青莲云嵌入式 SDK 开发使用文档

版本	编写/修订说明	修订人	修订日期	备注
3.0.0	创建文档	王科岩	20181120	
3.0.1	1.修改上传数据 一次可最多上传整数数据点个数为 60 2.增加章节 3.9 获取时间结构体	杨思嘉	20190213	
3.0.2	修改关于子设备 id 的声明	杨思嘉	20190319	
3.0.3	更新文档结构 优化 ota 接口，新增固件信息接口 增加补丁传输相关接口	王科岩	20190523	
3.0.4	1. 修改章节 3.7 数据发送成功报告 2. 增加章节 3.8 获取发送数据错误列表 3. 修改章节 4.1.1 上传数据全部接口 4. 修改章节 8.1 子设备上线接口 5. 修改章节 8.2 子设备下线接口	杨思嘉	20190711	

3.0.5	1. 章节 4.1.1 上传数据区分数据上传动作	杨思嘉	20190801	
3.0.6	1. 删除章节 3.8 获取发送数据错误列表 2. 修改章节 3.7 发送数据成功报告 1> 增加云端返回的数据列表参数 2> 此接口不再用于子设备上，下线接口的回调。 3. 增加章节 8.3 子设备状态报告接口	杨思嘉	20190807	
3.0.7	1. 增加章节 3.8 crc 校验码计算 2. 修改章节 5.1.2 ota 固件信息 1>增加文件 crc 信息 3. sub_id 长度扩展至 32 字节	杨思嘉	20191023	

目录

目录.....3

1 概要.....6

2 SDK 目录结构.....7

3 系统函数.....8

3.1 安全快速联网.....8

3.2 设置设备运行状态.....9

3.3 设备运行状态报告.....10

3.4 获取网络时间.....11

3.5 解析时间.....11

3.6 打印输出函数.....12

3.7 发送数据成功报告.....12

3.8 crc 校验码计算.....13

4 传输数据.....14

4.1 传输数据点数据.....14

4.1.1 上传数据.....14

4.1.2 接收数据.....16

4.2 传输自定义数据.....17

4.2.1 透传自定义数据.....17

4.2.2 接收自定义数据.....18

5 远程升级.....19

5.1 固件升级.....19

5.1.1 设置 ota 属性.....19

5.1.2 ota 固件信息.....20

5.1.3 接收固件数据块.....20

5.1.4 接收升级指令.....20

5.2 补丁升级.....21

5.2.1 补丁信息.....21

5.2.2 获取补丁.....21

5.2.3 接收补丁数据块.....22

5.2.4 补丁状态上报.....22

5.3 接收数据块接口.....22

6 设备信息.....23

6.1 请求设备信息.....23

6.2 接收设备信息.....24

6.3 配置设备信息.....25

7 本地数据存储.....26

7.1 保存数据至本地.....26

7.2 加载本地数据.....27

7.3 清空本地数据.....27

8 子设备功能.....27

8.1 子设备上线.....27

8.2 子设备离线.....29

8.3 子设备状态报告.....30

1 概要

青莲云作为领先的物联网安全服务商，为了让开发者不必关心数据的加密传输、网络链路的安全通信，适配了一系列 wifi、蓝牙、NB-IOT 模组。

我们提供了不同平台的嵌入式 SDK，开发者可以通过简单的调试，立即拥有强大的后端云能力，专注于具体业务研发。

各平台 SDK 可适用于联网硬件单品或网关类产品的开发。注意，开发硬件单品时没有“子设备”概念，全部涉及“子设备”的接口，将“sub_id”传入 NULL 即可。

本 SDK 提供以下功能的接口说明：

1) 安全快速联网
2) 连接状态报告
3) 网络时间同步
4) 实时数据上报
5) 实时获取命令
6) 实时/离线自定义消息推送
7) OTA 升级
8) 补丁升级
9) 获取用户/设备信息
10) 配置设备信息
11) 本地存储/加载数据
12) 子设备上线/下线

