ALIBABA CLOUD

阿里云

Android SDK

文档版本：20210425

阿里云

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
危险	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。	危险 重置操作将丢失用户配置数据。
警告	该类警示信息可能会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，恢复业务时间约十分钟。
注意	用于警示信息、补充说明等，是用户必须了解的内容。	注意 权重设置为0，该服务器不会再接受新请求。
说明	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。	单击**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。	在**结果确认**页面，单击**确定**。
Courier字体	命令或代码。	执行 `cd /d C:/window` 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。	`switch {active|stand}`

目录

1.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。LinkKit SDK Api Reference 参见： LinkKit API。

配置

1. 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/"
     }
   }
}
```

2. 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:1.7.2')
compile('com.aliyun.alink.linksdk:public-channel-core:0.7.7.1')
compile('com.aliyun.alink.linksdk:iot-device-manager:1.7.2.2')
```

混淆配置

混淆配置文件参考 Demo /app/proguard-rules.pro 文件

```
# 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.** {
  *;
}
-keepnames 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 class org.mozilla.**{*;}
# keep okhttp
-dontwarn okhttp3.**
-dontwarn okio.**
-dontwarn javax.annotation.**
-dontwarn org.mozilla.**
-keep class okio.**{*;}
-keep class okhttp3.**{*;}
-keep class org.apache.commons.codec.**{*;}
-keep class com.aliyun.alink.devicesdk.demo.FileProvider{*;}
-keep class android.support.**{*;}
-keep class android.os.**{*;}
```

2.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

SDK支持的设备认证方案如下：

设备密钥认证

设备密钥认证方式指设备在连接物联网平台时需要使用平台颁发的ProductKey、DeviceName、DeviceSecret进行设备认证，每个设备的设备密钥不能一样，否则会出现一个设备上线导致另外一个设备下线的情况。

注：有时候设备密钥也被称为三元组。

开发者需要实现初始化失败的设备连接平台重试逻辑，比如设备并未联网，初始化的时候连接物联网平台会失败，那么开发者需要再次对SDK进行初始化。 如果SDK初始化成功，也即设备成功连接物联网平台，之后如果因为网络原因导致连接断开，那么SDK会自动尝试连接物联网平台。

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
//如果使用deviceToken和clientID连接物联网平台，那么设备的deviceSecret需要设置为null
//一型一密免白名单动态注册之后建联需要设置deviceToken、clientId这两个值，这两个值由deviceDynamicRegister接口返回
MqttConfigure.deviceToken = deviceToken;
MqttConfigure.clientId = clientId;
/**
 * 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
  }
});
```

上面的代码会默认连接物联网平台上海站点的公共实例，如果您的实例位于其它站点或者是企业实例，那么需要对连接的域名进行更改：

- 公共实例

  公共实例的域名格式为：{ProductKey}.iot-as-mqtt.***RegionID***.aliyuncs.com"，其中的***RegionID***即代表相应的站点，其取值可以从“地域和可用区”中查看，需要注意的是物联网平台这个服务并不是所有的阿里云地域都进行了部署，因此这里填入的RegionID一定要和产品创建的地域相同，比如“华东2（上海）”的RegionID则为cn-shanghai，假设产品的ProductKey是abc，那么因此最终的域名为：abc.iot-as-mqtt.cn-shanghai.aliyuncs.com

- 企业实例

用户需要在企业实例中查看实例的接入URL，请参见文档“实例管理”中的“查看实例终端节点”章节了解如何查看实例详情，其中MQTT接入的域名如下图所示：

对于使用了企业实例的用户，我们提供demo用于演示如何连接物联网平台，您下载demo之后需要将app/src/main/res/raw/deviceinfo文件中的endpoint修改为自己实例的值。

动态注册

由于每个设备的DeviceName、DeviceSecret都是不同的，有的产品在产线上为每个设备烧写不同的设备密钥存在难度，为了简化厂商产线的设备密钥烧写难度，阿里云物联网平台提供了动态注册的方案。

动态注册指设备在出厂前烧写了ProductKey、ProductSecret，并使用设备已有的某个唯一标识用作DeviceName，通常为设备的MAC地址或者SN。由于ProductKey、ProductSecret对同一个产品的所有设备来说都是相同的，而设备即使不连接阿里云也需要烧写MAC地址或者SN，所以厂商可以将ProductKey、ProductSecret固化在设备的固件中，这就意味着设备厂商在产线上无需烧写设备密钥。

动态注册将会从云端获取设备的DeviceSecret，然后设备调用SDK初始化使用上面的设备密钥认证方式进行设备认证。设备使用设备密钥连接物联网平台成功之后，出于安全考虑平台不允许设备再次通过动态注册获取DeviceSecret，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**。

下面是动态注册的代码示例：

