

# 阿里云 物联网平台

## 设备端开发指南

文档版本：20181220

# 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
	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。	 禁止： 重置操作将丢失用户配置数据。
	该类警示信息可能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，恢复业务所需时间约10分钟。
	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 <b>Ctrl + A</b> 选中全部文件。
>	多级菜单递进。	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。	单击 确定。
<code>courier</code> 字体	命令。	执行 <code>cd /d C:/windows</code> 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	<code>bae log list --instanceid Instance_ID</code>
[ ]或者[a b]	表示可选项，至多选择一个。	<code>ipconfig [-all -t]</code>
{ }或者{a b}	表示必选项，至多选择一个。	<code>swich {stand   slave}</code>

# 目录

法律声明.....	I
通用约定.....	I
<b>1 下载设备端SDK.....</b>	<b>1</b>
<b>2 设备安全认证.....</b>	<b>4</b>
2.1 概览.....	4
2.2 一机一密.....	5
2.3 一型一密.....	7
<b>3 设备多协议连接.....</b>	<b>9</b>
3.1 MQTT协议规范.....	9
3.2 MQTT-TCP连接通信.....	10
3.3 MQTT-WebSocket连接通信.....	14
3.4 CoAP连接通信.....	16
3.5 HTTP连接通信.....	22
<b>4 物模型开发.....</b>	<b>27</b>
<b>5 设备OTA开发.....</b>	<b>37</b>
<b>6 子设备接入错误码.....</b>	<b>45</b>
6.1 子设备接入.....	45
6.2 子设备开发错误码.....	54
<b>7 设备影子.....</b>	<b>57</b>
7.1 设备影子JSON详解.....	57
7.2 设备影子数据流.....	58
7.3 设备影子开发.....	65
<b>8 编译与API参考.....</b>	<b>68</b>
<b>9 跨平台移植说明.....</b>	<b>78</b>
<b>10 Android-SDK.....</b>	<b>89</b>
<b>11 JAVA-SDK.....</b>	<b>99</b>
<b>12 基于Alink协议开发.....</b>	<b>101</b>
12.1 Alink协议.....	101
12.2 设备身份注册.....	108
12.3 添加拓扑关系.....	112
12.4 子设备上下线.....	118
12.5 设备属性、事件、服务.....	121
12.6 网关配置下发.....	133
12.7 设备禁用、删除.....	148

12.8 设备标签.....	151
12.9 TSL模板.....	153
12.10 固件升级.....	157
12.11 远程配置.....	160
12.12 设备端通用code.....	164



# 1 下载设备端SDK

物联网平台提供各类设备端SDK，简化开发过程，使设备快速上云。若您不使用SDK，请参考[Alink协议](#)文档自行开发。

## 前提条件

在设备端开发之前，您需要首先完成控制台所需操作，以获取设备开发阶段的必要信息，包括设备信息、Topic信息等。具体内容请参考用户指南。

## 设备端SDK

您可以根据实际环境的协议、功能需求，选择合适的设备端SDK进行开发。推荐使用实现了更多逻辑的C SDK。



说明：

如果提供的SDK不能满足您的需求，您也可以基于[Alink协议](#)自行开发。

	C SDK	Java SDK	Android SDK	iOS SDK	HTTP/2 SDK	泛化协议
MQTT	✓	✓	✓	✓		
CoAP	✓					
HTTP/HTTPS	✓					
HTTP/2					✓	
其他协议						✓
设备认证：一机一密	✓	✓	✓	✓	✓	✓
设备认证：一型一密	✓		✓			
OTA开发	✓					
子设备接入	✓		✓			
设备影子	✓	✓	✓			
基于物模型的设备开发	✓		✓			
远程配置	✓		✓			

## 适配平台

您可以单击[此处](#)，查询阿里云物联网平台适配的平台及详细信息。

如果您使用的平台未被适配，请访问[官方Github](#)，给我们提出建议。

## 下载SDK

### • C SDK

版本号	发布日期	开发环境	下载链接	更新内容
V2.2.1	2018/09/03	64位 Linux, GNU Make	<a href="#">RELEASED_V2.2.1</a>	<ul style="list-style-type: none"> <li>• WiFi配网、设备本地控制开源</li> <li>• 支持离线倒计时例程</li> <li>• OTA支持使用iTLs下载固件</li> </ul>
V2.1.0	2018/03/31	64位 Linux, GNU Make	<a href="#">RELEASED_V2_10_20180331.zip</a>	<ul style="list-style-type: none"> <li>• 支持cmake: 支持cmake编译方式，可以直接在linux和windows下使用QT或者VS2017打开工程进行编译运行。</li> <li>• 支持云端对物模型的抽象：设置FEATURE_CMP_ENABLED = y和FEATURE_DM_ENABLED = y, 可以支持物模型抽象，提供属性，服务和事件的接口。</li> <li>• 支持一型一密: 设置FEATURE_SUPPORT_PRODUCT_SECRET = y可以支持一型一密功能，优化产线流程。</li> <li>• 支持iTLS功能: 设置FEATURE_MQTT_DIRECT_NOITLS = y和FEATURE_MQTT_DIRECT_NOITLS = n可以支持ID2加密方式，使用iTLS进行数据建连，增加安全性，降低内存消耗。</li> <li>• 支持远程配置: 设置FEATURE_SERVICE_OTA_ENABLED = y和FEATURE_SERVICE_COTA_ENABLED = y，可以支持云端推送配置信息到设备。</li> <li>• 优化主子设备功能：主子设备添加部分功能。</li> </ul>

### • Java SDK

支持协议	更新历史	下载链接
MQTT	2017-05-27：支持华东2节点的设备认证流程，同时添加java端设备影子demo。	<a href="#">iotx-sdk-mqtt-java</a> ：MQTT的JAVA版只是使用开源库实现的一个demo，仅用于参考。

使用说明：[JAVA-SDK](#)。



- Android SDK

使用说明：[Android-SDK](#)

- iOS SDK

下载地址：

- <https://github.com/CocoaPods/Specs.git>
- <https://github.com/aliyun/aliyun-specs.git>

使用说明：[iOS-SDK](#)

- HTTP/2 SDK

下载地址：[iot-http2-sdk-demo](#)。

使用说明：[HTTP/2 SDK](#)

- 泛化协议

使用说明：[泛化协议](#)

- 其他开源库参考

下载地址为：<https://github.com/mqtt/mqtt.github.io/wiki/libraries>

## 2 设备安全认证

为保障设备安全，物联网平台为设备颁发证书，包括产品证书（ProductKey和ProductSecret）与设备证书（DeviceName和DeviceSecret）。其中，设备证书与设备一一对应，以确保设备的唯一合法性。设备通过协议接入IoT Hub之前，需依据不同的认证方案，上报产品证书和设备证书，云端通过认证后，方可接入物联网平台。针对不同的使用环境，物联网平台提供了多种认证方案。

物联网平台目前提供三种认证方案，分别是：

- 一机一密：每台设备烧录自己的设备证书。
- 一型一密：同一产品下设备烧录相同产品证书。
- 子设备认证：网关连接上云后，子设备的认证方案。

三种方案在易用性和安全性上各有优势，您可以根据设备所需的安全等级和实际的产线条件灵活选择，方案对比如下图所示。

表 2-1: 认证方案对比

对比项	一机一密	一型一密	子设备注册
设备端烧录信息	ProductKey、DeviceName、DeviceSecret	ProductKey、ProductSecret	ProductKey
云端是否需要开启	无需开启，默认支持。	需打开动态注册开关	需打开动态注册开关
是否需要预注册DeviceName	需要，确保产品下DeviceName唯一	需要，确保产品下DeviceName唯一	需要预注册
产线烧录要求	逐一烧录设备证书，需确保设备证书的安全性	批量烧录相同的产品证书，需确保产品证书的安全存储	子设备批量烧录相同的产品证书，但需要确保网关的安全性
安全性	较高	一般	一般
是否有配额限制	有，单个产品50万上限	有，单个产品50万上限	有，单个网关最多可注册200个子设备
其他外部依赖	无	无	依赖网关的安全性保障

### 2.1 概览

为保障设备安全，物联网平台为设备颁发证书，包括产品证书（ProductKey和ProductSecret）与设备证书（DeviceName和DeviceSecret）。其中，设备证书与设备一一对应，以确保设备的唯一合

法性。设备通过协议接入IoT Hub之前，需依据不同的认证方案，上报产品证书和设备证书，云端通过认证后，方可接入物联网平台。针对不同的使用环境，物联网平台提供了多种认证方案。

物联网平台目前提供三种认证方案，分别是：

- 一机一密：每台设备烧录自己的设备证书。
- 一型一密：同一产品下设备烧录相同产品证书。
- 子设备认证：网关连接上云后，子设备的认证方案。

三种方案在易用性和安全性上各有优势，您可以根据设备所需的安全等级和实际的产线条件灵活选择，方案对比如下图所示。

表 2-2: 认证方案对比

对比项	一机一密	一型一密	子设备注册
设备端烧录信息	ProductKey、 DeviceName、 DeviceSecret	ProductKey、 ProductSecret	ProductKey
云端是否需要开启	无需开启，默认支持。	需打开动态注册开关	需打开动态注册开关
是否需要预注册 DeviceName	需要，确保产品下 DeviceName唯一	需要，确保产品下 DeviceName唯一	需要预注册
产线烧录要求	逐一烧录设备证书，需 确保设备证书的安全性	批量烧录相同的产品证 书，需确保产品证书的 安全存储	子设备批量烧录相同的 产品证书，但需要确保 网关的安全性
安全性	较高	一般	一般
是否有配额限制	有，单个产品50万上限	有，单个产品50万上限	有，单个网关最多可注 册1500个子设备
其他外部依赖	无	无	依赖网关的安全性保障

## 2.2 一机一密

一机一密认证方法，即预先为每个设备烧录其唯一的设备证书 ( ProductKey、DeviceName和DeviceSecret )。当设备与物联网平台建立连接时，物联网平台对其携带的设备证书信息进行认证。认证通过，物联网平台激活设备，设备与物联网平台间才可传输数据。

### 背景信息

一机一密认证方式的安全性较高，推荐使用。

使用流程示意图：



## 操作步骤

1. 在[IoT控制台](#)，创建产品。如需帮助，请参考用户指南中，[创建产品与设备章节](#)。
2. 在产品下，添加设备，并获取设备证书信息。



3. 将设备证书信息烧录至设备中。
- 烧录设备证书信息操作步骤如下：
- a) 下载设备端SDK。
  - b) 初始化设备端SDK。在设备端SDK中，填入设备的证书信息（ProductKey、DeviceName和DeviceSecret）。
  - c) 根据实际需求，完成设备端SDK开发，如，OTA开发、子设备接入、设备物模型开发、设备影子开发等。
  - d) 在产线上，将已开发完成的设备SDK烧录至设备中。
4. 设备上电联网后，向物联网平台发起认证激活请求。
  5. 物联网平台对设备证书进行校验。认证通过后，与设备建立连接，设备便可通过发布消息至Topic和订阅Topic消息，与物联网平台进行数据通信。

## 2.3 一型一密

一型一密安全认证方式，即为同一产品下所有设备烧录相同固件（固件中写入产品证书，即ProductKey和ProductSecret）。设备发送激活请求时，物联网平台根据产品证书进行认证。认证通过，物联网平台下发该设备对应的DeviceSecret。

### 背景信息



说明：

- 采用一型一密方式认证，设备烧录相同固件，存在产品证书泄露风险。您可以在产品详情页面，手动关闭动态注册开关，拒绝新设备的认证请求。
- 一型一密认证方式用于获取设备的DeviceSecret。DeviceSecret仅下发一次，设备端需保存该DeviceSecret，用于后续登录。

使用流程示意图：



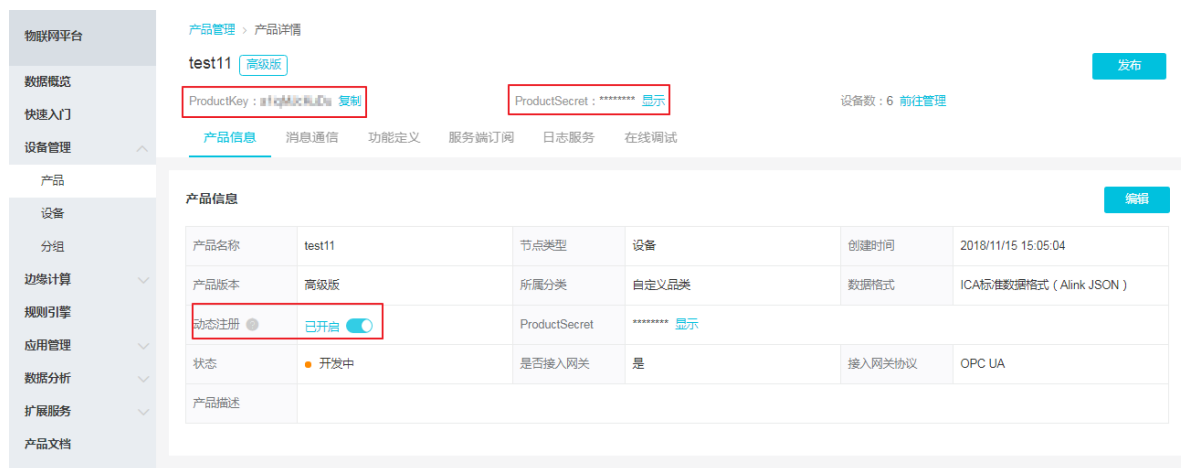
### 操作步骤

- 在[IoT控制台](#)，创建产品。如需帮助，请参考用户指南中，[创建产品与设备章节](#)。
- 在已创建产品的产品详情页面，开启动态注册开关。系统将进行短信验证，以确认是您本人操作。



说明：

若设备发出激活请求时，系统校验发现该开关未开启，将拒绝新设备的动态激活请求。已激活设备不受影响。



### 3. 在该产品下，添加设备。添加成功的设备状态为未激活。

因设备激活时会校验DeviceName，建议您采用可以直接从设备中读取到的ID，如设备的MAC地址、IMEI或SN号等，作为DeviceName使用。

### 4. 为设备烧录产品证书。

烧录产品证书操作步骤如下：

a) 下载设备端SDK。

b) 初始化设备端SDK，开通设备端SDK一型一密注册。在设备端SDK中，填入产品证书 ( ProductKey和ProductSecret ) 。

c) 根据实际需求，完成设备端SDK开发，如，OTA开发、子设备接入、设备物模型开发、设备影子开发等。

d) 在产线上，将已开发完成的设备SDK烧录至设备中。

### 5. 设备上电联网后，通过一型一密发起认证请求。

### 6. 物联网平台校验产品证书通过后，动态下发该设备对应的DeviceSecret。至此，设备获得连接云端所需的三元组信息 ( ProductKey、DeviceName和DeviceSecret )，便可使用三元组信息与云端建立连接，通过发布消息到Topic和订阅Topic消息，与云端进行数据通信。



说明：

仅在设备首次激活时，动态下发DeviceSecret。若需重新激活设备，请在物联网平台删除该设备后，再重新注册设备。

## 3 设备多协议连接

---

### 3.1 MQTT协议规范

#### 支持版本

目前阿里云支持MQTT标准协议接入，兼容3.1.1和3.1版本协议，具体的协议请参考 [MQTT 3.1.1](#) 和 [MQTT 3.1](#) 协议文档。

#### 与标准MQTT的区别

- 支持MQTT 的 PUB、SUB、PING、PONG、CONNECT、DISCONNECT和UNSUB等报文。
- 支持cleanSession。
- 不支持will、retain msg。
- 不支持QOS2。
- 基于原生的MQTT topic上支持RRPC同步模式，服务器可以同步调用设备并获取设备回执结果。

#### 安全等级

支持 TLSV1、 TLSV1.1和TLSV1.2 版本的协议建立安全连接。

- TCP通道基础 + 芯片级加密（ID2硬件集成）：安全级别高。
- TCP通道基础 + 对称加密（使用设备私钥做对称加密）：安全级别中。
- TCP方式（数据不加密）：安全级别低。

#### topic规范

默认情况下创建一个产品后，产品下的所有设备都拥有以下topic类的权限：

- `/${YourProductKey}/${YourDeviceName}/update pub`
- `/${YourProductKey}/${YourDeviceName}/update/error pub`
- `/${YourProductKey}/${YourDeviceName}/get sub`
- `/sys/${YourProductKey}/${YourDeviceName}/thing/# pub&sub`
- `/sys/${YourProductKey}/${YourDeviceName}/rrpc/# pub&sub`
- `/broadcast/${YourProductKey}/# pub&sub`

每个topic规则称为topic类，topic类实行设备维度隔离。每个设备发送消息时，将deviceName替换为自己设备的deviceName，防止topic被跨设备越权，topic说明如下：

- pub：表示数据上报到topic的权限。
- sub：表示订阅topic的权限。
- `/${YourProductKey}/${YourDeviceName}/xxx`类型的topic类：可以在物联网平台的控制台扩展和自定义。
- `/sys`开头的topic类：属于系统约定的应用协议通信标准，不允许用户自定义的，约定的topic需要符合阿里云ALink格式数据标准。
- `/sys/${YourProductKey}/${YourDeviceName}/thing/xxx`类型的topic类：网关主子设备使用的topic类，用于网关场景。
- `/broadcast`开头的topic类：广播类特定topic。
- `/sys/${YourProductKey}/${YourDeviceName}/rrpc/request/${yourmessageId}`：用于同步请求，服务器会对消息Id动态生成topic，设备端可以订阅通配符。
- `/sys/${YourProductKey}/${YourDeviceName}/rrpc/request/+`：收到消息后，发送pub消息到`/sys/${YourProductKey}/${YourDeviceName}/rrpc/response/${yourmessageId}`，服务器可以在发送请求时，同步收到结果。

## 3.2 MQTT-TCP连接通信

本文档主要介绍基于TCP的MQTT连接，并提供了两种设备认证模式示例：MQTT客户端域名直连和使用HTTPS认证再连接模式。



说明：

在进行MQTT CONNECT协议设置时，注意：

- Connect指令中的KeepAlive时间范围为30至1200秒。如果心跳时间不在此区间内，物联网平台会拒绝连接。建议取值300秒以上。
- 如果同一个设备三元组，同时用于多个物理设备连接，可能会导致客户端频繁上下线。
- MQTT连接模式中，默认开源SDK下线后会自动重连。您可以通过日志服务查看设备行为。

### MQTT客户端域名直连

1. 推荐使用TLS加密。如果使用TLS加密，需要[下载根证书](#)。
2. 使用MQTT客户端连接服务器。连接方法，请参见[开源MQTT客户端参考](#)。如果需了解MQTT协议，请参考 <http://mqtt.org> 相关文档。



说明：



若使用第三方代码，阿里云不提供技术支持。

### 3. MQTT 连接。

连接域名	<code>\${YourProductKey}.iot-as-mqtt.\${YourRegionId}.aliyuncs.com:1883</code> <code>\${YourProductKey}</code> 请替换为您的产品key。 <code>\${YourRegionId}</code> 请参考 <a href="#">地域和可用区</a> 替换为您的Region ID。
MQTT的Connect报文参数	<pre>mqttClientId: clientId+" securemode=3,signmethod= hmacsha1,timestamp=132323232 " mqttUsername: deviceName+"&amp;"+productKey mqttPassword: sign_hmac(deviceSecret,content)</pre> <p><b>mqttPassword</b> : sign签名需把提交给服务器的参数按字典排序后，根据signmethod加签。  <b>content</b>的值为提交给服务器的参数 ( ProductKey、DeviceName、timestamp和clientId )，按照字母顺序排序, 然后将参数值依次拼接。</p> <ul style="list-style-type: none"> <li>• <b>clientId</b> : 表示客户端ID，建议使用设备的MAC地址或SN码，64字符内。</li> <li>• <b>timestamp</b> : 表示当前时间毫秒值，可以不传递。</li> <li>• <b>mqttClientId</b> : 格式中  内为扩展参数。</li> <li>• <b>signmethod</b> : 表示签名算法类型。支持hmacmd5，hmacsha1和hmacsha256，默认为hmacmd5。</li> <li>• <b>securemode</b> : 表示目前安全模式，可选值有2 ( TLS直连模式 ) 和 3 ( TCP直连模式 )。</li> </ul> <p>示例：            假设clientId = 12345 , deviceName = device ,            productKey = pk , timestamp = 789 , signmethod=            hmacsha1 , deviceSecret=secret , 那么使用TCP方式提交            给MQTT的参数如下：</p> <pre>mqttclientId=12345 securemode=3,signmethod= hmacsha1,timestamp=789  mqttUsername=device&amp;pk mqttPassword=hmacsha1("secret","clientId12 345deviceNamedeviceproductKeypktimestamp789").</pre>

	<pre>toHexString(); //最后是二进制转16制字符串，大小写不敏感。</pre>
	加密后的Password结果为：
	FAFD82A3D602B37FB0FA8B7892F24A477F851A14

## 使用HTTPS认证再连接模式

### 1. 认证设备。

使用HTTPS进行设备认证，认证域名为：`https://iot-auth.${YourRegionId}.aliyuncs.com/auth/devicename`。其中，`${YourRegionId}`请参考[地域和可用区](#)替换为您的Region ID。

aliyuncs.com/auth/devicename。其中，`${YourRegionId}`请参考[地域和可用区](#)替换为您的Region ID。

- 认证请求参数信息：

参数	是否可选	获取方式
productKey	必选	ProductKey，从物联网平台的控制台获取。
deviceName	必选	DeviceName，从物联网平台的控制台获取。
sign	必选	签名，格式为hmacmd5(deviceSecret,content)，content值是将所有提交给服务器的参数（version、sign、resources和signmethod除外），按照字母顺序排序，然后将参数值依次拼接（无拼接符号）。
signmethod	可选	算法类型。支持hmacmd5，hmacsha1和hmacsha256，默认为hmacmd5。
clientId	必选	表示客户端ID，64字符内。
timestamp	可选	时间戳。时间戳不做时间窗口校验。
resources	可选	希望获取的资源描述，如MQTT。多个资源名称用逗号隔开。

- 返回参数：

参数	是否必选	含义
iotId	必选	服务器颁发的一个连接标记，用于赋值给MQTT connect报文中的username。

参数	是否必选	含义
iotToken	必选	token有效期为7天，赋值给MQTT connect报文中的password。
resources	可选	资源信息，扩展信息比如MQTT服务器地址和CA证书信息等。

- x-www-form-urlencoded请求示例：

```
POST /auth/devicename HTTP/1.1
Host: iot-auth.cn-shanghai.aliyuncs.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 123
productKey=123&sign=123&timestamp=123&version=default&clientId=123
&resources=mqtt&deviceName=test
sign = hmac_md5(deviceSecret, clientId123deviceName123productKey123timestamp123)
```

- 请求响应：

```
HTTP/1.1 200 OK
Server: Tengine
Date: Wed, 29 Mar 2017 13:08:36 GMT
Content-Type: application/json; charset=utf-8
Connection: close
{
  "code" : 200,
  "data" : {
    "iotId" : "42Ze0mk3556498a1AlTP",
    "iotToken" : "0d7fdeb9dc1f4344a2cc0d45edcb0bcb",
    "resources" : {
      "mqtt" : {
        "host" : "xxx.iot-as-mqtt.cn-shanghai.aliyuncs.com",
        "port" : 1883
      }
    },
    "message" : "success"
  }
}
```

## 2. 连接MQTT。

- 下载物联网平台 [root.crt](#)，建议使用TLS1.2。
- 设备端连接阿里云MQTT服务器地址。使用设备认证返回的MQTT地址和端口进行通信。
- 采用TLS建立连接。客户端通过CA证书验证物理网平台服务器；物联网平台服务器通过MQTT协议体内connect报文信息验证客户端，其中，username=iotId，password=iotToken，clientId=自定义的设备标记（建议MAC或SN）。

如果*iotId*或*iotToken*非法，则连接失败，收到的connect ack为3。

错误码说明如下：

- 401: request auth error：在这个场景中，通常在签名匹配不通过时返回。
- 460: param error：参数异常。
- 500: unknown error：一些未知异常。
- 5001: meta device not found：指定的设备不存在。
- 6200: auth type mismatch：未授权认证类型错误。

### MQTT保活

设备端在保活时间间隔内，至少需要发送一次报文，包括ping请求。

如果物联网平台在保活时间内无法收到任何报文，物联网平台会断开连接，设备端需要进行重连。

设置保活时间的取值，取值范围为30至1200秒，建议取值300秒以上。

## 3.3 MQTT-WebSocket连接通信

### 背景信息

支持基于WebSocket的MQTT协议，首先使用WebSocket建立连接，然后在WebSocket通道上，使用MQTT协议进行通信，即Mqtt-over-WebSocket。

使用WebSocket方式主要有以下优势：

- 使基于浏览器的应用程序可以像普通设备一样，具备和服务端建立MQTT长连接的能力。
- WebSocket方式使用443端口，使消息可以顺利穿过大多数防火墙。

### 操作步骤

#### 1. 证书准备。

WebSocket可以使用ws和wss两种方式，ws就是普通的WebSocket连接，wss就是增加了TLS加密。如果使用wss方式进行安全连接，需要使用和TLS直连一样的[根证书](#)。

#### 2. 客户端选择。

JAVA版本可以直接使用[官方客户端sdk](#)，只需要替换连接URL即可。其他语言版本客户端或者是自主接入，请参考[开源MQTT客户端](#)参考，使用前请阅读相关客户端的说明，是否支持WebSocket方式。

#### 3. 连接说明。

使用WebSocket方式进行连接，区别主要在MQTT连接URL的协议和端口号，MQTT连接参数和TCP直接连接方式完全相同，其中要注意securemode参数，使用wss方式连接时securemode=2，使用ws方式连接时securemode=3。

- 连接域名，华东2节点：`${YourProductKey}.iot-as-mqtt.cn-shanghai.aliyuncs.com:443`

`${YourProductKey}`请替换为您的产品key。

- MQTT的Connect报文参数如下：

```
mqttClientId: clientId+"|securemode=3,signmethod=hmacsha1,
timestamp=132323232|"
mqttUsername: deviceName+"&"+productKey
mqttPassword: sign_hmac(deviceSecret,content)sign签名需要把以下参数按
字典序排序后，再根据signmethod加签。
content=提交给服务器的参数 ( productKey,deviceName,timestamp,clientId
)，按照字母顺序排序，然后将参数值依次拼接
```

其中，

- `clientId`：表示客户端ID，建议mac或sn，64字符内。
- `timestamp`：表示当前时间毫秒值，可选。
- `mqttClientId`：格式中`||`内为扩展参数。
- `signmethod`：表示签名算法类型。
- `securemode`：表示目前安全模式，可选值有2（wss协议）和3（ws协议）。

参考示例，如果预置前提如下：

```
clientId = 12345,deviceName = device, productKey = pk, timestamp
= 789,signmethod=hmacsha1,deviceSecret=secret
```

- 使用ws方式

#### — 连接域名

```
ws://pk.iot-as-mqtt.cn-shanghai.aliyuncs.com:443
```

#### — 连接参数

```
mqttclientId=12345|securemode=3,signmethod=hmacsha1,timestamp=
789|
mqttUsername=device&pk
mqttPasswrod=hmacsha1("secret","clientId12345deviceNamedevicep
roductKeypktimestamp789").toHexString();
```

- 使用wss方式

### — 连接域名

```
wss://pk.iot-as-mqtt.cn-shanghai.aliyuncs.com:443
```

### — 连接参数

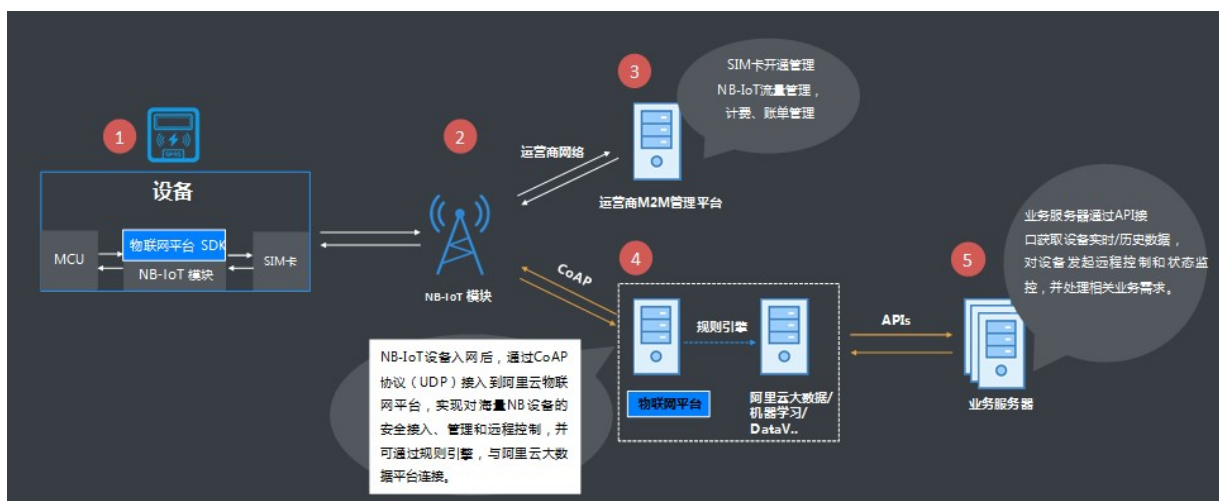
```
mqttclientId=12345|securemode=2,signmethod=hmacsha1,timestamp=789|
mqttUsername=device&pk
mqttPasswrod=hmacsha1("secret","clientId12345deviceNamedeviceproductKeypktimestamp789").toHexString();
```

## 3.4 CoAP连接通信

物联网平台支持CoAP协议连接通信。CoAP协议适用在资源受限的低功耗设备上，尤其是NB-IoT的设备使用。本文介绍基于CoAP协议进行设备接入的流程，及使用DTLS和对称加密两种认证方式下的自主接入流程。

### 基础流程

基于CoAP协议将NB-IoT设备接入物联网平台的流程如下图所示：



基础流程说明如下：

1. 在设备端NB-IoT模块中，集成阿里云物联网平台SDK。厂商在物联网平台的控制台申请设备证书 ( ProductKey、DeviceName和DeviceSecret ) 并烧录到设备中。
2. NB-IoT设备通过运营商的蜂窝网络进行入网。需要联系当地运营商，确保设备所属地区已经覆盖NB网络，并具备NB-IoT入网能力。
3. 设备入网成功后，NB设备产生的流量数据及产生的费用数据，将由运营商的M2M平台管理。此部分平台能力由运营商提供。

4. 设备开发者可通过 CoAP/UDP 协议，将设备采集的实时数据上报到阿里云物联网平台，借助物联网平台，实现海量亿级设备的安全连接和数据管理能力。并且，可通过规则引擎，将数据转发至阿里云的大数据产品、云数据库、表格存储等服务中进行处理。
5. 物联网平台提供相关的数据开放接口和消息推送服务，可将数据转发到业务服务器中，实现设备资产与实际应用的快速集成。

