阿里云
物联网平台

用户指南

文档版本：20180807

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**Ctrl** + **A**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**。
`courier` 字体	命令。	执行 `cd /d C:/windows` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *`Instance_ID`*
[]或者[a|b]	表示可选项，至多选择一个。	`ipconfig` *`[-all|-t]`*
{}或者{a|b}	表示必选项，至多选择一个。	`swich` *`{stand | slave}`*

目录

1 账号与登录

本章将详细介绍IoT控制台账号、登录的操作。

1.1 使用阿里云主账号登录控制台

阿里云主账号具有该账号下所有资源的完全操作权限，并且可以修改账号信息。

使用主账号登录 IoT 控制台

使用阿里云主账号登录物联网控制台，便对物联网平台的所有操作具有完全权限。

1. 访问阿里云官网。
2. 单击控制台。
3. 使用阿里云账号和密码登录。

> 说明：
> 若忘记账号或密码，请单击登录框中**忘记会员名**或**忘记密码**，进入账号或密码找回流程。

4. 在控制台中，单击**产品与服务**，页面显示所有阿里云产品和服务名称。
5. 搜索物联网平台，并单击搜索结果中的**物联网平台**产品名，进入**物联网控制台**。

> 说明：
> 如果您还没有开通物联网平台服务，**物联网控制台**主页会展示相关提示，您只需单击**立即开通**便可快速开通物联网平台服务。

进入**物联网控制台**后，您便可以进行产品管理、设备管理、规则管理等操作。

使用主账号创建访问控制

因为主账号具有账号的完全权限，主账号泄露会带来极严重的安全隐患。因此，若需要授权其他人访问您的阿里云资源，请勿将您的阿里云账号及密码直接泄露出去。应该通过访问控制 RAM 创建子账号，并给子账号授予需要的访问权限。非账号所有者或管理员的其他人通过子账号访问资源。有关子账号访问的具体方法，请参见子账号访问和自定义权限。

1.2 RAM授权管理

本章节将详细介绍物联网平台账号权限控制相关事宜。

1.2.1 RAM 和 STS 介绍

RAM 和 STS 是阿里云提供的权限管理系统。了解 RAM 和 STS 的详情，请参见访问控制产品帮助文档。

RAM 的主要作用是控制账号系统的权限。通过使用 RAM，创建、管理子账号，并通过给子账号授予不同的权限，控制子账号对资源的操作权限。

STS 是一个安全凭证（Token）的管理系统，为阿里云子账号（RAM 用户）提供短期访问权限管理。通过 STS 来完成对临时用户的访问授权。

背景介绍

RAM 和 STS 解决的一个核心问题是如何在不暴露主账号的 AccessKey 的情况下，安全地授权他人访问。因为一旦主账号的 AccessKey 被泄露，会带来极大的安全风险：获得该账号 AccessKey 的人可任意操作该账号下所有的资源，盗取重要信息等。

RAM 提供的是一种长期有效的权限控制机制。通过创建子账号，并授予子账号相应的权限，将不同的权限分给不同的用户。子账号的 AccessKey 也不能泄露。即使子账号泄露也不会造成全局的信息泄露。一般情况下，子账号长期有效。

相对于 RAM 提供的长效控制机制，STS 提供的是一种临时访问授权。通过调用 STS，获得临时的 AccessKey 和 Token。可以将临时 AccessKey 和 Token 发给临时用户，用来访问相应的资源。从 STS 获取的权限会受到更加严格的限制，并且具有时间限制。因此，即使出现信息泄露的情况，影响相对较小。

使用场景示例，请参见《使用示例》。

基本概念

使用 RAM 和 STS 涉及以下基本概念：

- 子账号：在 RAM 控制台中，创建的用户，每个用户即一个子账号。创建时或创建成功后，均可为子账号生成独立的 AccessKey。创建后，需为子账号配置密码和权限。使用子账号，可以进行已获授权的操作。子账号可以理解为具有某种权限的用户，可以被认为是一个具有某些权限的操作发起者。
- 角色（Role）：表示某种操作权限的虚拟概念，但是没有独立的登录密码和 AccessKey。子账号可以扮演角色。扮演角色时，子账号拥有的权限是该角色的权限。
- 授权策略（Policy）：用来定义权限的规则，比如允许子账号用户读取或者写入某些资源。

- 资源（Resource）：代表子账号用户可访问的云资源，比如表格存储所有的 Instance、某个 Instance 或者某个 Instance 下面的某个 Table 等。

子账号和角色可以类比为个人和其身份的关系。如，某人在公司的角色是员工，在家里的角色是父亲。同一人在不同的场景扮演不同的角色。在扮演不同角色的时候，拥有对应角色的权限。角色本身并不是一个操作的实体，只有用户扮演了该角色之后才是一个完整的操作实体。并且，一个角色可以被多个不同的用户同时扮演。

使用示例

为避免阿里云账号的 AccessKey 泄露而导致安全风险，某阿里云账号管理员使用 RAM 创建了两个子账号，分别命名为 A 和 B ，并为 A 和 B 生成独立的 AccessKey。A 拥有读权限，B 拥有写权限。 管理员可以随时在 RAM 控制台取消子账号用户的权限。

现在因为某些原因，需要授权给其他人临时访问物联网平台接口的权限。这种情况下，不能直接把 A 的 AccessKey 透露出去，而应该新建一个角色 C，并给这个角色授予读取物联网平台接口的权限。但请注意，目前角色 C 还无法直接使用。因为并不存在对应角色 C 的 AccessKey，角色 C 仅是一个拥有访问物联网平台接口权限的虚拟实体。

需调用 STS 的 AssumeRole 接口，获取访问物联网平台接口的临时授权。在调用 STS 的请求中，RoleArn 的值需为角色 C 的 Arn。如果调用成功，STS 会返回一个临时的 AccessKeyId、AccessKeySecret 、和 SecurityToken 作为访问凭证（凭证的过期时间，在调用 AssumeRole 的请求中指定）。将这个凭证发给需要访问的用户，该用户就可以获得访问物联网平台接口的临时权限。

为什么 **RAM** 和 **STS** 的使用这么复杂？

虽然 RAM 和 STS 的概念和使用比较复杂，但这是为了账号的安全性和权限控制的灵活性而牺牲了部分易用性。

将子账号和角色分开，主要是为了将执行操作的实体和代表权限集合的虚拟实体分开。如果某用户需要使用多种权限，比如读/写权限，但是实际上每次操作只需要其中的一部分权限，那么就可以创建两个角色。这两个角色分别具有读或写权限。然后，创建一个可以扮演这两个角色的用户子账号。当用户需要读权限的时候，就可以扮演其中拥有读权限的角色；使用写权限的时候同理。这样可以降低每次操作中权限泄露的风险。而且，通过扮演角色，可以将角色权限授予其他用户，更加方便了协同使用。

STS 对权限的控制更加灵活。如按照实际需求设置有效时长。但是，如果需要一个长期有效的临时访问凭证，则可以只适用 RAM 子账号管理功能，而无需使用 STS。

在后面的章节中，我们将提供一些 RAM 和 STS 的使用指南和使用示例。如果您需要了解更多 RAM 和 STS 的代码详情，请参见 *API 参考#RAM#*和 *API 参考#STS#* 。

1.2.2 自定义权限

权限指在某种条件下 ，允许 (Allow) 或 拒绝 (Deny) 对某些资源执行某些操作。

权限的载体是授权策略。自定义权限，即在自定义授权策略时定义某些权限。在 RAM 控制台，策略管理页面，单击新建授权策略开始创建自定义授权策略。创建自定义授权策略时，请选择模板为空白模板。

授权策略是 JSON 格式的字符串，需包含以下参数：

- **`Action`**：表示要授权的操作。IoT 操作都以*`iot:`* 开头。定义方式和示例，请参见本文档中Action 定义。
- **`Effect`** ： 表示授权类型，取值：`Allow`、`Deny`。
- **`Resource`** ：表示操作对象。资源命名格式如下：

  `acs:<service-name>:<region>:<account-id>:<relative-id>`
- **`Condition`** ：表示鉴权条件。详细信息，请参见本文档中 Condition 定义。

Action 定义

`Action`是 API 的名称。在创建 IoT 的授权策略时，每个 Action 前缀均为*`iot:`*，多个 Action 以逗号分隔。并且，支持使用星号通配符。IoT API 名称定义，请参见*IoT API* 授权映射表 。

下面介绍一些典型的 Action 定义示例。

- 定义单个 API。

```
"Action": "iot:CreateProduct"
```

- 定义多个API。

```
"Action": [
"iot:UpdateProduct",
"iot:QueryProduct"
]
```

- 定义所有只读 API。

```
{
"Version": "1",
"Statement": [
{
```

```
"Action": [
"rds:DescribeDBInstances",
"rds:DescribeDatabases",
"rds:DescribeAccounts",
"rds:DescribeDBInstanceNetInfo"
],
"Resource": "*",
"Effect": "Allow"
},
{
"Action": "ram:ListRoles",
"Effect": "Allow",
"Resource": "*"
},
{
"Action": [
"mns:ListTopic"
],
"Resource": "*",
"Effect": "Allow"
},
{
"Action": [
"dhs:ListProject",
"dhs:ListTopic",
"dhs:GetTopic"
],
"Resource": "*",
"Effect": "Allow"
},
{
"Action": [
"ots:ListInstance",
"ots:ListTable",
"ots:DescribeTable"
],
"Resource": "*",
"Effect": "Allow"
},
{
"Action": [
"log:ListShards",
"log:ListLogStores",
"log:ListProject"
],
"Resource": "*",
"Effect": "Allow"
},
{
"Effect": "Allow",
"Action": [
"iot:Query*",
"iot:List*",
"iot:Get*",
"iot:BatchGet*"
],
"Resource": "*"
}
]
```

```
}
```

- 定义所有读写 API。

```
{
"Version": "1",
"Statement": [
{
"Action": [
"rds:DescribeDBInstances",
"rds:DescribeDatabases",
"rds:DescribeAccounts",
"rds:DescribeDBInstanceNetInfo"
],
"Resource": "*",
"Effect": "Allow"
},
{
"Action": "ram:ListRoles",
"Effect": "Allow",
"Resource": "*"
},
{
"Action": [
"mns:ListTopic",
"mns:CreateQueue"
],
"Resource": "*",
"Effect": "Allow"
},
{
"Action": [
"dhs:ListProject",
"dhs:ListTopic",
"dhs:GetTopic"
],
"Resource": "*",
"Effect": "Allow"
},
{
"Action": [
"ots:ListInstance",
"ots:ListTable",
"ots:DescribeTable"
],
"Resource": "*",
"Effect": "Allow"
},
{
"Action": [
"log:ListShards",
"log:ListLogStores",
"log:ListProject"
],
"Resource": "*",
"Effect": "Allow"
},
{
"Effect": "Allow",
"Action": "iot:*",
"Resource": "*"
```

```
}
]
}
```

Condition 定义

目前 RAM 授权策略支持访问 IP 限制、是否通过 HTTPS 访问、是否通过 MFA（多因素认证）访问、访问时间限制等多种鉴权条件。物联网平台的所有 API 均支持这些条件。

访问 IP 限制

访问控制可以限制访问 IoT 的源 IP 地址，并且支持根据网段进行过滤。以下是典型的使用场景示例。

- 限制单个 IP 地址和 IP 网段。例如，只允许 IP 地址为 10.101.168.111 或 10.101.169.111/24 网段的请求访问。

```
{
"Statement": [
{
"Effect": "Allow",
"Action": "iot:*",
"Resource": "*",
"Condition": {
"IpAddress": {
"acs:SourceIp": [
"10.101.168.111",
"10.101.169.111/24"
]
}
}
}
],
"Version": "1"
}
```

- 限制多个 IP 地址。例如，只允许 IP 地址为 10.101.168.111 和 10.101.169.111 的请求访问。

```
{
"Statement": [
{
"Effect": "Allow",
"Action": "iot:*",
"Resource": "*",
"Condition": {
"IpAddress": {
"acs:SourceIp": [
"10.101.168.111",
"10.101.169.111"
]
}
}
}
],
```

```
"Version": "1"
}
```

HTTPS 访问限制

访问控制可以限制是否通过 HTTPS 访问。

示例：限制必须通过 HTTPS 请求访问。

```
{
"Statement": [
{
"Effect": "Allow",
"Action": "iot:*",
"Resource": "*",
"Condition": {
"Bool": {
"acs:SecureTransport": "true"
}
}
}
],
"Version": "1"
}
```

MFA 访问限制

访问控制可以限制是否通过 MFA (多因素认证) 访问。

示例：限制必须通过 MFA 请求访问。

```
{
"Statement": [
{
"Effect": "Allow",
"Action": "iot:*",
"Resource": "*",
"Condition": {
"Bool": {
"acs:MFAPresent ": "true"
}
}
}
],
"Version": "1"
}
```

访问时间限制

访问控制可以限制请求的访问时间，即只允许或拒绝在某个时间点范围之前的请求。

示例：用户可以在北京时间 2019 年 1 月 1 号凌晨之前访问，之后则不能访问。

```
{
"Statement": [
{
"Effect": "Allow",
"Action": "iot:*",
"Resource": "*",
"Condition": {
"DateLessThan": {
"acs:CurrentTime": "2019-01-01T00:00:00+08:00"
}
}
}
],
"Version": "1"
}
```

典型使用场景

结合以上对 Action、Resource 和 Condition 的定义，下面介绍一些典型使用场景的授权策略定义和授权方法。

允许访问的授权策略示例

场景：定义访问 IP 地址为 10.101.168.111/24 网段的用户访问 IoT 的权限，且要求只能在 2019-01-01 00:00:00 之前访问和通过 HTTPS 访问。

```
{
"Statement": [
{
"Effect": "Allow",
"Action": "iot:*",
"Resource": "*",
"Condition": {
"IpAddress": {
"acs:SourceIp": [
"10.101.168.111/24"
]
},
"DateLessThan": {
"acs:CurrentTime": "2019-01-01T00:00:00+08:00"
},
"Bool": {
"acs:SecureTransport": "true"
}
}
}
],
"Version": "1"
}
```

拒绝访问的授权策略示例

场景：拒绝访问 IP 地址为 10.101.169.111 的用户对 IoT 执行读操作。

```
{
"Statement": [
{
"Effect": "Deny",
"Action": [
"iot:Query*",
"iot:List*",
"iot:Get*",
"iot:BatchGet*"
],
"Resource": "*",
"Condition": {
"IpAddress": {
"acs:SourceIp": [
"10.101.169.111"
]
}
}
}
],
"Version": "1"
}
```

授权策略创建成功后，在访问控制 RAM 控制台，用户管理页面，将此权限授予子账号用户。获得授权的子账号用户就可以进行权限中定义的操作。创建子账号和授权操作帮助，请参见子账号访问。

1.2.3 IoT API 授权映射表

IoT API 名称即创建 IoT 授权策略时，可定义为**Action**的值。

IoT API	RAM 授权操作（Action)	资源（Resource）	接口说明
CreateProduct	iot:CreateProduct	*	创建产品
UpdateProduct	iot:UpdateProduct	*	修改产品
QueryProduct	iot:QueryProduct	*	查询产品
GetCats	iot:GetCats	*	获取产品类型信息
QueryProductByUserId	iot:QueryProductByUserId	*	根据用户id查询产品
RegistDevice	iot:RegistDevice	*	创建设备
QueryDevice	iot:QueryDevice	*	批量查询设备
QueryByDeviceId	iot:QueryByDeviceId	*	根据deviceId查询设备

IoT API	RAM 授权操作（Action)	资源（Resource）	接口说明
QueryDeviceByName	iot:QueryDeviceByName	*	根据deviceName查询设备
DeleteDevice	iot:DeleteDevice	*	删除设备
ApplyDeviceWithNamesRequest	iot:ApplyDeviceWithNamesRequest	*	批量创建设备
QueryApplyStatus	iot:QueryApplyStatus	*	查询申请状态
QueryPageByApplyId	iot:QueryPageByApplyId	*	根据申请id分页查询
BatchGetDeviceState	iot:BatchGetDeviceState	*	批量获取设备状态
QueryTopic	iot:QueryTopic	*	查询Topic
StartRule	iot:StartRule	*	启动规则
StopRule	iot:StopRule	*	暂停规则
ListRule	iot:ListRule	*	查询规则列表
GetRule	iot:GetRule	*	查询规则详情
CreateRule	iot:CreateRule	*	创建规则
UpdateRule	iot:UpdateRule	*	修改规则
DeleteRule	iot:DeleteRule	*	删除规则
CreateRuleAction	iot:CreateRuleAction	*	创建规则中的转发数据方法
UpdateRuleAction	iot:UpdateRuleAction	*	修改规则中的转发数据方法
DeleteRuleAction	iot:DeleteRuleAction	*	删除规则中的转发数据方法
GetRuleAction	iot:GetRuleAction	*	获取规则中的转发数据方法
ListRuleActions	iot:ListRuleActions	*	获取规则中的转发数据方法列表
Pub	iot:Pub	*	发布消息

IoT API	RAM 授权操作（Action)	资源（Resource）	接口说明
Sub	iot:Sub	*	订阅消息
Unsub	iot:Unsub	*	取消订阅
ServerOnline	iot:ServerOnline	*	服务端订阅者上线
DeviceGrant	iot:DeviceGrant	*	设备授权
DeviceRevokeByTopic	iot:DeviceRevokeByTopic	*	通过Topic名撤销设备权限
DeviceRevokeById	iot:DeviceRevokeById	*	通过ID撤销设备权限
DevicePermitModify	iot:DevicePermitModify	*	修改设备权限
ListPermits	iot:ListPermits	*	列出设备的权限
RRpc	iot:RRpc	*	发送消息给设备并得到设备响应
CreateProductTopic	iot:CreateProductTopic	*	创建产品Topic类
DeleteProductTopic	iot:DeleteProductTopic	*	删除产品Topic类
QueryProductTopic	iot:QueryProductTopic	*	查询产品Topic类列表
UpdateProductTopic	iot:UpdateProductTopic	*	修改产品Topic类
QueryDeviceTopic	iot:QueryDeviceTopic	*	查询设备Topic列表
DebugRuleSql	iot:DebugRuleSql	*	规则引擎SQL调试

1.2.4 子账号访问

子账号用户可以登录物联网控制台管理您的 IoT 资源，和使用子账号的 AccessKeyId 和AccessKeySecret 调用 IoT API。

您需先创建子账号，并通过授权策略授予子账号访问物联网平台的权限。创建自定义授权策略的方法，请参见自定义权限。

创建子账号

如果您已有子账号，请忽略此操作。

1. 用主账号登录访问控制 *RAM* 控制台。

2. 在左侧导航栏中，单击用户管理。
3. 单击新建用户。
4. 输入用户信息，并勾选为该用户自动生成 AccessKey前的复选框，再单击确定。

   说明：
   单击确定后，弹出保存 AccessKey 的提示。这是下载该子账号的 AccessKey 的唯一机会。请及时保存该 AccessKey 信息，并妥善保管。子账号用户调用 API 时，需传入该 AccessKey 信息。
5. 设置初始登录密码。
   a. 在用户管理页面，找到刚创建的用户名，单击管理，进入用户详情页面。
   b. 单击启用控制台登录。
   c. 为该子账号设置一个初始密码，并勾选要求该账号下次登录成功后重置密码前的复选框，再单击确定。
6. 启用多因素认证设备（可选）。

   在用户详情页面，单击启用虚拟MFA设备。

子账号创建完成后，子账号用户便可通过RAM 用户登录链接登录阿里云官网和控制台。RAM 用户的登录链接，请在访问控制 RAM 控制台的概览页面查看。

但是，在获得授权之前，该子账号无法访问您的阿里云资源。下一步，为子账号授予物联网平台的访问权限。

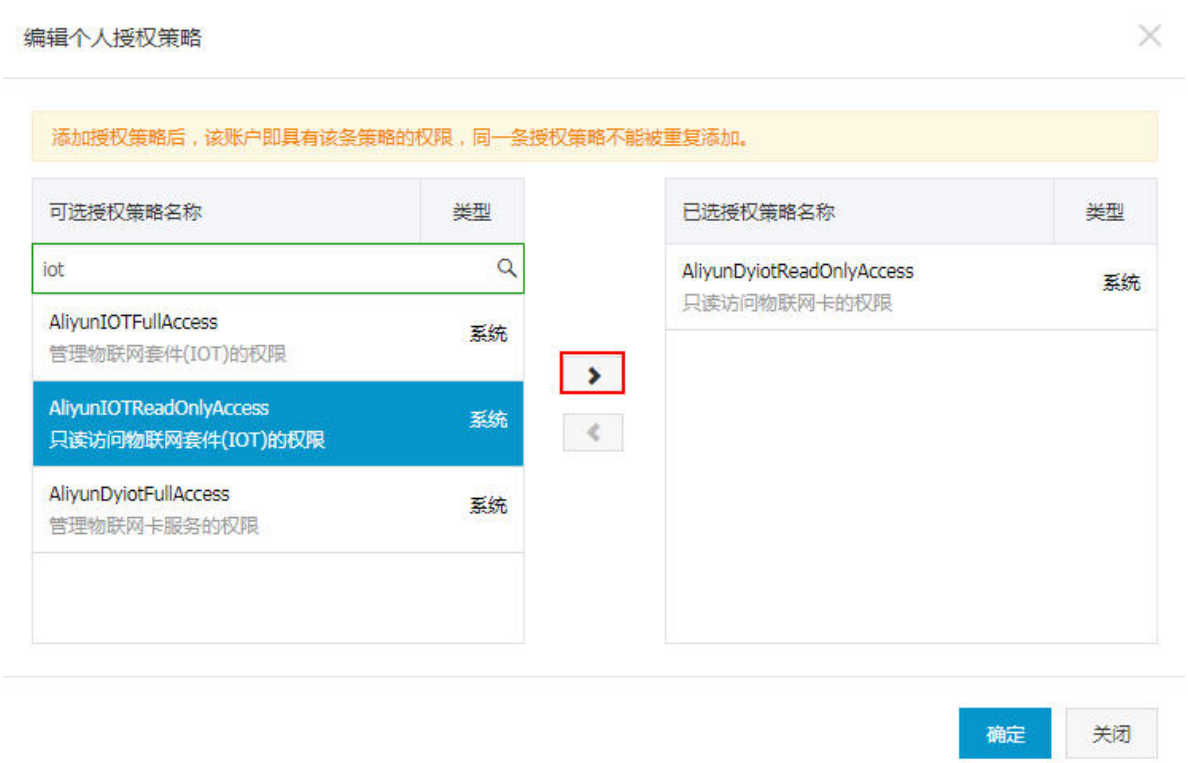
授权子账号访问 IoT

在访问控制 RAM 控制台中，您可以在用户管理页面，为单个子账号进行授权；也可以在群组管理页面，为整个群组授予相同的权限。下面我们以为单个子账号授权为例，介绍授权操作流程。

1. 用主账号登录访问控制 *RAM* 控制台。
2. 在左侧导航栏中，单击用户管理。
3. 找到要授权的子账号的用户名，单击授权操作按钮。
4. 在授权对话框中，选择您要授予该子账号的 IoT 授权策略，单击页面中间向右箭头将选中权限移入已选授权策略名称，再单击确定。

   说明：

如果您要为子账号授予自定义权限，请先创建授权策略。授权策略的创建方法，请参见自定义权限。



授权成功后，子账号用户便可访问授权策略中定义的资源，和进行授权策略中定义的操作。

子账号登录控制台

阿里云主账号登录是从阿里云官网主页直接登录，但是子账号需从**RAM** 用户登录页面登录。

1. 获取**RAM** 用户登录页面的链接地址。

   用主账号登录**访问控制 RAM 控制台**，在**概览**页面查看**RAM** 用户登录链接，并将链接地址分发给子账号的用户。

2. 子账号用户访问**RAM** 用户登录页面，并使用子账号和子账号密码登录。

   说明：

   子账号登录格式为：子用户名称@企业别名，如：username@company-alias。并且，首次登录成功后，需修改登录密码。

3. 单击页面右上角**控制台**按钮，进入管理控制台。
4. 单击**产品与服务**，选择**物联网平台**，即可进入物联网控制台。

子账号用户登录**物联网控制台**后，便可在控制台中，进行已获授权的操作。

1.2.5 进阶使用 STS

STS 权限管理系统是比访问控制（RAM）更为严格的权限管理系统。使用 STS 权限管理系统进行资源访问控制，需通过复杂的授权流程，授予子账号用户临时访问资源的权限。

子账号和授予子账号的权限均长期有效。删除子账号或解除子账号权限，均需手动操作。发生子账号信息泄露后，如果无法及时删除该子账号或解除权限，可能给您的阿里云资源和重要信息带来危险。所以，对于关键性权限或子账号无需长期使用的权限，您可以通过 STS 权限管理系统来进行控制。

图 1-1: 子账号获得临时访问权限的操作流程

步骤一：创建角色

RAM 角色是一种虚拟用户，是承载操作权限的虚拟概念。

1. 使用阿里云主账号登录访问控制 *RAM* 控制台。
2. 单击角色管理 > 新建角色，进入角色创建流程。
3. 选择用户角色。
4. 填写信息类型步骤中，直接使用默认的当前云账号信息，单击下一步。
5. 输入角色名称和备注后，单击创建。
6. 单击关闭或授权。

   如果您已创建要授予该角色的授权策略，则单击 授权，并对角色进行授权。

   如果您还没有创建授权策略，则单击关闭。然后，在策略管理中为该角色新建授权策略。

步骤二：创建角色授权策略

角色授权策略，即定义要授予角色的资源访问权限。

1. 在访问控制 *RAM* 控制台主页，单击策略管理 > 新建授权策略。
2. 选择空白模板。
3. 输入授权策略名称和策略内容，再单击新建授权策略。

授权策略内容的编写方法参考，请单击**授权策略格式定义**查看。

授权策略内容示例：IoT 资源只读权限。

```
{
"Version": "1",
"Statement": [
{
"Action": [
"rds:DescribeDBInstances",
"rds:DescribeDatabases",
"rds:DescribeAccounts",
"rds:DescribeDBInstanceNetInfo"
],
"Resource": "*",
"Effect": "Allow"
},
{
"Action": "ram:ListRoles",
"Effect": "Allow",
"Resource": "*"
},
{
"Action":[
"mns:ListTopic"
],
"Resource": "*",
"Effect": "Allow"
},
{
"Action": [
"dhs:ListProject",
"dhs:ListTopic",
"dhs:GetTopic"
],
"Resource": "*",
"Effect": "Allow"
},
{
"Action": [
"ots:ListInstance",
"ots:ListTable",
"ots:DescribeTable"
],
"Resource": "*",
"Effect": "Allow"
},
{
"Action":[
"log:ListShards",
"log:ListLogStores",
"log:ListProject"
],
"Resource": "*",
"Effect": "Allow"
},
{
"Effect": "Allow",
"Action": [
```

```
"iot:Query*",
"iot:List*",
"iot:Get*",
"iot:BatchGet*"
],
"Resource": "*"
}
]
}
```

授权策略内容示例：IoT 资源读写权限。

```
{
"Version": "1",
"Statement": [
{
"Action": [
"rds:DescribeDBInstances",
"rds:DescribeDatabases",
"rds:DescribeAccounts",
"rds:DescribeDBInstanceNetInfo"
],
"Resource": "*",
"Effect": "Allow"
},
{
"Action": "ram:ListRoles",
"Effect": "Allow",
"Resource": "*"
},
{
"Action":[
"mns:ListTopic"
],
"Resource": "*",
"Effect": "Allow"
},
{
"Action": [
"dhs:ListProject",
"dhs:ListTopic",
"dhs:GetTopic"
],
"Resource": "*",
"Effect": "Allow"
},
{
"Action": [
"ots:ListInstance",
"ots:ListTable",
"ots:DescribeTable"
],
"Resource": "*",
"Effect": "Allow"
},
{
"Action":[
"log:ListShards",
"log:ListLogStores",
"log:ListProject"
```

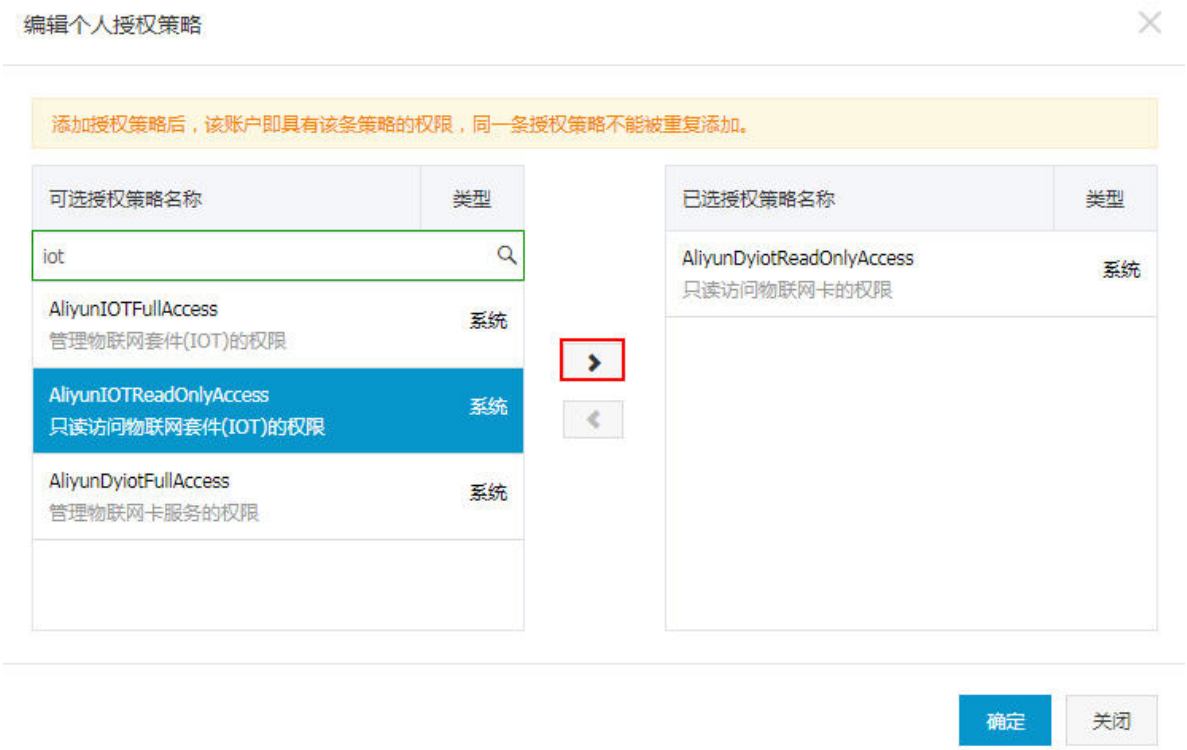
```
],
"Resource": "*",
"Effect": "Allow"
},
{
"Effect": "Allow",
"Action": "iot:*",
"Resource": "*"
}
]
}
```

授权策略创建成功后，您就可以将该授权策略中定义的权限授予角色。

步骤三：为角色授权

角色获得授权后，才具有资源访问权限。

1. 在访问控制 *RAM* 控制台主页，单击角色管理。
2. 找到要授权的角色，单击授权。
3. 在授权对话框中，选择要授予角色的自定义授权策略，单击中间的向右箭头，将选中的授权策略移至已选授权策略名称下，再单击确定。



授权完成后，该角色就具有了授权策略定义的权限。您可以单击该角色对应的管理操作按钮，进入角色详情页，查看该角色的基本信息和权限信息。

下一步，为子账号授予可以扮演该角色的权限。

步骤四：授予子账号角色扮演权限

虽然经过授权后，该角色已拥有了授权策略定义的访问权限，但角色本身只是虚拟用户，需要子账号用户扮演该角色，才能进行权限允许的操作。若任意子账号都可以扮演该角色，也会带来风险，因此只有获得角色扮演权限的子账号用户才能扮演角色。

授权子账号扮演角色的方法：先新建一个**Resource**参数值为角色 ID 的自定义授权策略，然后用该授权策略为子账号授权。

1. 在访问控制 *RAM* 控制台主页，单击策略管理 > 新建授权策略。
2. 选择空白模板。
3. 输入授权策略名称和策略内容，再单击新建授权策略。

   说明：

   授权策略内容中，参数**Resource** 的值需为角色 Arn。在角色管理页面，单击角色对应的管理按钮，进入角色详情页面中，查看角色的 Arn 。

   角色授权策略示例：

```
{
"Version": "1",
"Statement": [
{
"Effect": "Allow",
"Action": "iot:QueryProduct",
"Resource": "角色Arn"
}
]
}
```

4. 授权策略创建成功后，返回访问控制 **RAM** 控制台主页。
5. 单击左侧导航栏中的用户管理按钮，进入子账号管理页面。
6. 找到要授权的子账号，并单击对应的授权按钮。
7. 在授权对话框中，选择刚新建的角色授权策略，单击中间的向右箭头将授权策略移至已选授权策略名称下，再单击确定。

授权完成后，子账号便有了可以扮演该角色的权限，就可以使用 STS 获取扮演角色的临时身份凭证，和进行资源访问。

步骤五：子账号获取临时身份凭证

获得角色授权的子账号用户，可以通过直接调用 API 或使用 SDK 来获取扮演角色的临时身份凭证：AccessKeyId、AccessKeySecret、和 SecurityToken。STS API 和 STS SDK 详情，请参见访问控制文档中*API 参考#STS#*和*SDK 参考#STS#*。

使用 API 和 SDK 获取扮演角色的临时身份凭证需传入以下参数：

- RoleArn：需要扮演的角色 Arn。
- RoleSessionName：临时凭证的名称（自定义参数）。
- Policy：授权策略，即为角色增加一个权限限制。通过此参数限制生成的Token的权限。不指定此参数，则返回的 Token 将拥有指定角色的所有权限。
- DurationSeconds：临时凭证的有效期。单位是秒，最小为 900，最大为 3600，默认值是 3600。
- id 和 secret：指需要扮演该角色的子账号的 AccessKeyId 和 AccessKeySecret。

获取临时身份凭证示例

API 示例：子账号用户通过调用 STS 的**AssumeRole**接口获得扮演该角色的临时身份凭证。

```
https://sts.aliyuncs.com?Action=AssumeRole
&RoleArn=acs:ram::1234567890123456:role/iotstsrole
&RoleSessionName=iotreadonlyrole
&DurationSeconds=3600
&Policy=<url_encoded_policy>
&<公共请求参数>
```

SDK 示例：子账号用户使用 STS 的 Python 命令行工具接口获得扮演该角色的临时身份凭证。

```
$python ./sts.py AssumeRole RoleArn=acs:ram::1234567890123456:role/
iotstsrole RoleSessionName=iotreadonlyrole Policy='{"Version":"1","
Statement":[{"Effect":"Allow","Action":"iot:*","Resource":"*"}]}'
DurationSeconds=3600 --id=id --secret=secret
```

请求成功后，将返回扮演该角色的临时身份凭证：AccessKeyId、AccessKeySecret、和 SecurityToken。

步骤六：子账号临时访问资源

获得扮演角色的临时身份凭证后，子账号用户便可以在调用 SDK 的请求中传入该临时身份凭证信息，扮演角色。

Java SDK 示例：子账号用户在调用请求中，传入临时身份凭证的 AccessKeyId、AccessKeySecret 和 SecurityToken 参数，创建 IAcsClient 对象。

```
IClientProfile  profile = DefaultProfile.getProfile("cn-hangzhou",
AccessKeyId,AccessSecret);
RpcAcsRequest request.putQueryParameter("SecurityToken", Token);
IAcsClient client = new DefaultAcsClient(profile);
AcsResponse response = client.getAcsResponse(request);
```

2 创建产品与设备

本章节将介绍如何使用控制台创建、管理产品与设备。

2.1 基础版

物联网平台基础版适用于具有较强软硬件开发实例，并希望灵活组合阿里云各种云产品来搭建业务系统的开发者。

基础版特点：

- 提供设备与云端稳定的双向通信能力。
- 提供规则引擎，处理数据，连通阿里云产品。
- 提供设备认证、传输加密、多重防护，保障设备安全。

2.1.1 创建产品

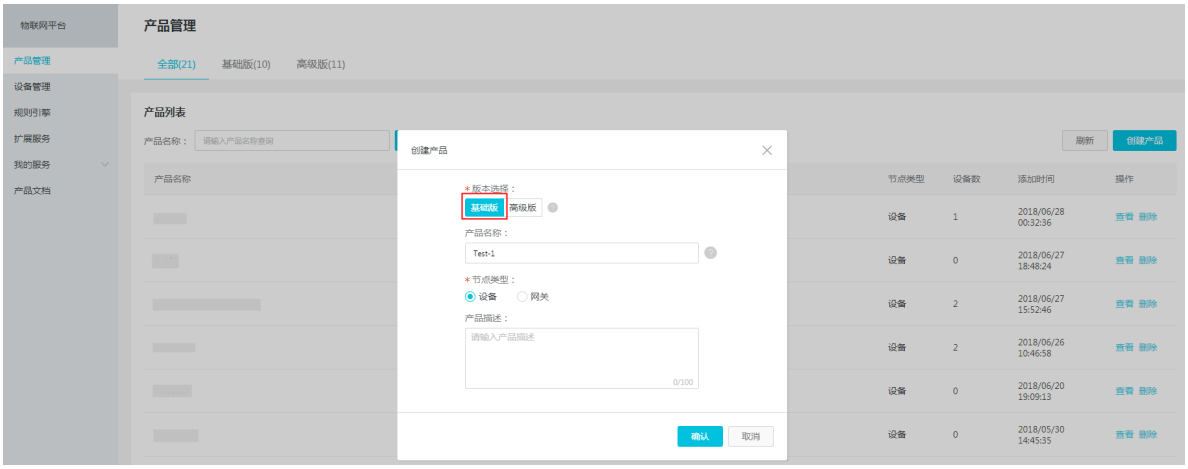
产品是设备的集合，通常是一组具有相同功能定义的设备集合。

背景信息

本文档介绍如何在控制台创建基础版产品。

操作步骤

1. 登录物联网平台控制台。
2. 在产品列表页面，单击创建产品。
3. 选择版本为基础版，输入合法的产品名称，选择合适的节点类型，再单击确定。



输入信息说明：

- 产品名称：为产品命名。产品名称在账号内具有唯一性。例如，可以填写为产品型号。
- 节点类型：设备或网关。
  - 设备：指不能挂载子设备的设备。这种设备可以直连 IoT Hub，也可以作为网关的子设备连接IoT Hub。
  - 网关：指可以挂载子设备的直连设备。网关具有子设备管理模块，维持子设备的拓扑关系，并且可以将拓扑关系同步到云端。

预期结果

产品创建成功后，页面自动跳转回产品列表页面。您可以查看或编辑产品信息。

2.1.2 创建设备

设备需归属于某个产品下。设备可以直接连接物联网平台，也可以作为子设备通过网关连接物联网平台。

背景信息

本文档介绍如何创建基础版设备。

操作步骤

1. 登录物联网控制台。
2. 单击左侧导航栏中设备管理。
3. 在设备管理页面上，选择已创建的基础版产品，再单击添加设备。

4. 在弹出对话框中，输入设备名称，单击确定。

> **说明：**
> 设备名称（即 DeviceName）需在产品内具有唯一性。可作为设备的唯一标识符，用于与 IoT Hub 进行通信。

5. 在添加成功页面，单击一键复制复制设备的 ProductKey、DeviceName、和 DeviceSecret，然后粘贴到文件中，并妥善保管。

ProductKey、DeviceName、 和 DeviceSecret 即设备三元组：

- ProductKey：物联网平台为您创建的产品颁发的全局唯一标识符。
- DeviceName：自定义的设备唯一标识符。用于设备认证和通信。
- DeviceSecret：物联网平台为设备颁发的设备秘钥，用于认证加密，需与DeiceName（或者deviceId）成对使用。

预期结果

设备创建成功，物联网平台会根据设备所属产品的消息通信中的 Topic 类，自动为设备生成对应的 Topic。设备可以基于 Topic 中定义的操作权限进行 Pub/Sub 通信。

2.2 高级版

高级版除了具有基础版的所有功能外，还具有更多设备管理功能，进一步降低机器智能化周期和成本投入，让开发者更聚焦于搭建垂直业务系统。

高级版特点：

- 具有基础版的所有功能。
- 具备更加完整的设备全生命周期管理能力。
- 支持设备模型定义、数据解析、在线调试和远程维护。
- 提供原始数据存储。
- 支持实时获取设备运行状态或历史数据。

