ALIBABA CLOUD

阿里云

3.x

文档版本：20210728

阿里云

法律声明

阿里云提醒您在阅读或使用本文档之前仔细阅读、充分理解本法律声明各条款的内容。如果您阅读或使用本文档，您的阅读或使用行为将被视为对本声明全部内容的认可。

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。

2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。

3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。阿里云保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在阿里云授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过阿里云授权渠道下载、获取最新版的用户文档。

4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的“现状”、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。

5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含“阿里云”、“Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。

6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

格式	说明	样例
危险	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	危险 重置操作将丢失用户配置数据。
警告	该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。	警告 重启操作将导致业务中断，恢复业务时间约十分钟。
注意	用于警示信息、补充说明等，是用户必须了解的内容。	注意 权重设置为0，该服务器不会再接受新请求。
说明	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	说明 您也可以通过按Ctrl+A选中全部文件。
>	多级菜单递进。	单击**设置**> **网络**> **设置网络类型**。
粗体	表示按键、菜单、页面名称等UI元素。	在**结果确认**页面，单击**确定**。
Courier字体	命令或代码。	执行 `cd /d C:/window` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *Instance_ID*
[] 或者 [a|b]	表示可选项，至多选择一个。	`ipconfig [-all|-t]`
{} 或者 {a|b}	表示必选项，至多选择一个。	`switch {active|stand}`

目录

1.SDK获取

C语言Link SDK适用于使用C语言开发业务处理逻辑的设备，由于C语言运行速度快、需要的运行内存较少，目前大多数的IoT设备使用C语言进行产品开发。

重要通知：生活物联网平台推出了专有的设备端SDK，针对生活场景增加了一些新的功能，若产品接入物联网平台请参照生活物联网平台的开发文档进行SDK获取以及产品开发。

SDK获取

SDK 3.X

当前最新版本：v3.2.0

> 说明　使用v3.0.1和v3.1.0开发产品的客户，如果用不到新增功能可不用升级SDK。

点击历史版本清单了解细节。

注：

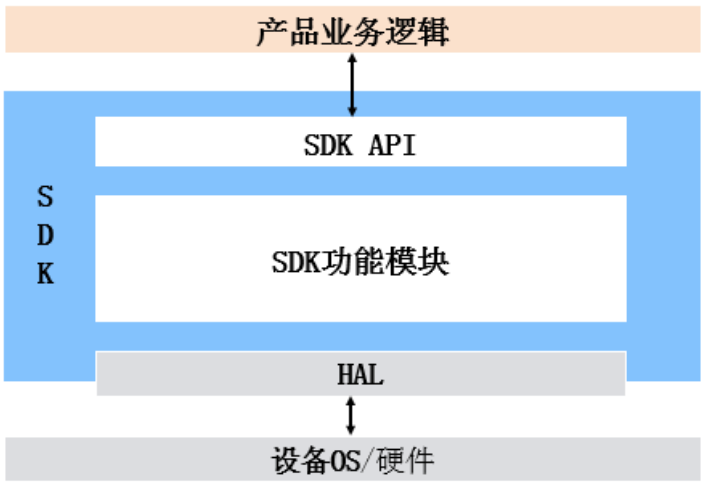
- SDK 2.3.0版本SDK的开发文档可以通过此处访问
- 用户如果正在使用SDK2.3.0开发产品，并且想升级到SDK3.0.1，可以查看版本变更记录

SDK使用说明

SDK提供了API供设备厂商调用，用于实现与阿里云IoT平台通信以及一些其它的辅助功能，例如WiFi配网、本地控制等。

另外，C语言版本的SDK被设计为可以在不同的操作系统上运行，例如Linux、FreeRTOS、Windows，因此SDK需要OS或者硬件支持的操作被定义为一些HAL函数，设备厂商在使用SDK开发产品时需要将这些HAL函数进行实现。

产品的业务逻辑、SDK、HAL的关系如下图所示。



其中产品业务逻辑和HAL需要设备厂商实现，SDK的目录wrappers\os下提供了针对Linux、FreeRTOS的部分HAL参考实现供参考。

对于初次接触阿里云IoT的用户，请单击《快速体验》了解如何在Ubuntu上将一个模拟设备接入阿里云IoT，从而理解一些基本的概念。

设备接入引导

SDK裁剪

- 如果您的产品基于嵌入式Linux进行开发，请单击《基于Make的编译说明》和《交叉编译示例》了解如何进行SDK裁剪和开发
- 如果您的产品基于KEIL、IAR等开发工具进行开发，请单击《基于代码抽取的移植说明》了解如何进行SDK裁剪、移植

MCU上集成SDK

如果您的产品使用MCU外接一个WiFi模组、2/3/4G、NB-IoT移动通信模组与互联网进行通信，并且您的产品业务需要在MCU上实现：

- 如果外接模组支持MQTT协议，请参见《MCU+支持MQTT的模组》了解移植过程
- 如果外接模组不支持MQTT、但是支持TCP协议，请参见《MCU+支持TCP的模组》了解移植过程

模组/SOC上集成SDK

如果您是模组商，或者是在一个支持TCP/IP的SOC上集成SDK，那么：

如果模组/SOC支持MQTT，请参见《在支持MQTT的模组上集成SDK》如果模组/SOC不支持MQTT但是支持TCP，请参见《在支持TCP的模组上集成SDK》

SDK功能列表

下面的表格列出了目前最新版本C Link SDK的功能：

功能模块	功能点
设备连云	• MQTT连云，设备可通过MQTT与阿里云IoT物联网平台通信 • CoAP连云，设备可通过CoAP与阿里云IoT物联网平台通信，用于设备主动上报信息的场景 • HTTPS连云，设备可通过HTTPS与阿里云IoT物联网平台通信，用于设备主动上报信息的场景
设备身份认证	• 一机一密 • 一型一密
物模型	使用属性、服务、事件对设备进行描述以及实现，包括： • 属性上报、设置 • 服务调用 • 事件上报
云端region设置	• 厂商指定region，告知设备连接到阿里云的具体的云端站点，例如中国上海、新加坡、美国、法国 • 动态连云，设备自动连接到距离设备延时最短的云端站点
OTA	设备固件升级
远程配置	设备配置文件获取

功能模块	功能点
子设备管理	用于让网关设备添加、删除子设备，以及对子设备进行控制
WiFi配网	将WiFi热点的SSID/密码传输给WiFi设备，包括： • 一键配网 • 手机热点配网 • 设备热点配网 • 零配
设备本地控制	局域网内，通过CoAP协议对设备进行控制，包括：ALCS Server，被控端实现ALCS Client，控制端实现，通常被希望通过本地控制设备的网关使用
设备绑定支持	设备绑定token维护，设备通过WiFI、以太网接入，并且通过阿里云开放智能生活平台管理时使用
设备影子	在云端存储设备指定信息供APP查询，避免总是从设备获取信息引入的延时
Reset支持	当设备执行Factory Reset时，通知云端清除记录。例如清除设备与用户的绑定关系，清除网关与子设备的关联关系等。
时间获取	从阿里云物联网平台获取当前最新的时间
文件上传	通过HTTP2上传文件

说明 注：表格中并没有对每种功能给出详细描述，在相应功能的开发章节中会对每个功能进行详细描述。

历史版本清单

版本号	发布日期	下载链接	更新内容
3.2.0	2020/03/03	sdk下载	• OTA支持软件模块升级 • TLS支持SNI • 提供设备影子的API
3.1.0	2019/10/18	sdk下载	• 对关键过程增加状态码机制 • WiFi配网增强，解决AP Isolation、移除路由器配网方案、新增Linux平台的配网HAL函数参考代码 • 增加对X.509的支持 • OTA新增断点续传 • 子设备管理新增子设备/网关多对多拓扑关系 • 系统topic订阅优化，缩短订阅时间

版本号	发布日期	下载链接	更新内容
3.0.1	2019/03/15	sdk下载	• 新增对异步/通知式的底层TCP/IP协议栈的支持 • 支持将选中功能对应的代码抽取出来，方便快速将SDK放入客户的编译环境进行编译 • 代码抽取时自动生成客户HAL适配文件 • 新增http2文件上传功能 • 配网增加设备热度配网方案
2.3.0	2018/11/19	sdk下载	• 新增一套物模型编程接口：风格为 IOT_Linkkit_XXX() ，旧版接口 linkkit_xxx() 仍然保留 • 新增图形化配置方式支持 • WiFi配网的四种模式可以分离
2.2.1	2018/09/03	sdk下载	• 新增一型一密/动态注册功能 • 新增OTA使用iTLS下载固件功能 • WiFi配网功能开源发布 • 本地控制功能开源发布
2.2.0	2018/08/06	sdk下载	• 离线reset支持 • 简化版TSL支持 • 设备禁用/使能支持 • TSL数组支持object类型 • MQTT海外多站点连接支持 • itls支持
2.1.0	2018/03/20	sdk下载	• 新增cmake编译系统 • 增加物模型支持
2.0.3	2018/01/31	sdk下载	• 子设备管理支持 • 优化tls，修复内存泄露 • 升级MQTT通道，支持更长的topic、支持更多的订阅请求、MQTT支持多线程
2.0.2	2017/11/30	sdk下载	• 新增mbedtls支持，目前适配Linux和Windows系统 • 优化HTTP接口，支持发送报文时不断开TLS连接 • 新增OpenSSL适配 • 支持用mingw32编译Win7的库和例程 • make reconfig可弹出和选择已适配平台

版本号	发布日期	下载链接	更新内容
2.0.1	2017/10/10	sdk下载	• 新增基于CoAP通知方式的OTA • 增加HTTP+TLS的云端连接通道 • 细化OTA状态 • 修正SDK在armcc编译器编译时出现的报错
2.0.0	2017/08/21	sdk下载	• 新增MQTT直连 • 新增CoAP通道 • 增加OTA功能 • 升级构建系统

2.快速体验

本章描述如何在Ubuntu上通过MQTT topic和通过物模型的编程方式，上报和接收业务报文。这个环节使用Ubuntu主机模拟IoT设备，让用户体验设备如何与阿里云物联网平台连接和交互，基于Link SDK3.0.1进行编写。

安装本地开发环境

通过Ubuntu系统进行开发

1. 安装Ubuntu16.04

   本文编写是对照的编译环境是64位主机上的 `Ubuntu16.04` ，在其它Linux版本上尚未验证过，推荐安装与阿里一致的发行版以避免碰到兼容性方面的问题。

   然后安装64位的Desktop版本的 `Ubuntu 16.04.x LTS` ，下载地址：http://releases.ubuntu.com/16.04。

   如果您使用 `Windows` 操作系统，可以安装虚拟机软件 `Virtualbox` 获得Linux开发环境，下载地址：https://www.virtualbox.org/wiki/Downloads。

2. 安装必备软件

   本SDK的开发编译环境使用如下软件： `make-4.1` ， `git-2.7.4` ， `gcc-5.4.0` ， `gcov-5.4.0` ， `lcov-1.12` ， `bash-4.3.48` ， `tar-1.28` ， `mingw-5.3.1`

   可使用如下命令行安装必要的软件：

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

以MQTT Topic编程方式接入设备

1. 创建产品和设备

   请登录阿里云IoT物联网平台进行产品创建，登录时通过您的阿里云账号进行登录。因为是直接通过MQTT的Topic进行产品功能实现，所以在创建产品时选择“基础版”即可。

   创建产品之后可以添加一个具体的设备，阿里云IoT物联网平台会为设备生成身份信息。

   如果您对云端如何创建产品不熟悉，请单击此处了解如何在阿里云IoT物联网平台进行产品和设备创建。

2. 产品功能实现

   i. 了解SDK根目录结构

   获取Linkkit SDK后，顶层目录结构如下：

   ```
   $ ls
   certs config.bat external_libs extract.bat extract.sh LICENSE makefile make.settings model.json
   README.md src tools wrappers
   ```

   ii. 配置SDK

   SDK默认配置打开了物模型选项，这里仅演示基础版的使用，先关闭物模型选项。

   ```
   $ make menuconfig
   ```

iii. 填写设备三元组到例程中

设备开发者需要实现SDK定义的相应HAL函数获取设备的身份信息，由于本文使用Ubuntu来模拟IoT设备，在SDK版本v3.0.1中，打开文件 wrappers/os/ubuntu/HAL_OS_linux.c ，在v3.1.0/v3.2.0中，打开文件 src/mqtt/examples/mqtt_example.c ，编辑如下代码片段，填入之前在物联网平台创建产品和设备后得到的设备身份信息：

- ProductKey：产品唯一标识
- ProductSecret：产品密钥
- DeviceName：设备唯一标识
- DeviceSecret：设备密钥

```
#ifdef DYNAMIC_REGISTER
...
...
#else
#ifdef DEVICE_MODEL_ENABLED
...
...
#else
char _product_key[IOTX_PRODUCT_KEY_LEN + 1]     = "xxxx"; /*使用实际的product key替换*/
char _product_secret[IOTX_PRODUCT_SECRET_LEN + 1] = "yyyy"; /*使用实际的product secret替换*
/
char _device_name[IOTX_DEVICE_NAME_LEN + 1]     = "zzzz"; /*使用实际的device name 替换*/
char _device_secret[IOTX_DEVICE_SECRET_LEN + 1]  = "ssss"; /*使用device secret替换*/
#endif
#endif
```

> **说明** 请在物联网平台的管理控制台将topic /${productKey}/${deviceName}/get 设置为“可订阅可发布”权限，下面的代码中将会用到。

iv. 初始化与建立连接

下面的代码片段来自MQTT上云功能的例程 `src/mqtt/examples/mqtt_example.c` ，它简单描述了设备的初始化以及连接过程。

定制化MQTT参数。

```
iotx_mqtt_param_t     mqtt_params;
memset(&mqtt_params, 0x0, sizeof(mqtt_params));
/* mqtt_params.request_timeout_ms = 2000; */
/* mqtt_params.clean_session = 0; */
/* mqtt_params.keepalive_interval_ms = 60000; */
/* mqtt_params.write_buf_size = 1024; */
/* mqtt_params.read_buf_size = 1024; */
mqtt_params.handle_event.h_fp = example_event_handle;
```

> **说明** 上面的代码中注释掉的地方是mqtt相关配置的默认数值，用户可以不用赋值，SDK会自动填写默认值。如果用户希望调整默认的连接参数，只需要去掉相应的注释，并填入数值即可。

尝试建立与服务器的MQTT连接。

```
pclient = IOT_MQTT_Construct(&mqtt_params);
if (NULL == pclient) {
  EXAMPLE_TRACE("MQTT construct failed");
  return -1;
}
```

> **说明** 将连接参数结构体传参给 `IOT_MQTT_Construct()` 接口，即可触发MQTT连接建立的动作成功返回非空值作为已建立连接的句柄，失败则返回空。

v. 上报数据到云端

在示例文件中定义了如下的topic。

```
/${productKey}/${deviceName}/get
```

下面的代码片段示例了如何向这个Topic发送数据。

```
int example_publish(void *handle)
{
    int         res = 0;
    const char    *fmt = "/%s/%s/get";
    char        *topic = NULL;
    int         topic_len = 0;
    char        *payload = "{\"message\":\"hello!\"}";
    topic_len = strlen(fmt) + strlen(DEMO_PRODUCT_KEY) + strlen(DEMO_DEVICE_NAME) + 1;
    topic = HAL_Malloc(topic_len);
    if (topic == NULL) {
        EXAMPLE_TRACE("memory not enough");
        return -1;
    }
    memset(topic, 0, topic_len);
    HAL_Snprintf(topic, topic_len, fmt, DEMO_PRODUCT_KEY, DEMO_DEVICE_NAME);
    res = IOT_MQTT_Publish_Simple(0, topic, IOTX_MQTT_QOS0, payload, strlen(payload));
```

> 说明 其中，IOT_MQTT_Publish_Simple() 的第1个参数可以填入之前调用 IOT_MQTT_Construct() 得到的句柄返回值，也可以直接填入0，代表告诉SDK，使用当前已建立的唯一MQTT连接来发送消息。

vi. 从云端订阅并处理数据

> 说明 示例程序为了尽量简单的演示发布/订阅，代码中对topic /${productKey}/${deviceName}/get 进行了订阅，意味着设备发送给物联网平台的数据将会被物联网平台发送回设备。

下面的代码订阅指定的topic并指定接收到数据时的处理函数。

```
res = example_subscribe(pclient);
if (res < 0) {
    IOT_MQTT_Destroy(&pclient);
    return -1;
}
...
...
int example_subscribe(void *handle)
{
    ...
    res = IOT_MQTT_Subscribe(handle, topic, IOTX_MQTT_QOS0, example_message_arrive, NULL);
    ...
```

> **说明** 其中，IOT_MQTT_Subscribe() 的第1个参数可以填入之前调用 IOT_MQTT_Construct() 得到的句柄返回值，也可以直接填入0，代表告诉SDK，使用当前已建立的唯一MQTT连接来订阅Topic。

示例程序中收到来自云端消息，在回调函数中处理时只是把消息打印出来。

```
void example_message_arrive(void *pcontext, void *pclient, iotx_mqtt_event_msg_pt msg)
{
  iotx_mqtt_topic_info_t    *topic_info = (iotx_mqtt_topic_info_pt) msg->msg;
  switch (msg->event_type) {
    case IOTX_MQTT_EVENT_PUBLISH_RECEIVED:
      /* print topic name and topic message */
      EXAMPLE_TRACE("Message Arrived:");
      EXAMPLE_TRACE("Topic  : %.*s", topic_info->topic_len, topic_info->ptopic);
      EXAMPLE_TRACE("Payload: %.*s", topic_info->payload_len, topic_info->payload);
      EXAMPLE_TRACE("\n");
      break;
    default:
      break;
  }
}
```

示例代码向该Topic周期性的发送数据，用户在实现自己的产品逻辑时不需要周期的发送数据，只是有需要上报的时候再发送数据。

```
while (1) {
  if (0 == loop_cnt % 20) {
    example_publish(pclient);
  }
  IOT_MQTT_Yield(pclient, 200);
  loop_cnt += 1;
}
```

vii. 编译例子程序

在SDK顶层目录运行如下命令：

```
make distclean
make
```

注：每次在根目录执行完make会自动生成代码在output/中，用户如已修改output/中代码需保存，请自行备份。

编译成功完成后，生成的样例程序在当前路径的 output/release/bin 目录下：

```
$ tree output/release
output/release/
+-- bin
...
...
|  +-- mqtt-example
...
...
```

