设备接入Link Kit SDK

Android SDK

Android SDK

工程配置

您可以使用物联网平台提供的Android SDK，搭建设备与云端的双向数据通道。SDK包含设备动态注册、初始化建联和数据上下行的接口等内容。

Android SDK Demo

Android SDK提供Demo，供您参考使用。
单击下载Android SDK Demo。下载本Demo将默认您同意本软件许可协议。

配置

在Android工程根目录下的build.gradle 基础配置文件中，加入阿里云仓库地址，进行仓库配置。

```
allprojects {
repositories {
jcenter()
google()
// 阿里云仓库地址
maven {
url "http://maven.aliyun.com/nexus/content/repositories/releases/"
}
maven {
url "http://maven.aliyun.com/nexus/content/repositories/snapshots"
}
}
}
```

在模块的 build.gradle 中，添加SDK的依赖，引入 SDK :iot-linkkit。

```
compile('com.aliyun.alink.linksdk:iot-linkkit:1.6.6')
```

混淆配置

```
# linkkit API
-keep class com.aliyun.alink.**{*;}
-keep class com.aliyun.linksdk.**{*;}
-dontwarn com.aliyun.**
-dontwarn com.alibaba.**
-dontwarn com.alipay.**
-dontwarn com.ut.**

# keep native method
-keepclasseswithmembernames class * {
native <methods>;
}

# keep netty
-keepattributes Signature,InnerClasses
-keepclasseswithmembers class io.netty.** {
*;
}
-dontwarn io.netty.**
-dontwarn sun.**

# keep mqtt
-keep public class org.eclipse.paho.**{*;}

# keep fastjson
-dontwarn com.alibaba.fastjson.**
-keep class com.alibaba.fastjson.**{*;}

# keep gson
-keep class com.google.gson.** { *;}

# keep network core
-keep class com.http.**{*;}

# keep okhttp
-dontwarn okhttp3.**
-dontwarn okio.**
-dontwarn javax.annotation.**
-keep class okio.**{*;}
-keep class okhttp3.**{*;}
-keep class org.apache.commons.codec.**{*;}
```

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三种方法，不同方法需填写不同信息。

- 使用三元组认证方式，需要用户为每个设备提供ProductKey、DeviceName和DeviceSecret。
- 使用动态注册认证方式，需要有ProductKey、ProductSecret和DeviceName，并在控制台开启动态注册。
- 使用ID2认证方式

三元组认证方式

三元组认证方式指设备在出厂前已经为每个设备烧写了ProductKey、DeviceName、DeviceSecret，注意每个设备的三元组不能一样，否则会出现一个设备上线导致另外一个设备下线的情况。

初始化代码如下所示：

```
/**
* 设置设备三元组信息
*/
DeviceInfo deviceInfo = new DeviceInfo();
deviceInfo.productKey = productKey;// 产品类型
deviceInfo.deviceName = deviceName;// 设备名称
deviceInfo.deviceSecret = deviceSecret;// 设备密钥

/**
* 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<>();
// 示例
// propertyValues.put("LightSwitch", new ValueWrapper.BooleanValueWrapper(0));

IoTMqttClientConfig clientConfig = new IoTMqttClientConfig(productKey, deviceName, deviceSecret);

LinkKitInitParams params = new LinkKitInitParams();
params.deviceInfo = deviceInfo;
params.propertyValues = propertyValues;
params.mqttClientConfig = clientConfig;
/**
* 设备初始化建联
* onError 初始化建联失败，需要用户重试初始化。如因网络问题导致初始化失败。
* onInitDone 初始化成功
*/
LinkKit.getInstance().init(context, params, new ILinkKitConnectListener() {
@Override
```

```
public void onError(AError error) {
// 初始化失败 error包含初始化错误信息
}

@Override
public void onInitDone(Object data) {
// 初始化成功 data 作为预留参数
}
});
```

动态注册

动态注册指设备出厂前烧写了ProductKey、ProductSecret以及DeviceName，其中DeviceName是厂商指定的，通常为设备的MAC地址或者SN，由于设备即使不连接阿里云也需要烧写MAC地址或者SN，所以使用动态注册方案对设备厂商的产线无需做更改，因为ProductKey、ProductSecret可以在软件固件中进行指定。

动态注册将会从云端获取设备的DeviceSecret，然后仍然使用三元组认证方式与云端交互来进行设备认证。所以**设备需要将收到的DeviceSecret持久化存储，确保设备进行OTA升级、配置清除之后仍然存在。**

动态初始化成功之后不能再执行动态初始化，后续应用重新启动（包括卸载后重装启动）都需要从持久化存储中获取三元组，然后执行初始化建联（即三元组认证方式与云端建立连接）。

```
// ####### 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，云端会返回已注册错误信息。
* 因此用户在编程时首先需要判断设备是否已获取过deviceSecret，没有获取过的情况下再
* 调用动态注册接口去获取deviceSecret
*/
DeviceInfo myDeviceInfo = new DeviceInfo();
myDeviceInfo.productKey = productKey;
myDeviceInfo.deviceName = deviceName;
myDeviceInfo.productSecret = productSecret;
LinkKitInitParams params = new LinkKitInitParams();
params.deviceInfo = myDeviceInfo;
// 设置动态注册请求 path 和 域名，域名使用默认即可
HubApiRequest hubApiRequest = new HubApiRequest();
hubApiRequest.path = "/auth/register/device";
LinkKit.getInstance().deviceRegister(context, params, hubApiRequest, new IConnectSendListener() {
@Override
public void onResponse(ARequest aRequest, AResponse aResponse) {
// aRequest 用户的请求数据
if (aResponse != null && aResponse.data != null) {
ResponseModel<Map<String, String>> response = JSONObject.parseObject(aResponse.data.toString(),
new TypeReference<ResponseModel<Map<String, String>>>() {
}.getType());
if ("200".equals(response.code) && response.data != null && response.data.containsKey("deviceSecret") &&
!TextUtils.isEmpty(response.data.get("deviceSecret"))) {
deviceInfo.deviceSecret = response.data.get("deviceSecret");
// getDeviceSecret success, to build connection.
// 持久化 deviceSecret 初始化建联的时候需要
// TODO 用户需要按照实际场景持久化设备的三元组信息，用于后续的连接
```

```
// 成功获取 deviceSecret，调用初始化接口建联
// TODO 调用设备初始化建联
}
}
}

@Override
public void onFailure(ARequest aRequest, AError aError) {
Log.d(TAG, "onFailure() called with: aRequest = [" + aRequest + "], aError = [" + aError + "]");
}
});
// ####### 一型一密动态注册接口结束 ######
```