### 使用DTLS自主接入

1. 连接CoAP服务器，endpoint 地址为`${YourProductKey}.coap.cn-shanghai.link.aliyuncs.com:${port}`。

其中：

- `${YourProductKey}`：请替换为您申请的产品Key。
- `${port}`：端口。使用DTLS时，端口为5684。

2. 使用DTLS安全通道，需下载[根证书](#)。
3. 设备认证。使用auth接口认证设备，获取token。上报数据时，需携带token信息。

设备认证请求：

```
POST /auth
Host: ${YourProductKey}.coap.cn-shanghai.link.aliyuncs.com
Port: 5684
Accept: application/json or application/cbor
Content-Format: application/json or application/cbor
payload: {"productKey": "ZG1EvTEa7NN", "deviceName": "NlwaSPXsCpTQuh8FxBGH", "clientId": "mylight1000002", "sign": "bccb3d2618afe74b3eab12b94042f87b"}
```

表 3-1: 设备认证参数说明

参数	说明
Method	请求方法。只支持POST方法。
URL	URL地址，取值： /auth。
Host	endpoint地址。取值格式： <code>\${YourProductKey}.coap.cn-shanghai.link.aliyuncs.com</code> 。其中，变量 <code>\${YourProductKey}</code> 需替换为您的产品Key。
Port	端口，取值：5684。
Accept	设备接收的数据编码方式。目前，支持两种方式：application/json和application/cbor。

参数	说明
Content-Format	设备发送给物联网平台的上行数据的编码格式，目前，支持两种方式： <code>application/json</code> 和 <code>application/cbor</code> 。
payload	设备认证信息内容，JSON数据格式。具体参数，请参见下表 <a href="#">payload 说明</a> 。

表 3-2: payload 说明

字段名称	是否必需	说明
productKey	是	设备三元组信息中ProductKey的值，是物联网平台为产品颁发的全局唯一标识。从物联网平台的控制台获取。
deviceName	是	设备三元组信息中DeviceName的值，在注册设备时自定义的或自动生成的设备名称。从物联网平台的控制台获取。
ackMode	否	通信模式。取值： <ul style="list-style-type: none"> <li>0：<code>request/response</code> 是携带模式，即客户端发送请求到服务端后，服务端处理完业务，回复业务数据和ACK。</li> <li>1：<code>request/response</code> 是分离模式，即客户端发送请求到服务端后，服务端先回复一个确认ACK，然后再处理业务后，回复业务数据。</li> </ul> 若不传入此参数，则默认为携带模式。
sign	是	签名。 签名计算格式为 <code>hmacmd5(DeviceSecret, content)</code> 。 其中， <b>content</b> 为将所有提交给服务器的参数（除 <b>version</b> 、 <b>sign</b> 、 <b>resources</b> 和 <b>signmethod</b> 外），按照英文字母升序，依次拼接排序（无拼接符号）。
signmethod	否	算法类型，支持 <code>hmacmd5</code> 和 <code>hmacsha1</code> 。
clientId	是	客户端ID，长度需在64字符内。建议使用设备的MAC地址或SN码作为clientId的值。
timestamp	否	时间戳。目前，时间戳不做时间窗口校验。



返回结果示例：

```
response : {"token": "f13102810756432e85dfd351eeb41c04"}
```

表 3-3: 返回码说明

Code	描述	Payload	备注
2.05	Content	认证通过： Token对象	正确请求。
4.00	Bad Request	no payload	请求发送的Payload非法。
4.01	Unauthorized	no payload	未授权的请求。
4.03	Forbidden	no payload	禁止的请求。
4.04	Not Found	no payload	请求的路径不存在。
4.05	Method Not Allowed	no payload	请求方法不是指定值。
4.06	Not Acceptable	no payload	Accept不是指定的类型。
4.15	Unsupported Content-Format	no payload	请求的content不是指定类型。
5.00	Internal Server Error	no payload	auth服务器超时或错误。

#### 4. 上行数据。

设备发送数据到某个Topic。自定义Topic，在物联网平台的控制台，设备所属产品的产品详情页的消息通信栏中进行设置。

目前，只支持发布权限的Topic用于数据上报，如/`${YourProductKey}/${YourDeviceName}/pub`。假设当前设备名称为device，产品Key为a1GFjLP3xxC，那么您可以调用a1GFjLP3xxC.coap.cn-shanghai.link.aliyuncs.com:5684/topic/a1GFjLP3xxC/device/pub 地址来上报数据。

上报数据请求：

```
POST /topic/${topic}
Host: ${YourProductKey}.coap.cn-shanghai.link.aliyuncs.com
Port: 5684
Accept: application/json or application/cbor
Content-Format: application/json or application/cbor
payload: ${your_data}
```

CustomOptions: number:2088 (标识token)

表 3-4: 上报数据请求参数说明

参数	是否必需	说明
Method	是	请求方法。支持POST方法。
URL	是	/topic/\${topic}。其中，变量\${topic}需替换为当前设备对应的Topic。
Host	是	endpoint地址。取值格式：\${YourProductKey}.coap.cn-shanghai.link.aliyuncs.com。其中，变量\${YourProductKey}需替换为您的产品Key。
Port	是	端口，取值：5684。
Accept	是	设备接收的数据编码方式。目前，支持两种方式：application/json和application/cbor。
Content-Format	是	上行数据的编码格式，服务端对此不做校验。目前，支持两种方式：application/json和application/cbor。
CustomOptions	是	<ul style="list-style-type: none"> <li>number取值：2088。</li> <li>token为设备认证#auth#返回的token值。</li> </ul> <div>  <p>说明： 每次上报数据都需要携带token信息。如果token失效，需重新进行设备认证获取token。</p> </div>

## 使用对称加密自主接入

1. 连接CoAP服务器，endpoint地址为\${YourProductKey}.coap.cn-shanghai.link.aliyuncs.com:\${port}。

其中：

- \${YourProductKey}：请替换为您申请的产品Key。
- \${port}：端口。使用对称加密时端口为5682。

2. 设备认证。

设备认证请求：

```
POST /auth
Host: ${YourProductKey}.coap.cn-shanghai.link.aliyuncs.com
```

```
Port: 5682
Accept: application/json or application/cbor
Content-Format: application/json or application/cbor
payload: { "productKey": "a1NUjcVkhZS", "deviceName": "ff1a11e7c08d4b3db2b1500d8e0e55", "clientId": "a1NUjcVkhZS&ff1a11e7c08d4b3db2b1500d8e0e55", "sign": "F9FD53EE0CD010FCA40D14A9FEAB81E0", "seq": "10" }
```

参数（除 **Port** 参数外。使用对称加密时的Port为 5682）及payload内容说明，可参见[参数说明](#)中。

返回结果示例：

```
{ "random": "ad2b3a5eb51d64f7", "seqOffset": 1, "token": "MZ8m37hp01w1SSqoDFzo0010500d00.ad2b" }
```

表 3-5: 返回参数说明

字段名称	说明
random	用于后续上、下行加密，组成加密Key。
seqOffset	认证seq偏移初始值。
token	设备认证成功后，返回的token值。


3. 上报数据。

上报数据请求：

```
POST /topic/${topic}
Host: ${YourProductKey}.coap.cn-shanghai.link.aliyuncs.com
Port: 5682
Accept: application/json or application/cbor
Content-Format: application/json or application/cbor
payload: ${your_data}
CustomOptions: number:2088(标识token), 2089(seq)
```

表 3-6: 上报数据参数说明

字段名称	是否必需	说明
Method	是	请求方法。支持POST方法。
URL	是	传入格式： <code>/topic/\${topic}</code> 。其中，变量 <code>\${topic}</code> 需替换为设备数据上行Topic。
Host	是	endpoint地址。传入格式： <code>\${YourProductKey}.coap.cn-shanghai.link.aliyuncs.com</code> 。其

字段名称	是否必需	说明
		中， <code>\${YourProductKey}</code> 需替换为设备所属产品的Key。
Port	是	端口。取值：5682。
Accept	是	设备接收的数据编码方式。目前，支持两种方式：application/json和application/cbor。
Content-Format	是	上行数据的编码格式，服务端对此不做校验。目前，支持两种方式：application/json和application/cbor。
payload	是	待上传的数据经高级加密标准（AES）加密后的数据。
CustomOptions	是	<p>option值有2088和2089两种类型，说明如下：</p> <ul style="list-style-type: none"> <li>2088：表示token，取值为设备认证后返回的token值。</li> </ul> <div style="border: 1px solid #ccc; padding: 5px; margin: 10px 0;">  说明： 每次上报数据都需要携带token信息。如果token失效，需重新认证获取token。 </div> <ul style="list-style-type: none"> <li>2089：表示seq，取值需比设备认证后返回的seqOffset值更大，且在认证生效周期内不重复的随机值。使用AES加密该值。</li> </ul> <p>option返回示例：</p> <pre>number:2090(云端消息ID)</pre>

消息上行成功后，返回成功状态码，同时返回物联网平台生成的消息ID。

### 3.5 HTTP连接通信

物联网平台支持使用HTTP接入，目前仅支持HTTPS协议。

#### 说明

- HTTP服务器地址为<https://iot-as-http.cn-shanghai.aliyuncs.com>。
- 目前，HTTP通信仅支持华东2（上海）地域。
- 只支持HTTPS协议。

- Topic规范和MQTT的Topic规范一致。使用HTTP协议连接，上报数据`https://iot-as-http.cn-shanghai.aliyuncs.com/topic/${topic}`使用的`${topic}`可以与MQTT连接通信的Topic 相复用。不支持`?query_String=xxx`形式传参
- 上行接口传输的数据大小限制为128 KB。

## 接入流程

1. 连接HTTP服务器。

endpoint地址：`https://iot-as-http.cn-shanghai.aliyuncs.com`

2. 认证设备，获取设备的token。

认证设备请求：

```
POST /auth HTTP/1.1
Host: iot-as-http.cn-shanghai.aliyuncs.com
Content-Type: application/json
body: {"version":"default","clientId":"mylight1000002","signmethod":"hmacsha1","sign":"4870141D4067227128CBB4377906C3731CAC221C","productKey":"ZG1EvTEa7NN","deviceName":"NlwaSPXsCpTQuh8FxBGH","timestamp":"1501668289957"}
```

表 3-7: 参数说明

参数	说明
Method	请求方法。支持POST方法。
URL	/auth，URL地址，只支持HTTPS。
Host	endpoint地址： <code>iot-as-http.cn-shanghai.aliyuncs.com</code>
Content-Type	设备发送给物联网平台的上行数据的编码格式。目前只支持 <code>application/json</code>
body	设备认证信息。JSON数据格式。具体信息，请参见下表body参数。

表 3-8: body参数

字段名称	是否必需	说明
productKey	是	设备所属产品的Key。从物联网平台的控制台获取。
deviceName	是	设备名称。从物联网平台的控制台获取。
clientId	是	客户端ID。长度为64字符内，建议以MAC地址或SN码作为clientId。

字段名称	是否必需	说明
timestamp	否	时间戳。校验时间戳15分钟内的请求有效。
sign	是	<p>签名。</p> <p>签名计算格式为<code>hmacmd5(DeviceSecret,content)</code>。</p> <p>其中，<b>content</b>为将所有提交给服务器的参数（除<b>version</b>、<b>sign</b>和<b>signmethod</b>外），按照英文字母升序，依次拼接排序（无拼接符号）的结果。</p> <p>签名示例：</p> <p>假设<code>clientId = 12345</code>，<code>deviceName = device</code>，<code>productKey = pk</code>，<code>timestamp = 789</code>，<code>signmethod=hmacsha1</code>，<code>deviceSecret=secret</code>，那么签名计算为<code>hmacsha1("secret","clientId12345deviceNamedeviceproductKeypktimestamp789").toHexString();</code>。最后二进制数据转十六进制字符串，大小写不敏感。</p>
signmethod	否	<p>算法类型，支持<code>hmacmd5</code>和<code>hmacsha1</code>。</p> <p>若不传入此参数，则默认为<code>hmacmd5</code>。</p>
version	否	版本号。若不传入此参数，则默认default。

设备认证返回结果示例：

```
body:
{
  "code": 0, //业务状态码
  "message": "success", //业务信息
  "info": {
    "token": "6944e5bfb92e4d4ea3918d1eda3942f6"
  }
}
```



说明：

- 返回的token可以缓存到本地。
- 每次上报数据时，都需要携带token信息。如果token失效，需要重新认证获取token。

表 3-9: 错误码说明

code	message	备注
10000	common error	未知错误。

code	message	备注
10001	param error	请求的参数异常。
20000	auth check error	设备鉴权失败。
20004	update session error	更新失败。
40000	request too many	请求次数过多，流控限制。

### 3. 上报数据。

发送数据到某个Topic。

自定义Topic，在物联网平台的控制台，设备所属产品的产品详情页，消息通信栏中设置

如Topic为/\${YourProductKey}/\${YourDeviceName}/pub。假设当前设备名称为device123，产品Key为a1GFjLP3xxC，那么您可以调用 <https://iot-as-http.cn-shanghai.aliyuncs.com/topic/a1GFjLP3xxC/device123/pub>地址来上报数据。

上报数据请求：

```
POST /topic/${topic} HTTP/1.1
Host: iot-as-http.cn-shanghai.aliyuncs.com
password:${token}
Content-Type: application/octet-stream
body: ${your_data}
```

表 3-10: 上报数据参数说明

参数	说明
Method	请求方法。支持POST方法。
URL	/topic/\${topic}。其中，变量\${topic}需替换为当前设备对应的Topic。只支持HTTPS。
Host	endpoint地址：iot-as-http.cn-shanghai.aliyuncs.com。
password	放在Header中的参数，取值为调用设备认证接口auth返回的token值。
body	发往\${topic}的数据内容，格式为二进制byte[]，UTF-8编码。

返回结果示例：

```
body:
{
  "code": 0, // 业务状态码
```

```
"message": "success", //业务信息
"info": {
  "messageId": 892687627916247040,
  "data": byte[] //utf-8编码,可能为空
}
```

表 3-11: 错误码说明

code	message	备注
10000	common error	未知错误。
10001	param error	请求的参数异常。
20001	token is expired	token失效。需重新调用 <b>auth</b> 进行鉴权，获取token。
20002	token is null	请求header中无token信息。
20003	check token error	根据token获取identify信息失败。需重新调用 <b>auth</b> 进行鉴权，获取token。
30001	publish message error	数据上行失败。
40000	request too many	请求次数过多，流控限制。



## 4 物模型开发

本文主要介绍如何基于物模型进行设备端开发。



说明：

该功能仅支持高级版产品。

### 准备工作

在云端创建合适的产品、设备和定义物模型，物模型定义包括属性，服务和事件，如图 4-1: 创建设备所示。

图 4-1: 创建设备

### 云端建连

1. 设备端与云端通过MQTT建立连接，具体请参考[MQTT-TCP连接通信](#)。
2. SDK中使用接口linkkit\_start，启动云端建连接，完成topic订阅。

使用SDK，SDK内保存一个影子，即设备的一个抽象，用于保存设备状态。设备与云端的交互就是设备与影子以及影子与云端的同步过程。

变量get\_tsl\_from\_cloud，用于设置是否需要运行时，从云端同步TSL。

- get\_tsl\_from\_cloud = 0：表明预置TSL，直接使用TSL\_STRING作为标准TSL。

将控制台产生的TSL复制后，作为linkkit\_sample.c中TSL\_STRING的定义，通过接口linkkit\_set\_tsl设置设备预置的TSL。



说明：

注意C语言中的转义符。

- get\_tsl\_from\_cloud = 1：表明没有预置TSL，运行时，动态从云端获取TSL。

动态获取TSL需要较大的运行内存并产生流量，具体数值取决于TSL的复杂程度。如果一个TSL有10k，则需要大约20k左右的运行内存和10k左右的流量。

3. 使用参数linkkit\_ops\_t，注册回调函数。

```
linkkit_start(8, get_tsl_from_cloud, linkkit_loglevel_debug, &
alinkops, linkkit_cloud_domain_sh, sample_ctx);
```

```
if (!get_tsl_from_cloud) {
    linkkit_set_tsl(TSL_STRING, strlen(TSL_STRING));
}
```

函数实现：

```
typedef struct _linkkit_ops {
    int (*on_connect)(void *ctx);
    int (*on_disconnect)(void *ctx);
    int (*raw_data_arrived)(void *thing_id, void *data, int len, void
*ctx);
    int (*thing_create)(void *thing_id, void *ctx);
    int (*thing_enable)(void *thing_id, void *ctx);
    int (*thing_disable)(void *thing_id, void *ctx);
#ifdef RRPC_ENABLED
    int (*thing_call_service)(void *thing_id, char *service, int
request_id, int rrpc, void *ctx);
#else
    int (*thing_call_service)(void *thing_id, char *service, int
request_id, void *ctx);
#endif /* RRPC_ENABLED */
    int (*thing_prop_changed)(void *thing_id, char *property, void *
ctx);
} linkkit_ops_t;
/**
 * @brief start linkkit routines, and install callback functions(
async type for cloud connecting).
 *
 * @param max_buffered_msg, specify max buffered message size.
 * @param ops, callback function struct to be installed.
 * @param get_tsl_from_cloud, config if device need to get tsl from
cloud(!0) or local(0), if local selected, must invoke linkkit_se
t_tsl to tell tsl to dm after start complete.
 * @param log_level, config log level.
 * @param user_context, user context pointer.
 * @param domain_type, specify the could server domain.
 *
 * @return int, 0 when success, -1 when fail.
 */
int linkkit_start(int max_buffered_msg, int get_tsl_from_cloud
, linkkit_loglevel_t log_level, linkkit_ops_t *ops, linkkit_cl
oud_domain_type_t domain_type, void *user_context);
/**
 * @brief install user tsl.
 *
 * @param tsl, tsl string that contains json description for thing
object.
 * @param tsl_len, tsl string length.
 *
 * @return pointer to thing object, NULL when fails.
 */
extern void* linkkit_set_tsl(const char* tsl, int tsl_len);
```

4. 与云端连接成功后，在控制台查看设备是否已经在线。

图 4-2: 设备在线

## 设备主动上报属性给云端

1. 当设备端属性发生变化，设备端可以通过topic：`/sys/{ProductKey}/{DeviceName}/thing/event/property/post`通知云端。

请求码流：

```
TOPIC: /sys/{ProductKey}/{DeviceName}/thing/event/property/post
REPLY TOPIC: /sys/{ProductKey}/{DeviceName}/thing/event/property/post_reply
request
{
  "id" : "123",
  "version": "1.0",
  "params" : {
    "PowerSwitch" : 1
  },
  "method": "thing.event.property.post"
}
response
{
  "id": "123",
  "code": 200,
  "data": {}
}
```

2. SDK通过接口linkkit\_set\_value修改设备中云端影子的属性值，linkkit\_trigger\_event将云端影子同步云端。



说明：

设备会把当前的属性值主动上报给云端。

函数原型：

```
linkkit_set_value(linkkit_method_set_property_value, sample->thing,
EVENT_PROPERTY_POST_IDENTIFIER, value, value_str); // set value
return linkkit_trigger_event(sample->thing, EVENT_PROPERTY_POST_IDENTIFIER, NULL); // update value to cloud
```

函数实现：

```
/**
 * @brief set value to property, event output, service output items.
 * if identifier is struct type or service output type or event
 * output type, use '.' as delimiter like "identifier1.identifier2"
 * to point to specific item.
 * value and value_str could not be NULL at the same time;
 * if value and value_str both as not NULL, value shall be used and
 * value_str will be ignored.
 * if value is NULL, value_str not NULL, value_str will be used.
```

```

* in brief, value will be used if not NULL, value_str will be used
only if value is NULL.
*
* @param method_set, specify set value type.
* @param thing_id, pointer to thing object, specify which thing to
set.
* @param identifier, property, event output, service output
identifier.
* @param value, value to set.(input int* if target value is int type
or enum or bool, float* if float type,
* long long* if date type, char* if text type).
* @param value_str, value to set in string format if value is null.
*
* @return 0 when success, -1 when fail.
*/
extern int linkkit_set_value(linkkit_method_set_t method_set, const
void* thing_id, const char* identifier,
const void* value, const char* value_str);
/**
* @brief trigger a event to post to cloud.
*
* @param thing_id, pointer to thing object.
* @param event_identifier, event identifier to trigger.
* @param property_identifier, used when trigger event with method "
event.property.post", if set, post specified property, if NULL, post
all.
*
* @return 0 when success, -1 when fail.
*/
extern int linkkit_trigger_event(const void* thing_id, const char*
event_identifier, const char* property_identifier);

```

## 云端获取设备属性

1. 在云端控制台，通过topic `/sys/{ProductKey}/{DeviceName}/thing/service/property/get` 主动获取设备端某个属性。

请求码流：

```

TOPIC: /sys/{ProductKey}/{DeviceName}/thing/service/property/get
REPLY TOPIC: /sys/{ProductKey}/{DeviceName}/thing/service/property/
get_reply
request
{
  "id" : "123",
  "version": "1.0",
  "params" : [
    "powerSwitch"
  ],
  "method": "thing.service.property.get"
}
response
{
  "id": "123",
  "code": 200,
  "data": {
    "powerSwitch": 0
  }
}

```

```
}

```

2. 设备端收到云端的获取指令后，SDK内部运行指令，从云端影子中直接读取属性值，自动返回给云端。

## 云端设置设备属性

1. 在云端控制台通过`topic/sys/{ProductKey}/{DeviceName}/thing/service/property/set`，修改设备端某个属性。

请求码流：

```
TOPIC: /sys/{ProductKey}/{DeviceName}/thing/service/property/set
REPLY TOPIC: /sys/{ProductKey}/{DeviceName}/thing/service/property/set_reply
payload:
{
  "id" : "123",
  "version": "1.0",
  "params" : {
    "PowerSwitch" : 0,
  },
  "method": "thing.service.property.set"
}
response
{
  "id": "123",
  "code": 200,
  "data": {}
}
```

2. SDK通过注册接口`linkkit_start`的参数`linkkit_ops_t`中的`thing_prop_changed`回调函数，响应云端设置设备属性的请求。
3. 在回调函数中，通过`linkkit_get_value`，可以获取到当前云端影子的设备属性，被云端修改后的属性值。
4. 调用者在处理完新的属性值后，通过函数`linkkit_answer_service`返回结果给云端，也可以不返回，由调用者的业务逻辑决定。

函数实现：

```
static int thing_prop_changed(void* thing_id, char* property, void* ctx)
{
  char* value_str = NULL;
  ...
  linkkit_get_value(linkkit_method_get_property_value, thing_id,
    property, NULL, &value_str);
  LINKKIT_PRINTF("#### property(%s) new value set: %s ####\n",
    property, value_str);
}
```

```
/* do user's process logical here. */
linkkit_trigger_event(thing_id, EVENT_PROPERTY_POST_IDENTIFIER,
property);
return 0;
}
```

回调函数原型：

```
int (*thing_prop_changed)(void *thing_id, char *property, void *ctx);
```

函数实现：

```
/**
 * @brief get value from property, event output, service input/output
 items.
 * if identifier is struct type or service input/output type or event
 output type, use '.' as delimiter like "identifier1.identifier2"
 * to point to specific item.
 * value and value_str could not be NULL at the same time;
 * if value and value_str both as not NULL, value shall be used and
 value_str will be ignored.
 * if value is NULL, value_str not NULL, value_str will be used.
 * in brief, value will be used if not NULL, value_str will be used
 only if value is NULL.
 * @param method_get, specify get value type.
 * @param thing_id, pointer to thing object, specify which thing to get
 .
 * @param identifier, property, event output, service input/output
 identifier.
 * @param value, value to get(input int* if target value is int type or
 enum or bool, float* if float type,
 * long long* if date type, char* if text type).
 * @param value_str, value to get in string format. DO NOT modify this
 when function returns,
 * user should copy to user's own buffer for further process.
 * user should NOT free the memory.
 *
 * @return 0 when success, -1 when fail.
 */
extern int linkkit_get_value(linkkit_method_get_t method_get, const
void* thing_id, const char* identifier,
void* value, char** value_str);
```

函数原型：

```
linkkit_set_value(linkkit_method_set_service_output_value, thing,
identifier, &sample->service_custom_output_contrastratio, NULL);
linkkit_answer_service(thing, service_identifier, request_id, 200);
```

函数实现：

```
/**
 * @brief set value to property, event output, service output items.
 * if identifier is struct type or service output type or event output
 type, use '.' as delimiter like "identifier1.identifier2"
```

```

* to point to specific item.
* value and value_str could not be NULL at the same time;
* if value and value_str both as not NULL, value shall be used and
value_str will be ignored.
* if value is NULL, value_str not NULL, value_str will be used.
* in brief, value will be used if not NULL, value_str will be used
only if value is NULL.
*
* @param method_set, specify set value type.
* @param thing_id, pointer to thing object, specify which thing to set
.
* @param identifier, property, event output, service output identifier
.
* @param value, value to set.(input int* if target value is int type
or enum or bool, float* if float type,
* long long* if date type, char* if text type).
* @param value_str, value to set in string format if value is null.
*
* @return 0 when success, -1 when fail.
*/
extern int linkkit_set_value(linkkit_method_set_t method_set, const
void* thing_id, const char* identifier,
const void* value, const char* value_str);
/**
* @brief answer to a service when a service requested by cloud.
*
* @param thing_id, pointer to thing object.
* @param service_identifier, service identifier to answer, user should
get this identifier from handle_dm_callback_fp_t type callback
* report that "dm_callback_type_service_requested" happened, use this
function to generate response to the service sender.
* @param response_id, id value in response payload. its value is from
"dm_callback_type_service_requested" type callback function.
* use the same id as the request to send response as the same
communication session.
* @param code, code value in response payload. for example, 200 when
service successfully executed, 400 when not successfully executed.
* @param rrpc, specify rrpc service call or not.
*
* @return 0 when success, -1 when fail.
*/
extern int linkkit_answer_service(const void* thing_id, const char*
service_identifier, int response_id, int code);

```

## 云端向设备发起服务

1. 云端通过topic `/sys/{ProductKey}/{DeviceName}/thing/service/{dsl.service.identifier}`发起设备的某个服务，定义在标准TSL的dsl.service.identifier中。

```

TOPIC: /sys/{ProductKey}/{DeviceName}/thing/service/{dsl.service.
identifier}
REPLY TOPIC:
/sys/{ProductKey}/{DeviceName}/thing/service/{dsl.service.identifier}
_reply
request
{
  "id" : "123",
  "version": "1.0",
  "params" : {

```

```

"SprinkleTime" : 50,
"SprinkleVolume" : 600
},
"method": "thing.service.AutoSprinkle"
}
response
{
  "id": "123",
  "code": 200,
  "data": {}
}

```

2. SDK通过注册接口linkkit\_start的参数linkkit\_ops\_t中的thing\_call\_service回调函数，响应云端发起服务的请求。
3. 调用者在处理完新的属性值后，必须调用函数linkkit\_answer\_service返回结果给云端。

函数原型：

```

int (*thing_call_service)(void *thing_id, char *service, int
request_id, void *ctx);

```

函数实现：

```

static int handle_service_custom(sample_context_t* sample, void*
thing, char* service_identifier, int request_id)
{
  char identifier[128] = {0};
  /*
   * get iutput value.
   */
  snprintf(identifier, sizeof(identifier), "%s.%s", service_identifier
, "SprinkleTime");
  linkkit_get_value(linkkit_method_get_service_input_value, thing,
identifier, &sample->service_custom_input_transparency, NULL);
  /*
   * set output value according to user's process result.
   */
  snprintf(identifier, sizeof(identifier), "%s.%s", service_identifier
, "SprinkleVolume");
  sample->service_custom_output_contrastratio = sample->service_cu
stom_input_transparency >= 0 ? sample->service_custom_input
_transparency : sample->service_custom_input_transparency * -1;
  linkkit_set_value(linkkit_method_set_service_output_value, thing,
identifier, &sample->service_custom_output_contrastratio, NULL);
  linkkit_answer_service(thing, service_identifier, request_id, 200);
  return 0;
}

```

## 设备端上报事件

1. 设备端通过topic `/sys/{ProductKey}/{DeviceName}/thing/event/{dsl.event.identifier}/post`，触发某个事件，定义在标准TSL的dsl.event.identifier中。



请求码流：

```

TOPIC: /sys/{ProductKey}/{DeviceName}/thing/event/{dsl.event.
identifer}/post
REPLY TOPIC: /sys/{ProductKey}/{DeviceName}/thing/event/{dsl.event.
identifer}/post_reply
request
{
  "id" : "123",
  "version":"1.0",
  "params" : {
    "ErrorCode" : 0
  },
  "method":"thing.event.Error.post"
}
response:
{
  "id" : "123",
  "code":200,
  "data" : {}
}

```

## 2. SDK通过接口linkkit\_trigger\_event来向云端触发事件。

函数原型：

```

static int post_event_error(sample_context_t* sample)
{
  char event_output_identifer[64];
  snprintf(event_output_identifer, sizeof(event_output_identifer),
    "%s.%s", EVENT_ERROR_IDENTIFIER, EVENT_ERROR_OUTPUT_INFO_IDENTIFIER
  );
  int errorCode = 0;
  linkkit_set_value(linkkit_method_set_event_output_value,
    sample->thing,
    event_output_identifer,
    &errorCode, NULL);
  return linkkit_trigger_event(sample->thing, EVENT_ERROR_IDENTIFIER,
    NULL);
}

```

函数实现：

```

/**
 * @brief trigger a event to post to cloud.
 *
 * @param thing_id, pointer to thing object.
 * @param event_identifer, event identifier to trigger.
 * @param property_identifer, used when trigger event with method "
event.property.post", if set, post specified property, if NULL, post
all.
 *
 * @return 0 when success, -1 when fail.
 */

