物联网通信

设备端接入手册

产品文档

腾讯云

文档目录

设备端接入手册
设备接入概述

最近更新时间：2024-12-27 16:55:24

功能概述

为方便用户设备的接入，保障设备的接入安全，物联网通信提供了完善的设备接入服务。设备接入物联网通信需先完成 设备注册/创建，设备注册/创建成功之后才可再基于物联网通信的接入流程完成接入。

设备接入服务

设备接入服务提供设备动态注册的功能，使设备注册可基于设备自身来完成。
设备接入服务支持多样化的协议接入，支持设备基于 MQTT 协议，WebSocket 协议，HTTP/HTTPS 协议，CoAP 协议完成接入。
设备接入服务提供接入认证服务，设备需基于接入的协议完成设备接入认证，保障接入的安全性。
设备接入服务提供设备端 SDK 服务，设备可基于 SDK 完成接入。

设备基于 SDK 接入

物联网通信提供 C SDK，Andriod SDK，Java SDK 等 SDK 接入。SDK 中集成设备接入服务中所包含的功能，用户只需设置 SDK 中的设备信息（密钥设备：ProductID、DeviceName、设备密钥；证书设备：ProductID、DeviceName、证书文件、密钥文件、CA 证书），并将 SDK 相应的功能集成到设备上，即可完成设备的接入。除接入服务功能之外，SDK 还包含设备影子、OTA、RRPC 等功能接口。接口详情可参考以下**基于 SDK 接入中 SDK 的使用说明**相关文档。
C SDK 使用说明。
Android SDK 使用说明。
Java SDK 使用说明。
说明：
物联网通信支持自定义接入，设备只需按照平台提供的协议及认证流程，即可完成设备的自定义接入。

基于C SDK接入

C SDK 下载

最近更新时间：2024-12-27 16:55:24

代码托管

自 v1.0.0 版本开始，设备端 SDK 代码使用 Github 托管。

下载最新版

版本 v3.2.1

发布日期：2020/08/04

开发语言：C语言

开发环境：Linux/Windows

内容：

1、新增 rrpc 同步通信功能及示例。

2、新增广播功能及示例。

3、网关增加绑定/解绑子设备接口。

4、更新文档。

版本 v3.2.0

发布日期：2020/04/30

开发语言：C语言

开发环境：Linux/Windows

内容：

1、合并 mtmc 分支代码，支持多设备连接，优化多线程接口。

2、修复一些潜在的内存泄漏及越界问题，以及跨平台编译运行问题。

3、使用 clang-format 格式化代码，引入代码检查工具 clang-tidy 及 cpplint。

版本 v3.1.3

发布日期：2020/03/06

开发语言：C语言

开发环境：Linux/Windows

内容：

1、优化 ota_mqtt_sample，将 OTA 流程以及需要文件操作的地方解耦分离，并且 sample 支持 MQTT 断开重连之后仍然可以恢复下载。

2、优化 gateway_sample，并增加代理一个以上子设备示例代码。

3、增加查询 MQTT 主题是否订阅成功的接口。

4、优化及更新文档。
5、修复一些编译警告及 bug。
6、统一代码缩进风格。

版本 v3.1.2

发布日期：2019/11/11
开发语言：C语言
开发环境：Linux/Windows
内容：
1、移除对 IoT Explorer 平台相关代码及文档，仅支持 IoT Hub。优化文档描述。
2、Bug 修复：OTA 模块内存泄漏问题，device_info.json 文件解析问题及 Windows 平台时间格式问题。
3、避免文件名冲突，ca.c/h 重命名为 qcloud_iot_ca.c/h，device.c/h 重命名为 qcloud_iot_device.c/h。

版本 v3.1.0

发布日期：2019/09/19
开发语言：C语言
开发环境：Linux/Windows
内容：
C-SDK 重构：
1、优化代码架构及目录层级，采用英文注释，完善文档说明，提高可用性和可移植性。
2、在原 makefile 编译基础上增加 cmake 编译方式和代码抽取方式，适应多种编译环境。
3、增加 Windows 平台支持，支持在 Microsoft Visual Studio 下开发。
4、增加 AT_socket 网络层以支持 MCU+TCP AT 模组设备的开发移植。
5、增加 FreeRTOS+lwIP 平台的移植适配。

版本 v3.0.3

发布日期：2019/08/26
开发语言：C语言
开发环境：Linux，GNU Make。
内容：
1、支持 OTA 断点续传：ota_mqtt_sample.c 示例增加本地固件版本信息管理（版本、断点、MD5），固件下载建立 HTTPS 连接时支持 range 参数。
2、SDK 版本号更新为 v3.0.3。

版本 v3.0.2

发布日期：2019/07/18
开发语言：C语言
开发环境：Linux，GNU Make。

内容：

1、数据模板字符串类型支持转义字符处理。

2、设备影子去除设备侧 version 管理。

3、优化数据模板相关示例。

版本 v3.0.1

发布日期：2019/06/11

开发语言：C语言

开发环境：Linux，GNU Make。

内容：

1、日志上报功能优化，动态分配缓冲区内存，支持较大日志分段上报，适合多种使用场景。

2、MQTT 增加 subscribe 的 event handler 回调，及时通知订阅 topic 的状态变化。

3、修复一些代码问题，例如对 MQTT API 的返回值判断不当问题。

版本 v3.0.0

发布日期：2019/05/17

开发语言：C语言

开发环境：Linux，GNU Make。

内容

1、基于影子增加数据模板功能。

2、增加事件上报功能。

3、增加数据模板代码生成脚本工具。

4、修复 JSON 处理的若干 bug。

5、新增数据模板示例、事件示例、数据模板智能灯场景示例。

6、调整文档结构，增加文档目录 docs 及平台 SDK 应用说明。

7、版本 v3.0.0 及以后版本同时支持物联网通信及物联网开发两个物联网平台。

版本 v2.3.5

发布日期：2019/05/15

开发语言：C语言

开发环境：Linux，GNU Make。

内容:

1、增加设备动态注册功能。

2、增加设备动态注册示例。

3、增加设备信息读写 HAL 接口。

4、增加 AES 加解密 API。

5、修改各 Sample 设备信息获取方式为 HAL 层接口实现。

版本 v2.3.3

发布日期：2019/05/06
开发语言：C语言
开发环境：Linux，GNU Make。
内容:
1、优化 MQTT keep alive 连接机制及 PING request 发包策略。
2、修改 MQTT 订阅/取消订阅的 topic name 使用动态内存方式存储，方便接口调用者使用。
3、修改 topic name 最大长度为128，与云端后台保持一致。
4、修复 HTTPC 以及 MQTT 获取 sys 及 log 消息的 bug。
5、优化错误码类型。

版本 v2.3.2

发布日期：2019/04/12
开发语言：C语言
开发环境：Linux，GNU Make。
内容：
1、修复体验问题：在 make.settings 里增加网关编译选项（默认关闭）以及修改固件升级打印级别。
2、修复 MQTT 接收缓冲区在影子消息下行时容易丢失问题：增加接收缓冲区不足时的错误提示，更改 MQTT 发送及接收缓冲区默认大小为2048字节。
3、修改成功订阅主题的最大个数为10条。

版本 v2.3.1

发布日期：2019/03/12
开发语言：C 语言
开发环境：Linux，GNU Make。
内容：
1、SDK 增加设备端日志上报功能，方便用户通过云端控制台远程监控及诊断设备联网状况。目前仅支持 MQTT 模式。
2、精简 SDK 日志打印内容，修复若干 Bug，优化部分代码设计。
3、修改设备名称最大长度为48位，与 IoT Hub 云端控制台保持一致。

版本 v2.3.0

发布日期：2019/02/25
开发语言：C 语言
开发环境：Linux，GNU Make。
内容：
1、增加网关功能，支持网关设备基于 MQTT 协议代理子设备上下线及收发消息。
2、针对多线程应用，优化线程安全设计，在 samples 中增加多线程例程及注意事项说明。

3、优化 MQTT 重连机制及心跳包定时器刷新策略。
4、若干 Bug 的修复，部分内存操作增加合法性检查。
5、去除若干结构体采用位域操作方式，减少跨平台错误。

版本 v2.2.0

发布日期：2018/07/20
开发语言：C 语言
开发环境：Linux，GNU Make。
内容：
1、新增 NBIoT 设备接入能力。
2、适配 TOPIC 的通配符 “#” 和 “+”。
3、整理第三方库的目录结构。
4、若干 bug 的修复。

版本 v2.1.0

发布日期：2018/05/02
开发语言：C 语言
开发环境：Linux，GNU Make。
内容：
1、新增固件升级（OTA-CoAP 通道）能力。
2、新增低端资源受限设备 hmac-sha1 鉴权接入能力。
3、新增获取后台时间的能力。

版本 v2.0.0

发布日期：2018/03/12
开发语言：C 语言
开发环境：Linux，GNU Make。
内容：
1、新增固件升级（OTA-MQTT 通道）能力。
2、修复设备影子心跳间隔失效的问题。
3、修复 MQTT 接收的数据长度在临界值时导致缓冲区溢出的问题。

