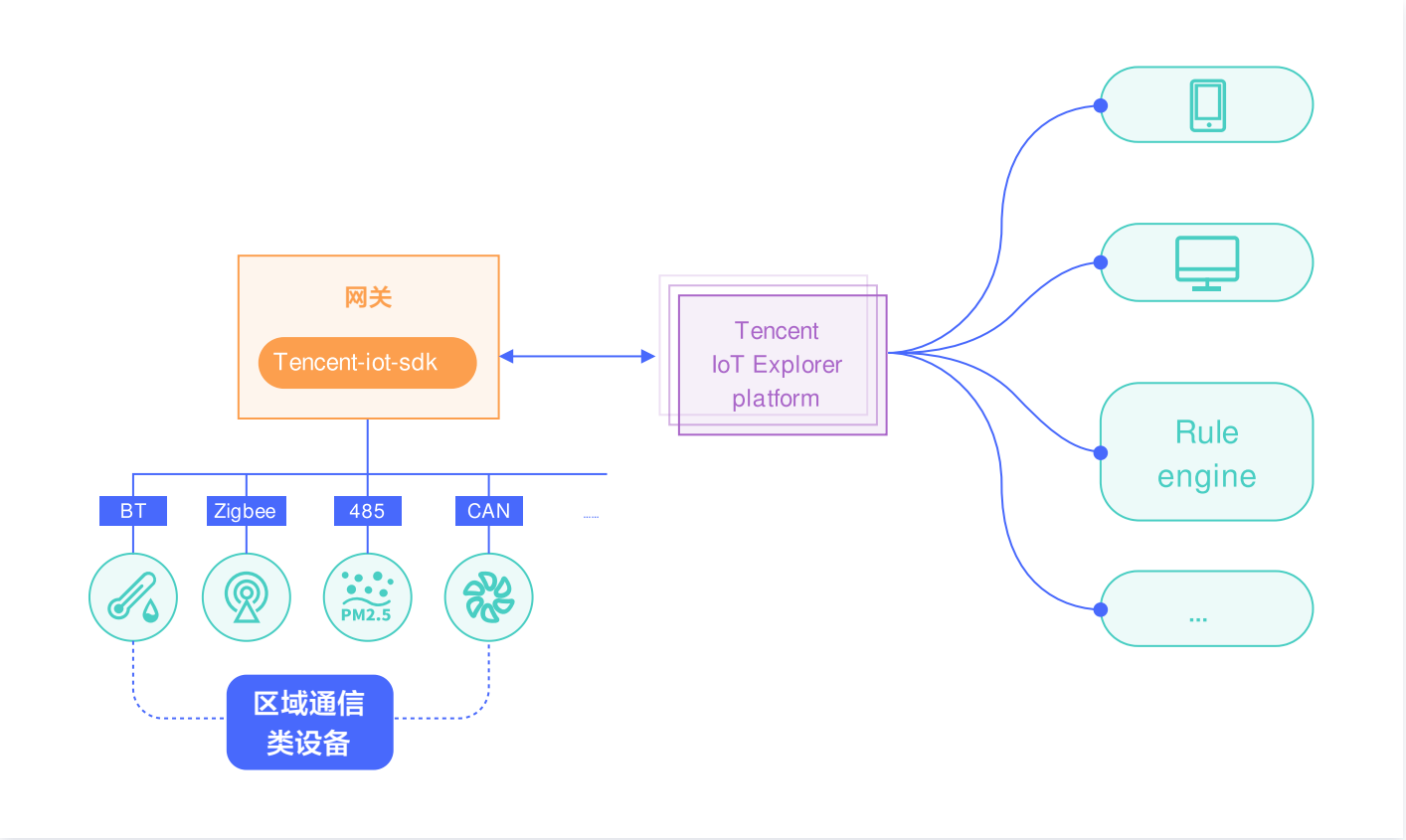
```
2023-08-09 12:04:20.0860 INFO IoT.Explorer.Test.DatatemplateTest - mqtt
connect success with F2F43QKKA4/5629fbfa12f4
2023-08-09 12:04:21.7918 DEBUG IoT.Explorer.Test.DatatemplateTest -
publish message :{"method":"report","clientToken":"1691553861","params":
{"brightness":0}}
2023-08-09 12:04:21.8791 DEBUG IoT.Explorer.Test.DatatemplateTest - down
message topic: $thing/down/property/F2F43QKKA4/5629fbfa12f4 paylaod :
{"method":"report_reply","clientToken":"1691553861","code":0,"status":"s
uccess"}
2023-08-09 12:04:25.6359 DEBUG IoT.Explorer.Test.DatatemplateTest - down
message topic: $thing/down/property/F2F43QKKA4/5629fbfa12f4 paylaod :
{"method":"control","clientToken":"v2149648760ozOCb::af400362-c384-40dd-
b39c-94d82950e1ad","params":{"power_switch":1}}
2023-08-09 12:04:29.4171 DEBUG IoT.Explorer.Test.DatatemplateTest - down
message topic: $thing/down/property/F2F43QKKA4/5629fbfa12f4 paylaod :
{"method":"control","clientToken":"v2146761678bxCmt::295125a1-6b64-4cba-
b2ca-036bbef43390","params":{"power_switch":0}}
2023-08-09 12:04:31.7989 DEBUG IoT.Explorer.Test.DatatemplateTest -
publish message :{"method":"report","clientToken":"1691553871","params":
{"brightness":1}}
```

```
2023-08-09 12:04:31.8919 DEBUG IoT.Explorer.Test.DatatemplateTest - down
message topic: $thing/down/property/F2F43QKKA4/5629fbfa12f4 paylaod :
{"method":"report_reply","clientToken":"1691553871","code":0,"status":"s
uccess"}
```

网关及子设备开发
网关及子设备接入说明

最近更新时间：2022-06-10 15:24:24

对于 BLE、Zigbee 和485等不具备直接访问网络能力的设备，需要先接入网关，然后通过网关代理，间接实现设备接入腾讯物联网开发平台 IoT Explorer，具体流程框架图如下：



术语定义

- 网关设备：具备北向和南向通信能力：
  - 北向：可以和云端进行数据交互。
  - 南向：可以通过有线或无线方式挂载子设备，具备子设备管理、代理子设备通信的能力。
- 子设备：不具备直接访问网络能力的设备，通过网关代理间接实现和云端数据交互，例如 BLE、Zigbee、485、433等设备。
- 拓扑关系：子设备和网关的关联关系为拓扑关系，网关和子设备需要建立关联的拓扑关系，网关才能代理子设备进行上下线和通信。网关设备和子设备建立关联，需要在产品级实现关联，然后在设备级实现关联。即先把子设备对应产品和网关对应的产品做关联，然后把具体的子设备关联到具体的网关设备。

开发流程

网关和子设备的场景，需要在网关侧和子设备侧分别进行开发，请分别参考下面的文档链接。

- 网关设备开发
- 子设备开发

网关设备接入指引

最近更新时间：2022-06-10 15:24:31

本文为您介绍如何基于 物联网开发平台 C SDK 开发网关接入腾讯物联网开发平台。

C SDK 网关特性

网关和子设备信息管理

网关和子设备的 HAL 层设备信息管理接口和直连设备有所区别，调用 `HAL_GetGwDevInfo` 获取网关和子设备的设备信息，此接口需要适配实现，即开发者需要根据目标平台和应用场景实现设备信息的管理，尤其是真实子设备和平台子设备的设备信息映射。

网关与云端建连

网关示例 `gateway_sample.c` 介绍如何使用 C SDK 提供的 API 实现设备信息获取、网关与云端建连、代理子设备上线及下线、代理子设备通信，并且展示了子设备如何基于数据模板进行通信。

代理子设备上下线

网关上线后，通过特定的协议（例如：BLE、Zigbee 等）进行子设备管理。

- 对于已经建立通信的子设备，调用 `IOT_Gateway_Subdev_Online` 实现子设备在云端的上线，详情可参考 上线数据请求格式
- 对于失去通信的子设备，调用 `IOT_Gateway_Subdev_Offline` 实现子设备在云端的下线，详情可参考 下线数据请求格式 。

动态绑定解绑子设备

网关能代理子设备上下线及通信，前提是网关设备和子设备已在平台建立拓扑关联关系。

- 产品级的关联，即子设备的产品和网关产品关联必须在 控制台建立 。
- 设备级的关联，可以在控制台建立，并且可以动态绑定和解绑。动态绑定和解绑，网关需要有子设备密钥实现签名，如果签名在网关侧计算，网关需要有对应待绑定子设备的密钥， 动态绑定解绑数据请求格式 。

