InGateway Documentation

发行版本 *0.0.1*

zhangning

2023 年 12 月 14 日

用户手册

CHAPTER 1

InGateway 文档网站导航

Device Supervisor App 为用户提供了便捷且可靠的数据采集、数据二次处理和数据上云等功能，支持 ISO on TCP、ModbusRTU 等多种工业协议解析。如果您想快速实现多种工业协议的数据采集并在本地预处理设备数据，处理过滤后的数据只需要简单配置后即可对接上传至自建 MQTT 云平台，那么 Device Supervisor 将是您理想的选择。

1.1 DeviceSupervisor Agent 用户手册（DS 2.0）

DeviceSupervisor Agent（以下简称 DSA）是运行在网关中的数采上云软件，为用户提供了便捷的数据采集、数据处理、数据上云和协议转换功能，支持 Modbus、ISO on TCP、EtherNet/IP 等多种工业协议和 DNP3、IEC 60870、IEC 61850 等电力协议解析。版本号为 **1.2.X** 的 DSA 简称为 **DSA 1.0**，版本号为 **2.X.X** 的 DSA 简称为 **DSA 2.0**。本手册以采集 PLC 的数据并上传至 EMQX 的在线 MQTT 服务器为例说明 DSA 2.0 如何通过 DSA 实现 PLC 数据采集和数据上云。以下将 InGateway902 简称为 “IG902”；InGateway502 简称为 “IG502”；InGateway974 简称为 “IG974”。

- 概览
- *1.* 准备硬件设备及其数据采集环境
  - *1.1* 硬件接线
    - *1.1.1* 以太网接线
    - *1.1.2* 串口接线
  - *1.2* 设置 *InGateway* 访问 *PLC*

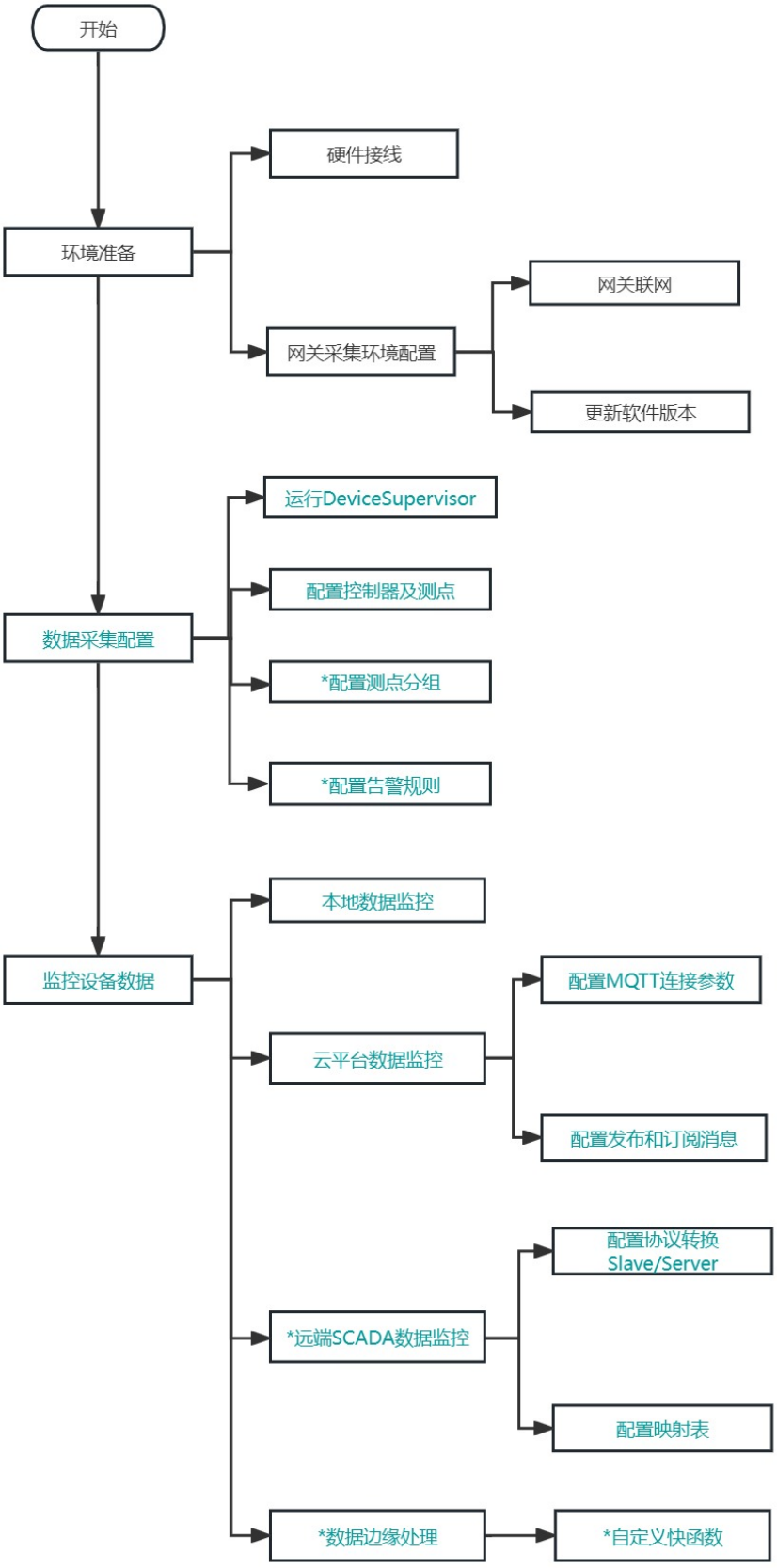
  - 查看 *App* 的云服务输出是否正确

1.1.1 概览

使用过程中，您需要准备以下项：

- 边缘计算网关 IG502/IG532/IG902/IG974
- PLC 设备
- 网线/串口线
- * 更新软件版本所需的固件、SDK 和 DSA
  - 固件版本
    - IG502：V2.0.0.r14045
    - IG532：V2.0.0.r14248
    - IG902：V2.0.0.r14047
    - IG974：V2.0.0.r14106
  - SDK 版本：py3sdk-V1.4.5 及以上
  - DSA 版本：2.2.0 及以上

整体流程如下图所示：


开始
环境准备
硬件接线
网关采集环境配置
网关联网
更新软件版本
数据采集配置
运行DeviceSupervisor
配置控制器及测点
*配置测点分组
*配置告警规则
监控设备数据
本地数据监控
云平台数据监控
配置MQTT连接参数
配置发布和订阅消息
*远端SCADA数据监控
配置协议转换
Slave/Server
配置映射表
*数据边缘处理
*自定义快函数

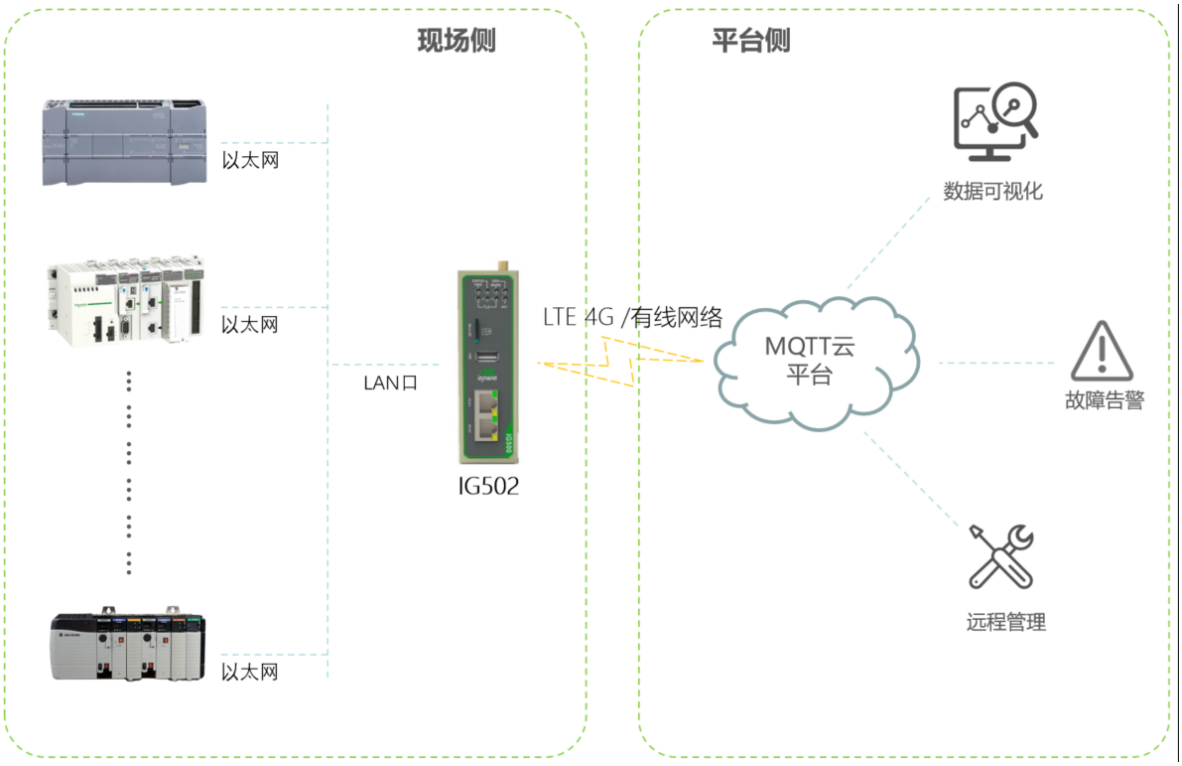
1.1.2 1. 准备硬件设备及其数据采集环境

1.1 硬件接线

1.1.1 以太网接线

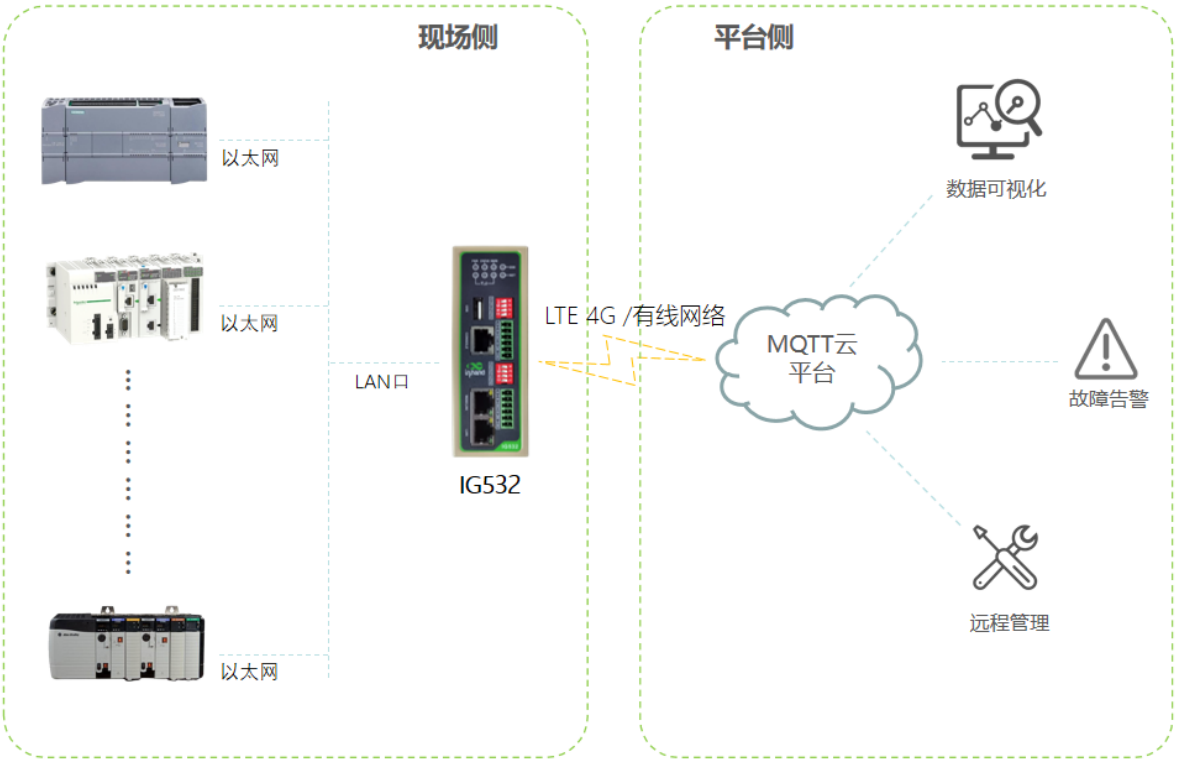
- IG502 以太网接线

  接通 IG502 的电源并按照拓扑使用以太网线连接 IG502 和 PLC。



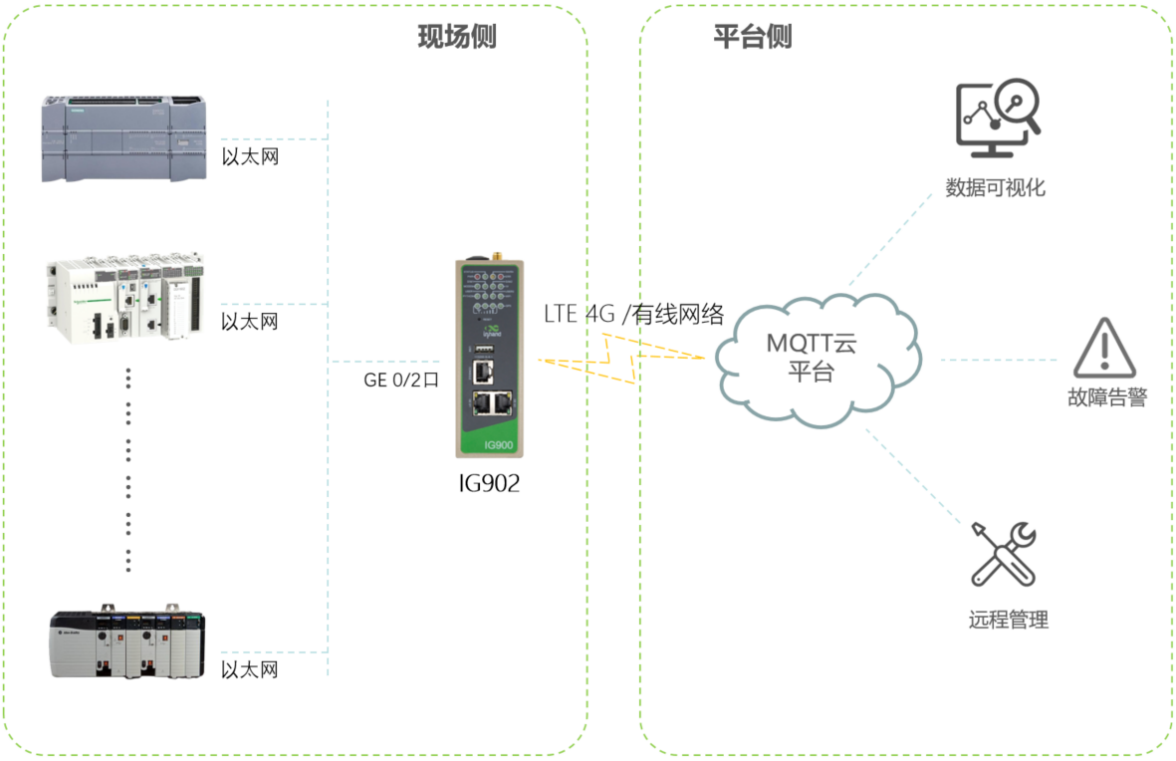
- IG532 以太网接线

  接通 IG532 的电源并按照拓扑使用以太网线连接 IG532 和 PLC。



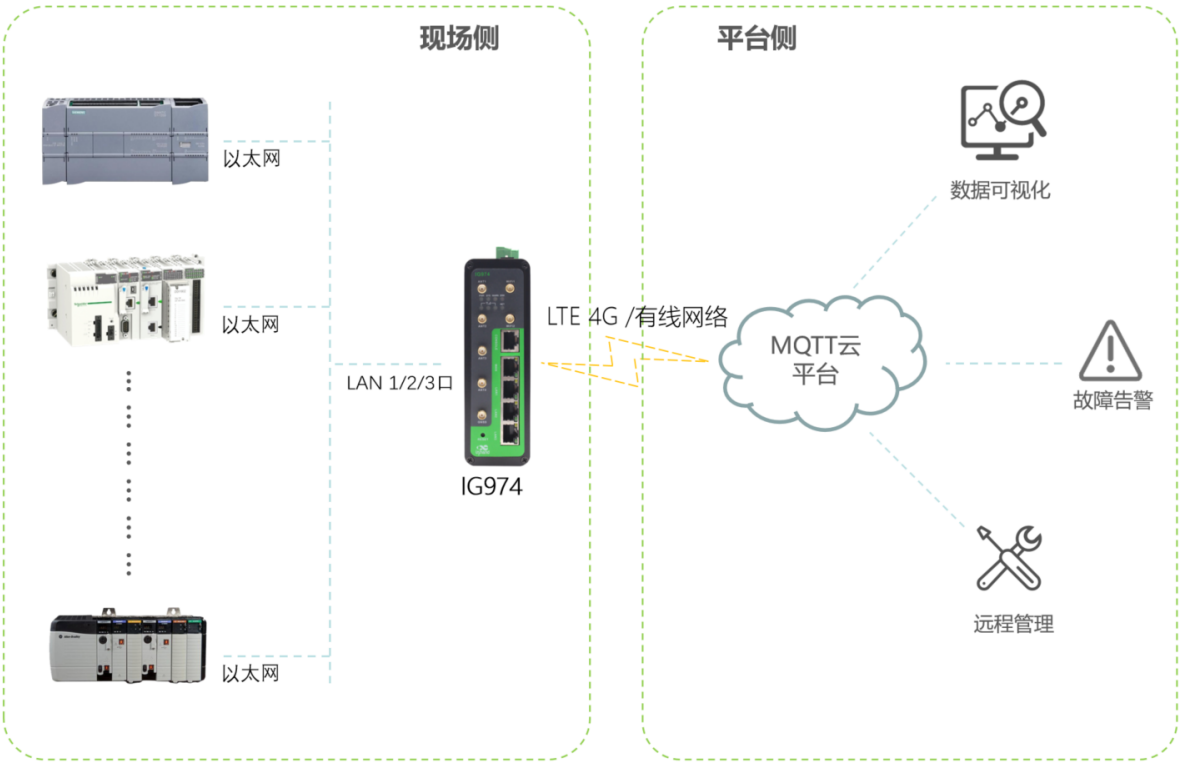
- IG902 以太网接线

  接通 IG902 的电源并按照拓扑使用以太网线连接 IG902 和 PLC。



- IG974 以太网接线

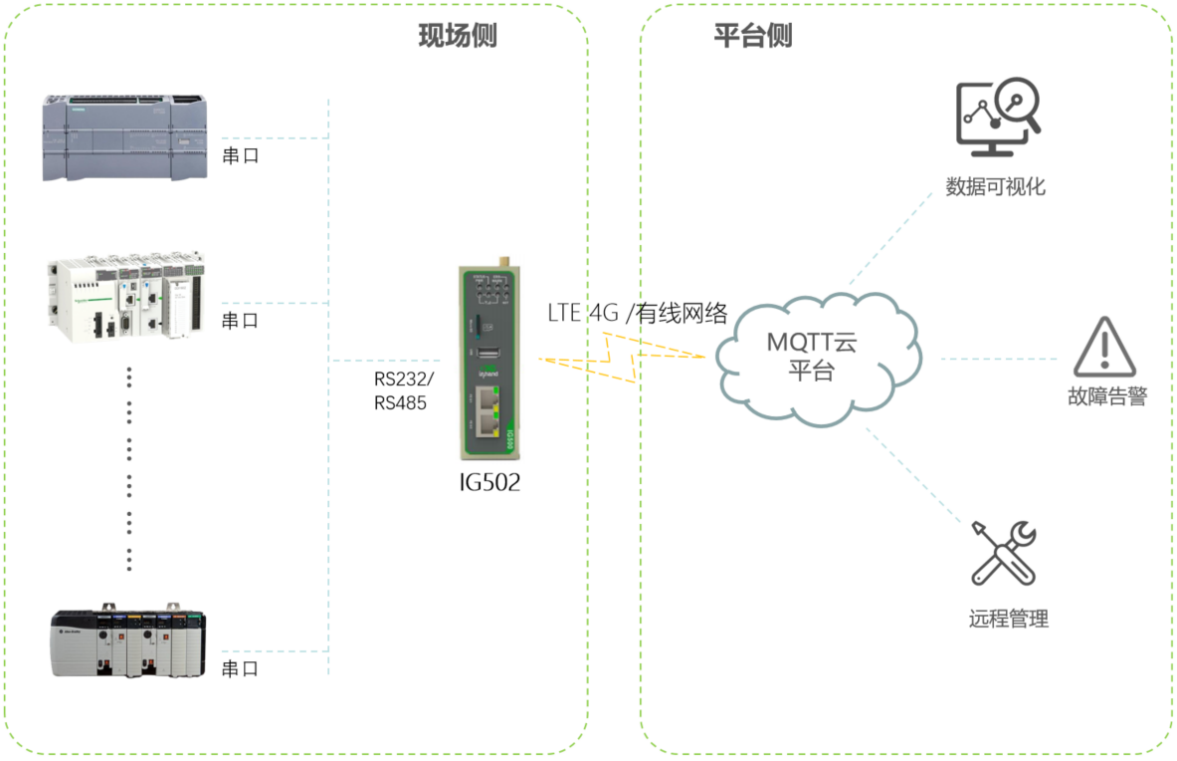
接通 IG974 的电源并按照拓扑使用以太网线连接 IG974 和 PLC。



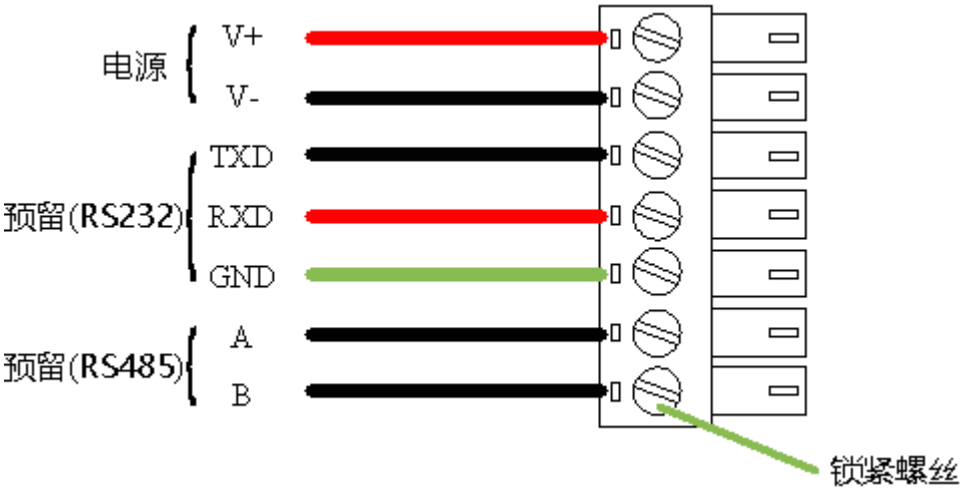
1.1.2 串口接线

- IG502 串口接线

  接通 IG502 的电源并按照拓扑连接 IG502 和 PLC。

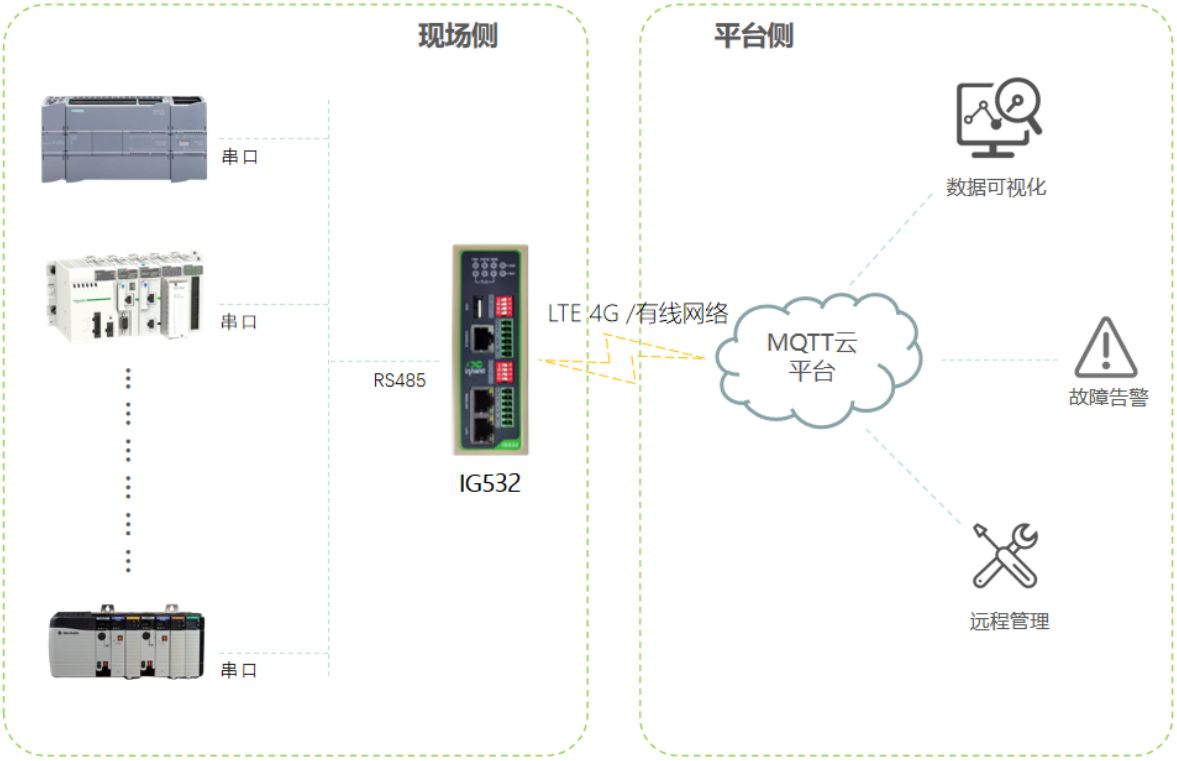


IG502 串口端子接线说明如下图：

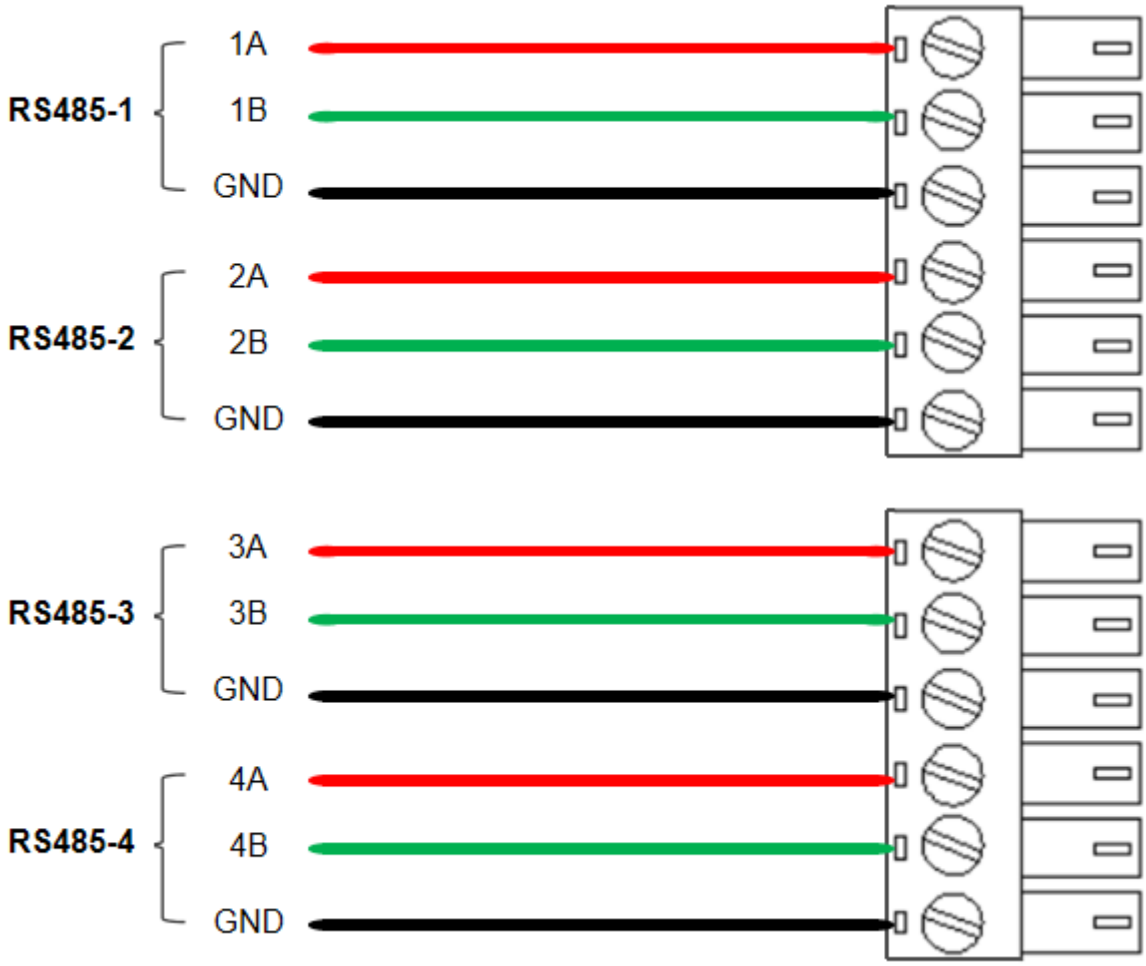


- IG532 串口接线

  接通 IG532 的电源并按照拓扑连接 IG532 和 PLC。

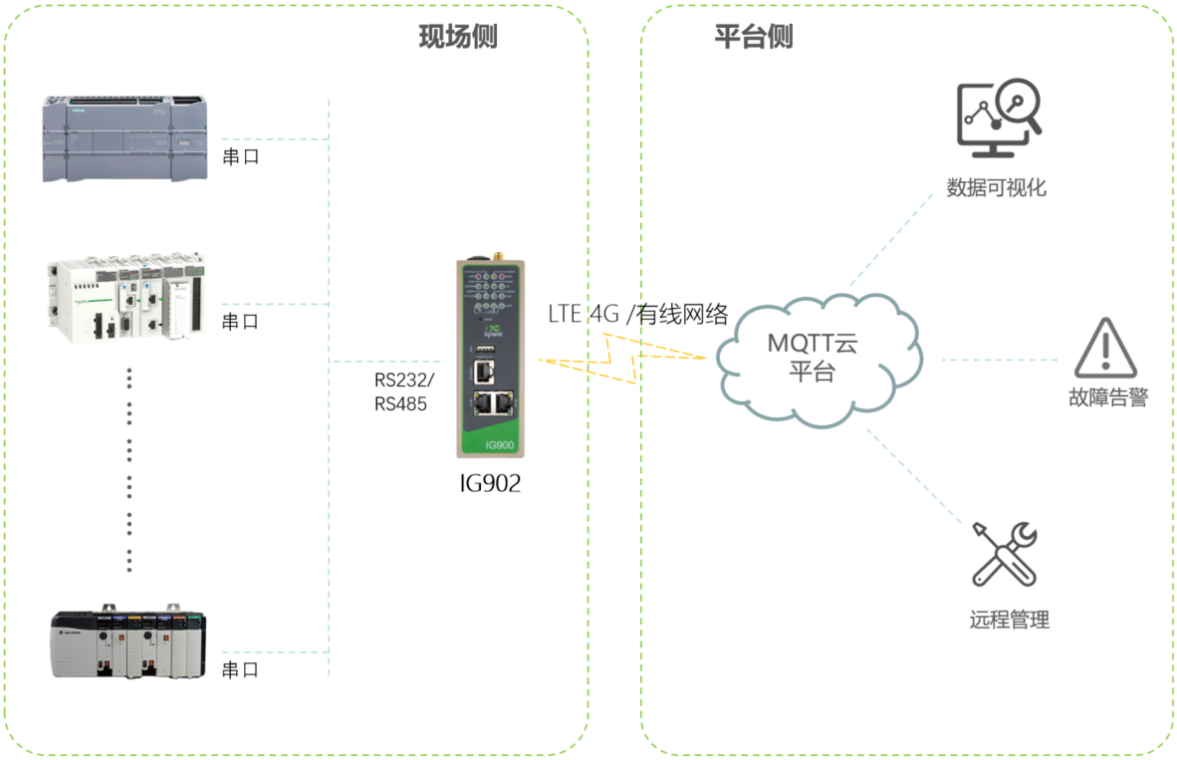


IG532 串口端子接线说明如下图：

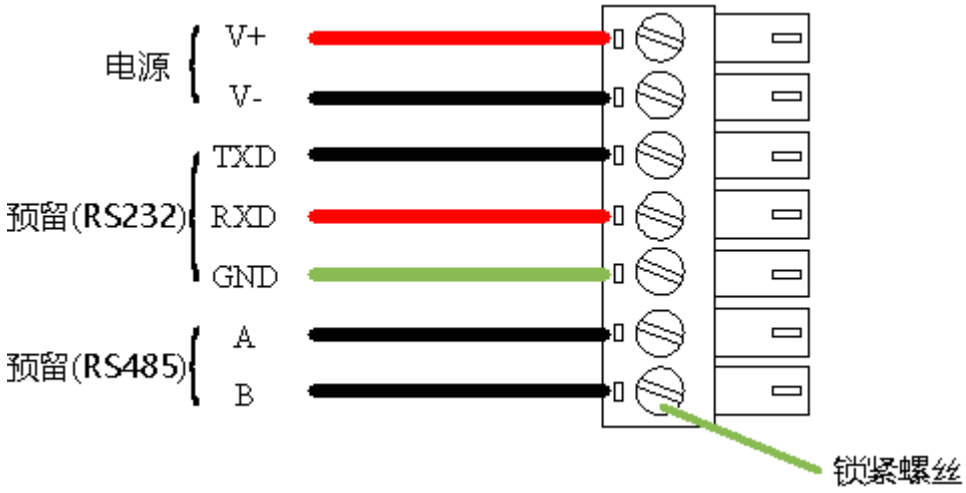


- IG902 串口接线

  接通 IG902 的电源并按照拓扑连接 IG902 和 PLC。

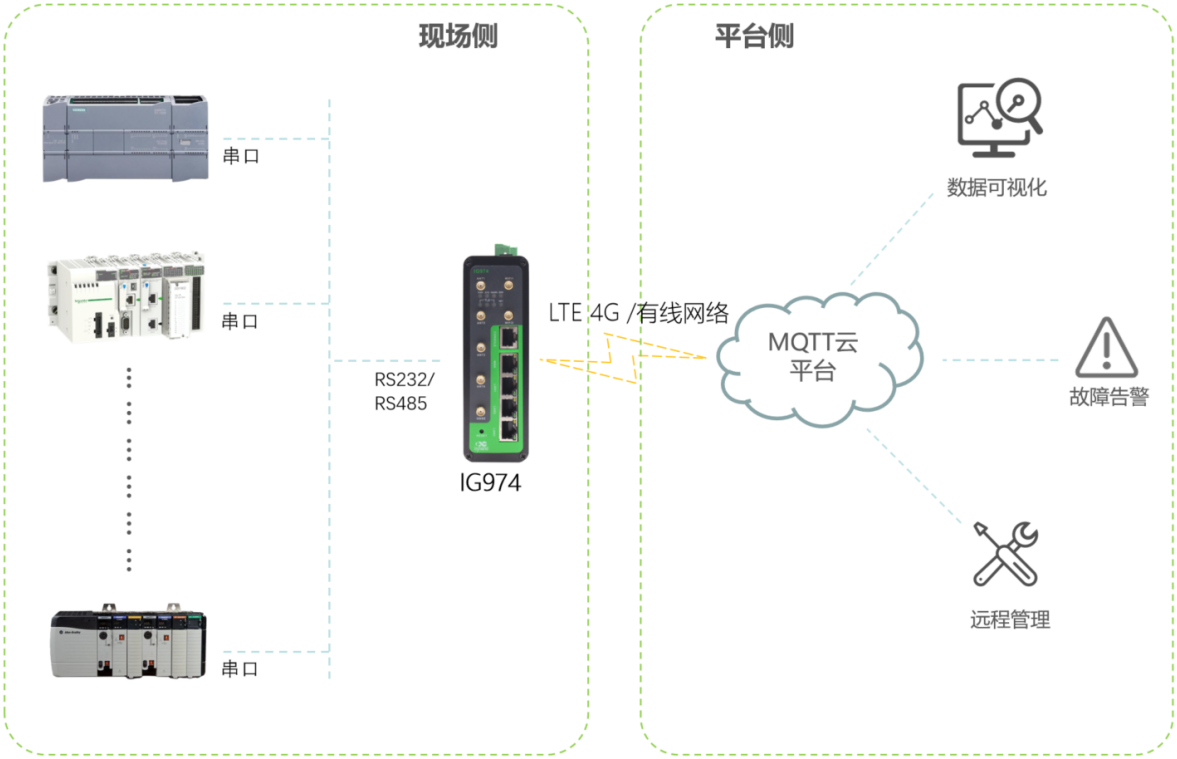


IG902 串口端子接线说明如下图：

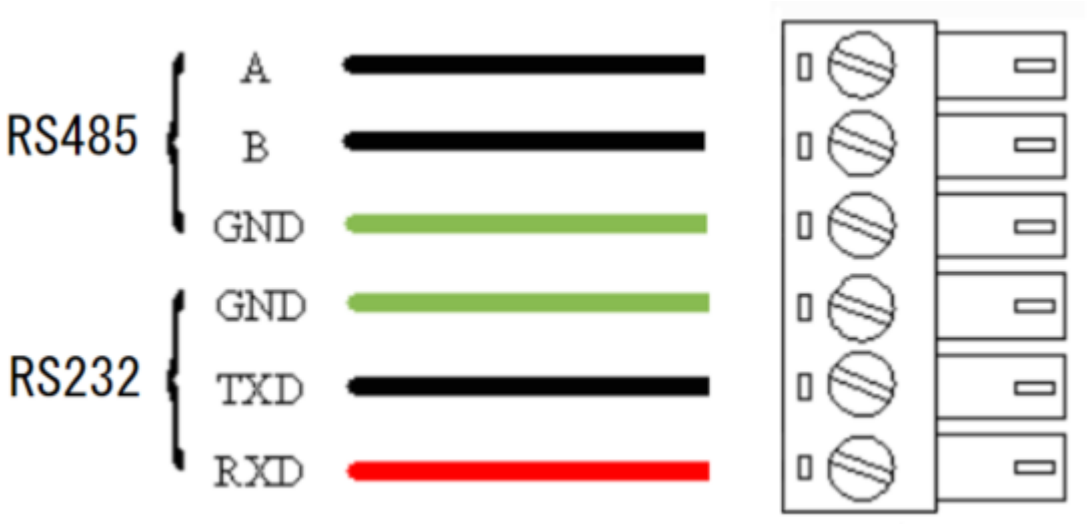


- IG974 串口接线

  接通 IG974 的电源并按照拓扑连接 IG974 和 PLC。



IG974 串口端子接线说明如下图：



1.2 设置 InGateway 访问 PLC

- IG502 的 LAN 口的默认 IP 地址为 `192.168.2.1`。为了使 IG502 能够通过 LAN 口访问以太网 PLC，需要设置 LAN 口与 PLC 处于同一网段，设置方法请参考访问 IG502。
- IG532 的 LAN 口的默认 IP 地址为 `192.168.2.1`。为了使 IG532 能够通过 LAN 口访问以太网 PLC，需要设置 LAN 口与 PLC 处于同一网段，设置方法请参考访问 IG532。
- IG902 的 GE 0/2 口的默认 IP 地址为 `192.168.2.1`。为了使 IG902 能够通过 GE 0/2 口访问以太网 PLC，需要设置 GE 0/2 口与 PLC 处于同一网段，设置方法请参考访问 IG902。
- IG974 的 LAN 口的默认 IP 地址为 `192.168.2.1`。为了使 IG902 能够通过 LAN 口访问以太网 PLC，需要设置 LAN 口与 PLC 处于同一网段，设置方法请参考访问 IG974。

1.3 设置 InGateway 联网

- 设置 IG502 联网请参考IG502 连接 Internet。
- 设置 IG532 联网请参考IG532 连接 Internet。
- 设置 IG902 联网请参考IG902 连接 Internet。
- 设置 IG974 联网请参考IG974 连接 Internet。

1.4 更新 InGateway 设备软件版本

如需获取 InGateway 产品最新软件版本及其功能特性信息，请访问资源中心。如需更新软件版本，请参考如下链接：

- 更新 IG502 软件版本

  使用 DSA 时，IG502 的固件版本应为 `V2.0.0.r14045` 及以上；SDK 版本应为 `py3sdk-V1.4.5` 及以上。

- 更新 IG532 软件版本

  使用 DSA 时，IG532 的固件版本应为 `V2.0.0.r14248` 及以上；SDK 版本应为 `py3sdk-V1.4.5` 及以上。

- 更新 IG902 软件版本

  使用 DDSA 时，IG902 的固件版本应为 `V2.0.0.r14047` 及以上；SDK 版本应为 `py3sdk-V1.4.5` 及以上。

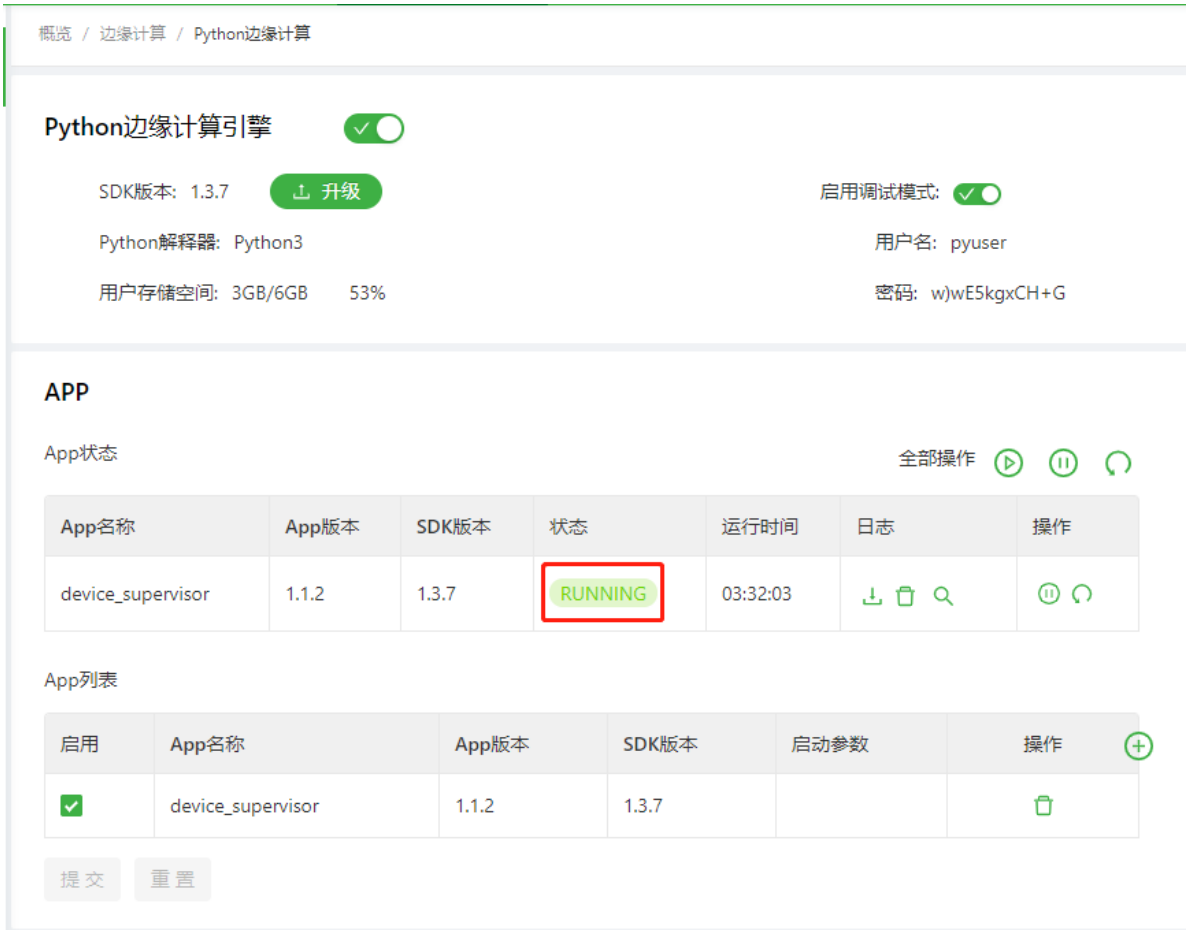
- 更新 IG974 软件版本

使用 DSA 时，IG974 的固件版本应为 `V2.0.0.r14106` 及以上；SDK 版本应为 `py3sdk-V1.4.5` 及以上。

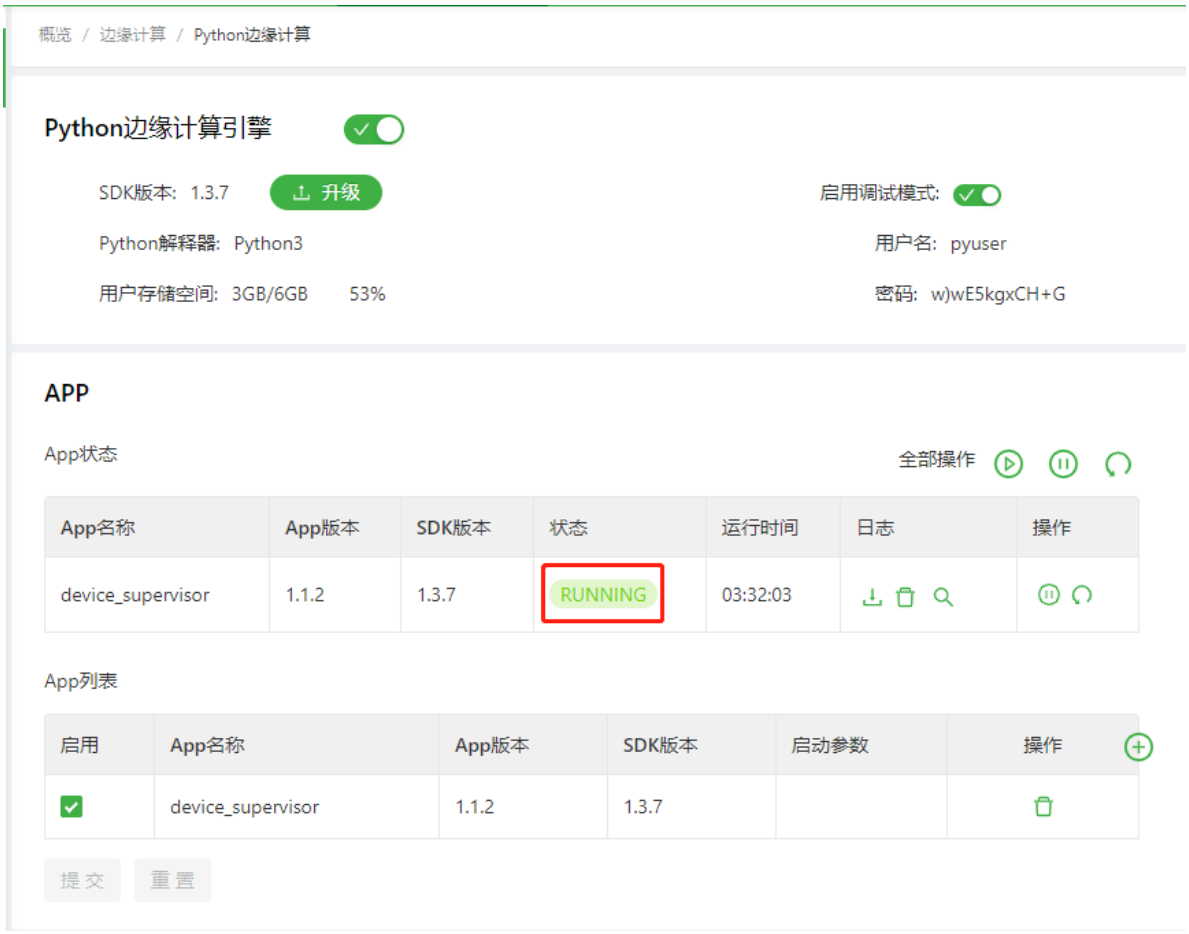
1.1.3 2.DSA 数据采集配置

2.1 安装并运行 DSA

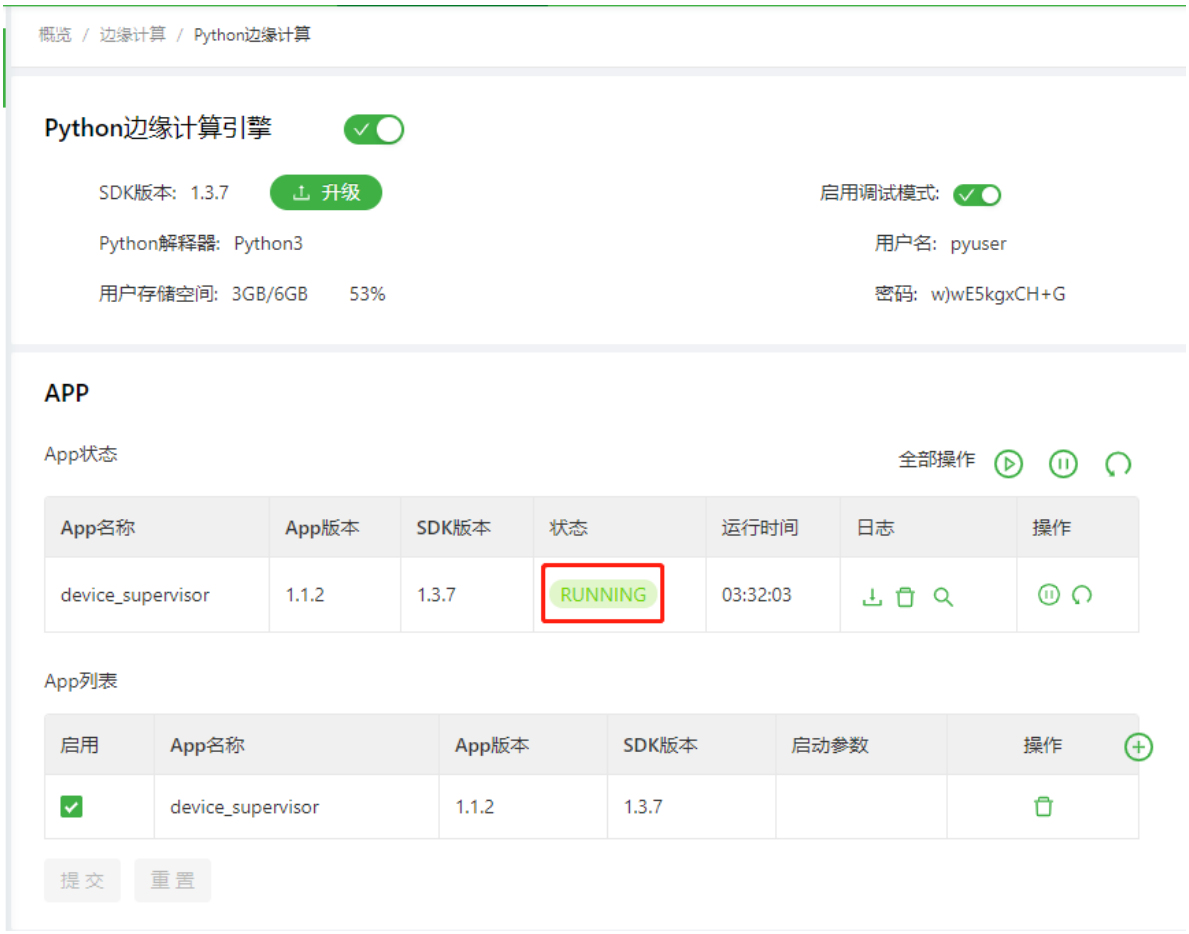
- IG502 如何安装并运行 Python App 请参考IG502 安装和运行 Python App，下载 DSA 请访问资源中心。DSA 正常运行后如下图所示：



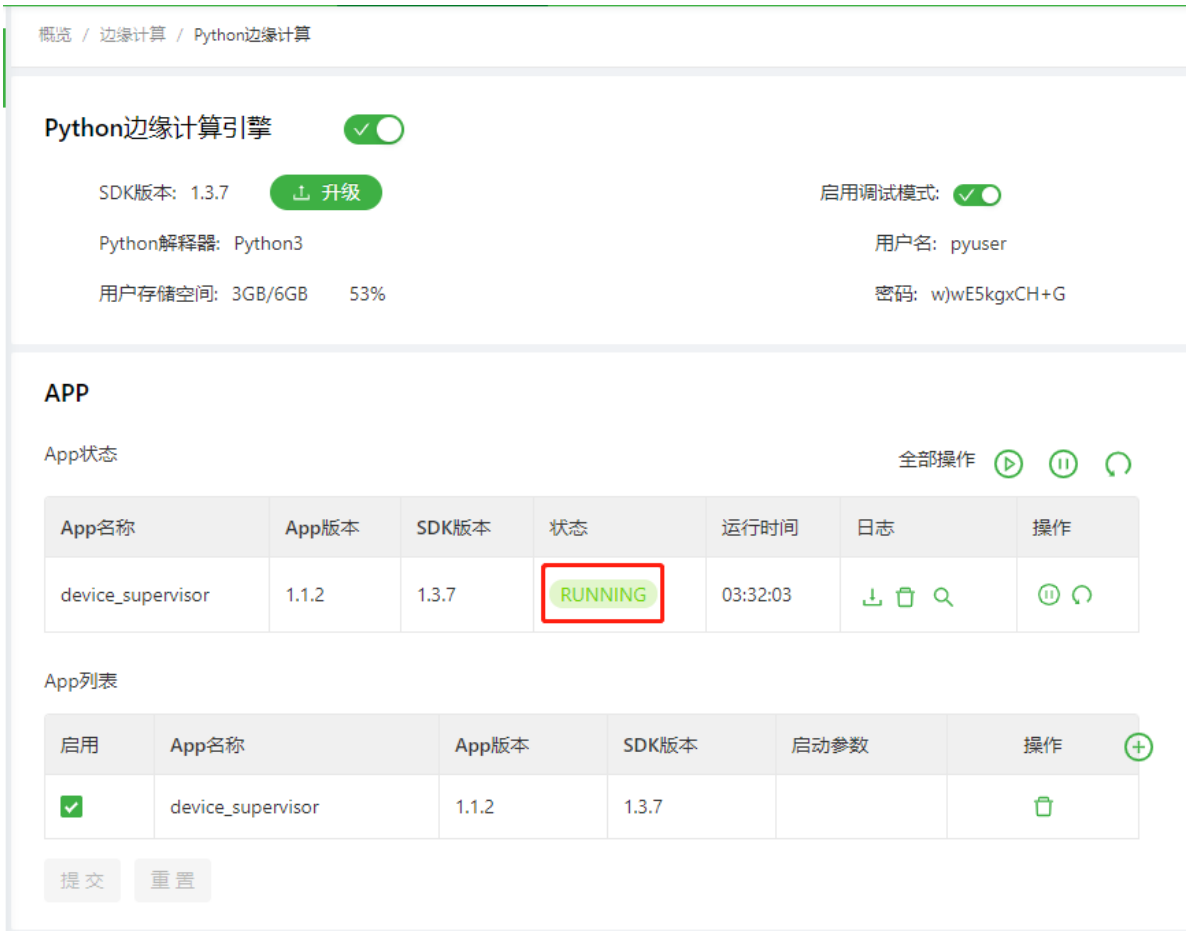
- IG532 如何安装并运行 Python App 请参考IG532 安装和运行 Python App，下载 DSA 请访问资源中心。DSA 正常运行后如下图所示：



- IG902 如何安装并运行 Python App 请参考IG902 安装和运行 Python App，下载 DSA 请访问资源中心。DSA 正常运行后如下图所示：



- IG974 如何安装并运行 Python App 请参考IG974 安装和运行 Python App，下载 DSA 请访问资源中心。DSA 正常运行后如下图所示：



