IoTHub v1.4 设计文档

注意

· 本文只解释一些和设备接入相关的关键功能解析，细节业务不涉及。

· 附文中仅仅给出技术方案指南，并不要求一定实现，或者一定要在iothub 实现。

架构

后端架构重新设计，新增了应用数据入口，抛弃了设备数据直接进 Nats 的做法。

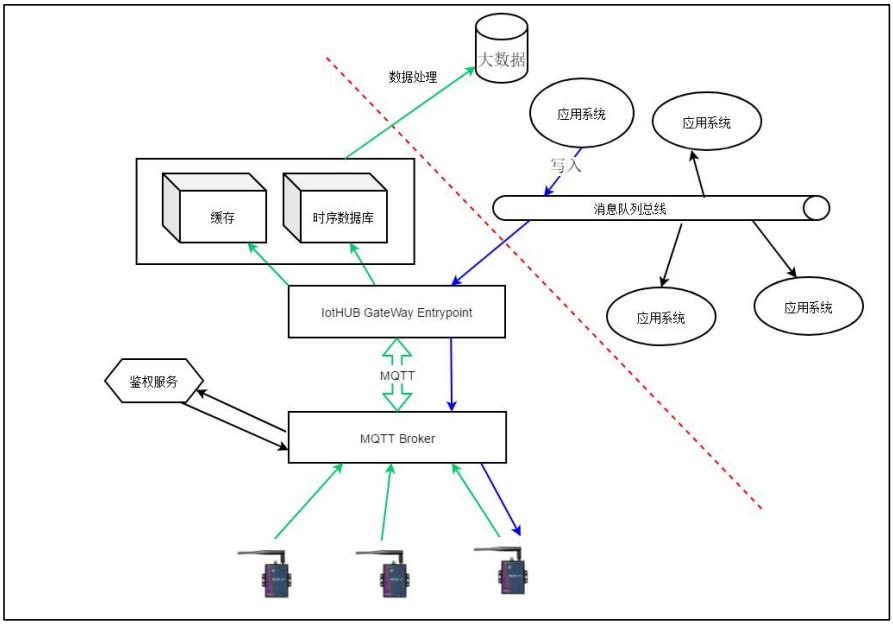


图 1

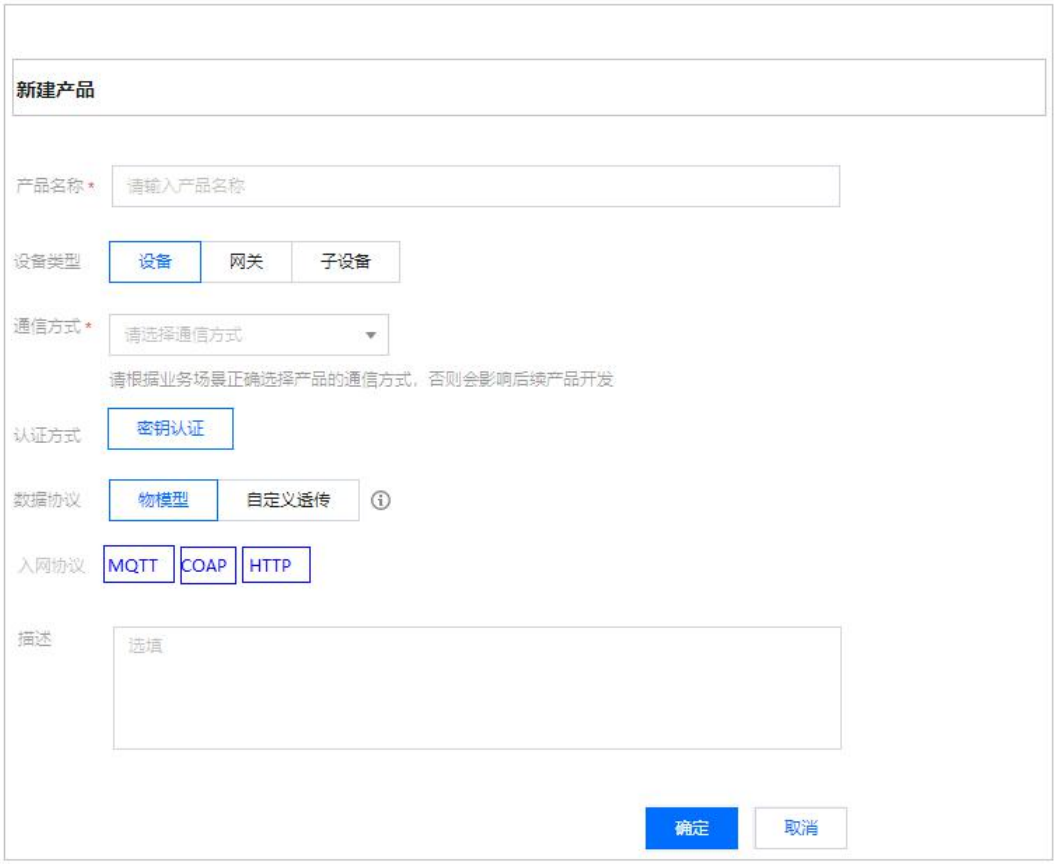
产品管理

新建

新建产品包含如下产品属性：

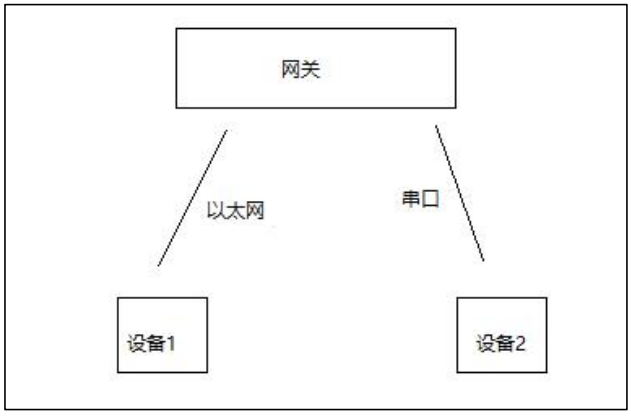
名称	类型	备注	必填
产品名称	String	不要超过10个字符不能出现*?/.[]$^%@符号，可以是中文	√
设备类型	Enum：网关、直连、子设备		√
通信方式	Enum：只有串口、以太网两种		√
认证方式	Enum：只支持秘钥认证一种形式		√
数据协议	Enum：物模型协议、透传协议（即无物模型转发类）		√
入网协议	Enum：只有 MQTT、COAP、HTTP		√
备注信息	随便填一些描述类信息	不要超过 256 个字符	×

原型界面效果如下：



新建产品数据仅仅是第一步，点击确认按钮后，同步还要做以下事情：

1. 如果是子设备，就需要配置整个拓扑结构，网关和子设备之间的关系如下：



根据通信方式对应的生成通信形式配置模板，配置模板如下

串口	以太网
波特率: 数据位: 停止位: 校验位: 校验类型:	IP: PORT:

2. 根据认证方式对应的生成秘钥，使用 HMAC256 非对称算法加密，具体的加密方式可由开发人员设计，所有的设备，秘钥都有 3 段组成:

名称	类型	含义
Username	String	入网用户名
Password	String	入网密码
Clientid	String	入网设备 ID

3. 根据入网方式对应的生成入网配置模板，具体的配置模板如下

名称	类型
MQTT	只支持 V3.1 和 V5.0 两个版本
CoAP	只支持单向 POST
HTTP	只支持单向 POST

lMQTT

订阅: 设备上线监控: $SYS/brokers/+/clients/+/connected
订阅: 设备下线监控: $SYS/brokers/+/clients/+/disconnected
订阅: 设备属性上报: $thing/up/property/${ProductID}/${DeviceID}
订阅: 设备事件上报: $thing/up/event/${ProductID}/${DeviceID}
属性下发: $thing/down/property/{ProductID}/{DeviceID}
动作下发: $thing/down/action/{ProductID}/{DeviceID}

举例：

假设此时网关(产品 ID=GW001)1 号（ID: smzx_sensor_gw_001）的数据上来了，则 MQTT 的 Topic 是:

$thing/up/property/GW001/smzx_sensor_gw_001

lCOAP

属性上报: POST: http://coap.xxx.com/property/{ProductID}/{DeviceID}
事件上报: POST: http://coap.xxx.com/event/{ProductID}/{DeviceID}

lHTTP

属性上报: POST: http://coap.xxx.com/property/{ProductID}/{DeviceID}
事件上报: POST: http://coap.xxx.com/event/{ProductID}/{DeviceID}

注意：只有${***}才是变量，其他的就是字面含义；${ProductID}是产品 ID，${DeviceID}是具体的设备 ID。

所有协议统一上传的数据格式

```
{
    "method":"report",
    "id":${uuid},
    "timestamp":1628106783,
    "data":{ 对应物模型的字段 }
}
```