版本 v1.2.2

发布日期：2018/02/07
开发语言：C 语言
开发环境：Linux，GNU Make。
内容：
1、新增 MQTT/CoAP 对称加密连接支持。

2、Linux c 编译优化。

版本 v1.2.1

发布日期：2018/02/02

开发语言：C 语言

开发环境：Linux，GNU Make。

内容：修复 Publish 消息超时回调的错误逻辑。

版本 v1.2.0

发布日期：2018/1/17

开发语言：C 语言

开发环境：Linux，GNU Make。

内容：

1、改造发布/订阅消息的 ACK 通过回调接收，不会阻塞发送线程。

2、增加终端与后台关于连接、日志对应的能力。

3、新增 CoAP 通道，基于 UDP，采用 DTLS 非对称加密，在纯上报数据场景耗能更少。

版本 v1.0.0

发布日期：2017/11/15

开发语言：C 语言

开发环境：Linux，GNU Make。

内容：

1、MQTT 协议支持：支持设备快捷轻便的链接 IoT Hub 云端服务器，可查看 MQTT 协议详解。

2、设备影子功能支持：具体可查看 设备影子详情。

3、提供对称和非对称两种加密方式支持。

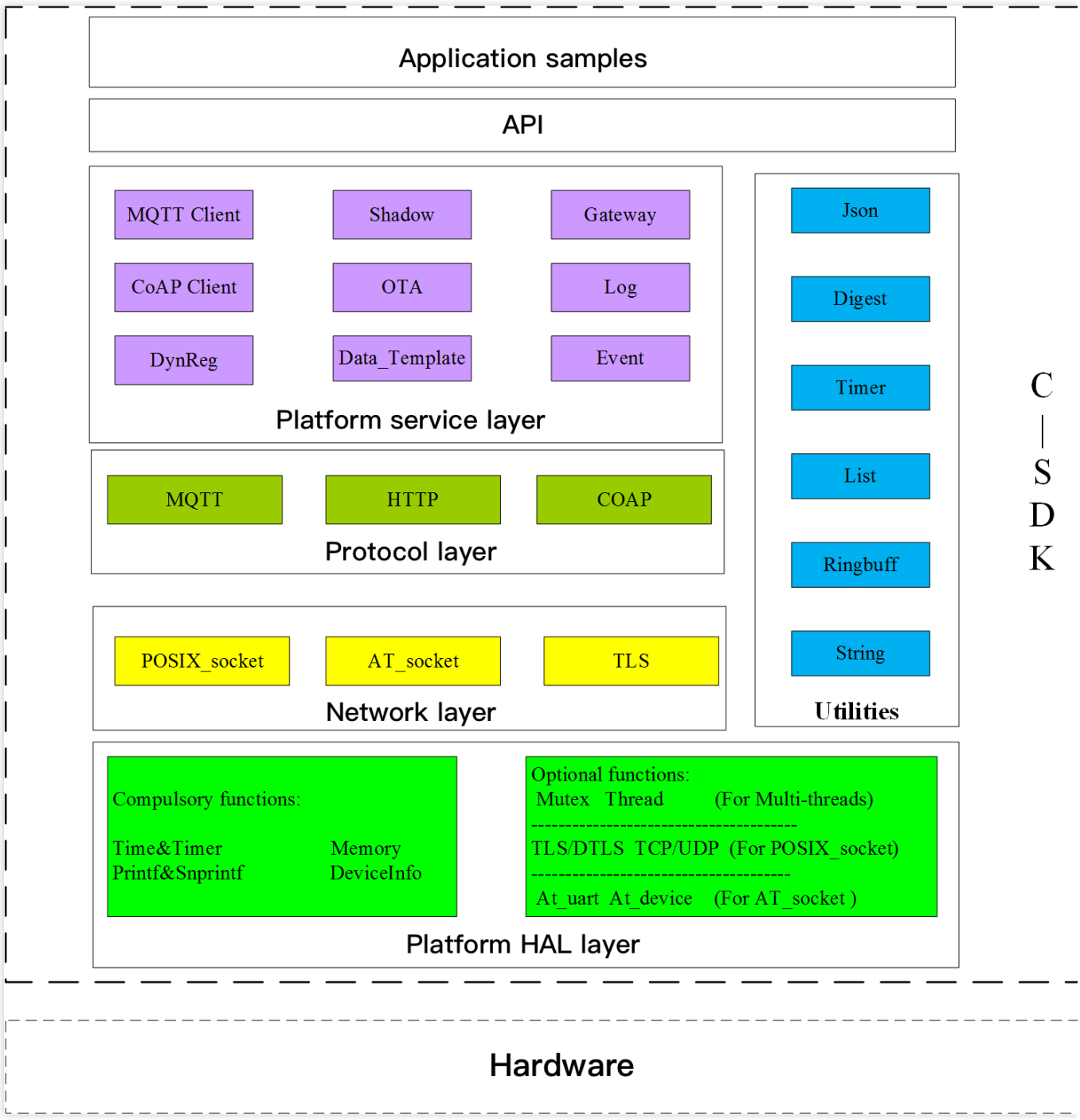
C SDK 跨平台移植
C SDK_Porting 跨平台移植概述

最近更新时间：2024-12-27 16:55:24

本文档介绍如何将设备端 C-SDK 移植到目标硬件平台。C-SDK 采用模块化设计，分离核心协议服务与硬件抽象层，在进行跨平台移植时，一般只需对硬件抽象层进行修改适配即可。

C-SDK 架构

架构图



架构说明

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。

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

实现基于 TLS/SSL（TLS/DTLS）方式，POSIX_socket（TCP/UDP）方式和 AT_socket 方式的网络协议栈，不同服务可根据需要使用不同的协议栈接口函数。

硬件抽象层

实现对不同硬件平台的底层操作的抽象封装，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。

硬件抽象层移植

HAL 层主要有几大块的移植，分别是 OS 相关的、网络及 TLS 相关的、时间及打印相关的、设备信息相关的。SDK 在 **platform/os** 目录示例了 Linux、Windows、FreeRTOS 及 nonOS 四个场景的硬件抽象层实现，可以参考最相近的目录展开目标平台的移植。

OS 相关接口

序号	函数名	说明
1	HAL_Malloc	动态申请内存块
2	HAL_Free	释放内存块
3	HAL_ThreadCreate	线程创建
4	HAL_ThreadDestroy	线程销毁
5	HAL_MutexCreate	创建互斥锁
6	HAL_MutexDestroy	销毁互斥锁
7	HAL_MutexLock	mutex 加锁
8	HAL_MutexUnlock	mutex 解锁
9	HAL_SemaphoreCreate	创建信号量
10	HAL_SemaphoreDestroy	销毁信号量
11	HAL_SemaphoreWait	等待信号量
12	HAL_SemaphorePost	释放信号量
13	HAL_SleepMs	休眠

网络及 TLS 相关的 HAL 接口

网络相关接口提供二选一的适配移植。对于具备网络通讯能力并且本身集成 TCP/IP 网络协议栈的设备，需要实现 POSIX_socket 的网络 HAL 接口，使用 TLS/SSL 加密通讯的还需要实现 TLS 相关的 HAL 接口。而对于 **MCU+ 通用 TCP_AT 模组** 的设备，则可以选择 SDK 提供的 AT_Socket 框架，并实现相关的 AT 模组接口。

基于 POSIX_socket 的 HAL 接口

其中 TCP/UDP 相关接口基于 POSIX socket 函数实现。TLS 相关接口依赖于 **mbedtls** 库，移植之前必须确保系统上有可用的 **mbedtls** 库。如果采用其他 TLS/SSL 库，可参考 **platform/tls/mbedtls** 相关实现进行移植适配。

UDP/DTLS 相关的函数仅在使能 **COAP** 通讯的时候才需要移植。

序号	函数名	说明
1	HAL_TCP_Connect	建立 TCP 连接
2	HAL_TCP_Disconnect	断开 TCP 连接
3	HAL_TCP_Write	TCP 写
4	HAL_TCP_Read	TCP 读
5	HAL_TLS_Connect	建立 TLS 连接
6	HAL_TLS_Disconnect	断开 TLS 连接
7	HAL_TLS_Write	TLS 写
8	HAL_TLS_Read	TLS 读
9	HAL_UDP_Connect	建立 UDP 连接
10	HAL_UDP_Disconnect	断开 UDP 连接
11	HAL_UDP_Write	UDP 写
12	HAL_UDP_Read	UDP 读
13	HAL_DTLS_Connect	建立 DTLS 连接
14	HAL_DTLS_Disconnect	断开 DTLS 连接
15	HAL_DTLS_Write	DTLS 写
16	HAL_DTLS_Read	DTLS 读

基于 AT_socket 的 HAL 接口

通过使能编译宏 **AT_TCP_ENABLED** 选择 AT_socket，则 SDK 会调用 `network_at_tcp.c` 的 `at_socket` 接口， at_socket 层不需要移植，需要实现 AT 串口驱动及 AT 模组驱动，AT 模组驱动只需要实现 AT 框架中 at_device 的驱动结构体 `at_device_op_t` 的驱动接口即可，可以参照 at_device 目录下的已支持的模组。AT 串

口驱动需要实现串口的中断接收，然后在中断服务程序中调用回调函数 `at_client_uart_rx_isr_cb` 即可，可以参考 `HAL_AT_UART_freertos.c` 实现目标平台的移植。

序号	函数名	说明
1	HAL_AT_Uart_Init	初始化 AT 串口
2	HAL_AT_Uart_Deinit	去初始化 AT 串口
3	HAL_AT_Uart_Send	AT 串口发送数据
4	HAL_AT_UART_IRQHandler	AT 串口接收中断服务程序

时间及打印相关的 HAL 接口

序号	函数名	说明
1	HAL_Printf	将格式化的数据写入标准输出流中
2	HAL_Snprintf	将格式化的数据写入字符串
3	HAL_UptimeMs	检索自系统启动以来已运行的毫秒数
4	HAL_DelayMs	阻塞延时，单位毫秒

设备信息相关的 HAL 接口

接入 IoT 平台需要在平台创建产品和设备信息，同时需要将产品及设备信息保存在设备侧的非易失存储介质。可以参考 `platform/os/linux/HAL_Device_linux.c` 示例实现。

序号	函数名	说明
1	HAL_GetDevInfo	设备信息读取
2	HAL_SetDevInfo	设备信息保存

FreeRTOS+lwIP 平台移植说明

最近更新时间：2024-12-27 16:55:24

本文档介绍如何将腾讯云物联 C-SDK 移植到 **FreeRTOS+lwIP** 平台。

FreeRTOS 移植简介

FreeRTOS 作为一个微内核系统，主要提供任务创建及调度和任务间通信等 OS 核心机制，在不同设备平台还需要搭配多个软件组件包括 C 运行库（例如 newlib 或者 ARM CMSIS 库）和 TCP/IP 网络协议栈（如 lwIP）才能形成完整的嵌入式运行平台。同时各个设备平台的编译开发环境也各不相同，因此在移植 C-SDK 时，需要根据不同设备的具体情况进行适配。

说明：

SDK 在 **platform/os/freertos** 里面提供了一个基于 **FreeRTOS+lwIP+newlib** 的参考实现，该实现已经在乐鑫 ESP8266 平台上验证测试过。

抽取代码

因为基于 RTOS 系统的平台编译方式各不相同，一般无法直接使用 SDK 的 cmake 或者 make 编译，因此 SDK 提供了代码抽取功能，可根据需要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。

1. 修改 CMakeLists.txt 中配置为 freertos 平台，并开启代码抽取功能：

```
set(BUILD_TYPE                    "release")
set(PLATFORM                      "freertos")
set(EXTRACT_SRC ON)
set(FEATURE_AT_TCP_ENABLED OFF)
```

2. 在 Linux 环境运行以下命令：

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

3. 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
```

说明：

include 目录：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。API 具体介绍请参考 接口及可变参数说明。

platform目录：平台相关的代码，可根据设备的具体情况进行修改适配。具体的函数说明请参考 C-SDK_Porting跨平台移植概述。

sdk_src：SDK 的核心逻辑及协议相关代码，一般不需要修改，其中 internal_inc 为 SDK 内部使用的头文件。

4. 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。

移植示例

在 Linux 开发环境基于乐鑫 ESP8266 RTOS 平台搭建一个工程示例。

1. 请参考 ESP8266_RTOS_SDK 获取 RTOS_SDK 和交叉编译器，并创建一个项目工程。

2. 将上面抽取的 qcloud_iot_c_sdk 目录，拷贝到 components/qcloud_iot 下。

3. 在 components/qcloud_iot 下，新建一个编译配置文件 component.mk，内容如下：

```
#
# Component Makefile
#
COMPONENT_ADD_INCLUDEDIRS := \\
qcloud_iot_c_sdk/include \\
qcloud_iot_c_sdk/include/exports \\
qcloud_iot_c_sdk/sdk_src/internal_inc
COMPONENT_SRCDIRS := \\
qcloud_iot_c_sdk/sdk_src \\
qcloud_iot_c_sdk/platform
```

至此，您可以将 qcloud_iot_c_sdk 作为一个组件进行编译了，之后在用户代码里面就可以调用物联 C-SDK 的接口进行连接和收发消息。

MCU+通用 TCP_AT 模组移植（FreeRTOS）

最近更新时间：2024-12-27 16:55:24

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须移植。本文阐述针对 MCU（FreeRTOS）+通用 TCP AT 模组的目标环境，如何移植 C-SDK 并接入腾讯云物联网平台。

SDK 下载

下载最新版本设备端 C-SDK。

SDK 功能配置

使用通用 TCP 模组编译配置选项配置如下：

名称	配置	说明
BUILD_TYPE	debug/release	根据需要设置
EXTRACT_SRC	ON	使能代码抽取
COMPILE_TOOLS	gcc/MSVC	根据需要设置，IDE 情况不关注
PLATFORM	Linux/Windows	根据需要设置，IDE 情况不关注
FEATURE_OTA_COMM_ENABLED	ON/OFF	根据需要设置
FEATURE_AUTH_MODE	KEY	资源受限设备认证方式建议选密钥认证
FEATURE_AUTH_WITH_NOTLS	ON/OFF	根据需要是否使能 TLS
FEATURE_EVENT_POST_ENABLED	ON/OFF	根据需要是否使能事件上报
FEATURE_AT_TCP_ENABLED	ON	AT 模组 TCP 功能开关
FEATURE_AT_UART_RECV_IRQ	ON	AT 模组中断接受功能开关
FEATURE_AT_OS_USED	ON	AT 模组多线程功能开关
FEATURE_AT_DEBUG	OFF	默认关闭 AT 模组调试功能，有调试需要再打开

代码抽取

1. 在 Linux 环境运行以下命令：

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

2. 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
```

说明：

include 目录：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：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 ：SDK 的核心逻辑及协议相关代码，一般不需要修改，其中 internal_inc 为 SDK 内部使用的头文件。

3. 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。

HAL 层移植

请先参考 C-SDK_Porting 跨平台移植概述 进行移植。

对于网络相关的 HAL 接口，通过上面的编译选项已选择 SDK 提供的 `AT_Socket` 框架，SDK 会调用 `network_at_tcp.c` 的 `at_socket` 接口， `at_socket` 层不需要移植，需要实现 AT 串口驱动及AT模组驱动，AT模组驱动只需要实现 AT 框架中 `at_device` 的驱动结构体 `at_device_op_t` 的驱动接口即可，可以参照 `at_device` 目录下的已支持的模组。

