设备接入Link Kit SDK

Java SDK

Java SDK

工程配置

用户可以使用下面两种依赖方式之一对工程进行配置：

Maven 依赖方式

配置Maven仓库地址，在pom添加如下maven仓库依赖。

```
<repositories>
<repository>
<id>alimaven</id>
<name>aliyun maven</name>
<url>http://maven.aliyun.com/nexus/content/groups/public/</url>
</repository>
</repositories>
```

maven 依赖，在工程 pom.xml dependencies 下添加如下依赖。

```
<dependencies>
<dependency>
<groupId>com.aliyun.alink.linksdk</groupId>
<artifactId>iot-linkkit-java</artifactId>
<version>1.2.0.1</version>
<scope>compile</scope>
</dependency>
<dependency>
<groupId>com.google.code.gson</groupId>
<artifactId>gson</artifactId>
<version>2.8.1</version>
<scope>compile</scope>
</dependency>
<dependency>
<groupId>com.alibaba</groupId>
<artifactId>fastjson</artifactId>
<version>1.2.40</version>
```

```
<scope>compile</scope>
</dependency>
</dependencies>
```

JAR 依赖方式

Java版设备端SDK依赖列表如下，可以到 阿里云Maven仓库 下载并依赖。

type	groupId	artifactId	version	description
LinkKit SDK	com.aliyun.alink.linksdk	iot-linkkit-java	1.2.0.1	LinkKit API
LinkKit SDK	com.aliyun.alink.linksdk	iot-device-manager-java	1.2.0.2	device manager
LinkKit SDK	com.aliyun.alink.linksdk	public-channel-gateway-java	1.2.2	gateway
LinkKit SDK	com.aliyun.alink.linksd	public-tmp-java	1.0.2	物模型
LinkKit SDK	com.aliyun.alink.linksdk	public-cmp-java	1.3.5	连接管理
LinkKit SDK	com.aliyun.alink.linksdk	iot-apiclient	1.0.0	https请求
LinkKit SDK	com.aliyun.alink.linksdk	network-core	1.0.0	基础网络请求库
LinkKit SDK	com.aliyun.alink.linksdk	iot-mqttclient	1.0.6	长连接
LinkKit SDK	com.aliyun.alink.linksdk	tools-java	1.0.1	
base SDK	com.google.code.gson	gson	2.8.1	json解析库
base SDK	com.alibaba	fastjson	1.2.40	json解析库
base SDK	com.squareup.okhttp3	okhttp	3.0.1	
base SDK	com.squareup.okio	okio	1.6.0	
base SDK	com.aliyun.alink.linksdk	opensource-paho-java	1.2.1	Mqtt协议实现

Java SDK Demo

Java SDK提供Demo，供您参考使用。
单击下载Java SDK Demo。下载本Demo将默认您同意本软件许可协议。
接口定义参见 Api Reference。

认证与连接

本文介绍如何进行 SDK 初始化，建立设备与云端的连接。

设备认证

设备的身份认证支持两种方法，不同方法需填写不同信息。

- 若使用一机一密认证方式，需要有ProductKey、DeviceName和DeviceSecret。
- 若使用一型一密认证方式，需要有ProductKey、ProductSecret和DeviceName，并在控制台开启动态注册。 **说明** 若使用一型一密认证方式，在调用初始化接口之前需先调用一型一密动态注册接口。

一型一密动态注册

如果设备认证选择了一型一密，需要先调用动态注册接口，一机一密设备可以跳过这一步，直接调用初始化接口。动态注册成功之后，持久化获取到的三元组信息，然后调用初始化接口。设备三元组信息（productKey、deviceName、deviceSecret）需要持久化存在本地。动态初始化成功，初始化建联之后，不能再执行动态初始化，后续重新运行程序都需要从持久化存储中获取三元组，然后执行初始化建联。LinkKitInitParams初始化参数。

```
// ####### 一型一密动态注册接口开始 ######
/**
* 注意：动态注册成功，设备上线之后，不能再次执行动态注册，云端会返回已注册。
*/
LinkKitInitParams params = new LinkKitInitParams();
IoTMqttClientConfig config = new IoTMqttClientConfig();
config.productKey = deviceInfo.productKey;
config.deviceName = deviceInfo.deviceName;

params.mqttClientConfig = config;
params.deviceInfo = deviceInfo;

final CommonRequest request = new CommonRequest();
request.setPath("/auth/register/device");
LinkKit.getInstance().deviceRegister(params, request, new IoTCallback() {
```

```
public void onFailure(CommonRequest commonRequest, Exception e) {
ALog.e(TAG, "动态注册失败 " + e);
}

public void onResponse(CommonRequest commonRequest, CommonResponse commonResponse) {
if (commonResponse == null || StringUtils.isEmptyString(commonResponse.getData())) {
ALog.e(TAG, "动态注册失败 response=null");
return;
}
try {
ResponseModel<Map<String, String>> response = new Gson().fromJson(commonResponse.getData(), new
TypeToken<ResponseModel<Map<String, String>>>() {
}.getType());
if (response != null && "200".equals(response.code)) {
ALog.d(TAG, "动态注册成功" + (commonResponse == null ? "" : commonResponse.getData()));
/** 获取 deviceSecret, 存储到本地，然后执行初始化建联
* 这个流程只能走一次，获取到 secret 之后，下次启动需要读取本地存储的三元组，
* 直接执行初始化建联，不可以再走动态初始化
*/
// deviceSecret = response.data.get("deviceSecret");
// init(pk,dn,ds);
return;
}
} catch (Exception e) {

}
ALog.d(TAG, "动态注册失败" + commonResponse.getData());
}
});
// ####### 一型一密动态注册接口结束 ######
```

SDK 初始化

SDK初始化，即一机一密建联。设备如果初始化失败，如网络问题，请做业务重试确保初始化成功，初始化成功之后和云端的连接的断连重试由SDK 负责。onInitDone表示初始化成功，onError表示初始化失败。

```
LinkKitInitParams params = new LinkKitInitParams();
/**
* 设置 Mqtt 初始化参数
*/
IoTMqttClientConfig config = new IoTMqttClientConfig();
config.productKey = pk;
config.deviceName = dn;
config.deviceSecret = ds;
/**
* 是否接受离线消息
* 对应 mqtt 的 cleanSession 字段
*/
config.receiveOfflineMsg = false;
params.mqttClientConfig = config;

/**
* 设置初始化三元组信息，用户传入
```

```
*/
DeviceInfo deviceInfo = new DeviceInfo();
deviceInfo.productKey = pk;
deviceInfo.deviceName = dn;
deviceInfo.deviceSecret = ds;

params.deviceInfo = deviceInfo;

/**
* 设置设备当前的初始状态值，属性需要和云端创建的物模型属性一致
* 如果这里什么属性都不填，物模型就没有当前设备相关属性的初始值。
* 用户调用物模型上报接口之后，物模型会有相关数据缓存。
*/
Map<String, ValueWrapper> propertyValues = new HashMap<String, ValueWrapper>();
// 示例
// propertyValues.put("LightSwitch", new ValueWrapper.BooleanValueWrapper(0));
params.propertyValues = propertyValues;

LinkKit.getInstance().init(params, new ILinkKitConnectListener() {
public void onError(AError aError) {
ALog.e(TAG, "Init Error error=" + aError);
}

public void onInitDone(InitResult initResult) {
ALog.i(TAG, "onInitDone result=" + initResult);
}
});
```