```
// ####### 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 ######
```

> **说明** 目前安卓SDK仅支持在上海站点支持上面所说的动态注册功能。

免白名单动态注册

区别于动态注册，这里的免白动态注册是不需要预先在后台录入deviceName的，可以直接使用该接口创建指定deviceName的设备。该动态注册接口返回的信息不包含deviceSecret，而是包含deviceToken和clientId，这个在建联的时候需要用到。需要注意的是，需要在云端控制台开启动态注册功能。

动态注册成功或失败之后，需要断开当前的动态注册长链接。如果动态注册成功，可以在断链之后执行建联的流程，具体可参考demo实现。下面是代码示例：

```
MqttInitParams initParams = new MqttInitParams(productKey, productSecret, deviceName, deviceSecret, MqttConfigure.MQTT_SECURE_MODE_TLS);
initParams.registerType = "regnwl"; // 一型一密免白
LinkKit.getInstance().deviceDynamicRegister(this, initParams, new IOnCallListener() {
  @Override
  public void onSuccess(com.aliyun.alink.linksdk.channel.core.base.ARequest request, com.aliyun.alink.linksdk.channel.core.base.AResponse response) {
    AppLog.i(TAG, "onSuccess() called with: request = [" + request + "], response = [" + response + "]");
    // response.data is byte array
    try {
      String responseData = new String((byte[]) response.data);
      JSONObject jsonObject = JSONObject.parseObject(responseData);
      String pk = jsonObject.getString("productKey");
      String dn = jsonObject.getString("deviceName");
      // 一型一密免白返回
      String ci = jsonObject.getString("clientId");
      String dt = jsonObject.getString("deviceToken");
      if ((!TextUtils.isEmpty(clientId) && !TextUtils.isEmpty(deviceToken)) || (!TextUtils.isEmpty(deviceSecret))) {
        // TODO by user 这里仅为测试代码，请将认证信息持久化到外部存储，确保app清除缓存或者卸载重装后仍能取到
        destroyRegisterConnect(true);
      } else {
        showToast("一型一密免白动态注册成功失败，返回信息无效 " + responseData);
        destroyRegisterConnect(false);
      }
    } catch (Exception e) {
      showToast("一型一密免白动态注册成功失败，返回数据信息无效");
      e.printStackTrace();
      destroyRegisterConnect(false);
    }
  }
  @Override
  public void onFailed(com.aliyun.alink.linksdk.channel.core.base.ARequest request, com.aliyun.alink.linksdk.channel.core.base.AError error) {
    AppLog.w(TAG, "onFailed() called with: request = [" + request + "], error = [" + error + "]");
    showToast("一型一密免白动态注册失败 " + error);
    destroyRegisterConnect(false);
  }
  @Override
  public boolean needUISafety() {
    return false;
  }
});
// 断开当前动态注册连接，并重新建联
private void destroyRegisterConnect(final boolean needConnect) {
    new Thread(new Runnable() {
      @Override
      public void run() {
        try {
          LinkKit.getInstance().stopDeviceDynamicRegister(10 * 1000, null, new IMqttActionListener() {
```