```

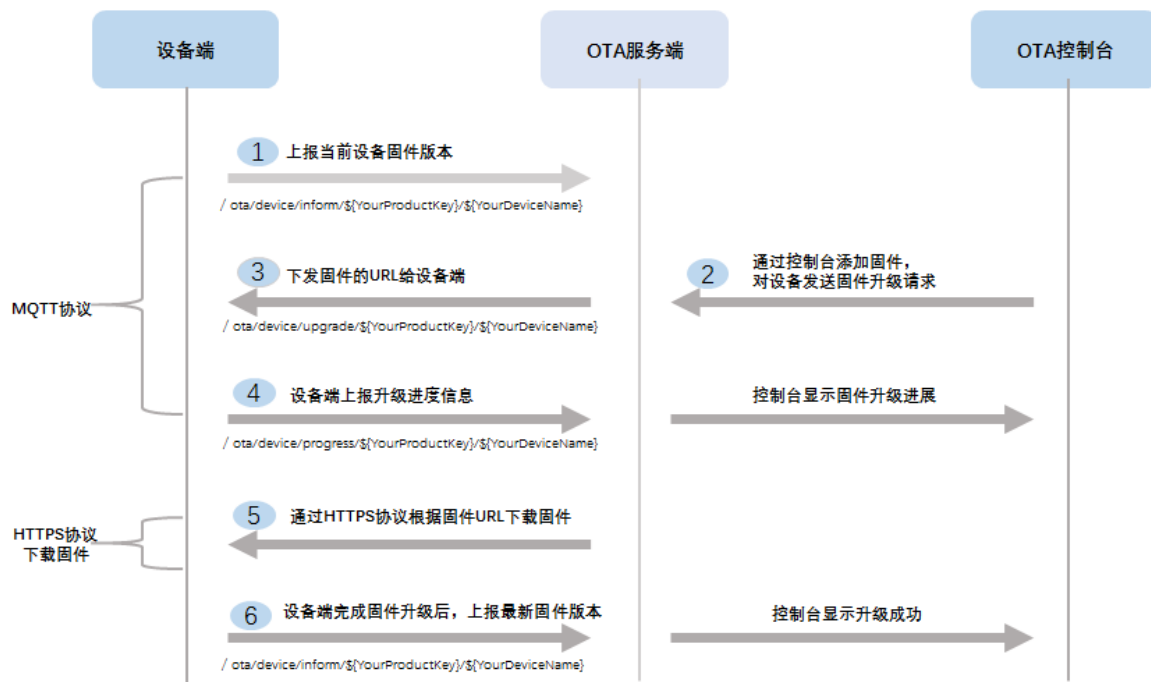
```
extern int linkkit_trigger_event(const void* thing_id, const char*
event_identfier, const char* property_identfier);
```

## 5 设备OTA开发

### 升级流程

以MQTT协议为例，固件升级流程如图 5-1: 固件升级所示。

图 5-1: 固件升级



固件升级Topic：

- 设备端上报固件版本给云端

```
/ota/device/inform/${YourProductKey}/${YourDeviceName}
```

- 设备端订阅该topic接收云端固件升级通知

```
/ota/device/upgrade/${YourProductKey}/${YourDeviceName}
```

- 设备端上报固件升级进度

```
/ota/device/progress/${YourProductKey}/${YourDeviceName}
```



说明：

- 设备固件版本号只需要在系统启动过程中上报一次即可，不需要周期循环上报。
- 根据版本号来判断设备端OTA是否升级成功。

- 从OTA服务端控制台发起批量升级，设备升级操作记录状态是待升级。

实际升级以OTA系统接收到设备上报的升级进度开始，设备升级操作记录状态是升级中。

- 设备离线时，接收不到服务端推送的升级消息。

当设备上线后，主动通知服务端上线消息，OTA服务端收到设备上线消息，验证该设备是否需要升级，如果需要，再次推送升级消息给设备，否则，不推送消息。

## 数据格式说明

1. 设备连接OTA服务，必须上报版本号。

设备端通过MQTT协议pub当前设备固件版本号到Topic：`/ota/device/inform/${YourProductKey}/${YourDeviceName}`，内容格式如下：

```
{
  "id": 1,
  "params": {
    "version": "xxxxxxx"
  }
}
```

- **id**：消息ID号，保留值。
- **version**：设备当前固件版本号。

2. 在物联网平台的控制台上添加固件和对指定版本号进行升级，具体请参考上面流程图。

升级分为几种：验证固件、批量升级、再次升级。

批量升级的前提是：该固件已通过验证，未验证的固件不可以进行批量升级。

- 验证固件：执行该功能后，控制台会返回此次验证的结果，选取的设备升级成功或者失败，升级记录可以在控制台的设备升级列表中查看。
- 批量升级：执行批量升级后，可以在控制台上设备升级列表中查看升级结果。
- 再次升级：该功能主要针对升级失败的设备进行操作，对于待升级，正在升级，升级成功的设备，是无效的，不可操作的。

3. 触发升级操作之后，设备会收到OTA服务推送的固件的URL地址。

设备端订阅Topic：`/ota/device/upgrade/${YourProductKey}/${YourDeviceName}`，控制台对设备发起固件升级请求后，设备端会通过该Topic收到固件的URL，格式如下：

```
{
  "code": "1000",
  "data": {
```

```

    "size": 432945,
    "version": "2.0.0",
    "url": "https://iotx-ota-pre.oss-cn-shanghai.aliyuncs.com/nopoll_0.4.4.tar.gz?Expires=1502955804&OSSAccessKeyId=XXXXXXXXXXXXXXXXXXXX&Signature=XfgJu7P6DWWejstKJgXJEH0qAKU%3D&security-token=CAISuQJlq6Ft5B2yfSjIpK6MGsyN1Jx5jo6mVnfBg1IPTvlvt5D50Tz2IHtIf3NpAusdsv03nWxT7v4flqFyTINVAEvYZJOPKGrGR0DzDbDasumZsJbo4f%2FMQBqEaXPS2MvVfJ%2BzLrf0ceusbFbpjzJ6xaCAGxypQ12iN%2B%2Fr6%2F5gdc9FcQSkL0B8ZrFsKxBltDUROfbIKP%2BpKWSKuGfLCldysQcOlwEP4K%2BkkMqH8Uic3h%2Boy%2BgJt8H2PpHhd9NhXuV2WMzn2%2FdtJOiTknxR7ARasaBqhelc4zqA%2FPPlWgAKvkXba7aIoo0lfV4jN5JXQfAU8KLO8tRjofHWmojNzBJAAPpYSSy3Rvr7m5efQrrybY1lLO6iZy%2BVio2VSZDxshI5Z3McKARWct06MWV9ABA2TTXOii40BOxuq%2B3JGoABXC54TOlo7%2F1wTLTsCUqzzeIiXVOK8CfNOkfTucMGHkeYeCdFkm%2FkADhXAnrnGf5a4FbmKMQph2cKsr8y8UfWLC6IzvJsClXTnbJBMeuWIqo5zIynSlpm7gf%2F9N3hVc6%2BEeIk0xfl2tycsUpbL2FoaGk6BAF8hWSWYUXsv59d5Uk%3D",
    "md5": "93230c3bde425a9d7984a594ac55ea1e"
  },
  "id": 1507707025,
  "message": "success"
}

```

- **size**：固件文件大小。
- **md5**：固件内容的MD5值，32位的hex String。
- **url**：固件的URL，带有时效30分钟，超过这个时间后再下载，服务端拒绝访问。
- **version**：固件的目的版本号。

#### 4. 设备收到URL之后，通过HTTPS协议根据URL下载固件。



说明：

URL是有时效限制的，目前限制时间是30分钟，超过这个时间后，会拒绝访问。

下载固件过程中，设备端必须pub升级进度到Topic：`/ota/device/progress/${YourProductKey}/${YourDeviceName}`，内容格式如下：

```

{
  "id": 1
  "params": {
    "step": "1",
    "desc": " xxxxxxxx "
  }
}

```

- **id**：消息ID号，保留值。
- **step**：[1, 100] 下载进度比。
  - -1：代表升级失败
  - -2：代表下载失败

- -3：代表校验失败
  - -4：代表烧写失败
  - **desc**: 当前步骤的描述信息，如果发生异常可以用此字段承载错误信息。
5. 设备端完成固件升级后，需要pub最新的固件版本到Topic：`/ota/device/inform/${YourProductKey}/${YourDeviceName}`，如果上报的版本与OTA服务要求的版本一致就认为升级成功，反之失败。



说明：

升级成功的唯一判断标志是设备上报正确的版本号。即使升级进度上报为100%，如果不上报新固件版本号，也视为升级失败。

#### 常见下载固件错误

- 签名错误，如果设备端获取的固件的URL不全或者手动修改了URL内容，就会出现如下错误：
- 访问拒绝，URL过期导致，目前过期时间为30分钟。

## OTA代码说明

1. 将固件烧录到设备中，并启动运行设备。

OTA相应的初始代码如下：

```
h_ota = IOT_OTA_Init(PRODUCT_KEY, DEVICE_NAME, pclient);
if (NULL == h_ota) {
    rc = -1;
    printf("initialize OTA failed\n");
}
```



说明：

OTA模块的初始化依赖于MQTT连接，即先获得的MQTT客户端句柄pclient。

函数原型：

```
/**
 * @brief Initialize OTA module, and return handle.
 * The MQTT client must be construct before calling this interface.
 *
 * @param [in] product_key: specify the product key.
 * @param [in] device_name: specify the device name.
```

```

* @param [in] ch_signal: specify the signal channel.
*
* @retval 0 : Successful.
* @retval -1 : Failed.
* @see None.
*/
void *IOT_OTA_Init(const char *product_key, const char *device_name
, void *ch_signal);
/**
* @brief Report firmware version information to OTA server (optional
).
* NOTE: please
*
* @param [in] handle: specify the OTA module.
* @param [in] version: specify the firmware version in string format
.
*
* @retval 0 : Successful.
* @retval < 0 : Failed, the value is error code.
* @see None.
*/
int IOT_OTA_ReportVersion(void *handle, const char *version);

```

## 2. 设备端收到URL之后，通过下载通道下载固件。

- IOT\_OTA\_IsFetching()接口：用于判断是否有固件可下载。
- IOT\_OTA\_FetchYield()接口：用于下载一个固件块。
- IOT\_OTA\_IsFetchFinish()接口：用于判断是否已下载完成。

具体参考代码如下：

```

//判断是否有固件可下载
if (IOT_OTA_IsFetching(h_ota)) {
    unsigned char buf_ota[OTA_BUF_LEN];
    uint32_t len, size_downloaded, size_file;
    do {
        //循环下载固件
        len = IOT_OTA_FetchYield(h_ota, buf_ota, OTA_BUF_LEN, 1);
        if (len > 0) {
            //写入到Flash等存储器中
        }
    } while (!IOT_OTA_IsFetchFinish(h_ota)); //判断固件是否下载完毕
}
exit: Ctrl ↵
/**
* @brief Check whether is on fetching state
*
* @param [in] handle: specify the OTA module.
*
* @retval 1 : Yes.
* @retval 0 : No.
* @see None.
*/
int IOT_OTA_IsFetching(void *handle);
/**
* @brief fetch firmware from remote server with specific timeout
value.

```

```

* NOTE: If you want to download more faster, the bigger 'buf' should
be given.
*
* @param [in] handle: specify the OTA module.
* @param [out] buf: specify the space for storing firmware data.
* @param [in] buf_len: specify the length of 'buf' in bytes.
* @param [in] timeout_s: specify the timeout value in second.
*
* @retval < 0 : Error occur..
* @retval 0 : No any data be downloaded in 'timeout_s' timeout
period.
* @retval (0, len] : The length of data be downloaded in 'timeout_s'
' timeout period in bytes.
* @see None.
*/
int IOT_OTA_FetchYield(void *handle, char *buf, uint32_t buf_len,
uint32_t timeout_s);
/**
* @brief Check whether is on end-of-fetch state.
*
* @param [in] handle: specify the OTA module.
*
* @retval 1 : Yes.
* @retval 0 : False.
* @see None.
*/
int IOT_OTA_IsFetchFinish(void *handle);

```



说明：

一般情况下，由于RAM不足，在下载的同时需要写入到系统OTA区。

### 3. 通过IOT\_OTA\_ReportProgress()接口上报下载状态。

参考代码如下：

```

if (percent - last_percent > 0) {
    IOT_OTA_ReportProgress(h_ota, percent, NULL);
}
IOT_MQTT_Yield(pclient, 100); //

```

升级进度可以上传，升级进度百分比（1%~100%），对应的进度会实时显示在控制台正在升级列表的进度列。

升级进度也可以上传失败码，目前失败码有以下四种：

- -1：升级失败（fail to upgrade）
- -2：下载失败（fail to download）
- -3：校验失败（fail to verify）
- -4：烧写失败（fail to flash）

### 4. 通过IOT\_OTA\_ioctl()接口，获取固件是否合法，硬件系统下次启动时，运行新固件。



参考代码如下：

```
int32_t firmware_valid;
IOT_OTA_Ioctl(h_ota, IOT_OTAG_CHECK_FIRMWARE, &firmware_valid, 4);
if (0 == firmware_valid) {
    printf("The firmware is invalid\n");
} else {
    printf("The firmware is valid\n");
}
```

如果固件合法，则需要通过修改系统启动参数等方式，使硬件系统下一次启动时运行新固件，不同系统的修改方式可能不一样。

```
/**
 * @brief Get OTA information specified by 'type'.
 * By this interface, you can get information like state, size of
 * file, md5 of file, etc.
 *
 * @param [in] handle: handle of the specific OTA
 * @param [in] type: specify what information you want, see detail '
 * IOT_OTA_CmdType_t'
 * @param [out] buf: specify buffer for data exchange
 * @param [in] buf_len: specify the length of 'buf' in byte.
 * @return
 * @verbatim
 * NOTE:
 * 1) When type is IOT_OTAG_FETCHED_SIZE, 'buf' should be pointer of
 * uint32_t, and 'buf_len' should be 4.
 * 2) When type is IOT_OTAG_FILE_SIZE, 'buf' should be pointer of
 * uint32_t, and 'buf_len' should be 4.
 * 3) When type is IOT_OTAG_MD5SUM, 'buf' should be a buffer, and '
 * buf_len' should be 33.
 * 4) When type is IOT_OTAG_VERSION, 'buf' should be a buffer, and '
 * buf_len' should be OTA_VERSION_LEN_MAX.
 * 5) When type is IOT_OTAG_CHECK_FIRMWARE, 'buf' should be pointer of
 * uint32_t, and 'buf_len' should be 4.
 * 0, firmware is invalid; 1, firmware is valid.
 * @endverbatim
 *
 * @retval 0 : Successful.
 * @retval < 0 : Failed, the value is error code.
 * @see None.
 */
int IOT_OTA_Ioctl(void *handle, IOT_OTA_CmdType_t type, void *buf,
size_t buf_len);
```

##### 5. 使用IOT\_OTA\_Deinit销毁连接，释放内存。

```
/**
 * @brief Deinitialize OTA module specified by the 'handle', and
 * release the related resource.
 * You must call this interface to release resource if reboot is not
 * invoked after downloading.
 *
 * @param [in] handle: specify the OTA module.
```

```
*  
* @retval 0 : Successful.  
* @retval < 0 : Failed, the value is error code.  
* @see None.  
*/  
int IOT_OTA_Deinit(void *handle);
```

6. 设备重启时运行新固件，并向云端上报新版本号。在OTA模块初始化之后，调用IOT\_OTA\_ReportVersion()接口上报当前固件的版本号，具体代码如下：

```
if (0 != IOT_OTA_ReportVersion(h_ota, "version2.0")) {  
    rc = -1;  
    printf("report OTA version failed\n");  
}
```

## 6 子设备接入错误码

### 6.1 子设备接入

[网关与子设备](#)，对物联网平台而言，都是单个的设备。两者都可以选择一机一密的认证方式，与云端建立通信。此时需要预烧录两者的三元组信息，包括ProductKey、DeviceName和DeviceSecret。若子设备烧录三元组，比如蓝牙及ZigBee设备，门槛较高，您可以选择动态注册方式认证子设备，此时子设备仅需在云端预注册ProductKey和DeviceName即可。

#### 前提条件

网关已使用[#unique\\_32](#)与云端建立连接。

#### 背景信息

使用动态注册时，子设备需在物联网平台预注册ProductKey和DeviceName。网关代替子设备进行注册时，云端校验子设备DeviceName，校验通过后，云端动态下发DeviceSecret。

以下是具体操作：

#### 操作步骤

1. 以阿里云账号登录[IoT控制台](#)。
2. 设置网关SDK。



#### 说明：

网关可以代理注册子设备、代理子设备上下线、维护主子设备拓扑关系和代理子设备与云端通信。网关厂商要基于该SDK实现应用程序，例如连接子设备、接收子设备消息、发消息到子设备Topic上报云端、订阅子设备Topic获取云端指令和路由消息给子设备等等。

- a) 下载SDK，具体请参见[下载SDK](#)。本文以C-SDK为例。
- b) 登录Linux虚拟机，配置网关三元组。
- c) 开启SDK中关于主子设备功能。

您可以使用`iotx-sdk-c\src\subdev`下的代码进行开发，开发时可参考`sample\subdev`中的Demo。

代码示例中包含三部分：

- 使用subdev的API直接进行开发的示例

```
demo_gateway_function(msg_buf, msg_readbuf);
```

- 使用subdev\_example\_api.h中对topic进行封装的API进行网关开发的示例

```
demo_thing_function(msg_buf, msg_readbuf);
```

- 使用subdev\_example\_api.h中对topic进行封装的API进行单品设备开发的示例

```
demo_only_one_device(msg_buf, msg_readbuf);
```

在网关上添加子设备：

- 如果子设备选择一机一密的认证方式，子设备需在平台预注册并将三元组信息提供给网关，网关通过接口IOT\_Thing\_Register/IOT\_Subdevice\_Register进行注册(IOTX\_Thing\_REGISTER\_TYPE\_STATIC)。
- 如果子设备选择动态注册的认证方式，子设备需在平台预注册，网关通过接口IOT\_Thing\_Register/IOT\_Subdevice\_Register进行动态注册(IOTX\_Thing\_REGISTER\_TYPE\_DYNAMIC)。

具体请参考demo\_gateway\_function 中关于动态注册的范例。

- example/subdev\_example\_api.c/h是对事物三要素property、event和service的topic的封装，使用者可以使用这些API直接进行操作，无需关心具体topic。
- 主子设备功能需要在make.settings中定义FEATURE\_SUBDEVICE\_ENABLED = y。

如果该设备是单品设备，请在make.settings中定义FEATURE\_SUBDEVICE\_STATUS = subdevice。

### 3. 网关设备与云端建立MQTT连接。

### 4. 子设备注册。

网关获得子设备的ProductKey与DeviceName（建议以子设备的唯一标识，例如MAC地址作为子设备的DeviceName），最后通过动态注册的方式从云端获取DeviceSecret。

- 请求Topic: /sys/{gw\_productKey}/{gw\_deviceName}/thing/sub/register

请求格式：

```
{ "id" : 123, "version": "1.0", "params" : [{ "deviceName" : "deviceName1234", "productKey" : "1234556554" }], "method": "thing.sub.register" }
```

- 响应Topic：/sys/{gw\_productKey}/{gw\_deviceName}/thing/sub/register\_reply

响应格式：

```
{ "id": 123, "code": 200, "data": [{ "iotId": "12344", "productKey": "xxx", "deviceName": "xxx", "deviceSecret": "xxx" } ] }
```

- JSON里面的ProductKey和DeviceName，不能和网关的ProductKey和DeviceName相同。
- 设备以QOS0方式发送消息。
- SDK中对应的API是 IOT\_Subdevice\_Register，其中register\_type 支持静态注册和动态注册两种，具体使用以及sign的计算，请参见sample\subdev\subdev-example.c 的实现。
- 如果register\_type是IOTX\_SUBDEV\_REGISTER\_TYPE\_DYNAMIC，调用该API后，可以看到控制台上网关下面新建一个子设备，并且该子设备处理离线状态。
- 如果register\_type是IOTX\_SUBDEV\_REGISTER\_TYPE\_DYNAMIC，请勿多次调用。第二次调用，云端会通知该设备已存在。
- 目前SDK内部实现存在一个限制，动态注册获取到device\_secret，只是写入到设备的全局变量中，并没有做持久化，如果重启该信息就会丢失。

因此，如果需要使用该API，请调整代码iotx-sdk-c\src\subdev\iotx\_subdev\_api.c的iotx\_subdevice\_parse\_register\_reply，将device\_secret写入到具体持久化功能的模块中。

- 如果register\_type是IOTX\_SUBDEV\_REGISTER\_TYPE\_STATIC，调用该API后，控制台上原来的子设备会添加到网关下面，作为网关的一个子设备，并且该子设备处理离线状态。

```
/**
 * @brief Device register
 * This function used to register device and add topo.
 *
 * @param pointer of handle, specify the gateway construction.
 * @param register type.
 * IOTX_SUBDEV_REGISTER_TYPE_STATIC
 * IOTX_SUBDEV_REGISTER_TYPE_DYNAMIC
 * @param product key.
 * @param device name.
 * @param timestamp. [if type = dynamic, must be NULL ]
 * @param client_id. [if type = dynamic, must be NULL ]
 * @param sign. [if type = dynamic, must be NULL ]
 * @param sign_method.
 * IOTX_SUBDEV_SIGN_METHOD_TYPE_SHA
 * IOTX_SUBDEV_SIGN_METHOD_TYPE_MD5
```

```

*
* @return 0, Logout success; -1, Logout fail.
*/
int IOT_Subdevice_Register(void* handle,
iotx_subdev_register_types_t type,
const char* product_key,
const char* device_name,
char* timestamp,
char* client_id,
char* sign,
iotx_subdev_sign_method_types_t sign_type);

```

##### 5. 在云端建立网关与子设备的拓扑关系。

- 添加拓扑关系

— 请求Topic : /sys/{gw\_productKey}/{gw\_deviceName}/thing/topo/add

请求格式 :

```

{
  "id" : "123",
  "version": "1.0",
  "params" : [{
    "deviceName" : "deviceName1234",
    "productKey" : "1234556554",
    "sign": "",
    "signmethod": "hmacSha1" //可支持hmacSha1,hmacSha256,hmacMd5
    "timestamp": "xxxx",
    "clientId": "xxx", //本地标记, 可以和productKey&deviceName保持一致
  }],
  "method": "thing.topo.add"
}

```

— 响应Topic : /sys/{gw\_productKey}/{gw\_deviceName}/thing/topo/add\_reply

请求格式 :

```

{
  "id": "123",
  "code": 200,
  "data": {}
}

```

- JSON里面的ProductKey和DeviceName, 不能和网关的ProductKey和DeviceName相同。
- 设备以QOS0方式发送消息。
- SDK中的这个部分已经封装到IOT\_Subdevice\_Register中, 无需再调用其他API。
- 删除拓扑关系
  - 请求Topic : /sys/{gw\_productKey}/{gw\_deviceName}/thing/topo/delete

请求格式：

```
{
  "id" : 123,
  "version": "1.0",
  "params" : [{
    "deviceName" : "deviceName1234",
    "productKey" : "1234556554"
  }],
  "method": "thing.topo.delete"
}
```

- 响应Topic：/sys/{gw\_productKey}/{gw\_deviceName}/thing/topo/delete\_reply

```
{
  "id":123,
  "code":200,
  "data":{}
}
```

- JSON里面的ProductKey和DeviceName，不能和网关的ProductKey和DeviceName相同。
- 设备以QOS0方式发送消息。
- SDK中的该部分已经封装到IOT\_Subdevice\_Unregister中，无需再调用其他API。
- 获取拓扑关系

— 请求Topic：/sys/{gw\_productKey}/{gw\_deviceName}/thing/topo/get

请求格式：

```
{
  "id" : 123,
  "version": "1.0",
  "params" : {},
  "method": "thing.topo.get"
}
```

— 响应Topic：/sys/{gw\_productKey}/{gw\_deviceName}/thing/topo/get\_reply

```
{
  "id":123,
  "code":200,
  "data": [{
    "deviceName" : "deviceName1234",
    "productKey" : "1234556554"
  }]
}
```

- JSON里面的ProductKey和DeviceName，不能与网关的ProductKey和DeviceName相同。

- 设备以QOS0方式发送消息。
- SDK中对应的API是IOT\_Gateway\_Get\_TOPO。
- 调用该接口后，可以获取网关下面全部子设备的信息，包括三元组信息。参

数get\_topo\_reply是输出结果，JSON格式，需要使用者自行解析。

```
/**
 * @brief Gateway get topo
 * This function publish a packet with topo/get topic and wait
 * for the reply (with TOPO_GET_REPLY topic).
 *
 * @param pointer of handle, specify the Gateway.
 * @param get_topo_reply.
 * @param length [in/out]. in -- get_topo_reply buffer length,
 * out -- reply length
 *
 * @return 0, logout success; -1, logout failed.
 */
int IOT_Gateway_Get_TOPO(void* handle,
char* get_topo_reply,
uint32_t* length);
```

## 6. 子设备上线。

- 请求Topic：/ext/session/{gw\_productKey}/{gw\_deviceName}/combine/login

请求格式：

```
{
  "id": "123",
  "params": {
    "productKey": "xxxxx", //子设备ProductKey
    "deviceName": "xxxx", //子设备DeviceName
    "clientId": "xxxx",
    "timestamp": "xxxx",
    "signMethod": "hmacmd5 or hmacsha1 or hmacsha256",
    "sign": "xxxxx", //子设备签名
    "cleanSession": "true or false" // 如果是true，那么清理所有子设备离线消息，即QoS1或者2的所有未接收内容
  }
}
//对于三元组设备 子设备签名规则同网关相同
sign = hmac_md5(deviceSecret, clientId123deviceNametestproductKey123timestamp123)
```

- 响应Topic：/ext/session/{gw\_productKey}/{gw\_deviceName}/combine/login\_reply

响应格式：

```
{
  "id": "123",
  "code": 200,
```



```
"message": "success"
}
```

- JSON里面的ProductKey和DeviceName，不能和网关的ProductKey和DeviceName相同。
- 设备以QOS0方式发送消息。
- SDK中对应API是IOT\_Subdevice\_Login，具体使用请参考sample\subdev\subdev-example.c实现。
- 调用该接口后，控制台可以看到子设备处于上线状态。

```
/**
 * @brief Subdevice login
 * This function publish a packet with LOGIN topic and wait for the
 * reply (with
 * LOGIN_REPLY topic), then subscribe some subdevice topic.
 *
 * @param pointer of handle, specify the Gateway.
 * @param product key.
 * @param device name.
 * @param timestamp. [if register_type = dynamic, must be NULL ]
 * @param client_id. [if register_type = dynamic, must be NULL ]
 * @param sign. [if register_type = dynamic, must be NULL ]
 * @param sign method, HmacShal or HmacMd5.
 * @param clean session, ture or false.
 *
 * @return 0, login success; -1, login failed.
 */
int IOT_Subdevice_Login(void* handle,
const char* product_key,
const char* device_name,
const char* timestamp,
const char* client_id,
const char* sign,
iotx_subdev_sign_method_types_t sign_method_type,
iotx_subdev_clean_session_types_t clean_session_type);
```

## 7. 子设备数据交互。

- 请求Topic：格式无限制，可以是物联网平台规定的Topic格式 /\${YourProductKey}/\${YourDeviceName}/xxx，也可以是 /sys/\${YourPproductKey}/\${YourDeviceName}/thing/xxx 格式。
- 网关只需要PUB数据到子设备Topic即可，即Topic中的/\${YourPproductKey}/\${YourDeviceName}/是子设备的三元组。
- mqtt payload数据格式无限制。

- SDK中提供了三个API：IOT\_Gateway\_Subscribe，IOT\_Gateway\_Unsubscribe和IOT\_Gateway\_Publish来实现消息的订阅与发布，具体使用可以参考sample\subdev\subdev-example.c 实现。

```
/**
 * @brief Gateway Subscribe
 * This function used to subscribe some topic.
 *
 * @param pointer of handle, specify the Gateway.
 * @param topic list.
 * @param QoS.
 * @param receive data function.
 * @param topic_handle_func's userdata.
 *
 * @return 0, Subscribe success; -1, Subscribe fail.
 */
int IOT_Gateway_Subscribe(void* handle,
const char *topic_filter,
int qos,
iotx_subdev_event_handle_func_fpt topic_handle_func,
void *pcontext);
/**
 * @brief Gateway Unsubscribe
 * This function used to unsubscribe some topic.
 *
 * @param pointer of handle, specify the Gateway.
 * @param topic list.
 *
 * @return 0, Unsubscribe success; -1, Unsubscribe fail.
 */
int IOT_Gateway_Unsubscribe(void* handle, const char* topic_filter);
/**
 * @brief Gateway Publish
 * This function used to Publish some packet.
 *
 * @param pointer of handle, specify the Gateway.
 * @param topic.
 * @param mqtt packet.
 *
 * @return 0, Publish success; -1, Publish fail.
 */
int IOT_Gateway_Publish(void* handle,
const char *topic_name,
iotx_mqtt_topic_info_pt topic_msg);
```

## 8. 子设备下线。

- 请求Topic：/ext/session/{gw\_productKey}/{gw\_deviceName}/combine/logout

请求格式：

```
{
  "id":123,
  "params":{
    "productKey":"xxxxx", //子设备ProductKey
```

```
"deviceName": "xxxxxx", //子设备DeviceName
}
}
```

- 响应Topic: /ext/session/{gw\_productKey}/{gw\_deviceName}/combine/logout\_reply

```
{
  "id": "123",
  "code": 200,
  "message": "success"
}
```

- JSON里面的ProductKey和DeviceName，不能和网关的ProductKey和DeviceName相同
- 设备以QOS0方式发送消息
- SDK中对应的API是IOT\_Subdevice\_Logout。具体使用可以参考sample\subdev\subdev-example.c 实现。
- 调用该接口后，控制台可以看到子设备处理离线状态。

```
/**
 * @brief Subdevice logout
 * This function unsubscribe some subdevice topic, then publish a
 * packet with
 * LOGOUT topic and wait for the reply (with LOGOUT_REPLY topic).
 *
 * @param pointer of handle, specify the Gateway.
 * @param product key.
 * @param device name.
 *
 * @return 0, logout success; -1, logout failed.
 */
int IOT_Subdevice_Logout(void* handle,
const char* product_key,
const char* device_name);
```

## 9. 子设备动态注销。

- 请求Topic : /sys/{gw\_productKey}/{gw\_deviceName}/thing/sub/unregister

请求格式：

```
{
  "id" : 123,
  "version": "1.0",
  "params" : [{
    "deviceName" : "deviceName1234",
    "productKey" : "123456554"
  }],
  "method": "thing.sub.unregister"
```

```
}
```

- 响应Topic: /sys/{gw\_productKey}/{gw\_deviceName}/thing/sub/unregister\_reply

```
{
  "id":123,
  "code":200,
  "data":{}}
}
```

- JSON里面的ProductKey和DeviceName，不能和网关的ProductKey和DeviceName相同。
- 设备以QOS0方式发送消息。
- SDK中对应的API是IOT\_Subdevice\_Unregister。具体使用可以参考sample\subdev\subdev-example.c 实现。
- 调用该接口后，子设备被销毁，该子设备不能再使用，因此请谨慎使用该API。

```
/**
 * @brief Device unregister
 * This function used to delete topo and unregister device.
 * The device must dynamic register again if it want to use after
 * unregister.
 *
 * @param pointer of handle, specify the gateway construction.
 * @param product key.
 * @param device name.
 *
 * @return 0, Unregister success; -1, Unregister fail.
 */
int IOT_Subdevice_Unregister(void* handle,
const char* product_key,
const char* device_name);
```



说明：

- gw\_productKey代表网关的Productkey
- gw\_deviceName代表网关的DeviceName

SDK中还提供其他API，请直接参考subdev-example的实现。

## 6.2 子设备开发错误码

本文汇总了子设备开发时出现的错误码。

### 介绍

- 普通设备直连，当云端发生业务上的错误时，客户端可以通过TCP连接断开感知到。
- 子设备通过网关和服务端通信出现异常，由于网关物理信道仍然保持着，因此必须通过物理通道向客户端发送错误消息，才能使客户端感知到错误。