SDK 反初始化

如果需要注销初始化，调用如下接口。IConnectNotifyListener 是全局下行数据和连接状态监听。

```
// 取消注册 notifyListener，notifyListener对象需和注册的时候是同一个对象
LinkKit.getInstance().unRegisterOnNotifyListener(notifyListener);
LinkKit.getInstance().deinit();
```

其他设置

日志开关

打开SDK内部日志输出开关：

```
ALog.setLevel(ALog.LEVEL_DEBUG);
```

连接状态监听

如果需要监听设备的上下线信息，云端下发的所有数据，可以设置以下监听器。

```
IConnectNotifyListener notifyListener = new IConnectNotifyListener() {
@Override
public void onNotify(String connectId, String topic, AMessage aMessage) {
// 云端下行数据回调
// connectId 连接类型 topic 下行 topic; aMessage 下行数据
//String pushData = new String((byte[]) aMessage.data);
// pushData 示例 {"method":"thing.service.test_service","id":"123374967","params":{"vv":60},"version":"1.0.0"}
// method 服务类型； params 下推数据内容
}

@Override
public boolean shouldHandle(String connectId, String topic) {
// 选择是否不处理某个 topic 的下行数据
// 如果不处理某个topic，则onNotify不会收到对应topic的下行数据
return true; //TODO 根基实际情况设置
}

@Override
public void onConnectStateChange(String connectId, ConnectState connectState) {
// 对应连接类型的连接状态变化回调，具体连接状态参考 SDK ConnectState
}
}
// 注册下行监听，包括长连接的状态和云端下行的数据
LinkKit.getInstance().registerOnNotifyListener(notifyListener);
```

请求域名

云端接口的请求域名，请参考**地域和可用区**查看支持的域名。

MQTT域名设置SDK初始化的时候添加以下设置。

```
// 设置 Mqtt 请求域名 LinkKitInitParams 初始化参数
IoTMqttClientConfig clientConfig = new IoTMqttClientConfig();
// 慎用 设置 mqtt 请求域名，默认productKey+".iot-as-mqtt.cn-shanghai.aliyuncs.com:1883" ,如果无具体的业务需求，请不要设置。
//clientConfig.channelHost = "xxx";
linkKitInitParams.mqttClientConfig = clientConfig;
```

一型一密域名设置SDK动态注册的时候添加以下设置。```javaHubApiRequest hubApiRequest = new HubApiRequest();// 一型一密域名 默认”iot-auth.cn-shanghai.aliyuncs.com”// hubApiRequest.domain = “xxx”; // 如无特殊需求，不要设置

```
### Mqtt 连接参数
* 设置Mqtt Keep-Alive 时间
> ```java
> // interval 单位秒
```

```
> MqttConfigure.setKeepAliveInterval(int interval);
>
```

- qos设置

```
MqttPublishRequest request = new MqttPublishRequest();
// 支持 0 和 1， 默认0
request.qos = 0;
request.isRPC = false;
request.topic = topic.replace("request", "response");
String resId = topic.substring(topic.indexOf("rrpc/request/")+13);
request.msgId = resId;
// TODO 用户根据实际情况填写 仅做参考
request.payloadObj = "{\"id\":\"" + resId + "\", \"code\":\"200\"" + ",\"data\":{} }";
```

- cleanSession 设置

```
/**
* 设置 Mqtt 初始化参数
*/
IoTMqttClientConfig config = new IoTMqttClientConfig();
config.productKey = deviceInfoData.productKey;
config.deviceName = deviceInfoData.deviceName;
config.deviceSecret = deviceInfoData.deviceSecret;
config.channelHost = pk + ".iot-as-mqtt." + deviceInfoData.region + ".aliyuncs.com:1883";
/**
* 是否接受离线消息
* 对应 receiveOfflineMsg = !cleanSession, 默认不接受离线消息
*/
config.receiveOfflineMsg = false;
params.mqttClientConfig = config;
```

自定义MQTT Topic通信

LinkKit SDK 提供了与云端长链接的基础能力接口，用户可以直接使用这些接口完成自定义 Topic 相关的功能。提供的基础能力包括：发布、订阅、取消订阅、RRPC、订阅下行。

上行接口请求

调用上行请求接口，SDK 封装了上行Publish请求、订阅Subscribe和取消订阅unSubscribe等接口。

```
/**
* 发布
*
* @param request 发布请求
* @param listener 监听器
*/
void publish(ARequest request, IConnectSendListener var2);

/**
* 订阅
*
* @param request 订阅请求
* @param listener 监听器
*/
void subscribe(ARequest request, IConnectSubscribeListener var2);

/**
* 取消订阅
*
* @param request 取消订阅请求
* @param listener 监听器
*/
void unsubscribe(ARequest request, IConnectUnscribeListener var2);
```

调用示例：

MqttPublishRequest 类路径参见 com.aliyun.alink.linksdk.cmp.connect.channel.MqttPublishRequest。

```
// 发布
MqttPublishRequest request = new MqttPublishRequest();
// topic 用户根据实际场景填写
request.topic = "/sys/" + pk + "/" + dn + "/thing/deviceinfo/update";
/**
* 订阅回复的 replyTopic
* 如果业务有相应的响应需求，可以设置 replyTopic，且 isRPC=true
*/
// request.replyTopic = request.topic + "_reply";
/**
* isRPC = true; 表示先订阅 replyTopic，然后再发布；
* isRPC = false; 不会订阅回复
*/
// request.isRPC = true;
/**
* 设置请求的 qos
*/
request.qos = 0;
// 更新标签 仅做测试
// payloadObj 替换成用户需要发布的数据 json String
//示例 属性上报 {"id":"160865432","method":"thing.event.property.post","params":{"LightSwitch":1},"version":"1.0"}
request.payloadObj = "{\"id\":2, \"params\":{\"version\":\"1.0.0\"}}";
LinkKit.getInstance().publish(request, new IConnectSendListener() {
@Override
public void onResponse(ARequest aRequest, AResponse aResponse) {
// publish 结果
ALog.d(TAG, "onResponse " + (aResponse==null?"":aResponse.data));
```

```
}

@Override
public void onFailure(ARequest aRequest, AError aError) {
// publish 失败
ALog.d(TAG, "onFailure " + (aError==null?"":(aError.getCode()+aError.getMsg())));
}
});
// 订阅
MqttSubscribeRequest request = new MqttSubscribeRequest();
// topic 用户根据实际场景填写
request.topic = "/sys/" + pk + "/" + dn + "/thing/deviceinfo/update";
request.isSubscribe = true;
LinkKit.getInstance().subscribe(request, new IConnectSubscribeListener() {
@Override
public void onSuccess() {
// 订阅成功
ALog.d(TAG, "onSuccess ");
}

@Override
public void onFailure(AError aError) {
// 订阅失败
ALog.d(TAG, "onFailure " + (aError==null?"":(aError.getCode()+aError.getMsg())));
}
});
// 取消订阅
MqttSubscribeRequest request = new MqttSubscribeRequest();
// topic 用户根据实际场景填写
request.topic = "/sys/" + pk + "/" + dn + "/thing/deviceinfo/update";
request.isSubscribe = false;
LinkKit.getInstance().unsubscribe(request, new IConnectUnscribeListener() {
@Override
public void onSuccess() {
// 取消订阅成功
ALog.d(TAG, "onSuccess ");
}

@Override
public void onFailure(AError aError) {
// 取消订阅失败
ALog.d(TAG, "onFailure " + (aError==null?"":(aError.getCode()+aError.getMsg())));
}
});
```