2 sdk 目录结构

-\SDK	
-\lib	
-\libiot_platform_os.a	正式版 sdk
-\libiot_platform_os_debug.a	debug 版 sdk
-\libmbedcrypto.a	SSL 库
-\libmbedtls.a	SSL 库
-\libmbedx509.a	SSL 库
-\libiotpatch.a	热补丁库
-\include	
-\iot_interface.h	云端对外接口
-\iot_aes.h	aes 库
-\iot_cjson.h	cjson 库
-\iot_md5.h	md5 库
-\iot_base64.h	base64 库
-\src	
-\main.c	使用样例
-\iot_interface.c	用户回调函数实现
-\makefile	
-readme.txt	相关说明

3 系统函数

3.1 安全快速联网

初始化设备与云端交互的上下文环境，并与云端会建立长连接。当长连接断开时，会自动重连。注意，填写密钥时，需将官网的一串字符串转换成相应的十六进制编码，即在每个字节前增加 0x 作为开头。如字符串是 5668，转换时应改为 0x56，0x68。

iot_s32 iot_start (struct iot_context* ctx);

struct iot_context 结构体的内容如下：

参数	长度	说明
product_id	4	产品 ID，云平台生成，4 字节的无符号整型数字
product_key	16	产品密钥，云平台生成，16 字节的十六进制编码
mcu_version	5	mcu 固件版本，"xx.xx"，0≤x≤9
recvbuf_size	4	接收数据 buffer 大小，范围 1024-1048576，默认 1024
sendbuf_size	4	发送数据 buffer 大小，范围 1024-1048576，默认 1024
server_addr	n	云服务器地址
server_port	2	云服务器端口
返回值	4	0 ：成功； -1：失败

3.2 设置设备运行状态

此函数用于设置设备的运行状态，设置结果请见 3.3 节“设备运行状态报告”。

```
void    iot_status_set(      DEV_STATUS_T    dev_status,
                             iot_u32         timeout      )
```

参数	说明
dev_status	DEV_STA_FACTORY_RESET(2)：恢复出厂设置，会删除云端设备信息，并解除与 APP 端的绑定关系，恢复出厂成功后自动调用 3.3 节函数告知。 DEV_STA_BIND_PERMIT(3) ：允许用户通过局域网绑定设备，可在连接云端成功后调用。绑定成功或失败都会调用 3.3 节函数告知，告知参数为 DEV_STA_BIND_FAILED(4) 或 DEV_STA_BIND_SUCCESS(5)。 (02.07 之后的版本必须调用此接口才能进行绑定操作！) DEV_STA_UNBIND(6) ：设置解绑，设备会解除与 APP 端的绑定关系，成功后自动调用 3.3 节函数告知。
timeout	设备运行状态保持时长，单位为秒; timeout = 0 表示不设置超时状态。

3.3 设备运行状态报告

当设备运行状态发生改变时，sdk 自动调用此函数。

void iot_status_cb(DEV_STATUS_T dev_status,

iot_u32 timestamp)

参数	说明
dev_status	DEV_STA_CONNECTED_CLOUD(0) ：连接云端成功 DEV_STA_DISCONN_CLOUD(1) ：网络异常连接断开 DEV_STA_FACTORY_RESET(2) ：恢复出厂成功，本地可重置设备 DEV_STA_BIND_FAILED（4） ：绑定失败（绑定超时或设备未联网），允许绑定终止 DEV_STA_BIND_SUCCESS （5） ：绑定成功，允许绑定终止 DEV_STA_UNBIND （6） ：解绑成功 DEV_STA_USER_SHARE_INC（7） ：分享用户增加 DEV_STA_USER_SHARE_DEC（8） ：分享用户减少 DEV_STA_AUTH_LIMIT(11) ：设备授权数达上限，无法通过云端认证 DEV_STA_MAC_BIND_OTHER_ID(12)：设备 mac 地址已绑定其他产品，无法通过云端认证
timestamp	状态改变的时间点

注意！回调函数中不可执行太多耗时代码。

3.4 获取网络时间

iot_u32　　iot_get_onlinetime(　void)

参数	说明
返回值	0：时间无效 >0：实时网络时间戳

3.5 解析时间

void　　iot_parse_timestamp (　iot_u32　timestamp,　struct s_time*　st　);