3. 观察数据

执行如下命令：

```
$ ./output/release/bin/mqtt-example
```

可以在物联网平台的控制台，找到指定的产品，在其日志服务中查看设备上报的消息。可以单击此处了解如何在云端查看设备上报的数据。

在Linux的console里面也可以看见示例程序打印的来自云端的数据：

```
example_message_arrive|031 :: Message Arrived:
example_message_arrive|032 :: Topic  : /a1MZxOdcBnO/test_01/get
example_message_arrive|033 :: Payload: {"message":"hello!"}
example_message_arrive|034 ::
```

以物模型编程方式接入设备

1. 创建产品和设备

可以在阿里云IoT物联网平台以及其上承载的多个行业服务中进行产品的创建，下面是在阿里云IoT物联网平台创建产品的帮助链接。

- 如何创建支持物模型的产品
- 如何定义物模型

若产品需要在生活物联网平台（注：基于阿里云IoT物联网平台创建的针对生活场景的行业服务）进行创建，可以登录生活物联网平台创建产品。

本示例产品的物模型描述文件 model_for_examples.JSON 存放在 ./src/dev_model/examples/ 目录下，为了简化用户在物联网平台控制台上的操作，用户可以在控制台创建自己的产品后，将该文件中的 productkey 替换为自己创建产品的 productKey ，然后在 **产品详情** - **功能定义** 页面单击 **导入物模型** 按钮将该JSON文件导入到自己创建的产品中，这样用户的产品将具备示例产品的全部物模型定义。

2. 产品功能实现

   i. 填写设备身份信息到例程中。

   设备的身份信息通过HAL调用返回给SDK，由于本体验基于Linux，因此相关的HAL实现位于 `wrappers/os/ubuntu/HAL_OS_linux.c` ，用户需要把文件中的以下设备身份信息替换成自己创建的设备的身份信息。

```
#ifdef DEVICE_MODEL_ENABLED
char _product_key[IOTX_PRODUCT_KEY_LEN + 1]     = "a1RIsMLz2BJ";
char _product_secret[IOTX_PRODUCT_SECRET_LEN + 1] = "fSAF0hle6xL0oRWd";
char _device_name[IOTX_DEVICE_NAME_LEN + 1]     = "example1";
char _device_secret[IOTX_DEVICE_SECRET_LEN + 1]  = "RDXf67itLqZCwdMCRrw0N5FHbv5D7jrE";
```

   > **说明** 用户也可以不用修改这些全局变量，而是直接修改 `HAL_GetProductKey()` 等函数返回设备身份信息。

   ii. 编译与运行程序。

   在SDK顶层目录执行如下命令。

```
$ make distclean
$ make
```

   编译成功完成后，生成的高级版例子程序在当前路径的 `output/release/bin` 目录下，名为 `linkkit-example-solo` 。

   在SDK顶层目录执行如下命令。

```
$ ./output/release/bin/linkkit-example-solo
```

3. 观察数据

   示例程序会定期将 `Counter` 属性的数值上报云端，因此可以在云端查看收到的属性。用户可以将该属性配置为可读写属性，并且可以在云端对该属性进行设置，然后再次查看从设备端上报的 `Counter` 值。

   - 属性上报

     示例中使用 `__user_post_property__` 作为上报属性的例子。该示例会循环上报各种情况的payload，用户可观察在上报错误payload时返回的提示信息。

     代码中上报属性的代码片段如下。

```
/* Post Proprety Example */
if (time_now_sec % 11 == 0 && user_master_dev_available()) {
    user_post_property();
}
```

     观察属性上报示例函数。

```
void user_post_property(void)
{
  static int example_index = 0;
  int res = 0;
  user_example_ctx_t *user_example_ctx = user_example_get_ctx();
  char *property_payload = "NULL";
  if (example_index == 0) {
```

正常上报属性的情况。

```
void user_post_property(void)
{
  static int cnt = 0;
  int res = 0;
  char property_payload[30] = {0};
  HAL_Snprintf(property_payload, sizeof(property_payload), "{\"Counter\": %d}", cnt++);
  res = IOT_Linkkit_Report(EXAMPLE_MASTER_DEVID, ITM_MSG_POST_PROPERTY,
                           (unsigned char *)property_payload, strlen(property_payload));
  EXAMPLE_TRACE("Post Property Message ID: %d", res);
}
```

下面是上报正常属性时的日志。

```
[inf] dm_msg_request(205): DM Send Message, URI: /sys/a1X2bEnP82z/test_06/thing/event/property/post, Payload: {"id":"2","version":"1.0","params":{"LightSwitch":1},"method":"thing.event.property.post"}
[inf] MQTTPublish(2546): Upstream Topic: '/sys/a1X2bEnP82z/test_06/thing/event/property/post'
```

这里是发送给云端的消息。

```
> {
>   "id": "2",
>   "version": "1.0",
>   "params": {
>     "Counter": 1
>   },
>   "method": "thing.event.property.post"
> }
```

收到的云端应答。

```
< {
<   "code": 200,
<   "data": {
<   },
<   "id": "1",
<   "message": "success",
<   "method": "thing.event.property.post",
<   "version": "1.0"
< }
```

用户回调函数的日志。

```
user_report_reply_event_handler.314: Message Post Reply Received, Devid: 0, Message ID: 2, Code: 2
00, Reply: {}
```

- 属性设置处理

  收到属性set请求时，会进入如下回调函数。

```
static int user_property_set_event_handler(const int devid, const char *request, const int request_l
en)
{
  int res = 0;
  user_example_ctx_t *user_example_ctx = user_example_get_ctx();
  EXAMPLE_TRACE("Property Set Received, Devid: %d, Request: %s", devid, request);
```

将属性设置的执行结果发回云端，更新云端设备属性。

```
  res = IOT_Linkkit_Report(user_example_ctx->master_devid, ITM_MSG_POST_PROPERTY,
                (unsigned char *)request, request_len);
  EXAMPLE_TRACE("Post Property Message ID: %d", res);
  return 0;
}
```

日志中可以看到从服务端下来的属性设置消息。

```
[dbg] iotx_mc_cycle(1774): PUBLISH
[inf] iotx_mc_handle_recv_PUBLISH(1549): Downstream Topic: '/sys/a1csED27mp7/AdvExample1/thi
ng/service/property/set'
[inf] iotx_mc_handle_recv_PUBLISH(1550): Downstream Payload:
```

从云端收到的属性设置报文内容。

```
< {
<   "method": "thing.service.property.set",
<   "id": "161430786",
<   "params": {
<     "LightSwitch": 1
<   },
<   "version": "1.0.0"
< }
```

发送回云端的应答消息。

```
> {
>   "id": "161430786",
>   "code": 200,
>   "data": {
>   }
> }
[inf] dm_client_publish(106): Publish Result: 0
[inf] _iotx_linkkit_event_callback(219): Receive Message Type: 15
[inf] _iotx_linkkit_event_callback(221): Receive Message: {"devid":0,"payload":{"LightSwitch":1}}
[dbg] _iotx_linkkit_event_callback(339): Current Devid: 0
[dbg] _iotx_linkkit_event_callback(340): Current Payload: {"LightSwitch":1}
```

`user_property_set_event_handler()` 示例回调函数中收到属性设置的日志。

```
user_property_set_event_handler.160: Property Set Received, Devid: 0, Request: {"LightSwitch":1}
```

这样，一条从服务端设置属性的命令就到达设备端并执行完毕了。

最后收到的对属性上报的应答。

```
<{
<   "code": 200,
<   "data": {
<   },
<   "id": "2",
<   "message": "success",
<   "method": "thing.event.property.post",
<   "version": "1.0"
<}
[dbg] iotx_mc_handle_recv_PUBLISH(1555):      Packet Ident : 00000000
[dbg] iotx_mc_handle_recv_PUBLISH(1556):      Topic Length : 60
[dbg] iotx_mc_handle_recv_PUBLISH(1560):       Topic Name : /sys/a1csED27mp7/AdvExample1/thing
/event/property/post_reply
[dbg] iotx_mc_handle_recv_PUBLISH(1563):    Payload Len/Room : 104 / 4935
[dbg] iotx_mc_handle_recv_PUBLISH(1564):     Receive Buflen : 5000
[dbg] iotx_mc_handle_recv_PUBLISH(1575): delivering msg ...
[dbg] iotx_mc_deliver_message(1291): topic be matched
[inf] dm_msg_proc_thing_event_post_reply(258): Event Id: property
[dbg] dm_msg_response_parse(167): Current Request Message ID: 2
[dbg] dm_msg_response_parse(168): Current Request Message Code: 200
[dbg] dm_msg_response_parse(169): Current Request Message Data: {}
[dbg] dm_msg_response_parse(174): Current Request Message Desc: success
[dbg] dm_ipc_msg_insert(87): dm msg list size: 0, max size: 50
[dbg] dm_msg_cache_remove(142): Remove Message ID: 2
[inf] _iotx_linkkit_event_callback(219): Receive Message Type: 30
[inf] _iotx_linkkit_event_callback(221): Receive Message: {"id":2,"code":200,"devid":0,"payload":{}}
[dbg] _iotx_linkkit_event_callback(476): Current Id: 2
[dbg] _iotx_linkkit_event_callback(477): Current Code: 200
[dbg] _iotx_linkkit_event_callback(478): Current Devid: 0
user_report_reply_event_handler.300: Message Post Reply Received, Devid: 0, Message ID: 2, Code: 2
00, Reply: {}
```

> **说明** 实际的产品收到属性设置时，应该解析属性并进行相应处理而不是仅仅将数值发送回云端。

- 事件上报

  示例中使用 `IOT_Linkkit_TriggerEvent` 上报属性。该示例会循环上报各种情况的payload，用户可观察在上报错误payload时返回的提示信息。

  正常上报事件的情况。

```
void user_post_event(void)
{
  int res = 0;
  char *event_id = "HardwareError";
  char *event_payload = "{\"ErrorCode\": 0}";
  res = IOT_Linkkit_TriggerEvent(EXAMPLE_MASTER_DEVID, event_id, strlen(event_id),
                    event_payload, strlen(event_payload));
  EXAMPLE_TRACE("Post Event Message ID: %d", res);
}
```

示例程序中 `Error` 事件（Event）是约每10s上报一次，在以上各种情况中循环。其中正常上报的日志如下。

```
[inf] dm_msg_request(218): DM Send Message, URI: /sys/a1csED27mp7/AdvExample1/thing/event/HardwareError/post, Payload: {"id":"1","version":"1.0","params":{"ErrorCode":0},"method":"thing.event.HardwareError.post"}
[dbg] MQTTPublish(319): ALLOC: (136) / [200] @ 0x1195150
[inf] MQTTPublish(378): Upstream Topic: '/sys/a1csED27mp7/AdvExample1/thing/event/HardwareError/post'
[inf] MQTTPublish(379): Upstream Payload:
```

向云端上报的事件消息内容及日志。

```
> {
>   "id": "1",
>   "version": "1.0",
>   "params": {
>     "ErrorCode": 0
>   },
>   "method": "thing.event.HardwareError.post"
> }
[inf] dm_client_publish(106): Publish Result: 0
[dbg] alcs_observe_notify(105): payload:{"id":"1","version":"1.0","params":{"ErrorCode":0},"method":"thing.event.Error.post"}
[inf] dm_server_send(76): Send Observe Notify Result 0
[dbg] dm_msg_cache_insert(79): dmc list size: 0
user_post_event.470: Post Event Message ID: 1
[dbg] iotx_mc_cycle(1774): PUBLISH
[inf] iotx_mc_handle_recv_PUBLISH(1549): Downstream Topic: '/sys/a1csED27mp7/AdvExample1/thing/event/HardwareError/post_reply'
[inf] iotx_mc_handle_recv_PUBLISH(1550): Downstream Payload:
```

从云端收到的应答消息内容及日志。

```
<{
<   "code": 200,
<   "data": {
<   },
<   "id": "1",
<   "message": "success",
<   "method": "thing.event.HardwareError.post",
<   "version": "1.0"
<}
[dbg] iotx_mc_handle_recv_PUBLISH(1555):       Packet Ident : 00000000
[dbg] iotx_mc_handle_recv_PUBLISH(1556):       Topic Length : 57
[dbg] iotx_mc_handle_recv_PUBLISH(1560):        Topic Name : /sys/a1csED27mp7/AdvExample1/thing
/event/Error/post_reply
[dbg] iotx_mc_handle_recv_PUBLISH(1563):    Payload Len/Room : 101 / 4938
[dbg] iotx_mc_handle_recv_PUBLISH(1564):     Receive Buflen : 5000
[dbg] iotx_mc_handle_recv_PUBLISH(1575): delivering msg ...
[dbg] iotx_mc_deliver_message(1291): topic be matched
[inf] dm_msg_proc_thing_event_post_reply(258): Event Id: Error
[dbg] dm_msg_response_parse(167): Current Request Message ID: 1
[dbg] dm_msg_response_parse(168): Current Request Message Code: 200
[dbg] dm_msg_response_parse(169): Current Request Message Data: {}
[dbg] dm_msg_response_parse(174): Current Request Message Desc: success
[dbg] dm_ipc_msg_insert(87): dm msg list size: 0, max size: 50
[dbg] dm_msg_cache_remove(142): Remove Message ID: 1
[inf] _iotx_linkkit_event_callback(219): Receive Message Type: 31
[inf] _iotx_linkkit_event_callback(221): Receive Message: {"id":1,"code":200,"devid":0,"eventid":"Err
or","payload":"success"}
[dbg] _iotx_linkkit_event_callback(513): Current Id: 1
[dbg] _iotx_linkkit_event_callback(514): Current Code: 200
[dbg] _iotx_linkkit_event_callback(515): Current Devid: 0
[dbg] _iotx_linkkit_event_callback(516): Current EventID: Error
[dbg] _iotx_linkkit_event_callback(517): Current Message: success
```

用户回调函数 `user_trigger_event_reply_event_handler()` 中的日志。

```
user_trigger_event_reply_event_handler.310: Trigger Event Reply Received, Devid: 0, Message ID: 1,
Code: 200, EventID: Error, Message: success
```

- 服务调用

  注册服务消息的处理函数。

  ```
  IOT_RegisterCallback(ITE_SERVICE_REQUEST, user_service_request_event_handler);
  ```

  收到服务请求消息时，会进入下面的回调函数。设备端演示了一个简单的加法运算服务，入参为 `NumberA` 和 `NumberB`，出参为 `Result`，例程中使用 `cJSON` 解析属性的值。

```
static int user_service_request_event_handler(const int devid, const char *serviceid, const int servic
eid_len,
                                              const char *request, const int request_len,
                                              char **response, int *response_len)
{
    int add_result = 0;
    cJSON *root = NULL, *item_number_a = NULL, *item_number_b = NULL;
    const char *response_fmt = "{\"Result\": %d}";
    EXAMPLE_TRACE("Service Request Received, Service ID: %.*s, Payload: %s", serviceid_len, serviceid
, request);
    /* Parse Root */
    root = cJSON_Parse(request);
    if (root == NULL || !cJSON_IsObject(root)) {
        EXAMPLE_TRACE("JSON Parse Error");
        return -1;
    }
    if (strlen("Operation_Service") == serviceid_len && memcmp("Operation_Service", serviceid, servic
eid_len) == 0) {
        /* Parse NumberA */
        item_number_a = cJSON_GetObjectItem(root, "NumberA");
        if (item_number_a == NULL || !cJSON_IsNumber(item_number_a)) {
            cJSON_Delete(root);
            return -1;
        }
        EXAMPLE_TRACE("NumberA = %d", item_number_a->valueint);
        /* Parse NumberB */
        item_number_b = cJSON_GetObjectItem(root, "NumberB");
        if (item_number_b == NULL || !cJSON_IsNumber(item_number_b)) {
            cJSON_Delete(root);
            return -1;
        }
        EXAMPLE_TRACE("NumberB = %d", item_number_b->valueint);
        add_result = item_number_a->valueint + item_number_b->valueint;
        /* Send Service Response To Cloud */
        *response_len = strlen(response_fmt) + 10 + 1;
        *response = (char *)HAL_Malloc(*response_len);
        if (*response == NULL) {
            EXAMPLE_TRACE("Memory Not Enough");
            return -1;
        }
        memset(*response, 0, *response_len);
        HAL_Snprintf(*response, *response_len, response_fmt, add_result);
        *response_len = strlen(*response);
    }
    cJSON_Delete(root);
    return 0;
}
```

此时在设备端可以看到如下日志。

收到的云端的服务调用，输入参数为 `NumberA` （值为1），`NumberB` （值为2）。

```
<{
<   "method": "thing.service.Operation_Service",
<   "id": "280532170",
<   "params": {
<     "NumberB": 2,
<     "NumberA": 1
<   },
<   "version": "1.0.0"
<}
```

在回调函数中将 `NumberA` 和 `NumberB` 的值相加后赋值给 `Result` 后，上报到云端。

```
>{
>   "id": "280532170",
>   "code": 200,
>   "data": {
>     "Result": 3
>   }
>}
```