代理子设备通信

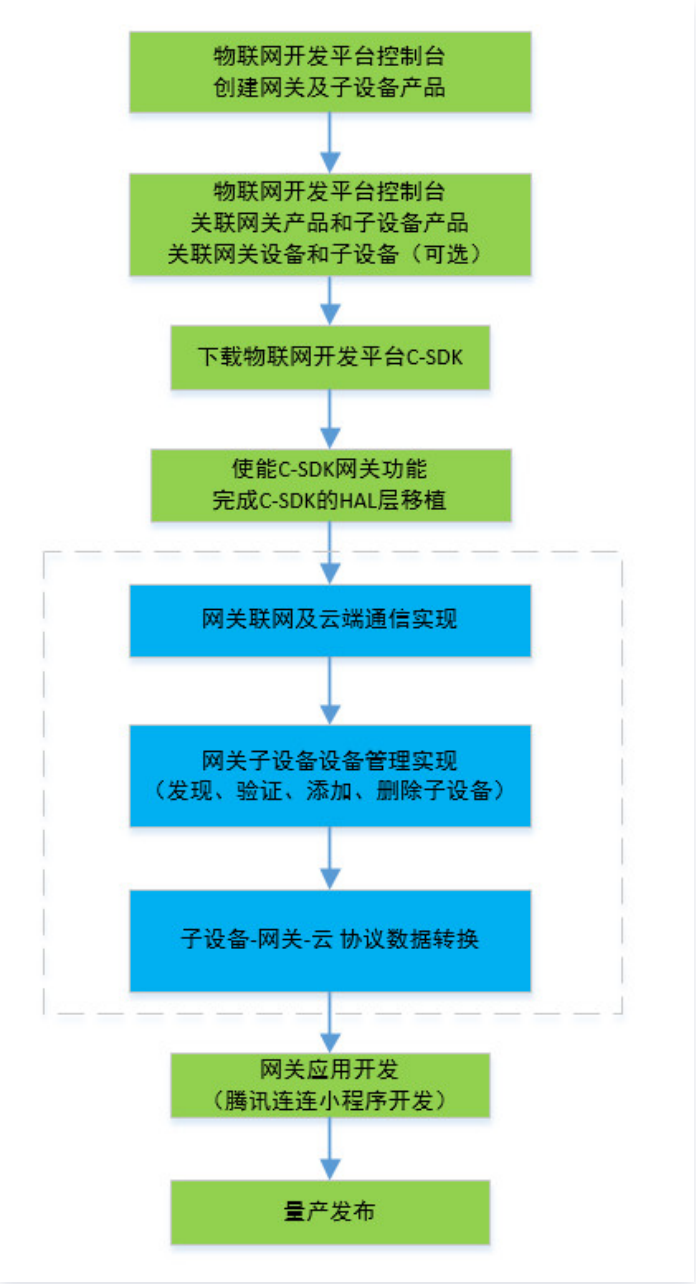
网关代理子设备上线成功后，即可调用 `IOT_Gateway_Subscribe` 代理子设备的订阅消息，调用 `IOT_Gateway_Publish` 向云端推送消息。

网关示例

网关示例 `gateway_sample.c` 介绍使用 C SDK 提供的 API，如何实现设备信息获取、网关与云端建连、代理子设备上线及下线、代理子设备通信，并且示例了子设备如何基于数据模板进行通信。

网关开发实现

网关产品通常的开发流程如下：



虚线框内的三个步骤由网关厂商实现，通常建议如下：

- 对于网关设备是 Wi-Fi 联网的方式，需要参照 Wi-Fi 配网协议 配合腾讯连连小程序实现配网。
- 对于网关和子设备的通信，平台没有限定，由网关厂商和子设备自由定义。但网关要实现对子设备的设备信息管理及合法性校验，需要和云端关联的子设备做好映射，一般会用子设备的唯一信息（MAC/EUI）作为平台子设备的设备名，在子设备入网过程网关实现映射关系的管理。
- 网关和物联网开发平台的通信协议必须是 数据模板协议，每一个产品类型、数据模板是不一样的，网关和子设备的通信一般是二进制。所以网关需要基于不同类型的产品，实现子设备二进制数据到数据模板协议数据的双向转换。一般的做法，创建两个消息队列和子设备到网关的消息队列 `stack_msg_queue` 及云端到网关的消息队列 `cloud_msg_queue`，独立线程中分别从消息队列中取消息，进行相应的转换后做消息的分发及推送。
- 对于蓝牙场景，腾讯有定义 连连 LLSync 协议，接入腾讯物联网平台的网关，网关和子设备的通信协议建议使用连连 LLSync 协议。

子设备接入指引

最近更新时间：2022-06-10 15:24:43

物联网开发平台 C SDK 是面向可以和平台直接通信的设备，由于子设备无法直接和平台进行通信，所以子设备不需要集成物联网开发平台 C SDK。理论上，用户可以完全自定义子设备和网关的交互逻辑，但为使子设备能和平台实现交互，本文提供一些常用的处理建议。

子设备设备信息管理

网关需要取得子设备的设备信息，才能代理子设备实现相应的功能。

子设备发现及鉴权验证

网关对子设备的发现逻辑，基于不同的通信方式（例如 BLE、Zigbee 和485等）自由实现，但设备发现后，建议做设备的合法性鉴权。鉴权可以结合平台的设备信息进行实现，通常建议子设备对应的产品选择密钥认证方式，每一个子设备在平台都有对应的三元组信息，即 productId + deviceName + psk。如果网关侧保存了子设备的 psk 信息，可以基于 psk 做签名验证。

1. 腾讯连连小程序内单击发现子设备按钮。
   腾讯连连小程序进入绑定主设备的流程有两种方式：
   - 在网关设备界面，单击 "+" 添加子设备按钮。

- 扫描控制台子设备的产品开发界面内批量投产处设备二维码。

2. 网关设备会收到一条 search_devices 的消息，此时，网关设备就可以启动发现搜索子设备的流程。网关收到搜索子设备的 payload：`{"payload":{"status":1},"type":"search_devices"}` ，网关回复开始搜索 payload：

```
{"type":"search_devices",
"payload":{"status":1, "result":0}}
```

3. 这里的搜索逻辑由客户自行实现，当成功发现子设备(发现条件由客户自定义)后，就可以绑定该子设备到网关，具体的绑定可以参考 IOT_Gateway_Subdev_Bind 的实现，然后小程序界面即显示刚才绑定成功的设备。

注意：
子设备一般是指不能直接连接物联网开发平台，需要通过网关连接平台的设备。网关产品需要添加子产品后才能添加网关设备下的子设备。网关子设备的拓扑关系及参数字段的描述，请参考这里。

子设备动态绑定

网关设备和子设备的拓扑绑定关系，可以在控制台预先完成，也可以在运行过程动态绑定，网关设备和子设备能够进行动态绑定的前置条件是网关产品和子设备产品已在控制台完成绑定关系。网关设备在发现子设备之后，是否需要发起动态绑定的判断逻辑，网关侧需要基于网关侧的设备管理逻辑开发实现。动态绑定 需要有基于子设备密钥的签名信息，签名计算参考 C SDK 接口 `subdev_bind_hmac_sha1_cal` ，签名的计算可以在子设备完成后发送给网关，也可以在网关侧完成，后者需要网关保存子设备的 psk 信息。
一个成功的绑定日志，可参考如下内容：

```
DBG|2022-01-14
16:57:25|mqtt_client_publish.c|qcloud_iot_mqtt_publish(346): publish
packetID=0|topicName=$gateway/operation/TZ9Y9CLTEA/gateway_001|payload=
{"type":"bind","payload":{"devices":
[{"product_id":"xxxx","device_name":"subdev_003","signature":"xxxxx","ra
ndom":1820758684,"timestamp":1642150644,"signmethod":"hmacsha1","authtyp
e":"psk"}]}}
```

```
INF|2022-01-14 16:57:25|gateway_sample.c|_message_handler(138): Receive
Message With topicName:$thing/down/property/HW9ME416BI/subdev_002,
payload:{"method":"report_reply","clientToken":"xxxx-
45","code":0,"status":"success"}
DBG|2022-01-14 16:57:25|gateway_common.c|_gateway_message_handler(419):
gateway recv : {"type":"bind","payload":{"devices":
[{"result":0,"product_id":"HW9ME416BI","device_name":"subdev_003"}]}}
INF|2022-01-14 16:57:25|gateway_common.c|_gateway_message_handler(510):
client_id(HW9ME416BI/subdev_003), bind result 0
DBG|2022-01-14 16:57:25|gateway_sample.c|show_subdev_bind_unbind(262):
bind HW9ME416BI/subdev_003 success
```

子设备动态注册

考虑产品的量产方便，子设备可以使用动态注册的方式获取子设备的三元组信息，动态注册 需要子设备的 产品密钥 作为签名，签名的计算可以在子设备侧完成后发送给网关，也可以在网关侧完成，后者需要在网关侧保存子设备的 psk 信息，动态注册的子设备名建议取设备发现过程中网关可以获取的设备唯一信息，例如 MAC 地址、EUI 等。