```
                @Override
                public void onSuccess(IMqttToken iMqttToken) {
                  AppLog.d(TAG, "onSuccess() called with: iMqttToken = [" + iMqttToken + "]");
                }
                @Override
                public void onFailure(IMqttToken iMqttToken, Throwable throwable) {
                  AppLog.w(TAG, "onFailure() called with: iMqttToken = [" + iMqttToken + "], throwable = [" + thro
wable + "]");
                }
              });
          } catch (Exception e) {
            e.printStackTrace();
          }
        }
      }).start();
  }
```

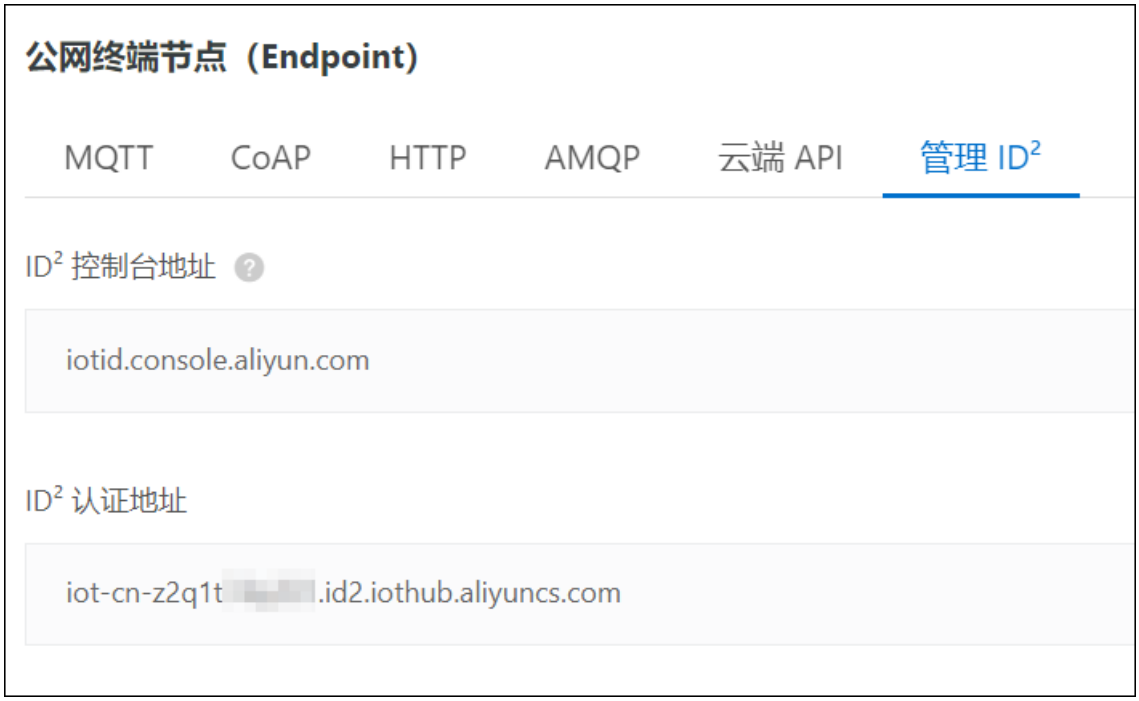
> **说明** 免白名单的动态注册目前仅在华东2(上海)站点进行了支持。

ID2设备认证

根据客户使用的物联网平台是公共实例或者企业实例，设备初始化时需要对域名进行修改。下面是连接上海站点的公共实例时SDK初始化的时候需要添加的设置设置：

```
/**
 * 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;
```

对于企业实例，域名前无需添加productKey，直接使用ID2的认证域名即可，ID2认证域名可以在企业实例中获取，如下图所示：



然后直接将认证域名设置到clientConfig.channelHost即可。

SDK 反初始化

如果需要注销初始化，调用如下接口。反初始化为同步接口，需要确保init的时候，上一次的deinit已经执行完成。如果正在deinit的时候执行init，会返回init失败。重复init也是会返回init失败的。

```
// 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();
```

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
    // 数据解析如下:
    //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);
// 取消下行监听，需要确保和注册的是同一个对象
// LinkKit.getInstance().unRegisterOnPushListener(notifyListener);
```

更多功能

目前支持以下功能的使能，默认都是关闭的，开发者可以根据使用场景做设置

```
LinkKitInitParams params = new LinkKitInitParams();
// ... 其它设置参数保持和认证参数一致，以下是新增的配置参数
/**
 * 设备是否支持被生活物联网平台APP发现
 * 需要确保开发的APP具备发现该类型设备的权限
*/
IoTDMConfig ioTDMConfig = new IoTDMConfig();
// 是否启用本地通信功能，默认不开启，
// 启用之后会初始化本地通信CoAP相关模块，设备将允许被生活物联网平台的应用发现、绑定、控制，依赖enableThi
ngModel开启
ioTDMConfig.enableLocalCommunication = false;
// 是否启用物模型功能，如果不开启，本地通信功能也不支持
// 默认不开启，开启之后init方法会等到物模型初始化（包含请求云端物模型）完成之后才返回onInitDone
ioTDMConfig.enableThingModel = false;
// 是否启用网关功能
// 默认不开启，开启之后，初始化的时候会初始化网关模块，获取云端网关子设备列表
ioTDMConfig.enableGateway = false;
// LinkKitInitParams 设置ioTDMConfig
params.ioTDMConfig = ioTDMConfig;
```

针对有休眠场景仍需要保持设备心跳建议参照 AlarmPingSender

```
MqttConfigure.pingSender = new UserDefinePingSender();
```

其他设置

日志开关

打开SDK内部日志输出开关：

```
PersistentNet.getInstance().openLog(true);
ALog.setLevel(ALog.LEVEL_DEBUG);
```

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;
```

- 设置clientId

clientId定义可以参考官方 MQTT-TCP连接通信 文档。

```
MqttConfigure.clientId = "abcdef003d27";
```

- 设置mqtt SDK是否自动重连

automaticReconnect设置为true，mqtt连接成功之后，如果因网络抖动导致mqtt断链，SDK会自动重新建联。如果设置为false，则mqtt断开之后，SDK不会自动重新建联，需要用户根据断链状态反馈，自己触发重连逻辑

```
MqttConfigure.automaticReconnect = true;
```

重连实现代码参见：

```
if(MqttNet.getInstance().getClient() instanceof MqttAsyncClient) {
   ((MqttAsyncClient) MqttNet.getInstance().getClient()).reconnect();
}
```

获取SDK的版本号

