阿里云
物联网边缘计算

边缘开发指南

文档版本：20181218

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
	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。	 **禁止：** 重置操作将丢失用户配置数据。
	该类警示信息可能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，恢复业务所需时间约10分钟。
	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。	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。	单击 **确定**。
`courier` 字体	命令。	执行 `cd /d C:/windows` 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。	`swich` *`{stand | slave}`*

目录

1 设备接入SDK

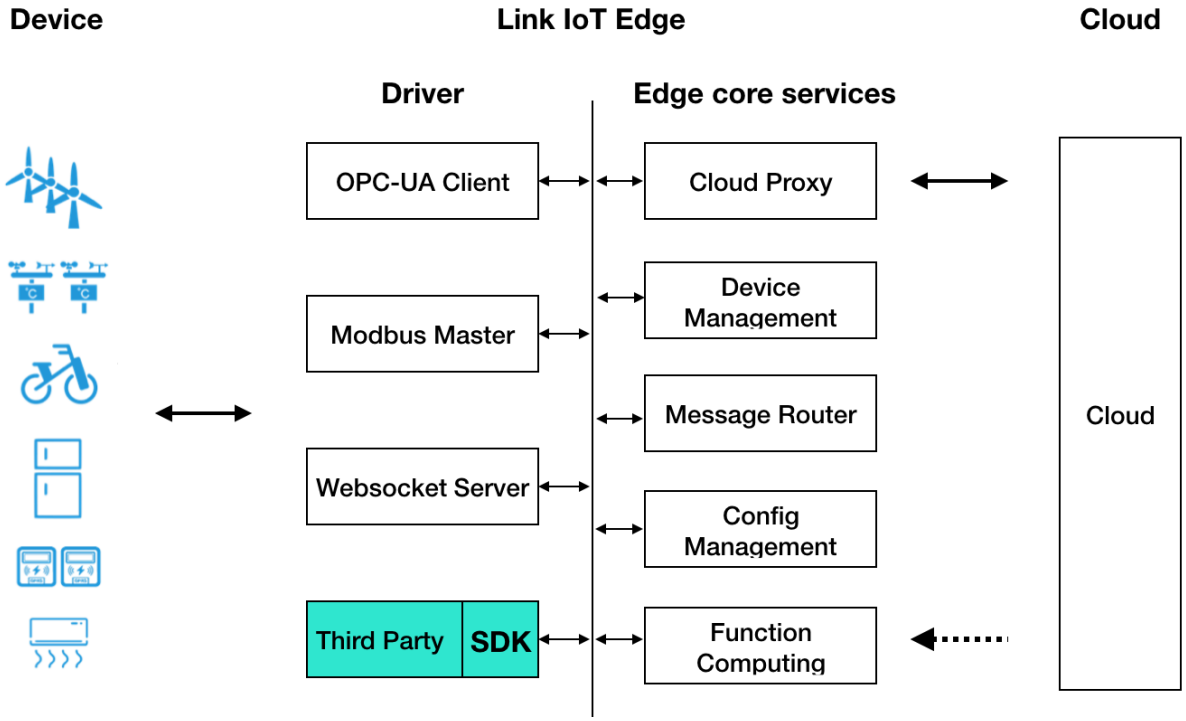
1.1 开发简介

本章为您介绍设备接入SDK及其开发流程。

设备接入

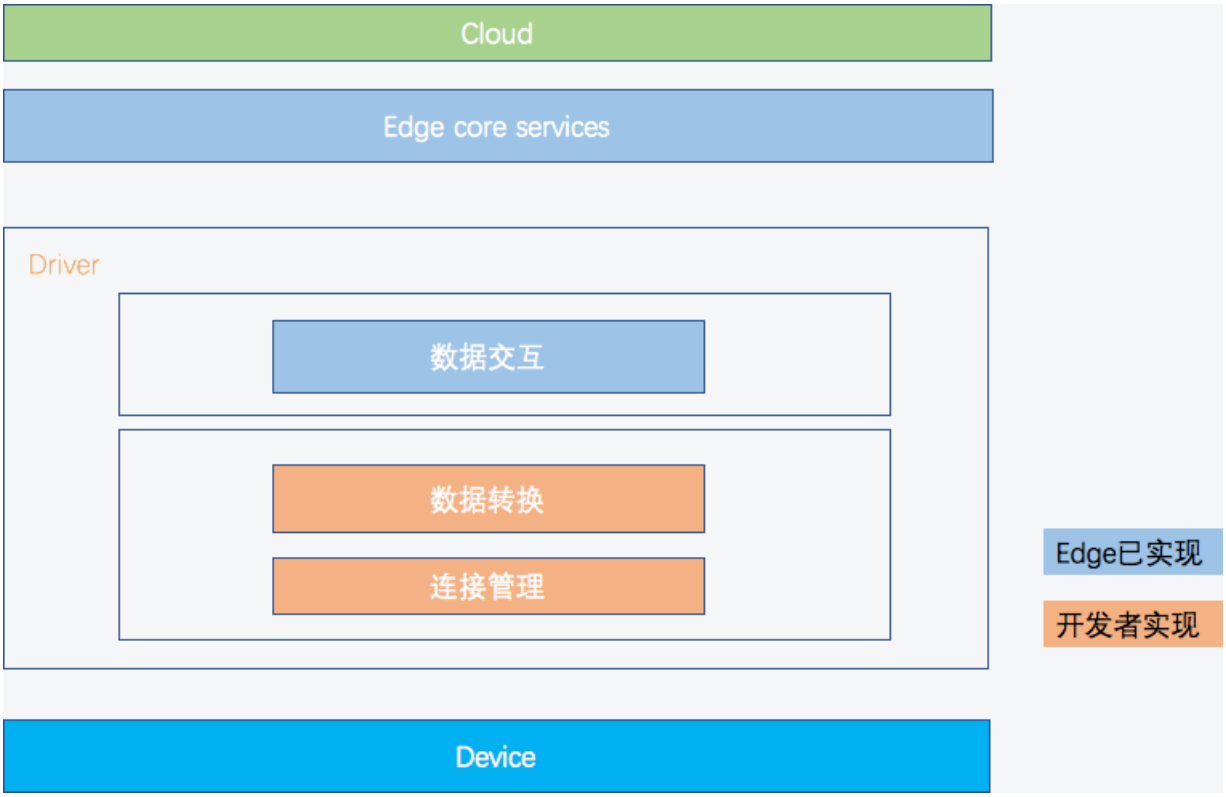
Link IoT Edge提供多种设备接入方式。Link IoT Edge内置支持常用的行业协议，如OPC-UA Client、Modbus Master和通用协议WebSocket Server，同时也提供SDK方便开发者做定制开发。

设备接入模块在Link IoT Edge中也称为驱动（driver）或设备驱动。Link IoT Edge的硬件载体称为边缘网关。



驱动的组成

驱动（设备接入模块）由设备的连接管理、设备的数据（协议）转换和设备的数据交互三个模块组成。如下图所示：



- 连接管理

  指设备与边缘网关建立通信连接。如，采用标准的网络协议（HTTP、HTTPS、WebSocket）、行业协议（Modbus、OPC-UA）和私有协议（基于TCP/UDP定制的私有协议）。