目前 SDK 针对物联网使用较广的 Wi-Fi 模组 ESP8266 提供了底层接口实现，供移植到其他通讯模组时作为参考。

业务逻辑开发

您可参考 SDK samples 目录下的例程进行开发。

MCU+通用 TCP_AT 模组移植（nonOS）

最近更新时间：2024-12-27 16:55:24

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须移植。本文阐述针对 MCU（无 OS）+通用 TCP AT 模组的目标环境，如何移植 C-SDK 并接入腾讯云物联网平台。
相较于有 RTOS 场景，at_socket 网络接收数据的处理会有差异，应用层需要周期性的调用 **IOT_MQTT_Yield** 来接收服务端下行数据，错过接收窗口则会存在数据丢失的情况，所以在业务逻辑较为复杂的场景建议使用 RTOS，通过配置 FEATURE_AT_OS_USED = OFF 选择无 OS 方式。

SDK 下载

下载最新版本设备端 C-SDK。

SDK 功能配置

无 RTOS 使用通用 TCP 模组编译配置选项配置如下：

名称	配置	说明
BUILD_TYPE	debug/release	根据需要设置
EXTRACT_SRC	ON	使能代码抽取
COMPILE_TOOLS	gcc/MSVC	根据需要设置，IDE 情况不关注
PLATFORM	Linux/Windows	根据需要设置，IDE 情况不关注
FEATURE_OTA_COMM_ENABLED	ON/OFF	根据需要设置
FEATURE_AUTH_MODE	KEY	资源受限设备认证方式建议选密钥认证
FEATURE_AUTH_WITH_NOTLS	ON/OFF	根据需要是否使能 TLS
FEATURE_EVENT_POST_ENABLED	ON/OFF	根据需要是否使能事件上报
FEATURE_AT_TCP_ENABLED	ON	使能 at_socket 组件
FEATURE_AT_UART_RECV_IRQ	ON	使能 AT 串口中断接收
FEATURE_AT_OS_USED	OFF	at_socket 组件无 RTOS 环境使用

FEATURE_AT_DEBUG	OFF	默认关闭 AT 模组调试功能，有调试需要再打开

代码抽取

1. 在 Linux 环境运行以下命令：

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

2. 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
```

说明：

include 目录：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：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 ：SDK 的核心逻辑及协议相关代码，一般不需要修改，其中 internal_inc 为 SDK 内部使用的头文件。

3. 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。

HAL 层移植

请先参考 C-SDK_Porting 跨平台移植概述。

对于网络相关的 HAL 接口，通过上面的编译选项已选择 SDK 提供的 AT_Socket 框架，SDK 会调用 `network_at_tcp.c` 的 `at_socket` 接口， `at_socket` 层不需要移植，需要实现 AT 串口驱动及 AT 模组驱动，AT 模组驱动只需要实现 AT 框架中 `at_device` 的驱动结构体 `at_device_op_t` 的驱动接口即可，可以参照 `at_device` 目录下的已支持的模组。AT 串口驱动需要实现串口的中断接收，然后在中断服务程序中调用回调函数 `at_client_uart_rx_isr_cb` 即可，可以参考 `HAL_OS_nonos.c` 实现目标平台的移植。

业务逻辑开发

您可参考 SDK samples 目录下的例程进行开发。

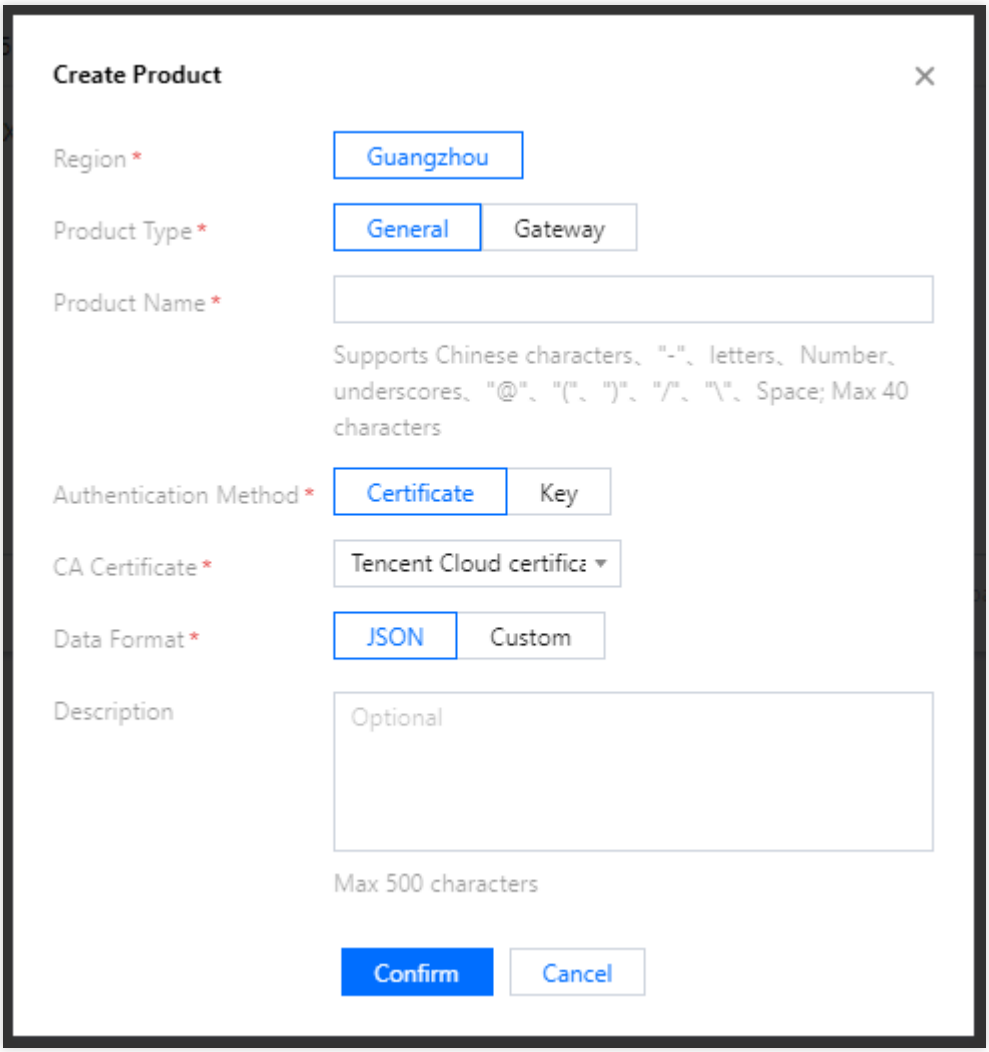
C SDK 接入说明

最近更新时间：2024-12-27 16:55:24

为保证安全，物联网平台会验证每个接入设备的合法性，为此提供了多种认证方式，满足不同的使用场景及各种资源的设备接入。

设备身份信息

从设备密钥形式区分，分为证书设备和密钥设备。证书方式安全性更高，但需要消耗较多的软硬件资源。
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。
设备密钥是创建产品时，通过设置认证方式来确定，如下图所示：



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区别。

预置烧录

创建产品后，在物联网通信 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：产品创建后使用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，然后使用申请到的三元组或者四元组信息进行设备认证。

动态烧录设备名（DeviceName）的生成：若使用动态注册的同时使用自动设备创建，则设备名是可以设备自己生成的，但必须保证同一产品 ID（ProductId）下的设备名不重复，一般取 IMEI 或者 MAC 地址。若使用动态注册的同时不使用自动设备创建，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，否则会产生较大的安全隐患。

预置烧录设备认证编程

设备信息写入

证书设备实现如下 HAL API：

HAL_API	说明
HAL_SetProductID	设置产品 ID，必须存放在非易失性存储介质
HAL_SetDevName	设置设备名，必须存放在非易失性存储介质
HAL_SetDevCertName	设置设备证书文件名，证书文件需要放置到 certs 文件目录
HAL_SetDevPrivateKeyName	设置设备证书私钥文件名，私钥文件需要放置到 certs 文件目录

密钥设备实现如下 HAL API：

HAL_API	说明
HAL_SetProductID	设置产品 ID，必须存放在非易失性存储介质
HAL_SetDevName	设置设备名，必须存放在非易失性存储介质
HAL_SetDevSec	设置设备密钥，必须存放在非易失性存储介质，建议加密加扰

设备信息获取

证书设备实现如下 HAL API：

HAL_API	说明
HAL_GetProductID	获取产品 ID
HAL_GetDevName	获取设备名
HAL_GetDevCertName	获取设备证书文件名
HAL_GetDevPrivateKeyName	获取设备证书私钥文件名

密钥设备实现如下 HAL API：

HAL_API	说明
HAL_GetProductID	获取产品 ID
HAL_GetDevName	获取设备名
HAL_GetDevSec	获取设备密钥，若写入时加密加扰，读取时需解密解扰

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_name;
    initParams->product_id = sg_devInfo.product_id;
     ......
}
```

设备信息获取

```
int HAL_GetDevInfo(void *pdevInfo)
{
    int ret;
    DeviceInfo *devInfo = (DeviceInfo *)pdevInfo;

    memset((char *)devInfo, 0, sizeof(DeviceInfo));
    ret = HAL_GetProductID(devInfo->product_id, MAX_SIZE_OF_PRODUCT_ID);
    ret |= HAL_GetDevName(devInfo->device_name, MAX_SIZE_OF_DEVICE_NAME);

#ifdef  AUTH_MODE_CERT
    ret |= HAL_GetDevCertName(devInfo->devCertFileName,
MAX_SIZE_OF_DEVICE_CERT_FILE_NAME);
    ret |= HAL_GetDevPrivateKeyName(devInfo->devPrivateKeyFileName,
MAX_SIZE_OF_DEVICE_KEY_FILE_NAME);
#else
    ret |= HAL_GetDevSec(devInfo->devSerc, MAX_SIZE_OF_DEVICE_SERC);
```

```
#endif

    if(QCLOUD_ERR_SUCCESS != ret){
        Log_e("Get device info err");
        ret = QCLOUD_ERR_DEV_INFO;
    }

    return ret;
}
```

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
            ......
}
```

动态烧录设备认证编程

判断是否发起动态申请

```
int main(int argc, char **argv) {
            ......
    memset((char *)&sDevInfo, 0, sizeof(DeviceInfo));
    ret = HAL_GetProductID(sDevInfo.product_id, MAX_SIZE_OF_PRODUCT_ID);
    ret |= HAL_GetProductKey(sDevInfo.product_key, MAX_SIZE_OF_PRODUCT_KEY);
```

```
    ret |= HAL_GetDevName(sDevInfo.device_name, MAX_SIZE_OF_DEVICE_NAME);  //动
态注册，建议用设备的唯一标识做设备名，譬如芯片ID、IMEI

#ifdef  AUTH_MODE_CERT
    ret |= HAL_GetDevCertName(sDevInfo.devCertFileName,
MAX_SIZE_OF_DEVICE_CERT_FILE_NAME);
    ret |= HAL_GetDevPrivateKeyName(sDevInfo.devPrivateKeyFileName,
MAX_SIZE_OF_DEVICE_KEY_FILE_NAME);
    if(QCLOUD_ERR_SUCCESS != ret){
        Log_e("Get device info err");
        return QCLOUD_ERR_FAILURE;
    }
    /*用户需要根据自己的产品情况修改设备信息为空的逻辑，此处仅为示例*/
    if(!strcmp(sDevInfo.devCertFileName, QCLOUD_IOT_NULL_CERT_FILENAME)
        ||!strcmp(sDevInfo.devPrivateKeyFileName,
QCLOUD_IOT_NULL_KEY_FILENAME)){
        Log_d("dev Cert not exist!");
        infoNullFlag = true;
    }else{
        Log_d("dev Cert exist");
    }
#else
    ret |= HAL_GetDevSec(sDevInfo.devSerc, MAX_SIZE_OF_PRODUCT_KEY);
    if(QCLOUD_ERR_SUCCESS != ret){
        Log_e("Get device info err");
        return QCLOUD_ERR_FAILURE;
    }
    /*用户需要根据自己的产品情况修改设备信息为空的逻辑，此处仅为示例*/
    if(!strcmp(sDevInfo.devSerc, QCLOUD_IOT_NULL_DEVICE_SECRET)){
        Log_d("dev psk not exist!");
        infoNullFlag = true;
    }else{
        Log_d("dev psk exist");
    }
#endif
            ......