子设备通信

网关和子设备通信

网关和子设备的通信协议及数据格式，平台不做限制，但网关代理子设备与云端通信时，则需要将子设备的消息转为 数据模板 格式数据。

子设备和子设备互通

通过控制台或者小程序配置子设备和子设备之间的消息转发规则，即可实现子设备互通。

音视频设备开发
P2P 接入指南

最近更新时间：2024-04-24 18:00:11

物联网开发平台为用户提供了基于 X-P2P 协议的 P2P 接入能力。本文档为您介绍 P2P 接入相关内容。

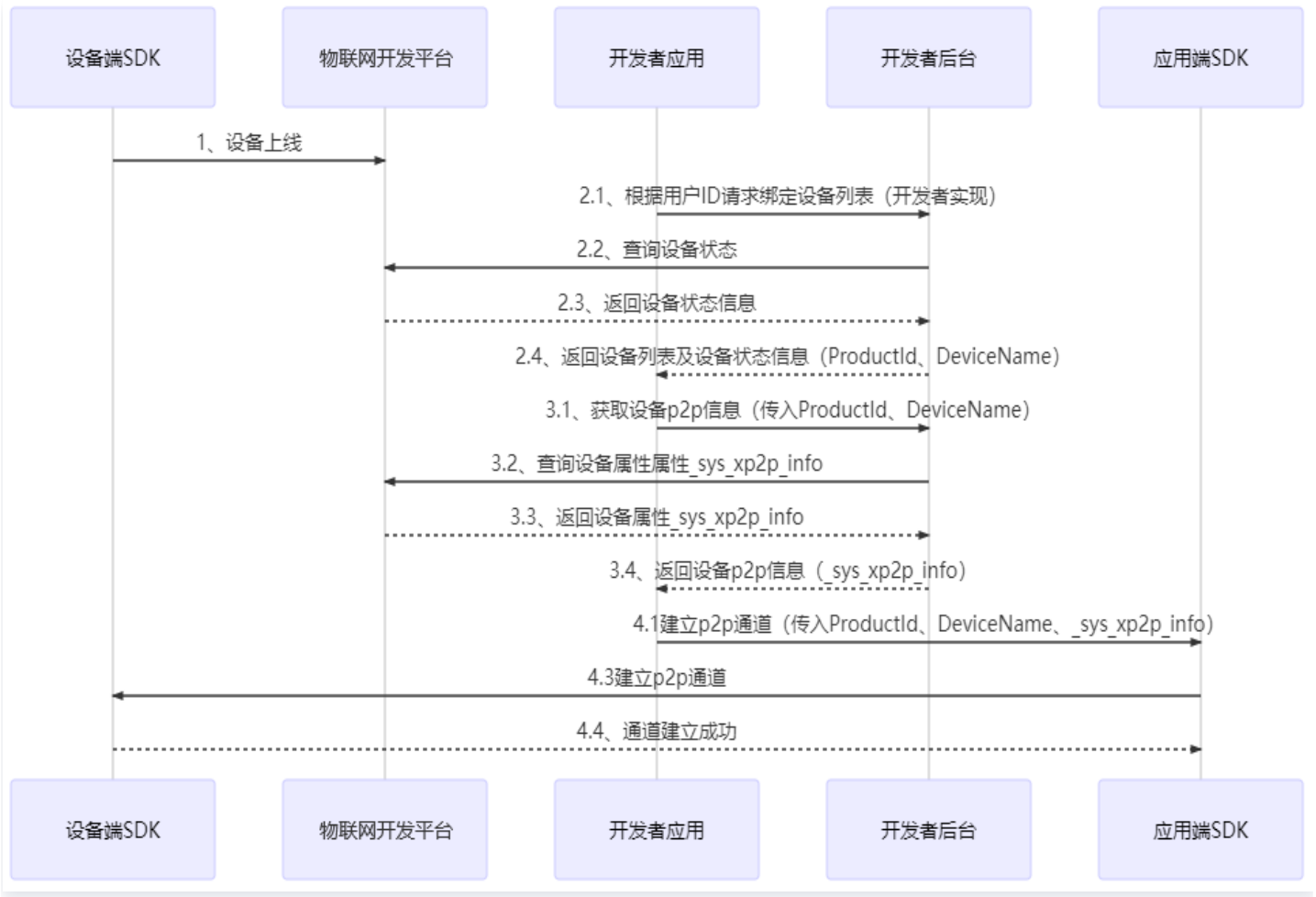
功能概述

- 目前一个设备最多支持建立4路 P2P 传输通道。
- P2P 通道除了支持音视频传输，还支持用户自定义数据传输。

操作步骤

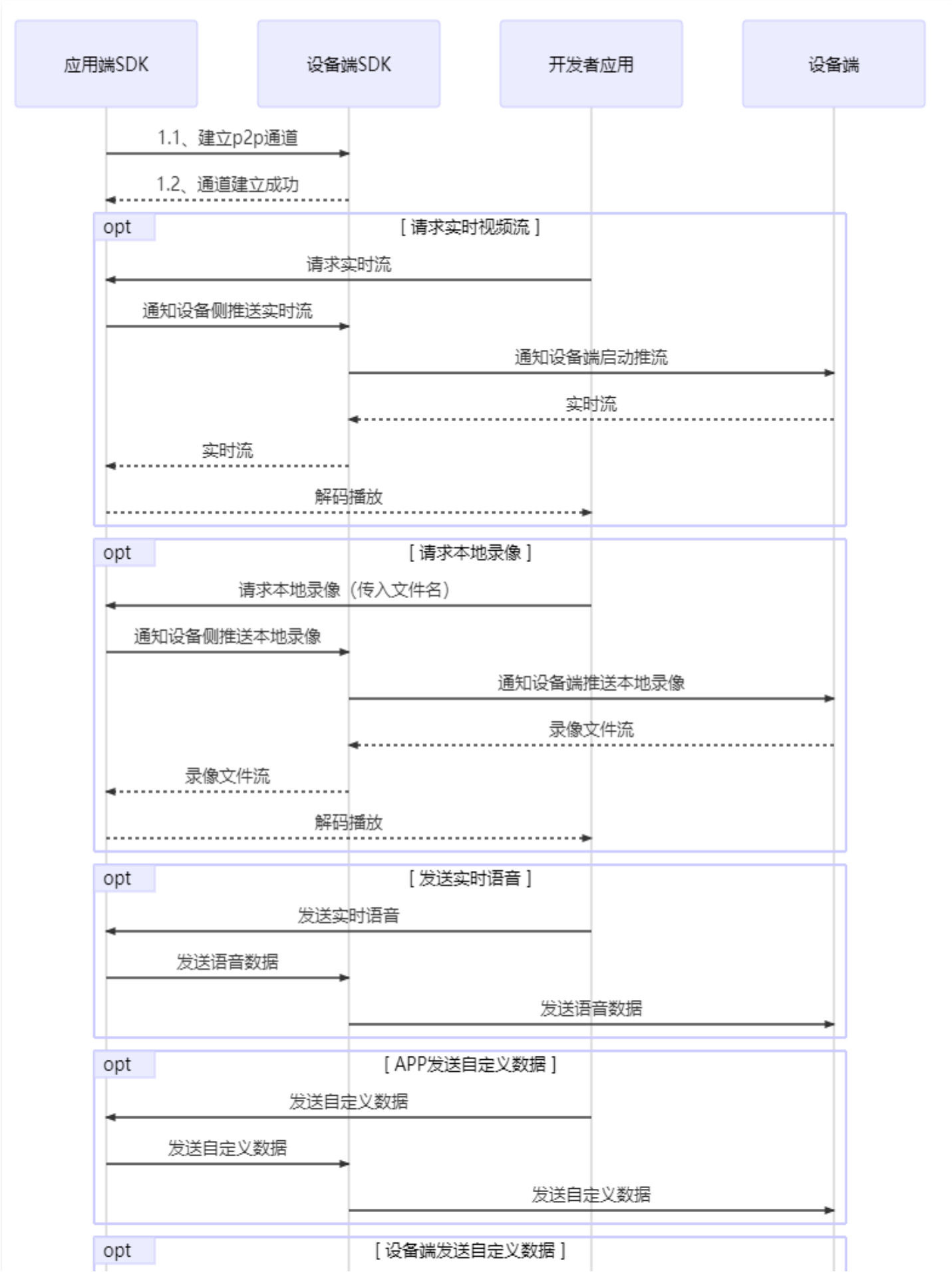
建立 P2P 连接

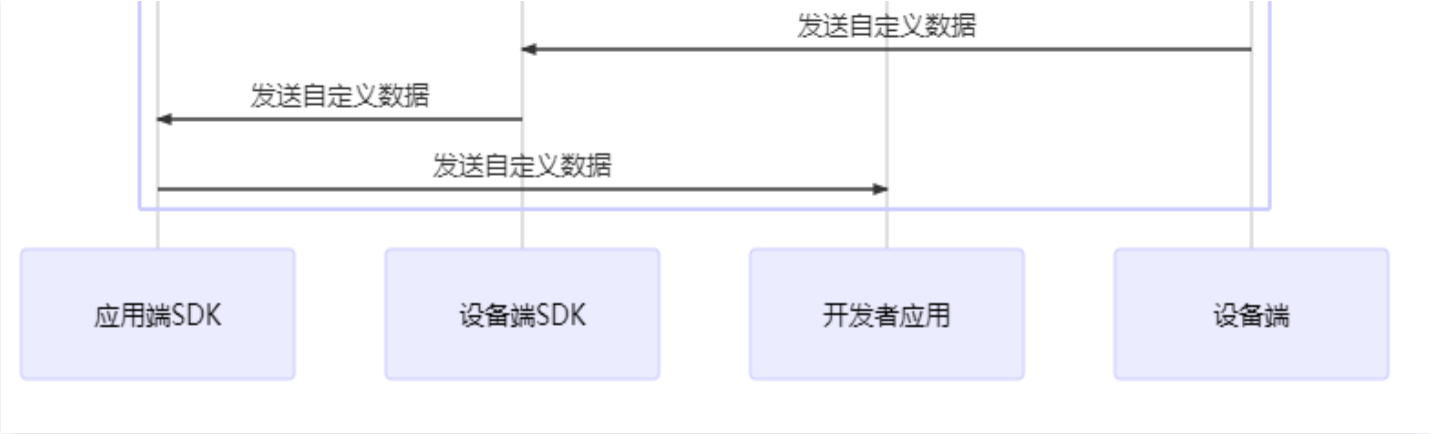
用户需要通过 X-P2P 进行应用端与已绑定的设备端之间的音视频传输，需要先建立 P2P 的连接，用户通过物联网开发平控制台查询设备状态 和 设备属性，获取_sys_xp2p_info。具体流程如下图：



P2P 音视频传输

应用端和设备端已建立 P2P 通道后，可进行音视频和自定义数据传输，流程如下图：


应用端SDK
设备端SDK
开发者应用
设备端
1.1、建立p2p通道
1.2、通道建立成功
opt
[请求实时视频流]
请求实时流
通知设备侧推送实时流
通知设备端启动推流
实时流
实时流
解码播放
opt
[请求本地录像]
请求本地录像（传入文件名）
通知设备侧推送本地录像
通知设备端推送本地录像
录像文件流
录像文件流
解码播放
opt
[发送实时语音]
发送实时语音
发送语音数据
发送语音数据
opt
[APP发送自定义数据]
发送自定义数据
发送自定义数据
发送自定义数据
opt
[设备端发送自定义数据]