- 数据转换

  指将设备的数据转换为符合阿里云物联网平台物模型规范的数据格式。阿里云物联网平台物模型规范请参考物模型。

- 数据交互

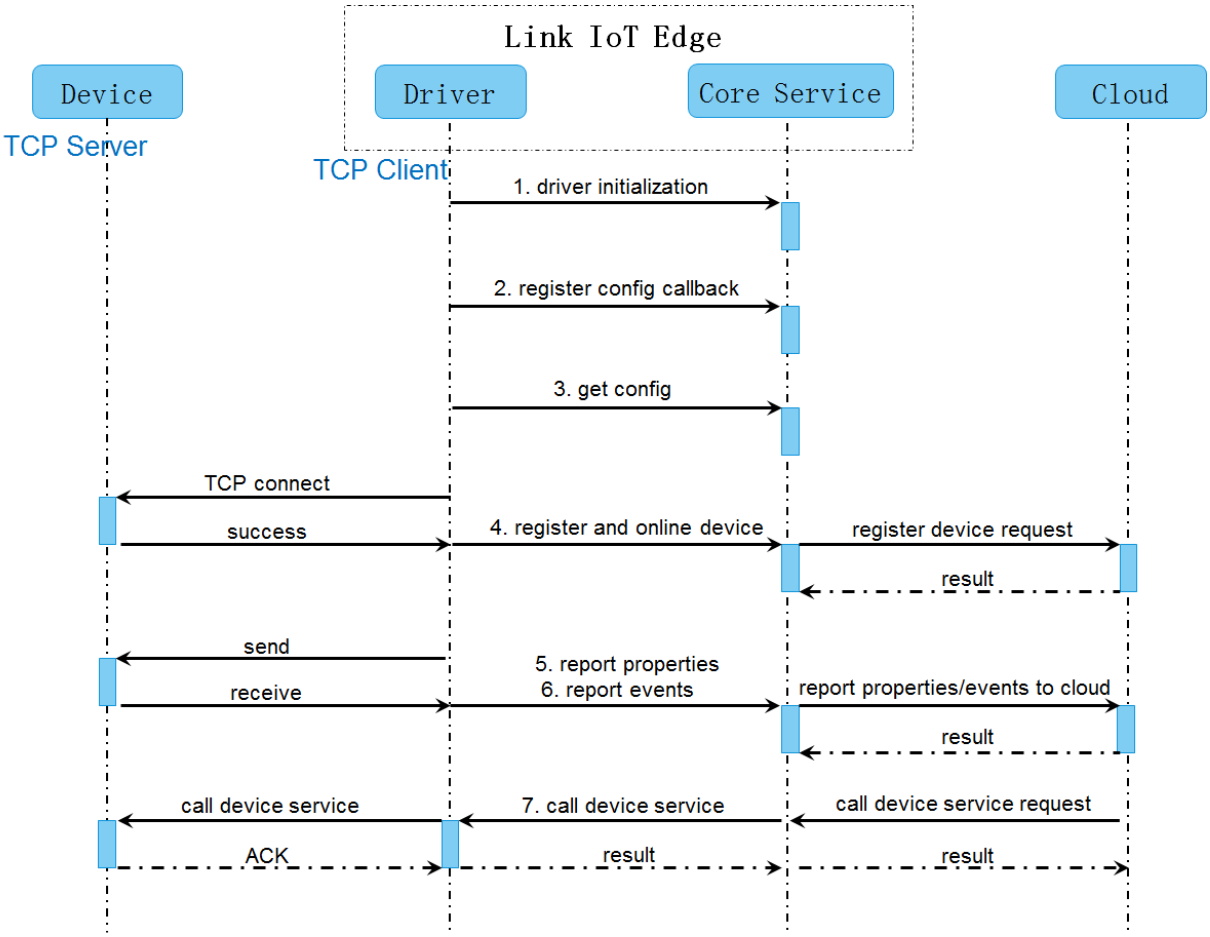
  指设备通过Link IoT Edge与阿里云物联网平台之间的数据交互。

开发流程

开发者需要实现连接管理和数据转换两部分，数据交互通过注册callback即可完成。开发流程图如下：

以温度计为例说明开发流程，该温度计内置TCP Server并提供TCP协议的温度读写接口。

> **说明：**
> 在开发驱动前需要在控制台上完成产品和设备创建工作，详情请参考创建产品(基础版)、创建产品(高级版)和创建设备。



操作步骤如下：

1. 驱动通过设备接入SDK提供的环境变量，获取温度计TCP Server的IP和Port，TCP Server的IP和Port需要根据驱动配置制定。
2. 驱动与温度计建立TCP连接，连接成功后调用设备接入SDK的注册上线接口，驱动即可连接到温度计。
3. 驱动通过TCP协议读取到当前温度，调用设备接入SDK的上报属性接口完成数据转换。

   根据业务逻辑，如果温度高于某个阈值，调用设备接入SDK封装的上报事件接口完成数据转换。
4. 驱动通过回调函数处理云端或其他设备数据。
5. 参考消息路由配置，完成配置后，即可在云端查看到设备上报的属性或者事件。

驱动配置

驱动配置以json格式存储在Link IoT Edge的核心服务模块配置中心。把驱动配置和驱动包解耦，目的是为了在不同的环境中复用驱动。

以温度计驱动为例，温度计驱动在运行时，通过设备接入SDK API从配置中心读取温度计设备TCP Server的IP和Port，进而与温度计设备建立连接。驱动配置在项目实施时在物联网平台进行配置，详情可参考自定义驱动。边缘实例部署后，物联网平台会将配置文件下发到Link IoT Edge的配置中心，进而传递给对应的驱动。

配置文件格式定义如下：

```
{
    "deviceList": [{
        "custom": "{\"ip\":\"192.168.1.1\",\"port\":12345}", //设备特有的配置信息，设备界面输入自定义配置
        "deviceName": "dn", //使用驱动的设备名
        "productKey": "pk", //使用驱动的产品信息
    }],
    "driverType": 1 // 1表示自定义驱动
}
```

参数说明如下：

字段名称	字段含义
deviceList	设备列表，从控制台获取。
custom	您的自定义信息，需要自定义参数和参数值。
deviceName	设备名称，从控制台获取。
productKey	产品唯一标识符，从控制台获取。
driverType	驱动类型，取值为1，表示自定义驱动。

第三方库依赖

针对设备接入驱动开发可能会涉及第三方库依赖问题，Link IoT Edge驱动根据不同开发语言特性，分别制定了每种开发语言的第三方库依赖规则。

 说明：

当前Link IoT Edge提供的Node.js和Python版本的SDK，不支持依赖库中包含so文件。

- C版本SDK：

  使用设备接入SDK C版本开发驱动时，若依赖第三方库，则第三方库必须和驱动的Binary在同一目录层级，不允许随意放置第三方库。

- Node.js版本SDK：

使用设备接入SDK Node.js版本开发驱动时，若依赖第三方库，可在项目根目录使用如下命令安装依赖。

```
npm install {第三方库名}
```

此时，依赖会被安装到项目根目录的*node_modules*目录下。当发布驱动时，将整个驱动项目打包并上传。

- Python版本SDK：

  使用设备接入SDK Python版本开发驱动时，若依赖第三方库，需要在驱动目录使用如下命令，将第三方库与驱动安装在同级目录。

  ```
  pip install -t . {第三方库名}
  ```

