概述

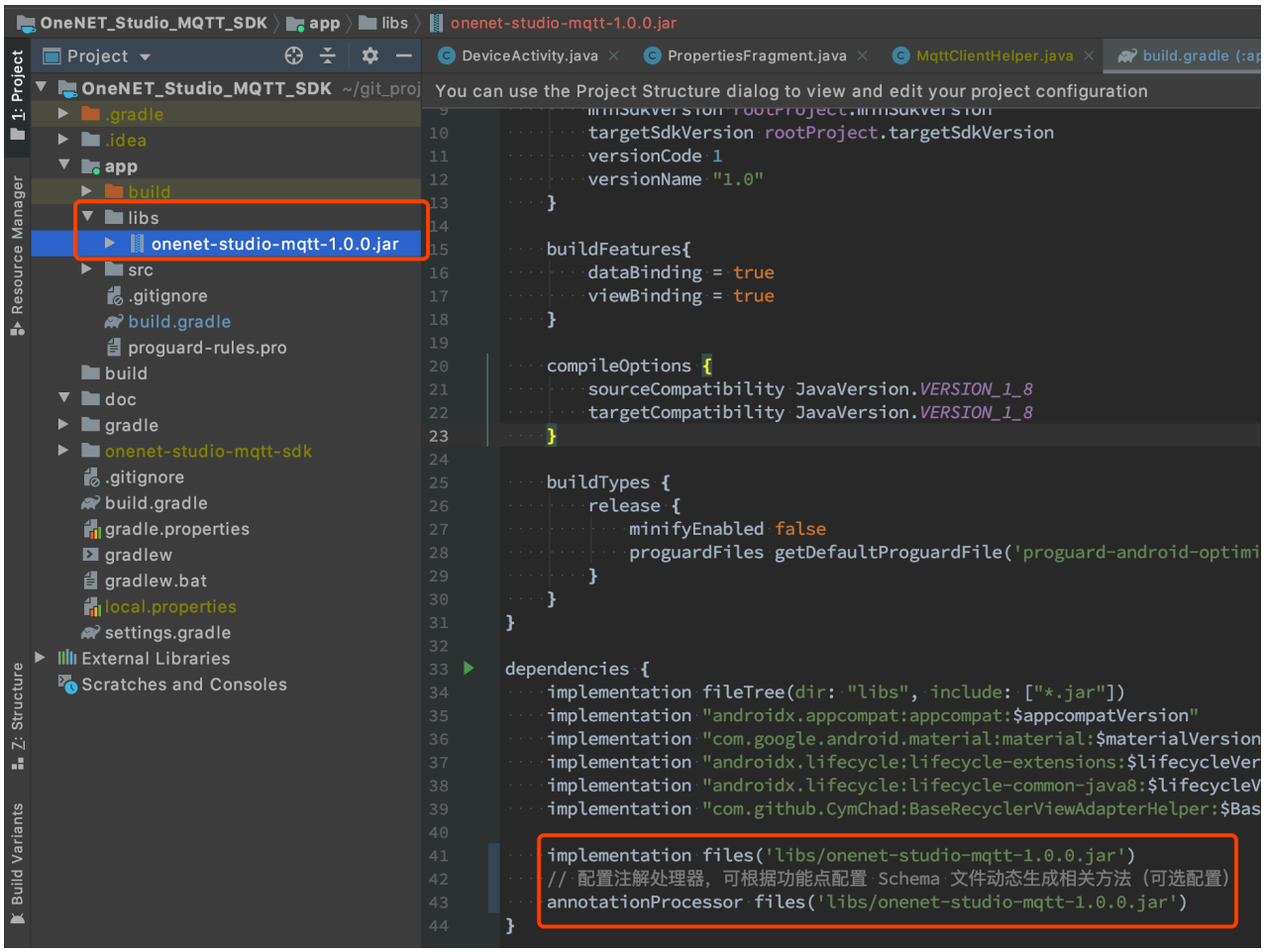
本文档是 OneNET Studio MQTT Android SDK 的开发指南，适用于基于 Android 操作系统的终端设备快速接入 OneNET Studio。我们默认读者是有经验的 Android 开发者，并且熟悉 Android Studio 的使用方法。

关于 OneNET Studio 请访问 OneNET Studio 文档页面 了解详情。

主要功能

- 设备属性功能点的上报和接收，设备事件上报
- 设备服务调用
- 子设备相关操作
- 支持自动重连
- 支持自定义超时时间、心跳间隔
- 可配置在编译期生成功能点相关代码