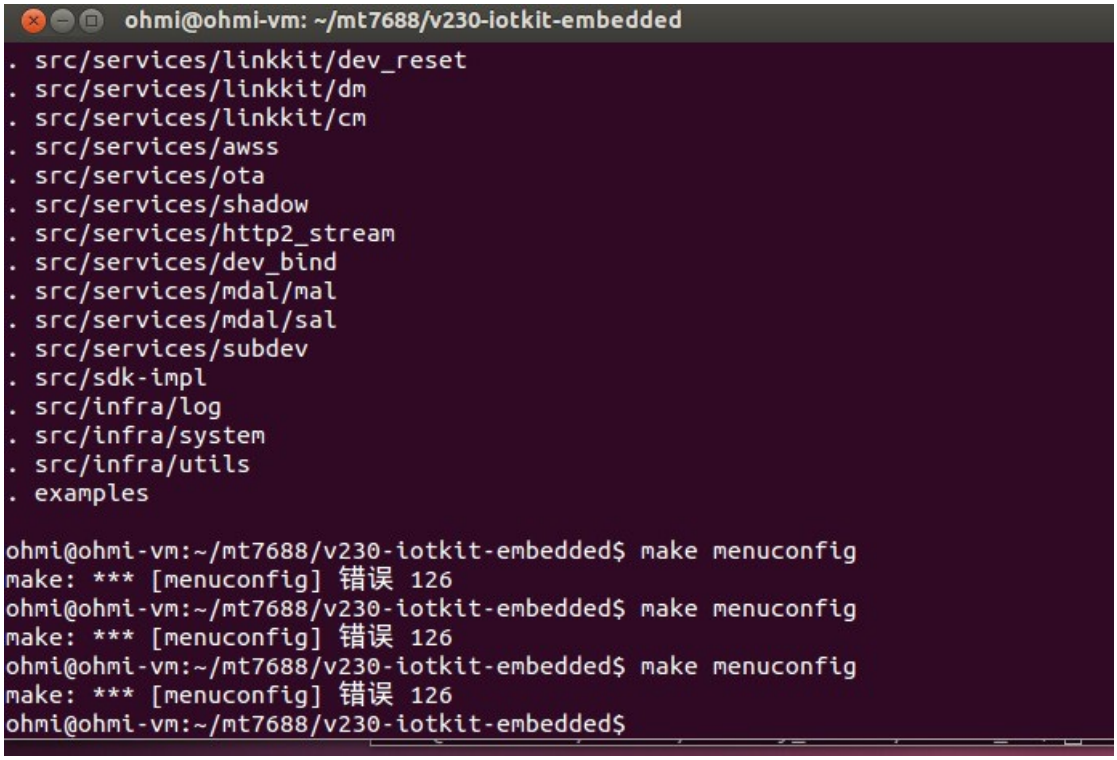
关于高级版单品例程中服务、属性、事件的说明就此结束。

3.常见问题列表

本文的常见问题与Link SDK的C语言版本相关，通用性的常见问题请参见本产品的“常见问题”章节。

make menuconfig提示126错误码

用户使用make menuconfig对SDK进行裁剪时，提示126的错误。



错误原因：Ubuntu的版本过低，导致调用make menuconfig出错

解决办法：将Ubuntu进行升级到16.04及以上

如何编译SDK能够减小二进制尺寸

请注意以下的编译选项在 `CFLAGS` 中能够起到的作用

选项	说明
`-Os`	尺寸优化选项, GNU系列的工具链一般都会支持
`-g3`	调试附加选项, 如果不需要使用 `gdb` 调试, 可以去掉来减小尺寸
`--coverage`	覆盖率统计选项, 如果不需要用 `lcov` 统计代码覆盖率, 可以去掉来减小尺寸
`-ffunction-sections`	将函数分段摆放, 不被使用的函数将不进入最终的二进制, **加上此选项可减小最终的可执行程序/镜像大小**
`-fdata-sections`	将数据分段摆放, 不被使用的变量将不进入最终的二进制, **加上此选项可减小最终的可执行程序/镜像大小**

选项	说明
-Wl,--gc-sections	链接的时候让未使用的符号不进入最终的二进制, 减小尺寸, **本选项需要和上面的2个选项组合使用**

同时如下的功能开关可以考虑关闭, 以减小尺寸

开关	说明
FEATURE_AWSS_SUPPORT_ROUTER	配网中的路由器配网模式, 一般可以直接关闭, 以减小尺寸
FEATURE_AWSS_SUPPORT_PHONEASAP	配网中的手机热点配网模式, 一般不使用这种模式的时候也可以关闭, 以减小尺寸

如何解决嵌入式平台上 strtod() 不工作问题

在有些嵌入式平台上, 由于C库被定制, 标准的C99库函数 strtod() 可能不工作甚至引起崩溃和死机, 可通过 setlocale(LC_ALL, "C"); 的方式使能C库能力全集来解决。

SDK几个库文件的编译时的链接顺序

当编译应用程序时链接SDK的顺序， 请务必保持使用

```
-liot_sdk -liot_hal -liot_tls
```

这样的顺序来链接SDK提供的几个分库, 因为写在后面的都是对前面库的支撑

对于移植到Linux上使用的情况, 还需要以

```
-liot_sdk -liot_hal -liot_tls -lpthread -lrt
```

的方式来书写, 这是因为SDK在Linux下的HAL参考实现使用了pthread库和librt实时库

例如, 使用 dlopen() 接口打开我们的 libiot_sdk.so 这样的情况, 那么在编译SDK和使用它的应用程序的时候, 就需要写成例如

```
$(TARGET): $(OBJS)
  $(CC) -o $@ $^ -liot_sdk -liot_hal -liot_tls -lpthread -lrt
```

的这个样子

不论是链接静态库, 还是链接动态库, 还是使用 dlopen() 等运行时动态加载的方式使用SDK, 应用程序链接的时候都请确保按照上述顺序编写链接指令。

获取动态链接库形态的SDK编译产物

由于C-SDK大部分情况下运行在非Linux的嵌入式操作系统上, 例如 AliOS Things , 或者 FreeRTOS 等。

而这些操作系统并无Linux的动态链接库概念, 所以默认情况下SDK都是以静态库(libiot_sdk.a + libiot_hal.a + libiot_tls.a)的方式输出。

可以用如下的修改方法调整默认的输出形态，将SDK的编译产物从静态库的方式改成动态库。

获取 libiot_sdk.so 代替 libiot_sdk.a 。

以默认的 config.ubuntu.x86 配置文件为例，如下的修改可以告诉构建系统要产生动态库形态的构建产物。

```
--- a/src/board/config.ubuntu.x86
+++ b/src/board/config.ubuntu.x86
@@ -1,6 +1,5 @@
 CONFIG_ENV_CFLAGS  += \
    -Os -Wall \
-   -g3 --coverage \
    -D_PLATFORM_IS_LINUX_ \
    -D__UBUNTU_SDK_DEMO__ \
@@ -19,6 +18,7 @@ CONFIG_ENV_CFLAGS  += \
    -DCONFIG_MQTT_RX_MAXLEN=5000 \
    -DCONFIG_MBEDTLS_DEBUG_LEVEL=0 \
+CONFIG_LIB_EXPORT := dynamic
 ifneq (Darwin,$(strip $(shell uname)))
 CONFIG_ENV_CFLAGS  += -rdynamic
```

- **改动点1：** 确保 CFLAGS 中没有 -g3 --coverage 这样的编译选项。
- **改动点2：** 新增一行 CONFIG_LIB_EXPORT := dynamic 。
- **改动点3：** 重新运行 make reconfig 选择刚才修改到的 config.ubuntu.x86 配置文件，或者被定制的config文件，然后以 make all 而不是 make 的方式来编译。

按照如上改法，make all 之后在 output/release/lib/libiot_sdk.so 就可以获取动态库形态的SDK了，其内容和默认的 libiot_sdk.a 是一致的。

```
$ ls output/release/lib/*.so
output/release/lib/libiot_sdk.so
```

获取 libiot_hal.so 代替 libiot_hal.a 。

修改 wrappers/iot.mk ，新增如下这行。

```
LIBSO_TARGET := libiot_hal.so
```

然后运行：

```
make reconfig
make all
```

之后便可以在 output/release/lib/libiot_hal.so 得到动态库形式的HAL参考实现的分库，其内容和默认的 libiot_hal.a 是一致的。

获取 libiot_tls.so 代替 libiot_tls.a 。

修改 external_libs/mbedtls/iot.mk ，新增如下这行。

```
LIBSO_TARGET := libiot_tls.so
```

然后运行:

```
make reconfig
make all
```

之后便可以在 output/release/lib/libiot_tls.so 得到动态库形式的TLS参考实现的分库，其内容和默认的 libiot_tls.a 是一致的。

TLS/SSL连接错误

-0x7880/-30848/MBEDTLS_ERR_SSL_PEER_CLOSE_NOTIFY

解释

云端把SSL连接断开了: The peer notified us that the connection is going to be closed

可能的原因和解决建议

- 设备端数据连接过于频繁, 触发云端限流, 断开设备
  - 建议关闭设备, 等待一段时间(5分钟以后)再发起连接重试, 观察错误仍会出现
- 有多个设备使用相同的productKey和deviceName与云端建立连接, 导致被云端踢下线
  - 建议检查当前使用的三元组是否可能被他人使用
- 设备端保活出错, 没有及时发送 MQTT ping packet, 或者被发送了没有及时到达云端
  - 建议用抓包等方式确认心跳包有成功发出或者观察有没有收到来自服务端的 MQTT ping response
- 如果一次都不能连接成功, 可以考虑是不是大小端字节序不匹配
  - 目前C-SDK 默认是适配小端设备, 如果需在大端硬件上工作, 请添加全局编译选项 REVERSED

-0x7800/-30720/MBEDTLS_ERR_SSL_PEER_VERIFY_FAILED

解释

认证错误: Verification of our peer failed

可能的原因和解决建议

- 证书错误
  - 如果使用官方C-SDK对接, 证书固化在SDK内部, 不会出现. 若自行对接, 则需检查使用的证书是否和阿里云官方证书匹配
- 日常环境SSL域名校验错误
  - 如果出错时, 连接的是日常环境, 则考虑日常不支持SSL域名校验, 请将 FORCE_SSL_VERIFY 的编译选项定义去掉

-0x7200/-29184/MBEDTLS_ERR_SSL_INVALID_RECORD

解释

收到非法数据: An invalid SSL record was received

可能的原因和解决建议

- TCP/IP协议栈收到的数据包出错, 需要排查协议栈方面问题
- SSL所运行的线程栈被设置的过小, 需调整线程栈大小
- SSL被配置的最大报文长度太小, 当网络报文长度超过该数值时, 则可能出现0x7200错误
  - 可调整 `MBEDTLS_SSL_MAX_CONTENT_LEN` 的值, 重新编译再试
  - `MBEDTLS_SSL_MAX_CONTENT_LEN` 的值, 目前已知最小不能小于 `4096`

-0x2700/-9984/MBEDTLS_ERR_X509_CERT_VERIFY_FAILED

解释

证书错误: `Certificate verification failed, e.g. CRL, CA or signature check failed`

可能的原因和解决建议

- 证书不匹配
  - 若未使用官方C-SDK对接, 请检查证书是否和阿里云官网提供下载的一致
- 时钟不对
  - 确认系统时间是否准确, 系统时间不对(例如默认的1970-01-01), 也会导致证书校验失败

-0x0052/-82/MBEDTLS_ERR_NET_UNKNOWN_HOST

解释

DNS域名解析错误: `Failed to get an IP address for the given hostname`

可能的原因和解决建议

- 大概率是设备端当前网络故障, 无法访问公网
  - 如果是WiFi设备, 检查其和路由器/热点之间的连接状况, 以及上联路由器是否可以正常访问公网
- 需要通过类似 `res_init()` 之类的调用强制C库刷新DNS解析文件
  - 在Linux系统上, 域名解析的系统调用 `getaddrinfo()` 的工作是依赖域名解析文件 `/etc/resolv.conf`
  - 对某些嵌入式Linux, 可能会有 `libc` 库读取过时的 `/etc/resolv.conf` 问题
  - 对于这类系统, 域名解析请求不论是早于还是晚于域名解析文件的更新, 都会读到过时的信息, 进而造成域名解析失败

这种现象产生的原因是DNS文件的更新晚于MQTT的域名解析请求. 这样 `getaddrinfo()` 系统调用就会得到一个 `EAI_AGAIN` 错误 然而, 如果不通过 `res_init()` , C库中的 `getaddrinfo()` 即使被重试逻辑调用也仍然读取过时的DNS文件, 并继续得到 `EAI_AGAIN` 错误

例如, 可以改动 `HAL_TLS_mbedtls.c`

```
#include <arpa/nameser.h>
#include <resolv.h>
...
...
if ((ret = getaddrinfo(host, port, &hints, &addr_list)) != 0) {
    if (ret == EAI_AGAIN)
      res_init();
      return (MBEDTLS_ERR_NET_UNKNOWN_HOST);
}
```

以上只是一个示意的改法，`res_init()` 本身也是一个过时的函数，所以不建议进入SDK的官方代码，用户可以酌情加在合适的位置

-0x0044/-68/MBEDTLS_ERR_NET_CONNECT_FAILED

解释

socket连接失败：`The connection to the given server / port failed`

可能的原因和解决建议

- TCP连接失败
  - 请确认连接的目标IP地址/域名，以及端口号是否正确

-0x0043/-67/MBEDTLS_ERR_NET_BUFFER_TOO_SMALL

解释

SSL报文缓冲区过短：`Buffer is too small to hold the data`

可能的原因和解决建议

- SSL被配置的最大报文长度太小
  - 可调整 `MBEDTLS_SSL_MAX_CONTENT_LEN` 的值，重新编译再试
  - `MBEDTLS_SSL_MAX_CONTENT_LEN` 的值，目前已知最小不能小于 `4096`

-0x0042/-66/MBEDTLS_ERR_NET_SOCKET_FAILED

解释

创建socket失败：`Failed to open a socket`

可能的原因和解决建议

- 系统可用的socket可能已全部被申请，在有些平台上，能被开启的socket数量，有上限的
  - 检查当前的流程是否有socket的泄漏，有些socket使用完后没有close
  - 如果流程正常，确需同时建立多个socket，请调整socket的个数上限

MQTT连接

-35/MQTT_CONNACK_BAD_USERDATA_ERROR

解释

发起MQTT协议中的Connect操作失败：`Connect request failed with the server returning a bad userdata error`

可能的原因和解决建议

- 使用了错误的设备三元组
  - 建议检查 productKey/deviceName/deviceSecret 是否正确并且是同一套
- 三元组传入的参数不正确
  - 建议检查C函数的传参过程
- 设备三元组被禁用
  - 若经检查确系禁用，请在控制台或是使用服务端相应API解除禁用

-37/MQTT_CONNACK_IDENTIFIER_REJECTED_ERROR

解释

云端认为设备权限不足而拒绝设备端的连接请求：`CONNACK` 报文中的错误码是 `2`

可能的原因和解决建议

- MQTT的Connect报文参数中, clientId的上报内容和协议预期不符, 这通常发生在不使用官方SDK而使用第三方MQTT自行对接时
- 建议检查在控制台创建当前设备三元组的账号, 是否有权限, 如是否有发生欠费等

-42/MQTT_PUSH_TO_LIST_ERROR

解释

订阅或是发布(QoS1)太多太快, 导致SDK内部队列满了而出错

可能的原因和解决建议

- 如果是订阅过程中发生的错误, 说明当前订阅使用的队列长度不够
  - 可以考虑调整 `IOTX_MC_SUB_NUM_MAX` 的值
- 如果是发布过程中发生的错误, 可能是发送的频率太快, 或是网络状态不佳
  - 可以考虑使用QoS0来发布
  - 可以考虑调整 `IOTX_MC_REPUB_NUM_MAX` 的值

什么是域名直连, 如何开启

C-SDK 的 2.0 以上版本, MQTT连接有两种方式, 一种是认证模式, 一种是域名直连模式

- 认证模式是设备先用 HTTPS 方式访问类似 `https://iot-auth.cn-shanghai.aliyuncs.com:443` 的地址, 进行鉴权
- 鉴权通过之后, 再用 MQTT 方式连接到类似 public.iot-as-mqtt.cn-shanghai.aliyuncs.com:1883 的 MQTT 服务器地址
- 域名直连模式是设备直接用 MQTT 方式连接到类似 ${productKey}.iot-as-mqtt.cn-shanghai.aliyuncs.com:1883的 MQTT 服务器地址
- 这样不需要访问 HTTPS+MQTT 两台服务器才建立连接, 更快建立连接, 耗费的端上资源也更少

2.0 以上的版本, 默认已经开启了直连模式, 如果配置文件 `make.settings` 不是默认的状态了, 那么

打开 `make.setting` 文件, 在文件尾部写入 `FEATURE_MQTT_DIRECT = y` , 然后运行 `make` 命令重新编译, 即可确保开启直连模式

MQTT协议版本是多少

C-SDK 的 2.0 以上版本, 封装的MQTT协议版本号是 `3.1.1`

例程 mqtt-example 连接上线后会很快下线, 如何修改可以一直处于在线状态

mqtt-example 例程本身的逻辑是发送一次消息后会自动退出

若需要此例程保持长期在线, 执行命令时加上 "loop" 参数即可, 例如:

```
.output/release/bin/mqtt-example loop
```

当您这样做时，请确保使用自己申请的 productKey/deviceName/deviceSecret 覆盖代码中的设备身份信息后后重新编译例程

因为官方SDK的代码中默认编写的是一台公用设备的信息，有其他人同时使用时，您会被踢下线，同一组设备信息只能同时在线一个连接

如何持续的接收MQTT消息

需要循环多次的调用 IOT_MQTT_Yield 函数，在其内部会自动的维持心跳和接收云端下发的MQTT消息

可以参考 examples 目录下，mqtt-example.c 里面的 while 循环部分

心跳和重连

心跳的时间间隔如何设置

在 IOT_MQTT_Construct 的入参中可以设置 `keepalive_interval_ms` 的值，C-SDK用这个值作为心跳间隔时间

keepalive_interval_ms 的取值范围是 60000 - 300000，即最短心跳间隔为1分钟，最长心跳间隔为5分钟

设备端是如何侦测到需要重连(reconnect)的

设备端C-SDK会在 `keepalive_interval_ms` 时间间隔过去之后，再次发送 MQTT ping request，并等待 MQTT ping response

如果在接下来的下一个 `keepalive_interval_ms` 时间间隔内，没有收到对应的 ping response

或是在进行报文的上行发送(send)或者下行接收(recv)时发生异常，则 C-SDK 就认为此时网络断开了，需要进行重连

设备端的重连机制是什么

C-SDK 的重连动作是内部触发，无需用户显式干预. 对于使用基础版的用户，需要时常调用IOT_MQTT_Yield 函数将 CPU 交给SDK，重连就在其中发生