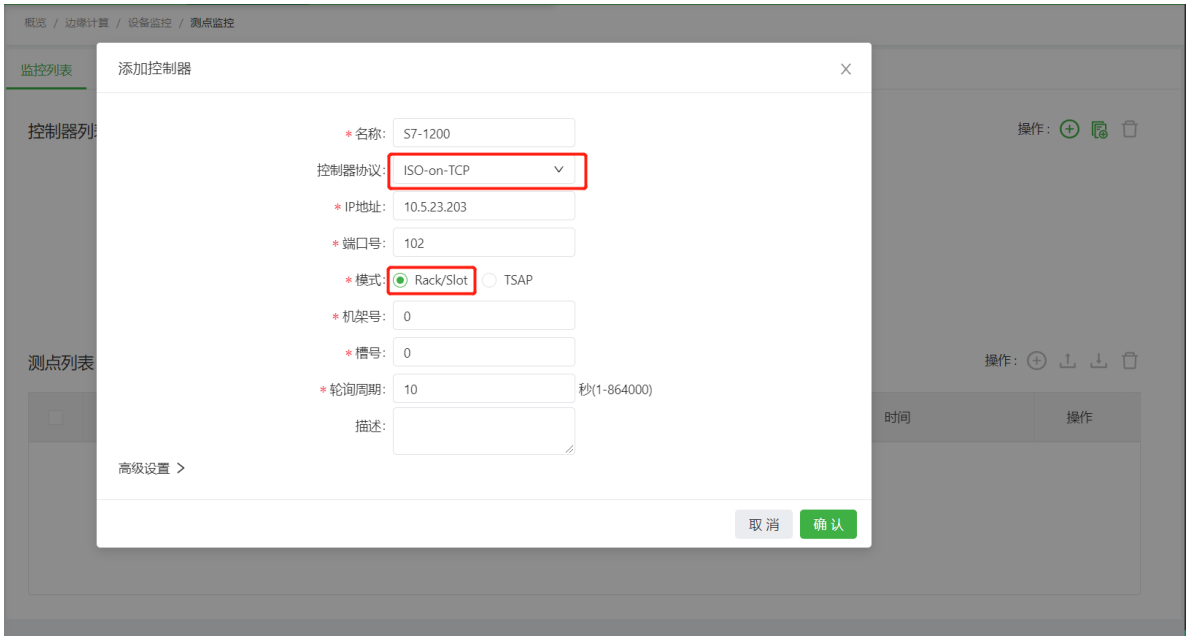
2.2 数据采集配置

2.2.1 添加控制器

- 添加以太网控制器（以 ISO on TCP 协议为例）

  进入“边缘计算 > 设备监控 > 测点监控”页面，点击“添加”按钮，在添加控制器页面选择控制器协议为“ISO on TCP”并配置控制器的通讯参数，轮询间隔为采集数据的时间间隔。注意：控制器名称不能重复。

  下图是添加 S7-1500、S7-1200、S7-400 和 S7-300 系列 PLC 的示例（模式选择 `Rack/Slot`）。机架号和槽号默认使用 0 即可：



下图是添加 S7-200、S7-200 Smart 和西门子 LOGO 系列 PLC 的示例（模式选择 `TSAP`）。注意：添加 S7-200 Smart 时，客户端 TSAP 配置为 02.00，服务端 TSAP 配置为 02.01；其余系列根据实际情况配置。

添加成功后如下图所示：

- 添加串口控制器（以 ModbusRTU 协议为例）

  进入“边缘计算 > 设备监控 > 测点监控”页面，点击“添加”按钮，在添加控制器页面选择控制器协议为“ModbusRTU”并配置控制器的通讯参数，轮询间隔为采集数据的时间间隔。注意：控制器名称不能重复。

添加成功后如下图所示：

如需修改 RS232/RS485 串口的通讯参数，请在“边缘计算 > 设备监控 > 参数设置”页面修改。修改后所有串口设备的通讯参数将自动修改并按照修改后的通讯参数通讯。

- 控制器部分参数说明
  - 启用多轮询周期：启用后，可以额外配置一个轮询周期并设置测点所使用的轮询周期。该功能可用于区分需要高频和低频采集的测点，高效利用网关和 PLC 性能。
    - 轮询周期 2：额外的轮询周期。
  - 启用组包上报：启用该功能的控制器下的测点可以配置为单独的轮询周期。在测点配置中启用了“组包上报”的测点按照设定的轮询周期和次数采集数据，并将多次采集的数据组包发布到本地的 MQTT 主题“ds2/eventbus/south/upload/msec/data/{controllerName}”上，可以在“自定义快函数”页面添加模式为“本地订阅消息触发”的快函数订阅该主题，在快函数中对组包数据进行处理。
    - 组包数据轮询周期：组包数据的轮询周期。
    - 组包数据上报：组包数据的轮询次数。

- 存储通讯报文：启用该功能的控制器会存储通讯报文，存储路径和条数可在“参数设置”页面的“默认参数”中设置。如需下载存储的通讯报文可在“测点监控”页面进入指定控制器的“实时通讯报文”页面，点击“下载”按钮下载。

2.2.2 添加测点

以 ISO on TCP 和 Modbus 协议添加测点为例。

- 添加 ISO on TCP 测点

  在“测点监控”页面点击“添加测点”按钮，在弹出框中配置测点参数：

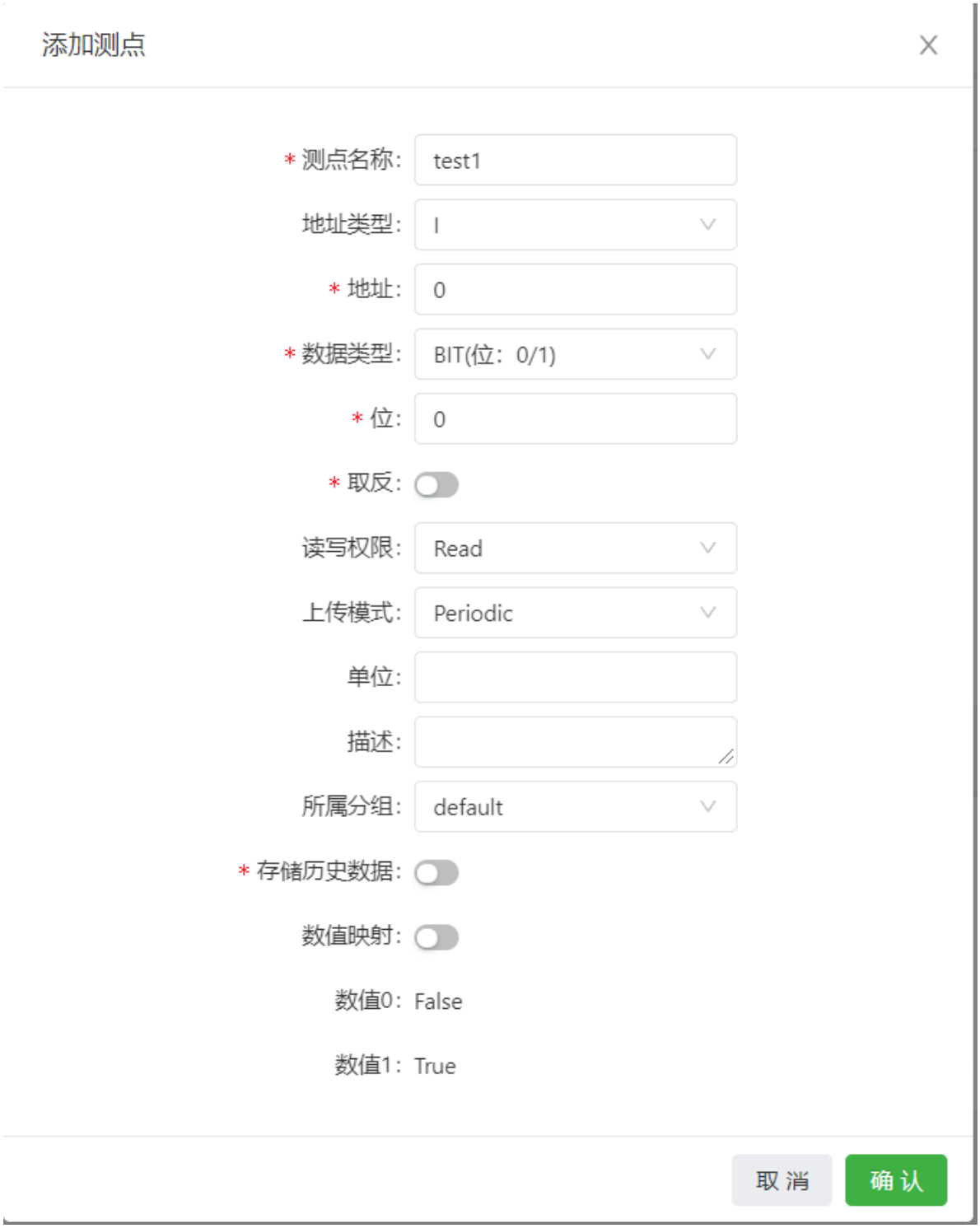
  - 测点名称：测点的名称（同一控制器测点名称不能重复）
  - 地址类型：测点地址类型，包括 I/Q/M/DB 四种类型
  - DB 索引：地址类型为 DB 时测点的 DB 号
  - 地址：测点地址
  - 数据类型：测点数据类型，包括：
    - BIT：0 或 1
    - BYTE：8 位无符号数据
    - SINT：8 位有符号数据
    - WORD：16 位无符号数据
    - INT：16 位有符号数据
    - DWORD：32 位无符号数据
    - DINT：32 位有符号数据
    - FLOAT：32 位浮点数
    - DOUBLE：64 位浮点数
    - STRING：8 位字符串
    - BCD：16 位 BCD 码
  - 小数位：数据类型为 FLOAT 或者 DOUBLE 时测点小数点后的数据长度，最大 6 位
  - 长度：数据类型为 STRING 时字符串长度，读取 1 个字符串的长度为 1
  - 按位取值：启用后可以读取整数中的任意位数据的值。
  - 位：读取位数据时测点的位偏移。
  - 读写权限：
    - Read：只读，不可写

    * Read/Write：可读可写

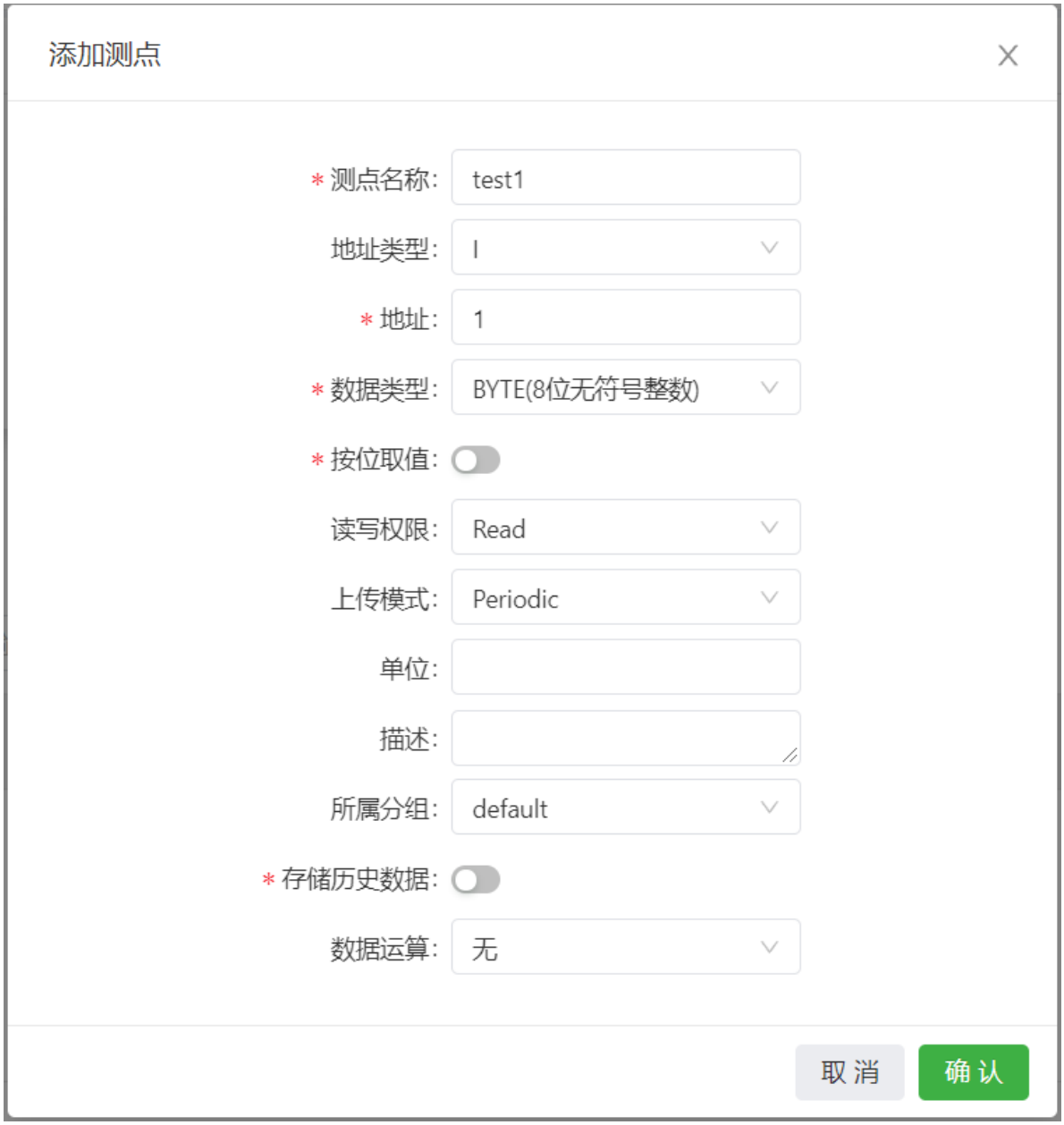
- 上传模式：
    * Periodic：按照所属分组的上报周期定时上报数据
    * Onchange：测点数值有变化时才按照分组的上报周期定时上报数据
    * Never：仅在本地使用，不需要上传云端的数据
- 变化死区：上传模式为 Onchange 时，可以设置数值变化在一定范围内视为数值无变化。如果数据有配置数据运算，则按照运算后的数值检测数值变化是否超过死区。
- 单位：测点单位
- 描述：测点描述
- 所属分组：测点所属分组
- 存储历史数据：启用后将以测点所属分组名称生成历史数据表，按照分组所设置的存储策略存储历史数据。存储的历史数据可导出为 CSV 文件或调用 Python API 获取。
- 数据运算：测点数据类型非 BIT 和 STRING 型时支持通过数据运算进行简单的运算处理，处理后的数据值可以上报至云平台。映射到协议转换中（Modbus Slave 或 OPC UA Server 等）的数值默认为采集的原始数据值而不是运算后的数据值
    * 无：不进行运算，使用采集的原始数据值
    * 比例运算：将采集的数据值按数据上下限范围等比例映射到比例上下限范围中的某个数据值，如将传感器中采集到的数据映射为实际的温湿度。计算公式为计算后的数据值 =（比例上限-比例下限）/（数据上限-数据下限）*（原始数据值-数据下限）+ 比例下限
        · 小数位：运算后需要保留的小数位
        · 数据上限：测点的数据上限值
        · 数据下限：测点的数据下限值
        · 比例上限：比例运算后的数据上限值
        · 比例下限：比例运算后的数据下限值
    * 偏移及缩放：按照倍率和偏移量计算原始数据值，公式为计算后的数据值 =（原始数据值 * 倍率）+ 偏移量
        · 小数位：运算后需要保留的小数位
        · 倍率：需要放大或缩小的倍率
        · 偏移量：倍率运算后需要增加或减少的数据值
    * 位截取：截取原始数据中的一段位数据作为数据值。FLOAT 型测点不支持此操作
        · 起始位：开始截取原始数据的位偏移
        · 结束位：结束截取原始数据的位偏移

    * PT/CT：按照倍率、偏移量、PT 和 CT 计算原始数据值，公式为 [(原始值 * 倍率)+ 偏移量]*PT*CT
      · 小数位：运算后需要保留的小数位
      · 倍率：需要放大或缩小的倍率
      · 偏移量：倍率运算后需要增加或减少的数据值
      · PT：额外的 PT 倍率
      · CT：额外的 CT 倍率
    * 数值映射：可配置测点的特定数值转换为其他数值。
      · 原始值：采集到的原始测点数值。
      · 映射值：期望转换后的测点数值。
  – 数值映射：启用后将 BIT 型测点的 0 和 1 值映射为 False 和 True
  – 组包上报：控制器启用组包上报后可见该字段。测点启用后，除了按照分组上报数据外还会按照控制器设定的组包上报逻辑上报数据

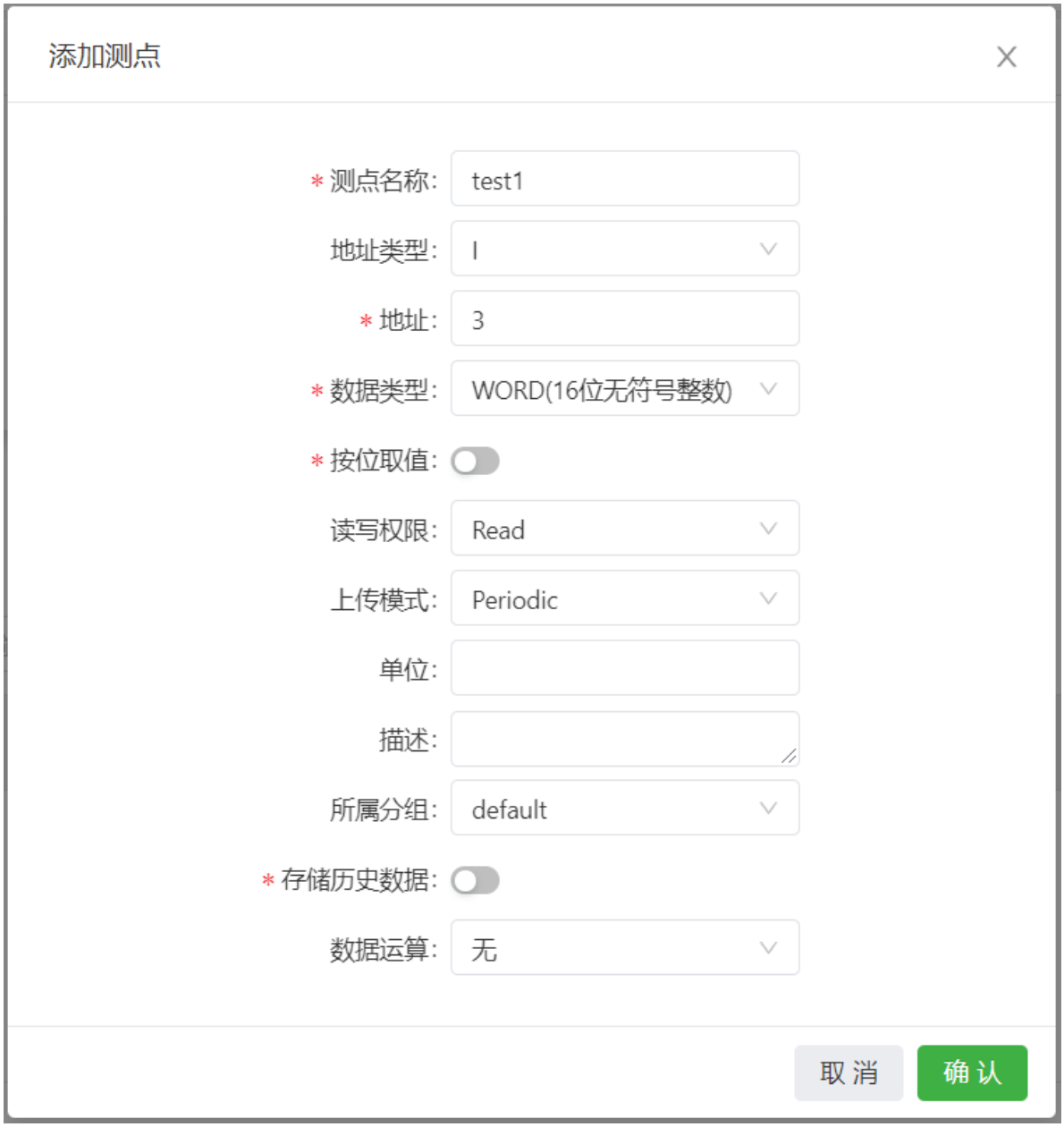
下图是添加一个地址为%I0.0 的开关测点的例子：



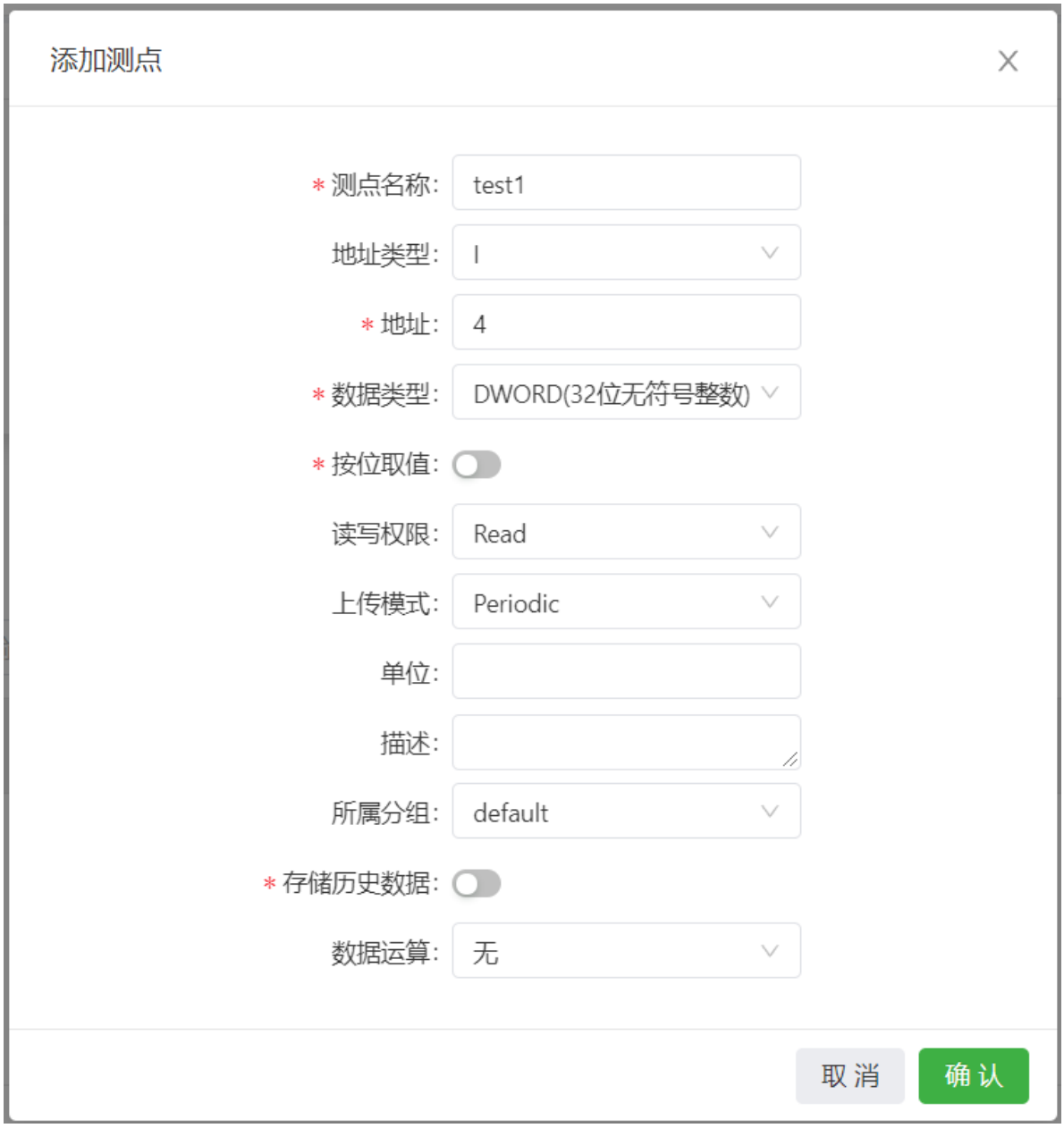
下图是添加一个地址为%IB1 的字节测点的例子:



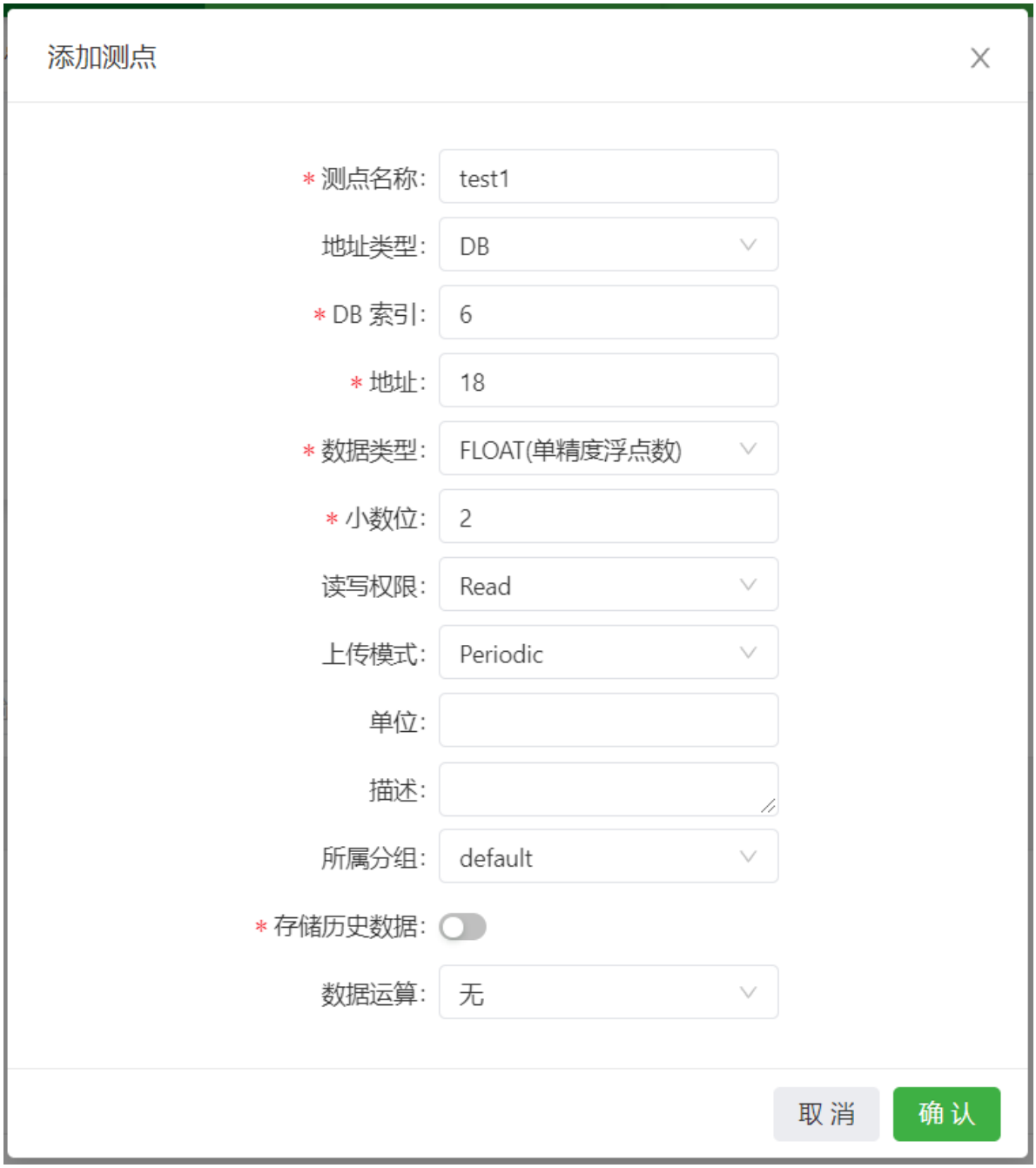
下图是添加一个地址为%IW3 的字测点的例子:



下图是添加一个地址为%ID4 的双字测点的例子:



下图是添加一个地址为%DB6.DBD18 的浮点数测点的例子:



- 添加 Modbus 测点

  在“测点监控”页面点击“添加测点”按钮，在弹出框中配置测点参数：

  - 测点名称：测点的名称（同一控制器下测点名称不能重复）
  - 地址：测点的寄存器地址
  - 数据类型：测点数据类型，包括：
    - BIT：0 或 1

    * WORD：16 位无符号数据
    * INT：16 位有符号数据
    * DWORD：32 位无符号数据
    * DINT：32 位有符号数据
    * FLOAT：32 位浮点数
    * STRING：8 位字符串
- 小数位：数据类型为 FLOAT 时测点小数点后的数据长度，最大 6 位
- 长度：数据类型为 STRING 时字符串长度
- 按位取值：启用后可以读取整数中的任意位数据的值。
- 位：读取位数据时测点的位偏移。
- 读写权限：
    * Read：只读，不可写
    * Read/Write：可读可写
- 上传模式：
    * Periodic：按照所属分组的上报周期定时上报数据
    * Onchange：测点数值有变化时才按照分组的上报周期定时上报数据
    * Never：仅在本地使用，不需要上传云端的数据
- 变化死区：上传模式为 Onchange 时，可以设置数值变化在一定范围内视为数值无变化。如果数据有配置数据运算，则按照运算后的数值检测数值变化是否超过死区。
- 单位：测点单位
- 描述：测点描述
- 所属分组：测点所属分组
- 存储历史数据：启用后将以测点所属分组名称生成历史数据表，按照分组所设置的存储策略存储历史数据。存储的历史数据可导出为 CSV 文件或调用 Python API 获取。
- 数据运算：测点数据类型非 BIT 和 STRING 型时支持通过数据运算进行简单的运算处理，处理后的数据值可以上报至云平台。映射到协议转换中（Modbus Slave 或 OPC UA Server 等）的数值默认为采集的原始数据值而不是运算后的数据值
    * 无：不进行运算，使用采集的原始数据值
    * 比例运算：将采集的数据值按数据上下限范围等比例映射到比例上下限范围中的某个数据值，如将传感器中采集到的数据映射为实际的温湿度。计算公式为计算后的数据值 =（比例上限-比例下限）/（数据上限-数据下限）*（原始数据值-数据下限）+ 比例下限
        · 小数位：运算后需要保留的小数位

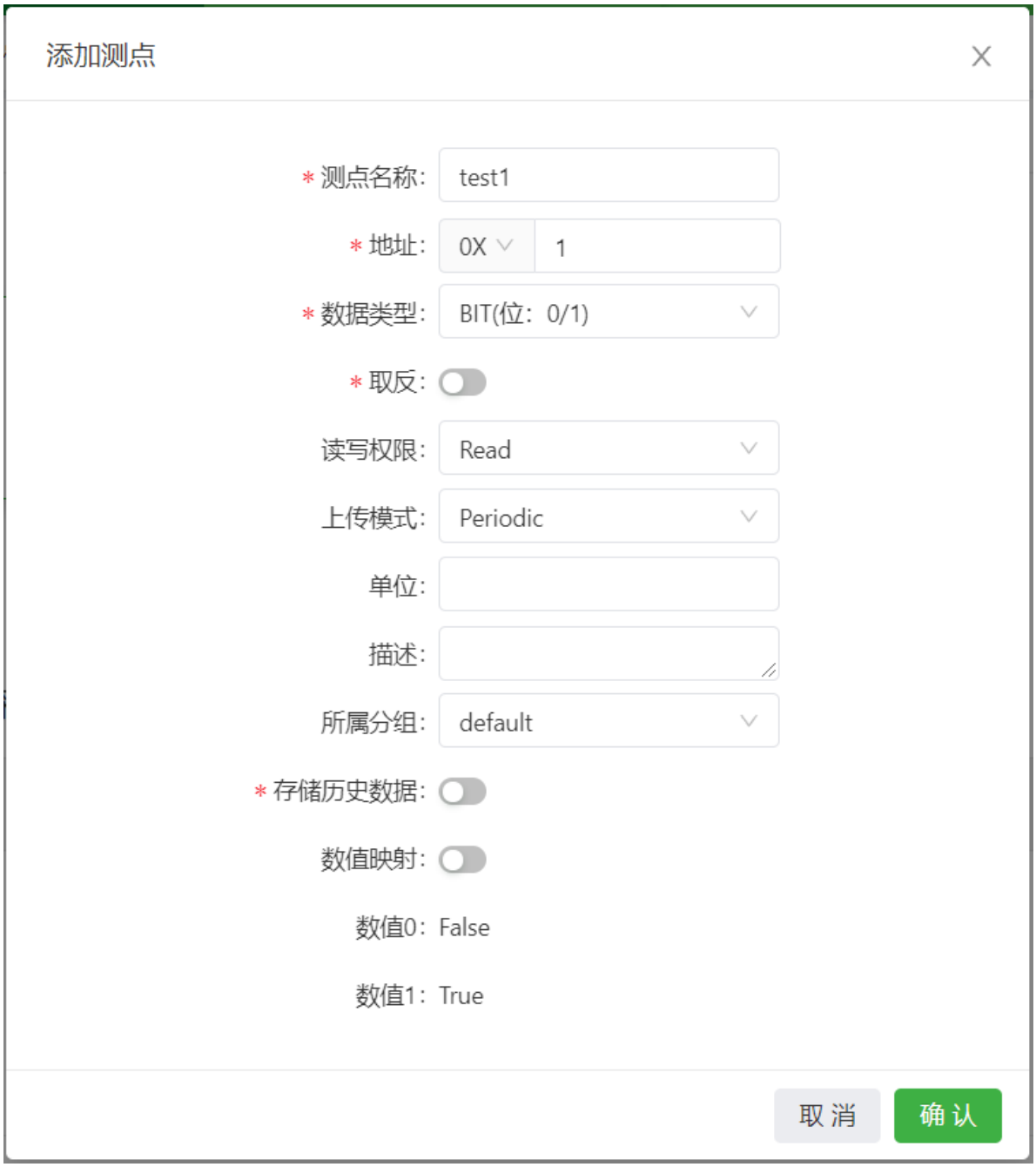
- 数据上限：测点的数据上限值
- 数据下限：测点的数据下限值
- 比例上限：比例运算后的数据上限值
- 比例下限：比例运算后的数据下限值

* 偏移及缩放：按照倍率和偏移量计算原始数据值，公式为计算后的数据值 =（原始数据值 * 倍率）+ 偏移量
  - 小数位：运算后需要保留的小数位
  - 倍率：需要放大或缩小的倍率
  - 偏移量：倍率运算后需要增加或减少的数据值
* 位截取：截取原始数据中的一段位数据作为数据值。FLOAT 型测点不支持此操作
  - 起始位：开始截取原始数据的位偏移
  - 结束位：结束截取原始数据的位偏移
* PT/CT：按照倍率、偏移量、PT 和 CT 计算原始数据值，公式为 [(原始值 * 倍率)+ 偏移量]*PT*CT
  - 小数位：运算后需要保留的小数位
  - 倍率：需要放大或缩小的倍率
  - 偏移量：倍率运算后需要增加或减少的数据值
  - PT：额外的 PT 倍率
  - CT：额外的 CT 倍率
* 数值映射：可配置测点的特定数值转换为其他数值。
  - 原始值：采集到的原始测点数值。
  - 映射值：期望转换后的测点数值。

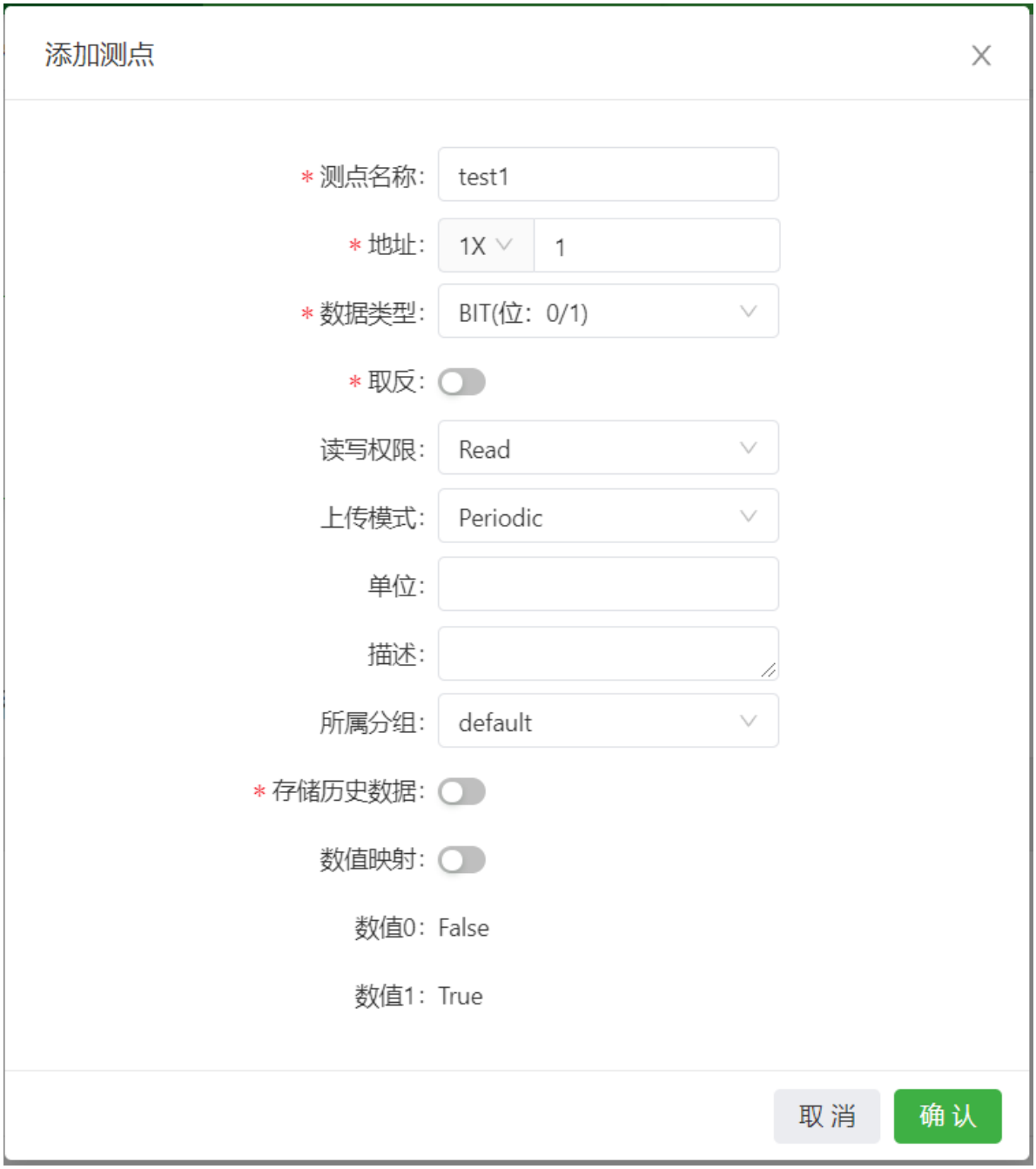
– 数值映射：启用后将 BIT 型测点的 0 和 1 值映射为 False 和 True

– 组包上报：控制器启用组包上报后可见该字段。测点启用后，除了按照分组上报数据外还会按照控制器设定的组包上报逻辑上报数据

下图是添加一个地址为 00001 的线圈测点的例子：



下图是添加一个地址为 10001 的开关测点的例子:



下图是添加一个地址为 30001 的整数测点的例子：

下图是添加一个地址为 40001 的浮点数测点的例子:

2.2.3 配置告警规则

你可以进入“边缘计算 > 设备监控 > 告警 > 告警规则”页面配置告警规则，点击“添加”按钮后，在弹出框中配置告警规则参数。参数如下：

- 名称：告警名称
- 控制器：告警测点所属控制器
- 测点名称：触发告警的测点名称
- 告警等级：从低到高分别支持“提醒”、“警告”、“次要”、“重要”、“严重”

- 告警条件
  - 判断条件：支持“=”、“!=”、“>”、“≥”、“<”、“≤”
  - 逻辑条件
    - 无逻辑条件：仅通过单个判断条件判断告警
    - &&：通过两个判断条件相与判断告警
    - ||：通过两个判断条件相或判断告警
- 告警内容：告警内容
- 告警标签：告警标签用于为告警分类，方便告警上云时快速选择

下图是一条告警等级为提醒的告警。该告警在测点数值 >30 且 <50 时产生告警；不在此范围时不产生告警或告警消除。

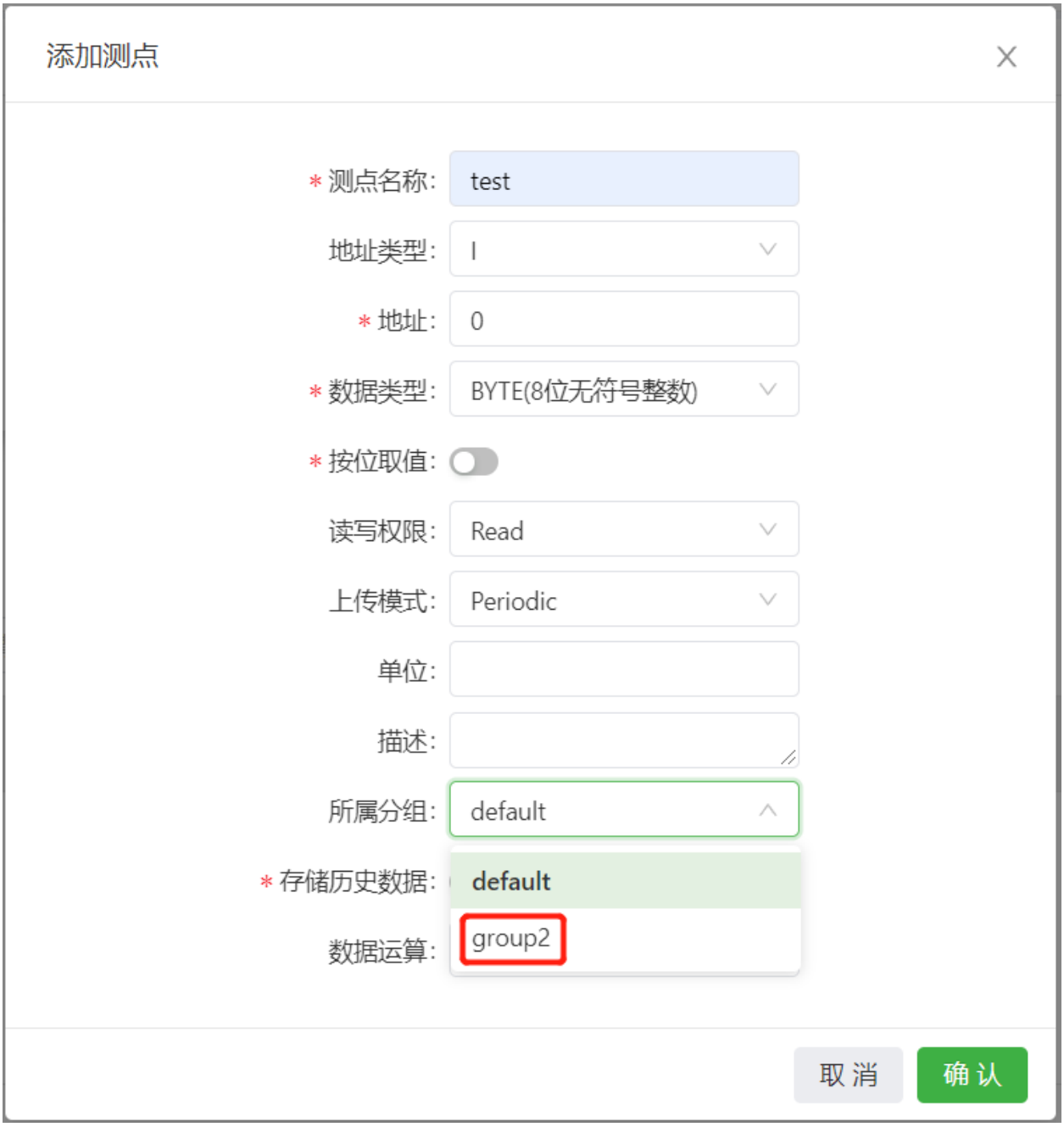
2.2.4 配置分组

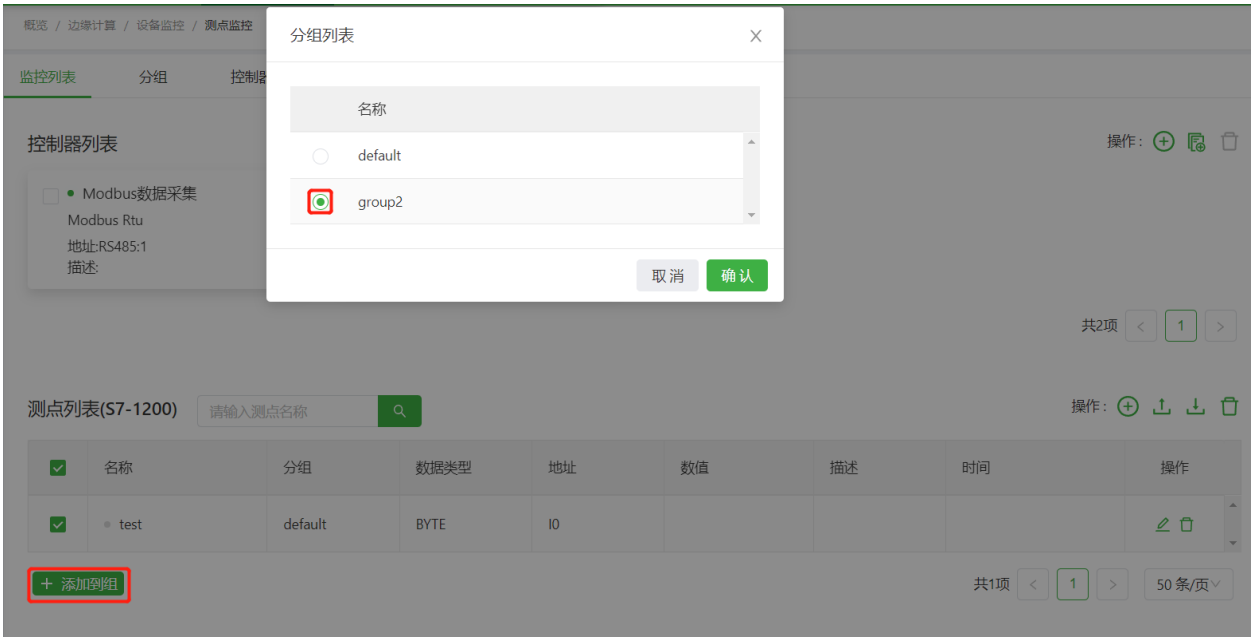
如需为测点配置不同的上报间隔或需要按照不同的 MQTT 主题上报相应的测点数据时，可在“边缘计算 > 设备监控 > 测点监控 > 分组”页面添加新分组。参数如下：

- 名称：分组名称
- 上报周期：分组内测点的上报周期。
- 周期上报 Onchange 数据：启用后，数值未变化的 Onchange 数据也会按照 Onchange 上报周期上传数据。
- Onchange 上报周期：Onchange 数据的固定上报周期。
- 历史数据
  - 最大条数：单个分组下最大存储的历史数据条数。
  - 存储策略：历史数据的存储周期
    - 与上报周期同步：按照上报周期存储历史数据。
    - 独立存盘周期：自定义历史数据的存储周期。
- 存储方式：历史数据的存储方式。切换存储路径后将清除存储的历史数据
  - 网关存储：历史数据存储在网关本身的存储空间中。
  - USB：历史数据存储在网关外接的 USB 存储空间中。
  - SD 卡：历史数据存储在网关外接的 SD 卡存储空间中。

下图添加了一个名为“group2”的分组，该分组每 30 秒上报一次分组中的测点：

添加分组后，添加测点时可以选择将测点关联到该分组或者在测点列表中选择测点添加到指定分组中。分组中的测点会按照分组的上报间隔上报数据。


添加测点
* 测点名称: test
地址类型: I
* 地址: 0
* 数据类型: BYTE(8位无符号整数)
* 按位取值:
读写权限: Read
上传模式: Periodic
单位:
描述:
所属分组: default
* 存储历史数据: default
数据运算: group2
取 消
确 认



点击“导出历史数据”可导出 CSV 格式的历史数据。

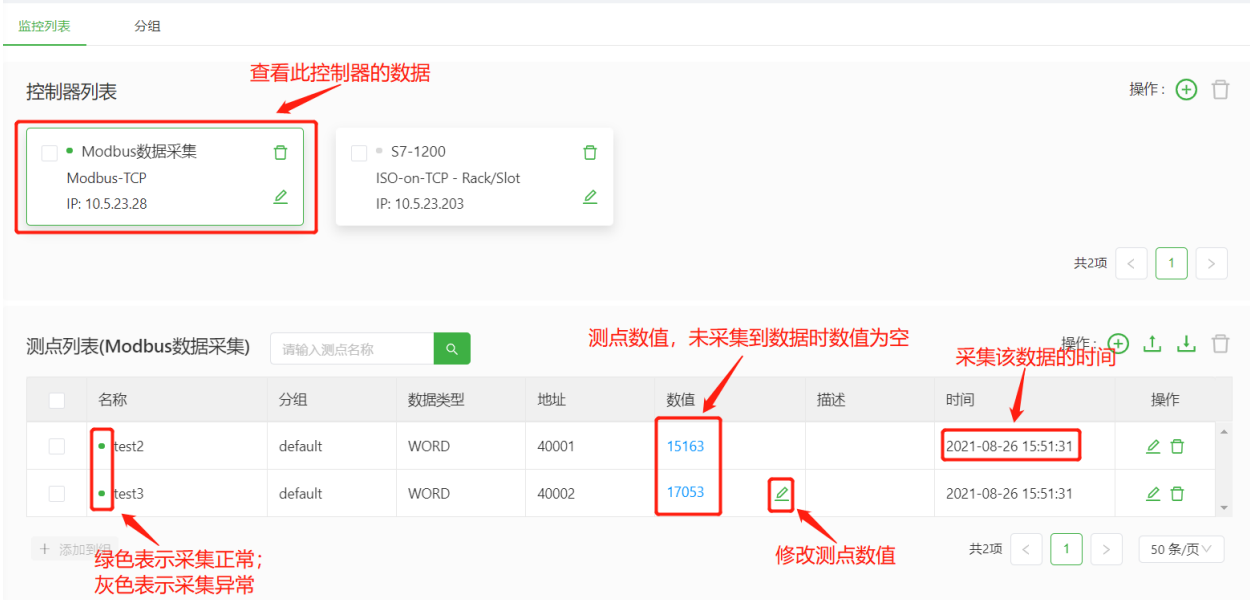
点击“清空历史数据”可清空已存储的历史数据。

1.1.4 3. 上报和监控 PLC 数据

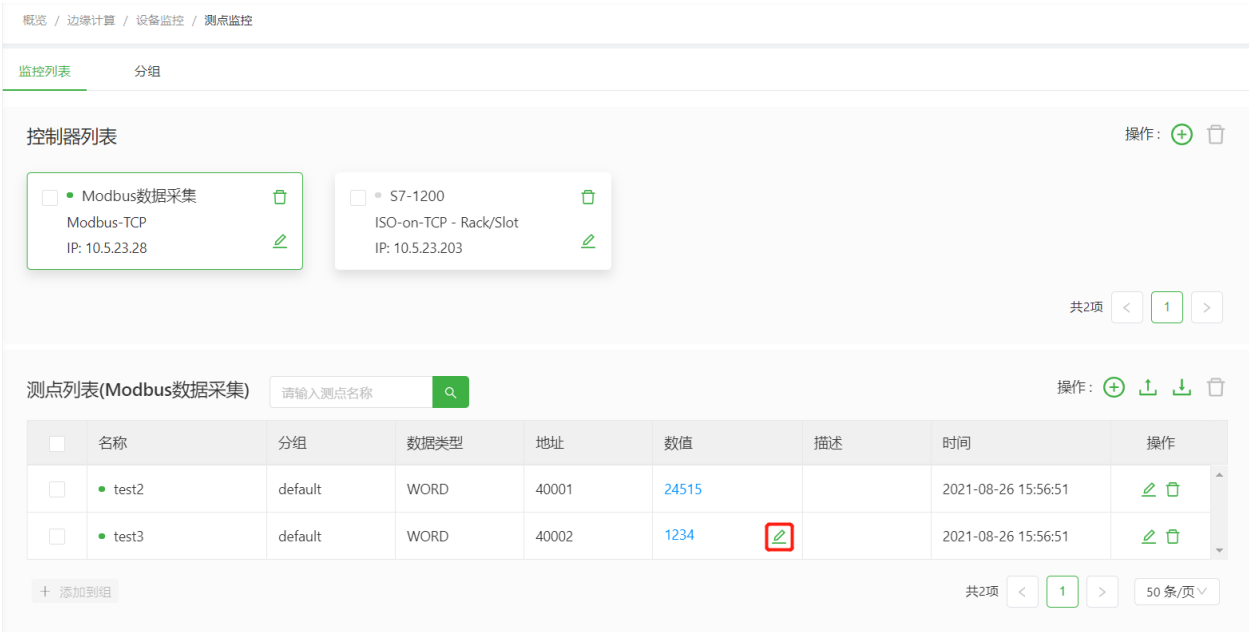
3.1 本地监控 PLC 数据

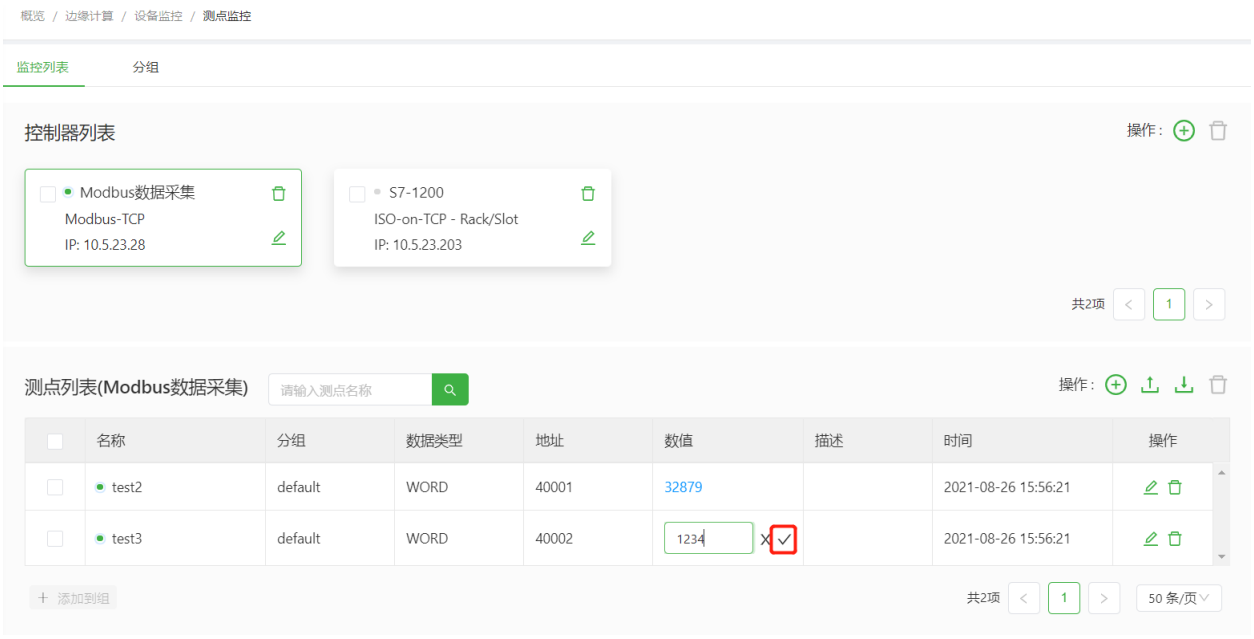
3.1.1 本地监控数据采集

数据采集配置完成后，可以在“边缘计算 > 设备监控 > 测点监控”页面查看数据采集情况。点击控制器列表中的控制器卡片可切换需要查看的 PLC 数据。

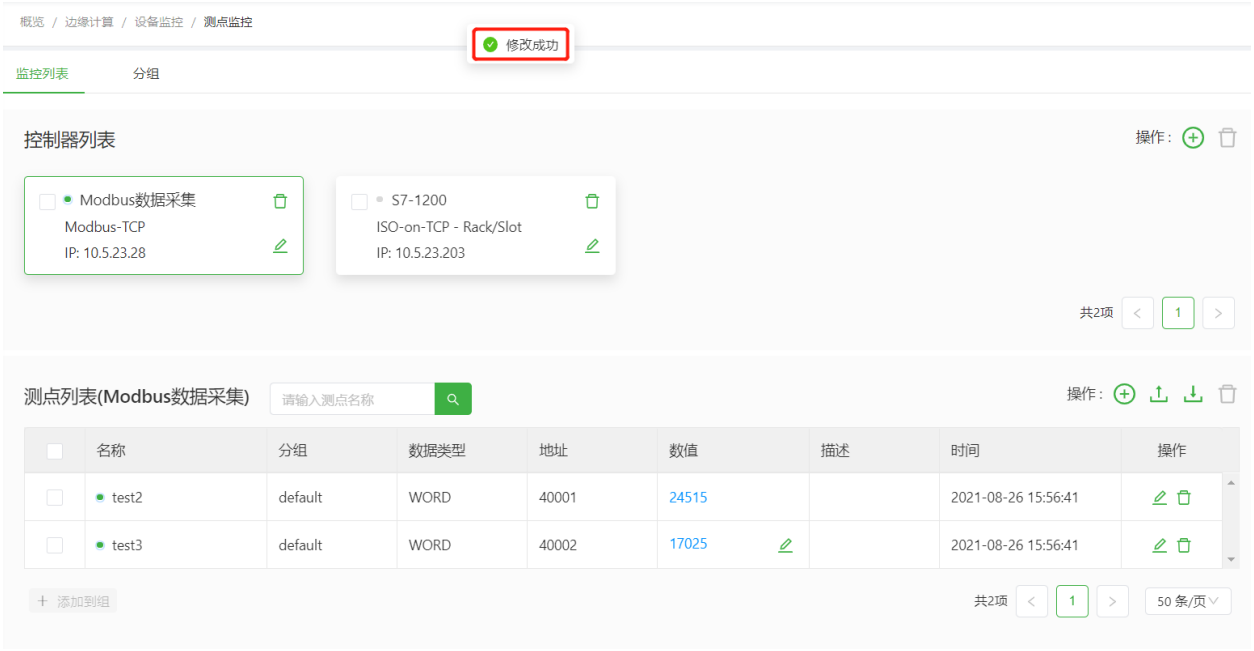


点击数值栏的按钮可进行写入操作。




概览 / 边缘计算 / 设备监控 / 测点监控
监控列表
分组
控制器列表
操作：
Modbus数据采集
Modbus-TCP
IP: 10.5.23.28
S7-1200
ISO-on-TCP - Rack/Slot
IP: 10.5.23.203
共2项
1
测点列表(Modbus数据采集)
请输入测点名称
操作：
名称
分组
数据类型
地址
数值
描述
时间
操作
test2
default
WORD
40001
32879
2021-08-26 15:56:21
test3
default
WORD
40002
1234
2021-08-26 15:56:21
添加到组
共2项
1
50 条/页