```
ALog.i(TAG, "sdk version = " + LinkKit.getInstance().getSDKVersion());
```

3.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。

> **说明** 物模型功能默认未使能，若需使用请参照初始化SDK中的“更多功能”章节将物模型使能。

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},"versio
n":"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 + "]");
    }
  });
}
```

4.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。如果不想使用物模型，可以通过这部分接口实现与云端的交互。

发布消息

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

```
// 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 json String
//示例 属性上报 {"id":"160865432","method":"thing.event.property.post","params":{"LightSwitch":1},"version":"1.0"}
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
  }
});
```

订阅消息

订阅关系保存在云端，设备如果订阅了某个topic，如果未主动执行unsbscribe，订阅关系会一直在。即如果设备重启之后不再订阅这个topic，订阅关系还是在的，云端收到对应topic发布，还是会转发给设备。订阅的topic下推可以在 设置的全局 IConnectNotifyListener 的 onNotify 处理。MqttSubscribeRequest 类路径参见 com.aliyun.alink.linksdk.cmp.connect.channel.MqttSubscribeRequest。

```
// 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;
subscribeRequest.qos = 0; // 支持0或者1
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

可以通过设置qos参数，指定订阅使用qos 0 或 1。

```
// 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
  }
});
```

5.RRPC能力

功能介绍

RRPC是指客户云端通过云端API发起一个 RRPC 调用，该调用将同步返回设备的响应。设备端会收到一个同步请求的 topic, 格式如 /ext/rrpc/{messageId}/{rrpc_topic} 或者 /sys/{pk}/{dn}/rrpc/request/${msgId}, 设备端接收到该消息后，进行处理，并将处理结果 publish 到 /ext/rrpc/{messageId}/{rrpc_topic} 或者 /ext/rrpc/${msgId}/${topic}。需要注意的是设备端首先需要订阅对应的下行topic，才能收到云端的RRPC同步调用。

消息通信API - RRPC

通过调用云端该 RRPC 接口，触发向设备发起 RRPC 同步请求。

设备管理API - InvokeThingsService

通过调用云端该 同步服务 接口或者在物联网平台定义设备的服务为同步服务并触发同步服务调用，即可向设备发起RRPC同步请求。

接口调用

在完成整个RRPC调用链路的过程中，设备端需要去订阅RRPC的topic，并在云端请求的时候，立即响应云端的请求。以下调用流程需要先确保SDK初始化成功，即MQTT建联成功。

RRPC订阅

${topic} 这部分替换成用户自定义的topic，如/a1pMraiYxxx/test/user/get，其中a1pMraiYxxx为具体产品productKey，test为deviceName。系统RRPC无需开发者订阅，在处理云端下行调用的时候系统 RRPC 调用的 topic 为/sys/${pk}/${dn}/rrpc/request/${requestId}。

```
MqttSubscribeRequest subscribeRequest = new MqttSubscribeRequest();
subscribeRequest.isSubscribe = true;
// ${topic} 替换成调用方的自定义 topic
subscribeRequest.topic = "/ext/rrpc/+/${topic}";
LinkKit.getInstance().subscribe(subscribeRequest, new IConnectSubscribeListener() {
  @Override
  public void onSuccess() {
    // 订阅成功
  }
  @Override
  public void onFailure(AError aError) {
    // 订阅失败
  }
);
```

RRPC监听和回复

云端RRPC请求可以在下行监听的地方接收到，具体接收到的topic由调用方指定。

```
// 这个是全局的下行监听设置接口
LinkKit.getInstance().registerOnPushListener(notifyListener);
 /**
   * 下行监听器，云端 MQTT 下行数据都会通过这里回调
   */
private static IConnectNotifyListener notifyListener = new IConnectNotifyListener() {
  /**
     * onNotify 会触发的前提是 shouldHandle 没有指定不处理这个topic
     * @param connectId 连接类型，这里判断是否长链 connectId == ConnectSDK.getInstance().getPersistentC
onnectId()
     * @param topic 下行的topic
     * @param aMessage 下行的数据内容
     */
  @Override
  public void onNotify(String connectId, String topic, AMessage aMessage) {
    String data = new String((byte[]) aMessage.data);
    // 服务端返回数据示例 data = {"method":"thing.service.test_service","id":"123374967","params":{"vv":60
},"version":"1.0.0"}
  }
  /**
     * @param connectId 连接类型，这里判断是否长链 connectId == ConnectSDK.getInstance().getPersistentC
onnectId()
     * @param topic 下行topic
     * @return 是否要处理这个topic，如果为true，则会回调到onNotify；如果为false，onNotify不会回调这个top
ic相关的数据。建议默认为true。
     */
  @Override
  public boolean shouldHandle(String connectId, String topic) {
    return true;
  }
  /**
     * @param connectId 连接类型，这里判断是否长链 connectId == ConnectSDK.getInstance().getPersistentC
onnectId()
     * @param connectState {@link ConnectState}
     *    CONNECTED, 连接成功
     *    DISCONNECTED, 已断链
     *    CONNECTING, 连接中
     *    CONNECTFAIL; 连接失败
     */
  @Override
  public void onConnectStateChange(String connectId, ConnectState connectState) {
    Log.d(TAG, "onConnectStateChange() called with: connectId = [" + connectId + "], connectState = [" + con
nectState + "]");
  }
};
```

调用示例

设备端

订阅自定义topic "/a1ExY4afKY1/testDevice/user/get"，并监听云端下行。RRPC 响应的数据内容根据调用者需要的数据来进行返回，以下示例默认返回空的data。