重连如果不成功，会在间隔一段时间之后持续重试，直到再次连接成功为止

相邻的两次重连之间的间隔时间是指数退避+随机数的关系，例如间隔1s+随机秒数，间隔2s+随机秒数，间隔4s+随机秒数，间隔8s+随机秒数... 直到间隔达到了间隔上限(默认60s)

关于 IOT_MQTT_Yield

IOT_MQTT_Yield 的作用

IOT_MQTT_Yield的作用主要是给SDK机会去尝试从网络上接收数据

因此在需要接收数据时(subscribe/unsubscribe之后，publish之后，以及希望收到云端下推数据时)，都需要主动调用该函数

IOT_MQTT_Yield 参数 timeout 的意义

IOT_MQTT_Yield 会阻塞住 timeout 指定的时间(单位毫秒)去尝试接收数据，直到超出这个时间，才会返回调用它的函数

IOT_MQTT_Yield 与 HAL_SleepMs 的区别

都会阻塞一段时间才返回，但是IOT_MQTT_Yield实质是去从网络上接收数据

而HAL_SleepMs则是什么也不做，单纯等待入参指定的时间间隔过去后返回

订阅相关

如果订阅了多个topic, 调用一次IOT_MQTT_Yield, 可能接收到多个topic的消息吗

调用一次 IOT_MQTT_Yield, 如果多个topic都被订阅成功并且都有数据下发, 则可以一次性接收到来自多个topic的消息

什么情况下会发生订阅超时

- 调用IOT_MQTT_Construct 和云端建立MQTT连接时, 其入参中可以设置请求超时时间 `request_timeout_ms` 的值
- 如果两倍 `request_timeout_ms` 时间过去, 仍未收到来自云端的 `SUBACK` 消息, 则会触发订阅(Subscribe)超时
- 超时的事件会通过 IOT_MQTT_Construct 入参中设置的 `event_handle` 回调函数通知到用户
- 在调用IOT_MQTT_Subscribe 之后, 需要尽快执行 IOT_MQTT_Construct 以接收SUBACK, 请勿使用HAL_SleepMs

发布相关

发布(Publish)的消息最长是多少, 超过会怎么样

- C-SDK的代码上来说, MQTT的消息报文长度, 受限于 IOT_MQTT_Construct 入参中 `write_buf` 和 `read_buf` 的大小
- 从云端协议上来说, MQTT消息报文长度不能超过256KB, 具体查阅 官网限制说明页面 为准
- 如果实际上报消息长度大于C-SDK的限制, 消息会被丢掉

被发布消息payload格式是怎么样的

阿里云物联网平台并没有指定pub消息的payload格式

需要客户根据应用场景制定自己的协议, 然后以JSON格式或其它格式, 放到pub消息载体里面传给服务端

物联网平台云端是否一有消息就立刻推送给订阅的设备而不做保存

消息一到达云端, 就会分发给不同的设备订阅者, 云端服务器不进行保存, 目前不支持 MQTT 协议中的 will 和 retain 特性

MQTT长连接时, 云端如何侦测到设备离线的

云端会根据用户调用 IOT_MQTT_Construct 时传入的 `keepalive_interval_ms` , 作为心跳间隔, 等待 MQTT ping request

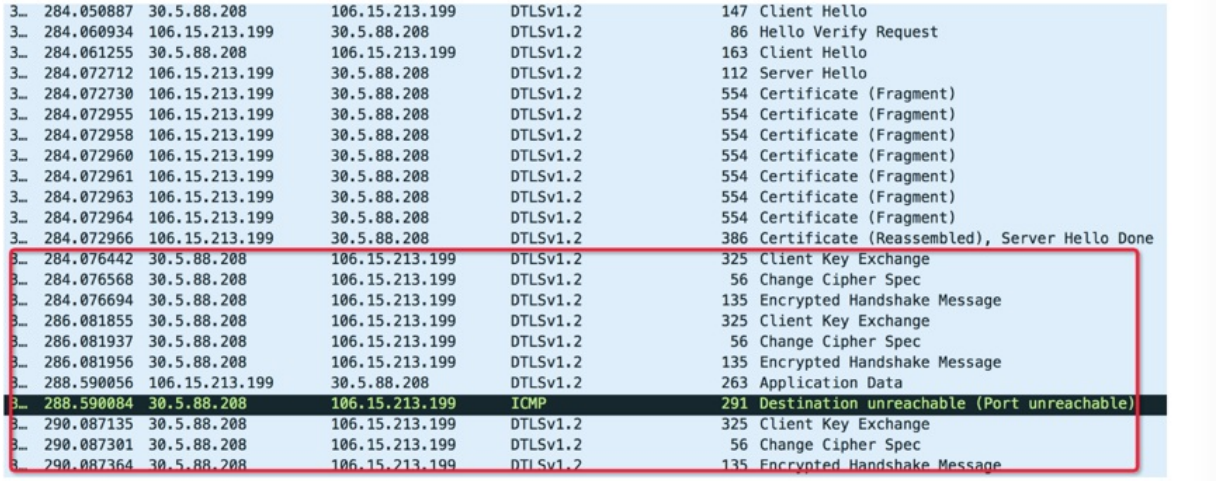
如果在约定的心跳间隔过去之后, 最多再等5秒钟, 若还是没有收到 MQTT ping request, 则认为设备离线

CoAP上云问题

认证超时失败, 目前总结下来分为下述两种情况

连接失败的时候, ssl->state为MBEDTLS_SSL_SERVER_CHANGE_CIPHER_SPEC的情况

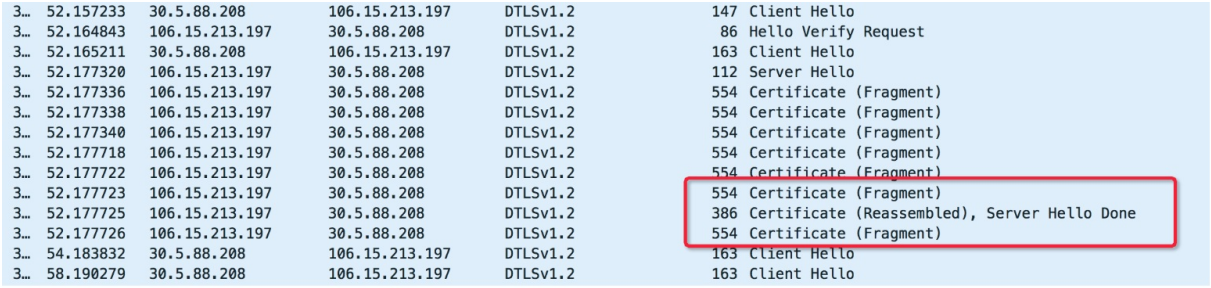
在出现问题的一次测试中通过tcpdump进行抓包, 结果通过wireshark展示如下图. 其中client(ip地址为30.5.88.208)发出了握手数据包(包括client key exchange, change cipher spec, encrypted handshake message),但是服务端并没有回复change cipher spec的数据包, 从而导致coap连云卡在这一步. 卡在这一步后, 但如果等待2s后没有等到数据, 客户端会再进行两次尝试超时等待(4s, 8s);如果3次都等不到服务端发来的change cipher spec的数据包, 则退出handshake过程, 导致连接失败由上分析可知, 这个问题的根因是在服务端.



3…	284.050887	30.5.88.208	106.15.213.199	DTLSv1.2	147	Client Hello
3…	284.060934	106.15.213.199	30.5.88.208	DTLSv1.2	86	Hello Verify Request
3…	284.061255	30.5.88.208	106.15.213.199	DTLSv1.2	163	Client Hello
3…	284.072712	106.15.213.199	30.5.88.208	DTLSv1.2	112	Server Hello
3…	284.072730	106.15.213.199	30.5.88.208	DTLSv1.2	554	Certificate (Fragment)
3…	284.072955	106.15.213.199	30.5.88.208	DTLSv1.2	554	Certificate (Fragment)
3…	284.072958	106.15.213.199	30.5.88.208	DTLSv1.2	554	Certificate (Fragment)
3…	284.072960	106.15.213.199	30.5.88.208	DTLSv1.2	554	Certificate (Fragment)
3…	284.072961	106.15.213.199	30.5.88.208	DTLSv1.2	554	Certificate (Fragment)
3…	284.072963	106.15.213.199	30.5.88.208	DTLSv1.2	554	Certificate (Fragment)
3…	284.072964	106.15.213.199	30.5.88.208	DTLSv1.2	554	Certificate (Fragment)
3…	284.072966	106.15.213.199	30.5.88.208	DTLSv1.2	386	Certificate (Reassembled), Server Hello Done
3…	284.076442	30.5.88.208	106.15.213.199	DTLSv1.2	325	Client Key Exchange
3…	284.076568	30.5.88.208	106.15.213.199	DTLSv1.2	56	Change Cipher Spec
3…	284.076694	30.5.88.208	106.15.213.199	DTLSv1.2	135	Encrypted Handshake Message
3…	286.081855	30.5.88.208	106.15.213.199	DTLSv1.2	325	Client Key Exchange
3…	286.081937	30.5.88.208	106.15.213.199	DTLSv1.2	56	Change Cipher Spec
3…	286.081956	30.5.88.208	106.15.213.199	DTLSv1.2	135	Encrypted Handshake Message
3…	288.590056	106.15.213.199	30.5.88.208	DTLSv1.2	263	Application Data
3…	288.590084	30.5.88.208	106.15.213.199	ICMP	291	Destination unreachable (Port unreachable)
3…	290.087135	30.5.88.208	106.15.213.199	DTLSv1.2	325	Client Key Exchange
3…	290.087301	30.5.88.208	106.15.213.199	DTLSv1.2	56	Change Cipher Spec
3…	290.087364	30.5.88.208	106.15.213.199	DTLSv1.2	135	Encrypted Handshake Message

连接失败的时候, ssl->state为MBEDTLS_SSL_HANDSHAKE_OVER的情况

在出现问题的一次测试中通过tcpdump进行抓包, 结果通过wireshark展示如下图. 服务端(106.15.213.197)发来的certificate数据包的顺序出现了错乱, server_hello_done这个certificate数据包并不是作为最后一个certicate包, 而是作为倒数第二个(对比上图即能发现). 在这种情况下, 客户端解析服务端的certificate信息失败, 重新发送client hello消息等待server端回复. 客户端尝试过进行三次尝试超时等待(2s, 4s, 8s)后还没收到数据包, 就主动放弃, 导致连接失败. 由上分析可知, 这个问题的根因是在服务端未能正确返回握手信息



3…	52.157233	30.5.88.208	106.15.213.197	DTLSv1.2	147	Client Hello
3…	52.164843	106.15.213.197	30.5.88.208	DTLSv1.2	86	Hello Verify Request
3…	52.165211	30.5.88.208	106.15.213.197	DTLSv1.2	163	Client Hello
3…	52.177320	106.15.213.197	30.5.88.208	DTLSv1.2	112	Server Hello
3…	52.177336	106.15.213.197	30.5.88.208	DTLSv1.2	554	Certificate (Fragment)
3…	52.177338	106.15.213.197	30.5.88.208	DTLSv1.2	554	Certificate (Fragment)
3…	52.177340	106.15.213.197	30.5.88.208	DTLSv1.2	554	Certificate (Fragment)
3…	52.177718	106.15.213.197	30.5.88.208	DTLSv1.2	554	Certificate (Fragment)
3…	52.177722	106.15.213.197	30.5.88.208	DTLSv1.2	554	Certificate (Fragment)
3…	52.177723	106.15.213.197	30.5.88.208	DTLSv1.2	554	Certificate (Fragment)
3…	52.177725	106.15.213.197	30.5.88.208	DTLSv1.2	386	Certificate (Reassembled), Server Hello Done
3…	52.177726	106.15.213.197	30.5.88.208	DTLSv1.2	554	Certificate (Fragment)
3…	54.183832	30.5.88.208	106.15.213.197	DTLSv1.2	163	Client Hello
3…	58.190279	30.5.88.208	106.15.213.197	DTLSv1.2	163	Client Hello

CoAP上云连接的云端URI是什么

- 在调用IOT_CoAP_Init 的时候, 可以设置其参数 `iotx_coap_config_t` 里面的 `p_url`
- 如果 `p_url` 值为NULL, 则C-SDK会自动使用 `IOTX_ONLINE_DTLS_SERVER_URL` 这个宏所定义的默认URL

```
#define IOTX_ONLINE_DTLS_SERVER_URL "coaps://%s.iot-as-coap.cn-shanghai.aliyuncs.com:5684"
```

- 其中 `%s` 是使用 `p_devinfo` 里面的 `product_key` , 所以请确保在初始化 `iotx_coap_config_t` 的时候一定要对 `p_devinfo` 赋值

IOT_CoAP_SendMessage 发送的消息必须是JSON格式吗, 如果不是JSON会出现什么错误

- 目前, 除了支持JSON格式外, 也可以支持CBOR格式
- 因为是与云端通信, 需要使用指定格式, 否则可能会出现无法解析的问题

关于 IOT_CoAP_Yield

如何设置 IOT_CoAP_Yield 处理结果最大等待时间?

目前默认设置是2000ms, 可以修改 `COAP_WAIT_TIME_MS` 这个宏进行调整

HTTP上云问题

认证连接

HTTPS进行设备认证时, Server会返回的错误码及其含义

- 10000: common error(未知错误)
  - HTTPS报文是有一定格式要求, 必须符合要求server才能支持
  - `ContentType` 只支持"application/json"
  - 只支持HTTPS
  - 只支持POST方法
- 40000: request too many(请求次数过多, 流控限制)
  - 同一个设备在一天内的认证次数是有限制的
  - 解决方法: 每次认证获得的token是有48小时有效期的, 过期前可以反复使用, 无需每次都去认证获取新的token
- 400: authorized failure(认证失败)
  - 服务器认为鉴权参数是不合法的, 鉴权失败
  - 解决方法: 检查 `IOTX_PRODUCT_KEY` , `IOTX_DEVICE_NAME` , `IOTX_DEVICE_SECRET` , `IOTX_DEVICE_ID` 是不是从控制台获得的正确参数

TLS连接问题

设备端TLS密码算法

目前C-SDK连接云端用到的默认TLS算法是: `TLS-RSA-WITH-AES-256-CBC-SHA256`

云端TLS密码算法

目前阿里云IoT平台支持的TLS算法清单如下:

- `TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384`
- `TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384`
- `TLS_RSA_WITH_AES_256_CBC_SHA256`
- `TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA384`
- `TLS_ECDH_RSA_WITH_AES_256_CBC_SHA384`
- `TLS_DHE_RSA_WITH_AES_256_CBC_SHA256`
- `TLS_DHE_DSS_WITH_AES_256_CBC_SHA256`
- `TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA`
- `TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA`

- TLS_RSA_WITH_AES_256_CBC_SHA
- TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA
- TLS_ECDH_RSA_WITH_AES_256_CBC_SHA
- TLS_DHE_RSA_WITH_AES_256_CBC_SHA
- TLS_DHE_DSS_WITH_AES_256_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
- TLS_RSA_WITH_AES_128_CBC_SHA256
- TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256
- TLS_ECDH_RSA_WITH_AES_128_CBC_SHA256
- TLS_DHE_RSA_WITH_AES_128_CBC_SHA256
- TLS_DHE_DSS_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
- TLS_RSA_WITH_AES_128_CBC_SHA
- TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA
- TLS_ECDH_RSA_WITH_AES_128_CBC_SHA
- TLS_DHE_RSA_WITH_AES_128_CBC_SHA
- TLS_DHE_DSS_WITH_AES_128_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_RC4_128_SHA
- TLS_ECDHE_RSA_WITH_RC4_128_SHA
- SSL_RSA_WITH_RC4_128_SHA
- TLS_ECDH_ECDSA_WITH_RC4_128_SHA
- TLS_ECDH_RSA_WITH_RC4_128_SHA
- TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA384
- TLS_ECDH_RSA_WITH_AES_256_GCM_SHA384
- TLS_DHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_DHE_DSS_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_RSA_WITH_AES_128_GCM_SHA256
- TLS_ECDH_ECDSA_WITH_AES_128_GCM_SHA256
- TLS_ECDH_RSA_WITH_AES_128_GCM_SHA256
- TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_DHE_DSS_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA
- TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA

- SSL_RSA_WITH_3DES_EDE_CBC_SHA
- TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA
- TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA
- SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA
- SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA
- SSL_RSA_WITH_RC4_128_MD5
- TLS_EMPTY_RENEGOTIATION_INFO_SCSV

4.设备认证

设备的身份认证分为一机一密以及一型一密两种：

- 一机一密： 在设备上烧写设备的ProductKey、DeviceName、DeviceSecret，然后适配相应的HAL并调用SDK提供的连接云端的函数即可，这种方式要求对设备的产线工具进行一定的修改，需要对每个设备烧写不同的DeviceName和DeviceSecret；
- 一型一密： 设备上烧写设备的ProductKey、ProductSecret，每个设备需要具备自己的唯一标识并将该标识预先上传到阿里云IoT物联网平台，然后调用SDK提供的函数连接云端。这种方式每个设备上烧写的信息是固定的ProductKey、ProductSecret

一机一密编程

1. 设置设备三元组信息：

在v3.0.1中，用户需要实现下面的三个HAL

- HAL_GetProductKey
- HAL_GetDeviceName
- HAL_GetDeviceSecret

在v3.1.0/v3.2.0中，通过IOT_Ioctl设置三元组

- IOT_Ioctl(IOTX_IOCTL_SET_PRODUCT_KEY, g_product_key)
- IOT_Ioctl(IOTX_IOCTL_SET_DEVICE_NAME, g_device_name)
- IOT_Ioctl(IOTX_IOCTL_SET_DEVICE_SECRET, g_device_secret)