Android Studio 工程配置



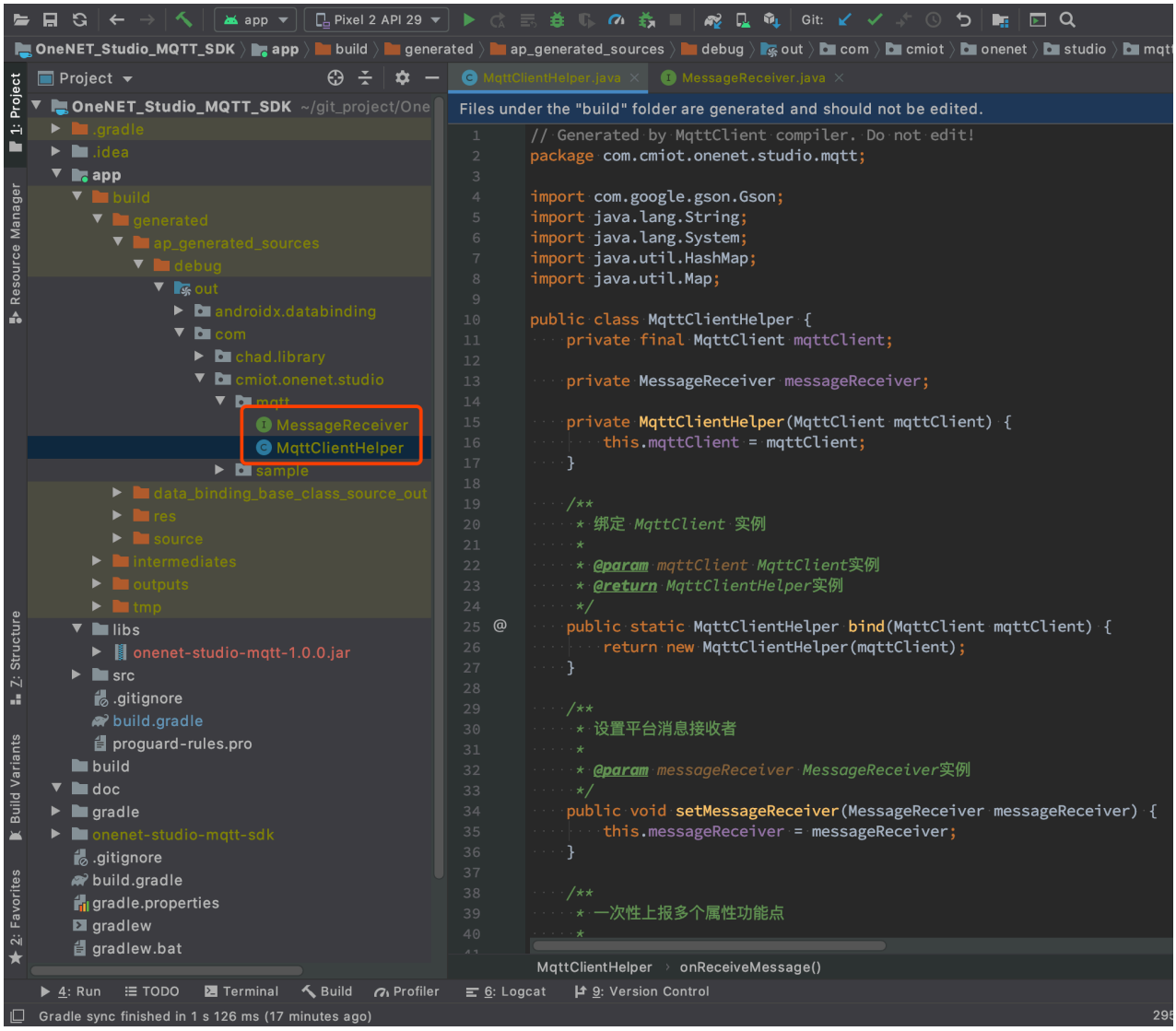
第一步，将 SDK jar 包拷贝至 app/libs 目录下。

第二步，在 app/build.gradle 文件中的 dependencies 代码块中加入

```
implementation files('libs/onenet-studio-mqtt-x.x.x.jar')
annotationProcessor files('libs/onenet-studio-mqtt-x.x.x.jar')
```

`x.x.x` 为版本号，其中配置`annotationProcessor`的作用是根据功能点配置文件生成`MqttClientHelper.java`和`MessageReceiver.java`两个类文件，包含了各个功能点的上报函数，和下发数据的解析。