}
```

发起动态申请，并保存申请的设备信息

```
    /*设备信息为空，发起设备注册 注意：成功注册并完成一次连接后则无法再次发起注册，请做好设备
信息的保存*/
    if(infoNullFlag){
        if(QCLOUD_ERR_SUCCESS == qcloud_iot_dyn_reg_dev(&sDevInfo)){

            ret = HAL_SetDevName(sDevInfo.device_name);
```

```
#ifdef  AUTH_MODE_CERT
            ret |= HAL_SetDevCertName(sDevInfo.devCertFileName);
            ret |= HAL_SetDevPrivateKeyName(sDevInfo.devPrivateKeyFileName);
#else
            ret |= HAL_SetDevSec(sDevInfo.devSerc);
#endif
            if(QCLOUD_ERR_SUCCESS != ret){
                Log_e("devices info save fail");
            }else{
#ifdef  AUTH_MODE_CERT
                Log_d("dynamic register success, productID: %s, devName: %s,
CertFile: %s, KeyFile: %s", \\
                        sDevInfo.product_id, sDevInfo.device_name,
sDevInfo.devCertFileName, sDevInfo.devPrivateKeyFileName);
#else
                Log_d("dynamic register success,productID: %s, devName: %s,
devSerc: %s", \\
                        sDevInfo.product_id, sDevInfo.device_name,
sDevInfo.devSerc);
#endif
            }
        }else{
            Log_e("%s dynamic register fail", sDevInfo.device_name);
        }
    }
```

设备信息动态申请成功后，即完成预置烧录的功能。后续的认证流程与预置烧录的流程一致。

C SDK 使用说明
使用概述

最近更新时间：2024-12-27 16:55:24

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。

说明：

在版本v3.1.0之后，SDK 对编译环境，代码及目录结构进行了重构优化，提高了可用性和可移植性。

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，则只要做简单适配（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须移植。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。
物联网通信平台快速体验设备端 MQTT 接入和收发消息，请参考 MQTT 快速入门。

注意事项

OTA 升级 API 变化

从 SDK 版本3.0.3开始，OTA 升级支持了断点续传，当固件下载过程中如果因为网络异常或其他原因被打断，可以将已经下载的固件部分保存，在下次恢复下载时候可以不用从零开始下载，而是从未下载的部分开始。
在支持这一新特性之后，OTA 相关 API 的使用方法发生了变化，对于从3.0.2及以前的版本升级的用户，需要修改用户逻辑代码，否则固件下载会失败，请参考 `samples/ota/ota_mqtt_sample.c` 进行修改。

代码命名变化

为了提高代码可读性，保证命名规范，SDK 3.1.0版本对部分变量、函数及宏命名进行了变更，对于从3.0.3及以前的版本升级的用户，可以在 Linux 环境下面执行 `tools/update_from_old_SDK.sh` 脚本，对自己的用户代码执行名字替换，替换完成就可以直接使用新版本的 SDK。

旧命名	新命名
QCLOUD_ERR_SUCCESS	QCLOUD_RET_SUCCESS
QCLOUD_ERR_MQTT_RECONNECTED	QCLOUD_RET_MQTT_RECONNECTED
QCLOUD_ERR_MQTT_MANUALLY_DISCONNECTED	QCLOUD_RET_MQTT_MANUALLY_DISC
QCLOUD_ERR_MQTT_CONNACK_CONNECTION_ACCEPTED	QCLOUD_RET_MQTT_CONNACK_CONN
QCLOUD_ERR_MQTT_ALREADY_CONNECTED	QCLOUD_RET_MQTT_ALREADY_CONN
MAX_SIZE_OF_DEVICE_SERC	MAX_SIZE_OF_DEVICE_SECRET
devCertFileName	dev_cert_file_name
devPrivateKeyFileName	dev_key_file_name
devSerc	device_secret
MAX_SIZE_OF_PRODUCT_KEY	MAX_SIZE_OF_PRODUCT_SECRET
product_key	product_secret。
DEBUG	eLOG_DEBUG
INFO	eLOG_INFO

WARN	eLOG_WARN
ERROR	eLOG_ERROR
DISABLE	eLOG_DISABLE
Log_writter	IOT_Log_Gen
qcloud_iot_dyn_reg_dev	IOT_DynReg_Device
IOT_SYSTEM_GET_TIME	IOT_Get_SysTime

编译配置说明

最近更新时间：2024-12-27 16:55:24

本文档对 C SDK 的编译方式和编译配置选项进行说明，并介绍了 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 方式与 SDK v3.0.3及之前的版本保持一致，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_MQTT_COMM_ENABLED	ON/OFF	MQTT 通道总开关。
FEATURE_MQTT_DEVICE_SHADOW	ON/OFF	设备影子总开关。
FEATURE_COAP_COMM_ENABLED	ON/OFF	CoAP 通道总开关。
FEATURE_GATEWAY_ENABLED	ON/OFF	网关功能总开关。
FEATURE_OTA_COMM_ENABLED	ON/OFF	OTA 固件升级总开关。
FEATURE_OTA_SIGNAL_CHANNEL	MQTT/COAP	OTA 信令通道类型。
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_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。

FEATURE_EVENT_POST_ENABLED	ON/OFF	事件上报开关。
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_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。
FEATURE_MULTITHREAD_TEST_ENABLED	ON/OFF	是否编译 Linux 多线程测试例程。

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_MQTT_DEVICE_SHADOW	FEATURE_MQTT_COMM_ENABLED	ON
FEATURE_GATEWAY_ENABLED	FEATURE_MQTT_COMM_ENABLED	ON
FEATURE_OTA_SIGNAL_CHANNEL(MQTT)	FEATURE_OTA_COMM_ENABLED FEATURE_MQTT_COMM_ENABLED	ON ON
FEATURE_OTA_SIGNAL_CHANNEL(COAP)	FEATURE_OTA_COMM_ENABLED FEATURE_COAP_COMM_ENABLED	ON ON
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（Linux 和 Windows）

最近更新时间：2024-12-27 16:55:24

Linux（Ubuntu）环境

说明：

本文演示使用 Ubuntu 的版本为16.04。

1. 必要软件安装

SDK 需要 cmake 版本在3.5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
```

2. 配置修改

修改 SDK 根目录下的 CMakeLists.txt 文件，并确保以下选项存在（以密钥认证设备为例）：

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

3. 执行脚本编译

3.1 完整编译库和示例如下：

```
./cmake_build.sh
```

3.2 输出的库文件，头文件及示例在 `output/release` 文件夹中。

在一次完整编译之后，若只需要编译示例，则执行以下代码：

```
./cmake_build.sh samples
```

3.3 填写设备信息

将在腾讯云物联网平台创建的设备的设备信息（以密钥认证设备为例），填写到 SDK 根目录下 device_info.json 中，示例代码如下：

```
"auth_mode":"KEY",
"productId":"S3EUVBQAZW",
"deviceName":"test_device",
"key_deviceinfo":{
```

```
  "deviceSecret":"vX6PQqazsGsMyf5SMfs6OA6y"
}
```

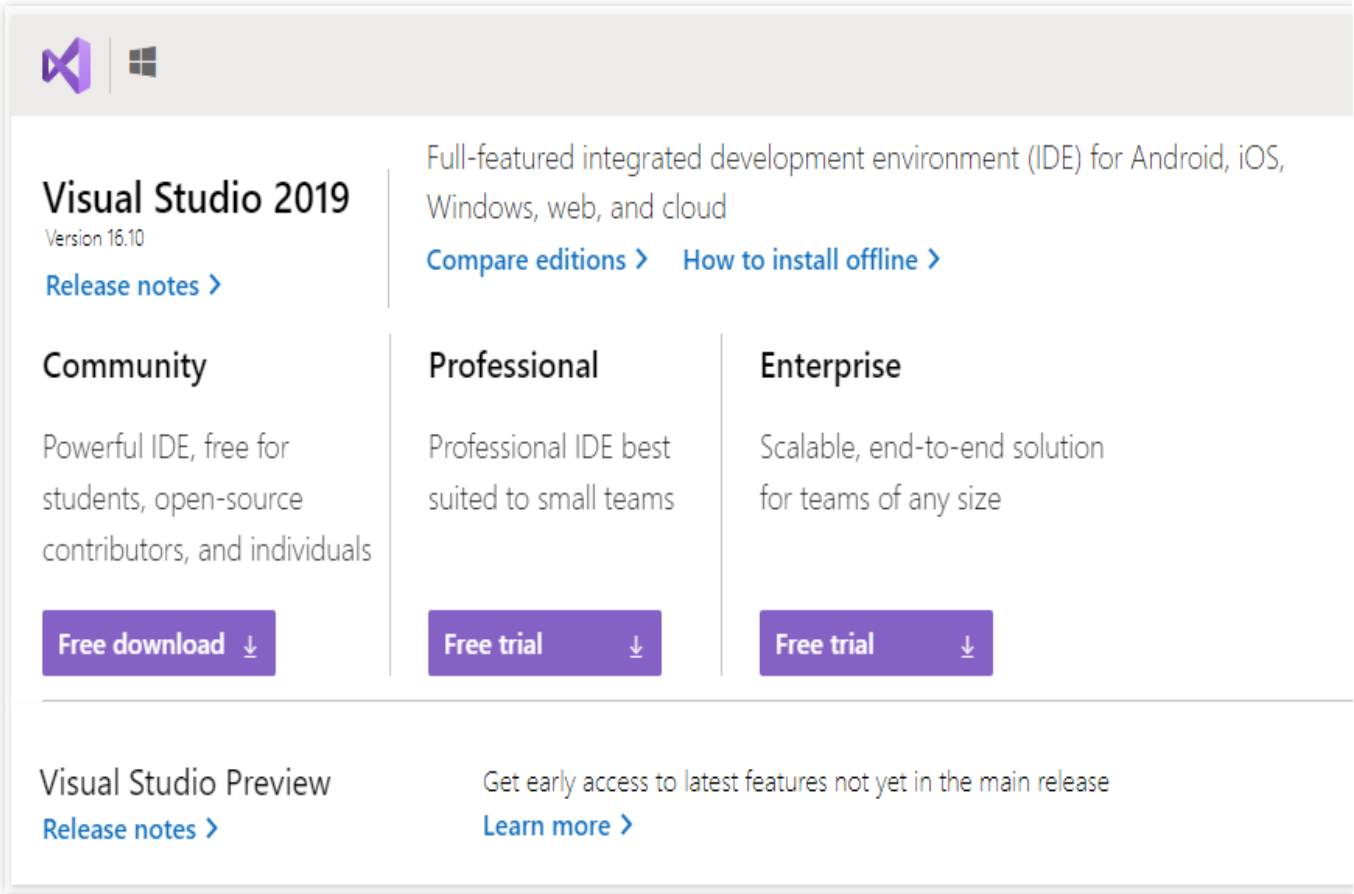
4. 运行示例

示例输出位于 `output/release/bin` 文件夹中，例如运行 data_template_sample 示例，输入 `./output/release/bin/data_template_sample` 即可。