下行数据监听

下行数据监听可以通过 RRPC 方式或者注册一个下行数据监听器实现。

```
/**
* RRPC 接口
* RRPC：先订阅 topic A；云端需要的时候调用设备的服务，通过 topic A下发数据；设备收到数据，回复云端；
* 另外一种方式是：单独调用订阅接口，然后在 registerOnNotifyListener 接收对应 topic 的下行数据，
* 并回复云端；
```

```
* @param topic 订阅 topic
* @param listener 监听器
*/
void registerResource(AResource var1, IResourceRequestListener var2);

/**
* 注册下行数据监听器,所有已订阅的 topic 下行数据都会在这里返回
*
* @param listener 监听器
*/
void registerOnNotifyListener(IConnectNotifyListener listener);

/**
* 取消注册下行监听器
*
* @param listener 监听器
*/
void unRegisterOnNotifyListener(IConnectNotifyListener listener);
```

调用示例：

```
/**
* 下行数据接收&处理
* 设备连接状态变化
*/
private IConnectNotifyListener notifyListener = new IConnectNotifyListener() {
public void onNotify(String connectId, String topic, AMessage aMessage) {
// 云端下行数据通知
}

public void onConnectStateChange(String connectId, ConnectState connectState) {
// 设备连接状态通知
}
public boolean shouldHandle(String connectId, String topic){
return true; // 根据实际场景设置
}
};

/**
* 所有topic的下行数据入口（前提是先订阅了该 topic，才会在这里收到）
* 如果想要在这里收到对应 topic 的下行数据，需要先订阅该 topic
*/
public void registerNotifyListener(){
LinkKit.getInstance().registerOnNotifyListener(notifyListener);
}

/**
* 取消注册下行的监听器，该 listener 需要保持和注册的 listener 是同一个对象
*/
public void unregisterNotifyListener() {
LinkKit.getInstance().unRegisterOnNotifyListener(notifyListener);
}
```

RRPC 调用示例：

```
final CommonResource resource = new CommonResource();
resource.topic = "/ext/rrpc/+/" + productKey + "/" + deviceName + "/get";
resource.replyTopic = resource.topic;

LinkKit.getInstance().registerResource(resource, new IResourceRequestListener() {
@Override
public void onHandleRequest(AResource aResource, ResourceRequest resourceRequest, IResourceResponseListener
iResourceResponseListener) {
// 收到云端数据下行
ALog.d(TAG, "onHandleRequest aResource=" + aResource + ", resourceRequest=" + resourceRequest + ",
iResourceResponseListener=" + iResourceResponseListener);
// 下行数据解析示例
// String downstreamData = new String((byte[]) resourceRequest.payloadObj);
// 示例 {"id":"269297015","version":"1.0","method":"thing.event.property.post","params":{"lightData":{"vv":12}}}

// 如果数据是json，且包含id字段，格式可以按照如下示例回复，传输数据请根据实际情况定制
// if (aResource instanceof CommonResource) {
// ((CommonResource) aResource).replyTopic = resourceRequest.topic;
// }
// if (iResourceResponseListener != null) {
// AResponse response = new AResponse();
//
// response.data = "{\"id\":\"123\", \"code\":\"200\"" + ",\"data\":{} }";
// iResourceResponseListener.onResponse(aResource, resourceRequest, response);
// }
// 如果不一定是json格式，可以参考如下方式回复
MqttPublishRequest rrpcResponse = new MqttPublishRequest();
rrpcResponse.topic = resourceRequest.topic;
rrpcResponse.payloadObj ="xxx";

LinkKit.getInstance().publish(rrpcResponse,null);
}

@Override
public void onSuccess() {
// 注册资源成功
ALog.d(TAG, "onSuccess ");
}

@Override
public void onFailure(AError aError) {
// 注册资源失败
ALog.d(TAG, "onFailure " + getError(aError));
}
});
```

物模型开发

设备可以使用物模型功能，实现属性上报（如上报设备状态）、事件上报（上报设备异常或错误）和服务调用（通过云端调用设备提供的服务）。

getDeviceThing() 返回的 IThing 接口 介绍参见 IThing ApiReference。

设备属性

- 设备属性上报

```
// 设备上报
Map<String, ValueWrapper> reportData = new HashMap<>();
// identifier 是云端定义的属性的唯一标识，valueWrapper是属性的值
// reportData.put(identifier, valueWrapper); // 参考示例，更多使用可参考demo
LinkKit.getInstance().getDeviceThing().thingPropertyPost(reportData, new IPublishResourceListener() {
public void onSuccess(String s, Object o) {
// 属性上报成功
}
public void onError(String s, AError aError) {
// 属性上报失败
}
});
```

设备属性获取

```
// 根据 identifier 获取当前物模型中该属性的值
String identifier = "xxx";
LinkKit.getInstance().getDeviceThing().getPropertyValue(identifier);

// 获取所有属性
LinkKit.getInstance().getDeviceThing().getProperties()
```

设备事件

设备事件上报

```
HashMap<String, ValueWrapper> hashMap = new HashMap<>();
// TODO 用户根据实际情况设置
// hashMap.put("ErrorCode", new ValueWrapper.IntValueWrapper(0));
OutputParams params = new OutputParams(valueWrapperMap);
LinkKit.getInstance().getDeviceThing().thingEventPost(identity, params, new IPublishResourceListener() {
public void onSuccess(String resId, Object o) {
// 事件上报成功
}
```

```
public void onError(String resId, AError aError) {
// 事件上报失败
}
});
```

设备服务

- 设备服务获取 Service 定义参见 Service API Reference。

```
// 获取设备支持的所有服务
LinkKit.getInstance().getDeviceThing().getServices()
```