发送自定义数据
发送自定义数据
发送自定义数据
应用端SDK
设备端SDK
开发者应用
设备端

云存接入指南

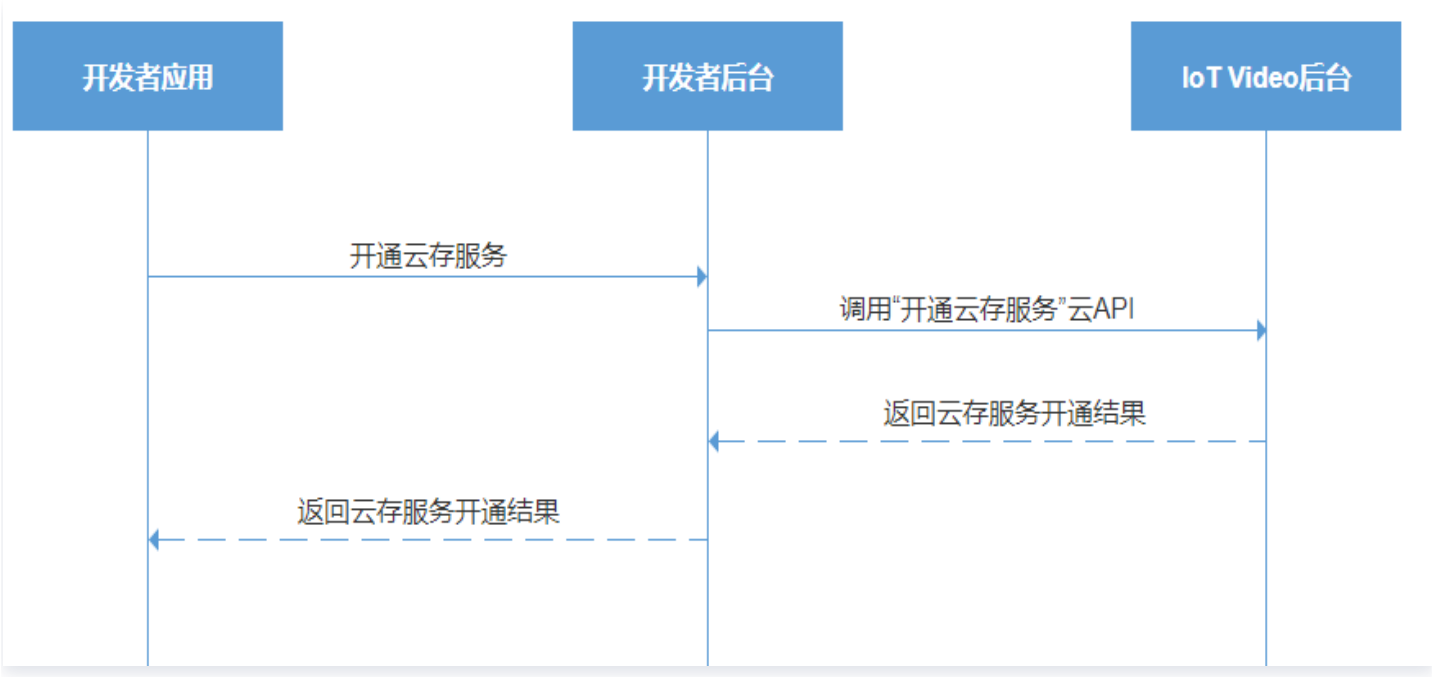
最近更新时间：2024-11-26 18:00:11

物联网开发平台为用户提供了基于云存储功能，支持设备端音视频数据存储在云端，需要观看时，由应用端从云端拉取数据。

操作步骤

开通云存套餐

使用云存储功能前，需要先为设备 开通云存套餐，通过调用云 API 支持为设备购买云存套餐。云存套餐购买成功后，设备将通过物模型获取设备的云存套餐状态。



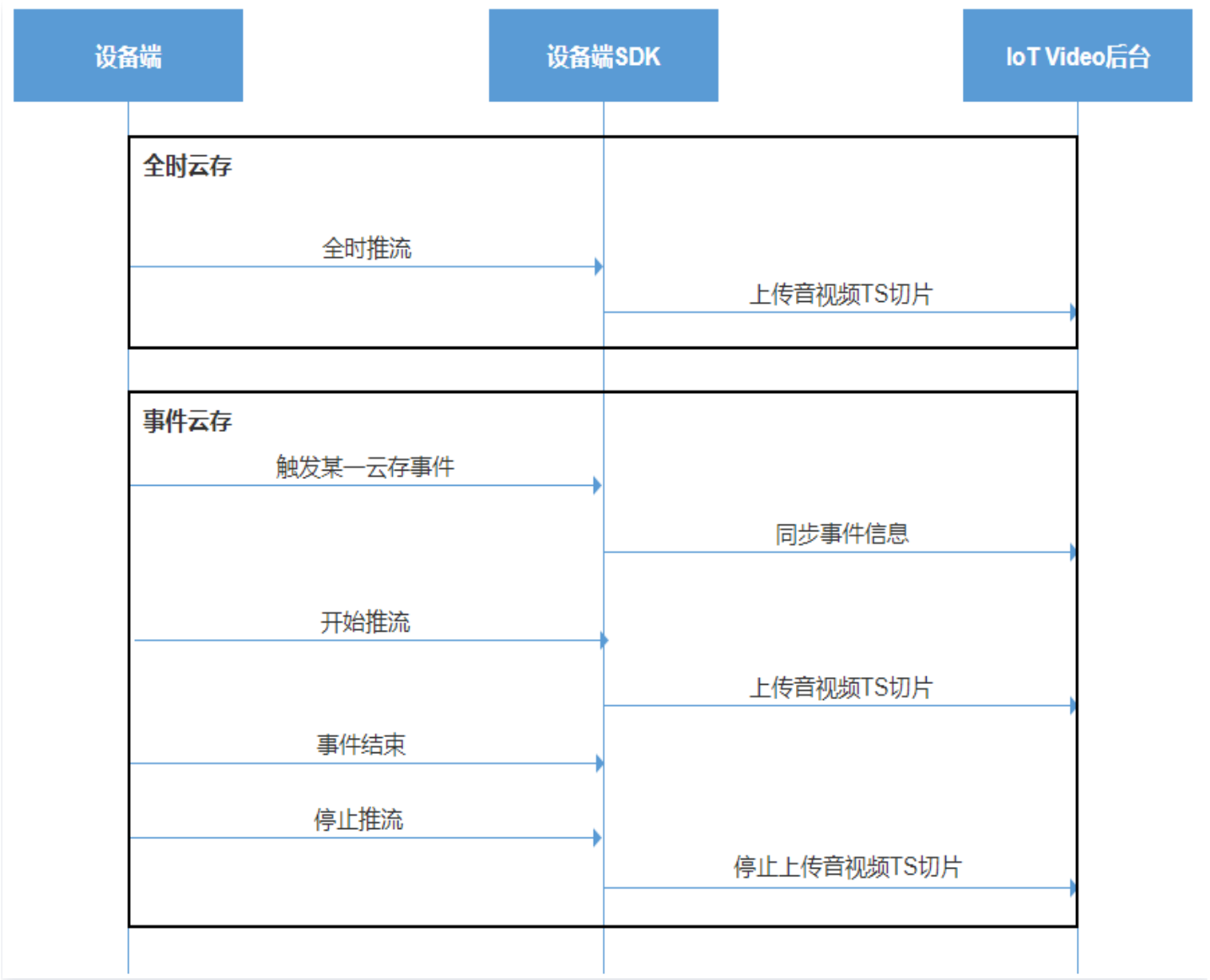
云存录像上传（视频流设备）

云存套餐分为全时云存套餐和事件云存套餐：

- **全时云存**：指设备在套餐生效时间内，可以把所有产生录像上传到云存进行存储。
- **事件云存**：是指设备在套餐生效时间内，当触发事件时，设备可以把事件发生时的录像上传到云端进行存储。