```
MqttSubscribeRequest subscribeRequest = new MqttSubscribeRequest();
subscribeRequest.isSubscribe = true;
subscribeRequest.topic = "/a1ExY4afKY1/testDevice/user/get";
LinkKit.getInstance().subscribe(subscribeRequest, new IConnectSubscribeListener() {
  @Override
  public void onSuccess() {
    // 订阅成功
  }
  @Override
  public void onFailure(AError aError) {
    // 订阅失败
  }
);
// 可使用全局的下行监听
LinkKit.getInstance().registerOnPushListener(notifyListener);
private IConnectNotifyListener notifyListener = new IConnectNotifyListener() {
  @Override
  public void onNotify(String connectId, String topic, AMessage aMessage) {
   if (CONNECT_ID.equals(connectId) && !TextUtils.isEmpty(topic) &&
            topic.startsWith("/ext/rrpc/")) {
      //示例 topic=/ext/rrpc/1138654706478941696//a1ExY4afKY1/testDevice/user/get
      //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},"ve
rsion":"1.0.0"}
      MqttPublishRequest request = new MqttPublishRequest();
      request.isRPC = false;
      request.topic = topic;
      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\":{} }";
      LinkKit.getInstance().publish(request, new IConnectSendListener() {
        @Override
        public void onResponse(ARequest aRequest, AResponse aResponse) {
   // 响应成功
        }
        @Override
        public void onFailure(ARequest aRequest, AError aError) {
   // 响应失败
        }
    });
    }
  }
  @Override
  public boolean shouldHandle(String connectId, String topic) {
    return true;
  }
  @Override
  public void onConnectStateChange(String connectId, ConnectState connectState) {
    Log.d(TAG, "onConnectStateChange() called with: connectId = [" + connectId + "], connectState = [" + con
nectState + "]");
  }
};
```

```
)
```

云端

云端触发RRPC调用代码如下，参考：调用自定义topic RRPC。

6.广播消息下发

功能介绍

云端可以对批量设备发送广播下行通知，设备端可以按照如下方式对广播进行处理。目前云端支持单设备广播和批量广播，这两种下发的topic格式是不一样的，设备端需要根据需要支持的形式做响应。

批量广播

批量广播是系统广播，无需端上进行订阅即可收到。topic格式：/sys/${pk}/${dn}/broadcast/request/+。

定向广播

定向广播是需要端上进行订阅才能收到广播下行的。topic格式：/broadcast/${pk}/${自定义action}。

广播接收

本功能需要在SDK正常初始化，且注册了下行监听器才可以收到通知。需要注意定向广播需要设备端进行订阅，topic是由发送端定义的，须符合 /broadcast/${pk}/${自定义action} 格式。其中自定义action可以是符合格式的任意值，设备端根据发送端的定义来进行订阅。

```
// 注册下行监听器
LinkKit.getInstance().registerOnPushListener(notifyListener);
/**
 * 下行监听器，云端 MQTT 下行数据都会通过这里回调
 */
private static IConnectNotifyListener notifyListener = new IConnectNotifyListener() {
  /**
     * onNotify 会触发的前提是 shouldHandle 没有指定不处理这个topic
     * @param connectId 连接类型，这里判断是否长链 connectId == ConnectSDK.getInstance().getPersistentC
onnectId()
     * @param topic 下行的topic
     * @param aMessage 下行的数据内容
     */
  @Override
  public void onNotify(String connectId, String topic, AMessage aMessage) {
    String data = new String((byte[]) aMessage.data);
    // 服务端返回数据示例 data = {"method":"thing.service.test_service","id":"123374967","params":{"vv":60
},"version":"1.0.0"}
    ALog.d(TAG, "onNotify() called with: connectId = [" + connectId + "], topic = [" + topic + "], aMessage = [" +
data + "]");
    if (ConnectSDK.getInstance().getPersistentConnectId().equals(connectId) && !TextUtils.isEmpty(topic) &
&
          topic.startsWith("/sys/" + DemoApplication.productKey + "/" + DemoApplication.deviceName + "/br
oadcast/request/")) {
       /**
          * topic 格式：/sys/${pk}/${dn}/broadcast/request/+
          * 无需订阅，云端免订阅，默认无需业务进行ack，但是也支持用户云端和设备端约定业务ack
          * 示例：/sys/a14NQ5RLiZA/android_lp_test1/broadcast/request/1229336863924294656
          * 注意：触发端数据需要进行Base64编码，否则会出现端上乱码，
          * 如云端： org.apache.commons.codec.binary.Base64.encodeBase64String("broadcastContent".get
Bytes())
          */
       //
```