参数	说明
timestamp	需要被转换的整型时间戳
st	转换后的时间结构体

时间结构体定义如下:

```
struct s_time {
    int   sec;                    //秒
    int   min;                    //分
    int   hour;                   //时
    int   day;                    //日
    int   mon;                    //月
    int   year;                   //年
    int   week;                   //周
```

};

3.6 打印输出函数

此函数用于输出日志信息，需用户自行实现，可根据需要重定向到串口、屏幕、文件等位置。

void iot_print (const char * str)

参数	说明
str	输出的日志内容

3.7 发送数据成功报告

发送数据成功后，sdk 自动调用下面这个回调函数。当调用 4.1.1 节上传数据、4.2.1 节透传自定义数据某个函数后，此回调函数会被调用。回调函数中不可执行太多耗时代码。

void iot_data_cb (iot_u32 data_seq， DATA_LIST_T * data_list);

参数	说明
data_seq	某条数据的序列号，如不关心何时上传成功，可不做处理
data_list	上传数据、透传数据的错误列表

数据列表结构体定义如下：

```
typedef struct data_list
{
    int                 count;          //列表信息个数
    DATA_INFO_T *       info;           //数据信息列表
}DATA_LIST_T;
```

数据信息结构体定义如下：

```
typedef struct data_info
{
    char            *       dev_id;         //设备 id(主设备:NULL，子设备:subid)
    DATA_INFO_TYPE_T    info_type;          //数据类型码
}DATA_INFO_T;
```

数据列表中信息类型结构体定义如下：

```
typedef enum DATA_INFO_TYPE {
    DATA_INFO_NULL              = 0,        //数据信息为空
    DATA_INFO_NOT_ONLINE    = 16,           //子设备发送数据、透传数据、下线时，如果未上线，则返回此错误
    DATA_INFO_BIND_OTHER    = 17,           //子设备上线、下线时，如果已被其他主设备绑定，则返回此错误
    DATA_INFO_BIND_NONE     = 18,           //子设备上线、下线时，如果未被绑定，则返回此错误
}DATA_INFO_TYPE_T;
```

3.8 crc 校验码计算

用户可使用该函数计算数据或文件的 crc16 校验码。

```
iot_u16     iot_crc16_calc(  iot_u8      *   data,
                             iot_u32         data_len,
                             iot_u16         pre_crc)
```

参数	说明

data	待校验数据
data_len	待校验数据长度
pre_crc	上一次 crc 校验码。（pre_crc 的初始值为 0xffff）
返回值	crc 校验码

4 传输数据

4.1 传输数据点数据

发送/接收数据的最大长度与 iot_ctx 中的 buffer 的大小有关。

如：buffer 设置为 2048，以整型类型数据点为例，最多可以一次性发送 60 个整型数据。

4.1.1 上传数据

- 开发者需明确每个数据点的 dpid、类型(通过云平台获取)。根据数据点的类型，调用不同的函数将数据点 id、对应数值添加到发送队列中。
- 目前支持的类型包括整数型、布尔型、枚举型、浮点型、字符型、故障型、二进制。
- 上传数据时，需保证在云端创建的数据点是可上报的。
- 发送给主设备数据点的 sub_id 为 NULL。

① 添加数据点到发送队列

```
void   dp_up_add_int   (const char sub_id[32],  iot_u8  dpid,   iot_s32    value)
void   dp_up_add_bool (const char sub_id[32],  iot_u8  dpid,   iot_u8     value)
void   dp_up_add_enum(const char sub_id[32], iot_u8  dpid,   iot_u8     value)
void   dp_up_add_float (const char sub_id[32],   iot_u8 dpid,   iot_f32    value)
```

```
void   dp_up_add_string(const   char   sub_id[32],   iot_u8     dpid,
                        const   char   *   str       ,   iot_u32   str_len)
void   dp_up_add_fault ( const   char   sub_id[32],   iot_u8     dpid,
                        const   char   *   fault     ,   iot_u32   fault _len)
void   dp_up_add_binary(const   char   sub_id[32],   iot_u8     dpid,
                        const    iot_u8 * bin      ,   iot_u32    bin_len)
```