ID2设备认证

SDK初始化的时候添加以下设置：

```
/**
* 云端创建使用 iTLS 认证方式的设备采用这种方式初始化
* 产品需要到 ID2 授权
*/
IoTMqttClientConfig clientConfig = new IoTMqttClientConfig(productKey, deviceName, deviceSecret);
clientConfig.channelHost = productKey + ".itls.cn-shanghai.aliyuncs.com:1883";
clientConfig.productSecret = productSecret;
clientConfig.secureMode = 8;
linkKitInitParams.mqttClientConfig = clientConfig;
```

SDK 反初始化

如果需要注销初始化，调用如下接口。

```
// 取消注册 notifyListener，notifyListener对象需和注册的时候是同一个对象
LinkKit.getInstance().unRegisterOnPushListener(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().registerOnPushListener(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(productKey, deviceName, deviceSecret);
// 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。

```
HubApiRequest 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
hubApiRequest.path = "/auth/register/device";
```

Mqtt 连接参数

- 设置Mqtt Keep-Alive 时间

```
// 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

```
IoTMqttClientConfig clientConfig = new IoTMqttClientConfig(productKey, deviceName,
deviceSecret);
// 对应 receiveOfflineMsg = !cleanSession, 默认不接受离线消息
clientConfig.receiveOfflineMsg = true;
```

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() {
@Override
public void onSuccess(String resID, Object o) {
// 属性上报成功 resID 设备属性对应的唯一标识
}

@Override
public void onError(String resId, 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(hashMap);
LinkKit.getInstance().getDeviceThing().thingEventPost(identifier, params, new IPublishResourceListener() {
@Override
public void onSuccess(String resId, Object o) { // 事件上报成功
}

@Override
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
LinkKit.getInstance().getDeviceThing().setServiceHandler(service.getIdentifier(),
mCommonHandler);
// 服务处理的handler
private ITResRequestHandler mCommonHandler = new ITResRequestHandler() {
@Override
public void onProcess(String identify, Object result, ITResResponseCallback
itResResponseCallback) {
// 收到云端异步服务调用 identify 设备端属性或服务唯一标识 result 下行服务调用数据
// iotResResponseCallback 用户处理完服务调用之后响应云端 具体使用参见 Demo 代码
try {
if (SERVICE_SET.equals(identify)) { // set 异步服务调用
// 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();
// 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)){ // get异步服务调用
// 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 { // 用户定义服务调用
// 根据不同的服务做不同的处理，跟具体的服务有关系
OutputParams outputParams = new OutputParams();
// outputParams.put("op", new ValueWrapper.IntValueWrapper(20));
itResResponseCallback.onComplete(identify,null, outputParams);
}
} catch (Exception e) {
e.printStackTrace();
}
}

@Override
public void onSuccess(Object o, OutputParams outputParams) {
// 服务注册成功 tag：用户传入的tag，未使用到 outputParams：异步回调成功的返回数据
,outparams等类型
}

@Override
public void onFail(Object o, ErrorInfo errorInfo) {
// 服务注册失败
}
};
```

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/")) {
showToast("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) {
Log.d(TAG, "onResponse() called with: aRequest = [" + aRequest + "], aResponse = [" + aResponse + "]");
}

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

自定义MQTT Topic通信

Android 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。如果不想使用物模型，可以通过这部分接口实现云端数据的上下行。

上行接口请求

调用上行请求接口，SDK封装了上行发布、订阅和取消订阅等接口。

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

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

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

调用示例：QoS 设置可以参照该示例进行设置。

```
// 发布
MqttPublishRequest request = new MqttPublishRequest();
request.isRPC = false;
// topic 替换成用户自己需要发布的 topic
request.topic = topic;
// 设置 qos
request.qos = 0;
// data 替换成用户需要发布的数据
request.payloadObj = data;
LinkKit.getInstance().publish(request, new IConnectSendListener() {
@Override
public void onResponse(ARequest aRequest, AResponse aResponse) {
// 发布成功
}

@Override
public void onFailure(ARequest aRequest, AError aError) {
// 发布失败
}
});
// 订阅
MqttSubscribeRequest subscribeRequest = new MqttSubscribeRequest();
// subTopic 替换成用户自己需要订阅的 topic
subscribeRequest.topic = subTopic;
subscribeRequest.isSubscribe = true;
LinkKit.getInstance().subscribe(subscribeRequest, new IConnectSubscribeListener() {
@Override
public void onSuccess() {
// 订阅成功
}