## 响应格式

子设备和服务端通信异常时，服务端通过网关通道发送一条MQTT错误消息给网关。

topic格式参考以下具体场景，消息内容 JSON格式：

```
{
  id: 子设备请求参数中上报的id
  code: 错误码(成功为200)
  message: 错误信息
}
```

## 子设备上线失败

错误消息发送Topic：/ext/session/{gw\_productKey}/{gw\_deviceName}/combine/login\_reply

表 6-1: 上线失败错误码说明

code	message	备注
460	request parameter error	参数格式错误，比如JSON格式错误，或者其他认证参数错误。
429	too many requests	认证被限流，单个设备认证过于频繁，一分钟内子设备上线次数超过5次会被限流。
428	too many subdevices under gateway	单个网关下子设备数目超过最大值，目前一个网关下子设备最大数量是1500。
6401	topo relation not exist	子设备没有和当前网关添加拓扑关系。
6100	device not found	子设备不存在。
521	device deleted	子设备被删除。
522	device forbidden	子设备已经被禁用。
6287	invalid sign	认证失败，用户名密码错误。
500	server error	云端异常。

## 子设备主动下线异常

发送到Topic：/ext/session/{gw\_productKey}/{gw\_deviceName}/combine/logout\_reply

表 6-2: 主动下线异常

code	message	备注
460	request parameter error	参数格式错误，JSON格式错误，或者参数错误。
520	device no session	子设备会话不存在，子设备已经下线，或者没有上线过。
500	server error	云端处理异常。

## 子设备被踢下线

发送到Topic：/ext/session/{gw\_productKey}/{gw\_deviceName}/combine/logout\_reply

表 6-3: 被踢下线

code	message	备注
427	device connect in elsewhere	设备重复登录，设备在其他地方上线，导致当前连接被断开。
521	device deleted	设备被删除。
522	device forbidden	设备被禁用。

## 子设备发送消息失败

发送到topic：/ext/error/{gw\_productKey}/{gw\_deviceName}

表 6-4: 发送消息失败

code	message	备注
520	device session error	子设备会话错误。 <ul style="list-style-type: none"><li>子设备会话不存在，可能没有connect，也可能已经被下线。</li><li>子设备会话在线，但是并不是通过当前网关会话上线的。</li></ul>

# 7 设备影子

## 7.1 设备影子JSON详解

本文档介绍设备影子的JSON格式表达方法。

设备影子JSON文档示例：

```
{
  "state": {
    "desired": {
      "color": "RED",
      "sequence": [
        "RED",
        "GREEN",
        "BLUE"
      ]
    },
    "reported": {
      "color": "GREEN"
    }
  },
  "metadata": {
    "desired": {
      "color": {
        "timestamp": 1469564492
      },
      "sequence": {
        "timestamp": 1469564492
      }
    },
    "reported": {
      "color": {
        "timestamp": 1469564492
      }
    }
  },
  "timestamp": 1469564492,
  "version": 1
}
```

JSON属性描述，如下表[表 7-1: JSON属性说明](#)所示。

表 7-1: JSON属性说明

属性	描述
<b>desired</b>	设备的预期状态。仅当设备影子文档具有预期状态时，才包含 <b>desired</b> 部分。 应用程序向 <b>desired</b> 部分写入数据，更新事物的状态，而无需直接连接到该设备。
<b>reported</b>	设备的报告状态。设备可以在 <b>reported</b> 部分写入数据，报告其最新状态。

属性	描述
	应用程序可以通过读取该参数值，获取设备的状态。 JSON文档中也可以不包含 <b>reported</b> 部分，没有 <b>reported</b> 部分的文档同样为有效影子JSON文档。
<b>metadata</b>	当用户更新设备状态文档后，设备影子服务会自动更新 <b>metadata</b> 的值。 设备状态的元数据的信息包含以 Epoch 时间表示的每个属性的时间戳，用来获取准确的更新时间。
<b>timestamp</b>	影子文档的最新更新时间。
<b>version</b>	用户主动更新版本号时，设备影子会检查请求中的 <b>version</b> 值是否大于当前版本号。 如果大于当前版本号，则更新设备影子，并将 <b>version</b> 值更新到请求的版本中，反之则会拒绝更新设备影子。 该参数更新后，版本号会递增，用于确保正在更新的文档为最新版本。



#### 说明：

设备影子支持数组。更新数组时必须全量更新，不能只更新数组的某一部分。

更新数组数据示例：

- 初始状态：

```
{
  "reported" : { "colors" : [ "RED", "GREEN", "BLUE" ] }
}
```

- 更新：

```
{
  "reported" : { "colors" : [ "RED" ] }
}
```

- 最终状态：

```
{
  "reported" : { "colors" : [ "RED" ] }
}
```

## 7.2 设备影子数据流

物联网平台为每个设备预定义了两个Topic实现数据流转，定义的Topic都以固定格式呈现。



- Topic : `/shadow/update/${YourProductKey}/${YourDeviceName}`

设备和应用程序发布消息到此Topic。物联网平台收到该Topic的消息后，将消息中的状态更新到设备影子中。

- Topic : `/shadow/get/${YourProductKey}/${YourDeviceName}`

设备影子更新状态到该Topic，设备订阅此Topic的消息。

以产品灯泡1号下某个具体灯泡设备为例，说明设备、设备影子以及应用程序之间的通信。在本示例中，Productkey:10000；DeviceName:lightbulb，设备以QoS=1发布订阅定义的两个Topic。

设备主动上报状态

处理流程图如[图 7-1: 设备主动上报](#)所示。

图 7-1: 设备主动上报

1. 当灯泡lightbulb联网时，设备使用Topic `/shadow/update/10000/lightbulb`上报最新状态到影子。

发送的JSON消息格式：

```
{
  "method": "update",
  "state": {
    "reported": {
      "color": "red"
    }
  },
  "version": 1
}
```

参数说明如[表 7-2: 上报参数说明](#)所示。

表 7-2: 上报参数说明

参数	说明
method	表示设备或者应用程序请求设备影子时的操作类型。 当执行更新操作时，method为必填字段，设置为update。
state	表示设备发送给设备影子的状态信息。 reported为必填字段，状态信息会同步更新到设备影子的reported部分。
version	表示设备影子检查请求中的版本信息。

参数	说明
	只有当新版本大于当前版本时，设备影子才会接受设备端的请求，更新设备影子版本到相应的版本。

2. 当设备影子接受到灯泡上报状态时，成功更新影子文档。

```
{
  "state" : {
    "reported" : {
      "color" : "red"
    }
  },
  "metadata" : {
    "reported" : {
      "color" : {
        "timestamp" : 1469564492
      }
    }
  },
  "timestamp" : 1469564492,
  "version" : 1
}
```

3. 更新设备影子之后，设备影子会返回结果给设备（灯泡），即发送消息到设备订阅的Topic / shadow/get/10000/lightbulb中。

- 若更新成功，发送到该Topic中的消息为：

```
{
  "method": "reply",
  "payload": {
    "status": "success",
    "version": 1
  },
  "timestamp": 1469564576
}
```

- 若更新失败，发送到该Topic中的消息为：

```
{
  "method": "reply",
  "payload": {
    "status": "error",
    "content": {
      "errorcode": "${errorcode}",
      "errormessage": "${errormessage}"
    }
  },
  "timestamp": 1469564576
}
```

错误码说明如表 7-3: 错误码说明所示。

表 7-3: 错误码说明

errorCode	errorMessage
400	不正确的JSON格式
401	影子JSON缺少method信息
402	影子JSON缺少state字段
403	影子JSON version不是数字
404	影子JSON缺少reported字段
405	影子JSON reported属性字段为空
406	影子JSON method是无效的方法
407	影子内容为空
408	影子reported属性个数超过128个
409	影子版本冲突
500	服务端处理异常

应用程序改变设备状态

处理流程图如图 7-2: 应用程序改变设备状态所示。

图 7-2: 应用程序改变设备状态

1. 应用程序下发指令给设备影子，更改灯泡状态。

应用程序发消息到Topic /shadow/update/10000/lightbulb/中，消息码流如下：

```
{
  "method": "update",
  "state": {
    "desired": {
      "color": "green"
    }
  },
  "version": 2
}
```

2. 应用程序发出更新请求，设备影子更新其文档，影子文档更改为:

```
{
```

```
"state" : {
  "reported" : {
    "color" : "red"
  },
  "desired" : {
    "color" : "green"
  }
},
"metadata" : {
  "reported" : {
    "color" : {
      "timestamp" : 1469564492
    }
  },
  "desired" : {
    "color" : {
      "timestamp" : 1469564576
    }
  }
},
"timestamp" : 1469564576,
"version" : 2
}
```

3. 设备影子更新完成后，发送消息到Topic/shadow/get/10000/lightbulb中，返回结果给设备，返回结果由设备影子决定其构成。

```
{
  "method": "control",
  "payload": {
    "status": "success",
    "state": {
      "reported": {
        "color": "red"
      },
      "desired": {
        "color": "green"
      }
    },
    "metadata": {
      "reported": {
        "color": {
          "timestamp": 1469564492
        }
      },
      "desired": {
        "color": {
          "timestamp": 1469564576
        }
      }
    },
    "version": 2,
    "timestamp": 1469564576
  }
}
```

```
}
```

4. 灯泡在线，并且订阅了Topic/shadow/get/10000/lightbulb，就会收到消息，并根据请求文档中**desired**的值更新状态，将灯泡颜色变成绿色。灯泡更新完状态后，上报最新状态到云端

```
{
  method: "update",
  "state": {
    "reported": {
      "color": "green"
    }
  },
  "version": 3
}
```

如果有时间戳判断指令过期，也可以选择不更新。

5. 最新状态上报成功后，设备端发消息到Topic/shadow/update/10000/lightbulb中清空**desired**属性。消息如下：

```
{
  "method": "update",
  "state": {
    "desired": "null"
  },
  "version": 4
}
```

6. 上报状态成功后，设备影子会同步更新，此时的影子文档如下：

```
{
  "state" : {
    "reported" : {
      "color" : "green"
    }
  },
  "metadata" : {
    "reported" : {
      "color" : {
        "timestamp" : 1469564577
      }
    }
  },
  "desired" : {
    "timestamp" : 1469564576
  },
  "version" : 4
}
```

## 设备主动获取设备影子内容

处理流程图如[图 7-3: 获取设备影子内容](#)所示。

图 7-3: 获取设备影子内容

1. 灯泡获取设备影子中保存的灯泡最新状态，发送固定消息到Topic/shadow/update/10000/lightbulb中，具体的消息如下：

```
{
  "method": "get"
}
```

2. 当设备影子收到这条消息时，发送消息到Topic/shadow/get/10000/lightbulb中，灯泡订阅该Topic，获得消息，消息内容如下：

```
{
  "method": "reply",
  "payload": {
    "status": "success",
    "state": {
      "reported": {
        "color": "red"
      },
      "desired": {
        "color": "green"
      }
    },
    "metadata": {
      "reported": {
        "color": {
          "timestamp": 1469564492
        }
      },
      "desired": {
        "color": {
          "timestamp": 1469564492
        }
      }
    },
    "version": 2,
    "timestamp": 1469564576
  }
}
```

## 设备端删除影子属性

处理流程图如[图 7-4: 删除影子属性](#)所示。

图 7-4: 删除影子属性

灯泡要删除设备影子中保存的灯泡某条属性状态，发送删除影子属性的JSON内容到Topic/shadow/update/10000/lightbulb中，具体的请求消息格式见以下示例。

删除属性时，只需要把**method**设置为delete，并且属性的值设置为null即可。

- 删除影子某一属性的JSON格式

```
{
  "method": "delete",
  "state": {
    "reported": {
      "color": "null",
      "temperature": "null"
    }
  },
  "version": 1
}
```

- 删除影子全部属性的JSON格式

```
{
  "method": "delete",
  "state": {
    "reported": "null"
  },
  "version": 1
}
```

## 7.3 设备影子开发

本章节介绍设备、设备影子和应用程序之间的通信。

### 背景信息

设备影子是指通过特别的Topic在云端构建一个设备的影子，设备同步状态至云端。当设备离线时，云端仍可以快速通过影子获取到设备的状态。

### 操作步骤

1. 设备C-SDK提供接口IOT\_Shadow\_Construct，创建设备影子。

函数声明如下：

```
/**
 * @brief Construct the Device Shadow.
 * This function initialize the data structures, establish MQTT
 * connection.
 * and subscribe the topic: "/shadow/get/${YourProduct_key}/${
 * YourDevice_name}".
 *
 * @param [in] pparam: The specific initial parameter.
 * @retval NULL : Construct shadow failed.
```

```

* @retval NOT_NULL : Construct success.
* @see None.
*/
void *IOT_Shadow_Construct(iotx_shadow_para_pt pparam);

```

## 2. 使用IOT\_Shadow\_RegisterAttribute接口，注册设备影子的属性。

函数声明如下：

```

/**
 * @brief Create a data type registered to the server.
 *
 * @param [in] handle: The handle of device shadow.
 * @param [in] pattr: The parameter which registered to the server.
 * @retval SUCCESS_RETURN : Success.
 * @retval other : See iotx_err_t.
 * @see None.
 */
iotx_err_t IOT_Shadow_RegisterAttribute(void *handle, iotx_shadow_attr_pt pattr);

```

## 3. 设备影子在每次开机时，设备C-SDK提供IOT\_Shadow\_Pull接口，从云端同步设备状态。

函数声明如下：

```

/**
 * @brief Synchronize device shadow data from cloud.
 * It is a synchronous interface.
 * @param [in] handle: The handle of device shadow.
 * @retval SUCCESS_RETURN : Success.
 * @retval other : See iotx_err_t.
 * @see None.
 */
iotx_err_t IOT_Shadow_Pull(void *handle);

```

## 4. 当设备端状态发生变化时，设备C-SDK提供接口IOT\_Shadow\_PushFormat\_Init、IOT\_Shadow\_PushFormat\_Add和IOT\_Shadow\_PushFormat\_Finalize接口更新状态，通过接口IOT\_Shadow\_Push将状态同步到云端。

函数声明如下：

```

/**
 * @brief Start a process the structure of the data type format.
 *
 * @param [in] handle: The handle of device shadow.
 * @param [out] pformat: The format struct of device shadow.
 * @param [in] buf: The buf which store device shadow.
 * @param [in] size: Maximum length of device shadow attribute.
 * @retval SUCCESS_RETURN : Success.
 * @retval other : See iotx_err_t.
 * @see None.
 */
iotx_err_t IOT_Shadow_PushFormat_Init(
    void *handle,

```



```

        format_data_pt pformat,
        char *buf,
        uint16_t size);

/**
 * @brief Format the attribute name and value for update.
 *
 * @param [in] handle: The handle of device shaodw.
 * @param [in] pformat: The format struct of device shadow.
 * @param [in] pattr: To have created the data type of the format in
 the add member attributes.
 * @retval SUCCESS_RETURN : Success.
 * @retval other : See iotx_err_t.
 * @see None.
 */
iotx_err_t IOT_Shadow_PushFormat_Add(
        void *handle,
        format_data_pt pformat,
        iotx_shadow_attr_pt pattr);

/**
 * @brief Complete a process the structure of the data type format.
 *
 * @param [in] handle: The handle of device shaodw.
 * @param [in] pformat: The format struct of device shadow.
 * @retval SUCCESS_RETURN : Success.
 * @retval other : See iotx_err_t.
 * @see None.
 */
iotx_err_t IOT_Shadow_PushFormat_Finalize(void *handle, format_data_pt pformat);

```

##### 5. 当设备需要与云端断开连接时，设备C-SDK提供接

口IOT\_Shadow\_DeleteAttribute和IOT\_Shadow\_Destroy，删除云端创建的属性并释放设备影子。

函数声明如下：

```

/**
 * @brief Deconstruct the specific device shadow.
 *
 * @param [in] handle: The handle of device shaodw.
 * @retval SUCCESS_RETURN : Success.
 * @retval other : See iotx_err_t.
 * @see None.
 */
iotx_err_t IOT_Shadow_Destroy(void *handle);

```

## 8 编译与API参考

详细技术说明文档，请访问[官方Wiki](#)。

### 编译配置项说明

请访问[SDK下载](#)，下载最新版本设备端C-SDK文档。

解压之后，打开编译配置文件`make.settings`，根据需要编辑配置项：

```
FEATURE_MQTT_COMM_ENABLED = y # 是否打开MQTT通道的总开关
FEATURE_MQTT_DIRECT = y # 是否打开MQTT直连的分开关
FEATURE_MQTT_DIRECT_NOTLS = n # 是否打开MQTT直连无TLS的分开关
FEATURE_COAP_COMM_ENABLED = y # 是否打开CoAP通道的总开关
FEATURE_HTTP_COMM_ENABLED = y # 是否打开HTTP通道的总开关
FEATURE_SUBDEVICE_ENABLED = n # 是否打开主子设备功能的总开关
FEATURE_SUBDEVICE_STATUS = gateway # 主子设备功能所处的功能状态
FEATURE_CMP_ENABLED = y # 是否打开CMP功能的总开关
FEATURE_CMP_VIA_MQTT_DIRECT = y # CMP功能连云部分是否直接使用MQTT的开关
FEATURE_MQTT_DIRECT_NOITLS = y # 是否打开MQTT直连无ITLS的分开关 目前itls只在id2模式支持
FEATURE_DM_ENABLED = y # 是否打开DM功能的总开关
FEATURE_SERVICE_OTA_ENABLED = y # 是否打开linkit中OTA功能的分开关
FEATURE_SERVICE_COTA_ENABLED = y # 是否打开linkit中远程配置功能的分开关
```

具体含义请参见下表：

表 8-1: `make.settings`参数解释

配置选项	含义
FEATURE_MQTT_COMM_ENABLED	是否使能MQTT通道功能的总开关
FEATURE_MQTT_DIRECT	是否用MQTT直连模式代替HTTPS三方认证模式做设备认证
FEATURE_MQTT_DIRECT_NOTLS	使用MQTT直连模式做设备认证时, 是否要关闭MQTT over TLS
FEATURE_COAP_COMM_ENABLED	是否使能CoAP通道功能的总开关
FEATURE_HTTP_COMM_ENABLED	是否使能Https通道功能的总开关
FEATURE_SUBDEVICE_ENABLED	是否使能主子设备通道功能的总开关
FEATURE_SUBDEVICE_STATUS	主子设备功能所处的功能状态，取值有网关gateway(gw=1)和子设备subdevice(gw=0)

配置选项	含义
FEATURE_CMP_ENABLED	是否打开CMP功能的总开关，CMP: connectivity management platform
FEATURE_CMP_VIA_CLOUD_CONN	CMP功能连云部分选择使用CLOUD_CONN，该开关选择具体协议：MQTT/CoAP/HTTP
FEATURE_MQTT_ID2_AUTH	ID2功能需打开ITLS开关支持
FEATURE_SERVICE_OTA_ENABLED	是否打开linkit中OTA功能的分开关，需打开FEATURE_SERVICE_OTA_ENABLED支持
FEATURE_SERVICE_COTA_ENABLED	是否打开linkit中COTA功能的分开关，需打开FEATURE_SERVICE_OTA_ENABLED支持
FEATURE_SUPPORT_PRODUCT_SECRET	是否打开一型一密开关，与id2互斥



说明：

- 当FEATURE\_SERVICE\_OTA\_ENABLED和FEATURE\_SERVICE\_COTA\_ENABLED的值均为y时，支持远程配置，不支持固件升级。
- 当FEATURE\_SERVICE\_OTA\_ENABLED值为y，FEATURE\_SERVICE\_COTA\_ENABLED的值为n时，支持固件升级，不支持远程配置。
- 当FEATURE\_SERVICE\_OTA\_ENABLED为n时，不支持service层级的ota，但是仍可以直接使用IOT\_OTA\_XXX来进行固件升级。

## 编译 & 运行

请参考[README.md](#)。

## C-SDK API参考

以下将为您介绍V2.0+版本C-SDK提供的功能和对应的API。

API的作用如下：

- 编写应用逻辑，以sample/\*/\*.c代码中的示例程序为准。
- 更加准确和权威的封装AT命令，以src/sdk-impl/iot\_export.h和src/sdk-impl/exports/\*.h代码中的注释为准。

执行如下命令，列出当前SDK代码提供的所有面向用户的API函数。

```
cd src/sdk-impl
```

```
grep -o "IOT_[A-Z][_a-zA-Z]*[^_]\> *(\" iot_export.h exports/*.h|sed 's  
!.*:\\(.*\\)(!\\|!'|cat -n
```

系统显示如下：

```
1 IOT_OpenLog
2 IOT_CloseLog
3 IOT_SetLogLevel
4 IOT_DumpMemoryStats
5 IOT_SetupConnInfo
6 IOT_SetupConnInfoSecure
7 IOT_Cloud_Connection_Init
8 IOT_Cloud_Connection_Deinit
9 IOT_Cloud_Connection_Send_Message
10 IOT_Cloud_Connection_Yield
11 IOT_CMP_Init
12 IOT_CMP_OTA_Start
13 IOT_CMP_OTA_Set_Callback
14 IOT_CMP_OTA_Get_Config
15 IOT_CMP_OTA_Request_Image
16 IOT_CMP_Register
17 IOT_CMP_Unregister
18 IOT_CMP_Send
19 IOT_CMP_Send_Sync
20 IOT_CMP_Yield
21 IOT_CMP_Deinit
22 IOT_CMP_OTA_Yield
23 IOT_CoAP_Init
24 IOT_CoAP_Deinit
25 IOT_CoAP_DeviceNameAuth
26 IOT_CoAP_Yield
27 IOT_CoAP_SendMessage
28 IOT_CoAP_GetMessagePayload
29 IOT_CoAP_GetMessageCode
30 IOT_HTTP_Init
31 IOT_HTTP_DeInit
32 IOT_HTTP_DeviceNameAuth
33 IOT_HTTP_SendMessage
34 IOT_HTTP_Disconnect
35 IOT_MQTT_Construct
36 IOT_MQTT_ConstructSecure
37 IOT_MQTT_Destroy
38 IOT_MQTT_Yield
39 IOT_MQTT_CheckStateNormal
40 IOT_MQTT_Disable_Reconnect
41 IOT_MQTT_Subscribe
42 IOT_MQTT_Unsubscribe
43 IOT_MQTT_Publish
44 IOT_OTA_Init
45 IOT_OTA_Deinit
46 IOT_OTA_ReportVersion
47 IOT_OTA_RequestImage
48 IOT_OTA_ReportProgress
49 IOT_OTA_GetConfig
50 IOT_OTA_IsFetching
51 IOT_OTA_IsFetchFinish
52 IOT_OTA_FetchYield
53 IOT_OTA_Ioctl
54 IOT_OTA_GetLastError
```

```

55 IOT_Shadow_Construct
56 IOT_Shadow_Destroy
57 IOT_Shadow_Yield
58 IOT_Shadow_RegisterAttribute
59 IOT_Shadow_DeleteAttribute
60 IOT_Shadow_PushFormat_Init
61 IOT_Shadow_PushFormat_Add
62 IOT_Shadow_PushFormat_Finalize
63 IOT_Shadow_Push
64 IOT_Shadow_Push_Async
65 IOT_Shadow_Pull
66 IOT_Gateway_Generate_Message_ID
67 IOT_Gateway_Construct
68 IOT_Gateway_Destroy
69 IOT_Subdevice_Register
70 IOT_Subdevice_Unregister
71 IOT_Subdevice_Login
72 IOT_Subdevice_Logout
73 IOT_Gateway_Get_TOPO
74 IOT_Gateway_Get_Config
75 IOT_Gateway_Publish_Found_List
76 IOT_Gateway_Yield
77 IOT_Gateway_Subscribe
78 IOT_Gateway_Unsubscribe
79 IOT_Gateway_Publish
80 IOT_Gateway_RRPC_Register
81 IOT_Gateway_RRPC_Response
82 linkkit_start
83 linkkit_end
84 linkkit_dispatch
85 linkkit_yield
86 linkkit_set_value
87 linkkit_get_value
88 linkkit_set_tsl
89 linkkit_answer_service
90 linkkit_invoke_raw_service
91 linkkit_trigger_event
92 linkkit_fota_init
93 linkkit_invoke_fota_service
94 linkkit_fota_init
95 linkkit_invoke_cota_get_config
96 linkkit_invoke_cota_service
97 linkkit_post_property

```

具体API解释如下表所示。

表 8-2: API功能说明

序号	函数名	说明
必选API		
1	IOT_OpenLog	开始打印日志信息(log), 接受一个 const char *为入参, 表示模块名字
2	IOT_CloseLog	停止打印日志信息(log), 入参为空

序号	函数名	说明
3	IOT_SetLogLevel	设置打印的日志等级, 接受入参从1到5, 数字越大, 打印越详细
4	IOT_DumpMemoryStats	调试函数, 打印内存的使用统计情况, 入参为1-5, 数字越大, 打印越详细
<b>CoAP功能相关</b>		
1	IOT_CoAP_Init	CoAP实例的构造函数, 入参为iotx_coap_config_t结构体, 返回创建的CoAP会话句柄
2	IOT_CoAP_Deinit	CoAP实例的摧毁函数, 入参为IOT_CoAP_Init()所创建的句柄
3	IOT_CoAP_DeviceNameAuth	基于控制台申请的DeviceName, DeviceSecret, ProductKey做设备认证
4	IOT_CoAP_GetMessageCode	CoAP会话阶段, 从服务器的CoAP Response报文中获取Respond Code
5	IOT_CoAP_GetMessagePayload	CoAP会话阶段, 从服务器的CoAP Response报文中获取报文负载
6	IOT_CoAP_SendMessage	CoAP会话阶段, 连接已成功建立后调用, 组织一个完整的CoAP报文向服务器发送
7	IOT_CoAP_Yield	CoAP会话阶段, 连接已成功建立后调用, 检查和收取服务器对CoAP Request的回复报文
<b>云端连接Cloud Connection功能相关</b>		
1	IOT_Cloud_Connection_Init	云端连接实例的构造函数, 入参为iotx_cloud_connection_param_pt结构体, 返回创建的云端连接会话句柄
2	IOT_Cloud_Connection_Deinit	云端连接实例的摧毁函数, 入参为IOT_Cloud_Connection_Init()所创建的句柄
3	IOT_Cloud_Connection_Send_Message	发送数据给云端
4	IOT_Cloud_Connection_Yield	云端连接成功建立后, 收取服务器发送的报文
<b>CMP功能相关</b>		
1	IOT_CMP_Init	CMP实例的构造函数, 入参为iotx_cmp_init_param_pt结构体, 只存在一个CMP实例

序号	函数名	说明
2	IOT_CMP_Register	通过CMP订阅服务
3	IOT_CMP_Unregister	通过CMP取消服务订阅
4	IOT_CMP_Send	通过CMP发送数据，可以送给云端，也可以送给本地设备
5	IOT_CMP_Send_Sync	通过CMP同步发送数据，暂不支持
6	IOT_CMP_Yield	通过CMP接收数据，单线程情况下才支持
7	IOT_CMP_Deinit	CMP示例的摧毁函数
8	IOT_CMP_OTAA_Start	初始化ota功能，上报版本
9	IOT_CMP_OTAA_Set_Callback	设置OTA回调函数
10	IOT_CMP_OTAA_Get_Config	获取远程配置
11	IOT_CMP_OTAA_Request_Image	获取固件
12	IOT_CMP_OTAA_Yield	通过CMP完成OTA功能
<b>MQTT功能相关</b>		
1	IOT_SetupConnInfo	MQTT连接前的准备，基于DeviceName + DeviceSecret + ProductKey产生MQTT连接的用户名和密码等
2	IOT_SetupConnInfoSecure	MQTT连接前的准备，基于ID2 + DeviceSecret + ProductKey产生MQTT连接的用户名和密码等，ID2模式启用
3	IOT_MQTT_CheckStateNormal	MQTT连接后，调用此函数检查长连接是否正常
4	IOT_MQTT_Construct	MQTT实例的构造函数，入参为iotx_mqtt_param_t结构体，连接MQTT服务器，并返回被创建句柄
5	IOT_MQTT_ConstructSecure	MQTT实例的构造函数，入参为iotx_mqtt_param_t结构体，连接MQTT服务器，并返回被创建句柄，ID2模式启用
6	IOT_MQTT_Destroy	MQTT实例的摧毁函数，入参为IOT_MQTT_Construct()创建的句柄

序号	函数名	说明
7	IOT_MQTT_Publish	MQTT会话阶段, 组织一个完整的MQTT Publish报文, 向服务端发送消息发布报文
8	IOT_MQTT_Subscribe	MQTT会话阶段, 组织一个完整的MQTT Subscribe报文, 向服务端发送订阅请求
9	IOT_MQTT_Unsubscribe	MQTT会话阶段, 组织一个完整的MQTT UnSubscribe报文, 向服务端发送取消订阅请求
10	IOT_MQTT_Yield	MQTT会话阶段, MQTT主循环函数, 包含了心跳的维持, 服务器下行报文的收取等
<b>OTA功能相关(模组实现时的可选功能)</b>		
1	IOT_OTA_Init	OTA实例的构造函数, 创建一个OTA会话的句柄并返回
2	IOT_OTA_Deinit	OTA实例的摧毁函数, 销毁所有相关的数据结构
3	IOT_OTA_Ioctl	OTA实例的输入输出函数, 根据不同的命令字可以设置OTA会话的属性, 或者获取OTA会话的状态
4	IOT_OTA_GetLastError	OTA会话阶段, 若某个IOT_OTA_*( )函数返回错误, 调用此接口可获得最近一次的详细错误码
5	IOT_OTA_ReportVersion	OTA会话阶段, 向服务端汇报当前的固件版本号
6	IOT_OTA_FetchYield	OTA下载阶段, 在指定的timeout时间内, 从固件服务器下载一段固件内容, 保存在入参buffer中
7	IOT_OTA_IsFetchFinish	OTA下载阶段, 判断迭代调用IOT_OTA_FetchYield()是否已经下载完所有的固件内容
8	IOT_OTA_IsFetching	OTA下载阶段, 判断固件下载是否仍在进行中, 尚未完成全部固件内容的下载
9	IOT_OTA_ReportProgress	可选API, OTA下载阶段, 调用此函数向服务端汇报已经下载了全部固件内容的百分之多少
10	IOT_OTA_RequestImage	可选API, 向服务端请求固件下载
11	IOT_OTA_GetConfig	可选API, 向服务端请求远程配置
<b>HTTP功能相关</b>		
1	IOT_HTTP_Init	Https实例的构造函数, 创建一个HTTP会话的句柄并返回