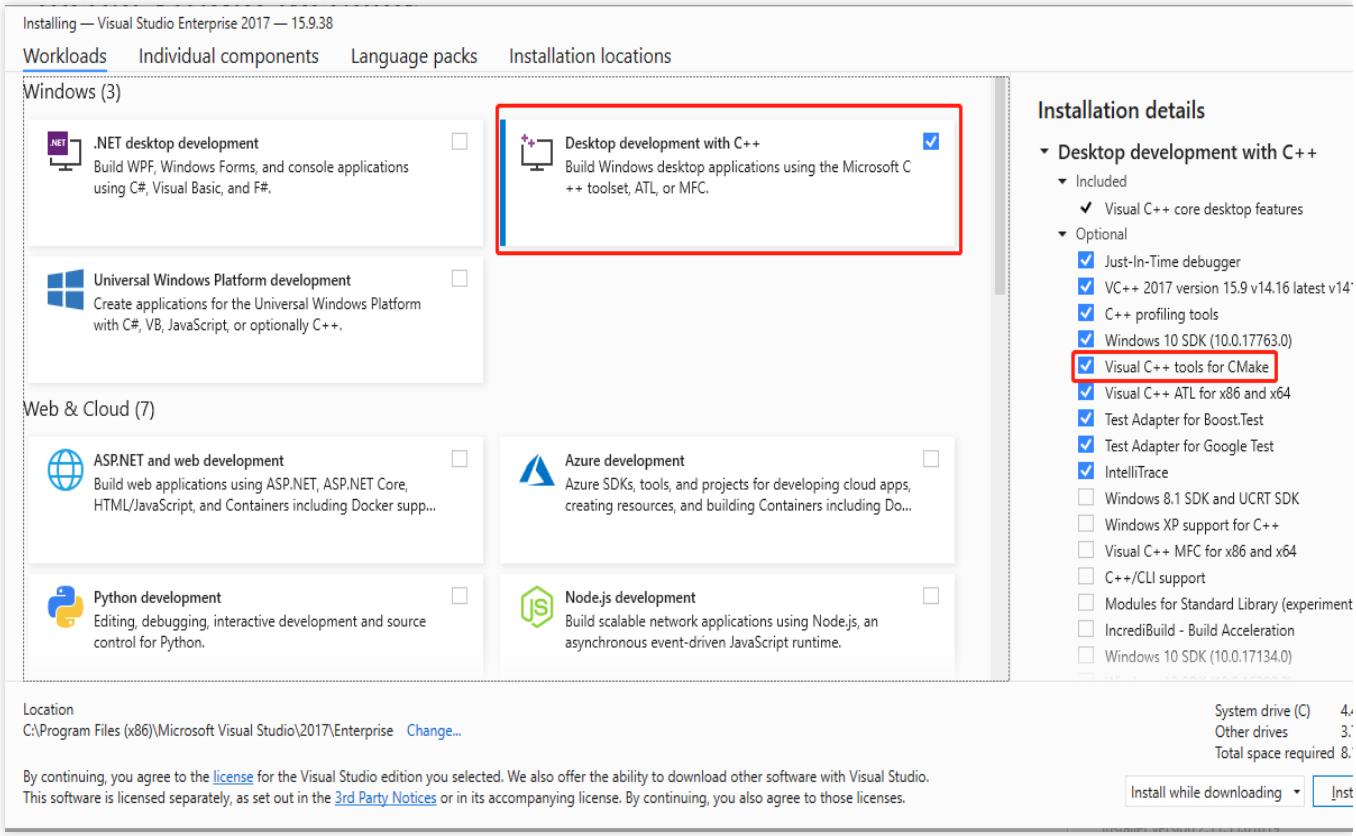
Windows 环境

获取和安装 Visual Studio 2019开发环境

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16.2版本 Community。

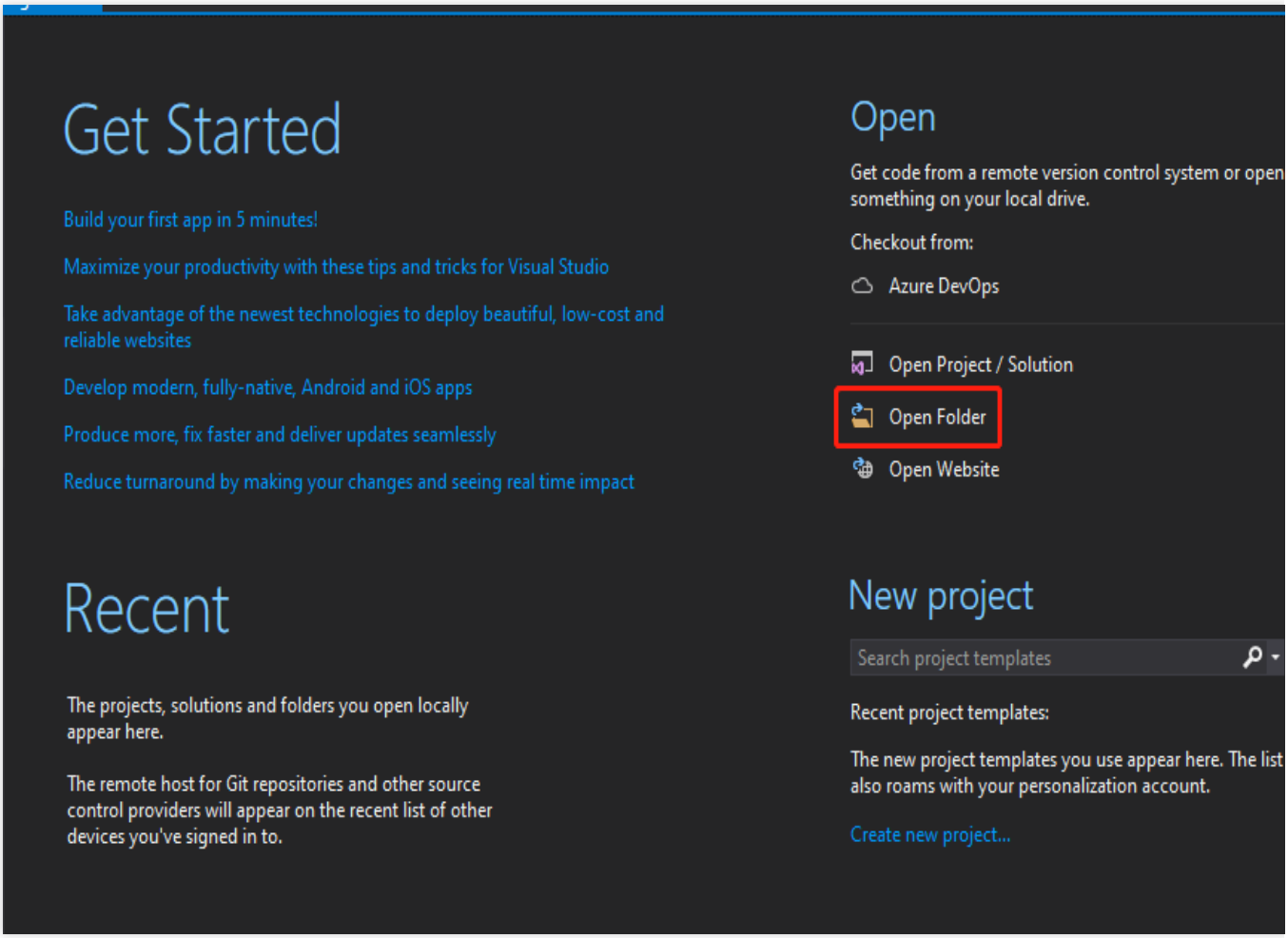


2. 选择【使用 C++ 的桌面开发】，并确保勾选【用于 Windows 的 C++ CMAKE 工具】。



编译并运行

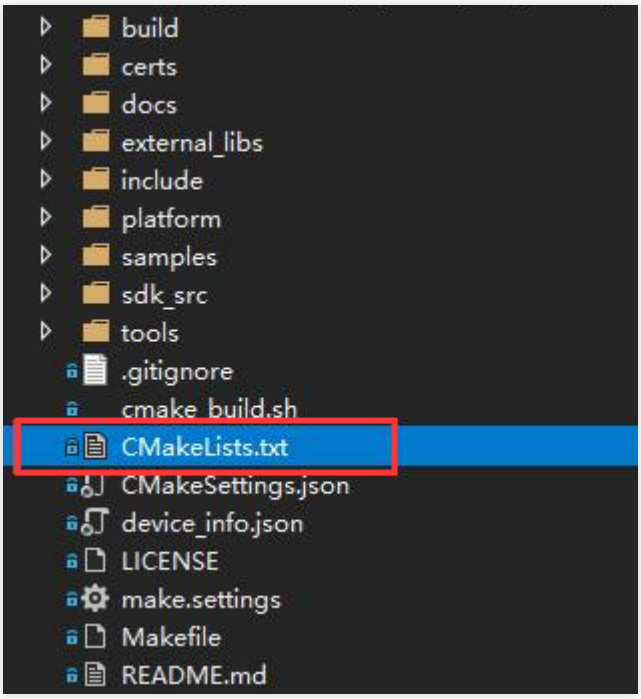
1. 运行 Visual Studio，选择【打开本地文件夹】，并选择下载的 C SDK 目录。



2. 将在腾讯云物联网通信控制台创建的设备的设备信息（以密钥认证设备为例），填写到 device_info.json 中，示例代码如下：

```
"auth_mode":"KEY",
"productId":"S3EUVBQAZW",
"deviceName":"test_device",
"key_deviceinfo":{
 "deviceSecret":"vX6PQqazsGsMyf5SMfs6OA6y"
}
```

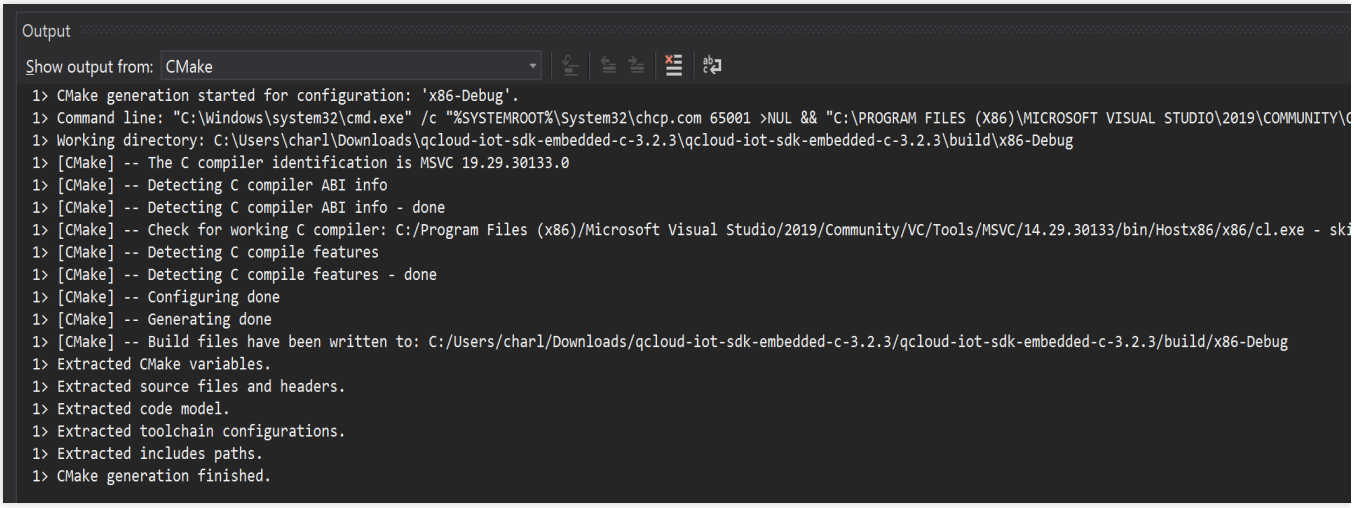
3. 双击打开根目录的 CMakeLists.txt，并确认编译工具链中设置的平台为 **Windows** 和编译工具为 **MSVC**。