2.2.1 创建产品

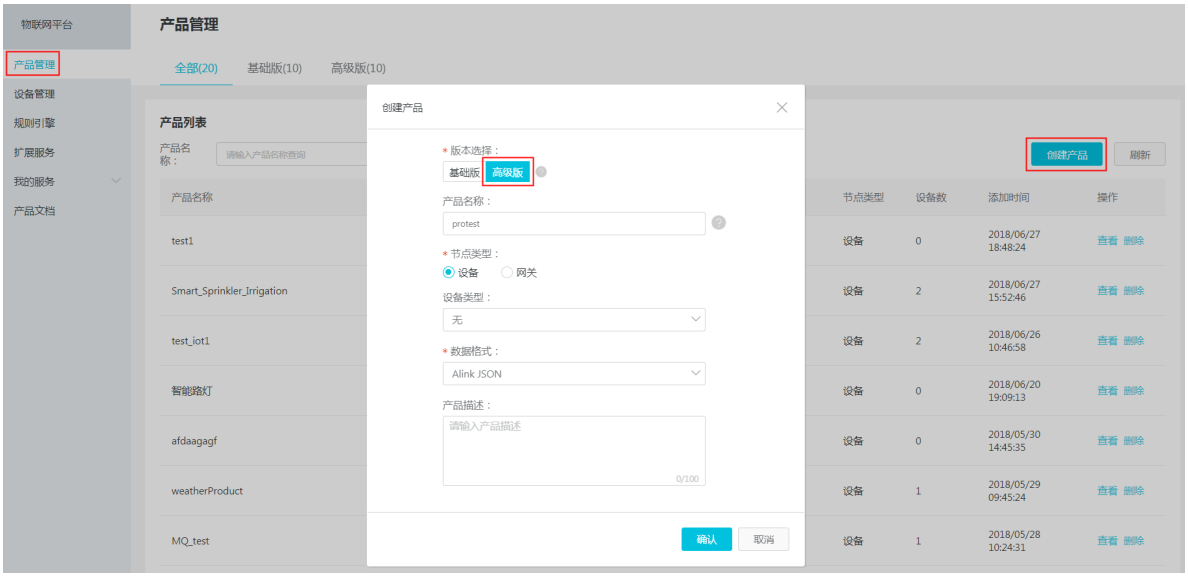
产品是设备的集合，通常是一组具有相同功能定义的设备集合。

背景信息

本文档介绍如何在物联网平台控制台创建高级版产品。

操作步骤

1. 登录物联网平台控制台。
2. 在产品列表页面，单击创建产品。
3. 选择版本为高级版，填写产品名称和选择节点类型、设备类型、和数据格式，然后单击确定。



输入信息说明：

- 产品名称：为产品命名。产品名称在账号内具有唯一性。例如，可以填写为产品型号。
- 节点类型：设备或网关。
  - 设备：指不能挂载子设备的设备。这种设备可以直连 IoT Hub，也可以作为网关的子设备连接IoT Hub。
  - 网关：指可以挂载子设备的直连设备。网关具有子设备管理模块，维持子设备的拓扑关系，并且可以将拓扑关系同步到云端。
- 设备类型：是一组预定义的标准功能模板。阿里云物联网平台已提供了多种产品功能模板，并已预定义相关标准功能。如，电表中已预定义：用电量、电压、电流、总累计量等标准功能。您可以选择某种产品功能模板，在标准功能模板的基础上编辑、修改功能，您还可

以添加更多自定义功能。如果选择无，将不会创建任何标准功能，需您完全自定义该产品的功能。

- 数据格式：设备上下行的数据格式，可选择**Alink JSON**或**透传/自定义**。
  - Alink JSON：是物联网平台高级版为开发者提供的设备与云端的数据交换协议，采用 JSON 格式。
  - 透传/自定义：如果您希望使用自定义的串口数据格式，可以选择**透传/自定义**。您需将自定义格式的数据转换为 Alink JSON 脚本，并在云端配置数据解析脚本。

预期结果

产品创建成功。下一步，为产品定义功能，即定义物模型。定义方法和示例，请参见定义物模型。

2.2.2 创建设备

设备需归属于某个产品下。设备可以直接连接物联网平台，也可以作为子设备通过网关连接物联网平台。

背景信息

本文档介绍如何创建高级版设备。

操作步骤

1. 登录物联网平台控制台。
2. 单击左侧导航栏中**设备管理**。
3. 在**设备管理**页面上，选择已创建的高级版产品，再单击**添加设备**。

> **说明：**
> 选择产品时，一定要正确选择该设备所属的高级版产品。

4. 在弹出对话框中，输入设备名称，单击**确定**。

> **说明：**

设备名称（即 DeviceName）需在产品内具有唯一性。可作为设备的唯一标识符，用于与 IoT Hub 进行通信。

5. 在添加成功页面，单击一键复制复制设备的 ProductKey、DeviceName、和 DeviceSecret，然后粘贴到文件中，并妥善保管。

   ProductKey、DeviceName、 和 DeviceSecret 即设备三元组：

   - ProductKey：物联网平台为您创建的产品颁发的全局唯一标识符。
   - DeviceName：自定义的设备唯一标识符。用于设备认证和通信。
   - DeviceSecret：物联网平台为设备颁发的设备秘钥，用于认证加密，需与DeiceName（或者deviceId）成对使用。

2.2.3 定义物模型

物模型指将物理空间中的实体数字化，并在云端构建该实体的数据模型。在物联网平台中，定义物模型即定义产品功能。完成功能定义后，系统将自动生成该产品的物模型。物模型描述产品是什么，能做什么，可以对外提供哪些服务。

产品功能分类

产品功能类型分为三类：属性、服务、和事件。

功能类型	说明
属性（Property）	一般用于描述设备运行时的状态，如环境监测设备所读取的当前环境温度等。属性支持 GET 和 SET 请求方式。应用系统可发起对属性的读取和设置请求。
服务（Service）	设备可被外部调用的能力或方法，可设置输入参数和输出参数。相比于属性，服务可通过一条指令实现更复杂的业务逻辑，如执行某项特定的任务。
事件（Event）	设备运行时的事件。事件一般包含需要被外部感知和处理的通知信息，可包含多个输出参数。如，某项任务完成的信息，或者设备发生故障或告警时的温度等，事件可以被订阅和推送。

定义功能

成功创建高级版产品后，您可以为产品逐条新增自定义功能。

1. 登录物联网平台控制台。
2. 在**产品列表**页面，单击产品所对应的**查看**操作按钮。
3. 单击**功能定义** > **新增**。
4. 选择要添加的功能类型：**属性**、**服务**、或**事件**，并输入信息后，单击**确定**。

在以下章节中，我们将分别具体说明新增属性、新增服务、和新增事件需定义的具体参数。

产品的功能定义完成后，您可以在**功能定义**页面，单击**查看物模型**查看 JSON 格式的功能描述。您还可以单击某个功能对应的**编辑**操作按钮，重新编辑该功能。

新增属性

定义功能时，在**添加功能**页面，选择功能类型为**属性**。

- 功能名称：属性的名称，如用电量。同 一个产品下功能名称不能重复。如果您创建产品时选择了功能模板，输入功能名称时，将自动从标准功能库中筛选匹配的标准属性，供您选择。
- 标识符：属性唯一标识符，在产品中具有唯一性。即 Alink JSON 格式中的**identifier**的值，作为设备上报该属性数据的 Key，云端根据该标识符校验是否接收数据。可包含英文、数字、下划线，如 PowerComsuption。
- 数据类型：属性类型。可选择**int32**（32位整型）、**float**（单精度浮点型）、**double**（双精度浮点型）、**enum**（枚举型）、**bool**（布尔型）、**text**（字符串）、**date**（时间戳）、**struct**（JSON对象）、和**array**（数组）。
  - int32/float/double：需定义取值范围和单位符号。
  - enum：定义枚举项的参数值和参数描述，如 1-加热模式、2-制冷模式。
  - bool：采用 0 或 1 来定义布尔值，如 0-关、1-开。
  - text：需定义字符串的数据长度，最长支持 1024 字节。
  - date：格式为 string 类型的 UTC 时间戳，单位：毫秒。
  - struct：定义一个 JSON 结构体，新增 JSON 参数项，如定义灯的颜色是由 Red、Green、Blue 三个参数组成的结构体。不支持结构体嵌套。
  - array：需声明数组内元素的数据类型，可选择**int32**、**float**、**double**、或**text**。需确保同一个数组元素类型相同，数组长度最长不超过 128 个元素。
- 读写类型：支持选择读写和只读。请求读写的方法支持 GET（获取）和 SET（设置），请求只读的方法仅支持 GET（获取）。
- 描述：对该功能进行说明或备注。

新增服务

定义功能时，在**添加功能**页面，选择功能类型为**服务**。


添加功能
* 功能类型：
属性
服务
事件
* 功能名称
自动喷灌
* 标识符
AutoSprinkle
* 调用方式
异步
同步
输入参数
参数名称：喷灌时间 编辑 删除
参数名称：喷灌量 编辑 删除
+增加参数
输出参数
参数名称： 土壤湿度 编辑 删除
+增加参数
描述
请输入描述
0/100
确认
取消

- 功能名称：服务名称。如果您创建产品时选择了功能模板，输入功能名称时，将自动从标准功能库中筛选匹配的标准服务，供您选择。
- 标识符：服务唯一标识符，在产品下具有唯一性。即 Alink JSON 格式中该服务的**identifier**的值，作为设备上报该属性数据的 Key。可包含英文、数字、和下划线。
- 调用方式：可选择**异步**或**同步**。服务为异步调用时，云端执行调用后直接返回结果，不会等待设备的回复消息。服务为同步调用时，云端会等待设备回复；若设备没有回复，则调用超时。
- 输入参数（可选）：该服务的入参。单击**新增参数**，在弹窗对话框中添加一个服务入参。您可以直接选择某个属性作为入参，也可以自定义参数。如在定义**自动喷灌**服务功能时，将已定义的属性**喷灌时间**和**喷灌量**作为自动喷灌服务的入参，则调用该参数时传入这两个参数，喷灌设备将按照设定的喷灌时间和喷灌量自动进行精准灌溉。

  说明：
  一个服务最多支持定义 10 个入参。
- 输出参数（可选）：该服务的出参。单击**新增参数**，在弹窗对话框中添加一个服务出参。您可以直接选择某个属性作为出参，也可以自定义参数，如将已定义的属性**土壤湿度**作为出参，则云端调用自动喷灌服务时，将返回当前土壤湿度的数据。

  说明：
  一个服务最多支持定义10个出参。
- 描述：对该服务功能进行说明或备注。

新增事件

定义功能时，在**添加功能**页面，选择功能类型为**事件**。

- 功能名称：事件的名称。
- 标识符：事件唯一标识符，在产品下具有唯一性。即 Alink JSON 格式中该事件的**identifier**的值，作为设备上报该属性数据的 Key，如 ErrorCode。
- 事件类型：可选信息、告警、或故障。信息是设备上报的一般性通知，如完成某项任务等。告警和故障是设备运行过程中主动上报的突发或异常情况，优先级高。您可以针对不同的事件类型进行业务逻辑处理和统计分析。

- 输出参数（可选）：该事件的出参。单击**新增参数**，在弹窗对话框中添加一个服务出参。您可以直接选择某个属性作为出参，也可以自定义参数。如，将已定义的属性**电压**作为出参，则设备上报该故障事件时，将携带当前设备的电压值，用于进一步判断故障原因。

  > **说明：**
  > 一个事件最多支持定义10个出参。

- 描述：对该事件功能进行说明或备注。

物模型 JSON 字段说明

完成产品的所有功能定义后，系统将自动生成该产品的物模型。物模型以 JSON 格式表述，称之为 TSL（即 Thing Specification Language）。在产品的**功能定义**页面，单击**查看物模型**即可查看 JSON 格式的 TSL。以下是物模型的 JSON 字段说明。

```
{
"schema": "物的TSL描述schema",
"link": "云端系统级uri,用来调用服务/订阅事件",
"profile": {
"productKey": "产品key",
"deviceName": "设备名称"
},
"properties": [
{
"identifier": "属性唯一标识符(产品下唯一)",
"name": "属性名称",
"accessMode": "属性读写类型，只读(r)，读写(rw)",
"required": "标准品类中的必选是否是标准功能的必选属性",
"dataType": {
"type": "属性类型: int(原生)，float(原生)，double(原生)， text(原生)，date(
String类型UTC毫秒)，bool(0或1的int类型)，enum(int类型)， struct(结构体类型，可
包含前面6种类型)，array(数组类型，支持int/double/float/text)",
"specs": {
"min": "参数最小值(int, float, double类型特有)",
"max": "参数最大值(int, float, double类型特有)",
"unit": "属性单位",
"unitName": "单位名称",
"size":"数组大小，默认最大128(数组特有)",
"item": {
"type":"数组元素的类型"
}
}
}
}
],
"events": [
{
"identifier": "事件唯一标识符(产品下唯一，其中post是默认生成的属性上报事件)",
"name": "事件名称",
```

```
"desc": "事件描述",
"type": "事件类型(info,alert,error)",
"required": "是否是标准功能的必选事件",
"outputData": [
{
"identifier": "参数唯一标识符",
"name": "参数名称",
"dataType": {
"type": "属性类型: int(原生) , float(原生) , double(原生), text(原生) , date(
String类型UTC毫秒) , bool(0或1的int类型) , enum(int类型), struct(结构体类型 , 可
包含前面6种类型) , array(数组类型 , 支持int/double/float/text)",
"specs": {
"min": "参数最小值(int, float, double类型特有)",
"max": "参数最大值(int, float, double类型特有)",
"unit": "属性单位",
"unitName": "单位名称",
"size":"数组大小 , 默认最大128(数组特有)",
"item": {
"type":"数组元素的类型"
}
}
}
}
],
"method": "事件对应的方法名称(根据identifier生成)"
}
],
"services": [
{
"identifier": "服务唯一标识符(产品下唯一 , 产品下唯一 , 其中set/get是根据属性的
accessMode默认生成的服务)",
"name": "服务名称",
"desc": "服务描述",
"required": "是否是标准功能的必选服务",
"inputData": [
{
"identifier": "入参唯一标识符",
"name": "入参名称",
"dataType": {
"type": "属性类型: int(原生) , float(原生) , double(原生), text(原生) , date(
String类型UTC毫秒) , bool(0或1的int类型) , enum(int类型), struct(结构体类型 , 可
包含前面6种类型) , array(数组类型 , 支持int/double/float/text)",
"specs": {
"min": "参数最小值(int, float, double类型特有)",
"max": "参数最大值(int, float, double类型特有)",
"unit": "属性单位",
"unitName": "单位名称",
"size":"数组大小 , 默认最大128(数组特有)",
"item": {
"type":"数组元素的类型"
}
}
}
}
],
"outputData": [
```

```
{
"identifier": "出参唯一标识符",
"name": "出参名称",
"dataType": {
"type": "属性类型: int(原生) , float(原生) , double(原生), text(原生) , date(
String类型UTC毫秒) , bool(0或1的int类型) , enum(int类型), struct(结构体类型，可
包含前面6种类型) , array(数组类型，支持int/double/float/text)",
"specs": {
"min": "参数最小值(int, float, double类型特有)",
"max": "参数最大值(int, float, double类型特有)",
"unit": "属性单位",
"unitName": "单位名称",
"size":"数组大小，默认最大128(数组特有)",
"item": {
"type":"数组元素的类型(数组特有)"
}
}
}
}
],
"method": "服务对应的方法名称(根据identifier生成)"
}
]
}
```

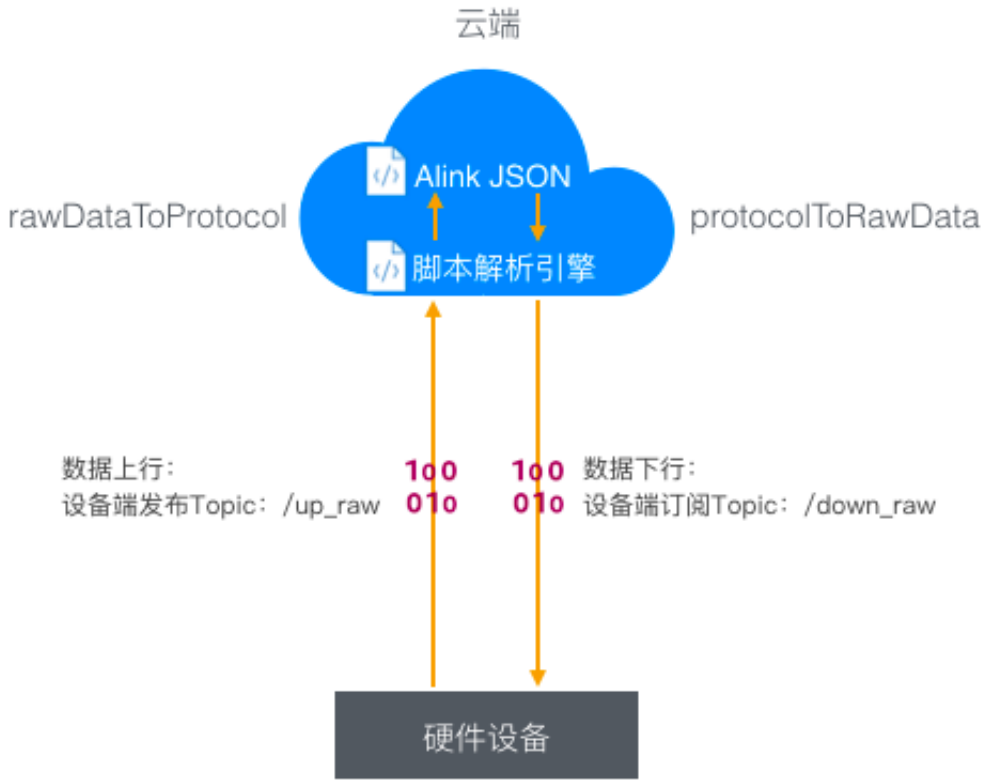
2.2.4 脚本解析

如果您在控制台创建产品时，数据格式选择了**透传/自定义**。您可以在物联网平台控制台上，编写脚本，解析设备数据。

关于脚本解析

由于低配置且资源受限或者对网络流量有要求的设备，不适合直接构造JSON数据和云端通信，因此选择将数据透传到云端，由云端运行转换脚本将透传的数据转换成Alink JSON格式的数据。您可以在创建产品时，选择数据格式为**透传/自定义格式**，目前转换脚本通过JavaScript语言开发，需要开发者自行开发转换脚本。物联网平台为开发者提供了用于数据解析的在线脚本编辑器，方便您进行在线的编辑和模拟调试。

数据解析流程：



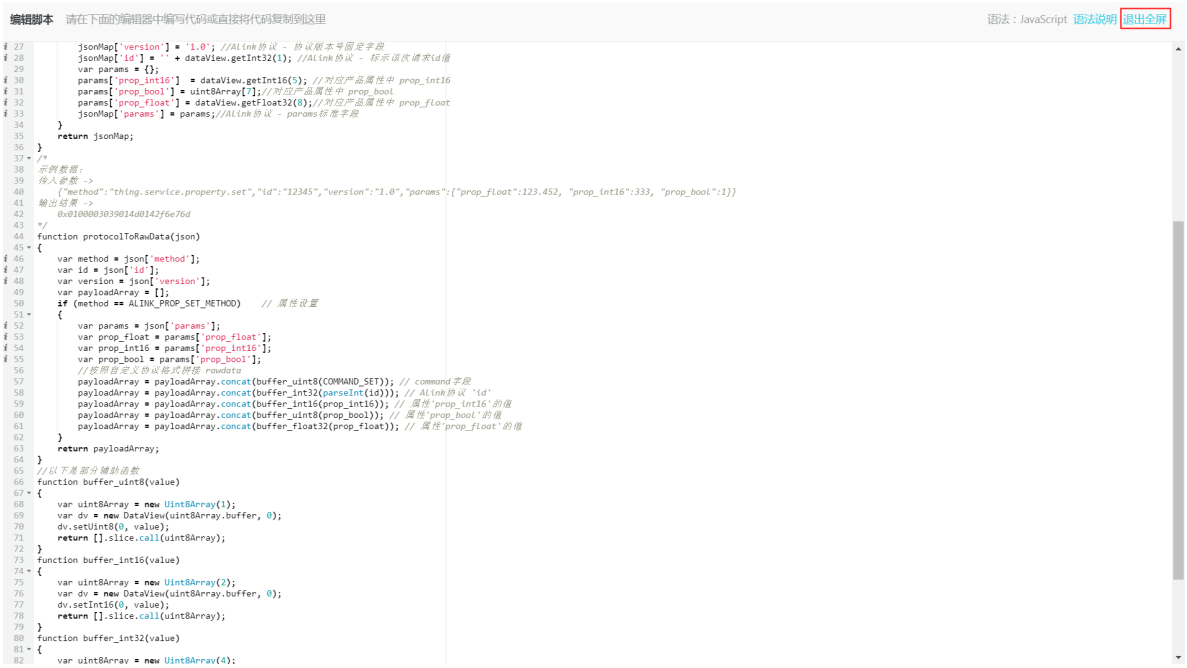
数据解析将为您提供：

- 脚本在线编辑器，支持 JavaScript 语言；
- 支持保存草稿，并从草稿中恢复编辑，支持删除草稿；
- 支持对脚本的模拟数据调试，可输入上下行数据进行模拟转换，查看脚本运行结果；
- 支持对脚本的静态语法检查（JavaScript 语法）；
- 提交脚本到运行环境，在设备上下行时进行调用；

在线编辑脚本

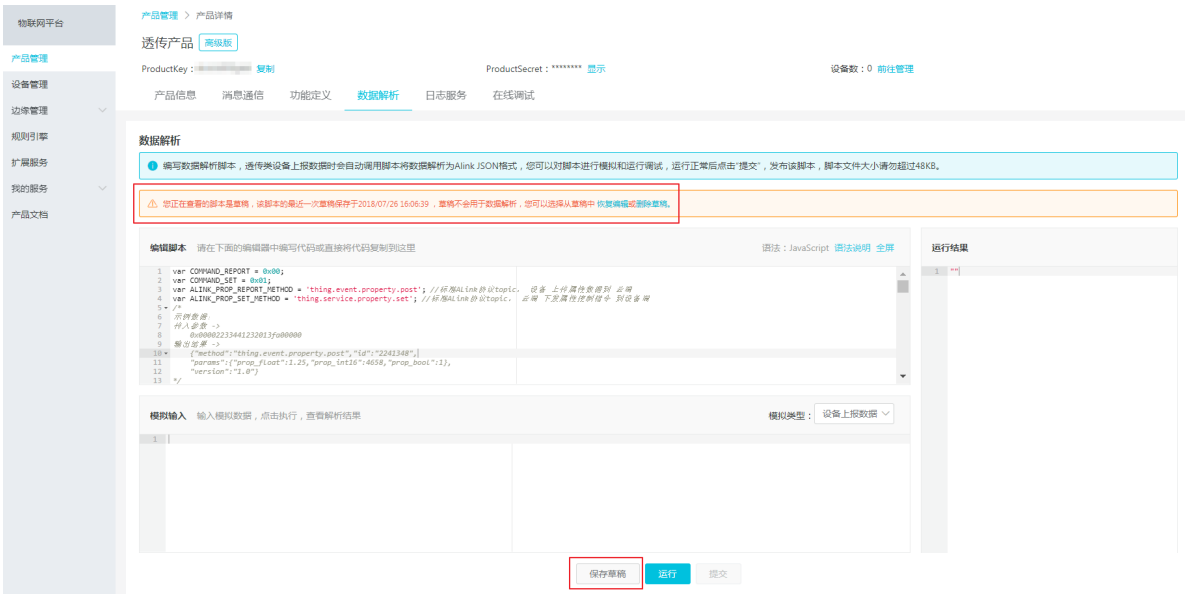
在产品详情页面，单击**数据解析**进入脚本编辑页面，在下方编辑框中编写数据解析脚本，目前支持编写 JavaScript 语言。

- 单击全屏，支持全屏查看或编辑脚本，单击退出全屏即可退出当前编辑界面。




```
        jsonMap['version'] = '1.0'; //Alink协议 - 协议版本号固定字段
        jsonMap['id'] = '' + dataView.getInt32(1); //Alink协议 - 标示该次请求id值
        var params = {};
        params['prop_int16']  = dataView.getInt16(5); //对应产品属性中 prop_int16
        params['prop_bool'] = uint8Array[7];//对应产品属性中 prop_bool
        params['prop_float'] = dataView.getFloat32(8);//对应产品属性中 prop_float
        jsonMap['params'] = params;//Alink协议 - params标准字段
    }
    return jsonMap;
}
/*
示例数据:
传入参数 ->
    {"method":"thing.service.property.set","id":"12345","version":"1.0","params":{"prop_float":123.452, "prop_int16":333, "prop_bool":1}}
输出结果 ->
    0x0100003039014d0142f6e76d
*/
function protocolToRawData(json)
{
    var method = json['method'];
    var id = json['id'];
    var version = json['version'];
    var payloadArray = [];
    if (method == ALINK_PROP_SET_METHOD)    // 属性设置
    {
        var params = json['params'];
        var prop_float = params['prop_float'];
        var prop_int16 = params['prop_int16'];
        var prop_bool = params['prop_bool'];
        //按照自定义协议格式拼接 rawdata
        payloadArray = payloadArray.concat(buffer_uint8(COMMAND_SET)); // command 字段
        payloadArray = payloadArray.concat(buffer_int32(parseInt(id))); // Alink协议 'id'
        payloadArray = payloadArray.concat(buffer_int16(prop_int16)); // 属性'prop_int16'的值
        payloadArray = payloadArray.concat(buffer_uint8(prop_bool)); // 属性'prop_bool'的值
        payloadArray = payloadArray.concat(buffer_float32(prop_float)); // 属性'prop_float'的值
    }
    return payloadArray;
}
//以下是部分辅助函数
function buffer_uint8(value)
{
    var uint8Array = new Uint8Array(1);
    var dv = new DataView(uint8Array.buffer, 0);
    dv.setUint8(0, value);
    return [].slice.call(uint8Array);
}
function buffer_int16(value)
{
    var uint8Array = new Uint8Array(2);
    var dv = new DataView(uint8Array.buffer, 0);
    dv.setInt16(0, value);
    return [].slice.call(uint8Array);
}
function buffer_int32(value)
{
    var uint8Array = new Uint8Array(4);
```

- 单击页面底部的保存草稿，系统将保存本次编辑的结果，您下次再进入平台时，将提示您最近一次保存的草稿，您可以选择恢复编辑或删除草稿。
  - 草稿不会进入到脚本解析的运行环境中，保存草稿不会影响已经提交的正式脚本；
  - 每次保存草稿将覆盖上一次保存的草稿；

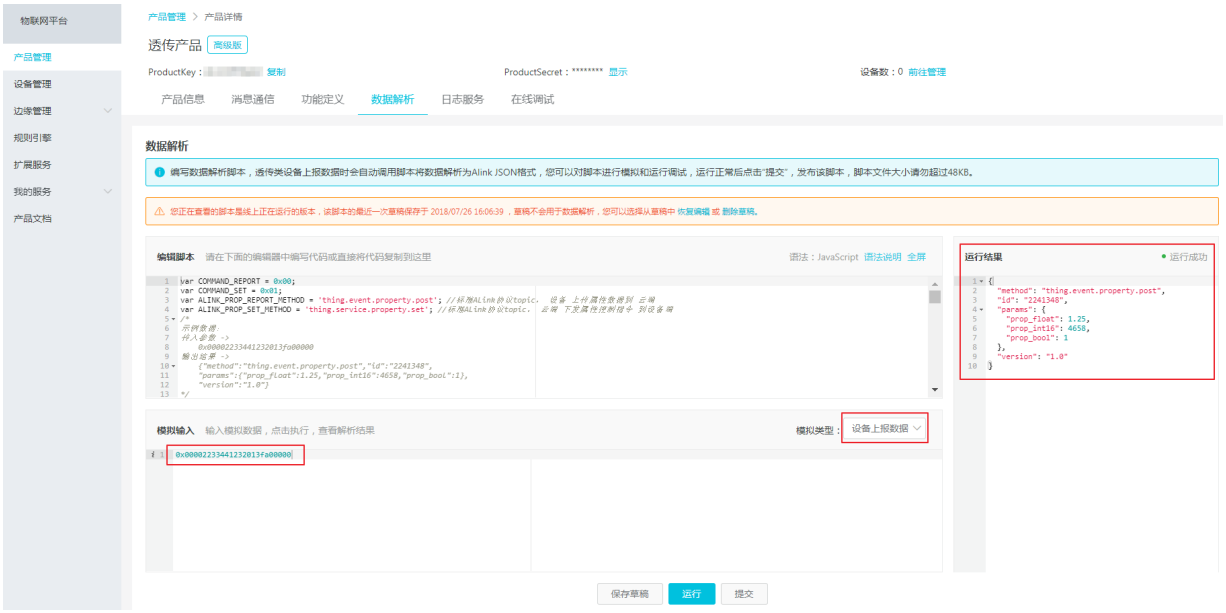


模拟运行

脚本编辑完成后，您可以在编辑器下方输入模拟数据，验证脚本的运行情况，输入模拟数据后，单击运行，系统将调用该脚本模拟进行数据解析，运行结果将显示在右侧的运行结果区域中，若脚本存在错误，将提示运行不通过，请重新修改脚本代码。

模拟上行数据解析

选择模拟类型为**设备上报数据**，输入透传二进制数据，单击**运行**，会将二进制透传数据按照脚本规则转换为JSON格式数据，您可以在运行结果区域看到透传数据的解析结果。

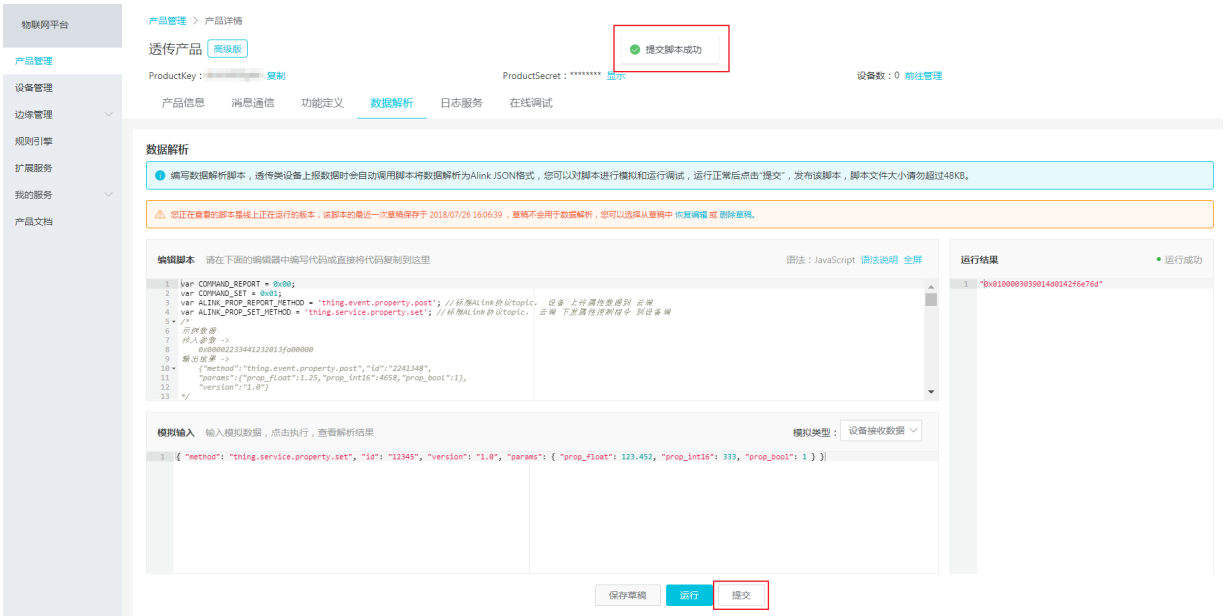


模拟下行数据解析

选择模拟类型为设备接收数据，输入JSON格式数据，单击运行，会将JSON数据按照脚本规则转换为二进制数据，您可以在运行结果区域看到解析后的数据。

提交脚本

为避免错误提交脚本造成数据无法上报，脚本运行通过后，您才可以将脚本提交到运行环境中，设备上下行时将自动调用脚本，实现数据的解析和转换。



说明：

提交脚本之前需至少运行过一次脚本并且运行通过才可提交。

脚本开发框架说明

概述

脚本只需要支持以下两个方法即可接入到物联网平台：

- Alink JSON格式数据转二进制数据：`protocolToRawData`
- 二进制数据转Alink JSON格式数据：`rawDataToProtocol`

脚本语言

目前脚本仅支持符合ECMAScript 5.1的JavaScript语法。

方法定义

- Alink JSON格式数据转二进制数据，方法如下：

```
//解析服务端发送的AlinkJson格式数据，并转换为二进制数据返回
function protocolToRawData(jsonObj){
    return rawdata;
}
```

参数定义：输入参数jsonObj ：符合产品 TSL 定义的 Alink JSON格式数据， 例如：

```
{
    "method": "thing.service.property.set",
    "id": "12345",
    "version": "1.0",
    "params": {
        "prop_float": 123.452,
        "prop_int16": 333,
        "prop_bool": 1
    }
}
```

返回参数 ：二进制byte数组， 如：

```
0x0100003039014d0142f6e76d
```

- 二进制数据转Alink JSON格式数据，方法如下：

```
//解析设备端发送的二进制数据，并转换为Alink JSON格式数据返回
function rawDataToProtocol(rawData){
    return jsonObj;
```

```
}
```

参数定义：输入参数rawData ：二进制byte数组， 例如：

```
0x00002233441232013fa00000
```

返回参数 ：符合产品 TSL 定义的 Alink JSON格式数据，如：

```
{
    "method": "thing.event.property.post",
    "id": "2241348",
    "params": {
        "prop_float": 1.25,
        "prop_int16": 4658,
        "prop_bool": 1
    },
    "version": "1.0"
}
```

脚本Demo示例

1. 创建产品并完成功能定义。

   a. 创建一款高级版产品，数据格式选择为**透传/自定义格式**；

   b. 定义产品功能，创建属性、服务和事件，在本示例中将创建以下三个属性：

标识符	数据类型
prop_float	浮点单精度 float
prop_int16	整数型 int32
prop_bool	布尔型 bool

2. 串口协议格式示例。

帧类型	ID	prop_int16	prop_bool	prop_float
1字节 0-上报；1-下发	请求序号	2字节 prop_int16属性值	1字节 prop_bool属性值	4字节 prop_float属性值

3. 拷贝脚本Demo代码。

   请将参考代码复制到如下输入框中：

```
var COMMAND_REPORT = 0x00;
var COMMAND_SET = 0x01;
var ALINK_PROP_REPORT_METHOD = 'thing.event.property.post'; //标准
ALink JSON格式topic， 设备 上传属性数据到 云端
```

```
var ALINK_PROP_SET_METHOD = 'thing.service.property.set'; //标准ALink
 JSON格式topic， 云端 下发属性控制指令 到设备端
/*
示例数据：
传入参数 ->
    0x000022334412320l3fa00000
输出结果 ->
    {"method":"thing.event.property.post","id":"2241348",
    "params":{"prop_float":1.25,"prop_int16":4658,"prop_bool":1},
    "version":"1.0"}
*/
function rawDataToProtocol(bytes) {
    var uint8Array = new Uint8Array(bytes.length);
    for (var i = 0; i < bytes.length; i++) {
        uint8Array[i] = bytes[i] & 0xff;
    }
    var dataView = new DataView(uint8Array.buffer, 0);
    var jsonMap = new Object();
    var fHead = uint8Array[0]; // command
    if (fHead == COMMAND_REPORT) {
        jsonMap['method'] = ALINK_PROP_REPORT_METHOD; //ALink JSON格
式 - 属性上报topic
        jsonMap['version'] = '1.0'; //ALink JSON格式 - 协议版本号固定字
段
        jsonMap['id'] = '' + dataView.getInt32(1); //ALink JSON格式
 - 标示该次请求id值
        var params = {};
        params['prop_int16'] = dataView.getInt16(5); //对应产品属性中
prop_int16
        params['prop_bool'] = uint8Array[7]; //对应产品属性中
prop_bool
        params['prop_float'] = dataView.getFloat32(8); //对应产品属性
中 prop_float
        jsonMap['params'] = params; //ALink JSON格式 - params标准字段
    }
    return jsonMap;
}
/*
示例数据：
传入参数 ->
    {"method":"thing.service.property.set","id":"12345","version":"
1.0","params":{"prop_float":123.452, "prop_int16":333, "prop_bool":1
}}
输出结果 ->
    0x0100003039014d0142f6e76d
*/
function protocolToRawData(json) {
    var method = json['method'];
    var id = json['id'];
    var version = json['version'];
    var payloadArray = [];
    if (method == ALINK_PROP_SET_METHOD) // 属性设置
    {
        var params = json['params'];
        var prop_float = params['prop_float'];
        var prop_int16 = params['prop_int16'];
        var prop_bool = params['prop_bool'];
        //按照自定义协议格式拼接 rawdata
```

```
        payloadArray = payloadArray.concat(buffer_uint8(COMMAND_SET
)); // command字段
        payloadArray = payloadArray.concat(buffer_int32(parseInt(id
))); // ALink JSON格式 'id'
        payloadArray = payloadArray.concat(buffer_int16(prop_int16
)); // 属性'prop_int16'的值
        payloadArray = payloadArray.concat(buffer_uint8(prop_bool
)); // 属性'prop_bool'的值
        payloadArray = payloadArray.concat(buffer_float32(prop_float
)); // 属性'prop_float'的值
    }
    return payloadArray;
}
//以下是部分辅助函数
function buffer_uint8(value) {
    var uint8Array = new Uint8Array(1);
    var dv = new DataView(uint8Array.buffer, 0);
    dv.setUint8(0, value);
    return [].slice.call(uint8Array);
}
function buffer_int16(value) {
    var uint8Array = new Uint8Array(2);
    var dv = new DataView(uint8Array.buffer, 0);
    dv.setInt16(0, value);
    return [].slice.call(uint8Array);
}
function buffer_int32(value) {
    var uint8Array = new Uint8Array(4);
    var dv = new DataView(uint8Array.buffer, 0);
    dv.setInt32(0, value);
    return [].slice.call(uint8Array);
}
function buffer_float32(value) {
    var uint8Array = new Uint8Array(4);
    var dv = new DataView(uint8Array.buffer, 0);
    dv.setFloat32(0, value);
    return [].slice.call(uint8Array);
}
```

4. 模拟数据解析。

   - 模拟设备上报数据

     模拟类型选择**设备上报数据**，填写测试数据：

     ```
     0x00022334412320 13fa00000
     ```

     点击运行，查看上报数据输出结果：

     ```
     {
         "method": "thing.event.property.post",
         "id": "2241348",
         "params": {
             "prop_float": 1.25,
             "prop_int16": 4658,
             "prop_bool": 1
         },
         "version": "1.0"
     ```

```
}
```

- 模拟设备接收数据模拟类型选择**设备接收数据**，填写测试数据：

```
{
    "method": "thing.service.property.set",
    "id": "12345",
    "version": "1.0",
    "params": {
        "prop_float": 123.452,
        "prop_int16": 333,
        "prop_bool": 1
    }
}
```

点击运行，查看接收数据输出结果：

```
0x0100003039014d0142f6e76d
```

附录：本地环境调试脚本的方法

目前物联网平台数据解析不支持debug调试，建议开发过程先在本地开发完成后再将脚本拷贝到物联网控制台的脚本编辑器中。可参考如下方式调用。

```
// Test Demo
function Test()
{
    //0x001232013fa00000
    var rawdata_report_prop = new Buffer([
        0x00, //固定command头, 0代表是上报属性
        0x00, 0x22, 0x33, 0x44, //对应id字段， 标记请求的序号
        0x12, 0x32, //两字节 int16, 对应属性 prop_int16
        0x01, //一字节 bool, 对应属性 prop_bool
        0x3f, 0xa0, 0x00, 0x00 //四字节 float, 对应属性 prop_float
    ]);
    rawDataToProtocol(rawdata_report_prop);
    var setString = new String('{"method":"thing.service.property.
set","id":"12345","version":"1.0","params":{"prop_float":123.452, "
prop_int16":333, "prop_bool":1}}');
    protocolToRawData(JSON.parse(setString));
}
```

```
Test();
```

2.3 Topic

物联网平台中，云端和设备端通过 Topic 来实现消息通信。设备上报消息至指定的Topic中，并从Topic中订阅消息。云端将指令下发到Topic中，并订阅具体Topic来获取设备信息。

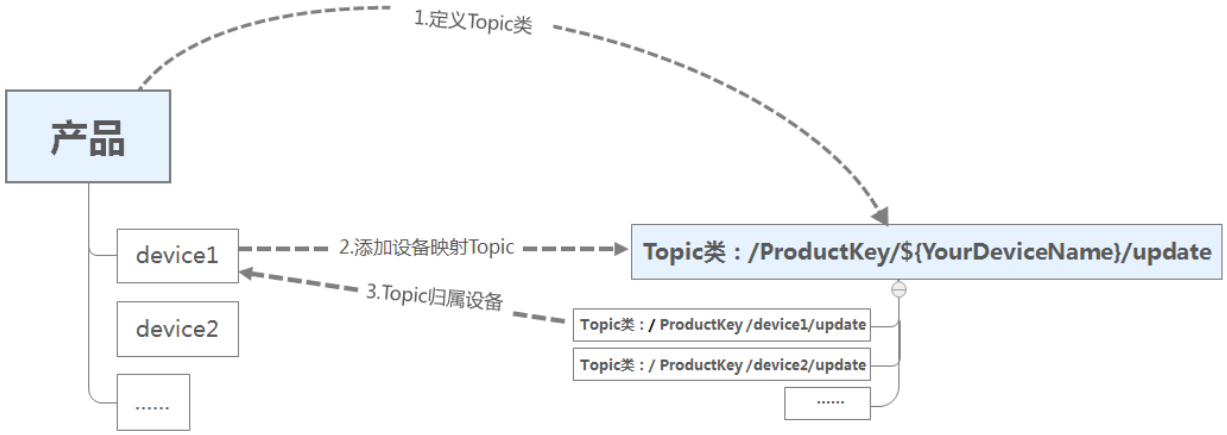
2.3.1 什么是Topic

物联网平台中，服务端和设备端通过 Topic 来实现消息通信。Topic是针对设备的概念，Topic类是针对产品的概念。

什么是**Topic**类？

为了方便海量设备基于海量 Topic 进行通信，简化授权操作，物联网平台增加了 Topic 类的概念。您创建产品后，物联网平台会为该产品自动创建默认的 Topic 类。并且，在您创建设备后，会自动将产品 Topic 类映射到设备上。您无需单独为每个设备授权 Topic。

图 **2-1: Topic** 自动生成示意图



在您创建产品后，物联网平台会自动为您的产品生成一些标准的 Topic 类。您可以在产品的**消息通信**页面，查看该产品的所有 Topic 类。

关于 Topic 类的说明：

- Topic类是一类 Topic 的集合。例如，Topic 类：`/${productKey}/${deviceName}/update`是具体 Topic：`/${productKey}/device1/update`和`/${productKey}/device2/update`的集合。