上面是背后生成的数据模板，最后需要体现在 UI 界面上：

除了路由以外，产品详情也需要体现出来：

搜索

名称	类型	备注	必填
模糊搜索	String	新增搜索框能搜索到任何字段及字段下的的数据信息	
精确搜索	String	产品的 ID 需要精确搜索	

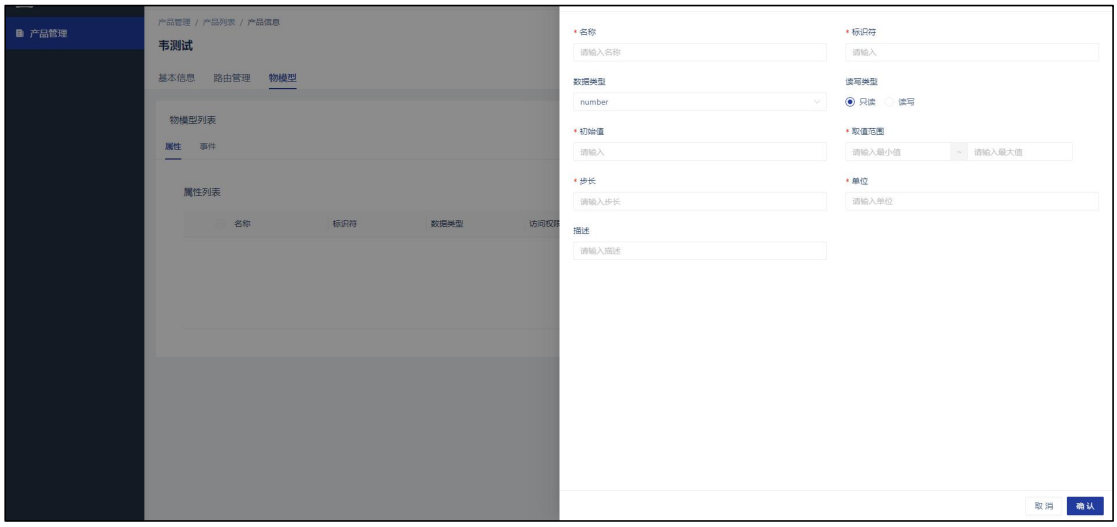
去除模块列表

名称	类型	备注	必填
去除模块列表	String	产品信息页里去掉模块列表，右边的组件块宽度铺满	×

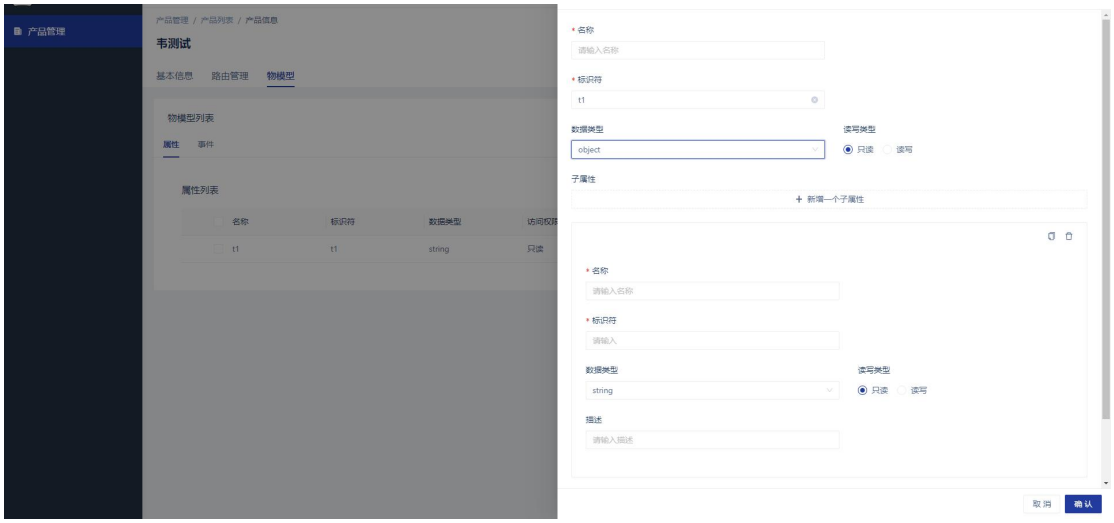
物模型

参考规范：https://www.w3.org/TR/wot-thing-description/#introduction

上面是介绍了创建一个产品以及背后做的事情，接下来看下最核心的物模型。



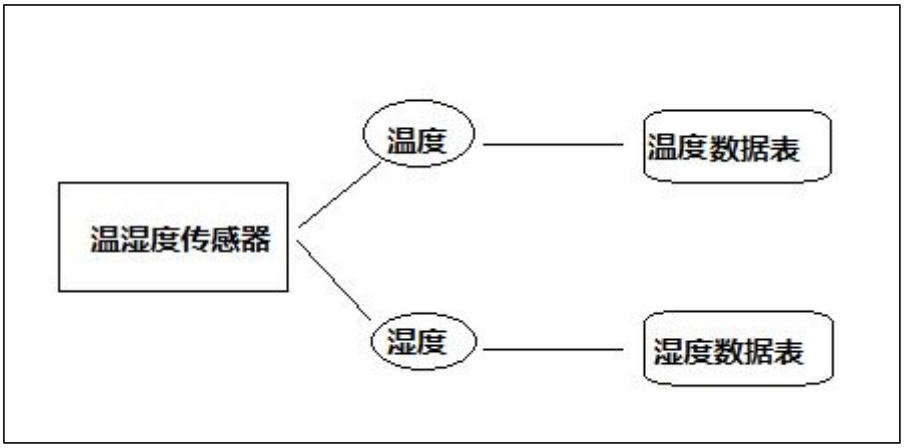
属性、命令、事件是最重要的 3 个功能。其中属性的字段类型要支持嵌套类型：



物模型创建成功以后，**马上需要把物模型的 schema 映射到时序数据库里**，也就是伴随着物模型的创建，时序数据库的建表动作也完成了。

建表规则：一个产品可以对应多个表，复合结构用一个表，单独属性用一个表。在时序数据库里面，表就是一等公民（一个产品有成千上百属性的时候，可能对应的海量表，其实是正常设计）：

新建物模型界面



物模型和实物映射

展示一个 8 路控制器物模型:

```
{
    "version": "1.0",
    "properties": [
        {
            "id": "switchers",
            "name": "继电器开关状态",
            "desc": "控制器的开关",
```

```
            "mode": "rw",
            "define": {
                "type": "struct",
                "specs": [
                    {
                        "id": "sw1",
                        "name": "sw1",
                        "dataType": {
                            "type": "bool",
                            "mapping": {
                                "0": "关",
                                "1": "开"
                            }
                        }
                    }
                ]
            },
            "required": false
        }
    ],
    "events": [
        {
            "id": "status",
            "name": "状态上报",
            "desc": "",
            "type": "info",
            "data": [
                {
                    "id": "sw1",
                    "name": "sw1",
                    "define": {
                        "type": "bool",
                        "mapping": {
                            "0": "关",
                            "1": "开"
                        }
                    }
                }
            ],
            "required": false
        }
    ],
    "actions": [
        {
```

```
            "id": "control",
            "name": "控制",
            "desc": "",
            "input": [
                {
                    "id": "sw1",
                    "name": "sw1",
                    "define": {
                        "type": "bool",
                        "mapping": {
                            "0": "关",
                            "1": "开"
                        }
                    }
                }
            ],
            "required": false
        }
    ],
    "configurations": [
        {
            "id": "config",
            "name": "串口配置",
            "desc": "串口配置",
            "mode": "rw",
            "define": {
                "type": "struct",
                "specs": [
                    {
                        "id": "baud_rate",
                        "name": "baud_rate",
                        "dataType": {
                            "type": "int"
                        }
                    },
                    {
                        "id": "parity",
                        "name": "parity",
                        "dataType": {
                            "type": "int"
                        }
                    }
                ]
            },
```

```
                "required": false
            }
        ],
        "profile": {
            "ProductId": "Y0ST19XLP1",
            "CategoryId": "1"
        }
}
```

注意:

物模型的属性、事件、动作总数都不能超过 10（*该值可配置，1.4 暂定 10，后期如果满足不了复杂场景可增加，通常也不会推荐有超过 30 个的属性 如果超了说明设计不合理 需要拆分*）个。

属性类型约束

属性包含如下关键字段

名称	类型	备注	必填
名称：name	String		√
类型：type	info：信息 alert：告警 fault：故障		√
备注：description	String		×
标识符：identify	String	事件 ID	√

物模型导入导出

需要支持物模型 JSON 的导入、导出功能。

Bool 类型

这是逻辑类型，只有 0、1 两个值，其中需要用户输入自定义 label，无单位，只有读写权限。

Date、Text

两类属于文本类型，不具备数值和 bool 类型的特性，所以其只有读写权限。

Object 类型

Object 对应的实际上是个结构体类型，该类型最多拥有 10 个属性。

Array 类型

Array 本质上对应的是数组，其子元素只能有相同的类型（数组的定义），该类型最多拥有 10 个子元素。数组的子元素只可新增、删除，不可修改。

输入条件

名称	初始值	步长	说明
int32	0	最小 1	注意：初始值、步长这些一般针对的是数值类型。文本类的没有该说法；所有的数值类型默认值全部是 0
float	0	最小 0.1	
double	0	最小 0.1	
bool	0（false）	无	
text	无	无	
date	无	无	
object	无	无	
array	无	无	

注意：上面的文档描述的是国内的一些规范，而本项目采用的规范是 W3C 规范。因此字段名称有不同，但是仅仅也是名称不同，其本质上是一样的。

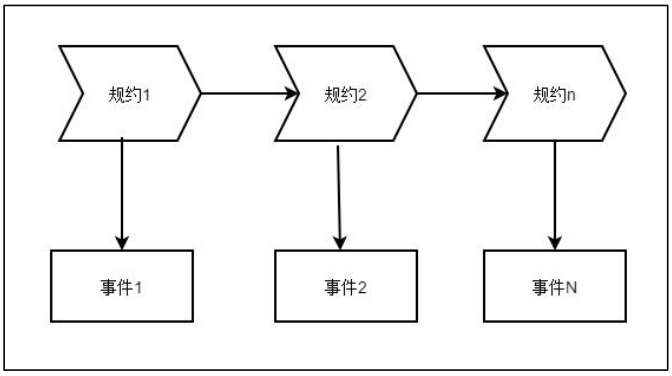
事件约束

事件是客户端上报的一些反馈自身硬件以及业务工作状态的描述信息。本次设计简化规范事件，**将规范事件的内容统一简化成文本**。事件的关键字段如下表：

名称	类型	备注	必填
事件名：name	String		√
事件类型：type	info：信息 alert：告警 fault：故障		√
事件内容：data	String	事件描述文本，例如“低电压”	√
备注：description	String		×

identify	String	事件 ID	√

注意：为什么简化事件？因为按照完整的设计思路，这个事件会有 2 个生产者：设备和平台。平台需要实时处理上传的数据，然后根据物模型定义的那些阈值或者规约来显示这些数据的限制权限，从而产生事件例如设备发送上来一个温度：100 °，设备并不知道阈值，因为只有服务器才能从物模型里提取。所以此时需要有一套完善的数据流处理机制。但是这套机制是极其复杂的，短时间内实现起来还是有一定难度，因此对此模型做了简化。
简化后我们默认就把数据生产者当成设备了【并且保留标准的时间模型】，设备自己上报事件即可，这样就不用走规则流判断，虽然简化了这个过程，但是随之而来的就是物模型无法发挥其作用。