序号	函数名	说明
2	IOT_HTTP_DeInit	Https实例的摧毁函数, 销毁所有相关的数据结构
3	IOT_HTTP_DeviceNameAuth	基于控制台申请的DeviceName、DeviceSecret和 ProductKey做设备认证
4	IOT_HTTP_SendMessage	Https会话阶段, 组织一个完整的HTTP报文向服务器发送,并同步获取HTTP回复报文
5	IOT_HTTP_Disconnect	Https会话阶段, 关闭HTTP层面的连接, 但是仍然保持TLS层面的连接
设备影子相关(模组实现时的可选功能)		
1	IOT_Shadow_Construct	建立一个设备影子的MQTT连接, 并返回被创建的会话句柄
2	IOT_Shadow_Destroy	摧毁一个设备影子的MQTT连接, 销毁所有相关的数据结构, 释放内存, 断开连接
3	IOT_Shadow_Pull	把服务器端被缓存的JSON数据下拉到本地, 更新本地的数据属性
4	IOT_Shadow_Push	把本地的数据属性上推到服务器缓存的JSON数据, 更新服务端的数据属性
5	IOT_Shadow_Push_Async	和IOT_Shadow_Push()接口类似, 但是异步的, 上推后便返回, 不等待服务端回应
6	IOT_Shadow_PushFormat_Add	向已创建的数据类型格式中增添成员属性
7	IOT_Shadow_PushFormat_Finalize	完成一个数据类型格式的构造过程
8	IOT_Shadow_PushFormat_Init	开始一个数据类型格式的构造过程
9	IOT_Shadow_RegisterAttribute	创建一个数据类型注册到服务端, 注册时需要*PushFormat*()接口创建的数据类型格式
10	IOT_Shadow_DeleteAttribute	删除一个已被成功注册的数据属性
11	IOT_Shadow_Yield	MQTT的主循环函数, 调用后接受服务端的下推消息, 更新本地的数据属性
主子设备相关(模组实现时的可选功能)		

序号	函数名	说明
1	IOT_Gateway_Construct	建立一个主设备，建立MQTT连接，并返回被创建的会话句柄
2	IOT_Gateway_Destroy	摧毁一个主设备的MQTT连接，销毁所有相关的数据结构，释放内存，断开连接
3	IOT_Subdevice_Login	子设备上线，通知云端建立子设备session
4	IOT_Subdevice_Logout	子设备下线，销毁云端建立子设备session及所有相关的数据结构，释放内存
5	IOT_Gateway_Yield	MQTT的主循环函数，调用后接受服务端的下推消息
6	IOT_Gateway_Subscribe	通过MQTT连接向服务端发送订阅请求
7	IOT_Gateway_Unsubscribe	通过MQTT连接向服务端发送取消订阅请求
8	IOT_Gateway_Publish	通过MQTT连接服务端发送消息发布报文
9	IOT_Gateway_RRPC_Register	注册设备的RRPC回调函数，接收云端发起的RRPC请求
10	IOT_Gateway_RRPC_Response	对云端的RRPC请求进行应答
11	IOT_Gateway_Generate_Message_ID	生成消息id
12	IOT_Gateway_Get_TOPO	向topo/get topic发送包并等待回复 ( TOPIC_GET_REPLY 回复 )
13	IOT_Gateway_Get_Config	向config/get topic发送包并等待回复 ( TOPIC_CONFIG_REPLY 回复 )
14	IOT_Gateway_Publish_Found_List	发现设备列表上报
<b>linkkit()</b>		
1	linkkit_start	启动 linkkit 服务，与云端建立连接并安装回调函数
2	linkkit_end	停止 linkkit 服务，与云端断开连接并回收资源
3	linkkit_dispatch	事件分发函数,触发 linkkit_start 安装的回调

序号	函数名	说明
4	linkkit_yield	linkkit 主循环函数，内含了心跳的维持，服务器下行报文的收取等。如果允许多线程，请不要调用此函数
5	linkkit_set_value	根据identifier设置物对象的 TSL 属性，如果标识符为struct类型、event output类型或者service output类型，使用'.'分隔字段，例如"identifier1.identifier2"指向特定的项
6	linkkit_get_value	根据identifier获取物对象的 TSL 属性
7	linkkit_set_tsl	从本地读取 TSL 文件，生成物的对象并添加到 linkkit 中
8	linkkit_answer_service	对云端服务请求进行回应
9	linkkit_invoke_raw_service	向云端发送裸数据
10	linkkit_trigger_event	上报设备事件到云端
11	linkkit_fota_init	初始化 OTA-fota 服务，并安装回调函数(需编译设置宏 SERVICE_OTA_ENABLED )
12	linkkit_invoke_fota_service	执行fota服务
13	linkkit_cota_init	初始化 OTA-cota 服务，并安装回调函数(需编译设置宏 SERVICE_OTA_ENABLED SERVICE_COTA_ENABLED )
14	linkkit_invoke_cota_get_config	设备请求远程配置
15	linkkit_invoke_cota_service	执行cota服务
16	linkkit_post_property	上报设备属性到云端

## 9 跨平台移植说明

### V2.0+设备端C-SDK移植

主要介绍V2.0+设备端C-SDK实现原理，和按照从下到上的顺序，逐个对每个层次做更加详细的说明。

详细技术文档请访问[官方Wiki](#)，物联网平台会逐渐增加已适配的平台，如果您使用的平台未被适配，请访问[官方Github](#)提出需求。

V2.0+设备端C-SDK概述：

```
+-----+ +-----+
| | | | => 构建完成后产生：
| IoT SDK Example Program | | sample/mqtt|coap|ota/*.c |
| | | | output/release/bin/*-example
+-----+ +-----+
| | | | => SDK提供功能的API，都在这里声明
| IoT SDK Interface Layer | | src/sdk-impl/iot_export.h | => 构建完成后
产生：
| | | |
| IOT_XXX_YYY() APIs | | Has all APIs' prototype | output/release/
include/
| | | | iot-sdk/iot_export.h
| | | | iot-sdk/exports/*.h
+-----+ +-----+
| | | | => SDK提供功能的API，都在这里实现
| | | | src/utills: utilities | => 构建完成后产生：
| | | | +---> | src/log: logging |
| | | | src/tls: security |
| IoT SDK Core Implements | | src/guider: authenticate | output/
release/lib/
| : => | <---+ | src/system: device mgmt | libiot_sdk.a
| : You SHOULD NOT Focus | | src/mqtt: MQTT client | |
| : on this unless | | src/coap: CoAP client |
| : you're debugging bugs | | src/http: HTTP client |
| | | | src/shadow: device shadow |
| | | | src/ota: OTA channel |
| | | |
+-----+ +-----+
| | | | => SDK仅含有示例代码，移植时需二次开发
| Hardware Abstract Layer | | src/sdk-impl/iot_import.h | => 构建完成后
产生：
| | | | : => |
| HAL_XXX_YYY() APIs | | : HAL_*() declarations | output/release/lib/
| | | | libiot_platform.a
| : You MUST Implement | | src/platform/*/*/*.c | output/release/
include/
| : this part for your | | : => | iot-sdk/iot_import.h
| : target device first | | : HAL_*() example impls | iot-sdk/imports/
*.h
```

+-----+ +-----+

2.0版本相对1.0.1版本在结构性方面，升级了编译系统，支持后续功能模块的灵活迭代和裁剪，但是在代码架构方面，和1.0.1版本保持恒定，也是分为三层的：

- 最底层称为硬件平台抽象层，也简称HAL层，对应Hardware Abstract Layer。

抽象不同的嵌入式目标板上，操作系统对SDK的支撑函数，包括网络收发、TLS/DTLS通道建立和读写，内存申请是否和互斥量加锁解锁等。



说明：

- 在任何跨平台移植时，都需要首先实现该层操作。
- 阿里的SDK里，不包含多平台的HAL层实现，提供了Linux OS(Ubuntu16.04)上的实现，移植时可以作为参考。

- 中间层称为SDK内核实现层，对应 IoT SDK Core Implements。

物联网平台C-SDK的核心实现部分，它基于HAL层接口完成了MQTT/CoAP通道等的功能封装，包括MQTT的连接建立、报文收发、CoAP的连接建立、报文收发、OTA的固件状态查询和OTA的固件下载等。



说明：

如果HAL层实现的好，这一层在跨平台移植时，理想情况不需要做任何修改。

- 最上层称为SDK接口声明层，对应IoT SDK Interface Layer。

如何使用这些API做业务逻辑，在sample目录提供了丰富的示例程序，并且只要填入了设备信息，就可以在Linux主机上运行体验。

各层功能实现如下：

- 硬件平台抽象层(HAL层)
  - 在头文件`src/sdk-impl/iot_import.h`中列出所有HAL层函数的声明。
  - 在`src/sdk-impl/imports/iot_import_*.h`中列出各功能点引入的HAL层接口依赖。
  - `src/sdk-impl/iot_import.h`中包含`imports`目录下的所有子文件。
  - 在V2.0+版本的编译系统中，该部分会被编译成`output/release/lib/libiot_platform.a`。

执行如下命令，可以列出所有跨平台移植时需要实现的HAL层接口。

```
src/sdk-impl$ grep -ro "HAL_[_A-Za-z0-9]*" *|cut -d':' -f2|sort -u|cat
-n
```

系统显示如下：

```
1 HAL_DTLSSession_create
2 HAL_DTLSSession_free
3 HAL_DTLSSession_read
4 HAL_DTLSSession_write
5 HAL_Free
6 HAL_GetModuleID
7 HAL_GetPartnerID
8 HAL_Malloc
9 HAL_MutexCreate
10 HAL_MutexDestroy
11 HAL_MutexLock
12 HAL_MutexUnlock
13 HAL_Printf
14 HAL_Random
15 HAL_SleepMs
16 HAL_Snprintf
17 HAL_Srandom
18 HAL_SSL_Destroy
19 HAL_SSL_Establish
20 HAL_SSL_Read
21 HAL_SSL_Write
22 HAL_TCP_Destroy
23 HAL_TCP_Establish
24 HAL_TCP_Read
25 HAL_TCP_Write
26 HAL_UDP_close
27 HAL_UDP_create
28 HAL_UDP_read
29 HAL_UDP_readTimeout
30 HAL_UDP_write
31 HAL_UptimeMs
32 HAL_Vsnprintf
```

函数的实现方法，请参考src/platform下已经写好的示例，该示例在Ubuntu16.04主机和Win7主机上已完整编写和测试过。

```
src/platform$ tree
.
+-- iot.mk
+-- os
|   +-- linux
|   |   +-- HAL_OS_linux.c
|   |   +-- HAL_TCP_linux.c
|   |   +-- HAL_UDP_linux.c
|   +-- ubuntu -> linux
|   +-- win7
|   +-- HAL_OS_win7.c
|   +-- HAL_TCP_win7.c
+-- ssl
+-- mbedtls
|   +-- HAL_DTLS_mbedtls.c
```

```
| +-- HAL_TLS_mbedtls.c
+-- openssl
+-- HAL_TLS_openssl.c
```

函数说明如下表所示，更多详细信息，请查阅代码中的注释，或关注[官方Wiki](#)。

**表 9-1: HAL层接口**

函数名	说明
HAL_DTLSSession_create	初始化DTLS资源并建立一个DTLS会话，用于CoAP功能。
HAL_DTLSSession_free	销毁一个DTLS会话并释放DTLS资源，用于CoAP功能。
HAL_DTLSSession_read	从DTLS会话中读数据，用于CoAP功能。
HAL_DTLSSession_write	向DTLS会话中写数据，用于CoAP功能。
HAL_Free	释放一片堆上内存。
HAL_GetModuleID	用于紧密合作伙伴，可实现为空函数。
HAL_GetPartnerID	用于紧密合作伙伴，可实现为空函数
HAL_Malloc	申请一片堆上内存。
HAL_MutexCreate	创建一个互斥量，用于同步控制。目前SDK仅支持单线程应用，可实现为空函数。
HAL_MutexDestroy	销毁一个互斥量，用于同步控制，目前SDK仅支持单线程应用，可实现为空函数。
HAL_MutexLock	加锁一个互斥量，用于同步控制，目前SDK仅支持单线程应用，可实现为空函数。
HAL_MutexUnlock	解锁一个互斥量，用于同步控制，目前SDK仅支持单线程应用，可实现为空函数。
HAL_Printf	打印函数，用于向串口或其它标准输出打印日志或调试信息。
HAL_Random	随机数函数，接受一个无符号数作为范围，返回0到该数值范围内的随机无符号数。
HAL_SleepMs	睡眠函数，使当前执行线程睡眠指定的毫秒数。
HAL_Snprintf	打印函数，向内存缓冲区格式化构建一个字符串，参考C99标准库函数snprintf。
HAL_Srandom	随机数播种函数，使HAL_Random的返回值每个执行序列各不相同，类似srand。
HAL_SSL_Destroy	销毁一个TLS连接，用于MQTT功能和HTTPS功能。

函数名	说明
HAL_SSL_Establish	建立一个TLS连接，用于MQTT功能和HTTPS功能。
HAL_SSL_Read	从一个TLS连接中读数据，用于MQTT功能和HTTPS功能。
HAL_SSL_Write	向一个TLS连接中写数据，用于MQTT功能和HTTPS功能。
HAL_TCP_Destroy	销毁一个TLS连接，用于MQTT功能和HTTPS功能。
HAL_TCP_Establish	建立一个TCP连接，包含了域名解析的动作和TCP连接的建立。
HAL_TCP_Read	在指定时间内，从TCP连接读取流数据，并返回读到的字节数。
HAL_TCP_Write	在指定时间内，向TCP连接发送流数据，并返回发送的字节数。
HAL_UDP_close	关闭一个UDP socket。
HAL_UDP_create	创建一个UDP socket。
HAL_UDP_read	阻塞的从一个UDP socket中读取数据包，并返回读到的字节数。
HAL_UDP_readTimeout	在指定时间内，从一个UDP socket中读取数据包，返回读到的字节数。
HAL_UDP_write	阻塞的向一个UDP socket中发送数据包，并返回发送的字节数。
HAL_UptimeMs	时钟函数，获取本设备从加电以来到目前时间点已经过去的毫秒数。
HAL_Vsnprintf	字符串打印函数，将va_list类型的变量，打印到指定目标字符串。

具体实现如下表所示。

**表 9-2: HAL接口实现**

函数名	说明
必须实现	
HAL_Malloc	申请一片堆上内存。
HAL_Free	释放一片堆上内存。
HAL_SleepMs	睡眠函数，使当前执行线程睡眠指定的毫秒数。



函数名	说明
HAL_Snprintf	打印函数，向内存缓冲区格式化构建一个字符串，参考C99标准库函数snprintf。
HAL_Printf	打印函数，用于向串口或其它标准输出打印日志或调试信息。
HAL_Vsnprintf	字符串打印函数，将va_list类型的变量，打印到指定目标字符串。
HAL_UptimeMs	时钟函数，获取本设备从加电以来到目前时间点已经过去的毫秒数。
可实现为空	
HAL_GetPartnerID	用于紧密合作伙伴，可实现为空函数。
HAL_GetModuleID	用于紧密合作伙伴，可实现为空函数。
HAL_MutexCreate	创建一个互斥量，用于同步控制，目前SDK仅支持单线程应用，可实现为空函数。
HAL_MutexDestroy	销毁一个互斥量，用于同步控制，目前SDK仅支持单线程应用，可实现为空函数。
HAL_MutexLock	加锁一个互斥量，用于同步控制，目前SDK仅支持单线程应用，可实现为空函数。
HAL_MutexUnlock	解锁一个互斥量，用于同步控制，目前SDK仅支持单线程应用，可实现为空函数。
没有MQTT时可实现为空	
HAL_SSL_Destroy	销毁一个TLS连接，用于MQTT功能和HTTPS功能。
HAL_SSL_Establish	建立一个TLS连接，用于MQTT功能和HTTPS功能。
HAL_SSL_Read	从一个TLS连接中读数据，用于MQTT功能和HTTPS功能。
HAL_SSL_Write	向一个TLS连接中写数据，用于MQTT功能和HTTPS功能。
HAL_TCP_Destroy	销毁一个TLS连接，用于MQTT功能和HTTPS功能。
HAL_TCP_Establish	建立一个TCP连接，包含了域名解析的动作和TCP连接的建立。
HAL_TCP_Read	在指定时间内，从TCP连接读取流数据，并返回读到的字节数。
HAL_TCP_Write	在指定时间内，向TCP连接发送流数据，并返回发送的字节数。
HAL_Random	随机数函数，接受一个无符号数作为范围，返回0到该数值范围内的随机无符号数。

函数名	说明
HAL_Srandom	随机数播种函数，使HAL_Random的返回值每个执行序列各不相同，类似srand。
没有CoAP时可实现为空	
HAL_DTLSSession_create	初始化DTLS资源并建立一个DTLS会话，用于CoAP功能。
HAL_DTLSSession_free	销毁一个DTLS会话并释放DTLS资源，用于CoAP功能。
HAL_DTLSSession_read	从DTLS连接中读数据，用于CoAP功能。
HAL_DTLSSession_write	向DTLS连接中写数据，用于CoAP功能。
没有ID2时可实现为空	
HAL_UDP_close	关闭一个UDP socket。
HAL_UDP_create	创建一个UDP socket。
HAL_UDP_read	阻塞的从一个UDP socket中读取数据包，并返回读到的字节数。
HAL_UDP_readTimeout	在指定时间内，从一个UDP socket中读取数据包，返回读到的字节数。
HAL_UDP_write	阻塞的向一个UDP socket中发送数据包，并返回发送的字节数。

- SDK内核实现层
  - 在src/sdk-impl/iot\_export.h头文件中列出所有被提供的函数的声明。
  - 在src/sdk-impl/exports/iot\_export\_\*.h中列出各功能点提供的接口。
  - 在src/sdk-impl/iot\_export.h下包含exports目录下的子文件。
  - 在V2.0+版本的编译系统中, 该部分会被编译成output/release/lib/libiot\_sdk.a。
- SDK接口声明层 + 例程

请参考[快速入门](#)和[官方SDK首页](#)。

### V1.0.1设备端C-SDK移植

详细描述如何将华东2节点设备端V1.0.1版本C-SDK移植到目标硬件平台。

SDK基本框架如下图所示。

图 9-1: 基本框架

- SDK可分为硬件抽象层、SDK内核代码和面向应用的API。
- 在移植到目标硬件平台时，需要根据硬件平台的情况实现硬件平台抽象接口。
- 在硬件平台抽象层中，包含OS层、network层、ssl层等3类。
  - OS层主要包括时间、互斥锁以及其它等接口，在目录`$(SDK_PATH)/src/platform/os` /下。`$(SDK_PATH)`为您下载SDK的路径。
  - network层主要包括网络相关的接口，目前为TCP接口，在目录`$(SDK_PATH)/src/platform/network` /下。
  - ssl层包含ssl或tls相关接口，在目录`$(SDK_PATH)/src/platform/ssl` /下。

硬件平台抽象层包含数据类型、OS（或硬件）接口、TCPIP网络接口和SSL（TLS）接口等4个部分。下面分别对这4部分进行叙述。

- 数据类型

表 9-3: 数据类型

序号	数据类型名称	说明
自定义数据类型		
1	bool	bool类型
2	int8_t	8比特有符号整型
3	uint8_t	8比特无符号整型
4	int16_t	16比特有符号整型
5	uint16_t	16比特无符号整型
6	int32_t	32比特有符号整型
7	uint32_t	32比特无符号整型
8	int64_t	64比特有符号整型
9	uint64_t	64比特无符号整型
10	uintptr_t	能够容纳指针类型长度的无符号整型
11	intptr_t	能够容纳指针类型长度的有符号整型
自定义关键字		
1	true	bool值：true，如果目标平台无此定义，可宏定义： <code>#define true (1)</code>

序号	数据类型名称	说明
2	false	bool值：false，如果目标平台无此定义，可宏定义：#define false (0)

- 此部分定义在源文件中：`$(SDK_PATH)/src/platform/os/aliot_platform_datatype.h`。
- 请根据目标平台情况实现，请实现在源文件`$(SDK_PATH)/src/platform/aliot_platform_datatype.h`中。



说明：

SDK所定义的数据类型是C99标准所定义的数据类型的一部分，如果目标硬件平台完全支持C99标准，则无需修改此部分代码即可满足于目标平台。

- OS（硬件）接口

表 9-4: OS相关接口说明

序号	接口名称	说明
1	aliot_platform_malloc	分配内存块
2	aliot_platform_free	释放内存块
3	aliot_platform_time_get_ms	获取系统时间（单位：ms），允许溢出
4	aliot_platform_printf	格式化输出
5	aliot_platform_ota_start	启动OTA，由于暂不支持OTA功能，该接口暂无需实现
6	aliot_platform_ota_write	写OTA固件，由于暂不支持OTA功能，该接口暂无需实现
7	al_platform_ota_finalize	完成OTA，由于暂不支持OTA功能，该接口暂无需实现
8	aliot_platform_msleep	睡眠指定时间，如果是无OS的平台，将函数实现为延时指定时间即可
9	aliot_platform_mutex_create	创建互斥锁，如果是无OS的平台，无需实现该接口
10	aliot_platform_mutex_destroy	销毁互斥锁，如果是无OS的平台，无需实现该接口

序号	接口名称	说明
11	alioth_platform_mutex_lock	锁住指定互斥锁，如果是无OS的平台，无需实现该接口
12	alioth_platform_mutex_unlock	释放指定互斥锁，如果是无OS的平台，无需实现该接口
13	alioth_platform_module_get_pid	该接口仅用于特定场景，若无涉及，返回NULL即可

- 详细的接口输入输出说明请参考源文件：( \$(SDK\_PATH)/src/platform/os/alioth\_platform\_os.h )。
- 实现时，请在路径\$(SDK\_PATH)/src/platform/os/下创建一个文件夹（ 请注意这个文件夹名，后续编译将用到这个名字 ），相应移植实现放置在该文件夹下。



说明：

如果是无OS平台，所有面向应用的接口都不能被并发调用，包括在中断服务程序中调用。

- TCPIP网络接口

表 9-5: TCPIP网络接口

序号	接口名称	说明
1	alioth_platform_tcp_establish	建立tcp连接，返回连接句柄
2	alioth_platform_tcp_destroy	释放一个tcp连接
3	alioth_platform_tcp_write	往TCP通道写入数据。注意实现超时参数
4	alioth_platform_tcp_read	从TCP通道读取数据。注意实现超时参数

- 详细的接口输入输出说明请参考源文件：( \$(SDK\_PATH)/src/platform/network/alioth\_platform\_network.h )。
- 实现时，请在路径\$(SDK\_PATH)/src/platform/network/下创建一个文件夹，移植实现放置在该文件夹下。



说明：

请保存该文件夹名，用于后续编译。

- SSL接口

表 9-6: SSL相关接口说明

序号	接口名称	说明
1	alioth_platform_ssl_establish	建立经SSL加密的传输通道
2	alioth_platform_ssl_destroy	释放一个SSL通道
3	alioth_platform_ssl_write	往SSL通道写入数据。注意实现超时参数
4	alioth_platform_ssl_read	从SSL通道读取数据。注意实现超时参数

- 详细的接口输入输出说明，请参考源文件：( \$(SDK\_PATH)/src/platform/ssl/alioth\_platform\_ssl.h )。
- 实现时，请在路径\$(SDK\_PATH)/src/platform/ssl/下创建一个文件夹，相应移植实现存放在该文件夹下。



说明：

请保存该文件夹名，用于后续编译。

## 10 Android-SDK

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。



说明：

Android SDK于2018年9月1日发布新版。旧版用户可参考[旧版文档](#)。

### 工程配置

1. 在Android工程根目录下的build.gradle 基础配置文件中，加入阿里云仓库地址，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SDK的依赖，引入 SDK :iot-linkkit。

```
compile('com.aliyun.alink.linksdk:iot-linkkit:1.3.0.1')
```

### 初始化

参考如下代码初始化设备信息，建立设备与云端的连接。

设备初始化参数包括：

- 设备证书信息
  - 若使用[#unique\\_32](#)认证方式，需要有ProductKey、DeviceName和DeviceSecret。
  - 若使用[#unique\\_43](#)认证方式，需要有ProductKey、ProductSecret和DeviceName，并在控制台开启动态注册。
- 云端接口请求域名

请参考[地域和可用区](#)查看支持的域名。
- 设备属性的初始值

仅在使用[物模型](#)时，才需设置此项。

```
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();
// userData.domain 默认值是 "iot.cn-shanghai.aliyuncs.com"
// 如果需要修改请按如下方式设置
// userData.domain = "xxxx";

// 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();
hubApiRequest.path = "/auth/register/device";
LinkKit.getInstance().deviceRegister(context, params, hubApiRequest,
new IConnectSuccessListener() {
    @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 ILinkKitConnectListe
ner() {
    @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
    }
});

```

## 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。

```

// 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 IConnectNotifyListen
er() {
    @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
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
    }
}

```

```
});
```

## 日志

打开SDK内部日志输出开关：

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

## 初始化结果

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](#)。

## 物模型数据上下行通道

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）。



说明：

`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 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 IResRequestHandler mCommonHandler = new IResRequestHandler
() {
    @Override
```

```

    public void onProcess(String identify, Object result, IResResponseCallback 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 仅供参考
            request.payloadObj = "{\"id\":\"" + resId + "\", \"code\": \"200\", \"data\": {}}";
            LinkKit.getInstance().publish(request, new IConnectSendListener() {
                @Override
                public void onResponse(ARequest aRequest, AResponse aResponse) {
                    // 上报结果
                }
            });
        }
    }
});

```

```

        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
connectState) {
    }
}
})

```

## 基础数据上下行通道

Android SDK 提供了与云端长链接的基础能力接口，用户可以直接使用这些接口完成自定义 Topic 相关的功能。提供的基础能力包括：发布、订阅、取消订阅、RRPC、订阅下行。如果不想使用物模型，可以通过这部分接口实现云端数据的上下行。

### 上行接口请求

调用上行请求接口，SDK封装了上行发布、订阅和取消订阅等接口。

```

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

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

/**
 * 取消订阅
 *
 * @param request 取消订阅请求
 * @param listener 监听器
 */

```

```
void unsubscribe(ARequest request, IConnectUnscribeListener listener);
```

调用示例：

```
// 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 IConnectSubscribeListener() {
    @Override
    public void onSuccess() {
        // 订阅成功
    }

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

// 取消订阅
MqttSubscribeRequest unsubRequest = new MqttSubscribeRequest();
unsubRequest.topic = unSubTopic;
unsubRequest.isSubscribe = false;
LinkKit.getInstance().unsubscribe(unsubRequest, new IConnectUnsubscribeListener() {
    @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
connectState) {
        // 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 MqttRrpcRe
gisterRequest();
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 IConnectRr
pcListener() {
    @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, IConnectRpcHandle
iConnectRpcHandle) {
        // 收到云端下行
        AResponse response = new AResponse();
        response.data = responseData;
        iConnectRpcHandle.onRpcResponse(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.**{*;}

```

## Android SDK Demo

单击下载[Android SDK Demo](#)。下载本Demo将默认您同意[本软件许可协议](#)。



## 11 JAVA-SDK

本章节以Java版SDK为例，介绍如何让设备通过MQTT协议连接到阿里云物联网平台。

### 前提条件

此Demo为maven工程，请先安装maven。

### 背景信息

本Demo并不适合Android特性，如果您是Android，请参考开源库 <https://github.com/eclipse/paho.mqtt.android>。

### 操作步骤

1. 下载mqttClient SDK，地址为[iotx-sdk-mqtt-java](#)。
2. 使用idea或者eclipse，导入该Demo到工程里面。
3. 登录阿里云物联网平台的控制台，选择设备管理，单击设备后的查看，获取三元组信息，即ProductKey、DeviceName、DeviceSecret的值。
4. 修改SimpleClient4IOT.java配置文件，修改完成后，直接运行。
  - a) 配置参数。

```
/** 从控制台获取productKey、deviceName、deviceSecret信息*/
private static String productKey = "";
private static String deviceName = "";
private static String deviceSecret = "";
/** 用于测试的topic */
private static String subTopic = "/" + productKey + "/" + deviceName + "/get";
private static String pubTopic = "/" + productKey + "/" + deviceName + "/pub";
```

- b) 连接MQTT服务器。

```
//客户端设备 自己的一个标记 建议是mac或sn，不能为空，32字符内
String clientId = InetAddress.getLocalHost().getHostAddress();
//设备认证
Map params = new HashMap();
params.put("productKey", productKey); //这个是对应用户在控制台注册的 设备productkey
params.put("deviceName", deviceName); //这个是对应用户在控制台注册的 设备name
params.put("clientId", clientId);
String t = System.currentTimeMillis() + "";
params.put("timestamp", t);
//mqtt服务器，tls的话ssl开头，tcp的话改成tcp开头
```

```
String targetServer = "ssl://" + productKey + ".iot-as-mqtt.cn-shanghai.aliyuncs.com:1883";
//客户端ID格式:
String mqttClientId = clientId + "|securemode=2,signmethod=hmacsha1,timestamp="+t+"|"; //设备端自定义的标记, 字符范围[0-9][a-z][A-Z], [MQTT-TCP连接通信](https://help.aliyun.com/document_detail/30539.html?spm=a2c4g.11186623.6.592.R3LqNT "MQTT-TCP连接通信")
String mqttUsername = deviceName+"&" + productKey; //mqtt用户名格式
String mqttPassword = SignUtil.sign(params, deviceSecret, "hmacsha1");//签名
//连接mqtt的代码片段
MqttClient sampleClient = new MqttClient(url, mqttClientId, persistence);
MqttConnectOptions connOpts = new MqttConnectOptions();
connOpts.setMqttVersion(4); // MQTT 3.1.1
connOpts.setSocketFactory(socketFactory);
//设置是否自动重连
connOpts.setAutomaticReconnect(true);
//如果是true 那么清理所有离线消息, 即qos1 或者 2的所有未接收内容
connOpts.setCleanSession(false);
connOpts.setUsername(mqttUsername);
connOpts.setPassword(mqttPassword.toCharArray());
connOpts.setKeepAliveInterval(80); //心跳时间 建议60s以上
sampleClient.connect(connOpts);
```

c) 数据发送。

```
String content = "要发送的数据内容, 这个内容可以是任意格式";
MqttMessage message = new MqttMessage(content.getBytes("utf-8"));
message.setQos(0); //消息qos 0:最多一次, 1:至少一次
sampleClient.publish(topic, message); //发送数据到某个topic
```

d) 数据接收。