设备服务调用监听 云端在添加设备服务时，需设置该服务的调用方式，由Service中的callType 字段表示。同步服务调用时callType="sync"；异步服务调用callType="async"。设备属性设置和获取也是通过服务调用监听方式实现云端服务的下发。

异步服务调用 设备先注册服务的处理监听器，当云端触发异步服务调用的时候，下行的请求会到注册的监听器中。一个设备会有多种服务，通常需要注册所有服务的处理监听器。onProcess 是设备收到的云端下行的服务调用，第一个参数是需要调用服务对应的 identifier，用户可以根据 identifier（identifier 是云端在创建属性或事件或服务的时候的标识符，可以在云端产品的功能定义找到每个属性或事件或服务对应的identifier）做不同的处理。云端调用设置服务的时候，设备需要在收到设置指令后，调用设备执行真实操作，操作结束后上报一条属性状态变化的通知。

```
// 用户可以根据实际情况注册自己需要的服务的监听器
List<Service> srviceList = LinkKit.getInstance().getDeviceThing().getServices();
for (int i = 0; srviceList != null && i < srviceList.size(); i++) {
Service service = srviceList.get(i);
LinkKit.getInstance().getDeviceThing().setServiceHandler(service.getIdentifier(),
mCommonHandler);
}
// 服务处理的handler
private ITResRequestHandler mCommonHandler = new ITResRequestHandler() {
public void onProcess(String identify, Object result, ITResResponseCallback
itResResponseCallback) {
ALog.d(TAG, "onProcess() called with: s = [" + identify + "], o = [" + result + "],
itResResponseCallback = [" + itResResponseCallback + "]");
ALog.d(TAG, "收到云端异步服务调用 " + identify);
try {
if (SERVICE_SET.equals(identify)) {
// TODO 用户按照真实设备的接口调用 设置设备的属性
// 设置完真实设备属性之后，上报设置完成的属性值
// 用户根据实际情况判断属性是否设置成功 这里测试直接返回成功
boolean isSetPropertySuccess = true;
if (isSetPropertySuccess) {
```

```
if (result instanceof InputParams) {
Map<String, ValueWrapper> data = (Map<String, ValueWrapper>) ((InputParams)
result).getData();
// data.get()
ALog.d(TAG, "收到异步下行数据 " + data);
// 响应云端 接收数据成功
itResResponseCallback.onComplete(identify, null, null);
} else {
itResResponseCallback.onComplete(identify, null, null);
}
} else {
AError error = new AError();
error.setCode(100);
error.setMsg("setPropertyFailed.");
itResResponseCallback.onComplete(identify, new ErrorInfo(error), null);
}

} else if (SERVICE_GET.equals(identify)) {
// 初始化的时候将默认值初始化传进来，物模型内部会直接返回云端缓存的值

} else {
// 根据不同的服务做不同的处理，跟具体的服务有关系
ALog.d(TAG, "用户根据真实的服务返回服务的值，请参照set示例");
OutputParams outputParams = new OutputParams();
// outputParams.put("op", new ValueWrapper.IntValueWrapper(20));
itResResponseCallback.onComplete(identify, null, outputParams);
}
} catch (Exception e) {
e.printStackTrace();
ALog.d(TAG, "TMP 返回数据格式异常");
}
}

public void onSuccess(Object o, OutputParams outputParams) {
ALog.d(TAG, "onSuccess() called with: o = [" + o + "], outputParams = [" + outputParams + "]");
ALog.d(TAG, "注册服务成功");
}

public void onFail(Object o, ErrorInfo errorInfo) {
ALog.d(TAG, "onFail() called with: o = [" + o + "], errorInfo = [" + errorInfo + "]");
ALog.d(TAG, "注册服务失败");
}
};
```

- 同步服务调用 同步服务调用是一个 RRPC 调用。用户可以注册一个下行数据监听，当云端触发服务调用时，可以在 onNotify 收到云端的下行服务调用。收到云端的下行服务调用之后，用户根据实际服务调用对设备做服务处理，处理之后需要回复云端的请求，即发布一个带有处理结果的请求到云端。**当前版本添加了支持使用自定义RRPC，云端的同步服务属性下行将走自定义RRPC通道下行，用户在升级SDK之后要注意这个改动点。**

```
if (CONNECT_ID.equals(connectId) && !TextUtils.isEmpty(topic) &&
topic.startsWith("/ext/rrpc/")) {
ALog.d(TAG, "收到云端自定义RRPC下行");
```

```
// ALog.d(TAG, "receice Message=" + new String((byte[]) aMessage.data));
// 服务端返回数据示例
{"method":"thing.service.test_service","id":"123374967","params":{"vv":60},"version":"1.0.0
"}
MqttPublishRequest request = new MqttPublishRequest();
// 支持 0 和 1， 默认0
// request.qos = 0;
request.isRPC = false;
request.topic = topic.replace("request", "response");
String[] array = topic.split("/");
String resId = array[3];
request.msgId = resId;
// TODO 用户根据实际情况填写 仅做参考
request.payloadObj = "{\"id\":\"" + resId + "\", \"code\":\"200\"" + ",\"data\":{} }";
// aResponse.data =
LinkKit.getInstance().publish(request, new IConnectSendListener() {
@Override
public void onResponse(ARequest aRequest, AResponse aResponse) {
ALog.d(TAG, "onResponse() called with: aRequest = [" + aRequest + "], aResponse = [" +
aResponse + "]");
}

@Override
public void onFailure(ARequest aRequest, AError aError) {
ALog.d(TAG, "onFailure() called with: aRequest = [" + aRequest + "], aError = [" + aError
+ "]");
}
});
}
```

远程配置

使用该功能前，需在云端开启产品的**远程配置**功能。远程配置可以用于更新设备的配置信息，包括设备的系统参数、网络参数或者本地策略等等。

说明 远程配置相关接口参见 设备 IDeviceCOTA。

主动获取配置

```
RequestModel<Map> requestModel = new RequestModel<Map>();
requestModel.id = "123";
requestModel.method = "thing.config.get";
requestModel.version = "1.0";
Map<String, String> paramsMap = new HashMap<String, String>();
paramsMap.put("configScope", "product");
```

```
paramsMap.put("getType", "file");
requestModel.params = paramsMap;

LinkKit.getInstance().getDeviceCOTA().COTAGet(requestModel, new IConnectSendListener() {
@Override
public void onResponse(ARequest aRequest, AResponse aResponse) {
ALog.d(TAG, "onResponse() called with: aRequest = [" + aRequest + "], aResponse = [" + (aResponse == null ? null :
aResponse.data) + "]");
}

@Override
public void onFailure(ARequest aRequest, AError aError) {
ALog.d(TAG, "onFailure() called with: aRequest = [" + aRequest + "], aError = [" + getError(aError) + "]");
}
});
```