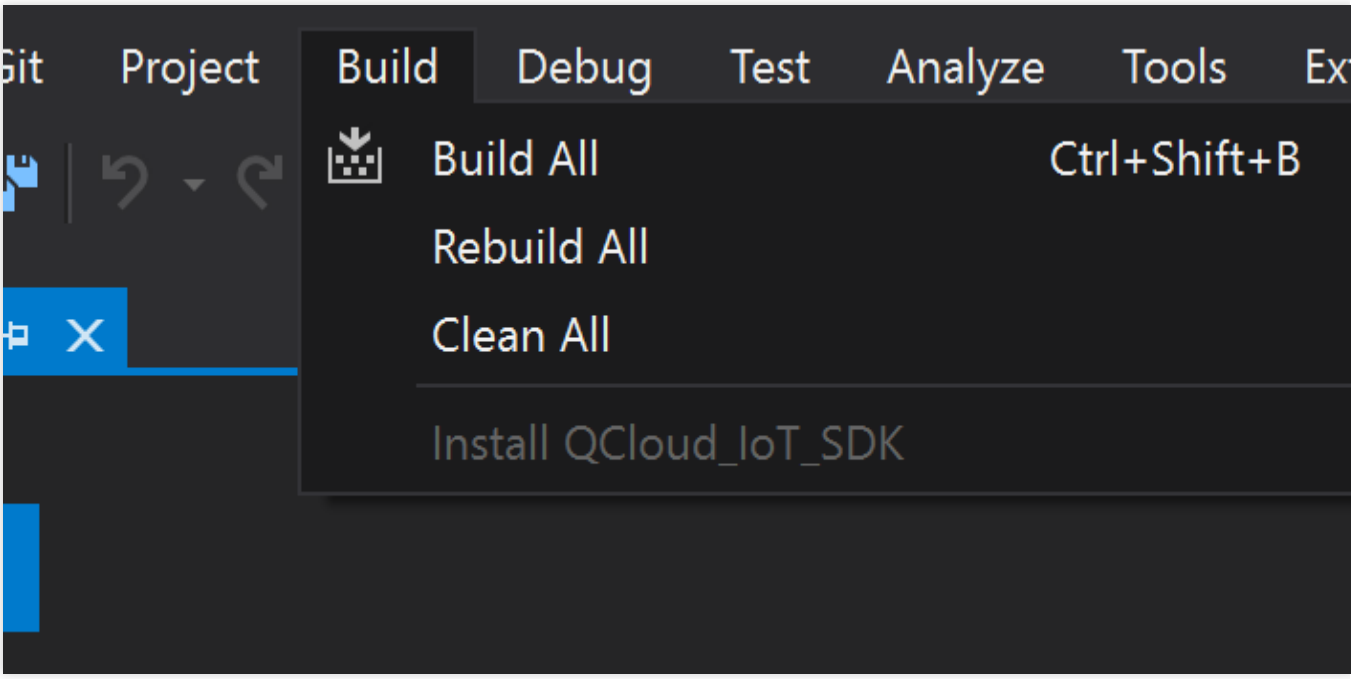

```
# 编译工具链
#set(COMPILE_TOOLS "gcc")
#set(PLATFORM      "linux")

set(COMPILE_TOOLS "MSVC")
set(PLATFORM      "windows")
```

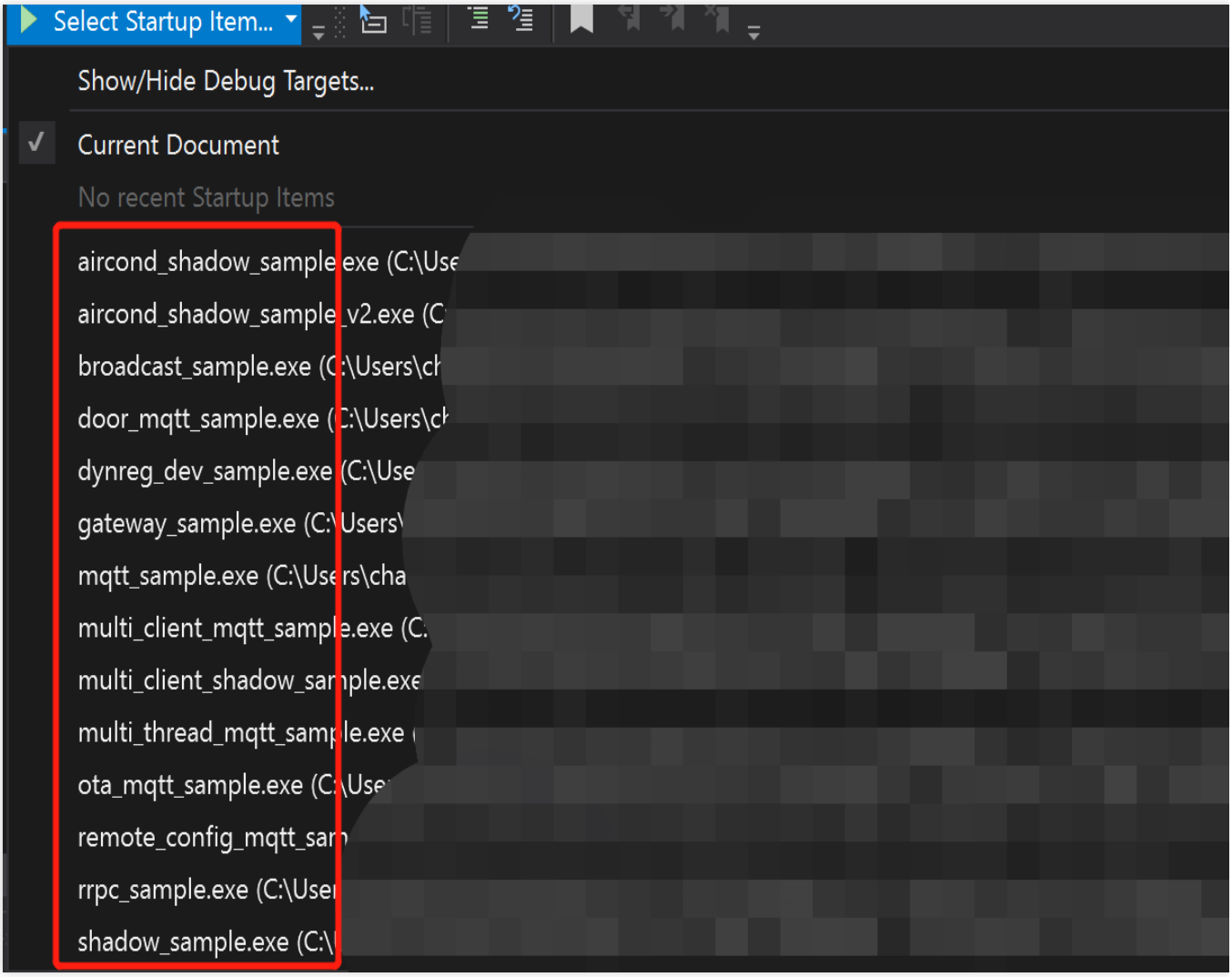
4. Visual Studio 会自动生成 cmake 缓存，请等待 cmake 缓存生成完毕。



5. 缓存生成完毕后，选择【生成】>【全部生成】。



6. 选择相应的示例运行，示例应与用户信息相对应。



MQTT 快速入门

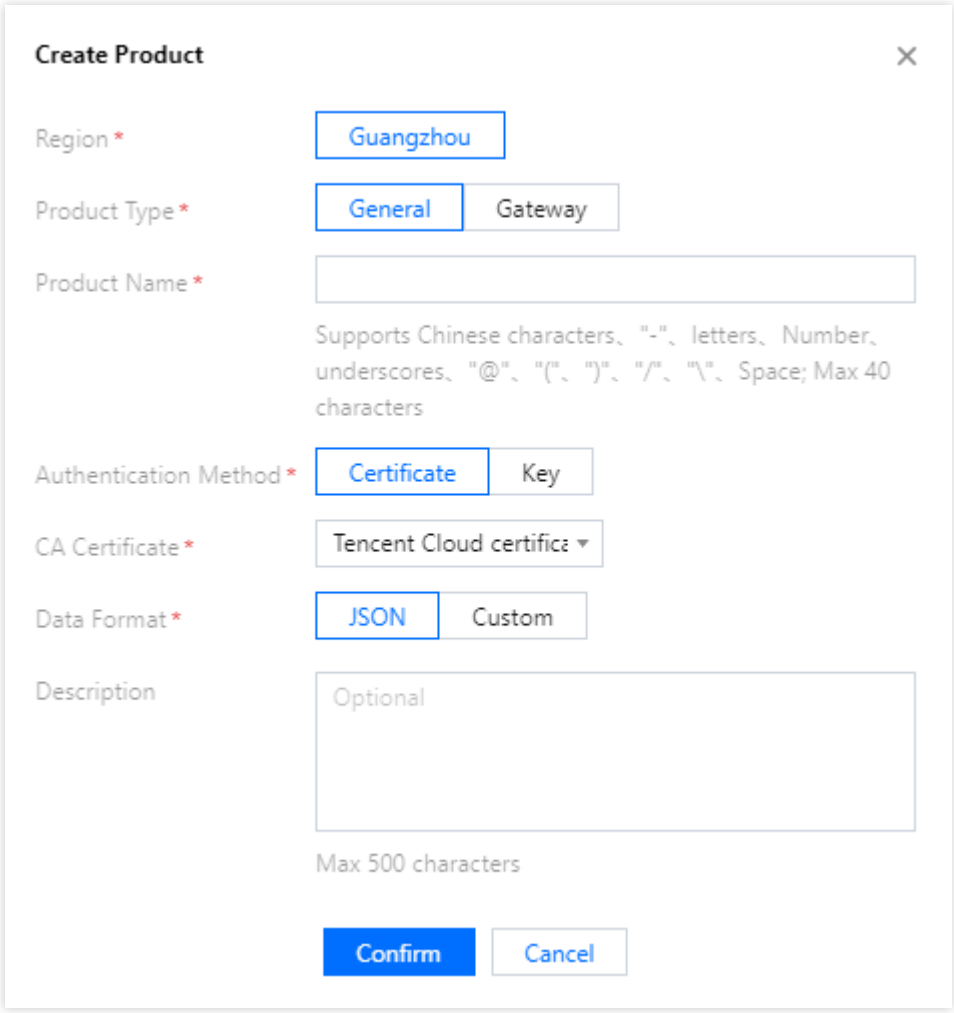
最近更新时间：2024-12-27 16:55:24

本文档主要介绍如何在腾讯云物联网通信 IoT Hub 控制台创建设备和权限，并结合 C-SDK 的 **mqtt_sample** 快速体验设备端通过 MQTT 协议连接到腾讯云 IoT Hub，进行消息发送和接收。

控制台操作

创建产品和设备

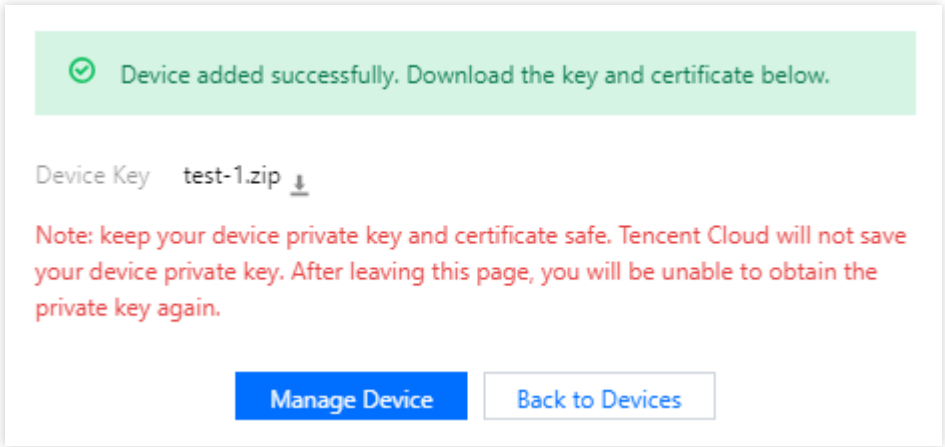
1. 登录 物联网通信控制台，单击左侧菜单**产品列表**。
2. 进入产品列表页面，单击**创建新产品**。
3. 在弹出的添加新产品页面中，选择节点类型和产品类型，输入产品名称，选择认证方式和数据格式，并输入产品描述，
4. 然后单击**确定**。（对于普通直连设备，可按下图选择）