② 上传一条数据，设备功能发生改变时将最新数据上传，一条数据可包含多个子设备的多个数据点。

```
iot_s32 iot_upload_dps (   DATA_ACT_T   data_act,   iot_u32*   data_seq );
```

参数	说明
data_act	发送上行数据的动作
data_seq	传出参数，本条数据的序列号，如果需要确定数据何时上传成功，可记录此发送序列号，与收到的进行对比。
返回值	0 ： 成功； -1：失败

```
typedef enum DATA_ACT_TYPE
{
    DATA_ACT_NORMAL                 = 0,     //默认发送数据，并触发联动
    DATA_ACT_FORBIT_LINKAGE         = 1,     //发送数据，但该数据不会触发联动
}DATA_ACT_T;
```

4.1.2 接收数据

- 开发者需明确每个数据点的 dpid、类型(通过云平台获取)。根据数据点的类型，调用不同的转换函数转换成需要的数值。
- 目前支持的类型包括整数型、布尔型、枚举型、浮点型、字符型、故障型、二进制。
- 接收数据时，需保证在云端创建的数据点是可下发的。

① 填写以下数据结构

```
iot_download_dps_t      iot_down_dps[]  =
{   云端数据点 ID              数据点类型              处理函数
    { DP_ID_DP_SWITCH,     DP_TYPE_BOOL,     dp_down_handle_switch    },
};
```

② 定义针对某个数据点的处理函数，函数类型为

```
void     dp_down__handle*(  char     sub_id[32],
                            iot_u8*  indata,
                            iot_u32  inlen    )
```

其中 sub_id 为 NULL 时可能是硬件单品，也可能是网关自身的数据。

如，数据点 switch 的类型为 bool，则接受其数据的处理函数实现如下：

```
void   dp_down_handle_switch (  char     sub_id[32],
                                iot_u8*   in_data,
                                iot_u32  inlen    )
{
    iot_u8   dp_switch   =   bytes_to_bool(   in_data  );   //转换成对应数值
    if(   dp_switch   ==  0   )
```

```
    {    }
    else
    {    }
}
```

③ 处理函数要根据数据点的类型，调用不同的转换函数

```
iot_s32     bytes_to_int (    const iot_u8     bytes[4]     );
iot_u8      bytes_to_bool (   const iot_u8     bytes[1]     );
iot_u8      bytes_to_enum (const iot_u8      bytes[1]     );
iot_f32     bytes_to_float (  const iot_u8     bytes[4]     );
```

其他包括字符串、故障、二进制类型请直接对原始数据进行处理。

④ 接收到合法数据后，sdk 会自动调用数据点对应的处理函数，处理函数一般会操作改变设备状态。

比如收到开/关命令，处理函数中应先执行设备开/关，接着将最新的设备开关状态通过调用 5.1 节接口上传至云端。

4.2 传输自定义数据

4.2.1 透传自定义数据

- 可透传任意格式自定义数据，请与 app 开发者自行约定
- 手机端在线，云端不保存数据，直接转发至手机
- 手机端离线，云端最多会保存 20 条数据，待手机上线时发送，发送后清空

```
iot_s32  iot_tx_data (const    char          sub_id[32],
                               iot_u32*      data_seq,
```

iot_u8*　　data,

iot_u32　　data_len　　)

参数	说明
sub_id	硬件单品：NULL 网关自身数据：NULL 子设备数据：自定义的子设备 id，仅限字母数字组合
data_seq	传出参数，本条数据的序列号，如果需要确定数据何时上传成功，可记录此发送序列号，与收到的进行对比。
data	字符形式、二进制形式、特殊字符的自定义数据
data_len	自定义数据长度
返回值	0 ： 成功； -1：失败

4.2.2　接收自定义数据

- 接收来自 app 的透传数据，格式请与 app 开发者自行约定
- 设备在线，云端收到 app 数据后，直接透传至设备
- 设备离线，云端最多会保存 20 条 app 数据，待设备上线时发送，发送后清空

回调函数中不可执行太多耗时代码。

void　　iot_rx_data_cb (char　　sub_id[32],

iot_u32　　data_ timestamp,

iot_u8*　　data,

iot_u32 data_len)