代码示例(mqtt_example.c)：

```
int main(int argc, char *argv[])
{
    void            *pclient = NULL;
    int             res = 0;
    int             loop_cnt = 0;
    iotx_mqtt_param_t     mqtt_params;
    memset(&mqtt_params, 0x0, sizeof(mqtt_params));
    mqtt_params.handle_event.h_fp = example_event_handle;
    pclient = IOT_MQTT_Construct(&mqtt_params);
    if (NULL == pclient) {
        EXAMPLE_TRACE("MQTT construct failed");
        return -1;
    }
    ...
}
```

++注：IOT_MQTT_Construct()将会调用上面提到的HAL_GetProductKey()等三个HAL函数去获取设备的身份信息。++

一型一密编程

使用的流程示意

简写说明

- PK: Product Key, 设备品类标识字符串
- PS: Product Secret, 设备品类密钥
- DN：DeviceName, 某台设备的标识字符串
- DS: DeviceSecret, 某台设备的设备密钥

流程简述：

1. 设备使用PK、PS和设备标识（DN）到阿里云物联网获取该设备对应的DS
2. 阿里云物联网的动态注册Server将查找该设备的标识（DN）是否在该PK对应的设备列表，如果该设备在列表中则将该设备的DS返回
3. 设备收到DS之后，将使用一机一密的方式计算MQTT连接参数以及签名
4. 设备使用计算出来的MQTT连接参数连接阿里云物联网平台

注1: 用户必须将获取的DeviceSecret持久化到设备, 以备后续使用. 若获取的DeviceSecret丢失可能导致设备无法上线等严重后果, 云端服务器不会接受已激活设备重复的动态注册请求

注2: 使用一型一密功能, 用户必须对每个设备进行预注册, 即在阿里云物联网平台的控制台上传每个设备的DeviceName，并且在控制台上打开对应产品的动态注册功能

例子程序讲解

一型一密功能的例子程序在 `src/dynamic_register/examples/dynreg_example.c` , 以下对其逐段讲解

要使用一型一密功能, 要包含它的头文件 `dynreg_api.h`

```
#include <stdio.h>
#include <string.h>
#include "infra_types.h"
#include "infra_defs.h"
#include "dynreg_api.h"
```

准备输入参数 `region` 和出入参结构体 `meta`

```
iotx_http_region_types_t region = IOTX_HTTP_REGION_SHANGHAI;
HAL_Printf("dynreg example\n");
memset(&meta,0,sizeof(iotx_dev_meta_info_t));
HAL_GetProductKey(meta.product_key);
HAL_GetProductSecret(meta.product_secret);
HAL_GetDeviceName(meta.device_name);
```

以上例子程序用 `IOTX_CLOUD_REGION_SHANGHAI` 代表的华东二站点作为例子, 演示连接上海服务器时候的情况, 另一个入参 `meta` 其实就是填入设备的 `PK/PS/DN`

调用一型一密的API获取 `DeviceSecret`

```
res = IOT_Dynamic_Register(region, &meta);
if (res < 0) {
  HAL_Printf("IOT_Dynamic_Register failed\n");
  return -1;
}
HAL_Printf("\nDevice Secret: %s\n\n", meta.device_secret);
```

这个 `IOT_Dynamic_Register()` 接口就是一型一密功能点唯一提供的用户接口，若执行成功，在参数 `meta` 中将填上从服务器成功获取到的 `DeviceSecret`

其它

参考上面的图示

- 第1步和第2步对应用户接口：`IOT_Dynamic_Register()`
- 第3步对应用户接口：`IOT_Sign_MQTT()`
- 第4步对应用户接口：`IOT_MQTT_Construct()`

注：

- 因为当设备获取到DeviceSecret之后再次调用 `IOT_Dynamic_Register()` 将会返回失败，因此用户编程时获取到DeviceSecret之后需要将其保存到Flash中；
- 用户的程序在调用 `IOT_Dynamic_Register()` 之前应该先调用HAL_GetDeviceSecret()查看设备是否已经获取到了DeviceSecret，如果已经获取到，则无需再次去调用 `IOT_Dynamic_Register()`

功能API接口

原型

```
int32_t IOT_Dynamic_Register(iotx_http_region_types_t region, iotx_dev_meta_info_t *meta);
```

接口说明

根据输入参数中指定的站点区域，以及 `productKey` 和 `productSecret`，去云端为 `deviceName` 指定的设备去云端申请 `deviceSecret`

参数说明

参数	数据类型	方向	说明
region	iotx_http_region_types_t	输入	表示设备将要工作的区域，例如美西/新加坡/日本/华东2站点等
meta	iotx_dev_meta_info_t *	输入输出	输入的时候带入设备的 `PK/PS/DN`，输出的时候返回从服务器取到的 `DS`

5.物模型编程

物模型管理功能是指使用 属性/事件/服务 的方式对产品支持的能力进行描述，在设备开发时也需要以物模型的方式进行编程。

下面的讲解中使用了示例代码*./src/dev_model/examples/linkkit_example_solo.c*。

> 注意
>
> 1. 在Linux环境下，用户可通过修改wrappers/os/ubuntu/HAL_OS_linux.c文件中的默认三元组来使用自己在云端控制台创建的设备的身份信息。
> 2. 我们在*src/dev_model/examples*目录下提供了名为*model_for_example.json*的物模型描述文件，用户可以用自己产品的productkey替换掉该文件中的productKey值后，将该物模型文件导入到物联网平台上的产品定义中，这样用户可以快速的参照示例代码来体验基于物模型的编程方式。

设备属性

- 属性上报说明

  用户可以调用IOT_Linkkit_Report()函数来上报属性，属性上报时需要按照云端定义的属性格式使用JSON编码后进行上报。示例中函数user_post_property展示了如何使用IOT_Linkkit_Report进行属性上报（对于异常情况的上报，详见example）。

```
void user_post_property(void)
{
    static int cnt = 0;
    int res = 0;
    char property_payload[30] = {0};
    HAL_Snprintf(property_payload, sizeof(property_payload), "{\"Counter\": %d}", cnt++);
    res = IOT_Linkkit_Report(EXAMPLE_MASTER_DEVID, ITM_MSG_POST_PROPERTY,
                             (unsigned char *)property_payload, strlen(property_payload));
    EXAMPLE_TRACE("Post Property Message ID: %d", res);
}
```

  > 注意　property_payload = “{\” Counter\”:1}” 即是将属性编码为JSON对象。

  当上报属性或者事件时需要云端应答时，通过 IOT_Ioctl 对 IOTX_IOCTL_RECV_EVENT_REPLY 选项进行设置。

- 属性设置说明

  示例代码在回调函数 user_property_set_event_handler 中获取云端设置的属性值，并原样将收到的数据发回给云端，这样可以更新在云端的设备属性值，用户可在此处对收到的属性值进行处理。

  > 说明　该回调函数是在example初始化时使用 IOT_RegisterCallback 注册的 ITE_PROPERTY_SET 事件对应的回调函数。

```
static int user_property_set_event_handler(const int devid, const char *request, const int request_len)
{
    int res = 0;
    user_example_ctx_t *user_example_ctx = user_example_get_ctx();
    EXAMPLE_TRACE("Property Set Received, Devid: %d, Request: %s", devid, request);
    res = IOT_Linkkit_Report(user_example_ctx->master_devid, ITM_MSG_POST_PROPERTY,
                             (unsigned char *)request, request_len);
    EXAMPLE_TRACE("Post Property Message ID: %d", res);
    return 0;
}
```

设备服务

在示例程序中，当收到服务调用请求(包括同步服务和异步服务)时，会进入下面列出的回调函数：

```
static int user_service_request_event_handler(const int devid, const char *serviceid, const int serviceid_len
,
                        const char *request, const int request_len,
                        char **response, int *response_len){    int add_result = 0;
  cJSON *root = NULL, *item_number_a = NULL, *item_number_b = NULL;
  const char *response_fmt = "{\"Result\": %d}";
  EXAMPLE_TRACE("Service Request Received, Service ID: %.*s, Payload: %s", serviceid_len, serviceid, reque
st);
  /* Parse Root */
  root = cJSON_Parse(request);
  if (root == NULL || !cJSON_IsObject(root)) {
    EXAMPLE_TRACE("JSON Parse Error");
    return -1;
  }
  if (strlen("Operation_Service") == serviceid_len && memcmp("Operation_Service", serviceid, serviceid_le
n) == 0) {
    /* Parse NumberA */
    item_number_a = cJSON_GetObjectItem(root, "NumberA");
    if (item_number_a == NULL || !cJSON_IsNumber(item_number_a)) {
      cJSON_Delete(root);
      return -1;
    }
    EXAMPLE_TRACE("NumberA = %d", item_number_a->valueint);
    /* Parse NumberB */
    item_number_b = cJSON_GetObjectItem(root, "NumberB");
    if (item_number_b == NULL || !cJSON_IsNumber(item_number_b)) {
      cJSON_Delete(root);
      return -1;
    }
    EXAMPLE_TRACE("NumberB = %d", item_number_b->valueint);
    add_result = item_number_a->valueint + item_number_b->valueint;
    /* 服务应答数据，数据长度通过response，response_len参数传给SDK */
    *response_len = strlen(response_fmt) + 10 + 1;
    *response = (char *)HAL_Malloc(*response_len);
    if (*response == NULL) {
      EXAMPLE_TRACE("Memory Not Enough");
      return -1;
    }
    memset(*response, 0, *response_len);
    HAL_Snprintf(*response, *response_len, response_fmt, add_result);
    *response_len = strlen(*response);
  }
  cJSON_Delete(root);
  return 0;
}
```

用户需要动态分配内存空间用于存放服务应答数据，并通过response参数返回给SDK，SDK在完成应答数据发送后会负责response指向内存的释放。

> **说明** 对服务输出参数为空的情况，`*response` 应指向存放JSON对象 `{}` 的内存，不能让 `*response` 指向空指针。

设备事件

示例中使用 `IOT_Linkkit_TriggerEvent` 上报事件。其中展示了如何使用 `IOT_Linkkit_Report` 进行事件上报（对于异常情况的上报，详见example）。

```
void user_post_event(void){
  int res = 0;
  char *event_id = "HardwareError";
  char *event_payload = "{\"ErrorCode\": 0}";
  res = IOT_Linkkit_TriggerEvent(EXAMPLE_MASTER_DEVID, event_id, strlen(event_id),
                  event_payload, strlen(event_payload));
  EXAMPLE_TRACE("Post Event Message ID: %d", res);
}
```

关于上报消息的格式说明及示例

上报属性时，属性ID和值以JSON格式的形式放在IOT_Linkkit_Report()的 `payload` 中，不同数据类型以及多个属性的格式示例如下。

```
/* 整型数据 */
char *payload = "{\"Brightness\":50}";
/* 浮点型数据上报 */
char *payload = "{\"Temperature\":11.11}";
/* 枚举型数据上报 */
char *payload = "{\"WorkMode\":2}";
/* 布尔型数据上报, 在物模型定义中, 布尔型为整型, 取值为0或1, 与JSON格式的整型不同 */
char  *payload = "{\"LightSwitch\":1}";
/* 字符串数据上报 */
char *payload = "{\"Description\":\"Amazing Example\"}";
/* 时间型数据上报, 在物模型定义中, 时间型为字符串 */
char *payload = "{\"Timestamp\":\"1252512000\"}";
/* 复合类型属性上报, 在物模型定义中, 符合类型属性为JSON对象 */
char *payload = "{\"RGBColor\":{\"Red\":11,\"Green\":22,\"Blue\":33}}";
/* 多属性上报, 如果需要上报以上各种数据类型的所有属性, 将它们放在一个JSON对象中即可 */
char *payload = "{\"Brightness\":50,\"Temperature\":11.11,\"WorkMode\":2,\"LightSwitch\":1,\"Description\
":\"Amazing Example\",\"Timestamp\":\"1252512000\",\"RGBColor:{\"Red\":11,\"Green\":22,\"Blue\":33}\"}";
/* 属性payload准备好以后, 就可以使用如下接口进行上报了 */
IOT_Linkkit_Report(devid, ITM_MSG_POST_PROPERTY, payload, strlen(payload));
```

上报事件时，与上报属性的区别是，事件ID需要单独拿出来，放在IOT_Linkkit_TriggerEvent()的eventid中，而事件的上报内容，也就是物模型定义中事件的输出参数，则使用与上报属性相同的格式进行上报，示例如下。

```
/* 事件ID为Error, 其输出参数ID为ErrorCode, 数据类型为枚举型 */
char *eventid = "Error";
char *payload = "{\"ErrorCode\":0}";
/* 事件ID为HeartbeatNotification, 其输出参数有2个. 第一个是布尔型参数ParkingState, 第二个是浮点型参数Voltag
eValue */
char *eventid = "HeartbeatNotification";
char *payload = "{\"ParkingState\":1,\"VoltageValue\":3.0}";
/* 事件payload准备好以后, 就可以使用如下接口进行上报了 */
IOT_Linkkit_TriggerEvent(devid, event_id, strlen(event_id), payload, strlen(payload));
/* 从上面的示例可以看出, 当事件的输出参数有多个时, payload的格式与多属性上报是相同的 */
```

基于MQTT Topic进行数据收发

> **注意** 虽然物模型编程的API并未返回MQTT编程接口IOT_MQTT_XXX()所需要的pClient参数，但基于MQTT Topic进行数据收发仍可和物模型编程混用。

- 所有MQTT数据收发的接口，第1个参数都可接受参数 0 作为输入，表示 “使用当前唯一的MQTT通道进行数据收发等操作”，包括：
  - IOT_MQTT_Construct
  - IOT_MQTT_Destroy
  - IOT_MQTT_Yield
  - IOT_MQTT_CheckStateNormal
  - IOT_MQTT_Subscribe
  - IOT_MQTT_Unsubscribe
  - IOT_MQTT_Publish
  - IOT_MQTT_Subscribe_Sync
  - IOT_MQTT_Publish_Simple
- 例如要在使用物模型编程API的程序代码段落中表示对某个Topic进行订阅，可以用

```
IOT_MQTT_Subscribe(0, topic_request, IOTX_MQTT_QOS0, topic_callback, topic_context);
```

- 例如要在使用物模型编程API的程序代码段落中表示在某个Topic进行发布（数据上报），可以用

```
IOT_MQTT_Publish_Simple(0, topic, IOTX_MQTT_QOS0, payload, payload_len);
```

与物模型功能相关的API列表

函数名	说明
IOT_Linkkit_Open	创建本地资源，在进行网络报文交互之前，必须先调用此接口，得到一个会话的句柄
IOT_Linkkit_Connect	对主设备/网关来说，将会建立设备与云端的通信。对于子设备来说，将向云端注册该子设备（若需要），并添加主子设备拓扑关系
IOT_Linkkit_Yield	若SDK占有独立线程，该函数只将接收到的网络报文分发到用户的回调函数中，否则表示将CPU交给SDK让其接收网络报文并将消息分发到用户的回调函数中
IOT_Linkkit_Close	若入参中的会话句柄为主设备/网关，则关闭网络连接并释放SDK为该会话所占用的所有资源
IOT_Linkkit_TriggerEvent	向云端发送事件报文、错误码、异常告警等
IOT_Linkkit_Report	向云端发送没有云端业务数据下发的上行报文，包括属性值/设备标签/二进制透传数据/子设备管理等各种报文

函数名	说明
IOT_Linkkit_Query	向云端发送存在云端业务数据下发的查询报文，包括OTA状态查询/OTA固件下载/子设备拓扑查询/NTP时间查询等各种报文

函数名	说明
IOT_RegisterCallback	对SDK注册事件回调函数，如云端连接成功/失败，有属性设置/服务请求到达，子设备管理报文答复等
IOT_Ioctl	对SDK进行各种参数运行时设置和获取，以及运行状态的信息获取等，实参可以是任何数据类型

6.标签

设备标签用于展示如设备信息, 如厂商/设备型号的静态扩展信息, 以键值对的形式存储在云端， 详细说明可参见设备标签

向云端发送更新设备标签的消息

上报设备标签信息更新的消息时, 标签信息放在成对的一组或多组attrKey和attrValue中. attrKey的定义支持大小写字母及数字, 但不能以数字开头, 长度为2~32个字节, payload为JSON数组类型, 可包含多对{"attrKey":"","attrValue":""}, 示例如下

```
/* 设备标签有两组, 第一组attrKey为abc, attrValue为Hello,World. 第二组attrKey为def, attrValue为Hello,Aliyun */
char *payload = "[{\"attrKey\":\"abc\",\"attrValue\":\"Hello,World\"},{\"attrKey\":\"def\",\"attrValue\":\"Hello
,Aliyun\"}]";
/* 设备标签payload准备好以后, 就可以使用如下接口进行上报了 */
IOT_Linkkit_Report(devid, ITM_MSG_DEVICEINFO_UPDATE, (unsigned char *)payload, strlen(payload));
```