修改成功如下图所示：


概览 / 边缘计算 / 设备监控 / 测点监控
修改成功
监控列表
分组
控制器列表
操作：
Modbus数据采集
Modbus-TCP
IP: 10.5.23.28
S7-1200
ISO-on-TCP - Rack/Slot
IP: 10.5.23.203
共2项
1
测点列表(Modbus数据采集)
请输入测点名称
操作：
名称
分组
数据类型
地址
数值
描述
时间
操作
test2
default
WORD
40001
24515
2021-08-26 15:56:41
test3
default
WORD
40002
17025
2021-08-26 15:56:41
添加到组
共2项
1
50 条/页

3.1.2 本地监控告警

告警策略配置完成后，可以在“边缘计算 > 设备监控 > 告警”页面查看测点告警情况。

- 实时告警：查看当前未消除的告警信息

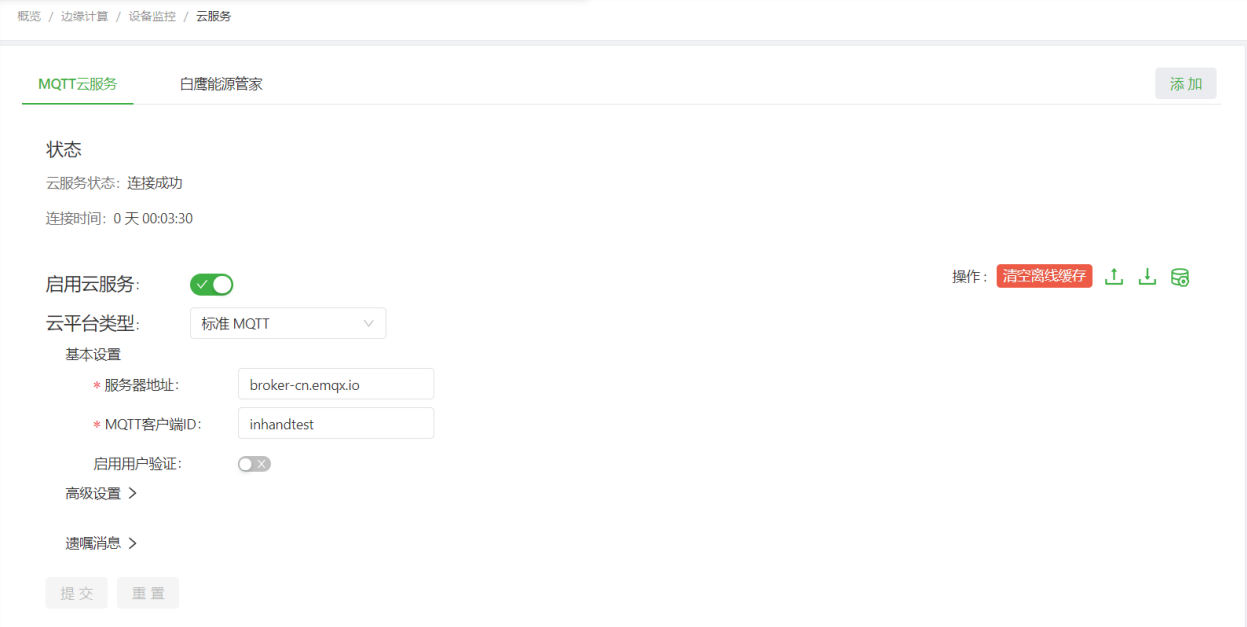
- 历史告警：筛选查看任意告警信息

3.2 云平台监控 PLC 数据

进入“边缘计算 > 设备监控 > 云服务”页面，勾选启用云服务并配置相应的 MQTT 连接参数，配置完成后点击提交。

- 类型：EMQX 的在线 MQTT 服务器的连接方式为标准 `MQTT`。阿里云 `IoT` 的使用方法请参考阿里云 IoT 使用说明；`AWS IoT` 的使用方法请参考AWS IoT 使用说明；`Azure IoT` 的使用方法请参考Azure IoT 使用说明
- 服务器地址：EMQX 的在线 MQTT 服务器地址为 `broker-cn.emqx.io`
- `MQTT 客户端 ID`：任一唯一 ID。支持调用自定义参数设置客户端 ID，如配置客户端 ID 为“${SN}”
- 其余项使用默认配置即可

配置完成后如下图所示：



提交后在“消息管理”中配置发布和订阅消息，已缓存的历史数据不需要上报时，可点击“清空离线缓存”清除已缓存的历史数据。发布和订阅消息的配置方法请参考消息管理（自定义 *MQTT* 发布/订阅）。以下是示例配置：

- 发布消息：
  - 名称：任意不重复的名称
  - 数据源类型：上报测点数据时选择测点分组消息；上报告警数据时选择告警消息。本示例使用测点分组消息
  - 分组/标签：需要上传数据至 thingsboard 的分组或告警标签名称，本文档为 `default`
  - `Topic: devices/telemetry`
  - `Qos(MQTT): 1`
  - 入口函数：入口函数名称，本文档为 `main`
  - 函数代码：

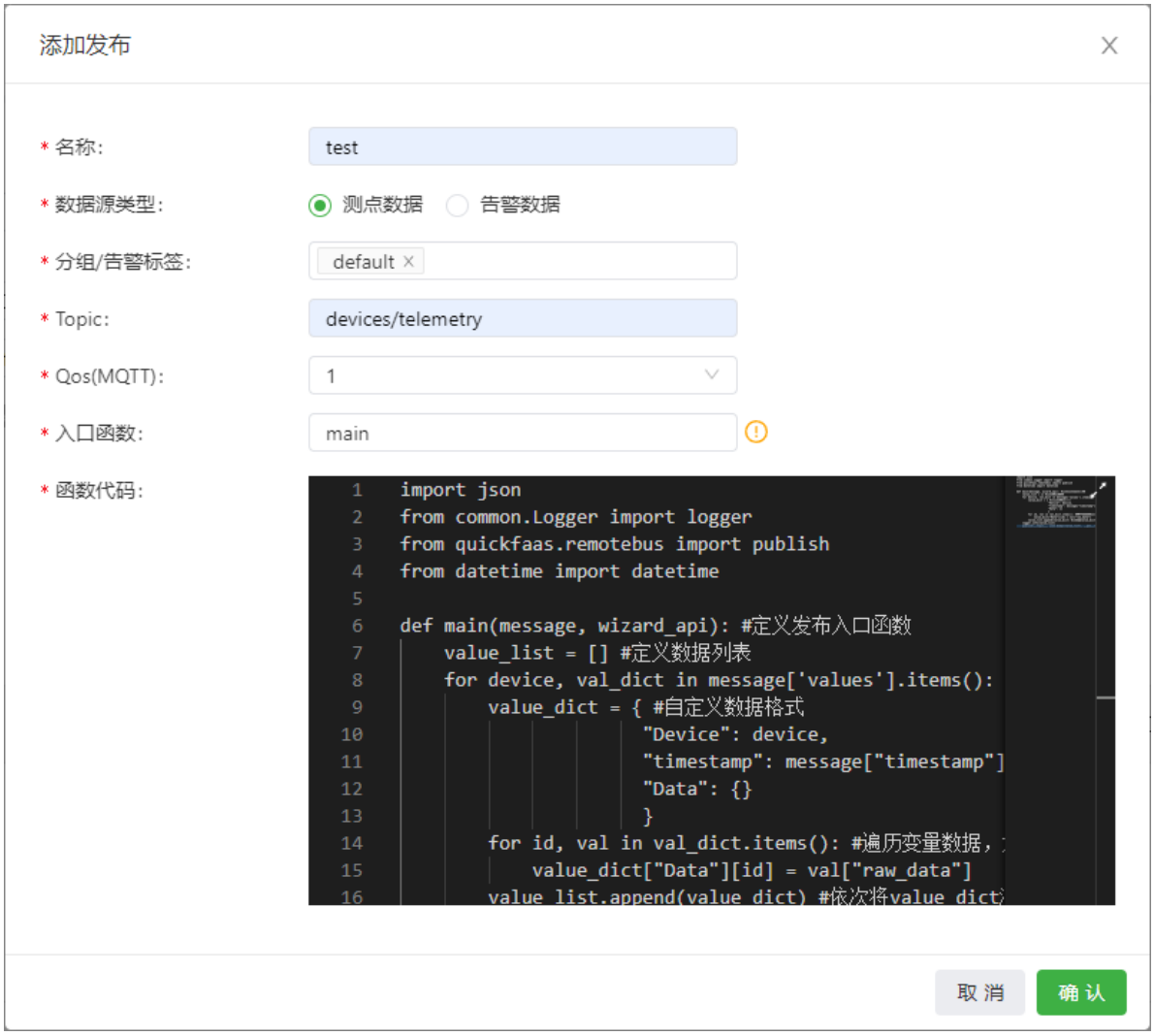
```
import json
from common.Logger import logger
from quickfaas.remotebus import publish
from datetime import datetime

def main(message, wizard_api): #定义发布入口函数
    value_list = [] #定义数据列表
    for device, val_dict in message['values'].items():
↪#遍历values字典，该字典中包含设备名称和设备下的变量数据
        value_dict = { #自定义数据格式
```

(续下页)

(接上页)

```
                    "Device": device,
                    "timestamp": message["timestamp"],
                    "Data": {}
                    }
        for id, val in val_dict.items(): #遍历变量数据，为数据字典赋值
            value_dict["Data"][id] = val["raw_data"]
        value_list.append(value_dict) #依次将value_dict添加到value_list中
    logger.info(value_list)
    publish(__topic__, json.dumps(value_list), __qos__)
↪#使用发布消息中定义的主题和qos发送value_
↪list至云平台。当qos不为0时，数据发送失败则缓存数据等待连接恢复后按先存先传的顺序上传至MQT
```

配置完成后如下图所示：

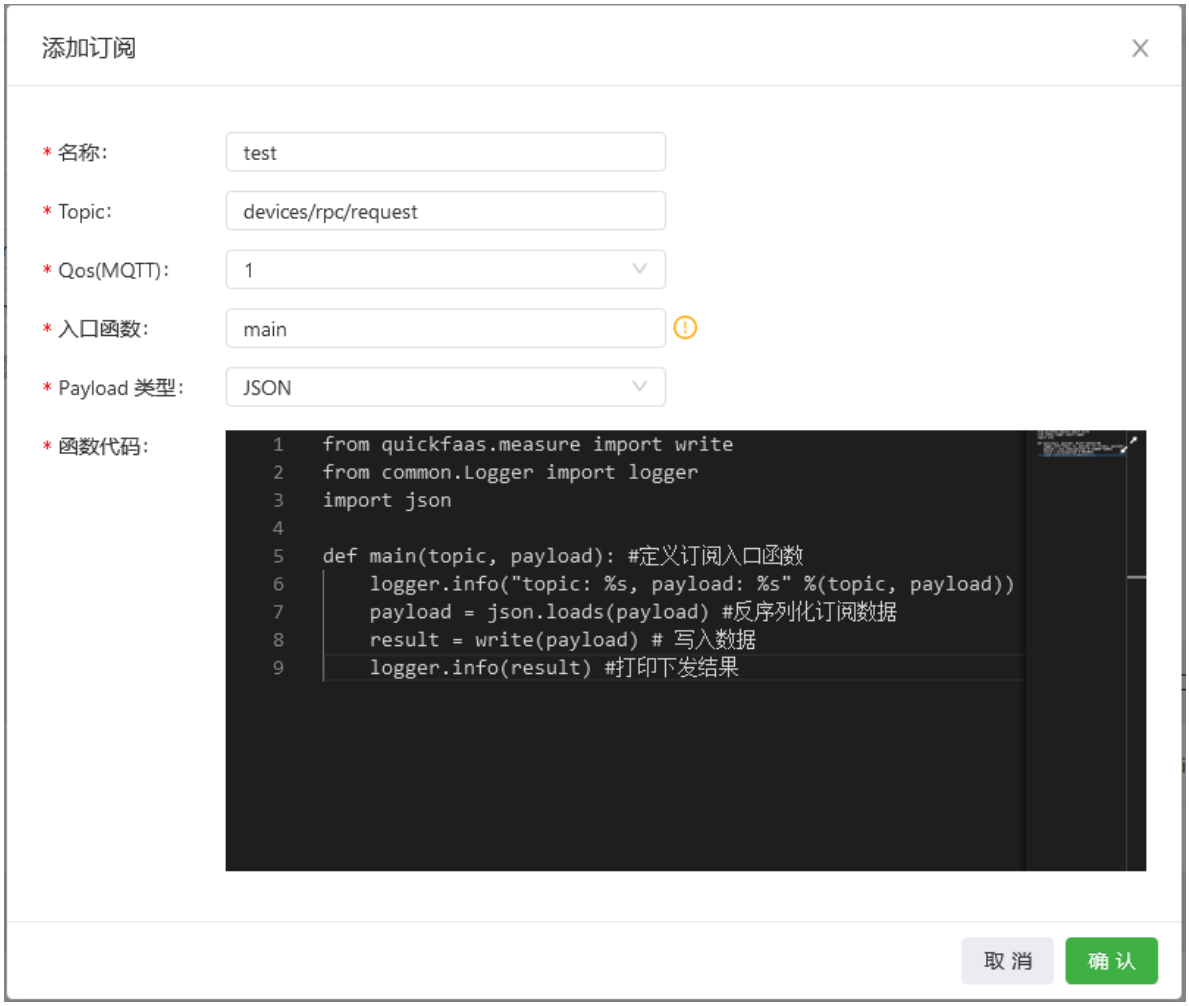


- 订阅消息:
  - 名称：任意不重复的名称

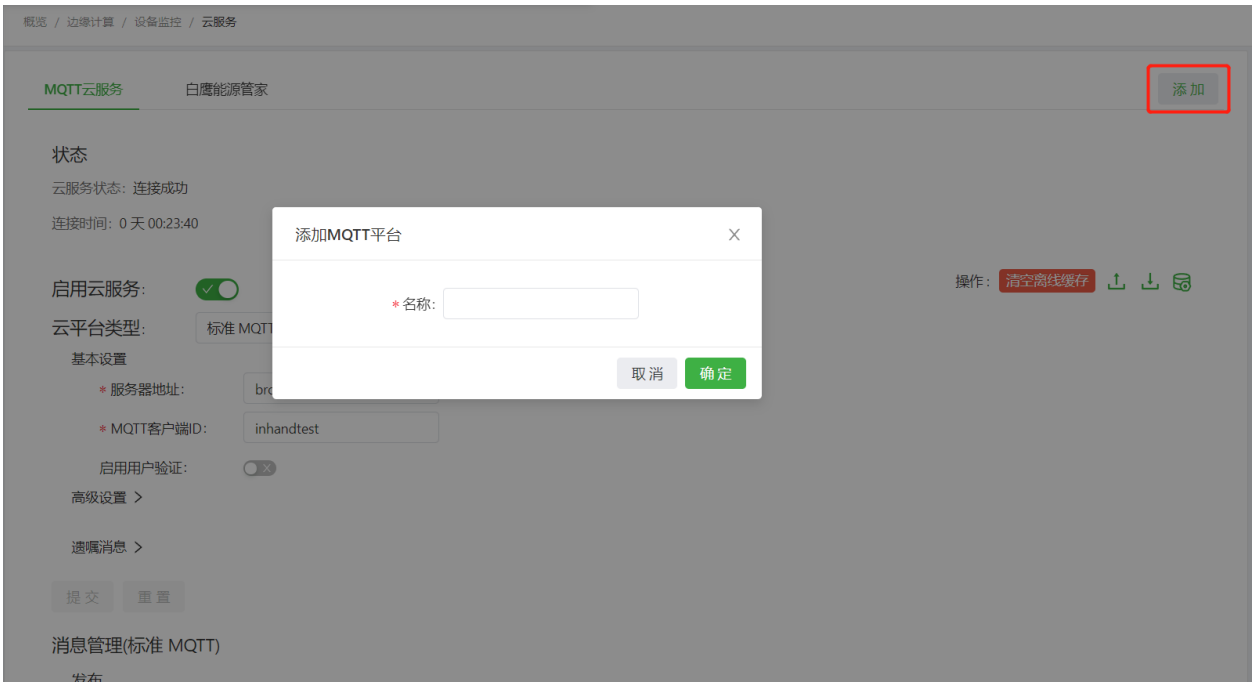
- Topic：devices/rpc/request
- Qos(MQTT)：1
- 入口函数：入口函数名称，本文档为 main
- Payload 类型：负载类型，本文档为 JSON
- 函数代码：

```
from quickfaas.measure import write_plc_values
from common.Logger import logger
import json

def main(topic, payload): #定义订阅入口函数
    logger.info("topic: %s, payload: %s" %(topic, payload))
↪#打印订阅主题和数据,假定payload数据为{"method":"setValue","Device":"Modbus_
↪test", "TagName":"SP1", "TagValue":12.3}
    payload = json.loads(payload) #反序列化订阅数据
    if payload["method"] == "setValue": #检测是否为写入数据
        message = {payload["Device"]: {payload["TagName"]:payload["TagValue"]}
↪} #定义下发消息，包括下发的变量名称和变量值
        write_plc_values(message) # 写入数据
```

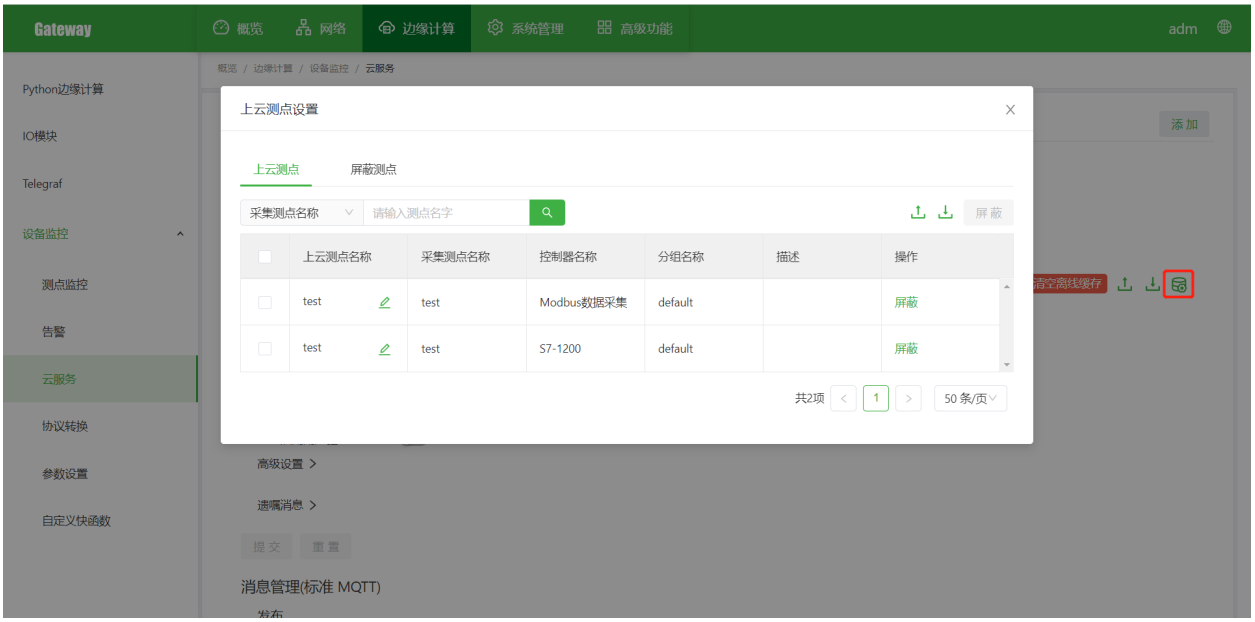
配置完成后如下图所示：



点击“添加”按钮可以添加额外的 MQTT 连接。



点击“上云测点设置”可以设置当前 MQTT 连接中不需要上报的测点（屏蔽测点）或者修改上报时的测点名称。



3.3 远端 SCADA 数据监控

3.3.1 配置协议转换 Slave/Server

在“映射值设置”中，选择“数据原始值”则映射的测点值为不经过数据运算时的数值；选择“数据运算值”则映射的测点值为经过数据运算时的数值。

在添加映射时，也可以按照自己的需求将测点原始的数据类型转换为其他数据类型。如将 WORD 数据类型转换为 FLOAT

数据类型。

- 配置 Modbus TCP Slave

  进入“协议转换 > Modbus TCP Slave > 配置”页面，点击“启用”Modbus TCP Slave。启用后，配置相应的通讯参数，示例如下：

- 配置 IEC104 Server

  进入“协议转换 > IEC104 Server > 配置”页面，点击“启用”IEC104 Server。启用后，配置相应的通讯参数，示例如下：

- 配置 OPC UA Server

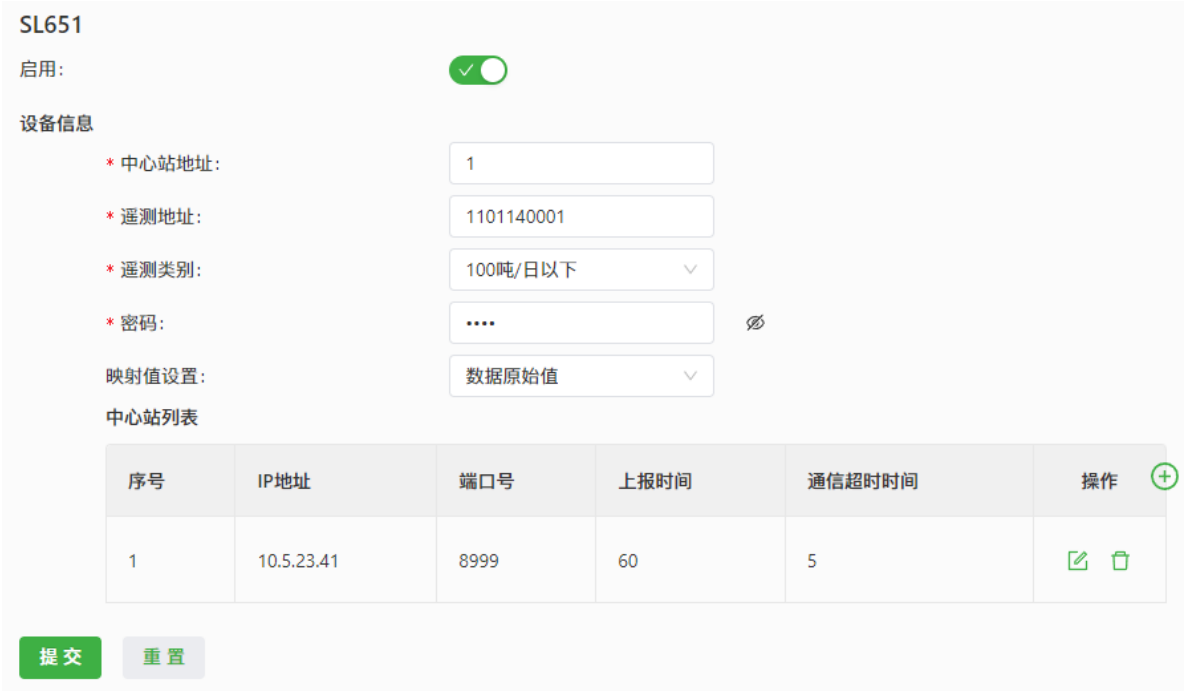
进入“协议转换 > OPC UA Server > 配置”页面，点击“启用”OPC UA Server。启用后，配置相应的通讯参数，示例如下：

- 配置 Modbus RTU Slave

  进入“协议转换 > Modbus RTU Slave > 配置”页面，点击“启用”Modbus RTU Slave。启用后，配置相应的通讯参数，示例如下：

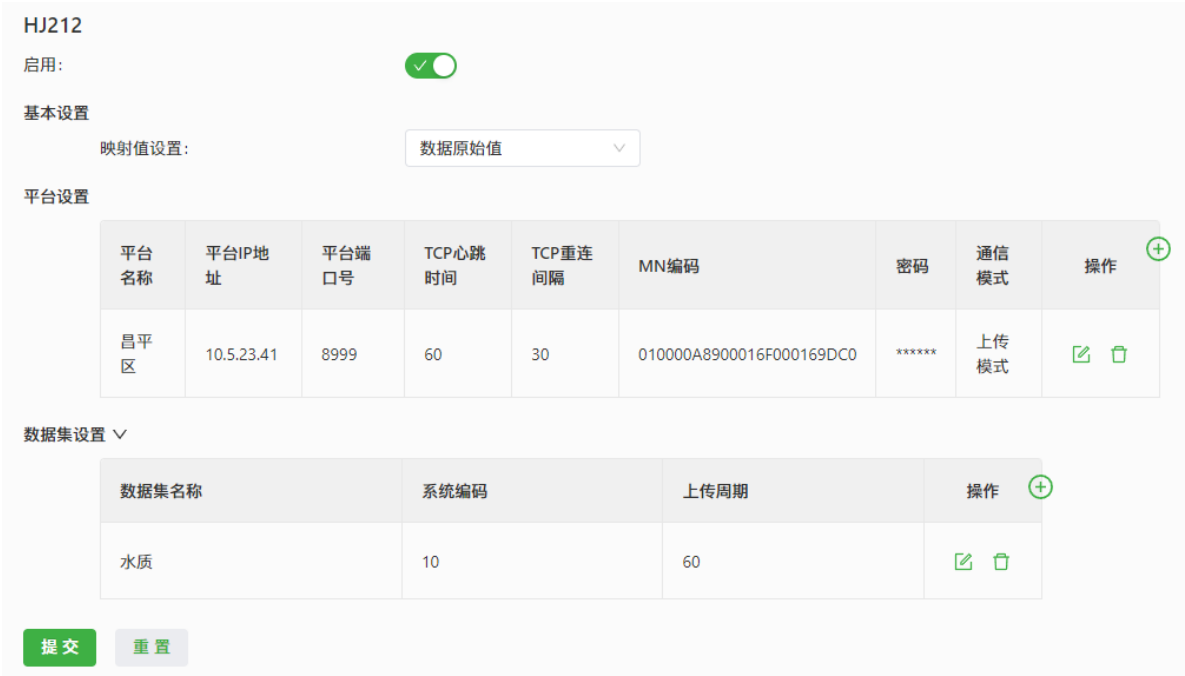
- 配置 SL651

  进入“协议转换 > SL651 > 配置”页面，点击“启用”SL651。启用后，配置相应的通讯参数，示例如下:



- 配置 HJ212 Client

  进入“协议转换 > HJ212 Client > 配置”页面，点击“启用”HJ212 Client。启用后，配置相应的通讯参数，示例如下：



- 配置 BACnet IP Server

  进入“协议转换 > BACnet IP Server > 配置”页面，点击“启用”BACnet IP Server。启用后，配置相应的通讯参数，示例如下：

3.3.2 配置映射表

- 配置 Modbus 映射表

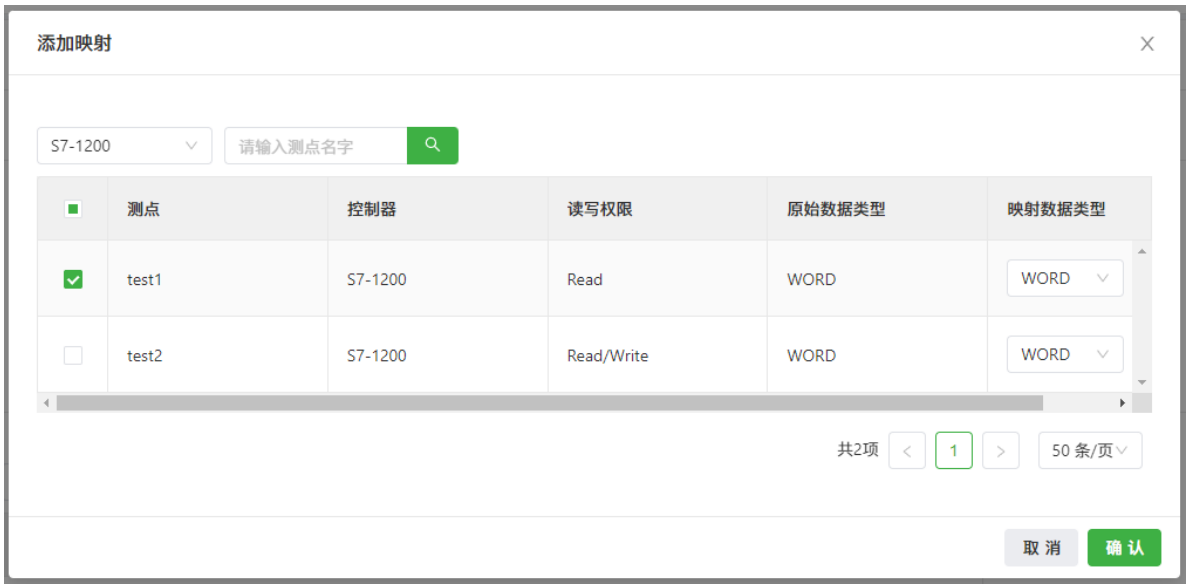
  以添加单个 Modbus 映射为例，点击“添加”按钮，在弹出框中选择相应的测点并设置映射地址，示例如下：

- 配置 IEC 101/104 映射表

  以添加单个 IEC 101/104 映射为例，点击“添加”按钮，在弹出框中选择相应的测点并设置映射地址，示例如下：

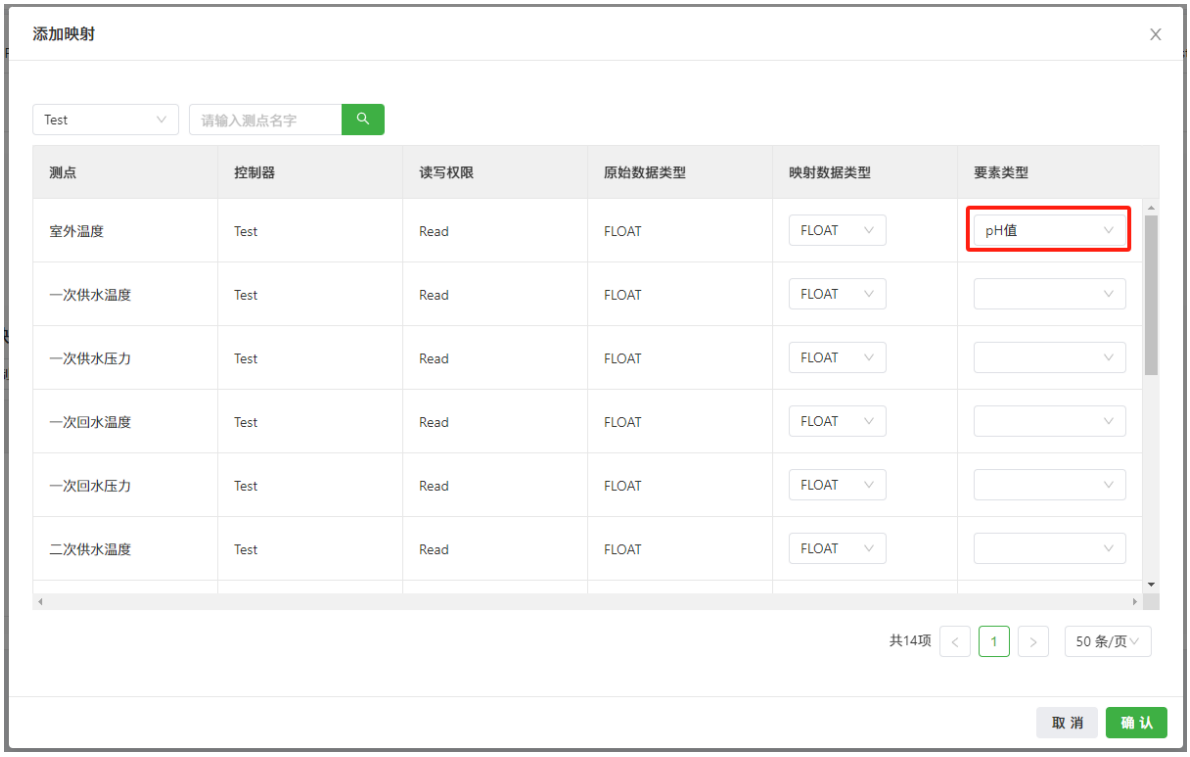
- 配置 OPC UA 映射表

  以添加单个 OPC UA 映射为例，点击“添加”按钮，在弹出框中选择相应的测点即可，示例如下：



- 配置 SL651 映射表

  以添加 pH 值映射为例，点击“添加”按钮，在弹出框中为相应的测点选择 pH 值即可，示例如下：



- 配置 HJ212 映射表

  以添加水质映射为例，点击“添加”按钮，在弹出框中为相应的测点选择水质数据集并设置监控因子编码即可，示例如下：

- 配置 BACnet 映射表

  以添加单个 BACnet 映射为例，点击“添加”按钮，在弹出框中选择相应的测点并设置映射地址，示例如下：

- 配置 DNP3 映射表

  以添加单个 DNP3 映射为例，点击“添加”按钮，在弹出框中选择相应的测点并设置映射地址，示例如下：

3.4 数据边缘处理

简单的数据处理可以在“添加测点”时通过配置“数据运算”功能解决，如需进行复杂的业务数据处理则可以通过“自定义快函数”解决。进入“边缘计算 > 设备监控 > 自定义快函数”页面，点击“添加”按钮，在弹出框中配置业务逻辑：

- 名称：快函数名称，名称不能重复。
- 模式：支持两种模式
  - 周期触发：按照时间周期性触发快函数
  - 本地订阅消息触发：网关本地的 MQTT 服务器接收到指定 topic 的消息后触发执行快函数
  - 快函数启动触发：DSA 重启或者每当快函数有变更时运行一次。
- 周期：模式为“周期触发”时的触发间隔
- 订阅 Topic：模式为“本地订阅消息触发”时，触发执行快函数的 topic
- 入口函数：入口函数名称
- 函数代码：使用 Python 代码编写的业务逻辑

示例配置如下：

1.1.5 附录

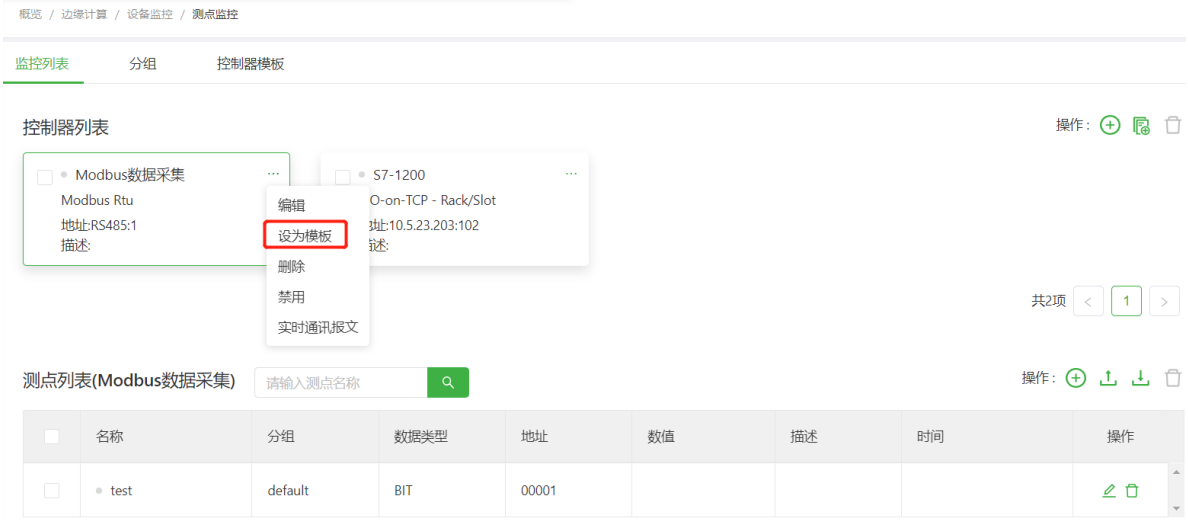
控制器相关功能说明

在“监控列表”页面，控制器支持以下功能：

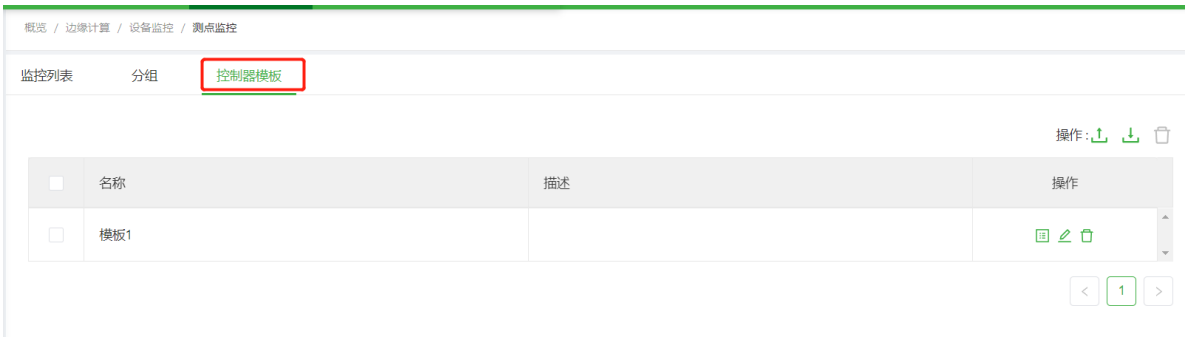
- 实时通讯报文: 支持查看和下载控制器的通讯报文。

- 设为模板：支持将控制器的通讯参数和测点采集配置存储为模板，通过模板快速完成多个类似控制器的添加。比如采集多台电表的数据。

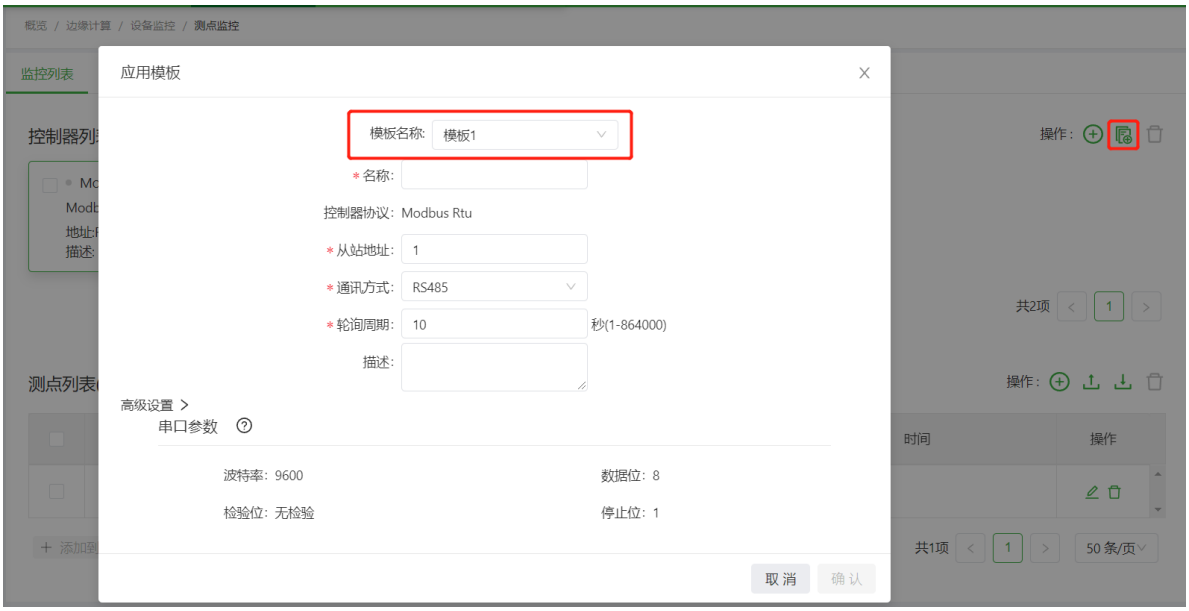
  完成单台控制器的采集配置后点击“设为模板”。



可以在“控制器模板”页面管理模板。



在“监控列表”可以点击“应用模板”快速配置控制器和测点。



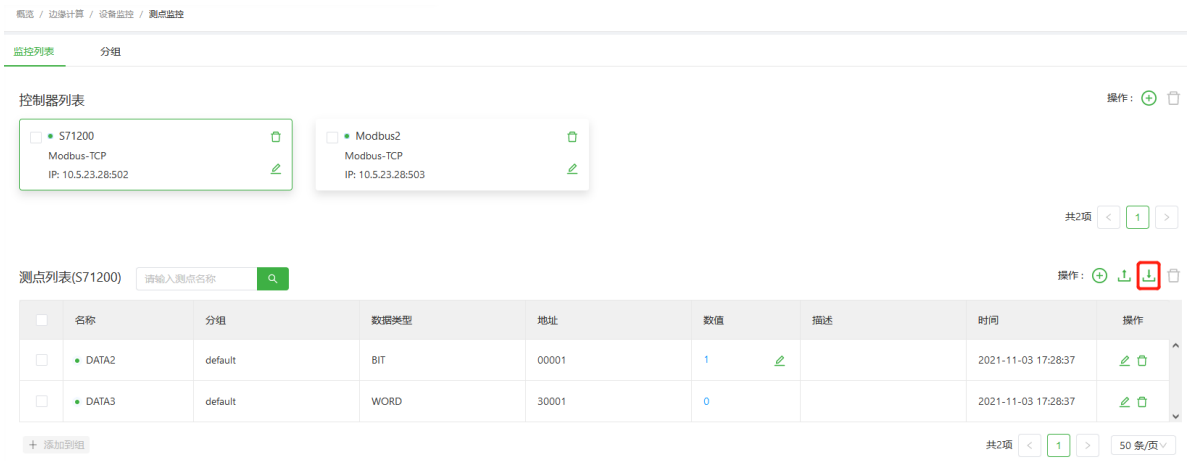
导入导出数据采集配置

DSA 支持导入导出部分 CSV 格式的配置文件，您可以通过导入导出配置文件快速实现采集配置。各配置文件内容如下：

- `measure.csv`: 测点配置文件，详细参数如下
  - `MeasuringPointName`：测点名称
  - `ControllerName`：测点所属设备
  - `GroupName`：测点所属分组
  - `UploadType`：上传模式，包括 `realtime`、`onchange`、`never`
  - `DeadZonePercent`：变化死区
  - `Data Type`：数据类型

- ArrayIndex：数组索引
- EnableBit：启用按位取值
- BitIndex：取位索引
- reverseBit：BIT 数值取反。0 为不取反，1 为取反
- Address：测点地址
- Decimal：小数位，1~6
- Len：字符串长度
- CodeType：字符串编码格式，支持 ASCII、UTF-8、UTF-16、UTF-16-BIG、GB2312
- ReadWrite：读写权限，包括 Read/Write、Read
- Unit：单位
- Description：描述
- Transform Type：数据运算类型，包括 gain（比例换算）、zoom（偏移及缩放）、bit（位截取）、none（无）
- MaxValue：Transform Type 为 gain 时有效，测点的数据上限值
- MinValue：Transform Type 为 gain 时有效，测点的数据下限值
- MaxScale：Transform Type 为 gain 时有效，比例运算后的数据上限值
- MinScale：Transform Type 为 gain 时有效，比例运算后的数据下限值
- Gain：Transform Type 为 zoom 时有效，需要放大或缩小的倍率
- Offset：Transform Type 为 zoom 时有效，倍率运算后需要增加或减少的数据值
- startBit：Transform Type 为 bit 时有效，开始截取原始数据的位偏移
- endBit：Transform Type 为 bit 时有效，结束截取原始数据的位偏移
- Pt：Transform Type 为 ptct 时有效，PT 值
- Ct：Transform Type 为 ptct 时有效，CT 值
- bitMap：仅对 BIT 类型测点生效，为 1 时将 BIT 型测点的 0 和 1 值映射为 False 和 True
- msecSample：组包上报。为 0 时不启用，为 1 时启用
- storageLwTSDB：存储历史数据。为 0 时不存储，为 1 时存储
- pollCycle：启用多轮询周期后。为 0 时使用默认的轮询周期，为 1 时使用轮询周期 2

导出方式为监控列表页面的测点列表导出。



示例配置如下：

MeasuringPointName	ControllerName	GroupName	UploadType	DataType	Address	Decimal	Len	ReadWrite	Unit	Description	Transform Type	MaxValue	MinValue	MaxScale	MinScale	Gain	Offset	startBit	endBit	bitMap
DATA2	S71200	default	periodic	BIT	0X0001			rw			none									0
DATA3	S71200	default	periodic	WORD	3X0001			ro			zoom					1	0			

- `group.csv`: 分组配置文件，详细参数如下
  - `GroupName`：分组名称
  - `UploadInterval`：上报周期
  - `EnablePerOnchange`：启用周期上报 Onchange 数据。为 0 时不器用，为 1 时启用
  - `OnchangePeriod`：启用周期上报 Onchange 数据后，Onchange 数据的固定上报周期
  - `LwTSDBSize`：历史数据最大存储条数
  - `strategy`：历史数据存储策略。为 1 时代表“与上报周期同步”，为 2 时代表“独立存盘周期”
  - `storagePeriod`：历史数据存储周期
  - `historyDataPath`:历史数据存储路径。网关存储路径为“/var/user/data/dbhome/device_supervisor/LwTSDB”

导出方式为分组页面的分组导出。

示例配置如下：

GroupName	UploadInterval	EnablePerOnchange	OnchangePeriod	LwTSDBSize	strategy	storagePeriod	historyDataPath
default	10	0		1000	1		/var/user/data/dbhome/device_supervisor/LwTSDB

- alarm.csv：告警规则配置文件，详细参数如下：
  - AlarmName：告警名称
  - ControllerName：告警测点所属控制器
  - MeasuringPointName：引用的测点名称
  - AlarmLevel：告警等级。1：提醒，2：警告，3：次要，4：重要，5：严重
  - Condition1：告警条件 1。Eq：等于,Neq：不等于,Gt：大于,Gne：大于等于,Lne：小于等于,Lt：小于
  - Operand1：告警阈值 1
  - CombineMethod：告警条件连接方式。None：空，And：&&，Or：||
  - Condition2：告警条件 2
  - Operand2：告警阈值 2
  - AlarmContent：告警内容
  - AlarmTag：告警标签

导出方式为告警策略页面的告警导出。

消息管理（自定义 MQTT 发布/订阅）

您可以在“边缘计算 > 设备监控 > 云服务”配置你的 MQTT 连接参数，通过消息管理功能配置上报数据的 MQTT 主题、数据来源等参数并支持使用 Python 语言自定义 MQTT 发布和订阅消息的数据上报、处理等逻辑。无需二次开发即可实现与多种 MQTT 服务器进行数据上传和下发。以下将为您说明“消息管理”的使用方法。

配置发布消息

自定义发布消息中包含以下配置项：

- 名称：用户自定义发布名称
- 数据源类型：需要发布的数据类型
  - 测点数据：发布指定分组下的测点数据
  - 告警数据：发布指定告警标签下的告警数据
- 分组/标签：
  - 分组：选择相应的分组后，分组下所有测点通过该发布配置将数据上传至 MQTT 服务器。可选择多个分组，当选择多个分组时，按照分组的上报周期分别对各分组下的测点执行发布中的脚本逻辑。分组中必须包含测点，否则不会执行发布中的脚本逻辑
  - 告警标签：选择相应的告警标签后，告警标签下所有告警通过该发布配置将数据上传至 MQTT 服务器，可选择多个告警标签。告警触发后立即上报
- Topic：发布主题，与 MQTT 服务器订阅的主题保持一致
- Qos(MQTT)：发布 Qos，建议与 MQTT 服务器的 Qos 保持一致
  - 0：只发送一次消息，不进行重试
  - 1：最少发送一次消息，确保消息到达 MQTT 服务器
  - 2：确保消息到达 MQTT 服务器且只收到一次
- 入口函数：入口函数名称，与函数代码中的入口函数名称保持一致
- 函数代码：使用 Python 代码自定义组包和处理逻辑，发布中的入口函数参数包括：
  - 参数 1：DSA 将采集后的测点/告警数据发送给该参数，数据格式如下：
    - 测点数据格式：

```
{
    'timestamp': 1646966325,  #数据产生时间戳
    'group_name': 'default',  #采集组名称
    'values':  #变量数据字典，包含PLC名称，变量名称和变量值
    {
        'S7-1200':  #PLC名称
        {
            'Test1':  #变量名称
            {
                'raw_data': False,  #变量值
                'timestamp': 1646966325,  #数据产生时间戳
                'status': 1  #采集状态，非1即采集异常
            },
```

(续下页)

(接上页)

```
            'Test2':
            {
                'raw_data': 2,
                'timestamp': 1646966325,
                'status': 1
            }
        }
    }
}
```

* 告警数据格式：

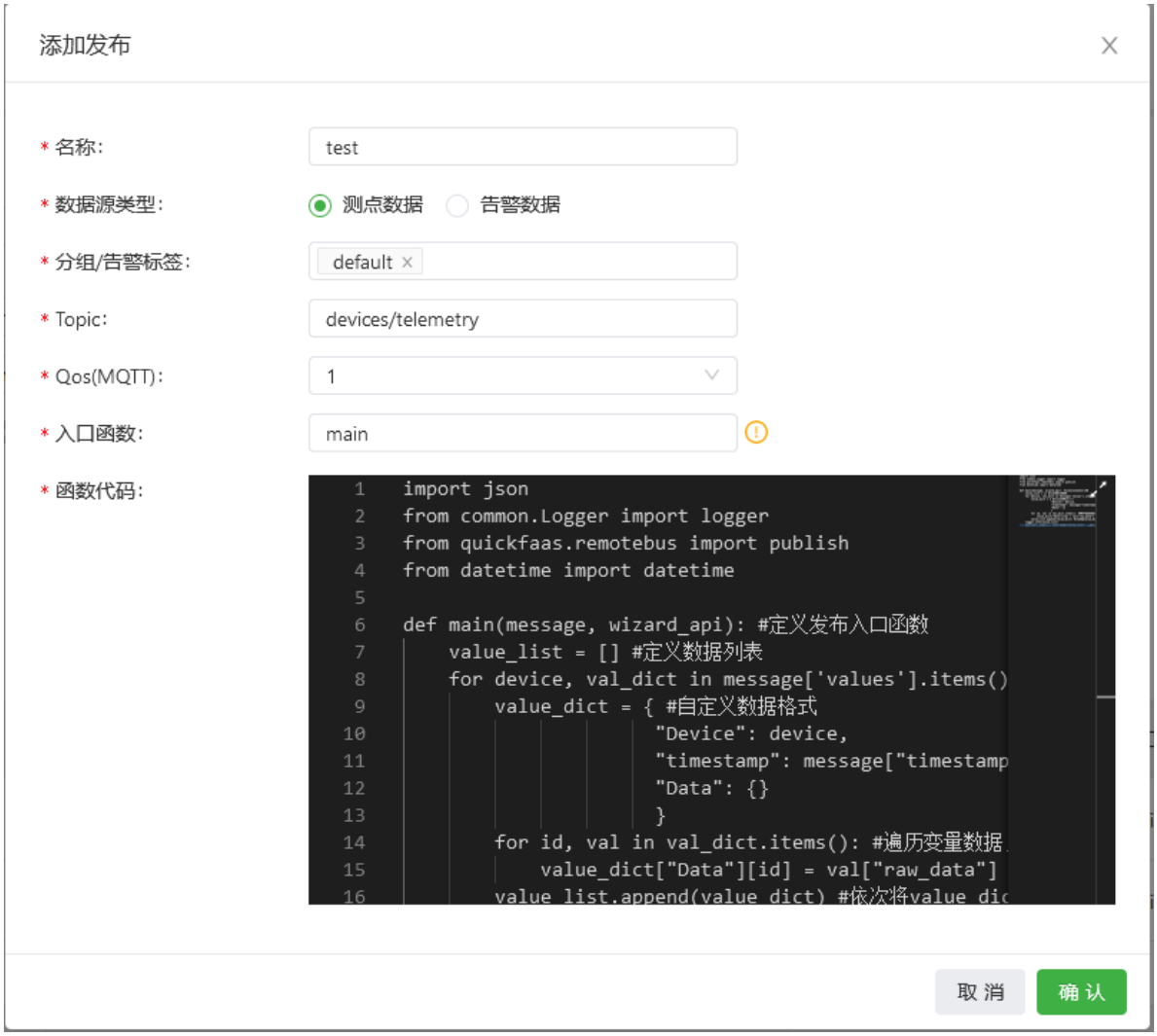
```
{
        'timestamp': 1641975441,  #告警产生时间戳
        'group_name': 'default',  #告警组名称
        'values': {  #告警数据字典，包含告警名称等告警信息
                'Warn': {  #告警名称
                        'ctrlName': 'S71200',  #告警控制器名称
                        'measureName': 'DATA2',  #告警测点名称
                        'timestamp': 1641975441,  #告警产生时间戳
                        'current': 'on',  #告警状态。on:已触发，off:已消除
                        'status': 0,  #告警状态。0:已触发，1:已消除
                        'value': 1,  #告警触发时告警变量的数值
                        'alarm_content': 'Test',  #告警描述
                        'level': 1  #预留字段
                }
        }
}
```

– 参数 2：兼容 DS 1.0 版本 Device Supervisor 提供的 api 接口。不兼容 save_data api 接口。脚本中需要提供此参数，否则会影响参数 1 的格式

以下是常见的自定义发布方法示例：

- 发布示例 1：使用 publish 上传测点数据

  本示例实现了使用 publish 上传测点数据，将处理后的测点数据使用 publish 命令上传至 MQTT 服务器。当 qos 不为 0 时，如果发送失败则缓存测点数据并等待 MQTT 连接正常后按先存先传的顺序上传至 MQTT 服务器。发布和代码配置示例如下：




```
import json
from common.Logger import logger
from quickfaas.remotebus import publish
from datetime import datetime

def main(message, wizard_api): #定义发布入口函数
    value_list = [] #定义数据列表
    for device, val_dict in message['values'].items():
↪#遍历values字典，该字典中包含设备名称和设备下的变量数据
        value_dict = { #自定义数据格式
                       "Device": device,
                       "timestamp": message["timestamp"],
                       "Data": {}
                       }
        for id, val in val_dict.items(): #遍历变量数据，为数据字典赋值
            value_dict["Data"][id] = val["raw_data"]
```

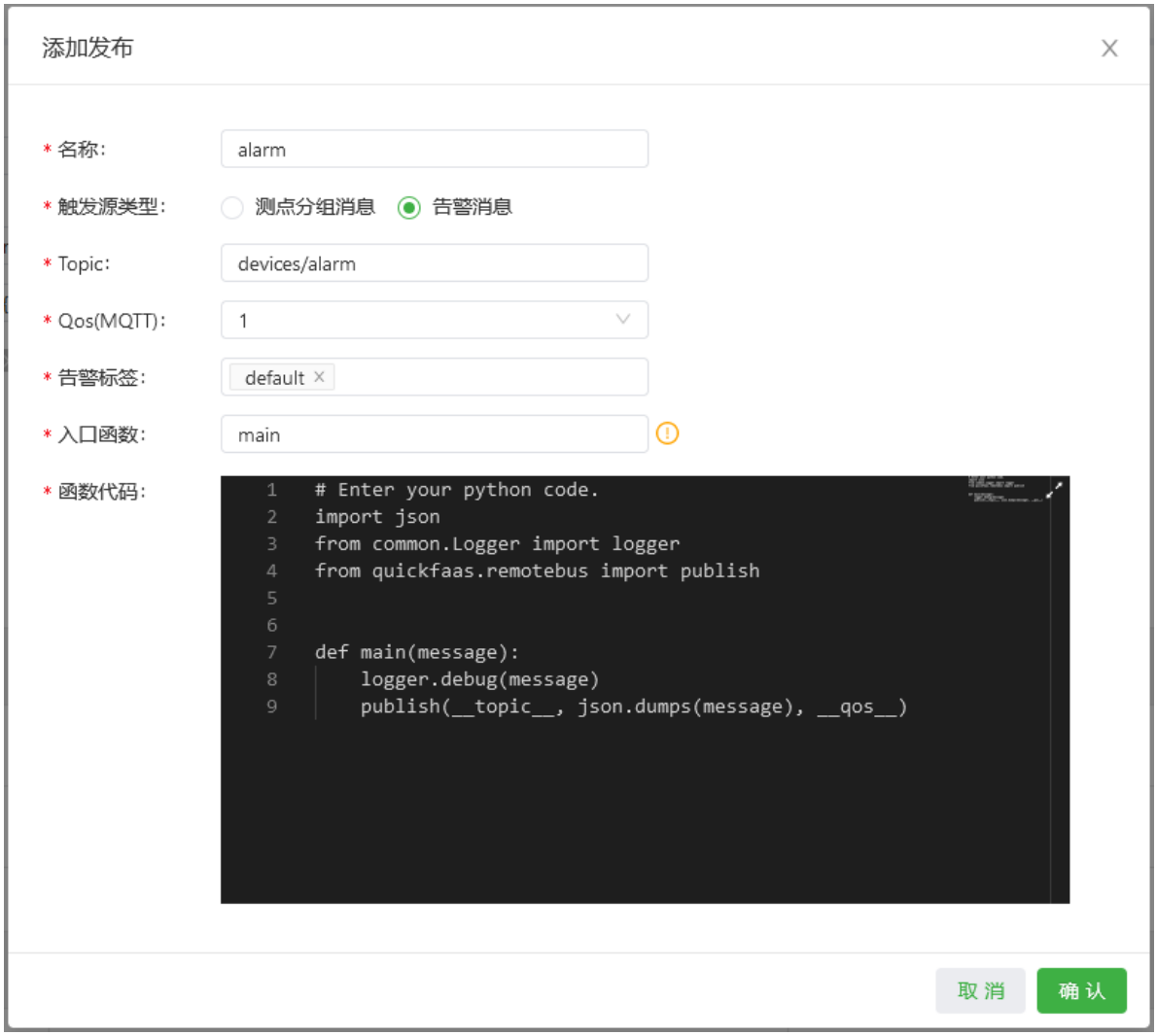
(续下页)