- Topic类中必须以正斜线(/)进行分层，区分每个类目。其中，有两个类目为既定类目：`${productKey}`表示产品的标识符 ProductKey；`${deviceName}`表示设备名称。
- 类目命名只能包含字母，数字和下划线(_)。每级类目不能为空。
- 设备操作权限：**发布**表示设备可以往 Topic 发布消息；**订阅**表示设备可以从 Topic 订阅消息。
- 基础版产品支持自定义 Topic 类。您可以根据业务需求，通过自定义 Topic 类灵活地进行消息通信。高级版不支持自定义 Topic 类和修改类目名称。
- 系统 Topic 类是由系统预定义的 Topic 类，不支持用户自定义，不采用`/${productKey}`开头。例如，高级版中，针对物模型所提供的 Topic 类一般以`/sys/`开头；固件升级相关的Topic类以`/ota/`开头；设备影子的 Topic 类以`/shadow/`开头。

什么是**Topic**？

产品的 Topic 类不用于通信，只是定义 Topic。用于消息通信的是具体的 Topic。

- Topic 格式和Topic 类格式一致。区别在于 Topic 类中的变量`${deviceName}`，在 Topic 中则是具体的设备名称。
- 设备对应的 Topic 是从产品 Topic 类映射出来，根据设备名称而动态创建的。设备的具体 Topic 中带有设备名称（即`deviceName`），只能被该设备用于 Pub/Sub 通信。例如，Topic：`/${productKey}/device1/update`归属于设备名为**device1**的设备，所以只能被设备 **device1** 用于发布、订阅消息，而不能被设备 **device2** 用于发布订阅消息。
- 在配置规则引擎时，配置的 Topic 中可使用通配符，且同一个类目中只能出现一个通配符。

表 **2-1: Topic** 通配符

通配符	描述
#	这个通配符必须出现在 Topic 的最后一个类目，代表本级及下级所有类目。例如， Topic：`/productKey/device1/#`，可以代表`/productKey/device1/update`和`productKey/device1/update/error`。
+	代表本级所有类目。例如，Topic：`/productKey/+/update`，可以代表`/${productKey}/device1/update`和`/${productKey}/device2/update`。

2.3.2 Topic列表

本文列出了系统定义的Topic。

基础版默认Topic类

基础版支持自定义Topic类。创建产品之后，系统自动创建三个默认的用户Topic类，您可以直接使用或添加更多自定义Topic类。

- `/${productKey}/${deviceName}/update`：用于设备上报数据。设备操作权限：发布。
- `/${productKey}/${deviceName}/update/error`：用于设备上报错误。设备操作权限：发布。
- `/${productKey}/${deviceName}/get`：用于云端获取设备数据。设备操作权限：订阅。

高级版默认Topic类

高级版提供默认的系统 Topic，不支持自定义Topic类。您可以直接使用 SDK 实现消息通信，或者 Pub 或 Sub 相关的系统 Topic 实现设备数据的上下行通信。

高级版默认Topic类分为 Alink Topic 类和 透传 Topic 类。Alink Topic 类是设备采用 Alink JSON 进行消息通信时使用的 Topic 类。透传 Topic 类是设备采用透传/自定义格式进行消息通信时使用的Topic类。

Alink Topic 类：

- `/sys/${productKey}/${deviceName}/thing/event/property/post`：用于设备上报属性。设备操作权限：发布。
- `/sys/${productKey}/${deviceName}/thing/service/property/set`：用于设置设备属性。设备操作权限：订阅。
- `/sys/${productKey}/${deviceName}/thing/event/{tsl.event.identifer}/post`：用于设备上报事件。设备操作权限：发布。
- `/sys/${productKey}/${deviceName}/thing/service/{tsl.service.identifer}`：用于调用设备服务。设备操作权限：订阅。
- `/sys/${productKey}/${deviceName}/thing/deviceinfo/update`：用于设备上报标签。设备操作权限：发布。

透传 Topic 类：

- `/sys/{ProductKey}/{deviceName}/thing/model/up_raw`：设备透传数据上报，用于设备上报属性、事件和扩展信息的数据。设备操作权限：发布。

- `/sys/{ProductKey}/{deviceName}/thing/model/down_raw`：设备透传数据下行，用于获取设备属性、设置设备属性和调用设备服务。设备操作权限：订阅。

设备影子相关Topic类

设备影子提供系统 Topic 类，主要用于基础版产品的影子更新。

- `/shadow/update/${productKey}/${deviceName}`：用于更新设备影子。设备和应用程序操作权限：发布。
- `/shadow/get/${productKey}/${deviceName}`：用于获取设备影子。设备操作权限：订阅。

远程配置相关Topic类

远程配置提供系统Topic类，用于给设备下发配置文件。

- /sys/${productKey}/${deviceName}/thing/config/push：用于设备接收云端推送的配置信息。设备操作权限：订阅。
- `/sys/${productKey}/${deviceName}/thing/config/get`：用于设备主动请求更新配置。设备操作权限：发布。
- `/sys/${productKey}/${deviceName}/thing/config/get_reply`：用于设备主动请求更新配置，接收云端返回的配置信息。设备操作权限：订阅。

固件升级相关Topic类

固件升级提供系统Topic类，用于设备上报固件版本和接收升级通知。

- `/ota/device/inform/${productKey}/${deviceName}`：用于设备端上报固件版本给云端。设备操作权限：发布。
- `/ota/device/upgrade/${productKey}/${deviceName}`：用于设备端接收云端固件升级通知。设备操作权限：订阅。
- `/ota/device/progress/${productKey}/${deviceName}`：用于设备端上报固件升级进度。设备操作权限：发布。
- `/ota/device/request/${productKey}/${deviceName}`：设备端请求是否固件升级。设备操作权限：发布。

设备广播Topic类

设备广播提供系统Topic类，但是可以自定义广播设备范围。

`/broadcast/${productKey}/+`：用于设备接收广播消息。设备操作权限：订阅。通配符（+）可自定义为广播的设备范围。

RRPC通信相关Topic类

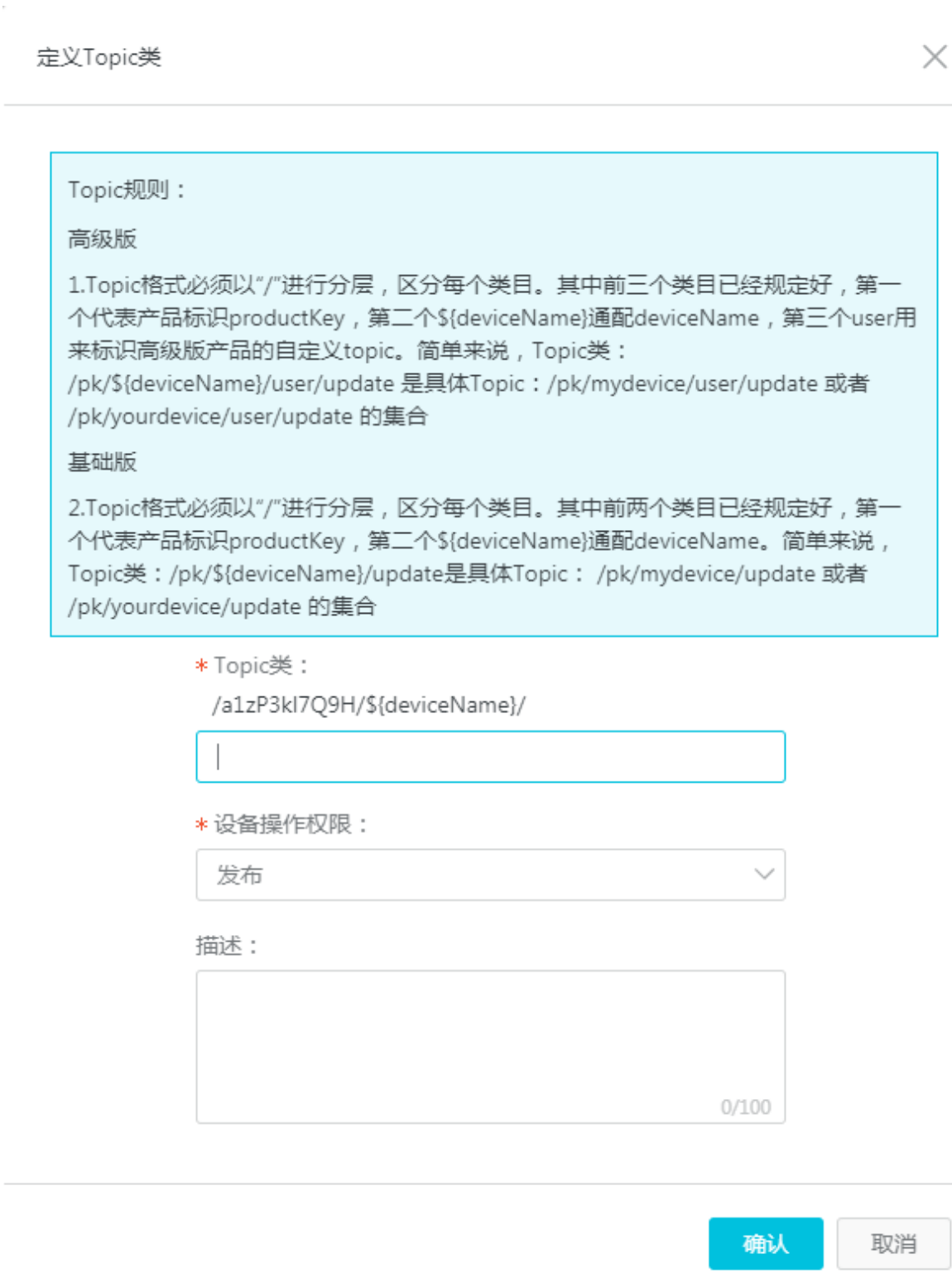
IoT Hub基于开源协议MQTT封装了同步的通信模式，服务端下发指令给设备可以同步得到设备端的响应。以下是

- `/sys/${productKey}/${deviceName}/rrpc/request/${messageId}`：用于设备发布RRPC请求。设备操作权限：发布。
- `/sys/${productKey}/${deviceName}/rrpc/response/${messageId}`：用于设备响应RRPC请求。设备操作权限：发布。
- `/sys/${productKey}/${deviceName}/rrpc/request/+`：用于设备订阅RRPC请求。设备操作权限：订阅。

2.3.3 自定义Topic

本文介绍基础版产品和高级版产品如何自定义Topic。

1. 在产品列表页面，选择需要自定义Topic的产品，单击右侧的查看，进入产品详情页面。
2. 选择消息通信，单击定义**Topic**类，系统显示定义**Topic**类窗口。
3. 自定义Topic类。



- **Topic类**：根据界面上**Topic规则**设置Topic类。
- **设备操作权限**：设备对该Topic的操作权限，可设置为发布、订阅、发布和订阅。
- **描述**：对自定义的Topic进行描述，可以为空。

4. 物联网平台也支持自定义带通配符的Topic类。如果您需要订阅通配符，则首先需要自定义带通配符的Topic，再订阅。

   创建Topic类时，参数设置如下：

   - **Topic类**：您可以创建带有通配符#或者+的Topic类。

     > **说明：**
     > 通配符#只能出现在Topic类的最后面。

   - **设备操作权限**：设备对该Topic的操作权限，只有选择订阅，才能支持填写通配符。
   - **描述**：对自定义的Topic进行描述，可以为空。

   已创建的带通配符的Topic，在设备的**Topic列表**页面不支持执行**发布消息**的操作。

   > **说明：**
   > 当前基础版产品的**设备详情**页有**Topic列表**，高级版产品的**设备详情**页，暂无**Topic列表**。

5. 单击**确认**，完成自定义Topic类。

2.4 标签

物联网平台的标签是您给产品或设备自定义的标识。您可以使用标签功能来灵活管理产品与设备。

物联网往往涉及量级产品与设备的管理。如何区分不同批次的产品与设备，如何实现批量管理，成为一大挑战。阿里云物联网平台为解决这一问题提供了标签功能。您可以为不同产品或设备贴上不同标签，然后根据标签实现分类统一管理。

标签包括产品标签与设备标签。产品标签是设备标签的母版。创建产品标签后，该产品下新增的设备将自动继承产品标签。您也可以单独为单个设备添加设备标签。标签的结构为`Key:Value`。

本文将详细讲解产品标签与设备标签的创建。

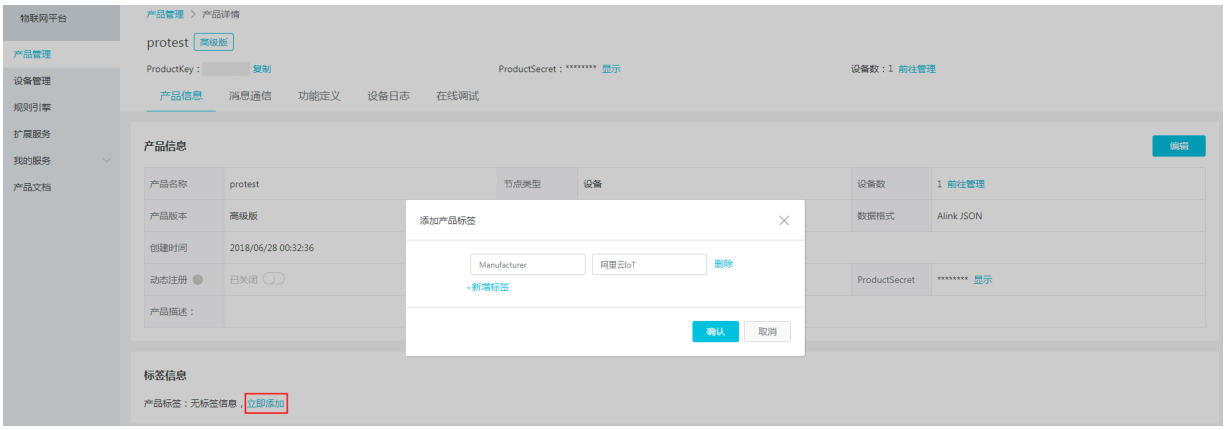
产品标签

产品标签通常描述的是对一个产品下所有设备所具有的共性信息。如产品的制造商、所属单位、外观尺寸、操作系统等。需在创建产品后，再为该产品添加产品标签。

添加产品标签操作步骤：

1. 登录物联网平台控制台。
2. 在**产品列表**页面，单击要添加标签的产品所对应的**查看**操作按钮，进入**产品信息**页面。

3. 单击产品标签对应的编辑按钮。
4. 在编辑标签对话框中，输入标签的 **Key** 和 **Value**，然后单击确定。



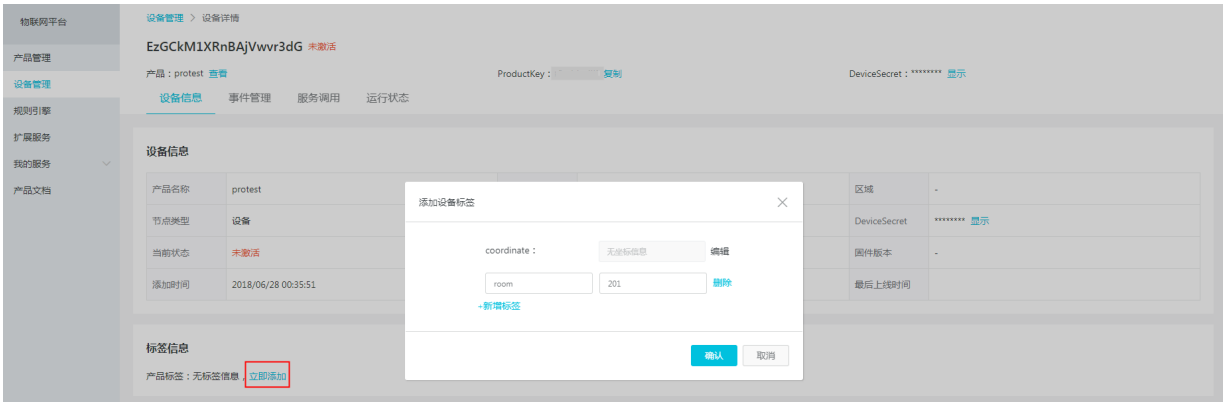
产品标签创建后，该产品下的新增设备将自动继承产品的标签。

设备标签

在产品下添加设备后，设备将自动继承所属产品的标签。但是，从产品中继承的标签一般只定义了产品下所有设备的共性。您可以根据设备的特性为设备添加特有的标签，方便对设备进行管理。例如，为房间 201 的智能电表定义一个标签为`room:201`。您可以在控制台管理设备标签，也可以通过 API 管理设备标签。

在控制台添加设备标签的操作步骤：

1. 登录物联网平台控制台。
2. 单击设备管理。
3. 在设备管理页面，单击要添加标签的设备所对应的查看操作按钮，进入设备详情页面。
4. 单击设备标签对应的编辑按钮。
5. 在编辑标签对话框中，输入标签的 **Key** 和 **Value**，然后单击确定。



设备标签信息会跟随设备在系统内部流转。并且，物联网平台可以基于规则引擎，将设备标签发给阿里云其他云服务。

2.5 网关与子设备

物联网平台支持设备直连，也支持设备挂载在网关上，作为网关的子设备，由网关直连。

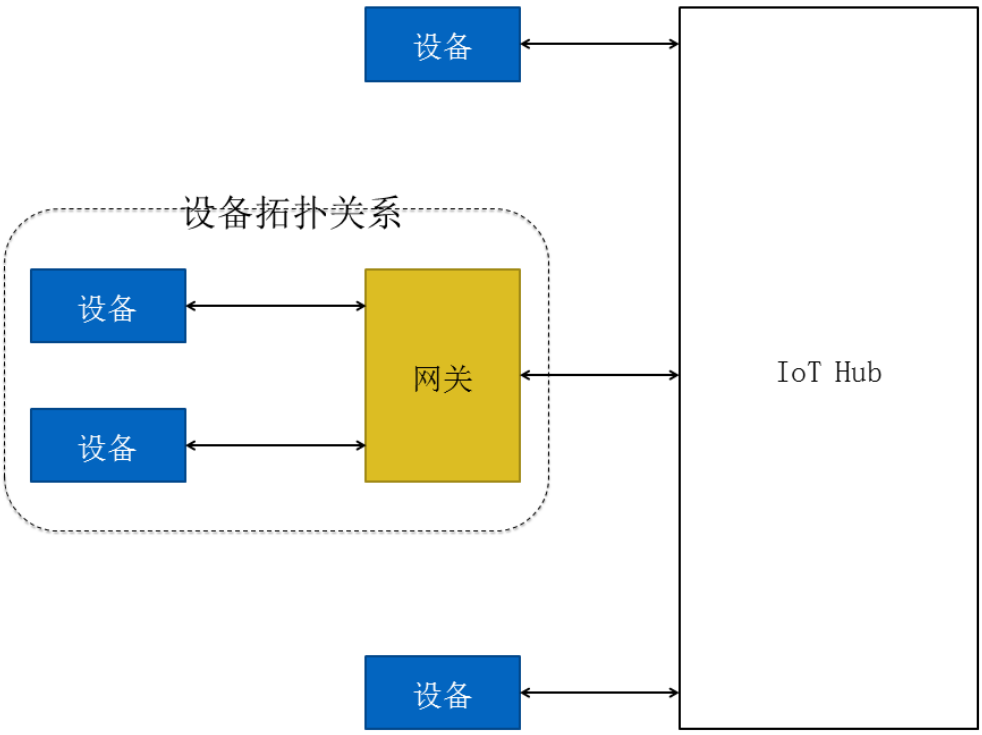
网关与设备

创建产品与设备时，需要选择节点类型。平台目前支持两种节点类型：设备和网关。

- 设备：指不能挂载子设备的设备。设备可以直连IoT Hub，也可以作为网关的子设备，由网关代理连接IoT Hub。
- 网关：指可以挂载子设备的直连设备。网关可以管理子设备、可以维持与子设备的拓扑关系，并将该拓扑关系同步到云端。

网关与子设备的拓扑关系

图 **2-2:** 设备拓扑关系



创建完产品与设备后，如果拓扑中包含网关。在设备开发阶段，需由网关将拓扑关系同步至云端，代替子设备完成设备认证、消息上传、指令接收等与平台的通信。而子设备将由网关统一管理。

网关代替子设备与平台通信的具体设置方法，请参考子设备接入。

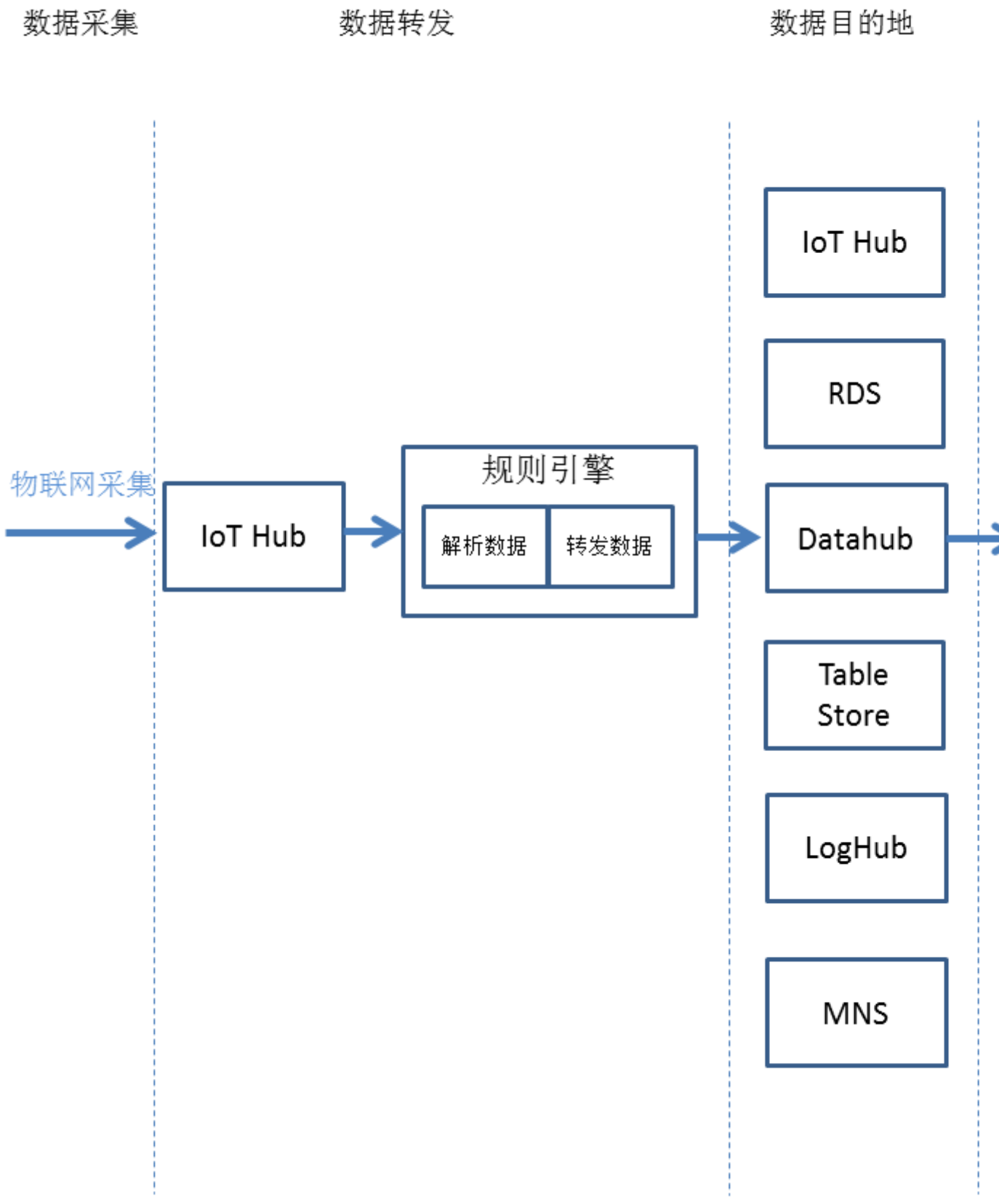
3 规则引擎

3.1 概览

当设备基于*Topic*进行通信时，您可以使用规则引擎，编写SQL对Topic中的数据进行处理，并配置转发规则将处理后的数据转发到阿里云其他服务。例如：

- 可以转发到*RDS*、*Table Store*、*HiTSDB*中进行存储。
- 可以转发到*DataHub*中，使用*Streacompute*进行流计算，使用*Maxcompute*进行大规模离线计算。
- 可以转发到函数计算进行事件计算。
- 可以转发到另一个Topic中实现M2M通信。
- 可以转发到队列*MQ*实现高可靠消费数据·。

使用规则引擎后，您无需购买服务器部署分布式架构，即可实现采集 + 计算 + 存储的全栈服务。


数据采集
数据转发
数据目的地
物联网采集
IoT Hub
规则引擎
解析数据
转发数据
IoT Hub
RDS
Datahub
Table Store
LogHub
MNS

说明：

使用规则引擎时，您需特别注意：

- 规则引擎基于Topic对数据进行处理。只有设备通过Topic进行通信时，才能使用规则引擎。
- 规则引擎通过SQL对Topic中的数据进行处理。
- SQL语法目前不支持子查询、不支持like操作符。
- 支持部分函数，比如`deviceName()`获取当前设备名称，具体请参考函数列表。

3.2 地域和可用区

物联网平台使用规则引擎将设备数据转发至其他阿里云产品。在转发时，需确认目的云产品已经在该地域和可用区上线，并且支持相应格式数据的转发。

表 3-1: 地域和可用区列表

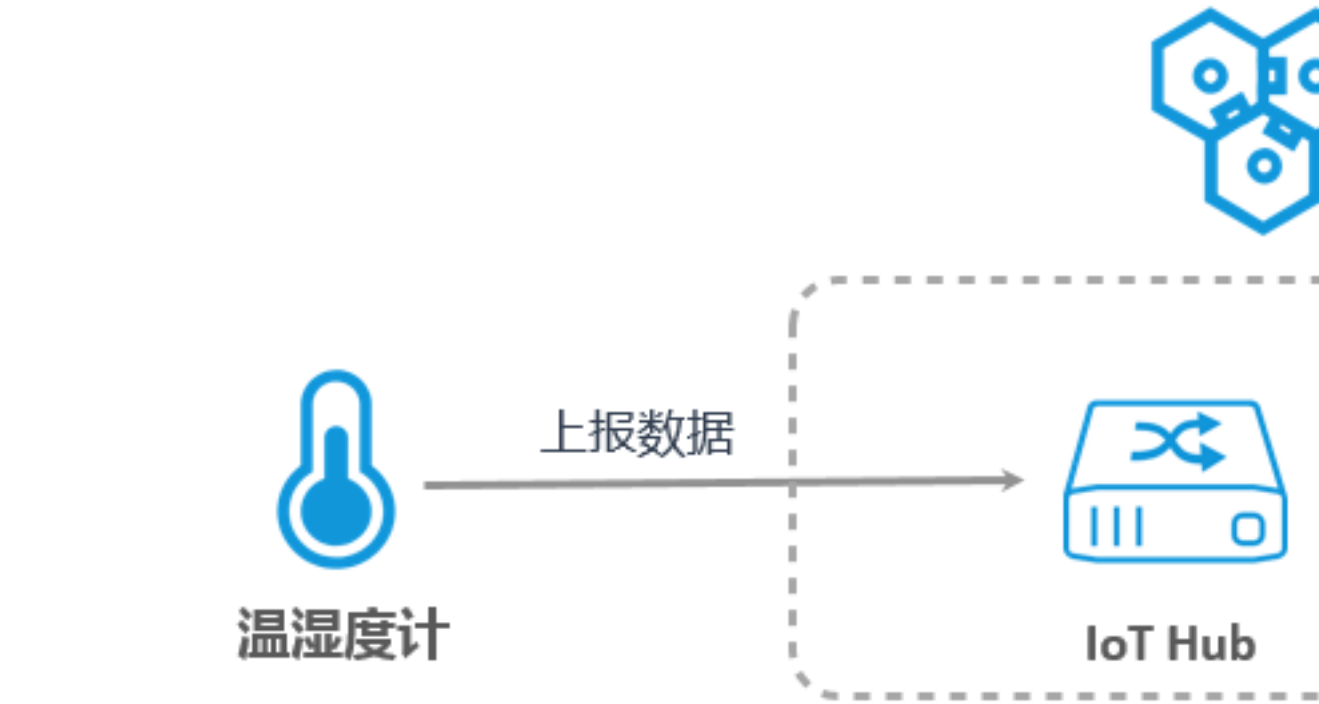
	华东2（上海）		新加坡		日本（东京）		美国（硅谷）		德国（法兰克福）	
	JSON	二进制	JSON	二进制	JSON	二进制	JSON	二进制	JSON	二进制
Table Store（表格存储）	√	-	√	-	√	-	√	-	√	-
DataHub	√	√	√	√	-	-	-	-	-	-
RDS	√	-	√	-	√	-	√	-	√	-
MNS（消息服务）	√	√	√	√	√	√	√	√	√	√
ONS（消息队列，Message Queue）	√	√	-	-	-	-	-	-	-	-
HITSDB（时间序列数据库，TSDB）	√	-	-	-	-	-	-	-	-	-
FC（函数计算，Function Compute）	√	√	√	√	√	√	-	-	-	-

3.3 数据流转

规则引擎仅能处理发送至Topic的数据。本文将为您讲解使用规则引擎时，数据的流转和不同阶段的数据格式。

基础版产品

基础版产品设备数据直接透传至IoT Hub，数据结构不变。数据流转图如下：



设备：D39383

```
topic："/{PK}/{DN}/data"
payload:
{
"temperature":23,
"humidity":63
}
```

设备原始payload数据

```
topic："/{PK}/{DN}/da
payload:
{
"temperature":23,
"humidity":63
}
```

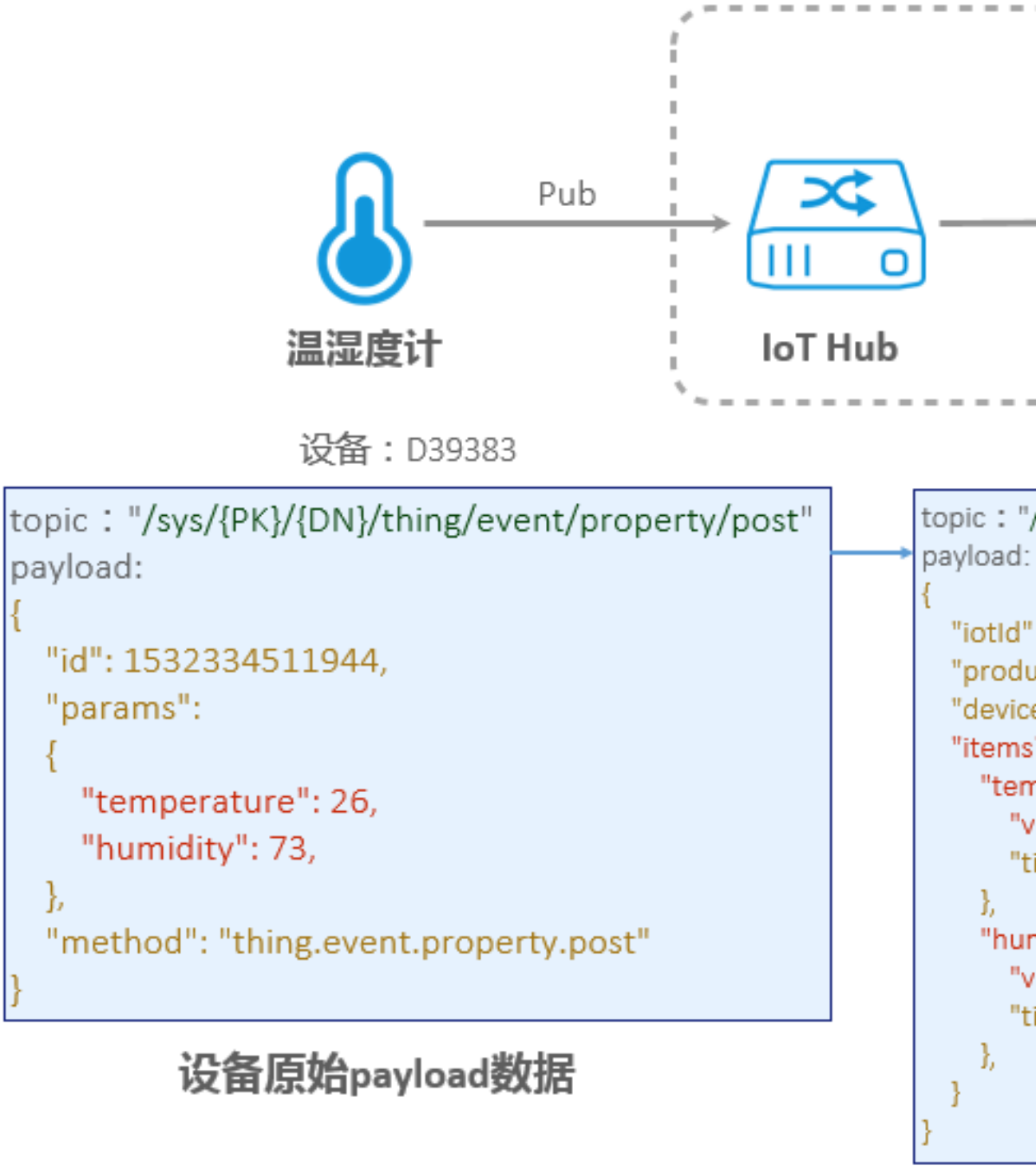
透传结构不

```
SELECT
deviceName() as deviceNa
timestamp('HH:mm') as tir
temperature,
humidity
FROM "/{PK}/+/data"
```

高级版产品

创建高级版产品时，可选择数据格式为Alink JSON格式或者二进制格式。

- 二进制格式数据规则引擎不做处理，直接透传，数据流转请参考基础版产品的数据流转图。
- Alink JSON格式数据会经物模型解析，规则引擎SQL处理解析后的数据。



设备原始payload数据

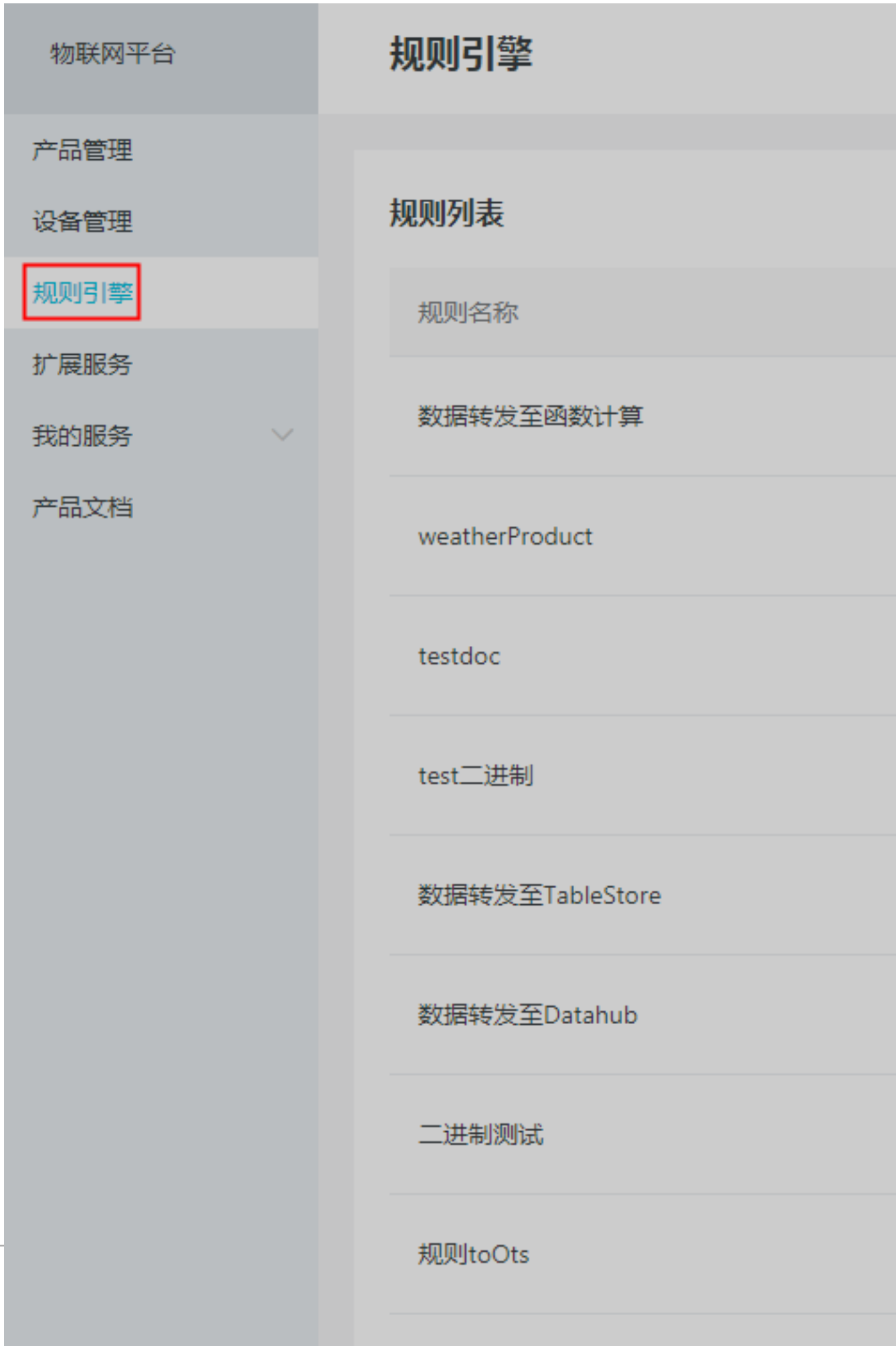
3.4 设置规则引擎

本文将为您详细讲解如何设置一条完整的规则。

背景信息

操作步骤

1. 单击**规则引擎**，单击**创建规则**，创建一条新的规则。
2. 填写**规则名称**，选择**数据格式**。


物联网平台
产品管理
设备管理
规则引擎
扩展服务
我的服务
产品文档
规则引擎
规则列表
规则名称
数据转发至函数计算
weatherProduct
testdoc
test二进制
数据转发至TableStore
数据转发至Datahub
二进制测试
规则toOts

- 规则名称：输入规则名称，用以区别各条规则。
- 数据格式：支持JSON和二进制。因规则引擎基于Topic处理数据，此处的数据格式需与被处理Topic中的数据格式保持一致。

3. 找到刚创建的规则，单击**管理**，进入**规则详情**页，设置规则具体信息。


物联网平台
产品管理
设备管理
边缘管理
规则引擎
扩展服务
我的服务
产品文档
规则引擎 > 规则详情
数据转发至函数计算
数据格式：JSON
产品描述：
处理数据
转发数据
数据目的地

a) 单击**编写SQL**，编写SQL，设置数据处理的具体规则。

> **说明：**
> 二进制数据不支持提取字段，仅支持条件筛选。

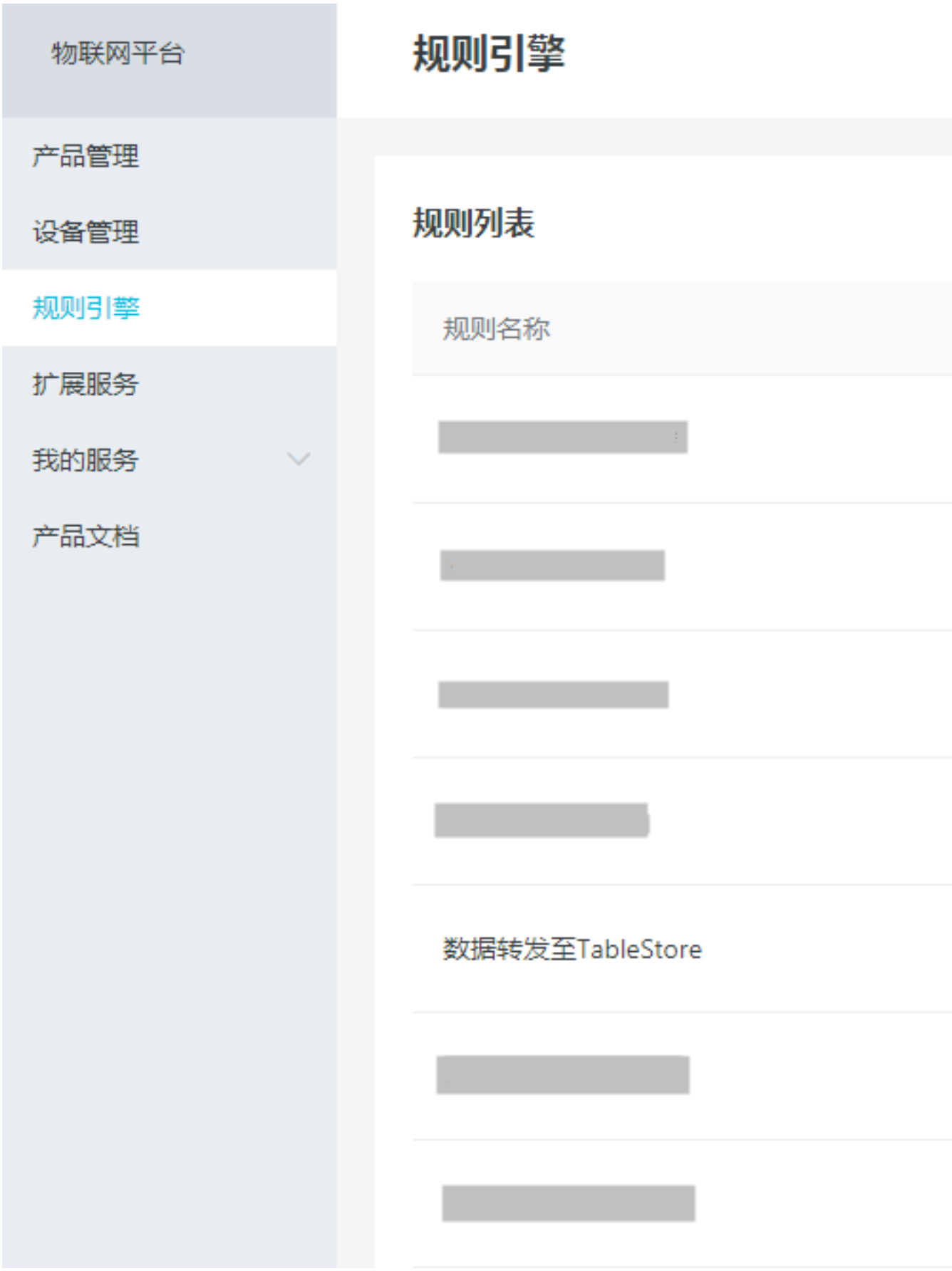
例如：以下SQL可以将deviceName从Basic_Light_001产品下，后缀为data的Topic类中取出。