驱动文件打包说明

基于Link IoT Edge提供的SDK开发驱动并完成调试后，需将产物打包为*.zip*包，并确保驱动Binary或index源文件在*.zip*包第一级目录，以备后续控制台创建自定义驱动时使用。

每个版本SDK开发的驱动在打包时，有不同的打包规则：

- 基于C SDK开发的驱动：驱动Binary需要命名为main。
- 基于Python SDK或Node.js SDK开发的驱动：驱动包文件中须包含*index.py*或*index.js*，并且在该文件中定义handler函数。

1.2 C版本SDK

本章为您介绍C版本的SDK使用方法及相关API。Link IoT Edge提供C版本的SDK，名称为*link-iot-edge-access-sdk-c*。C版本开源的SDK源码请见开源C库。

get_properties_callback

```
/*
     * 获取属性的回调函数, Link IoT Edge 需要获取某个设备的属性时, SDK 会调用该接口间接获取到数据并封装成固定格式后回传给 Link IoT Edge.
     * 开发者需要根据设备id和属性名找到设备, 将属性值获取并以@device_data_t格式返回.
     *
     * @dev_handle:          Link IoT Edge 需要获取属性的具体某个设备.
     * @properties:          开发者需要将属性值更新到properties中.
     * @properties_count:    属性个数.
     * @usr_data:            注册设备时, 用户传递的私有数据.
     * 所有属性均获取成功则返回LE_SUCCESS, 其他则返回错误码(参考错误码宏定义).
     * */
typedef int (*get_properties_callback)(device_handle_t dev_handle,
```

```
                                        leda_device_data_t properties
[],
                                        int properties_count,
                                        void *usr_data);
```

set_properties_callback

```
/*
     * 设置属性的回调函数, Link IoT Edge 需要设置某个设备的属性时, SDK 会调用
该接口将具体的属性值传递给应用程序, 开发者需要在本回调
     * 函数里将属性设置到设备.
     *
     * @dev_handle:         Link IoT Edge 需要设置属性的具体某个设备.
     * @properties:         Link IoT Edge 需要设置的设备的属性名和值.
     * @properties_count:   属性个数.
     * @usr_data:           注册设备时, 用户传递的私有数据.
     *
     * 若获取成功则返回LE_SUCCESS, 失败则返回错误码(参考错误码宏定义).
     * */
typedef int (*set_properties_callback)(device_handle_t dev_handle,
                                        const leda_device_data_t
properties[],
                                        int properties_count,
                                        void *usr_data);
```

call_service_callback

```
/*
     * 服务调用的回调函数, Link IoT Edge 需要调用某个设备的服务时, SDK 会调用
该接口将具体的服务参数传递给应用程序, 开发者需要在本回调
     * 函数里调用具体的服务, 并将服务返回值按照设备 tsl 里指定的格式返回.
     *
     * @dev_handle:   Link IoT Edge 需要调用服务的具体某个设备.
     * @service_name: Link IoT Edge 需要调用的设备的具体某个服务名.
     * @data:         Link IoT Edge 需要调用的设备的具体某个服务参数, 参数与
设备 tsl 中保持一致.
     * @data_count:   Link IoT Edge 需要调用的设备的具体某个服务参数个数.
     * @output_data:  开发者需要将服务调用的返回值, 按照设备 tsl 中规定的服务
格式返回到output中.
     * @usr_data:     注册设备时, 用户传递的私有数据.
     *
     * 若获取成功则返回LE_SUCCESS, 失败则返回错误码(参考错误码宏定义).
     * */
typedef int (*call_service_callback)(device_handle_t dev_handle,
                                     const char *service_name,
                                     const leda_device_data_t data[],
                                     int data_count,
                                     leda_device_data_t output_data
[],
                                     void *usr_data);
```

leda_init

```
/*
```

```
 * 模块初始化, 模块内部会创建工作线程池, 异步执行云端下发的指令, 工作线程数目通过
worker_thread_nums配置.
 *
 * module_name:          模块名称.
 * worker_thread_nums : 线程池工作线程数.
 *
 * 阻塞接口, 成功返回LE_SUCCESS, 失败返回错误码.
 */
int leda_init(const char *module_name, int worker_thread_nums);
```

config_changed_callback

```
/*
 * 设备配置变更回调.
 *
 * module_name:   模块名称.
 * config:        配置信息.
 *
 * 阻塞接口, 成功返回LE_SUCCESS, 失败返回错误码.
 */
typedef int (*config_changed_callback)(const char *module_name, const
char *config);
```

leda_register_config_changed_callback

```
/*
 * 注册设备配置变更回调.
 *
 * module_name:    模块名称.
 * config_cb:      配置变更通知回调接口.
 *
 * 阻塞接口, 成功返回LE_SUCCESS,失败返回错误码.
 */
int leda_register_config_changed_callback(const char *module_name,
config_changed_callback config_cb);
```

leda_register_and_online_by_device_name

```
/*
 * 通过已在云平台注册的device_name, 注册设备并上线设备, 申请设备唯一标识符.
 *
 * 设备默认注册后即上线.
 *
 * product_key: 产品pk.
 * device_name: 云平台注册的device_name, 可以通过配置文件导入.
 * device_cb:   设备回调函数结构体，详细描述@leda_device_callback.
 * usr_data:    设备注册时传入私有数据, 在回调中会传给设备.
 *
 * 阻塞接口, 返回值设备在Link IoT Edge本地唯一标识, >= 0表示有效, < 0 表示无
效.
 *
 */
```

```
device_handle_t leda_register_and_online_by_device_name(const char
 *product_key, const char *device_name, leda_device_callback_t *
device_cb, void *usr_data);
```

leda_register_and_online_by_local_name

```
/*
 * 注册设备并上线设备, 申请设备唯一标识符, 若需要注册多个设备, 则多次调用该接口即
可.
 *
 * 设备默认注册后即上线.
 *
 * product_key: 产品pk.
 * local_name:  由设备特征值组成的唯一描述信息, 必须保证同一个product_key
时，每个待接入设备名称不同.
 * device_cb:   设备回调函数结构体，详细描述@leda_device_callback.
 * usr_data:    设备注册时传入私有数据, 在回调中会传给设备.
 *
 * 阻塞接口, 返回值设备在Link IoT Edge本地唯一标识, >= 0表示有效, < 0 表示无
效.
 *
 */
device_handle_t leda_register_and_online_by_local_name(const char *
product_key, const char *local_name, leda_device_callback_t *device_cb
, void *usr_data);
```

leda_online

```
/*
 * 上线设备, 设备只有上线后, 才能被 Link IoT Edge 识别.
 *
 * dev_handle:  设备在Link IoT Edge本地唯一标识.
 *
 * 阻塞接口,成功返回LE_SUCCESS, 失败返回错误码.
 */
int leda_online(device_handle_t dev_handle);
```