(接上页)

```
        value_list.append(value_dict) #依次将value_dict添加到value_list中
    logger.info(value_list)
    publish(__topic__, json.dumps(value_list), __qos__)
↪#使用发布消息中定义的主题和qos发送value_
↪list至云平台。当qos不为0时，数据发送失败则缓存数据等待连接恢复后按先存先传的顺序上传至MQTT服务
```

- 发布示例 2：使用 `publish` 上传告警数据

  本示例实现了使用 `publish` 上传告警数据，将处理后的告警数据使用 publish 命令上传至 MQTT 服务器。当 qos 不为 0 时，如果发送失败则缓存告警数据并等待 MQTT 连接正常后按先存先传的顺序上传至 MQTT 服务器。发布和代码配置示例如下：




```
import json
from common.Logger import logger
```

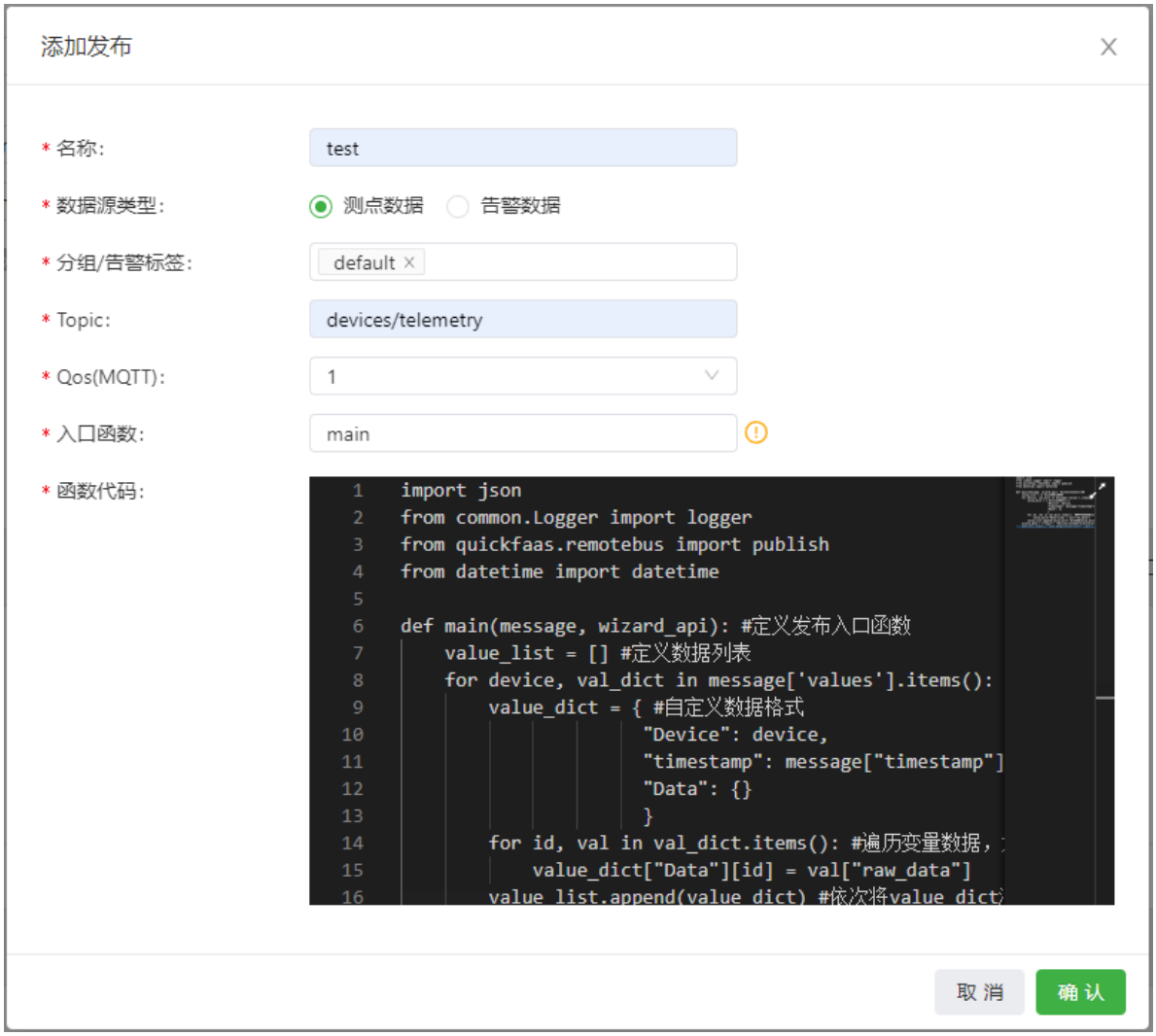
(续下页)

(接上页)

```
from quickfaas.remotebus import publish


def main(message, wizard_api): #定义发布入口函数
    logger.info(message) #打印告警数据
    publish(__topic__, json.dumps(message), __qos__)
↪#使用发布消息中定义的主题和qos发送message至云平台。当qos不为0时，数据发送失败则缓存数据等待连接
```

- 发布示例 3：使用 `publish` 上传测点数据并使用新的 topic 上报历史数据

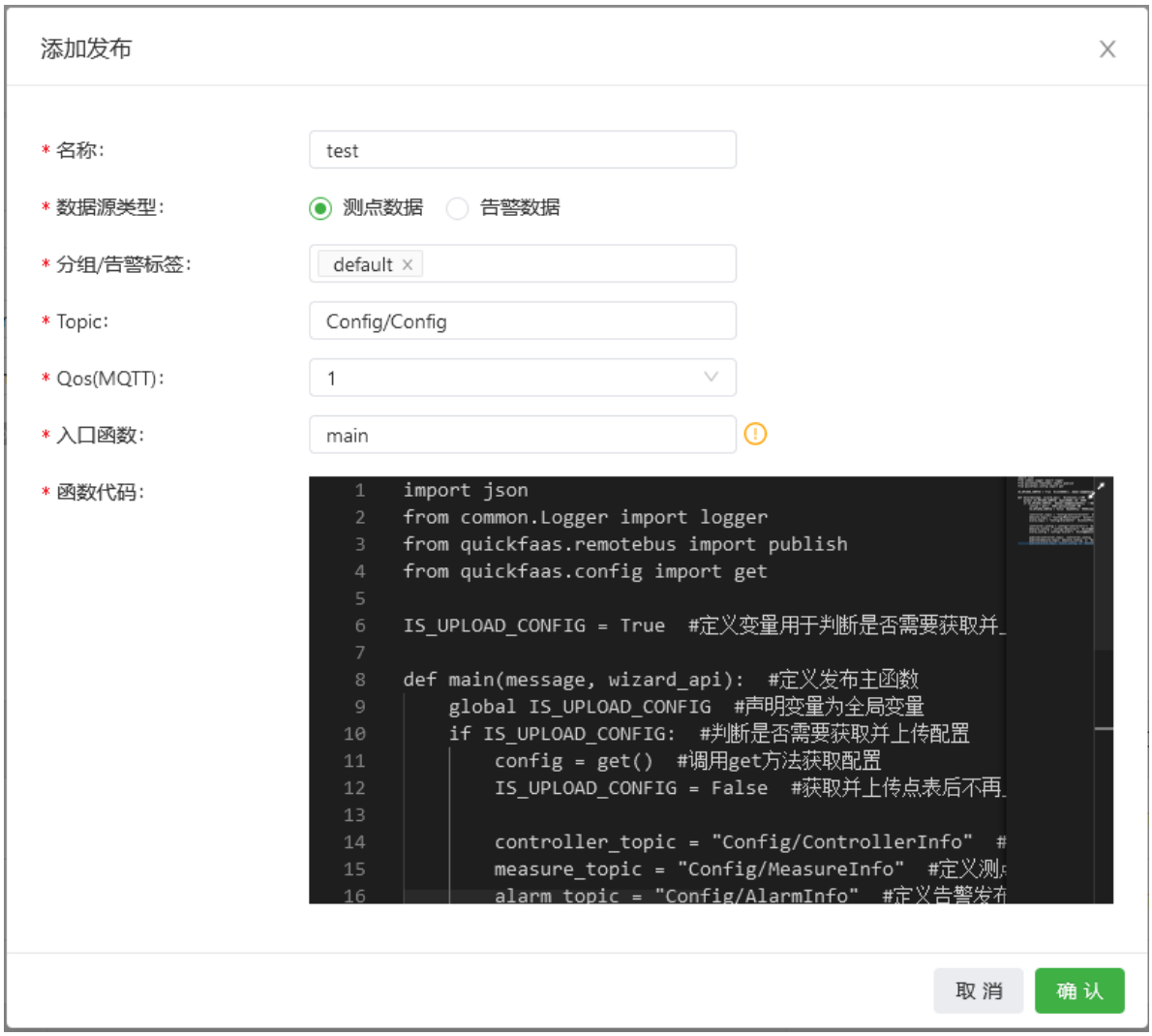
  本示例实现了使用 `publish` 上传测点数据，将处理后的测点数据使用 publish 命令上传至 MQTT 服务器。如果发送失败则缓存测点数据并等待 MQTT 连接正常后使用历史数据的 topic 上传至 MQTT 服务器。发布和代码配置示例如下：



```
import json
from common.Logger import logger
from quickfaas.remotebus import publish
from datetime import datetime

def main(message, wizard_api): #定义发布入口函数
    value_list = [] #定义数据列表
    for device, val_dict in message['values'].items():
↪#遍历values字典，该字典中包含设备名称和设备下的变量数据
        value_dict = { #自定义数据格式
                        "Device": device,
                        "timestamp": message["timestamp"],
                        "Data": {}
                        }
        for id, val in val_dict.items(): #遍历变量数据，为数据字典赋值
            value_dict["Data"][id] = val["raw_data"]
        value_list.append(value_dict) #依次将value_dict添加到value_list中
    wizard_data = {"topic": "devices/telemetry/history", "qos": 1, "payload":␣
↪json.dumps(value_list)} #定义历史数据重发信息
    publish(__topic__, json.dumps(value_list), __qos__, wizard_data = wizard_
↪data) #使用发布消息中定义的主题和qos发送value_
↪list至云平台。当qos不为0时，数据发送失败则缓存数据等待连接恢复后按采集时间顺序上传至MQTT服务器
↪data定义的topic、qos和负载上报
```

- 发布示例 4：使用 `get` 获取 DSA 配置信息

  本示例实现了每次重启 DSA 时使用 get 方法获取控制器、测点和告警配置并分别上传至 MQTT 服务器。发布和代码配置示例如下：




```
import json
from common.Logger import logger
from quickfaas.remotebus import publish
from quickfaas.config import get

IS_UPLOAD_CONFIG = True  #定义变量用于判断是否需要获取并上传配置

def main(message, wizard_api):  #定义发布主函数
    global IS_UPLOAD_CONFIG  #声明变量为全局变量
    if IS_UPLOAD_CONFIG:  #判断是否需要获取并上传配置
        config = get()  #调用get方法获取配置
        IS_UPLOAD_CONFIG = False  #获取并上传点表后不再上传点表

        controller_topic = "Config/ControllerInfo"  #定义控制器发布主题
        measure_topic = "Config/MeasureInfo"  #定义测点发布主题
        alarm_topic = "Config/AlarmInfo"  #定义告警发布主题
```

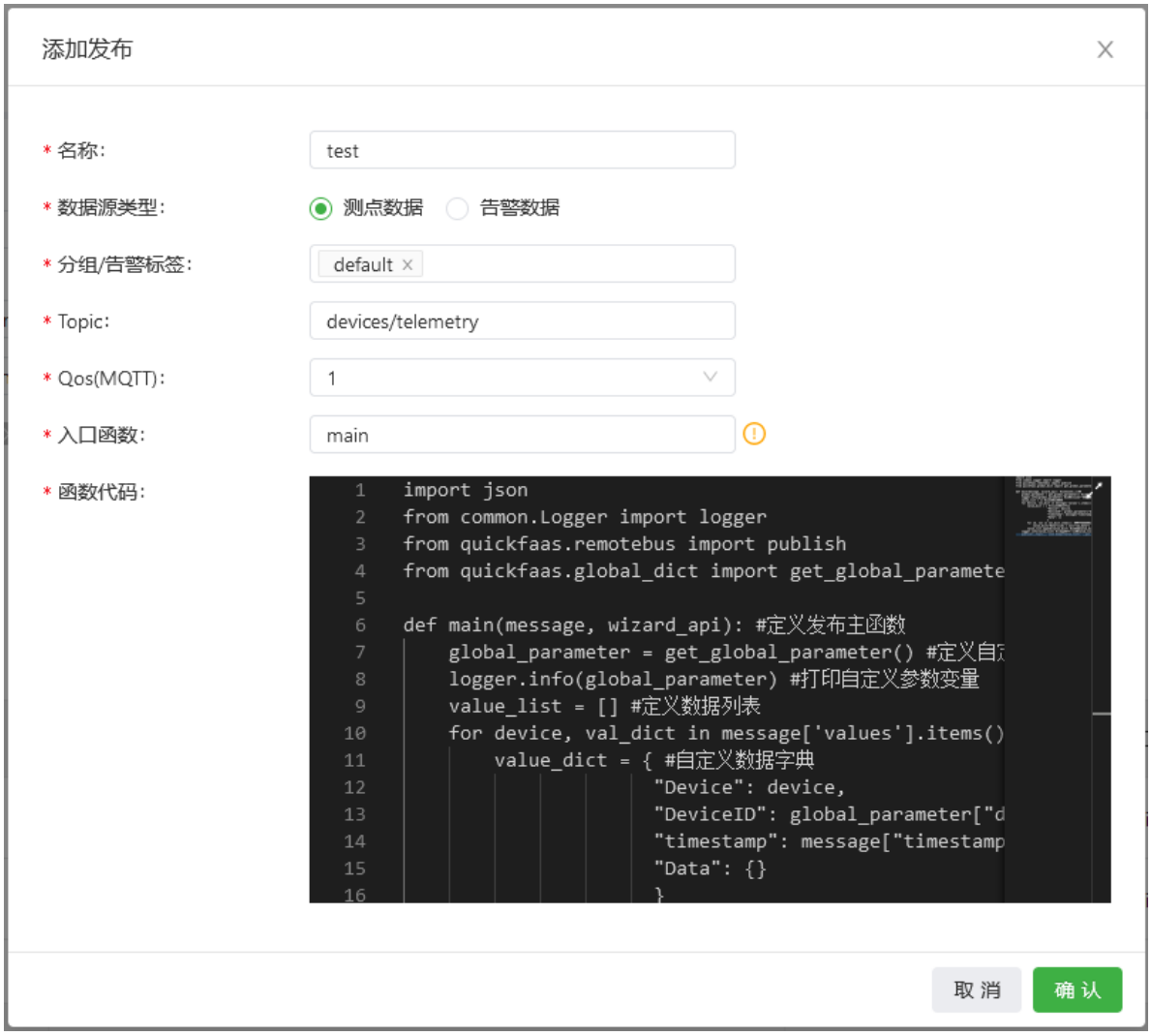
(续下页)

(接上页)

```
controller_config = config["controllers"]  #定义需要发布的控制器配置信息
measure_config = config["measures"]  #定义需要发布的测点配置信息
alarm_config = config["alarms"]  #定义需要发布的告警配置信息

publish(controller_topic, controller_config, 1)  #发布的控制器配置
publish(measure_topic, measure_config, 1)  #发布的测点配置
publish(alarm_topic, alarm_config, 1)  #发布的告警配置
```

- 发布示例 5：使用 `get_global_parameter` 获取参数设置中的自定义参数

  本示例实现了获取“参数设置”中的自定义参数 device_id，并通过通配符 ${device_id} 的配置方式配置 MQTT 主题。发布和代码配置示例如下：




```
import json
from common.Logger import logger
from quickfaas.remotebus import publish
from quickfaas.global_dict import get_global_parameter

def main(message, wizard_api): #定义发布主函数
    global_parameter = get_global_parameter() #定义自定义参数变量
    logger.info(global_parameter) #打印自定义参数变量
    value_list = [] #定义数据列表
    for device, val_dict in message['values'].items():
↪#遍历values字典，该字典中包含设备名称和设备下的变量数据
        value_dict = { #自定义数据字典
                       "Device": device,
                       "DeviceID": global_parameter["device_id"],
↪#获取自定义参数中定义的设备ID
                       "timestamp": message["timestamp"],
```

(续下页)

(接上页)

```
                    "Data": {}
                    }
        for id, val in val_dict.items(): #遍历变量数据，为Data字典赋值
            value_dict["Data"][id] = val["raw_data"]
        value_list.append(value_dict) #依次将value_dict添加到value_list中
    logger.info(value_list) #在App日志中打印value_list，数据格式为[{'Device': 'S7-
↪1200', 'DeviceID': '1', 'timestamp': 1589538347.5604711, 'Data': {'Test1':␣
↪False, 'Test2': 12}}]
    publish(__topic__, json.dumps(value_list), __qos__)
↪#使用发布消息中定义的主题和qos发送message至云平台。当qos不为0时，数据发送失败则缓存数据等待连接
```

配置订阅消息

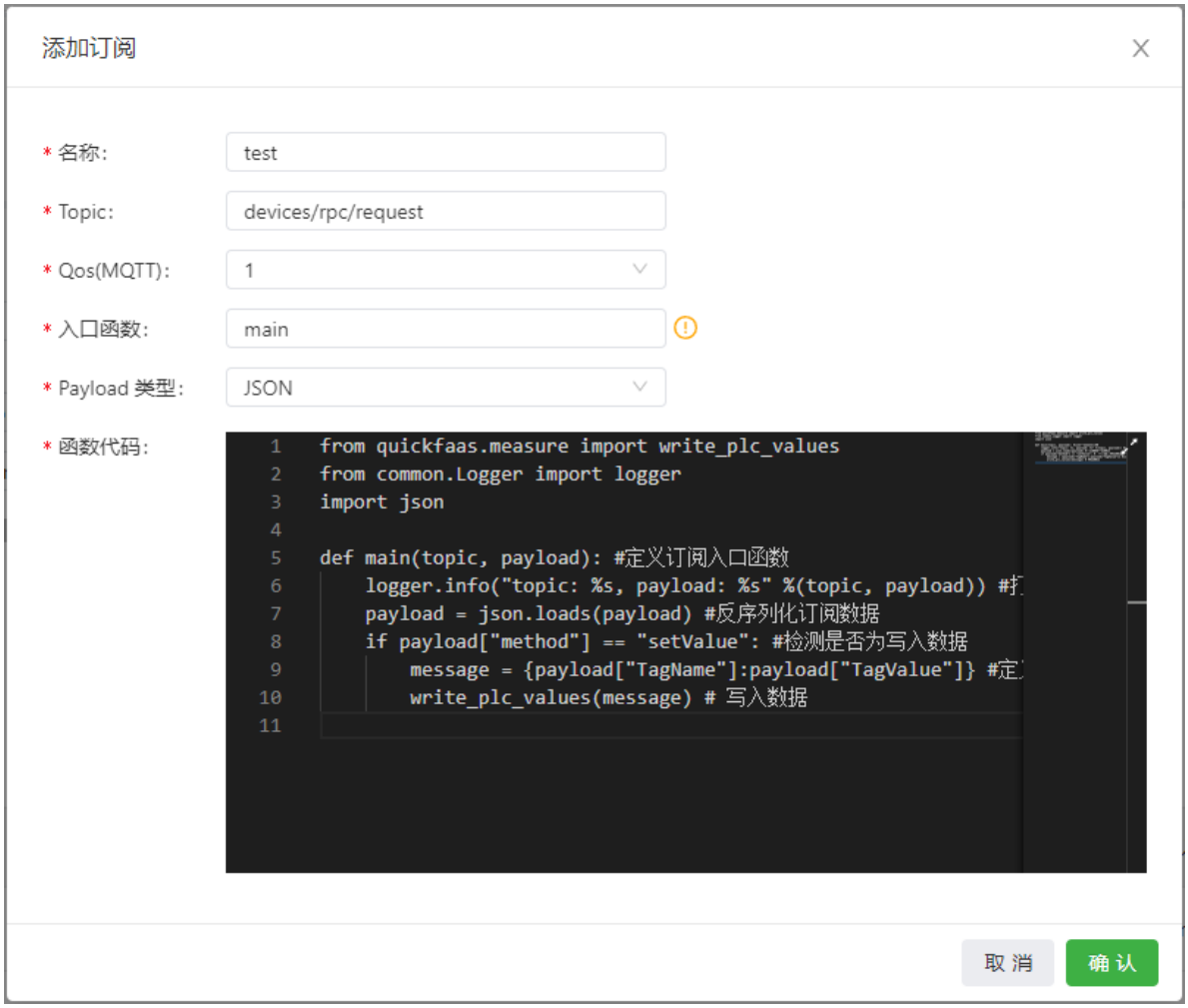
自定义订阅消息中包含以下项：

- 名称：用户自定义订阅名称
- Topic：订阅主题，与 MQTT 服务器发布的数据主题保持一致
- Qos(MQTT)：订阅 Qos，建议与 MQTT 服务器的 Qos 保持一致
- 入口函数：入口函数名称，与函数代码中的入口函数名称保持一致
- Payload 类型：Payload 类型，与云平台要求的类型保持一致。通常为 JSON
- 函数代码：使用 Python 代码自定义组包和处理逻辑，订阅中的入口函数参数包括：
  - 参数 1：该参数为接收到的主题，数据类型为 string
  - 参数 2：该参数为接收到的数据，数据类型为 string
  - 参数 3：兼容 DS 1.0 版本 Device Supervisor 提供的 api 接口。不兼容 save_data api 接口

以下是常见的自定义订阅方法示例：

- 订阅示例 1：下发测点名称和测点值写入 PLC 数据且不返回写入结果

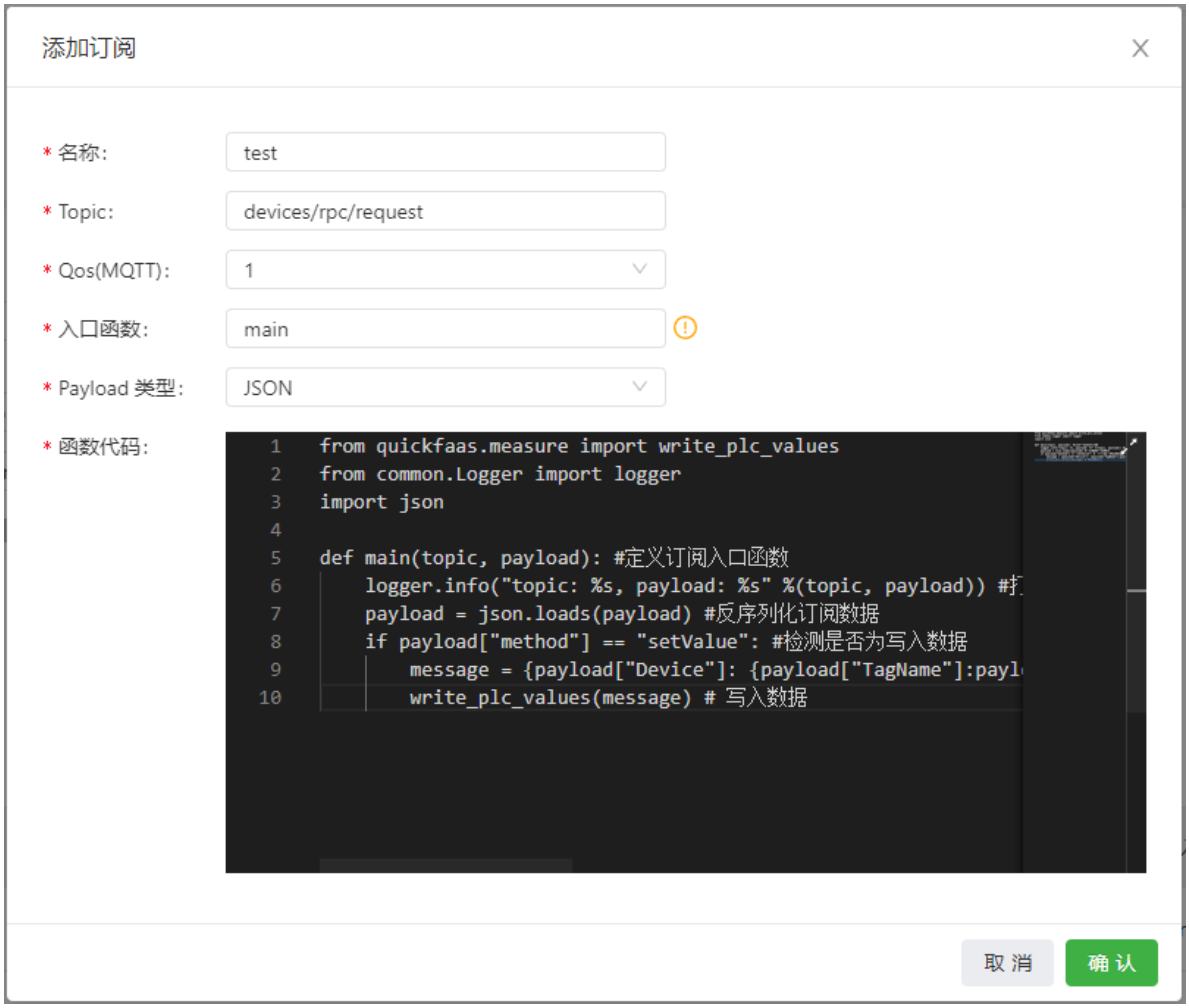
  本示例实现了从 MQTT 服务器下发指定命令修改测点数值，发布和代码配置示例如下：




```
from quickfaas.measure import write_plc_values
from common.Logger import logger
import json

def main(topic, payload): #定义订阅入口函数
    logger.info("topic: %s, payload: %s" %(topic, payload)) #打印订阅主题和数据,
↪假定payload数据为{"method":"setValue", "TagName":"SP1", "TagValue":12.3}
    payload = json.loads(payload) #反序列化订阅数据
    if payload["method"] == "setValue": #检测是否为写入数据
        message = {payload["TagName"]:payload["TagValue"]}
↪#定义下发消息，包括下发的变量名称和变量值
        write_plc_values(message) # 写入数据
```

- 订阅示例 2：下发控制器名称，测点名称和测点值写入 PLC 数据且不返回写入结果

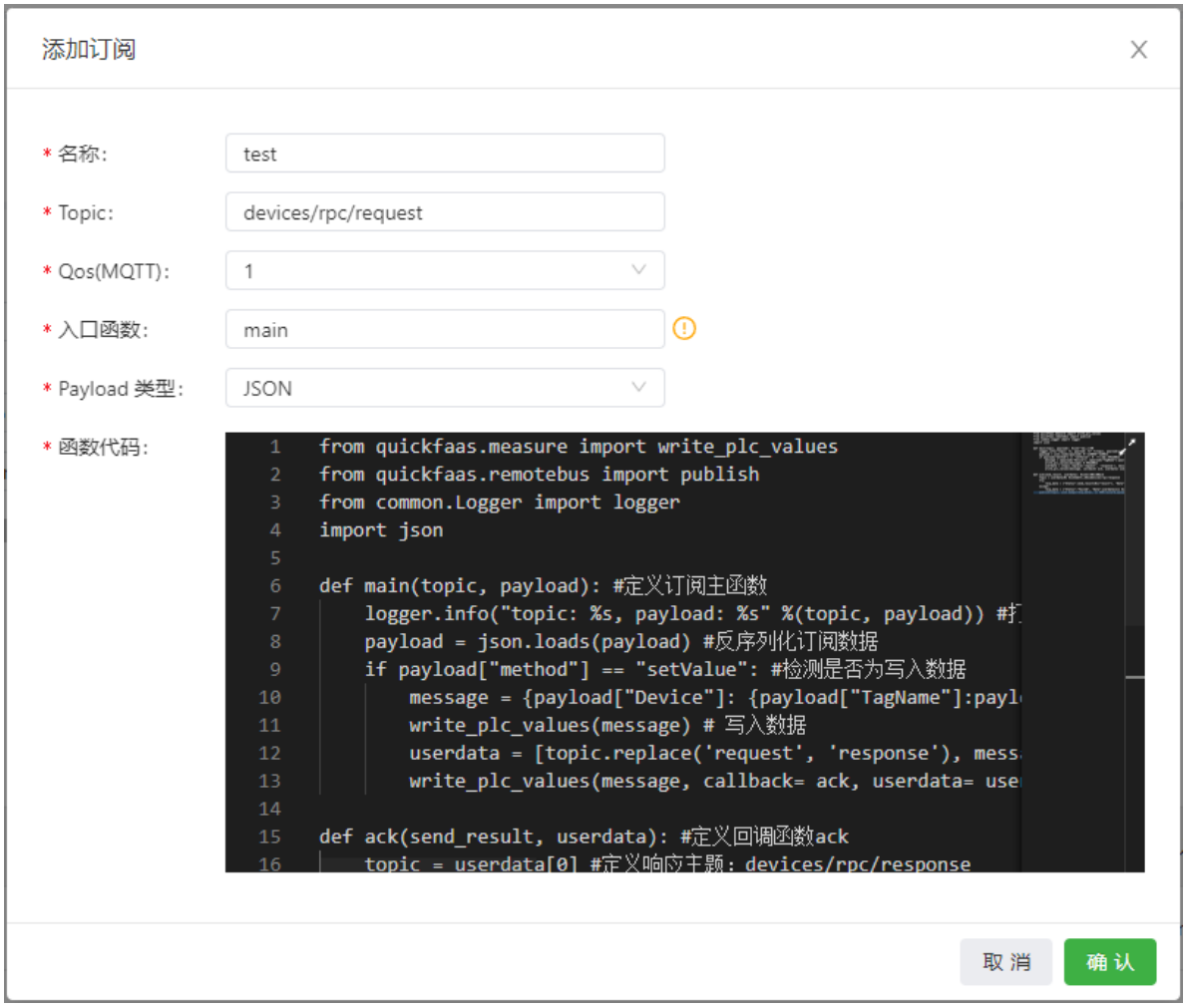
  本示例实现了从 MQTT 服务器下发指定命令修改测点数值，发布和代码配置示例如下：




```
from quickfaas.measure import write_plc_values
from common.Logger import logger
import json

def main(topic, payload): #定义订阅入口函数
    logger.info("topic: %s, payload: %s" %(topic, payload)) #打印订阅主题和数据,
↪假定payload数据为{"method":"setValue","Device":"Modbus_test", "TagName":"SP1",
↪"TagValue":12.3}
    payload = json.loads(payload) #反序列化订阅数据
    if payload["method"] == "setValue": #检测是否为写入数据
        message = {payload["Device"]: {payload["TagName"]:payload["TagValue"]}}
↪#定义下发消息，包括下发的变量名称和变量值
        write_plc_values(message) # 写入数据
```

- 订阅示例 3：下发写入 PLC 数据且返回写入结果

  本示例实现了从 MQTT 服务器下发指定命令修改测点数值并返回结果，发布和代码配置示例如下：




```
from quickfaas.measure import write_plc_values
from quickfaas.remotebus import publish
from common.Logger import logger
import json

def main(topic, payload): #定义订阅主函数
    logger.info("topic: %s, payload: %s" %(topic, payload)) #打印订阅主题和数据,
↪假定topic为devices/rpc/request;定payload数据为{"method":"setValue","Device":
↪"Modbus_test", "TagName":"SP1", "TagValue":12.3}
    payload = json.loads(payload) #反序列化订阅数据
    if payload["method"] == "setValue": #检测是否为写入数据
        message = {payload["Device"]: {payload["TagName"]:payload["TagValue"]}}
↪#定义下发消息，包括下发的变量名称和变量值
        write_plc_values(message) # 写入数据
        userdata = [topic.replace('request', 'response'), message]
↪#定义确认数据，包括响应的主题和消息
        write_plc_values(message, callback= ack, userdata= userdata, timeout =␣
```

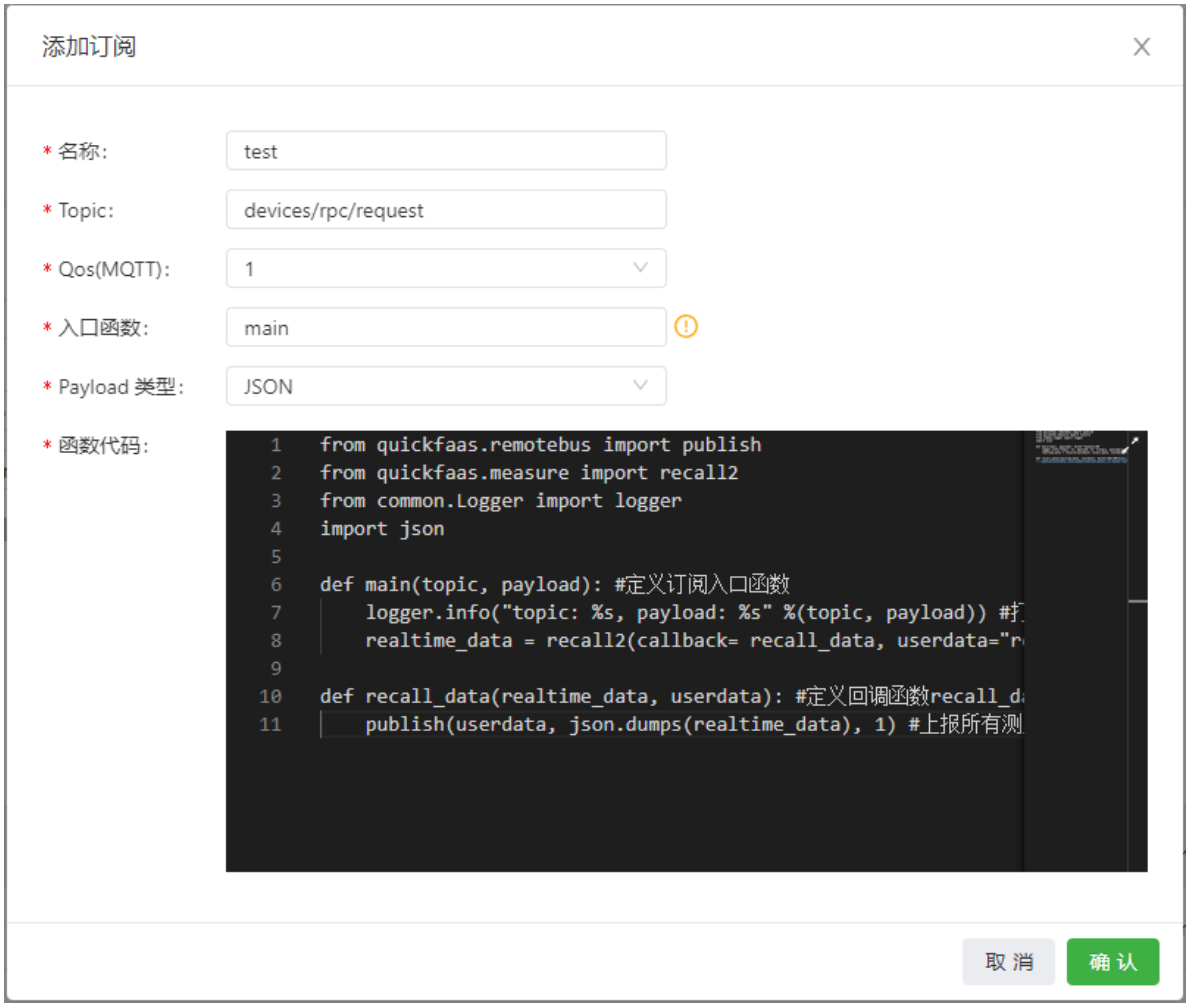
(续下页)

(接上页)

```
↪10) #调用write_plc_
↪values方法，将message字典中的数据下发至指定测点；定义该方法的回调函数名称为ack并将userdata传递

def ack(send_result, userdata): #定义回调函数ack
    topic = userdata[0] #定义响应主题：devices/rpc/response
    try:
        resp_data = {"Status":send_result[0]["result"], "Data":userdata[1]}
↪#定义响应数据
    except:
        resp_data = {"Status":"Failed", "Data":userdata[1]} #定义异常时的响应数据
    publish(topic, json.dumps(resp_data), 1) #调用wizard_api模块中的mqtt_
↪publish将响应数据发送给MQTT服务器
```

- 订阅示例 4：立即召回数据

  本示例实现了从 MQTT 服务器下发指定命令时，立即读取所有变量数值并发送至 MQTT 服务器，发布和代码配置示例如下：




```
from quickfaas.remotebus import publish
from quickfaas.measure import recall2
from common.Logger import logger
import json

def main(topic, payload): #定义订阅入口函数
    logger.info("topic: %s, payload: %s" %(topic, payload)) #打印订阅主题和数据
    realtime_data = recall2(callback= recall_data, userdata="response/recall",␣
↪timeout= 10)
↪#调用recall2方法，获取所有测点数据；定义该方法的回调函数名称为recall_
↪data并将userdata传递给回调函数recall_data

def recall_data(realtime_data, userdata): #定义回调函数recall_data
    publish(userdata, json.dumps(realtime_data), 1) #上报所有测点的数据值
```

Device Supervisor 的 api 接口说明

Device Supervisor 提供以下 api 接口：

- 发布消息到云端：将指定数据通过相应的主题和 qos 发送到云端 MQTT 服务器，在 qos 不为 0 时发送失败则缓存数据并重新发送。
  - 调用方法

```
from quickfaas.remotebus import publish
```

  - api 参数
    - 参数 1：发送数据的主题，数据类型为 string
    - 参数 2：需要发送的数据
    - 参数 3：发布数据的 qos 等级（包括 0/1/2 三种等级）
    - 参数 4（可选参数 wizard_data）：发送历史数据时使用的主题、qos 和数据。格式为：

```
{"topic":<topic>, "qos":<qos>, "payload":<payload>}
```

  - 使用示例请参考*发布示例 3*

```
publish(<topic>, <payload>, <qos>, <wizard_data>)
```

- 发布消息到本地：将指定数据通过相应的主题和 qos 发送到本地 MQTT 服务器，在 qos 不为 0 时发送失败则缓存数据（掉电后丢失）并重新发送。
  - 调用方法

```
from quickfaas.messagebus import publish
```

  - api 参数
    - 参数 1：发送数据的主题，数据类型为 string
    - 参数 2：需要发送的数据
    - 参数 3：发布数据的 qos 等级（包括 0/1/2 三种等级）
  - 使用示例

```
publish(<topic>, <payload>, <qos>)
```

- 修改测点值：写入数值至测点对应的地址并返回写入结果
  - 调用方法

```
from quickfaas.measure import write_plc_values
```

- api 参数
  - 参数 1：写入消息，支持两种格式：
    - 格式 1（推荐）：

```
{"<controller_name>": {"<measure_name>": <value>, "<measure_name>":
↪<value>}}
```

    - 格式 2（需要确保测点名称在所有控制器下唯一，否则会出错）：

```
{"<measure_name>": <value>, "<measure_name>": <value>}
```

  - 参数 2（可选参数 callback）：返回修改结果的回调函数名称，回调函数说明见 *write_plc_values 回调函数说明*
  - 参数 3（可选参数 userdata）：已有参数 2 时，可将需要传递给参数 2 的数据赋值给参数 3
  - 参数 4（可选参数 timeout）：写入超时时间，默认为 60 秒
- 使用示例请参考订阅示例 *3*

```
write_plc_values(message, callback= ack, userdata= userdata, timeout = 10)
```

- 立即获取测点值：立即获取相应测点的数据值
  - 调用方法

```
from quickfaas.measure import recall2
```

  - api 参数
    - 参数 1（可选参数 names）：定义如何召回测点数据，默认召回所有控制器下的所有测点数据
      - 召回所有控制器下的所有测点数据

```
names= [] #names为None或者[]均可
```

      - 召回控制器“controller1”下的所有测点数据

```
names= [{"name": "controller1", "measures": []}]
```

      - 召回控制器“controller1”下测点”measure1”和”measure2”的数据

```
[{"name": "controller1", "measures": ["measure1", "measure2"]}]
```

  * 参数 2（可选参数 callback）：立即读取所有测点数值的回调函数的名称，回调函数说明见 *recall2 回调函数*
  * 参数 3（可选参数 userdata）：将需要传递给参数 2 的数据赋值给参数 3
  * 参数 4（可选参数 timeout）：立即读取测点数值的超时时间，默认为 10 秒

- 使用示例请参考*订阅示例 4*

```
recall2(callback= ack, userdata= userdata, timeout= 10)
```

- 获取全局参数：获取“参数设置”中的“自定义参数”信息
  - 调用方法

```
from quickfaas.global_dict import get_global_parameter
```

  - api 参数

    无

  - 使用示例

```
global_parameter = get_global_parameter()
```

  - 响应数据

```
{'SN': 'GF5021111111111', 'MAC': '00:00:00:00:00:00'}
```

- 获取控制器连接状态：获取网关与控制器的连接状态
  - 调用方法

```
from quickfaas.controller import get_controller_status
```

  - api 参数

    可选参数 controller：需要获取连接状态的控制器名称，默认获取所有控制器的连接状态

  - 使用示例

```
controller_status = get_controller_status()
```

  - 响应数据

```
{
        "controller1": { # 控制器名称
```

(续下页)

(接上页)

```
                "health": 1, #␣
↪控制器连接状态，0代表控制器离线；1代表控制器在线
                "timestamp": 1582771955 #状态时间戳
        }
}
```

- 获取云平台连接状态：获取网关与云平台的连接状态
  - 调用方法

```
from quickfaas.clouds import get_status
```

  - api 参数

    无

  - 使用示例

```
cloud_status = get_status()
```

  - 响应数据

```
Ture/False # Ture代表连接成功；False代表连接断开
```

- 获取全局配置：获取 DSA 所有配置信息
  - 调用方法

```
from quickfaas.config import get
```

  - api 参数

    无

  - 使用示例

```
config = get()
```

  - 响应数据

```
{
        'controllers': [{  // 控制器列表
                'protocol': 'Modbus-TCP',  // 控制器协议
                'name': 'S71200',  // 控制器名称
                'args': {  // 协议特有参数
                        'slaveAddr': 1,
                        'int16Ord': 'ab',
```

(续下页)

(接上页)

```
                        'int32Ord': 'abcd',
                        'float32Ord': 'abcd',
                        'continuousAcquisition': 1,
                        'maxContinuousNumber': 64,
                        'communicationInterval': 3
                },
                'samplePeriod': 1,  // 控制器轮询周期
                'expired': 10000,  // 超时时间
                'endpoint': '10.5.23.28:502'  // 控制器通用访问参数
        }],
        'measures': [{  // 测点列表
                'name': 'DATA1',  // 测点名称
                'ctrlName': 'S71200',  // 所属控制器名称
                'group': 'default',  // 所属分组名称
                'uploadType': 'periodic',  // 上报类型
                'dataType': 'WORD',  // 测点数据类型
                'addr': '40001',  // 测点地址
                'readWrite': 'rw',  // 测点读写权限
                'unit': '',  // 测点单位
                'desc': '',  // 测点描述
                'transformType': 0,  // 测点数据运算模式
                'gain': '1.0',  // 倍率,仅 transformType=2␣
↪时有效，支持任意整数或浮点数
                'offset': '0.0'  // 偏移,仅 transformType=2␣
↪时有效，支持任意整数或浮点数
        }, {
                'name': 'DATA2',
                'ctrlName': 'S71200',
                'group': 'default',
                'uploadType': 'periodic',
                'dataType': 'WORD',
                'addr': '40002',
                'readWrite': 'rw',
                'unit': '',
                'desc': '',
                'transformType': 0,
                'gain': '1.0',
                'offset': '0.0'
        }],
        'alarmLables': ['default'],  // 告警标签列表
        'alarms': [],  // 告警规则列表
        'groups': [{  // 测点分组列表
                'name': 'default',  // 分组名称
```

(续下页)

(接上页)

```
            'uploadInterval': 10  // 上报间隔
        }],
        'misc': {  // 其他配置
                'coms': [{ // 串口设置
                        'name': 'rs232',
                        'baud': 9600,
                        'bits': 8,
                        'parityChk': 'n',
                        'stopbits': 1
                }, {
                        'name': 'rs485',
                        'baud': 9600,
                        'bits': 8,
                        'parityChk': 'n',
                        'stopbits': 1
                }],
                'logLvl': 'INFO',  // 日志等级
                'maxAlarmRecordSz': 2000  // 最大存储的告警条数
        },
        'clouds': [{  // 云服务参数
                'cacheSize': 10000,  // 缓存大小设置
                'enable': 1,  // 0: 禁用云服务, 1: 启用云服务
                'name': 'default',  // 预留字段, 填写default即可
                'type': 'Standard MQTT',  // 云服务类型, 仅支持Standard MQTT
                'args': {  // 不同云服务特定的参数集
                        'host': 'broker-cn.emqx.io',
                        'port': 1883,
                        'clientId': 'zntest',
                        'auth': 0,
                        'tls': 0,
                        'cleanSession': 0,
                        'mqttVersion': 'v3.1.1',
                        'keepalive': 60,
                        'key': '',
                        'cert': '',
                        'rootCA': '',
                        'verifyServer': 0,
                        'verifyClient': 0,
                        'username': '',
                        'passwd': ''
                }
        }],
        'quickfaas': { // quickfaas信息
```

(续下页)

(接上页)

```
            'uploadFuncs': [],   // 发布消息快函数列表
            'downloadFuncs': [],  // 订阅消息快函数列表
            'genericFuncs': []   // 自定义快函数列表
        },
        'labels': [{  // 自定义参数列表
            'key': 'SN',
            'value': 'GF5022117001693'
        }, {
            'key': 'MAC',
            'value': '00:18:05:16:fa:4c'
        }],
        'modbusSlave': {  // Modbus协议转换信息
            'enable': 0,
            'protocol': 'Modbus-TCP',
            'port': 502,
            'slaveAddr': 1,
            'int16Ord': 'ab',
            'int32Ord': 'abcd',
            'float32Ord': 'abcd',
            'maxConnection': 5,
            'mapping_table': []
        },
        'iec104Server': {  // IEC 104协议转换信息
            'enable': 0,
            'cotSize': 2,
            'port': 2404,
            'serverList': [{
                    'asduAddr': 1
            }],
            'kValue': 12,
            'wValue': 8,
            't0': 15,
            't1': 15,
            't2': 10,
            't3': 20,
            'maximumLink': 5,
            'timeSet': 1,
            'byteOrder': 'abcd',
            'mapping_table': []
        },
        'opcuaServer': {  // OPC UA协议转换信息
            'enable': 0,
            'port': 2404,
```

(续下页)

(接上页)

```
                'maximumLink': 5,
                'securityMode': 0,
                'identifierType': 'String',
                'mapping_table': []
        },
        'version': '2.2.1'  // 版本信息，请勿修改
}
```

- 设置全局配置：设置 DSA 配置信息
  - 调用方法

```
from quickfaas.config import set
```

  - api 参数
    - 参数 1: 新的 DSA 配置信息
  - 使用示例

```
from quickfaas.config import get, set

def mian():
    global_config = get()
    set(global_config)
```

  - 响应数据

    无

- 数据库插入数据：在网关本地的数据库中插入历史数据。
  - 调用方法

```
from quickfaas.LWTSDB import insert_request
```

  - api 参数
    - 参数 1: 数据表名称。
    - 参数 2: 需要插入到数据库的历史数据。数据格式：{”“：{”controller1”：{”measure1”：[<measure1_health>, <measure1_value>], “measure2” : [<measure2_health>, <measure2_value>]}}}
    - 参数 3(可选参数 noack): 定义是否需要响应，0：需要响应，1：不需要响应，默认为 0。
    - 参数 4(可选参数 callback): 定义响应数据的回调函数，noack 为 0 时该参数才有意义。
    - 参数 5(可选参数 userdata): 定义回调函数的传参，noack 为 0 时该参数才有意义。
    - 参数 6(可选参数 timeout): 定义请求超时时间，默认 30 秒。

- 使用示例

```
import time
from common.Logger import logger
from quickfaas.LWTSDB import insert_request

def main():
    insert_data = [
        {str(int(time.time())): {"controller1": {"measure1": [1, 100],
↪"measure2": [1, "test"]}}}
    ]

    insert_request('default', insert_data, 0, callback=insert_callback,␣
↪userdata="insert")

def insert_callback(message, userdata):
    logger.info("%s response message:%s" % (userdata, message))
```

- 响应数据

  回调函数中的响应示例如下：

```
{
  "return_code": 0,
  "return_msg":"okay"
}
```

- 删除历史数据：在网关本地的数据库中插入历史数据。
  - 调用方法

```
from quickfaas.LWTSDB import remove_request
```

  - api 参数
    - 参数 1: 数据表名称。
    - 参数 2(可选参数 start_time): 需要删除的历史数据的起始时间，为 None 即从时间戳最早数据的开始删除，时间格式为%Y-%m-%d %H:%M:%S。
    - 参数 3(可选参数 end_time): 需要删除的历史数据的截止时间，为 None 即删除至时间戳最晚的数据，时间格式为%Y-%m-%d %H:%M:%S。
    - 参数 4(可选参数 noack): 定义是否需要响应，0：需要响应，1：不需要响应，默认为 0。
    - 参数 5(可选参数 callback): 定义响应数据的回调函数，noack 为 0 时该参数才有意义。
    - 参数 6(可选参数 userdata): 定义回调函数的传参，noack 为 0 时该参数才有意义。
    - 参数 7(可选参数 timeout): 定义请求超时时间，默认 30 秒。

- 使用示例

```
from common.Logger import logger
from quickfaas.LWTSDB import remove_request

def main():
    remove_request('default', '2023-01-09 12:00:00', '2022-12-09 16:00:00', 0,
↪ callback=remove_callback, userdata="remove") #删除时序数据库数据(从'2023-
↪01-09 12:00:00'到'2023-01-09 16:00:00'),并且需要响应。

def remove_callback(message, userdata):
    logger.info("%s response message:%s" % (userdata, message))
```

- 响应数据

  回调函数中的响应示例如下：

```
{
  "return_code": 0,
  "return_msg":"okay"
}
```

- 查找历史数据：在网关本地的数据库中查找历史数据。
  - 调用方法

```
from quickfaas.LWTSDB import query_request
```

  - api 参数
    - 参数 1: 数据表名称。
    - 参数 2(可选参数 start_time): 需要查找的历史数据的起始时间，为 None 即从时间戳最早数据的开始查找，时间格式为%Y-%m-%d %H:%M:%S。
    - 参数 3(可选参数 end_time): 需要查找的历史数据的截止时间，为 None 即查找至时间戳最晚的数据，时间格式为%Y-%m-%d %H:%M:%S。
    - 参数 4(可选参数 filter): 查找数据的过滤规则。数据格式：{" default" : "accept_all" , "black_list" : {" controller1" : [" measure1" , "measure2"]}, "white_list" : []}。
      - default
      - accept_all：默认返回所有测点，黑名单列表中的测点除外。
      - deny_all：默认所有测点都不返回，白名单列表中的测点除外。
      - black_list：黑名单测点名称列表。
      - white_list：白名单测点名称列表。

* 参数 5(可选参数 limit): 单次需要获取的历史数据条数，默认 1000。
* 参数 6(可选参数 offest): 单次获取历史数据的起始位置，为 0 则表示从所有历史数据中的第一条开始获取。通常和 limit 组合使用，默认为 0。
* 参数 7(可选参数 callback): 定义响应数据的回调函数，noack 为 0 时该参数才有意义。
* 参数 8(可选参数 userdata): 定义回调函数的传参，noack 为 0 时该参数才有意义。
* 参数 9(可选参数 timeout): 定义请求超时时间，默认 30 秒。

– 使用示例

```
def main():
    filter = {"default": "deny_all", "white_list": {"controller1": ["measure1
↪"]}}
    query_request('default', '2023-01-09 12:00:00', '2023-01-09 16:00:00',␣
↪filter, callback=query_callback, userdata="query") #查询default数据表数据(从
↪'2023-01-09 12:00:00'到'2023-01-09 16:00:00', 并只要测点名称为"measure1
↪"的测点)

def query_callback(message, userdata):
    logger.info("%s response message:%s" % (userdata, message))
```

– 响应数据

```
{
  "total": 1000,
    "offset": 0,
    "size": 100,
  "data":[
    {"1669630340": {"controller1": {"measure1": [1, 100], "measure2": [1,
↪"test1"]}}},
    {"1669630350": {"controller1": {"measure1": [1, 101], "measure2": [1,
↪"test2"]}}}
  ]
}
```

Device Supervisor 的回调函数说明

- write_plc_values 回调函数说明 write_plc_values 回调函数包含以下参数，使用示例请参考 *订阅示例 3*：
  - 参数 1: write_plc_values 方法的写入结果

```
[
    {
        "device": "controller1",  //控制器名称
        "var_name": "measures1",  //测点名称
        "result": "OK",          // "OK"代表成功，"Failed"代表失败
        "error": "Success", // 错误原因描述
        "value": None          // 兼容字段
    }
]
```

  - 参数 2: write_plc_values 方法中配置的参数 3，如果未在 write_plc_values 中配置参数 3，则该参数的值为 None

- recall2 回调函数说明 recall2 回调函数包含以下参数*订阅示例 4*：
  - 参数 1: recall2 方法返回的变量数据。获取变量数据超时时返回值为 ("error", -110, "timeout")，正常返回变量数据时数据格式如下：

```
{
    "timestamp": 1589507333.2521989,  //采集时间戳
    "values": {
        "controller1": {  //控制器名称
            "measure1": {  //测点名称
                "raw_data": 12,  //测点值
                      "timestamp": 1582771955,  //采集时间戳
                "status": 1  //测点状态
            },
            "measure2": {
                "raw_data": 1.23,
                      "timestamp": 1582771955,
                "status": 1
            }
        }
    }
}
```

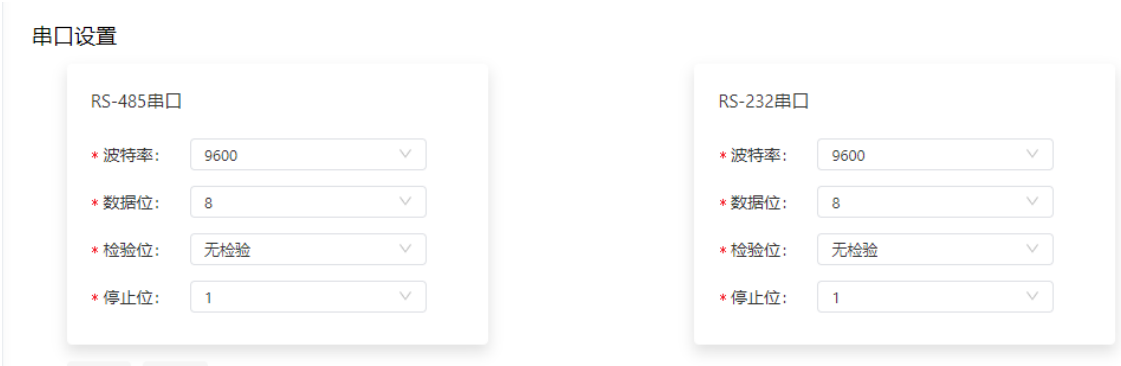
  - 参数 2：recall2 方法中配置的参数 3，如果未在 recall2 中配置参数 3，则该参数的值为 None