订阅获取

设备端可以通过订阅获取远程配置信息

```
LinkKit.getInstance().getDeviceCOTA().setCOTAChangeListener(new IConnectRrpcListener() {
@Override
public void onSubscribeSuccess(ARequest aRequest) {
ALog.d(TAG, "onSubscribeSuccess() called with: aRequest = [" + aRequest + "]");
}

@Override
public void onSubscribeFailed(ARequest aRequest, AError aError) {
ALog.d(TAG, "onSubscribeFailed() called with: aRequest = [" + aRequest + "], aError = [" + getError(aError) + "]");
}

@Override
public void onReceived(ARequest aRequest, IConnectRrpcHandle iConnectRrpcHandle) {
ALog.d(TAG, "onReceived() called with: aRequest = [" + aRequest + "], iConnectRrpcHandle = [" +
iConnectRrpcHandle + "]");
if (aRequest instanceof MqttRrpcRequest) {
// 云端下行数据 拿到
String cotaData = new String((byte[]) ((MqttRrpcRequest) aRequest).payloadObj);
ALog.d(TAG, "received data=" + cotaData);
// ((MqttRrpcRequest) aRequest).payloadObj;
// ResponseModel<Map<String, String>> responseModel = JSONObject.parseObject(((MqttRrpcRequest)
aRequest).payloadObj, new TypeReference<ResponseModel<Map<String, String>>>(){}.getType());

}
// 返回数据示例
/*{
"id": "123",
"version": "1.0",
"code": 200,
"data": {
"configId": "123dagdah",
"configSize": 1234565,
"sign": "123214adfadgadg",
"signMethod": "Sha256",
```

```
"url": "https://iotx-config.oss-cn-
shanghai.aliyuncs.com/nopoll_0.4.4.tar.gz?Expires=1502955804&OSSAccessKeyId=XXXXXXXXXXXXXXXXXXXX&Sig
nature=XfgJu7P6DWWejstKJgXJEH0qAKU%3D&security-
token=CAISuQJ1q6Ft5B2yfSjIpK6MGsyN1Jx5jo6mVnfBglIPTvlvt5D50Tz2IHtIf3NpAusdsv03nWxT7v4flqFyTINVAEvYZ
JOPKGrGR0DzDbDasumZsJbo4f%2FMQBqEaXPS2MvVfJ%2BzLrf0ceusbFbpjzJ6xaCAGxypQ12iN%2B%2Fr6%2F5gdc9
FcQSkL0B8ZrFsKxBltdUROFbIKP%2BpKWSKuGfLC1dysQcO1wEP4K%2BkkMqH8Uic3h%2Boy%2BgJt8H2PpHhd9NhX
uV2WMzn2%2FdtJOiTknxR7ARasaBqhelc4zqA%2FPPlWgAKvkXba7aIoo01fV4jN5JXQfAU8KLO8tRjofHWmojNzBJAA
PpYSSy3Rvr7m5efQrrybY1lLO6iZy%2BVio2VSZDxshI5Z3McKARWct06MWV9ABA2TTXXOi40BOxuq%2B3JGoABXC54
TOlo7%2F1wTLTsCUqzzeIiXVOK8CfNOkfTucMGHkeYeCdFkm%2FkADhXAnrnGf5a4FbmKMQph2cKsr8y8UfWLC6IzvJ
sClXTnbJBMeuWIqo5zIynS1pm7gf%2F9N3hVc6%2BEeIk0xfl2tycsUpbL2FoaGk6BAF8hWSWYUXsv59d5Uk%3D",
"getType": "file"
}
}*/

}

@Override
public void onResponseSuccess(ARequest aRequest) {
ALog.d(TAG, "onResponseSuccess() called with: aRequest = [" + aRequest + "]");
}

@Override
public void onResponseFailed(ARequest aRequest, AError aError) {
ALog.d(TAG, "onResponseFailed() called with: aRequest = [" + aRequest + "], aError = [" + getError(aError) + "]");
}
});
```

标签

支持设备端上报标签到云端，以及删除设备标签。

说明 设备标签相关接口参见设备 IDeviceLabel。

上报标签

```
// 标签是 key，value的形式 可以替换 attrKey 和 attrValue
RequestModel<List<Map<String, String>>> requestModel = new RequestModel<List<Map<String, String>>>();
requestModel.id = "123";
requestModel.method = "thing.deviceinfo.update";
requestModel.version = "1.0";
List<Map<String, String>> paramsList = new ArrayList<Map<String, String>>();

Map<String, String> listItemMap = new HashMap<String, String>();
listItemMap.put("attrKey", "Temperature");
listItemMap.put("attrValue", "26.8");

paramsList.add(listItemMap);
```

```
requestModel.params = paramsList;
LinkKit.getInstance().getDeviceLabel().labelDelete(requestModel, new IConnectSendListener() {
@Override
public void onResponse(ARequest aRequest, AResponse aResponse) {
ALog.d(TAG, "onResponse() called with: aRequest = [" + aRequest + "], aResponse = [" + (aResponse == null ? "" :
aResponse.data) + "]");
}

@Override
public void onFailure(ARequest aRequest, AError aError) {
ALog.d(TAG, "onFailure() called with: aRequest = [" + aRequest + "], aError = [" + getError(aError) + "]");
}
});
```

删除标签

```
// 标签是 key，value的形式 可以替换 attrKey 和 attrValue
RequestModel<List<Map<String, String>>> requestModel = new RequestModel<List<Map<String, String>>>();
requestModel.id = "123";
requestModel.method = "thing.deviceinfo.delete";
requestModel.version = "1.0";
List<Map<String, String>> paramsList = new ArrayList<Map<String, String>>();

Map<String, String> listItemMap = new HashMap<String, String>();
listItemMap.put("attrKey", "Temperature");

paramsList.add(listItemMap);
requestModel.params = paramsList;

LinkKit.getInstance().getDeviceLabel().labelUpdate(requestModel, new IConnectSendListener() {
@Override
public void onResponse(ARequest aRequest, AResponse aResponse) {
ALog.d(TAG, "onResponse() called with: aRequest = [" + aRequest + "], aResponse = [" + (aResponse == null ? "" :
aResponse.data) + "]");
}

@Override
public void onFailure(ARequest aRequest, AError aError) {
ALog.d(TAG, "onFailure() called with: aRequest = [" + aRequest + "], aError = [" + getError(aError) + "]");
}
});
```

子设备管理

如果当前设备是一个网关，且该网关下的子设备需接入云端从而可以通过云端对子设备进行控制与管理，此时需要使用子设备管理功能。

网关子设备管理提供了子设备动态注册、获取云端网关下子设备列表、添加子设备、删除子设备、子设备上线、子设备下线、监听子设备禁用和删除的消息、代理子设备数据上下行的能力。