leda_offline

```
/*
 * 下线设备, 假如设备工作在不正常的状态或设备退出前, 可以先下线设备, 这样Link IoT
 Edge就不会继续下发消息到设备侧.
 *
 * dev_handle:  设备在Link IoT Edge本地唯一标识.
 *
 * 阻塞接口, 成功返回LE_SUCCESS,  失败返回错误码.
 *
 */
int leda_offline(device_handle_t dev_handle);
```

leda_report_properties

```
/*
 * 上报属性, 设备具有的属性在设备能力描述 tsl 里有规定.
```

```
 *
 * 上报属性, 可以上报一个, 也可以多个一起上报.
 *
 * dev_handle:           设备在Link IoT Edge本地唯一标识.
 * properties:           @leda_device_data_t, 属性数组.
 * properties_count:     本次上报属性个数.
 *
 * 阻塞接口, 成功返回LE_SUCCESS,  失败返回错误码.
 *
 */
int leda_report_properties(device_handle_t dev_handle, const
leda_device_data_t properties[], int properties_count);
```

leda_report_event

```
/*
 * 上报事件, 设备具有的事件上报能力在设备 tsl 里有约定.
 *
 *
 * dev_handle:  设备在Link IoT Edge本地唯一标识.
 * event_name:  事件名称.
 * data:        @leda_device_data_t, 事件参数数组.
 * data_count:  事件参数数组长度.
 *
 * 阻塞接口, 成功返回LE_SUCCESS,  失败返回错误码.
 *
 */
int leda_report_event(device_handle_t dev_handle, const char *
event_name, const leda_device_data_t data[], int data_count);
```

leda_get_tsl_size

```
/*
 * 获取配置相关信息长度.
 *
 * product_key:   产品pk.
 *
 * 阻塞接口, 成功返回product_key对应的tsl长度, 失败返回0.
 */
int leda_get_tsl_size(const char *product_key);
```

leda_get_tsl

```
/*
 * 获取配置相关信息.
 *
 * product_key:  产品pk.
 * tsl:          输出物模型, 需要提前申请好内存传入.
 * size:         tsl长度, 该长度通过leda_get_tsl_size接口获取, 如果传入tsl
比实际配置内容长度短, 会返回LE_ERROR_INVAILD_PARAM.
 *
 * 阻塞接口, 成功返回LE_SUCCESS, 失败返回错误码.
 */
```

```
int leda_get_tsl(const char *product_key, char *tsl, int size);
```

leda_get_config_size

```
/*
 * 获取配置信息长度.
 *
 * module_name:  模块名称
 *
 * 阻塞接口, 成功返回config长度, 失败返回0.
 */
int leda_get_config_size(const char *module_name);
```

leda_get_config

```
/*
 * 获取配置信息.
 *
 * module_name:  模块名称
 * config:       配置信息, 需要提前申请好内存传入.
 * size:         config长度, 该长度通过leda_get_config_size接口获取, 如果传
入config比实际配置内容长度短, 会返回LE_ERROR_INVAILD_PARAM.
 *
 * 阻塞接口, 成功返回LE_SUCCESS, 失败返回错误码.
 */
int leda_get_config(const char *module_name, char *config, int size);
```

leda_get_device_handle

```
/*
 * 获取设备句柄.
 *
 * product_key: 产品pk.
 * device_name: 设备名称.
 *
 * 阻塞接口, 成功返回device_handle_t, 失败返回小于0数值.
 */
device_handle_t leda_get_device_handle(const char *product_key, const
char *device_name);
```

leda_exit

```
/*
 * 模块退出.
 *
 * 模块退出前, 释放资源.
 *
 * 阻塞接口.
 */
void leda_exit(void);
```

leda_get_module_name

```
/*
```

```
 * 获取驱动模块名称.
 *
 * 阻塞接口, 成功返回驱动模块名称, 失败返回NULL.
 */
char* leda_get_module_name(void);
```

1.3 Nodejs版本SDK

设备接入SDK用于您在边缘计算节点上开发驱动，将设备连接到边缘计算节点，进而连接到物联网平台。设备接入SDK分为Node.js版本和Python版本，Node.js版本开源的SDK源码请见开源的*Node.js*库。

Node.js版本设备接入SDK使用说明

使用设备接入SDK前需要执行**`npm install linkedge-thing-access-sdk`**命令，安装SDK。

设备接入SDK的使用格式如下：

```
const {
    ThingAccessClient
} = require('linkedge-thing-access-sdk');
```

ThingAccessClient

设备接入客户端，您可以通过该客户端来主动上报设备属性或事件，也可被动接受云端下发的指令。

常量定义

名称	类型	描述
PRODUCT_KEY	String	配置对象（传给ThingAccessClient构造函数）的键值，指定云端分配的productKey。
DEVICE_NAME	String	配置对象（传给ThingAccessClient构造函数）的键值，指定云端分配的deviceName。
LOCAL_NAME	String	配置对象（传给ThingAccessClient构造函数）的键值，指定本地自定义的设备名。
RESULT_SUCCESS	Number	操作成功。用于回调函数返回状态码。
RESULT_FAILURE	Number	操作失败。用于回调函数返回状态码。
CALL_SERVICE	String	回调对象（传给ThingAccessClient构造函数）的键值，指定调用设备服务回调函数。
GET_PROPERTIES	String	回调对象（传给ThingAccessClient构造函数）的键值，指定获取设备属性回调函数。

名称	类型	描述
SET_PROPERTIES	String	回调对象（传给ThingAccessClient构造函数）的键值，指定设置设备属性回调函数。

ThingAccessClient(config, callbacks)

功能介绍

构造函数，使用指定的config和callbacks构造。

请求参数

名称	类型	描述
config	Object	元数据，用于配置该客户端。取值格式为： `{ "productKey": "Your Product Key", "deviceName": "Your Device Name" }`
callbacks	Object	回调函数对象。 • 指定设置设备属性的回调参数，请参见*callbacks.setProperties* • 指定获取设备属性的回调参数，请参见*callbacks.getProperties* • 指定调用设备服务的回调参数，请参见*callbacks.callService*

回调说明

```
callbacks: {
  setProperties: function(properties) {},
  getProperties: function(keys) {},
  callService: function(name, args) {}
}
```

callbacks.setProperties

功能介绍

设置具体设备属性的回调函数。通过回调函数，实现设置设备的属性。

请求参数

名称	类型	描述
properties	Object	设置属性的对象，取值格式为： `{ "key1": "value1", "key2": "value2" }`

callbacks.getProperties

功能介绍

获取具体设备属性的回调函数。通过回调函数，实现获取设备的属性。

请求参数

名称	类型	描述
keys	String[]	获取属性对应的名称，取值格式为： `['key1', 'key2']`

callbacks.callService

功能介绍

调用设备服务回调函数。通过回调函数，实现调动设备服务。

请求参数

名称	类型	描述
name	String	设备服务名称。
args	Object	服务入参列表，取值格式为： `{ "key1": "value1", "key2": "value2" }`

setup() -> {Promise<Void>}

功能介绍

设备接入客户端初始化。该接口必须调用。

返回参数

```
Promise<Void>
```

registerAndOnline() -> {Promise<Void>}

功能介绍

将设备注册到边缘节点中并通知边缘节点上线设备。设备需要注册并上线后，设备端才能收到云端下发的指令或者发送数据到云端。

返回参数

```
Promise<Void>
```

reportEvent(eventName, args)