具体编写方法请参考*SQL*表达式和函数列表。

b) 单击**转发数据**一栏的**添加操作**，将数据转发至目的云产品。

数据转发的具体实例，请参考使用实例。

4. 返回至规则引擎页。单击启动，数据即可按照规则进行转发。


物联网平台
产品管理
设备管理
规则引擎
扩展服务
我的服务
产品文档
规则引擎
规则列表
规则名称
数据转发至TableStore

您也可以：

- 单击管理修改规则具体设置。
- 单击删除删除对应规则。其中，启动中的规则不可删除。
- 单击停止停止某条规则转发数据。

3.5 SQL表达式

使用规则引擎时，若您的数据为JSON格式，可以编写SQL来解析和处理数据。本文主要讲解SQL表达式。

SQL表达式

JSON数据可以映射为虚拟的表，其中Key对应表的列，Value对应列值，这样就可以使用SQL处理。为便于理解，我们将规则引擎的一条规则抽象为一条SQL表达（类试MySQL语法）：

例如某环境传感器用于火灾预警，可以采集温度、湿度及气压数据，上报数据内容如下：

```
{
"temperature":25.1
"humidity":65
"pressure":101.5
"location":"xxx,xxx"
}
```

假定温度大于38，湿度小于40时，需要触发报警，可以编写如下的SQL语句：

```
SELECT temperature as t, deviceName() as deviceName, location FROM /ProductA/+/update WHERE temperature > 38 and humidity < 40
```

当上报的数据中，温度大于38且湿度小于40时，会触发该规则，并且解析数据中的温度、设备名称、位置，用于进一步处理。

FROM

FROM 需要填写Topic通配符，用于匹配需要处理的消息Topic。当有符合Topic规则的消息到达时，消息的payload数据以json格式解析，并根据SQL语句进行处理（如果消息格式不合法，将忽略此消息）。您可以使用`topic()`函数引用具体的Topic值。

上文例子中，"FROM /ProductA/+/update"语句表示该SQL仅处理符合/ProductA/+/update格式的消息，具体匹配参考 *Topic*。

SELECT

- JSON数据格式

  select语句中的字段，可以使用上报消息的payload解析结果，即json中的键值，也可以使用SQL内置的函数，比如`deviceName()`。不支持子SQL查询。

  上报的json数据格式，可以是数组或者嵌套的json，SQL语句支持使用json path获取其中的属性值，如对于`{a:{key1:v1, key2:v2}}`，可以通过`a.key2` 获取到值`v2`。使用变量时需要注意单双引号区别：单引号表示常量，双引号或不加引号表示变量。如使用单引号`'a.key2'`，值为`a.key2`。

  内置的SQL函数可以参考函数列表。

  例如上文，"SELECT temperature as t, deviceName() as deviceName, location"语句，其中temperature和loaction来自于上报数据中的字段，deviceName()则使用了内置的SQL函数。

- 二进制数据格式

  目前二进制数据不支持解析payload中的字段，SELECT语句固定为`SELECT *`，表示透传二进制数据。

WHERE

- JSON数据格式

规则触发条件，条件表达式。不支持子SQL查询。WHERE中可以使用的字段和SELECT语句一致，当接收到对应topic的消息时，WHERE语句的结果会作为规则是否触发的判断条件。具体条件表达式列表见下方表格。

上文例子中， `"WHERE temperature > 38 and humidity < 40"` 表示温度大于38且湿度小于40时，才会触发该规则，执行配置。

- 二进制数据格式

  目前二进制格式WHERE语句中仅支持内置函数及条件表达式，无法使用payload中的字段。

SQL结果

SQL语句执行完成后，会得到对应的SQL结果，用于下一步转发处理。如果payload数据解析过程中出错会导致规则运行失败。 转发数据动作中的表达式需要使用 ${表达式} 引用对应的值。

对于上文例子，配置转发动作时，可以`${t}`、`${deviceName}`和`${loaction}`获取SQL解析结果，如果要将数据存储到`TableStore`，配置中可以使用`${t}`、`${deviceName}`和`${loaction}`。

数组使用说明

数组表达式需要使用双引号，比如设备消息为：`{a:[{v:1},{v:2}]}`，那么select中引用`a[0]`的写法为：`select ".a[0]" data1,".a[1].v" data2`，则`data1=[{v:1}]`，`data2=2`

条件表达式支持列表

操作符	描述	举例
=	相等	color = ‘red’
<>	不等于	color <> ‘red’
AND	逻辑与	color = ‘red’ AND siren = ‘on’
OR	逻辑或	color = ‘red’ OR siren = ‘on’
( )	括号代表一个整体	color = ‘red’ AND (siren = ‘on’ OR isTest)
+	算术加法	4 + 5
-	算术减	5 - 4
/	除	20 / 4
*	乘	5 * 4
%	取余数	20 % 6

<	小于	5 < 6
<=	小于或等于	5 <= 6
>	大于	6 > 5
>=	大于或等于	6 >= 5
函数调用	支持函数，详细列表请参考函数列表。	deviceId()
JSON属性表达式	可以从消息payload以json表达式提取属性。	state.desired.color,a.b.c[0].d
CASE … WHEN … THEN … ELSE … END	Case 表达式	CASE col WHEN 1 THEN ‘Y’ WHEN 0 THEN ‘N’ ELSE ‘’ END as flag
IN	仅支持枚举，不支持子查询。	比如 where a in(1,2,3)。不支持以下形式： where a in(select xxx)
like	匹配某个字符， 仅支持%号通配符，代表匹配任意字符串。	比如 where c1 like ‘%abc’, where c1 not like ‘%def%’

3.6 函数列表

规则引擎提供多种函数，您可以在编写SQL时使用它们，实现多样化数据处理。

使用方法

您可以在SQL语句中使用函数获取数据或者对数据做处理。

例如：下面代码中用到了deviceName()，abs(number)，topic(number)三个函数。

```
SELECT case flag when 1 then '开灯' when 2 then '关灯' else '' end flag
 ,deviceName(),abs(temperature) tmr FROM "/topic/#" WHERE temperature>
10 and topic(2)='123'
```

说明：

使用时请注意，通常单引号代表常量，不带引号或双引号代表变量。如`select “a” a1, ‘a’ a2, a a3`中，`a1`与`a3`等效，`a2`代表常量`a`。

函数名	函数说明
abs(number)	返回绝对值。
asin(number)	返回 number的反正弦。

attribute(key)	返回key所对应的设备属性值。
concat(string1, string2)	字符串连接 示例：concat(field,’a’)
cos(number)	返回number的余弦。
cosh(number)	返回number的双曲余弦（hyperbolic cosine）。
crypto(field,String)	对field的值进行加密。 第二个参数String为算法字符串。可选：MD2，MD5，SHA1，SHA-256，SHA-384，SHA-512。
deviceName()	返回当前设备名称。
endswith(input, suffix)	判断input是否以suffix结尾。
exp(number)	返回指定数字的指定次幂。
floor(number)	返回一个最接近它的整数，它的值小于或等于这个浮点数。
log(n, m)	返回自然对数。 如果不传m，则返回log(n)。
lower(string)	返回小写字符串。
mod(n, m)	n%m 余数。
nanvl(value, default)	返回属性值。 若属性值为null，则返回default。
newuuid()	返回一个随机uuid字符串。
payload(textEncoding)	返回设备发布消息的payload转字符串。 字符编码默认UTF-8，即 payload()默认等价于payload(‘utf-8’)。
power(n,m)	返回n的m次幂。
rand()	返回[0~1)之间随机数。
replace(source, substring, replacement)	对某个目标列值进行替换。 示例：replace(field,’a’,’1’)。
sin(n)	返回 n的正弦。
sinh(n)	返回 n的双曲正弦（hyperbolic sine）。
tan(n)	返回 n的正切。
tanh(n)	返回n的双曲正切（hyperbolic tangent）。
timestamp(format)	返回当前系统时间。

	format可选。如果为空则返回当前系统时间戳毫秒值，比如 timestamp() = 1232323233，timestamp('yyyy-MM-dd HH:mm:ss.SSS')=2016-05-30 12:00:00.000。
topic(number)	返回Topic分段信息。 如，有一个Topic： /abcdef/ghi。函数 topic() 返回“ /abcdef/ghi”，topic(1) 返回“abcdef”， topic(2) 返回“ghi”。
upper(string)	返回大写字符。

3.7 Topic数据格式

使用规则引擎，您需要基于Topic编写SQL处理数据。基础版Topic，数据格式是您自定义的，物联网平台不做处理。高级版Topic，物联网平台定义了Topic格式，此时您需要根据平台定义的数据格式，处理数据。本文将为您讲解高级版Topic的数据格式。

设备属性上报

通过该Topic获取设备上报的属性信息。

数据流转TOPIC：`/{productKey}/{deviceName}/thing/event/property/post`

数据格式：

```
{
"iotId":"4z819VQHk6VSLmmBJfrf00107ee200",
"productKey":"1234556554",
"deviceName":"deviceName1234",
"gmtCreate":1510799670074,
"deviceType":"Ammeter",
"items":{
"Power":{
"value":"on",
"time": 1510799670074
},
"Position":{
"time":1510292697470,
"value":{
"latitude":39.90,
"longitude":116.38
}
}
}
}
```

参数说明：

参数	类型	说明
iotId	String	设备在平台内的唯一标识

参数	类型	说明
productKey	String	产品的唯一标识
deviceName	String	设备名称
deviceType	String	设备类型
items	Object	设备类型
Power	String	属性名称，产品所具有的属性名称请参考TSL描述
Position	String	属性名称，产品所具有的属性名称请参考TSL描述
value	根据TSL定义	属性值
time	Long	属性产生时间，如果设备没有上报默认采用云端生成时间
gmtCreate	Long	数据流转消息产生时间

设备上报事件

通过该topic获取设备上报的事件信息。

数据流转TOPIC：`/{productKey}/{deviceName}/thing/event/{tsl.event.identifier}/post`

数据格式：

```
{
"identifier":"BrokenInfo",
"name":"损坏率上报",
"type":"info",
"iotId":"4z819VQHk6VSLmmBJfrf00107ee200",
"productKey":"X5eCzh6fEH7",
"deviceName":"5gJtxDVeGAkaEztpisjX",
"gmtCreate":1510799670074,
"value":{
"Power":"on",
"Position":{
"latitude":39.90,
"longitude":116.38
}
},
"time":1510799670074
}
```

参数说明：

参数	类型	说明
iotId	String	设备在平台内的唯一标识
productKey	String	产品的唯一标识
deviceName	String	设备名称
type	String	事件类型，事件类型参考TSL描述
value	Object	事件的参数
Power	String	事件参数名称
Position	String	时间参数名称
time	Long	事件产生时间，如果设备没有上报默认采用远端时间
gmtCreate	Long	数据流转消息产生时间

网关发现子设备数据流转

在一些场景中网关能够检测到子设备，并将检测到的子设备信息上报。此时可以通过该Topic获取到上报的信息。

数据流转TOPIC：`/{productKey}/{deviceName}/thing/list/found`

数据格式：

```
{
"gwIotId":"4z819VQHk6VSLmmBJfrf00107ee200",
"gwProductKey":"1234556554",
"gwDeviceName":"deviceName1234",
"devices":[{
"productKey":"12345565569",
"deviceName":"deviceName1234",
"iotId":"4z819VQHk6VSLmmBJfrf00107ee201"
}]
}
```

参数说明：

参数	类型	说明
gwIotId	String	网关设备在平台内的唯一标识
gwProductKey	String	网关产品的唯一标识
gwDeviceName	String	网关设备名称
iotId	String	子设备在平台内的唯一标识

参数	类型	说明
productKey	String	子设备产品的唯一标识
deviceName	String	子设备名称

设备上下线状态数据流转

通过该Topic获取设备的上下线状态。

数据流转TOPIC：`/{productKey}/{deviceName}/mqtt/status`

数据格式：

```
{
"productKey":"1234556554",
"deviceName":"deviceName1234",
"gmtCreate":1510799670074,
"deviceType":"Ammeter",
"iotId":"4z819VQHk6VSLmmBJfrf00107ee200",
"action":"online",
"status"{
"value":"1",
"time":1510292697471
}
}
```

参数说明：

参数	类型	说明
iotId	String	设备在平台内的唯一标识
productKey	String	产品的唯一标识
deviceName	String	设备名称
status	Object	设备状态
value	String	状态值 1上线，0离线
time	Long	设备上下线时间
gmtCreate	Long	数据流转消息产生时间
action	String	设备状态变更动作，online上线， offline离线

设备下行指令结果数据流转

通过该Topic可以获取，通过异步方式下发指令给设备，设备进行处理后返回的结果信息。如果下发指令过程中出现错误，也可以通过该Topic得到指令下发的错误信息。

数据流转TOPIC：`/{productKey}/{deviceName}/thing/downlink/reply/message`

数据格式：

```
{
"gmtCreate": 1510292739881,
"iotId": "4z819VQHk6VSLmmBJfrf00107ee200",
"productKey": "1234556554",
"deviceName": "deviceName1234",
"requestId": 1234,
"code": 200,
"message": "success",
"topic": "/sys/1234556554/deviceName1234/thing/service/property/set",
"data": {}
}
```

参数说明：

参数	类型	说明
gmtCreate	Long	UTC时间戳
iotId	String	设备在平台内的唯一标识
productKey	String	产品Key
deviceName	String	设备名称
requestId	Long	阿里云产生和设备通信的信息id
code	Integer	调用的结果信息
message	String	结果信息说明
data	Object	设备返回的结果，非透传之间返回设备结果，透传则需要经过脚本转换

错误信息：

参数	类型	说明
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
9200	device not actived	设备没有激活
9201	device offline	设备不在线

参数	类型	说明
403	request forbidden	请求被禁止，由于欠费导致

3.8 使用实例

3.8.1 数据转发到另一Topic

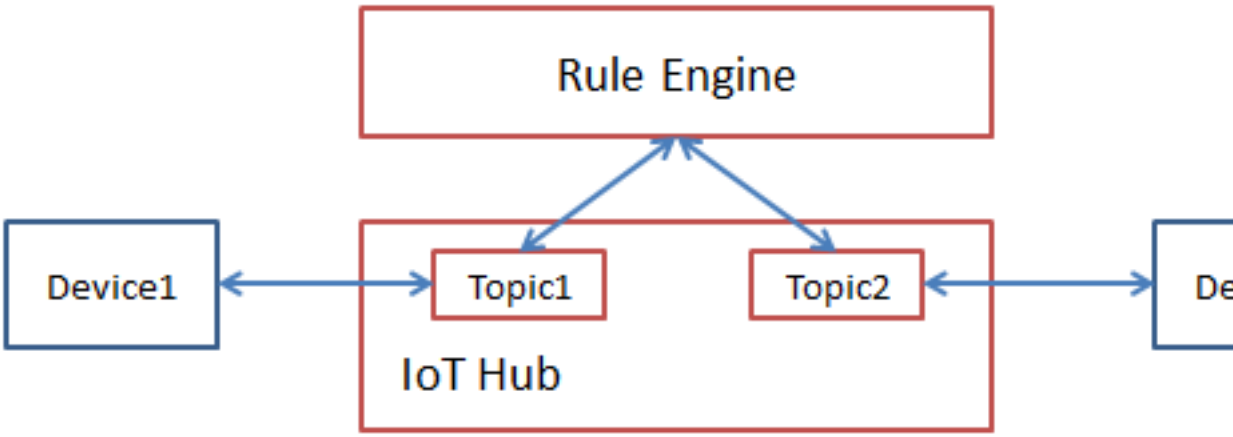
您可以将依据SQL规则处理完的数据，设置转发到另一个Topic中，实现M2M或者更多其他场景。

前提条件

在设置转发之前，您需要参考设置规则引擎编写SQL完成对数据的处理。

背景信息

本文将教您如何设置数据从Topic1中依照规则引擎设置转发到Topic2内：



操作步骤

1. 单击数据转发一栏的添加操作。出现添加操作页面。

2. 按照页面提示，设置参数。
   - 选择操作：此处选择发布到另一个**Topic**。
   - Topic：选择您需要把数据转发到哪一个Topic中。此处选择产品后，还需补充完整该Topic。您可以使用`${}`表达式引用上下文值，如填`${dn}/get`表示从消息中选中devicename，该Topic的后缀为get。

3.8.2 数据转发到MQ

您可以使用规则引擎，将数据转发到消息队列（以下简称为MQ）中，从而实现消息依次从设备、物联网平台、MQ到应用服务器之间的全链路高可靠传输能力。

前提条件

在设置转发之前，您需要参考设置规则引擎编写SQL完成对数据的处理。

操作步骤

1. 单击数据转发一栏的添加操作，出现添加操作页面。选择发送数据到消息队列**(Message Queue)**中。

2. 按照界面提示，设置参数。
   - 选择操作：选择MQ。
   - 地域：选择MQ Topic所在地域。
   - Topic：根据业务选择需要写入数据的Topic。
   - 授权：授权平台将数据写入MQ Topic的权限。

说明：

MQ非铂金实例的Topic不支持跨地域的访问写入，此时需保证规则引擎与MQ Topic在同一地域下。若您已购买并使用铂金实例，则无需关心地域问题。

转入MQ的规则启动之后，平台即可将数据写入MQ的Topic，您可以使用MQ实现消息收发，具体使用方法，请参考*MQ*订阅消息。

3.8.3 数据转发到表格存储

您可以使用规则引擎，将数据转发到表格存储（Table Store）中。

前提条件

在设置转发之前，您需要参考设置规则引擎编写SQL完成对数据的处理。

操作步骤

1. 单击数据转发一栏的添加操作，出现添加操作页面。选择存储到表格存储**(Table Store)**。

添加操作

选择操作：

存储到表格存储(Table Store)

该方法将数据插入到表格存储(Table Store)中, 详情请参考
中, 详情请参考文档

* 地域：

华东 2

* 实例：

ShanghaiRegion

* 数据表：

shanghai_61034

* 主键：

pk1 *值： 12

* 角色：

AliyunIOTAccessingOTSRole

2. 按照界面提示，设置参数。
   - 选择操作：选择Table Store。
   - 地域、实例、数据表：选择Table Store数据表的基本信息，用于数据存储。
   - 主键：Table Store数据表都包含主键，选择好数据表之后，控制台将自动读出该表的主键，此处需要您配置主键的值。
   - 角色：授权平台将数据写入Table Store。此处需要您先创建一个具有Table Store写入权限的角色，然后将该角色赋予给规则引擎。这样，规则引擎才可以将处理完成的数据写入数据表中。

后续操作

示例

使用SQL抽取JSON数据：`{"device":"bike","product":"xxx","data2":[{...}]}`。因业务需要，需将此JSON数据存入Table Store中，主键是`device`，`product`，`id`。

配置及效果：

1. 在控制台配置主键的值，输入`${device}`。当有消息过来并触发规则，主键device就会存入JSON中device的value值。主键product同理。

   
   说明：
   ${}是转义符，如果不输入该转义符，存入的将会是一个常量。
2. 规则会自动匹配主键是否为自增列，如果是自增列，将自动返填`AUTO_INCREMENT`，并且不能编辑。
3. 主键配置完成后，平台将自动解析JSON中除主键外的key值，并依据key值自动创建Table Store的数据列。此例中，将会自动创建data1和data2两个列，且在每列下面存入对应的value值。

   
   说明：
   目前只支持一级JSON解析，不支持嵌套JSON的解析。因此，该示例中data2下面将会以字符串形式存入整个嵌套JSON，而不会对嵌套JSON再次进行解析创建列。

3.8.4 数据转发到DataHub

物联网平台用于接入和管理设备，数据存储和计算将交给阿里云其他产品。比如，您可以使用规则引擎将数据转到*DataHub*上，由DataHub为下游流计算、MaxCompute等提供实时数据，帮助用户实现更多计算场景。

前提条件

在设置转发之前，您需要参考设置规则引擎编写SQL完成对数据的处理。

操作步骤

1. 单击数据转发一栏的添加操作，出现添加操作页面。选择发送数据到**DataHub**中。

添加操作

选择操作：

发送数据到DataHub中

该操作将数据插入到Datahub中, 详情请参考文档

* 地域：

华东 2

* Project：

iotdata

* Topic：

iotdata

* 角色：

AliyunIOTAccessingDataHubRole

2. 按照界面提示，设置参数。

   - 选择操作：选择发送数据到DataHub中。
   - 选择DataHub对应的地域、Project和Topic。选完Topic后，规则引擎会自动获取Topic中的Schema，规则引擎筛选出来的数据将会映射到对应的Schema中。

     说明：

     - 将数据映射到Schema时，需使用`${}`，否则存入表中的将会是一个常量。
     - Schema与规则引擎的的数据类型必须保持一致，不然无法存储。
   - 角色：授权物联网平台将数据写入DataHub。此处需要您先创建一个具有DataHub写入权限的角色，然后将该角色赋予给规则引擎。这样，规则引擎才可以将处理完成的数据写入DataHub中。

3.8.5 数据转发到RDS

您可以设置规则引擎，将处理后的数据转发到云数据库（以下简称RDS）的VPC实例中。

使用须知

- 目前转发至RDS只发布在华东2、硅谷和新加坡三个节点。
- 只支持同region转发，不支持跨区转发。比如，华东2节点的平台数据只能转发到华东2的RDS中。
- 只支持转发到VPC实例。
- 只支持MySQL实例。
- 支持普通数据库和高权限数据库的转发。

准备工作

在设置转发之前，您需要参考设置规则引擎编写SQL完成对数据的处理。

操作步骤

1. 单击数据转发一栏的添加操作，出现添加操作页面。选择存储到云数据库(RDS)中。

添加操作

选择操作：

存储到云数据库(RDS)中

该操作将数据插入到云数据库(RDS)中, 详情请参考云数据库(RDS)中, 详情请参考文档

特别提醒：此操作仅针对专有网络的RDS实例，并且将会在您的RDS白名单中添加一条记录100.104.123.0/24，用于IoT访问您的数据库，请勿删除。

区域：

华东2

* VPC实例：

rm-uf63b1cji3z6xgpk5 创建实

* MySQL数据库：

asdf

* 账号：

请选择 创建账

* 请输入密码：

请输入该账号的密码

* 表名：

请输入已存在的表名

* 键：

RDS表中的字段

2. 按照界面提示，设置参数。

   - 选择操作：选择存储到云数据库(RDS)中。
   - VPC实例、MySQL数据库：根据您的业务选择当前区域下的VPC实例和MySQL数据库。

     说明：
     如果是高权限数据库，需要您手动输入数据库名称。

   - 账号、密码：输入数据库的账号、密码。此账号密码应具有该数据库的读写权限，否则规则引擎无法将数据写入RDS。

     说明：
     规则引擎获得账号后，仅负责将规则匹配的数据写进数据库中，不会做其他操作。

   - 表名：输入数据库中已建立的数据表名，规则引擎将把数据写入这张表上。
   - 字段：此处输入数据表的字段，规则引擎将把处理后的数据存入该字段中。
   - 值：此处填写输入数据表字段的值。可以使用转义符`$`，格式为`${key}`，即提取Topic中key对应的Value值作为输入值。

     规则引擎的SQL为：`SELECT tem FROM mytopic`；RDS数据库有一张表，表中有`tem`字段，类型是`String`。

     控制台配置时，字段处填入RDS数据表的字段**`tem`**，值处填入规则引擎筛选出来的JSON字段**`${key}`**。

     说明：
     - 值处需使用`${}`，否则填入表中的将是一个常量。
     - 字段与值的数据类型需保持一致，否则无法存储成功。

3. 配置完成后，规则引擎为了连接RDS，会在RDS的白名单中添加下列IP。若这些IP未出现，请手动添加。

   - 华东2：100.104.123.0/24
   - 亚太东南1（新加坡）：100.104.106.0/24
   - 美国西部1（硅谷）：100.104.8.0/24

   RDS控制台的白名单示例如下：

3.8.6 数据转发到MNS

您可以使用规则引擎，将处理后的数据转发到消息服务（以下简称MNS）中。两者结合使用，实现设备端与服务端之间高性能的消息闭环传输。

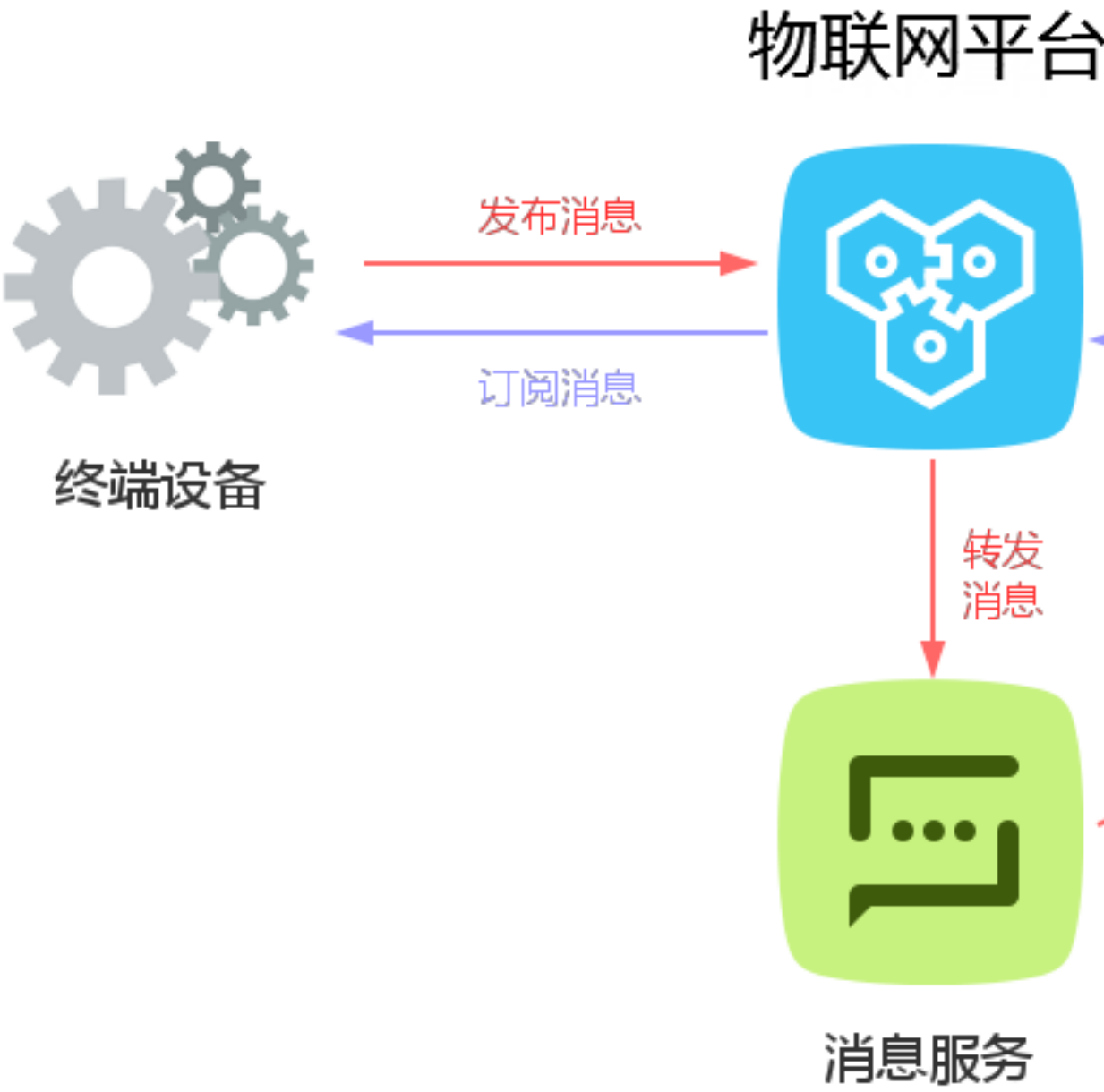
数据转发到消息服务（Message service）中

- 设备发送数据到服务端

设备发布消息到物联网平台中，物联网平台通过规则引擎将消息进行处理并转发到MNS的主题中，最后客户的应用服务器调用消息服务的接口订阅消息。**这种方式优势是MNS可以保证消息的可靠性，避免了服务端不可用时的消息丢失，同时MNS在处理大量消息并发时有削峰填谷的作用，保证服务端不会因为突然的并发压力导致服务不可用。**

- **服务端发送数据到设备**

  客户的应用服务器调用物联网平台的OpenAPI发布数据到物联网平台中，然后设备从物联网平台中订阅消息。



3.8.7 数据转发到HiTSDB

您可以配置规则引擎将处理过的数据转发到时间序列数据库（*HiTSDB*）的实例中。

使用须知

- 目前转发至HiTSDB仅发布在华东2节点。
- 只支持专有网络下HiTSDB实例.
- 只支持同region转发。例如：华东2节点的套件数据只能转发到华东2的HiTSDB中。

- 只支持JSON格式数据转发。

准备工作

操作步骤

1. 单击数据转发一栏的添加操作，出现添加操作页面。选择存储到高性能时间序列数据库**(HiTSDB)**中。

添加操作

选择操作：

存储到高性能时间序列数据库(HiTSDB)中

该操作将数据插入到时序数据库(HiTSDB)中, 详情请参考文档

区域：

华东2

* VPC实例：

ts-uf6n532yvs088o47o

* timestamp：

请输入时间戳标签

* tag：

RDS表中的字段

* 值：

Topic中的字段 删除

添加字段

* 角色：

请选择RAM角色...

确定

2. 按照界面提示，设置参数。

   - 选择操作：此处选择高性能时间序列数据库(HiTSDB)
   - 地域、实例：选择处理后的数据将要存入哪个数据库实例中。
   - timestamp：此处的值必须为Unix时间戳，如1404955893000。有如下两种配置方式：
     - 使用转义符`${}`表达式，例如`${time}`。此时timestamp的值为Topic中time字段对应的值。建议使用此方式。
     - 使用规则引擎函数`timestamp()`，此时timestamp的值为规则引擎服务器的时间戳。
   - tag：必须使用常量配置，值有三种配置方式：
     - 使用转义符`${}`表达式，例如`${city}`，此时值为Topic中city字段对应的值。建议使用此方式。
     - 使用规则引擎函数规定的一些函数，例如`deviceName()`，此时值为DeviceName。
     - 使用常量配置，例如beijing，此时值为beijing。
   - 授权：勾选同意物联网平台向HiTSDB写数据。此时，规则引擎会向时间序列数据库实例中添加网络白名单，用于IoT访问您的数据库， 请勿删除这些IP段。

示例

规则引擎的SQL：

```
SELECT time,city,power,distance, FROM "/myproduct/myDevice/update"；
```

配置的规则如操作步骤所示。

发送的消息：

```
{
"time": 1513677897,
"city": "beijing",
"distance": 8545,
"power": 93.0
}
```

规则引擎会向高性能时间序列数据库中写入的两条数据：

```
数据：timestamp:1513677897, [metric:distance value:8545]
tag： device=myDevice,product=bikes,cityName=beijing
```

```
数据：timestamp:1513677897, [metric:power value:93.0]
```

```
tag:device=myDevice,product=bikes,cityName=beijing
```

注意

- 发送的消息中除了在规则引擎中配置为timestamp或者tag值的字段外，其他字段都将作为metric写入时间序列数据库。如上例所示除time和city以外，distance和power作为两个metric写入数据库。
- metric的值只能为**数值类型**，否则会导致写入数据库失败。
- 用户要保证在规则引擎中配置的tag-值键值对能够获取到，如果获取不到任意一个tag-值键值对，会导致写入数据库失败。
- tag-值键值对限制最多输入8个。
- metric、tag和tag值只可包含大小写英文字母、中文、数字，以及特殊字符`-_./():,[]=‘`
- 规则引擎在HiTSDB的白名单中添加以下IP段用以访问数据库。若IP未出现，请手动添加。

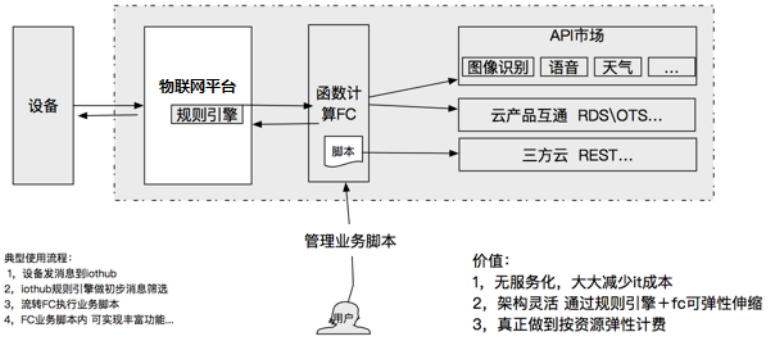
  华东2：100.104.76.0/24

  HiTSDB控制台白名单示例如下：


实例详情
实例详情
实例监控
数据时效
数据清理
时间线清理
数据查询
产品文档
实例详情
基础信息
实例ID： ts-
地域： 华东 2
专有网络： vpc-
VPC网络地址： ts-
公共网络地址： 申请公共网络
运行状态
运行状态： 运行中
创建时间： 2017-09-14 15:54:20
配置信息
存储容量： 200 GB
最大支持时间线： 1000000
白名单状态
网络地址白名单：
100.104.76.0/24

3.8.8 转发数据到函数计算

您可以使用规则引擎，将处理后的数据转发至函数计算（Function Compute）中。



操作流程概述：

1. 函数计算控制台创建好服务、函数。
2. 创建规则，将IoT数据处理并转发到函数计算中，启动规则。
3. 使用配置了规则的Topic发送一条消息。
4. 查看函数计算服务实时监控大盘查询函数执行情况，或者根据函数的具体业务逻辑查看结果是否正确。

操作步骤

1. 登录函数计算控制台，创建服务与函数。
   a. 创建服务。其中，**服务名称**必须填写，其余参数请根据您的需求设置。


函数计算
服务列表
搜索服务
1/2
计量数据
监控数据每小时更新并尽最大可能
本月执行次数
5.901万
触发器
对象存储
结合OSS事件即时触发函数处
数据
HTTP 触发器
NEW
通过访问HTTP URL来调用函
函数模版
图像分类
您仅需要通过OSS上传图片，
计算服务进行图像识别并对结
类存储的模板。

b. 创建服务成功后，创建函数。

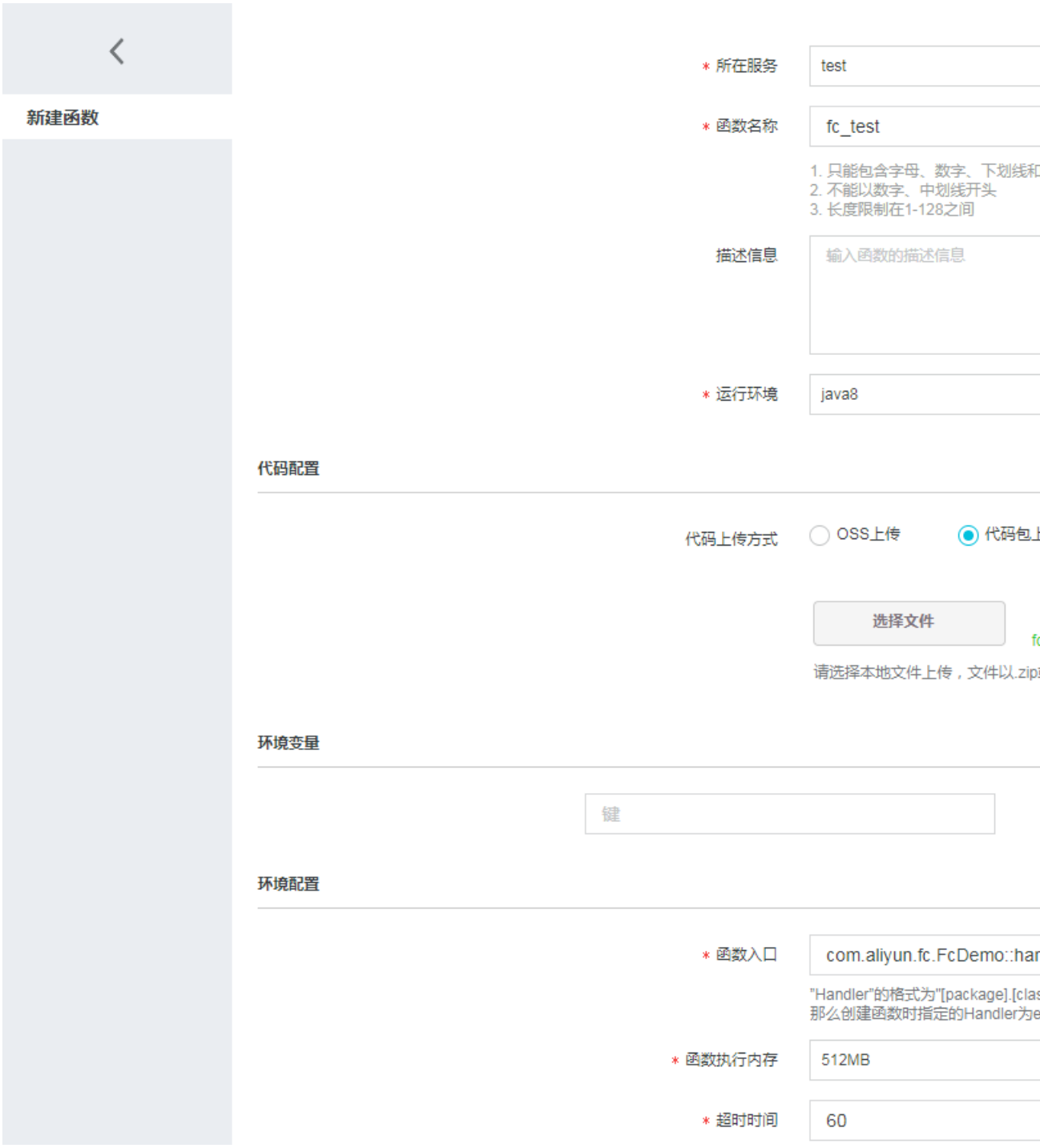

华东2 (上海) > test
服务概览
函数列表
搜索函数
1/1
计量数据
监控数据每小时更新并尽最大可能推送
本月执行次数
0.0
基础配置
服务名称 test
创建时间 06/28/2018, 09:
功能描述
高级配置
日志项目
服务角色

c. 选择函数模板，此处以空白函数模板为示例。


新建函数
新建函数
函数模版
函数模版选择
业务模板提供了示例配置和代码，为您创建函数时
选择全部的语言
空白函数
空白函数模板会创建一个空白函数，通过
置和代码开发，完成函数的创建。
flask-web
python2.7
通过该模板的示例代码，用户可以通过u
serverless.
Previous
1
2
3
Next

d. 设置函数参数。

此处设置函数的逻辑为直接在函数计算中显示获取的数据。



其中，

所在服务：选择*1.a*中创建的服务。

函数名称：设置您的函数名称。

运行环境：设置函数运行的环境，此示例中选择java8。

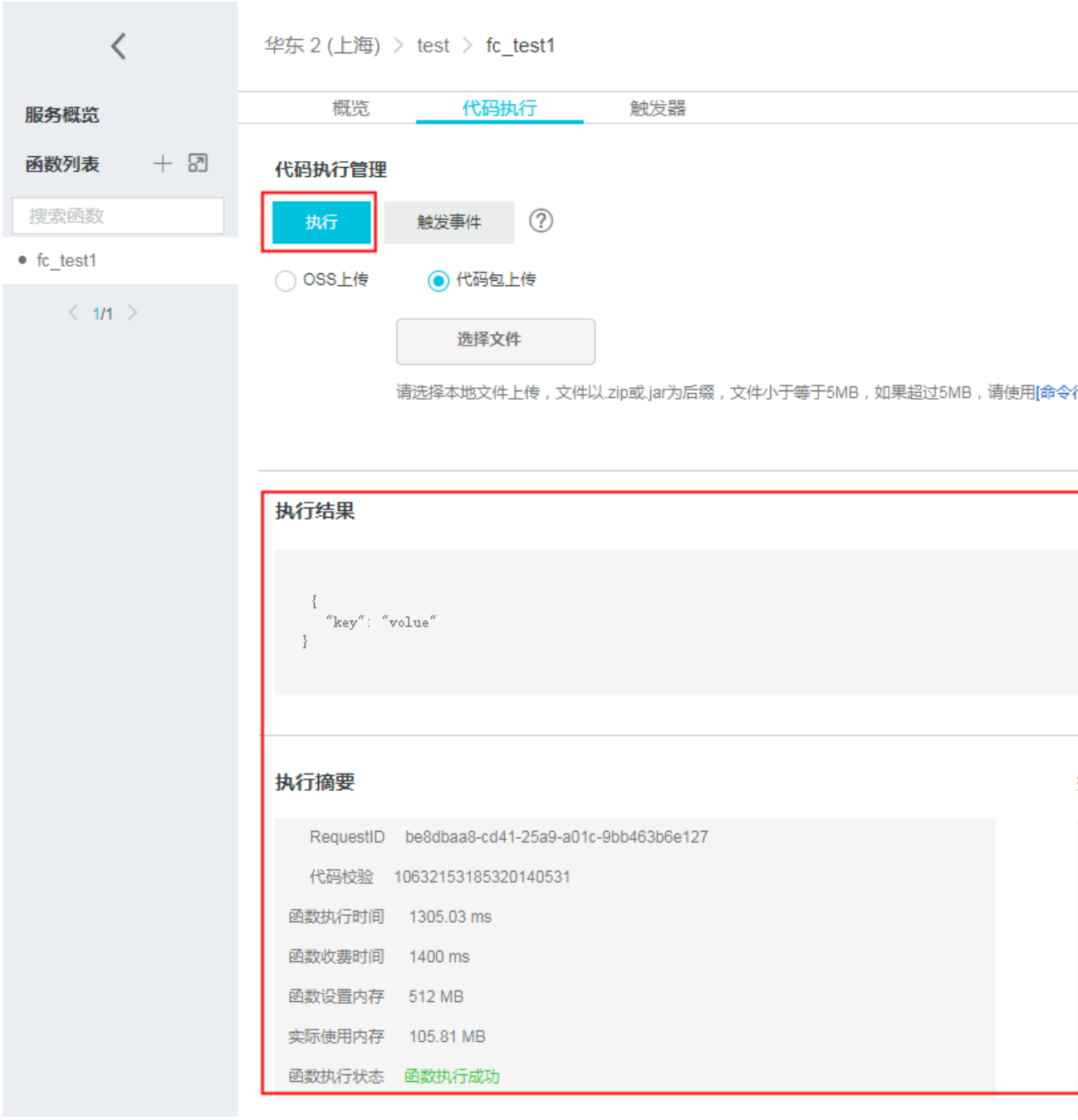
代码配置：上传您的代码。

函数入口：设置您的函数在函数计算系统运行的调用入口，此示例中必须设置为`com.aliyun.fc.FcDemo::handleRequest`。

其余参数的值请根据您的需求，参见函数计算设置。

e. 函数执行验证。

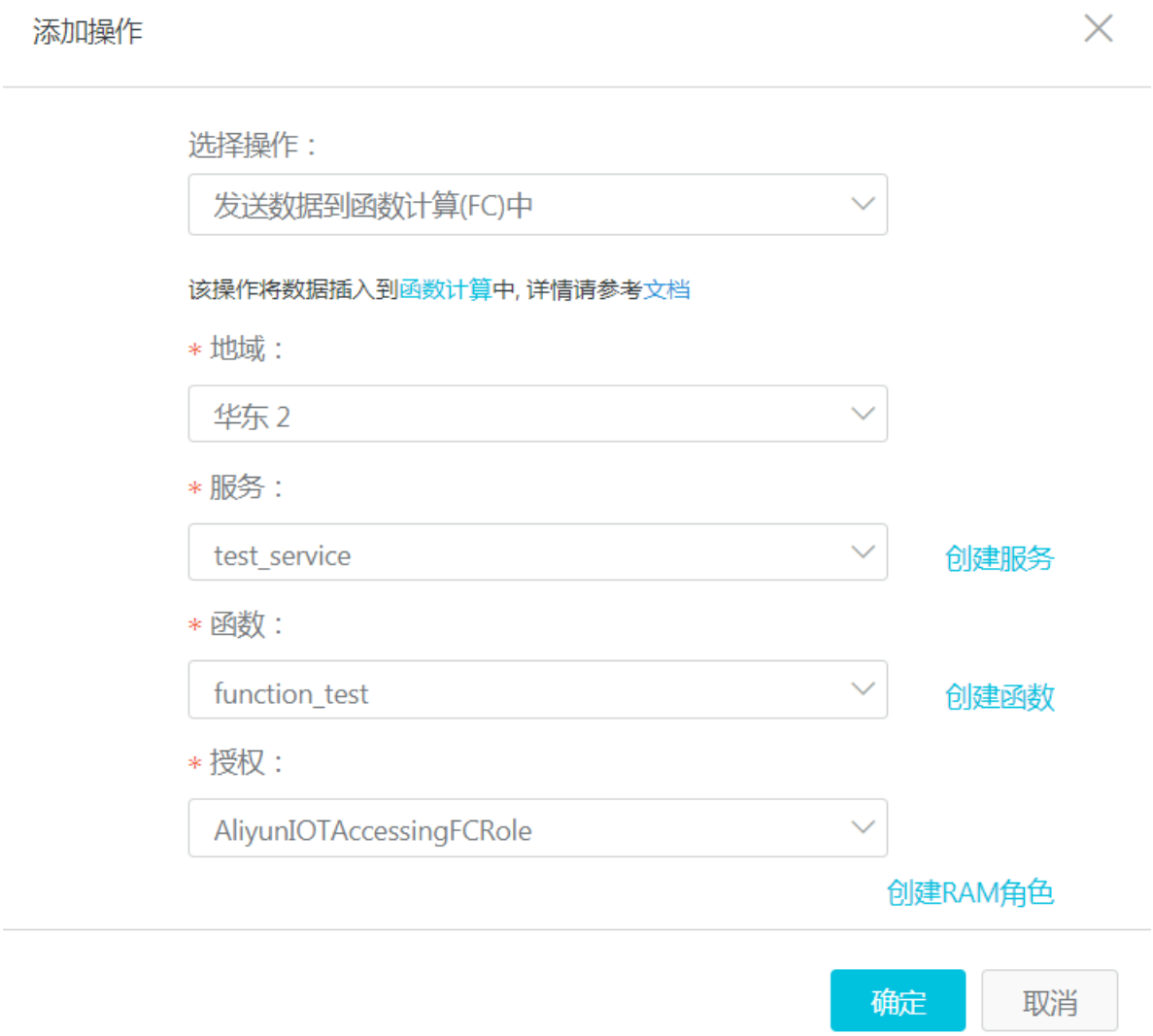
函数成功创建后，可以直接在函数计算的控制台执行，以验证函数执行情况。函数计算会直接将函数的输出和请求的相关信息打印在控制台上。