事件流程

示例事件数据

```
{
    "method":"event",
    "id":"123",
    "eventName":"PowerAlarm",
    "type":"fault",
    "timestamp":1212121221,
    "data": “机器人低电压”
}
```

动作约束

经过调研，大致上可以确定我们这期或者说长期内暂时不实现 Action 里面的东西。属性足够满足大部分场景。

示例数据：

```
{
    "method": "action",
    "id": "20a4ccfd-d308",
    "actionId": "openDoor",
    "timestamp": 1212121221,
    "data": { 。 。 。 。 }
```

```
}
```

注意：Action 先留空不实现。因为动作相当于是将节点当成服务来用，适用于边缘计算方面，我们现阶段暂时没这方面的需求。

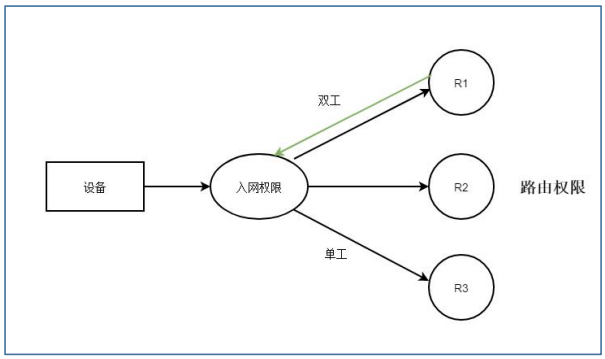
设备管理

新建

新建设备的时候需要指定产品，还需要增加自定义 Tag，Tag 遵守 K-V 格式，用来扩展外部业务。**新建的设备相当于一个产品的实例，需要把定义从产品继承过来，但是值填写自己的。**

认证

设备的认证也是非常重要的一部分，绝大部分设备包含了 2 个认证：**入网许可和路由许可**，也就是通常所说的 Auth 和 ACL。
为了统一认证，本设计将所有的认证字段抽象成 3 段：Username、password、Clientid。分别用来标识入网能力（Username、password）、路由能力（Clientid、ACL）。
路由能力有 2 类：上行和下行，抽象成 up 和 down 两类，但是每个协议不同，比如 MQTT 支持双向、可以同时支持 up 和 down，但是 HTTP 只允许单向，因此只支持 down，其关系如下图所示:



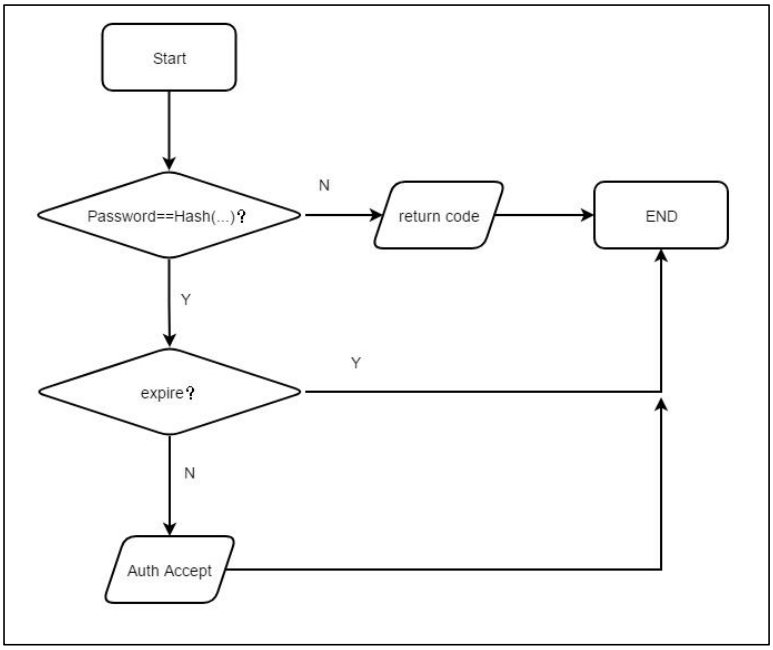
设备上下行数据流程

目前我们选择使用秘钥认证，也就是要对 username、password、clientid 做一套算法来实现设备的认证行为。具体的算法设计本文不阐述，具体实现由任务中实现。这边给出一个极其简单的加密设计设计仅供参考:
新建设备时，自动生成 username、password、clientid、expire-time，private-key 等参数。

Password = Hash(client-id#expire-time, username, private-key)

- ***client-id：设备唯一 ID, 32 位 Md5 字符串***
- ***username 登录用户名, 32 位 Md5 字符串***
- ***expire-time：设备有效期，是个毫秒级时间戳，如果是 0 则表示永不过期***
- ***private-key：存在数据库里的私钥, 32 位 Md5 字符串***

需要注意：username 和 expire-time 是用‘#’连接在一起的!



设备认证流程图

当前阶段，通信能力暂时分上行和下行，**因为时间关系以及人力不足关系，同步消息暂时不实现**。

上行消息很好理解，就是设备直接传上来的数据，设备上报数据分为数据和事件。数据类型的上报消息其格式如下:

```
{
    "method":"report",                  // 固定为 “report”
    "id":${uuid},                       // 用来追寻请求的自定义字段，可设置为随机数字
    "timestamp":1628106783,             // 时间戳
    "data":{ 对应数据物模型的字段 } // 物模型的数据
}
```

事件类的消息格式如下:

```
{
    "method":"event_post",                  // 固定为 “event_post”
    "id":${uuid},                           // 用来追寻请求的自定义字段，可设置为随机数字
    "timestamp":1628106783,                 // 时间戳
    "data":{ 对应事件物模型的字段 } // 物模型的数据
}
```

不难看出，事件只是一类特殊数据罢了。

下行消息是通过外部接口给下面连接的设备发送消息，分属性下发和动作下发。**现阶段只有 MQTT 协议支持下行消息**，下行消息的 Topic:

```
属性下发: $thing/down/property/{ProductID}/{DeviceID}
动作下发: $thing/down/action/{ProductID}/{DeviceID}
```

下行消息的格式如下:

```
{
    "method":"control",                    // 固定为“control”
    "id":${uuid},                  // 用来追寻请求的自定义字段，可设置为随机数字
    "timestamp":1628106783,          // 时间戳
    "data":{ 对应物模型的字段 } // 物模型的数据
}
```