```
//订阅某个topic, 一旦有数据会回调到这里
sampleClient.subscribe(topic, new IMqttMessageListener() {
@Override
public void messageArrived(String topic, MqttMessage message)
throws Exception {
//设备订阅topic成功后, 数据会回调这里
//重复订阅无影响
}
});
```



说明：

MQTT连接参数详细说明, 请参见[MQTT-TCP连接通信](#)。

## 12 基于Alink协议开发

### 12.1 Alink协议

物联网平台为设备端开发提供了SDK，这些SDK已封装了设备端与云端的交互协议。您可以直接使用设备端SDK来进行开发。如果嵌入式环境复杂，已提供的设备端SDK不能满足您的需求，请参考本文，自行封装Alink协议数据，建立设备与云端的通信。

Alink协议是针对物联网开发领域设计的一种数据交换规范，数据格式是JSON，用于设备端和云端的双向通信，更便捷地实现和规范了设备端和云端之间的业务数据交互。

以下为您介绍Alink协议下，设备的上线流程和数据上下行原理。

#### 上线流程

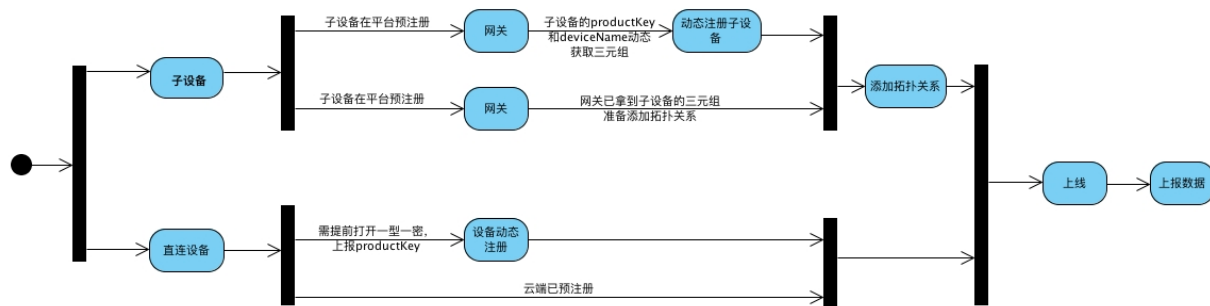
设备上线流程，可以按照设备类型，分为直连设备接入与子设备接入。主要包括：设备注册、上线和数据上报三个流程。

直连设备接入有两种方式：

- 使用**一机一密**方式提前烧录三元组，注册设备，上线，然后上报数据。
- 使用**一型一密**动态注册提前烧录产品证书（ProductKey和ProductSecret），注册设备，上线，然后上报数据。

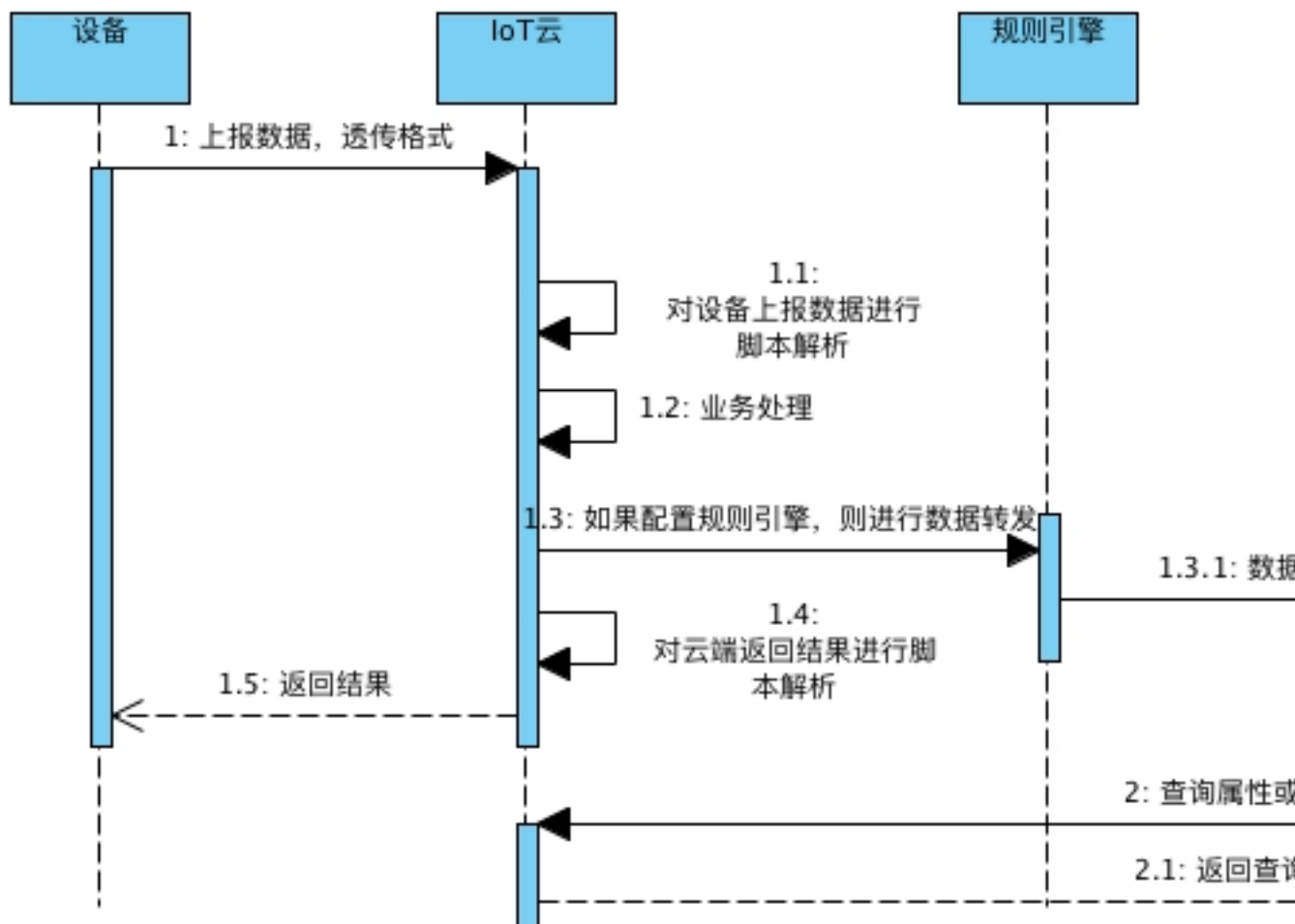
子设备接入流程通过网关发起，具体接入方式有两种：

- 使用**一机一密**提前烧录三元组，子设备上报三元组给网关，网关添加拓扑关系，复用网关的通道上报数据。
- 使用动态注册方式提前烧录ProductKey，子设备上报ProductKey和DeviceName给网关，云端校验DeviceName成功后，下发DeviceSecret。子设备将获得的三元组信息上报网关，网关添加拓扑关系，通过网关的通道上报数据。



## 设备上报属性或事件

- 透传格式（透传/自定义）数据

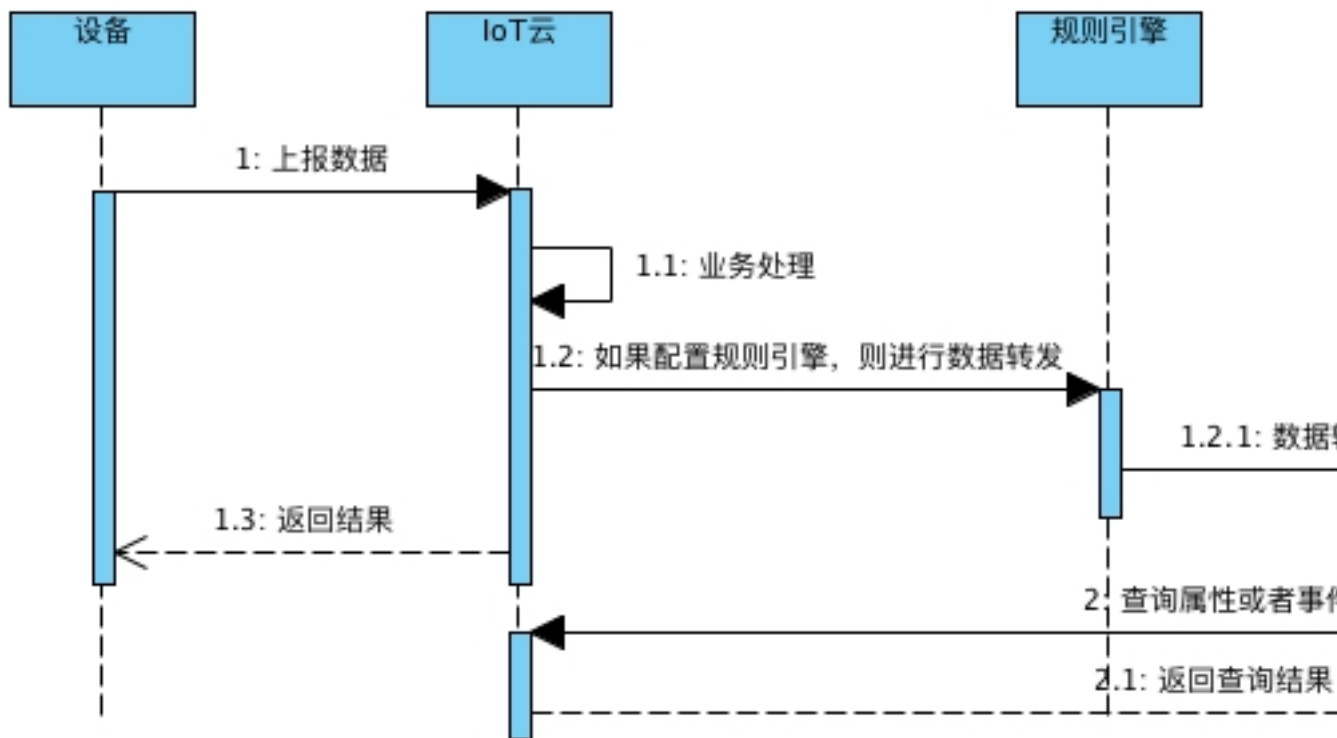


- 使用透传格式的Topic，设备上报透传数据。
- 云端通过脚本先对设备上报的数据进行解析。调用脚本中的rawDataToProtocol方法将设备上报的数据转换为IoT平台标准数据格式（Alink JSON格式）。
- 使用转换后的Alink JSON格式数据进行业务处理。  
如果配置了规则引擎，则通过规则引擎将数据输出到规则引擎配置的目的云产品中。
- 对于云端返回的结果，通过脚本进行解析。
- 转换后的返回结果推送给设备。
- 开发者可以通过QueryDevicePropertyData接口查询设备上报的属性历史数据，通过QueryDeviceEventData接口查询设备上报的事件历史数据。



说明：

- 规则引擎获取到的数据是经过脚本解析之后的数据。
  - 可以在规则引擎中使用Topic：`/sys/{productKey}/{deviceName}/thing/event/property/post`，获取设备属性
  - 可以在规则引擎中使用Topic：`/sys/{productKey}/{deviceName}/thing/event/{tsl.event.identifier}/post`，获取设备事件。
- 非透传格式（Alink JSON）数据



1. 使用非透传格式的Topic，设备上报数据。

2. 云端进行业务处理。

如果配置了规则引擎，则通过规则引擎将数据输出到规则引擎配置的目的云产品中。

3. 云端返回结果。

4. 开发者可以通过`QueryDevicePropertyData`接口查询设备上报的属性历史数据，通过`QueryDeviceEventData`接口查询设备上报的事件历史数据。



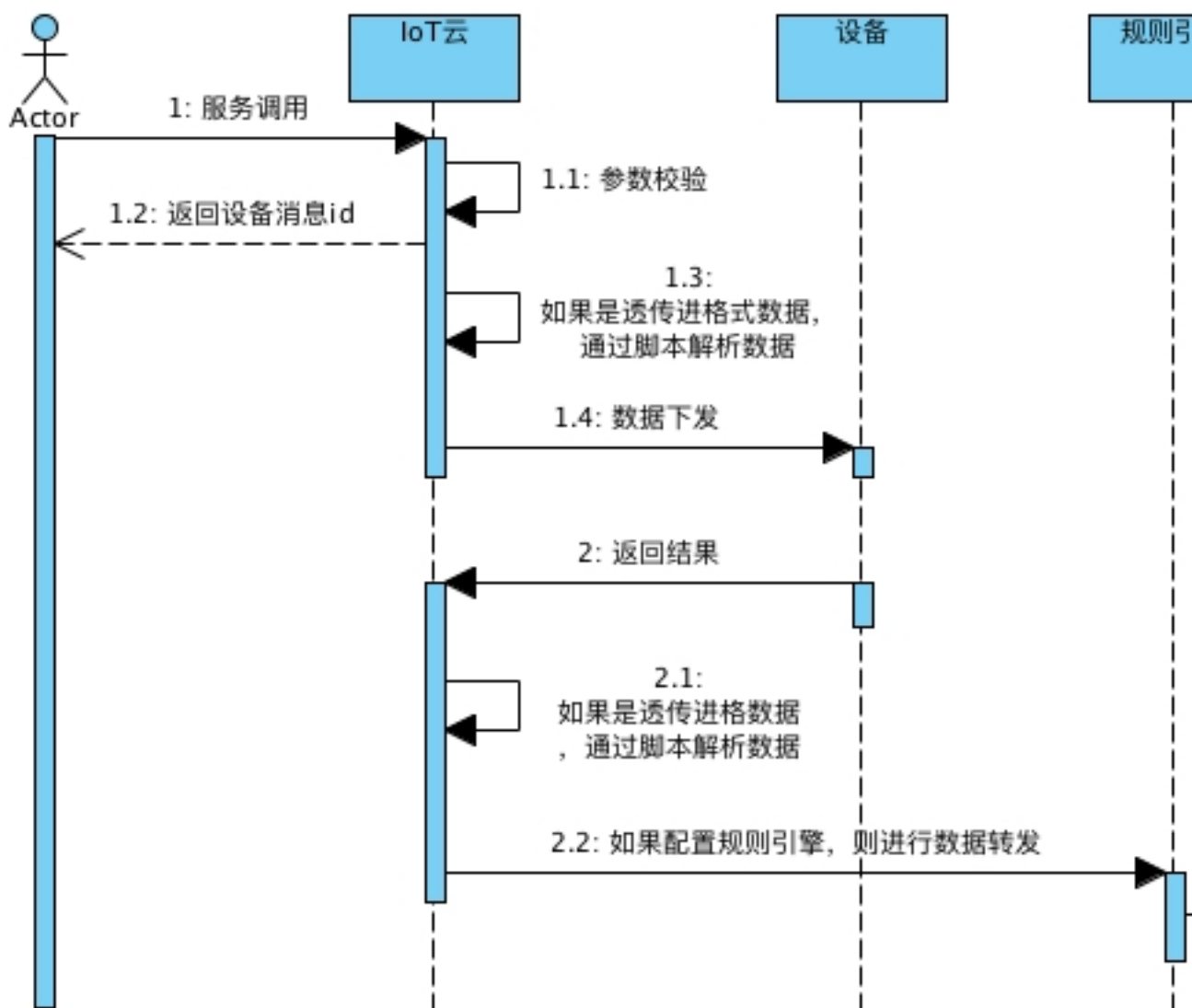
说明：

- 可以在规则引擎中使用Topic：`/sys/{productKey}/{deviceName}/thing/event/property/post`，获取设备属性。

- 可以在规则引擎中使用Topic：`/sys/{productKey}/{deviceName}/thing/event/{tsl.event.identifier}/post`，获取设备事件。

### 设备服务调用或属性设置

- 异步服务调用或属性设置



1. 设置设备属性或者异步调用服务。
2. 对参数进行校验。
3. IoT采用异步调用进行业务处理并返回调用结果，若没有报错，则结果中携带下发给设备的信息id。

如果是透传格式（透传/自定义）数据，通过脚本解析，调用脚本中的`protocolToRawData`方法，对下发给设备的数据进行数据格式转换。

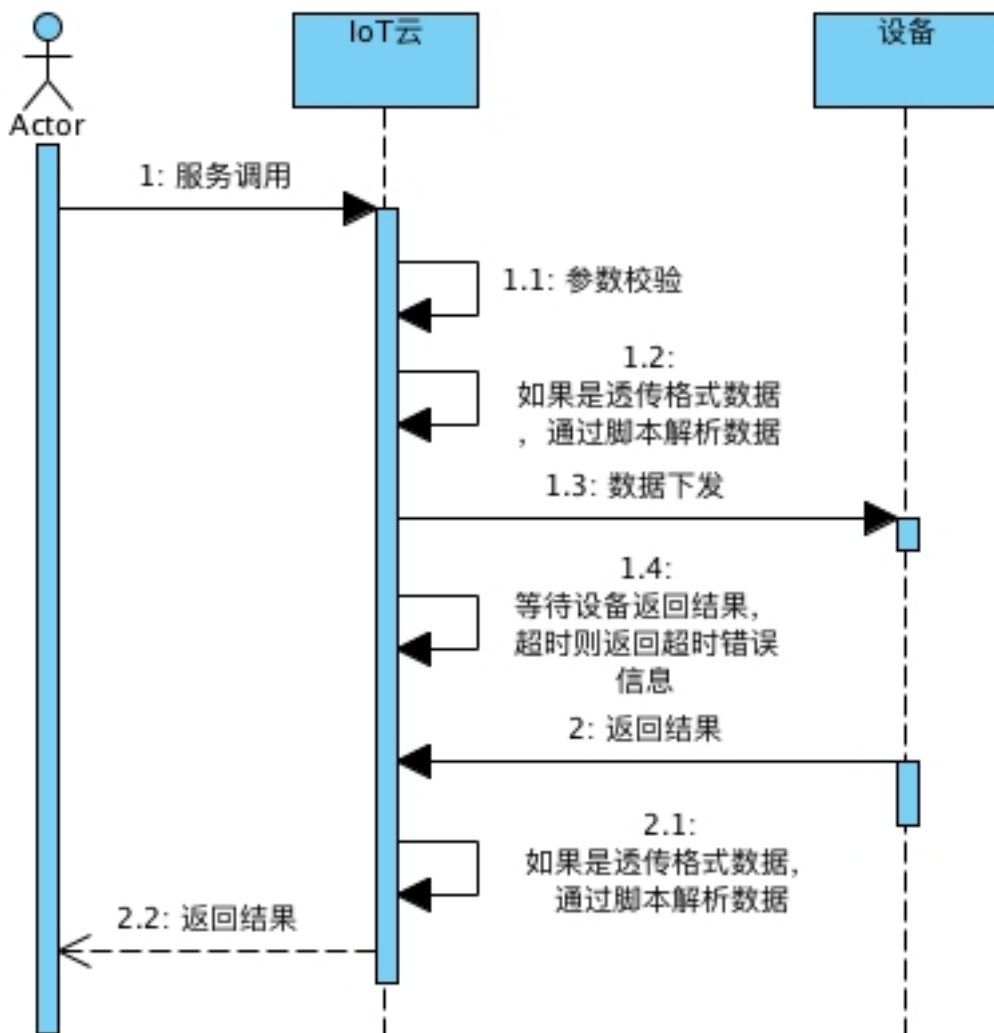
4. 下发数据给设备，设备异步处理结果。

- 如果是透传格式（透传/自定义）数据，则使用透传格式Topic。
  - 如果是非透传格式（Alink JSON）数据，则使用非透传格式Topic。
5. 设备处理完成业务后，返回处理结果。
- 如果是透传格式（透传/自定义）数据，通过脚本解析，调用脚本中的rawDataToProtocol方法，对设备返回的结果进行数据格式转换。
  - 如果配置了规则引擎，则通过规则引擎将数据输出到规则引擎配置的目的云产品中。



说明：

- 可以在规则引擎中，使用Topic：`/sys/{productKey}/{deviceName}/thing/downlink/reply/message`，获取异步调用的返回结果。
  - 透传格式（透传/自定义）规则引擎获取到的数据是经过脚本解析之后的数据。
- 同步服务调用或属性设置



1. 调用同步服务。
2. 对参数进行校验。

如果是透传格式（透传/自定义）数据，通过脚本解析，调用脚本中的`protocalToRawData`方法，对下发给设备的数据进行数据格式转换。

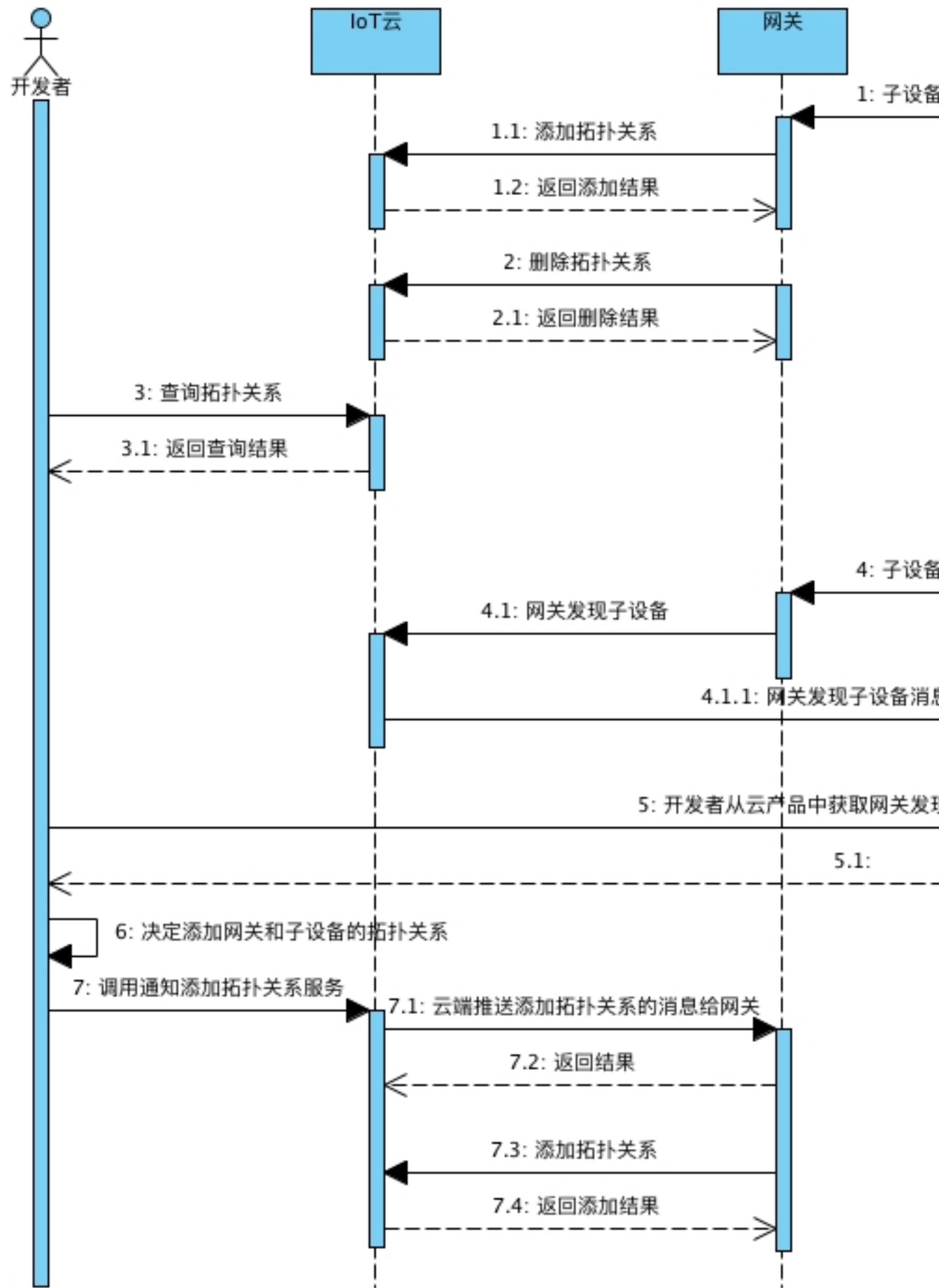
3. 使用同步调用方式，调用RRPC的Topic，下发数据给设备，云端同步等待设备返回结果。
4. 设备处理完成业务后，返回处理结果。若超时则返回超时的错误信息。

如果是透传格式（透传/自定义）数据，通过脚本解析，调用脚本中的`rawDataToProtocal`方法，对设备返回的结果进行数据格式转换。

5. 返回结果



拓扑关系



1. 子设备连接到网关后，网关通过添加拓扑关系Topic，添加拓扑关系，云端返回添加的结果。
2. 网关可以通过删除拓扑关系的Topic，来删除网关和子设备的拓扑关系。
3. 开发者可以调用**GetThingTopo**接口来查询网关和子设备的拓扑关系。
4. 当添加拓扑关系需要第三方介入时，可以通过下面的步骤添加拓扑关系。
  - a. 网关通过发现设备列表的Topic，上报发现的子设备信息。
  - b. 云端收到上报数据后，如果配置了规则引擎，可以通过规则引擎将数据流转到对应的云产品中，开发者可以从云产品中获取该数据。
  - c. 当开发者从云产品中获取了网关发现的子设备后，可以决定是否添加与网关的拓扑关系。如果需要添加拓扑关系，可以调用**NotifyAddThingTopo**接口，通知网关发起添加拓扑关系。
  - d. 云端收到**NotifyAddThingTopo**接口调用后，会通知添加拓扑关系的Topic将命令推送给网关。
  - e. 网关收到通知添加拓扑关系的命令后，通过添加拓扑关系Topic，添加拓扑关系。



说明：

- 网关通过Topic：`/sys/{productKey}/{deviceName}/thing/topo/add`，添加拓扑关系。
- 网关通过Topic：`/sys/{productKey}/{deviceName}/thing/topo/delete`，删除拓扑关系。
- 网关通过Topic：`/sys/{productKey}/{deviceName}/thing/topo/get`，获取网关和子设备的拓扑关系。
- 网关通过发现设备列表的Topic：`/sys/{productKey}/{deviceName}/thing/list/found`，上报发现的子设备信息。
- 网关通过Topic：`/sys/{productKey}/{deviceName}/thing/topo/add/notify`，通知网关设备对子设备发起添加拓扑关系。

## 12.2 设备身份注册

设备上线之前您需要对设备进行身份注册，标识您的设备。

接入IoT平台的设备身份注册有两种方式：

- 使用一机一密的方式，在控制台申请三元组（`ProductKey`，`DeviceName`，`DeviceSecret`）做为设备唯一标识，您只需要将申请到的三元组预烧录到固件，固件在完成上线建连后即可向云端上报数据。
- 使用动态注册的方式，包括直连设备使用一型一密动态注册和子设备动态注册。

### — 直连设备使用一型一密动态注册的流程：

1. 在控制台预注册，并获取产品证书（ProductKey和ProductSecret）。其中，预注册时DeviceName使用可以直接从设备中读出的数据，如mac地址或SN序列号等。
2. 在控制台开启产品的动态注册开关。
3. 将产品证书烧录至固件。
4. 设备直接向云端发起身份认证。云端认证成功后，下发DeviceSecret。
5. 设备使用三元组与云端建立连接。

### — 子设备动态注册流程：

1. 在控制台预注册获取ProductKey，其中，预注册时DeviceName使用可以直接从设备中读出的数据，如mac地址或SN序列号等。
2. 在控制台开启动态注册开关。
3. 将子设备ProductKey烧录至固件或网关上。
4. 网关代替子设备向云端发起身份注册。

## 子设备的动态注册

### 上行

- TOPIC: /sys/{productKey}/{deviceName}/thing/sub/register
- REPLY TOPIC: /sys/{productKey}/{deviceName}/thing/sub/register\_reply

### Alink请求数据格式

```
{
  "id": "123",
  "version": "1.0",
  "params": [
    {
      "deviceName": "deviceName1234",
      "productKey": "1234556554"
    }
  ],
  "method": "thing.sub.register"
}
```

### Alink响应数据格式

```
{
  "id": "123",
  "code": 200,
  "data": [
    {
      "iotId": "12344",

```

```
{
  "productKey": "1234556554",
  "deviceName": "deviceName1234",
  "deviceSecret": "xxxxxx"
}
```

#### 参数说明

参数	类型	说明
id	Long	消息ID号，保留值
version	String	协议版本号，目前协议版本1.0
params	List	设备动态注册的参数
deviceName	String	子设备的名称
productKey	String	子设备的产品Key
iotId	String	设备的唯一标识Id
deviceSecret	String	设备秘钥
method	String	请求方法
code	Integer	结果信息

#### 错误信息

错误码	消息	描述
460	request parameter error	请求参数错误
6402	topo relation cannot add by self	设备不能将自己添加为自己的子设备
401	request auth error	签名校验失败

#### 直连设备使用一型一密动态注册

直连设备动态注册，通过HTTP请求进行。使用时需在控制台上开通该产品的一型一密动态注册功能。

- URL TMLATE： <https://iot-auth.cn-shanghai.aliyuncs.com/auth/register/device>
- HTTP METHOD：POST

## Alink请求数据格式

```
POST /auth/register/device HTTP/1.1
Host: iot-auth.cn-shanghai.aliyuncs.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 123
productKey=1234556554&deviceName=deviceName1234&random=567345&sign=
adfvl23hdfdh&signMethod=HmacMD5
```

## Alink响应数据格式

```
{
  "code": 200,
  "data": {
    "productKey": "1234556554",
    "deviceName": "deviceName1234",
    "deviceSecret": "adsfweafdsf"
  },
  "message": "success"
}
```

## 参数说明

参数	类型	说明
productKey	String	产品唯一标识
deviceName	String	设备名称
random	String	随机数
sign	String	签名
signMethod	String	签名方法，目前支持hmacmd5、hmacsha1、hmacsha256
code	Integer	结果信息
deviceSecret	String	设备密钥

## 签名方法

加签内容为将所有提交给服务器的参数（**sign**、**signMethod**除外）按照字母顺序排序，然后将参数和值依次拼接（无拼接符号）。

对加签内容，需使用**signMethod**指定的加签算法进行加签。

示例如下：

```
sign = hmac_shal(productSecret, deviceNamedeviceName1234productKey1234556554random123)
```

## 12.3 添加拓扑关系

子设备身份注册后，下一步需由网关向云端上报[网关与子设备](#)的拓扑关系，然后进行上线。

上线过程中，云端会校验子设备的身份和与拓扑关系。验证合法，才会建立并绑定子设备逻辑通道至网关物理通道上。子设备与云端的数据上下行与直连设备协议可以完全保持一致，协议上不需要露出网关信息。

删除拓扑关系后，子设备利用该网关再次上线时，系统将提示拓扑关系不存在，认证不通过。

### 添加物的拓扑关系

上行

- TOPIC : /sys/{productKey}/{deviceName}/thing/topo/add
- REPLY TOPIC : sys/{productKey}/{deviceName}/thing/topo/add\_reply

Alink请求数据格式

```
{
  "id": "123",
  "version": "1.0",
  "params": [
    {
      "deviceName": "deviceName1234",
      "productKey": "1234556554",
      "sign": "xxxxxxx",
      "signmethod": "hmacShal",
      "timestamp": "1524448722000",
      "clientId": "xxxxxxx"
    }
  ],
  "method": "thing.topo.add"
}
```

Alink响应数据格式