```
        ToastUtils.showToast("收到云端批量广播下行：topic=" + topic + ",data=" + data);
        //TODO 根据批量广播做业务逻辑处理
    } else if (ConnectSDK.getInstance().getPersistentConnectId().equals(connectId) && !TextUtils.isEmpty(to
pic) &&
            topic.startsWith("/broadcast/" + DemoApplication.productKey )) {
        //
        /**
         * topic 需要用户自己订阅才能收到，topic 格式：/broadcast/${pk}/${自定义action}，需要和云端发送topic
一致
         * 示例：/broadcast/a14NQ5RLiZA/oldBroadcast
         * 注意：触发端数据需要进行Base64编码，否则会出现端上乱码，
         * 如云端： org.apache.commons.codec.binary.Base64.encodeBase64String("broadcastContent".get
Bytes())
         */
        ToastUtils.showToast("收到云端广播下行：topic=" + topic + ",data=" + data);
        //TODO 根据广播做业务逻辑处理
    } else {
        ToastUtils.showToast("收到云端下行：topic=" + topic + ",data=" + data);
        /**
         * TODO
         * 根据订阅的具体 topic 做业务处理
         */
    }
  }
```

7.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<RequestMod
el<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
  }
});
```

8.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<Re
questModel<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<Req
uestModel<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
  }
});
```

9.子设备接入

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
    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。

> **说明** 动态注册的免白名单方案不适用于子设备接入，因此子设备的DeviceName仍然需要厂家事先上传到物联网平台。

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 == nu
ll ? "null" : aResponse.data) + "]");
    try {
      // 子设备动态注册成功
      ResponseModel<List<DeviceInfo>> response = JSONObject.parseObject(aResponse.data.toString(), ne
w 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(aRespons
e.data.toString(),
                                                  new TypeReference<ResponseModel<Map<String, List<DeviceInf
o>>>>() {
                                                  }.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","deviceNam
e":"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(), ne
w 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 可为任意固定值，不可以是随机值
    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<I
nitResult>() {
  @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

```
// 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 = [" + iConnect
RrpcHandle + "]");
    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(((MqttRrpcReques
t) 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 + "]");
  }
});
```

10.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 iConnectRrpcHa
ndle) {
```

```
    // 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":{"timestamp":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
  }
});
```

11.文件上传

SDK的文件上传功能使用HTTP2流式传输协议，将文件上传至阿里云物联网平台服务器.

功能说明

- 支持多种上传模式，如以创建文件的方式上传，或以覆盖文件的方式上传
- 支持指定上传长度，并在下次上传时续传，用户可在上传时根据网络带宽配置上传分配大小(`part_len`)，以提高带宽利用效率

> **说明** 如果设备需要本文件上传功能，设备接入认证时不能选择动态注册免白名单方案。

本节以 `src/main/java/.../H2FileManagerActivity.java` 为例讲解如何使用文件上传功能

初始化

IStreamSender 是 HTTP2 流通道接口类。

```
LinkKitInitParams params = new LinkKitInitParams();
params.deviceInfo = deviceInfo; // 参考 Demo
// params.propertyValues = propertyValues; // 其他初始化参数
// params.connectConfig = userData;
// H2初始化参数
IoTH2Config ioTH2Config = new IoTH2Config();
ioTH2Config.clientId = "client-id";
// 注意替换{pk}为真实productKey
// ioTH2Config.endPoint = "https://{pk}.iot-as-http2.cn-shanghai.aliyuncs.com";
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();
```

注意：上面的代码中的endPoint需要根据region进行相应修改，不同region对应的Region ID请参见《地域和可用区》。

建立连接

CompletableListener 是通用的异步回调接口。

```
// 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 = LinkKit.getInstance().getH2StreamClient();
client.connect(new CompletableListener<Object>() {
  @Override
  public void complete(Object o) {
    // 建联成功
  }
  @Override
  public void completeExceptionally(Throwable throwable) {
    // 建联失败
  }
});
```