区别是“method”：上行为“report”，下行为“control”。

外部接口

外部应用可以通过 HTTP 或者 GRPC 接口来调用 entry-point 服务，从而实现下发指令。外部系统可以发送两类消息：

1. 设备的属性（property）
2. 设备的动作（action）

外部系统请求 entry-point，必须包含如下参数：

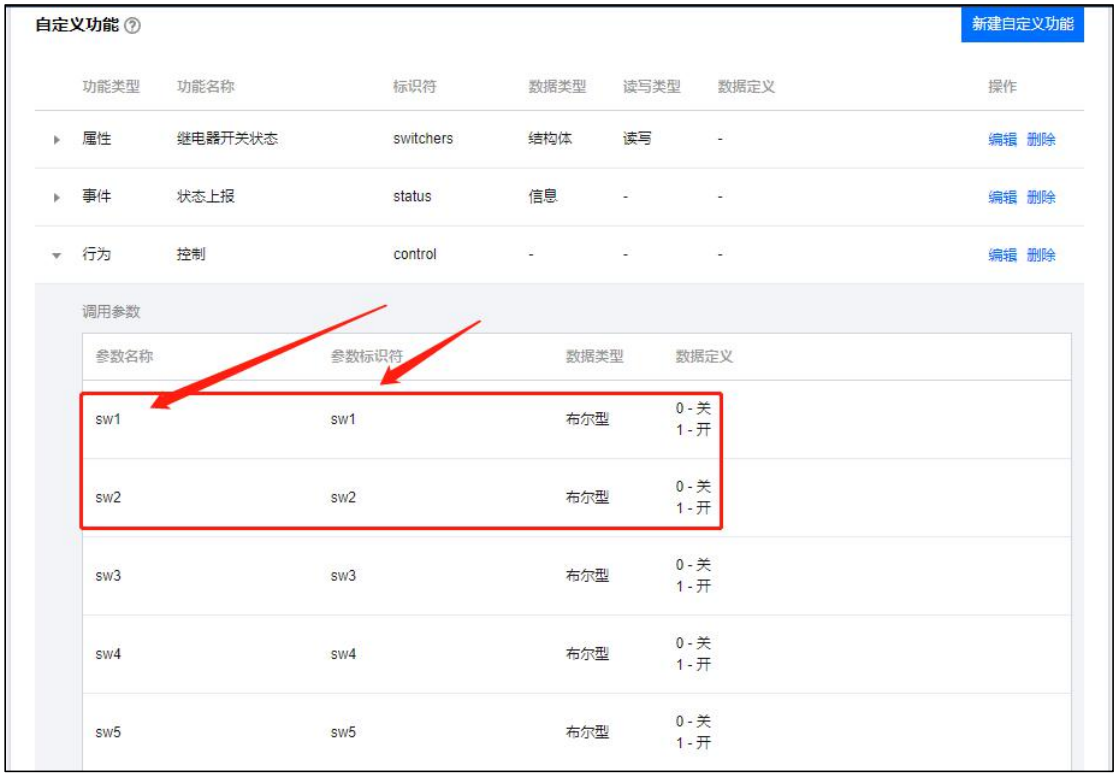
名称	类型	备注	必填
deviceId	String	设备的 SN 序列号，表示要给某个设备发送数据	√
data	String	发送给设备的数据内容，这个数据结构对应设备的物模型规定的字段	√
id	String	唯一请求 ID，用来做请求识别	√
productId	String	产品 ID	√
method	String	消息类型只有两个值:“property”、“action”	√

当 Entry-point 收到数据以后，需要将其转发到下面的 MQTT Topic，才能到最终设备：

$thing/down/${method}/{ProductID}/{DeviceID}

消息内容：

```
{
        "method": "${method}",
        "id": "20a4ccfd-d308",
        "actionId": "动作名",    // 只有 Action 时才有该字段
        "timestamp": 时间戳,
        "data": {
            "a": "1"     // 对应的物模型 actionId 动作的参数
        }
}
```



设备收到这条指令后，会向下面的 Topic 发送一个回复消息：

$thing/up/${method}/{ProductID}/{DeviceID}

回复内容是：

```
{
        "method":"${method}_reply",
```

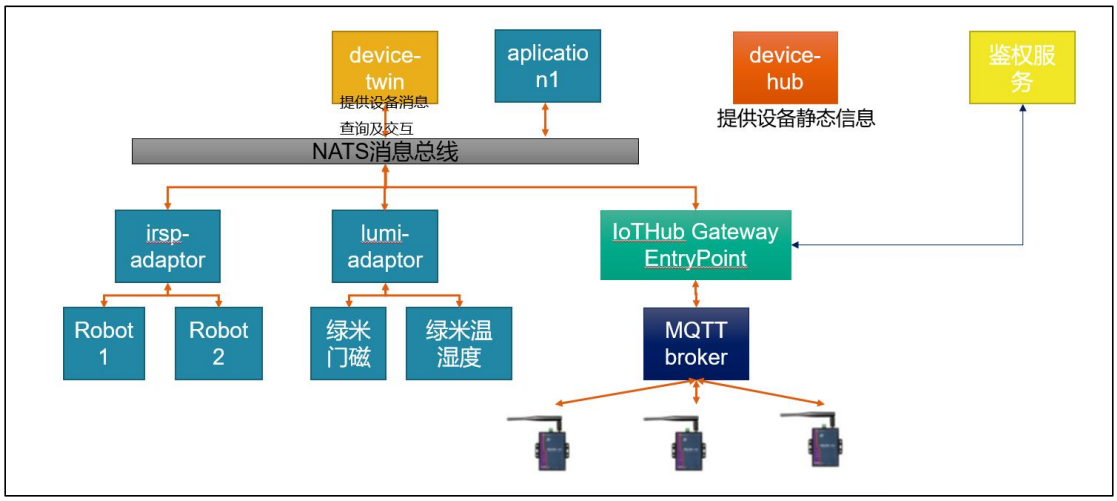
```
        "id":"20a4ccfd-d308",
        "code": 0,
        "status":"some message"
}
```

当 Entry-point 收到包含该请求 ID 的时候认为通信成功。

注意：该接口需要用公共权限过滤机制。

IRSP 协议设备通讯架构

device-hub 提供设备静态消息。
device-twin 提供设备动态消息的查询及交互入口，
adaptor 适配规整设备消息并接入设备总线 nats。



机器人 IRSP 协议类设备消息定义

设备消息分为以下几种：

状态	state	设备直接产生的，不可更改的信息
事件	event	由原始信息产生的二次信息，旨在强调或引起注意的消息
属性	property	可以被配置，被改变的设备属性
命令	command	应用服务主动下发给设备，设备必须回应的消息
回复	reply	设备回复应用程序的消息

nat 格式定义

nats 与设备服务交互的桥梁，写入 nats 的设备消息必须是格式定义且明确的。
nats 的消息定义格式 详见 git 文档：
《https://git.infore-robotics.cn/smart-cloud/docs/-/blob/develop/tech/%E8%AE%BE%E5%A4%87%E9%80%9A%E8%AE%AF%E6%A0%BC%E5%BC%8F%E5%8F%8Anats%E6%B6%88%E6%81%AF%E8%A7%84%E5%88%99%E5%AE%9A%E4%B9%89.md》

twin 文档说明

twin 是物理设备的数字抽象，提供与设备交互入口。
twin 提供以下两种类型接口：
1. 三种设备上行消息的(state event property)的四种交互方式(get,watch,search,getHistory)
2. 一种下行消息(command) 的两种(set, getHistory) 交互方式。