网关本身是一个直连设备，网关产品的开发参见前面章节中介绍的“认证与连接”、“自定义MQTT Topic通信”。网关使用子设备管理的功能前，需要网关已经连接到阿里云物联网平台。

说明 网关子设备管理相关接口参见设备 IGateway。

网关开发过程说明

厂商在物联网平台定义网关产品（基础版或者高级版），设置“节点类型”为“网关”，设置网关的身份认证模式，并根据网关功能定义topic或者定义物模型，并参照前面的章节中的说明对网关自身的功能进行开发；

实现子设备的管理功能

i. 实现子设备的发现与连接功能，该部分功能由厂商自行实现，阿里并未提供网关如何发现以及如何将子设备连接到网关的代码实现
ii. 实现子设备三元组的获取方式，下面的内容有介绍几种获取子设备三元组的方式供厂商参考
iii. 当网关发现并将一个子设备连接到网关后，如果需要该子设备能够通过物联网平台进行远程管理，需要调用SDK的添加子设备接口将其告知物联网平台，添加之前需要先获得该子设备的三元组信息；然后调用SDK提供的子设备上线接口通知物联网平台，因为如果一个子设备处于离线状态，如果远程对子设备进行控制，物联网平台将直接返回失败、而不是将命令发送给网关之后等待错误提示或者超时提示
iv. 当网关告知物联网平台一个子设备上线之后，需要将子设备的状态信息上报云端以保证子设备在云端的状态与当前子设备的状态一致。特别是使用物模型定义子设备功能时，子设备上线需要将属性的最新数值通知云端；
v. 当网关的一个已添加到物联网平台的子设备离线时，网关需要调用子设备离线接口告知物联网平台这个子设备已离线；
vi. 当一个在线子设备的属性发生变化时，也需要实时告知物联网平台
vii. 当网关离线并再次上线时（比如网络连接断开，或者网关重启），网关需要对所有已添加到云端的子设备再次调用添加子设备接口，再次调用子设备上线接口，如果网关不知道子设备的属性与网关离线前上报到云端的是否一致，那么网关需要将子设备的最新属性再次上报云端。
viii. 当网关接收到来自物联网平台对子设备的控制消息时，网关如何将该消息转换成子设备识别的格式并发送给子设备，由网关厂商进行实现

子设备开发过程

- 设备厂商在物联网平台定义子设备产品（基础版或者高级版），设置“节点类型”为“设备”，设置产品的身份认证模式
- 阿里并不在子设备上提供任何SDK，因此网关如何发现子设备、如何连接子设备、网关如何发现子设备上线或者离线、网关如何将来自物联网平台的命令发送给子设备，均由网关厂商与子设备厂商定义

协议并实现

子设备三元组的获取方式

子设备是通过网关到阿里云物联网进行注册的，注册时也需要使用到子设备的三元组进行设备验证。下面是网关获取子设备三元组的几种方式，网关厂家根据自己的实际情况进行选用：

- 网关从子设备获取子设备三元组

由网关与子设备之间定义一套协议，当网关发现与连接子设备之后，获取到子设备的三元组。阿里云并不提供参考协议实现，该协议由网关厂商与子设备厂商自行定义与实现；

- 网关预置子设备的三元组

如果网关设备预先可以得知自己需要连接的子设备，并且网关提供了某种配置方式输入子设备的三元组信息，那么可以通过这种方式获取子设备的三元组。同样，该功能由网关厂商实现。

- 网关通过动态注册获取子设备三元组

网关可以通过某种协议发现与连接子设备，并获取到子设备的型号（model）以及唯一标识（比如SN、MAC地址），但是并不知道子设备的DeviceSecret。由于子设备也需要在阿里云物联网平台进行产品定义（云端会为子设备生成ProductKey），网关可以建立子设备型号（model）到阿里云物联网平台ProductKey的映射（网关厂家在网关上实现该映射），并将设备的唯一标识作为阿里云物联网平台的DeviceName，然后通过阿里云物联网平台提供的动态注册功能从云端获取子设备的DeviceSecret，从而得到完整的子设备的三元组信息。

子设备动态注册

子设备厂商需要在云端对子设备开启动态注册功能，并事先上传子设备的唯一标识（比如SN、MAC地址）。

设备端的代码示例如下：

```
LinkKit.getInstance().getGateway().gatewaySubDevicRegister(getSubDevList(), new IConnectSendListener() {
@Override
public void onResponse(ARequest aRequest, AResponse aResponse) {
ALog.d(TAG, "onResponse() called with: aRequest = [" + aRequest + "], aResponse = [" + (aResponse == null ?
"null" : aResponse.data) + "]");
try {
// 子设备动态注册结果
ResponseModel<List<DeviceInfo>> response = JSONObject.parseObject(aResponse.data.toString(), new
TypeReference<ResponseModel<List<DeviceInfo>>>() {
}.getType());
if (response != null && "200".equals(response.code)) {
/** 获取 deviceSecret, 存储到本地，在添加子设备的时候需要使用deviceSecret
*/
// deviceSecret = response.data.get("deviceSecret");
return;
}
} catch (Exception e) {
e.printStackTrace();
```

```
}
}

@Override
public void onFailure(ARequest aRequest, AError aError) {
// 子设备动态注册失败
}
});
```

获取子设备列表

获取网关当前在云端已经注册了哪些子设备。但是一般来说，网关会记录自己已在云端注册了哪些子设备，不需要从云端去获取该列表。

```
LinkKit.getInstance().getGateway().gatewayGetSubDevices(new IConnectSendListener() {
@Override
public void onResponse(ARequest aRequest, AResponse aResponse) {
// 获取子设备列表结果
try {
ResponseModel<List<DeviceInfo>> response = JSONObject.parseObject(aResponse.data.toString(), new
TypeReference<ResponseModel<List<DeviceInfo>>>() {
}.getType());
// TODO 根据实际应用场景处理
} catch (Exception e) {
e.printStackTrace();
}
}

@Override
public void onFailure(ARequest aRequest, AError aError) {
// 获取子设备列表失败
}
});
```

添加子设备

网关发现并连接了一个新的子设备、并获取到子设备的三元组后，可以告知云端网关需要添加一个子设备。代码示例如下：

```
final DeviceInfo info = new DeviceInfo();
info.productKey = productKey; // 三元组 产品型号（必填）
info.deviceName = deviceName; // 三元组 设备标识（必填）
info.deviceSecret= deviceSecret; // 三元组 设备密钥（必填）
LinkKit.getInstance().getGateway().gatewayAddSubDevice(info, new ISubDeviceConnectListener() {
@Override
public String getSignMethod() {
// 使用的签名方法
return "hmacsha1";
}
```

```
@Override
public String getSignValue() {
// 获取签名，用户使用 deviceSecret 获得签名结果
Map<String, String> signMap = new HashMap<>();
signMap.put("productKey", info.productKey);
signMap.put("deviceName", info.deviceName);
signMap.put("clientId", getClientId());
return SignUtils.hmacSign(signMap, info.deviceSecret);
}

@Override
public String getClientId() {
// clientId 可为任意值
return "id";
}
@Override
public Map<String, Object> getSignExtraData() {
return null;
}

@Override
public void onConnectResult(boolean isSuccess, ISubDeviceChannel iSubDeviceChannel, AError aError) {
// 添加结果
if (isSuccess) {
// 子设备添加成功，接下来可以做子设备上线的逻辑
// subDevOnline(null);
}
}

@Override
public void onDataPush(String s, AMessage message) {
// 收到子设备下行数据 topic=" + s + ", data=" + message
// 如禁用 删除 已经 设置、服务调用等 返回的数据message.data 是 byte[]
}
});
```