@Override
public void onFailure(AError aError) {
// 订阅失败
}
});
// 取消订阅
MqttSubscribeRequest unsubRequest = new MqttSubscribeRequest();
// unSubTopic 替换成用户自己需要取消订阅的 topic
unsubRequest.topic = unSubTopic;
unsubRequest.isSubscribe = false;
LinkKit.getInstance().unsubscribe(unsubRequest, new IConnectUnscribeListener() {
@Override
public void onSuccess() {
// 取消订阅成功
}

@Override
public void onFailure(AError aError) {
// 取消订阅失败
}
});
```

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
*
* @param request RRPC 请求
* @param listener 监听器
*/
void subscribeRRPC(ARequest request, IConnectRrpcListener listener);

/**
* 注册下行数据监听器
*
* @param listener 监听器
*/
void registerOnPushListener(IConnectNotifyListener listener);

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

调用示例（数据监听）：

```
// 下行数据监听
IConnectNotifyListener onPushListener = new IConnectNotifyListener() {
@Override
public void onNotify(String connectId, String topic, AMessage aMessage) {
// 下行数据通知
}

@Override
public boolean shouldHandle(String connectId, String topic) {
return true; // 是否需要处理 该 topic
}

@Override
public void onConnectStateChange(String connectId, ConnectState connectState) {
// 连接状态变化
}
};
// 注册
LinkKit.getInstance().registerOnPushListener(onPushListener);
// 取消注册
LinkKit.getInstance().unRegisterOnPushListener(onPushListener);
```

调用示例（RRPC）：

```
final MqttRrpcRegisterRequest registerRequest = new MqttRrpcRegisterRequest();
// rrpcTopic 替换成用户自己自定义的 RRPC topic
registerRequest.topic = rrpcTopic;
// rrpcReplyTopic 替换成用户自己定义的RRPC 响应 topic
registerRequest.replyTopic = rrpcReplyTopic;
// 根据需要填写，一般不填
// registerRequest.payloadObj = payload;
// 先订阅 rrpcTopic
// 云端发布消息到 rrpcTopic
// 收到下行数据 回复云端（rrpcReplyTopic）具体可参考 Demo 同步服务调用
LinkKit.getInstance().subscribeRRPC(registerRequest, 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) {
// 收到云端下行
// 响应获取成功
if (iConnectRrpcHandle != null){
AResponse aResponse = new AResponse();
// 仅供参考，具体返回云端的数据用户根据实际场景添加到data结构体
aResponse.data = "{\"id\":\"" + 123 + "\", \"code\":\"200\"" + ",\"data\":{} }";
iConnectRrpcHandle.onRrpcResponse(registerRequest.replyTopic, aResponse);
}
}

@Override
public void onResponseSuccess(ARequest aRequest) {
// RRPC 响应成功
}

@Override
public void onResponseFailed(ARequest aRequest, AError aError) {
// RRPC 响应失败
}
});
```

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

```
String COTA_get = "{" + "  \"id\": 123," + "  \"version\": \"1.0\"," +
" \"params\": {" + "\"configScope\": \"product\"," + "\"getType\": \"file\"" +
" }," + " \"method\": \"thing.config.get\"" + "}";

RequestModel<Map> requestModel = JSONObject.parseObject(COTA_get, new
TypeReference<RequestModel<Map>>() {
}.getType());
LinkKit.getInstance().getDeviceCOTA().COTAGet(requestModel, new IConnectSendListener() {
@Override
public void onResponse(ARequest aRequest, AResponse aResponse) {
// 获取远程配置结果成功
}

@Override
public void onFailure(ARequest aRequest, AError aError) {
// 获取远程配置失败
}
});
```

订阅获取

设备端可以通过订阅获取远程配置信息

```
// 注意：云端目前有做限制，一个小时只能全品类下发一次 COTA 更新
LinkKit.getInstance().getDeviceCOTA().setCOTAChangeListener(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) {
// 接收到远程配置下行数据
if (aRequest instanceof MqttRrpcRequest){
// 云端下行数据 返回数据示例参见 Demo
// ((MqttRrpcRequest) aRequest).payloadObj;
}
}

@Override
public void onResponseSuccess(ARequest aRequest) {
```

```
// 回复远程配置成功
}

@Override
public void onResponseFailed(ARequest aRequest, AError aError) {
// 回复远程配置失败
}
});
```

设备OTA开发

OTA基本流程：

1. 设备上报版本号
2. 设备订阅 OTA 相关topic
3. 在 OTA 后台配置 OTA 任务，可以按多纬度知道要升级的设备，地址：https://iot.console.aliyun.com/product
4. 成功订阅了 OTA 相关topic的设备在当前配置的 OTA 任务中，设备会收到一个推送信息，其中包括：
    i. 可升级的版本号
    ii. 固件地址，大小，MD5
5. 下载固件，开始升级，并上报升级进度
6. 升级完成后，自动上报新的版本号，整个 OTA 流程结束

如何接入

在后台配置 OTA 升级任务，获取 OTA 实例：

```
mOta = LinkKit.getInstance().getOta()
```

准备OTA升级任务，目前自动完成版本上报，订阅 topic, 当有 ota 推送时，会通过 IOta.STEP_RCVD_OTA 回调：

```
mOta.tryStartOta(mConfig, new OtaListener(){
public boolean onOtaProgress(int step, IOta.OtaResult otaResult) {
int code = otaResult.getErrorCode();
Object data = otaResult.getData();
```