详细接口定义：

序号	说明	接口
1	获取设备状态	GetStates(GetStatesReq) returns (GetStatesRsp)
2	下发命令	Command(CommandReq) returns (stream CommandRsp)
3	监听状态	WatchStates(WatchStatesReq) returns (stream WatchStatesRsp)
4	搜索状态	SearchStates(SearchStatesReq) returns (SearchStatesRsp)
5	获取设备消息	GetDeviceMessages(GetDeviceMessagesReq) returns (GetDeviceMessagesRsp)
6	搜索设备消息	SearchDeviceMessages(SearchDeviceMessagesReq) returns (SearchDeviceMessagesRsp)
7	获取历史设备消息	GetHistoryMessages(GetHistoryMessagesReq) returns (GetHistoryMessagesRsp)
8	订阅设备消息	WatchMessages(WatchMessagesReq) returns (stream WatchMessagesRsp)
9	获取历史设备命令	GetHistoryCommands(GetHistoryCommandsReq) returns (GetHistoryCommandsRsp)

详细 proto 接口定义与约束参见如下
《 https://git.infore-robotics.cn/smart-cloud/arch-protos/-/blob/master/platform/twin/twin.proto》

约束

除了上面的基础，还有一些原则上的约束条件：

1. 新建产品后不可修改关键信息，只能改基础备注之类的基础数据，当产品有设备以后禁止修改其物模型，如果修改物模型会引起设备版本不一致；
2. 产品删除的时候如其包含设备则不可删除；
3. 产品、设备名称均支持 I18N，最长不可超过 10 个字符；
4. 修改、删除设备需要检查其是否正在运行，如果运行则不允许删除、修改。如果要删除运行产品，首先需要将其停止或者移除到回收站；
5. 网关类设备包含子设备也不可删除；
6. 设备认证信息应该用包含时间戳的加密算法生成

结束

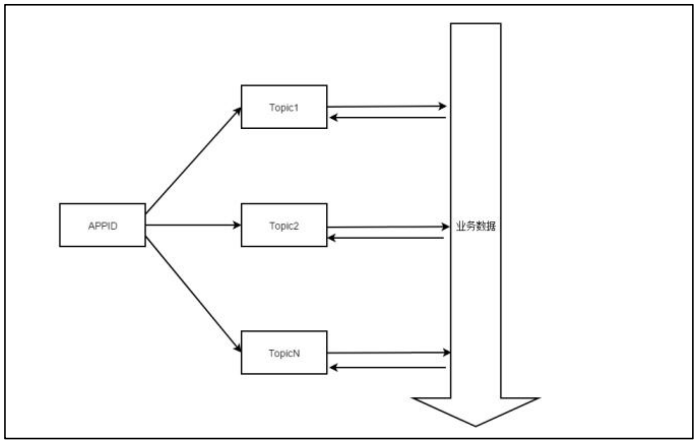
到此为止，所有关于设备接入相关的全部介绍完毕。

附 1：APP 机制

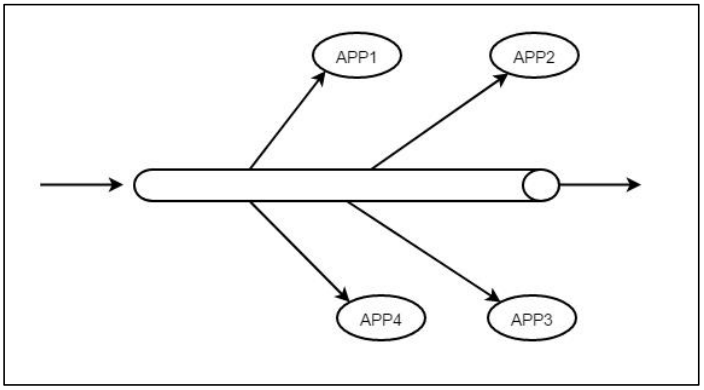
现阶段的应用层和总线接入形式是很混乱而且没有做过滤的，随便来个都能从 Nats 里拉数据，就和下面一样：

```
signal.Notify(c, syscall.SIGINT, syscall.SIGABRT)
connection, err1 := nats.Connect("emqx.dev.inrobot.cloud:4222", func(o *nats.Options) error {
    o.User = "nats_client"
    o.Password = "Pwa43zr2kS"
    return nil
})
if err1 != nil {
    log.Fatal(err1)
    return
}
_, err2 := connection.Subscribe("iothub.upstream.>", func(m *nats.Msg) {
    fmt.Println(m.Subject, "==> "+string(m.Data))
    m.Ack()
})
```

IoTHUB 包含了数据的应用目的地，因此需要一个纯净的应用消费机制，为了解决这个问题，在 NATS 之上做一层设计：即使用 NATS 做业务层总线，每个上层服务称之为一个应用（app）。服务要接入总线需要认证权限，不可随便乱连接，防止消息串流。每个应用有自己的 ID 和自己的分流权限，是一对多的关系，通过 APP 管理中心来控制。



消息转发



总线机制

优化后看起来应该和下面这个效果一样：

```
// APPID: NHOGYHLGrt970086IBHNJL
```

```
func main() {
	c := make(chan os.Signal, 1)
	signal.Notify(c, syscall.SIGINT, syscall.SIGABRT)
	app := goodbus.NewApplication("a-demo-app")
	if err := app.Auth("127.0.0.1:4222", "nats_client", "Pwa43zr2kS"); err != nil {
		panic(err)
	}
	app.SetErrorHandler(func(err error) {
		fmt.Println(err)
	})
	if err := app.JoinChannel("mychannelabcd", func(msg goodbus.Message) {
		fmt.Println("Received:", msg.String())

	}); err != nil {
		panic(err)
	}
	signal := <-c
	fmt.Printf("Received stop signal:%v", signal)
	os.Exit(1)
}
```

这当然是个理想的设计, 希望大家在做这类设计的时候有意识的考虑到合理性以及安全性等因素，对技术方案有评估过程。

附 2：任务列表

下面的任务列表为 1.4.0 一期。具体的细节内容可由项目经理专门细化分配，例如新建产品包含了新建物模型等，此处仅列出关键大项。

任务名	备注
新建产品	
更新产品	
搜索产品	
删除产品	
新建设备	
更新设备	
搜索设备	
删除设备	
设备入网认证	
设备路由认证	

时序数据和物模型映射关联	将物模型字段和 TSDB 的 SQL 进行关联映射建表，举例：
	{
	“temp” :{“type” : float}
	“hum” :{“type” : float}
	}
	映射成 SQL:
	Create table (
	`temp` float
	`hum` float
	)