2. 函数正常执行后，配置IoT规则引擎。
3. 在设置转发之前，您需要参考设置规则引擎编写SQL完成对数据的处理。

> 说明：
> JSON格式和二进制格式都支持转发到函数计算中

4. 单击规则名称，进入规则详情页面。
5. 单击数据转发一栏的添加操作，并在弹出的添加操作页面中设置参数。



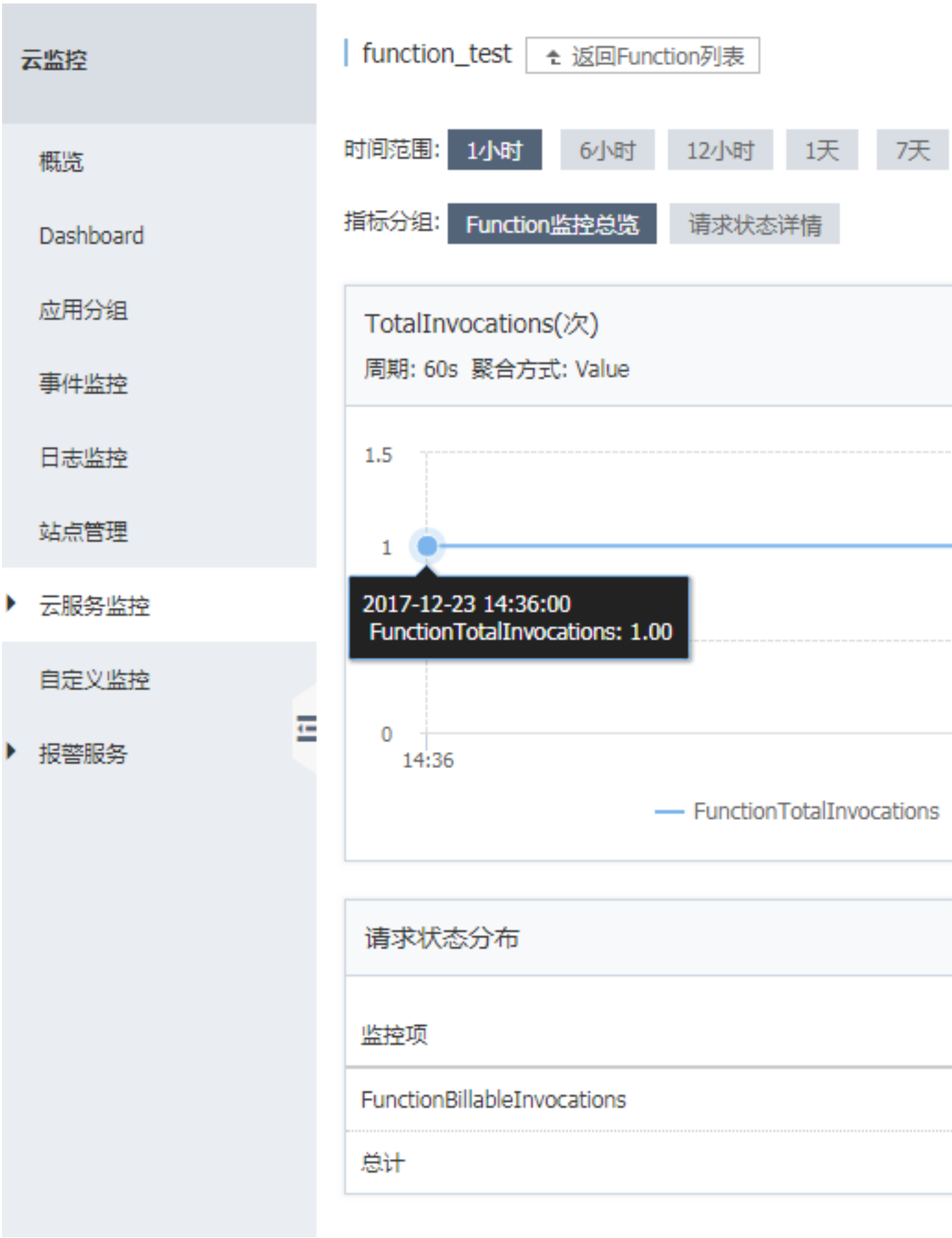
- 选择操作：发送数据到函数计算(FC)中。
- 地域：根据您的业务选择将处理后的数据转发至某个地域。如果该地域没有对应资源，需要到函数计算控制台创建相应的资源。

  说明：
  目前只支持华东2节点
- 服务：根据您的地域选择服务。若没有服务，请单击**创建服务**。
- 函数：根据您的地域选择函数。若没有函数，请单击**创建函数**。
- 角色：授权物联网平台操作函数的权限。此处需要您先创建一个具有执行函数权限的角色，然后将该角色赋予给规则引擎。

6. 启动规则。规则运行后，IoT将根据编写的SQL，将处理后的数据转发到函数计算中。函数计算根据您函数定义的逻辑，直接将接收到的数据显示在函数计算控制台上。

结果验证

函数计算控制台针对函数的执行情况有监控统计。统计有大概5分钟的延时，可以通过监控大盘查询函数的执行情况。


云监控
概览
Dashboard
应用分组
事件监控
日志监控
站点管理
云服务监控
自定义监控
报警服务
function_test
返回Function列表
时间范围:
1小时
6小时
12小时
1天
7天
指标分组:
Function监控总览
请求状态详情
TotalInvocations(次)
周期: 60s 聚合方式: Value
1.5
1
0
2017-12-23 14:36:00
FunctionTotalInvocations: 1.00
14:36
FunctionTotalInvocations
请求状态分布
监控项
FunctionBillableInvocations
总计

4 设备开发指南

4.1 下载设备端SDK

本章节介绍各种版本的SDK文件的下载路径。

- C语言版、JAVA版、IOS版和Android版的SDK支持MQTT协议。
- 仅C语言版的SDK支持HTTPS和CoAP协议。

开发者也可以使用开源的MQTT、CoAP和HTTP客户端自行开发，实现和云端的连接。

其中，官方提供的C语言版实现了更多逻辑，比如认证逻辑、OTA、设备影子和网关主子设备能力等功能，如果您使用开源版本，扩展能力需要按照已有规范实现。

C语言版

设备端SDK代码托管迁移到Github，下载地址为

- *https://github.com/aliyun/iotkit-embedded*
- *https://github.com/aliyun/iotkit-embedded/wiki*

已适配平台，如表 *4-1: 已适配平台*所示，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。

表 4-1: 已适配平台

开发板	网络支持	厂商SDK链接	开发板购买链接	阿里云SDK版本
ESP32	Wi-Fi	*esp32-aliyun*	乐鑫信息科技	V2.01
ESP8266	Wi-Fi	*esp8266-aliyun*	乐鑫信息科技	V2.01

支持的设备端SDK版本如表 *4-2: SDK*版本所示。

表 4-2: SDK版本

版本号	发布日期	开发语言	开发环境	下载链接	更新内容
版本V2.10	2018/03/31	C语言	64位 Linux, GNU Make	*RELEASED_V2_10_20180331.zip*	• 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。

版本号	发布日期	开发语言	开发环境	下载链接	更新内容
					• 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。 • 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。 • 支持iTLS功能: 设置FEATURE_MQTT_DIRECT_NOTLS = 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。 • 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。 • 优化主子设备功能：主子设备添加部分功能。
版本V2.03	2018/01/31	C语言	64位 Linux, GNU Make	*RELEASED_V2_03_20180131.zip*	• 支持主子设备功能: 设置FEATURE_SUBDEVICE_ENABLED = y，可以支持子设备通过主设备(网关设备)进行数据交互。 • 升级HTTP通道: 优化HTTP流程。 • 优化TLS: 修复内存泄漏问题。 • 优化OTA的配置: 可以更合理的开关OTA功能。 • 升级MQTT通道: 支持topic更长，更多的订阅请求，MQTT支持多线程。
版本V2.02	2017/11/30	C语言	64位 Linux, GNU Make	*RELEASED_V2_02_20171130.zip*	• 正式的多平台支持: 使用make reconfig可弹出和选择Ubuntu16.04以外的已适配平台。 • 新增Windows版本: 支持用mingw32工具链编译Win7版本的库和例程。 • 新增OpenSSL适配: 新增了配合openssl-0.9.x+Windows版本的HAL参考实现。

版本号	发布日期	开发语言	开发环境	下载链接	更新内容
					• 优化HTTP接口: HTTP通道方面接口优化, 支持发送报文而不断开TLS连接。 • 自包含的安全库: 新增裁剪版本的安全库 `mbedtls`, 目前可适配Linux/Windows平台。
版本V2.01	2017/10/10	C语言	64位 Linux, GNU Make	*RELEASED_V2_01_20171010.zip*	• 新增CoAP+OTA: 允许配置成基于CoAP通知方式的OTA。 • 新增HTTP+TLS: 在MQTT/CoAP之外, 新增HTTP的通道。 • 细化OTA状态: 优化OTA部分代码, 使云端可以更细化的区分设备的OTA固件下载状态。 • ArmCC支持: 修正了SDK在ArmCC编译器编译时会出现的报错。
版本V2.00	2017/08/21	C语言	64位 Linux, GNU Make	*RELEASED_V2_00_20170818.zip*	• 新增MQTT直连: 支持更快更轻的连接IoT套件, 去掉对HTTPS/HTTP的依赖, 可看公告。 • 新增CoAP通道: 基于UDP, 在纯上报数据场景更节省资源 ， 可看公告。 • 新增OTA通道: 提供一系列OTA相关的API, 可查询/触发/下载用户自主上传的固件。 • 升级构建系统: 支持更灵活的组织和配置SDK。
版本V1.0.1	2017/06/29	C语言	64位 Linux, GNU Make	*RELEASED_V1_0_1_20170629.zip*	• **华东2站点:** 第一个正式配合华东2站点的设备端SDK, 全源码发布。 • **新增设备影子功能:** 具体可参看设备影子介绍页面。

JAVA版本

表 4-3: JAVA版本

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。	*iotx-sdk-mqtt-java* ：MQTT的JAVA版只是使用开源库实现的一个demo，仅用于参考。

Android版本

下载地址：*https://github.com/eclipse/paho.mqtt.android*

IOS版本

下载地址为：

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

其它开源库参考

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

4.2 设备多协议连接

4.2.1 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：

- /${productKey}/${deviceName}/update pub
- /${productKey}/${deviceName}/update/error pub
- /${productKey}/${deviceName}/get sub
- /sys/${productKey}/${deviceName}/thing/# pub&sub
- /sys/${productKey}/${deviceName}/rrpc/# pub&sub
- /broadcast/${productKey}/# 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。
- /${productKey}/${deviceName}/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数据标准。
- /sys/${productKey}/${deviceName}/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/${productKey}/${deviceName}/rrpc/request/${messageId}：用于同步请求，服务器会对消息Id动态生成topic， 设备端可以订阅通配符。
- /sys/${productKey}/${deviceName}/rrpc/request/+：收到消息后，发送pub消息到/sys/${productKey}/${deviceName}/rrpc/response/${messageId}，服务器可以在发送请求时，同步收到结果。

4.2.2 MQTT-TCP连接通信

本篇文档主要介绍基于TCP的MQTT连接，提供了两种设备认证模式。

- MQTT客户端域名直连，资源受限设备推荐。
- HTTPS发送授权后，连接MQTT特殊增值服务，比如设备级别的引流。

说明：

在进行MQTT CONNECT协议设置时：

- Connect指令中的KeepAlive有效范围为[60秒,300秒]，否则会拒绝连接。

- 如果同一个设备三元组，同时用于多个连接，可能导致客户端互相上下线。
- MQTT默认开源SDK会自动重连，您可以通过日志服务查看设备行为。

具体使用方式请参看\sample\mqtt\mqtt-example.c的示例。

MQTT客户端域名直连

- 不使用已有Demo

  完全使用开源MQTT包自主接入，可以参考以下流程：

  1. 如果使用TLS，需要 下载根证书。
  2. 使用MQTT客户端连接服务器，自主接入可以使用开源*MQTT*客户端参考，如果您对MQTT不了解，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 连接使用说明。

     - 连接域名：

       — 华东2节点：${productKey}.iot-as-mqtt.cn-shanghai.aliyuncs.com:1883

       — 美西节点：${productKey}.iot-as-mqtt.us-west-1.aliyuncs.com:1883

       — 新加坡节点：${productKey}.iot-as-mqtt.ap-southeast-1.aliyuncs.com:1883

       ${productKey}请替换为您的产品key。

     - mqtt的Connect报文参数：

       ```
       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排序后，根据signmethod加签。

       content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。

       - 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，可以不传递。

- 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（TLS直连模式）和3（TCP直连模式）。

示例：如果`clientId = 12345`，`deviceName = device`，`productKey = pk`，`timestamp = 789`，`signmethod=hmacsha1`，`deviceSecret=secret`，那么使用tcp方式提交给mqtt参数如下：

```
mqttclientId=12345|securemode=3,signmethod=hmacsha1,timestamp=789|
username=device&pk
password=hmacsha1("secret","clientId12345deviceNamedevicep
roductKeypktimestamp789").toHexString(); //最后是二进制转16制字符
串，大小写不敏感。
```

结果为：

```
FAFD82A3D602B37FB0FA8B7892F24A477F851A14
```

说明：

上面3个参数分别是mqtt Connect登录报文的mqttClientId、mqttUsername和mqttPasswrod。

- 使用已有Demo

1. 在域名直连的情况下，才支持TCP的MQTT连接，将make.settings中FEATURE_MQTT_DIRECT、FEATURE_MQTT_DIRECT和FEATURE_MQTT_DIRECT_NOTLS的值设置修改成**y**。

```
FEATURE_MQTT_DIRECT = y
FEATURE_MQTT_DIRECT = y
FEATURE_MQTT_DIRECT_NOTLS = y
```

2. 在SDK中，直接调用IOT_MQTT_Construct完成与云端建立连接的过程。

```
pclient = IOT_MQTT_Construct(&mqtt_params);
if (NULL == pclient) {
EXAMPLE_TRACE("MQTT construct failed");
rc = -1;
goto do_exit;
```

```
}
```

函数声明：

```
/**
* @brief Construct the MQTT client
* This function initialize the data structures, establish MQTT
connection.
*
* @param [in] pInitParams: specify the MQTT client parameter.
*
* @retval NULL : Construct failed.
* @retval NOT_NULL : The handle of MQTT client.
* @see None.
*/
void *IOT_MQTT_Construct(iotx_mqtt_param_t *pInitParams);
```

使用HTTPS认证再连接模式

1. 设备认证

   设备认证使用https，认证域名为*https://iot-auth.cn-shanghai.aliyuncs.com/auth/devicename*。

   - 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，hmacmd5或hmacsha1，默认为hmacmd5。
clientId	必选	表示客户端id，64字符内。
timestamp	可选	时间戳，不做窗口校验。
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。
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请求举例：

```
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
&resouces=mqtt&deviceName=test
sign = hmac_md5(deviceSecret, clientId123deviceNametestprodu
ctKey123timestamp123)
```

- 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

   a. 下载物联网平台 *root.crt*，建议使用TLS1.2。

   b. 设备端连接阿里云MQTT服务器地址，认证返回的MQTT地址和端口，进行通信。

c. 采用TLS建立连接，客户端验证服务器通过CA证书完成，服务器验证客户端通过MQTT协议体内connect报文中的username=iotId，password=iotToken，clientId=自定义设备标记（建议MAC或SN）。

如果iotId或iotToken非法，mqtt connect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: unknow 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。

d. 如果您使用已有Demo，请将make.settings 中FEATURE_MQTT_DIRECT的值设置修改成n，再调用IOT_MQTT_Construct接口，完成认证再连接的整个过程。

```
FEATURE_MQTT_DIRECT = n
```

HTTPS认证的过程具体实现在：\src\system\iotkit-system\src\guider.c 中 iotx_guider_authenticate。

SDK API介绍

- IOT_MQTT_Construct 与云端建立MQTT连接

mqtt-example 程序发送一次消息后会自动退出，可以尝试以下任意一种方式实现长期在线。

- 执行mqtt-example时，使用命令行`./mqtt-example loop`，设备会保持长期在线。
- 修改demo代码，example的代码调用IOT_MQTT_Destroy，设备变成离线状态。如果希望设备一直处于上线状态，请去掉IOT_MQTT_Unregister 和IOT_MQTT_Destroy的部分，使用while 保持长连接状态。

代码如下：

```
while(1)
{
IOT_MQTT_Yield(pclient, 200);
HAL_SleepMs(100);
```

```
}
```

返回结果如下：

```
/**
* @brief Construct the MQTT client
* This function initialize the data structures, establish MQTT
connection.
*
* @param [in] pInitParams: specify the MQTT client parameter.
*
* @retval NULL : Construct failed.
* @retval NOT_NULL : The handle of MQTT client.
* @see None.
*/
void *IOT_MQTT_Construct(iotx_mqtt_param_t *pInitParams);
```

- IOT_MQTT_Subscribe 同云端订阅某个topic

  请保留topic_filter的memory，callback topic_handle_func才能准确送达。

  代码如下：

```
/* Subscribe the specific topic */
rc = IOT_MQTT_Subscribe(pclient, TOPIC_DATA, IOTX_MQTT_QOS1,
_demo_message_arrive, NULL);
if (rc < 0) {
IOT_MQTT_Destroy(&pclient);
EXAMPLE_TRACE("IOT_MQTT_Subscribe() failed, rc = %d", rc);
rc = -1;
goto do_exit;
}
```

  返回结果如下：

```
/**
* @brief Subscribe MQTT topic.
*
* @param [in] handle: specify the MQTT client.
* @param [in] topic_filter: specify the topic filter.
* @param [in] qos: specify the MQTT Requested QoS.
* @param [in] topic_handle_func: specify the topic handle callback-
function.
* @param [in] pcontext: specify context. When call 'topic_hand
le_func', it will be passed back.
*
* @retval -1 : Subscribe failed.
* @retval >=0 : Subscribe successful.
The value is a unique ID of this request.
The ID will be passed back when callback 'iotx_mqtt_param_t:
handle_event'.
* @see None.
*/
int IOT_MQTT_Subscribe(void *handle,
const char *topic_filter,
```

```
iotx_mqtt_qos_t qos,
iotx_mqtt_event_handle_func_fpt topic_handle_func,
void *pcontext);
```

- IOT_MQTT_Publish 发布信息到云端

  代码如下：

```
/* Initialize topic information */
memset(&topic_msg, 0x0, sizeof(iotx_mqtt_topic_info_t));
strcpy(msg_pub, "message: hello! start!");
topic_msg.qos = IOTX_MQTT_QOS1;
topic_msg.retain = 0;
topic_msg.dup = 0;
topic_msg.payload = (void *)msg_pub;
topic_msg.payload_len = strlen(msg_pub);
rc = IOT_MQTT_Publish(pclient, TOPIC_DATA, &topic_msg);
EXAMPLE_TRACE("rc = IOT_MQTT_Publish() = %d", rc);
```

  返回结果如下：

```
/**
* @brief Publish message to specific topic.
*
* @param [in] handle: specify the MQTT client.
* @param [in] topic_name: specify the topic name.
* @param [in] topic_msg: specify the topic message.
*
* @retval -1 : Publish failed.
* @retval 0 : Publish successful, where QoS is 0.
* @retval >0 : Publish successful, where QoS is >= 0.
The value is a unique ID of this request.
The ID will be passed back when callback 'iotx_mqtt_param_t:
handle_event'.
* @see None.
*/
int IOT_MQTT_Publish(void *handle, const char *topic_name,
iotx_mqtt_topic_info_pt topic_msg);
```

- IOT_MQTT_Unsubscribe 取消云端订阅

  代码如下：

```
IOT_MQTT_Unsubscribe(pclient, TOPIC_DATA);
```

  返回结果如下：

```
/**
* @brief Unsubscribe MQTT topic.
*
* @param [in] handle: specify the MQTT client.
* @param [in] topic_filter: specify the topic filter.
*
* @retval -1 : Unsubscribe failed.
* @retval >=0 : Unsubscribe successful.
```

```
The value is a unique ID of this request.
The ID will be passed back when callback 'iotx_mqtt_param_t:
handle_event'.
* @see None.
*/
int IOT_MQTT_Unsubscribe(void *handle, const char *topic_filter);
```

- IOT_MQTT_Yield 数据接收函数

  请在任何需要接收数据的地方调用这个API。如果系统允许，请起一个单独的线程，执行该接口。

  代码如下：

```
/* handle the MQTT packet received from TCP or SSL connection */
IOT_MQTT_Yield(pclient, 200);
```

  返回结果如下：

```
/**
* @brief Handle MQTT packet from remote server and process timeout
request
* which include the MQTT subscribe, unsubscribe, publish(QOS >= 1),
reconnect, etc..
*
* @param [in] handle: specify the MQTT client.
* @param [in] timeout_ms: specify the timeout in millisecond in this
 loop.
*
* @return status.
* @see None.
*/
int IOT_MQTT_Yield(void *handle, int timeout_ms);
```

- IOT_MQTT_Destroy 销毁 MQTT连接，释放内存

  代码如下：

```
IOT_MQTT_Destroy(&pclient);
```

  返回结果如下：

```
/**
* @brief Deconstruct the MQTT client
* This function disconnect MQTT connection and release the related
resource.
*
* @param [in] phandle: pointer of handle, specify the MQTT client.
*
* @retval 0 : Deconstruct success.
* @retval -1 : Deconstruct failed.
* @see None.
*/
```

```
int IOT_MQTT_Destroy(void **phandle);
```

- IOT_MQTT_CheckStateNormal 查看当前的连接状态

  该接口用于查询MQTT的连接状态。但是，该接口并不能立刻检测到设备断网，只会在有数据发送或是keepalive时才能侦测到disconnect。

  返回结果如下：

```
/**
* @brief check whether MQTT connection is established or not.
*
* @param [in] handle: specify the MQTT client.
*
* @retval true : MQTT in normal state.
* @retval false : MQTT in abnormal state.
* @see None.
*/
int IOT_MQTT_CheckStateNormal(void *handle);
```

MQTT保活

设备端在keepalive_interval_ms时间间隔内，至少需要发送一次报文，包括ping请求。

如果服务端在keepalive_interval_ms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。

在IOT_MQTT_Construct函数可以设置keepalive_interval_ms的取值，物联网平台通过该取值作为心跳间隔时间。keepalive_interval_ms的取值范围是60000~300000。

4.2.3 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：${productKey}.iot-as-mqtt.cn-shanghai.aliyuncs.com:443

     ${productKey}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","clientId12345deviceNamedevicep
roductKeypktimestamp789").toHexString();
```

4.2.4 CoAP协议规范

协议版本

支持 RFC 7252 Constrained Application Protocol协议，具体请参考：*RFC 7252*。

通道安全

使用 DTLS v1.2保证通道安全，具体请参考：*DTLS v1.2*。

开源客户端参考

http://coap.technology/impls.html。

> 说明：
> 若使用第三方代码, 阿里云不提供技术支持

阿里CoAP约定

- 不支持？号形式传参数。
- 暂时不支持资源发现。

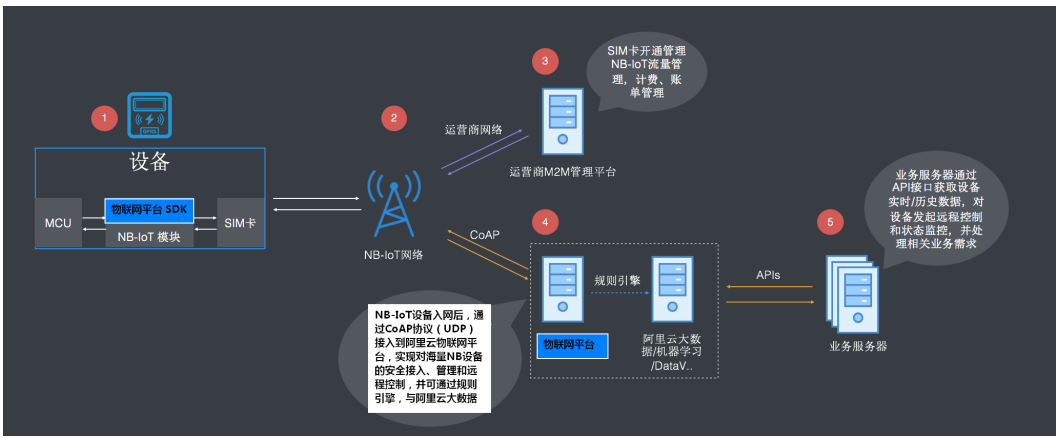
- 仅支持UDP协议，并且目前必须通过DTLS。
- URI规范，CoAP的URI资源和MQTT TOPIC保持一致，参考*MQTT规范*。

4.2.5 CoAP连接通信

流程说明

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，基于CoAP协议将NB-IoT设备接入物联网平台的流程如图 *4-1: CoAP连接*所示。

图 **4-1: CoAP连接**



流程说明如下：

1. 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/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，并已具备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，并可通过规则引擎，与阿里云的各类大数据产品、云数据库和报表系统打通，快速实现从连接到智能的跨越。
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SDK接入

提供了C版本SDK示例，可以参考*DEMO*。

CoAP协议自主接入流程

使用CoAP自主接入流程：

1. CoAP服务器地址`endpoint = ${productKey}.iot-as-coap.cn-shanghai.aliyuncs.com:5684`，其中productKey请替换为您申请的产品Key。
2. 目前只支持DTLS，必须走安全通道，下载根证书。
3. 设备在发送数据前，首先发起认证，获取设备的token。
4. 每次上报数据时，需要携带token信息，如果token失效，需要重新认证获取token，token可以缓存本地，有效期48小时。

CoAP协议自主接入操作

1. 设备认证，此接口用于传输数据前获取token，只需要请求一次。

```
POST /auth
Host: ${productKey}.iot-as-coap.cn-shanghai.aliyuncs.com
Port: 5684
Accept: application/json or application/cbor
Content-Format: application/json or application/cbor
payload: {"productKey":"ZG1EvTEa7NN","deviceName":"NlwaSPXsCp
TQuh8FxBGH","clientId":"mylight1000002","sign":"bccb3d2618afe74b3eab
12b94042f87b"}
```

参数说明如下：

- Method: POST，只支持POST方法。
- URL: /auth，URL地址。
- Accept：接收的数据编码方式，目前支持 application/json和application/cbor。
- Content-Format： 上行数据的编码格式，目前支持 application/json和application/cbor。

payload内容，JSON数据格式，具体属性值如下：

表 **4-4: payload**参数说明

字段名称	是否必选	说明
productKey	必选	productKey，从物联网平台的控制台获取。
deviceName	必选	deviceName，从物联网控制台获取。
sign	必选	签名，格式为hmacmd5(deviceSecret,content)， content为将所有提交给服务器的参数（version、sign,resources

字段名称	是否必选	说明
		和signmethod除外），按照字母顺序排序，将参数值依次拼接，无拼接符号。
signmethod	可选	算法类型，hmacmd5或hmacsha1。
clientId	必选	表示客户端Id，64字符内。
timestamp	可选	时间戳，目前时间戳并不做窗口校验。

返回如下：

```
 response：{"token":"eyJ0b2tlbiI6IjBkNGUyNjkyZTNjZDQxOGU5MTA4
Njg4ZDdhNWI3MjUxIiwiZXhwIjoxNDk4OTg1MTk1fQ.DeQLSwVX8iBjdazjzNHG
5zcRECWcL49UoQfq1lXrJvI"}
```

返回码说明如下：

表 **4-5:** 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	返回错误信息	请求发送的Payload非法。
4.04	Not Found	404 Not found	请求的路径不存在。
4.05	Method Not Allowed	支持的方法	请求方法不是指定值。
4.06	Not Acceptable	要求的Accept	Accept不是指定的类型。
4.15	Unsupported Content-Format	支持的content信息	请求的content不是指定类型。
5.00	Internal Server Error	返回错误信息	auth服务器超时或错误。

SDK中使用IOT_CoAP_Init 和 IOT_CoAP_DeviceNameAuth与云端建立CoAP认证。

示例代码：

```
iotx_coap_context_t *p_ctx = NULL;
p_ctx = IOT_CoAP_Init(&config);
if (NULL != p_ctx) {
  IOT_CoAP_DeviceNameAuth(p_ctx);
```

```
  do {
    count ++;
    if (count == 11) {
      count = 1;
    }
    IOT_CoAP_Yield(p_ctx);
  } while (m_coap_client_running);
  IOT_CoAP_Deinit(&p_ctx);
} else {
  HAL_Printf("IoTx CoAP init failed\r\n");
}
```

函数声明：

```
/**
* @brief Initialize the CoAP client.
* This function initialize the data structures and network,
* and create the DTLS session.
*
* @param [in] p_config: Specify the CoAP client parameter.
*
* @retval NULL : Initialize failed.
* @retval NOT_NULL : The contex of CoAP client.
* @see None.
*/
iotx_coap_context_t *IOT_CoAP_Init(iotx_coap_config_t *p_config);

/**
* @brief Handle device name authentication with remote server.
*
* @param [in] p_context: Pointer of contex, specify the CoAP client.
*
* @retval IOTX_SUCCESS : Authenticate success.
* @retval IOTX_ERR_SEND_MSG_FAILED : Send authentication message
failed.
* @retval IOTX_ERR_AUTH_FAILED : Authenticate failed or timeout.
* @see iotx_ret_code_t.
*/
int IOT_CoAP_DeviceNameAuth(iotx_coap_context_t *p_context);
```

2. 上行数据(${endpoint}/topic/${topic})

发送数据到某个topic，${topic}可以在物联网平台的控制台，选择产品管理 > 消息通信进行设置。

对于topic/productkey/${deviceName}/pub，如果当前设备名称为device，那么您可以调用 ${productKey}.iot-as-coap.cn-shanghai.aliyuncs.com:5684/topic/productkey/device/pub 地址来上报数据，目前只支持pub权限的topic用于数据上报。

示例：

```
POST /topic/${topic}
Host: ${productKey}.iot-as-coap.cn-shanghai.aliyuncs.com
Port: 5683
```

```
Accept: application/json or application/cbor
Content-Format: application/json or application/cbor
payload: ${your_data}
CustomOptions: number:61(标识token)
```

参数说明如下：

- Method： POST，支持POST方法。
- URL： /topic/${topic}，${topic}替换为当前设备对应的topic。
- Accept：接收的数据编码方式，目前只支持 application/json和application/cbor。
- Content-Format： 上行数据的编码格式，服务端不做校验。
- CustomOptions： 设备认证（Auth）获取到token值，Option Number使用61。

3. SDK使用接口IOT_CoAP_SendMessage发送数据，使用IOT_CoAP_GetMessagePayload和IOT_CoAP_GetMessageCode接收数据。

示例代码：

```
/* send data */
static void iotx_post_data_to_server(void *param)
{
  char path[IOTX_URI_MAX_LEN + 1] = {0};
  iotx_message_t message;
  iotx_deviceinfo_t devinfo;
  message.p_payload = (unsigned char *)"{\"name\":\"hello world\"}";
  message.payload_len = strlen("{\"name\":\"hello world\"}");
  message.resp_callback = iotx_response_handler;
  message.msg_type = IOTX_MESSAGE_CON;
  message.content_type = IOTX_CONTENT_TYPE_JSON;
  iotx_coap_context_t *p_ctx = (iotx_coap_context_t *)param;
  iotx_set_devinfo(&devinfo);
  snprintf(path, IOTX_URI_MAX_LEN, "/topic/%s/%s/update/", (char *)
devinfo.product_key,
  (char *)devinfo.device_name);
  IOT_CoAP_SendMessage(p_ctx, path, &message);
}

/* receive data */
static void iotx_response_handler(void *arg, void *p_response)
{
  int len = 0;
  unsigned char *p_payload = NULL;
  iotx_coap_resp_code_t resp_code;
  IOT_CoAP_GetMessageCode(p_response, &resp_code);
  IOT_CoAP_GetMessagePayload(p_response, &p_payload, &len);
  HAL_Printf("[APPL]: Message response code: 0x%x\r\n", resp_code);
  HAL_Printf("[APPL]: Len: %d, Payload: %s, \r\n", len, p_payload);
}
```

```
/**
* @brief Send a message with specific path to server.
* Client must authentication with server before send message.
```

```
*
* @param [in] p_context : Pointer of contex, specify the CoAP client
.
* @param [in] p_path: Specify the path name.
* @param [in] p_message: Message to be sent.
*
* @retval IOTX_SUCCESS : Send the message success.
* @retval IOTX_ERR_MSG_TOO_LOOG : The message length is too long.
* @retval IOTX_ERR_NOT_AUTHED : The client hasn't authenticated with
 server
* @see iotx_ret_code_t.
*/
int IOT_CoAP_SendMessage(iotx_coap_context_t *p_context, char *
p_path, iotx_message_t *p_message);

/**
* @brief Retrieves the length and payload pointer of specified
message.
*
* @param [in] p_message: Pointer to the message to get the payload.
Should not be NULL.
* @param [out] pp_payload: Pointer to the payload.
* @param [out] p_len: Size of the payload.
*
* @retval IOTX_SUCCESS : Get the payload success.
* @retval IOTX_ERR_INVALID_PARAM : Can't get the payload due to
invalid parameter.
* @see iotx_ret_code_t.
**/
int IOT_CoAP_GetMessagePayload(void *p_message, unsigned char **
pp_payload, int *p_len);

/**
* @brief Get the response code from a CoAP message.
*
* @param [in] p_message: Pointer to the message to add the address
information to.
* Should not be NULL.
* @param [out] p_resp_code: The response code.
*
* @retval IOTX_SUCCESS : When get the response code to message
success.
* @retval IOTX_ERR_INVALID_PARAM : Pointer to the message is NULL.
* @see iotx_ret_code_t.
**/
int IOT_CoAP_GetMessageCode(void *p_message, iotx_coap_resp_code_t *
p_resp_code);
```

限制条件及注意事项

- topic规范和MQTT topic一致，CoAP协议内 `coap://host:port/topic/${topic}`接口对于所有${topic}和MQTT topic是可以复用的，不支持`?query_String=xxx`形式传参。
- 客户端缓存认证返回的token是请求的令牌，通过DTLS传输。
- 传输的数据大小依赖于MTU的大小，最好在1K以内。

C版本SDK其他接口介绍

- IOT_CoAP_Yield 接口接收数据。

  请在任何需要接收数据的地方调用这个API，如果系统允许，请起一个单独的线程，执行该接口。

```
/**
* @brief Handle CoAP response packet from remote server,
* and process timeout request etc..
*
* @param [in] p_context : Pointer of contex, specify the CoAP client
.
*
* @return status.
* @see iotx_ret_code_t.
*/
int IOT_CoAP_Yield(iotx_coap_context_t *p_context);
```

- IOT_CoAP_Deinit 接口释放内存。

```
/**
* @brief De-initialize the CoAP client.
* This function release CoAP DTLS session.
* and release the related resource. *
* @param [in] p_context: Pointer of contex, specify the CoAP client.
*
* @return None.
* @see None.
*/
void IOT_CoAP_Deinit(iotx_coap_context_t **p_context);
```

C版本SDK具体使用，请参考*\sample\coap\coap-example.c*的实现。

4.2.6 HTTP协议规范

HTTP协议版本

- 支持 Hypertext Transfer Protocol — HTTP/1.0 协议，具体请参考：*RFC 1945*
- 支持 Hypertext Transfer Protocol — HTTP/1.1 协议，具体请参考：*RFC 2616*

通道安全

使用HTTPS(Hypertext Transfer Protocol Secure协议)保证通道安全。

- 不支持？号形式传参数。
- 暂时不支持资源发现。
- 仅支持HTTPS。

- URI规范, HTTP的URI资源和MQTT TOPIC保持一致，参考*MQTT规范*。

4.2.7 HTTP连接通信

接入说明

设备使用HTTP接入说明：

- HTTP服务器地址endpoint = *https://iot-as-http.cn-shanghai.aliyuncs.com*。
- 只支持HTTPS协议。
- 设备在发送数据前，首先发起认证，获取设备的token。
- 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，token可以到缓存本地，有效期48小时。

设备认证(${endpoint}/auth)

此接口用于传输数据前获取token，只需要请求一次：