```
switch (step) {
case IOta.STEP_REPORT_VERSION:
// 上报版本
break;
case IOta.STEP_SUBSCRIBE:
// 订阅回调
break;

case IOta.STEP_RCVD_OTA:
// 有新的OTA固件，返回true 表示继续升级
break;
case IOta.STEP_DOWNLOAD:
// 下载固件中
break;
}

return true;
}
});
```

收到推送的OTA 时，返回true ,会自动下载固件，下载进度通过IOta.STEP_DOWNLOAD 回调

4. 固件下载完成后，设备厂商通过自己的OTA 方式升级设备即可

注意：本SDK只提供固件的管理，推送，下载，具体OTA 进度需要开发者者自己定义，通过接口 reportProgress() 完成，同时SDK提供接口 reportVersion() 用于上报版本功能。

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
String update_label = "{" + " \"id\": \"123\"," + " \"version\": \"1.0\"," +
" \"params\": [" + " {" + " \"attrKey\": \"Temperature\"," +
" \"attrValue\": \"36.8\"" + " }" + " ]," +
" \"method\": \"thing.deviceinfo.update\"" + "}";
RequestModel<List<Map>> requestModel = JSONObject.parseObject(update_label, new
TypeReference<RequestModel<List<Map>>>() {
}.getType());
LinkKit.getInstance().getDeviceLabel().labelUpdate(requestModel, new IConnectSendListener() {
@Override
public void onResponse(ARequest aRequest, AResponse aResponse) {
```

```
// 更新标签成功
}

@Override
public void onFailure(ARequest aRequest, AError aError) {
// 更新标签失败
}
});
```

删除标签

```
// 标签是 key，value的形式 可以替换 attrKey 和 attrValue
String deleteLabel = "{" + " \"id\": \"123\"," + " \"version\": \"1.0\"," +
" \"params\": [" + " {" + " \"attrKey\": \"Temperature\"" +
" }" + " ]," + " \"method\": \"thing.deviceinfo.delete\"" + "}";
RequestModel<List<Map>> requestModel = JSONObject.parseObject(deleteLabel, new
TypeReference<RequestModel<List<Map>>>() {
}.getType());
LinkKit.getInstance().getDeviceLabel().labelDelete(requestModel, new IConnectSendListener() {
@Override
public void onResponse(ARequest aRequest, AResponse aResponse) {
// 删除标签成功
}

@Override
public void onFailure(ARequest aRequest, AError aError) {
// 删除标签失败
}
});
```

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，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、代理子设备上下行的能力。

网关本身是一个直连设备可直接使用上述介绍的所有能力。网关和子设备之间的连接、数据通信需要用户处理。

说明 网关子设备管理相关接口参见设备 IGateway

设备每次初始建联的时候都需要调用添加、登录接口。

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

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

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

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

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；

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；

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

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。

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

使用PK&DN动态注册

需要在云端开启允许动态注册功能，可以同时注册多个，用于获取子设备三元组。子设备添加到网关之前需要先进行动态注册获取子设备三元组信息，如果本地已有子设备三元组信息，可以跳过这一步。如果子设备已经被绑定到其他网关设备，动态注册不会返回该子设备的三元组信息。

```
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 {
// 子设备动态注册成功
ResponseModel<List<DeviceInfo>> response = JSONObject.parseObject(aResponse.data.toString(), new
TypeReference<ResponseModel<List<DeviceInfo>>>() {
}.getType());
// 根据 response 的数据判断是否成功 code=200
//TODO 保存子设备的三元组信息
} catch (Exception e) {
e.printStackTrace();
}
}

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

使用PK&DN&PS动态注册

需要注意的是这种动态注册方式涉及到子设备的pk需要预先获取，安全性会低于第一种动态注册方式。如果当前已经有子设备的productKey、deviceName、productSecret信息，并且需要进行抢占式动态注册，则可以使用 这种方式进行子设备动态注册。抢占式动态注册，即被其它网关设备绑定的子设备三元组也会返回。
推荐通过云端远程配置下发（COTA）下发子设备的PK、DN、PS信息到网关设备，然后再进行抢占式动态注册。
应用场景：需要将子设备从A网关绑定到B网关的场景，如根据信号强度切换网关。使用PK&DN实现的话需要子设备先在A网关登出，解除拓扑关系，然后与B建立拓扑关系。

```
// 该动态注册方案需要提前知道子设备的productSecret，安全性会比下面一种子设备动态注册低一点
// 这种动态注册方式可以考虑和COTA-远程配置下发配合使用，在云端下发子设备的pk、dn、ps，网关收到后
// 完成动态注册
// 使用于需要抢占绑定关系时使用，会返回被其他网关设备的子设备
MqttPublishRequest request = new MqttPublishRequest();
final RequestModel requestModel = new RequestModel();
requestModel.id = String.valueOf(IDGenerater.generateId());
requestModel.version = "1.0";
requestModel.method = GatewayChannel.METHOD_PRESET_SUBDEV_REGITER;
request.isRPC = true;
JSONObject jsonObject = new JSONObject();
JSONArray jsonArray = new JSONArray();
for (int i = 0; i < presetSubdevList.size(); i++) {
DeviceInfo itemDev = presetSubdevList.get(i);
Map<String, String> itemMap = new HashMap<>();
itemMap.put("productKey", itemDev.productKey);
itemMap.put("deviceName", itemDev.deviceName);
itemMap.put("random", RandomStringUtil.getRandomString(10));
String sign = SignUtils.hmacSign(itemMap, itemDev.productSecret);
itemMap.put("sign", sign);
itemMap.put("signMethod", "hmacsha1");
jsonArray.add(itemMap);
```