注: 网关重启之后并连接到阿里云之后，对连接的子设备需要再次调用添加子设备方法。

删除子设备

此处指删除网关和子设备的拓扑关系。

```
final DeviceInfo deviceInfo = new DeviceInfo();
deviceInfo.productKey = productKey; // 三元组 产品型号（必填）
deviceInfo.deviceName = deviceName; // 三元组 设备标识 （必填）
LinkKit.getInstance().getGateway().gatewayDeleteSubDevice(deviceinfo, new ISubDeviceRemoveListener() {
@Override
public void onSuceess() {
// 成功删除子设备 删除之前可先做下线操作
}

@Override
public void onFailed(AError aError) {
```

```
// 删除子设备失败
}
});
```

子设备上线

调用子设备上线之前，请确保已完成子设备添加。网关发现子设备连上网关之后，需要告知云端子设备上线，子设备上线之后可以执行子设备的订阅、发布等操作。**注意：由于接口调用都是异步的，子设备上线接口不能在子设备添加的下一行调用，而是要放到子设备添加成功的回调里面调用。**

```
final DeviceInfo deviceInfo = new DeviceInfo();
deviceInfo.productKey = productKey; // 三元组 产品型号（必填）
deviceInfo.deviceName = deviceName; // 三元组 设备标识（必填）
LinkKit.getInstance().getGateway().gatewaySubDeviceLogin(deviceinfo, new ISubDeviceActionListener() {
@Override
public void onSuccess() {
// 代理子设备上线成功
// 上线之后可订阅 删除和禁用的下行通知
// subDevDisable(null);
}

@Override
public void onFailed(AError aError) {
ALog.d(TAG, "onFailed() called with: aError = [" + aError + "]");
}
});
```

子设备下线

当子设备离线之后，网关需要告知云端子设备离线，以避免云端向子设备发送数据。子设备下线之后不可以进行子设备的发布、订阅、取消订阅等操作。

```
final DeviceInfo deviceInfo = new DeviceInfo();
deviceInfo.productKey = productKey; // 三元组 产品型号（必填）
deviceInfo.deviceName = deviceName; // 三元组 设备标识（必填）
LinkKit.getInstance().getGateway().gatewaySubDeviceLogout(deviceinfo, new ISubDeviceActionListener() {
@Override
public void onSuccess() {
// 代理子设备下线成功
}

@Override
public void onFailed(AError aError) {
// 代理子设备下线失败
}
});
```

监听子设备禁用

网关设备可以在云端操作子设备，如禁用子设备、启用子设备、删除和子设备的拓扑关系。目前服务端只支持禁用子设备的下行通知。服务端在禁用子设备的时候会对子设备做下线处理，后续网关将不能代理子设备和云端做通信。

```
final DeviceInfo deviceInfo = new DeviceInfo();
deviceInfo.productKey = productKey; // 三元组 产品型号（必填）
deviceInfo.deviceName = deviceName; // 三元组 设备标识 （必填）
LinkKit.getInstance().getGateway().gatewaySetSubDeviceDisableListener(deviceinfo, new IConnectRrpcListener() {
@Override
public void onSubscribeSuccess(ARequest aRequest) {
// 订阅成功
}

@Override
public void onSubscribeFailed(ARequest aRequest, AError aError) {
// 订阅失败
}

@Override
public void onReceived(ARequest aRequest, IConnectRrpcHandle iConnectRrpcHandle) {
// 子设备禁用通知
iConnectRrpcHandle.onRrpcResponse(null, null);
}

@Override
public void onResponseSuccess(ARequest aRequest) {
Log.d(TAG, "onResponseSuccess() called with: aRequest = [" + aRequest + "]");
}

@Override
public void onResponseFailed(ARequest aRequest, AError aError) {
Log.d(TAG, "onResponseFailed() called with: aRequest = [" + aRequest + "], aError = [" + aError + "]");
}
});
```

代理子设备物模型上下行

- 子设备物模型初始化子设备物模型初始化必须在子设备添加到网关下，且子设备已经登录的情况下才可以调用。**注意：由于接口调用都是异步的，子设备物模型初始化接口不能在子设备登录的下一行调用，而是要放到子设备登录成功的回调里面调用。**

```
DeviceInfo deviceInfo = new DeviceInfo();
deviceInfo.productKey = productKey;
deviceInfo.deviceName = deviceName;
// deviceInfo.deviceSecret = "xxxx";
Map<String, ValueWrapper> subDevInitState = new HashMap<>();
// subDevInitState.put(); //TODO 用户根据实际情况设置
```

```
String tsl = null;// 用户根据实际情况设置，默认为空 直接从云端获取最细的 TSL
LinkKit.getInstance().getGateway().initSubDeviceThing(tsl, deviceInfo, subDevInitState, new
IDMCallback<InitResult>() {
@Override
public void onSuccess(InitResult initResult) {
// 物模型初始化成功之后 可以做服务注册 上报等操作
}

@Override
public void onFailure(AError aError) {
// 子设备初始化失败
}
});
```

- 子设备物模型使用 接口使用和直连设备的物模型使用一致，获取 IThing 接口实现使用如下方式获取
  。

```
// 获取 IThing 实例
IThing thing = LinkKit.getInstance().getGateway().getSubDeviceThing(mBaseInfo).first;
// thing 有可能为空，如子设备未登录、物模型未初始化、子设备未添加到网关、子设备处于离线状态等。
// error 信息在 LinkKit.getInstance().getGateway().getSubDeviceThing(mBaseInfo).second 返回
// 参考示例 注意判空
thing.thingPropertyPost(reportData, new IPublishResourceListener() {
@Override
public void onSuccess(String s, Object o) {
// 设备上报状态成功
}

@Override
public void onError(String s, AError aError) {
// 设备上报状态失败
}
});
```

- 子设备物模型销毁 反初始化物模型，后续如果需要重新使用需要重新走登录、物模型初始化流程。

```
LinkKit.getInstance().getGateway().uninitSubDeviceThing(mBaseInfo);
```