文件上传

非断点续传

文件上传类型请参考IANA Media Type文档。

```
final Http2Request request = new Http2Request();
//OSS上存储的文件名。文件名校验规则正则表达式为[a-zA-Z][a-zA-Z0-9_.]*。
request.getHeaders().add("x-file-name", "fileName");
//是否覆盖同名文件。0(不覆盖), 1(覆盖)。默认为0。如果文件已存在，并指定了不默认覆盖，则创建流失败
request.getHeaders().add("x-file-overwrite", "1");
//文件类型，不指定则由OSS自动指定。
// request.getHeaders().add("x-file-content-type", "jpg");
String serviceName = "/c/iot/sys/thing/file/upload";
// 注意替换成真实上传文件的路径
String filePath = "/sdcard/demo.jpg";
client.uploadFile(serviceNme, request, filePath, new CompletableDataListener<Http2Response>() {
  @Override
  public void complete(Http2Response http2Response) {
    // 上传成功
  }
  @Override
  public void completeExceptionally(Throwable throwable) {
    // 上传失败
  }
  @Override
  public void callBack(String fileUploadID) {
    // 本次上传的ID，如果需要断点续传，需要传入这个fileUploadID
    // 回回调多次，请保存最后一次的fileUploadID 用于后续断点续传
  }
});
```

断点续传

如果文件之前上传失败了，需要进行文件续传的话，需要传入上一次的fileUploadId。

```
final Http2Request request = new Http2Request();
// fileUploadId 为上一次未上传完成的ID
request.getHeaders().add("x-file-upload-id", fileUploadId);
String serviceName = "/c/iot/sys/thing/file/upload";
// 注意替换成真实上传文件的路径
String filePath = "/sdcard/demo.jpg";
client.uploadFile(serviceNme, request, filePath, new CompletableDataListener<Http2Response>() {
  @Override
  public void complete(Http2Response http2Response) {
    showToast(http2Response.toString() + "success");
  }
  @Override
  public void completeExceptionally(Throwable throwable) {
    showToast(throwable.toString() + "fail");
  }
  @Override
  public void callBack(String fileUploadID) {
    fileUploadId = fileUploadID;
  }
});
```

断开连接

文件上传完成之后断开连接。

```
// 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) {
  }
});
```

12.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 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，例如单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：

- 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。
- 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。

13.错误码

以下是mqtt 开源库 paho 的原生错误码，具体连接断开的时候，可以看到mqtt断开相关的错误日志和如下相关的错误码。

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
	123	ERROR_STATE_LINKKIT_DEINITING	deinit未执行完成，请等待deinit执行完后再初始化
-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

错误码	子错误码	描述	备注
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
1101312		LINKKIT_INIT_FAIL	初始化失败，一般是指mqtt建联成功了，但是在执行业务初始化的时候mqtt因网络问题中断导致整体初始化失败
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

错误码	子错误码	描述	备注
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
1101204		ERROR_DM_RESET_FAILED	设备重置失败，比如当前mqtt尚未建立连接或已掉线
1101205		ERROR_DM_INIT_THING_FAILED_IS_INITING	物模型初始化中
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
1102000		ERROR_CALL_INTERFACE_PARAMS_ERROR	*接口调用参数错误，具体调用的哪个接口 可查看msg或sub error code*
1102002		ERROR_PARAMS_ERROR	参数错误
200201		DM_INIT_PARAMS_INVALID	初始化参数无效
200301		DM_INIT_GET_TSL_RESPONSE_ERROR	初始化过程中获取物模型失败
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

错误码	子错误码	描述	备注
4101		ERROR_NETWORK_ERROR	网络错误
4102		ERROR_SERVER	业务网关错误
4103		ERROR_BUSINESS	业务错误

Mqtt错误码

1=无效协议版本
2=无效客户机标识
3=代理程序不可用
4=错误的用户名或密码
5=无权连接
6=意外错误
32000=等待来自服务器的响应时超时
32100=已连接客户机
32101=已断开客户机连接
32102=客户机正在断开连接
32103=无法连接至服务器
32104=客户机未连接
32105=指定的 SocketFactory 类型与代理程序 URI 不匹配
32106=SSL 配置错误
32107=不允许通过回调方法断开连接
32108=不可识别的包
32109=已断开连接
32110=已在进行连接
32111=客户机已关闭
32200=持久性已在使用中
32201=令牌已在使用中
32202=正在进行过多的发布