附 3：MQTT 连接

内网 HOST: (172.16.0.11, 172.16.0.12, 172.16.0.15):1883
公网 HOST: 42.193.180.26:1883

下面这个账户是开了白名单的，用来给【iothub-entry-point】服务使用:
ID: iothub-entry-point-{唯一 ID}
用户: iothub-entry-point
密码: iothub-entry-point

注意：公网是给设备连接用，内网是给应用连接用。

附 4：TdEngine 指南

TDengine 是一款高性能、分布式、支持 SQL 的时序数据库 (Database)，其核心代码，包括集群功能全部开源（开源协议，AGPL v3.0）。TDengine 能被广泛运用于物联网、工业互联网、车联网、IT 运维、金融等领域。除核心的时序数据库 (Database) 功能外，TDengine 还提供缓存、数据订阅、流式计算等大数据平台所需要的系列功能，最大程度减少研发和运维的复杂度。
TDengine 在 Iothub 的应用：联通物模型、数据、设备，建立设备的时序数据存储系统。

超级表

超级表可用来建立设备的**物模型。**回顾一下上面的设计，我们在业务层做了物模型，其实 TDEngine 直接在数据库层面就支持上了，对应的概念是超级表。下面这个是一个最简单的 Demo 展示：

```
CREATE STABLE 产品 (
    ts timestamp,
    current float,
    voltage int,
    phase float
) TAGS (
    location binary(10)
);
```

具体的**设备**可以关联到超级表子表：

```
CREATE TABLE 设备_SNbOJHBHJLN USING 产品 TAGS ("深圳联合金融大厦");
```

插入数据:

```
INSERT INTO 设备_SNbOJHBHJLN USING 产品
TAGS ("深圳联合金融大厦")
VALUES (now, 10.2, 219, 0.32);
```

例如市民花园传感器，物模型包含 3 字段：
- current float,
- voltage int,
- phase float

映射到时序数据库，本质上是个 SQL 建表语句：

```
CREATE STABLE meters (
      ts timestamp,
      current float,
      voltage int,
      phase float
)TAGS (产品的 TAG);
```

数据类型

我们在 IOTHUB 创建物模型的时候已经指定了属性的类型，和 Tdengine 映射的时候有个疑问：
我们既然说了，每个属性一个表，那么有些属性看起来可以合并的时候怎么办呢？
有个原则：
1. 默认通用情况是所有属性每个字段一个超级表；
2. 当字段类型为 struct、object 类的时候，合并成一个超级表，而不用拆分其属性；
3. 尽可能的认为所有属性都应该是单独的表，struct 和 object 仅仅是特殊情况。

举个例子：

某传感器具备温湿度、CO2 等参数，看起来他是一个传感器的数据，但是实际上他们是不同的指标参数，相互之间没有任何关联，所以他们是每个属性一个超级表；

同样的有个 GPS 设备，他传上来的数据有经纬度、海拔，他们单独来说是没有意义的，因此必须合起来才能表达一个空间地理坐标，因此此时这个 GPS 设备三个属性合并成一个超级表来描述

附 5：Topic 规范

上报 topic

```
$thing/up/event/{ProductID}/{DeviceID}
$thing/up/property/{ProductID}/{DeviceID}
$thing/up/action/{ProductID}/{DeviceID}
```

下发 topic

```
$thing/down/property/{ProductID}/{DeviceID}
$thing/down/action/{ProductID}/{DeviceID}
```

回复 Topic

```
$thing/property/reply/{ProductID}/{DeviceID}
$thing/action/reply/{ProductID}/{DeviceID}
```

属性下发

Topic：$thing/down/property/{ProductID}/{DeviceID}

```
{
    "method": "property",
    "id": "20a4ccfd-d308",
    "timestamp": 时间戳,
    "data": {
        "a": "1"
    }
}
```

动作下发

Topic：$thing/down/action/{ProductID}/{DeviceID}

```
{
```

```
    "method": "action",
    "id": "20a4ccfd-d308",
    "actionId": "动作名",                    // 只有 Action 时才有该字段
    "timestamp": 时间戳,
    "data": {
        "a": "1"                              // 对应的物模型 actionId 动作的参数
    }
}
```

属性 回复消息体

Topic：$thing/property/reply/{ProductID}/{DeviceID}

```
{
    "method":"property_reply",
    "id":"20a4ccfd-d308",
    "code": 0,
    "status":"some message"
}
```

动作 回复消息体

Topic：$thing/action/reply/{ProductID}/{DeviceID}

```
{
    "method":"action_reply",
    "id":"20a4ccfd-d308",
    "actionId": "动作",
    "code": 0,
    "status":"some message"
    "out": {action 的出参}
}
```

附 6：网关设备

设备分类

物联网开发平台根据设备功能性的不同将设备分为如下三类（即节点的分类）：
普通设备：此类设备可直接接入物联网开发平台且无挂载子设备。
网关设备：此类设备可直接接入物联网开发平台，并且可接受子设备加入局域网络。
子设备：此类设备必须依托网关设备才可与物联网开发平台进行通信，例如 Zigbee、蓝牙、RF433 等设备。

操作场景

对于不具备直接接入因特网的设备，可先接入本地网关设备的网络，利用网关设备与云端的通信功能，代理子设备接入物联网开发平台。
对于在局域网中加入或退出网络的子设备，网关设备可对其在平台进行绑定或解绑操作，并上报与子设备的拓扑关系，实现平台对于整个局域网子设备的管理。

接入方式

网关设备接入物联网开发平台的方式与普通设备一致。网关设备接入之后可代理同一局域网下的子设备上/下线，代理子设备上报数据，代理子设备接收云端下发给子设备的数据，并管理与子设备之间的拓扑关系。子设备的接入需通过网关设备来完成，子设备通过网关设备完成身份的认证之后即可成功接入云端。

代理子设备上下线

网关类型的设备，可通过与云端的数据通信，代理其下的子设备进行上线与下线操作。此类功能所用到的 Topic 与网关子设备拓扑管理的 Topic 一致：
数据上行 Topic（用于子设备发布）：
$thing/up/property/subdevice/${productid}/${device-id}
数据下行 Topic（用于子设备订阅）：
$thing/down/property/subdevice/${productid}/${device-id}

上线

请求上线：

```
{
  "method": "subdevice_online",
     "devices": [
```

```
        {
            "product_id": "11111",
            "device_id": “11111”
        }
    ]
}
```

上线结果：

```
{
   "method": "subdevice_online",
      "devices": [
        {
            "product_id": "11111",
            "device_id": “11111”,
            “result”:0
        }
    ]
}
```

下线

请求下线：

```
{
   "method": "subdevice_offline",
      "devices": [
        {
            "product_id": "11111",
            "device_id": “11111”
        }
    ]
}
```