参数	说明
sub_id	硬件单品：NULL 网关自身数据：NULL 子设备数据：自定义的子设备 id，仅限字母数字组合
data_timestamp	时间戳，云端收到本条数据的时间点，如果不需要过期消息可以通过时间戳过滤掉
data	自定义数据，仅支持字符型
data_len	自定义数据长度

5 远程升级

青莲云提供安全的设备远程升级服务，可支持整机器固件升级和系统补丁升级操作。其中，固件升级完成后，需设备端执行重启操作，并以高版本固件重新连接云平台；补丁升级完成后，需告知云平台补丁升级状态。

5.1 固件升级

具体流程，请参考 ota 升级文档《青莲云 OTA 升级操作流程》及本小节内容进行实现。

5.1.1 设置 ota 属性

iot_s32 iot_ota_option_set(iot_u32 expect_time,

iot_u32 chunk_size)

参数	说明

expect_time	期望升级倒计时，单位秒，范围 120-3600
chunk_size	云端会将固件按 chunk_size 分块，分块后一次下发一块。 2 的 n 次幂，范围 256-1048576，默认 1024
返回值	0 ： 设置属性成功，可在 log 中查看实际升级时间 -1 ： 失败，参数错误

5.1.2 ota 固件信息

```
void    iot_ota_info_cb （  char        name[16],
                            iot_u32     total_len,
                            char        version[6]，
                            iot_u16     file_crc ）
```

参数	说明
name	待升级固件名称
version	待升级固件版本，"xx.xx"，0≤x≤9
total_len	待升级固件文件总大小，单位字节
file_crc	待升级固件文件 crc

5.1.3 接收固件数据块

参考 5.3 节，chunk_stat 为 OTA 数据块的情况。

5.1.4 接收升级指令

收到此命令，重启，运行新版本固件。回调函数中不可执行太多耗时代码。

void iot_ota_upgrade_cb(void)

5.2 补丁升级

5.2.1 补丁信息

设备定时查询云端是否有新补丁发布，自动调用此函数。

void iot_ patchs_list_cb(iot_s32 count, patchs_list_t patchs[ ])

参数	说明
count	云端已发布，本设备未升级的补丁数量，范围 0-n
patchs	补丁信息结构体，如已发布多个补丁，将以列表形式展示

```
typedef struct patchs_list{
    char *      name;           //补丁名称
    char *      version;        //补丁版本
    iot_u32     total_len;      //补丁文件总大小
}patchs_list_t;
```

5.2.2 获取补丁

收到补丁列表后，请调用此函数，请求需要升级的补丁文件。

iot_s32 iot_patch_req(char name[16], char version[16])

参数	说明
name	欲升级的补丁名称
version	补丁版本，格式自定义
返回值	0 ： 成功，可在 5.3 节中收到补丁数据块

	-1 ： 失败，参数错误

5.2.3 接收补丁数据块

参考 5.3 节，chunk_stat 为补丁数据块的情况。

5.2.4 补丁状态上报

补丁升级结束后，调用此函数告知云端完成状态。

```
iot_s32  iot_patch_end (  const   char      name[16],
                          const   char      version[16],
                          DEV_STATUS_T   patch_state   )
```

参数	说明
name	升级补丁的名称
version	升级补丁的版本
patch_state	升级补丁的结束状态 DEV_STA_PATCH_FAILED（13） ： 失败 DEV_STA_PATCH_SUCCESS（14）： 成功

5.3 接收数据块接口

sdk 自动调用该函数，接受云端发来的数据块，收到后需更新本地固件或补丁。回调函数中不可执行太多耗时代码。

```
iot_s32  iot_chunk_cb(   iot_u8          chunk_stat,
                         iot_u32         chunk_offset,
```

iot_u32 chunk_size,

const iot_s8* chunk)

参数	说明
chunk_stat	高四位代表 chunk 的类型： 0：OTA 数据块； 1：补丁数据块 低四位代表 chunk 是否是最后一个分块： 0：不是最后一个分块； 1：是最后一个分块，固件传输结束
chunk_offset	本数据块相对于完整固件的偏移量 即，第一个数据块，此参数值是 0
chunk_size	本数据块的长度
chunk	固件数据块
返回值	0 ：写入固件块成功 -1：写入固件块失败