```
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"}
```

参数说明如下：

- Method: POST，支持POST方法。
- URL: /auth，URL地址，只支持HTTPS。
- Content-Type：目前只支持 application/json。

JSON数据格式，具体属性值如下：

表 4-6: 请求参数

字段名称	是否必选	说明
productKey	必选	从物联网平台的控制台获取。
deviceName	必选	从物联网平台的控制台获取。
clientId	必选	客户端自表示Id,64字符内，建议是MAC或SN。

字段名称	是否必选	说明
timestamp	可选	时间戳，校验时间戳15分钟内请求有效。
sign	必选	签名，格式为hmacmd5(deviceSecret,content), content = 将所有提交给服务器的参数（version、sign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，hmacmd5或hmacsha1，不填默认hmacmd5。
version	可选	版本，不填默认default。

返回如下：

```
body:
{
  "code": 0,//业务状态码
  "message": "success",//业务信息
  "info": {
    "token":  "eyJ0eXBlIjoiSldUIiwiYWxnIjoiaG1hY3NoYTEifQ.eyJleHBpcmUiOjE1MDI1MzE1MDc0NzcsInRva2VuIjoiODA0ZmFjYTBiZTE3NGUxNjliZjY0ODVlNWNiNDg3MTkifQ.OjMwu29F0CY2YR_6oOyiOLXz0c8"
  }
}
```

业务状态码说明如下：

表 4-7: 状态码

code	message	备注
10000	common error	未知错误。
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。

SDK使用IOT_HTTP_Init和IOT_HTTP_DeviceNameAuth来进行认证。

```
handle = IOT_HTTP_Init(&http_param);
if (NULL != handle) {
  IOT_HTTP_DeviceNameAuth(handle);
  HAL_Printf("IoTx HTTP Message Sent\r\n");
} else {
  HAL_Printf("IoTx HTTP init failed\r\n");
  return 0;
}
```

函数声明：

```
/**
* @brief Initialize the HTTP client
* This function initialize the data.
*
* @param [in] pInitParams: Specify the init param infomation.
*
* @retval NULL : Initialize failed.
* @retval NOT_NULL : The contex of HTTP client.
* @see None.
*/
void *IOT_HTTP_Init(iotx_http_param_t *pInitParams);
/**
* @brief Handle device name authentication with remote server.
*
* @param [in] handle: Pointer of context, specify the HTTP client.
*
* @retval 0 : Authenticate success.
* @retval -1 : Authenticate failed.
* @see iotx_err_t.
*/
int IOT_HTTP_DeviceNameAuth(void *handle);
```

上行数据(${endpoint}/topic/${topic})

发送数据到某个topic，${topic}可以在物联网平台的控制台，选择**产品管理 > 消息通信**进行设置，比如对于${topic} `/productkey/${deviceName}/pub`，如果设备名称为device123，该设备可以通过 *https://iot-as-http.cn-shanghai.aliyuncs.com/topic/productkey/device123/pub* 地址来上报数据。

- 示例：

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

参数说明：

- Method： POST，支持POST方法。
- URL： /topic/${topic}，${topic}替换为设备对应的topic，只支持HTTPS。
- Content-Type：目前只支持 application/octet-stream。
- password： 放在Header中的参数，${token}为设备认证auth接口返回的token。

- body内容：发往${topic}的内容，二进制byte[]，utf-8编码。
- 返回值：

```
body:
{
  "code": 0,//业务状态码
  "message": "success",//业务信息
  "info": {
    "messageId": 892687627916247040,
    "data": byte[]//可能为空，utf-8编码
  }
}
```

业务状态码说明：

表 **4-8:** 状态码说明

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，需要重新调用auth，进行，鉴权，获取token。
20002	token is null	请求header无token信息。
20003	check token error	根据token获取identify信息失败，需要重新调用auth，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。

C版本SDK

SDK使用接口IOT_HTTP_SendMessage来发送和接收数据。

```
static int iotx_post_data_to_server(void *handle)
{
  int ret = -1;
```

```
  char path[IOTX_URI_MAX_LEN + 1] = {0};
  char rsp_buf[1024];
  iotx_http_t *iotx_http_context = (iotx_http_t *)handle;
  iotx_device_info_t *p_devinfo = iotx_http_context->p_devinfo;
  iotx_http_message_param_t msg_param;
  msg_param.request_payload = (char *)"{\"name\":\"hello world\"}";
  msg_param.response_payload = rsp_buf;
  msg_param.timeout_ms = iotx_http_context->timeout_ms;
  msg_param.request_payload_len = strlen(msg_param.request_payload) +
1;
  msg_param.response_payload_len = 1024;
  msg_param.topic_path = path;
  HAL_Snprintf(msg_param.topic_path, IOTX_URI_MAX_LEN, "/topic/%s/%s/
update",
  p_devinfo->product_key, p_devinfo->device_name);
  if (0 == (ret = IOT_HTTP_SendMessage(iotx_http_context, &msg_param
))) {
    HAL_Printf("message response is %s\r\n", msg_param.response_p
ayload);
  } else {
    HAL_Printf("error\r\n");
  }
  return ret;
}
```

```
/**
* @brief Send a message with specific path to server.
* Client must authentication with server before send message.
*
* @param [in] handle: Pointer of contex, specify the HTTP client.
* @param [in] msg_param: Specify the topic path and http payload
configuration.
*
* @retval 0 : Success.
* @retval -1 : Failed.
* @see iotx_err_t.
*/
int IOT_HTTP_SendMessage(void *handle, iotx_http_message_param_t *
msg_param);
```

C版本SDK其他接口

IOT_HTTP_Disconnect关闭http连接，释放内存。

```
/**
* @brief close tcp connection from client to server.
*
* @param [in] handle: Pointer of contex, specify the HTTP client.
* @return None.
* @see None.
*/
```

```
void IOT_HTTP_Disconnect(void *handle);
```

限制条件及注意事项

- TOPIC规范和MQTT的TOPIC一致，HTTP协议内 *https://iot-as-http.cn-shanghai.aliyuncs.com/topic/${topic}* ，该接口对于所有${topic}和MQTTtopic是可以复用的，不支持`?query_String=xxx`形式传参。
- 客户端缓存认证返回的Token，待Token失效进行重新认证时，刷新缓存。
- 上行接口传输的数据大小限制为128k。

4.3 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 4-9: 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

对比项	一机一密	一型一密	子设备注册
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

4.3.1 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 4-10: 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

对比项	一机一密	一型一密	子设备注册
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

4.3.2 一机一密

本章节介绍一机一密认证方法，与该方法的使用步骤。

什么是一机一密

物联网平台默认采用一机一密，即设备需要预先烧录唯一的设备证书，设备与云端建连时携带ProductKey、DeviceName和DeviceSecret进行接入认证，认证通过后云端完成设备激活才可传输数据。

> **说明：**
> 此种认证方式安全性较高，推荐使用。

使用流程图

图 4-2: 一机一密

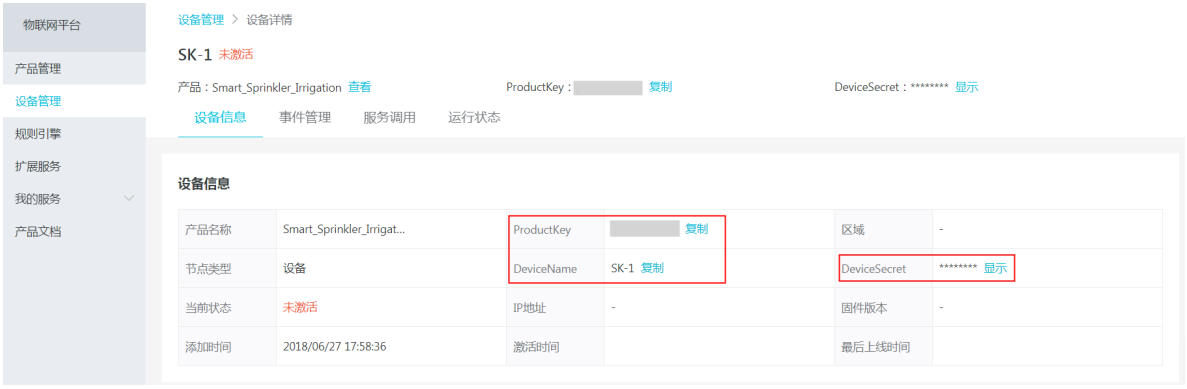
流程说明如下：

1. 参考创建产品与设备章节，创建产品和设备，获得认证信息，包括ProductKey、DeviceName和DeviceSecret。
2. 产线上为每个设备烧录ProductKey、DeviceName和DeviceSecret。
3. 设备上电联网，向云端发起认证激活。
4. 云端认证设备，通过后与设备建立连接，设备发布/订阅相关Topic，开始上下行数据通信。

使用步骤

1. 以阿里云账号登录*IoT*控制台。
2. 参考创建产品与设备章节，创建产品与设备。
3. 选择设备管理，单击设备后的查看，获取三元组信息，如下图所示：

图 4-3: 三元组信息



4. 下载设备端SDK，选择连接协议，开发设备端SDK。
5. 在设备端SDK中填入获取到的三元组信息。
6. 根据实际需求完成其他功能，如OTA开发、子设备接入、设备模型开发、设备影子开发等。
7. 在产线上将设备SDK（已写入三元组信息）烧录至设备中。

4.3.3 一型一密

本章节介绍一机一密的认证方法，与该方法的使用步骤。

什么是一型一密

选择一型一密，可以简化烧录流程，为同一产品烧录相同固件（固件中写入产品证书，即ProductKey和ProductSecret）。固件烧录后，设备激活时将从云端动态获取DeviceSecret。

说明：

- 设备端仅C-SDK支持一型一密功能。
- 使用前，您需确认设备支持一型一密的认证方式。
- 采用一型一密方式认证时，设备烧录相同固件，存在产品证书泄露风险。此时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。

使用流程图

图 4-4: 一型一密

流程说明如下：

1. 参考创建产品与设备章节，创建产品，获得系统颁发的产品证书。
2. 在产品详情页面，开启动态注册开关。系统将开启短信验证，确保您是本人操作。

   说明：
   若系统校验时，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. 在设备管理页面，为该产品添加设备。您可以选择将设备的MAC地址、IMEI或SN号等ID信息作为DeviceName预注册到平台，此时的设备处于未激活状态。

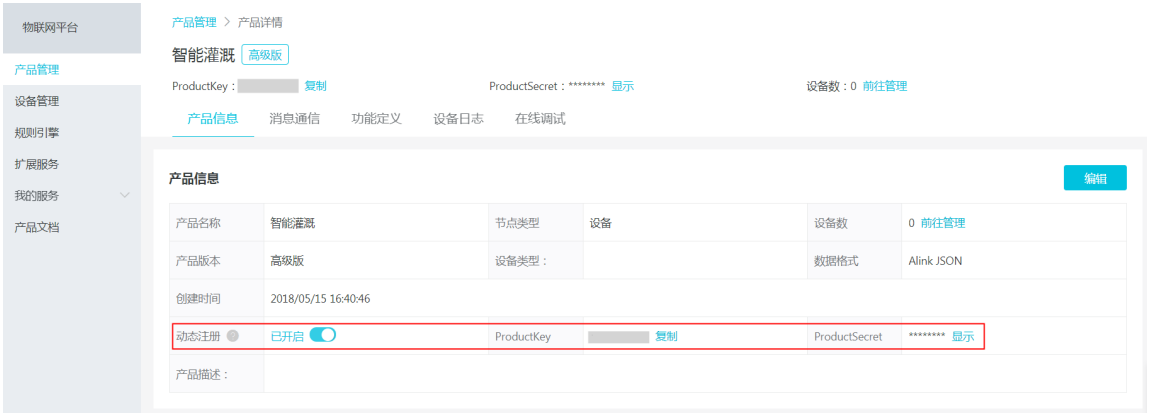
   说明：
   因设备激活时会校验DeviceName，建议您采用可以直接从设备中读取到的ID作为DeviceName使用。
4. 产线为该产品下的所有设备烧录相同的产品证书，设备端SDK（C-SDK）设置`FEATURE_SUPPORT_PRODUCT_SECRET = y`开启一型一密功能。
5. 设备上电联网，首先通过一型一密发起认证请求，云端校验通过后动态下发该设备对应的DeviceSecret。至此，设备获得了连接云端所需的三元组信息，包括ProductKey、DeviceName和DeviceSecret。

   说明：
   该认证方式下，仅在设备首次激活时动态下发DeviceSecret。若需重新激活设备，请在云端删除设备后，重新添加。
6. 设备使用三元组与云端建立连接，发布/订阅相关Topic，开始上下行数据通信。

使用步骤

1. 以阿里云账号登录*IoT*控制台。
2. 参考创建产品与设备章节，创建产品。
3. 获取系统颁发的产品证书，即ProductKey和ProductSecret。
   a. 单击产品后的查看，进入产品信息页面。
   b. 获取产品证书信息，如下图所示：

图 4-5: 产品信息



c. 将动态注册开启，填写短信验证码，确认是您本人操作。

4. 在设备管理，创建设备。以MAC地址、IMEI或SN序列号等可以从设备中直接读取到的信息作为DeviceName。
5. 选择连接协议，下载设备端C-SDK，开发设备端SDK。
6. 在设备端SDK中填入获取到的产品证书（ProductKey和ProductSecret）。
7. 在设备端SDK中，设置`FEATURE_SUPPORT_PRODUCT_SECRET = y`开启一型一密功能。
8. 具体实现可以参考`/src/cmp/iotx_cmp_api.c`中`IOT_CMP_Init`，代码示例如下：

```
#ifdef SUPPORT_PRODUCT_SECRET
        /* 一型一密 */
        if (IOTX_CMP_DEVICE_SECRET_PRODUCT == pparam->secret_type
 && 0 >= HAL_GetDeviceSecret(device_secret)) {
            HAL_GetProductSecret(product_secret);
            if (strlen(product_secret) == 0) {
                CMP_ERR(cmp_log_error_secret_1);
                return FAIL_RETURN;
            }
            /* auth */
            if (FAIL_RETURN == iotx_cmp_auth(product_key, device_nam
e, device_id)) {
                CMP_ERR(cmp_log_error_auth);
                return FAIL_RETURN;
            }
        }
    #endif /**< SUPPORT_PRODUCT_SECRET*/
```

9. 根据实际需求完成其他功能，如OTA开发、子设备接入、设备模型开发、设备影子开发等。
10. 在产线上将设备SDK（已写入产品证书信息）烧录至设备中。

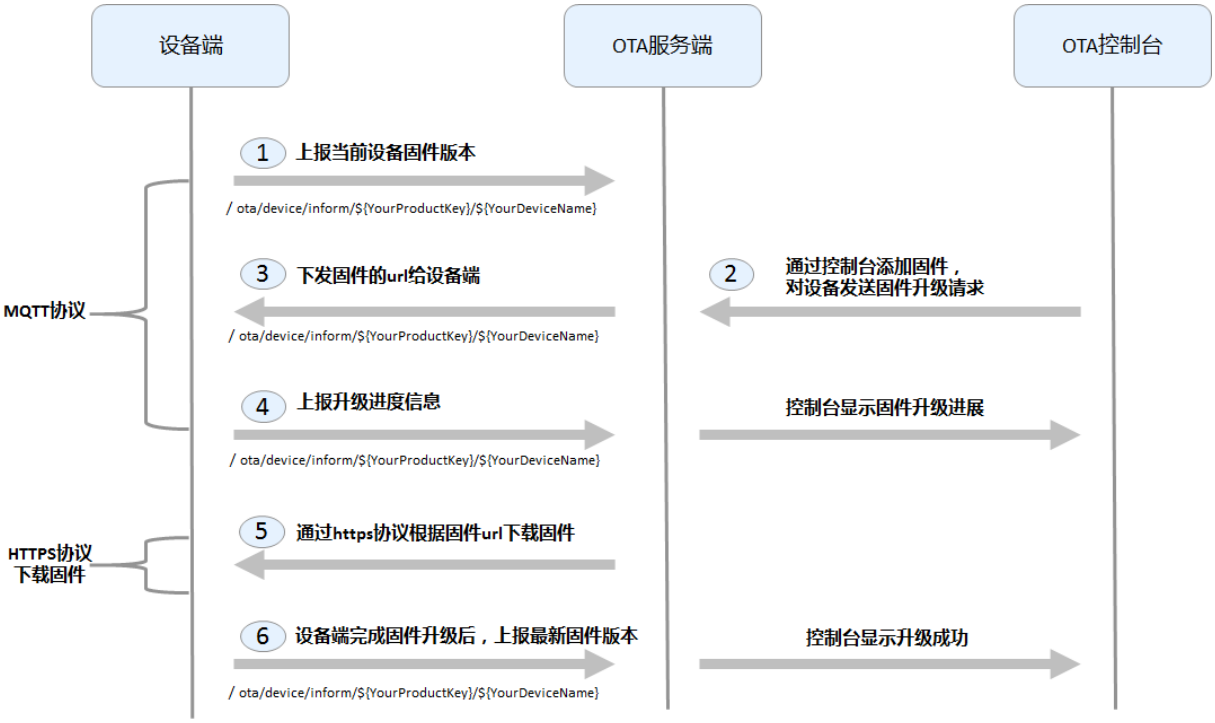
4.4 设备OTA开发

4.4.1 基础版

升级流程

以MQTT协议为例，固件升级流程如图 *4-6: 固件升级*所示。

图 **4-6:** 固件升级



固件升级topic：

- 设备端上报固件版本给云端

```
/ota/device/inform/${productKey}/${deviceName}
```

- 设备端订阅该topic接收云端固件升级通知

```
/ota/device/upgrade/${productKey}/${deviceName}
```

- 设备端上报固件升级进度

```
/ota/device/progress/${productKey}/${deviceName}
```

- 设备端请求是否固件升级

```
/ota/device/request/${productKey}/${deviceName}
```

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。

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");
}
```

4.5 子设备接入

本章节介绍如何将节点类型为设备的设备作为网关的子设备与云端建立连接通信。

物联网套件目前支持两种节点类型接入：设备和网关。

- 设备：指不能挂载子设备的设备，这种设备可以直连IoT Hub，也可以作为网关的子设备连接IoT Hub。
- 网关：指可以挂载子设备的直连设备，网关具有子设备管理模块，维持子设备的拓扑关系，并且可以将拓扑关系同步到云端。

4.5.1 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

网关已使用一机一密与云端建立连接。

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

   - 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eviceName，不能和网关的pPoductKey和dDviceName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PoductKey和DviceName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PoductKey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PoductKey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, HmacSha1 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 /${productKey}/${deviceName}/xxx，也可以是 /sys/${productKey}/${deviceName}/thing/xxx格式。
- 网关只需要PUB数据到子设备Topic即可，即Topic中的/${productKey}/${deviceName}/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":"xxxxx",//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" : "1234556554"
}],
"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。

4.5.2 错误码说明

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 **4-11:** 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200。
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。

code	message	备注
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 **4-12:** 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 **4-13:** 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 **4-14:** 发送消息失败

code	message	备注
520	device session error	子设备会话错误。 • 子设备会话不存在，可能没有connect，也可能已经被下线。 • 子设备会话在线，但是并不是通过当前网关会话上线的。

4.6 设备影子

4.6.1 设备影子介绍

设备影子是一个 JSON 文档，用于存储设备上报状态、应用程序期望状态信息。

- 每个设备有且只有一个设备影子，设备可以通过MQTT获取和设置设备影子以此来同步状态，该同步可以是影子同步给设备，也可以是设备同步给影子。
- 应用程序通过物联网平台的SDK获取和设置设备影子，获取设备最新状态或者下发期望状态给设备。

场景一

由于网络不稳定，设备频繁上下线。应用程序发出需要获取当前的设备状态请求时，设备掉线，无法获取设备状态，但下一秒设备又连接成功，应用程序无法正确发起请求。

使用设备影子机制存储设备最新状态，一旦设备状态产生变化，设备会将状态同步到设备影子。应用程序在请求设备当前状态时，只需要获取影子中的状态即可，不需要关心设备是否在线。

场景二

如果设备网络稳定，很多应用程序请求获取设备状态，设备需要根据请求响应多次，即使响应的结果是一样的，设备本身处理能力有限，无法负载被请求多次的情况。

使用设备影子机制，设备只需要主动同步状态给设备影子一次，多个应用程序请求设备影子获取设备状态，即可获取设备最新状态，做到应用程序和设备的解耦。

场景三

- 设备网络不稳定，导致设备频繁上下线，应用程序发送控制指令给设备时，设备掉线，指令无法下达到设备。

- 通过QoS=1或者2实现，但是该方法对于服务端的压力比较大，一般不建议使用。
- 使用设备影子机制，应用程序发送控制指令，指令携带时间戳保存在设备影子中。当设备掉线重连时，获取指令并根据时间戳确定是否执行。

- 设备真实掉线，指令发送失败。设备再上线时，设备影子功能通过指令加时间戳的模式，保证设备不会执行过期指令。

4.6.2 设备影子JSON详解

设备影子JSON格式

格式为：

```
{
"state": {
"desired": {
"attribute1": integer2,
"attribute2": "string2",
...
"attributeN": boolean2
},
"reported": {
"attribute1": integer1,
"attribute2": "string1",
...
"attributeN": boolean1
}
},
"metadata": {
"desired": {
"attribute1": {
"timestamp": timestamp
},
"attribute2": {
"timestamp": timestamp
},
...
"attributeN": {
"timestamp": timestamp
}
},
"reported": {
"attribute1": {
"timestamp": timestamp
},
"attribute2": {
"timestamp": timestamp
},
...
"attributeN": {
"timestamp": timestamp
}
}
},
"timestamp": timestamp,
```

```
"version": version
}
```

JSON属性描述，如表 *4-15: JSON*属性所示。

表 **4-15: JSON**属性

属性	描述
desired	设备的预期状态。 应用程序向desired部分写入数据更新事物的状态，无需直接连接到该设备。
reported	设备的报告状态。设备可以在reported部分写入数据，报告其最新状态。 应用程序可以通过读取该参数值，获取设备的状态。
metadata	当用户更新State文档，设备影子服务会自动更新metadata的值。 state 部分的元数据的信息，包括 state 部分中每个属性的时间戳，以 Epoch 时间表示，用来获取确定的更新时间。
timestamp	影子文档的最新更新时间。
version	用户主动更新version，设备影子会检查请求中的version是否大于当前版本。 如果大于当前版本，则更新设备影子，并将version更新到请求的版本中，反之拒绝更新设备影子。 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。

示例影子JSON：

```
{
"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
}
```

空白部分

- 仅当设备影子文档具有预期状态时，才包含desired部分。没有desired部分的文档同样为有效影子JSON文档：

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

- reported部分可以为空，没有reported部分的文档同样为有效影子JSON文档：

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

数组

设备影子支持数组，数组必须全量更新，不能只更新数组的某一部分。

- 初始状态：

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

- 更新：

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

- 最终状态：

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

4.6.3 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/${productKey}/${deviceName}*

  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/${productKey}/${deviceName}*

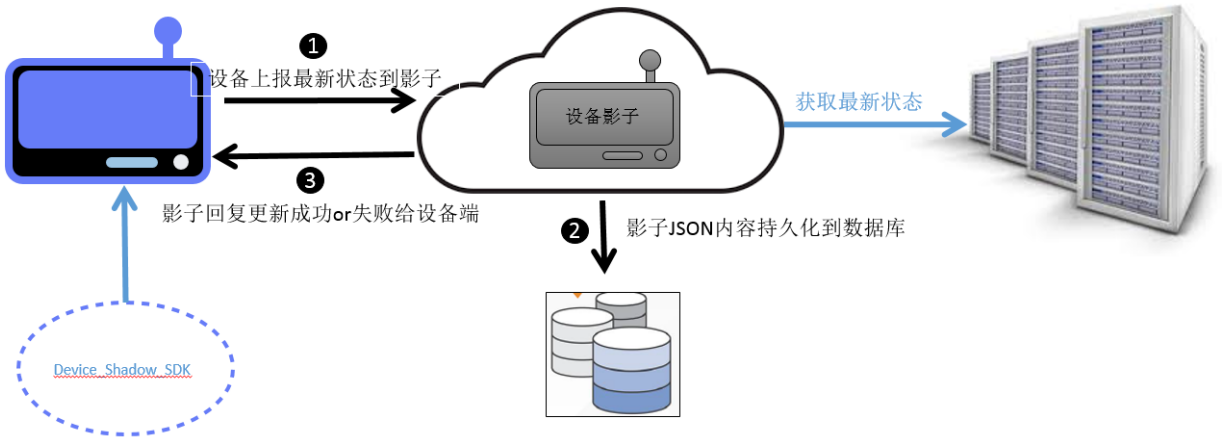
  设备影子更新状态到该Topic，设备订阅此Topic的消息。

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

设备主动上报状态

处理流程图如图 *4-7:* 设备主动上报所示。

图 4-7: 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 *4-16: 上报参数说明*所示。

   表 4-16: 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。 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），即发消息到/shadow/get/10000/lightbulb中，设备订阅该Topic。

   - 若更新成功，发送到Topic中的消息为：

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

   - 若更新失败，发送到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 *4-17:* 错误码说明所示。

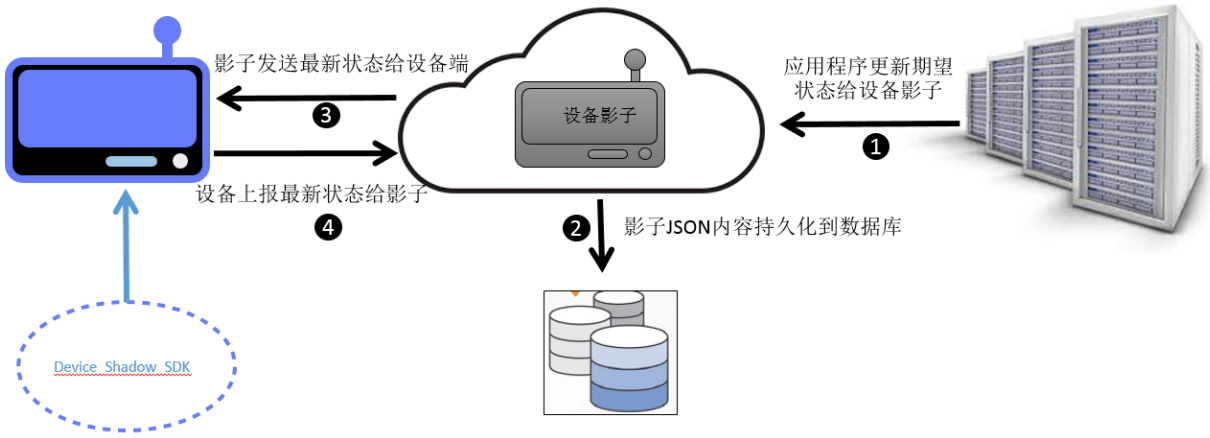
表 **4-17:** 错误码说明

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 *4-8:* 应用程序改变设备状态所示。

图 **4-8:** 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。

   如果有时间戳判断指令过期，也可以选择不更新。

5. 更新完之后，发消息到topic`/shadow/update/10000/lightbulb`中上报最新状态，消息如下：

```
{
"method": "update",
"state": {
"desired":"null"
},
"version": 3
}
```

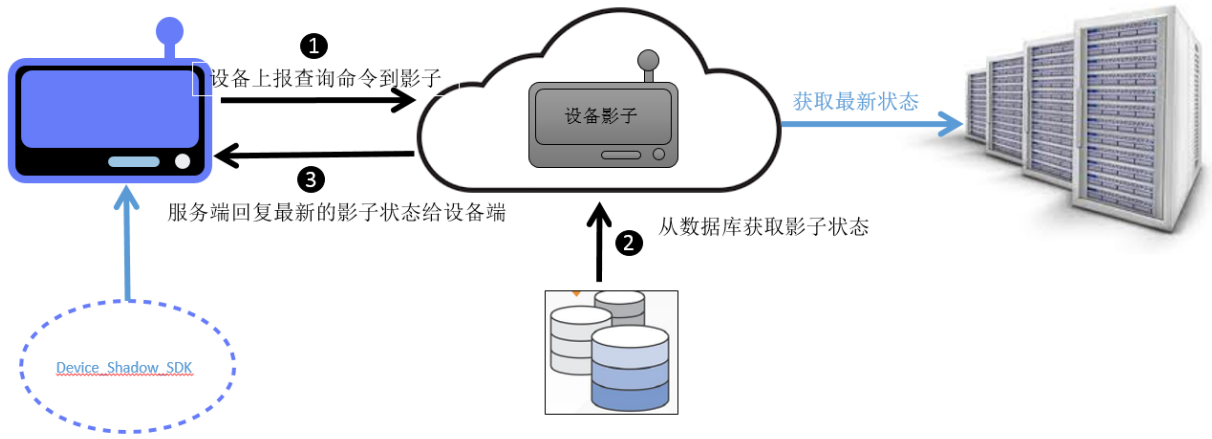
6. 上报状态成功后，设备影子会同步更新，此时的影子文档如下：

```
{
"state" : {
"reported" : {
"color" : "green"
}
},
"metadata" : {
"reported" : {
"color" : {
"timestamp" : 1469564577
}
},
"desired" : {
"timestamp" : 1469564576
}
},
"version" : 3
}
```

设备主动获取设备影子内容

处理流程图如图 *4-9: 获取设备影子内容*所示。

图 4-9: 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
}
```

3. SDK使用API IOT_Shadow_Pull 完成获取设备影子的动作。

```
IOT_Shadow_Pull(h_shadow);
```

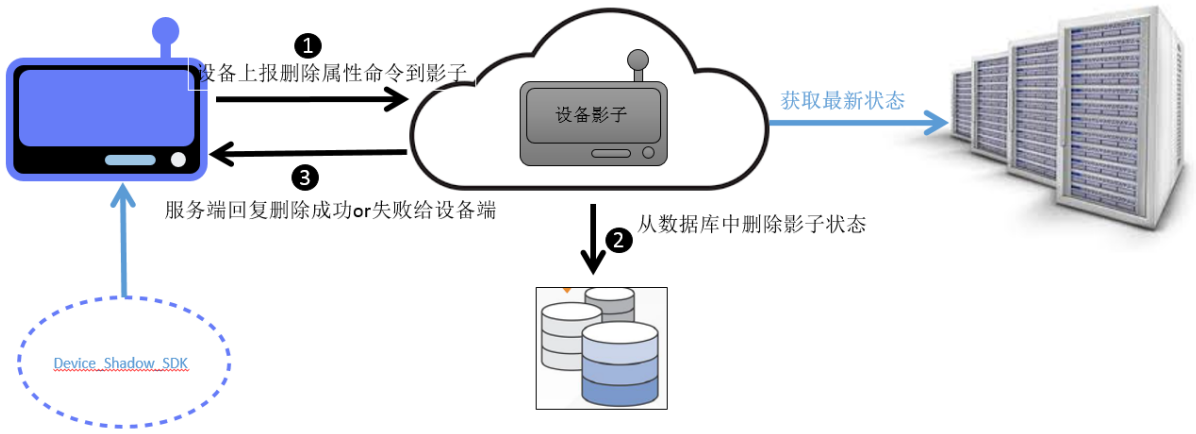
函数原型：

```
/**
* @brief Synchronize device shadow data from cloud.
* It is a synchronous interface.
* @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_Pull(void *handle);
```

设备端删除影子属性

处理流程图如图 *4-10:* 删除影子属性所示。

图 **4-10:** 删除影子属性



1. 灯泡想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，具体的消息如下：

   - 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
}
```

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

2. SDK使用IOT_Shadow_DeleteAttribute参数，取消设备影子的属性。

```
IOT_Shadow_DeleteAttribute(h_shadow, &attr_temperature);
IOT_Shadow_DeleteAttribute(h_shadow, &attr_light);
IOT_Shadow_Destroy(h_shadow);
```

函数原型：

```
/**
* @brief Delete the specific attribute.
*
* @param [in] handle: The handle of device shaodw.
* @param [in] pattr: The parameter to be deleted from server.
* @retval SUCCESS_RETURN : Success.
* @retval other : See iotx_err_t.
* @see None.
*/
iotx_err_t IOT_Shadow_DeleteAttribute(void *handle, iotx_shado
w_attr_pt pattr);
```

4.6.4 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/${product_key}/${device_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 shaodw.
* @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_shado
w_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，同步设备状态至云端。

   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 shaodw.
* @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 shaodw.
* @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_dat
a_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);
```

4.7 设备模型开发

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 *4-11: 创建设备*所示。

图 4-11: 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));
}
```

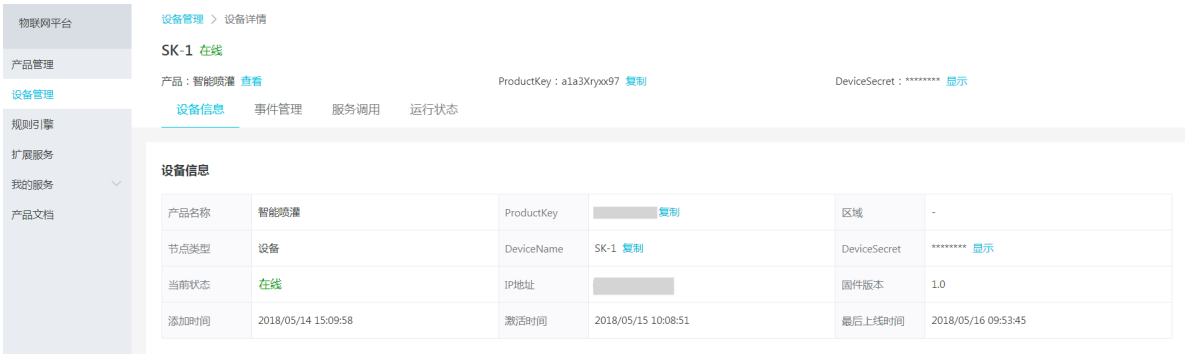
函数实现：

```
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 funstions(
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 **4-12:** 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 delimeter like "identifier1.ientifier2"
* 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 delimeter like "identifier1.ientifier2"
* 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 delimeter like "identifier1.ientifier2"
* 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.identifer}`发起设备的某个服务，定义在标准TSL的dsl.service.identifer中。

```
TOPIC: /sys/{productKey}/{deviceName}/thing/service/{dsl.service.
identifer}
REPLY TOPIC:
/sys/{productKey}/{deviceName}/thing/service/{dsl.service.identifer}
_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.identifer}/post`，触发某个事件，定义在标准TSL的dsl.event.identifer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_identifier[64];
snprintf(event_output_identifier, sizeof(event_output_identifier),
 "%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_identifier,
&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_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);
```

4.8 日志服务

本文主要介绍日志服务的三种类型及日志格式。

使用说明

日志类型分为三类：

- 设备行为分析
- 上行消息分析
- 下行消息分析

物联网平台支持按如下方式搜索日志：

搜索日志方式	说明
DeviceName	设备名称，该产品下设备唯一标识。可以根据deviceName搜索出该设备的相关日志。
MessageId	消息ID，某条消息在套件里的唯一标识。可以用messageId追踪到这条消息全链路的流转状况。
状态	日志有两个状态：成功和失败。
时间范围	可以筛选某一时间段打印出的日志。

说明：

- {}是变量，日志根据实际运行打印相应结果
- 日志提供的是英文，不会打印中文
- 一旦出现失败状态的日志内容，非system error的原因都是由于用户使用不当或者产品限制导致的，请仔细排查。

设备行为分析

设备行为主要有设备上线（online），设备下线(offline)的日志。

可按DeviceName，时间范围来筛选日志，操作界面如下图所示：

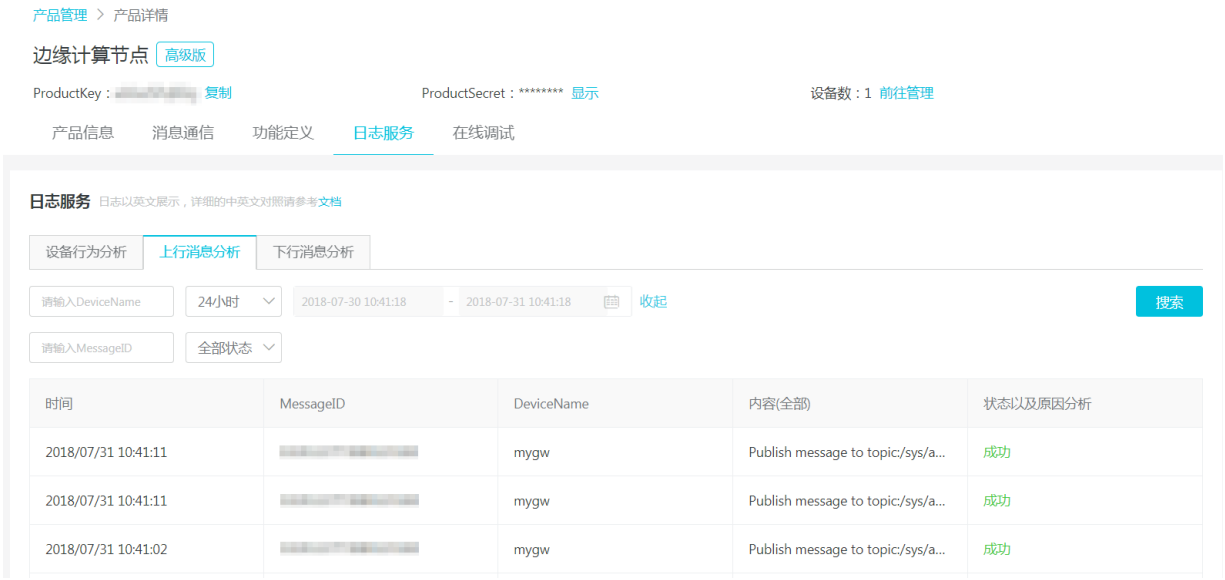
设备离线原因中英文对查表

英文	中文
Kicked by the same device	被相同设备踢下线
Connection reset by peer	TCP连接被对端重置
Connection occurs exception	通信异常,IoT服务端主动关闭连接
Device disconnect	设备主动发送MQTT断开连接请求
Keepalive timeout	心跳超时，IoT服务端关闭连接

上行消息分析

上行消息主要包括设备发送消息到topic，消息流转到规则引擎，规则引擎转发到云产品的日志。

可按DeviceName、MessageId、状态、时间范围来筛选日志，操作界面如下图所示：



上行消息中英文对查表

说明：

其中包括context（打印的英文日志+中文注释），error reason（打印的英文日志），失败的原因（中文注释）。

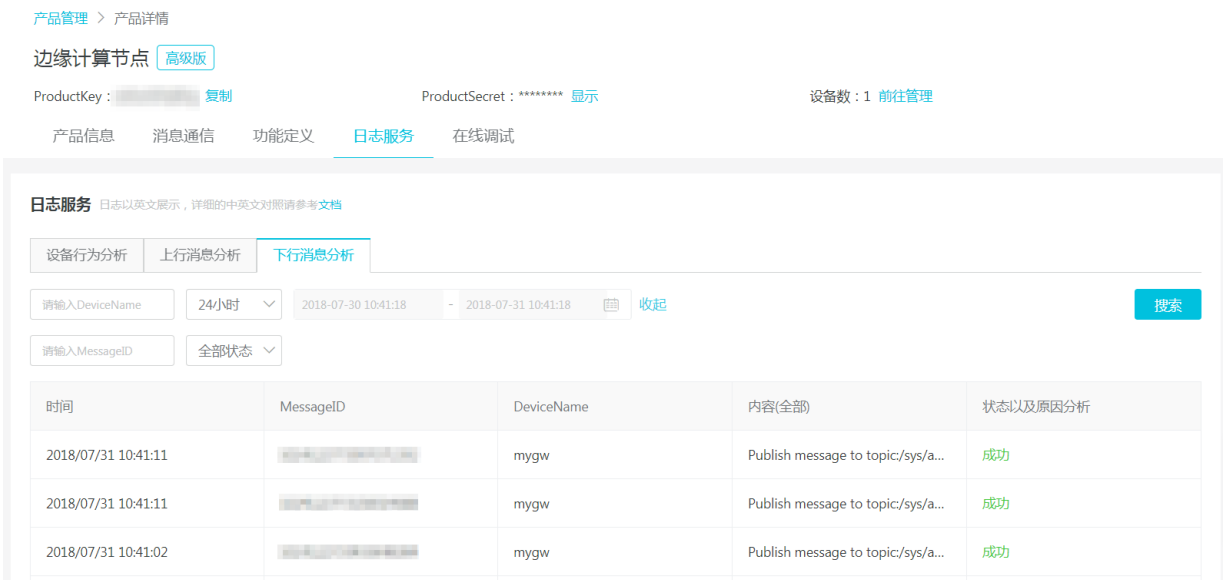
context	error reason	失败原因
Device publish message to topic:{},QoS={},protocolMessageId:{}设备发送消息到topic:{},QoS={},protocolMessageId:{}	Rate limit:{maxQps},current qps:{}	限流{最大流量},当前qps:{}
	No authorization	未授权
	System error	系统错误
send message to RuleEngine，topic:{} protocolMessageId:{} IoT Hub发送消息给规则引擎，topic:{} protocolMessageId:{}	{eg，too many requests}	失败信息，例如太多请求被限流导致失败
	System error	系统错误
Transmit data to DataHub, project:{},topic:{},from IoT topic:{}规则引擎发送数据到Datahub,project:{}，topic:{},来自IoT的topic:{}	DataHub Schema:{} is invalid!	DataHub Schema:{}无效，类型不匹配
	DataHub IllegalArgumentException:{}	Datahub 参数异常:{}
	Write record to dataHub occurs error! errors:[code:{},message:{}]	写数据到datahub出错，错误:[code:{},message{}]
	Datahub ServiceException:{}	Datahub服务异常:{}
	System error	系统错误
Transmit data to MNS,queue:{},theme:{},from IoT topic:{}发送数据到MNS，queue:{},topic:{},来自IoT的topic:{}	MNS IllegalArgumentException:{}	MNS参数异常:{}
	MNS ServiceException:{}	MNS服务异常:{}
	MNS ClientException:{}	MNS客户端异常:{}
	System error	系统错误
Transmit data to MQ,topic:{}, from IoT topic:{}规则引擎发送数据到MQ,topic:{},来自IoT的topic:{}	MQ IllegalArgumentException:{}	MQ参数异常:{}
	MQ ClientException:{}	MQ客户端异常:{}
	System error	系统错误
Transmit data to TableStore, instance:{},tableName:{},from IoT topic:{}规则引擎发送数据到TableStore，实例名:{},表名:{},来自IoT的topic:{}	TableStore IllegalArgumentException:{}	TableStore参数异常:{}
	TableStore ServiceException:{}	TableStore服务异常:{}
	TableStore ClientException:{}	TableStore客户端异常:{}
	System error	系统错误

Transmit data to RDS, instance:{},databaseName:{}, tableName:{},from IoT topic:{}规则引擎发送数据到RDS,实例名:{},数据库名:{},表名:{}	RDS IllegalArgumentException:{}	RDS参数异常:{}
	RDS CannotGetConnectionException:{}	RDS无法连接:{}
	RDS SQLException:{}	RDS SQL语句异常
	System error	系统错误
Republish topic, from topic:{} to target topic:{}规则引擎转发topic,从topic:{}到目标topic:{}	System error	系统错误
RuleEngine receive message from IoT topic:{}规则引擎接收消息，来自IoT的topic:{}	Rate limit:{maxQps},current qps:{}	限流{最大流量},当前qps:{}
	System error	系统错误
Check payload, payload:{}检测payload,payload:{}	Payload is not json	Payload的json格式不合法

下行消息分析

下行消息主要是云端发送消息到设备的日志。

可按DeviceName、MessageId、执行状态、时间范围来筛选日志，操作界面如下图所示：



下行消息中英文对查表

说明：

其中包括context（打印的英文日志+中文注释），error reason（打印的英文日志），失败的原因（中文注释）。

context	error reason	失败原因
Publish message to topic:{}, protocolMessageId:{}推送消息给topic:{},protocolMessageId:{}	No authorization	未授权
Publish message to device, QoS={}IoT Hub发送消息给设备	IoT hub cannot publish messages	因为服务端没有得到设备的puback，会一直发消息，直到超过50条的阈值然后IoT Hub就会无法发送消息
	Device cannot receive messages	设备端接受消息的通道阻塞，可能由于网络慢，或者设备端消息能力不足导致了服务端发送消息失败
	Rate limit:{maxQps},current qps:{}	限流{最大流量},当前qps:{}
Publish RRPC message to deviceIoT发送RRPC消息给设备	IoT hub cannot publish messages	设备端一直没有回复response，而且服务端一直发送消息超出阈值导致发送失败
	Response timeout	设备响应超时
	System error	系统错误
RRPC finishedRRPC结束	{e.g rrpcCode}	错误信息，会打出相应的RRPCCode,比如UNKNOW,TIMEOUT,OFFLINE, HALFCONN
Publish offline message to device IoT Hub发送离线消息给设备	Device cannot receive messages	设备端接受消息的通道阻塞，可能由于网络慢，或者设备端消息能力不足导致了服务端发送消息失败

4.9 SDK使用参考

4.9.1 iOS-SDK

本章节介绍如何使用iOS-SDK，将iOS手机接入物联网平台。

本章节中SDK 采用 cocoapods 管理依赖，建议采用 1.1.1 以上版本。

iOS SDK封装了 MQTT 建连、长连接维护和基于MQTT协议的上下行请求的功能。

快速接入

1. 在xcode工程Podfile中添加以下行，集成SDK。