参数设置

你可以访问“边缘计算 > 设备监控 > 参数设置”页面配置 DSA 的通用设置。

- 串口设置

  你可以在串口设置中配置 RS485 和 RS232 串口的通讯参数，如下图所示：

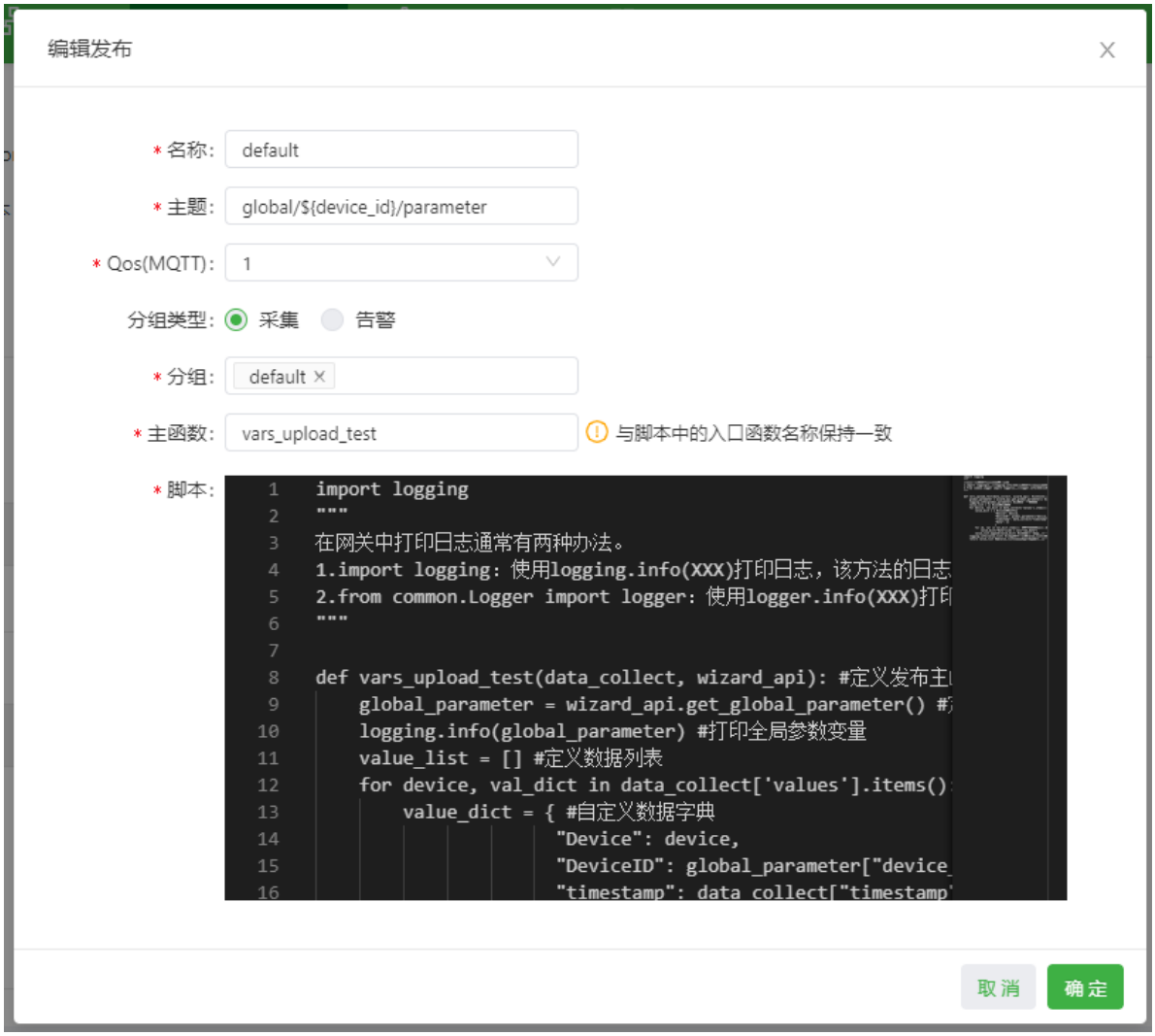


- 默认参数

  你可以在默认参数中设置日志等级、历史告警条数等设置信息。

- 自定义参数

  你可以在自定义参数中自行添加常用参数作为云服务中的通配符使用，默认包含 SN 和 MAC 参数。使用方法为 ${参数名称}，如下图所示：


编辑发布
* 名称: default
* 主题: global/${device_id}/parameter
* Qos(MQTT): 1
分组类型: 采集 告警
* 分组: default
* 主函数: vars_upload_test
与脚本中的入口函数名称保持一致
* 脚本:
import logging
"""
在网关中打印日志通常有两种办法。
1.import logging: 使用logging.info(XXX)打印日志，该方法的日志
2.from common.Logger import logger: 使用logger.info(XXX)打印
"""
def vars_upload_test(data_collect, wizard_api): #定义发布主
global_parameter = wizard_api.get_global_parameter() #
logging.info(global_parameter) #打印全局参数变量
value_list = [] #定义数据列表
for device, val_dict in data_collect['values'].items()
value_dict = { #自定义数据字典
"Device": device,
"DeviceID": global_parameter["device_
"timestamp": data_collect["timestamp'
取消
确定


启用云服务:
云平台类型:
标准 MQTT
基础设置
* 服务器地址:
broker-cn.emqx.io
* MQTT客户端ID:
${device_id}
启用用户验证:
高级设置 >

网关的其他配置

关于网关的其他常用操作请查看IG502 快速使用手册、IG902 快速使用手册或IG974 快速使用手册。

1.1.6 FAQ

查看云服务脚本是否正确

打开 DSA App 日志。脚本编写完成并点击“确定”后，通过日志中的 `Build module: < 入口函数名称>` 信息查看脚本是否构建成功。

脚本构建成功如下图所示：

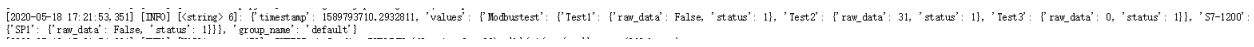
```
2021-11-08 14:11:56,896 <INFO> [faas_handler.py:111]: Build OK.
2021-11-08 14:11:56,904 <INFO> [quick_function.py:99]: uploadFuncs
2021-11-08 14:11:56,908 <INFO> [faas_handler.py:106]: Build module: test, script: import json
from common.Logger import logger
from quickfaas.remotebus import publish
from datetime import datetime
```

脚本构建失败如下图所示：

```
2021-11-08 14:32:50,734 <WARNING> [faas_handler.py:315]: The script may run incorrectly(name 'value_lit' is not defined)
```

查看 App 的云服务输出是否正确

您可以使用使用 `logger` 输出重要日志。下图是在运行脚本中的第 6 行使用了 `logger.info` 方法，在日志中可以通过搜索`<string> 6` 查看输出结果是否符合预期。



1.2 Device Supervisor App 用户手册

Device Supervisor App（以下简称 Device Supervisor）为用户提供了便捷的数据采集、数据处理和数据上云功能，支持 ISO on TCP、ModbusRTU 等多种工业协议解析。本手册以采集 PLC 的数据并上传至 Thingsboard 云平台为例说明如何通过 Device Supervisor App 实现 PLC 数据采集和数据上云。以下将 InGateway501 简称为“IG501”；InGateway902 简称为“IG902”；InGateway502 简称为“IG502”。

- 概览
- *1.* 准备硬件设备及其数据采集环境

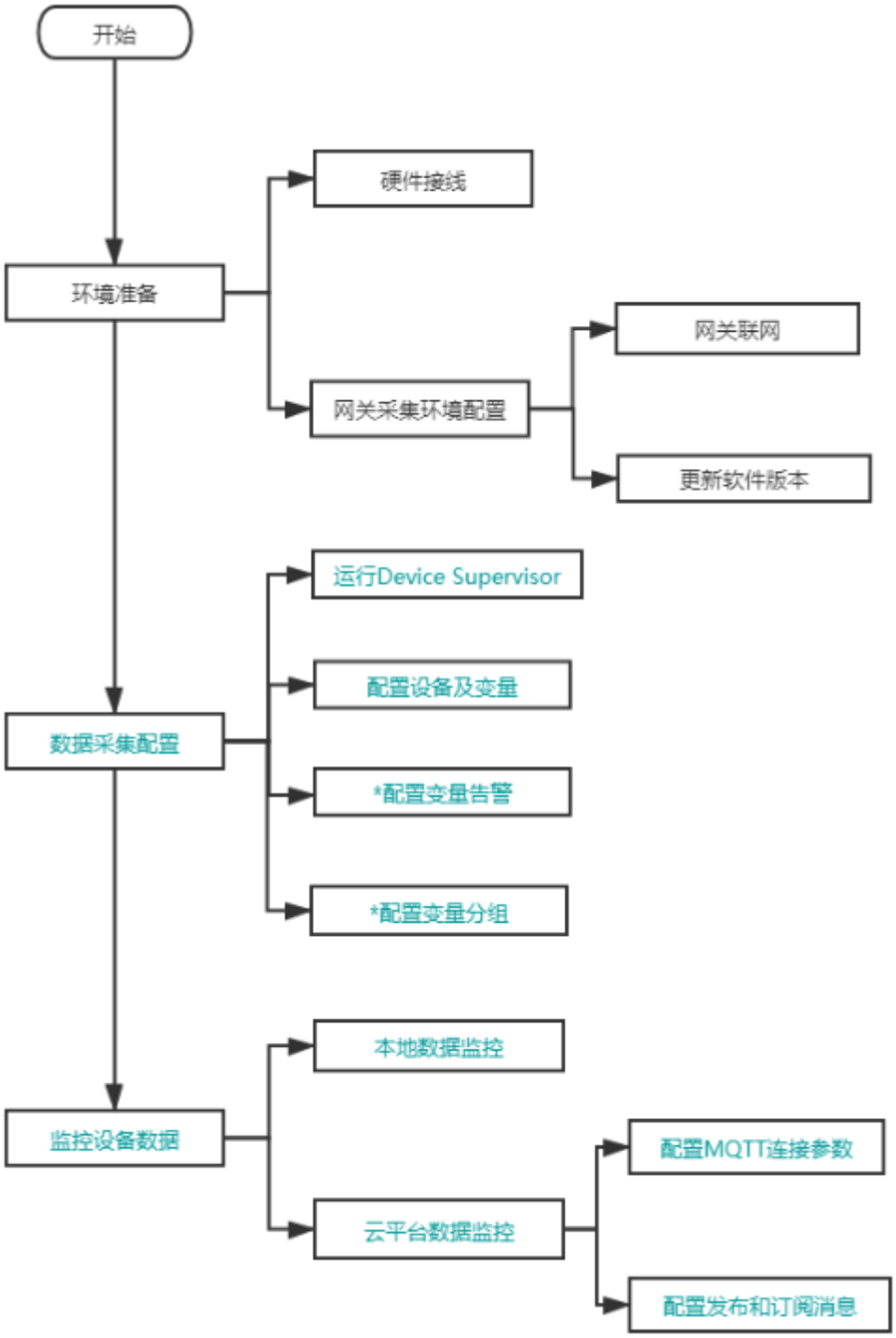
        * 添加设备和资产
        * 传输 *PLC* 数据到 *Thingsboard* 设备
        * 配置可视化仪表板
- *FAQ*
    - 查看云服务脚本是否正确
    - 查看 *App* 的云服务输出是否正确

1.2.1 概览

使用过程中，您需要准备以下项：

- 边缘计算网关 IG501/IG502/IG902
- PLC 设备
- 网线/串口线
- * 更新软件版本所需的固件、SDK 和 App
    - 固件版本：V2.0.0.r12622 及以上
    - SDK 版本：py3sdk-V1.3.7 及以上
    - App 版本：1.1.2 及以上
- *Thingsboad 演示账号

整体流程如下图所示：


开始
环境准备
硬件接线
网关采集环境配置
网关联网
更新软件版本
数据采集配置
运行Device Supervisor
配置设备及变量
*配置变量告警
*配置变量分组
监控设备数据
本地数据监控
云平台数据监控
配置MQTT连接参数
配置发布和订阅消息

1.2.2 1. 准备硬件设备及其数据采集环境

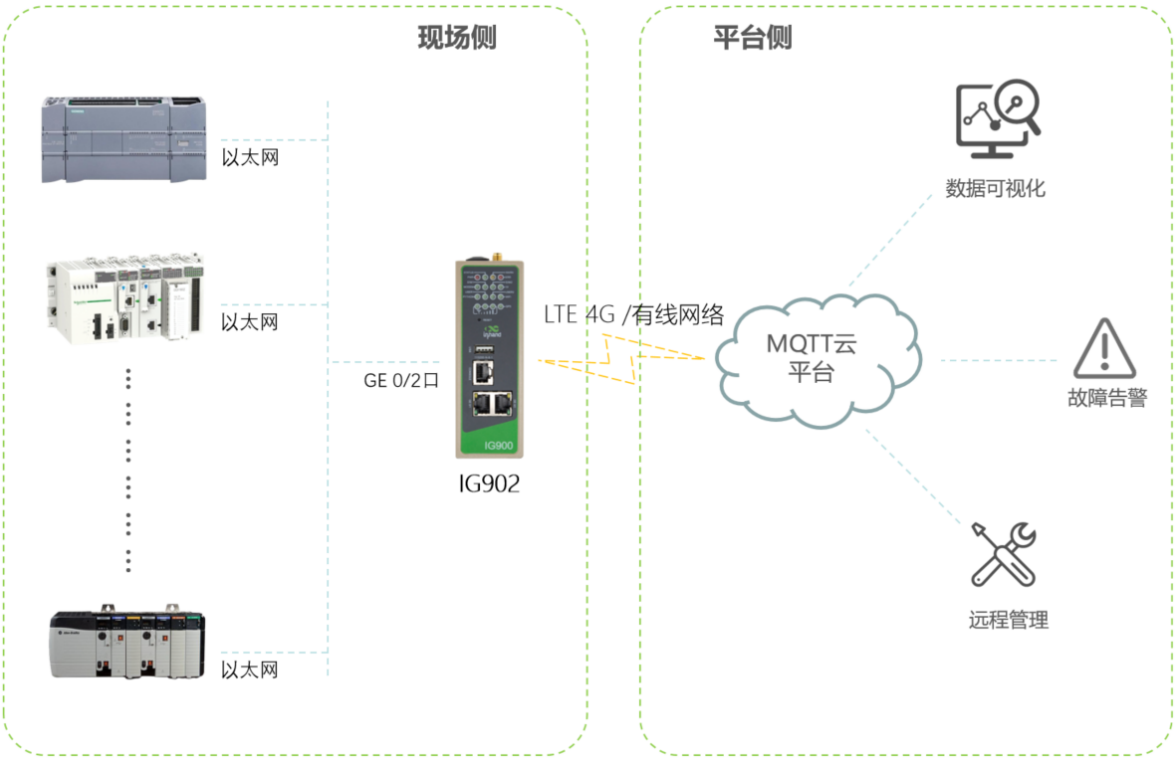
- *1.1* 硬件接线
- *1.2* 设置 *InGateway* 访问 *PLC*
- *1.3* 设置 *InGateway* 联网
- *1.4* 更新 *InGateway* 设备软件版本

1.1 硬件接线

1.1.1 以太网接线

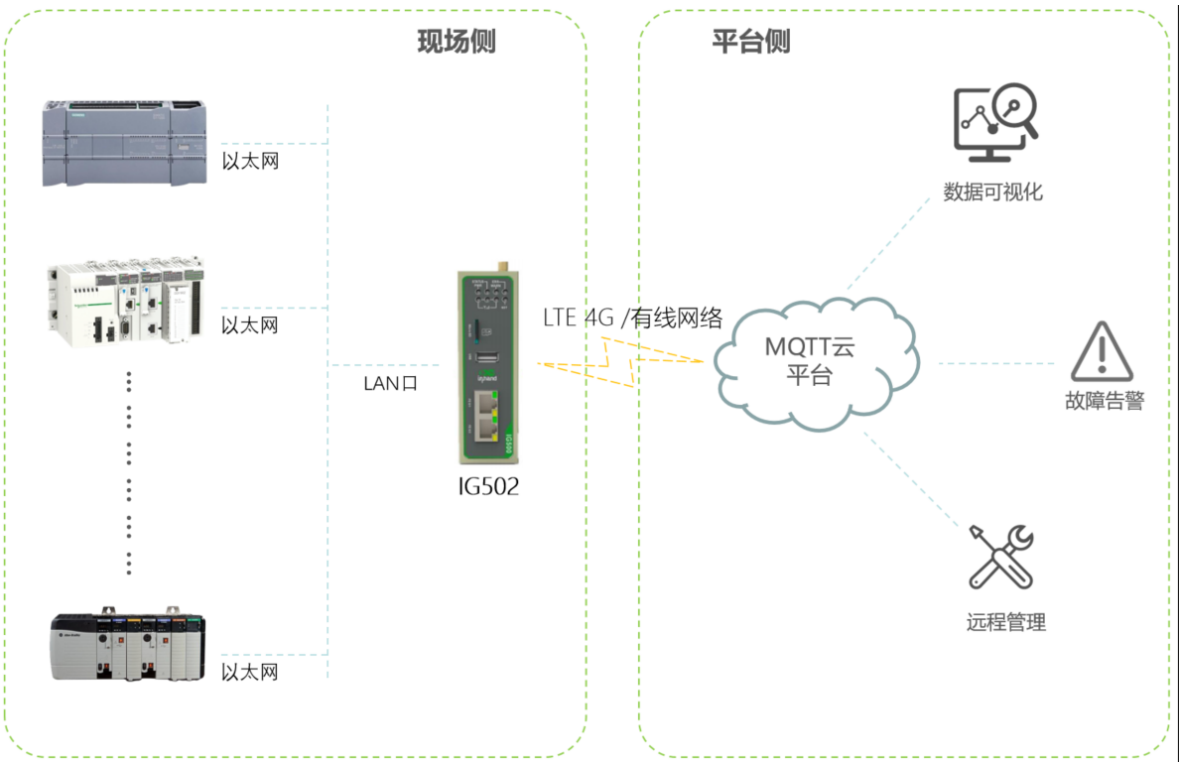
- IG902 以太网接线

  接通 IG902 的电源并按照拓扑使用以太网线连接 IG902 和 PLC。



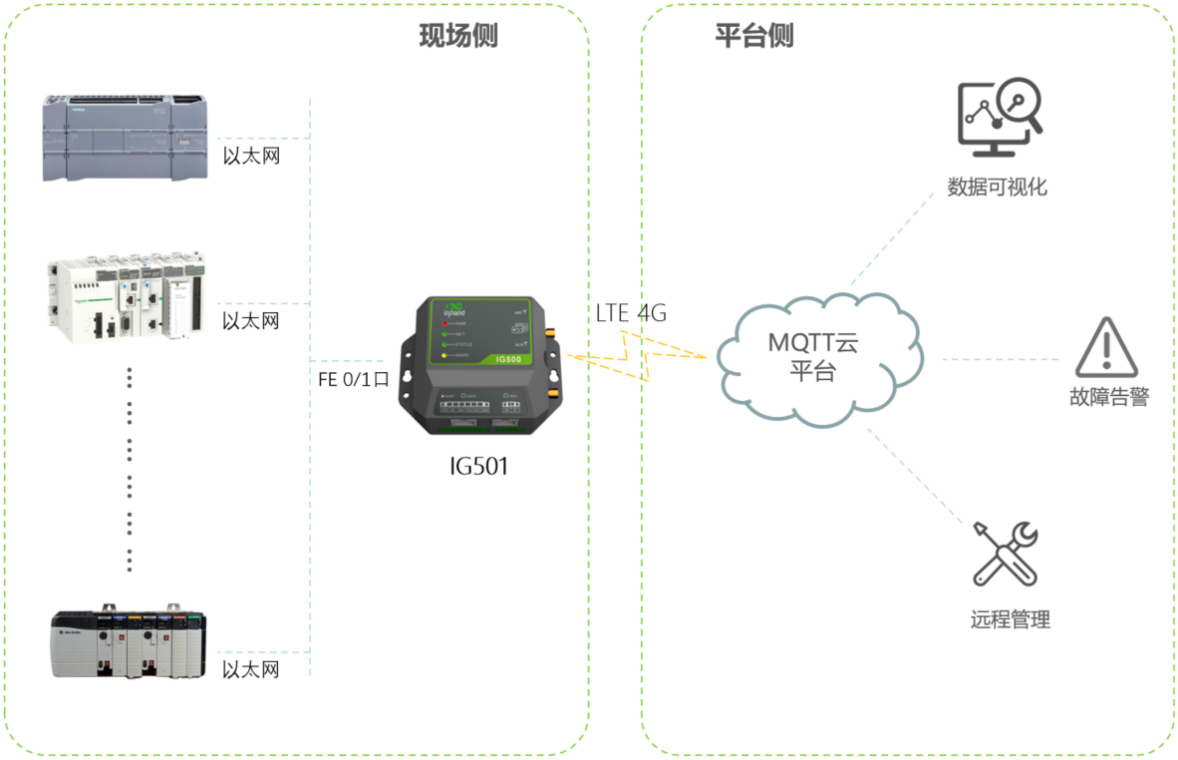
- IG502 以太网接线

  接通 IG502 的电源并按照拓扑使用以太网线连接 IG502 和 PLC。



- IG501 以太网接线

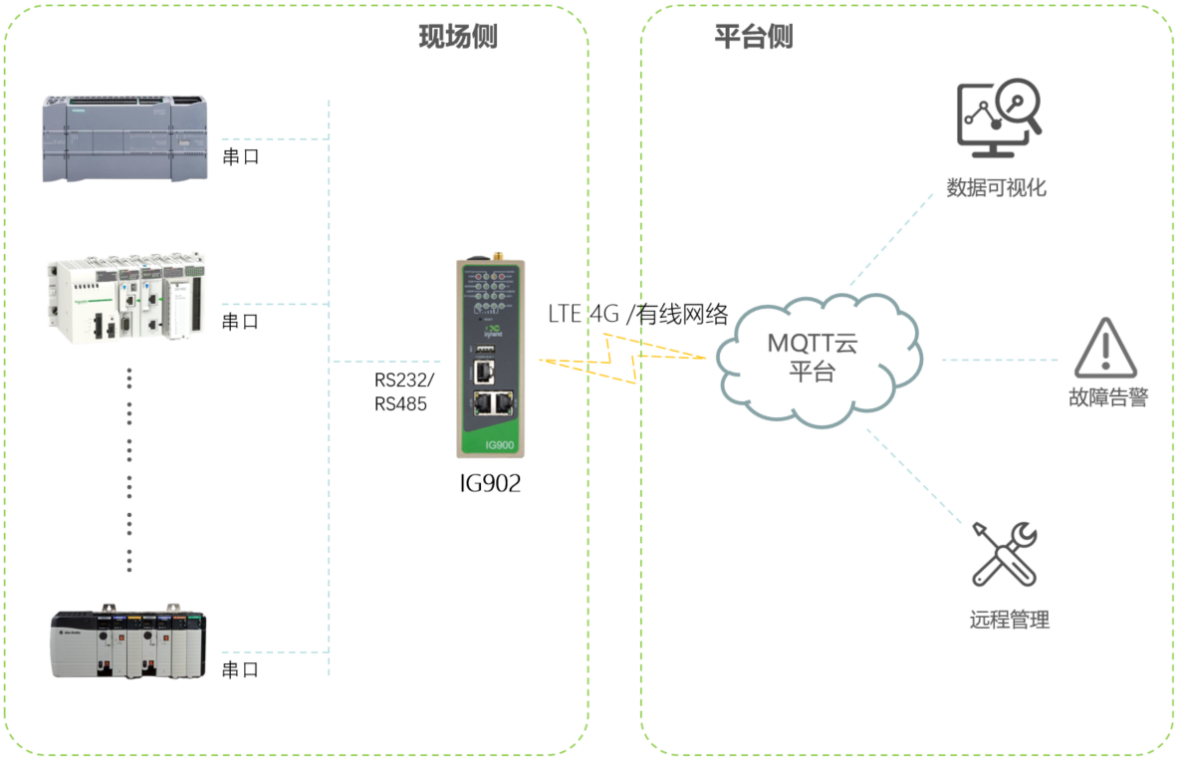
  接通 IG501 的电源并按照拓扑使用以太网线连接 IG501 和 PLC。



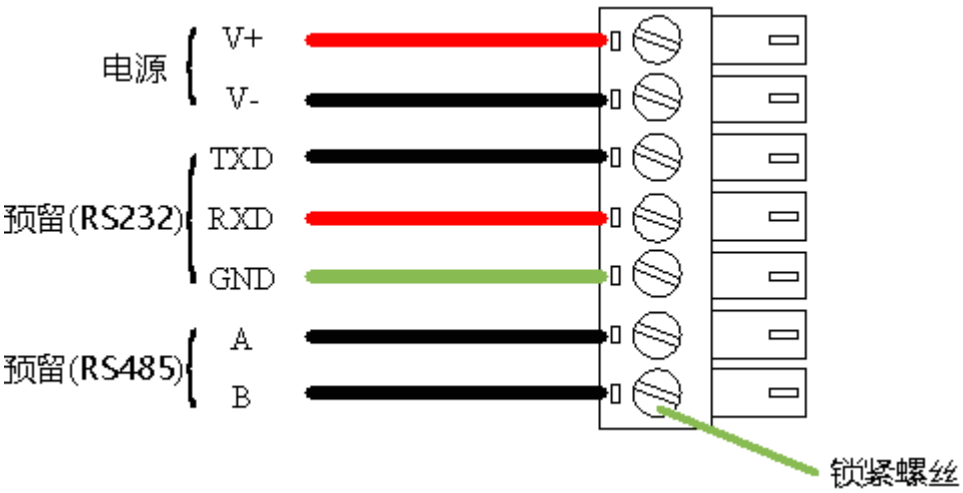
1.1.2 串口接线

- IG902 串口接线

  接通 IG902 的电源并按照拓扑连接 IG902 和 PLC。

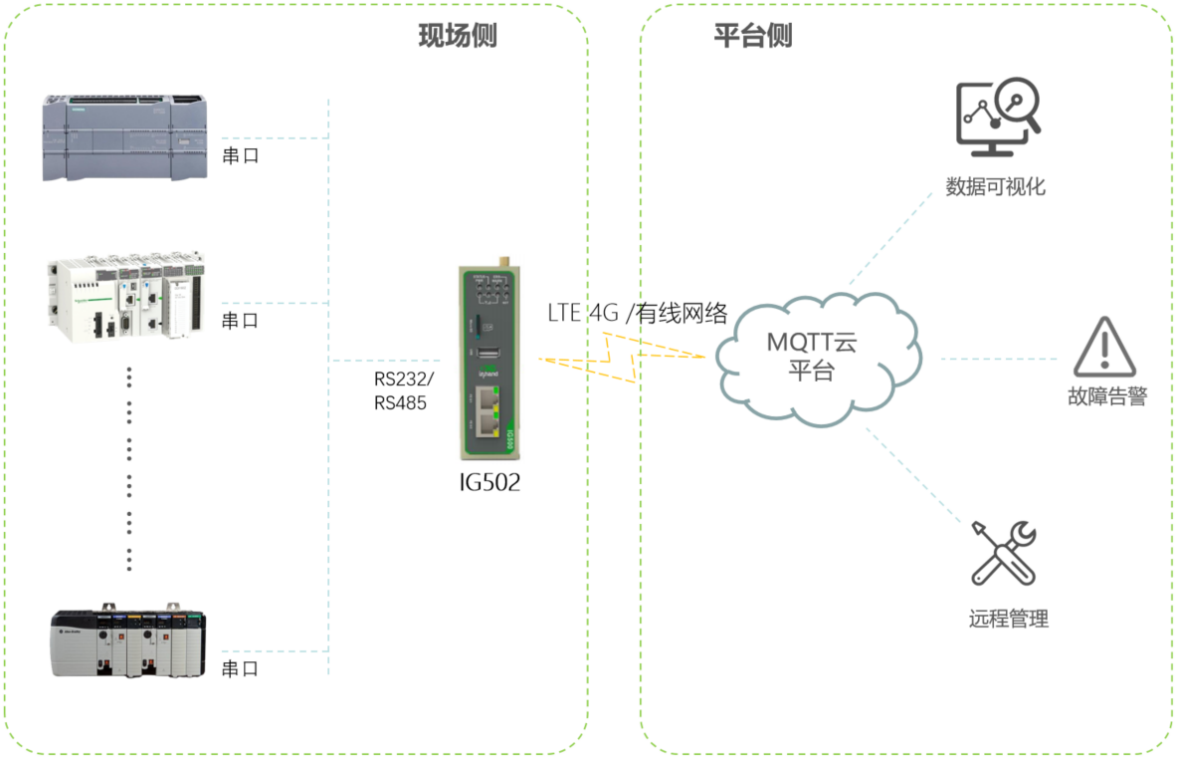


  IG902 串口端子接线说明如下图：

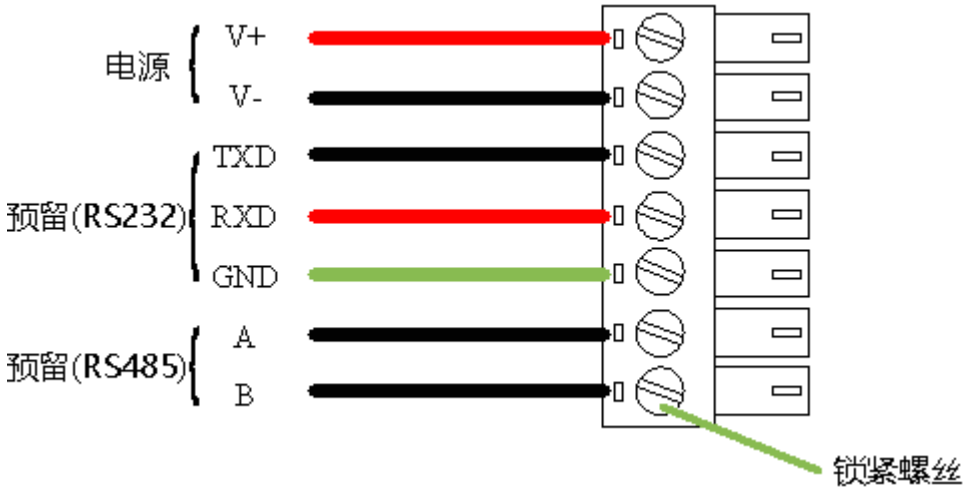


- IG502 串口接线

  接通 IG502 的电源并按照拓扑连接 IG502 和 PLC。

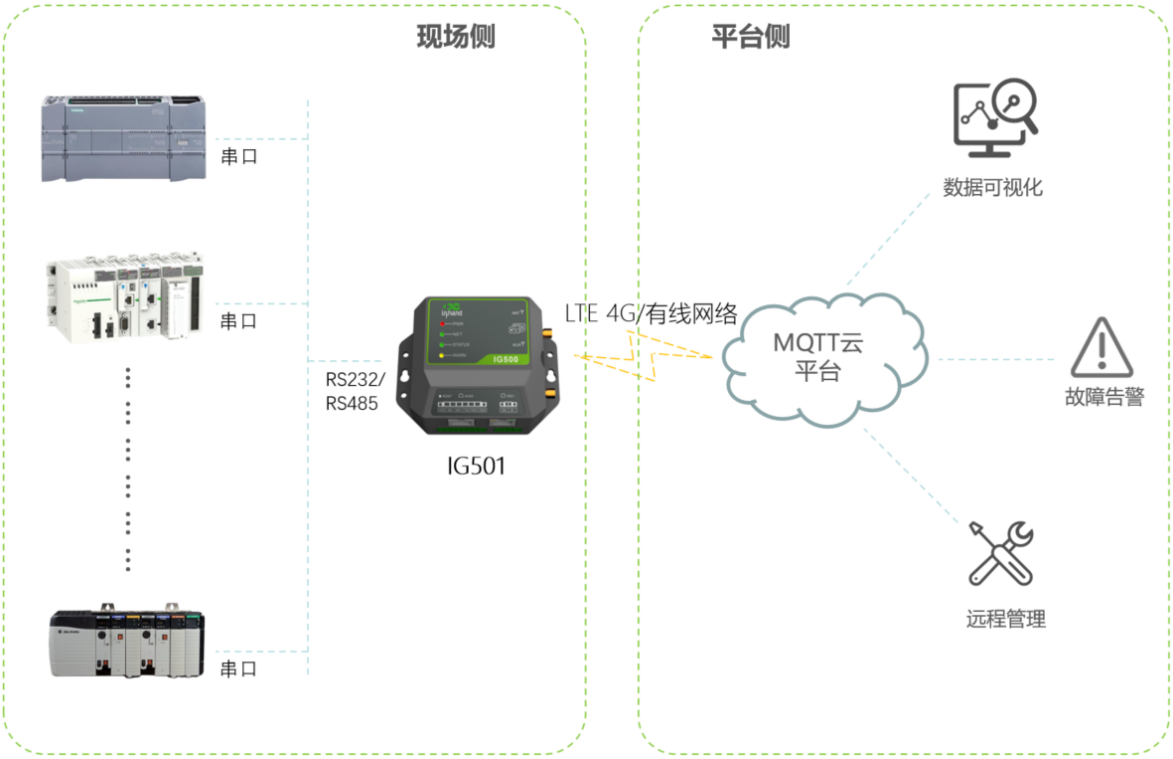


IG502 串口端子接线说明如下图：

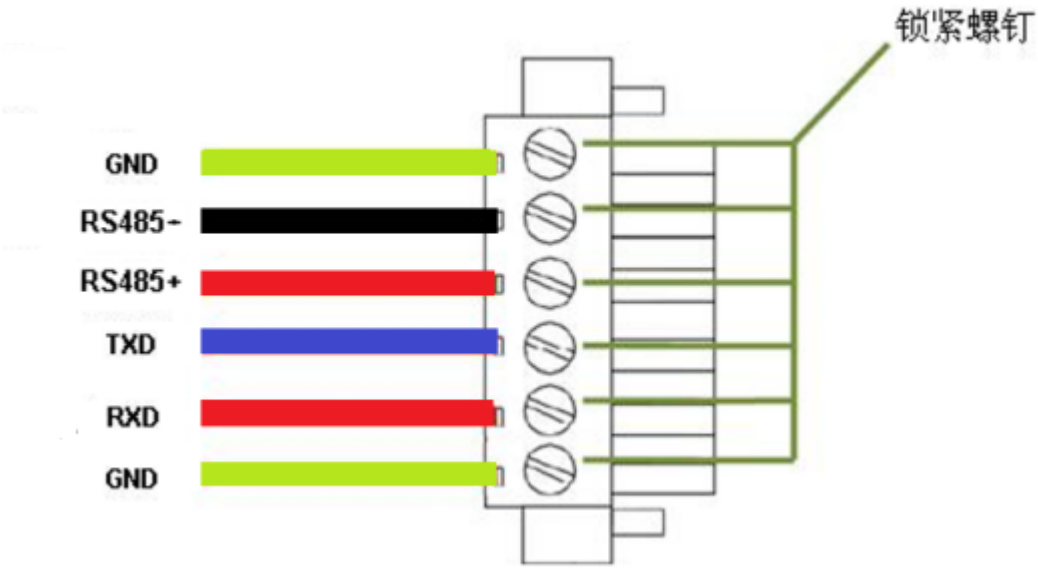


- IG501 串口接线

  接通 IG501 的电源并按照拓扑连接 IG501 和 PLC。



IG501 串口端子接线说明如下图：



1.2 设置 InGateway 访问 PLC

- IG902 的 GE 0/2 口的默认 IP 地址为 `192.168.2.1`。为了使 IG902 能够通过 GE 0/2 口访问以太网 PLC，需要设置 GE 0/2 口与 PLC 处于同一网段，设置方法请参考访问 IG902。
- IG502 的 LAN 口的默认 IP 地址为 `192.168.2.1`。为了使 IG502 能够通过 LAN 口访问以太网 PLC，需要设置 LAN 口与 PLC 处于同一网段，设置方法请参考访问 IG502。
- IG501 的 FE 0/1 口的默认 IP 地址为 `192.168.1.1`。为了使 IG501 能够通过 FE 0/1 口访问以太网 PLC，需要设置 FE 0/1 口与 PLC 处于同一网段，设置方法请参考访问 IG501。

1.3 设置 InGateway 联网

- 设置 IG902 联网请参考IG902 连接 Internet。
- 设置 IG502 联网请参考IG502 连接 Internet。
- 设置 IG501 联网请参考IG501 连接 Internet。

1.4 更新 InGateway 设备软件版本

如需获取 InGateway 产品最新软件版本及其功能特性信息，请访问资源中心。如需更新软件版本，请参考如下链接：

- 更新 IG902 软件版本

  使用 Device Supervisor 时，IG902 的固件版本应为 `V2.0.0.r12537` 及以上；SDK 版本应为 `py3sdk-V1.4.2` 及以上。

- 更新 IG502 软件版本

  使用 Device Supervisor 时，IG502 的固件版本应为 `V2.0.0.r13595` 及以上；SDK 版本应为 `py3sdk-V1.4.2` 及以上。

- 更新 IG501 软件版本

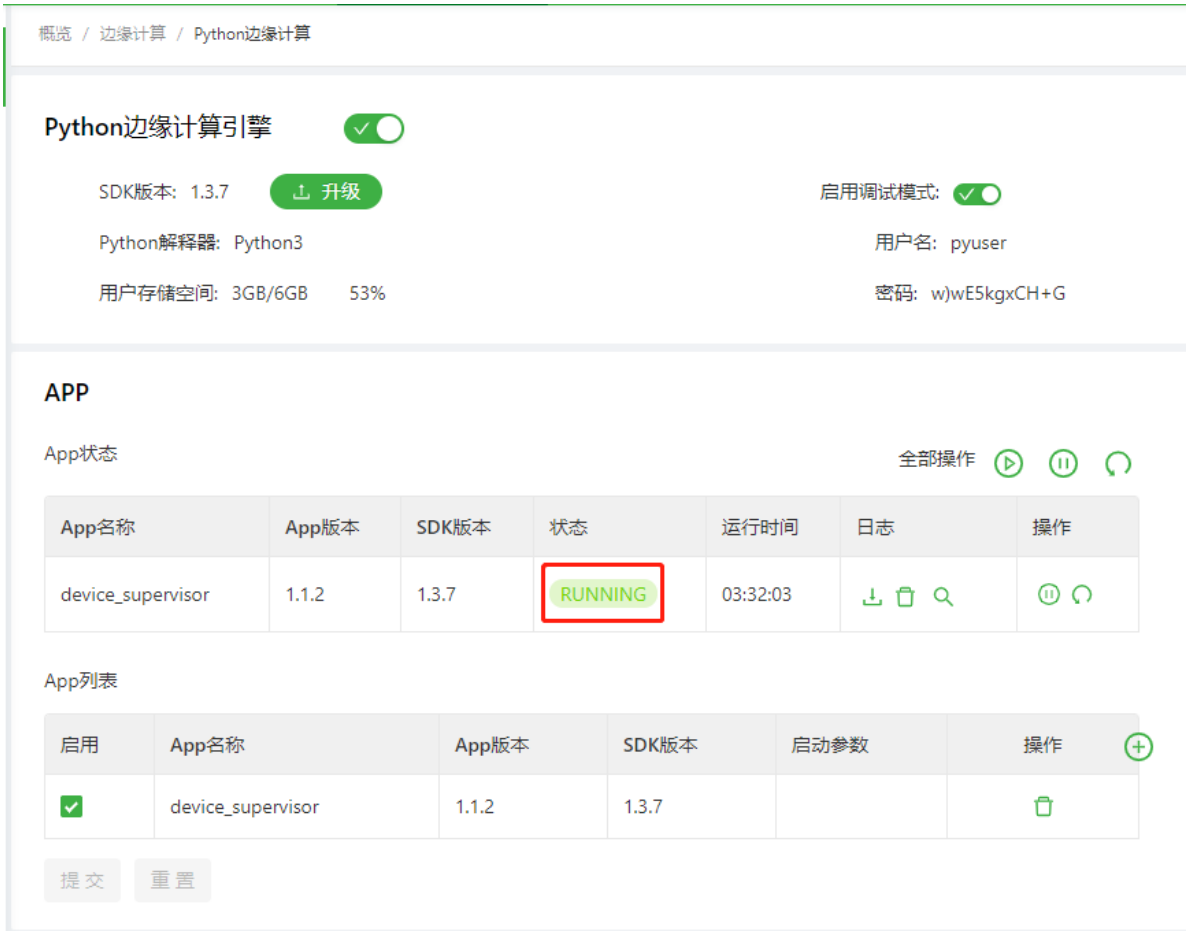
  使用 Device Supervisor 时，IG501 的固件版本应为 `V2.0.0.r12884` 及以上；SDK 版本应为 `py3sdk-V1.4.0` 及以上。

1.2.3 2.Device Supervisor 数据采集配置

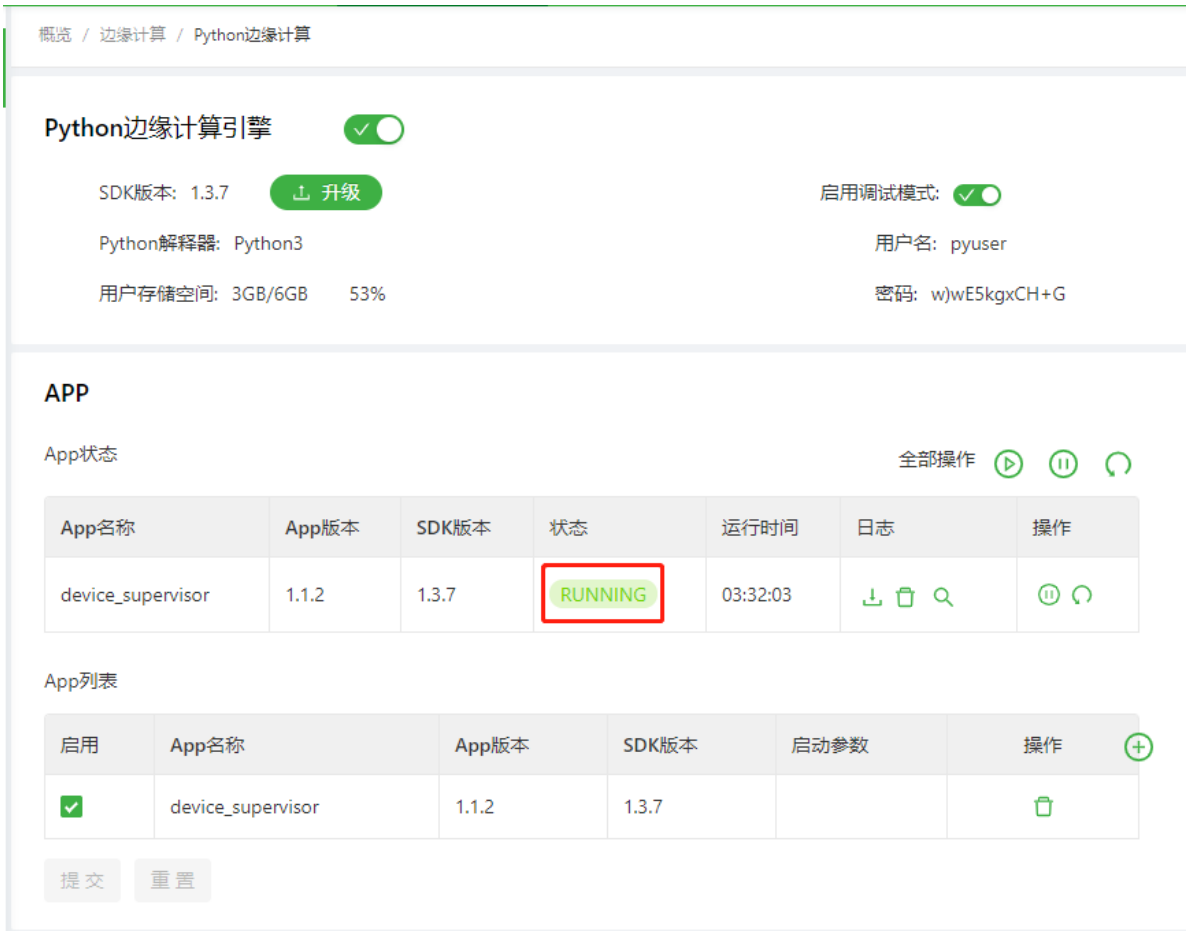
- *2.1 安装并运行 Device Supervisor*
- *2.2 数据采集配置*

2.1 安装并运行 Device Supervisor

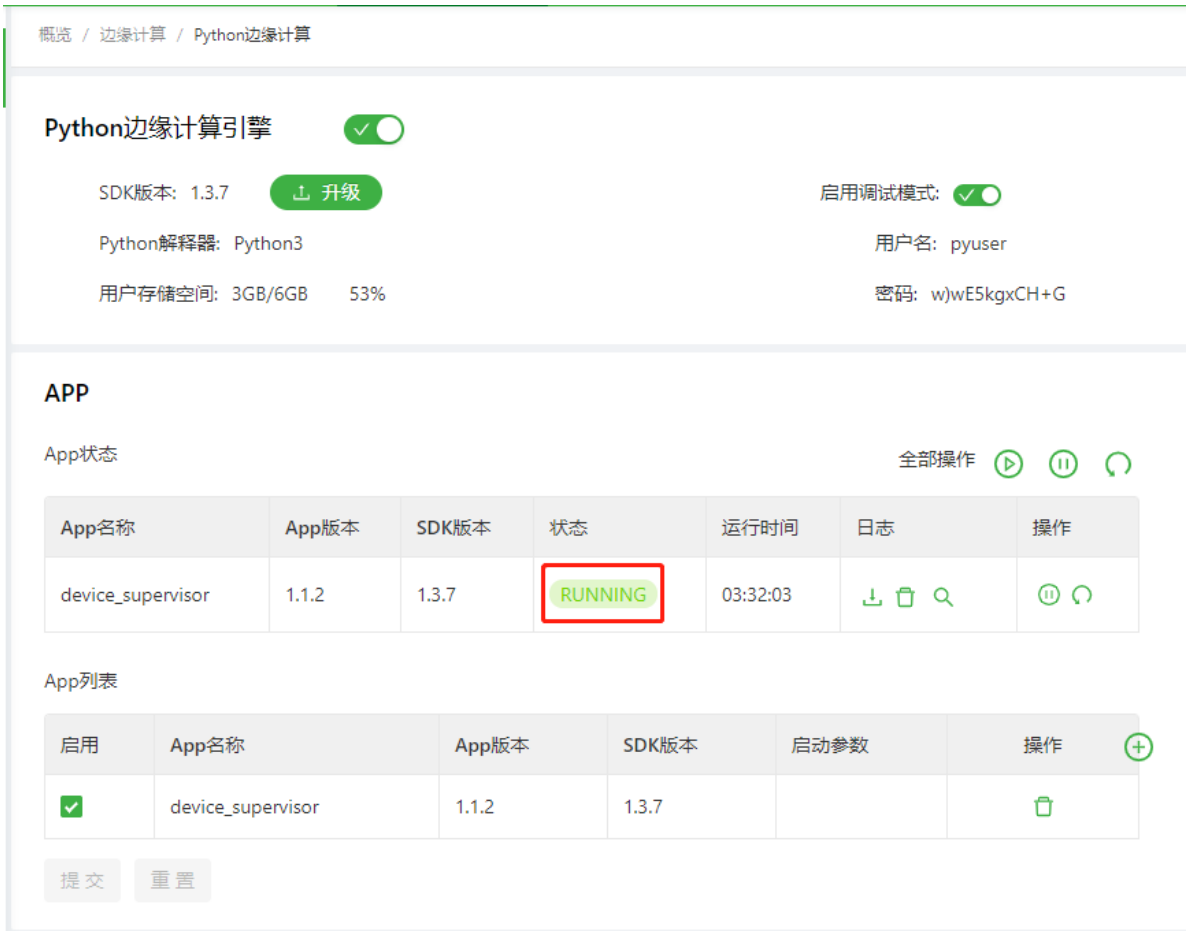
- IG902 如何安装并运行 Python App 请参考IG902 安装和运行 Python App，下载 Device Supervisor 请访问资源中心。Device Supervisor 正常运行后如下图所示：



- IG502 如何安装并运行 Python App 请参考IG502 安装和运行 Python App，下载 Device Supervisor 请访问资源中心。Device Supervisor 正常运行后如下图所示：



- IG501 如何安装并运行 Python App 请参考IG501 安装和运行 Python App，下载 Device Supervisor 请访问资源中心。Device Supervisor 正常运行后如下图所示：



2.2 数据采集配置

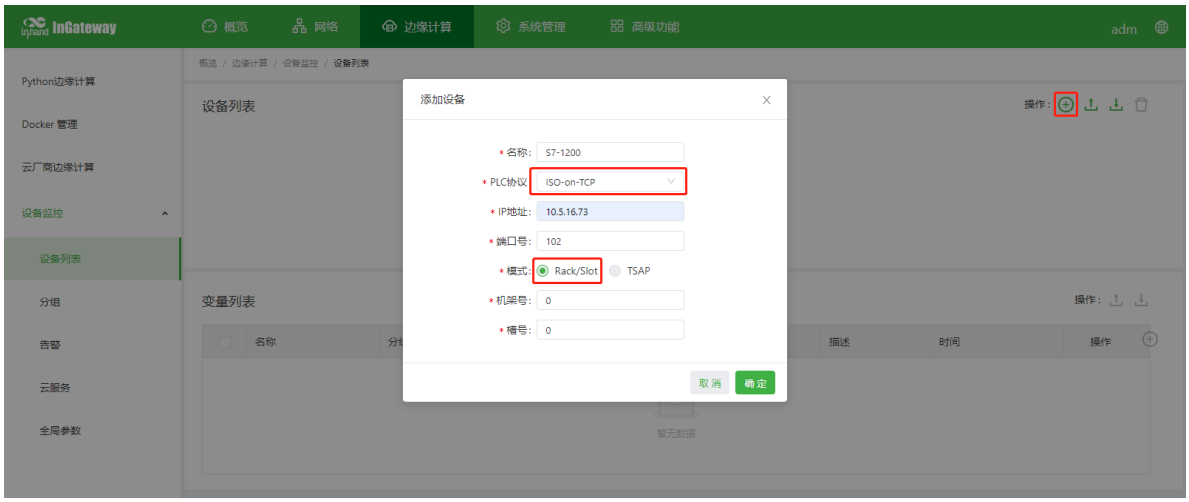
- *2.2.1* 添加 *PLC* 设备
- *2.2.2* 添加变量
- *2.2.3* 配置告警策略
- *2.2.4* 配置分组

2.2.1 添加 PLC 设备

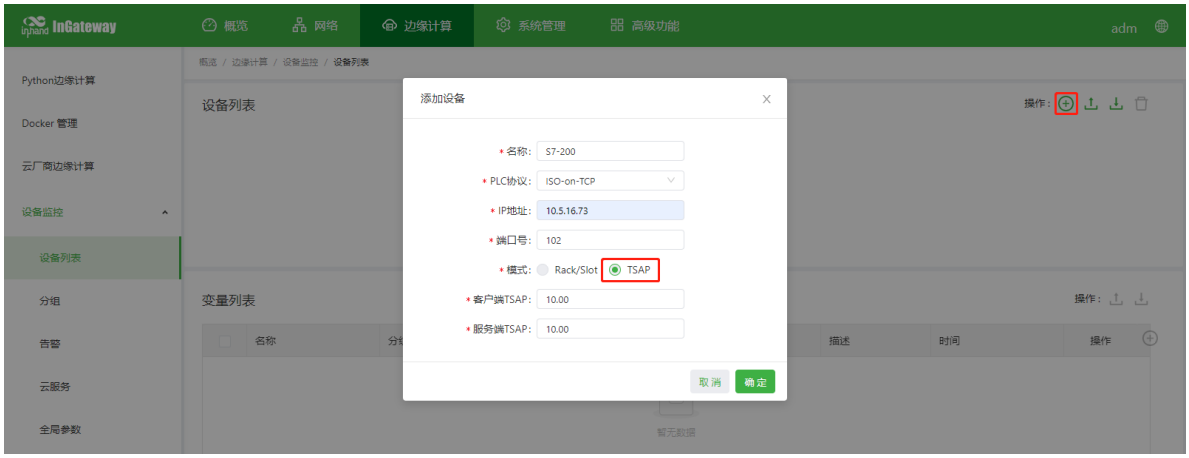
- 添加 ISO on TCP 通讯的 PLC 设备

  进入“边缘计算 > 设备监控 > 设备列表”页面，点击“添加 PLC”按钮，在添加设备页面选择 PLC 协议为“ISO on TCP”并配置 PLC 的通讯参数。注意：设备名称不能重复。

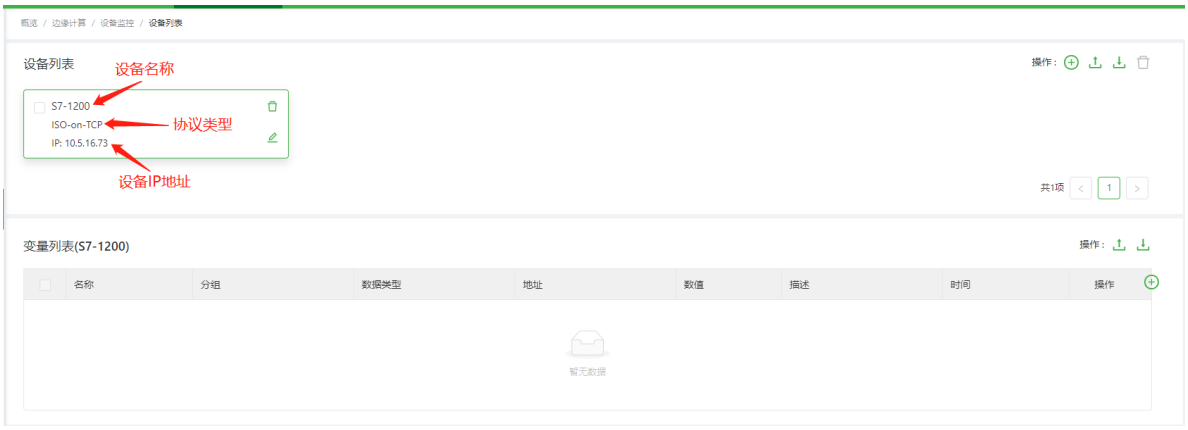
  下图是添加 S7-1500、S7-1200、S7-400 和 S7-300 系列 PLC 的示例（模式选择 `Rack/Slot`）。机架号和槽号默认使用 0，0 即可：



  下图是添加 S7-200、S7-200 Smart 和西门子 LOGO 系列 PLC 的示例（模式选择 `TSAP`）。注意：添加 S7-200 Smart 时，客户端 TSAP 配置为 02.00，服务端 TSAP 配置为 02.01；其余系列根据实际情况配置。

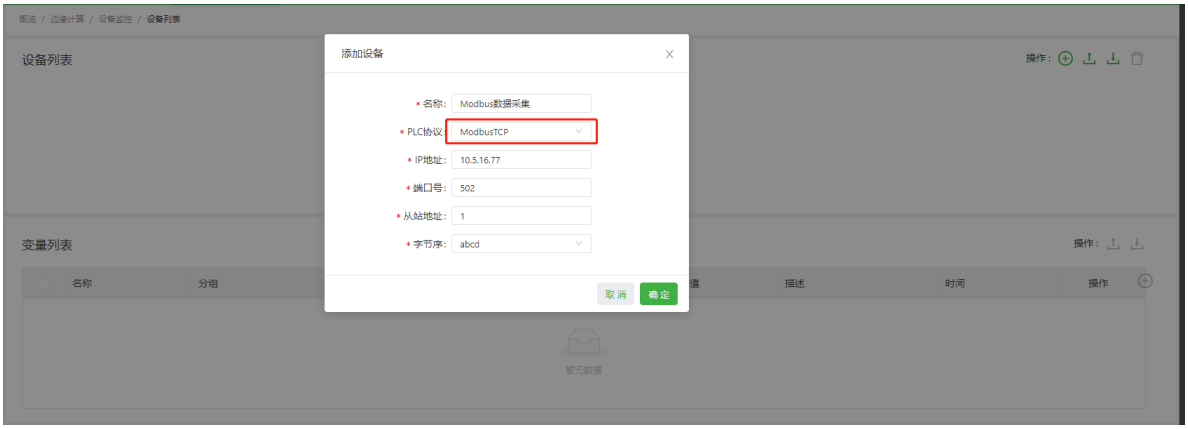


  添加成功后如下图所示：



- 添加 ModbusTCP 通讯的 PLC 设备

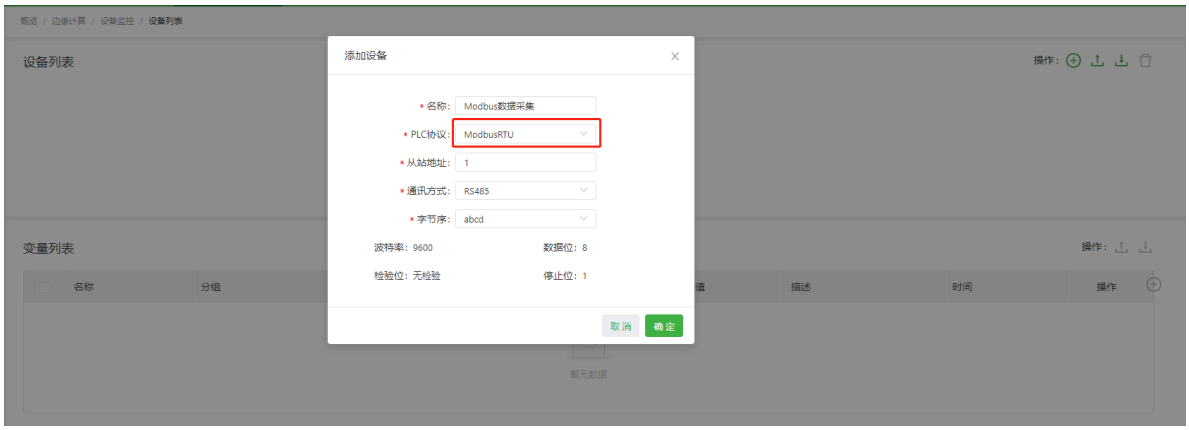
  进入“边缘计算 > 设备监控 > 设备列表”页面，点击“添加 PLC”按钮，在添加设备页面选择 PLC 协议为“ModbusTCP”并配置 PLC 的通讯参数。(端口号和字节序默认为 502 和 abcd；使用时需根据实际情况调整）注意：设备名称不能重复。