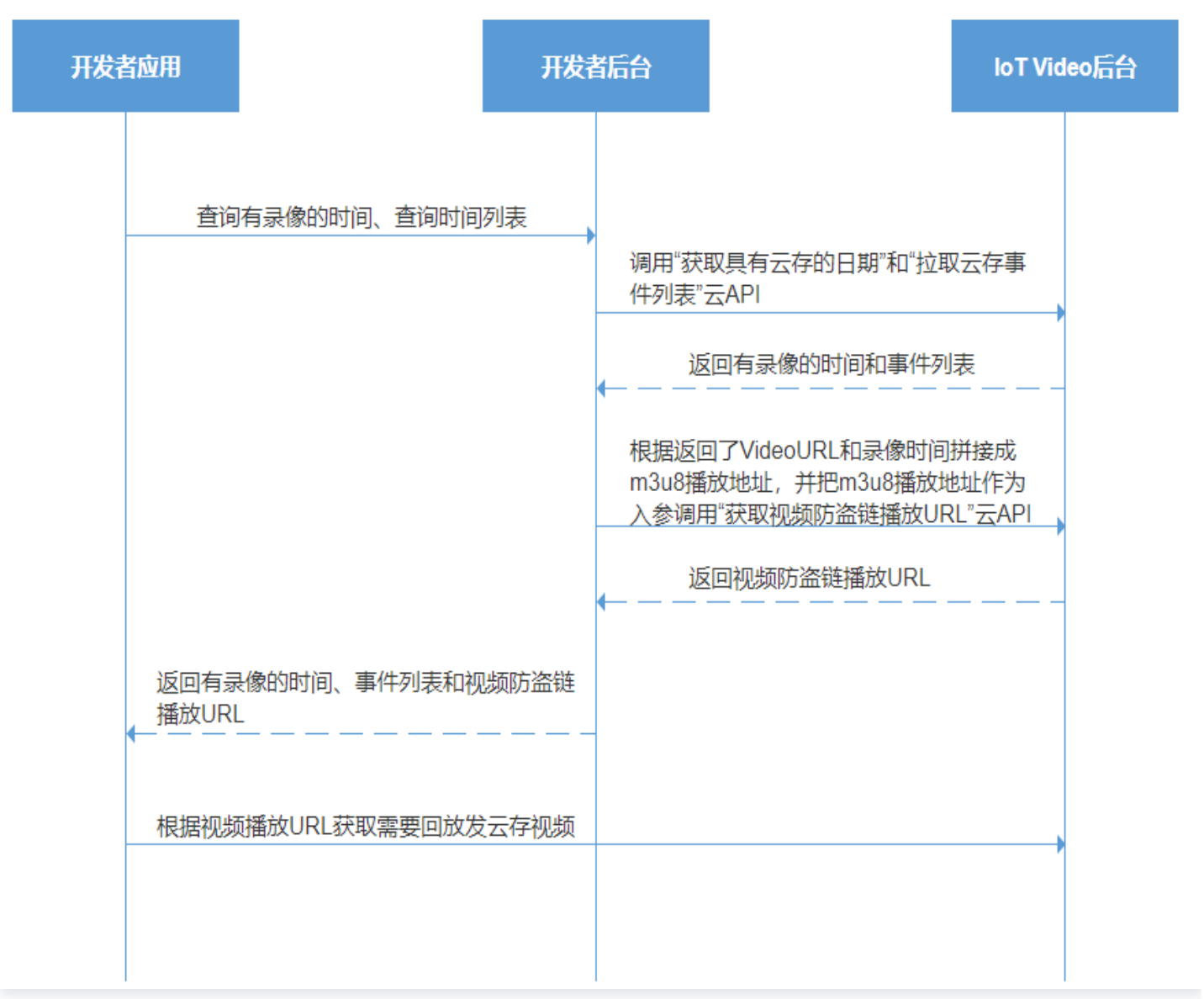
> **说明：**
> 一个设备同时只能存在一个云存套餐。

录像上传到云端存储流程如下：



查看云存录像（视频流设备）

IoT Video（Consumer Version）提供云 API 支持查询有录像的 日期 和 事件列表，并根据返回的日期和 VideoURL 拼接成 m3u8 播放地址，通过 m3u8 地址获取视频流。



m3u8 播放地址拼接

1. 通过调用 获取具有云存的日期 API接口获取云存日期。
2. 调用 获取某一天云存时间轴 API获取指定日期的云存录像播放地址（VideoUrl）和每个录像片段的开始时间（StartTime）、结束时间（EndTime）。
3. 将返回的 VideoUrl、StartTime 和 EndTime 按 UNIX 时间戳以如下格式进行拼接。

```
{VideoUrl}?starttime_epoch={StartTime}&endtime_epoch={EndTime}
```