```
source 'https://github.com/CocoaPods/Specs.git'
source 'https://github.com/aliyun/aliyun-specs.git'
target "necslinkdemo" do
pod 'IMLChannelCore'
pod 'OpenSSL'
end
```

2. 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**，获取需要的三元组信息，即productKey、deviceName及deviceSecret。
3. SDK API开发，具体请参见下文各SDK实现说明。

SDK 初始化

初始化时，使用三元组信息跟物联网平台建立可信安全的长连接通道，配置自己的server地址以及端口。

```
#import
#import
LKIoTConnectConfig * config = [LKIoTConnectConfig new];
config.productKey = @"your product key";
config.deviceName = @"your device name";
config.deviceSecret = @"your device secret";
config.server = @"www.youserver.com";//设为nil表示使用IoT套件作为连接服务器
config.port = 1883,//your server port。如果server被设置为nil。则port也不要
设置。
config.receiveOfflineMsg = NO;//如果希望收到客户端离线时的消息，可以设为YES.
[[LKIoTExpress sharedInstance]startConnect:config connectListener:self
];
//如果config.server置为nil，则默认连接的地址为上海节点：${yourproductKey}.
iot-as-mqtt.cn-shanghai.aliyuncs.com:1883`
```

上行请求接口

SDK 封装了上行Publish请求、订阅Subscribe和取消订阅unSubscribe等接口。

上行请求需要在 SDK 初始化建联成功之后才能正常调用。

```
/**
RPC请求接口，封装了业务的上行request以及下行resp。request业务报文由SDK内部按
alink标准协议封装，形如
{
"id":"msgId" //消息id
"system": {
```

```
"version": "1.0", // 必填，消息版本，目前为1.0
"time": "" // 必填，消息发生的时间，毫秒,
},
"request": {
},
"params": {
}
}
@param topic rpc请求的topic，由具体业务确定，完整topic形如：
/sys/${productKey}/${deviceName}/app/abc/cba
@param opts 可选配置项，可空，预留
示例，{"extraParam":{"method":"thing.topo.add"}}
则会将"method":"thing.topo.add"塞到最终的业务报文里,跟业务报文中的params平级。
@param params 业务参数。
@param responseHandler 业务服务器响应回调接口，参见LKExpressResponse
*/
-(void)invokeWithTopic:(NSString *)topic opts:(NSDictionary* _Nullable
)opts
params:(NSDictionary*)params respHandler:(LKExpressResponseHandler)
responseHandler;
/**
上行数据，直接透传，不会再按alink业务报文协议封装
@param topic 消息topic
@param dat 需透传的数据
@param completeCallback 数据上行结果回调
*/
-(void)uploadData:(NSString *)topic data:(NSData *)dat complete:(
LKExpressOnUpstreamResult)completeCallback;
/**
上行数据，不会有回执。SDK会按alink标准协议封装业务报文。
@param topic 消息topic,完整的topic
@param params 业务参数
@param completeCallback 数据上行结果回调
*/
-(void)publish:(NSString *) topic params:(NSDictionary *)params
complete:(LKExpressOnUpstreamResult)completeCallback;
/**
订阅topic的接口
@param topic 订阅的消息的topic，由具体业务确定，需要传完整的topic区段,形如：
/sys/${productKey}/${deviceName}/app/abc/cba
@param completionHandler 订阅流程结束的callback，如果error为空表示订阅成
功，否则订阅失败
*/
- (void)subscribe:(NSString *)topic complete: (void (^)(NSError *
_Nullable error))completionHandler;
/**
取消订阅topic的接口
@param topic 订阅的消息的topic，由具体业务确定，需要传完整的topic区段,形如：
/sys/${productKey}/${deviceName}/app/abc/cba
@param completionHandler 取消订阅流程结束的callback，如果error为空表示订阅成
功，否则订阅失败
*/
- (void)unsubscribe : (NSString *)topic complete: (void (^)(NSError *
_Nullable error))completionHandler;
```

三个上行方法的区别如下：

- `-(void)invokeWithTopic:(NSString )topic opts:(NSDictionary _Nullable) opts respHandler:(LKExpressResponseHandler)responseHandler;` ：这个是业务请求响应模型，发一个请求回去，服务端的响应会在responseHandler回调中抛回来。
- `uploadData:(NSString )topic data:(NSData )dat complete:(LKExpressOnUpstreamResult)completeCallback;`：这个是数据透传接口，不进行任何处理，直接上行到云端，也不会有响应回来。
- `-(void)publish:(NSString ) topic params:(NSDictionary )params complete:(LKExpressOnUpstreamResult)completeCallback;`：数据上行时会按alink业务报文协议封装后再上行。alink业务报文协议在api reference里有较详细的说明。

业务请求响应模型调用示例：

```
NSString *topic = @"/sys/${productKey}/${deviceName}/account/bind";
NSDictionary *params = @{
@"iotToken": token,
};
[[LKIoTExpress sharedInstance] invokeWithTopic:topic
opts:nil
params:params
respHandler:^(LKExpressResponseHandler * _Nonnull response) {
if (![response successed]) {
NSLog(@"业务请求失败");
}
}];
//其中${productKey}为三元组信息之productKey
//其中${deviceName}为三元组信息之deviceName
```

订阅topic调用示例：

```
NSString *topic = @"/sys/${productKey}/${deviceName}/app/down/event";
[[LKIoTExpress sharedInstance] subscribe:topic complete: ^(NSError *
error) {
if (error != nil) {
NSLog(@"业务请求失败");
}
}];
```

下行数据监听

订阅Topic后，云端推送的下行消息监听。

```
@protocol LKExpressDownListener<NSObject>
-(void)onDownstream:(NSString *) topic data: (id _Nullable) data;///<
topic-消息topic，data-消息内容,NSString 或者 NSDictionary

-(BOOL)shouldHandle:(NSString *)topic;///<数据使用onDownstream:data:上抛
时，可以先过滤一遍，如返回NO，则不上传，返回YES，则会使用onDownstream:data:上抛
```

```
@end

[[LKIoTExpress sharedInstance] addDownstreamListener:LKExpressD
ownListenerTypeGw listener:(id<LKExpressDownListener>)downListener];
//downListener在本SDK中 是 weak reference，所以需要调用者保证生命周期。
```

设计建议

建议通道建连时，所使用的三元组跟C端用户账号解耦。即应用只使用一个三元组去物联网平台建立连接。

C端用户账号可以任意切换，切换账号时，只需要变更C端账号跟三元组的绑定关系即可。

最终的形态是C端账号A可以使用该通道，切到另一个C端账号B，重新绑定三元组关系后，也可以使用该通道，即C端账号跟通道的绑定关系可以动态变更。

4.9.2 Android-SDK

本章节介绍如何使用Android-SDK，将Android设备接入物联网平台。

Android SDK封装了 MQTT 建连、长连接维护和基于MQTT协议的上下行请求的功能。

快速接入

1. SDK引入。
   a. 在根目录下的build.gradle 基础配置文件中，加入仓库地址，进行仓库配置。

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

   b. 在模块的 build.gradle 中，添加 public-channel-core 的依赖，引入 SDK :public-channel-core。

```
compile('com.aliyun.alink.linksdk:public-channel-core:0.2.1')
```

   c. SDK 基于Paho mqttv3 Java开源库完成MQTT连接，自动间接依赖相关库。

```
compile 'org.eclipse.paho:org.eclipse.paho.client.mqttv3:1.2.0'
```

```
compile 'com.alibaba:fastjson:1.2.40'
```

2. 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，获取需要的三元组信息，即productKey、deviceName及deviceSecret。
3. SDK API开发指引，具体请参见下文各个SDK详细说明。

SDK 日志开关

打开SDK内部日志输出开关：

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

SDK 环境配置

SDK环境配置，需要在SDK初始化之前。

SDK支持自定义切换Host。

默认连接的地址为上海节点： `ssl://[productKey].iot-as-mqtt.cn-shanghai.aliyuncs.com:1883`

还包括美西节点和新加坡节点：

```
MqttConfigure.HOST = "ssl://[productKey].iot-as-mqtt.us-west-1.aliyuncs.com:1883";//美西节点
MqttConfigure.HOST = "ssl://[productKey].iot-as-mqtt.ap-southeast-1.aliyuncs.com:1883";//新加坡节点
```

SDK 初始化

使用三元组信息对SDK进行初始化，SDK会进行 MQTT 建联。

```
MqttInitParams initParams = new MqttInitParams(productKey,deviceName,deviceSecret);
PersistentNet.getInstance().init(context,initParams);
```

添加通道状态变化监听

添加通道状态变化监听，获取MQTT建连结果的反馈和监听后续通道状态变化。

```
PersistentEventDispatcher.getInstance().registerOnTunnelStateListener(channelStateListener,isUiSafe);// 注册监听
PersistentEventDispatcher.getInstance().unregisterOnTunnelStateListener(connectionStateListener);//取消监听
    public interface IConnectionStateListener {
        void onConnectFail(String msg); //通道连接失败
        void onConnected(); // 通道已连接
```

```
        void onDisconnect(); // 通道断连
}
```

上行请求接口

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

```
/**
* @param request 发送数据对象基类，需要根据不同的INet实现类传入不同的ARequest子类对象
*                长连接传入：PersistentRequest对象
*                短连接传：TransitoryRequest对象
* @param listener 发送结果回调接口
* @return AConnect对象，可以用于重试
*/
ASend asyncSend(ARequest request, IOnCallListener listener);
/**重新发送请求
* @param connect 上次调用asyncSend返回值
*/
void retry(ASend connect);
/** 订阅下行消息，订阅后的消息统一在 IOnPushListener 中接收
* @param topic
* @param listener
*/
void subscribe(String topic,IOnSubscribeListener listener);
/** 取消订阅下行消息
* @param topic
* @param listener
*/
void unSubscribe(String topic,IOnSubscribeListener listener);
```

调用示例：

```
//Publish 请求
MqttPublishRequest publishRequest = new MqttPublishRequest();
publishRequest.isRPC = false;
publishRequest.topic = "/productKey/deviceName/update";
publishRequest.payloadObj = "content";
PersistentNet.getInstance().asyncSend(publishRequest, new IOnCallLis
tener() {
        @Override
        public void onSuccess(ARequest request, AResponse response) {
            ALog.d(TAG,"send , onSuccess");
}
         @Override
         public void onFailed(ARequest request, AError error) {
             ALog.d(TAG,"send , onFailed");
}
         @Override
         public boolean needUISafety() {
              return false;
}
});
```

```
//订阅
PersistentNet.getInstance().subscribe(topic, listener);
//取消订阅
PersistentNet.getInstance().unSubscribe(topic, listener);
```

下行数据监听

订阅Topic后，云端推送的下行消息监听。

```
PersistentEventDispatcher.getInstance().registerOnPushListener(
onPushListener,isUiSafe);// 注册监听
PersistentEventDispatcher.getInstance().unregisterOnPushListener(
onPushListener);//取消监听
         public interface IOnPushListener {
             /**下推消息回调接口
               * @param topic
               * @param data 下推的消息内容
              */
             void onCommand(String topic,String data);
            /**根据method过滤消息
             * @param topic : 本次下推消息的命令名称
              * @return : 若返回true，则onCommand被调用，返回false，则
onCommand不会被调用
             */
             boolean shouldHandle(String topic);
}
```

实例代码

完整Android工程项目Demo，请访问*aliyunIoTAndroidDemo.zip*下载。

4.9.3 JAVA-SDK使用(MQTT)

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。

4.9.4 C-SDK

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 4-18: 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)
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

介绍华东2站点V2.0+版本的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!.*:\(.*\)(!\1!'|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
```

```
IOT_CoAP_SendMessage
IOT_CoAP_GetMessagePayload
IOT_CoAP_GetMessageCode
IOT_HTTP_Init
IOT_HTTP_DeInit
IOT_HTTP_DeviceNameAuth
IOT_HTTP_SendMessage
IOT_HTTP_Disconnect
IOT_MQTT_Construct
IOT_MQTT_ConstructSecure
IOT_MQTT_Destroy
IOT_MQTT_Yield
IOT_MQTT_CheckStateNormal
IOT_MQTT_Disable_Reconnect
IOT_MQTT_Subscribe
IOT_MQTT_Unsubscribe
IOT_MQTT_Publish
IOT_OTA_Init
IOT_OTA_Deinit
IOT_OTA_ReportVersion
IOT_OTA_RequestImage
IOT_OTA_ReportProgress
IOT_OTA_GetConfig
IOT_OTA_IsFetching
IOT_OTA_IsFetchFinish
IOT_OTA_FetchYield
IOT_OTA_Ioctl
IOT_OTA_GetLastError
IOT_Shadow_Construct
IOT_Shadow_Destroy
IOT_Shadow_Yield
IOT_Shadow_RegisterAttribute
IOT_Shadow_DeleteAttribute
IOT_Shadow_PushFormat_Init
IOT_Shadow_PushFormat_Add
IOT_Shadow_PushFormat_Finalize
IOT_Shadow_Push
IOT_Shadow_Push_Async
IOT_Shadow_Pull
IOT_Gateway_Generate_Message_ID
IOT_Gateway_Construct
IOT_Gateway_Destroy
IOT_Subdevice_Register
IOT_Subdevice_Unregister
IOT_Subdevice_Login
IOT_Subdevice_Logout
IOT_Gateway_Get_TOPO
IOT_Gateway_Get_Config
IOT_Gateway_Publish_Found_List
IOT_Gateway_Yield
IOT_Gateway_Subscribe
IOT_Gateway_Unsubscribe
IOT_Gateway_Publish
IOT_Gateway_RRPC_Register
IOT_Gateway_RRPC_Response
linkkit_start
linkkit_end
linkkit_dispatch
linkkit_yield
linkkit_set_value
linkkit_get_value
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 4-19: 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
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
CoAP功能相关		
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

序号	函数名	说明
云端连接**Cloud Connection**功能相关		
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
CMP功能相关		
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
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_OTA_Start	初始化ota功能，上报版本
9	IOT_CMP_OTA_Set_Callback	设置OTA回调函数
10	IOT_CMP_OTA_Get_Config	获取远程配置
11	IOT_CMP_OTA_Request_Image	获取固件
12	IOT_CMP_OTA_Yield	通过CMP完成OTA功能

序号	函数名	说明
MQTT功能相关		
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
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, 内含了心跳的维持, 服务器下行报文的收取等
OTA功能相关(模组实现时的可选功能)		
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

序号	函数名	说明
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
HTTP功能相关		
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
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

序号	函数名	说明
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)		
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

序号	函数名	说明
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conifg/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
linkkit()		
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
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

序号	函数名	说明
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

4.9.5 HTTP/2 SDK

您可以使用HTTP/2协议建立设备与物联网平台之间的通信。此处为您提供一个HTTP/2的SDK Demo，您可以参考此Demo，开发您的SDK。

前提条件

此Demo为Maven工程，请先安装Maven。

操作步骤

1. 下载HTTP/2 SDK。下载地址为：*iot-http2-sdk-demo*。
2. 使用idea或者eclipse，将该Demo导入到工程里面。
3. 从控制台获取设备的三元组信息。具体请参考**用户指南>创建产品与设备**相关文档。
4. 修改H2Client.java配置文件。
   a. 配置参数。

```
// TODO 从控制台获取productKey、deviceName、deviceSecret信息
String productKey = "";
String deviceName = "";
String deviceSecret = "";

// 用于测试的topic
String subTopic = "/" + productKey + "/" + deviceName + "/get";
String pubTopic = "/" + productKey + "/" + deviceName + "/update";
```

   b. 连接HTTP/2服务器，并接收数据。

```
// endPoint: https://${uid}.iot-as-http2.${region}.aliyuncs.com
String endPoint = "https://" + productKey + ".iot-as-http2.cn-shanghai.aliyuncs.com";

// 客户端设备唯一标记
```

```
String clientId = InetAddress.getLocalHost().getHostAddress();

// 连接配置
Profile profile = Profile.getDeviceProfile(endPoint, productKey,
deviceName, deviceSecret, clientId);

// 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
profile.setCleanSession(false);

// 构造客户端
MessageClient client = MessageClientFactory.messageClient(profile
);

// 数据接收
client.connect(messageToken -> {
Message m = messageToken.getMessage();
System.out.println("receive message from " + m);
return MessageCallback.Action.CommitSuccess;
});
```

c. 订阅Topic。

```
// topic订阅。订阅成功后，即可在建连时的回调接口中收到消息
CompletableFuture subFuture = client.subscribe(subTopic);
System.out.println("sub result : " + subFuture.get());
```

d. 发送数据。

```
// 发布消息
MessageToken messageToken = client.publish(pubTopic, new Message("
hello iot".getBytes(), 0));
System.out.println("publish success, messageId: " + messageToken.
getPublishFuture().get().getMessageId());
```

5. 配置修改完成后，直接运行。

接口说明

- 身份认证

  设备连接物联网平台时，需要使用Profile配置设备身份及相关参数。具体接口参数如下：

```
Profile profile = Profile.getDeviceProfile(endPoint, productKey,
deviceName, deviceSecret, clientId);
MessageClient client = MessageClientFactory.messageClient(profile);
client.connect(messageToken -> {
    Message m = messageToken.getMessage();
    System.out.println("receive message from " + m);
    return MessageCallback.Action.CommitSuccess;
});
```

  Profile参数说明

名称	类型	是否必须	描述
endPoint	String	是	接入点地址，格式为https://${productKey}.iot-as-http2.${region}.aliyuncs.com。目前仅支持cn-shanghai。
productKey	String	是	设备所隶属产品的Key，可从控制台获取。
deviceName	String	是	设备的名称，可从控制台获取。
deviceSecret	String	是	设备的密钥，可从控制台获取。
clientId	String	是	客户端设备的唯一标识
cleanSession	Boolean	否	清除缓存消息
heartBeatInterval	Long	否	心跳间隔，单位为ms。
heartBeatTimeOut	Long	否	心跳超时时间，单位为ms。
multiConnection	Boolean	否	是否使用多连接，使用productkey和deivceName接入时请设置为false。
callbackThreadCorePoolSize	Integer	否	回调线程池corePoolSize
callbackThreadMaximumPoolSize	Integer	否	回调线程池MaximumPoolSize
callbackThreadBlockingQueueSize	Integer	否	回调线程池BlockingQueueSize
authParams	Map	否	自定义认证参数

- 建立连接

  获取MessageClient之后需要与物联网平台建立连接。身份认证成功后才可收发消息。连接建立成功，服务端会立即向SDK推送已订阅的消息，因此建连时需要提供默认消息接收接口，用于处理未设置回调的消息。建连接口如下：

```
void connect(MessageCallback messageCallback);
```

- 消息订阅

```
/**
* 订阅topic
* @param topic topic
* @return completableFuture for subscribe result
*/

CompletableFuture subscribe(String topic);
```

```
/**
* 订阅topip并制定该topic回调
* @param topic topic
* @param messageCallback callback when message received on this
topic
* @return completableFuture for subscribe result
*/
CompletableFuture subscribe(String topic, MessageCallback messageCal
lback);

/**
* 取消订阅
*
* @param topic topic
* @return completableFuture for unsubscribe result
*/
CompletableFuture unsubscribe(String topic);
```

- 消息接收

  消息接收需要实现MessageCallback接口，并在连接或订阅Topic时传入MessageCleint，具体接口如下：

```
/**
*
* @param messageToken message token
* @return Action after consuming
*/
Action consume(final MessageToken messageToken);
```

  通过MessageToken.getMessage获取消息后，该方法会被调用。因接口在线程池中调用，所以请注意现成安全问题。该方法返回值用于处理QoS1及QoS2的ACK，返回值说明如下：

返回值	描述
Action.CommitSuccess	回执ACK
Action.CommitFailure	不回执ACK，消息稍后会重新收到
Action.CommitAckManually	不回执ACK，需要手动调用MessageClient.ack() 方法回复

- 消息发送

```
/**
* 发送消息到指定topic
*
* @param topic topic
* @param message message entity
* @return completableFuture for publish result
*/
```

```
MessageToken publish(String topic, Message message);
```

4.9.6 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/utils: 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 **4-20: 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 4-21: 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

  请参考快速接入页面和官方*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 **4-13:** 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*$(SDK_PATH)/src/platform/os*/下。
  - network层主要包括网络相关的接口，目前为TCP接口，在目录*$(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 **4-22:** 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

序号	数据类型名称	说明
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： #define true (1)
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 4-23: 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

序号	接口名称	说明
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
11	aliot_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	aliot_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	aliot_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：（$(SDK_PATH)/src/platform/os/aliot_platform_os.h）。
- 实现时，请在路径$(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 4-24: TCPIP网络接口

序号	接口名称	说明
1	aliot_platform_tcp_establish	建立tcp连接，返回连接句柄
2	aliot_platform_tcp_destroy	释放一个tcp连接
3	aliot_platform_tcp_write	往TCP通道写入数据。注意实现超时参数
4	aliot_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：（$(SDK_PATH)/src/platform/network/aliot_platform_network.h）。
- 实现时，请在路径$(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 4-25: SSL相关接口说明

序号	接口名称	说明
1	aliot_platform_ssl_establish	建立经SSL加密的传输通道
2	aliot_platform_ssl_destroy	释放一个SSL通道
3	aliot_platform_ssl_write	往SSL通道写入数据。注意实现超时参数
4	aliot_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：（$(SDK_PATH)/src/platform/ssl/aliot_platform_ssl.h）。
- 实现时，请在路径$(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。

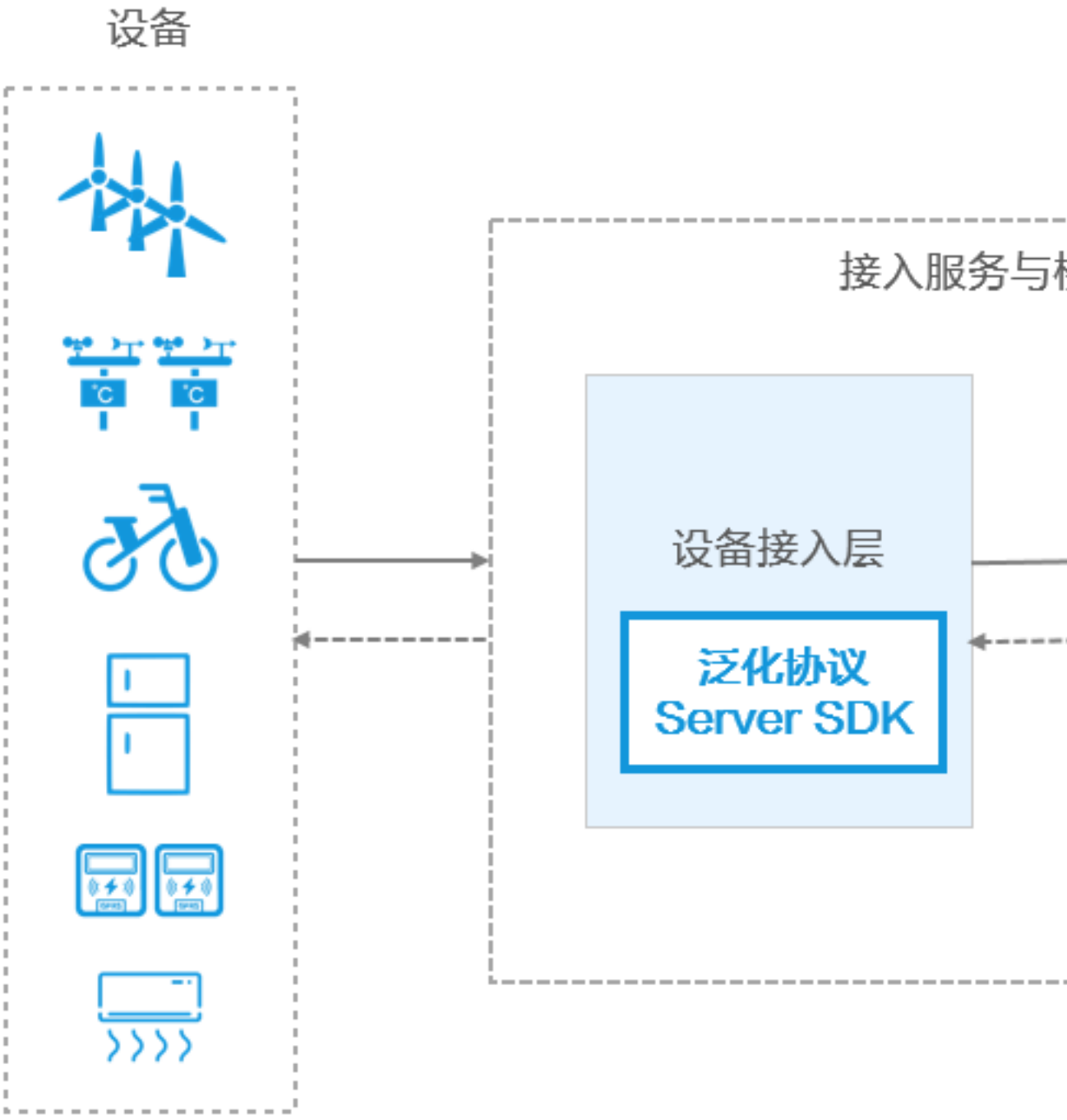
5 泛化协议

5.1 概览

阿里云物联网平台已经支持基于MQTT、CoAP和HTTP等一些协议的通信，其他类型协议，如消防协议GB/T 26875.3-2011、Modbus、JT808等暂未接入，在特定场景下，部分设备也可能无法直接接入物联网平台。此时，您需要使用泛化协议SDK，快速构建桥接服务，搭建设备或平台与阿里云物联网平台的双向数据通道。

泛化协议SDK

泛化协议SDK是协议自适应的框架，用以构建与阿里云物联网平台进行高效双向通信的桥接服务。服务架构如下图所示：



泛化协议一共提供两个SDK：核心SDK与Server SDK。

- **泛化协议核心SDK**

  核心SDK抽象了如Session管理、配置管理等通用能力，在设备与物联网平台中间，扮演网桥角色，代替设备与物联网平台通信，有效降低了开发者适配物联网平台的复杂性。其主要功能如下：

— 提供非持久化的Session管理能力。

— 提供基于配置文件的配置管理能力。

— 提供设备连接管理能力。

— 提供上行通讯能力。

— 提供下行通讯能力。

— 支持设备认证。

当您的设备已经使用传统方式连接物联网，现在希望将已有设备或平台桥接至阿里云物联网平台时，请使用核心SDK。

- **Server SDK**

  泛化协议Server SDK在核心SDK基础上提供了基于TCP/UDP的设备端接入服务。主要功能如下：

  — 支持基于TCP/UDP传输的任意协议。

  — 支持TLS/SSL加密传输。

  — 支持接入容量水平扩展。

  — 基于Netty的通信服务。

  — 提供自动化可定制的设备连接管理能力。

  若您希望从头构建接入服务，请使用附带了Socket通信功能的Server SDK。

开发和部署

控制台创建产品与设备

参考创建产品章节，创建网桥产品与设备。获取网桥三元组。

SDK依赖

泛化协议SDK目前仅提供Java语言版本，支持JDK 1.8及以上版本。Maven依赖如下：

```
<!-- 核心SDK -->
<dependency>
    <groupId>com.aliyun.openservice</groupId>
    <artifactId>iotx-as-bridge-sdk-core</artifactId>
    <version>1.0.0</version>
</dependency>

<!-- Server SDK -->
<dependency>
    <groupId>com.aliyun.openservice</groupId>
    <artifactId>iotx-as-bridge-sdk-server</artifactId>
    <version>1.0.0</version>
```

```
</dependency>
```

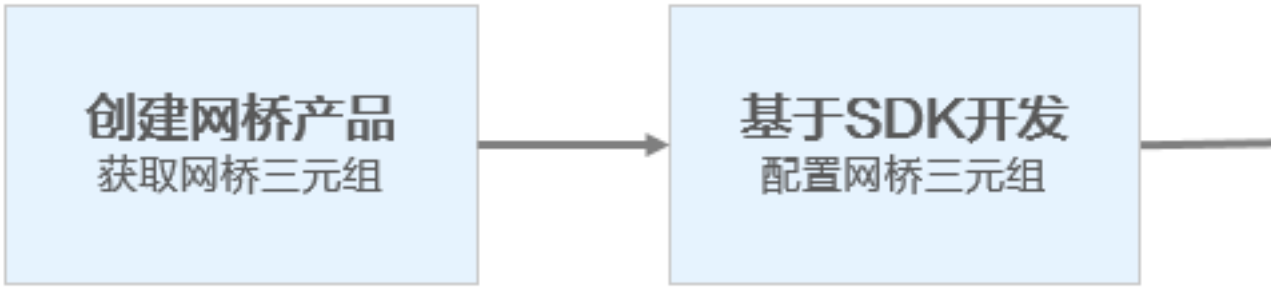
开发SDK

核心SDK开发和Server SDK开发为您简单介绍了开发流程。具体实现细节，请参考Javadoc。

部署服务

已开发完成的可运行的桥接服务，可以使用阿里云ECS和SLB等服务，以高度可扩展的方式部署至阿里云上；也可以直接部署到本地环境中，以保证可信通信环境。

以基于阿里云云服务器ECS为例，上线流程如下:



5.2 核心SDK开发

基于泛化协议核心SDK，您可以将物联网平台桥接服务与已有的接入服务或平台进行高效整合，使设备或者服务得以快速接入阿里云物联网平台。

准备工作

泛化协议核心SDK概念、功能及Maven依赖，请参考概览。

配置管理

泛化协议核心SDK默认使用基于文件的配置管理。自定义配置请参见自定义组件>配置管理。

- 支持格式包括Java Proterties、JSON以及可读性较好的HOCON（JSON超集）。
- 支持结构化配置项以提升可维护性。
- 可以通过Java系统属性覆盖文件配置，如java -Dmyapp.foo.bar=10。
- 支持配置文件分离和嵌套引用。

application.conf

名称	描述	是否必须
productKey	网桥所属产品的ProductKey	是
deviceName	网桥对应的DeviceName，默认值为ECS MAC地址	否
deviceSecret	网关对应的DeviceSecret	否
http2Endpoint	HTTP2网关服务地址	是
authEndpoint	设备认证服务地址	是
popClientProfile	Open API调用客户端相关配置，参见*API Client*配置。	是

API Client配置

名称	描述	是否必须
accessKey	Open API调用用户身份标识	是
accessSecret	Open API调用用户身份标识对应的秘钥	是
name	API客户端profile名称	是
region	调用的服务端所在地域	是
product	产品名称，除非特殊情况否则请配置为 *Iot*	是
endpoint	调用的服务端地址	是

devices.conf

配置设备在阿里云物联网平台的映射，即配置设备三元组。自定义配置文件请参见自定义组件>配置管理。

```
XXXX // 表示设备的原始标识
{
    productKey: "",
    deviceName: "",
    deviceSecret: ""
}
```

开发接口

初始化

com.aliyun.iot.as.bridge.core.BridgeBootstrap负责初始化设备到阿里云物联网平台的通信。创建新的BridgeBootstrap实例后，网关基本配置的管理组件会被初始化。使用自定义的配置管理，请参见自定义组件>配置管理。

请通过以下接口完成初始化。

- bootstrap(): 不实现消息下行功能并进行初始化。
- bootstrap(DownlinkChannelHandler handler): 采用开发者指定的DownlinkChannelHandler进行初始化。

示例代码：

```
BridgeBootstrap bootstrap = new BridgeBootstrap();
// 不实现下行通讯
bootstrap.bootstrap();
```

设备上线

当设备建立连接或者发送连接请求时，应发起设备上线操作，只有设备上线后，才能与阿里云物联网平台通讯。设备上线分为两个部分：本地Session的初始化以及设备认证。

1. Session初始化

   泛化协议SDK提供非持久化的本地Session管理功能，关于自定义Session管理，请参见自定义组件>*Session*管理。

   创建接口：

   - com.aliyun.iot.as.bridge.core.model.Session.newInstance(String originalIdentity, Object channel)
   - com.aliyun.iot.as.bridge.core.model.Session.newInstance(String originalIdentity, Object channel, int heartBeatInterval)
   - com.aliyun.iot.as.bridge.core.model.Session.newInstance(String originalIdentity, Object channel, int heartBeatInterval, int heartBeatProbes)

   其中originalIdentity表示原始协议中设备的唯一标识，如序列号、编号等；channel为设备连接至桥接服务的通信通道如Netty的Channel；heartBeatInterval与heartBeatProbes用于自动的心跳过期监测，分别代表心跳的间隔时间（以秒为单位）以及允许的最大心跳丢失次数，超过该次数后将发送心跳超时事件，开发者可通过注册com.aliyun.iot.as.bridge.core.session.SessionListener来处理心跳超时事件。

2. 设备认证

设备本地Session初始化完成后请调用com.aliyun.iot.as.bridge.core.handler.UplinkChannelHandler.doOnline(Session newSession, String originalIdentity, String... credentials)完成本地设备认证和阿里云物联网平台上线认证，并根据结果允许设备后续通信或者断开连接。默认无本地认证，到阿里云物联网平台上线认证由SDK提供支持，如需使用自定义本地认证，请参见自定义组件>连接认证。

示例代码：

```
UplinkChannelHandler uplinkHandler = new UplinkChannelHandler();
Session session = Session.newInstance(device, channel);
boolean success = uplinkHandler.doOnline(session, originalIdentity);
if (success) {
    // 上线成功，接受后续通信请求
} else {
    // 上线失败，拒绝后续通信请求，断开连接（如有）
}
```

设备下线

当设备连接断开或者探测到需要断开连接的时候，应发起设备下线操作，通过接口com.aliyun.iot.as.bridge.core.handler.UplinkChannelHandler.doOffline(String originalIdentity)实现设备下线。

示例代码：

```
UplinkChannelHandler uplinkHandler = new UplinkChannelHandler();
uplinkHandler.doOffline(originalIdentity);
```

上报数据

开发者可通过com.aliyun.iot.as.bridge.core.handler.UplinkChannelHandler上报数据到阿里云物联网平台。操作包括以下三步：获取上行数据关联的设备、找到设备关联的Session、上报数据到阿里云物联网平台。请通过以下接口完成数据上报。

说明：

请开发者务必注意控制是否上报数据以及安全相关前置处理。

- CompletableFuture doPublishAsync(String originalIdentity, String topic, byte[] payload, int qos)：异步发送数据并立即返回，开发者需根据future判断发送结果。
- CompletableFuture doPublishAsync(String originalIdentity, ProtocolMessage protocolMsg)：异步发送数据并立即返回，开发者需根据future判断发送结果。
- boolean doPublish(String originalIdentity, ProtocolMessage protocolMsg, int timeout)：异步发送数据并等待发送结果返回。

- boolean doPublish(String originalIdentity, String topic, byte[] payload, int qos, int timeout)：异步发送数据并等待发送结果返回。

示例代码：

```
UplinkChannelHandler uplinkHandler = new UplinkChannelHandler();
DeviceIdentity identity = ConfigFactory.getDeviceConfigManager().
getDeviviceIdentity(device);
if (identity == null) {
    // 设备未映射到阿里云物联网平台上的设备，丢弃消息
    return;
}
Session session = SessionManagerFactory.getInstance().getSession(
device);
if (session == null) {
    // 设备尚未上线，上报数据到阿里云物联网平台前请务必确保设备已上线，请上线设备或者丢弃消息
}
boolean success = uplinkHandler.doPublish(session, topic, payload, 0,
10);
if(success) {
    // 上报数据到阿里云物联网平台成功
} else {
    // 上报数据到阿里云物联网平台失败
}
```

下行消息

泛化协议SDK提供了com.aliyun.iot.as.bridge.core.handler.DownlinkChannelHandler作为下行到设备的数据分发处理器，支持单播和广播(如果云端下发的数据不包含特定设备信息则为广播)。

示例代码：

```
public class SampleDownlinkHandler implements DownlinkChannelHandler {
    @Override
    public boolean pushToDevice(Session session, String topic, byte[]
payload) {
        // 处理设备消息推送
    }