下线结果：

```
{
   "method": "subdevice_offline",
      "devices": [
        {
            "product_id": "11111",
            "device_id": “11111”,
            “result”:0
        }
    ]
}
```

错误码

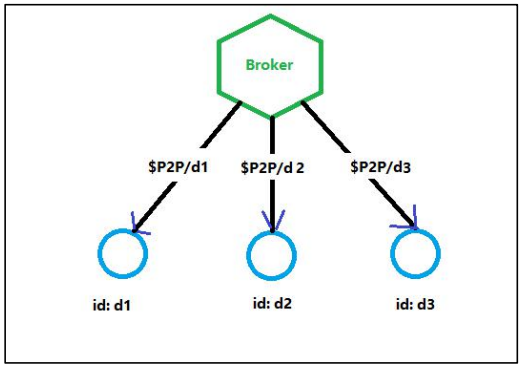
错误码	描述
0	成功
1	网关设备未绑定该子设备
2	系统错误，子设备上线或者下线失败
3	请求参数错误
4	设备名非法，或者设备不存在
5	不支持的子设备

附 7：名词对照表

英文名	中文名	含义	规约
Name	名称	界面上的显示效果	
Title	名称	界面上的显示效果	
Identify	标识符	唯一标识符	
Topic	主题	MQTT Topic	
Status	状态		
Schema	模型		
Object	对象		
Array	数组		
Bool	布尔		
Int	整形		
Action	动作		
Event	事件		
Payload	消息负载		
Up	上行		
Down	下行		
Reply	回复		

附 8：MQTT 非标特性

为了简化设备消息下发过程，IotHUB 底层组件扩展了 MQTT3.1 协议规范，引入新的功能：无需订阅即可直接接受消息。



直接发送消息示意图

规则

如果要直接给 Client Id 为 test1 的客户端发送消息,则对方的路由 Topic 为:

$P2P/test1

任何点对点发送的数据 TOPIC 前缀全部是: ***$P2P/***

注意：该特性是非标准 MQTT 协议，如果设备不支持自定义开发，还是需要订阅 Topic。

SDK 设计规范

SDK 是封装了技术细节的代码工具包，对于 IOTHUB 而言，需要设计一套统一接入的 SDK，目前对于 SDK 的设计规范如下。其示例代码并不是具体某个语言，而是一种伪代码描述方法。其中涉及到的名词和数据格式规范可以在【附 5】找到。

鉴权

第一步首先要鉴权：

```
Device *device = iothub.NewMQTTDevice(clientid,username,password)
//  下面是具体每个设备的实现
// Device *device = iothub.NewHTTPDevice(clientid,username,password)
// Device *device = iothub.NewCOAPDevice(clientid,username,password)
// Device *device = iothub.NewUDPDevice(clientid,username,password)

If (!device->Auth()){
    Return “Error info”
}else{
    // do something
}
```

上行

数据类上行：

```
Device->PropertyUp(“key”, value)
Device->ActionUp(“key”, value)
Device->EventUp(“key”, value)
```

回复类上行：

```
Device->ControlReply(“request-id”, value, “message”)
Device->ActionReply(“request-id”, value, “message”)
```

下行

下行仅限于 MQTT 协议：

```
//  属性下发
Device->OnPropertyControl((PropertySchema) =>[Reply] {
        // do something
```

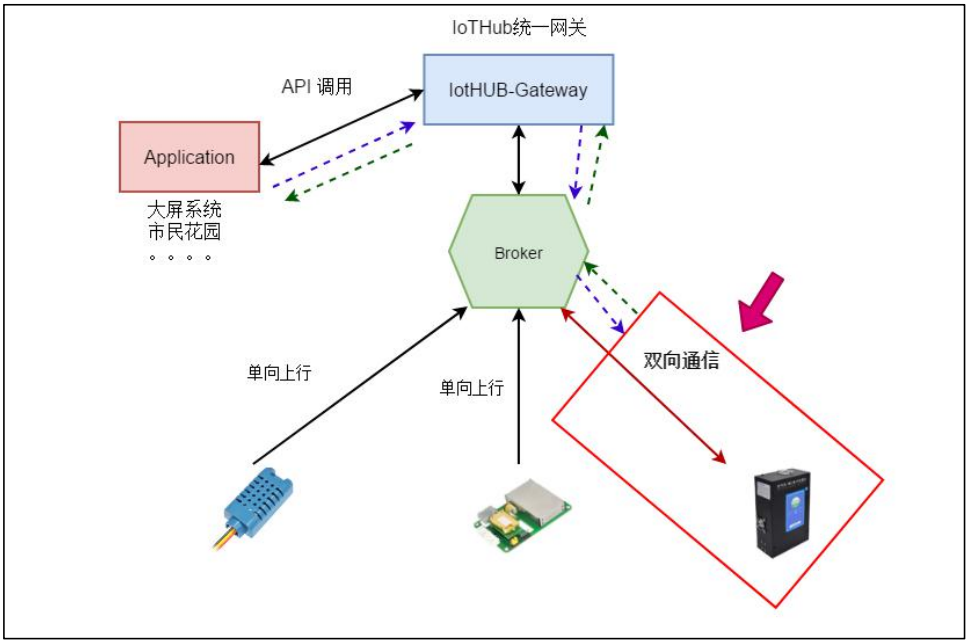
```
})
// 指令下发
Device->OnActionInvoke((PropertySchema) => [Reply] {
        // do something
})
```

退出

```
Device->LogOut()
```

下行数据规范

本节主要规范下行数据。



指令下发过程

接口字段

如果要下发指令，需要请求下发接口，下发接口请求字段规范如下：

参数名	类型	限制	备注
deviceID	String	Len == 固定长度	
data	JSON String	JSON 格式的字符串	这个就是真实物模型数据 JSON 表示
id	String	非空字符串	请求凭据，建议用 Nano 时间戳

下面是个多开关控制器的数据示例：

```
{
   "id": "11223344556677889900",
   "deviceId": "SN1234567890",
   "data": {
      "sw1": 1,
      "sw2": 0,
      "sw3": 0,
      "sw4": 0,
      "sw5": 0,
      "sw6": 0,
      "sw7": 1,
      "sw8": 0
```

```
  }
}
```

设备返回：

```
{
    "method":"control_reply",
    "id":"11223344556677889900",
    "code": 0, // 自定义状态码
    "status":"这里是一些文本，比如“成功”等"
}
```

注意：更多细节可参考【附 5】。

类型映射

物模型里面有一些数据类型可能对于其他编程语言来说并不支持或者语义不明确。因此设计一套映射关系来处理 SDK 生成时类型不匹配问题。

- 参考：物模型和编程语言数据类型映射.md · master · 终端设备 / 常见设计文档 · GitLab (infore-robotics.cn)