代理子设备基础上下行

使用网关的通道执行子设备的数据上下行。

```
final DeviceInfo deviceInfo = new DeviceInfo();
deviceInfo.productKey = productKey; // 三元组 产品型号（必填）
deviceInfo.deviceName = deviceName; // 三元组 设备标识 （必填）
String topic = xxx;
String publishData = xxx;
// 订阅
```

```
LinkKit.getInstance().getGateway().gatewaySubDeviceSubscribe(topic, deviceinfo, new ISubDeviceActionListener() {
@Override
public void onSuccess() {
// 代理子设备订阅成功
}

@Override
public void onFailed(AError aError) {
// 代理子设备订阅失败
}
});

//发布
LinkKit.getInstance().getGateway().gatewaySubDevicePublish(topic, publishData, deviceinfo, new
ISubDeviceActionListener() {
@Override
public void onSuccess() {
// 代理子设备发布成功
}

@Override
public void onFailed(AError aError) {
// 代理子设备发布失败
}
});

// 取消订阅
LinkKit.getInstance().getGateway().gatewaySubDeviceUnsubscribe(topic, deviceinfo, new ISubDeviceActionListener()
{
@Override
public void onSuccess() {
// 代理子设备取消订阅成功
}

@Override
public void onFailed(AError aError) {
// 代理子设备取消订阅事变
}
});
```

设备影子

如果当前产品不具备物模型的能力，可以通过设备影子将当前设备的最新状态缓存到云端。云端缓存的是一个最新的 JSON 格式数据，需要用户自己根据实际情况做解析。物模型具备更高级的设备影子能力，能根据各个属性、事件、服务做独立展示，并具有所有操作的历史记录。

说明 设备影子相关接口参见设备 IDeviceShadow

数据上行

- 获取云端设备影子
- 更新云端设备影子
- 删除云端设备影子

```
// 更新设备影子 需要根据获得到的设备影子读取返回的 version值，在更新的时候 {ver} 替换为version+1
String shadowUpdate = "{" + "\"method\": \"update\"," + "\"state\": {" + "\"reported\": {" +
"\"color\": \"red\"" + ",\"mode\": \"1\"" + "}" + "}," + "\"version\": {ver}" + "}";

// 获取设备影子
String shadowGet = "{" + "\"method\": \"get\"" + "}";

//删除设备影子 color 属性 {ver}需要替换
String shadowDelete = "{" + "\"method\": \"delete\"," + "\"state\": {" + "\"reported\": {" +
"\"color\": \"null\"" + "}" + "}," + "\"version\": {ver}" + "}";

//删除设备影子所有属性 {ver}需要替换
String shadowDeleteAll = "{" + "\"method\": \"delete\"," + "\"state\": {" +
"\"reported\":\"null\"" + "}," + "\"version\": {ver}" + "}";

String requestData = shadowUpdate.replace("{ver}";// shadowGet;shadowDelete.replace("{ver}"
LinkKit.getInstance().getDeviceShadow().shadowUpload(requestData, new IConnectSendListener() {
@Override
public void onResponse(ARequest aRequest, AResponse aResponse) {
try {
if (aRequest instanceof MqttPublishRequest && aResponse != null) {
String dataStr = null;
if (aResponse.data instanceof byte[]) {
dataStr = new String((byte[]) aResponse.data, "UTF-8");
} else if (aResponse.data instanceof String) {
dataStr = (String) aResponse.data;
} else {
dataStr = String.valueOf(aResponse.data);
}
// 返回数据示例
//
{"method":"reply","payload":{"status":"success","state":{"reported":{}},"metadata":{"reported":{}}},"timestamp":154764
1855,"version":7,"clientToken":"null"}

ShadowResponse<String> response = JSONObject.parseObject(dataStr, new
TypeReference<ShadowResponse<String>>() {
}.getType());
if (response != null && response.version != null) {
version = Integer.valueOf(response.version);
}
}
} catch (NumberFormatException e) {
e.printStackTrace();
//ALog.e(TAG, "update version failed.");
} catch (Exception e) {
//ALog.e(TAG, "update response parse exception.");
}
```

```
}

@Override
public void onFailure(ARequest aRequest, AError aError) {
//ALog.d(TAG, "onFailure() called with: aRequest = [" + aRequest + "], aError = [" + aError + "]");
}
});
```

数据下行

```
// 监听云端设备影子更新
LinkKit.getInstance().getDeviceShadow().setShadowChangeListener(new IShadowRRPC() {
@Override
public void onSubscribeSuccess(ARequest aRequest) {
// ALog.d(TAG, "设备影子下行订阅成功");
// ALog.d(TAG, "onSubscribeSuccess() called with: aRequest = [" + aRequest + "]");
}

@Override
public void onSubscribeFailed(ARequest aRequest, AError aError) {
// ALog.d(TAG, "设备影子下行订阅失败");
// ALog.d(TAG, "onSubscribeFailed() called with: aRequest = [" + aRequest + "], aError = [" + aError + "]");
}

@Override
public void onReceived(ARequest aRequest, AResponse aResponse, IConnectRrpcHandle iConnectRrpcHandle) {
// ALog.d(TAG, "onReceived() called with: aRequest = [" + aRequest + "], iConnectRrpcHandle = [" +
iConnectRrpcHandle + "]");
// TODO user logic
// ALog.d(TAG, "收到设备影子下行指令");
try {
if (aRequest != null) {
String dataStr = null;
if (aResponse.data instanceof byte[]) {
dataStr = new String((byte[]) aResponse.data, "UTF-8");
} else if (aResponse.data instanceof String) {
dataStr = (String) aResponse.data;
} else {
dataStr = String.valueOf(aResponse.data);
}
ALog.d(TAG, "dataStr = " + dataStr);
// 返回数据示例
//
{"method":"reply","payload":{"status":"success","state":{"reported":{}},"metadata":{"reported":{}}},"timestamp":154764
1855,"version":7,"clientToken":"null"}
ShadowResponse<String> shadowResponse = JSONObject.parseObject(dataStr, new
TypeReference<ShadowResponse<String>>() {
}.getType());
if (shadowResponse != null && shadowResponse.version != null) {
version = Integer.valueOf(shadowResponse.version);
}

AResponse response = new AResponse();
// TODO 用户实现控制设备
```

```
// 用户控制设备之后 上报影子的值到云端
// 上报设置之后的值到云端
// 根据当前实际值上报
response.data = shadowUpdate.replace("{ver}", String.valueOf(++version));
// 第一个值 replyTopic 有默认值 用户不需要设置
iConnectRrpcHandle.onRrpcResponse(null, response);
}
} catch (Exception e) {
e.printStackTrace();
}
}

@Override
public void onResponseSuccess(ARequest aRequest) {
ALog.d(TAG, "onResponseSuccess() called with: aRequest = [" + aRequest + "]");
}

@Override
public void onResponseFailed(ARequest aRequest, AError aError) {
ALog.w(TAG, "onResponseFailed() called with: aRequest = [" + aRequest + "], aError = [" + aError + "]");
}
});
```