14.常见问题

RRPC发送应答出错，返回错误码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。

```
714 10591-10591/com.haier.intelrefriger D/ConnectSDK: send() common request
714 10591-10591/com.haier.intelrefriger D/PersistentConnect: send()
715 10591-10591/com.haier.intelrefriger D/MqttNet: getConnectState()
715 10591-10591/com.haier.intelrefriger D/MqttNet: getConnectState() paho state = true
728 10591-10591/com.haier.intelrefriger E/MqttSendExecutor: asyncSend(): convert payload Obj to byte array error
728 10591-10591/com.haier.intelrefriger W/System.err: java.io.NotSerializableException: org.json.JSONObject
728 10591-10591/com.haier.intelrefriger W/System.err:     at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:
729 10591-10591/com.haier.intelrefriger W/System.err:     at java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:3
729 10591-10591/com.haier.intelrefriger W/System.err:     at
k.linksdk.channel.core.persistent.mqtt.send.b.asyncSend(MqttSendExecutor.java:124)
729 10591-10591/com.haier.intelrefriger W/System.err:     at com.aliyun.alink.linksdk.channel.core.persistent.mqtt.b.asyncSen
729 10591-10591/com.haier.intelrefriger W/System.err:     at
k.linksdk.channel.core.persistent.PersistentNet.asyncSend(PersistentNet.java:75)
```

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。

mqtt 重连策略是

mqtt建联成功之后，mqtt如果中间断开了是会自动进行重试的。重试的时间间隔：1-2-4-8-16-32-64-128-128S...重试间隔是指上一次请求结束之后等待下一次重试建联的时间间隔

如何判断mqtt连接断开了

```
// 断连通知
2020-01-17 11:51:40.607 32730-32730/com.aliyun.alink.devicesdk.demo D/LK-DM-DeviceManager: onConnectStateChange() called with: s = [LINK_PERSISTENT], connectState = [DISCONNECTED]
// 断连原因，断连不一定能感知到所有原因，这里会保留开源库层的disconnect错误码
2020-01-17 11:56:30.807 2617-2671/com.aliyun.alink.devicesdk.demo W/LK-core-PersistentEventDispatch: mqtt disconnected, method = [null], content = [null], errorCode = [32109], message = [Connection lost，Connection lost (32109) - javax.net.ssl.SSLException: Read error: ssl=0x7dbe00f708: I/O error during system call, Software caused connection abort]
```

如何关闭 SDK 自动进行Mqtt重连功能？

```
MqttConfigure.automaticReconnect = false;
```

如何从日志判断SDK是否已经发起了自动重连？

看到以下日志表明 paho 自动发起了一次重连，发起的时机对上层不可见

```
2020-01-17 11:38:19.007 32730-32730/com.aliyun.alink.devicesdk.demo I/LK-core-IoTMqttAsyncClient: mqtt-paho connect start, userContext = [null], callback = [com.aliyun.alink.linksdk.channel.core.persistent.mqtt.MqttNet$3@18e0bf], [ clientId = lp_test&a14NQ5RLiZA|securemode=2,_v=1.7.0,lan=Android,os=9,signmethod=hmacsha1,ext=1,timestamp=1579232298867|]
```

如何从日志判断SDK重连结果？

```
// paho 日志
2020-01-07 14:57:06.097 9955-9955/com.aliyun.alink.devicesdk.demo D/LK-core-MqttDefaulCallback: connectComplete, reconnect=true, client=com.aliyun.alink.linksdk.channel.core.itls.IoTMqttAsyncClient@c212ad5, threadId=Thread[main,5,main]
// channel-core 日志
2020-01-07 14:57:05.095 9955-10005/com.aliyun.alink.devicesdk.demo I/LK-core-MqttDefaulCallback: mqtt connectComplete,reconnect = true ,ssl://a14NQ5RLiZA.iot-as-mqtt.cn-shanghai.aliyuncs.com:1883
```

心跳请求和心跳响应日志

```
// 心跳请求发送成功
2020-01-17 14:04:00.090 9442-9490/com.aliyun.alink.devicesdk.demo D/LK-core-MqttPaho: fine, c= org.eclipse.paho.client.mqttv3.internal.CommsSender , method = run , msg = 802, inserts = Ping,PINGREQ, throwable = null
// 收到云端心跳响应
2020-01-17 14:04:00.103 9442-9489/com.aliyun.alink.devicesdk.demo D/LK-core-MqttPaho: fine, c= org.eclipse.paho.client.mqttv3.internal.Token , method = markComplete , msg = 404, inserts = Ping,PINGRESP msgId 0, null, throwable = null
```

是否可以自定义心跳实现

可以通过实现MqttPingSender接口，可参考 官方实现 TimerPingSender，这里可以设置下一次发送心跳的时间点，以及停止发送心跳。发送心跳的时间点不要超过保活时间。

```
// 其中 PrivateMqttPingSender 实现了 MqttPingSender接口
MqttConfigure.pingSender = new PrivateMqttPingSender();
```

SDK 执行报错，找不到Log相关的类

原因：用户方法数超过65536，未支持多dex，导致类未打包进去