  添加成功后如下图所示：

- 添加 ModbusRTU 通讯的 PLC 设备

  进入“边缘计算 > 设备监控 > 设备列表”页面，点击“添加 PLC”按钮，在添加设备页面选择 PLC 协议为“ModbusRTU”并配置 PLC 的通讯参数。注意：设备名称不能重复。



添加成功后如下图所示：

如需修改 RS232/RS485 串口的通讯参数，请在“边缘计算 > 设备监控 > 参数设置”页面修改。修改后所有串口设备的通讯参数将自动修改并按照修改后的通讯参数通讯。

- 添加 EtherNET/IP 设备（**要求 App 版本为 1.2.5 及以上**）

  进入“边缘计算 > 设备监控 > 设备列表”页面，点击“添加 PLC”按钮，在添加设备页面选择 PLC 协议为“EtherNET/IP”并配置 PLC 的通讯参数。注意：设备名称不能重复。

2.2.2 添加变量

- 添加 ISO on TCP 变量

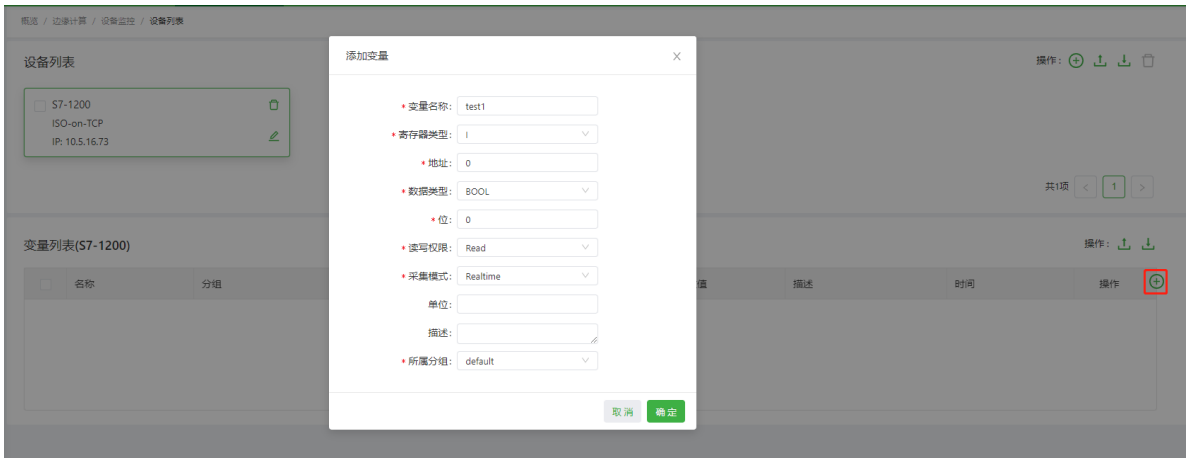
  在“设备列表”页面点击“添加变量”按钮，在弹出框中配置变量参数：

  - 变量名：变量名称（同一设备下变量名称不能重复）
  - 寄存器类型：变量寄存器类型，包括 I/Q/M/DB 四种类型
  - DB 索引：寄存器类型为 DB 时变量的 DB 号
  - 地址：变量的寄存器地址
  - 数据类型：变量数据类型，包括：
    - BOOL：True 或 False
    - BIT：0 或 1
    - BYTE：8 位无符号数据
    - SINT：8 位有符号数据
    - WORD：16 位无符号数据
    - INT：16 位有符号数据
    - DWORD：32 位无符号数据
    - DINT：32 位有符号数据
    - FLOAT：32 位浮点数
    - STRING：8 位字符串
    - BCD：16 位 BCD 码
  - 小数位：数据类型为 FLOAT 时变量小数点后的数据长度，最大 6 位
  - 长度：数据类型为 STRING 时字符串长度，读取 1 个字符串的长度为 1
  - 位：数据类型为 BOOL 或 BIT 时变量的位偏移，可输入 0~7 中任一数字
  - 读写权限：
    - Read：只读，不可写
    - Write：只写，不可读
    - Read/Write：可读可写
  - 采集模式：
    - Realtime：按照所属分组的采集间隔采集变量并按照上报间隔上报数据

    * Onchange：变量数值有变化时才采集变量并按照上报间隔上报数据

- 单位：变量单位
- 描述：变量描述
- 所属分组：变量所属的采集组

下图是添加一个地址为%I0.0 的开关变量的例子：

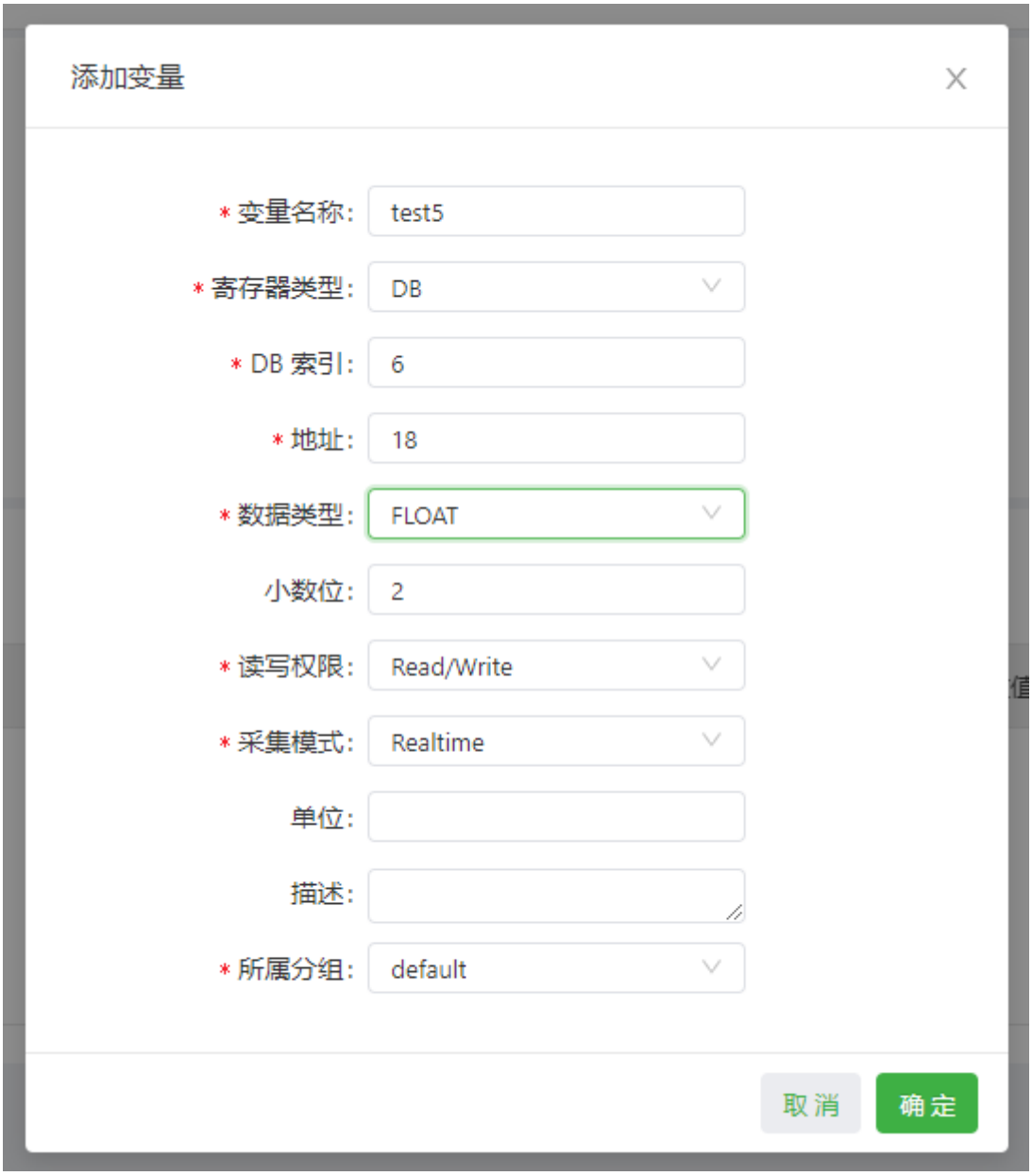


下图是添加一个地址为%IB1 的字节变量的例子：

下图是添加一个地址为%IW3 的字变量的例子:

下图是添加一个地址为%ID4 的双字变量的例子:

下图是添加一个地址为%DB6.DBD18 的浮点数变量的例子:



- 添加 Modbus 变量

  在“设备列表”页面点击“添加变量”按钮，在添加变量弹出框中配置 PLC 变量参数：

  - 变量名：变量名称（同一设备下变量名称不能重复）
  - 地址：变量的寄存器地址
  - 数据类型：变量数据类型，包括：
    - BOOL：True 或 False
    - BIT：0 或 1
    - WORD：16 位无符号数据

    - INT：16 位有符号数据
    - DWORD：32 位无符号数据
    - DINT：32 位有符号数据
    - FLOAT：32 位浮点数
    - STRING：8 位字符串
  - 小数位：数据类型为 FLOAT 时变量小数点后的数据长度，最大 6 位
  - 长度：数据类型为 STRING 时字符串长度
  - 位：地址为 `30001~40000`,`310001~365535`，`40001~50000`，`410001~465535` 且数据类型为 BOOL 或 BIT 时变量的位偏移，可输入 0~15 中任一数字
  - 读写权限：
    - Read：只读，不可写
    - Write：只写，不可读
    - Read/Write：可读可写
  - 采集模式：
    - Realtime：按照固定采集间隔采集变量并按照上报间隔上报数据
    - Onchange：变量数值变化后才采集并按照上报间隔上报数据
  - 单位：变量单位
  - 描述：变量描述
  - 所属分组：变量所属的采集组

下图是添加一个地址为 00001 的线圈变量的例子：

下图是添加一个地址为 10001 的开关变量的例子:

下图是添加一个地址为 30001 的整数变量的例子:

下图是添加一个地址为 40001 的浮点数变量的例子:

- 添加 EtherNET/IP 变量（**要求 App 版本为 1.2.5 及以上**）

  在“设备列表”页面点击“添加变量”按钮，在添加变量弹出框中配置变量参数。EtherNET/IP 变量无须配置数据类型，Device Supervisor 会自行判断数据的类型（目前支持的 EIP 数据类型包括 `BOOL`、`SINT`、`INT`、`DINT`、`REAL`、`STRING`）：

  - 变量名：变量名称（同一设备下变量名称不能重复）
  - 标签：PLC 中变量的标签
  - 小数位：数据类型为浮点数时变量小数点后的数据长度，最大 6 位
  - 读写权限：
    - Read：只读，不可写
    - Write：只写，不可读
    - Read/Write：可读可写

- 采集模式：
  - Realtime：按照固定采集间隔采集变量并按照上报间隔上报数据
  - Onchange：变量数值变化后才采集并按照上报间隔上报数据
- 单位：变量单位
- 描述：变量描述
- 所属分组：变量所属的采集组

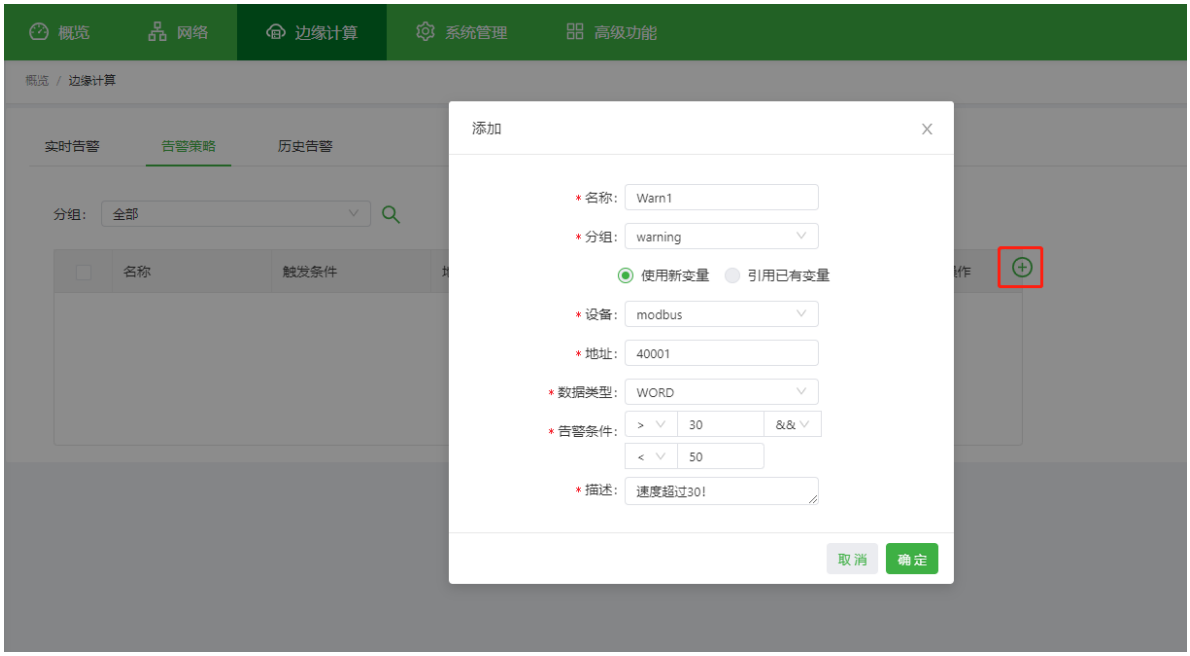
以下是添加一个标签名称为 `ZB.LEN.16` 的变量的例子：

2.2.3 配置告警策略

你可以进入“边缘计算 > 设备监控 > 告警 > 告警策略”页面配置告警策略，点击“添加”按钮后，在弹出框中配置告警策略参数。告警策略支持两种配置方式“使用新变量”和“引用已有变量”，参数如下：

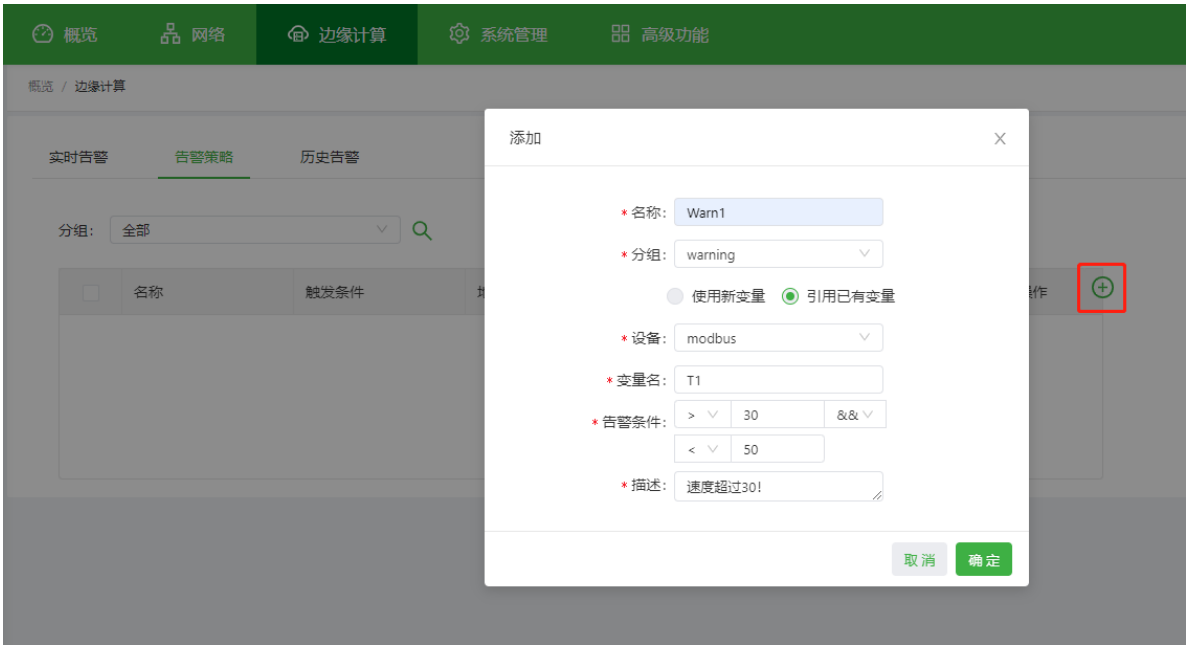
- 使用新变量
  - 名称：告警名称
  - 分组：告警所属分组
  - 变量来源：“使用新变量”即告警变量未在“设备列表”中配置，需要自行设置变量参数（该操作不会在“设备列表”中新增变量）
  - 设备：告警变量所属设备
  - 寄存器类型：变量寄存器类型，包括 I/Q/M/DB 四种类型（ISO on TCP 变量）
  - 地址：告警变量地址
  - 数据类型：告警变量数据类型
  - 告警条件
    - 判断条件：支持“=”、“!=”、“>”、“≥”、“<”、“≤”
    - 逻辑条件
      - 无逻辑条件：仅通过单个判断条件判断告警
      - &&：通过两个判断条件相与判断告警
      - ||：通过两个判断条件相或判断告警
  - 描述：告警描述

下图是添加一个告警变量，该变量数值 >30 且 <50 时产生告警；不在此范围时不产生告警或告警消除。



- 引用已有变量
  - 名称：告警名称
  - 分组：告警所属分组
  - 变量来源："引用已有变量"即告警变量已在"设备列表"中配置，可以输入已有变量名称直接使用
  - 设备：告警变量所属设备
  - 变量名：引用的变量名称
  - 告警条件
    - 判断条件：支持"="、"!="、">"、"≥"、"<"、"≤"
    - 逻辑条件
      - 无逻辑条件：仅通过单个判断条件判断告警
      - &&：通过两个判断条件相与判断告警
      - ||：通过两个判断条件相或判断告警
  - 描述：告警描述

下图是引用已有变量生成一条告警变量，该变量数值 >30 且 <50 时产生告警；不在此范围时不产生告警或告警消除。



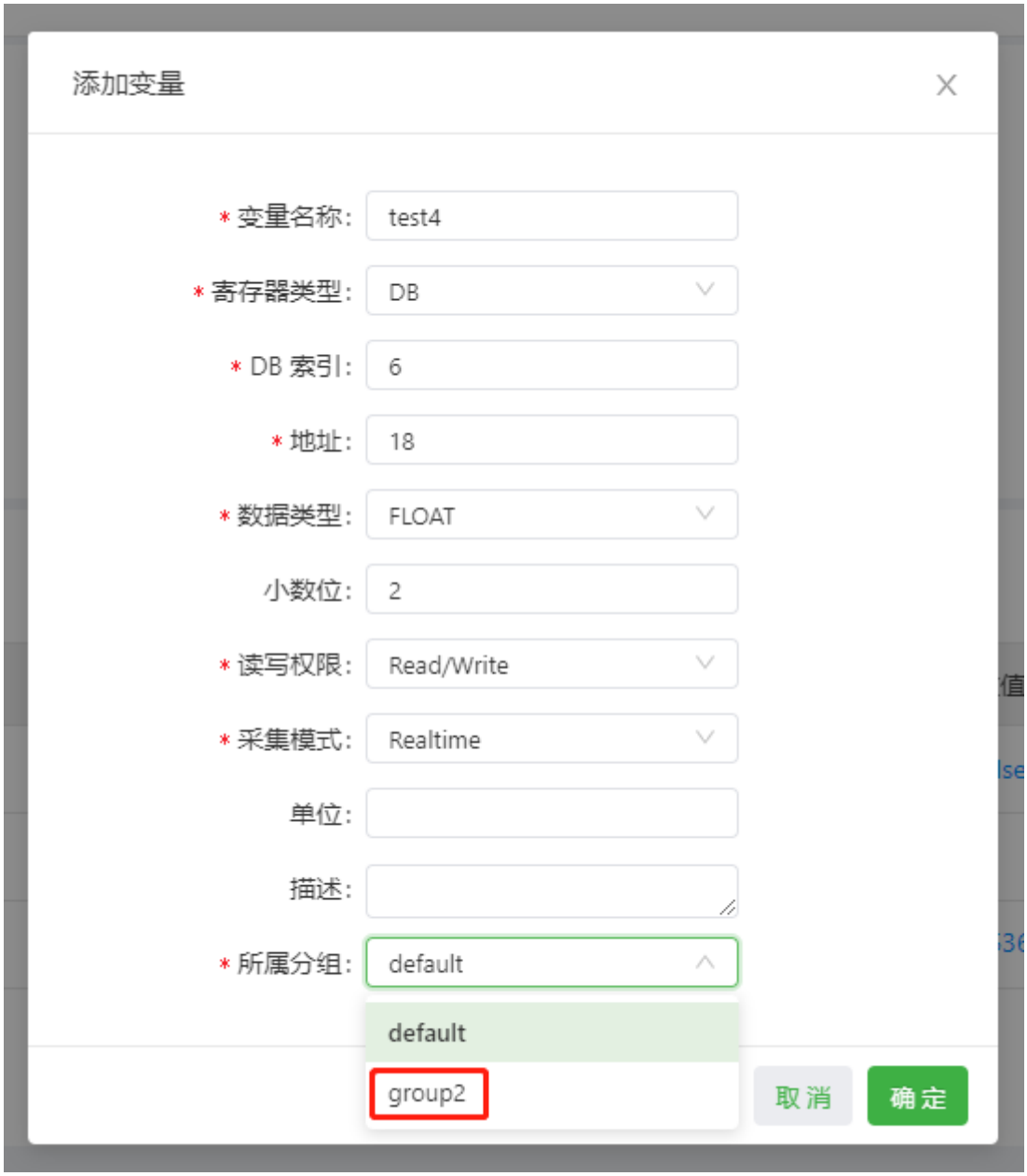
2.2.4 配置分组

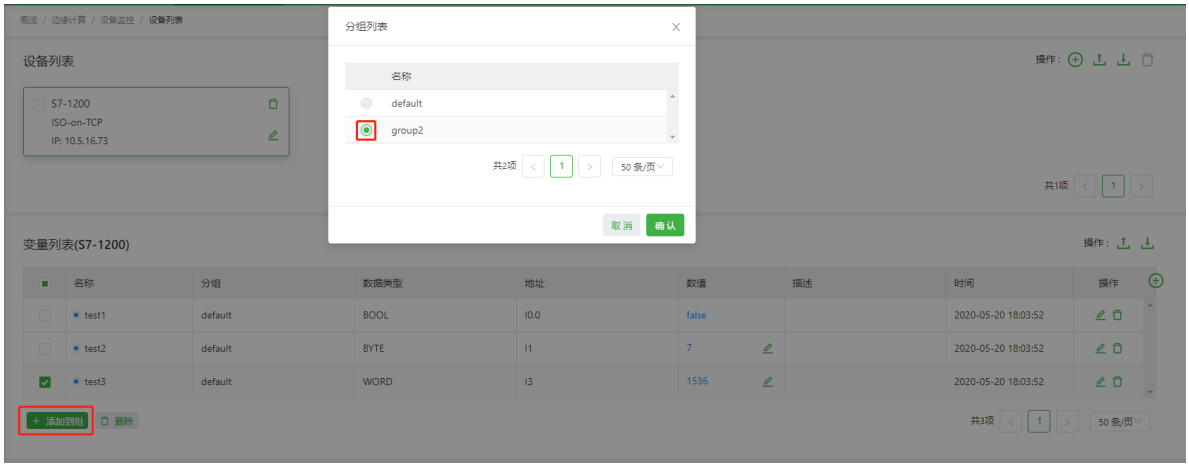
如需为变量或告警配置不同的采集间隔或需要按照不同的 MQTT 主题上报相应的变量数据时，可在“边缘计算 > 设备监控 > 分组”页面添加新分组。

- 添加采集分组

  下图添加了一个名为“group2”的采集分组，该采集分组每 5 秒采集一次分组中的变量：

添加采集分组后，添加变量时可以选择将变量关联到该分组或者在变量列表中选择需要关联的变量添加到指定分组中，分组中的变量会按照分组的采集间隔采集数据。

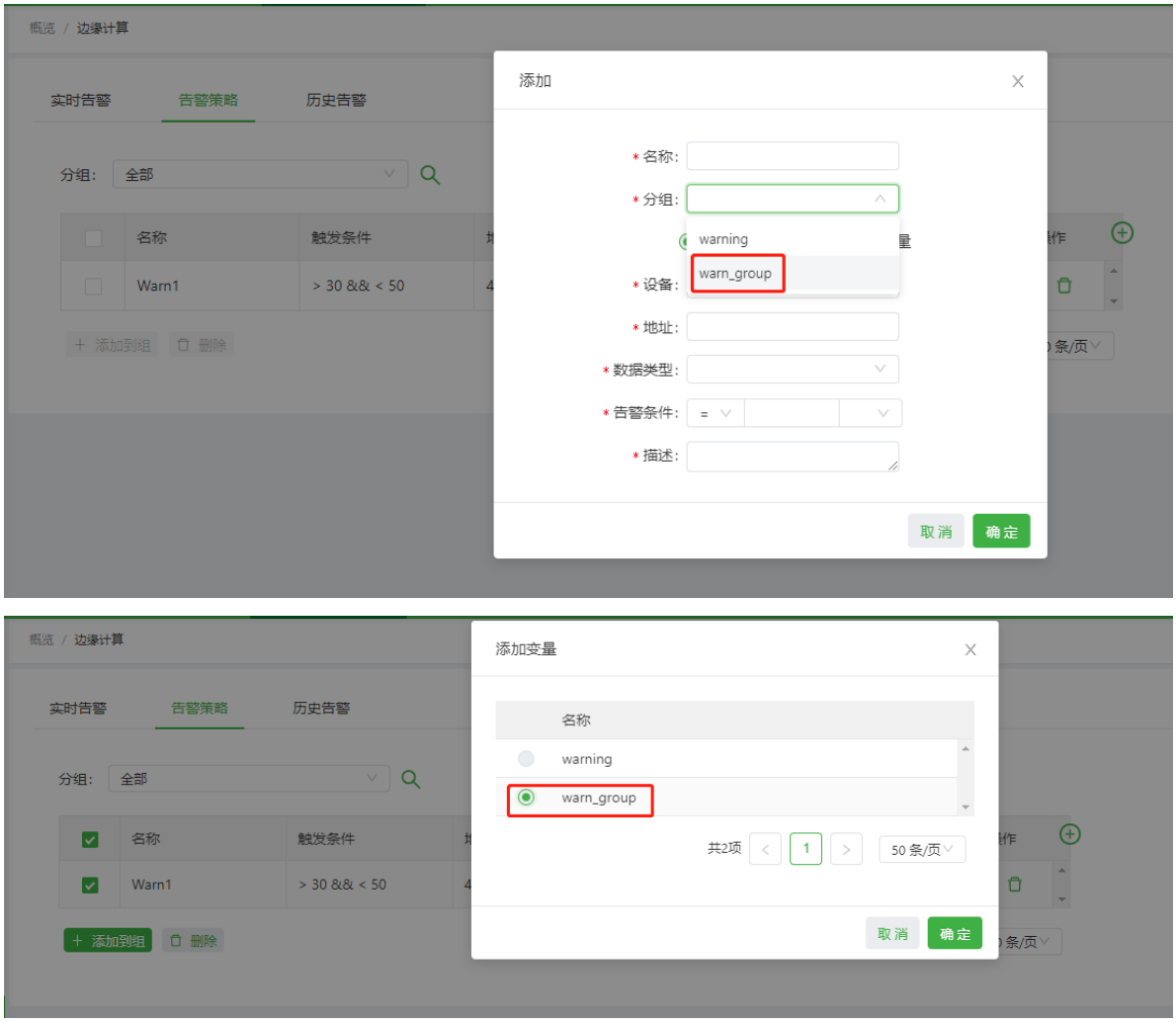

添加变量
* 变量名称: test4
* 寄存器类型: DB
* DB 索引: 6
* 地址: 18
* 数据类型: FLOAT
小数位: 2
* 读写权限: Read/Write
* 采集模式: Realtime
单位:
描述:
* 所属分组: default
default
group2
取消
确定



- 添加告警分组

  下图添加了一个名为 warn_group 的告警分组，该告警分组每 5 秒检测一次分组中的告警变量是否处于告警状态：

  添加告警分组后，添加告警策略时可以选择将告警策略关联到该分组或者在告警列表中选择需要关联的告警策略添加到指定分组中，分组中的告警策略会按照分组的采集间隔检测变量告警状态。



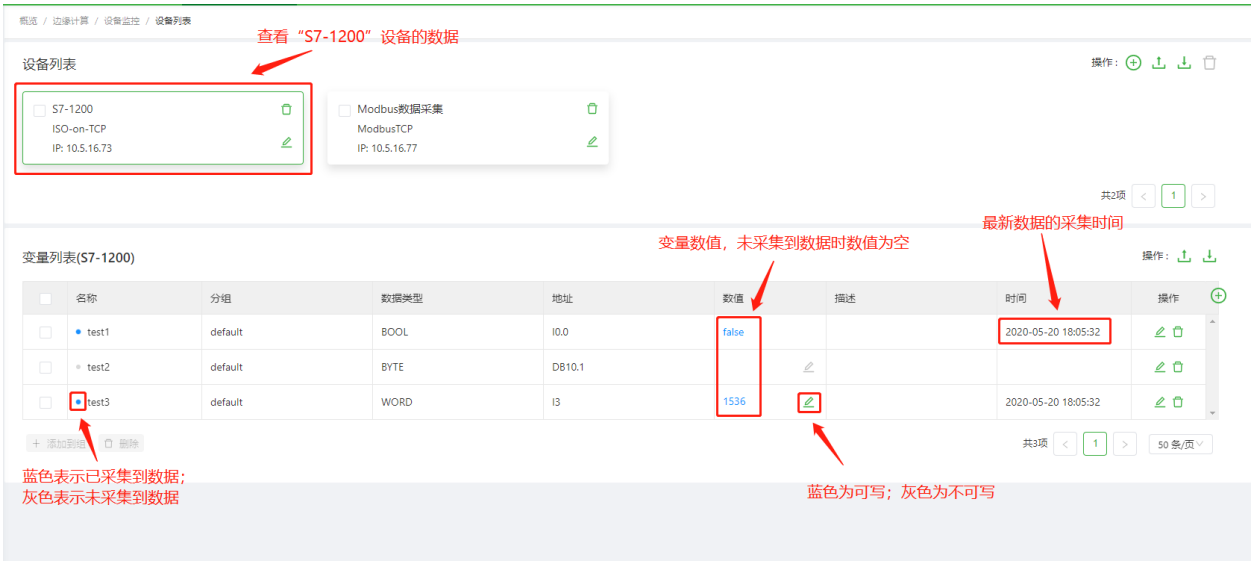
1.2.4 3. 上报和监控 PLC 数据

- *3.1 本地监控 PLC 数据*
- *3.2 云平台监控 PLC 数据*

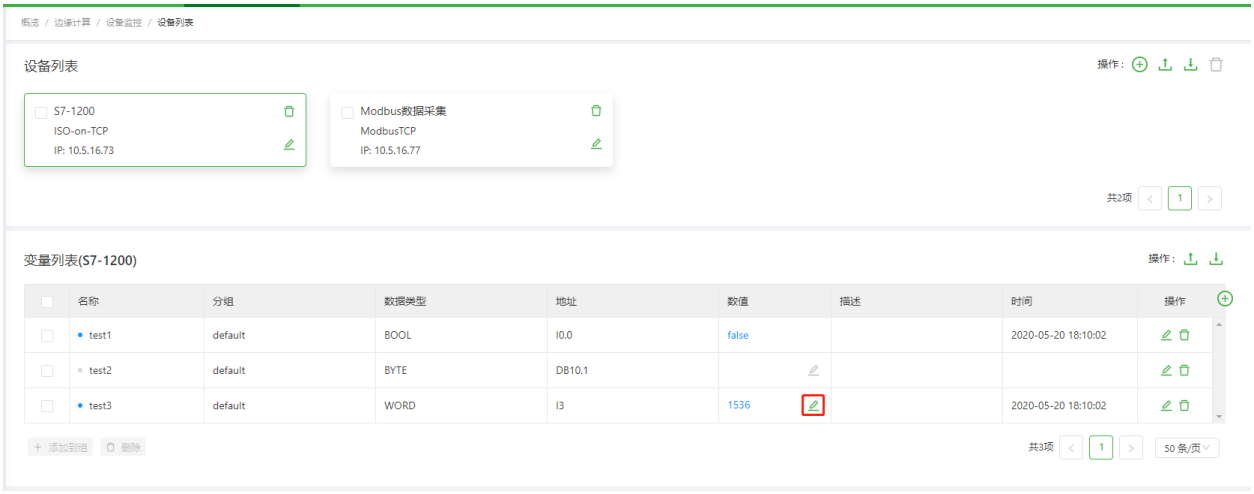
3.1 本地监控 PLC 数据

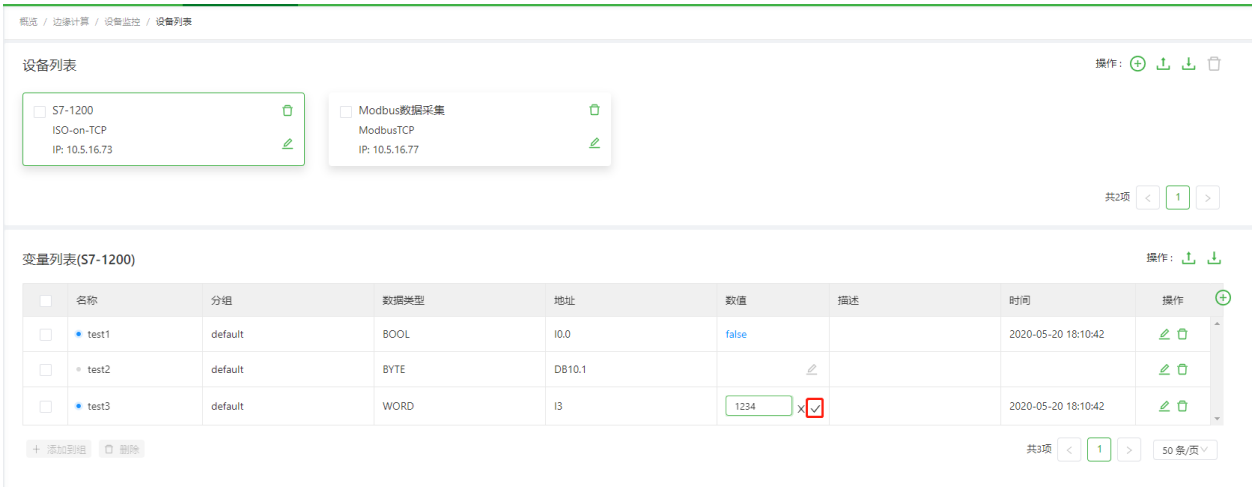
3.1.1 本地监控数据采集

数据采集配置完成后，可以在“边缘计算 > 设备监控 > 设备列表”页面查看数据采集情况。点击设备列表中的设备卡片可切换需要查看的 PLC 数据。



点击数值栏的按钮可进行写入操作。





修改成功如下图所示：

3.1.2 本地监控告警

告警策略配置完成后，可以在“边缘计算 > 设备监控 > 告警”页面查看变量告警情况。

- 实时告警：查看当前未消除的告警信息

- 历史告警：筛选查看任意告警信息

3.2 云平台监控 PLC 数据

- *3.2.1 配置 Thingsboard*
- *3.2.2 配置云服务上报和接收下发数据*

3.2.1 配置 Thingsboard

Thingsboard 的详细使用方法请查看Thingsboard 入门手册，您也可以按照 *Thingsboard 参考流程*进行测试。

3.2.2 配置云服务上报和接收下发数据

进入“边缘计算 > 设备监控 > 云服务”页面，勾选启用云服务并配置相应的 MQTT 连接参数，配置完成后点击提交。

- 类型：Thingsboard 的连接方式为标准 `MQTT`。阿里云 `IoT` 的使用方法请参考阿里云 IoT 使用说明；`AWS IoT` 的使用方法请参考AWS IoT 使用说明；`Azure IoT` 的使用方法请参考Azure IoT 使用说明
- 服务器地址：Thingsboard 的 demo 地址为 `demo.thingsboard.io`
- `MQTT 客户端 ID`：任一唯一 ID
- `MQTT 用户名`：Thingsboard 设备的访问令牌，访问令牌获取方式见*传输 PLC 数据到 Thingsboard 设备*
- `MQTT 密码`：任意 6~32 位密码

- 其余项使用默认配置即可

配置完成后如下图所示：

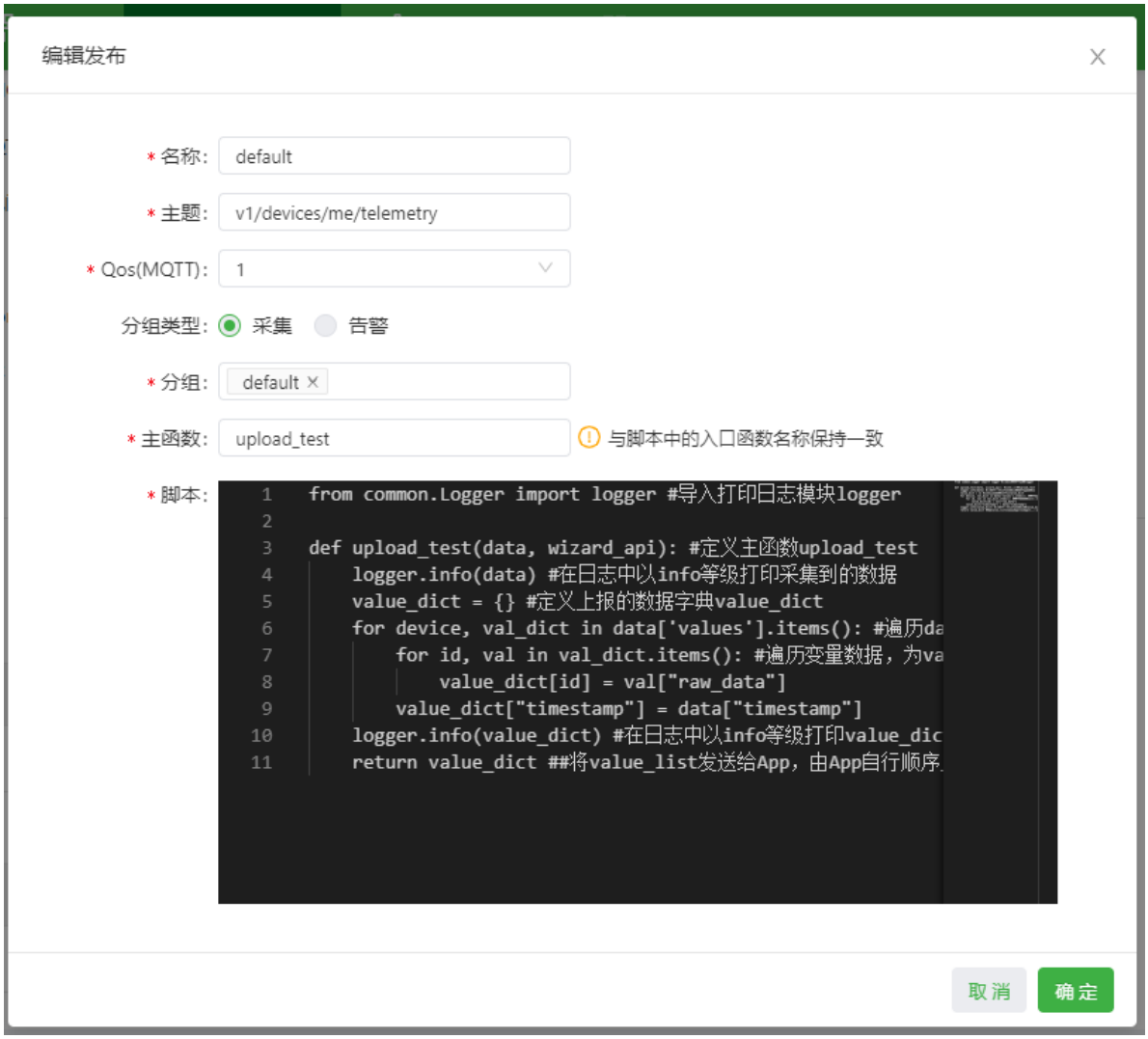
提交后点击“消息管理”以配置发布和订阅消息。发布和订阅消息的配置方法请参考消息管理（自定义 *MQTT* 发布/订阅）。以下是示例配置：

- 发布消息：
  - 主题：`v1/devices/me/telemetry`
  - Qos(MQTT)：`1`
  - 分组类型：采集
  - 分组：需要上传数据至 thingsboard 的分组名称，本文档为 `default`
  - 主函数：入口函数名称，本文档为 `upload_test`

- 脚本：

```
from common.Logger import logger #导入打印日志模块logger

def upload_test(data, wizard_api): #定义主函数upload_test
    logger.info(data) #在日志中以info等级打印采集到的数据
    value_dict = {} #定义上报的数据字典value_dict
    for device, val_dict in data['values'].items():
↪#遍历data中的values字典，该字典中包含设备名称和设备下的变量数据
        for id, val in val_dict.items(): #遍历变量数据，为value_dict字典赋值
            value_dict[id] = val["raw_data"]
        value_dict["timestamp"] = data["timestamp"]
    logger.info(value_dict) #在日志中以info等级打印value_dict
    return value_dict #将value_
↪list发送给App，由App自行顺序上传至MQTT服务器。最终的value_list格式为{'bool
↪': False, 'byte': 7, 'real': 0.0, 'timestamp': 1583990892.5429199}
```

配置完成后如下图所示：



- 订阅消息:
  - 主题：v1/devices/me/rpc/request/+
  - Qos(MQTT)：1
  - 主函数：入口函数名称，本文档为 ctl_test
  - 脚本：

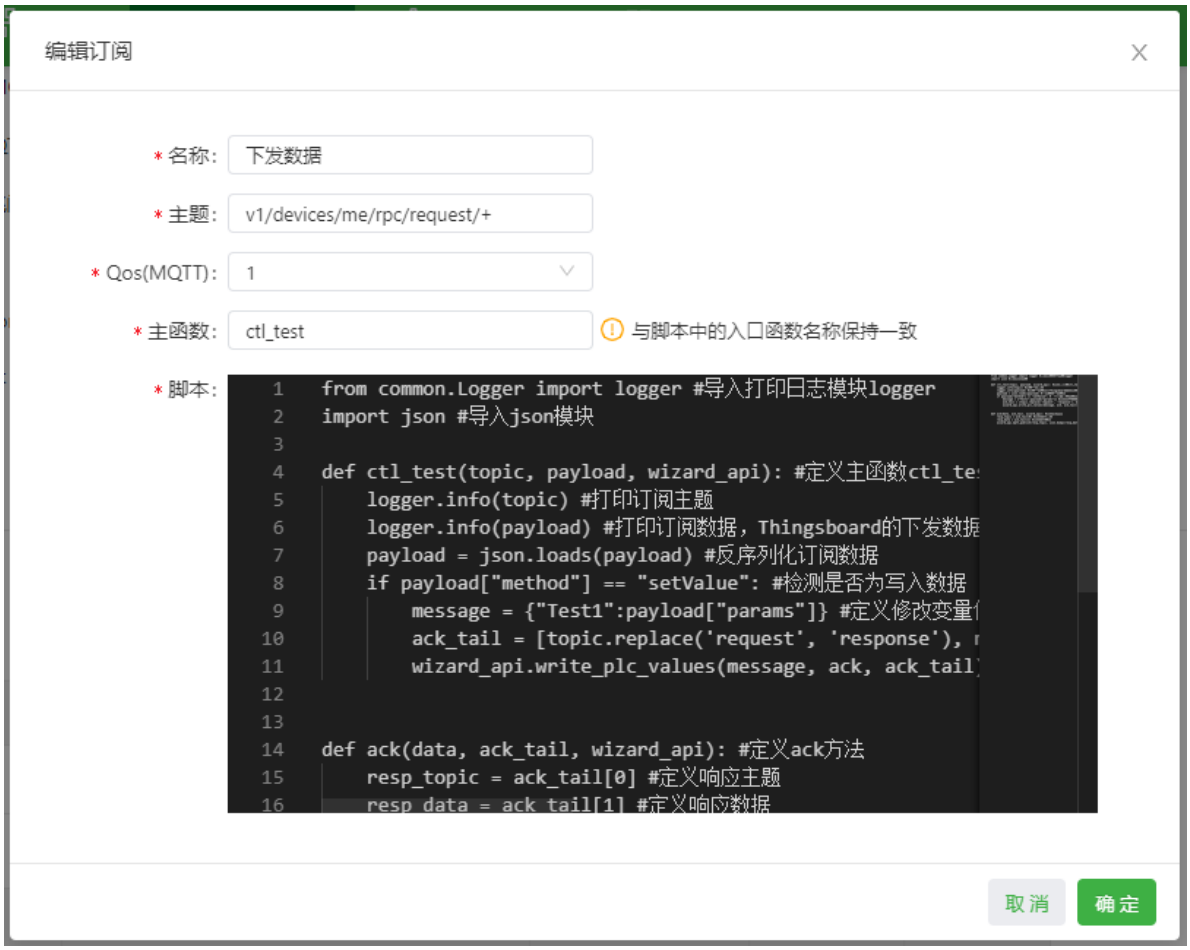
```
from common.Logger import logger #导入打印日志模块logger
import json #导入json模块

def ctl_test(topic, payload, wizard_api): #定义主函数ctl_test
    logger.info(topic) #打印订阅主题
    logger.info(payload) #打印订阅数据，Thingsboard的下发数据格式为{"method":
↪"setValue","params":true}
    payload = json.loads(payload) #反序列化订阅数据
```

(续下页)

(接上页)

```
    if payload["method"] == "setValue": #检测是否为写入数据
        message = {"bool":payload["params"]}
↪#定义修改变量值消息，包括变量名称和变量值
        ack_tail = [topic.replace('request', 'response'), message]
↪#定义确认数据，包括响应的主题和消息
        wizard_api.write_plc_values(message, ack, ack_tail) #调用write_plc_
↪values方法，将message字典中的数据下发至指定变量；调用ack方法并发送ack_
↪tail给ack方法


def ack(data, ack_tail, wizard_api): #定义ack方法
    resp_topic = ack_tail[0] #定义响应主题
    resp_data = ack_tail[1] #定义响应数据
    wizard_api.mqtt_publish(resp_topic, json.dumps(resp_data), 1) #调用mqtt_
↪publish将响应数据发送给MQTT服务器，响应数据格式为{'bool': True}
```

配置完成后如下图所示：



1.2.5 附录

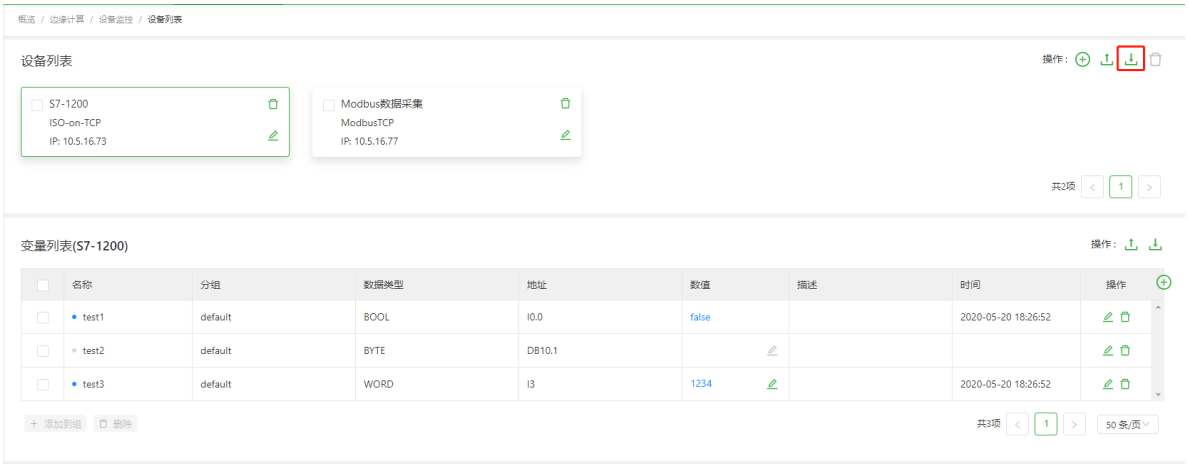
- 导入导出数据采集配置
- 消息管理（自定义 *MQTT* 发布/订阅）
- 参数设置
- 网关的其他配置
- *Thingsboard* 参考流程

导入导出数据采集配置

Device Supervisor 的数据采集配置总共包含四个 CSV 格式的配置文件（App 版本 `1.2.5` 及以上才支持告警策略配置文件），您可以通过导入导出配置文件快速实现采集配置。各配置文件内容如下：

- `device.csv`：设备配置文件，详细参数如下
  - `Device Name`：设备名称
  - `Protocol`：通讯协议名称，如 `ModbusTCP`
  - `Ip/Serial`：以太网设备填写 ip 地址；串口设备填写 `RS485` 或 `RS232`
  - `Port`：以太网设备的通讯端口号
  - `Rack`（仅 ISO on TCP 设备）：设备机架号
  - `Slot`（仅 ISO on TCP 设备）：设备槽号
  - `Mode`（仅 ISO on TCP 设备）：ISO on TCP 模式，包括 `TSAP` 和 `Rack/Slot`
  - `Slave`（仅 Modbus 设备）：从站地址
  - `Byte Order`（仅 Modbus 设备）：字节序，包括 `abcd`、`badc`、`cdab`、`dcba`

  导出方式为设备列表页面的设备列表导出。

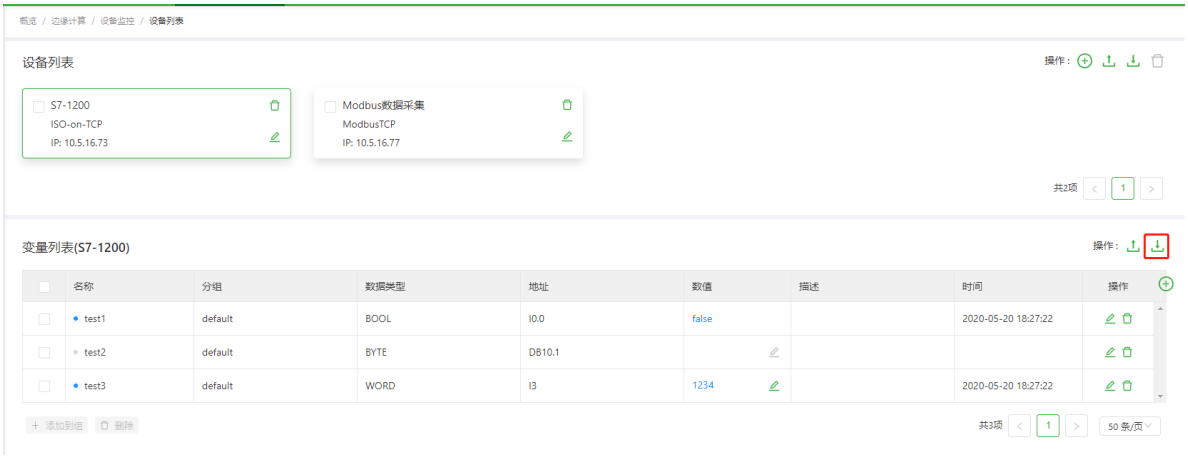


示例配置如下：

Device Name	Protocol	Ip	Port	Slave	Byte Order
Modbustest	ModbusTCP	10.5.16.82	502	1	cdab

- `var.csv`: 变量配置文件，详细参数如下
  - `Var Name`：变量名称
  - `Device`：变量所属设备
  - `Protocol`：通讯协议名称
  - `Dbnumber`（仅 ISO on TCP 设备）：DB 号
  - `Register Type`（仅 ISO on TCP 设备）：寄存器类型，如 `DB`
  - `Register Addr`：寄存器地址
  - `Register Bit`：位偏移
  - `Data Type`：数据类型
  - `Read Write`：读写权限，包括 `Read/Write`、`Write`、`Read`
  - `Float Repr`：小数位，`1~6`
  - `Mode`：采集模式，包括 `realtime`、`onchange`
  - `Unit`：单位
  - `Size`：字符串长度
  - `Desc`：描述
  - `Group`：所属分组

导出方式为设备列表页面的变量列表导出。



示例配置如下：

Var Name	Device	Protocol	Dbnumber	Register Type	Register Addr	Register Bit	Data Type	Read Write	Float Repr	Mode	Unit	Size	Desc	Group
SP1	S7-1200	ISO-on-TCP	0	I	0	0	BOOL	read/write	2	realtime		1		default

- group.csv: 分组配置文件，详细参数如下
  - Group Name：分组名称
  - Polling Interval：采集间隔
  - Upload Interval：上传间隔。分组类型为 alarm 时此项为空即可
  - Group Type：分组类型，支持 alarm 和 collect

  导出方式为分组页面的分组导出。

  示例配置如下：

Group Name	Polling Interval	Upload Interval	Group Type
default	10	10	collect

- warn.csv：告警策略配置文件，详细参数如下：
  - Warn Name：告警名称
  - Group：告警所属分组
  - Quotes：是否引用变量。0 为“使用新变量”，1 为“引用已有变量”
  - Device：告警变量所属设备
  - Var Name：引用的变量名称。未引用变量时留空即可
  - Condition1：告警条件 1。Eq：等于,Neq：不等于,Gt：大于,Gne：大于等于,Lne：小于等于,Lt：小于
  - Operand1：告警阈值 1
  - Combine Method：告警条件连接方式。None：空，And：&&，Or：||
  - Condition2：告警条件 2
  - Operand2：告警阈值 2

- `Alarm Content`：告警描述
- `Register Addr`：告警变量地址
- `Dbnumber`：告警变量寄存器类型为 DB 时变量的 DB 号
- `Data Type`：告警变量数据类型。配置 EtherNET/IP 和 OPCUA 变量时留空即可
- `Symbol`：告警变量标签名称。配置 EtherNET/IP 变量时需要填写
- `Register Type`：告警变量寄存器类型
- `Register Bit`：告警变量数据类型为 BOOL 或 BIT 时变量的位偏移
- `Namespace Index`：告警变量为 OPCUA 协议时的命名空间索引
- `Identifier`：告警变量为 OPCUA 协议时的识别码
- `Identifier Type`：告警变量为 OPCUA 协议时的 ID 类型
- `Float Repr`：小数位

导出方式为告警策略页面的告警导出。

消息管理（自定义 MQTT 发布/订阅）

您可以在“边缘计算 > 设备监控 > 云服务”配置你的 MQTT 连接参数，通过消息管理功能配置上报数据的 MQTT 主题、数据来源等参数并支持使用 Python 语言自定义 MQTT 发布和订阅消息的数据上报、处理等逻辑。无需二次开发即可实现与多种 MQTT 服务器进行数据上传和下发。以下将为您说明“消息管理”的使用方法。

- 配置发布消息
- 配置订阅消息
- *Device Supervisor* 的 *api* 接口说明（*wizard_api*）
- *Device Supervisor api* 回调函数说明

配置发布消息

自定义发布消息中包含以下配置项：

- 名称：用户自定义发布名称
- 主题：发布主题，与 MQTT 服务器订阅的主题保持一致
- Qos(MQTT)：发布 Qos，建议与 MQTT 服务器的 Qos 保持一致
  - 0：只发送一次消息，不进行重试
  - 1：最少发送一次消息，确保消息到达 MQTT 服务器
  - 2：确保消息到达 MQTT 服务器且只收到一次
- 分组类型：发布变量数据时请选择“采集”，随后在分组中仅能选择“采集组”；发布告警数据时请选择“告警”，随后在分组中仅能选择“告警组”
- 分组：选择相应的分组后，分组下所有变量通过该发布配置将数据上传至 MQTT 服务器；可选择多个分组，当选择多个分组时，按照分组的采集间隔分别对各分组下的变量执行发布中的脚本逻辑。分组中必须包含变量，否则不会执行发布中的脚本逻辑
- 主函数：主函数名称，即入口函数名称，与脚本中的入口函数名称保持一致
- 脚本：使用 Python 代码自定义组包和处理逻辑，发布中的主函数参数包括：
  - 参数 1：Device Supervisor 将采集后的变量数据发送给该参数，数据格式如下：
    - 变量数据格式：

```
{
    'timestamp': 1589434519.5458372,  #数据产生时间戳
    'group_name': 'default',  #采集组名称
    'values':  #变量数据字典，包含PLC名称，变量名称和变量值
    {
        'S7-1200':  #PLC名称
        {
            'Test1':  #变量名称
            {
                'raw_data': False,  #变量值
                'status': 1  #采集状态，非1即采集异常
            },
            'Test2':
            {
                'raw_data': 2,
                'status': 1
            }
        }
```