```
{
  "id": "123",
  "code": 200,
  "data": {}
}
```

参数说明

参数	类型	说明
id	Long	消息ID号，保留值
version	String	协议版本号，目前协议版本1.0
params	List	请求入参
deviceName	String	设备的名称，在request取子设备名称
productKey	String	设备的产品Key，在request中为子设备产品的Key
sign	String	签名
signmethod	String	签名方法，支持hmacSha1，hmacSha256，hmacMd5，Sha256
timestamp	String	时间戳
clientId	String	本地标记，非必填，可以和productKey&deviceName保持一致
code	Integer	结果信息，返回200表示成功

### 加签算法



注意：

物联网平台的加签算法都是通用的。

加签内容为将所有提交给服务器的参数（**sign**，**signMethod**除外）按照字母顺序排序，然后将参数和值依次拼接（无拼接符号）。

对加签内容，需使用**signMethod**指定的加签算法进行加签。

例如，在如下request请求中，对**params**中的参数按照字母顺序依次拼接后进行加签。

```
sign= hmac_md5(deviceSecret, clientId123deviceNameetestproductKey123timestamp1524448722000)
```

### 错误信息

错误码	消息	描述
460	request parameter error	请求参数错误

错误码	消息	描述
6402	topo relation cannot add by self	设备不能把自己添加为自己的子设备
401	request auth error	签名校验授权失败

## 删除物的拓扑关系

网关类型的设备，可以通过该Topic上行请求删除它和子设备之间的拓扑关系。

上行

- TOPIC : /sys/{productKey}/{deviceName}/thing/topo/delete
- REPLY TOPIC : /sys/{productKey}/{deviceName}/thing/topo/delete\_reply

## Alink请求数据格式

```
{
  "id": "123",
  "version": "1.0",
  "params": [
    {
      "deviceName": "deviceName1234",
      "productKey": "123456554"
    }
  ],
  "method": "thing.topo.delete"
}
```

## Alink响应数据格式

```
{
  "id": "123",
  "code": 200,
  "data": {}
}
```

## 参数说明

参数	类型	说明
id	Long	消息ID号，保留值
version	String	协议版本号，目前协议版本1.0
params	List	请求参数
deviceName	String	设备名称，在request中为子设备的名称
productKey	String	设备产品Key，在request中为子设备产品的Key



参数	类型	说明
method	String	请求方法
code	Integer	结果信息，200表示成功

#### 错误信息

错误码	消息	描述
460	request parameter error	请求参数错误
6100	device not found	设备不存在

#### 获取物的拓扑关系

##### 上行

- TOPIC : /sys/{productKey}/{deviceName}/thing/topo/get
- REPLY TOPIC : /sys/{productKey}/{deviceName}/thing/topo/get\_reply

网关类型的设备，可以通过该Topic获取该设备和关联的子设备拓扑关系。

#### Alink请求数据格式

```
{
  "id": "123",
  "version": "1.0",
  "params": {},
  "method": "thing.topo.get"
}
```

#### Alink响应数据格式

```
{
  "id": "123",
  "code": 200,
  "data": [
    {
      "deviceName": "deviceName1234",
      "productKey": "1234556554"
    }
  ]
}
```

#### 参数说明

参数	类型	说明
id	Long	消息ID号，保留值

参数	类型	说明
version	String	协议版本号，目前协议版本1.0
params	Object	请求参数，可为空
method	String	请求方法
deviceName	String	子设备的名称
productKey	String	子设备的产品Key
code	Integer	结果信息，200表示成功

#### 错误信息

错误码	消息	描述
460	request parameter error	请求参数错误

#### 发现设备列表上报

##### 上行

- TOPIC : /sys/{productKey}/{deviceName}/thing/list/found
- REPLY TOPIC : /sys/{productKey}/{deviceName}/thing/list/found\_reply。

在一些场景下，网关可以发现新接入的子设备。发现后，需将新接入子设备的信息上报云端，然后通过数据流转到第三方应用，选择将哪些子设备接入该网关。

#### Alink请求数据格式

```
{
  "id": "123",
  "version": "1.0",
  "params": [
    {
      "deviceName": "deviceName1234",
      "productKey": "1234556554"
    }
  ],
  "method": "thing.list.found"
}
```

#### Alink响应数据格式

```
{
  "id": "123",
  "code": 200,
  "data": {}
}
```

```
}
```

### 参数说明

参数	类型	说明
id	Long	消息ID号，保留值
version	String	协议版本号，目前协议版本1.0
params	Object	请求参数，可为空
method	String	请求方法
deviceName	String	子设备的名称
productKey	String	子设备的产品Key
code	Integer	结果信息，200表示成功

### 错误信息

错误码	消息	描述
460	request parameter error	请求参数错误
6250	product not found	上报的子设备产品不存在
6280	devicename not meet specs	上报的子设备的名称不符规范，设备名称支持英文字母、数字和特殊字符- _@.，长度限制4~32

### 通知添加设备拓扑关系

#### 下行

- TOPIC : /sys/{productKey}/{deviceName}/thing/topo/add/notify
- REPLY TOPIC : /sys/{productKey}/{deviceName}/thing/topo/add/notify\_reply

通知网关设备对子设备发起添加拓扑关系，可以配合发现设备列表上报功能使用。可以通过数据流转获取设备返回的结果，数据流转topic为 /{productKey}/{deviceName}/thing/downlink/reply/message。

### Alink请求数据格式

```
{
  "id": "123",
  "version": "1.0",
  "params": [
    {
```

```
    "deviceName": "deviceName1234",
    "productKey": "1234556554"
  },
  "method": "thing.topo.add.notify"
}
```

#### Alink响应数据格式

```
{
  "id": "123",
  "code": 200,
  "data": {}
}
```

#### 参数说明

参数	类型	说明
id	Long	消息ID号，保留值
version	String	协议版本号，目前协议版本1.0
params	Object	请求参数，可为空
method	String	请求方法
deviceName	String	子设备的名称
productKey	String	子设备的产品Key
code	Integer	结果信息，200表示成功

## 12.4 子设备上下线

直连子设备上线之前仅需确保已在云端注册身份。

子设备上线需要确保身份已经注册、已添加拓扑关系。因为子设备上线，云端需要根据拓扑关系进行身份校验，以确定子设备是否具备复用网关通道的能力。

#### 子设备上线

##### 上行

- TOPIC : /ext/session/{productKey}/{deviceName}/combine/login
- REPLY TOPIC : /ext/session/{productKey}/{deviceName}/combine/login\_reply

#### Alink请求数据格式

```
{
  "id": "123",
  "params": {
```

```
{
  "productKey": "123",
  "deviceName": "test",
  "clientId": "123",
  "timestamp": "123",
  "signMethod": "hmacmd5",
  "sign": "xxxxxx",
  "cleanSession": "true"
}
```

### Alink响应数据格式

```
{
  "id": "123",
  "code": 200,
  "message": "success",
  "data": ""
}
```

### 参数说明

参数	取值	说明
id	Long	消息ID号，保留值
params	List	请求入参
deviceName	String	子设备的设备名称
productKey	String	子设备所属的产品Key
sign	String	子设备签名，规则与网关相同
signmethod	String	签名方法，支持hmacSha1，hmacSha256，hmacMd5，Sha256
timestamp	String	时间戳
clientId	String	设备端标识，可以写productKey&deviceName
cleanSession	String	如果是true，那么清理所有子设备离线消息，即QoS1的所有未接收内容
code	Integer	结果信息，返回200表示成功
message	String	结果信息
data	String	结果中附加信息，json格式



注意：

网关下同时在线的子设备数目不能超过200，超过后，新的子设备上线请求将被拒绝。

#### 签名方法

加签内容为将所有提交给服务器的参数（**sign**，**signMethod**除外）按照字母顺序排序，然后将参数和值依次拼接（无拼接符号）。

对加签内容，需使用**signMethod**指定的加签算法进行加签。

示例如下：

```
sign= hmac_md5(deviceSecret, cleanSessiontrueclientId123deviceNametes  
tproductKey123timestamp123)
```

#### 错误信息

错误码	消息	描述
460	request parameter error	请求参数错误
429	rate limit, too many subDeviceOnline msg in one minute	单个设备认证过于频繁被限流
428	too many subdevices under gateway	网关下同时上线子设备过多
6401	topo relation not exist	网关和子设备没有拓扑关系
6100	device not found	子设备不存在
521	device deleted	子设备被删除
522	device forbidden	子设备被禁用
6287	invalid sign	子设备密码或者签名错误

#### 子设备下线

##### 上行

- TOPIC : /ext/session/{productKey}/{deviceName}/combine/logout
- REPLY TOPIC : /ext/session/{productKey}/{deviceName}/combine/logout\_reply

#### Alink请求数据格式

```
{  
  "id": 123,  
  "params": {  
    "productKey": "xxxxxx",  
    "deviceName": "xxxxxx"  
  }  
}
```

```
}
```

### Alink响应数据格式

```
{
  "id": "123",
  "code": 200,
  "message": "success",
  "data": ""
}
```

### 参数说明

参数	取值	说明
id	Long	消息ID号，保留值。
params	List	请求入参。
deviceName	String	子设备的设备名称。
productKey	String	子设备所属的产品Key。
code	Integer	结果代码，200表示成功。
message	String	结果信息。
data	String	结果中附加信息，JSON格式。

### 错误信息

错误码	消息	描述
460	request parameter error	请求参数错误。
520	device no session	子设备会话不存在。

子设备接入具体可参考[设备身份注册](#)，错误码参考[错误码说明](#)。

## 12.5 设备属性、事件、服务

如果为产品定义了[物模型](#)，设备可以按照属性、事件、服务协议分别上报数据。其中，属性、事件和服务的物模型参数格式，请参考[物模型数据格式](#)。本文仅讲解使用物模型时，如何上报数据。

设备的数据上报方式有两种：ICA 标准数据格式 (Alink JSON)和透传/自定义。两者二选一，推荐您使用Alink JSON方式。

- ICA 标准数据格式 (Alink JSON)：设备按照物联网平台定义的标准数据格式生成数据，然后上报数据。具体格式，请参见本文档中Alink请求数据格式示例和Alink相应数据格式示例。

- 透传/自定义：设备上报原始数据如二进制数据流，阿里云物联网平台会运行您在控制台提交的解析脚本，将原始数据转成标准数据格式。编写解析脚本，请参见[数据解析](#)。

## 设备上报属性

### 上行（透传）

- 请求Topic：/sys/{productKey}/{deviceName}/thing/model/up\_raw
- 响应Topic：/sys/{productKey}/{deviceName}/thing/model/up\_raw\_reply

### 上行（Alink JSON）

- 请求Topic：/sys/{productKey}/{deviceName}/thing/event/property/post
- 响应Topic：/sys/{productKey}/{deviceName}/thing/event/property/post\_reply

您可以配置[规则引擎](#)，将设备上报的属性信息转发至其他目的云产品。规则引擎设置示例如下：

编写SQL

\* 规则查询语句：

SELECT deviceName() as deviceName FROM "/sys/a1Cs40ZnKII/LightSensor/thing/event/pr

\* 字段：

deviceName() as deviceName

\* Topic：

sys

光照感应器

LightSensor

/thing/event...

条件：

可以使用规则引擎函数,例如:deviceName()=mydevice

Q

✓ /thing/event/property/post

/thing/downlink/reply/message

确认

取消

## Alink请求数据格式

```
{
  "id": "123",
  "version": "1.0",
  "params": {
    "Power": {
      "value": "on",
      "time": 1524448722000
    }
  }
}
```



```
    },
    "WF": {
      "value": 23.6,
      "time": 1524448722000
    }
  },
  "method": "thing.event.property.post"
}
```

### Alink响应数据格式

```
{
  "id": "123",
  "code": 200,
  "data": {}
}
```


表 12-1: 请求参数说明

参数	类型	说明
id	Long	消息ID号。
version	String	协议版本号，目前协议版本1.0。
params	Object	请求参数。如以上示例中，设备上报了的两个属性Power和WF。具体属性信息，包含属性上报时间（time）和上报的属性值（value）。
time	Long	属性上报时间。
value	String	上报的属性值。
method	String	请求方法。

表 12-2: 响应参数说明

参数	类型	说明
id	Long	消息ID号。
code	Integer	结果状态码。具体参考 <a href="#">设备端通用code</a> 。
data	String	请求成功时，返回的数据。

### 错误信息

错误码	消息	描述
460	request parameter error	请求参数错误。
6106	map size must less than 200	一次最多只能上报200条属性。
6313	tsl service not available	<p>物模型校验服务不可用。 物联网平台会校验上报的属性信息，通过产品TSL描述判断上传的属性是否符合您定义的属性格式。当校验服务不可用时会报这个错。属性校验请参考<a href="#">概述</a>。</p> <div>  <p><b>说明：</b> 校验结果为不合格的属性将被直接过滤掉，仅保留合格属性。若所有属性都不合格，会过滤掉全部属性，返回的response仍是校验成功。</p> </div>

## 设置设备属性

### 下行（透传）

- 请求Topic：/sys/{productKey}/{deviceName}/thing/model/down\_raw
- 响应Topic：/sys/{productKey}/{deviceName}/thing/model/down\_raw\_reply

### 下行（Alink JSON）

- 请求Topic：/sys/{productKey}/{deviceName}/thing/service/property/set
- 响应Topic：/sys/{productKey}/{deviceName}/thing/service/property/set\_reply

属性设置的结果，可以通过订阅数据流转信息获取。数据流转Topic为/sys/{productKey}/{deviceName}/thing/downlink/reply/message。您可以配置[规则引擎](#)，将属性设置后，设备返回的结果转发至其它目的云产品。规则引擎设置示例如下：

编写SQL

×

\* 规则查询语句：

SELECT deviceName() as deviceName FROM "/sys/a1Cs40ZnKII/LightSensor/thing/downlink

\* 字段：

deviceName() as deviceName

\* Topic：

sys光照感应器LightSensor

/thing/downlink/reply/message

条件：

可以使用规则引擎函数,例如:deviceName()=mydevice

确认

取消

## Alink请求数据格式

```
{
  "id": "123",
  "version": "1.0",
  "params": {
    "temperature": "30.5"
  },
  "method": "thing.service.property.set"
}
```

## Alink响应数据格式

```
{
  "id": "123",
  "code": 200,
  "data": {}
}
```

### 表 12-3: 请求参数说明

参数	类型	说明
id	Long	消息ID号。
version	String	协议版本号，目前协议版本1.0。

参数	类型	说明
params	Object	属性设置参数。如以上示例中，设置属性： <pre>{ "temperature": "30.5" }</pre> 。
method	String	请求方法。

表 12-4: 响应参数说明

参数	类型	说明
id	Long	消息ID号。
code	Integer	结果状态码，具体参考 <a href="#">设备端通用code</a> 。
data	String	请求成功时，返回的数据。

## 设备事件上报

### 上行（透传）

- 请求Topic：/sys/{productKey}/{deviceName}/thing/model/up\_raw
- 响应Topic：/sys/{productKey}/{deviceName}/thing/model/up\_raw\_reply

### 上行（Alink JSON）

- 请求Topic：/sys/{productKey}/{deviceName}/thing/event/{tsl.event.identifier}/post
- 响应Topic：/sys/{productKey}/{deviceName}/thing/event/{tsl.event.identifier}/post\_reply

您可以配置[规则引擎](#)，将设备上报的事件信息转发至其他目的云产品。规则引擎设置示例如下：

### 编写SQL

\* 规则查询语句 :

SELECT \* FROM "/sys/a1nDqW3wwrL" WHERE

\* 字段 :

\*

\* Topic :

sys

高级版

hyDZmQWIZ...

条件 :

可以使用规则引擎函数,例如:deviceName()=mydevice

/thing/event/property/post

/thing/event/testAlertEvent/post

/thing/event/testInfoEvent/post

/thing/event/testErrorEvent/post

/thing/downlink/reply/message

请选择Topic

确认

取消

### Alink请求数据格式

```
{
  "id": "123",
  "version": "1.0",
  "params": {
    "value": {
      "Power": "on",
      "WF": "2"
    },
    "time": 1524448722000
  },
  "method": "thing.event.{tsl.event.identifier}.post"
}
```

### Alink响应数据格式

```
{
  "id": "123",
  "code": 200,
  "data": {}
}
```

文档版本：20181220

127

```
}
```

表 12-5: 请求参数说明

参数	类型	说明
id	Long	消息ID号。
version	String	协议版本号，目前协议版本1.0。
params	List	上报事件的参数。
value	Object	具体的事件信息。如以上示例中 <pre>{     "Power": "on",     "WF": "2" }</pre>
time	Long	事件生成的时间戳，类型为UTC毫秒级时间。
method	String	请求方法。

表 12-6: 响应参数说明

参数	类型	说明
id	Long	消息ID号。
code	Integer	结果状态码，具体参考 <a href="#">设备端通用code</a> 。
data	String	请求成功时，返回的数据。

### 示例

假设产品中定义了一个alarm事件，它的TSL描述如下：

```
{  
  "schema": "https://iot-tsl.oss-cn-shanghai.aliyuncs.com/schema.json",  
  "link": "/sys/${productKey}/airCondition/thing/",  
  "profile": {  
    "productKey": "${productKey}，请替换为您的ProductKey",  
    "deviceName": "airCondition，请替换为您的Devicename"  
  },  
  "events": [  
    {
```

```

    "identifier": "alarm",
    "name": "alarm",
    "desc": "风扇警报",
    "type": "alert",
    "required": true,
    "outputData": [
      {
        "identifier": "errorCode",
        "name": "错误码",
        "dataType": {
          "type": "text",
          "specs": {
            "length": "255"
          }
        }
      }
    ],
    "method": "thing.event.alarm.post"
  }
]
}

```

当设备上报事件时，Alink请求数据格式：

```

{
  "id": "123",
  "version": "1.0",
  "params": {
    "value": {
      "errorCode": "error"
    },
    "time": 1524448722000
  },
  "method": "thing.event.alarm.post"
}

```



说明：

- `tsl.event.identifier` 为TSL模板中，事件的描述符。TSL模板具体参考[概述](#)。
- 物联网平台会对设备上报的事件做校验。通过产品的TSL描述判断上报的事件是否符合定义的事件格式。不合格的事件会直接被过滤掉，并返回失败的错误码。

## 设备服务调用

- 透传（下行）
  - 请求Topic：`/sys/{productKey}/{deviceName}/thing/model/down_raw`
  - 响应Topic：`/sys/{productKey}/{deviceName}/thing/model/down_raw_reply`
- Alink JSON（下行）
  - 请求Topic：`/sys/{productKey}/{deviceName}/thing/service/{tsl.service.identifier}`

### — 响应Topic：/sys/{productKey}/{deviceName}/thing/service/{tsl.service.identifier}\_reply

服务调用按照调用方式可以分为：同步调用和异步调用。[物模型定义服务](#)时，需设置此项。

- 同步方式：物联网平台直接使用RRPC同步方式下行推送请求。设备RRPC的集成方式，请参见[什么是RRPC](#)。
- 异步方式：物联网平台则采用异步方式下行推送请求，设备也采用异步方式返回结果。

只有当前服务选择为异步调用方式，物联网平台才会订阅该异步响应Topic。异步调用的结果，可以通过数据流转获取。数据流转Topic为/sys/{productKey}/{deviceName}/thing/downlink/reply/message。

您可以配置[规则引擎](#)，将设备返回的服务调用结果转发至其它目的云产品。规则引擎设置示例如下：

编写SQL

\* 规则查询语句：

SELECT deviceName() as deviceName FROM "/sys/a1Cs40ZnKII/LightSensor/thing/downlink"

\* 字段：

deviceName() as deviceName

\* Topic：

sys

光照感应器

LightSensor

/thing/down... ^

条件：

可以使用规则引擎函数,例如:deviceName()=mydevice

/thing/event/property/post

✓ /thing/downlink/reply/message

确认

取消

### Alink请求数据格式

```
{
  "id": "123",
  "version": "1.0",
  "params": {
    "Power": "on",
    "WF": "2"
  },
  "method": "thing.service.{tsl.service.identifier}"
}
```



```
}
```

### Alink响应数据格式

```
{
  "id": "123",
  "code": 200,
  "data": {}
}
```

表 12-7: 请求参数说明

参数	类型	说明
id	Long	消息ID号。
version	String	协议版本号，目前协议版本1.0。
params	List	服务调用参数。包含服务标识符和服务的值。如以上示例中： <pre>{   "Power": "on",   "WF": "2" }</pre>
method	String	请求方法。

表 12-8: 返回参数说明

参数	类型	说明
id	Long	消息ID号。
code	Integer	结果状态码，具体参考 <a href="#">设备端通用code</a> 。
data	String	返回的结果信息。 <b>data</b> 参数的值和物模型定义相关。如果服务没有返回结果，则 <b>data</b> 的值为空。如果服务有返回结果，则返回的数据会严格遵循服务的定义。

### 示例

比如产品中定义了服务SetWeight，它的TSL描述如下：

```
{
  "schema": "https://iotx-tsl.oss-ap-southeast-1.aliyuncs.com/schema.
  json",
  "profile": {
    "productKey": "testProduct01"
  },
  "services": [
    {
      "outputData": [
        {
          "identifier": "OldWeight",
          "dataType": {
            "specs": {
              "unit": "kg",
              "min": "0",
              "max": "200",
              "step": "1"
            },
            "type": "double"
          },
          "name": "OldWeight"
        },
        {
          "identifier": "CollectTime",
          "dataType": {
            "specs": {
              "length": "2048"
            },
            "type": "text"
          },
          "name": "CollectTime"
        }
      ],
      "identifier": "SetWeight",
      "inputData": [
        {
          "identifier": "NewWeight",
          "dataType": {
            "specs": {
              "unit": "kg",
              "min": "0",
              "max": "200",
              "step": "1"
            },
            "type": "double"
          },
          "name": "NewWeight"
        }
      ],
      "method": "thing.service.SetWeight",
      "name": "设置重量",
      "required": false,
      "callType": "async"
    }
  ]
}
```

```
}
```

当调用服务时，Alink请求数据格式

```
{
  "method": "thing.service.SetWeight",
  "id": "105917531",
  "params": {
    "NewWeight": 100.8
  },
  "version": "1.0.0"
}
```

Alink响应数据格式

```
{
  "id": "105917531",
  "code": 200,
  "data": {
    "CollectTime": "1536228947682",
    "OldWeight": 100.101
  }
}
```



说明：

tsl.service.identifier为tsl模板中定义的服务描述符。TSL使用参考[概述](#)。

## 12.6 网关配置下发

物模型扩展配置及子设备的连接协议配置，由云端推送至设备端。

- TOPIC : /sys/{productKey}/{deviceName}/thing/model/config/push

Alink配置推送数据格式

```
{
  "id": 123,
  "version": "1.0",
  "method": "thing.model.config.push",
  "data": {
    "digest": "",
    "digestMethod": "",
    "url": ""
  }
}
```

参数说明

参数	类型	说明
id	String	消息ID号

参数	类型	说明
version	String	协议版本号，默认为1.0
method	String	方法，使用thing.model.config.push
data	Object	数据
digest	String	签名，用于校验从url中获取的数据完整性
digestMethod	String	签名方法，默认sha256
url	String	数据url，从oss中请求获取，数据格式如下文所示。

#### url中的数据格式

```
{
  "modelList": [
    {
      "profile": {
        "productKey": "test01"
      },
      "services": [
        {
          "outputData": "",
          "identifier": "AngleSelfAdaption",
          "inputData": [
            {
              "identifier": "test01",
              "index": 0
            }
          ],
          "displayName": "test01"
        }
      ],
      "properties": [
        {
          "identifier": "identifier",
          "displayName": "test02"
        },
        {
          "identifier": "identifier_01",
          "displayName": "identifier_01"
        }
      ],
      "events": [
        {
          "outputData": [
            {
              "identifier": "test01",
              "index": 0
            }
          ],
          "identifier": "event1",
```

```
        "displayName": "abc"
      }
    ]
  },
  {
    "profile": {
      "productKey": "test02"
    },
    "properties": [
      {
        "originalDataType": {
          "specs": {
            "registerCount": 1,
            "reverseRegister": 0,
            "swap16": 0
          },
          "type": "bool"
        },
        "identifier": "test01",
        "registerAddress": "0x03",
        "scaling": 1,
        "operateType": "inputStatus",
        "pollingTime": 1000,
        "trigger": 1
      },
      {
        "originalDataType": {
          "specs": {
            "registerCount": 1,
            "reverseRegister": 0,
            "swap16": 0
          },
          "type": "bool"
        },
        "identifier": "test02",
        "registerAddress": "0x05",
        "scaling": 1,
        "operateType": "coilStatus",
        "pollingTime": 1000,
        "trigger": 2
      }
    ]
  }
],
"serverList": [
  {
    "baudRate": 1200,
    "protocol": "RTU",
    "byteSize": 8,
    "stopBits": 2,
    "parity": 1,
    "name": "modbus01",
    "serialPort": "0",
    "serverId": "D73251B4277742"
  },
  {
    "protocol": "TCP",
    "port": 8000,
    "ip": "192.168.0.1",
    "name": "modbus02",
    "serverId": "586CB066D6A34"
  }
],
```

```

    {
      "password": "XIJTginONohPEUAYZxLB7Q==",
      "secPolicy": "Basic128Rsa15",
      "name": "server_01",
      "secMode": "Sign",
      "userName": "123",
      "serverId": "55A9D276A7ED470",
      "url": "tcp:00",
      "timeout": 10
    },
    {
      "password": "hAaX5s13gwX2JwyvUkOafQ==",
      "name": "service_09",
      "secMode": "None",
      "userName": "1234",
      "serverId": "44895C63E3FF401",
      "url": "tcp:00",
      "timeout": 10
    }
  ],
  "deviceList": [
    {
      "deviceConfig": {
        "displayNamePath": "123",
        "serverId": "44895C63E3FF4013924CEF31519ABE7B"
      },
      "productKey": "test01",
      "deviceName": "test_02"
    },
    {
      "deviceConfig": {
        "displayNamePath": "1",
        "serverId": "55A9D276A7ED47"
      },
      "productKey": "test01",
      "deviceName": "test_03"
    },
    {
      "deviceConfig": {
        "slaveId": 1,
        "serverId": "D73251B4277742D"
      },
      "productKey": "test02",
      "deviceName": "test01"
    },
    {
      "deviceConfig": {
        "slaveId": 2,
        "serverId": "586CB066D6A34E"
      },
      "productKey": "test02",
      "deviceName": "test02"
    }
  ],
  "tslList": [
    {
      "schema": "https://iotx-tsl.oss-ap-southeast-1.aliyuncs.com/schema.json",
      "profile": {
        "productKey": "test02"
      },
      "services": [

```

```

{
  "outputData": [],
  "identifier": "set",
  "inputData": [
    {
      "identifier": "test02",
      "dataType": {
        "specs": {
          "unit": "mm",
          "min": "0",
          "max": "1"
        },
        "type": "int"
      },
      "name": "测试功能02"
    }
  ],
  "method": "thing.service.property.set",
  "name": "set",
  "required": true,
  "callType": "async",
  "desc": "属性设置"
},
{
  "outputData": [
    {
      "identifier": "test01",
      "dataType": {
        "specs": {
          "unit": "m",
          "min": "0",
          "max": "1"
        },
        "type": "int"
      },
      "name": "测试功能01"
    },
    {
      "identifier": "test02",
      "dataType": {
        "specs": {
          "unit": "mm",
          "min": "0",
          "max": "1"
        },
        "type": "int"
      },
      "name": "测试功能02"
    }
  ],
  "identifier": "get",
  "inputData": [
    "test01",
    "test02"
  ],
  "method": "thing.service.property.get",
  "name": "get",
  "required": true,
  "callType": "async",
  "desc": "属性获取"
}

```

```

],
"properties": [
  {
    "identifier": "test01",
    "dataType": {
      "specs": {
        "unit": "m",
        "min": "0",
        "max": "1"
      },
      "type": "int"
    },
    "name": "测试功能01",
    "accessMode": "r",
    "required": false
  },
  {
    "identifier": "test02",
    "dataType": {
      "specs": {
        "unit": "mm",
        "min": "0",
        "max": "1"
      },
      "type": "int"
    },
    "name": "测试功能02",
    "accessMode": "rw",
    "required": false
  }
],
"events": [
  {
    "outputData": [
      {
        "identifier": "test01",
        "dataType": {
          "specs": {
            "unit": "m",
            "min": "0",
            "max": "1"
          },
          "type": "int"
        },
        "name": "测试功能01"
      },
      {
        "identifier": "test02",
        "dataType": {
          "specs": {
            "unit": "mm",
            "min": "0",
            "max": "1"
          },
          "type": "int"
        },
        "name": "测试功能02"
      }
    ],
    "identifier": "post",
    "method": "thing.event.property.post",
    "name": "post",

```



```

        "type": "info",
        "required": true,
        "desc": "属性上报"
    }
  ]
},
{
  "schema": "https://iotx-tsl.oss-ap-southeast-1.aliyuncs.com/
schema.json",
  "profile": {
    "productKey": "test01"
  },
  "services": [
    {
      "outputData": [],
      "identifier": "set",
      "inputData": [
        {
          "identifier": "identifier",
          "dataType": {
            "specs": {
              "length": "2048"
            },
            "type": "text"
          },
          "name": "7614"
        },
        {
          "identifier": "identifier_01",
          "dataType": {
            "specs": {
              "length": "2048"
            },
            "type": "text"
          },
          "name": "测试功能1"
        }
      ],
      "method": "thing.service.property.set",
      "name": "set",
      "required": true,
      "callType": "async",
      "desc": "属性设置"
    },
    {
      "outputData": [
        {
          "identifier": "identifier",
          "dataType": {
            "specs": {
              "length": "2048"
            },
            "type": "text"
          },
          "name": "7614"
        },
        {
          "identifier": "identifier_01",
          "dataType": {
            "specs": {
              "length": "2048"
            },
            "type": "text"
          },
          "name": "测试功能1"
        }
      ],
      "method": "thing.service.property.set",
      "name": "set",
      "required": true,
      "callType": "async",
      "desc": "属性设置"
    }
  ]
}

```

```

        "type": "text"
      },
      "name": "测试功能1"
    }
  ],
  "identifier": "get",
  "inputData": [
    "identifier",
    "identifier_01"
  ],
  "method": "thing.service.property.get",
  "name": "get",
  "required": true,
  "callType": "async",
  "desc": "属性获取"
},
{
  "outputData": [],
  "identifier": "AngleSelfAdaption",
  "inputData": [
    {
      "identifier": "test01",
      "dataType": {
        "specs": {
          "min": "1",
          "max": "10",
          "step": "1"
        },
        "type": "int"
      },
      "name": "参数1"
    }
  ],
  "method": "thing.service.AngleSelfAdaption",
  "name": "角度自适应校准",
  "required": false,
  "callType": "async"
}
],
"properties": [
  {
    "identifier": "identifier",
    "dataType": {
      "specs": {
        "length": "2048"
      },
      "type": "text"
    },
    "name": "7614",
    "accessMode": "rw",
    "required": true
  },
  {
    "identifier": "identifier_01",
    "dataType": {
      "specs": {
        "length": "2048"
      },
      "type": "text"
    },
    "name": "测试功能1",

```