功能介绍

主动上报设备事件。

请求参数

名称	类型	描述
eventName	String	事件对应的名称，与您在产品定义中创建事件的名称一致。
args	Object	事件中包含的属性key与value，取值格式为： `{ "key1": "value1", "key2": "value2" }`

reportProperties(properties)

功能介绍

主动上报设备属性。

请求参数

名称	类型	描述
properties	Object	属性中包含的属性key与value，取值格式为： `{ "key1": "value1", "key2": "value2"`

名称	类型	描述
		}

cleanup() -> {Promise<Void>}

功能介绍

资源回收接口，您可以使用该接口回收您的资源。

返回参数

```
Promise<Void>
```

getTsl() -> {Promise<Void>}

功能介绍

返回TSL字符串，数据格式与云端一致。

返回参数

```
Promise<Void>
```

unregister() -> {Promise<Void>}

功能介绍

从边缘计算节点移除设备。请谨慎使用该接口。

返回参数

```
Promise<Void>
```

online() -> {Promise<Void>}

功能介绍

通知边缘节点设备上线，该接口一般在设备离线后再次上线时使用。

返回参数

```
Promise<Void>
```

offline() -> {Promise<Void>}

功能介绍

通知边缘节点设备已离线。

返回参数

```
Promise<Void>
```

1.4 Python版本SDK

本章为您介绍Python版本的SDK使用方法及相关API。Link IoT Edge提供Python版本的SDK，名称为*lethingaccesssdk*。Python版本开源的SDK源码请见开源的*Python*库。

Python版本设备接入SDK使用说明

在使用设备接入SDK前需要在您的代码库中导入**import lethingaccesssdk**。

ThingCallback

首先根据真实设备，命名一个类（如Demo_device）继承ThingCallback，然后在该类（Demo_device）中实现*setProperties*、*getProperties*和*callService*三个函数。

setProperties

功能介绍

设置具体设备属性函数。

请求参数

名称	类型	描述
properties	dict	设置属性，取值格式为： `{ "property1": "value1", "property2": "value2" }`

返回参数

名称	类型	描述
code	int	若获取成功则返回LEDA_SUCCESS，失败则返回错误码。
output	dict	数据内容自定义，例如： `{ "key1": xxx, "key2": yyy, ...`

名称	类型	描述
		} 若无返回数据，则值为空{}。

getProperties

功能介绍

获取具体设备属性的函数。

请求参数

名称	类型	描述
keys	list	获取属性对应的名称，取值格式为： `['key1', 'key2']`

返回参数

名称	类型	描述
code	int	若获取成功则返回LEDA_SUCCESS，失败则返回错误码。
output	dict	返回值，例如： `{ 'property1': xxx, 'property2': yyy, ..}. }`

callService

功能介绍

调用设备服务函数。

请求参数

名称	类型	描述
name	str	设备服务名称。
args	dict	服务入参列表，取值格式为： `{ "key1": "value1", "key2": "value2"`

名称	类型	描述
		}

返回参数

名称	类型	描述
code	int	若获取成功则返回LEDA_SUCCESS，失败则返回错误码。

output	dict	数据内容自定义，例如：
		{
		"key1": xxx,
		"key2": yyy,
		...
		}
		若无返回数据，则值为空{}。

ThingAccessClient(config)

功能介绍

设备接入客户端，您可以通过该客户端来主动上报设备属性或事件，也可被动接受云端下发的指令。

请求参数

名称	类型	描述

config	dict	包括云端分配的ProductKey和deviceName。例如，
		{
		"productKey": "xxx",
		"deviceName": "yyy"
		}

registerAndOnline(ThingCallback)

功能介绍

将设备注册到边缘节点中并通知边缘节点上线设备。设备需要注册并上线后，设备端才能收到云端下发的指令或者发送数据到云端。

请求参数

名称	类型	描述
ThingCallback	object	设备的callback对象。

reportEvent(eventName, args)

功能介绍

主动上报设备事件。

请求参数

名称	类型	描述
eventName	str	事件对应的名称，与您在产品定义中创建事件的名称一致。
args	dict	事件中包含的属性key与value，取值格式为： `{ "key1": "value1", "key2": "value2" }`

reportProperties(properties)

功能介绍

主动上报设备属性。

请求参数

名称	类型	描述
properties	dict	属性中包含的属性key与value，取值格式为： `{ "key1": "value1", "key2": "value2" }`

getTsl()

功能介绍

返回TSL字符串，数据格式与云端一致。

返回参数

TSL字符串

online()

功能介绍

通知边缘节点设备上线，该接口一般在设备离线后再次上线时使用。

offline()

功能介绍

通知边缘节点设备已离线。

unregister()

功能介绍

移除设备和边缘节点的绑定关系。请谨慎使用该接口。

2 设备接入SDK综合示例

本文展示一段使用设备接入SDK，开发驱动的代码片段。设备使用thing标识，拥有一个temperature的只读属性，在连接到边缘计算节点后，每隔2秒向平台上报属性和高温事件。

