Android 设备端SDK接口文档

概述

工程配置

初始化

物模型

- 设备属性上报
- 设备属性获取
- 设备事件上报
- 设备服务获取
- 设备服务调用监听

COTA 远程配置

- 数据上行
- 数据下行

标签

- 数据上行-上报标签
- 数据下行-删除标签

网关子设备管理

- 子设备动态注册
- 获取子设备列表
- 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、删除
- 代理子设备物模型上下行
  - 子设备物模型初始化
  - 子设备物模型使用
  - 子设备物模型销毁
- 代理子设备基础上下行

设备影子

代码示例：
数据上行
数据下行
基础能力通道
上行接口请求
下行数据监听
混淆配置

概述

Android 设备端 SDK 为设备提供与云端的双向数据通道，从而实现对设备的远程管理。 SDK 提供了设备动态注册，初始化建联，数据上下行的接口。基础能力通道是基于 MQTT 实现的发布订阅模型实现，所有与云端上下行通信都是基于该基础能力做的业务封装，如物模型、网关子设备管理、标签、COTA、设备影子。

工程配置

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

```
allprojects {
    repositories {
      jcenter()
      // 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"
          }
     }
}
```

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

```
compile('com.aliyun.alink.linksdk:iot-linkkit:1.4.0')
```

初始化

设备端初始化包括设备信息的初始化，以及设备与云端的建联。设备初始化参数包括：

- 设备三元组信息
- 设备产品密钥（一型一密必要参数）
- 云端接口请求域名
- 设备属性的初始值

初始化成功会收到 onInitDone 回调，失败会收到 onError 回调。DeviceInfo 结构体内容可参见 DeviceInfo ApiReference。

```
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）
deviceInfo.deviceSecret = deviceSecret; // 三元组 设备密钥 （一机一密必填）
deviceInfo.productSecret = productSecret; // 产品密钥（一型一密必填）

// 请求域名
IoTApiClientConfig userData = new IoTApiClientConfig();
// 预留 目前不需要设置

// 设备属性初始值
Map<String, ValueWrapper> propertyValues = new HashMap<>();
// TODO 开发者需根据实际产品从设备获取属性值，如果不使用物模型无需设置属性初始值
//propertyValues.put("LightSwitch", new ValueWrapper.BooleanValueWrapper
(0));

LinkKitInitParams params = new LinkKitInitParams();
params.deviceInfo = deviceInfo;
params.propertyValues = propertyValues;
params.connectConfig = userData;

// 如果不设置建联之后会从云端更新最新的TSL
// 如果主动设置TSL，需确保TSL和线上完全一致，且功能定义与云端一致
// params.tsl = "{xxx}"; // 不建议用户设置，直接依赖SDK从云端更新最新的TSL即可

//  #######  一型一密动态注册接口开始 ######
// 如果是一型一密的设备，需要先调用动态注册接口；一机一密设备不需要执行此流程
HubApiRequest hubApiRequest = new HubApiRequest();
// 一型一密域名 默认"iot-auth.cn-shanghai.aliyuncs.com"
// hubApiRequest.domain = "xxx";  // 如果特殊需求，不要设置
hubApiRequest.path = "/auth/register/device";
// 设置 Mqtt 请求域名
IoTMqttClientConfig clientConfig = new IoTMqttClientConfig(productKey, devic
eName, deviceSecret);
```

```
// 慎用 设置 mqtt 请求域名，默认".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";
params.mqttClientConfig = clientConfig;

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  ######

// 设备初始化建联  如果是一型一密设备，需要在获取deviceSecret成功之后执行该操作。
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
    }
});
```

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。

```
// 注册下行监听
// 包括长连接的状态
// 云端下行的数据
LinkKit.getInstance().registerOnPushListener(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
    }

    @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 conne
ctState) {
        // 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
    }
});
```

打开SDK内部日志输出开关：

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

物模型

设备可以使用物模型实现设备状态变化的属性上报、设备出现异常或错误的事件上报、以及设备提供的服务的调用（如通过云端调用设备提供的 get 、set 服务，或去设备状态或者下发控制设备的指令）。

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

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 IPu
blishResourceListener() {
    @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, ne
```

```
w 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

云端在添加服务的时候，可以设置当前服务需要使用异步方式调用还是同步方式调用。Service 里面的 callType 字段表示当前服务是使用的同步服务调用（callType=”sync”）还是异步服务调用（callType=”async”）。设备属性设备和属性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的指令之后，调用设备执行真实的操作，在操作结束之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
    }