向云端发送删除设备标签的消息

上报设备标签信息删除的消息时, 将标签信息的attrKey放在一个或多个 `attrKey` 中

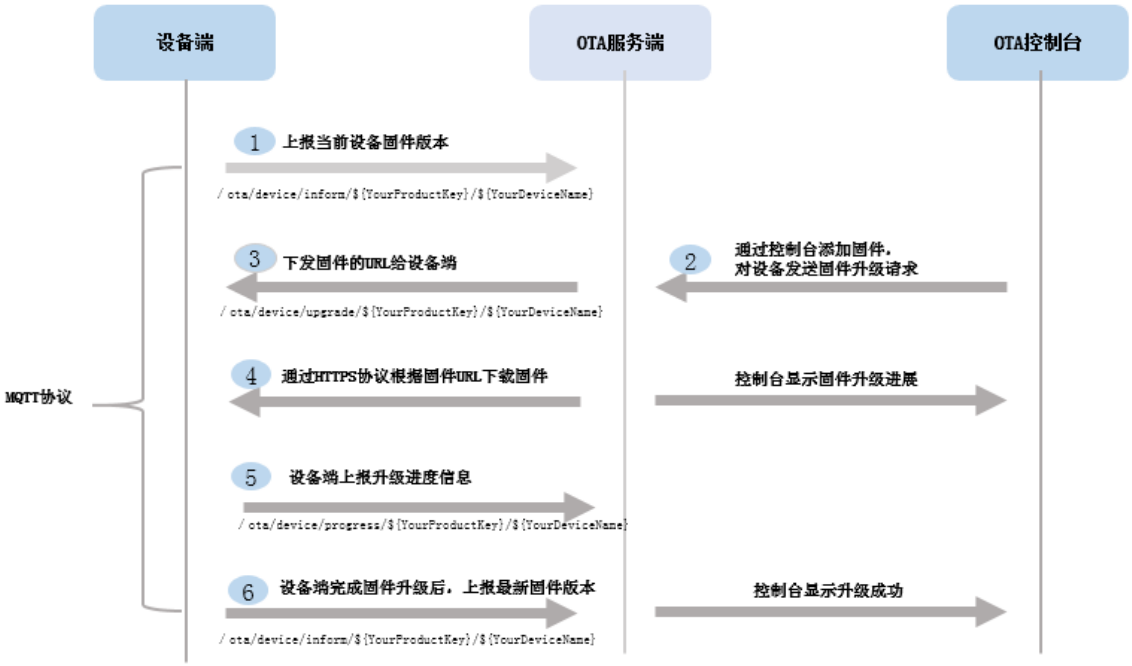
```
/* 设备标签attrKey有两个, 第一个attrKey为abc. 第二个attrKey为def */
char *payload = "[{\"attrKey\":\"abc\"},{\"attrKey\":\"def\"}]";
/* 设备标签payload准备好以后, 就可以使用如下接口进行上报了 */
IOT_Linkkit_Report(devid, ITM_MSG_DEVICEINFO_DELETE, (unsigned char *)payload, strlen(payload));
```

7.设备OTA开发

OTA（Over-the-Air Technology）即空中下载技术，物联网平台支持通过OTA方式进行设备固件升级。

背景信息

基于MQTT协议下固件升级流程如下。



OTA例程讲解

- 过OTA的API可以实现设备端固件下载。但是如何存储/使用下载到的固件，需要用户实现。
- 存储固件是指将下载到的固件存储到FLASH等介质上。
- 使用固件，包括加载新下载的固件，需要用户根据业务的具体需求（例如需要用户单击升级按钮）来实现。

> **说明** OTA整体流程请见OTA服务。

下面用两个例子分别说明如何用基础版接口和高级版接口来实现OTA功能。

用基础版接口实现的OTA例程

现对照 src/ota/examples/ota_example_mqtt.c 例程分步骤讲解如何使用基础版的接口实现OTA的功能。

1. OTA业务建立前的准备：导入设备三元组，初始化连接信息。

```
int main(int argc, char *argv[]) {
  ...
  /**< get device info*/
  HAL_SetProductKey(PRODUCT_KEY);
  HAL_SetDeviceName(DEVICE_NAME);
  HAL_SetDeviceSecret(DEVICE_SECRET);
  /**< end*/
  _ota_mqtt_client()
}
```

2. 在_ota_mqtt_client函数完成建连和OTA的主要配置逻辑。

```
  /* Device AUTH */
  if (0 != IOT_SetupConnInfo(g_product_key, g_device_name, g_device_secret, (void **)&pconn_info)) {
    EXAMPLE_TRACE("AUTH request failed!");
    rc = -1;
    goto do_exit;
  }
  /* Initialize MQTT parameter */
  memset(&mqtt_params, 0x0, sizeof(mqtt_params));
  mqtt_params.port = pconn_info->port;
  mqtt_params.host = pconn_info->host_name;
  mqtt_params.client_id = pconn_info->client_id;
  mqtt_params.username = pconn_info->username;
  mqtt_params.password = pconn_info->password;
  mqtt_params.pub_key = pconn_info->pub_key;
  mqtt_params.request_timeout_ms = 2000;
  mqtt_params.clean_session = 0;
  mqtt_params.keepalive_interval_ms = 60000;
  mqtt_params.read_buf_size = OTA_MQTT_MSGLEN;
  mqtt_params.write_buf_size = OTA_MQTT_MSGLEN;
  mqtt_params.handle_event.h_fp = event_handle;
  mqtt_params.handle_event.pcontext = NULL;
  /* Construct a MQTT client with specify parameter */
  pclient = IOT_MQTT_Construct(&mqtt_params);
  if (NULL == pclient) {
    EXAMPLE_TRACE("MQTT construct failed");
    rc = -1;
    goto do_exit;
  }
```

3. 在 `_ota_mqtt_client` 函数进行OTA有关的初始化工作（主要是订阅跟这个设备有关的固件升级信息）。

```
  h_ota = IOT_OTA_Init(PRODUCT_KEY, DEVICE_NAME, pclient);
  if (NULL == h_ota) {
    rc = -1;
    EXAMPLE_TRACE("initialize OTA failed");
    goto do_exit;
  }
```

4. 建立一个循环，一直去尝试接收OTA升级的消息。

```
int ota_over = 0;
do {
  uint32_t firmware_valid;
  EXAMPLE_TRACE("wait ota upgrade command....");
  /* 接收MQTT消息 */
  IOT_MQTT_Yield(pclient, 200);
  /* 判断接收到的消息中是否有固件升级的消息 */
  if (IOT_OTA_IsFetching(h_ota)) {
   /* 下载OTA内容, 上报下载进度,见章节 "5. 下载OTA内容, 上报下载进度" */
   /* 校验固件的md5, 见章节 "6. 校验md5的值" */
  }
  } while (!ota_over);
```

需要到服务端推送一个固件升级事件下去， `IOT_OTA_IsFetching` 返回才能结果为1，才能走入固件升级的逻辑。推送固件升级事件的具体步骤如下。

i. 到IoT控制台的 OTA服务页面，单击**新增固件**。

ii. 单击**创建固件**，**验证固件**。

iii. 单击这个新增固件的**批量升级**按钮，从中选择设备所属产品为*examples/ota/ota_mqtt-example.c*中三元组对应的产品。

iv. 待升级版本号点开下拉框选当前版本号，升级范围选*定向升级*，再从设备范围中选当前的三元组对应的设备，单击**确定**即可。

5. 下载OTA内容，上报下载进度。

```
do {
  /* 下载OTA固件 */
  len = IOT_OTA_FetchYield(h_ota, buf_ota, OTA_BUF_LEN, 1);
  if (len > 0) {
    if (1 != fwrite(buf_ota, len, 1, fp)) {
      EXAMPLE_TRACE("write data to file failed");
      rc = -1;
      break;
    }
  } else {
    /* 上报已下载进度 */
    IOT_OTA_ReportProgress(h_ota, IOT_OTAP_FETCH_FAILED, NULL);
    EXAMPLE_TRACE("ota fetch fail");
  }
  /* get OTA information */
  /* 获取已下载到的数据量, 文件总大小, md5信息, 版本号等信息 */
  IOT_OTA_Ioctl(h_ota, IOT_OTAG_FETCHED_SIZE, &size_downloaded, 4);
  IOT_OTA_Ioctl(h_ota, IOT_OTAG_FILE_SIZE, &size_file, 4);
  IOT_OTA_Ioctl(h_ota, IOT_OTAG_MD5SUM, md5sum, 33);
  IOT_OTA_Ioctl(h_ota, IOT_OTAG_VERSION, version, 128);
  last_percent = percent;
  percent = (size_downloaded * 100) / size_file;
  if (percent - last_percent > 0) {
    /* 上报已下载进度 */
    IOT_OTA_ReportProgress(h_ota, percent, NULL);
    IOT_OTA_ReportProgress(h_ota, percent, "hello");
  }    IOT_MQTT_Yield(pclient, 100);
  /* 判断下载是否结束 */
} while (!IOT_OTA_IsFetchFinish(h_ota));
```

6. 校验md5的值。

```
IOT_OTA_Ioctl(h_ota, IOT_OTAG_CHECK_FIRMWARE, &firmware_valid, 4);
if (0 == firmware_valid) {
  EXAMPLE_TRACE("The firmware is invalid");
} else {
  EXAMPLE_TRACE("The firmware is valid");
}
ota_over = 1;
```

7. 用户通过 `IOT_OTA_Deini t` 释放所有资源。

```
if (NULL != h_ota) {
  IOT_OTA_Deinit(h_ota);
}
if (NULL != pclient) {
  IOT_MQTT_Destroy(&pclient);
}
if (NULL != msg_buf) {
  HAL_Free(msg_buf);
}
if (NULL != msg_readbuf) {
  HAL_Free(msg_readbuf);
}
if (NULL != fp) {
  fclose(fp);
}
return rc;
```

8. 固件的存储。

   在 `_ota_mqtt_client` 函数中通过下述方式打开，写入和关闭一个文件。

```
fp = fopen("ota.bin", "wb+")
...
if (1 != fwrite(buf_ota, len, 1, fp)) {
  EXAMPLE_TRACE("write data to file failed");
  rc = -1;
  break;
}
...
if (NULL != fp) {
  fclose(fp);
}
```

用高级版接口实现的OTA例程

现对照*src/dev_model/examples/linkkit_example_solo.c*分步骤讲解如何使用高级版的接口实现OTA的功能。

1. 初始化主设备，注册FOTA的回调函数，建立与云端的连接。

```
int res = 0;
int domain_type = 0, dynamic_register = 0, post_reply_need = 0;
iotx_linkkit_dev_meta_info_t master_meta_info;
memset(&g_user_example_ctx, 0, sizeof(user_example_ctx_t));
memset(&master_meta_info, 0, sizeof(iotx_linkkit_dev_meta_info_t));
memcpy(master_meta_info.product_key, PRODUCT_KEY, strlen(PRODUCT_KEY));
memcpy(master_meta_info.product_secret, PRODUCT_SECRET, strlen(PRODUCT_SECRET));
memcpy(master_meta_info.device_name, DEVICE_NAME, strlen(DEVICE_NAME));
memcpy(master_meta_info.device_secret, DEVICE_SECRET, strlen(DEVICE_SECRET));
/* Register Callback */
...
...
IOT_RegisterCallback(ITE_FOTA, user_fota_event_handler);
domain_type = IOTX_CLOUD_REGION_SHANGHAI;
IOT_Ioctl(IOTX_IOCTL_SET_DOMAIN, (void *)&domain_type);
/* Choose Login Method */
dynamic_register = 0;
IOT_Ioctl(IOTX_IOCTL_SET_DYNAMIC_REGISTER, (void *)&dynamic_register);
/* post reply doesn't need */
post_reply_need = 1;IOT_Ioctl(IOTX_IOCTL_RECV_EVENT_REPLY, (void *)&post_reply_need);
/* Create Master Device Resources */
g_user_example_ctx.master_devid = IOT_Linkkit_Open(IOTX_LINKKIT_DEV_TYPE_MASTER, &master_m
eta_info);
if (g_user_example_ctx.master_devid < 0) {
  EXAMPLE_TRACE("IOT_Linkkit_Open Failed\n");
  return -1;}
/* Start Connect Aliyun Server */
res = IOT_Linkkit_Connect(g_user_example_ctx.master_devid);
if (res < 0) {
  EXAMPLE_TRACE("IOT_Linkkit_Connect Failed\n");
  return -1;
}
```

2. 实现上述代码中的回调函数 `user_fota_event_handler` 。

   该回调函数在如下两种情况下会被触发：

   - 直接收到云端下发的新固件通知时。
   - 由设备端主动发起新固件查询，云端返回新固件通知时。

   在收到新固件通知后，可调用 `IOT_Linkkit_Query` 进行固件下载。

```
int user_fota_event_handler(int type, const char *version){
  char buffer[128] = {0};
  int buffer_length = 128;
  /* 0 - new firmware exist, query the new firmware */
  if (type == 0) {
    EXAMPLE_TRACE("New Firmware Version: %s", version);
    IOT_Linkkit_Query(EXAMPLE_MASTER_DEVID, ITM_MSG_QUERY_FOTA_DATA, (unsigned char *)buff
er, buffer_length);
  }
  return 0;
}
```

3. 固件的存储。

用户需要实现如下3个HAL接口来实现固件的存储。

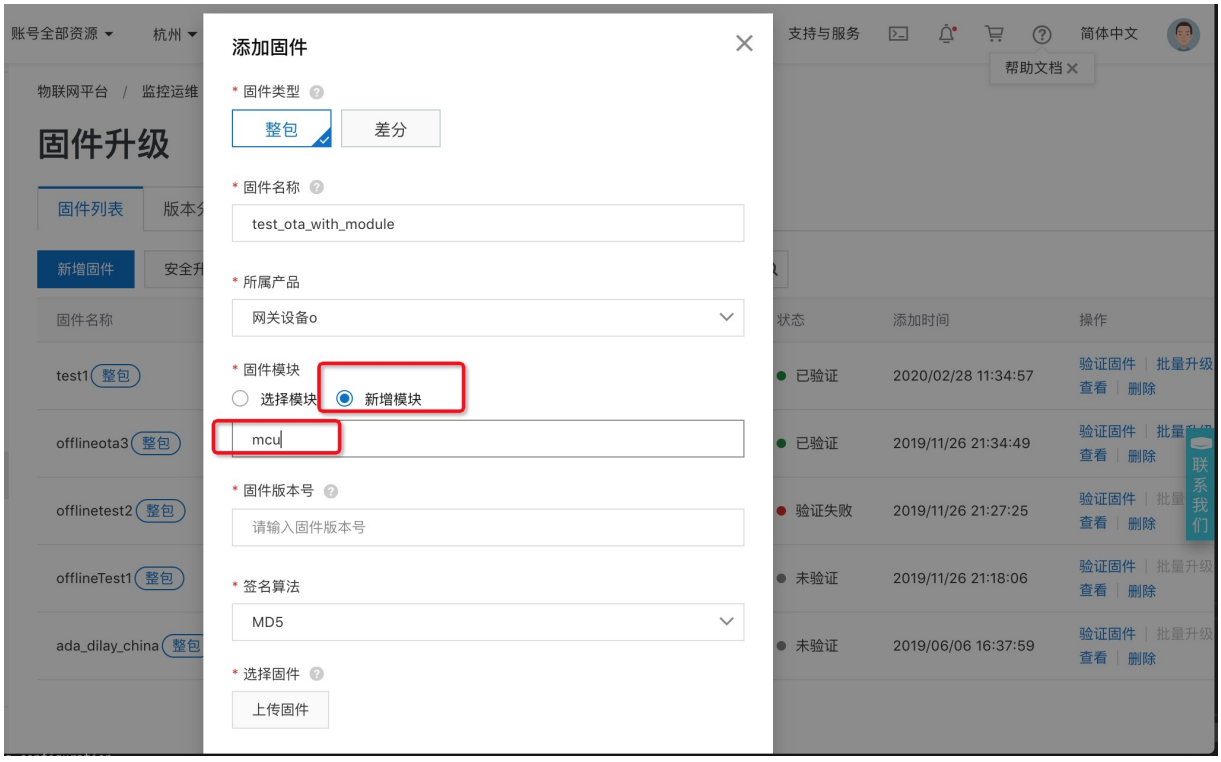
```
/* SDK在开始下载固件之前进行调用 */
void HAL_Firmware_Persistence_Start(void);
/* SDK在接收到固件数据时进行调用 */
int HAL_Firmware_Persistence_Write(char *buffer, uint32_t length);
/* SDK在固件下载结束时进行调用 */
int HAL_Firmware_Persistence_Stop(void);
```

4. 用户主动发起新固件查询。

```
IOT_Linkkit_Query(user_example_ctx->master_devid, ITM_MSG_REQUEST_FOTA_IMAGE,
                  (unsigned char *)("app-1.0.0-20180101.1001"), 30);
```

OTA支持多模块

OTA除了可以升级设备固件外，还可以下载并升级设备的软件模块，客户需要在物联网平台的控制台上创建模块，如下图所示：



设备端开发时，需要将module设置为与控制台的模块名一致：

```
char* module = "mcu";
IOT_Ioctl(IOTX_IOCTL_SET_MODULE, (void *)module);
```

其他流程如正常的ota测试流程，在云端控制台部署任务后，设备端会有如下日志，具体见其中的module字段：

```
[dbg] otamqtt_UpgrageCb(111): topic=/ota/device/upgrade/a1u8Mae7PjW/foDzDjPtmG2GC53PDJ9d
[dbg] otamqtt_UpgrageCb(112): len=431, topic_msg={"code":"1000","data":{"size":143360,"module":"mcu",
"sign":"867f1536fb5a9af8c40e3a2205436252","version":"111","url":"https://iotx-ota-daily.oss-cn-shanghai.
aliyuncs.com/ota/338ac9db05545dcab910cb211cdf2c52/ck75mi9h800003h75xbgz61vl.tar?Expires=15829517
57&OSSAccessKeyId=aS4MT0lYrPSPj6Gy&Signature=urw%2F9WAlizQuiQOlzHxvxxEGd0A%3D","signMethod
":"Md5","md5":"867f1536fb5a9af8c40e3a2205436252"},"id":1582865357795,"message":"success"}
[dbg] otamqtt_UpgrageCb(129): receive device upgrade
[inf] ofc_Init(47): protocol: https
received state: -0x092C(msg queue size: 0, max size: 50)
received state: -0x092C(msg enqueue w/ message type: 43)
received state: -0x092C(msg dequeue)
received state: -0x0938(alink event type: 43)
received state: -0x0938(new fota information received, 111)
user_fota_module_event_handler.219: New Firmware Version: 111, module: mcu
```

OTA功能API

- 用基础版接口实现OTA功能涉及的API。

函数名	说明
IOT_OTA_Init	OTA实例的构造函数，创建一个OTA会话的句柄并返回。
IOT_OTA_Deinit	OTA实例的摧毁函数，销毁所有相关的数据结构。
IOT_OTA_Ioctl	OTA实例的输入输出函数，根据不同的命令字可以设置OTA会话的属性，或者获取OTA会话的状态。
IOT_OTA_GetLastError	OTA会话阶段，若某个 `IOT_OTA_XXX()` 函数返回错误，调用此接口可获得最近一次的详细错误码。
IOT_OTA_ReportVersion	OTA会话阶段，向服务端汇报当前的固件版本号。
IOT_OTA_FetchYield	OTA下载阶段，在指定的timeout时间内，从固件服务器下载一段固件内容，保存在入参buffer中。
IOT_OTA_IsFetchFinish	OTA下载阶段，判断迭代调用 `IOT_OTA_FetchYield` 是否已经下载完所有的固件内容。
IOT_OTA_IsFetching	OTA下载阶段，判断固件下载是否仍在进行中，尚未完成全部固件内容的下载。
IOT_OTA_ReportProgress	可选API，OTA下载阶段，调用此函数向服务端汇报已经下载了全部固件内容的百分之多少。