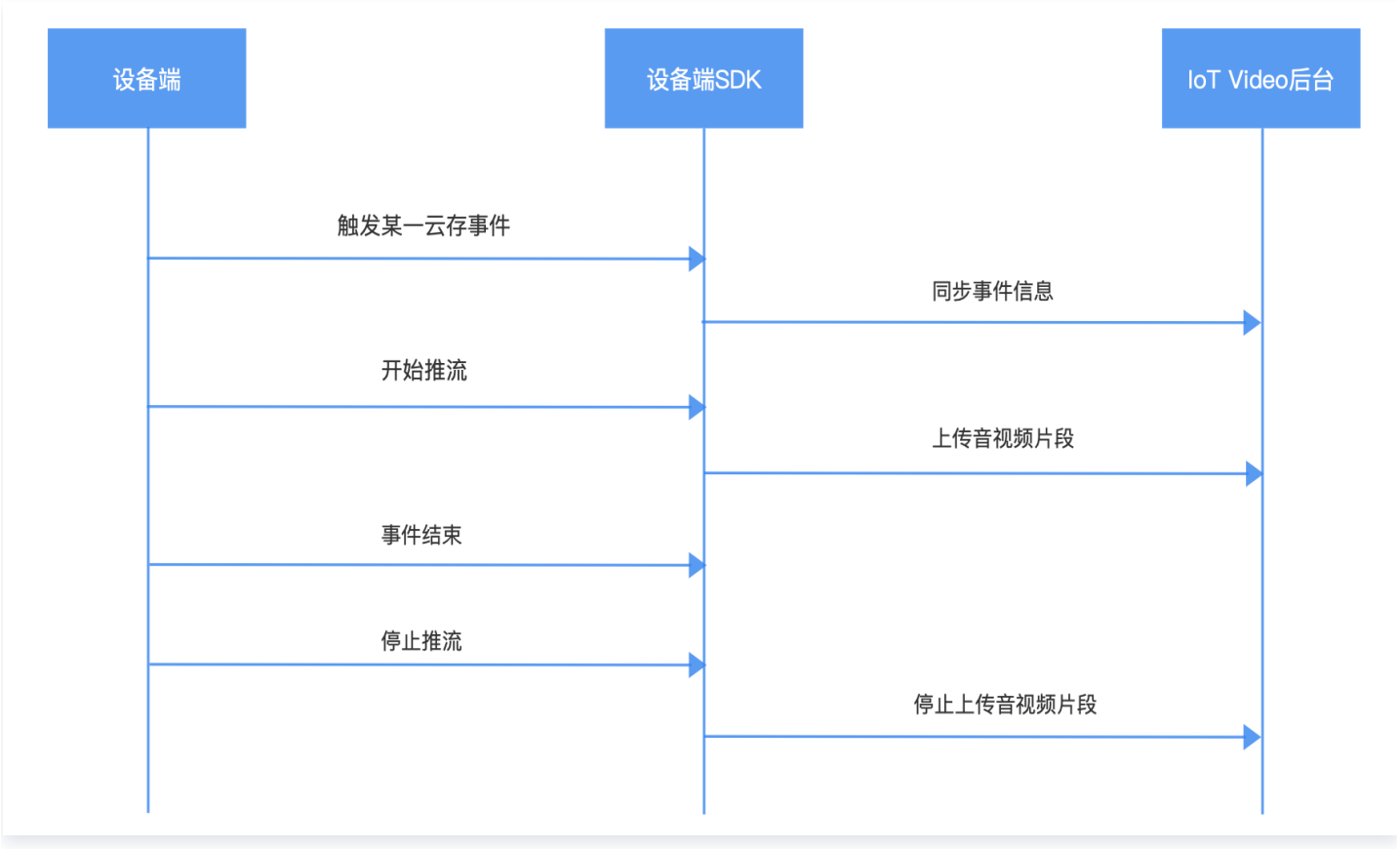
4. 将拼接好的 VideoURL 作为入参调用 获取视频防盗链播放 URL API，即可获得该云存视频的防盗链的播放地址，进而调用小程序插件能力实现正常播放。

云存录像上传（图片流设备）

云存套餐仅有事件云存套餐。

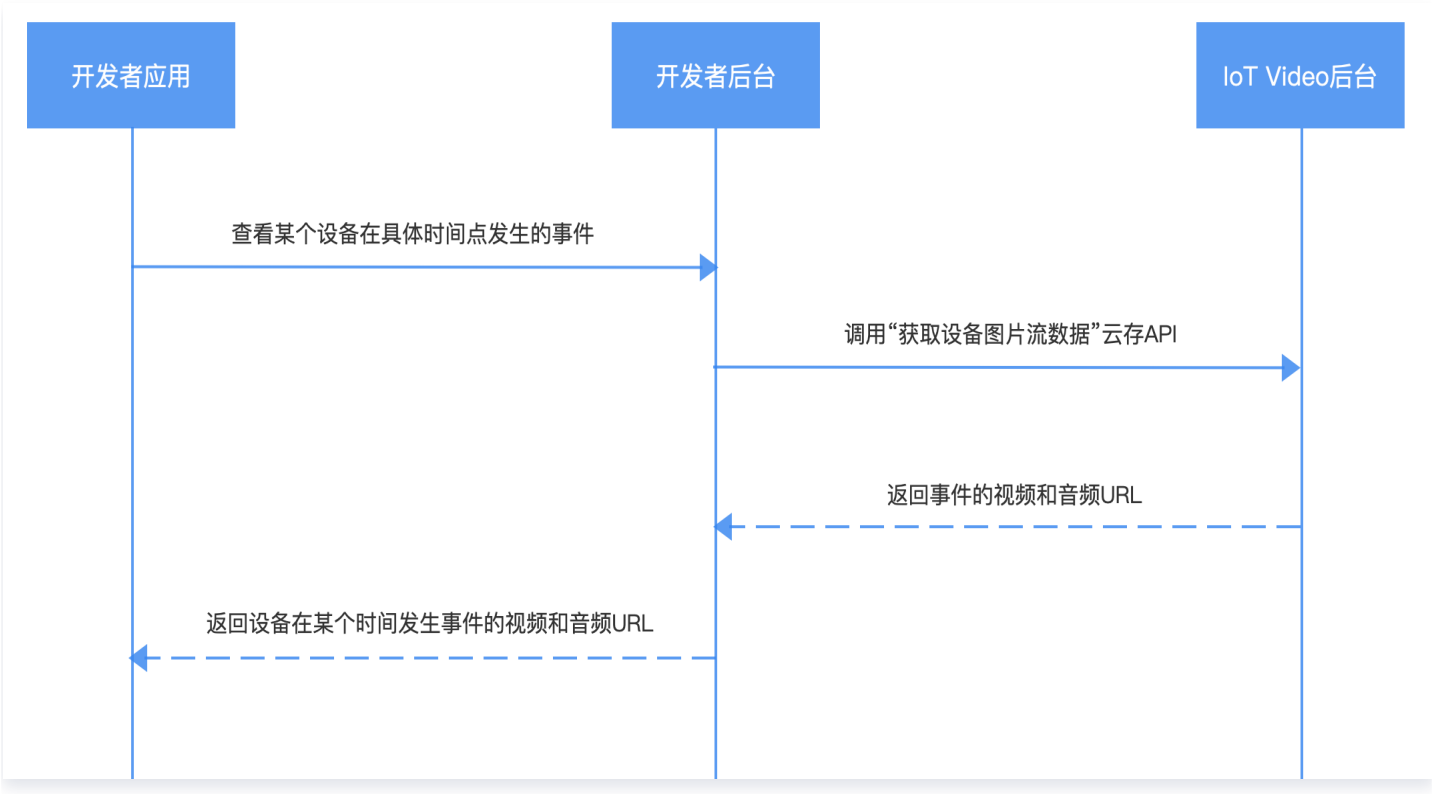
事件云存：是指设备在套餐生效时间内，当触发事件时，设备可以把事件发生时的录像上传到云端进行存储。

> **说明：**
> 一个设备同时只能存在一个云存套餐。



查看云存录像（图片流设备）

平台提供 云 API 支持获取图片流设备的事件云存录像(mjpg 格式的视频 + acc 格式的音频)。



获取图片流播放地址

1. 通过调用 获取具有云存的日期 API 接口获取事件云存的日期。
2. 通过调用 拉取云存事件列表 API 获取某个日期下的事件列表，包括每个事件的 StartTime（开始时间）、EndTime（结束时间）、Thumbnail（缩略图文件名）和 EventId（事件 ID）。
3. 调用 拉取云存事件缩略图 API 获取指定事件缩略图的访问地址。
4. 调用 获取设备图片流数据 API 获取事件的图片流视频、音频播放地址，进而调用小程序插件能力实现正常播放。

> ⚠ **注意：**
> 每次事件记录的存储时间不超过10s。

设备端与应用端信令交互说明

最近更新时间：2024-11-14 15:47:32

概述

底层应用端和设备端 P2P 传输通道建立后，将基于该通道实现 P2P 消息发送和音视频传输功能。下面将详细描述设备端与应用端的信令交互流程，分为音视频传输（设备推音视频流给应用端）、对讲音视频传输（应用端推音视频流给设备端，适用于语音对讲或音视频对讲）、接收信令（应用端发送消息给设备）、发送信令（设备发送信息给应用端）。

> **说明：**
> 未提及参数为内部参数，请勿使用。

音视频传输

请求方法 get
请求路径 / ipc.flv

基本请求参数

_crypto

类型：可选
可见性：外部参数
描述：加密开关，值为 on 或 off，无此参数或不为 off 则默认开启加密。
示例：

```
_crypto=off
```

action

类型：必选
可见性：外部参数
描述：本次拉流请求的类型，有 live 直播，playback 回放，download 下载。
示例：

```
action=live
```

channel

类型：必选

可见性：外部参数
描述：本次拉流请求的通道号，单摄像头为0，多摄像头设备指的是第几个摄像头，nvr 设备指的是第几路数据，均从0开始计数。
示例：

```
channel=0
```

user_args

类型：可选
可见性：外部参数
描述：用户自定义参数，不宜过长，建议控制在1KB 以内，内容和格式不限，但不得出现非法 URL 字符。
示例：

```
user_args=this_is_user_args%E4%BD%A0%E5%A5%BD
```

requester

类型：可选（后续开发建议作为必选参数）
可见性：外部参数
描述：发起拉流请求的角色，不宜过长，建议控制在64字节以内。目前有这几种：app-android，app-ios，wxmp-android，wxmp-ios，server-voip，如有必要可扩展版本号等额外信息，例如：app-android-xiaomi，wxmp-ios-p2p-player-version123。
示例：

```
requester=app-android
```