这一项以及第三步之后的配置都是可选配置，如果不需要，可以不配置。



第三步，导出产品物模型配置文件。

第四步，将物模型配置文件拷贝到工程目录下，建议放在 assets 目录下，这样 Android 项目代码也能读取。

第五步，在任意类文件的类名上添加注解 。如果有子设备，可连同子设备一起配置，注意将 subDevice 设置为 true 即可。

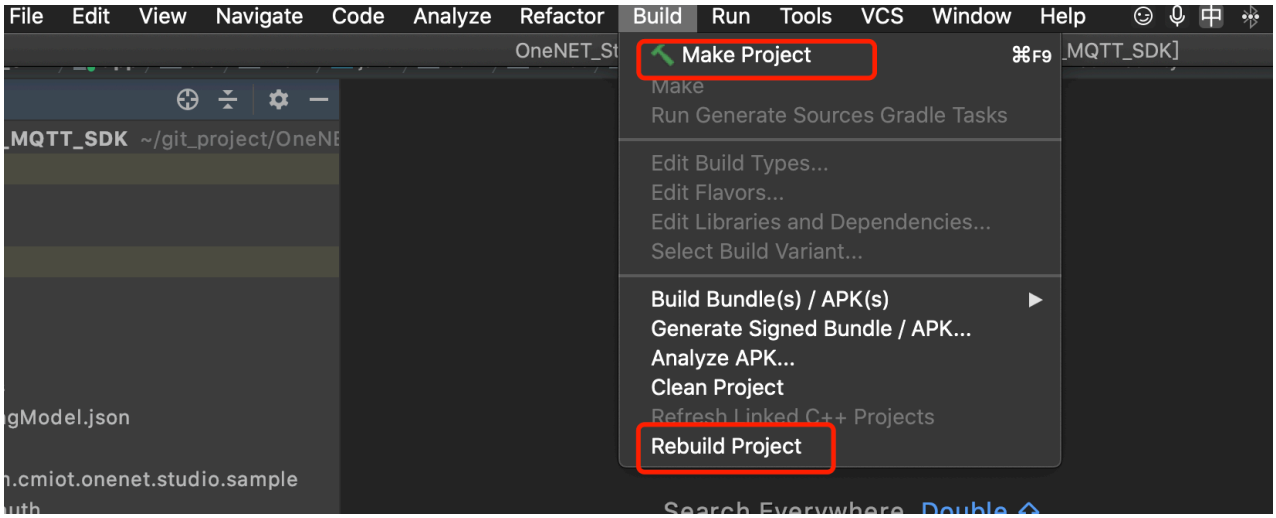
```
@Schema("schema文件的路径")
@ThingModels({
        @ThingModel(
                path = "GatewayDemo/src/main/assets/model-I2ShgOIGdw.json",
                productId = "I2ShgOIGdw",
                productName = "DeviceTest",
                productKey = "raM9//M9ORHVC5VRR+91QVsfOXG1VZnnrZNtC+dpS4k="
        ),
        @ThingModel(
                path = "GatewayDemo/src/main/assets/model-REy3k0pk6N.json",
                productId = "REy3k0pk6N",
                productName = "SubDeviceTest",
                productKey = "P41PiWW5wxnTOzcXo3Pp83OZ+y9x968RJZWnO7sHFwU=",
                subDevice = true // 子设备
        )
})
public class SomeClass {
}
```

```
@ThingModels({
        @ThingModel(
                path = "GatewayDemo/src/main/assets/model-I2ShgOIGdw.json",
                productId = "I2ShgOIGdw",
                productName = "DeviceTest",
                productKey = "raM9//M9ORHVC5VRR+91QVsfOXG1VZnnrZNtC+dpS4k="
        ),
        @ThingModel(
                path = "GatewayDemo/src/main/assets/model-REy3k0pk6N.json",
                productId = "REy3k0pk6N",
                productName = "SubDeviceTest",
                productKey = "P41PiWW5wxnTOzcXo3Pp83OZ+y9x968RJZWnO7sHFwU=",
                subDevice = true // 子设备
        )
})
public class DeviceActivity extends BaseActivity<ActivityDeviceBinding, DeviceViewModel> {

    private List<BaseFragment> mFragments = new ArrayList<>(2);
    private LogFragment mLogFragment;

    @Override
    protected int getLayoutId() {
        return R.layout.activity_device;
    }
```