- 用高级版接口实现OTA功能涉及的API。

函数名	说明
IOT_Linkkit_Open	创建本地资源，在进行网络报文交互之前，必须先调用此接口，得到一个会话的句柄。

函数名	说明
IOT_Linkkit_Connect	对主设备/网关来说，将会建立设备与云端的通信。对于子设备来说，将向云端注册该子设备（若需要），并添加主子设备拓扑关系。
IOT_Linkkit_Yield	若SDK占有独立线程，该函数只将接收到的网络报文分发到用户的回调函数中，否则表示将CPU交给SDK让其接收网络报文并将消息分发到用户的回调函数中。
IOT_Linkkit_Close	若入参中的会话句柄为主设备/网关，则关闭网络连接并释放SDK为该会话所占用的所有资源。
IOT_Linkkit_Query	向云端发送存在云端业务数据下发的查询报文，包括OTA状态查询/OTA固件下载/子设备拓扑查询/NTP时间查询等各种报文。
IOT_RegisterCallback	对SDK注册事件回调函数，如云端连接成功/失败，有属性设置/服务请求到达，子设备管理报文答复等。

需要实现的HAL

- 用基础版接口实现OTA功能需要实现的API。

  无。

- 用高级版接口实现OTA功能需要实现的API。

函数名	说明
HAL_Firmware_Persistence_Start	固件持久化功能开始
HAL_Firmware_Persistence_Write	固件持久化写入固件
HAL_Firmware_Persistence_Stop	固件持久化功能结束

8.子设备管理

本节内容将结合 src/dev_model/examples/linkkit_example_gateway.c 为例讲解网关产品的编程方法。

背景信息

名词	说明
网关	能够直接连接物联网平台的设备，且具有子设备管理功能，能够代理子设备连接云端。
子设备	本质上也是设备。子设备不能直接连接物联网平台，只能通过网关连接。
设备ID	Device ID，也就是设备句柄，在网关场景中用于标识一个具体的设备，调用 IOT_Linkkit_Open 时返回。
拓扑关系	子设备和网关的关联关系为拓扑关系，子设备与网关建立拓扑关系后，便可以复用网关的物理通道进行数据通信。
子设备动态注册	子设备在注册时只需将productKey和deviceName上报给网关，网关代替子设备向云端发起身份认证并获取云端返回的deviceSecret用之后的上线操作。

管理子设备

1. 网关与云端建连。

   网关的建连过程与单品直连设备的建连过程完全一致。

   i. 调用 IOT_RegisterCallback 注册必要的回调处理函数，如连接事件处理，设备连云初始化完成处理，属性设置事件处理等回调函数。子设备和网关共用一组回调处理函数，以参数DeviceID来区分不同的设备。

   ```
   IOT_RegisterCallback(ITE_CONNECT_SUCC, user_connected_event_handler);
   IOT_RegisterCallback(ITE_DISCONNECTED, user_disconnected_event_handler);
   IOT_RegisterCallback(ITE_PROPERTY_SET, user_property_set_event_handler);
   IOT_RegisterCallback(ITE_REPORT_REPLY, user_report_reply_event_handler);
   IOT_RegisterCallback(ITE_TIMESTAMP_REPLY, user_timestamp_reply_event_handler);
   IOT_RegisterCallback(ITE_INITIALIZE_COMPLETED, user_initialized);
   IOT_RegisterCallback(ITE_PERMIT_JOIN, user_permit_join_event_handler);
   ```

   ii. 调用 IOT_Ioctl 进行必要的配置，如选择服务器站点，选择是否使用一型一密等等。

   ```
   /* Choose Login Server */
   int domain_type = IOTX_CLOUD_REGION_SHANGHAI;
   IOT_Ioctl(IOTX_IOCTL_SET_DOMAIN, (void *)&domain_type);
   /* Choose Login Method */
   int dynamic_register = 0;
   IOT_Ioctl(IOTX_IOCTL_SET_DYNAMIC_REGISTER, (void *)&dynamic_register);
   ```

iii. 使用 IOTX_LINKKIT_DEV_TYPE_MASTER 参数调用 IOT_Linkkit_Open 初始化主设备资源。

```
iotx_linkkit_dev_meta_info_t master_meta_info;
memset(&master_meta_info, 0, sizeof(iotx_linkkit_dev_meta_info_t));
memcpy(master_meta_info.product_key, PRODUCT_KEY, strlen(PRODUCT_KEY));
memcpy(master_meta_info.product_secret, PRODUCT_SECRET, strlen(PRODUCT_SECRET));
memcpy(master_meta_info.device_name, DEVICE_NAME, strlen(DEVICE_NAME));
memcpy(master_meta_info.device_secret, DEVICE_SECRET, strlen(DEVICE_SECRET));
/* Create Master Device Resources */
user_example_ctx->master_devid = IOT_Linkkit_Open(IOTX_LINKKIT_DEV_TYPE_MASTER, &mast
er_meta_info);
if (user_example_ctx->master_devid < 0) {
  EXAMPLE_TRACE("IOT_Linkkit_Open Failed\n");
  return -1;
}
```

iv. 同样使用 IOTX_LINKKIT_DEV_TYPE_MASTER 参数调用 IOT_Linkkit_Connect 与云端建立连接。

```
/* Start Connect Aliyun Server */
res = IOT_Linkkit_Connect(user_example_ctx->master_devid);
if (res < 0) {
  EXAMPLE_TRACE("IOT_Linkkit_Connect Failed\n");
  return -1;
}
```

v. 在 ITE_INITIALIZE_COMPLETED 事件处理函数中确认网关连云初始化完成后，便可以进行下一步添加子设备的操作。

> **说明** 不要在步骤1中注册的回调函数中进行会阻塞线程操作，如调用 IOT_Linkkit_Connect 进行子设备建连，调用 IOT_Linkkit_Report 进行子设备上下线等。

2. 添加子设备。

   添加子设备主要由4个步骤完成。

   i. 使用 IOTX_LINKKIT_DEV_TYPE_SLAVE 参数调用 IOT_Linkkit_Open 初始化子设备资源。

   > **说明** 如果需要使用动态注册，只需要将设备信息参数的 device_secret 配置为空字符串即可。启用动态注册功能需要把子设备的DeviceName事先在物联网控制台预注册。

   ii. 调用 IOT_Linkkit_Connect 将子设备连上云端，这个接口为同步接口，会自动完成子设备注册和拓扑关系的添加。

   iii. 使用 ITM_MSG_LOGIN 参数调用 IOT_Linkkit_Report 完成子设备上线操作。

iv. 在 ITE_INITIALIZE_COMPLETED 事件处理函数中确认对应的子设备连云初始化完成后，便可以进行子设备与云端的数据交互了。

```
int example_add_subdev(iotx_linkkit_dev_meta_info_t *meta_info)
{
    int res = 0, devid = -1;
    devid = IOT_Linkkit_Open(IOTX_LINKKIT_DEV_TYPE_SLAVE, meta_info);
    if (devid == FAIL_RETURN) {
        EXAMPLE_TRACE("subdev open Failed\n");
        return FAIL_RETURN;
    }
    EXAMPLE_TRACE("subdev open susseed, devid = %d\n", devid);
    res = IOT_Linkkit_Connect(devid);
    if (res == FAIL_RETURN) {
        EXAMPLE_TRACE("subdev connect Failed\n");
        return res;
    }
    EXAMPLE_TRACE("subdev connect success: devid = %d\n", devid);
    res = IOT_Linkkit_Report(devid, ITM_MSG_LOGIN, NULL, 0);
    if (res == FAIL_RETURN) {
        EXAMPLE_TRACE("subdev login Failed\n");
        return res;
    }
    EXAMPLE_TRACE("subdev login success: devid = %d\n", devid);
    return res;
}
```

注意 使用相同 ProductKey ， DeviceName 重复调用 IOT_Linkkit_Open 初始化子设备资源，将返回相同的 devid ，SDK不会重复创建子设备资源。因此，在子设备创建成功后，用户可通过重复调用 IOT_Linkkit_Open 来查询 ProductKey ， DeviceName 对应的子设备 devid 。

3. 子设备管理相关操作。

i. 子设备登出。

使用 ITM_MSG_LOGOUT 选项调用 IOT_Linkkit_Report 即可完成子设备登出。子设备登出功能主要用于通知云端控制台设备处于离线状态。

```
    res = IOT_Linkkit_Report(devid, ITM_MSG_LOGOUT, NULL, 0);
    if (res == FAIL_RETURN) {
        EXAMPLE_TRACE("subdev logout Failed\n");
        return res;
    }
    EXAMPLE_TRACE("subdev logout success: devid = %d\n", devid);
```

ii. 获取子设备列表。

网关可以用 ITM_MSG_QUERY_TOPOLIST 选项来调用 IOT_Linkkit_Query 以获取与其存在拓扑关系的所有子设备信息。列表信息将在 ITE_TOPOLIST_REPLY 事件回调中返回。

iii. 删除子设备拓扑关系。

用户可以用 ITM_MSG_DELETE_TOPO 选项调用 IOT_Linkkit_Report 以删除参数 devid 指定的子设备拓扑关系，执行该命令后云端不会再将该子设备与本网关进行关联。

iv. 子设备OTA。

用户使用 `IOT_Ioctl` 配置要升级的子设备，再用 `IOT_Linkkit_Query` 来触发子设备升级。

- 调用 `IOT_RegisterCallback` 注册固件升级所用的回调函数 `user_fota_event_handler` 。

```
IOT_RegisterCallback(ITE_FOTA, user_fota_event_handler);
IOT_RegisterCallback(ITE_INITIALIZE_COMPLETED, user_initialized);
```

- 使用IOTX_LINKKIT_DEV_TYPE_MASTER参数调用IOT_Linkkit_Open初始化主设备资源。

```
iotx_linkkit_dev_meta_info_t master_meta_info;
memset(&master_meta_info, 0, sizeof(iotx_linkkit_dev_meta_info_t));
memcpy(master_meta_info.product_key, PRODUCT_KEY, strlen(PRODUCT_KEY));
memcpy(master_meta_info.product_secret, PRODUCT_SECRET, strlen(PRODUCT_SECRET));
memcpy(master_meta_info.device_name, DEVICE_NAME, strlen(DEVICE_NAME));
memcpy(master_meta_info.device_secret, DEVICE_SECRET, strlen(DEVICE_SECRET));
/* Create Master Device Resources */
user_example_ctx->master_devid = IOT_Linkkit_Open(IOTX_LINKKIT_DEV_TYPE_MASTER, &mas
ter_meta_info);
if (user_example_ctx->master_devid < 0) {
  EXAMPLE_TRACE("IOT_Linkkit_Open Failed\n");
  return -1;
}
```

- 调用IOT_Linkkit_Connect与云端建立连接。

```
/* Start Connect Aliyun Server */
res = IOT_Linkkit_Connect(user_example_ctx->master_devid);
if (res < 0) {
  EXAMPLE_TRACE("IOT_Linkkit_Connect Failed\n");
  return -1;
}
```

- 添加子设备，进行子设备OTA。

```
/* Add subdev */
while (user_example_ctx->subdev_index < EXAMPLE_SUBDEV_ADD_NUM) {
  if (user_example_ctx->master_initialized && user_example_ctx->subdev_index >= 0 &&
    (0 == current_ota_running) && (user_example_ctx->auto_add_subdev == 1 || user_example_
ctx->permit_join != 0)) {
    /* Add next subdev */
    if (example_add_subdev((iotx_linkkit_dev_meta_info_t *)&subdevArr[user_example_ctx->s
ubdev_index]) == SUCCESS_RETURN) {
      EXAMPLE_TRACE("subdev %s add succeed", subdevArr[user_example_ctx->subdev_index].
device_name);
    } else {
      EXAMPLE_TRACE("subdev %s add failed", subdevArr[user_example_ctx->subdev_index].de
vice_name);
    }
    user_example_ctx->subdev_index++;
    user_example_ctx->permit_join = 0;
    /* check each sub dev has remote ota message */
    do_subdev_ota(user_example_ctx->subdev_index);
    /* wait remote ota message */
    HAL_SleepMs(10000);
  }
}
```

- `do_subdev_ota` 函数详解。

  用户通过 `IOT_Ioctl` 这个接口来切换OTA通道为某个子设备（以devid区分_使用。切换后，只有这个子设备的升级消息能够被接收到。同时通过 `IOT_Linkkit_Query` 接口向云端查询是否有适合当前子设备的固件版本信息，如果有则走入OTA流程（触发用户自定义的OTA回调函数user_fota_event_handler）。

```
void do_subdev_ota(int devid)
{
  if (status_list[devid] == SUB_OTA_SUCCESS) {
    HAL_Printf("current devid is %d, its has checked remote ota message, skip\n", devid);
    return;
  }
  if (current_ota_running == 1) {
    return;
  }
  HAL_Printf("current devid running ota is %d\n", devid);
  int ota_result = 0;
  ota_result = IOT_Ioctl(IOTX_IOCTL_SET_OTA_DEV_ID, (void *)(&devid));
  if (0 != ota_result) {
    status_list[devid] = SUB_OTA_FAILED_SET_DEV_ID;
    HAL_Printf("IOTX_IOCTL_SET_OTA_DEV_ID failed, id is %d\n", devid);
    return;
  }
  ota_result = IOT_Linkkit_Query(devid, ITM_MSG_REQUEST_FOTA_IMAGE, (unsigned char *)cu
rrent_subdev_version,
                     strlen(current_subdev_version));
  HAL_Printf("current devid is %d, ITM_MSG_REQUEST_FOTA_IMAGE ret is %d\n", devid, ota_re
sult);
  if (0 != ota_result) {
    status_list[devid] = SUB_OTA_FAILED_QUERY;
    return;
  }
  status_list[devid] = SUB_OTA_SUCCESS;
}
```

4. 子设备数据交互。

   子设备与云端的数据交互方法与单品产品完全一致，详情可查看物模型编程章节。

   - 调用 `IOT_Linkkit_Report` 上报属性，上报透传数据，更新设备标签信息，删除设备标签信息。
   - 调用 `IOT_Linkkit_TriggerEvent` 进行Event的主动上报。
   - 在 `ITE_RAWDATA_ARRIVED` 事件回调中接收云端下发的透传数据。
   - 在 `ITE_SERVICE_REQUST` 事件回调中接收服务请求（同步服务和异步服务）。
   - 在 `ITE_PROPERTY_SET` 事件回调中处理云端下发的属性设置。
   - 在 `ITE_PROPERTY_GET` 事件回调中处理本地通信下发的属性获取

注意事项

1. 网关设备必须支持多线程，并使用独立线程用于执行 `IOT_Linkkit_Yield` 。

```
void *user_dispatch_yield(void *args)
{
  while (1){
    IOT_Linkkit_Yield(USER_EXAMPLE_YIELD_TIMEOUT_MS);
  }
  return NULL;
}
res = HAL_ThreadCreate(&g_user_dispatch_thread, user_dispatch_yield, NULL, NULL, NULL);
if (res < 0) {
  EXAMPLE_TRACE("HAL_ThreadCreate Failed\n");
  IOT_Linkkit_Close(user_example_ctx->master_devid);
  return -1;
}
```

2. 对接智能生活开放平台时，只有在收到云端下发的ITE_PERMIT_JOIN事件后，才可以执行子设备添加流程。

   当用户通过App扫码发起子设备添加时，App将会向云端发送PermitJoin命令，之后云端会将该命令转发给网关。ITE_PERMIT_JOIN事件会下发子设备的productKey和允许子设备接入的时间窗口timeoutSec（一般为60秒），厂商可在此窗口时间内去执行子设备的发现和绑定，并执行添加子设备流程上报云端，上报成功后便可以在App界面查看到添加的子设备。此功能让子设备的添加被有效的管控起来，只有在窗口时间内才可以添加子设备 `int user_permit_join_event_handler(const char product_key, const int time){ user_example_ctx_t user_example_ctx = user_example_get_ctx();` 。

```
EXAMPLE_TRACE(“Product Key: %s, Time: %d”, product_key, time);
user_example_ctx->permit_join = 1;
return 0;
}
```

3. 在SDK主目录的*make.setting*文件中添加 `FEATURE_DEVICE_MODEL_GATEWAY=y` ，再运行make命令即可编译出网关例程。

网关相关API

函数名	说明
IOT_Linkkit_Open	创建本地资源，在进行网络报文交互之前，必须先调用此接口，得到一个会话的句柄。
IOT_Linkkit_Connect	对主设备/网关来说，将会建立设备与云端的通信。对于子设备来说，将向云端注册该子设备（若需要），并添加主子设备拓扑关系。
IOT_Linkkit_Yield	若SDK占有独立线程，该函数内容为空，否则表示将CPU交给SDK让其接收网络报文并将消息分发到用户的回调函数中。
IOT_Linkkit_Close	若入参中的会话句柄为主设备/网关，则关闭网络连接并释放SDK为该会话所占用的所有资源。
IOT_Linkkit_Report	向云端发送没有云端业务数据下发的上行报文，包括属性值/设备标签/二进制透传数据/子设备管理等各种报文。

函数名	说明
IOT_Linkkit_Query	向云端发送存在云端业务数据下发的查询报文，包括OTA状态查询/OTA固件下载/子设备拓扑查询/NTP时间查询等各种报文。
IOT_Linkkit_TriggerEvent	向云端发送事件报文、如错误码、异常告警等。

其他通用函数名	说明
IOT_Ioctl	设置SDK运行时的可配置选项
IOT_RegisterCallback	注册事件回调函数

9.文件上传

SDK的文件上传功能使用HTTP2流式传输协议，将文件上传至阿里云物联网平台服务器.