```
}

jsonObject.put("proxieds", jsonArray);
requestModel.params = jsonObject;
request.payloadObj = requestModel.toString();
LinkKit.getInstance().getGateway().subDevicRegister(request, new IConnectSendListener() {
@Override
public void onResponse(ARequest aRequest, AResponse aResponse) {
ALog.d(TAG, "onResponse() called with: aRequest = [" + aRequest + "], aResponse = [" + aResponse + "]");
try {
showToast("收到子设备动态结果");
ResponseModel<Map<String, List<DeviceInfo>>> responseModel =
JSONObject.parseObject(aResponse.data.toString(),
new TypeReference<ResponseModel<Map<String, List<DeviceInfo>>>>() {
}.getType());
// TODO 保存子设备的三元组信息
ALog.d(TAG, "onResponse responseModel=" + JSONObject.toJSONString(responseModel));
//
{"code":200,"data":{"failures":[],"successes":[{"deviceSecret":"xxx","productKey":"xxx","deviceName":"xxx"}]},"id":"1","
message":"success","method":"thing.proxy.provisioning.product_register","version":"1.0"}
// 动态注册成功列表
List<DeviceInfo> successList = responseModel.data.get("successes");
// 动态注册失败列表
List<DeviceInfo> failList = responseModel.data.get("failures");
} catch (Exception e) {
e.printStackTrace();
}
}

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

获取子设备列表

获取网关当前在云端已经有哪些子设备。

```
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

子设备动态注册完成之后，可以通过该接口将子设备添加到网关下。

```
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().gatewayAddSubDevice(deviceinfo, 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("timestamp", String.valueOf(System.currentTimeMillis()));
signMap.put("clientId", getClientId());
return SignUtils.hmacSign(signMap, info.deviceSecret);
}

@Override
public String getClientId() {
// clientId 可为任意值
return "id";
}

@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[]
}
});
```

删除子设备

删除网关下的子设备。

```
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);
// subDevDelete(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
}
});
```

支持PermitJoin

如果网关是接入智能生活开放平台（又称飞燕平台），网关需要在收到来自云端的PermitJoin消息时，再连接子设备、以及添加子设备。目前的版本还不支持对PermitJoin的封装（将在下一个迭代中进行支持），可以通过下面的办法实现对PermitJoin消息的处理：

> 网关需要去发现并连接指定型号的子设备，如果网关找不到指定型号的子设备，那么就无需添加子设备；添加子设备的接口见本章前面描述的“添加子设备”
>
> 如果网关收到PermitJoin之后，发现网关连接了多个指定型号的子设备，只向云端添加其中一个子设备即可

```java// 订阅并监听云端 permitJoin 下行LinkKit.getInstance().getGateway().permitJoin(new IConnectRrpcListener() { @Override public void onSubscribeSuccess(ARequest aRequest) { ALog.d(TAG, “onSubscribeSuccess() called with: aRequest = [ “ + aRequest + “]”); showToast( “permitJoin subscribe success.”); }

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

@Override
public void onReceived(ARequest aRequest, IConnectRrpcHandle iConnectRrpcHandle) {
ALog.d(TAG, "onReceived() called with: aRequest = [" + aRequest + "], iConnectRrpcHandle = [" +
iConnectRrpcHandle + "]");
showToast("接收到permitJoin指令");
// TODO user add sub device &
if (aRequest instanceof MqttRrpcRequest){
// 云端下行数据 拿到
// ((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":"", //消息id
```

// “version”:”1.0”, //ALink协议的版本号// “params”:{// “productKey”:”xxx”, //子设备的型号，如果内容为空，表示网关允许任何子设备接入// “time”:60 //网关发现与添加子设备的窗口时间，int类型，单位为秒// }// } // TODO 发现、添加、登录 子设备
```

```
        // 网关处理完子设备拓扑关系添加、登录后
if (iConnectRrpcHandle != null) {
AResponse response = new AResponse();
// 回复示例 TODO edit by user
// response.data 里的 id 字段需要与收到的 permitJoin 消息的 id 保持一致
response.data = "{\"id\":\"xxx\", \"code\":\"200\"" + ",\"data\":{} }";
iConnectRrpcHandle.onRrpcResponse(((MqttRrpcRequest) aRequest).replyTopic, response);
}
}
}

@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 = [" + aError + "]");
}
```

});```

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

```
// 获取设备影子 具体业务可参考 Demo
String data = "{" + "\"method\": \"get\"" + "}";
// 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 data = "{" + "\"method\": \"update\"," + "\"state\": {" + "\"reported\": {" +
// "\"color\": \"red\"" + "}" + "}," + "\"version\": {ver}" + "}";
//删除设备影子 color 属性 {ver}需要替换
// String data = "{" + "\"method\": \"delete\"," + "\"state\": {" + "\"reported\": {" +
```

```
// "\"color\": \"null\"" + "}" + "}," + "\"version\": {ver}" + "}";
LinkKit.getInstance().getDeviceShadow().shadowUpload(data, new IConnectSendListener() {
@Override
public void onResponse(ARequest aRequest, AResponse aResponse) {
// 设备影子更新成功
// 数据解析参考数据下行或 Demo
}

@Override
public void onFailure(ARequest aRequest, AError aError) {
// 设备影子更新失败
}
});
```