最后，点击 Android Studio 的 Build 菜单下的 Make Project 或 Rebuild Project，就能看到`MqttClientHelper.java`和`MessageReceiver.java`两个文件，并在代码中引入使用了。



连接平台

SDK 连接平台需要三个参数，分别是“产品 id”、“产品 key”（或“设备秘钥”）、“设备名称”。

创建产品

在“设备接入管理” - “产品管理”中，点击“添加产品”，填写产品信息后确定。

点击产品列表右侧的详情，进入产品详情页，可以查看产品 id 和产品 key。

创建设备

在“设备接入管理” - “设备管理”中，点击“添加设备”，填写设备信息后确定。

点击设备列表右侧的详情，进入设备详情页，可以查看设备名称和设备秘钥。

连接平台

现在连接平台所需的信息都有了，下面开始调用 SDK。

首先，构建 `MqttClient` 实例

```
MqttClient mqttClient = new MqttClient.Builder()
        .productId("产品id")
        .accessKey("产品key或设备秘钥")
        .deviceName("设备名称")
        .ssl(true)                          // 是否使用 ssl 加密通信，默认 true
        .expireTime(100 * 24 * 60 * 60) // 鉴权 token 超时时间，单位秒，默认 100 天
        .autoReconnect(true)                // 是否自动重连
        .connectionTimeout(10)              // 连接超时时间，单位秒，默认15秒
        .keepAliveInterval(60)              // keep-alive 发送间隔，单位秒，默认120秒
        .showDebugLog(true)                 // 是否显示调试日志
        .build();
```

其中，productKey 和 deviceKey 可以二选一，也可以同时设置，优先使用 deviceKey。

然后就可以调用 `connect()` 方法进行连接了。

```
try {
    mqttClient.connect(); // 连接服务器
} catch (Exception e) {
    e.printStackTrace();
}
```

私有化部署平台参数设置

```
MqttClient mqttClient = new MqttClient.Builder()
        // ...
        .host("xxx.xxx.xxx.xxx") // 私有化部署平台 MQTT 服务器地址
        .port(xxxx)              // 私有化部署平台 MQTT 服务器端口
        .cert(certFile)          // 证书文件，如果选择加密通信则要设置此项
        .ssl(true);              // 是否要加密通信，默认是 true
```

状态监听

设置回调

```
private MqttClientCallback callback = new MqttClientCallback() {
    @Override
    public void onConnectFailed(String serverUrl, Throwable throwable) {}

    @Override
    public void onConnectionLost(Throwable throwable) {}

    @Override
    public void onReceiveMessage(final String topic, final byte[] payload) {}
};

mqttClient.addCallback(callback);
```

需要释放资源的时候记得移除回调

```
mqttClient.removeCallback(callback);
```

`MqttClientCallback` 是一个抽象类，一共有8个回调方法：

```
public void onConnected(String serverUrl) {}
public abstract void onConnectFailed(String serverUrl, Throwable exception);
public abstract void onConnectionLost(Throwable exception);
public abstract void onReceiveMessage(String topic, byte[] payload);
public void onReconnected(String serverUrl) {}
public void onMessageDelivered(byte[] payload) {}
public void onSubscribeSuccess(String topic) {}
public void onSubscribeFailed(String topic, Throwable exception){}
```

分别用于监听

- 连接成功
- 连接失败
- 连接断开
- 收到消息
- 重连成功
- 消息送达
- 订阅主题成功
- 订阅主题失败

其中 `onConnectFailed`、`onConnectionLost` 和 `onReceiveMessage` 三个方法为抽象方法，其余方法可根据个人需求选择覆写。

功能点代码生成

SDK提供了注解处理器，可生成功能点相关操作代码，配置方式如前文所述。这是一个可选功能，主要优点是屏蔽了物模型 OneJSON 协议的具体细节，使得开发更加简单和直观。

使用该功能，编译代码后，会在 app 的 build 目录中生成 `MqttClientHelper.java` 和 `MessageReceiver.java` 两个文件，如果配置了子设备，则会生成多个，并在文件名末尾加上产品 id 表示区分。