6 设备信息

6.1 请求设备信息

此函数用于获取设备相关信息，与 4.2 节“接收设备信息”配合使用。

iot_s32 iot_get_info (INFO_TYPE_T info_type)

参数	说明

info_type	INFO_TYPE_USER_MASTER：绑定用户的信息，包括用户 ID、手机号、邮箱等 INFO_TYPE_USER_SHARE ：分享用户的信息，包括用户 ID、手机号、邮箱等 INFO_TYPE_DEVICE ：设备的详细信息，包括绑定状态等

6.2 接收设备信息

void iot_info_cb (INFO_TYPE_T info_type, void * info)

参数	说明
info_type	参考 iot_get_info 函数中 info_type 的说明
info	根据 info_type 的值传入不同格式的数据

该函数用来处理接收的设备信息数据，接收的数据根据 info_type 的值数据格式不同

注意！回调函数中不可执行太多耗时代码。

① 当 info_type= INFO_TYPE_USER_MASTER 时，表示获取的是设备绑定用户的信息，数据结构体如下

```
typedef struct user_info {
    char *  id;                 //用户 id
    char *  email;              //用户 email
    char *  country_code;       //用户手机号的国家代码，比如中国是 0086
    char *  phone;              //用户手机号
    char *  name;               //用户昵称
```

```
}user_info_t;
```

② 当 info_type= INFO_TYPE_USER_SHARE 时，表示获取的是分享用户的信息，由于可能有多个分享用户，因此分享用户的数据是列表形式的，数据结构体如下

```
typedef struct share_info      {
    iot_s32          count;          //分享用户的数量
    user_info_t*     user;           //分享用户信息结构体的列表
}share_info_t;
```

③ 当 info_type= INFO_TYPE_DEVICE 时，表示获取的是设备信息，数据结构体如下

```
typedef struct dev_info   {
    iot_u8   is_bind;                //设备是否被绑定
    char     sdk_ver[6];             //SDK 版本号
}dev_info_t;
```

6.3 配置设备信息

注意！请在调用 iot_start 前调用此函数。

```
iot_s32   iot_config_info (     INFO_TYPE_T     info_type,    void * info        )
```

参数	说明
info_type	INFO_TYPE_ENCRYPT：通信链路加密算法选择 INFO_TYPE_SN：设置设备序列号信息
info	根据 info_type 的不同，传入不同参数数据 INFO _TYPE_ENCRYPT："AES"：AES 算法（默认）； "SSL"：SSL 算法；

	"SM4"：SM4 算法 INFO_TYPE_SN：自定义字符串，数字字母组合，最长 16 字节

① 当 info_type = INFO_TYPE_ENCRYPT 时，表示设置通信链路的加密算法，取值如下：

"AES"：AES 算法（默认）；

"SSL"：SSL 算法；(用 SSL 加密时，需要调用 void iot_ssl_load(void) 加载 SSL 库)

"SM4"：SM4 算法

例如：iot_config_info (INFO_TYPE_ENCRYPT, "SSL")；设置通信加密算法为 SSL

② 当 info_type = INFO_TYPE_SN 时，表示设置第三方序列号，厂商可设置自定义格式的设备唯一序列号，此功能可用于实现维修设备替换通信模块时，用户设备历史数据不变的功能。

注意！同一产品（product_id 相同）下，该序列号须保证唯一，如需更改 SN，请先删除设备。

例如：iot_config_info(INFO_TYP_SN, "SNDQ201811200003")；设置设备的序列号为 SNDQ201811200003，此序列号可唯一标识一个设备。

7 本地数据存储

具有文件系统的平台，会将数据保存成文件；不具有文件系统，将数据存至 flash。

7.1 保存数据至本地

iot_s32 iot_local_save (iot_u32 data_len, const void * data)

参数	说明
data_len	自定义数据长度，范围 1-4064
data	需要保存的自定义数据

返回值	0 ： 成功； -1：失败

7.2 加载本地数据

iot_s32 iot_local_load (iot_u32 data_len, void * data)