    @Override
    public void onSuccess(Object tag, 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 tag, 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。用户可以在同步服务的场景注册一个下行数据监听，当云端触发同步服务调用的时候，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。RRPC 回复的 topic 将请求的 request 替换成 response， msgId 保持不变。

```
LinkKit.getInstance().registerOnPushListener(new IConnectNotifyListener() {
    @Override
    public void onNotify(String connectId, String topic, AMessage aMessage)
{
        if ("LINK_PERSISTENT".equals(connectId) && !TextUtils.isEmpty(topic)
 &&
                topic.startsWith("/sys/" + DemoApplication.productKey + "/"
 + DemoApplication.deviceName + "/rrpc/request")){
            // showToast("收到云端同步下行");
            MqttPublishRequest request = new MqttPublishRequest();
            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\":{} }";
            LinkKit.getInstance().publish(request, new IConnectSendListener
() {
                @Override
```

```
                public void onResponse(ARequest aRequest, AResponse aRespon
se) {
                    // 上报结果
                }

                @Override
                public void onFailure(ARequest aRequest, AError aError) {
                    // 上报失败 aError: 错误原因
                }
            });
        }
    }

    @Override
    public boolean shouldHandle(String connectId, String topic) {
        return true;
    }

    @Override
    public void onConnectStateChange(String connectId, ConnectState connect
State) {

    }
})
```

COTA 远程配置

该功能需要在云端开启产品的远程配置功能才可以使用。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。

> 远程配置相关接口参见 设备 IDeviceCOTA。

数据上行

- 获取设备远程配置

```
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 TypeRe
ference<RequestModel<Map>>() {
    }.getType());
LinkKit.getInstance().getDeviceCOTA().COTAGet(requestModel, new IConnectSen
```

```
dListener() {
    @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
    }
});
```

数据下行

- 订阅设备远程配置

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

标签

允许设备端上报标签到云端，或者删除设备标签。

> 设备标签相关接口参见 设备 IDeviceLabel。

数据上行-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\": \"Temperatu
re\"," +
            "      \"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 IConne
ctSendListener() {
    @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
    }
});
```

数据下行-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\": \"Temperatu
re\"" +
```

```
            "    }" + "  ]," + "  \"method\": \"thing.deviceinfo.delete\""
 + "}";
RequestModel<List<Map>> requestModel = JSONObject.parseObject(deleteLabel, n
ew TypeReference<RequestModel<List<Map>>>() {
}.getType());
LinkKit.getInstance().getDeviceLabel().labelDelete(requestModel, new IConne
ctSendListener() {
    @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
    }
});
```

网关子设备管理

如果当前设备是一个网关，并需要帮助网关下的子设备接入云端，就需要接入网关子设备管理的能力。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、添加网关子设备、删除网关子设备、网关子设备上线、网关子设备下线、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。

> 网关子设备管理相关接口参见 设备 IGateway。
> 注意：设备每次初始建联的时候都需要调用添加、登录的接口。

子设备动态注册

需要在云端开启允许动态注册功能，可以同时注册多个。子设备在添加到网关之前先要动态注册设备。

```
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.parseObje
```

```
ct(aResponse.data.toString(), new TypeReference<ResponseModel<List<DeviceIn
fo>>>() {
            }.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
    }
});
```

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 IConnectSendList
ener() {
    @Override
    public void onResponse(ARequest aRequest, AResponse aResponse) {
        // 获取子设备列表结果
        try {
            ResponseModel<List<DeviceInfo>> response = JSONObject.parseObje
ct(aResponse.data.toString(), new TypeReference<ResponseModel<List<DeviceIn
fo>>>() {
            }.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 ISubD
eviceConnectListener() {
    @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.currentTim
eMillis()));
        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 iSubDev
iceChannel, 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, String s1) {
        // 收到子设备下行数据 topic=" + s + ", data=" + s1
        // 如禁用 删除 已经 设置、服务调用等
```

```
    }
});
```

删除子设备

这里的删除子设备是指在子设备的产品里面删除该子设备，不是指删除和网关的拓扑关系。

```
final DeviceInfo deviceInfo = new DeviceInfo();
deviceInfo.productKey = productKey; // 三元组 产品型号（必填）
deviceInfo.deviceName = deviceName; // 三元组 设备标识 （必填）
LinkKit.getInstance().getGateway().gatewayDeleteSubDevice(deviceinfo, new ISub
DeviceRemoveListener() {
    @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