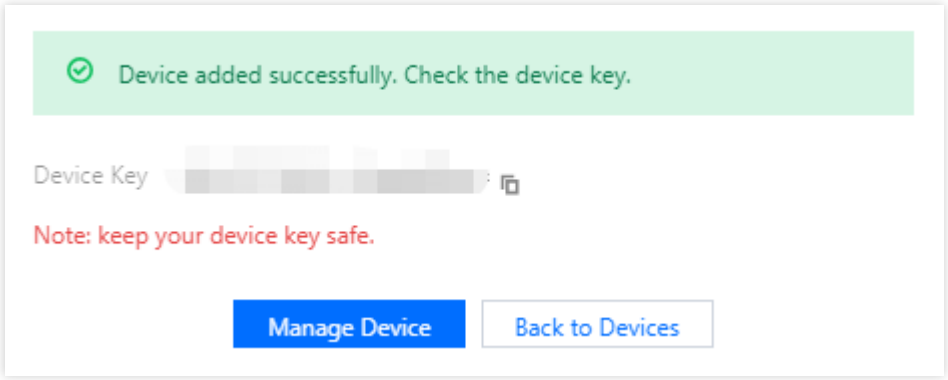
5. 产品创建完成后，在生成的产品页面下，单击**设备列表**。

6. 在设备列表页面，单击**添加新设备**。

如果产品认证方式为证书认证，输入设备名称成功后，切记单击弹窗中的**下载按钮**，下载所得包中的设备密钥文件和设备证书用于设备连接物联网通信的鉴权。

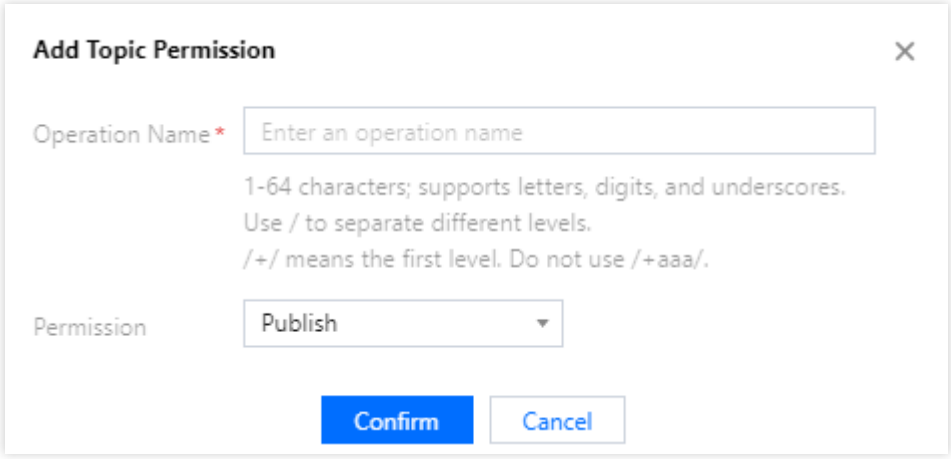


如果产品认证方式为密钥认证，输入设备名称成功后，会在弹窗中显示新添加设备的密钥。



创建 Topic

1. 在生成的产品页面下，单击**权限列表**。
2. 进入权限列表页面，单击**添加 Topic 权限**。
3. 在弹出的 Topic 权限页面中，输入 data，并设置操作权限为**订阅和发布**，单击**确定**



4. 随后将会创建出 `productID/\\${deviceName}/data` 的 Topic，在产品页面的权限列表中即可查看该产品的所有权限。

编译运行示例程序

下面讲述在 Linux 环境编译运行 mqtt_sample 示例（以密钥认证设备为例）

1. 编译 SDK

1.1. 修改 CMakeLists.txt 确保以下选项存在：

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

1.2. 执行脚本编译。

```
./cmake_build.sh
```

1.3. 示例输出位于 `output/release/bin` 文件夹中。

2. 填写设备信息

将上面在腾讯云物联网 IoT Hub 创建的设备的设备信息，填写到 device_info.json 中。

```
"auth_mode":"KEY",
"productId":"S3EUVBRJLB",
"deviceName":"test_device",
"key_deviceinfo":{
"deviceSecret":"vX6PQqazsGsMyf5SMfs6OA6y"
    }
```

3. 执行 mqtt_sample 示例程序

```
./output/release/bin/mqtt_sample
INF|2019-09-12 21:28:20|device.c|iot_device_info_set(67): SDK_Ver: 3.1.0,
Product_ID: S3EUVBRJLB, Device_Name: test_device
DBG|2019-09-12 21:28:20|HAL_TLS_mbedtls.c|HAL_TLS_Connect(204): Setting up the
SSL/TLS structure...
DBG|2019-09-12 21:28:20|HAL_TLS_mbedtls.c|HAL_TLS_Connect(246): Performing the
SSL/TLS handshake...
DBG|2019-09-12 21:28:20|HAL_TLS_mbedtls.c|HAL_TLS_Connect(247): Connecting to
/S3EUVBRJLB.iotcloud.tencentdevices.com/8883...
INF|2019-09-12 21:28:20|HAL_TLS_mbedtls.c|HAL_TLS_Connect(269): connected with
/S3EUVBRJLB.iotcloud.tencentdevices.com/8883...
INF|2019-09-12 21:28:20|mqtt_client.c|IOT_MQTT_Construct(125): mqtt connect
with id: p8t0W success
INF|2019-09-12 21:28:20|mqtt_sample.c|main(303): Cloud Device Construct Success
DBG|2019-09-12 21:28:20|mqtt_client_subscribe.c|qcloud_iot_mqtt_subscribe(138):
topicName=$sys/operation/result/S3EUVBRJLB/test_device|packet_id=1932
INF|2019-09-12 21:28:20|mqtt_sample.c|_mqtt_event_handler(71): subscribe
success, packet-id=1932
```

```
DBG|2019-09-12 21:28:20|system_mqtt.c|_system_mqtt_sub_event_handler(80): mqtt
sys topic subscribe success
DBG|2019-09-12 21:28:20|mqtt_client_publish.c|qcloud_iot_mqtt_publish(337):
publish packetID=0|topicName=$sys/operation/S3EUVBRJLB/test_device|payload=
{"type": "get", "resource": ["time"]}
DBG|2019-09-12 21:28:20|system_mqtt.c|_system_mqtt_message_callback(63): Recv
Msg Topic:$sys/operation/result/S3EUVBRJLB/test_device, payload:
{"type":"get","time":1568294900}
INF|2019-09-12 21:28:21|mqtt_sample.c|main(316): system time is 1568294900
DBG|2019-09-12 21:28:21|mqtt_client_subscribe.c|qcloud_iot_mqtt_subscribe(138):
topicName=S3EUVBRJLB/test_device/data|packet_id=1933
INF|2019-09-12 21:28:21|mqtt_sample.c|_mqtt_event_handler(71): subscribe
success, packet-id=1933
DBG|2019-09-12 21:28:21|mqtt_client_publish.c|qcloud_iot_mqtt_publish(329):
publish topic seq=1934|topicName=S3EUVBRJLB/test_device/data|payload={"action":
"publish_test", "count": "0"}
INF|2019-09-12 21:28:21|mqtt_sample.c|_mqtt_event_handler(98): publish success,
packet-id=1934
INF|2019-09-12 21:28:21|mqtt_sample.c|on_message_callback(195): Receive Message
With topicName:S3EUVBRJLB/test_device/data, payload:{"action": "publish_test",
"count": "0"}
INF|2019-09-12 21:28:22|mqtt_client_connect.c|qcloud_iot_mqtt_disconnect(437):
mqtt disconnect!
INF|2019-09-12 21:28:22|system_mqtt.c|_system_mqtt_sub_event_handler(98): mqtt
client has been destroyed
INF|2019-09-12 21:28:22|mqtt_client.c|IOT_MQTT_Destroy(186): mqtt release!
```

4. 观察消息发送

如下日志信息显示示例程序通过 MQTT 的 Publish 类型消息，上报数据到 `/{productID}/{deviceName}/data`，服务器已经收到并成功完成了该消息的处理。

```
INF|2019-09-12 21:28:21|mqtt_sample.c|_mqtt_event_handler(98): publish success,
packet-id=1934
```

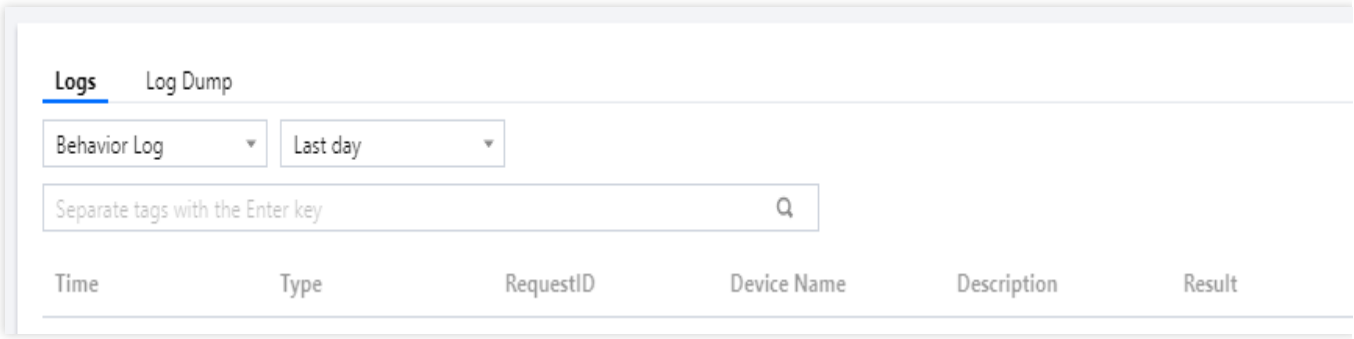
5. 观察消息接收

如下日志信息显示该消息因为是到达已被订阅的 Topic，所以又被服务器原样推送到示例程序，并进入相应的回调函数。

```
INF|2019-09-12 21:28:21|mqtt_sample.c|on_message_callback(195): Receive Message
With topicName:S3EUVBRJLB/test_device/data, payload:{"action": "publish_test",
"count": "0"}
```

6. 观察控制台日志

登录 物联网通信控制台，单击该产品名称，单击上方菜单**云日志**，即可查看刚上报的消息。



接口及可变参数说明

最近更新时间：2024-12-27 16:55:24

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 的使用可以根据具体场景需求，配置相应的参数，满足实际业务的运行。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 协议发送消息和接受消息的 buffer 大小默认为2048字节，最大支持16KB。
3. COAP 协议发送消息和接受消息的 buffer 大小默认为512字节，最大支持1KB。
4. MQTT 心跳消息发送周期，最大值为690秒，单位：毫秒。
5. 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` 文件如下宏定义可以改变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，更加详细的说明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。

MQTT 接口

序号	函数名	说明
1	IOT_MQTT_Construct	构造 MQTTClient 并连接 MQTT 云端服务
2	IOT_MQTT_Destroy	关闭 MQTT 连接并销毁 MQTTClient
3	IOT_MQTT_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_MQTT_Publish	发布 MQTT 消息
5	IOT_MQTT_Subscribe	订阅 MQTT 主题
6	IOT_MQTT_Unsubscribe	取消订阅已订阅的 MQTT 主题
7	IOT_MQTT_IsConnected	查看当前 MQTT 是否已连接
8	IOT_MQTT_GetErrCode	获取IOT_MQTT_Construct失败的错误码

多线程环境使用说明

SDK 对于 MQTT 接口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_MQTT_Yield，IOT_MQTT_Construct 以及IOT_MQTT_Destroy。

可以多线程调用 IOT_MQTT_Publish，IOT_MQTT_Subscribe 及 IOT_MQTT_Unsubscribe。

IOT_MQTT_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。

设备影子接口

关于设备影子功能介绍，可以参考 设备影子详情。

序号	函数名	说明
1	IOT_Shadow_Construct	构造设备影子客户端 ShadowClient，并连接 MQTT 云端服务
2	IOT_Shadow_Publish	影子客户端发布 MQTT 消息

3	IOT_Shadow_Subscribe	影子客户端订阅 MQTT 主题
4	IOT_Shadow_Unsubscribe	影子客户端取消订阅已订阅的 MQTT 主题
5	IOT_Shadow_IsConnected	查看当前影子客户端的 MQTT 是否已连接
6	IOT_Shadow_Destroy	关闭 Shadow MQTT 连接并销毁 ShadowClient
7	IOT_Shadow_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
8	IOT_Shadow_Update	异步更新设备影子文档
9	IOT_Shadow_Update_Sync	同步方式更新设备影子文档
10	IOT_Shadow_Get	异步方式获取设备影子文档
11	IOT_Shadow_Get_Sync	同步方式获取设备影子文档
12	IOT_Shadow_Register_Property	注册当前设备的设备属性
13	IOT_Shadow_UnRegister_Property	删除已经注册过的设备属性
14	IOT_Shadow_JSON_ConstructReport	在 JSON 文档中添加 reported 字段，不覆盖更新
15	IOT_Shadow_JSON_Construct_OverwriteReport	在 JSON 文档中添加 reported 字段，覆盖更新
16	IOT_Shadow_JSON_ConstructReportAndDesireAllNull	在 JSON 文档中添加 reported 字段，同时清空 desired 字段
17	IOT_Shadow_JSON_ConstructDesireAllNull	在 JSON 文档中添加 "desired": null 字段