直播请求

用于观看摄像头实时音视频内容
直播请求参数 action = live

附加请求参数

quality

类型：必选
可见性：外部参数
描述：本次拉流请求的视频质量，有低中高三种，分别是：standard，high，super。
示例：

```
quality=standard
```

请求示例

```
http://ipc.p2p.com/ipc.flv?
action=live&channel=0&quality=high&_token=123456789&_peername=xxxxxxxx&r
equester=app-
android&req_id=xxxxxxxx&user_args=this_is_user_args%E4%BD%A0%E5%A5%BD
```

回放请求

用于回放摄像头 SD 卡内录像
回放请求参数 action=playback

附加请求参数

start_time

类型：必选
可见性：外部参数
描述：请求回放的开始时间，UNIX 秒级时间戳， 与 end_time 间隔建议在5秒以上，间隔过小可能导致索引错误。
示例：

```
start_time=123456789
```

end_time

类型：必选
可见性：外部参数
描述：请求回放的结束时间，UNIX 秒级时间戳， 与start_time间隔建议在5秒以上，间隔过小可能导致索引错误。
示例：

```
end_time=123456789
```

请求示例

```
http://ipc.p2p.com/ipc.flv?
action=playback&channel=0&start_time=1633060850&end_time=1633060880&_tok
```

```
en=123456789&_peername=xxxxxxxx
```

下载请求

用于下载 SD 卡内文件，配合“查询文件列表”信令使用。
回放请求参数 action=download

附加请求参数

file_name

类型：必选
可见性：外部参数
描述：下载的文件名，若包含文件路径请进行URL编码处理。
示例：

```
file_name=test_file.mp4
```

offset

类型：必选
可见性：外部参数
描述：下载文件的起始位置，用户可借助此参数开发断点续传功能。
示例：

```
offset=0
```

请求示例

```
http://ipc.p2p.com/ipc.flv?
action=download&channel=0&file_name=test_file.mp4&offset=0&_crypto=off&_
peername=xxxxxxxx
```

对讲音视频传输

请求方法 post
请求路径 / voice

请求参数

基本请求参数

请参考“音视频传输 – 基本请求参数”中除 action，channel 的其余参数。

附加请求参数

channel

类型：摄像头类设备可选，NVR 类设备必选。
可见性：外部参数
描述：本次对讲请求的通道号，对于摄像头类设备此参数暂未使用，可省略；对于 NVR 类设备指的是第几路，从0开始计数。
示例：

```
channel=0
```

请求示例

```
http://ipc.p2p.com/voice?channel=0&_token=123456789&_peername=xxxxxxxx
```

接收信令

请求方法 post
请求路径 / command
默认开启加密，不可关闭。

请求参数

action

类型：必选
可见性：外部参数
描述： `inner_define` 为内部信令，请勿使用； `user_define` 为外部信令即用户自定义信令。
示例：

```
action=inner_define
action=user_define
```

channel

类型：摄像头类设备可选，NVR 类设备必选。
可见性：外部参数
描述：本次对讲请求的通道号，对于摄像头类设备此参数暂未使用，可省略；对于 NVR 类设备指的是第几路，从0开始计数。
示例：

```
channel=0
```

cmd

类型：必选
可见性：外部参数
描述：消息类型
示例：

```
cmd=my_message
```

内部信令

本地录像回放相关信令

按起止时间查询录像

请求参数：

键	值	描述
cmd	get_record_index	信令类型
start_time	UNIX 时间戳	查询录像的开始时间，单位为秒，与 end_time 间隔建议在5秒以上，间隔过小可能导致索引错误
end_time	UNIX 时间戳	查询录像的结束时间，单位为秒，与 start_time 间隔建议在5秒以上，间隔过小可能导致索引错误
type	枚举	表示录像的类型，是一个整数，由用户自己定义具体含义，目前暂不支持

返回值：
json 格式的录像索引列表，基本格式如下：

```
{
    "video_list": [
        {
            "type": 0,
            "start_time": "<unix时间戳>",
            "end_time": "<unix时间戳>"
        },
        {
```

```
            "type": 0,
            "start_time": "<unix时间戳>",
            "end_time": "<unix时间戳>"
        }
    ]
}
```

请求示例：

```
http://ipc.p2p.com/command?
_token=123456789&_peername=xxxxxxxx&action=inner_define&cmd=get_record_i
ndex&start_time=000&end_time=111
```

按月查询录像

请求参数：

键	值	描述
cmd	get_month_record	信令类型
time	时间	6位整数，前4位表示年，后两位表示月

返回值：
json 格式，其中 `video_list` 的值转换为一个32位的整数，从低位到高位每一比特代表月份的第几天是否有录像；例如：8320（0010000010000000）表示8号和14号有录像。

```
{"video_list": "123456"}
```

请求示例：

```
http://ipc.p2p.com/command?
_token=123456789&_peername=xxxxxxxx&action=inner_define&cmd=get_month_re
cord&time=202401
```

回放暂停

请求参数：

键	值	描述
cmd	playback_pause	信令类型

返回值：

json格式， `<code>` 为信令执行结果返回值，0正常，其他异常。

```
{"status":"<code>"}
```

请求示例：

```
http://ipc.p2p.com/command?
_token=123456789&_peername=xxxxxxxx&action=inner_define&cmd=playback_see
k&progress=123456
```

回放继续

请求参数：

键	值	描述
cmd	playback_resume	信令类型

返回值：

json 格式， `<code>` 为信令执行结果返回值，0正常，其他异常。

```
{"status":"<code>"}
```

请求示例：

```
http://ipc.p2p.com/command?
_token=123456789&_peername=xxxxxxxx&action=inner_define&cmd=playback_res
ume
```

设置播放位置

请求参数：

键	值	描述
cmd	playback_seek	信令类型
progress	整数	毫秒单位的时间戳，表示在当前文件内希望跳转播放的位置

返回值：

json 格式，`<code>` 为信令执行结果返回值，0正常，其他异常。

```
{"status":"<code>"}
```

请求示例：

```
http://ipc.p2p.com/command?
_token=123456789&_peername=xxxxxxxx&action=inner_define&cmd=playback_see
k&progress=123456
```

获取播放位置

请求参数：

键	值	描述
cmd	playback_progress	信令类型

返回值：

json 格式，`<code>` 为信令执行结果返回值，0正常，其他异常；`<timestamp>` 当前播放进度，单位毫秒。

```
{
    "status":"<code>",
    "progress":"<timestamp>"
}
```

请求示例：

```
http://ipc.p2p.com/command?
_token=123456789&_peername=xxxxxxxx&action=inner_define&cmd=playback_pro
gress
```

设置快进

请求参数：

键	值	描述
cmd	playback_ff	信令类型
value	整数	• 0表示正常播放

		• 1表示只播放I帧 • 2表示从相邻的两个I帧中取一个I帧播放 • 3表示从相邻的三个I帧中取一个I帧播放 • 以此类推

返回值：

json 格式， `<code>` **为信令执行结果返回值，0正常，其他异常。**

```
{"status":"<code>"}
```

请求示例：

```
http://ipc.p2p.com/command?
_token=123456789&_peername=xxxxxxxx&action=inner_define&cmd=playback_ff&
value=1
```

设置快退

请求参数：

键	值	描述
cmd	playback_rewind	信令类型
start_time	时间戳	单位毫秒，表示倒放的结束时间
end_time	时间戳	单位毫秒，表示倒放的开始时间

返回值：

json 格式， `<code>` **为信令执行结果返回值，0正常，其他异常。**

```
{"status":"<code>"}
```

请求示例：

```
http://ipc.p2p.com/command?
_token=123456789&_peername=xxxxxxxx&action=inner_define&cmd=playback_rew
ind&start_time=100&end_time=200
```

设置倍速

请求参数：

键	值	描述
cmd	playback_speed	信令类型
speed	整数	单位毫秒，表示帧间隔，以视频为准进行同步，假设视频的正常帧率为25fps，即帧间隔是50ms，数值小于正常帧间隔则为快放，例如设置为10则相当于5倍速快放；数值大于正常帧间隔则为慢放，例如设置为100则相当于0.5倍速慢放

返回值：

json 格式， `<code>` **为信令执行结果返回值，0正常，其他异常。**

```
{"status":"<code>"}
```

请求示例：

```
http://ipc.p2p.com/command?
_token=123456789&_peername=xxxxxxxx&action=inner_define&cmd=playback_spe
ed&speed=50
```

TWeCall及语音对讲相关信令

挂断

说明：由 client 端发送给设备端，用于接听以后，表示主动挂断。

请求参数：

键	值	描述
cmd	call_hang_up	信令类型

返回值：

json 格式， `<code>` **为信令执行结果返回值，0正常，其他异常。**

```
{"status":"<code>"}
```

请求示例：

```
http://ipc.p2p.com/command?
_token=123456789&_peername=xxxxxxxx&action=inner_define&cmd=call_hang_up
```