(续下页)

(接上页)

```
    }
}
```

* 告警数据格式：

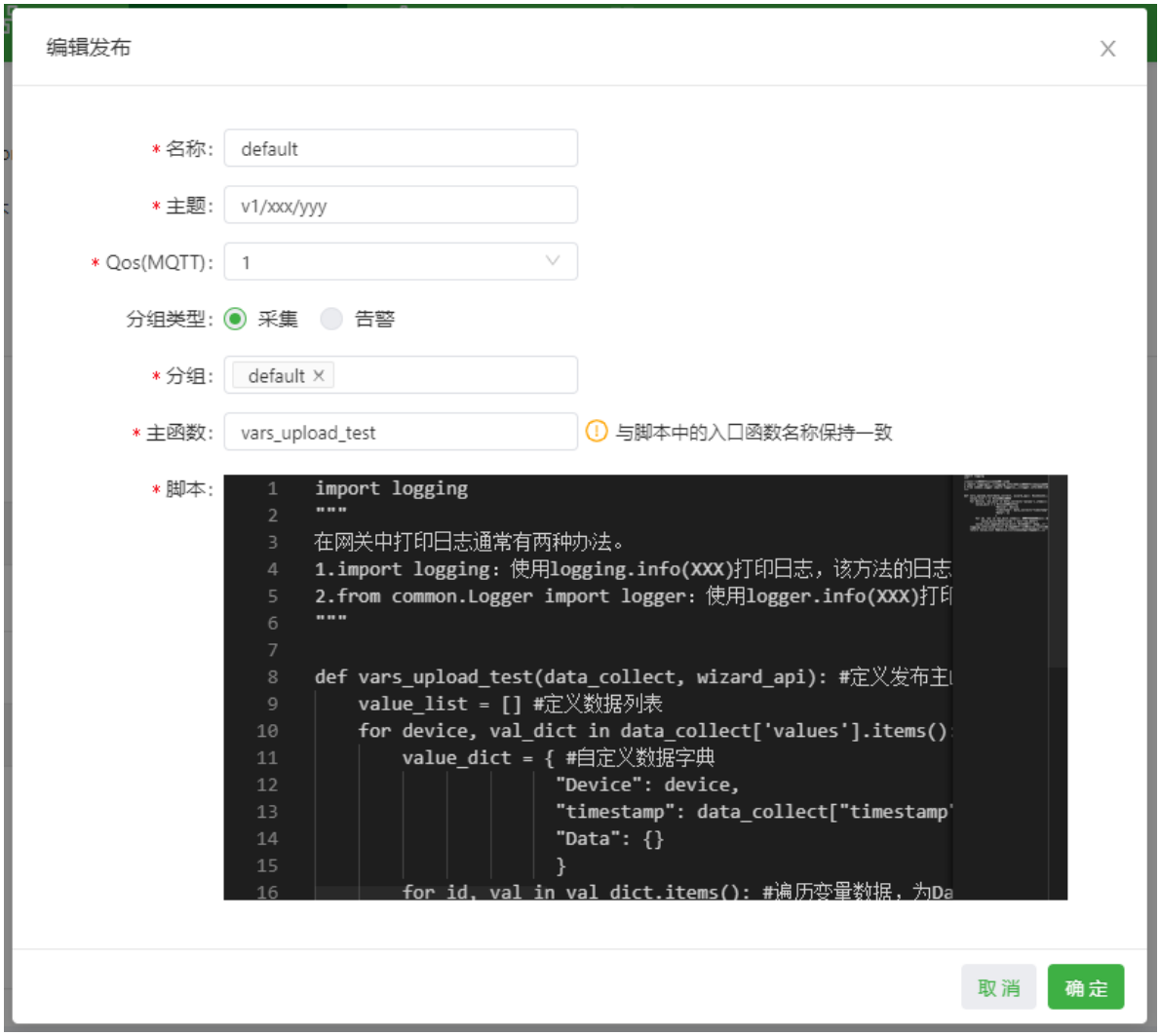
```
{
    'timestamp': 1589434527.3628697,  #告警产生时间戳
    'group_name': 'warning',  #告警组名称
    'values':  #告警数据字典，包含告警名称等告警信息
    {
        'Warn1':  #告警名称
        {
            'timestamp': 1589434527,  #告警产生时间戳
            'current': 'on',  #告警状态。on:已触发，off:已消除
            'status': 0,  #告警状态。0:已触发，1:已消除
            'value': 33,  #告警触发时告警变量的数值
            'alarm_content': '速度超过30! ',  #告警描述
            'level': 1  #预留字段
        }
    }
}
```

– 参数 2：该参数为 Device Supervisor 提供的 api 接口，参数说明见 *Device Supervisor 的 api 接口说明（wizard_api）*

以下是常见的自定义发布方法示例（请勿将 `mqtt_publish` 或 `save_data` 方法与 `return` 命令同时使用）：

- 发布示例 1：使用 `return` 方式上传变量数据

  本示例实现了使用 `return` 方式上传变量数据，将处理后的变量数据使用 `return` 命令发送给 Device Supervisor。Device Supervisor 自行根据发布中配置的主题和 Qos 将变量数据按照采集时间顺序上传至 MQTT 服务器。如果发送失败则缓存变量数据等待 MQTT 连接正常后按采集时间顺序上传至 MQTT 服务器。发布和代码配置示例如下：




```
import logging
"""
在网关中打印日志通常有两种办法。
1.import logging: 使用logging.info(XXX)打印日志，该方法的日志显示不受参数设置页面中的日志等级参数控制。
↪
2.from common.Logger import logger: 使用logger.info(XXX)打印日志，该方法的日志显示受参数设置页面中的日志等级参数控制。
↪
"""

def vars_upload_test(data_collect, wizard_api): #定义发布主函数
    value_list = [] #定义数据列表
    for device, val_dict in data_collect['values'].items():
↪#遍历values字典，该字典中包含设备名称和设备下的变量数据
        value_dict = { #自定义数据字典
                        "Device": device,
                        "timestamp": data_collect["timestamp"],
```

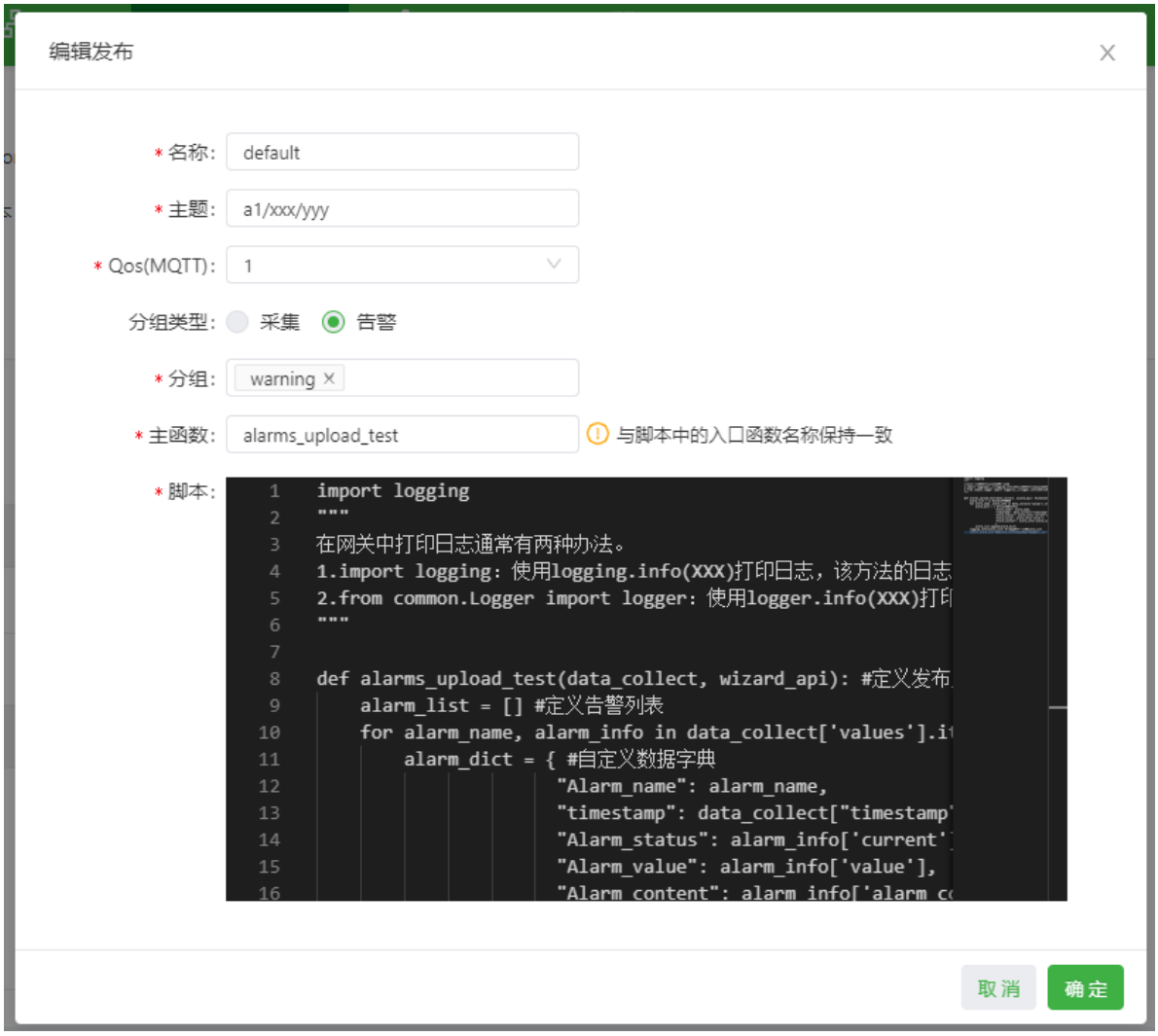
(续下页)

(接上页)

```
                    "Data": {}
                    }
        for id, val in val_dict.items(): #遍历变量数据，为Data字典赋值
            value_dict["Data"][id] = val["raw_data"]
        value_list.append(value_dict) #依次将value_dict添加到value_list中
    logging.info(value_list) #在App日志中打印value_list，数据格式为[{'Device':
↪'S7-1200', 'timestamp': 1589538347.5604711, 'Data': {'Test1': False, 'Test2':␣
↪12}}]
    return value_list #将value_
↪list发送给App，App将自行按照采集时间顺序上传至MQTT服务器。如果发送失败则缓存数据等待连接恢复后
```

- 发布示例 2：使用 `return` 方式上传告警数据

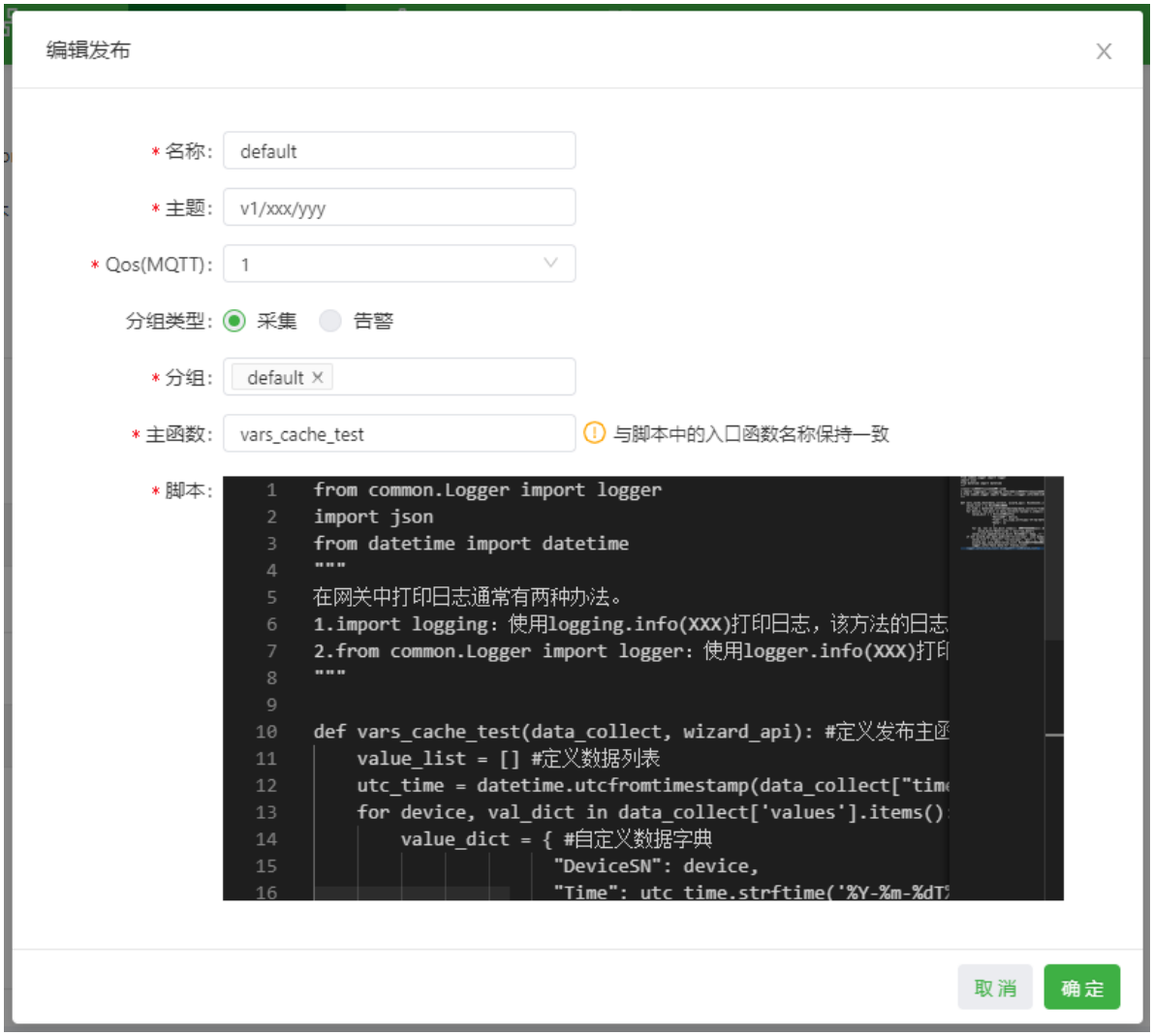
  本示例实现了上传告警数据。发布和代码配置示例如下：



```
import logging
"""
在网关中打印日志通常有两种办法。
1.import logging: 使用logging.
↪info(XXX)打印日志，该方法的日志显示不受参数设置页面中的日志等级参数控制。
2.from common.Logger import logger: 使用logger.
↪info(XXX)打印日志，该方法的日志显示受参数设置页面中的日志等级参数控制。
"""

def alarms_upload_test(data_collect, wizard_api): #定义发布主函数
    alarm_list = [] #定义告警列表
    for alarm_name, alarm_info in data_collect['values'].items():
↪#遍历values字典，该字典中包含告警名称和告警时间等信息
        alarm_dict = { #自定义数据字典
                        "Alarm_name": alarm_name,
                        "timestamp": data_collect["timestamp"],
                        "Alarm_status": alarm_info['current'],
                        "Alarm_value": alarm_info['value'],
                        "Alarm_content": alarm_info['alarm_content']
                        }
        alarm_list.append(alarm_dict) #依次将alarm_dict添加到alarm_list中
    logging.info(alarm_list) #在App日志中打印alarm_list
    return alarm_list #将alarm_
↪list发送给App，App将自行按照采集时间顺序上传至MQTT服务器。如果发送失败则缓存数据等待连接恢复后
```

- 发布示例 3：使用 `mqtt_publish` 上传变量数据并使用 `save_data` 存储上传失败的变量数据

  本示例实现了使用 `mqtt_publish` 方法将变量数据上传至 MQTT 服务器，当 MQTT 连接异常导致变量数据上传失败时，使用 `save_data` 方法存储主题、qos 和变量数据至数据库，存储的变量数据会在 MQTT 连接正常时按照先存先传的方式通过存储数据中的主题和 Qos 上传至 MQTT 服务器。发布和代码配置示例如下：




```
from common.Logger import logger
import json
from datetime import datetime
"""
在网关中打印日志通常有两种办法。
1.import logging: 使用logging.
↪info(XXX)打印日志，该方法的日志显示不受参数设置页面中的日志等级参数控制。
2.from common.Logger import logger: 使用logger.
↪info(XXX)打印日志，该方法的日志显示受参数设置页面中的日志等级参数控制。
"""

def vars_cache_test(data_collect, wizard_api): #定义发布主函数
    value_list = [] #定义数据列表
    utc_time = datetime.utcfromtimestamp(data_collect["timestamp"])
↪#转换LINUX时间戳为UTC时间
    for device, val_dict in data_collect['values'].items():
```

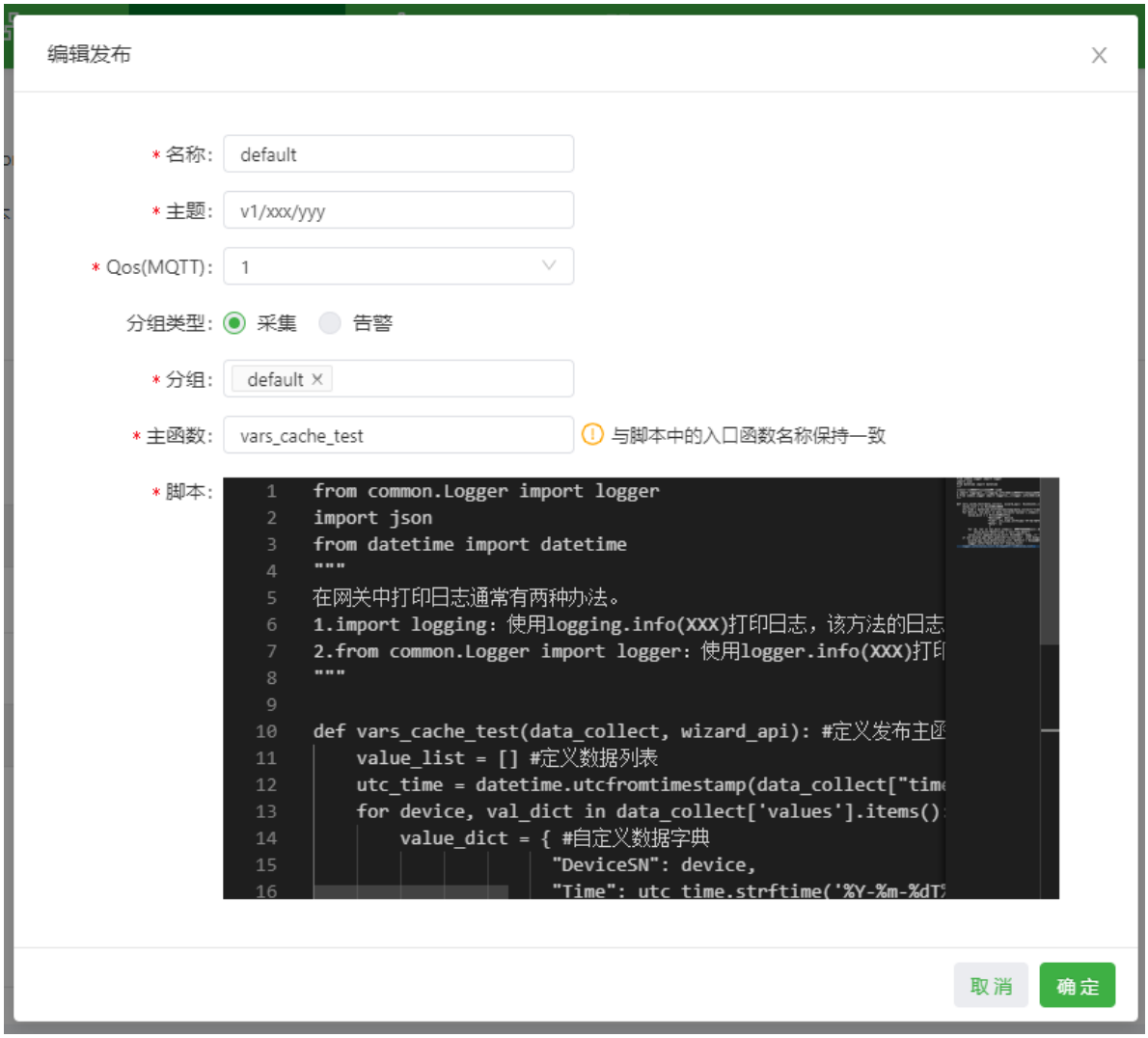
(续下页)

(接上页)

```
↪#遍历values字典，该字典中包含设备名称和设备下的变量数据
        value_dict = { #自定义数据字典
                        "DeviceSN": device,
                        "Time": utc_time.strftime('%Y-%m-%dT%H:%M:%S.%fZ'),
                        "Data": {}
                         }
        for id, val in val_dict.items(): #遍历变量数据，为Data字典赋值
            value_dict["Data"][id] = val["raw_data"]
        value_list.append(value_dict) #依次将value_dict添加到value_list中
    if not wizard_api.mqtt_publish("v1/xxx/yyy", json.dumps(value_list), 1):
↪#调用wizard_api模块中的mqtt_publish方法将value_list数据通过主题“v1/xxx/
↪yyy”，qos等级1发送至MQTT服务器并检测是否发送成功
        value_list = {"topic": "v1/xxx/yyy", "qos": 1, "payload": value_list}
        wizard_api.save_data(value_list)
↪#发送失败则存储数据，等待连接恢复后将按时间顺序上传存储数据
        logger.info("Save data:%s" %value_list)
    logger.info(value_list) #在App日志中打印value_list
```

- 发布示例 4：使用 `mqtt_publish` 上传变量数据并使用 `save_data` 存储上传失败的变量数据

  本示例实现了使用 `mqtt_publish` 方法将变量数据上传至 MQTT 服务器，当 MQTT 连接异常导致变量数据上传失败时，使用 `save_data` 方法存储变量数据和分组名称，存储的变量数据会在 MQTT 连接正常时按照先存先传的方式将变量数据按照分组在云服务中关联的主题和 Qos 上传至 MQTT 服务器（请勿将 `mqtt_publish` 或 `save_data` 方法与 `return` 命令同时使用）。发布和代码配置示例如下：




```
from common.Logger import logger
import json
from datetime import datetime
"""
在网关中打印日志通常有两种办法。
1.import logging: 使用logging.info(XXX)打印日志，该方法的日志显示不受参数设置页面中的日志等级参数控制。
2.from common.Logger import logger: 使用logger.info(XXX)打印日志，该方法的日志显示受参数设置页面中的日志等级参数控制。
"""

def vars_cache_test(data_collect, wizard_api): #定义发布主函数
    value_list = [] #定义数据列表
    utc_time = datetime.utcfromtimestamp(data_collect["timestamp"])
#转换LINUX时间戳为UTC时间
```

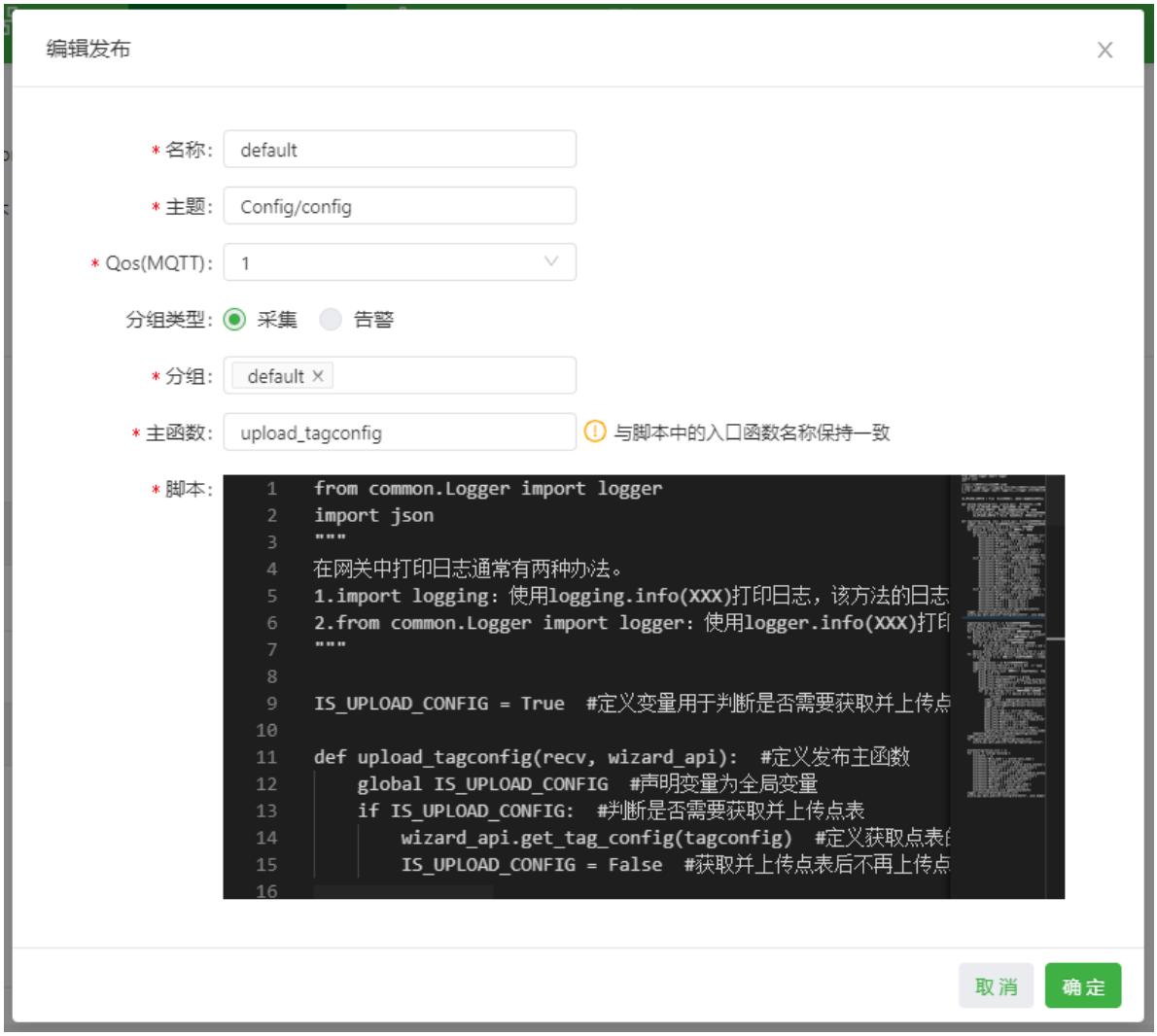
(续下页)

(接上页)

```
    for device, val_dict in data_collect['values'].items():
↪#遍历values字典，该字典中包含设备名称和设备下的变量数据
        value_dict = { #自定义数据字典
                        "DeviceSN": device,
                        "Time": utc_time.strftime('%Y-%m-%dT%H:%M:%S.%fZ'),
                        "Data": {}
                         }
        for id, val in val_dict.items(): #遍历变量数据，为Data字典赋值
            value_dict["Data"][id] = val["raw_data"]
        value_list.append(value_dict) #依次将value_dict添加到value_list中
    if not wizard_api.mqtt_publish("v1/xxx/yyy", json.dumps(value_list), 1):
↪#调用wizard_api模块中的mqtt_publish方法将value_list数据通过主题“v1/xxx/
↪yyy”，qos等级1发送至MQTT服务器并检测是否发送成功
        wizard_api.save_data(value_list,'default')
↪#发送失败则存储数据，等待连接恢复后将按时间顺序上传存储数据
        logger.info("Save data:%s" %value_list)
    logger.info(value_list) #在App日志中打印value_list
```

- 发布示例 5：使用 `get_tag_config` 获取设备、变量和告警点表

  本示例实现了每次重启 App 时使用 `get_tag_config` 方法获取设备、变量和告警点表并分别上传至 MQTT 服务器（该示例仅适用于获取 Modbus 以及 ISO on TCP 的 Rack/slot 模式点表）。发布和代码配置示例如下：




```
from common.Logger import logger
import json
"""
在网关中打印日志通常有两种办法。
1.import logging: 使用logging.
↪info(XXX)打印日志，该方法的日志显示不受参数设置页面中的日志等级参数控制。
2.from common.Logger import logger: 使用logger.
↪info(XXX)打印日志，该方法的日志显示受参数设置页面中的日志等级参数控制。
"""

IS_UPLOAD_CONFIG = True  #定义变量用于判断是否需要获取并上传点表

def upload_tagconfig(recv, wizard_api):  #定义发布主函数
    global IS_UPLOAD_CONFIG  #声明变量为全局变量
    if IS_UPLOAD_CONFIG:  #判断是否需要获取并上传点表
        wizard_api.get_tag_config(tagconfig)  #调用wizard_api模块中的recall_
```

(续下页)

(接上页)

```
↪data方法，定义该方法的回调函数名称为tagconfig
        IS_UPLOAD_CONFIG = False  #获取并上传点表后不再上传点表

def tagconfig(config, tail, wizard_api): #定义获取点表的回调函数tagconfig
    logger.info(config) #打印点表信息，包含设备、分组、变量和告警点表
    deviceConfiguration_list = [] #定义设备点表列表
    for device in config['devices']: #遍历设备点表
        deviceInfo = {} #定义设备信息字典
        if device['protocol'] == "ModbusTCP": #判断设备通讯协议是否为ModbusTCP
            deviceInfo["Device"] = device['device_name']
            deviceInfo["PLCProtocol"] = device['protocol']
            deviceInfo["IP Address"] = device['ip']
            deviceInfo["Port"] = device['port']
            deviceInfo["SlaveAddress"] = device['slave']
            deviceInfo["Endian"] = device['byte_order']
            deviceConfiguration_list.append(deviceInfo)
↪#依次将deviceInfo添加到deviceConfiguration_list中
        elif device['protocol'] == "ModbusRTU": #判断设备通讯协议是否为ModbusRTU
            deviceInfo["Device"] = device['device_name']
            deviceInfo["PLCProtocol"] = device['protocol']
            deviceInfo["Port"] = device['serial']
            deviceInfo["Baudrate"] = device['baudrate']
            deviceInfo["DataBits"] = device['bytesize']
            deviceInfo["Parity"] = device['parity']
            deviceInfo["StopBits"] = device['stopbits']
            deviceInfo["SlaveAddress"] = device['slave']
            deviceInfo["Endian"] = device['byte_order']
            deviceConfiguration_list.append(deviceInfo)
        elif device['protocol'] == "ISO-on-TCP": #判断设备通讯协议是否为ISO-on-TCP
            deviceInfo["Device"] = device['device_name']
            deviceInfo["PLCProtocol"] = device['protocol']
            deviceInfo["IP Address"] = device['ip']
            deviceInfo["Port"] = device['port']
            deviceInfo["Rack"] = device['rack']
            deviceInfo["Slot"] = device['slot']
            deviceConfiguration_list.append(deviceInfo)
    logger.info(deviceConfiguration_list)
    wizard_api.mqtt_publish("Config/DeviceInfo", json.dumps(deviceConfiguration_
↪list), 1) #调用wizard_api模块中的mqtt_publish方法将deviceConfiguration_
↪list数据通过主题“Config/DeviceInfo”，qos等级1发送至MQTT服务器


    tagConfiguration_list = [] #定义变量点表列表
```

(续下页)

(接上页)

```
    device_group_info_dict = {} #定义设备下的分组信息字典
    group_info_dict = {} #定义分组信息字典
    for groupinfo in config['groups']: #遍历点表中的分组
        group_info_dict[groupinfo["group_name"]] = groupinfo
↪#建立组名与分组信息的对应关系
    for device in config['devices']: #遍历点表中的设备
        group_list= [] #定义设备下的分组列表
        for var in config['vars']: #遍历点表中的变量
            if device['device_name'] == var['device'] and var['group'] not in␣
↪group_list: #判断设备下存在变量，且该变量所属的分组未存在与group_
↪list中，则将该分组添加到group_list中
                group_list.append(var['group'])
        device_group_info_dict[device['device_name']] = group_list
↪#建立设备与设备下的分组列表的对应关系
    for device, group_list in device_group_info_dict.items():
↪#遍历设备下的分组信息字典
        if group_list == []:
↪#如果设备下的分组列表为空，即该设备下未定义变量，则跳过该设备
            continue
        tagConfiguration = {} #定义变量点表字典
        tagConfiguration["Device"] = device #添加设备信息
        tagConfiguration["Collections"] = []
        for group in group_list: #遍历设备下的分组并添加分组信息
            group_info = {}
            group_info["CollectionName"] = group
            group_info["SampleRate"] = group_info_dict[group]["polling_interval"]
            group_info["PublishInterval"] = group_info_dict[group]["upload_
↪interval"]
            group_info["TagData"] = []
            tagConfiguration["Collections"].append(group_info)
            for var in config['vars']: #遍历点表中的变量
                if var['device'] != device or var['group'] != group:
↪#如果该变量不属于当前设备和分组，则跳过该变量
                    continue
                index_number = tagConfiguration["Collections"].index(group_info)
↪#获取该分组在tagConfiguration["Collections"]中的索引
                data_info = {} #定义变量信息字典
                data_info["Tag"] = var["var_name"]
                data_info["Address"] = var["address"]
                data_info["ValueType"] = var["data_type"]
                data_info["AccessLevel"] = var["read_write"]
                data_info["Mode"] = var["mode"]
                data_info["Unit"] = var["unit"]
```

(续下页)

(接上页)

```
                data_info["Description"] = var["desc"]
                tagConfiguration["Collections"][index_number]["TagData"].
↪append(data_info) #将变量信息添加至指定分组的TagData中
        tagConfiguration_list.append(tagConfiguration)
↪#依次将设备点表添加至tagConfiguration_list中
    logger.info(tagConfiguration_list) #打印变量点表
    for tagConfiguration in tagConfiguration_list:
↪#遍历每个设备下的变量点表并依次上传至MQTT服务器
        wizard_api.mqtt_publish("Config/TagConfiguration", json.
↪dumps(tagConfiguration), 1) #调用wizard_api模块中的mqtt_
↪publish方法将tagConfiguration_list中的数据依次通过主题“Config/
↪TagConfiguration”，qos等级1发送至MQTT服务器


    alarmConfiguration_list = [] #定义告警点表
    for alarm in config['warning']: #遍历点表中的告警
        alarmInfo = {} #定义告警信息字典
        alarmInfo['Warn_name'] = alarm['warn_name']
        alarmInfo['Group'] = alarm['group']
        alarmInfo['Alarm_content'] = alarm['alarm_content']
        alarmInfo['Condition1'] = alarm['condition1']
        alarmInfo['Operand1'] = alarm['operand1']
        alarmInfo['Combine_method'] = alarm['combine_method']
        alarmInfo['Condition2'] = alarm['condition2']
        alarmInfo['Operand2'] = alarm['operand2']
        alarmInfo['Device'] = alarm['device']
        alarmInfo['Var_name'] = alarm['var_name']
        alarmInfo['Address'] = alarm['address']
        alarmConfiguration_list.append(alarmInfo)
↪#依次将告警信息添加至alarmConfiguration_list中
    logger.info(alarmConfiguration_list) #打印告警点表
    wizard_api.mqtt_publish("Config/AlarmInfo", json.dumps(alarmConfiguration_
↪list), 1) #调用wizard_api模块中的mqtt_publish方法将alarmConfiguration_
↪list数据通过主题“Config/AlarmInfo”，qos等级1发送至MQTT服务器
```

- 发布示例 6：使用 `get_global_parameter` 获取参数设置中的自定义参数

  本示例实现了获取“参数设置”中的自定义参数 `device_id`，并通过通配符 `${device_id}` 的配置方式配置 MQTT 主题。发布和代码配置示例如下：

全局参数

参数	参数值	操作
catch_recording	100000	
device_id	1	
log_level	INFO	
warning_recording	2000	


```
import logging
```

(续下页)

(接上页)

```
"""
在网关中打印日志通常有两种办法。
1.import logging: 使用logging.
↪info(XXX)打印日志，该方法的日志显示不受参数设置页面中的日志等级参数控制。
2.from common.Logger import logger: 使用logger.
↪info(XXX)打印日志，该方法的日志显示受参数设置页面中的日志等级参数控制。
"""

def vars_upload_test(data_collect, wizard_api): #定义发布主函数
    global_parameter = wizard_api.get_global_parameter() #定义自定义参数变量
    logging.info(global_parameter) #打印自定义参数变量
    value_list = [] #定义数据列表
    for device, val_dict in data_collect['values'].items():
↪#遍历values字典，该字典中包含设备名称和设备下的变量数据
        value_dict = { #自定义数据字典
                       "Device": device,
                       "DeviceID": global_parameter["device_id"],
↪#获取自定义参数中定义的设备ID
                       "timestamp": data_collect["timestamp"],
                       "Data": {}
                       }
        for id, val in val_dict.items(): #遍历变量数据，为Data字典赋值
            value_dict["Data"][id] = val["raw_data"]
        value_list.append(value_dict) #依次将value_dict添加到value_list中
    logging.info(value_list) #在App日志中打印value_list，数据格式为[{'Device':
↪'S7-1200', 'DeviceID': '1', 'timestamp': 1589538347.5604711, 'Data': {'Test1':␣
↪False, 'Test2': 12}}]
    return value_list #将value_
↪list发送给App，App将自行按照采集时间顺序上传至MQTT服务器。如果发送失败则缓存数据等待连接恢复后
```

配置订阅消息

自定义订阅消息中包含以下项：

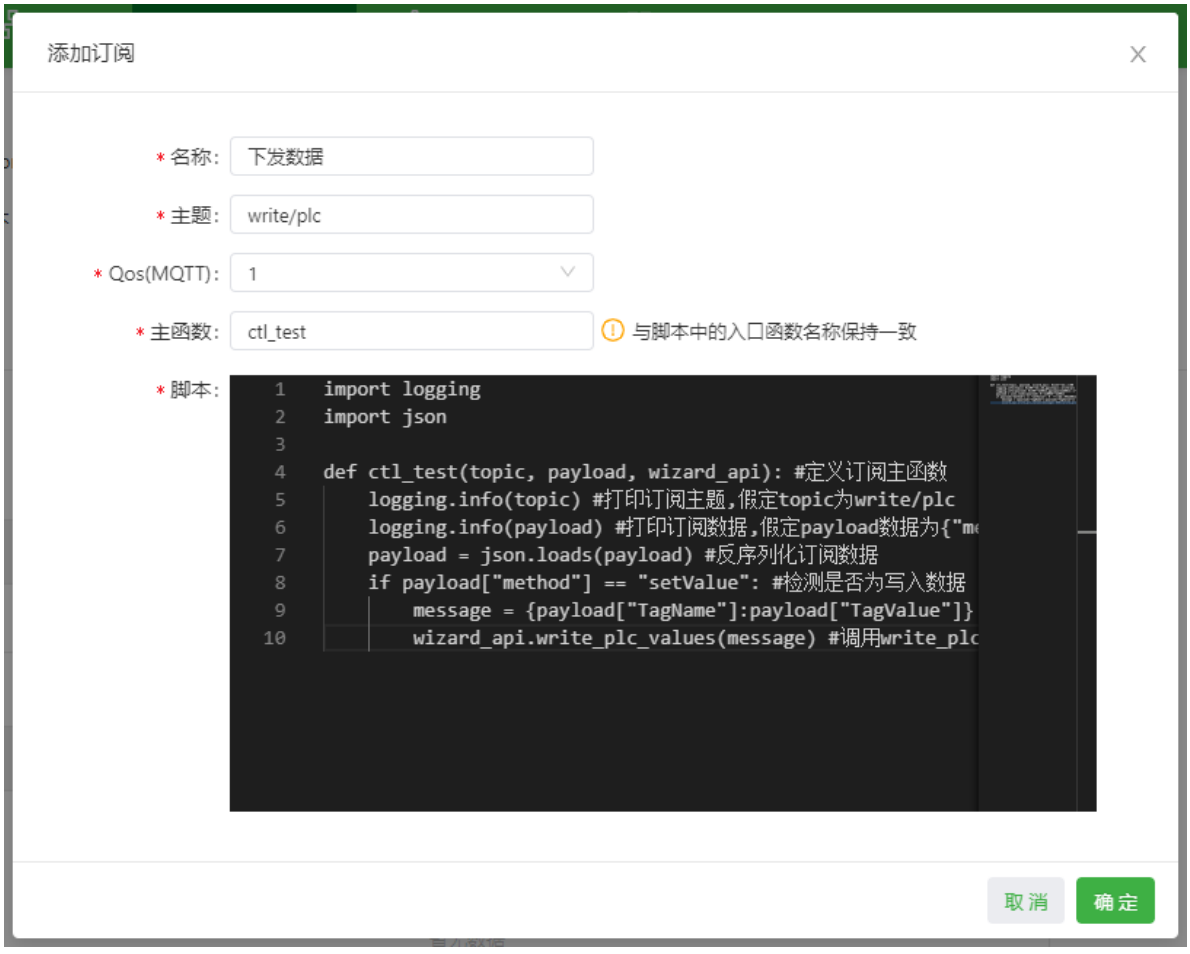
- 名称：自定义订阅名称
- 主题：订阅主题，与 MQTT 服务器发布的数据主题保持一致
- Qos(MQTT)：订阅 Qos，建议与 MQTT 服务器的 Qos 保持一致
- 主函数：主函数名称，即入口函数名称，与脚本中的入口函数名称保持一致
- 脚本：使用 Python 代码自定义组包和处理逻辑，订阅中的主函数参数包括：

- 参数 1：该参数为接收到的主题，数据类型为 string
- 参数 2：该参数为接收到的数据，数据类型为 string
- 参数 3：该参数为 Device Supervisor 提供的 api 接口，参数说明见 *Device Supervisor 的 api 接口说明（wizard_api）*

以下是四个常见的自定义订阅方法示例：

- 订阅示例 1：下发变量名称和变量值写入 PLC 数据且不返回写入结果

  本示例实现了从 MQTT 服务器下发指定命令修改变量数值，发布和代码配置示例如下：




```
import logging
import json

def ctl_test(topic, payload, wizard_api): #定义订阅主函数
    logging.info(topic) #打印订阅主题,假定topic为write/plc
    logging.info(payload) #打印订阅数据,假定payload数据为{"method":"setValue",
↪"TagName":"SP1", "TagValue":12.3}
```

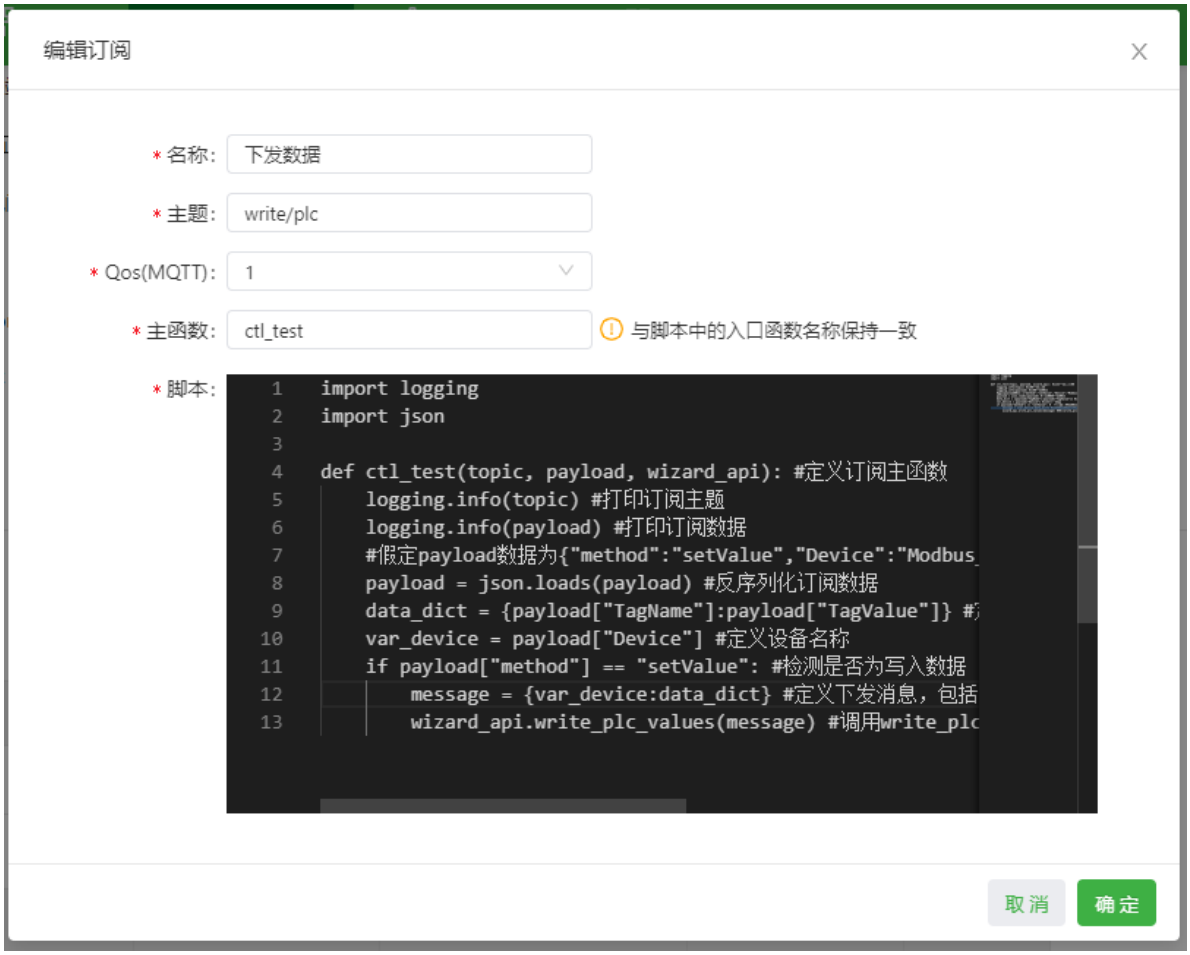
(续下页)

(接上页)

```
    payload = json.loads(payload) #反序列化订阅数据
    if payload["method"] == "setValue": #检测是否为写入数据
        message = {payload["TagName"]:payload["TagValue"]}
↪#定义下发消息，包括下发的变量名称和变量值
        wizard_api.write_plc_values(message) #调用wizard_api模块中的write_plc_
↪values方法，将message字典中的数据下发至指定变量
```

- 订阅示例 2：下发设备名称，变量名称和变量值写入 PLC 数据且不返回写入结果

  本示例实现了从 MQTT 服务器下发指定命令修改变量数值，发布和代码配置示例如下：




```
import logging
import json

def ctl_test(topic, payload, wizard_api): #定义订阅主函数
    logging.info(topic) #打印订阅主题
    logging.info(payload) #打印订阅数据
```

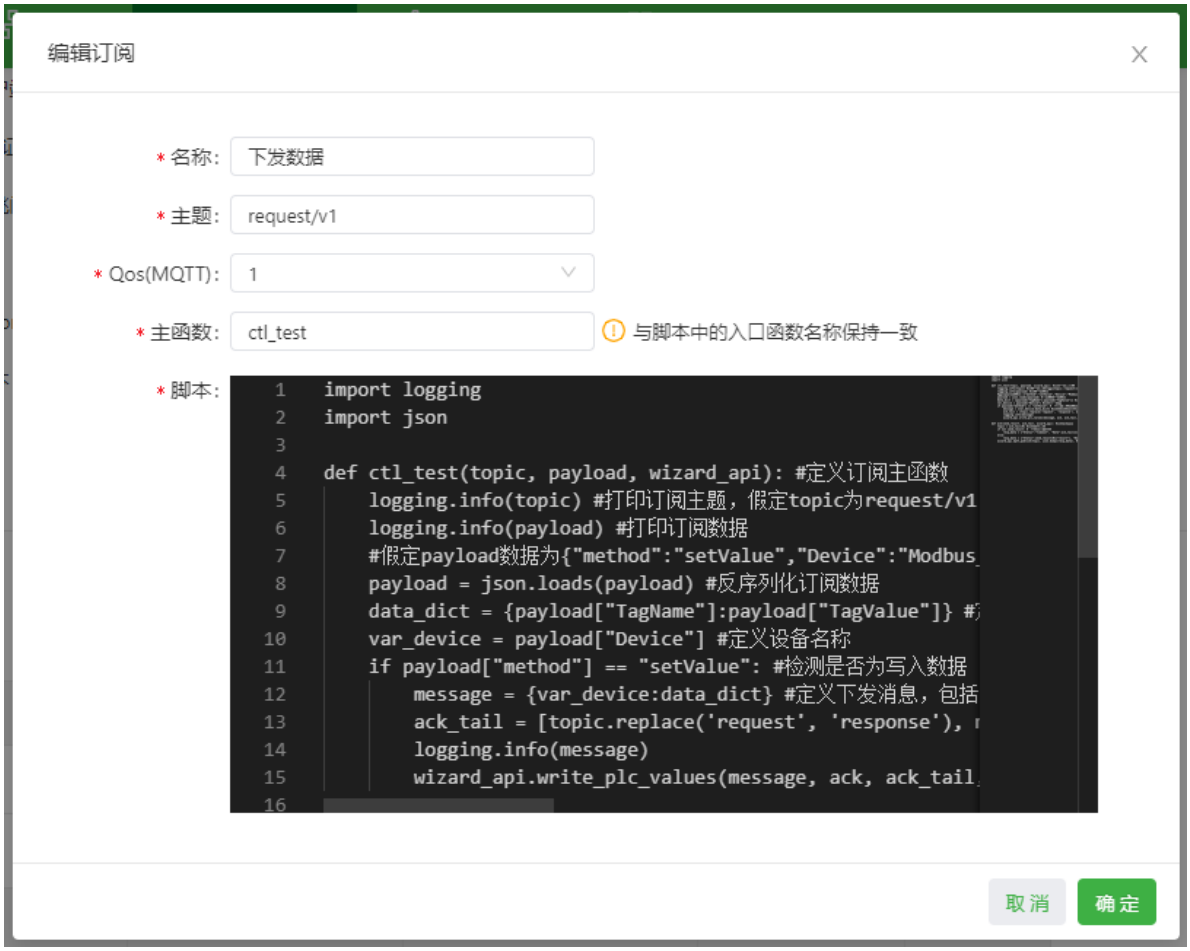
(续下页)

(接上页)

```
    #假定payload数据为{"method":"setValue","Device":"Modbus_test", "TagName":"SP1
↪", "TagValue":12.3}
    payload = json.loads(payload) #反序列化订阅数据
    data_dict = {payload["TagName"]:payload["TagValue"]}
↪#定义下发的数据字典，包含下发的变量名称和变量值
    var_device = payload["Device"] #定义设备名称
    if payload["method"] == "setValue": #检测是否为写入数据
        message = {var_device:data_dict}
↪#定义下发消息，包括设备名称和下发的数据字典
        wizard_api.write_plc_values(message) #调用wizard_api模块中的write_plc_
↪values方法，将message字典中的数据下发至指定变量
```

- 订阅示例 3：写入变量数据并返回写入结果

  本示例实现了从 MQTT 服务器下发指定命令修改变量数值并返回修改结果，发布和代码配置示例如下：




```
import logging
```

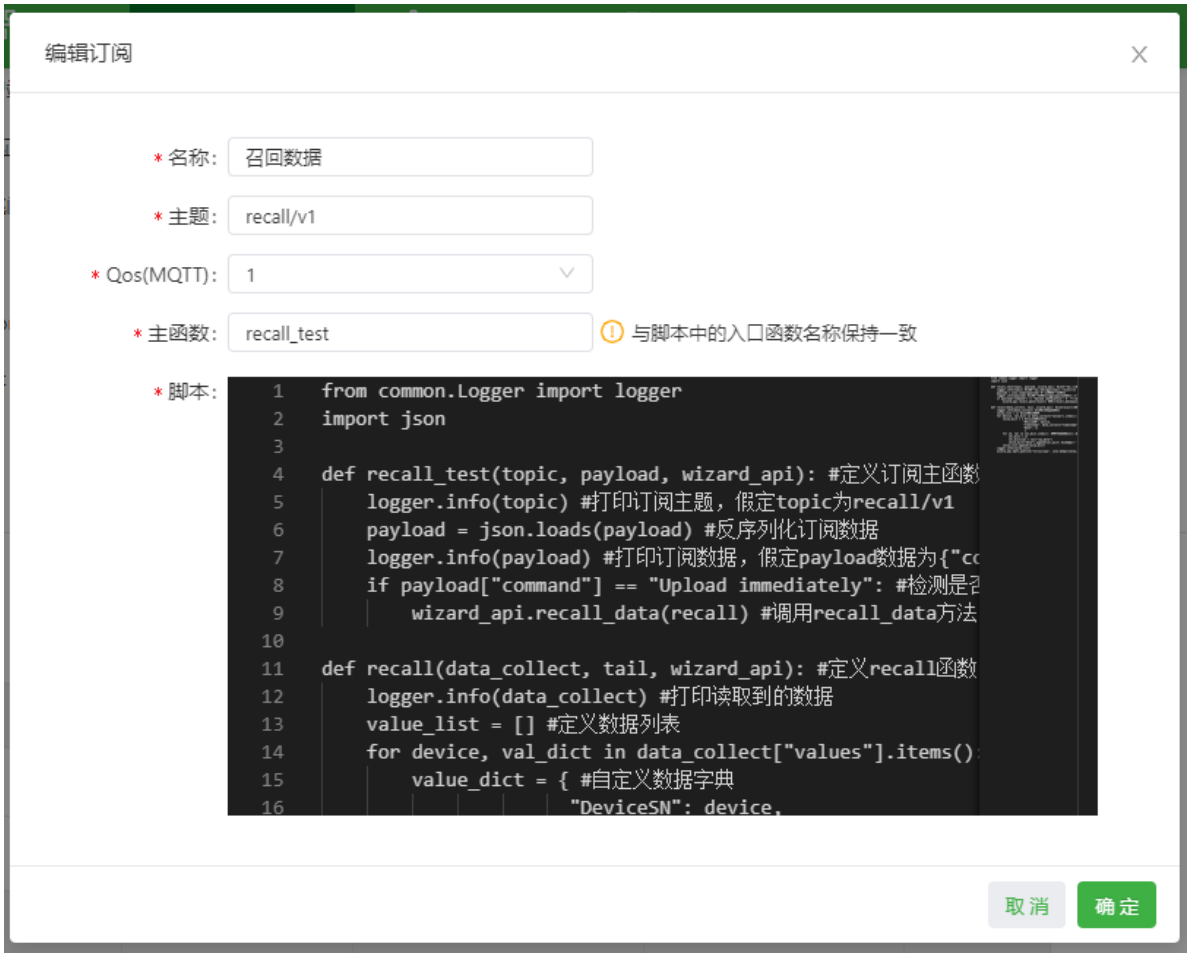
(续下页)

(接上页)

```
import json

def ctl_test(topic, payload, wizard_api): #定义订阅主函数
    logging.info(topic) #打印订阅主题，假定topic为request/v1
    logging.info(payload) #打印订阅数据
    #假定payload数据为{"method":"setValue","Device":"Modbus_test", "TagName":"SP1
↪", "TagValue":12.3}
    payload = json.loads(payload) #反序列化订阅数据
    data_dict = {payload["TagName"]:payload["TagValue"]}
↪#定义下发的数据字典，包含下发的变量名称和变量值
    var_device = payload["Device"] #定义设备名称
    if payload["method"] == "setValue": #检测是否为写入数据
        message = {var_device:data_dict}
↪#定义下发消息，包括设备名称和下发的数据字典
        ack_tail = [topic.replace('request', 'response'), message]
↪#定义确认数据，包括响应的主题和消息
        logging.info(message)
        wizard_api.write_plc_values(message, ack, ack_tail, timeout = 0.5)
↪#调用wizard_api模块中的write_plc_
↪values方法，将message字典中的数据下发至指定变量；定义该方法的回调函数名称为ack并将ack_
↪tail传递给回调函数ack

def ack(send_result, ack_tail, wizard_api): #定义回调函数ack
    topic = ack_tail[0] #定义响应主题：response/v1
    if isinstance(send_result,tuple): #检测send_
↪result的数据类型是否为元组，为元组则说明下发超时
        resp_data = {"Status":"timeout", "Data":ack_tail[1]}
↪#定义下发超时的响应数据
    else:
        resp_data = {"Status":send_result[0]["result"], "Data":ack_tail[1]}
↪#定义下发未超时的响应数据
    wizard_api.mqtt_publish(topic, json.dumps(resp_data), 0) #调用wizard_
↪api模块中的mqtt_publish将响应数据发送给MQTT服务器
```

- 订阅示例 4：立即召回数据

  本示例实现了从 MQTT 服务器下发指定命令时，立即读取所有变量数值并发送至 MQTT 服务器，发布和代码配置示例如下：




```
from common.Logger import logger
import json

def recall_test(topic, payload, wizard_api): #定义订阅主函数
    logger.info(topic) #打印订阅主题，假定topic为recall/v1
    payload = json.loads(payload) #反序列化订阅数据
    logger.info(payload) #打印订阅数据，假定payload数据为{"command":"Upload␣
↪immediately"}
    if payload["command"] == "Upload immediately": #检测是否需要数据召回
        wizard_api.recall_data(recall) #调用wizard_api模块中的recall_
↪data方法，定义该方法的回调函数名称为recall

def recall(data_collect, tail, wizard_api): #定义回调函数recall
    logger.info(data_collect) #打印读取到的数据
    value_list = [] #定义数据列表
    for device, val_dict in data_collect["values"].items():
↪#遍历values字典，该字典中包含设备名称和设备下的变量数据
        value_dict = { #自定义数据字典
                    "DeviceSN": device,
```