使用方法

```
// 1 绑定 MqttClient，获得 MqttClientHelper 实例
MqttClientHelper mqttClientHelper = MqttClientHelper.bind(mqttClient);

// 2 添加 MessageReceiver
mqttClientHelper.addMessageReceiver(messageReceiver);

MqttClientCallback callback = new MqttClientCallback() {
    @Override
    public void onConnectFailed(String serverUrl, Throwable throwable) {}

    @Override
    public void onConnectionLost(Throwable throwable) {}

    @Override
    public void onReceiveMessage(final String topic, final byte[] payload) {
        // 3 在 MqttClientCallback 的 onReceiveMessage 回调中调用
MqttClientHelper 的 onReceiveMessage 方法
        mqttClientHelper.onReceiveMessage(topic, payload);
    }
};

/**
 * MessageReceiver 中的方法是根据物模型自动生成的，所以物模型不同，方法也不同。
 */
MessageReceiver messageReceiver = new MessageReceiver() {
    @Override
    onReceiveProperty1(String msgId, Integer value) {
    }

    @Override
    onReceiveProperty2(String msgId, String value) {
    }

    // ...
};

// 上报 property1
mqttClientHelper.uploadProperty1(msgId, value);

// 4 移除 MessageReceiver
```

```
mqttClientHelper.removeMessageReceiver(messageReceiver);
```

`MqttClientHelper` 的 `onReceiveMessage` 会将服务端的原始数据进行解析，分发到对应的功能点消息回调中执行。

数据上报和下发

设备和平台之间的数据通信采用一种定制化的 JSON 格式，称为 OneJSON。基本的数据格式为：

```
{
  "id": "消息id",
  "version": "1.0",
  "params": {}
}
```

其中 `id` 表示消息 id，是一个长度不超过 13 个字符的自定义字符串，一个会话周期内请求和响应的消息 id 是一致的，`version` 默认为 1.0，`params` 是用户定义的参数，其中的字段和类型来自于用户在平台定义的产品功能点物模型。

设备属性上报

首先在平台“设备接入与管理” - “产品管理”中，查看产品详情，点击“添加系统功能点”、“添加标准功能点”或“添加自定义功能点”，填写功能点信息，然后点击“保存”，随后即可导出物模型配置文件。

SDK 提供了生成功能点相关代码的功能，配置方式如前文所述。如果使用了这项功能，则不用再去关注物模型配置文件的细节，直接调用 `MqttClientHelper` 类的相关函数即可。

如果不使用自动生成的代码，则可以通过 `MqttClient` 类提供的 `postProperties` 方法进行数据的上报。

`postProperties` 方法声明如下

```
public <T> void postProperties(String msgId, T params)
```

`msgId` 代表消息 id，`params` 代表设备的功能点属性数据。

假如用户定义的物模型文件格式如下：

```
{
  "properties": [
    {
      "accessMode": "r",
      "dataType": {
        "specs": {
          "length": "5"
        },
        "type": "string"
      },
      "desc": "电源状态",
      "functionMode": "property",
      "functionType": "u",
      "identifier": "power",
      "name": "电源"
    }
  ]
}
```

那么，对应的上报消息格式为：

```
{
  "id": "xxxxx",
  "version": "1.0",
  "params": {
    "power": {
      "value": "on",
      "time": 1524448722123
    }
  }
}
```

用代码实现如下：

```
long time = System.currentTimeMillis();
Map params = new HashMap();
Map power = new HashMap();
power.put("value", "on");
power.put("time", time);
params.put("power", power);
mqttClient.postProperties(time + "", params);
```

如果使用 `MqttClientHelper`，则简化如下：

```
MqttClientHelper helper = MqttClientHelper.bind(mqttClient);
helper.uploadPower(System.currentTimeMillis() + "", "on");
```

设备事件上报

设备事件上报总体上和属性上报类似，但物模型格式稍有不同，详细请查看 OneJSON数据协议。

`MqttClient` 同样也提供了一个 `postEvents` 方法，声明如下：

```
public <T> void postEvents(String msgId, T params)
```

平台响应

设备每次上报数据，平台都会返回一个响应，格式为：

```
{
  "id": "xxxx",
  "code":200,
  "msg":"xxxxxx"
}
```

上一节中提到，实例化 `MqttClient` 时可以传入一个 `MqttClientCallback` 实例，其中有一个抽象类

```
public abstract void onReceiveMessage(String topic, byte[] payload);
```

设备收到的所有消息都是通过这个方法通知设备，平台下发了数据，可以通过 topic 来区分不同的消息。

如果使用 `MqttClientHelper`，则可以使用其提供的一个成员方法

```
public void onReceiveMessage(String topic, byte[] payload) {}
```