数据下行

监听云端设备影子数据更新，一般使用在 APP 去控制设备的时候。APP控制设备的时候，通过云端下发设备影子更新到设备端，设备端拿到云端下行的设备影子之后，根据 desired 的值去执行设备更新。

```
// 监听云端设备影子更新
LinkKit.getInstance().getDeviceShadow().setShadowChangeListener(new IShadowRRPC() {
@Override
public void onSubscribeSuccess(ARequest aRequest) {
// 订阅设备影子下行数据成功
}

@Override
public void onSubscribeFailed(ARequest aRequest, AError aError) {
// 订阅设备影子下行数据失败
}

@Override
public void onReceived(ARequest aRequest, AResponse aResponse, IConnectRrpcHandle iConnectRrpcHandle) {
// 接收到云端数据下行，下行数据在 aResponse 想里面
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);
}
// Log.d(TAG, "dataStr = " + dataStr);
// 返回数据示例
//{"method":"control","payload":{"state":{"desired":{"mode":2,"color":"white"},"reported":{"mode":"1","color":"red"}},"
metadata":{"desired":{"mode":{"timestamp":1547642408},"color":{"timestamp":1547642408}},"reported":{"mode":{"ti
mestamp":1547642408},"color":{"timestamp":1547642408}}}},"timestamp":1547642408,"version":12}
// 仅供参考
ShadowResponse<String> shadowResponse = JSONObject.parseObject(dataStr, new
TypeReference<ShadowResponse<String>>() {
}.getType());
```

```
if (shadowResponse != null && shadowResponse.version != null &&
TextUtils.isDigitsOnly(shadowResponse.version)) {
version = Long.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 + 1));
// 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) {
// 下行处理之后上报成功
}

@Override
public void onResponseFailed(ARequest aRequest, AError aError) {
// 下行处理之后上报失败
}
});
```

HTTP2 流通道

简介

HTTP 2 流通到 SDK 提供流式数据接入。在一个连接上可以建立多个流通道，使用完流通到需要关闭流通道。不同于视频流，该流通道较轻量级，可以快速打开关闭，适合较有限流数据传输，比如文件上传等。
1.建立和云端的连接；
2.打开一个流通道；
3.发送数据流，可以一次性发完也可以多次发送完；
4.发送方接收云端的响应，可以是一次响应完，也可以一直下推数据；
5.流使用完需要关闭；
6.不需要再使用 HTTP2 的时候需要关闭连接。

初始化

SDK 目前支持两种方式的流通道认证：设备接入认证、APP接入认证。IStreamSender 是 HTTP2 流通道接口类。如果当前Android设备作为设备三元组接入，选择 设备认证初始化，如果是作为一个应用只集成H2能力，则选择 APP接入初始化方式。

设备初始化

```javaLinkKitInitParams params = new LinkKitInitParams();params.deviceInfo = deviceInfo; // 参考Demo// params.propertyValues = propertyValues; // 其他初始化参数// params.connectConfig = userData;

/**

- 如果用户需要设置域名*/IoTH2Config ioTH2Config = new IoTH2Config();

```
 ioTH2Config.clientId = "client-id";
ioTH2Config.endPoint = "https://10.125.15.223:9999";
```

params.iotH2InitParams = ioTH2Config;// 执行linkkit 初始化// linkkit 初始化成功之后可以使用如果方式获取 IStreamSender实例IStreamSender client = LinkKit.getInstance().getH2StreamClient();```

## APP 初始化

Profile 是 APP 认证方式初始化入口，作为设备的时候不需要走 APP 初始化。

```
/**
* replace url appKey appSecret
* url = "https://"
* signMethod = AuthSignMethod.SHA1.getMethod();
*/
Profile profile = Profile.getAppKeyProfile("url", "appKey", new IAuthSign() {
@Override
public String getAuthSign(String signContent) {
@Override
public String getSignMethod() {
return AuthSignMethod.SHA1.getMethod();// set bu user
}
// TODO by user
// use appSecret to sign signContent and return signValue
// hmacMd5Hex、hmacSha1Hex、hmacSha256Hex、hmacSha384Hex、hmacSha512Hex
HmacUtils utils = new HmacUtils(HmacAlgorithms.HMAC_SHA_1, "appSecret");
return new String(Hex.encodeHex(utils.hmac(signContent)));
```
```

```
}
});
IStreamSender client = StreamSenderFactory.streamSender(profile);
```

流通道

建联

CompletableListener 是通用的异步回调接口。

```java// 有可能会抛出已建联、网络相关 exception// client 即初始化获得的 IStreamSender 实现实例
client.connect(new CompletableListener() { @Override public void complete(Object o) { }
```

```
@Override
public void completeExceptionally(Throwable throwable) {
}
```

}););

```
<a name="t2vexk"></a>
## [](#t2vexk)打开流
> ```java
final Http2Request request = new Http2Request();
// 添加请求参数
// request.getHeaders().add("nothing", "ddd");
client.openStream(serviceNme,
request, new CompletableListener<Http2Response>() {
@Override
public void complete(Http2Response http2Response) {
ALog.w(TAG, "open stream result: " + http2Response);
// save dataStreamId of this stream
if (http2Response != null) {
dataStreamId = StreamUtil.getDataStreamId(http2Response.getHeaders());
}
}

@Override
public void completeExceptionally(Throwable throwable) {
showToast("open Stream failed");
Log.w(TAG, "completeExceptionally: ", throwable);
dataStreamOpened.set(false);
}
});
```

发送流

发送流需要使用到 openStream 返回的云端的 dataStreamId，以及需要发送的数据。一次请求分多次发送可以有两种方式：一种是当做普通请求，每次发送header和部分数据，并设置endStream为true；一种是第一次发送 header+data，后续只发送data。
数据发送可以分为以下几种（具体和业务服务端实现的功能有关）：

一次请求，一次响应；类似 HTTP 1.1

一次请求，多次响应；数据下推

一次请求，分多次发送； |header+data | header+data| ... 或 | header + data |+ data|+ data|+...