C版本

```
/*
 * Copyright (c) 2014-2018 Alibaba Group. All rights reserved.
 * License-Identifier: Apache-2.0
 *
 * Licensed under the Apache License, Version 2.0 (the "License"); you
 may
 * not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *    http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT
 * WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 *
 */

#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <string.h>
#include <signal.h>
#include <pthread.h>
#include <cJSON.h>
#include <unistd.h>
#include <sys/prctl.h>

#include "log.h"
#include "le_error.h"
#include "leda.h"

#ifdef __cplusplus
extern "C" {
#endif

#define LED_TAG_NAME            "demo_led"
#define MAX_DEVICE_NUM          32

static int g_dev_handle_list[MAX_DEVICE_NUM] =
{
    INVALID_DEVICE_HANDLE
};
static int  g_dev_handle_count = 0;

static int  g_LedRunState      = 0;

static int get_properties_callback_cb(device_handle_t dev_handle,
                                      leda_device_data_t properties[],
```

```
                                int properties_count,
                                void *usr_data)
{
    int i = 0;

    for (i = 0; i < properties_count; i++)
    {
        log_i(LED_TAG_NAME, "get_property %s: ", properties[i].key);

        if (!strcmp(properties[i].key, "temperature"))
        {
            sprintf(properties[i].value, "%d", 30); /* 作为演示，填写模拟
数据 */
            properties[i].type = LEDA_TYPE_INT;
            log_i(LED_TAG_NAME, "%s\r\n",  properties[i].value);
        }
    }

    return LE_SUCCESS;
}

static int set_properties_callback_cb(device_handle_t dev_handle,
                                const leda_device_data_t properties[],
                                int properties_count,
                                void *usr_data)
{
    int i = 0;

    for (i = 0; i < properties_count; i++)
    {
        log_i(LED_TAG_NAME, "set_property type:%d %s: %s\r\n",
properties[i].type, properties[i].key, properties[i].value);
    }

    return LE_SUCCESS;
}

static int call_service_callback_cb(device_handle_t dev_handle,
                                const char *service_name,
                                const leda_device_data_t data[],
                                int data_count,
                                leda_device_data_t output_data[],
                                void *usr_data)
{
    int i = 0;

    log_i(LED_TAG_NAME, "service_name: %s\r\n", service_name);

    for (i = 0; i < data_count; i++)
    {
        log_i(LED_TAG_NAME, "    input_data %s: %s\r\n", data[i].key,
data[i].value);
    }

    output_data[0].type = LEDA_TYPE_INT;
    sprintf(output_data[0].key, "%s", "temperature");
    sprintf(output_data[0].value, "%d", 50); /* 作为演示，填写模拟数据 */

    return LE_SUCCESS;
}
```

```
static int parse_driverconfig_and_onlinedevice(const char* driver_con
fig)
{
    char                        *productKey     = NULL;
    char                        *deviceName     = NULL;
    cJSON                       *root           = NULL;
    cJSON                       *object         = NULL;
    cJSON                       *item           = NULL;
    cJSON                       *result         = NULL;
    leda_device_callback_t      device_cb;
    device_handle_t             dev_handle      = -1;

    root = cJSON_Parse(driver_config);
    if (NULL == root)
    {
        log_e(LED_TAG_NAME, "driver_config parser failed\r\n");
        return LE_ERROR_INVAILD_PARAM;
    }

    object = cJSON_GetObjectItem(root, "deviceList");
    cJSON_ArrayForEach(item, object)
    {
        if (cJSON_Object == item->type)
        {
            result = cJSON_GetObjectItem(item, "productKey");
            productKey = result->valuestring;

            result = cJSON_GetObjectItem(item, "deviceName");
            deviceName = result->valuestring;

            /*  parse user custom config
                因为本demo主要介绍驱动开发流程和SDK接口的使用，所以并没有完整的
实现设备和驱动的连接。实际应用场景中这部分是
                必须要实现的，要实现驱动和设备的连接，开发者可以通过在设备关联驱
动配置流程时通过自定义配置来配置设备连接信息，
                操作流程可以参考如下链接
                https://help.aliyun.com/document_detail/85236.html?spm
=a2c4g.11186623.6.583.64bd3265NR9Ouo

                如果用户在自定义配置中填写内容为
                {
                    "ip" : "192.168.1.199",
                    "port" 9999
                }

                则实际生成结果为
                "custom" : {
                    "ip" : "192.168.1.199",
                    "port" 9999
                }

                下面给出解析自定义配置的示例代码，具体如下
                cJSON          *custom = NULL;
                char           *ip     = NULL;
                unsigned short port    = 65536;

                custom = cJSON_GetObjectItem(item, "custom");
                result = cJSON_GetObjectItem(custom, "ip");
                ip = result->valuestring;

                result = cJSON_GetObjectItem(custom, "port");
```

```
                port = (unsigned short)result->valueint;

                获取到设备ip和port后即可连接该设备，连接成功则注册上线设备，否则
继续下一个配置的解析
            */

            /* register and online device */
            device_cb.get_properties_cb           = get_proper
ties_callback_cb;
            device_cb.set_properties_cb           = set_proper
ties_callback_cb;
            device_cb.call_service_cb             = call_servi
ce_callback_cb;
            device_cb.service_output_max_count    = 5;

            dev_handle = leda_register_and_online_by_local_name(
productKey, deviceName, &device_cb, NULL);
            if (dev_handle < 0)
            {
                log_e(LED_TAG_NAME, "product:%s device:%s register
failed\r\n", productKey, deviceName);
                continue;
            }
            g_dev_handle_list[g_dev_handle_count++] = dev_handle;
            log_i(LED_TAG_NAME, "product:%s device:%s register success
\r\n", productKey, deviceName);
        }
    }

    if (NULL != root)
    {
        cJSON_Delete(root);
    }

    if (0 == g_dev_handle_count)
    {
        log_e(LED_TAG_NAME, "current driver no device online\r\n");
    }

    return LE_SUCCESS;
}

static void *thread_device_data(void *arg)
{
    int i = 0;

    prctl(PR_SET_NAME, "demo_led_data");
    while (0 != g_LedRunState)
    {
        for (i = 0; i < g_dev_handle_count; i++)
        {
            /* report device properties */
            leda_device_data_t dev_proper_data[1] =
            {
                {
                    .type  = LEDA_TYPE_INT,
                    .key   = {"temperature"},
                    .value = {"20"}
                }
            };
            leda_report_properties(g_dev_handle_list[i], dev_proper
_data, 1);
```

```
            /* report device event */
            leda_device_data_t dev_event_data[1] =
            {
                {
                    .type  = LEDA_TYPE_INT,
                    .key   = {"temperature"},
                    .value = {"50"}
                }
            };
            leda_report_event(g_dev_handle_list[i], "high_temperature
", dev_event_data, 1);
        }

        sleep(5);
    }

    pthread_exit(NULL);

    return NULL;
}

static void ledSigInt(int sig)
{
    if (sig && (SIGINT == sig))
    {
        if (0 != g_LedRunState)
        {
            log_e(LED_TAG_NAME, "Caught signal: %s, exiting...\r\n",
strsignal(sig));
        }

        g_LedRunState = -1;
    }

    return;
}

int main(int argc, char** argv)
{
    int         size            = 0;
    char        *driver_config  = NULL;
    pthread_t   thread_id;
 struct      sigaction sig_int;

    /* listen SIGINT singal */
 memset(&sig_int, 0, sizeof(struct sigaction));
    sigemptyset(&sig_int.sa_mask);
    sig_int.sa_handler = ledSigInt;
    sigaction(SIGINT, &sig_int, NULL);

    /* set log level */
    log_init(LED_TAG_NAME, LOG_FILE, LOG_LEVEL_DEBUG, LOG_MOD_VERBOSE
);

    log_i(LED_TAG_NAME, "demo startup\r\n");

    /* init sdk */
    if (LE_SUCCESS != leda_init(leda_get_module_name(), 5))
    {
        log_e(LED_TAG_NAME, "leda_init failed\r\n");
        return LE_ERROR_UNKNOWN;
```

```
    }

    /* take driver config from config center */
    size = leda_get_config_size(leda_get_module_name());
    if (size >0)
    {
        driver_config = (char*)malloc(size);
        if (NULL == driver_config)
        {
            return LE_ERROR_ALLOCATING_MEM;
        }

        if (LE_SUCCESS != leda_get_config(leda_get_module_name(),
driver_config, size))
        {
            return LE_ERROR_INVAILD_PARAM;
        }
    }

    /* parse driver config and online device */
    if (LE_SUCCESS != parse_driverconfig_and_onlinedevice(driver_con
fig))
    {
        log_e(LED_TAG_NAME, "parse driver config or online device
failed\r\n");
        return LE_ERROR_UNKNOWN;
    }
    free(driver_config);

    /* report device data */
    if (0 != pthread_create(&thread_id, NULL, thread_device_data, NULL
))
    {
        log_e(LED_TAG_NAME, "create device data thread failed\r\n");
        return LE_ERROR_UNKNOWN;
    }

    /* keep driver running */
    prctl(PR_SET_NAME, "demo_led_main");
    while (0 != g_LedRunState)
    {
        sleep(5);
    }

    /* exit sdk */
    leda_exit();
    log_i(LED_TAG_NAME, "demo exit\r\n");

 return LE_SUCCESS;
}

#ifdef __cplusplus
}
#endif
```