- 支持多种上传模式，如以创建文件的方式上传，或以覆盖文件的方式上传
- 支持指定上传长度，并在下次上传时续传，用户可在上传时根据网络带宽配置上传分配大小(part_len)，以提高带宽利用效率

本节以 src/http2/http2_example_uploadfile.c 为例讲解如何使用文件上传功能

1. 与云端建立连接

调用 IOT_HTTP2_UploadFile_Connect 建立HTTP2连接，用户指定设备的三元组信息和服务器地址/端口号。

```
http2_upload_conn_info_t conn_info;
void *handle;
memset(&conn_info, 0, sizeof(http2_upload_conn_info_t));
conn_info.product_key = HTTP2_PRODUCT_KEY;
conn_info.device_name = HTTP2_DEVICE_NAME;
conn_info.device_secret = HTTP2_DEVICE_SECRET;
conn_info.url = HTTP2_ONLINE_SERVER_URL;
conn_info.port = HTTP2_ONLINE_SERVER_PORT;
handle = IOT_HTTP2_UploadFile_Connect(&conn_info, NULL);
if(handle == NULL) {
  return -1;
}
```

目前各个区域对应的域名和端口如下，其中 * 符号应使用设备的 ProductKey 替换，如 ProductKey 为 a1IgnOND7vI 时对应的URL、PORT如下：

```
#define HTTP2_ONLINE_SERVER_URL        "a1IgnOND7vI.iot-as-http2.cn-shanghai.aliyuncs.com"
#define HTTP2_ONLINE_SERVER_PORT        443
```

```
*.iot-as-http2.cn-shanghai.aliyuncs.com:443       // 上海正式
*.iot-as-http2.us-west-1.aliyuncs.com:443         // 美西正式
*.iot-as-http2.us-east-1.aliyuncs.com:443         // 美东正式
*.iot-as-http2.eu-central-1.aliyuncs.com:443      // 德国正式
*.iot-as-http2.ap-southeast-1.aliyuncs.com:443    // 新加坡正式
*.iot-as-http2.ap-northeast-1.aliyuncs.com:443    // 日本正式
```

如果用户关心网络状态，可以注册相应的回调函数，目前支持网络断开连接，和网络重连成功两个回调函数。

2. 文件上传

使用 IOT_HTTP2_UploadFile_Request 请求文件上传，例程以 UPLOAD_FILE_OPT_BIT_OVERWRITE 的方式上传，每次上传都会覆盖云端的文件。此接口为异步接口，用户可以插入多个上传请求到内部队列中。

```
http2_upload_params_t fs_params;
http2_upload_result_cb_t result_cb;
memset(&result_cb, 0, sizeof(http2_upload_result_cb_t));
result_cb.upload_completed_cb = upload_file_result;
result_cb.upload_id_received_cb = upload_id_received_handle;
memset(&fs_params, 0, sizeof(fs_params));
fs_params.file_path = argv[1];    /* 文件名称以命令行参数传入 */
fs_params.opt_bit_map = UPLOAD_FILE_OPT_BIT_OVERWRITE;
ret = IOT_HTTP2_UploadFile_Request(handle, &fs_params, &result_cb, NULL);
if(ret < 0) {
  return -1;
}
```

例程中注册了2个回调函数, 分别用于接收上传的结果, 和接收云端返回的上传标示符(upload_id). 在SDK调用了 upload_file_result 后, 文件上传操作便结束了, 用户可进行下一步操作

```
void upload_file_result(const char *file_path, int result, void *user_data)
{
  upload_end++;
  EXAMPLE_TRACE("========== file_path = %s, result = %d, finish num = %d ==========", file_path, res
ult, upload_end);
}
void upload_id_received_handle(const char *file_path, const char *upload_id, void *user_data)
{
  EXAMPLE_TRACE("========== file_path = %s, upload_id = %s ==========", file_path, upload_id);
  if (upload_id != NULL) {
    memcpy(g_upload_id, upload_id, strlen(upload_id));
  }
}
```

在上传过程中我们可以在log中看到HTTP2的报文交互:

1. 设备端请求云端打开文件上传的通道:

```
[inf] on_frame_send_callback(143): [INFO] C ---------> S (HEADERS) stream_id [1]
[inf] on_frame_send_callback(145): > :method: POST
[inf] on_frame_send_callback(145): > :path: /stream/open/c/iot/sys/thing/file/upload
[inf] on_frame_send_callback(145): > :scheme: https
[inf] on_frame_send_callback(145): > x-auth-name: devicename
[inf] on_frame_send_callback(145): > x-auth-param-client-id: a1IgnOND7vI.H2_FS01
[inf] on_frame_send_callback(145): > x-auth-param-signmethod: hmacsha1
[inf] on_frame_send_callback(145): > x-auth-param-product-key: a1IgnOND7vI
[inf] on_frame_send_callback(145): > x-auth-param-device-name: H2_FS01
[inf] on_frame_send_callback(145): > x-auth-param-sign: 8d6b80749ed63823dc16b2c1e7f049bbdd00bf2b
[inf] on_frame_send_callback(145): > x-sdk-version: 301
[inf] on_frame_send_callback(145): > x-sdk-version-name: 3.0.1
[inf] on_frame_send_callback(145): > x-sdk-platform: c
[inf] on_frame_send_callback(145): > content-length: 0
[inf] on_frame_send_callback(145): > x-file-name: upload1M
[inf] on_frame_send_callback(145): > x-file-overwrite: 1
[inf] on_begin_headers_callback(393): [INFO] C <--------- S (HEADERS) stream_id [1]
[inf] on_header_callback(363): < :status: 200
[inf] on_header_callback(363): < x-request-id: 1103919500797702144
[inf] on_header_callback(363): < x-next-append-position: 0
[inf] on_header_callback(363): < x-data-stream-id: DS1103919500889976832
[inf] on_header_callback(363): < x-file-upload-id: ULDS1103919500889976832
[inf] on_header_callback(363): < x-response-status: 200
```

1. 通道打开成功，接收到云端返回的文件上传标示符并调用用户回调函数：

```
upload_id_received_handle|037 :: ========== file_path = upload1M, upload_id = ULDS11039195008899768
32 ==========
```

1. 通道打开成功后，设备端通过HTTP2请求上传文件：

```
[inf] on_frame_send_callback(143): [INFO] C ---------> S (HEADERS) stream_id [3]
[inf] on_frame_send_callback(145): > :method: POST
[inf] on_frame_send_callback(145): > :path: /stream/send/c/iot/sys/thing/file/upload
[inf] on_frame_send_callback(145): > :scheme: https
[inf] on_frame_send_callback(145): > content-length: 1048576
[inf] on_frame_send_callback(145): > x-data-stream-id: DS1103919500889976832
[inf] on_frame_send_callback(145): > x-sdk-version: 301
[inf] on_frame_send_callback(145): > x-sdk-version-name: 3.0.1
[inf] on_frame_send_callback(145): > x-sdk-platform: c
[inf] on_frame_send_callback(145): > x-file-upload-id: ULDS1103919500889976832
[dbg] http2_stream_node_search(168): stream node not exist, stream_id = 3
[inf] send_callback(63): send_callback data len 10249, session->remote_window_size=16777215!
[inf] send_callback(72): send_callback data ends len = 10249!
[dbg] http2_stream_node_search(168): stream node not exist, stream_id = 3
[inf] iotx_http2_client_send(563): nghttp2_session_send 0
[dbg] _http2_fs_part_send_sync(250): send len = 10240
[inf] send_callback(63): send_callback data len 10249, session->remote_window_size=16766975!
[inf] send_callback(72): send_callback data ends len = 10249!
[inf] iotx_http2_client_send(563): nghttp2_session_send 0
[dbg] _http2_fs_part_send_sync(250): send len = 20480
[inf] send_callback(63): send_callback data len 10249, session->remote_window_size=16756735!
[inf] send_callback(72): send_callback data ends len = 10249!
[inf] iotx_http2_client_send(563): nghttp2_session_send 0
[dbg] _http2_fs_part_send_sync(250): send len = 30720
[inf] send_callback(63): send_callback data len 10249, session->remote_window_size=16746495!
[inf] send_callback(72): send_callback data ends len = 10249!
[inf] iotx_http2_client_send(563): nghttp2_session_send 0
[dbg] _http2_fs_part_send_sync(250): send len = 40960
```

1. 文件上传结束，等待云端上传结构应答，应答中的 `x-next-append-position` 便是已上传文件的大小

```
[inf] on_frame_recv_callback(196): on_frame_recv_callback, type = 8
[inf] on_frame_recv_callback(197): on_frame_recv_callback, stream_id = 3
[inf] on_frame_recv_callback(205): stream user data is not exist
[inf] on_begin_headers_callback(393): [INFO] C <--------- S (HEADERS) stream_id [3]
[inf] on_header_callback(363): < :status: 200
[inf] on_header_callback(363): < x-request-id: 1103919501166800896
[inf] on_header_callback(363): < x-next-append-position: 1048576
[inf] on_header_callback(363): < x-data-stream-id: DS1103919500889976832
[inf] on_header_callback(363): < x-response-status: 200
[inf] on_frame_recv_callback(196): [dbg] _http2_fs_part_send_sync(250): on_frame_recv_callback, type = 1
[inf] on_frame_recv_callback(197): on_frame_recv_callback, stream_id = 3
[inf] on_frame_recv_callback(205): send len = 1048576
[inf] _http2_fs_node_handle(350): file offset = 1048576 now
```

1. 最后SDK会关闭文件上传通道：

```
[inf] on_frame_send_callback(143): [INFO] C ---------> S (HEADERS) stream_id [5]
[inf] on_frame_send_callback(145): > :method: POST
[inf] on_frame_send_callback(145): > :path: /stream/close/c/iot/sys/thing/file/upload
[inf] on_frame_send_callback(145): > :scheme: https
[inf] on_frame_send_callback(145): > x-data-stream-id: DS1103919500889976832
[inf] on_frame_send_callback(145): > x-sdk-version: 301
[inf] on_frame_send_callback(145): > x-sdk-version-name: 3.0.1
[inf] on_frame_send_callback(145): > x-sdk-platform: c
[dbg] http2_stream_node_search(168): stream node not exist, stream_id = 5
[inf] iotx_http2_client_send(563): nghttp2_session_send 0
[inf] on_begin_headers_callback(393): [INFO] C <--------- S (HEADERS) stream_id [5]
[inf] on_header_callback(363): < :status: 200
[inf] on_header_callback(363): < x-request-id: 1103919502177628160
[inf] on_header_callback(363): < x-data-stream-id: DS1103919500889976832
[inf] on_header_callback(363): < x-file-crc64ecma: 6947770692288575170
[inf] on_header_callback(363): < x-response-status: 200
[inf] on_header_callback(363): < x-file-store-id: 101184
```

3. 断开连接

所有文件上传结束后使用 `IOT_HTTP2_UploadFile_Disconnect` 断开云端连接。

```
ret = IOT_HTTP2_UploadFile_Disconnect(handle);
```

功能API接口

`src/http2/http2_upload_api.h` 列出了HTTP2文件上传的所有API和相关数据类型定义

`src/http2/http2_wrapper.h` 列出了HTTP2所需的底层接口

HTTP2建立连接

接口原型

```
void *IOT_HTTP2_UploadFile_Connect(http2_upload_conn_info_t *conn_info, http2_status_cb_t *cb);
```

接口说明

创建HTTP2连接, 并注册相关状态回调函数. 此接口为同步接口, 当建连成功后会返回HTTP2连接句柄. 否则返回NULL

参数说明

参数	数据类型	方向	说明
conn_info	http2_upload_conn_info_t *	输入	设备连接信息

参数	数据类型	方向	说明
cb	http2_status_cb_t *	输入	设备状态回调函数结构体指针

返回值说明

值	说明
非NULL	建连成功
NULL	建连失败

参数附加说明

```
typedef struct {
  char *product_key;
  char *device_name;
  char *device_secret;
  char *url;
  int  port;
} http2_upload_conn_info_t;
```

- product_key: 三元组之一, 产品标示符
- device_name: 三元组之一, 设备名称
- device_secret: 三元组之一, 识别秘钥
- url: 云端服务器地址
- port: 云端服务器端口

```
typedef struct {
  http2_disconnect_cb_t  on_disconnect_cb;
  http2_reconnect_cb_t   on_reconnect_cb;
} http2_status_cb_t;
```

- on_disconnect_cb: HTTP2断连回调函数
- on_reconnect_cb: HTTP2重连回调函数

文件上传请求

接口原型

```
int IOT_HTTP2_UploadFile_Request(void *http2_handle, http2_upload_params_t *params, http2_upload_result_cb_t *cb, void *user_data);
```

接口说明

按照指定参数配置上传文件，并注册相关结果回调函数. 此接口为异步接口，上传结果由回调函数返回

参数说明

参数	数据类型	方向	说明
http2_handle	void *	输入	调用 IOT_HTTP2_UploadFile_Connect 建连成功后返回的句柄
params	http2_upload_params_t *	输入	上传参数结构体指针
cb	http2_upload_result_cb_t	输入	上传结构回调函数结构体指针
user_data	void *	输入	用户数据

返回值说明

值	说明
0	函数调用成功
< 0	函数调用失败

```
typedef struct {
  const char *file_path;    /* file path, filename must be ASCII string and strlen < 2014 */
  uint32_t part_len;      /* maximum content len of one http2 request, must be in range of 100KB ~ 100MB */
  const char *upload_id;    /* a specific id used to indicate one upload session, only required when UPLOAD_FILE_OPT_BIT_RESUME option set */
  uint32_t upload_len;    /* used to indicate the upload length, only required when UPLOAD_FILE_OPT_BIT_SPECIFIC_LEN option set */
  uint32_t opt_bit_map;    /* option bit map, support UPLOAD_FILE_OPT_BIT_OVERWRITE, UPLOAD_FILE_OPT_BIT_RESUME and UPLOAD_FILE_OPT_BIT_SPECIFIC_LEN */
} http2_upload_params_t;
```

- file_path: 文件路径，注意文件名必须为ASCII编码，且不能使用数字开头
- part_len: 文件上传分片大小，也即HTTP2请求content_len的最大长度，必须在100KB ~ 100MB范围内，否则会使用http2_config.h里的默认长度
- upload_id: 上传标示符，由首次上传时返回. 在需要使用断点续传方式上传时需添加 opt_bit_map 参数 UPLOAD_FILE_OPT_BIT_RESUME ，并指定对应上传标示符
- upload_len: 指定本次请求的长度，仅在 opt_bit_map 参数中有配置 UPLOAD_FILE_OPT_BIT_SPECIFIC_LEN 时才能起作用
- opt_bit_map: 位定义的选项表，可以使用按位或的方式配置此选项
  - 如 opt_bit_map = UPLOAD_FILE_OPT_BIT_OVERWRITE | UPLOAD_FILE_OPT_BIT_SPECIFIC_LEN 表示使用覆盖方式上传指定的文件长度

```
#define UPLOAD_FILE_OPT_BIT_OVERWRITE     (0x00000001)
#define UPLOAD_FILE_OPT_BIT_RESUME        (0x00000002)
#define UPLOAD_FILE_OPT_BIT_SPECIFIC_LEN  (0x00000004)
```

- UPLOAD_FILE_OPT_BIT_OVERWRITE: 使用覆盖的方式上传文件. 如果云端文件已存在, 而未使用覆盖方式, 则文件上传会失败. 未使用此选项, 将使用创建文件的方式上传文件
- UPLOAD_FILE_OPT_BIT_RESUME: 使用断点续传的方式上传文件. 使用此选项需填写上传标示符参数(`upload_id`)
- UPLOAD_FILE_OPT_BIT_SPECIFIC_LEN: 使用指定长度的方式上传. 使用此选项需要填写上传长度参数(`upload_len`). 否则无需填写 `upload_len` , 将上传整个文件

```
typedef struct {
  http2_upload_id_received_cb_t  upload_id_received_cb;
  http2_upload_completed_cb_t    upload_completed_cb;
} http2_upload_result_cb_t;
```

- upload_id_received_cb: 接收到云端服务器返回的上传标示符时, 将调用此回调函数
- upload_completed_cb: 文件上传结束时, 将调用此回调函数, result参数指示了上传结果

HTTP2断开连接

接口原型

```
int IOT_HTTP2_UploadFile_Disconnect(void *handle);
```

接口说明

断开参数handle指定的HTTP2连接

参数说明

参数	数据类型	方向	说明
http2_handle	void *	输入	调用 `IOT_HTTP2_UploadFile_Connect` 建连成功后返回的句柄

返回值说明

值	说明
0	函数调用成功
< 0	函数调用失败

需要对接的HAL接口

文件 `src/http2/http2_wrapper.h` 中包含了用户对接HTTP2文件上传需要适配的部分HAL接口

函数名	说明
HAL_SSL_Destroy	销毁一个TLS连接, 用于MQTT功能, HTTPS功能
HAL_SSL_Establish	建立一个TLS连接, 用于MQTT功能, HTTPS功能
HAL_SSL_Read	从一个TLS连接中读数据, 用于MQTT功能, HTTPS功能
HAL_SSL_Write	向一个TLS连接中写数据, 用于MQTT功能, HTTPS功能
HAL_MutexCreate	创建一个互斥量对象
HAL_MutexDestroy	销毁一个互斥量对象
HAL_MutexLock	锁住一个互斥量
HAL_MutexUnlock	解锁一个互斥量
HAL_SemaphoreCreate	创建信号量
HAL_SemaphoreDestroy	销毁信号量
HAL_SemaphorePost	post信号量
HAL_SemaphoreWait	等待信号量
HAL_ThreadCreate	创建线程
HAL_ThreadDelete	销毁线程
HAL_ThreadDetach	分离线程
HAL_Fopen	打开文件
HAL_Fread	读取文件数据
HAL_Fwrite	向文件写入数据
HAL_Fseek	设置文件指针stream的位置
HAL_Fclose	关闭文件
HAL_Ftell	得到文件位置指针当前位置相对于文件首的偏移字节数
HAL_Printf	打印函数
HAL_SleepMs	睡眠函数
HAL_Malloc	内存分配
HAL_Free	内存释放