CoAP 接口

序号	函数名	说明
1	IOT_COAP_Construct	构造 CoAPClient 并完成 CoAP 连接
2	IOT_COAP_Destroy	关闭 CoAP 连接并销毁 CoAPClient
3	IOT_COAP_Yield	在当前线程上下文中，进行 CoAP 报文读取和消息处理等任务
4	IOT_COAP_SendMessage	发布 CoAP 消息

5	IOT_COAP_GetMessageId	获取 COAP Response 消息 msgId
6	IOT_COAP_GetMessagePayload	获取 COAP Response 消息内容
7	IOT_COAP_GetMessageCode	获取 COAP Response 消息错误码

OTA 接口

关于 OTA 固件下载功能介绍，可以参考 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可以参考 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

网关功能接口

关于网关功能介绍，可以参考 SDK docs 目录下物联网通信平台文档网关产品部分。

序号	函数名	说明
1	IOT_Gateway_Construct	构造 Gateway client 并完成 MQTT 连接
2	IOT_Gateway_Destroy	关闭 MQTT 连接并销毁 Gateway client
3	IOT_Gateway_Subdev_Online	子设备上线
4	IOT_Gateway_Subdev_Offline	子设备下线
5	IOT_Gateway_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
6	IOT_Gateway_Publish	发布 MQTT 消息
7	IOT_Gateway_Subscribe	订阅 MQTT 主题
8	IOT_Gateway_Unsubscribe	取消订阅已订阅的 MQTT 主题

设备信息存储

最近更新时间：2024-12-27 16:55:24

概述

腾讯云物联网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。设备端需要存储这些设备身份信息。C-SDK 提供了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区别。

预置烧录

创建产品后，在 物联网通信控制台 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，然后用申请到的三元组或者四元组信息进行设备认证。
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 IMEI 或者 MAC 地址。若使能动态注册的同时不使能自动设备创建，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，否则会产生较大的安全隐患。

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_ID",
        "sub_devName":"YOUR_SUBDEV_DEVICE_NAME"
    }
}
```

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_name;
    initParams->product_id = sg_devInfo.product_id;
     ......
}
```

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
            ......
}
```

基于 Android SDK 接入
Android SDK 版本说明

最近更新时间：2024-12-27 17:03:42

代码托管

自 v1.0.0 版本开始，Android 设备端 SDK 代码使用 Github 托管。

版本信息

自 v1.0.0 版本开始，Android 设备端 SDK 版本迭代信息，请参见 Github。

Android SDK 工程配置

最近更新时间：2024-12-27 17:03:43

腾讯云物联网设备端 Android 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

已按照 设备接入准备 创建好产品和设备。

引用方式

集成 SDK 方式

若不需要将 IoT SDK 运行在 service 组件中，则只需要依赖 iot_core。
依赖 maven 远程构建，示例如下：

```
dependencies {
        implementation 'com.tencent.iot.hub:hub-device-android-core:x.x.x'
        implementation 'com.tencent.iot.hub:hub-device-android-service: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。
若不需要将 IoT SDK 运行在 service 组件中，则只需要依赖 iot_core。
若需要将 IoT SDK 运行在 service 组件中，则只需依赖 iot_service。
依赖本地 SDK 源码构建：
修改应用模块的 build.gradle，使应用模块依赖 iot_core 和 iot_service 源码，示例如下：

```
dependencies {
        implementation project(':hub:hub-device-android:iot_core')
        implementation project(':hub:hub-device-android:iot_service')
}
```

认证连接

编辑 app-config.json 文件中的配置信息，可在 IoTMqttFragment.java 读取对应以下数据：

```
{
            "PRODUCT_ID":         "",
            "DEVICE_NAME":        "",
            "DEVICE_PSK":         "",
            "SUB_PRODUCT_ID":     "",
            "SUB_DEV_NAME":       "",
            "SUB_PRODUCT_KEY":    "",
            "TEST_TOPIC":         "",
            "SHADOW_TEST_TOPIC":  "",
            "PRODUCT_KEY":        ""
 }
```

SDK 提供证书认证与密钥认证两种认证方式，需按照已创建的产品认证类型进行选择设置。

密钥认证须在 app-config.json 配置信息中填入 PRODUCT_ID、DEVICE_NAME、DEVICE_PSK 所对应的参数。

SDK 会根据设备配置信息自动生成签名，作为接入物联网通信的凭证。

证书认证须在 app-config.json 配置信息中填入 PRODUCT_ID、DEVICE_NAME 等内容并读取设备证书、设备私钥文件的内容。读取方式分为两种：

通过 AssetManager 进行读取，此时需在工程 `hub/hub-android-demo/src/main` 路径下创建 assets 目录并将设备证书、私钥放置在该目录中。

通过 InputStream 进行读取，此时需传入设备证书、私钥的全路径信息。

1.1 成功读取证书文件与私钥文件之后，需在 IoTMqttFragment.java 中设置 mDevCertName 证书名称与 mDevKeyName 私钥名称。

```
  private String mDevCertName = "YOUR_DEVICE_NAME_cert.crt";
  private String mDevKeyName  = "YOUR_DEVICE_NAME_private.key";
```

1.2 配置完成之后，在工程中调用 SDK 中 MQTT 连接的相关接口，即可完成设备的接入。

```
  mMqttConnection = new TXGatewayConnection(mContext, mBrokerURL, mProductID, mDevNa
  mMqttConnection.connect(options, mqttRequest);
```

Android SDK 使用说明

最近更新时间：2024-12-27 17:03:43

Android SDK 在提供设备的接入功能之外，还提供了网关子设备，设备影子，OTA 等功能。相关功能接口描述如下。

MQTT 接口

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到后台服务器
reportOTAState	上报设备升级状态到后台服务器

MQTT 网关接口

TXGatewayConnection

方法名	说明
connect	网关连接
reconnect	网关重连
disConnect	断开网关 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到后台服务器
reportOTAState	上报设备升级状态到后台服务器
addSubDev	添加子设备
removeSubdev	移除子设备
findSubdev	查找子设备
gatewaySubdevOffline	子设备下线
gatewaySubdevOnline	子设备上线

设备影子接口

TXShadowConnection

方法名	说明
connect	Shadow 连接
disConnect	关闭 Shadow 连接
getConnectStatus	获取 MQTT 连接状态
update	更新设备影子文档
get	获取设备影子文档
registerProperty	注册设备属性
unRegisterProperty	取消注册设备属性
reportNullDesiredInfo	更新 delta 信息后，上报空的 desired 信息
setBufferOpts	设置断连状态 buffer 缓冲区

getMqttConnection	获取 TXMqttConnection 实例

MQTT 远程服务客户端

TXMqttClient

方法名	说明
setMqttActionCallBack	设置 MqttAction 回调接口
setServiceConnection	设置远程服务连接回调接口
init	初始化远程服务客户端
startRemoteService	开启远程服务
stopRemoteService	停止远程服务
connect	MQTT 连接
disConnect	断开 MQTT 连接
subscribe	订阅 MQTT 主题
unSubscribe	取消订阅 MQTT 主题
publish	发布 MQTT 消息
setBufferOpts	设置断连状态 buffer 缓冲区
clear	释放资源

Shadow 远程服务客户端

TXShadowClient

方法名	说明
setShadowActionCallBack	设置 ShadowAction 回调接口
setServiceConnection	设置远程服务连接回调接口
init	初始化远程服务客户端
startRemoteService	开启远程服务
stopRemoteService	停止远程服务

connect	Shadow 连接
disConnect	断开 Shadow 连接
getMqttClient	获取 MQTT 客户端实
get	获取设备影子
update	更新设备影子
registerProperty	注册设备属性
unRegisterProperty	取消注册设备属性
reportNullDesiredInfo	更新 delta 信息后，上报空的 desired 信息
setBufferOpts	设置断连状态 buffer 缓冲区
clear	释放资源

MQTT 通道固件升级

TXMqttClient

方法名	说明
initOTA	初始化 OTA 功能
reportCurrentFirmwareVersion	上报设备当前版本信息到后台服务器
reportOTAState	上报设备升级状态到后台服务器

基于 Java SDK 接入

Java SDK 版本说明

最近更新时间：2024-12-27 17:03:43

代码托管

自 v1.0.0 版本开始，Java 设备端 SDK 代码使用 Github 托管。

版本信息

自 v1.0.0 版本开始，Java 设备端 SDK 版本迭代信息，请参见 Github。

Java SDK 工程配置

最近更新时间：2024-12-27 17:03:43

腾讯云物联网设备端 Java 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

已按照 设备接入准备 创建好产品和设备。

引用方式

如果您需要通过 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.hub:hub-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。
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 Java SDK 源码。

认证连接

设备的身份认证支持两种认证方式，密钥认证和证书认证：
若使用密钥认证方式，需 ProductID，DevName 和 DevPSK。
若使用证书认证方式，需 ProductID，CertFile 和 PrivateKeyFile。
认证连接示例代码如下：

```
private String mProductID = "YOUR_PRODUCT_ID";
private String mDevName = "YOUR_DEVICE_NAME";
private String mDevPSK = "YOUR_DEV_PSK";
private String mCertFilePath = null;
private String mPrivKeyFilePath = null;
TXMqttConnection mqttconnection = new TXMqttConnection(mProductID, mDevName,
mDevPSK, new callBack());
mqttconnection.connect(options, null);
```

```
try {
        Thread.sleep(20000);
} catch (InterruptedException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
mqttconnection.disConnect(null);
```

Java SDK 使用说明

最近更新时间：2024-12-27 17:03:43

Java SDK 除提供设备的接入功能之外，还提供网关子设备，设备影子等功能。相关功能接口如下。

MQTT 接口

MQTT 的相关接口定义在 TXMqttConnection 类中，支持发布和订阅功能；如果需支持设备影子功能，则需使用 TXShadowConnection 类及其方法，TXMqttConnection 类接口，介绍如下：

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

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 IoT Hub 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，后续只要网关链接成功，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 TXGatewayConnection 类接口中，介绍如下：

方法名	说明
connect	网关 MQTT 连接
reconnect	网关 MQTT 重连

disConnect	断开网关 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
gatewaySubdevOffline	子设备下线
gatewaySubdevOnline	子设备上线
gatewayBindSubdev	子设备绑定
gatewayUnbindSubdev	子设备解绑

设备影子接口

如果需要支持设备影子功能，需使用 TXShadowConnection 类中的接口，介绍如下：

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
update	更新设备影子文档
get	获取设备影子文档
reportNullDesiredInfo	更新 delta 信息后，上报空的 desired 信息
setBufferOpts	设置断连状态 buffer 缓冲区
getMqttConnection	获取 TXMqttConnection 实例

getConnectStatus	获取 MQTT 连接状态

基于 Python SDK 接入
Python SDK 版本说明

最近更新时间：2024-12-27 17:03:42

代码托管

自 v1.0.0 版本开始，Python 设备端 SDK 代码使用 Github 托管。

版本信息

自 v1.0.0 版本开始，Python 设备端 SDK 版本迭代信息，请参见 Github。

Python SDK 工程配置

最近更新时间：2024-12-27 17:03:42

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

已按照 设备接入准备 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。

Python SDK 使用说明

最近更新时间：2024-12-27 17:03:43

Python SDK 除提供设备的接入功能之外，还提供网关子设备，设备影子等功能。相关功能接口如下。

MQTT 接口

MQTT 的相关接口定义在 hub.py 类中，支持发布和订阅功能；如果需支持设备影子功能，则需使用 shadow.py 类及其方法，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 IoT Hub 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，后续只要网关链接成功，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

动态注册接口

如果需要使用动态注册功能，需使用 hub.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 hub.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