Node.js版本

```
/*
 * Copyright (c) 2018 Alibaba Group Holding Ltd.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
```

```
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

'use strict';

const {
  RESULT_SUCCESS,
  RESULT_FAILURE,
  ThingAccessClient,
} = require('linkedge-thing-access-sdk');

var driverConfig;
try {
  driverConfig = JSON.parse(process.env.FC_DRIVER_CONFIG)
} catch (err) {
  throw new Error('The driver config is not in JSON format!');
}
var configs = driverConfig['deviceList'];
if (!Array.isArray(configs) || configs.length === 0) {
  throw new Error('No device is bound with the driver!');
}

var args = configs.map((config) => {
  var self = {
    config,
    thing: {
      temperature: 41,
    },
    callbacks: {
      setProperties: function (properties) {
        // Usually, in this callback we should set properties to the
physical thing and
        // return the result. Here we just return a failed result
since the properties
        // are read-only.
        console.log('Set properties %s to thing %s-%s', JSON.stringify
(properties),
          config.productKey, config.deviceName);
        // Return an object representing the result in the following
form or the promise
        // wrapper of the object.
        return {
          code: RESULT_FAILURE,
          message: 'failure',
        };
      },
      getProperties: function (keys) {
        // Usually, in this callback we should get properties from the
 physical thing and
        // return the result. Here we return the simulated properties.
        console.log('Get properties %s from thing %s-%s', JSON.
stringify(keys),
```

```
          config.productKey, config.deviceName);
        // Return an object representing the result in the following
form or the promise
        // wrapper of the object.
        if (keys.includes('temperature')) {
          return {
            code: RESULT_SUCCESS,
            message: 'success',
            params: {
              temperature: self.thing.temperature,
            }
          };
        }
        return {
          code: RESULT_FAILURE,
          message: 'The requested properties does not exist.',
        }
      },
      callService: function (name, args) {
        // Usually, in this callback we should call services on the
physical thing and
        // return the result. Here we just return a failed result
since no service
        // provided by the thing.
        console.log('Call service %s with %s on thing %s-%s', JSON.
stringify(name),
          JSON.stringify(args), config.productKey, config.deviceName);
        // Return an object representing the result in the following
form or the promise
        // wrapper of the object
        return new Promise((resolve) => {
          resolve({
            code: RESULT_FAILURE,
            message: 'The requested service does not exist.',
          })
        });
      }
    },
  };
  return self;
});

args.forEach((item) => {
  var client = new ThingAccessClient(item.config, item.callbacks);
  client.setup()
    .then(() => {
      return client.registerAndOnline();
    })
    .then(() => {
      // Push events and properties to LinkEdge platform.
      return new Promise(() => {
        setInterval(() => {
          var temperature = item.thing.temperature;
          if (temperature > 40) {
            client.reportEvent('high_temperature', {'temperature':
temperature});
          }
          client.reportProperties({'temperature': temperature});
        }, 2000);
      });
    })
    .catch(err => {
```

```
      console.log(err);
      return client.cleanup();
    })
    .catch(err => {
      console.log(err);
    });
});

module.exports.handler = function (event, context, callback) {
  console.log(event);
  console.log(context);
  callback(null);
};
```

Python版本

```
# -*- coding: utf-8 -*-
import lethingaccesssdk
import time
import json
import os
import logging


# Base on device, user need to implement the callService, getPropert
ies and getProperties function.
class Temperature_device(lethingaccesssdk.ThingCallback):
  def __init__(self):
    self.temperature = 41

  def callService(self, name, input_value):
    return -1, {}

  def getProperties(self, input_value):
    if input_value[0] == "temperature":
      return 0, {input_value[0]: self.LightSwitch}
    else:
      return -1, {}

  def setProperties(self, input_value):
    if "temperature" in input_value:
      self.LightSwitch = input_value["temperature"]
      return 0, {}

try:
  driver_conf = json.loads(os.environ.get("FC_DRIVER_CONFIG"))
  if "deviceList" in driver_conf and len(driver_conf["deviceList"]) >
0:
    device_list_conf = driver_conf["deviceList"]
    device = device_list_conf[0]
    app_callback = Temperature_device()
    client = lethingaccesssdk.ThingAccessClient(device)
    client.registerAndonline(app_callback)
    while True:
      time.sleep(2)
      if app_callback.temperature > 40:
        client.reportEvent('high_temperature', {'temperature':
app_callback.temperature})
        client.reportProperties({'temperature': app_callback.
temperature})
except Exception as e:
```

```
  logging.error(e)

# don't remove this function
def handler(event, context):
  return 'hello world'
```

3 Link IoT Edge核心SDK

3.1 Nodejs版本SDK

3.1.1 IoTData

Link IoT Edge核心SDK允许开发人员使用Node.js编写FC函数。核心SDK源码已开源，详情请见核心SDK开源的Node.js库。

包含IoT操作相关API的类。

publish(params, callback)

发布指定消息。您可以通过消息路由功能进一步设置消息的流转路径。

名称	类型	描述
params	Object	参数对象，包含如下必选参数。 • topic：{String}，消息主题。 • payload：{String}，消息负载。
callback(err)	function	回调函数。遵循JavaScript标准实践。成功则err为null，否则err包含发生的错误。

getThingProperties(params, callback)

获取指定的设备属性。

名称	类型	描述
params	Object	参数对象，包含如下必选参数。 • productKey：{String}，设备的ProductKey，创建设备时生成。 • deviceName：{String}，设备的DeviceName，创建设备时生成。 • payload：{Array}，包含设备属性的数组。
callback(err, data)	function	回调函数。遵循JavaScript标准实践。 • err：{Error}，成功则为null，否则为发生的错误。 • data：{Object}，指定设备属性的键值。

setThingProperties(params, callback)

设置指定的设备属性。

名称	类型	描述
params	Object	参数对象，包含如下必选参数。 • productKey：{String}，设备的ProductKey，创建设备时生成。 • deviceName：{String}，设备的DeviceName，创建设备时生成。 • payload：{Object}，包含设置给设备的属性键值。
callback(err)	function	回调函数。遵循JavaScript标准实践。 • err：{Error}，成功则返回null，否则为发生的错误。

callThingService(params, callback)

调用指定的设备服务。

名称	类型	描述
params	Object	参数对象，包含如下参数。 • productKey：{String} (Required)，设备的ProductKey，创建设备时生成。 • deviceName：{String} (Required)，设备的DeviceName，创建设备时生成。 • service：{String} (Required)，被调用的服务名。 • payload：{String|Buffer}，传给服务的参数。
callback(err, data)	function	回调函数。遵循JavaScript标准实践。 • err：{Error}，成功则返回null，否则为发生的错误。 • data：被调用服务的返回值。

getThingsWithTags(params, callback)