IDownStreamListener 是下推数据流的数据接收接口。

```javaHttp2Request request = new Http2Request();request.setContent(sendData.getBytes());// 如果数据要多次分多次上传，比如文件上传，需要先设置总的大小，然后分多次上传；// request.setEndOfStream(endOfStream);// request.getHeaders().set(“content-length”, String.valueOf(length));// dataStreamId openStream 云端返回的 dataStreamIdclient.sendStream(dataStreamId, request, new IDownStreamListener() { @Override public void onHeadersRead(String s, Http2Headers http2Headers, boolean b) { ALog.d(TAG, “onHeadersRead() called with: s = [ “ + s + “], http2Headers = [ “ + http2Headers + “], b = [ “ + b + “]”); }

```
@Override
public void onDataRead(String s, byte[] bytes, boolean b) {
ALog.d(TAG, "onDataRead() called with: s = [" + s + "], bytes = [" + new String(bytes) + "], b = [" + b + "]");
}

@Override
public void onStreamError(String s, IOException e) {
ALog.w(TAG, "onStreamError() called with: s = [" + s + "], e = [" + e + "]");
}
}, new CompletableListener<StreamWriteContext>() {
@Override
public void complete(StreamWriteContext streamWriteContext) {
ALog.d(TAG, "complete() called with: streamWriteContext = [" + streamWriteContext + "]");
}

@Override
public void completeExceptionally(Throwable throwable) {
ALog.w(TAG, "completeExceptionally() called with: throwable = [" + throwable + "]");
}
});
```
```


[](#oc3ris)关闭流

> ```java
Http2Request request = new Http2Request();
request.setEndOfStream(true);
client.closeStream(dataStreamId, request, new CompletableListener<Http2Response>() {
@Override
public void complete(Http2Response http2Response) {
ALog.d(TAG, "complete() called with: http2Response = [" + http2Response + "]");
}

@Override
public void completeExceptionally(Throwable throwable) {
ALog.d(TAG, "completeExceptionally() called with: throwable = [" + throwable + "]");
}
});

断连

```java// client 即初始化获得的 IStreamSender 实现实例client.disconnect(new CompletableListener() { @Override public void complete(Object o) { }

@Override
public void completeExceptionally(Throwable throwable) {
}

});

<a name="gl8txp"></a>
## [](#gl8txp)文件上传
文件上传基于流通道实现，在使用文件上传功能的时候需要先建联、打开流，并拿到 dataStreamId。filePath 是文件要上传的绝对路径，包含文件名。需要用户确保该文件具备访问的权限，以及动态申请了文件的读写权限。文件上传采用的是 |header + data| data| data |... 多次请求发送，只有第一个请求会带header。

> ```java
Http2Request request = new Http2Request();
request.getHeaders().add("xxx", "xxx");
client.upload(dataStreamId, filePath, request, new CompletableListener<Http2Response>() {
@Override
public void complete(Http2Response o) {
ALog.d(TAG, "complete() called with: o = [" + o + "]");
// upload success
}
```

```
@Override
public void completeExceptionally(Throwable throwable) {
ALog.w(TAG, "completeExceptionally() called with: throwable = [" + throwable + "]");
// upload fail
}
});
```

设备reset

有的产品设计了reset按钮，用于清除设备上的配置，让设备恢复到出厂状态，这些功能由设备厂商去实现。但是有的场景下，设备商需要云端感知设备上执行了reset操作，那么需要调用Link Kit SDK提供的API告知云端。

Android设备只有用户主动触发reset调用的场景，比如点击按钮或者按键的时候开发者调用SDK的reset接口
reset调用之后，SDK分两个流程处理：
（1）如果设备在线，调用reset接口到云端，成功之后销毁整个linkkit（调用deinit），失败按照流程2处理；
（2）如果设备不在线，设置一个reset的标志位，然后销毁整个linkkit，后续linkkit被重新初始化的时候，根据是否有这个reset标志位在建联成功之后再调用reset接口，reset完成之后才会响应设备的发现；如果APP卸载或删除APP数据之后，则下次启动则不再执行reset操作了，标志位被清除了；

```
LinkKit.getInstance().reset(new IConnectSendListener() {
@Override
public void onResponse(ARequest aRequest, AResponse aResponse) {
ALog.d(TAG, "reset onResponse ");
}

@Override
public void onFailure(ARequest aRequest, AError aError) {
ALog.d(TAG, "reset onFailure");
}
});
```

智能生活开放平台的用户解绑操作

背景说明：

如果产品是在阿里云IoT的智能生活开放平台注册，那么用户只有通过手机APP与设备之间建立绑定关系之后，才能对设备进行控制。如果设备被出售或者转让给了另外一个用户，那么新用户可以通过触按reset按键告知云端解除设备与原用户的绑定关系，避免原用户对设备可以继续进行控制。

此时就需要调用 LinkKit.getInstance().reset 接口来告知云端解除设备与原有用户的绑定关系。

允许重新进行动态注册的操作

背景说明：

如果产品的认证选用了动态注册功能，设备出厂时只需要烧写ProductKey、ProductSecret，设备连接到阿里云物联网时可以通过动态注册过程去获取设备的DeviceSecret；但是当设备获取到了DeviceSecret之后，如果通过动态注册过程去获取设备的DeviceSecret，云端将会拒绝。

因此设备获取到DeviceSecret之后，需要将DeviceSecret固化到非应用存储中，并且即使用户触按reset按键或者卸载APP或者删除APP数据也不要将DeviceSecret删除掉。 如果设备因为某些原因导致DeviceSecret被删掉，为了让该设备能够继续被使用，开发者需要将该设备的ProductKey、DeviceName记录下来，到阿里云控制台删除对应deviceName 的设备，然后重新创建一个相同deviceName的设备即可。厂家需要注意正式发布的产品不允许出现deviceSecret可以被用户删除的情况。