拒接

说明：由 client 端发送给设备端，用于接听以前，表示拒接。
请求参数：

键	值	描述
cmd	call_reject	信令类型

返回值：
json 格式， `<code>` 为信令执行结果返回值，0正常，其他异常。

```
{"status":"<code>"}
```

请求示例：

```
http://ipc.p2p.com/command?
_token=123456789&_peername=xxxxxxxx&action=inner_define&cmd=call_reject
```

取消

说明：由 client 端发送给设备端，用于接听以前，表示主动取消呼叫。
请求参数：

键	值	描述
cmd	call_cancel	信令类型

返回值：
json格式， `<code>` 为信令执行结果返回值，0正常，其他异常。

```
{"status":"<code>"}
```

请求示例：

```
http://ipc.p2p.com/command?
_token=123456789&_peername=xxxxxxxx&action=inner_define&cmd=call_cancel
```

占线

说明：由 client 端发送给设备端，用于接听以后，表示有其他设备正在通话。

请求参数：

键	值	描述
cmd	call_busy	信令类型

返回值：
json 格式，`<code>` 为信令执行结果返回值，0正常，其他异常。

```
{"status":"<code>"}
```

请求示例：

```
http://ipc.p2p.com/command?
_token=123456789&_peername=xxxxxxxx&action=inner_define&cmd=call_busy
```

无应答

说明：由 client 端发送给设备端，用于接听以前，表示呼叫超时无人接听。
请求参数：

键	值	描述
cmd	call_timeout	信令类型

返回值：
json 格式，`<code>` 为信令执行结果返回值，0正常，其他异常。

```
{"status":"<code>"}
```

请求示例：

```
http://ipc.p2p.com/command?
_token=123456789&_peername=xxxxxxxx&action=inner_define&cmd=call_timeout
```

其他信令

查询设备状态

请求参数：

键	值	描述

cmd	get_device_st	信令类型
type	live / voice / playback	必选，询问设备是否接受直播、对讲、回放请求
quality	standard / high / super	type 为 live 时必选，询问设备是否支持低中高图像质量的直播请求

返回值：

json 格式，status 状态码，0接受连接请求，其他值拒绝连接请求； `appConnectNum` **当前已连接到设备的 App 或小程序数；** `maxConnectNum` **设备端支持的最大连接数。**

```
[
    {
        "status":"code",
        "appConnectNum":"2",
        "maxConnectNum": "3"
    }
]
```

请求示例：

```
http://ipc.p2p.com/command?
_token=123456789&_peername=xxxxxxxx&action=inner_define&cmd=get_device_s
t&type=live&quality=standard
```

查询文件列表

请求参数：

键	值	描述
cmd	get_file_list	信令类型

返回值：

json 格式， `file_type` **文件类型；** `file_name` **文件名；** `file_size` **文件大小；** `start_time` **开始时间；** `end_time` **结束时间；** `extra_info` **扩展信息，可用于携带哈希值或其他自定义文件信息。**

```
{
    "file_list": [
        {
            "file_type": "video",
```

```
            "file_name": "1234.mp4",
            "file_size": "1024000",
            "start_time": "2024-01-23_12-34-00",
            "end_time": "2024-01-23_12-34-59",
            "extra_info": {
                "md5": "abvdefg"
            }
        },
        {
            "file_type": "picture",
            "file_name": "abcd.jpg",
            "file_size": "10240",
            "start_time": "2024-01-23_12-34-56",
            "end_time": "2024-01-23_12-34-56"
        }
    ]
}
```

请求示例：

```
http://ipc.p2p.com/command?
_token=123456789&_peername=xxxxxxxx&action=inner_define&cmd=get_file_lis
t
```

查询NVR下设备名称

请求参数：

键	值	描述
cmd	get_nvr_list	信令类型

返回值：

json 格式， `DeviceName` **设备名；** `Channel` **位于几通道；** `Online` **是否在线。**

```
[
    {
        "DeviceName": "name1",
        "Channel": "1",
        "Online": "0"
    },
    {
        "DeviceName": "name2",
```

```
        "Channel": "2",
        "Online": "1"
    }
]
```

请求示例：

```
http://ipc.p2p.com/command?
_token=123456789&_peername=xxxxxxxx&action=inner_define&cmd=get_nvr_list
```

外部信令

即用户自定义信令
请求参数：

键	值	描述
cmd	信令内容	用户自定义信令，不宜过长，建议控制在1KB 以内，内容和格式不限，但不得出现非法 URL 字符

返回值：
格式不限，类型不限，不宜过长，建议控制在1KB 以内。
请求示例：

自定义信令进行云台控制示例：

```
http://ipc.p2p.com/command?
_token=123456789&_peername=xxxxxxxx&action=user_define&cmd=ptz_left_spee
d_1
http://ipc.p2p.com/command?
_token=123456789&_peername=xxxxxxxx&action=user_define&cmd=ptz_right_spe
ed_1
```

自定义信令传输少量二进制数据：

```
http://ipc.p2p.com/command?
_token=123456789&_peername=xxxxxxxx&action=user_define&cmd=36D7F336FCEC9
E57318ECA0323BA94970296A776CE028D19F5CA5AC44E92952B4CE77642354CBAE7036F4
DA954317462C61A1BC033DAC882FAAB9EDCBFEE47DA
```

发送信令

请求方法 post
请求路径 / feedback
默认开启加密，不可关闭。

消息在 HTTP 报文正文中发送，采用 json 格式。

请求参数

键	值	描述
product_id	设备三元组信息	供对方识别设备、消息管理等场景使用
device_name	设备三元组信息	供对方识别设备、消息管理等场景使用

请求示例

```
http://127.0.0.1:34567/ipc.p2p.com/forward/xxxxxxxxxxxx/proxy.sample.server/feedback?_token=123456789&product_id=xxxxxxxx&device_name=xxxxxxxx
```

用户自定义信令

用户自定义消息格式采用 json 格式，不得含有 `iv_private_cmd` 字段，其余内容不限，长度不宜过大，建议控制在16KB 以内。
示例：

```
发送
{
"my_message":"hello world"
}
接收
{
"status":"<code>"
}
```

`<code>` 为返回值，0正常，其他异常。

内部信令

内部消息使用以下 json 格式

```
发送
{"iv_private_cmd":"<cmd>"}
接收
{"status":"<code>"}
```

挂断

说明：由设备端发送给 client 端，用于接听以后，表示主动挂断。

```
{"iv_private_cmd":"call_hang_up"}
```

响应：

```
{"status":"<code>"}
```

`<code>` 为返回值，0正常，其他异常。

取消

说明：由设备端发送给 client 端，用于接听以前，表示主动取消呼叫。

```
{"iv_private_cmd":"call_cancel"}
```

响应：

```
{"status":"<code>"}
```

`<code>` 为返回值，0正常，其他异常。

其他信令

设备端主动停止接收对端发送的音视频数据

说明：由设备端发送给 client 端，用于双向对讲的情景。

```
{"iv_private_cmd":"recv_stop"}
```

响应：

```
{"status":"<code>"}
```

`<code>` 为返回值，0正常，其他异常。

腾讯连连小程序交互信令说明

最近更新时间：2024-11-26 18:00:11

设备端对接腾讯连连小程序 IoT Video 标准面板时使用的信令交互协议。

请求方法

```
/**
 * command 为字符串或对象
 */
function sendCommand(command) {
    message_id++
    if (typeof command !== 'string') command =
'action=user_define&channel=0&cmd=' + JSON.stringify(command)
    return p2pExports.sendCommand(id, command)
}
```

内部信令

参考文档 设备端与应用端信令交互说明_内部信令 。

外部信令

操作云台

请求字段

字段	类型	说明
topic	string	ptz 操作类型。
message_id	number | string	请求 ID。
cmd	string	• ptz_release_pre：松手 。 • ptz_up_press：上 。 • ptz_right_press：右 。 • ptz_down_press：下 。 • ptz_left_press：左。

请求示例

```
{
    "topic": "ptz",
    "message_id": 0,
    "data": {
        "cmd": "ptz_release_pre"
    }
}
```

返回字段

字段	类型	说明
apex	string	是否到顶。 • yes：到顶。 • no：未到顶。
current_x	number	横向位置。
current_y	number	纵向位置。
max_x	number	横向位置最大值。
max_y	number	纵向位置最大值。
min_x	number	横向位置最小值。
min_y	number	纵向位置最小值。

返回示例

```
{
    "confirmation_topic": "ptz",
    "result": "0",
    "apex": "yes",
      "current_x": 123,
      "current_y": 123,
      "max_x": 123,
      "max_y": 123,
      "min_x": 123,
      "min_y": 123,
    "message_id": 0
}
```