参数	说明
data_len	需要加载的数据长度，需小于实际保存的数据长度
data	需要加载的自定义数据
返回值	0 ： 成功； -1：失败，加载数据出错或 data_len 大于实际数据长度

7.3 清空本地数据

iot_s32 iot_local_reset (void)

参数	说明
返回值	0 ： 成功； -1：失败

8 子设备功能

8.1 子设备上线

① 添加子设备信息到发送队列

```
iot_s32  sub_dev_add ( const  char  sub_id[32],
                       const  char  sub_name[32],
                       const  char  sub_version[5],
```

iot_u16 sub_type)

参数	说明
sub_id	自定义的子设备 id，仅限字母数字组合。请自行维护唯一性，支持单个网关下唯一或产品维度下唯一（请根据实际需求填写）。
sub_name	子设备名称，仅限字母数字组合
sub_version	子设备固件版本，"xx.xx"，0≤x≤9
sub_type	子设备类型，请与 APP 端自行约定
返回值	0 ：成功 -1：失败

② 上线子设备，一次可同时上线多个子设备。

注：关于一次可以同时上线多少个子设备，代码中不做限制，但是和 iot_start 函数中的 sendbuf_size 参数大小有关。

iot_s32 iot_sub_dev_active (SUB_OPT_TYPE_T opt_type,

iot_u32* data_seq)

参数	说明
opt_type	子设备上线操作类型。 SUB_OPT_BIND_ONLINE ：执行绑定主设备+上线操作 SUB_OPT_ONLINE ：仅执行子设备上线操作
data_seq	传出参数，本条数据的序列号，如果需要确定数据何时上传成功，可记录此发送序列号，与收到的进行对比。

返回值	0 ： 成功； -1：失败

子设备上/下线操作类型定义如下：

```
typedef enum SUB_OPT_TYPE {
    SUB_OPT_BIND_ONLINE          = 0,  //执行绑定主设备+上线操作
    SUB_OPT_ONLINE               = 1,  //仅执行子设备上线操作
    SUB_OPT_OFFLINE              = 0,  //仅执行子设备下线操作
    SUB_OPT_UNBIND_OFFLINE       = 1,  //执行从主设备解绑+下线操作
}SUB_OPT_TYPE_T;
```

8.2 子设备离线

子设备离线时，调用此接口。

```
iot_s32   iot_sub_dev_inactive (   const    char          sub_id[32],
                                   SUB_OPT_TYPE_T    opt_type,
                                   iot_u32 *         data_seq    )
```

参数	说明
sub_id	自定义的子设备 id，仅限字母数字组合
opt_type	子设备下线操作类型。 SUB_OPT_OFFLINE ： 仅执行子设备下线操作

	SUB_OPT_UNBIND_OFFLINE ： 执行从主设备解绑+下线操作
data_seq	传出参数，本条数据的序列号，如果需要确定数据何时上传成功，可记录此发送序列号，与收到的进行对比。
返回值	0 ： 成功； -1：失败

8.3 子设备状态报告

当设备运行状态发生改变时，sdk 自动调用此函数。当调用 8.1 节子设备上线、8.2 节子设备下线后，此回调函数会被调用。当 sdk 收到子设备被动解绑通知（包括 app 解绑子设备，其他主设备绑定当前主设备下的子设备）时，此回调函数回避调用。回调函数中不可执行太多耗时代码。

```
void iot_sub_status_cb(   SUB_STATUS_T      sub_status,
                          iot_u32           timestamp,
                          DATA_LIST_T *     list ,
                          iot_u32           seq);
```

参数	说明
sub_status	子设备状态
timestamp	时间戳
list	子设备信息列表
seq	此发送序列号，与收到的进行对比。
返回值	0 ： 成功；

	-1：失败

```
typedef enum SUB_STATUS {
    SUB_STA_ONLINE              = 0,  //子设备上线结果
    SUB_STA_BIND_ONLINE         = 1,  //子设备绑定上线结果
    SUB_STA_OFFLINE             = 2,  //子设备下线结果
    SUB_STA_UNB_OFFLINE         = 3,  //子设备解绑下线结果
    SUB_STA_FORCED_UNBIND       = 4,  //子设备被强制解绑
}SUB_STATUS_T;
```