```
        "accessMode": "rw",
        "required": false
    },
    ],
    "events": [
        {
            "outputData": [
                {
                    "identifier": "identifier",
                    "dataType": {
                        "specs": {
                            "length": "2048"
                        },
                        "type": "text"
                    },
                    "name": "7614"
                },
                {
                    "identifier": "identifier_01",
                    "dataType": {
                        "specs": {
                            "length": "2048"
                        },
                        "type": "text"
                    },
                    "name": "测试功能1"
                }
            ],
            "identifier": "post",
            "method": "thing.event.property.post",
            "name": "post",
            "type": "info",
            "required": true,
            "desc": "属性上报"
        },
        {
            "outputData": [
                {
                    "identifier": "test01",
                    "dataType": {
                        "specs": {
                            "min": "1",
                            "max": "20",
                            "step": "1"
                        },
                        "type": "int"
                    },
                    "name": "测试参数1"
                }
            ],
            "identifier": "event1",
            "method": "thing.event.event1.post",
            "name": "event1",
            "type": "info",
            "required": false
        }
    ]
}
```

```
}
```

参数说明如下：

参数	类型	描述
modelList	Object	网关下面所有子设备的产品扩展信息
serverList	Object	网关下面所有的子设备通道管理
deviceList	Object	网关下面所有子设备的连接配置
tslList	Object	网关下面所有子设备的TSL描述

### modelList说明

设备协议目前支持Modbus和OPC UA两种协议，两种协议的扩展信息不一致。

- Modbus

```
{
  "profile": {
    "productKey": "test02"
  },
  "properties": [
    {
      "originalDataType": {
        "specs": {
          "registerCount": 1,
          "reverseRegister": 0,
          "swap16": 0
        },
        "type": "bool"
      },
      "identifier": "test01",
      "registerAddress": "0x03",
      "scaling": 1,
      "operateType": "inputStatus",
      "pollingTime": 1000,
      "trigger": 1
    },
    {
      "originalDataType": {
        "specs": {
          "registerCount": 1,
          "reverseRegister": 0,
          "swap16": 0
        },
        "type": "bool"
      },
      "identifier": "test02",
      "registerAddress": "0x05",
      "scaling": 1,
      "operateType": "coilStatus",
      "pollingTime": 1000,
      "trigger": 2
    }
  ]
}
```

```
}

```

参数说明如下：

参数	类型	说明
identifier	String	属性、事件和服务的描述符
operateType	String	参数类型，取值可以为： <ul style="list-style-type: none"> <li>线圈状态：coilStatus</li> <li>输入状态：inputStatus</li> <li>保持寄存器：holdingRegister</li> <li>输入寄存器：inputRegister</li> </ul>
registerAddress	String	寄存器地址
originalDataType	Object	数据原始类型
type	String	取值如下： int16，uint16，int32，uint32，int64，uint64，float，double，string，customized data
specs	Object	描述信息
registerCount	Integer	寄存器的数据个数
swap16	Integer	把寄存器内16位数据的前后8个bits互换。0表示false、1表示true
reverseRegister	Integer	把原始数据32位数据的bits互换。0表示false，1表示true
scaling	Integer	缩放因子
pollingTime	Integer	采集间隔
trigger	Integer	数据的上报方式，1 按时上报，2 变更上报

#### • OPC UA

```
{
  "profile": {
    "productKey": "test01"
  },
  "services": [
    {
      "outputData": "",
      "identifier": "AngleSelfAdaption",
      "inputData": [
        {
          "identifier": "test01",
          "index": 0
        }
      ]
    }
  ]
}
```

```
    ],
    "displayName": "test01"
  },
],
"properties": [
  {
    "identifier": "identifier",
    "displayName": "test02"
  },
  {
    "identifier": "identifier_01",
    "displayName": "identifier_01"
  }
],
"events": [
  {
    "outputData": [
      {
        "identifier": "test01",
        "index": 0
      }
    ],
    "identifier": "event1",
    "displayName": "abc"
  }
]
}
```

参数说明如下：

参数	类型	说明
services	Object	服务
properties	Object	属性
events	Object	事件
outputData	Object	输出参数，如事件上报数据、服务调用的返回结果
identifier	String	描述信息，用于标识
inputData	Object	输入参数，如服务的入参
index	Integer	索引信息
displayName	String	展示名称

serverList说明

通道的信息也分为Modbus和OPC UA协议两种。

- Modbus协议

```
[
  {
```

```

    "baudRate": 1200,
    "protocol": "RTU",
    "byteSize": 8,
    "stopBits": 2,
    "parity": 1,
    "name": "modbus01",
    "serialPort": "0",
    "serverId": "D73251B4277742"
  },
  {
    "protocol": "TCP",
    "port": 8000,
    "ip": "192.168.0.1",
    "name": "modbus02",
    "serverId": "586CB066D6A34"
  }
]

```

参数	类型	说明
protocol	String	协议类型，TCP或者RTU
port	Integer	端口号
ip	String	IP地址
name	String	通道名称
serverId	String	通道的ID
baudRate	Integer	波特率
byteSize	Integer	字节数
stopBits	Integer	停止位
parity	Integer	奇偶校验位 <ul style="list-style-type: none"> <li>E：偶校验</li> <li>O：奇校验</li> <li>N：无校验</li> </ul>
serialPort	String	串口号

- OPC UA协议

```

{
  "password": "XIjTginONohPEUAYzxLB7Q==",
  "secPolicy": "Basic128Rsa15",
  "name": "server_01",
  "secMode": "Sign",
  "userName": "123",
  "serverId": "55A9D276A7ED470",
  "url": "tcp:00",
  "timeout": 10
}

```

```
}
```

参数说明如下：

参数	类型	说明
password	String	密码，采用AES算法加密。具体说明见下文 OPC UA的password加密方式
secPolicy	String	加密策略，取值：None，Basic128Rsa15，Basic256
secMode	String	加密模式，取值：None，Sign，SignAndEncrypt
name	String	server名称
userName	String	用户名
serverId	String	server的ID
url	String	服务器连接地址
timeout	Integer	超时时间

#### OPC UA的password加密方式

加密算法采用AES算法，使用128位（16字节）分组，缺省的mode为CBC，缺省的padding为PKCS5Padding，密钥使用设备的deviceSecret，加密后的内容采用BASE64进行编码。

加解密示例代码：

```
private static String instance = "AES/CBC/PKCS5Padding";

private static String algorithm = "AES";

private static String charsetName = "utf-8";
/**
 * 加密算法
 *
 * @param data          待加密内容
 * @param deviceSecret  设备的密钥
 * @return
 */
public static String aesEncrypt(String data, String deviceSecret) {
    try {
        Cipher cipher = Cipher.getInstance(instance);
        byte[] raw = deviceSecret.getBytes();
        SecretKeySpec key = new SecretKeySpec(raw, algorithm);
        IvParameterSpec ivParameter = new IvParameterSpec(
            deviceSecret.substring(0, 16).getBytes());
        cipher.init(Cipher.ENCRYPT_MODE, key, ivParameter);
```



```

        byte[] encrypted = cipher.doFinal(data.getBytes(
charsetName));

        return new BASE64Encoder().encode(encrypted);
    } catch (Exception e) {
        e.printStackTrace();
    }

    return null;
}

public static String aesDecrypt(String data, String deviceSecret
) {
    try {
        byte[] raw = deviceSecret.getBytes(charsetName);
        byte[] encrypted1 = new BASE64Decoder().decodeBuffer(
data);

        SecretKeySpec key = new SecretKeySpec(raw, algorithm);
        Cipher cipher = Cipher.getInstance(instance);
        IvParameterSpec ivParameter = new IvParameterSpec(
deviceSecret.substring(0, 16).getBytes());
        cipher.init(Cipher.DECRYPT_MODE, key, ivParameter);
        byte[] originalBytes = cipher.doFinal(encrypted1);
        String originalString = new String(originalBytes,
charsetName);
        return originalString;
    } catch (Exception ex) {
        ex.printStackTrace();
    }

    return null;
}

public static void main(String[] args) throws Exception {
    String text = "test123";
    String secret = "testTNmjyWHQzniA8wEkTNmjyWHQtest";
    String data = null;
    data = aesEncrypt(text, secret);
    System.out.println(data);
    System.out.println(aesDecrypt(data, secret));
}

```

## deviceList说明

- Modbus协议

```

{
  "deviceConfig": {
    "slaveId": 1,
    "serverId": "D73251B4277742D"
  },
  "productKey": "test02",
  "deviceName": "test01"
}

```

参数说明如下：

参数	类型	取值
deviceConfig	Object	设备信息
slaveId	Integer	从站ID
serverId	String	通道ID
productKey	String	产品ID
deviceName	String	设备名称

- OPC UA协议

```
{
  "deviceConfig": {
    "displayNamePath": "123",
    "serverId": "44895C63E3FF4013924CEF31519ABE7B"
  },
  "productKey": "test01",
  "deviceName": "test_02"
}
```

参数说明如下：

参数	类型	说明
deviceConfig	Object	设备连接信息
productKey	String	产品ID
deviceName	String	设备名称
displayNamePath	String	展示名称
serverId	String	关联的通道ID

## 12.7 设备禁用、删除

网关类设备可以禁用和删除子设备。

### 设备禁用

下行

- TOPIC : /sys/{productKey}/{deviceName}/thing/disable
- REPLY TOPIC : /sys/{productKey}/{deviceName}/thing/disable\_reply

设备禁用提供设备侧的通道能力，云端使用异步的方式推送该消息，设备订阅该Topic。具体禁用功能适用于网关类型设备，网关可以使用该功能禁用相应的子设备。

### Alink请求数据格式

```
{
  "id": "123",
  "version": "1.0",
  "params": {},
  "method": "thing.disable"
}
```

### Alink响应数据格式

```
{
  "id": "123",
  "code": 200,
  "data": {}
}
```

### 参数说明

参数	取值	说明
id	Long	消息ID号，保留值
version	String	协议版本号，目前协议版本1.0
params	Object	请求参数，为空即可
method	String	请求方法
code	Integer	结果信息，具体参考 <a href="#">设备端通用code</a>

### 恢复禁用

#### 下行

- TOPIC : /sys/{productKey}/{deviceName}/thing/enable
- REPLY TOPIC : /sys/{productKey}/{deviceName}/thing/enable\_reply

设备恢复禁用提供设备侧的通道能力，云端使用异步的方式推送消息，设备订阅该topic。具体恢复禁用功能适用于网关类型设备，网关可以使用该功能恢复被禁用的子设备。

### Alink请求数据格式

```
{
  "id": "123",
  "version": "1.0",
  "params": {},
  "method": "thing.enable"
}
```

```
}
```

### Alink响应数据格式

```
{
  "id": "123",
  "code": 200,
  "data": {}
}
```

### 参数说明

参数	取值	说明
id	Long	消息ID号，保留值
version	String	协议版本号，目前协议版本1.0
params	Object	请求参数，为空即可
method	String	请求方法
code	Integer	结果信息，具体参考设备端通用code

## 删除设备

### 下行

- TOPIC : /sys/{productKey}/{deviceName}/thing/delete
- REPLY TOPIC : /sys/{productKey}/{deviceName}/thing/delete\_reply

删除设备提供设备侧的通道能力，云端使用异步的方式推送消息，设备订阅该topic。具体删除设备的功能适用于网关类型设备，网关可以使用该功能删除相应的子设备。

### Alink请求数据格式

```
{
  "id": "123",
  "version": "1.0",
  "params": {},
  "method": "thing.delete"
}
```

### Alink响应数据格式

```
{
  "id": "123",
  "code": 200,
  "data": {}
}
```

```
}
```

#### 参数说明

参数	取值	说明
id	Long	消息ID号，保留值
version	String	协议版本号，目前协议版本1.0
params	Object	请求参数，为空即可
method	String	请求方法
code	String	结果信息，具体参考设备端通用code

## 12.8 设备标签

部分设备信息，如厂商、设备型号等，展示用的静态扩展信息，可以保存为设备标签。

#### 标签信息上报

上行

- TOPIC : /sys/{productKey}/{deviceName}/thing/deviceinfo/update
- REPLY TOPIC : /sys/{productKey}/{deviceName}/thing/deviceinfo/update\_reply

#### Alink请求数据格式

```
{
  "id": "123",
  "version": "1.0",
  "params": [
    {
      "attrKey": "Temperature",
      "attrValue": "36.8"
    }
  ],
  "method": "thing.deviceinfo.update"
}
```

#### Alink响应数据格式

```
{
  "id": "123",
  "code": 200,
  "data": {}
}
```

参数说明：

参数	取值	说明
id	Long	消息ID号，保留值
version	String	协议版本号，目前协议版本1.0
params	Object	请求参数 params元素个数不超过200个
method	String	请求方法
attrKey	String	标签的名称 <ul style="list-style-type: none"> <li>长度不超过100字节</li> <li>仅允许字符集为[a-z, A-Z, 0-9]和下划线</li> <li>首字符必须是字母或者下划线</li> </ul>
attrValue	String	标签的值
code	Integer	结果信息，200表示成功

#### 错误码

错误码	消息	描述
460	request parameter error	请求参数错误
6100	device not found	设备不存在

#### 删除标签信息

##### 上行

- TOPIC : /sys/{productKey}/{deviceName}/thing/deviceinfo/delete
- REPLY TOPIC : /sys/{productKey}/{deviceName}/thing/deviceinfo/delete\_reply

##### Alink请求数据格式

```
{
  "id": "123",
  "version": "1.0",
  "params": [
    {
      "attrKey": "Temperature"
    }
  ],
  "method": "thing.deviceinfo.delete"
```

```
}
```

### Alink响应数据格式

```
{
  "id": "123",
  "code": 200,
  "data": {}
}
```

参数说明：

参数	取值	说明
id	Long	消息ID号，保留值
version	String	协议版本号，目前协议版本1.0
params	Object	请求参数
method	String	请求方法
attrKey	String	标签的名称 <ul style="list-style-type: none"><li>长度不超过100字节</li><li>仅允许字符集为[a-z, A-Z, 0-9]和下划线</li><li>首字符必须是字母或者下划线</li></ul>
attrValue	String	标签的值
code	Integer	结果信息，200表示成功

错误信息

错误码	消息	描述
460	request parameter error	请求参数错误
6100	device not found	设备不存在

## 12.9 TSL模板

设备可以通过上行请求获取设备的TSL模板。

- TOPIC : /sys/{productKey}/{deviceName}/thing/dsltemplate/get
- REPLY TOPIC : /sys/{productKey}/{deviceName}/thing/dsltemplate/get\_reply

### Alink请求数据格式

```
{
  "id": "123",
  "version": "1.0",
  "params": {},
  "method": "thing.dsltemplate.get"
}
```

### Alink响应数据格式

```
{
  "id": "123",
  "code": 200,
  "data": {
    "schema": "https://iot-ssl.oss-cn-shanghai.aliyuncs.com/schema.json",
    "link": "/sys/1234556554/airCondition/thing/",
    "profile": {
      "productKey": "1234556554",
      "deviceName": "airCondition"
    },
    "properties": [
      {
        "identifier": "fan_array_property",
        "name": "风扇数组属性",
        "accessMode": "r",
        "required": true,
        "dataType": {
          "type": "array",
          "specs": {
            "size": "128",
            "item": {
              "type": "int"
            }
          }
        }
      }
    ]
  },
  "events": [
    {
      "identifier": "alarm",
      "name": "alarm",
      "desc": "风扇警报",
      "type": "alert",
      "required": true,
      "outputData": [
        {
          "identifier": "errorCode",
          "name": "错误码",
          "dataType": {
            "type": "text",
            "specs": {
              "length": "255"
            }
          }
        }
      ]
    }
  ],
  "method": "thing.event.alarm.post"
}
```



```

],
"services": [
  {
    "identifier": "timeReset",
    "name": "timeReset",
    "desc": "校准时间",
    "inputData": [
      {
        "identifier": "timeZone",
        "name": "时区",
        "dataType": {
          "type": "text",
          "specs": {
            "length": "512"
          }
        }
      }
    ]
  },
  {
    "identifier": "curTime",
    "name": "当前的时间",
    "dataType": {
      "type": "date",
      "specs": {}
    }
  }
],
"method": "thing.service.timeReset"
},
{
  "identifier": "set",
  "name": "set",
  "required": true,
  "desc": "属性设置",
  "method": "thing.service.property.set",
  "inputData": [
    {
      "identifier": "fan_int_property",
      "name": "风扇整数型属性",
      "accessMode": "rw",
      "required": true,
      "dataType": {
        "type": "int",
        "specs": {
          "min": "0",
          "max": "100",
          "unit": "g/ml",
          "unitName": "毫升"
        }
      }
    }
  ]
},
{
  "identifier": "get",
  "name": "get",
  "required": true,
  "desc": "属性获取",
  "method": "thing.service.property.get",

```

```
"inputData": [
  "array_property",
  "fan_int_property",
  "batch_enum_attr_id",
  "fan_float_property",
  "fan_double_property",
  "fan_text_property",
  "fan_date_property",
  "batch_boolean_attr_id",
  "fan_struct_property"
],
"outputData": [
  {
    "identifier": "fan_array_property",
    "name": "风扇数组属性",
    "accessMode": "r",
    "required": true,
    "dataType": {
      "type": "array",
      "specs": {
        "size": "128",
        "item": {
          "type": "int"
        }
      }
    }
  }
]
}
```

参数说明：

参数	取值	说明
id	Long	消息ID号，保留值
version	String	协议版本号，目前协议版本1.0
params	Object	为空即可
method	String	请求方法
productKey	String	产品的Key，示例中取值为1234556554
deviceName	String	设备名称，示例中取值为airCondition
data	Object	物的TSL描述，具体参考 <a href="#">概述</a>

错误码

错误码	消息	描述
460	request parameter error	请求参数错误
6321	tsl: device not exist in product	设备不存在

## 12.10 固件升级

物联网平台提供固件升级与管理服务。本文介绍固件升级数据上下行Topic和Alink协议下，固件升级的上下行数据格式，包括设备上报固件版本、物联网平台推送固件信息、设备上报升级进度和设备请求获取最新固件信息。

关于固件升级开发设置流程，请参考[设备OTA开发](#)和用户指南[固件升级](#)。

### 设备上报固件版本

数据上行

- Topic : /ota/device/inform/\${YourProductKey}/\${YourDeviceName}

设备通过这个Topic上报当前使用的固件版本信息。

Alink请求数据格式

```
{
  "id": 1,
  "params": {
    "version": "1.0.1"
  }
}
```

参数说明

参数	取值	说明
id	Long	消息ID号，保留值。
version	String	设备固件的版本信息。

### 物联网平台推送固件信息

数据下行

- Topic : /ota/device/upgrade/\${YourProductKey}/\${YourDeviceName}

物联网平台通过这个Topic推送固件信息，设备订阅该Topic可以获得固件信息。

## Alink请求数据格式

```
{
  "code": "1000",
  "data": {
    "size": 432945,
    "version": "2.0.0",
    "url": "https://iotx-ota-pre.oss-cn-shanghai.aliyuncs.com/nopoll_0.4.4.tar.gz?Expires=1502955804&OSSAccessKeyId=XXXXXXXXXXXXXXXXXXXX&Signature=XfgJu7P6DWWejstKJgXJEH0qAKU%3D&security-token=CAISuQJlq6Ft5B2yfSjIpK6MGsyNlJx5jo6mVnfBglIPTvlvt5D50Tz2IHtIf3NpAusdsv03nWxT7v4flqFyTINVAEvYZJOPKGrGR0DzDbDasumZsJbo4f%2FMQBqEaXPS2MvVfJ%2BzLrf0ceusbfBpJzJ6xaCAGxypQ12iN%2B%2Fr6%2F5gdc9FcQSkL0B8ZrFsKxBltDUOfbIKP%2BpKWSKuGfLCldysQcOlwEP4K%2BkkMqH8Uic3h%2Boy%2BgJt8H2PpHhd9NhXuV2WMzn2%2FdtJOiTkxr7ARasaBqhelc4zqA%2FPPlWgAKvkXba7aIoo01fv4jN5JXQfAU8KLO8tRjofHWmojNzBJAAPpYSSy3Rvr7m5efQrrybY1lLO6iZy%2BVio2VSZDxshI5Z3McKARWct06MWV9ABA2TTXXOi40B0xuq%2B3JGoABXC54Tolo7%2F1wTLTScUqzzeIiXVOK8CfNOKfTucMGHkeYeCdFkm%2FkADhXAnrnGf5a4FbmKMQph2cKsr8y8UfWLC6IzvJsClXTnbJBMeuWIqo5zIynSlpm7gf%2F9N3hVc6%2BEeIk0xf12tycsUpbL2FoaGk6BAF8hWSWYUXsv59d5Uk%3D",
    "md5": "93230c3bde425a9d7984a594ac55eale",
    "sign": "93230c3bde425a9d7984a594ac55eale",
    "signMethod": "Md5"
  },
  "id": 1507707025,
  "message": "success"
}
```

## 参数说明

参数	取值	说明
id	Long	消息ID号，保留值。
message	String	结果信息。
code	String	状态码。
version	String	设备固件的版本信息。
size	Long	固件大小，单位：字节。
url	String	固件在对象存储（OSS）上的存储地址。
sign	String	固件签名。
signMethod	String	签名方法，目前支持Md5，Sha256两种签名方法。
md5	String	作为保留字段，兼容老的设备信息。当签名方法为Md5时，除了会给sign赋值外还会给md5赋值。

## 设备上报升级进度

### 数据上行

- Topic: /ota/device/progress/\${YourProductKey}/\${YourDeviceName}

固件升级过程中，设备可以通过这个Topic上报固件升级的进度百分比。

### Alink请求数据格式

```
{
  "id": 1,
  "params": {
    "step": "-1",
    "desc": "固件升级失败，请求不到固件信息"
  }
}
```

### 参数说明

参数	取值	说明
id	Long	消息ID号，保留值。
step	String	固件升级进度信息。 取值范围： <ul style="list-style-type: none"><li>[1，100] 之间的数字：表示升级进度百分比。</li><li>-1：表示升级失败。</li><li>-2：表示下载失败。</li><li>-3：表示校验失败。</li><li>-4：表示烧写失败。</li></ul>
desc	String	当前步骤的描述信息。如果发生异常，此字段可承载错误信息。

## 设备请求固件信息

### 数据上行

- Topic：/ota/device/request/\${YourProductKey}/\${YourDeviceName}

设备通过这个Topic主动请求固件信息。

### Alink请求数据格式

```
{
  "id": 1,
```

```
"params": {  
  "version": "1.0.1"  
}
```

参数说明

参数	取值	说明
id	Long	消息ID号，保留值
version	String	设备固件的版本信息

物联网平台收到设备请求后，响应请求。物联网平台响应数据格式如下：

- 下发固件信息。返回数据格式如下：

```
{  
  "code": "1000",  
  "data": {  
    "size": 93796291,  
    "sign": "f8d85b250d4d787a9f483d89a9747348",  
    "version": "1.0.1.9.20171112.1432",  
    "url": "https://the_firmware_url",  
    "signMethod": "Md5",  
    "md5": "f8d85b250d4d787a9f483d89a9747348"  
  },  
  "id": 8758548588458,  
  "message": "success"  
}
```

- 无固件信息下发。返回数据格式如下：

```
: {  
  "code": 500,  
  "message": "none upgrade operation of the device."  
}
```

返回参数说明，请参见物联网平台推送固件信息章节中的[参数说明](#)。

## 12.11 远程配置

本文档介绍设备主动请求配置信息和平台推送配置信息的Topic及Alink数据格式。远程配置的具体使用方法，请参考用户指南[远程配置](#)文档。

设备主动请求配置信息

上行

- Topic：/sys/{productKey}/{deviceName}/thing/config/get
- Reply topic：/sys/{productKey}/{deviceName}/thing/config/get\_reply

## Alink请求数据格式

```
{
  "id": 123,
  "version": "1.0",
  "params": {
    "configScope": "product",
    "getType": "file"
  },
  "method": "thing.config.get"
}
```

## Alink响应数据格式

```
{
  "id": "123",
  "version": "1.0",
  "code": 200,
  "data": {
    "configId": "123dagdah",
    "configSize": 1234565,
    "sign": "123214adfadgadg",
    "signMethod": "Sha256",
    "url": "https://iotx-config.oss-cn-shanghai.aliyuncs.com/nopoll_0.4.4.tar.gz?Expires=1502955804&OSSAccessKeyId=XXXXXXXXXXXXXXXXXXXX&Signature=XfgJu7P6DWWejstKJgXJEH0qAKU%3D&security-token=CAISuQJlq6Ft5B2yfSjIpK6MGsyNlJx5jo6mVnfBgLIPTvlvt5D50Tz2IHtIf3NpAusdsv03nWxT7v4flqFyTINVAEvYZJOPKGrGR0DzDbDasumZsJbo4f%2FMQBqEaXPS2MvVfJ%2BzLrf0ceusbfbpjzJ6xaCAGxypQ12iN%2B%2Fr6%2F5gdc9FcQSkL0B8ZrFsKxBltDUROFbIKP%2BpKWSKuGfLCldysQcOlwEP4K%2BkkMqH8Uic3h%2Boy%2BgJt8H2PpHhd9NhXuV2WMzn2%2FdtJOiTkxr7ARasaBqhelc4zqA%2FPPlWgAKvkXba7aIoo01fV4jN5JXQfAU8KLO8tRjofHWmojNzBJAAPPYSSy3Rvr7m5efQrrybY1lL06iZy%2BVio2VSZDxshI5Z3McKARWct06MWV9ABA2TTXXOi40BOxuq%2B3JGoABXC54TOlo7%2F1wTLTsCUqzzeIiXVOK8CfNokfTucMGHkeYeCdFkm%2FkADhXAnrnGf5a4FbmKMQph2cKsr8y8UfWLC6IzvJsClXTnbJBMeuWIqo5zIynSlpm7gf%2F9N3hVc6%2BEeIk0xfl2tycsUpbL2FoaGk6BAF8hWSWYUXsv59d5Uk%3D",
    "getType": "file"
  }
}
```

## 参数说明

参数	类型	说明
id	Long	消息ID号，保留值。
version	String	协议版本号，目前协议版本1.0。
configScope	String	配置范围，目前只支持产品维度配置。取值：product。
getType	String	获取配置类型。目前支持文件类型，取值：file。
configId	String	配置的ID。

参数	类型	说明
configSize	Long	配置大小，按字节计算。
sign	String	签名。
signMethod	String	签名方法，仅支持Sha256。
url	String	配置的OSS地址。
code	Integer	结果码。返回200表示成功，返回其他状态码，表示失败。

#### 错误码

错误码	消息	描述
6713	thing config function is not available	产品的远程配置功能不可用，需要在控制台，远程配置页面打开配置开关。
6710	no data	没有配置的数据。

#### 配置推送

##### 下行

- Topic : /sys/{productKey}/{deviceName}/thing/config/push
- Reply topic : /sys/{productKey}/{deviceName}/thing/config/push\_reply

设备订阅该Topic后，用户在物联网控制台批量推送配置信息时，物联网平台采用异步推送方式向设备推送信息。设备返回的结果，可以通过数据流转获取，数据流转Topic为/sys/{productKey}/{deviceName}/thing/downlink/reply/message。

此时，您可以配置[规则引擎](#)，将配置推送后设备返回的结果转发至其它目的云产品。规则引擎设置示例如下：



编写SQL

×

\* 规则查询语句 :

SELECT deviceName() as deviceName FROM "/sys/a1Cs40ZnKII/LightSensor/thing/downlink"

\* 字段 :

deviceName() as deviceName

\* Topic :

sys

光照感应器

LightSensor

/thing/down... ^

条件 :

可以使用规则引擎函数,例如:deviceName()=mydevice

Q

/thing/event/property/post

✓ /thing/downlink/reply/message

确认

取消

Alink请求数据格式 :

```
{
  "id": "123",
  "version": "1.0",
  "params": {
    "configId": "123dagdah",
    "configSize": 1234565,
    "sign": "123214adfadgadg",
    "signMethod": "Sha256",
    "url": "https://iotx-config.oss-cn-shanghai.aliyuncs.com/nopoll_0
.4.4.tar.gz?Expires=1502955804&OSSAccessKeyId=XXXXXXXXXXXXXXXXXXXX
&Signature=XfgJu7P6DWWejstKJgXJEH0qAKU%3D&security-token=CAISuQJlq6
Ft5B2yfSjIpK6MGsyN1Jx5jo6mVnfBgLIPTvlvt5D50Tz2IHtIf3NpAusdsv03nWxT7v4f
lqFYtINVAEvYZJOPKGrGR0DzDbDasumZsJbo4f%2FMQBqEaXPS2MvVfJ%2BzLrf0ceu
sbFbpjzJ6xaCAGxypQ12iN%2B%2Fr6%2F5gdc9FcQSkL0B8ZrFsKxBltDUOfbIKP%
2BpKWSKuGfLCl dysQcOlwEP4K%2BkkMqH8Uic3h%2Boy%2BgJt8H2PpHhd9NhXuV2WMzn2
%2FdtJOiTkxr7ARasaBqhelc4zqA%2FPPlWgAKvkXba7aIoo01fv4jN5JXQfAU8KLO8tR
jofHWmojNzBJAAPpYSSy3Rvr7m5efQrrybY1lLO6iZy%2BVio2VSZDxshi5Z3McK
ARWct06MWV9ABA2TTXXOi40B0xuq%2B3JGoABXC54Tolo7%2F1wTLTsCUqzzeIiXVOK
8CfNOKfTucMGHkeYeCdFkm%2FkADhXAnrnGf5a4FbmKMQph2cKsr8y8UfWLC6Iz
vJsClXTnbJBMeuWIqo5zIynSlpm7gf%2F9N3hVc6%2BEEIk0xfl2tycsUpbL2
FoaGk6BAF8hWSWYUXsv59d5Uk%3D",
    "getType": "file"
  },
  "method": "thing.config.push"
}
```

Alink响应数据格式

```
{
```

```

    "id": "123",
    "code": 200,
    "data": {}
  }

```

#### 参数说明

参数	类型	说明
id	Long	消息ID号，保留值。
version	String	协议版本号，目前协议版本1.0。
configScope	String	配置范围，目前只支持产品维度配置。取值：product。
getType	String	获取配置类型，目前支持文件类型，取值：file。
configId	String	配置的ID。
configSize	Long	配置大小，按字节计算。
sign	String	签名。
signMethod	String	签名方法，仅支持sha256。
url	String	配置的OSS地址。
code	Integer	结果信息，具体参考设备端通用code。

## 12.12 设备端通用code

设备通用code信息，用于表达云端下行推送时设备侧业务处理的返回结果。

错误码	消息	描述
200	success	请求成功。
400	request error	内部服务错误，处理时发生内部错误
460	request parameter error	请求参数错误，设备入参校验失败
429	too many requests	请求过于频繁，设备端处理不过来时可以使用
100000-110000	自定义的错误信息	从100000到110000的错误码用于设备自定义错误信息，和云端错误信息加以区分