错误码

错误码（新增）

常见错误码

错误码	子错误码	描述	备注
1101100		ERROR_SDK_ERROR	SDK 初始化内部异常
1101101		ERROR_SDK_INIT_ERROR	
	120	ERROR_PARAMS_DEVICEINFO_INVALID	初始化 三元组信息 productKey、deviceName 为空
	121	ERROR_PARAMS_DEVICE_SECRET_NULL	初始化三元组 deviceSecret（设备密钥）为空
	122	ERROR_PARAMS_SECURE_MODE_ITLS_WITH_PS_NULL	使用 itls 认证模式，但是三元组 productSecret （产品密钥）为空
-33		MQTT_CONNECT_ERROR	mqtt建联失败 *检查是否网络正常；* 检查三元组是否正确； *域名是否设置正确；* 是否多个设备使用同一

			个三元组
-4		ERROR_HTTP	HTTP 请求接口错误，如一型一密动态注册失败 *检测域名是否设置正确；* 网络是否有异常； * 系统时间是否正常；
1101220		ERROR_COTA_GET_PARAMS_ERROR	获取远程配置请求参数错误
1101230		ERROR_SHADOW_INVALID_STATE	SDK尚未初始化调用设备影子相关接口
1101231		ERROR_SHADOW_UPDATE_FAILED	设备影子更新失败，具体错误信息参考error message
1101232		ERROR_SHADOW_PARAMS_INVALID	设备影子更新参数错误
1101312		ERROR_GATEWAY_PERMIT_JOIN_DEVICE_INFO_INVALID	permitJoin调用的时候初始化的设备信息无效
1101300		ERROR_GATEWAY_TOPO_NOT_ADDED	尚未添加拓扑关系
1101301		ERROR_GATEWAY_SUBDEV_NOT_LOGIN	子设备尚未登录
1101302		ERROR_GATEWAY_SUBDEV_WRAPPER_INFO_NULL	SDK内部子设备信息为空
1101303		ERROR_GATEWAY_SUBDEV_WRAPPER_NULL	SDK内部子设备不存在
1101304		ERROR_GATEWAY_SUBDEV_THING_NOT_INITED	子设备物模型未初始化
1101305		ERROR_GATEWAY_SUBDEV_LABEL_NULL	获取子设备标签为空，一般是子设备未添加后登录
1101306		ERROR_GATEWAY_SUBDEV_SHADOW_NULL	获取子设备设备影子为空，一般是子设备未添加后登录
1101307		ERROR_GATEWAY_SUBDEV_COTA_NULL	获取子设备远程配置为空，一般是子设备未添加后登录
1101308		ERROR_GATEWAY_SUBDEV_INFO_INVALID	子设备三元组信息无效
1101309		ERROR_GATEWAY_SUBDEV_DISABLED	子设备已被云端禁用

1101310		ERROR_GATEWAY_SUBDEV_DELETED	子设备已被删除
1101311		ERROR_GATEWAY_LABEL_PARAMS_INVALID	标签请求参数无效
1101200		ERROR_TMP_INIT	初始化失败，TMP初始化失败
1101201		ERROR_DM_GET_TSL_INFO_INVALID	初始化失败，获取TSL信息无效
1101202		ERROR_DM_INIT_THING_PARAMS_INVALID	初始化失败，物模型获取参数无效
1101203		ERROR_DM_GET_TMP_IDEVICE	初始化失败，物模型获取失败
1101020		ERROR_DUPLICATE_SDK_INIT	重复初始化，当前初始化已完成或者正在初始化
100		ERROR_DUPLICATE_SDK_INIT_DM	设备管理模块重复初始化
101		ERROR_DUPLICATE_SDK_INIT_LK	LinkKit SDK重复初始化
510		ERROR_CMP_PARAMS_ERROR	CMP 参数错误
514		ERROR_CMP_REGISTER_CONNECT_ERROR_EXIST	该链接类型已注册，一般可以忽略
517		ERROR_CMP_SEND_ERROR_CONNECT_NOT_FOUND	发送失败，当前连接类型不存在
521		ERROR_CMP_SEND_ERROR_CONNECT_NOT_CONNECTED	发送失败，连接未建立
529		ERROR_CMP_REGISTER_CONNECT_IS_REGISTERING	连接类型正在注册
4201		ERROR_UNKNOW	客户端内部错误
4101		ERROR_NETWORK_ERROR	网络错误
4102		ERROR_SERVER	业务网关错误
4103		ERROR_BUSINESS	业务错误

常见问题

RRPC发送应答出错，返回错误码4201

问题描述：当设备端收到RRPC调用后，发送应答出错，错误码4201

问题原因：客户组装的数据不是一个正确的JSON数据，导致转换时出错。

解决办法：将发送的数据转换成正确的JSON数据后，错误消失

网关子设备动态注册问题

Q: 网关代理子设备动态注册失败，动态注册参数是子设备列表，云端返回注册成功，但是返回的列表为空？A:目前遇到两种常见的导致该问题的原因：

- 子设备未开启动态注册；
- 子设备已经被添加到其他网关设备下了；

Q: 网关代理子设备动态注册，动态注册列表包含3个子设备，云端返回注册成功信息，但是只返回了2个子设备信息？A:导致该问题可能的原因除了该设备未开启动态注册、添加到其他网关设备下，还有可能是子设备信息填写错误；

直连设备动态注册问题

Q: LinkKit SDK动态注册接口失败，导致初始化失败，无发建联？A:根据用户日志看到”SSLHandshakeException: com.android.org.bouncycastle.jce.exception.ExtCertPathValidatorException: Could not validate certificate: null”，请求还没有到达云端，mqtt证书检验出了问题，原因是用户的设备时间非正确时间。