代理子设备上线，子设备上线之后可以执行子设备的订阅、发布等操作。

```
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 ISu
bDeviceActionListener() {
    @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 + "]");
        showToast("代理子设备上线失败");
```

```
            log(TAG ,"代理子设备上线失败" + getPkDn(info));
        }
    });
```

子设备下线

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 IS
ubDeviceActionListener() {
    @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、删除

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(device
info, 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 iConnectRr
pcHandle) {
        // 子设备禁用通知
        iConnectRrpcHandle.onRrpcResponse(null, null);
    }

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

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

代理子设备物模型上下行

子设备物模型初始化

```
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, subDe
vInitState, new IDMCallback<InitResult>() {
    @Override
    public void onSuccess(InitResult initResult) {
        // 物模型初始化成功之后 可以做服务注册  上报等操作
    }

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

```
    }
});
```

子设备物模型使用

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

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

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

子设备物模型销毁

反初始化物模型，后续如果需要重新使用需要重新走登录、物模型初始化流程。

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

代理子设备基础上下行

使用网关的通道执行子设备的数据上下行。

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

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

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

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

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

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

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

设备影子

如果当前产品不具备物模型的能力，可以通过设备影子实现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。

> 设备影子相关接口参见 设备 IDeviceShadow。

代码示例：

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

```
// 监听云端设备影子更新
LinkKit.getInstance().getDeviceShadow().setShadowChangeListener(new IShadow
RRPC() {
    @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, IConnect
RrpcHandle 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);

                ShadowResponse<String> shadowResponse = JSONObject.parseObj
ect(dataStr, new TypeReference<ShadowResponse<String>>() {
                }.getType());
                if (shadowResponse != null && shadowResponse.version != nu
ll && 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.value
Of(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
        }
});
```

基础能力通道

LinkKit SDK 提供了与云端长链接的基础能力接口，用户可以直接使用这些接口完成自定义 Topic 相关的功能。提供的基础能力包括：发布、订阅、取消订阅、RRPC、订阅下行。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。如果有自定 Topic 需求，或者不想使用封装的其他能力接口，都可以通过基础上下行通道能力实现。

上行接口请求

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

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

```
// 发布
MqttPublishRequest request = new MqttPublishRequest();
request.isRPC = false;
request.topic = topic;
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();
subscribeRequest.topic = subTopic;
subscribeRequest.isSubscribe = true;
LinkKit.getInstance().subscribe(subscribeRequest, new IConnectSubscribeListen
er() {
    @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();
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：

```
// 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 connect
State) {
        // 连接状态变化
    }
};
// 注册
LinkKit.getInstance().registerOnPushListener(onPushListener);
// 取消注册
LinkKit.getInstance().unRegisterOnPushListener(onPushListener);
```

RRPC 调用示例：

```
final MqttRrpcRegisterRequest registerRequest = new MqttRrpcRegisterRequest()
;
registerRequest.topic = rrpcTopic;
registerRequest.replyTopic = rrpcReplyTopic;
registerRequest.payloadObj = payload;
// 先订阅回复的 replyTopic
// 云端发布消息到 replyTopic
// 收到下行数据 回复云端 具体可参考 Demo 同步服务调用
LinkKit.getInstance().subscribeRRPC(registerRequest, new IConnectRrpcListene
r() {
    @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 iConnectRr
pcHandle) {
        // 收到云端下行
        AResponse response = new AResponse();
        response.data = responseData;
        iConnectRrpcHandle.onRrpcResponse(registerRequest.topic, response);
    }

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

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

混淆配置

```
-dontwarn com.alibaba.fastjson.**
-dontwarn com.aliyun.alink.linksdk.**
-dontwarn com.facebook.**
-dontwarn okhttp3.internal.**

-keep class com.alibaba.** {*;}
-keep class com.aliyun.alink.linkkit.api.**{*;}
-keep class com.aliyun.alink.dm.api.**{*;}
-keep class com.aliyun.alink.dm.model.**{*;}
-keep class com.aliyun.alink.dm.shadow.ShadowResponse{*;}
## 设备sdk keep
```

```
-keep class com.aliyun.alink.linksdk.channel.**{*;}
-keep class com.aliyun.alink.linksdk.tmp.**{*;}
-keep class com.aliyun.alink.linksdk.cmp.**{*;}
-keep class com.aliyun.alink.linksdk.alcs.**{*;}
-keep public class com.aliyun.alink.linksdk.alcs.coap.**{*;}

-keep class com.http.helper.**{*;}
-keep class com.aliyun.alink.apiclient.**{*;}
```