还是以上面提到的物模型为例，使用方法如下：

```
private MqttClientCallback callback = new MqttClientCallback() {

    @Override
    public void onConnected(String serverUrl) {

    }

    @Override
    public void onConnectFailed(String serverUrl, Throwable throwable) {

    }
```

```
    @Override
    public void onConnectionLost(Throwable throwable) {

    }

    @Override
    public void onReceiveMessage(String topic, byte[] payload) {
        mqttClientHelper.onReceiveMessage(topic, payload);
    }
};
```

然后通过 `MqttClientHelper` 提供的 `setMessageReceiver` 方法，传入一个 `MessageReceiver` 实例

```
mqttClientHelper.setMessageReceiver(new MessageReceiver() {
    // 省略无关代码

    @Override
    public void onUploadPowerReply(final String msgId, final int code,
        final String msg) {
        // 平台收到上报Power功能点数据的响应
    }
});
```

`MqttClientHelper` 在内部解析了相应的功能点响应数据，这样就不用自己去处理这些细节问题了。

平台设备属性下发

平台下发的数据都要经过 `MqttClientCallback` 的 `onReceiveMessage` 方法，如果使用了 `MqttClientHelper`，同时也设置了 `MessageReceiver`，那么 `MqttClientHelper` 内部会将下发的消息数据解析，通过生成的各个功能点的 `onReceive` 方法返回给设备。

```
mqttClientHelper.setMessageReceiver(new MessageReceiver() {
    // 省略无关代码

    @Override
    public void onReceivePower(final String msgId, final String power) {
        // 设备收到Power属性功能点下发数据
    }
});
```

> 设备收到平台下发的数据后一般需要立刻向平台发送响应，否则平台会认为设备没有正常收到数据，判定为超时。
>
> 如果使用了 `MqttClientHelper` 的 `onReceiveMessage` 方法，那么会自动发送响应，无需开发者自己处理。

设备服务

以下面这个服务的物模型为例

```
{
  "services": [
    {
      "name": "停止录像观看",
      "identifier": "StopVod",
      "functionType": "st",
      "functionMode": "service",
      "desc": "",
      "callType": "a",
      "input": [
        {
          "identifier": "SessionID",
          "name": "会话ID",
          "dataType": {
            "type": "string",
            "specs": {
              "length": "255"
            }
          }
        }
      ],
      "output": []
    }
  ]
}
```

编译后会在 `MessageReceiver` 中生成 `onStopvodInvoke` 方法，如下

```
mqttClientHelper.setMessageReceiver(new MessageReceiver() {
    // 省略无关代码

    @Override
    public void onStopvodInvoke(final String msgId, final String sessionid) {
        // 设备收到启动服务请求
        // 回复服务请求
        mqttClient.replyInvoke("SessionID", msgId, 200, "success", null);
        // TODO 启动设备服务
        // ...
    }
});
```

设备期望属性值

平台允许配置设备属性的期望值，如果设备是在线的，那么设备端会立即收到数据；如果设备是离线的，那么在设备上线之后，需要调用 `MqttClient` 的 `getDesiredProperties` 方法，并传入要获取期望值的功能点名称。

如果使用了 `MqttClientHelper`，那么可以用其提供的 `getAllDesiredProperties` 方法，获取所有属性功能点的期望值，并通过 `MessageReceiver` 的 `onGetPropertyDesiredReply` 方法返回给设备。

```
mqttClientHelper.setMessageReceiver(new MessageReceiver() {
    // 省略无关代码

    @Override
    public void onGetPropertyDesiredReply(final String msgId, final int code,
        final Map msg) {
        // 设备收到属性期望值数据
    }
});
```

子设备相关

以下方法均位于 `MqttClient.java` 中

```
// 绑定子设备
public void bindSubDevice(String msgId, String subProductId, String
subDeviceName, String subAccessKey);

// 解绑子设备
public void unbindSubDevice(String msgId, String subProductId, String
subDeviceName, String subAccessKey);

// 获取网关拓扑关系
public void getSubDevices(String msgId);

// 回复网关拓扑关系变化通知
public void replySubDeviceChange(String msgId, int code, String msg);

// 子设备上线
public void subDeviceLogin(String msgId, String subProductId, String
subDeviceName);

// 子设备下线
public void subDeviceLogout(String msgId, String subProductId, String
subDeviceName);

// 回复子设备属性获取请求
```

```
public void replySubDevicePropertiesGet(String msgId, int code, String msg, Map
data);

// 回复子设备属性设置请求
public void replySubDevicePropertiesSet(String msgId, int code, String msg);

// 回复子设备服务调用
public void replySubDeviceServiceInvoke(String msgId, int code, String msg, Map
data);
```