获取满足所有标签的设备信息。

名称	类型	描述
params	Object	参数对象，包含如下必选参数。 • payload：{Array}，一个{<标签名>:<标签值>}组成的数组。
callback(err, data)	function	回调函数。遵循JavaScript标准实践。 • err：{Error}，成功则返回null，否则为发生的错误。 • data：{Array}，满足条件的设备信息数组。

3.1.2 FCClient

包含函数计算相关API的类。

invokeFunction(params, callback)

调用指定函数。

名称	类型	描述
params	object	参数对象，包含的参数请见下表*params*参数说明。
callback(err, data)	function	回调函数。遵循JavaScript标准实践。 • err：{Error} 成功则返回null，否则为发生的错误。 • data：被调函数的返回值。

表 3-1: params参数说明

名称	类型	描述
functionId	string	被调用函数的唯一ID。您还可以通过serviceName与functionName同时指定被调函数，无需使用functionId。
serviceName	string	服务名，必须与functionName共同指定被调函数。您也可以通过functionId指定被调函数，无需使用serviceName。
functionName	string	函数名，必须与serviceName共同指定被调函数。您也可以通过functionId指定被调函数，无需使用functionName。

名称	类型	描述
invocationType	string	调用类型。同步为Sync，异步为Async。默认为同步。
payload	string|Buffer	参数信息作为函数的输入。

3.2 Python版本SDK

3.2.1 IoTData

Link IoT Edge核心SDK同时允许开发人员使用Python编写FC函数。核心SDK源码已开源，详情请见核心SDK开源的Python库。

包含IoT操作相关API的类。

publish(params)

发布指定消息。您可以通过消息路由功能进一步设置消息的流转路径。

名称	类型	描述
params	dict	参数对象，包含如下必选参数。 • topic：{str}，消息主题。 • payload：{str}，消息负载。

getThingProperties(params)

获取指定的设备属性。

名称	类型	描述
params	dict	参数对象，包含如下必选参数。 • productKey：{str}，设备的ProductKey，创建设备时生成。 • deviceName：{str}，设备的DeviceName，创建设备时生成。 • payload：{list}，包含设备属性的数组。
return	dict	指定设备属性的键值。

setThingProperties(params)

设置指定的设备属性。

名称	类型	描述
params	dict	参数对象，包含如下必选参数。 • productKey：{str}，设备的ProductKey，创建设备时生成。 • deviceName：{str}，设备的DeviceName，创建设备时生成。 • payload：{dict}，包含设置给设备的属性键值。
return	dict	成功则返回True，否则为发生的错误。

callThingService(params)

调用指定的设备服务。

名称	类型	描述
params	dict	参数对象，包含如下参数。 • productKey：{str} (Required)，设备的ProductKey，创建设备时生成。 • deviceName：{str} (Required)，设备的DeviceName，创建设备时生成。 • service：{str} (Required)，被调用的服务名。 • payload：{str|Bytes}，传给服务的参数。
return	dict	被调用服务的返回值。

getThingsWithTags(params)

获取满足所有标签的设备信息。

名称	类型	描述
params	dict	参数对象，包含如下必选参数。 • payload：{list}，一个由{<标签名>:<标签值>}组成的数组。
return	list	满足条件的设备信息数组。

3.2.2 Client

包含函数计算相关API的类。

invoke_function(params)

调用指定函数。

名称	类型	描述
params	dict	参数对象，包含的参数请见下表*params*参数说明。
return	dict	被调函数的返回值。

表 3-2: params参数说明

名称	类型	描述
functionId	str	被调用函数的唯一ID。您还可以通过serviceName与functionName同时指定被调函数，无需使用functionId。
serviceName	str	服务名，必须与functionName共同指定被调函数。您也可以通过functionId指定被调函数，无需使用serviceName。
functionName	str	函数名，必须与serviceName共同指定被调函数。您也可以通过functionId指定被调函数，无需使用functionName。
invocationType	str	调用类型。同步为`Sync`，异步为`Async`。默认为同步。
payload	str|bytes	参数信息作为函数的输入。

4 Link IoT Edge核心SDK综合示例

本文展示使用Link IoT Edge核心SDK控制灯设备的操作代码。示例中，当光照传感器上报的光照度超过500时关闭灯，光照度不足100时打开灯。

Node.js版本综合示例

```
/*
 * Copyright (c) 2018 Alibaba Group Holding Ltd.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

/*
 * The example demonstrates closing a light when it monitor the
illuminance
 * reported by a light sensor is greater than 500.
 */

'use strict';

const leSdk = require('linkedge-core-sdk');

const iotData = new leSdk.IoTData();

const productKey = 'Light Product Key';
const deviceName = 'Light Device Name';

module.exports.handler = function (event, context, callback) {

  console.log(`LightMonitor is invoking with ${event.toString()}.`);

  var illuminance;
  try {
    var obj = JSON.parse(event.toString());
    if (!obj.payload) {
      callback(new Error(`Can't find "payload" in event.`));
      return;
    }
    var payload = JSON.parse(obj.payload);
    if (!payload['MeasuredIlluminance']) {
      callback(new Error(`Can't find "MeasuredIlluminance" in event
.`));
      return;
    }
    illuminance = payload['MeasuredIlluminance'].value || 0;
  } catch (err) {
```

```
    err = new Error(`Parse event failed due to ${err}.`);
    callback(err);
    return;
  }

  if (illuminance > 500) {
    turnOff(callback);
  } else if (illuminance <= 100) {
    turnOn(callback);
  } else {
    console.log(`Illuminance value is ${illuminance}, ignore.`);
    callback(null);
  }
};

function turnOff(callback) {
  // Turn off the light according to product key and device name.
  iotData.setThingProperties({
    productKey,
    deviceName,
    payload: {'LightSwitch': 0},
  }, function (err) {
    if (err) {
      console.log(`Failed to turn off the light due to ${err}.`);
      callback(err);
    } else {
      console.log(`Turns off light successfully.`);
      callback(null);
    }
  });
}

function turnOn(callback) {
  // Turn on the light according to product key and device name.
  iotData.setThingProperties({
    productKey,
    deviceName,
    payload: {'LightSwitch': 1},
  }, function (err) {
    if (err) {
      console.log(`Failed to turn on the light due to ${err}.`);
      callback(err);
    } else {
      console.log(`Turns on light successfully.`);
      callback(null);
    }
  });
}
```

Python版本综合示例

```
# -*- coding: utf-8 -*-
import lecoresdk
import json


ON = 1
OFF = 0

def light_turn(status):
  it = lecoresdk.IoTData()
```

```
  set_params = {"productKey": "Light Product Key",
                            "deviceName": "Light Device Name",
                            "payload": {"LightSwitch":status}}
  res = it.setThingProperties(set_params)

def handler(event, context):
  event_json = json.loads(event.decode("utf-8"))
  if "payload" in event_json:
    payload_json = json.loads(event_json["payload"])
    if "illuminance" in payload_json and "value" in payload_json["
illuminance"]:
       illuminance = payload_json["illuminance"]["value"]
       if illuminance > 500:
          light_turn(OFF)
       elif illuminance <= 100:
          light_turn(ON)
  return 'hello world'
```

5 云端开发指南