    @Override
    public boolean broadcast(String topic, byte[] payload) {
        // 处理广播
    }
}
```

自定义组件

开发者可以自定义设备连接认证、Session管理以及配置管理组件，如果期望使用自定义的组件，请务必在调用BridgeBootstrap初始化之前完成这些组件的初始化和替换。

连接认证

自定义设备连接认证需实现接口com.aliyun.iot.as.bridge.core.auth.AuthProvider，并在BridgeBootstrap初始化前调用com.aliyun.iot.as.bridge.core.auth.AuthProviderFactory.init(AuthProvider customizedProvider)将认证组件替换为自定义实现。

Session管理

自定义Session管理需实现接口com.aliyun.iot.as.bridge.core.session.SessionManager，并在BridgeBootstrap初始化前调用com.aliyun.iot.as.bridge.core.session.SessionManagerFactory.init(SessionManager<?> customizedSessionManager)将Session组件替换为自定义实现。

配置管理

自定义配置管理需实现接口com.aliyun.iot.as.bridge.core.config.DeviceConfigManager和com.aliyun.iot.as.bridge.config.BridgeConfigManager，并在BridgeBootstrap初始化前调用 com.aliyun.iot.as.bridge.core.config.ConfigFactory.init(BridgeConfigManager bcm, DeviceConfigManager dcm)将配置管理组件替换为自定义实现，其中参数均可为空，如果为空则使用泛化协议SDK默认实现。

5.3 Server SDK开发

5.3.1 Server SDK开发

基于泛化协议Server SDK，您可以快速搭建桥接物联网平台的接入服务，使设备或者服务快速接入阿里云物联网平台。

准备工作

泛化协议Server SDK概念、功能及Maven依赖，请参考概览。

配置管理

泛化协议Server SDK默认提供基于文件的配置管理，可通过在*application.conf* 中增加 **socketServer** 配置项设置如下表所示的Socket Server相关参数。自定义配置管理请参见自定义组件>配置管理。

名称	描述	是否必须
address	连接服务监听地址，支持网卡名称如eth1，IPv4地址前缀如10.30，建议明确指定	否
backlog	TCP连接backlog的数量	否

名称	描述	是否必须
ports	连接服务监听端口，可指定多个，默认为：9123	否
listenType	socket server类型。取值为udp或tcp，默认为tcp。不区分大小写	否
broadcastEnabled	是否支持udp广播。当**listenType**为udp时有效，默认为true	否
unsecured	是否支持不加密的TCP连接。当**listenType**为tcp时有效	否
keyPassword	证书库密码。当**listenType**为tcp时有效	否 说明： 只有同时配置keyPassword、keyStoreFile和keyStoreType才会生效，否则配置将被忽略。
keyStoreFile	证书库文件地址。当**listenType**为tcp时有效	否
keyStoreType	证书库类型。当**listenType**为tcp时有效	否

开发接口

以下两篇文档假设开发者对基于Netty的开发已有基本了解。关于Netty的知识，请参考*Netty*文档。

- *TCP*类型*Server*开发接口
- *UDP*类型*Server*开发接口

自定义组件

除基于文件的配置管理外，开发者还可以自定义server的配置管理。

自定义配置管理需实现接口com.aliyun.iot.as.bridge.server.config.BridgeServerConfigManager，调用 com.aliyun.iot.as.bridge.server.config.ServerConfigFactory.init(BridgeServerConfigManager bcm)将默认配置管理组件替换为自定义组件，完成自定义组件的初始化。然后，再启动网桥。

5.3.2 TCP Server开发接口

基于泛化协议SDK的TCP Server开发接口，您可以快速构建以TCP为传输协议的接入服务，并将其与阿里云物联网平台桥接。

启动引导

com.aliyun.iot.as.bridge.server.BridgeServerBootstrap是启动Socket服务器和桥接服务的引导类。创建新的BridgeServerBootstrap实例后，会初始化基于配置文件的配置管理组件。

示例代码：

```
BridgeServerBootstrap bootstrap = new BridgeServerBootstrap(new
TcpDecoderFactory() {
    @Override
    public ByteToMessageDecoder newInstance() {
        // 返回解码器
    }
}, new TcpEncoderFactory() {
    @Override
    public MessageToByteEncoder<?> newInstance() {
        // 返回编码器
    }
}, new TcpBasedProtocolAdaptorHandlerFactory() {
    @Override
    public CustomizedTcpBasedProtocolHandler newInstance() {
        // 返回协议适配器
    }
}, new DefaultDownlinkChannelHandler());
try {
    bootstrap.start();
} catch (BootException | ConfigException e) {
    // 处理启动失败
}
```

实例化TCP类型BridgeServerBootstrap

- com.aliyun.iot.as.bridge.server.channel.factory.TcpDecoderFactory: 协议解码器工厂类具体实现，负责如何创建一个新的解码器实例，用于上行数据的解析，线程安全，可为null。
- com.aliyun.iot.as.bridge.server.channel.factory.TcpEncoderFactory: 协议编码器工厂类具体实现，负责如何创建一个新的编码器实例，用于下行数据到协议编码的转换，线程安全，可为null。
- com.aliyun.iot.as.bridge.server.channel.factory.TcpBasedProtocolAdaptorHandlerFactory: 协议适配器工厂类具体实现，负责如何创建一个新的协议适配器实例，用于解码器解析出来的数据到云端数据的转换，线程安全，不可为null。

- com.aliyun.iot.as.bridge.core.handler.DownlinkChannelHandler: 云端下行数据分发处理器，支持单播和广播。单播默认直接转发云端下发的数据到设备，广播需开发者自定义具体实现。可为null，为null时表示不支持下行。

启动Socket Server

创建BridgeServerBootstrap完成后，需调用com.aliyun.iot.as.bridge.server.BridgeServerBootstrap.start()完成Socket Server启动。

协议解码

协议解码组件需派生自io.netty.handler.codec.ByteToMessageDecoder，具体内容请参考*ByteToMessageDecoder*文档。

示例代码：

```
public class SampleDecoder extends ByteToMessageDecoder {
    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf in, List<
Object> out) throws Exception {
        // 解码协议
    }
}
```

协议编码

协议编码组件需派生自io.netty.handler.codec.MessageToByteEncoder<I>，具体内容请参考*MessageToByteEncoder*文档。

示例代码：

```
public class SampleEncoder extends MessageToByteEncoder<String>{
    @Override
    protected void encode(ChannelHandlerContext ctx, String msg,
ByteBuf out) throws Exception {
        // 协议编码
    }
}
```

协议适配器

为了降低开发成本，提高泛化协议设备接入与桥接服务的开发效率，泛化协议Server SDK为协议适配器提供了可扩展、可定制的基础能力类com.aliyun.iot.as.bridge.server.channel.CustomizedTcpBasedProtocolHandler。其封装了与阿里云物联网平台的对接细节，使您可以专注于协议本身的业务，无需关注具体的对接细节。泛化协议的协议处理适配器需派生自该类。

设备上线

当设备建立连接或者发送连接请求时，应触发设备上线操作。设备上线分为两个部分：网桥本地Session的初始化以及设备认证。

1. Session初始化

   具体内容请参见核心*SDK*>设备上线>*Session*初始化。

2. 设备认证

   本地Session初始化完成后即可调用doOnline(ChannelHandlerContext ctx, Session newSession, String originalIdentity, String... credentials)完成本地设备认证和阿里云物联网平台上线认证。认证成功，设备可以进行通信；认证失败，设备将断开连接。

   示例代码：

```
Session session = Session.newInstance(device, channel);
boolean success = doOnline(session, originalIdentity);
if (success) {
    // 上线成功，接受后续通信请求
} else {
    // 上线失败，拒绝后续通信请求，断开连接（如有）
}
```

设备下线

当设备连接断开或者网桥探测到需要断开连接时，应触发设备下线操作。Server SDK自身提供了设备连接断开时自动下线设备的能力，开发者只需负责处理主动触发设备下线操作的情形即可，下线某个设备可通过接口doOffline(Session session)实现。

上报数据

协议适配器需override channelRead(ChannelHandlerContext ctx, Object msg)方法，该方法是所有设备上行数据的入口，msg为decoder所返回的数据。

上报数据到物联网平台包括三步：获取上行数据关联的设备、找到设备关联的Session、上报数据到物联网平台。需通过以下接口完成数据上报：

- CompletableFuture doPublishAsync(Session session, String topic, byte[] payload, int qos)：异步发送数据并立即返回，开发者需根据future判断发送结果。
- CompletableFuture doPublishAsync(Session session, ProtocolMessage protocolMsg)：异步发送数据并立即返回，开发者需根据future判断发送结果。
- boolean doPublish(Session session, ProtocolMessage protocolMsg, int timeout)：异步发送数据并等待发送结果返回。

- boolean doPublish(Session session, String topic, byte[] payload, int qos, int timeout)：异步发送数据并等待发送结果返回。

示例代码：

```
DeviceIdentity identity = ConfigFactory.getDeviceConfigManager().
getDeviviceIdentity(device);
if (identity == null) {
    // 设备未成功映射到阿里云物联网平台上的设备，丢弃消息
    return;
}
Session session = SessionManagerFactory.getInstance().getSession(
device);
if (session == null) {
    // 设备尚未上线，请上线设备或者丢弃消息。上报数据到物联网平台前请务必确保设备已
上线。
}
boolean success = doPublish(session, topic, payload, 0, 10);
if(success) {
    // 成功上报数据到阿里云物联网平台
} else {
    // 上报数据到阿里云物联网平台失败
}
```

下行消息

具体请参考核心*SDK*>下行消息文档。

SDK提供了com.aliyun.iot.as.bridge.core.handler.DefaultDownlinkChannelHandler作为下行数据的分发处理器。支持单播和广播，其中单播默认直接转发云端下发的数据到设备，广播需开发者自定义具体实现。开发者可以通过派生子类实现行为定制。

示例代码：

```
import io.netty.channel.Channel;
import io.netty.channel.ChannelFuture;
...

public class SampleDownlinkChannelHandler implements DownlinkCh
annelHandler {
    @Override
    public boolean pushToDevice(Session session, String topic, byte[]
payload) {
        // 从设备对应的session中取出通信通道
        Channel channel = (Channel) session.getChannel().get();
        if (channel != null && channel.isWritable()) {
            String body = new String(payload, StandardCharsets.UTF_8);
            // 发送下行数据到设备
            ChannelFuture future = channel.pipeline().writeAndFlush(
body);
            future.addListener(ChannelFutureListener.FIRE_EXCEP
TION_ON_FAILURE);
            return true;
        }
```

```
        return false;
    }

    @Override
    public boolean broadcast(String topic, byte[] payload) {
        throw new RuntimeException("not implemented");
    }
}
```

5.3.3 UDP Server开发接口

基于泛化协议SDK的UDP Server开发接口，您可以快速构建以UDP为传输协议的接入服务，并将其与阿里云物联网平台桥接。

启动引导

com.aliyun.iot.as.bridge.server.BridgeServerBootstrap是启动Socket服务器和桥接服务的引导类。创建新的BridgeServerBootstrap实例后，会初始化基于配置文件的基本配置组件。

示例代码：

```
BridgeServerBootstrap bootstrap = new BridgeServerBootstrap(new
UdpDecoderFactory() {
    @Override
    public MessageToMessageDecoder newInstance() {
        // 返回解码器
    }
}, new UdpEncoderFactory() {
    @Override
    public MessageToMessageEncoder<?> newInstance() {
        // 返回编码器
    }
}, new UdpBasedProtocolAdaptorHandlerFactory() {
    @Override
    public CustomizedUdpBasedProtocolHandler newInstance() {
        // 返回协议适配器
    }
});
try {
    bootstrap.start();
} catch (BootException | ConfigException e) {
    // 处理启动失败
}
```

实例化UDP类型BridgeServerBootstrap

- com.aliyun.iot.as.bridge.server.channel.factory.UdpDecoderFactory: 协议解码器工厂类具体实现，负责如何创建一个新的解码器实例，用于上行数据的解析，线程安全，可为null。
- com.aliyun.iot.as.bridge.server.channel.factory.UdpEncoderFactory: 协议编码器工厂类具体实现，负责如何创建一个新的编码器实例，用于下行数据到协议编码的转换，线程安全，可为null。

- com.aliyun.iot.as.bridge.server.channel.factory.UdpBasedProtocolAdaptorHandlerFactory: 协议适配器工厂类具体实现，负责如何创建一个新的协议适配器实例，用于解码器解析出来的数据到云端数据的转换，线程安全，不可为null。

启动Socket Server

创建BridgeServerBootstrap完成后，需调用com.aliyun.iot.as.bridge.server.BridgeServerBootstrap.start()完成Socket Server启动。

协议解码

协议解码组件需派生自io.netty.handler.codec.MessageToMessageDecoder<I>，具体内容请参考*MessageToMessageDecoder*文档。

示例代码：

```
public class SampleDecoder extends  MessageToMessageDecoder<DatagramPacket> {
    @Override
    protected void decode(ChannelHandlerContext ctx, DatagramPacket in, List<Object> out) throws Exception {
        // 解码协议
    }
}
```

协议编码

协议编码组件需派生自io.netty.handler.codec.MessageToMessageEncoder<I>，具体内容请参考*MessageToMessageEncoder*文档。

示例代码：

```
public class SampleEncoder extends MessageToMessageEncoder<T>{
    @Override
    protected void encode(ChannelHandlerContext ctx, T msg, ByteBuf out) throws Exception {
        // 协议编码
    }
}
```

协议适配器

为了降低开发成本，提高泛化协议设备接入与桥接服务的开发效率，Server SDK为协议适配器提供了可扩展、可定制的基础能力类com.aliyun.iot.as.bridge.server.channel.CustomizedUdpBasedProtocolHandler。其封装了与阿里云物联网平台的对接细节，使您可以专注于协议本身的业务，无需关注具体的对接细节。泛化协议的协议处理适配器需派生自该类。

设备上线

当设备建立连接或发送连接请求时，应触发设备上线操作。设备上线分为两个部分：网桥本地Session的初始化以及设备认证。

1. Session初始化

   请参见核心*SDK*>设备上线>*Session*初始化。

2. 设备认证

   本地Session初始化完成后即可调用doOnline(Session newSession, String originalIdentity, String ... credentials)或doOnline(String originalIdentity, String... credentials)完成本地设备认证和阿里云物联网平台上线认证。认证成功，设备可以进行通信；认证失败，设备将断开连接。

   示例代码：

```
Session session = Session.newInstance(device, channel);
boolean success = doOnline(session, originalIdentity);
if (success) {
    // 上线成功，接受后续通信请求
} else {
    // 上线失败，拒绝后续通信请求，断开连接（如有）
}
```

设备下线

当设备连接断开或者网桥探测到需要断开连接时，应触发设备下线操作。泛化协议Server SDK自身提供了设备连接断开时自动下线设备的能力，开发者只需负责处理主动触发设备下线操作的情形即可，下线某个设备可通过接口 doOffline(Session session)实现。

上报数据

协议适配器需override channelRead(ChannelHandlerContext ctx, Object msg)方法，该方法是所有设备上行数据的入口，msg为decoder所返回的数据。

上报数据到物联网平台包括三步：获取上行数据关联的设备、找到设备关联的Session、上报数据到物联网平台。需通过以下接口完成数据上报：

- CompletableFuture doPublishAsync(String originalIdentity, String topic, byte[] payload, int qos)：异步发送数据并立即返回，开发者需根据future判断发送结果。
- CompletableFuture doPublishAsync(String originalIdentity, ProtocolMessage protocolMsg)：异步发送数据并立即返回，开发者需根据future判断发送结果。

- boolean doPublish(String originalIdentity, ProtocolMessage protocolMsg, int timeout)：异步发送数据并等待发送结果返回。
- boolean doPublish(String originalIdentity, String topic, byte[] payload, int qos, int timeout)：异步发送数据并等待发送结果返回。

示例代码：

```
DeviceIdentity identity = ConfigFactory.getDeviceConfigManager().
getDeviviceIdentity(device);
if (identity == null) {
    // 设备未成功映射到阿里云物联网平台上的设备，丢弃消息
    return;
}
Session session = SessionManagerFactory.getInstance().getSession(
device);
if (session == null) {
    // 设备尚未上线，请上线设备或者丢弃消息。上报数据到物联网平台前请务必确保设备已
上线。
}
boolean success = doPublish(session, topic, payload, 0, 10);
if(success) {
    // 成功上报数据到阿里云物联网平台
} else {
    // 上报数据到阿里云物联网平台失败
}
```

下行消息

暂不支持

6 扩展服务

6.1 固件升级

本章节主要介绍固件服务的控制台使用说明，帮助用户快速使用固件服务。

前提条件

- 已开通固件升级服务，如果未开通，登录物联网平台的控制台，选择**扩展服务**，单击固件升级下的**使用服务**。
- 设备端SDK已默认开启支持OTA升级服务。

背景信息

固件升级的步骤如下：

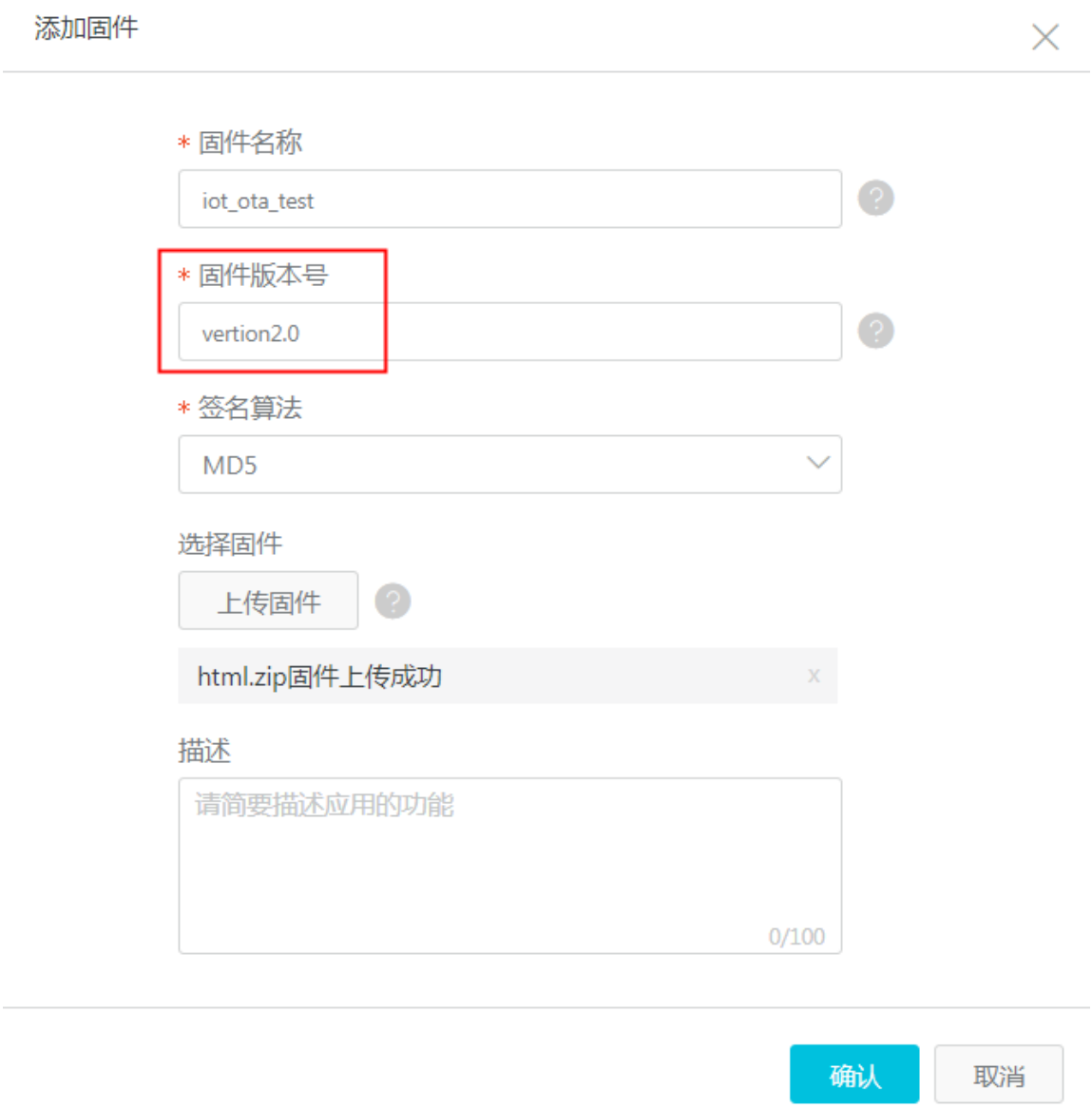
1. 添加固件
2. 验证固件
3. 批量升级
4. 再次升级

目前只有华东2节点支持固件服务。

操作步骤

1. 登录物联网平台的控制台。
2. 选择华东2区域节点。
3. 添加固件。
   a) 选择**我的服务** > **固件升级**。
   b) 在固件升级页面，单击**新增固件**，如图 *6-1:* 添加固件所示。

图 6-1: 添加固件



c) 设置固件参数。

- 上传的固件文件名称不能包含特殊字符，仅支持中文、英文字母、数字和下划线，长度限制1~32。
- 上传的固件文件大小不能超过10M。
- 用户最多能上传100个固件文件。

> **说明：**
> 已删除的也包括在100个固件文件内。

4. 验证固件。

   添加完固件后，必须先使用少量设备来验证固件是否可用。若验证固件可用，此固件才可以在大量设备上投入使用。

   在固件列表上选择某一固件，单击**验证固件**。

   - 验证固件会向MQTT接入的设备推送升级通知，在线设备会立即接收到升级通知，不在线的设备下次接入时，系统会再重新推送一次。

     其他接入方式（如CoAP或者HTTPS）的设备都是短连接的，所以不在线无法收到。
   - 可以反复发起验证固件，验证固件的本质就是指定少量设备进行升级。
   - 只要用户进行过验证固件操作，未验证的固件的状态就会被置为已验证，不会取决于设备的实际升级结果。
   - 验证固件操作结束，系统会记录下设备相应的升级操作记录（初始状态是待升级），只有设备收到升级通知后上传升级进度后，系统才会认为升级正式开始（升级操作记录状态更新为升级中）。升级开始后，可以在固件详情页，查看设备的升级进度信息。

5. 当进行过验证固件后，确认固件对设备是可用的，可以进行批量升级。在固件列表选择某一固件，单击**批量升级**。

   批量升级的本质是对大批设备定向推送升级通知。

   - 禁止使用未验证的固件进行批量升级操作。
   - 设备从收到升级通知开始直至升级完成是一个渐进的过程，OTA系统在收到设备主动上报的升级进度后更新设备升级进度百分比信息。
   - 批量升级所覆盖的设备可能会因为设备上一次的升级动作没有结束，导致本次升级中部分设备升级失败。
   - 设备在实际升级过程中出现错误，比如说下载失败、校验失败和解压失败等，并且通知给OTA系统后，系统会将本次升级动作置为完成，升级失败。
   - 可以在固件详情页，查看批量升级对应设备的升级情况，升级失败列表选项卡会显示简要的升级失败原因。

   批量升级任务冲突，同一产品下，创建批量升级时，如果选择的待升级版本号，已经处于另一个固件的批量升级过程中，则会提示升级任务冲突。

示例：同一产品下，某一个设备固件版本号为A，用户在控制台创建了两个固件，版本号分别为B和C。用户在控制台创建了固件B的批量升级，升级策略为从版本A到版本B。此时，如果用户想创建固件C的批量升级，升级策略为从版本A到版本C，则云端会触发升级任务冲突。

6. 若某些设备上一次升级失败，用户可以使用失败的升级操作记录尝试再次发起升级。

6.2 远程配置

前提条件

- 已开通远程配置服务，如果未开通，登录物联网平台的控制台，选择**扩展服务**，单击远程配置下的**开通服务**。
- 设备端SDK已开启支持远程配置服务。需要在设备端SDK中定义 `FEATURE_SERVICE_OTA_ENABLED = y`，SDK提供接口**`linkkit_cota_init`**来初始化远程配置（Config Over The Air，COTA）。

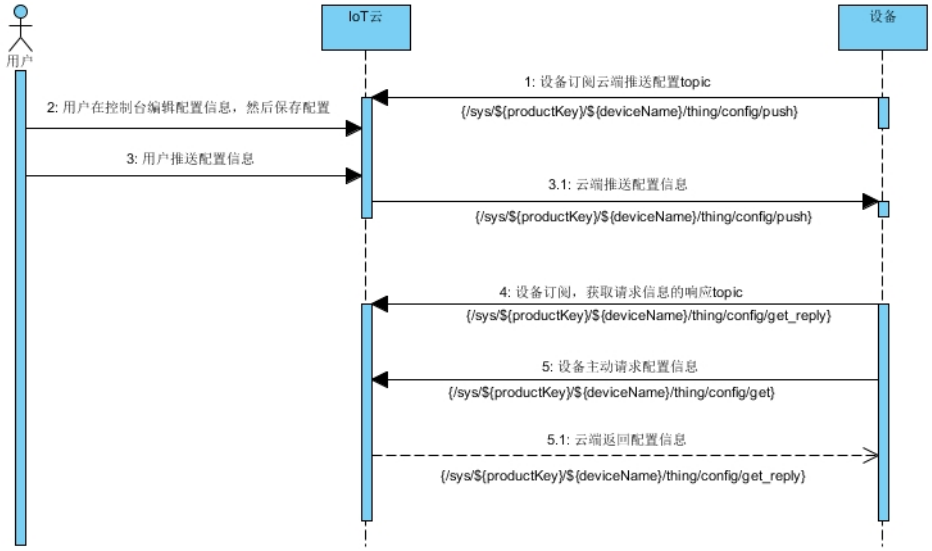
远程配置说明

很多场景下，开发者需要更新设备的配置信息，包括设备的系统参数、网络参数或者本地策略等等。通常情况下更新设备的配置信息是通过固件升级的方式完成的。这将加大固件版本的维护工作，并且需要设备中断运行以完成更新。为了解决上述问题，物联网平台为您提供了远程配置更新的功能，设备无需重启或中断运行即可在线完成配置信息的更新。

物联网平台提供的远程配置功能，可支持开发者：

- 开启/关闭远程配置；
- 在线编辑配置文件并支持版本管理；
- 向设备批量更新配置信息；
- 支持设备主动请求更新配置信息。

远程配置流程图如下所示：



远程配置大致分为三部分：

- 配置生成，您在IoT控制台编辑并保存配置信息；
- 配置使用，您在IoT控制台批量推送配置信息给设备，设备接收后修改本地配置文件；
- 主动请求，设备主动向云端请求新的配置文件并进行更新。

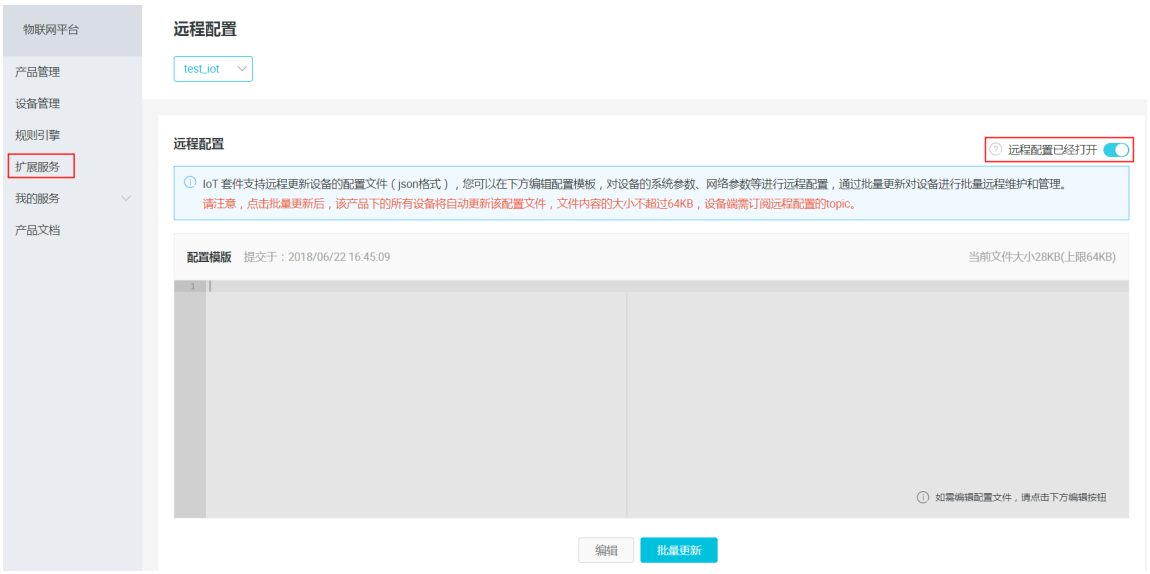
开启远程配置

开启远程配置分为两种场景，一种是云端下发配置信息给设备端的场景，一种是设备端主动查询配置信息的场景。根据场景的不同，开启远程配置的步骤也有所区别。

场景一

设备接收云端的配置信息，可按如下步骤进行远程配置。

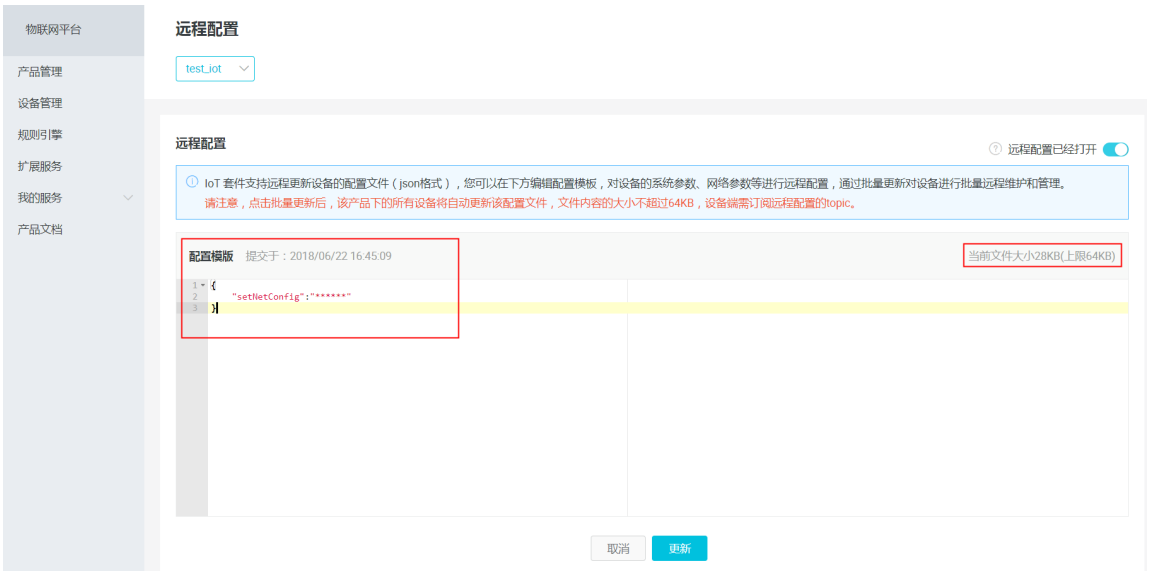
1. 设备上线，订阅推送配置信息的topic(/sys/${productKey}/${deviceName}/thing/config/push)。
2. 在IoT控制台中开启远程配置。
   a. 登录物联网平台的控制台，选择**扩展服务**。
   b. 单击**远程配置**，进入服务详情页面，单击**使用服务**。
   c. 选择产品，打开**远程配置**后的远程配置开关。



说明：

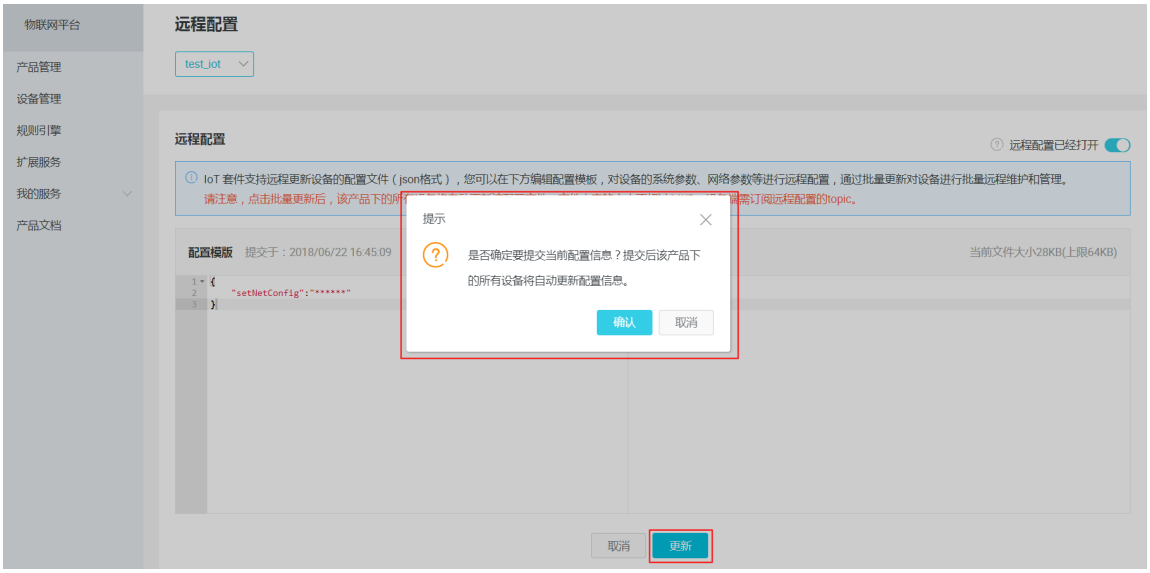
- 必须开启远程配置功能后，才可以编辑配置信息。
- 切换为关闭状态，即可关闭远程配置。

d. 在编辑区，单击**编辑**直接编写配置信息或拷贝配置信息到编辑区。产品配置模板适用于该产品下的所有设备，目前不支持对单个设备进行配置更新。

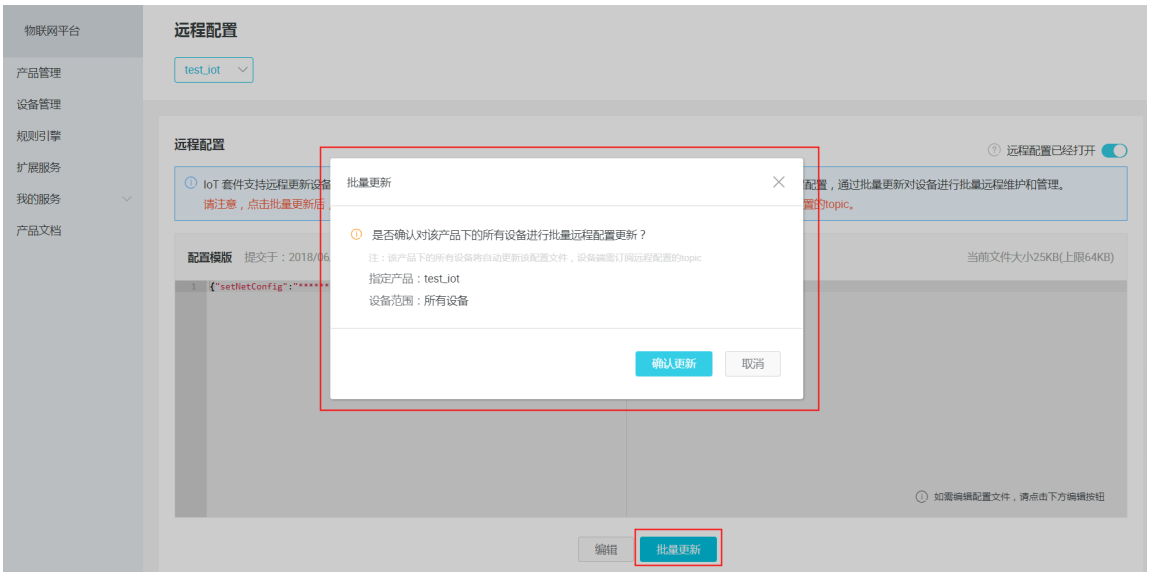


- 远程配置采用JSON格式，物联网套件对配置内容没有特殊要求，但系统会在提交时对内容进行JSON格式校验，避免错误格式引起配置异常；
- 配置文件最大支持64KB，编辑框右上角将实时显示当前文件的大小，超过64KB的配置文件无法提交。

e. 编辑完成配置信息后，需要单击更新，此时将生成正式的配置文件，允许设备主动请求更新该配置信息。



f. 配置文件提交后，云端不会立即向设备推送更新，需要单击批量更新，此时系统会向该产品下的所有设备批量推送配置文件更新。

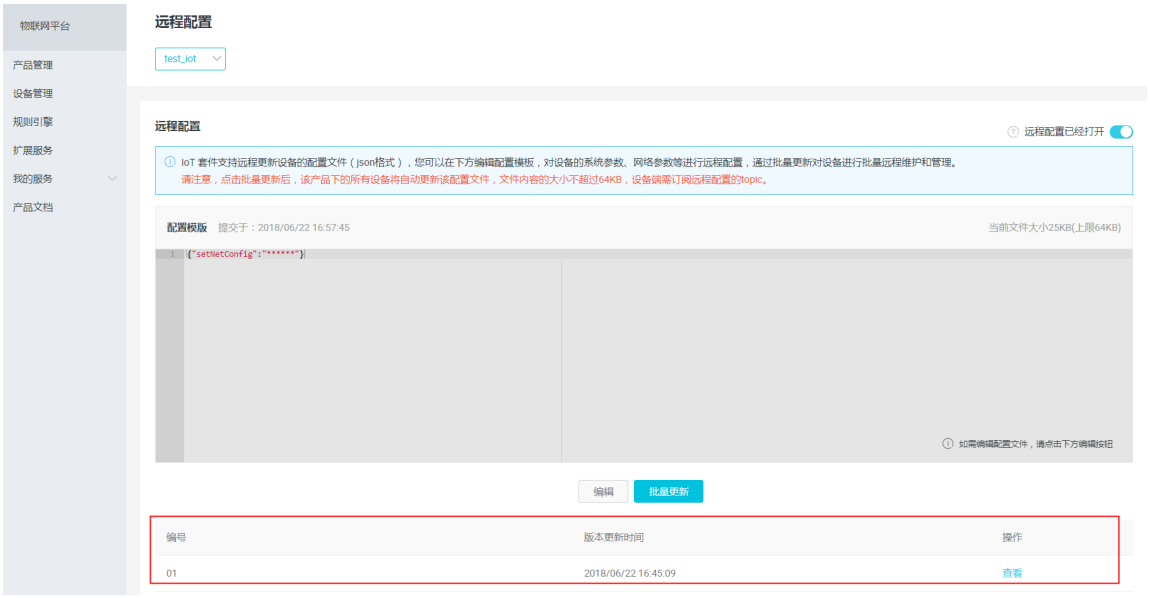


说明：

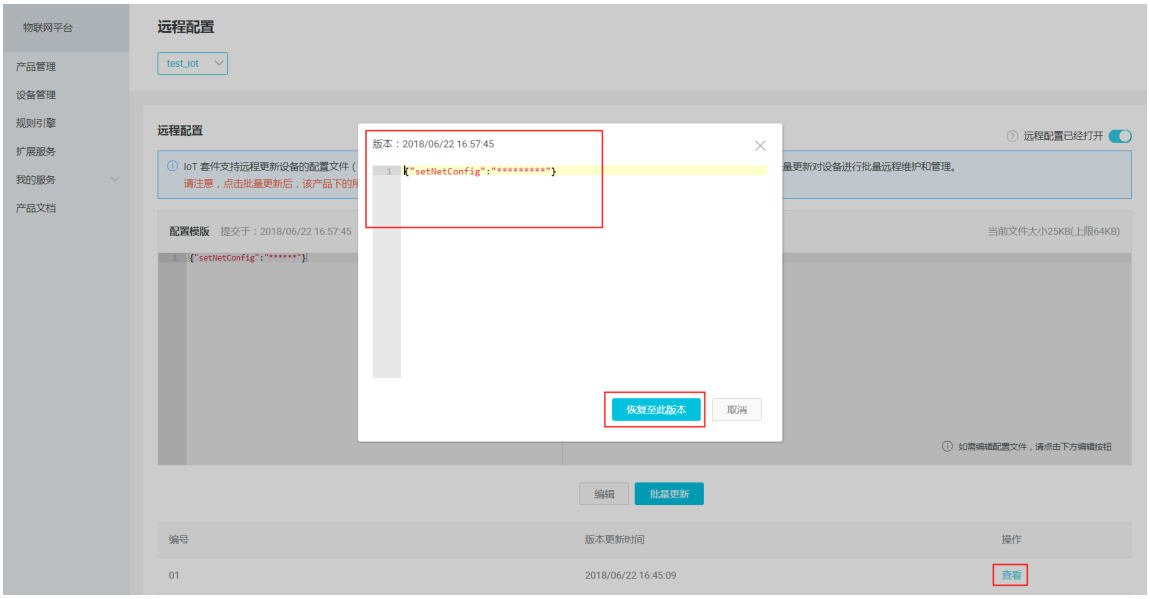
- 批量更新一小时内仅允许操作一次 ，请勿频繁操作；
- 如果您希望停止批量推送更新，单击远程配置开关，关闭远程配置，系统将停止所有更新推送，并且拒绝设备主动请求更新。

g. 可查看远程配置修改记录。

远程配置默认保存最近5次的配置修改记录，每次编辑并提交后，上一次配置将显示在版本记录列表中，您可以查看版本更新时间和配置内容，方便追溯。



在列表中单击查看，将显示该版本的配置内容，单击恢复至此版本，自动将所选版本中的内容恢复至编辑区中，您可以直接编辑修改并更新。



3. 设备端接收云端下发的配置信息后，自动更新配置。

场景二

在一些场景中，设备需要主动查询配置信息，可按如下步骤进行远程配置。

1. 设备端订阅topic(/sys/${productKey}/${deviceName}/thing/config/get_reply)。
2. 在IoT控制台中开启远程配置。详细步骤请参见2。

3. 设备端用户使用接口**linkkit_invoke_cota_get_config**来触发云端远程配置的请求。
4. 设备通过topic(/sys/${productKey}/${deviceName}/thing/config/get)，主动查询最新的配置信息。
5. 接收到设备的请求后云端会返回最新的配置信息到设备中。
6. 设备端用户在回调函数**cota_callback**中去处理远程配置下发的配置文件。

7 物联网平台通信模组

物联网通信模组

与阿里云物联网平台深度合作的通信模组，已具备连接阿里云物联网平台的能力，客户可以很方便进行应用场景业务开发。

广域网通信模组

表 7-1: 通信模组

厂商	型号	模组类型	模块链接	开发板链接	支持协议	物联网平台AT指令接口
合宙(Luat)	Air202	GPRS	点击查看	点击查看	MQTT+TLS	不支持
合宙(Luat)	Air800	GPRS	点击查看	点击查看	MQTT+TLS	不支持
合宙(Luat)	Air801	GPRS	点击查看		MQTT+TLS	不支持
有方(Neoway)	N10	GPRS	点击查看	点击查看	MQTT+TLS	
有方(Neoway)	N720	4G	点击查看	点击查看	MQTT+HTTPS	支持
移柯	L206	GPRS	点击查看		MQTT+HTTPS	支持
移远	EC20	4G	点击查看		MQTT+HTTPS	支持
美格	SLM750	4G	点击查看		MQTT+HTTPS	支持
	SIMCOM7000C	NB-IOT	点击查看	点击查看	MQTT, CoAP+Direct	支持
	YFM801D	4G			MQTT+Direct	支持