(续下页)

(接上页)

```
                    "timestamp": data_collect["timestamp"],
                    "Data": []
                     }
        for id, val in val_dict.items(): #遍历变量数据，为Data列表赋值
            var_dict = {} #定义变量字典
            var_dict[id] = val["raw_data"]
            value_dict["Data"].append(var_dict) #依次将变量字典添加到value_dict中
        value_list.append(value_dict) #依次将数据字典添加到value_list中
    logger.info(value_list) #打印value_list
    wizard_api.mqtt_publish("v1/xxx/yyy", json.dumps(value_list), 1) #调用wizard_
↪api模块中的mqtt_publish方法将value_list数据通过主题“v1/xxx/
↪yyy”，qos等级1发送至MQTT服务器
```

Device Supervisor 的 api 接口说明（wizard_api）

Device Supervisor 提供的 api 接口，包含以下方法：

- `mqtt_publish`：MQTT 发布消息方法，用于将指定数据通过相应的主题发送到 MQTT 服务器并返回发送结果：发送成功（True），发送失败（False），使用示例请参考*发布示例 3*。该方法包含以下参数：
  - 参数 `1`：MQTT 主题，数据类型为 `string`。通过至该主题发送数据到 MQTT 服务器
  - 参数 `2`：需要发送的数据
  - 参数 `3`：qos 等级（包括 0/1/2 三种等级）
- `save_data`：存储数据至数据库方法，被存储的数据将在 MQTT 连接正常时按先存先传的方式上传至 MQTT 服务器，使用示例请参考*发布示例 3* 和*发布示例 4*。该方法包含以下参数：
  - 参数 `1`：需要存储的数据。如果调用 `save_data` 时只提供了参数 `1`，则参数 `1` 的数据类型为 `dict`，在储存的数据中需具备以 `topic`、`qos` 和 `payload` 为键的键值对，当 MQTT 连接正常后将以存储数据中的 `topic` 和 `qos` 将 `payload` 发送至 MQTT 服务器。
  - 参数 `2`（可选参数 `group`）：需要存储的数据的分组名称，数据类型为 `string`。如果调用 `save_data` 时提供参数 `2`，则将以该分组在云服务中关联的主题和 qos 上传参数 `1` 至 MQTT 服务器。
- `write_plc_values`：下发数据至指定变量方法并支持返回修改结果，使用示例请参考*订阅示例 1*，*订阅示例 2* 和*订阅示例 3*。该方法包含以下参数：
  - 参数 `1`：下发数据，该数据可以有两种形式：
    - 形式 `1`：传入一个以变量名称和变量值为键值对的 `dict`。使用此方法修改变量数值时，需要确保该变量的名称在“设备列表”中唯一，数据格式示例如下：

```
{
    "SP1": 12.3,  #变量名称和变量值的键值对
    "SP2": 12.4
}
```

* 形式 2：传入一个包含设备名称、变量名称和变量值的 dict，数据格式示例如下：

```
{
    "S7-1200":  #设备名称
    {
        "SP1": 12.3,  #变量名称和变量值的键值对
        "SP2": 12.4
    }
}
```

  - 参数 2（可选参数 callback）：返回修改结果的回调函数名称，回调函数说明见 *write_plc_values 回调函数说明*
  - 参数 3（可选参数 tail）：已有参数 2 时，可将需要传递给参数 2 的数据赋值给参数 3
  - 参数 4（可选参数 timeout）：写入超时时间，数据类型为整数或浮点数。默认为 60 秒

- get_tag_config：获取点表配置方法，点表配置包括 PLC、变量、分组和告警配置，使用示例请参考*发布示例 5*。该方法包含以下参数：
  - 参数 1：获取点表配置的回调函数的名称，回调函数说明见 *get_tag_config 回调函数说明*
  - 参数 2（可选参数 tail）：可将需要传递给参数 1 的数据赋值给参数 2
  - 参数 3（可选参数 timeout）：获取点表超时时间，数据类型为整数。默认为 60 秒
- recall_data：立即读取所有变量数值方法，使用示例请参考*订阅示例 4*。该方法包含以下参数：
  - 参数 1：立即读取所有变量数值的回调函数的名称，回调函数说明见 *recall_data 回调函数*
  - 参数 2（可选参数 tail）：可将需要传递给参数 1 的数据赋值给参数 2
  - 参数 3（可选参数 timeout）：立即读取所有变量的超时时间，数据类型为整数。默认为 60 秒
- get_global_parameter：获取全局参数方法，使用示例请参考*发布示例 6*。该方法会返回一个参数设置的字典，数据格式如下：

```
{
    'gateway_sn': 'GT902XXXXXXXXX', #系统参数，网关序列号
    'log_level': 'INFO', #系统参数，日志等级
    'catch_recording': 100000, #系统参数，最大可缓存的变量数据的MQTT消息数量
    'warning_recording': 2000, #系统参数，最大可缓存的告警数据的MQTT消息数量
    'device_id': '1' #自定义参数
}
```

Device Supervisor api 回调函数说明

- write_plc_values 回调函数说明 write_plc_values 回调函数包含以下参数，使用示例请参考 *订阅示例 3*:
  - 参数 1: write_plc_values 方法的写入结果
    - 写入超时时返回值为

```
("error", -110, "timeout")
```

    - 写入成功时返回值格式为:

```
[
{
    'value': 12, #写入值
    'device': 'S7-1200', #写入设备
    'var_name': 'Test2', #写入变量的名称
    'result': 'OK', #写入结果。写入成功: OK, 写入失败: Failed
    'error': '' #写入错误，当写入成功时该项为空
}]
```

    - 写入失败时返回值格式为:

```
[
{
    'value': 12.3,
    'device': 'Modbus_test',
    'var_name': 'SP1',
    'result': 'Failed',
    'error': "Device 'Modbus_test' not found."
}]
```

  - 参数 2: write_plc_values 方法中配置的参数 3，如果未在 write_plc_values 中配置参数 3，则该参数为 None
  - 参数 3: 该参数为 Device Supervisor 提供的 api 接口，参数说明见 *Device Supervisor 的 api 接口说明（wizard_api）*

- get_tag_config 回调函数说明 get_tag_config 回调函数包含以下参数，使用示例请参考发布 *示例 5*:

- 参数 1: get_tag_config 方法返回的点表配置。获取点表超时时返回值为 ("error", -110, "timeout")，正常返回点表配置时数据格式如下（以 ISO-on-TCP 协议的 Rack/slot 模式为例）:

```
{
    'devices': [ #设备点表
    {
        'protocol': 'ISO-on-TCP', #设备协议
        'device_name': 'S7-1200', #设备名称
        'ip': '10.5.16.73', #IP地址
        'port': 102, #端口号
        'rack': 0, #机架号
        'slot': 0, #槽号
        'id': '6358f50294dc11ea8d890018050ff046' #设备ID
    }],
    'groups': [ #分组点表
    {
        'group_name': 'warning', #分组名称
        'polling_interval': 10, #采集间隔
        'upload_interval': '', #上报间隔间隔
        'group_type': 'alarm', #分组类型。collect: 采集组，alarm: 告警组
        'id': '84c371902eb911eabab11a4f32d1ee44' #分组ID
    }],
    'warning': [ #告警点表
    {
        'warn_name': 'Warn1', #告警名称
        'group': 'warning', #告警所属分组
        'quotes': 1, #告警变量来源。0: 直接使用地址，1: 引用地址
        'device': 'S7-1200', #告警变量所属设备
        'alarm_content': '速度超过30! ', #告警描述
        'condition1': 'Gt', #告警条件1。Eq: 等于,Neq: 不等于,Gt: 大于,
↪Gne: 大于等于,Lne: 小于等于,Lt: 小于
        'operand1': '30', #告警阈值1
        'combine_method': 'And', #告警条件连接方式。None: 空，And: &&，Or: ||
        'condition2': 'Lt', #告警条件2
        'operand2': '50', #告警阈值2
        'var_name': 'Test2', #告警变量名称
        'var_id': '96c93c3094dd11eabd400018050ff046', #告警变量ID
        'size': 1, #告警变量的数据类型为STRING时的字符串长度
        'float_repr': 2, #告警变量的数据类型为FLOAT时变量小数点后的数据长度
        'id': '9165ed78943e11ea8a000018050ff046', #告警ID
        'address': 'DB6.2', #告警变量地址
        'protocol': 'ISO-on-TCP', #告警设备通讯协议
        'data_type': 'WORD', #告警变量数据类型
        'register_type': 'DB', #告警变量寄存器类型
```

(续下页)

(接上页)

```
        'register_addr': 2, #告警变量寄存器地址
        'read_write': 'read/write', #告警读写权限。read: 只读、write: 只写、
↪'read/write: 可读可写
        'mode': 'realtime', #告警变量采集模式
        'unit': '', #告警变量单位
        'desc': '', #告警变量描述
        'dbnumber': 6, #告警变量寄存器类型为DB时变量的DB号
        'register_bit': '' #告警变量数据类型为BOOL或BIT时变量的位偏移
    }],
    'vars': [ #变量点表
    {
        'device': 'S7-1200', #变量所属设备的名称
        'protocol': 'ISO-on-TCP', #变量所属设备的通讯协议
        'data_type': 'BOOL', #变量数据类型
        'register_type': 'I', #变量寄存器类型
        'var_name': 'Test1', #变量名称
        'register_addr': 0, #变量寄存器地址
        'read_write': 'read/write', #变量读写权限。read: 只读、write: 只写、
↪'read/write: 可读可写
        'mode': 'realtime', #变量采集模式
        'unit': '', #变量单位
        'desc': '', #变量描述
        'group': 'default', #变量所属分组
        'register_bit': 0, #变量数据类型为BOOL或BIT时变量的位偏移
        'size': 1, #变量的数据类型为STRING时的字符串长度
        'float_repr': 2, #变量的数据类型为FLOAT时变量小数点后的数据长度
        'dbnumber': 0, #变量寄存器类型为DB时变量的DB号
        'id': 'a1d9439a94dc11eaa2830018050ff046', #变量ID
        'address': 'I0.0' #变量地址
    },
    {
        'device': 'S7-1200',
        'protocol': 'ISO-on-TCP',
        'data_type': 'WORD',
        'register_type': 'DB',
        'var_name': 'Test2',
        'register_addr': 2,
        'read_write': 'read/write',
        'mode': 'realtime',
        'unit': '',
        'desc': '',
        'group': '2222',
        'dbnumber': 6,
```

(续下页)

(接上页)

```
        'size': 1,
        'float_repr': 2,
        'register_bit': '',
        'id': '96c93c3094dd11eabd400018050ff046',
        'address': 'DB6.2'
    }]
}
```

- 参数 2：get_tag_config 方法中配置的参数 3，如果未在 get_tag_config 中配置参数 3，则该参数为 None
- 参数 3：该参数为 Device Supervisor 提供的 api 接口，参数说明见 *Device Supervisor 的 api 接口说明（wizard_api）*

- recall_data 回调函数说明 recall_data 回调函数包含以下参数订阅示例 *4*：
  - 参数 1：recall_data 方法返回的变量数据。获取变量数据超时时返回值为 ("error", -110, "timeout")，正常返回变量数据时数据格式如下：

```
{
    'timestamp': 1589507333.2521989,  #数据产生时间戳
    'values':  #数据字典，包括PLC名称，变量名称和变量值
    {
        'S7-1200':  #PLC名称
        {
            'Test1':  #变量名称
            {
                'raw_data': False,  #变量值
                'status': 1  #采集状态，非1即采集异常
            },
            'Test2':
            {
                'raw_data': 33,
                'status': 1
            }
        }
    }
}
```

  - 参数 2：recall_data 方法中配置的参数 3，如果未在 recall_data 中配置参数 3，则该参数为 None
  - 参数 3：该参数为 Device Supervisor 提供的 api 接口，参数说明见 *Device Supervisor 的 api 接口说明（wizard_api）*

参数设置

你可以访问“边缘计算 > 设备监控 > 参数设置”页面配置 Device Supervisor 的通用设置。

- 默认参数

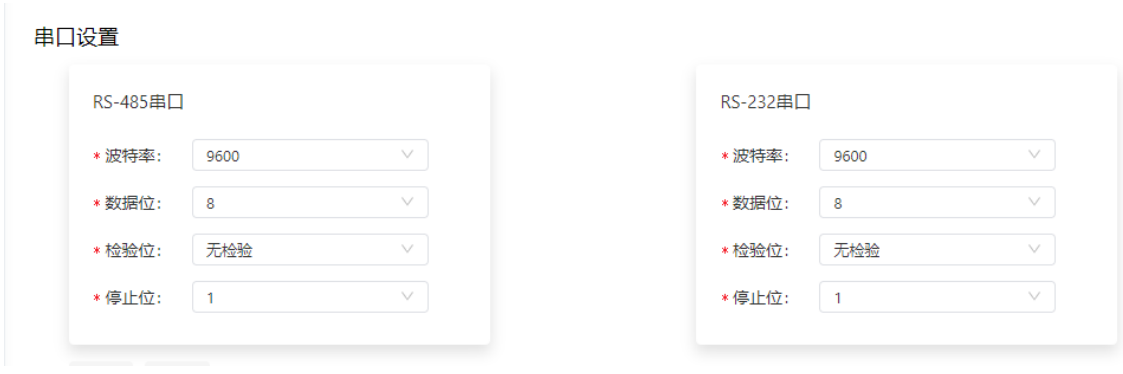
  你可以在默认参数中设置日志等级、历史告警和历史数据条数。

- 自定义参数

  你可以在自定义参数中自行添加常用参数作为云服务中的通配符使用。使用方法为 `${参数名称}`，如下图所示：

- 串口设置

  你可以在串口设置中配置 RS485 和 RS232 串口的通讯参数，如下图所示：



网关的其他配置

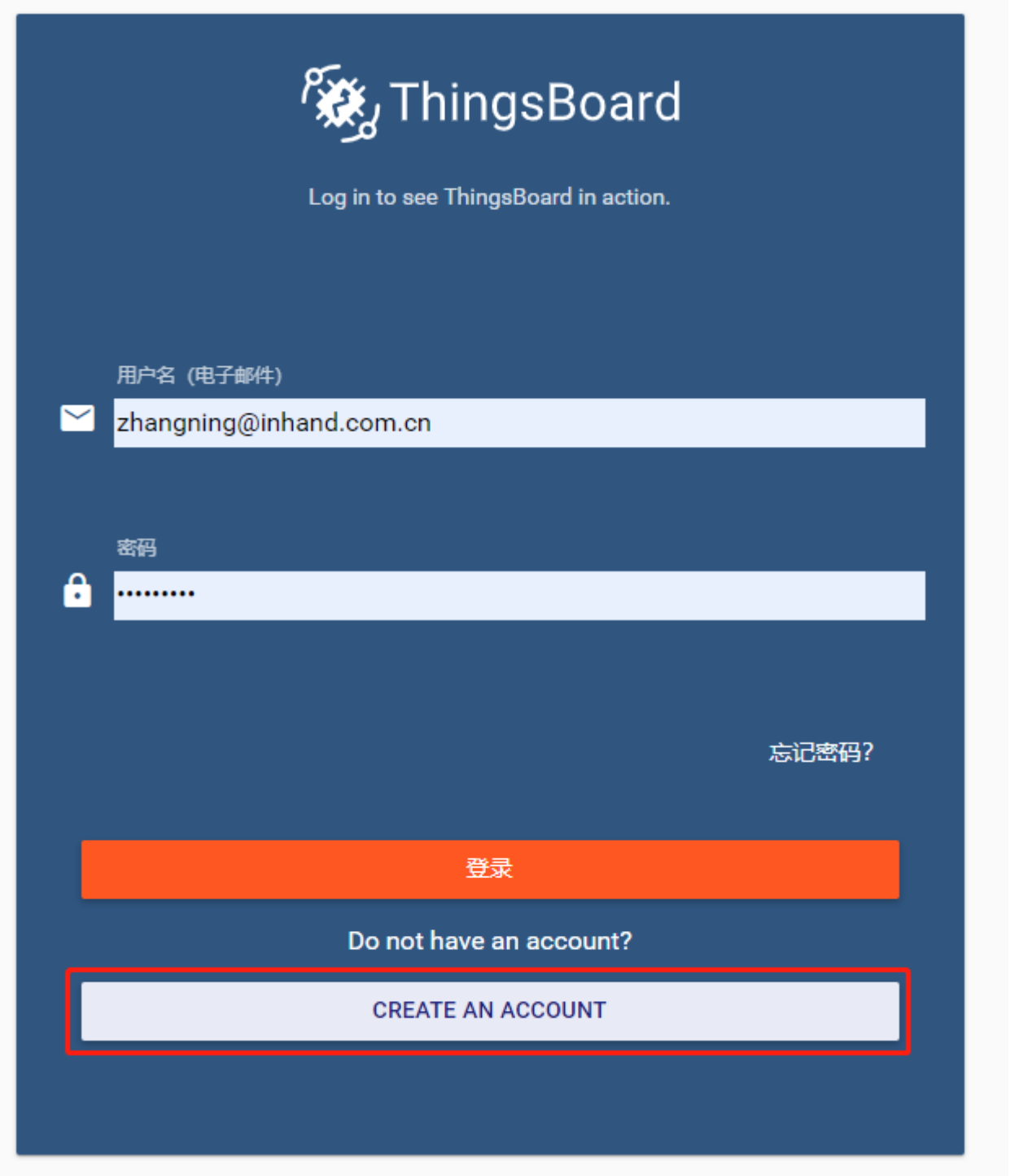
关于网关的其他常用操作请查看IG501 快速使用手册、IG502 快速使用手册或IG902 快速使用手册。

Thingsboard 参考流程

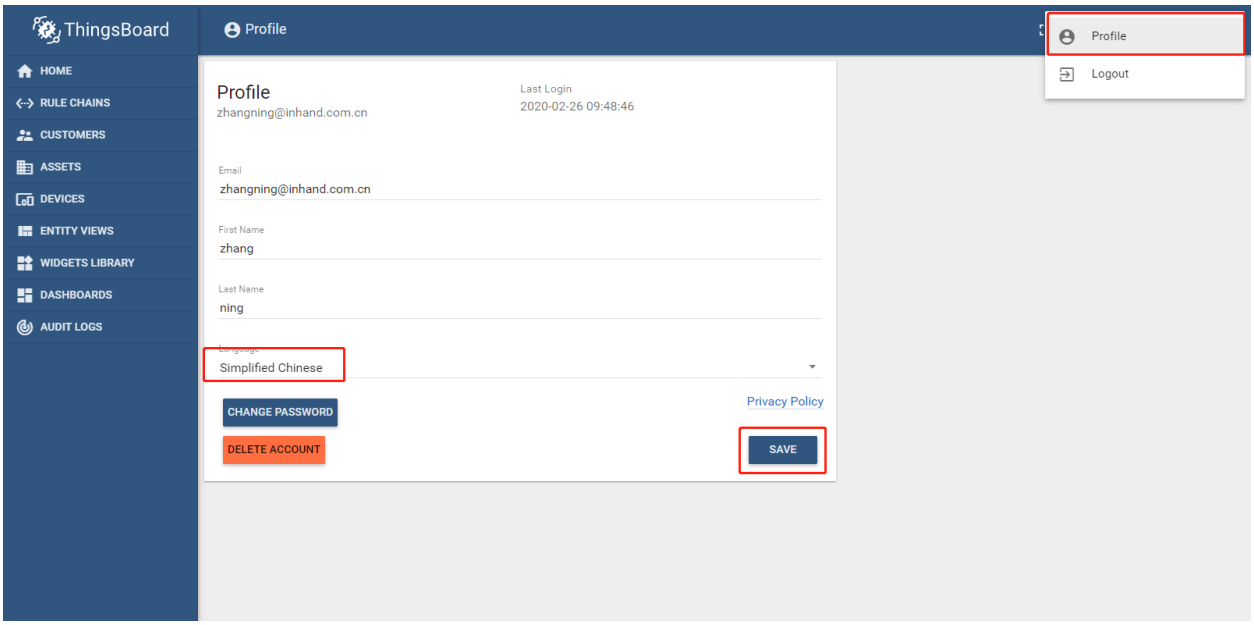
- 添加设备和资产
- 传输 *PLC* 数据到 *Thingsboard* 设备
- 配置可视化仪表板

添加设备和资产

访问 `https://demo.thingsboard.io/login`，输入登录账号和密码。如果未注册过账号则需要先注册账号后再登录（注册账号时需要能够访问海外网络，否则可能无法正常注册）。

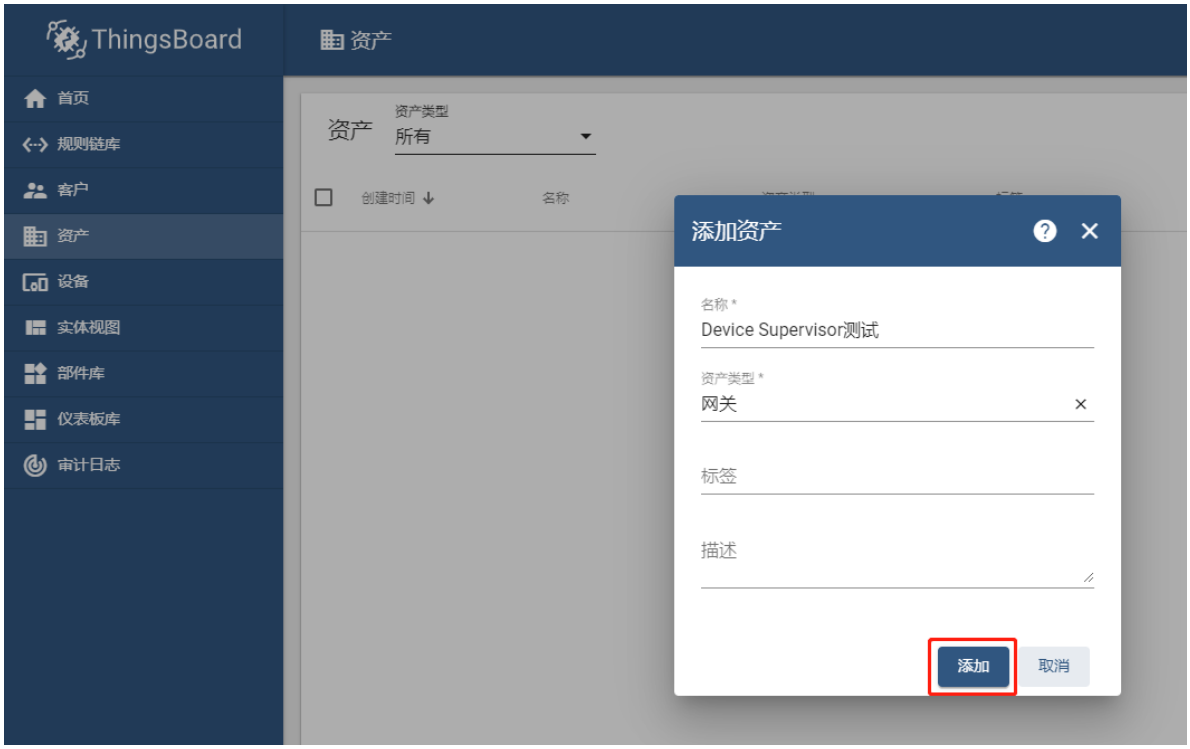


登录后，进入属性页面修改语言为简体中文。

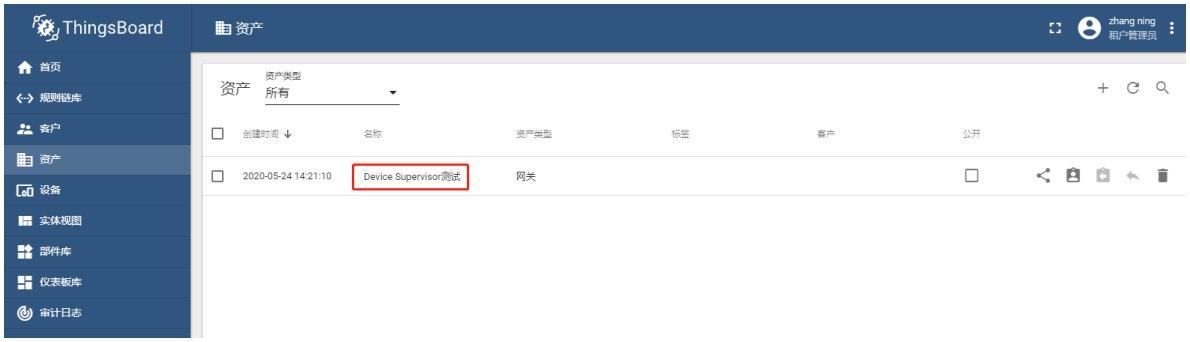

ThingsBoard
Profile
HOME
RULE CHAINS
CUSTOMERS
ASSETS
DEVICES
ENTITY VIEWS
WIDGETS LIBRARY
DASHBOARDS
AUDIT LOGS
Profile
Logout
Profile
zhangning@inhand.com.cn
Last Login
2020-02-26 09:48:46
Email
zhangning@inhand.com.cn
First Name
zhang
Last Name
ning
Language
Simplified Chinese
CHANGE PASSWORD
DELETE ACCOUNT
Privacy Policy
SAVE

- 添加一个资产

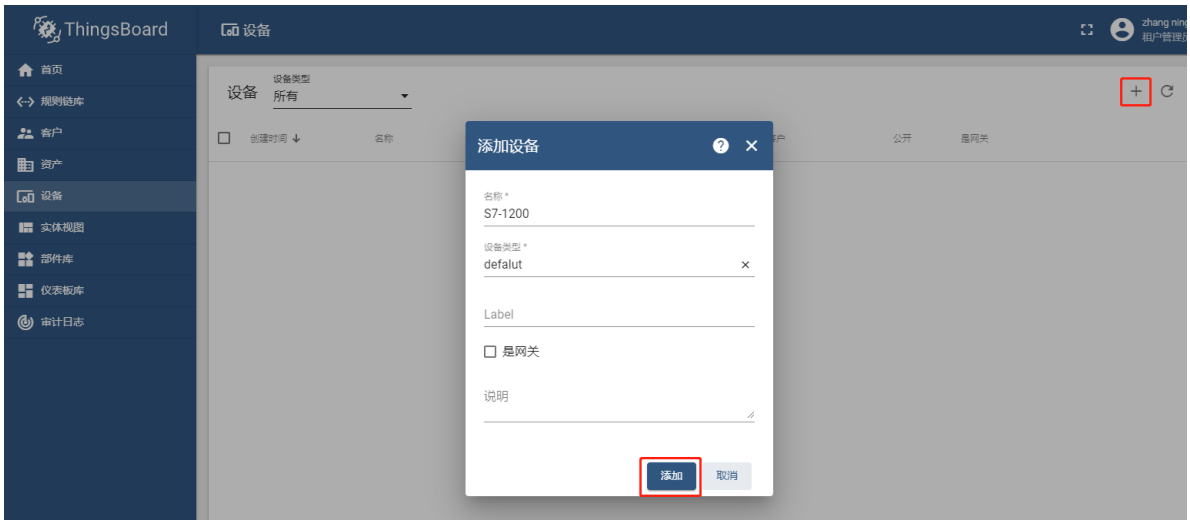

ThingsBoard
资产
zhang ning
租户管理员
首页
规则链库
客户
资产
设备
实体视图
部件库
仪表板库
审计日志
资产
资产类型
所有
创建时间
名称
资产类型
标签
客户
公开
未找到资产

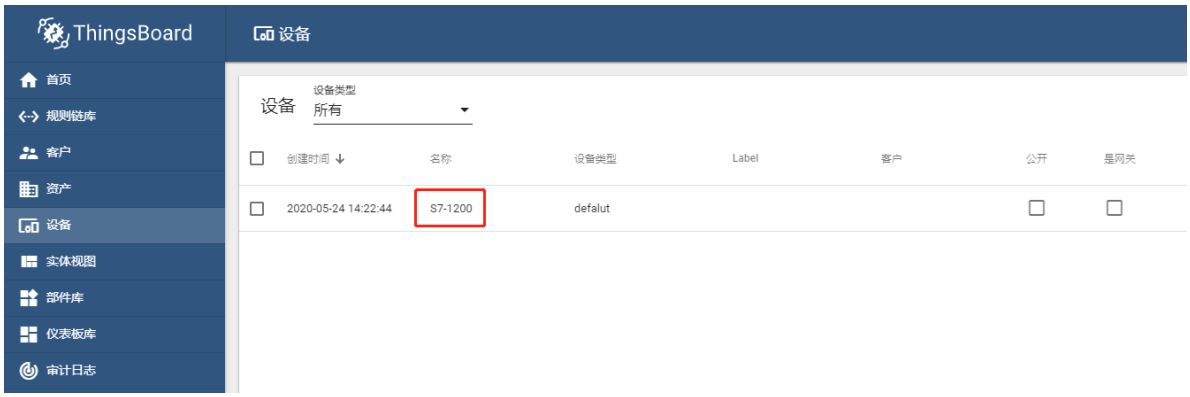


添加成功后如下图所示：

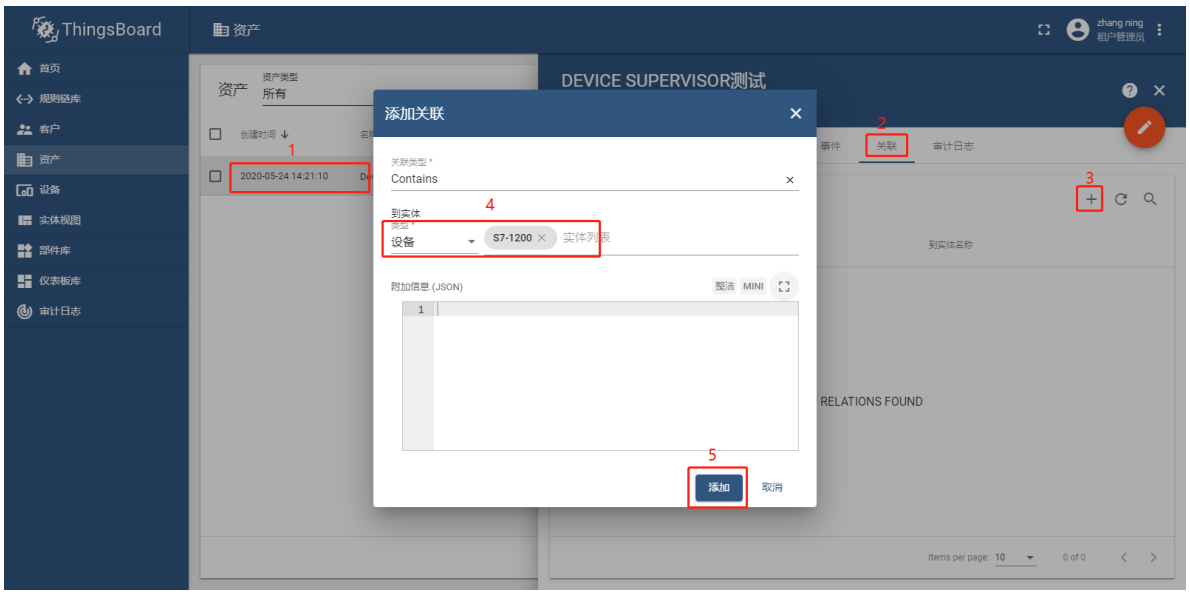


- 添加一个设备





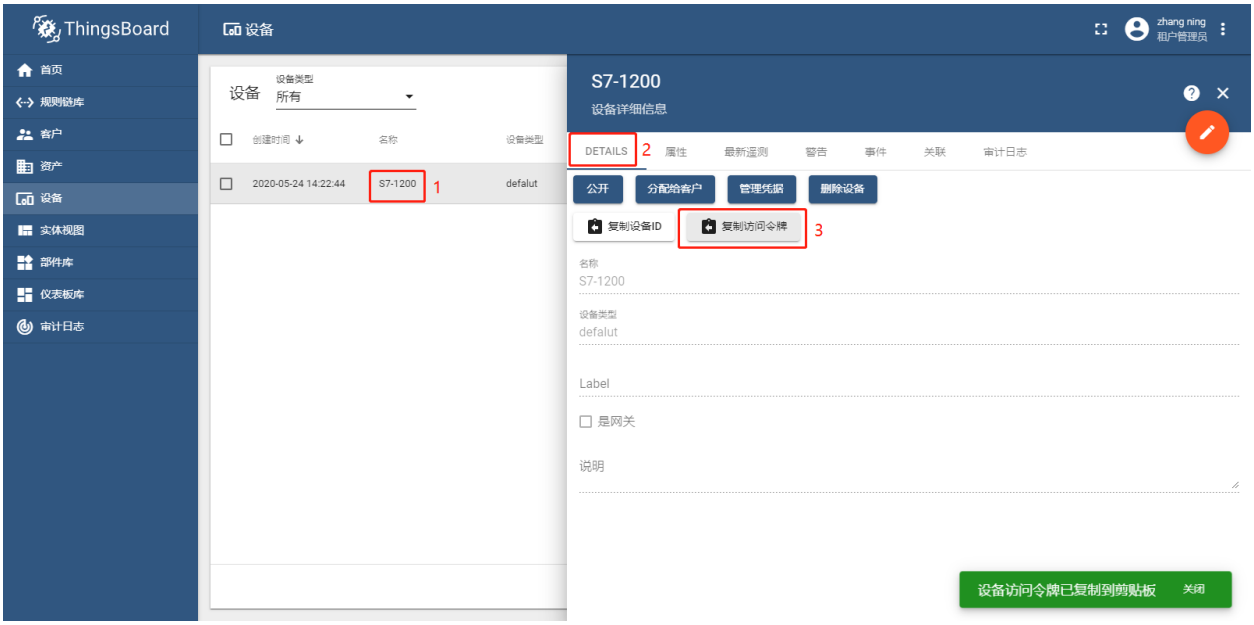
建立资产与设备的关联。



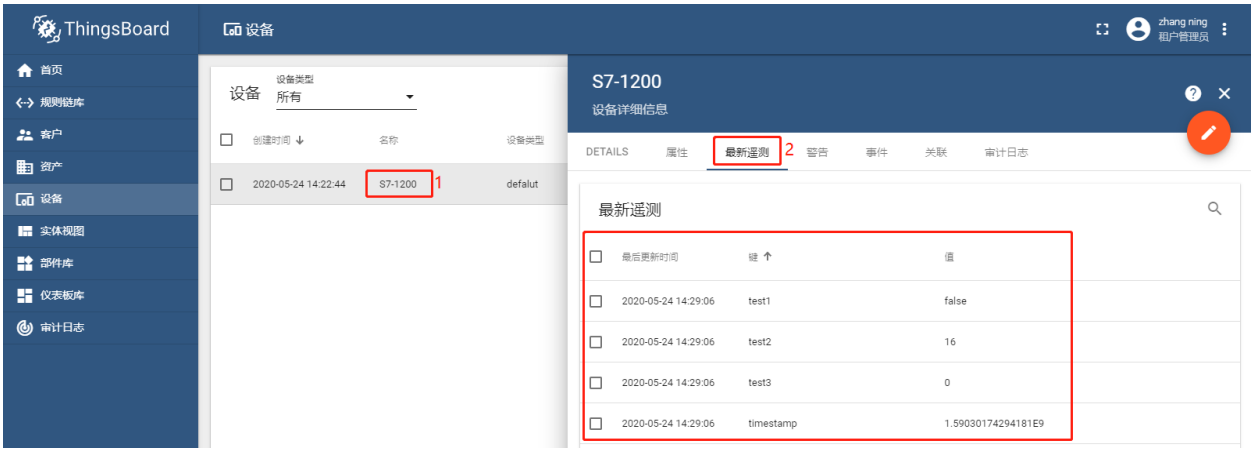
添加完成后如下图所示：

传输 PLC 数据到 Thingsboard 设备

资产和设备配置完成后，复制已添加设备的访问令牌并粘贴至网关的云服务页面的用户名参数中以将数据传输至 Thingsboard 中的 `S7-1200` 设备。



随后可在设备的最新遥测中查看已上传的数据。

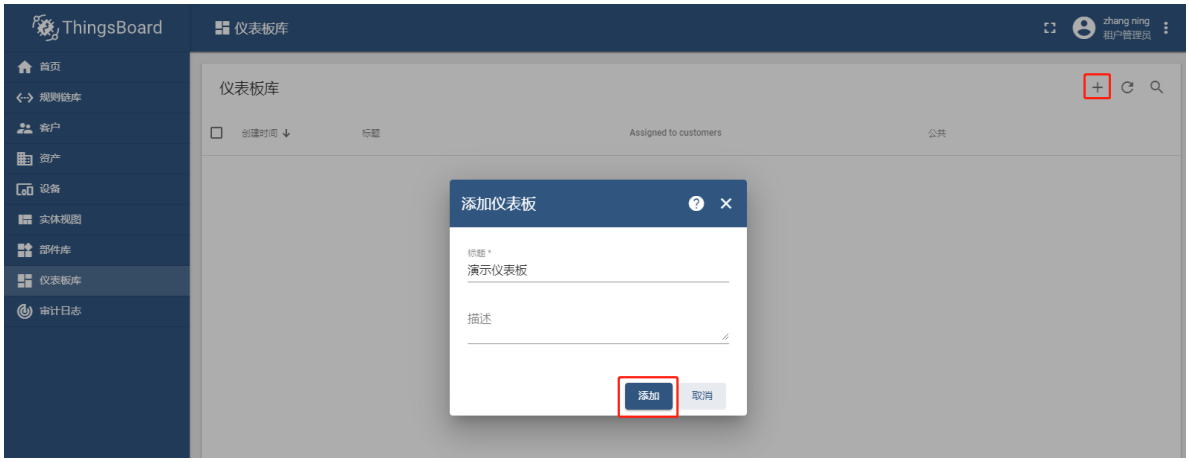


配置可视化仪表板

- 添加仪表板
- 添加趋势图
- 添加开关

- 添加仪表板

  点击“添加仪表板”，选择“创建新的仪表板”。在“添加仪表板”配置并添加一个仪表板。



  添加完成后单击仪表板名称并选择“打开仪表板”。

点击“进入编辑模式”。

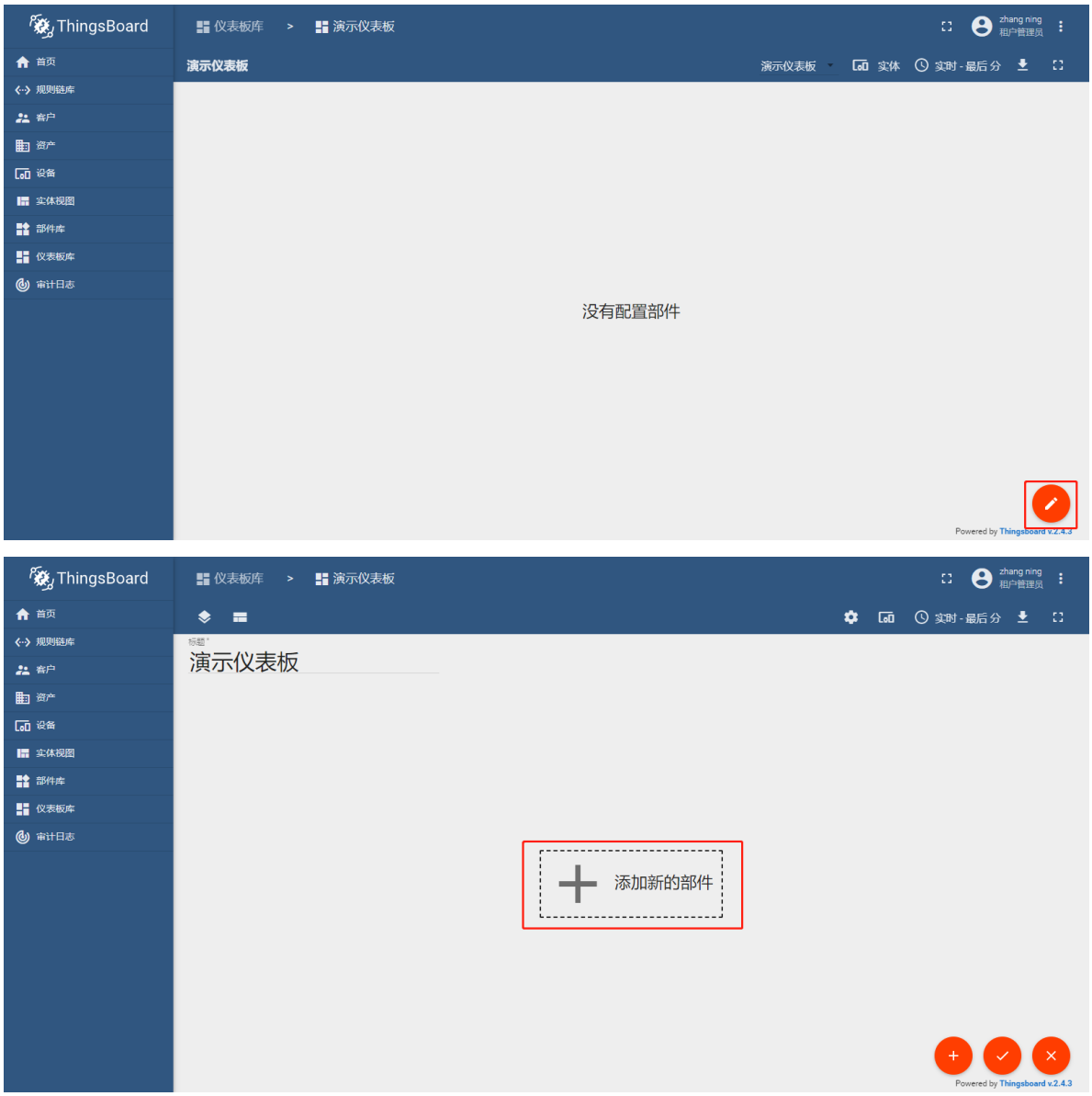
点击“实体别名”为仪表板添加实体别名。

在“添加别名”页面参考下图进行配置：

配置完成后保存配置即可。

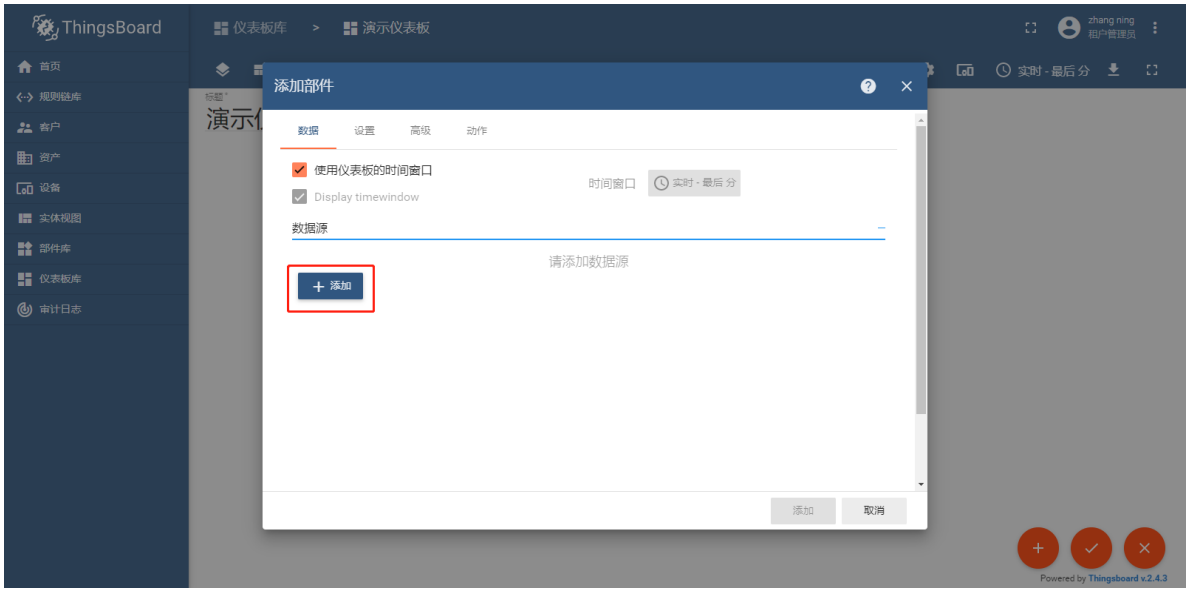
- 添加趋势图

  在仪表板中点击“进入编辑模式”并点击“添加新的部件”。



在部件中选择“Charts”并点击“Timeseries-Flot”。

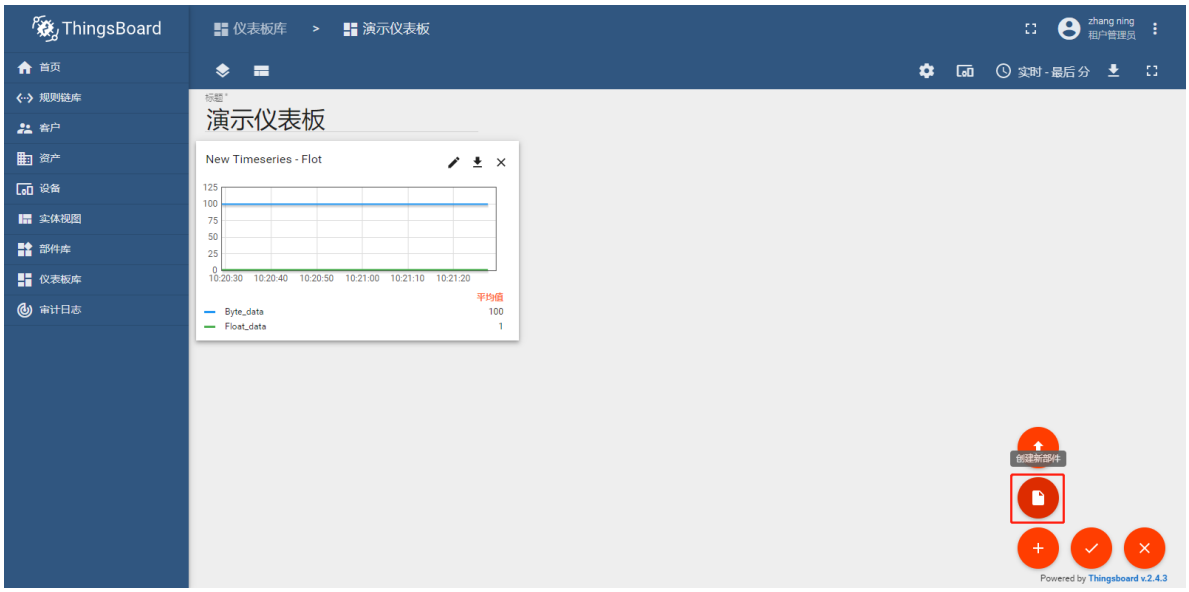
在图表的“数据”页面为趋势图添加展示数据。



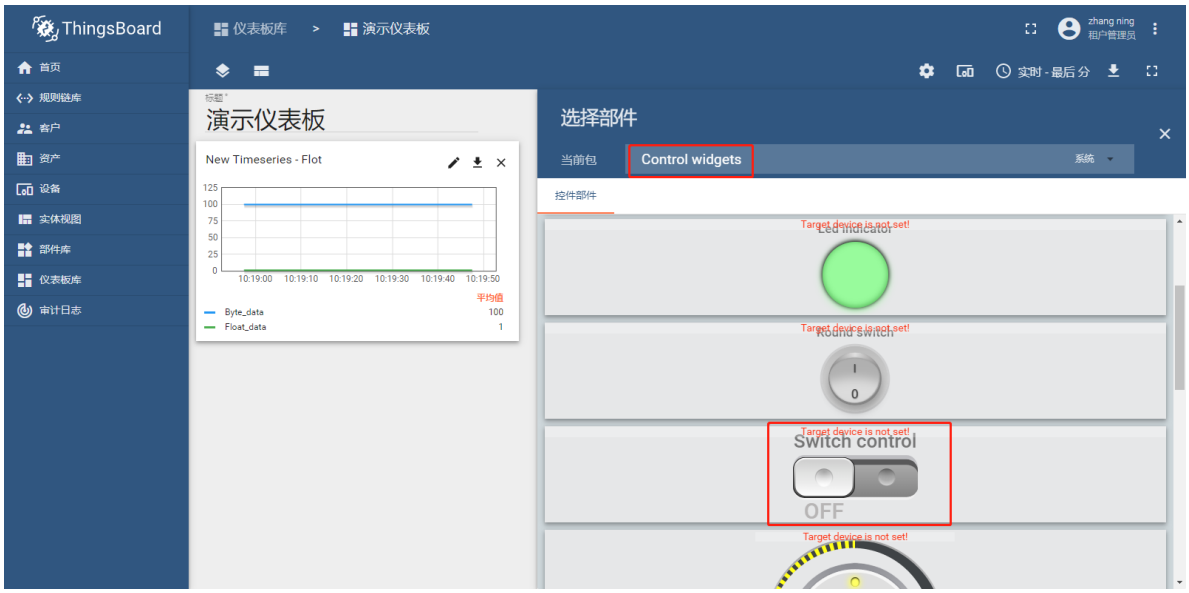
添加完成后如下图所示：

- 添加开关

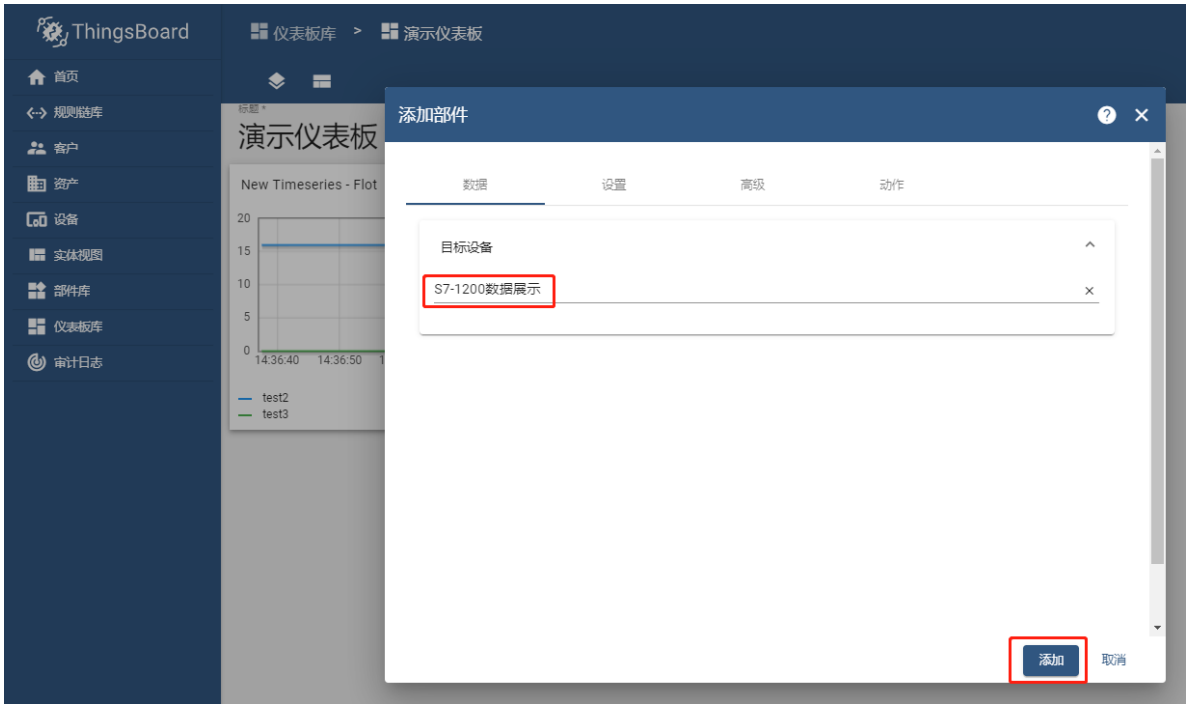
  点击“添加新的部件”，选择“创建新部件”以添加一个控制开关。



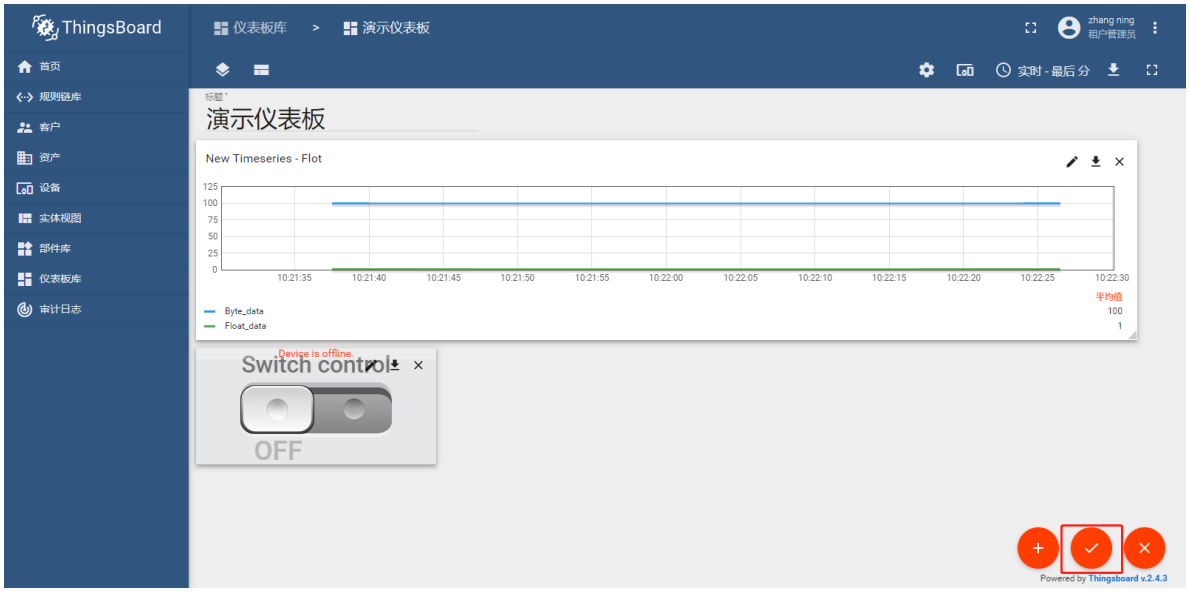
在部件中选择“Control widgets”并点击“Switch control”。



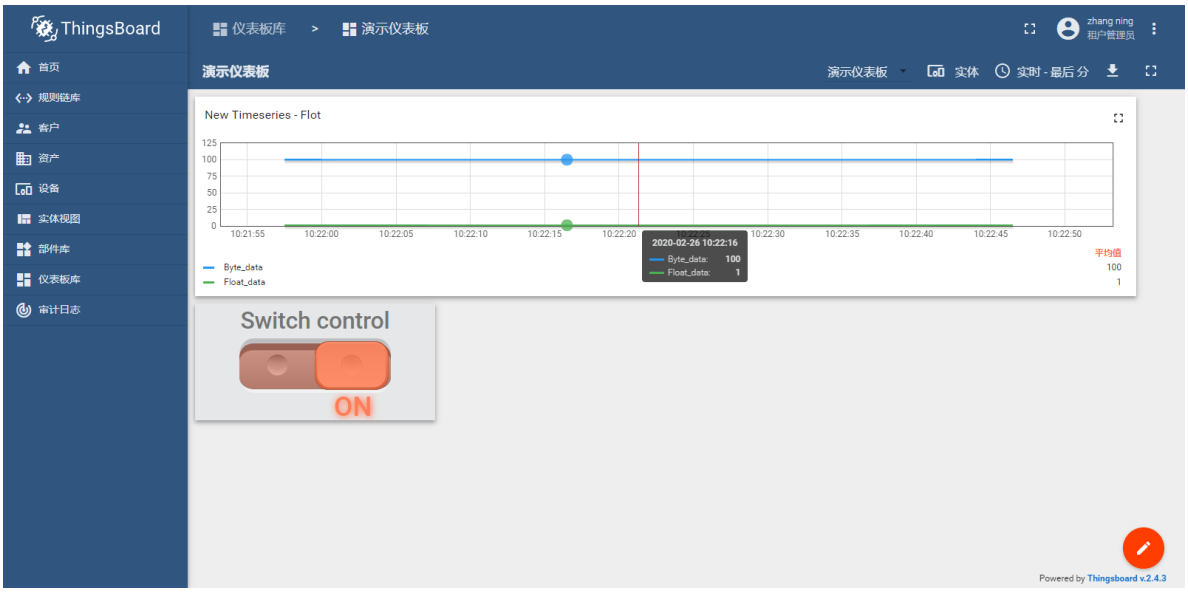
随后选择目标设备。



配置完成后调整部件的大小和布局并保存。



随后可以通过开关下发控制命令以及通过趋势图查看数据趋势。



1.2.6 FAQ

- 查看云服务脚本是否正确
- 查看 *App* 的云服务输出是否正确

查看云服务脚本是否正确

打开 Device Supervisor App 日志。脚本编写完成并点击“确定”后，通过日志中的 `Build module: <主函数名称>, type: <publish/subscribe>` 信息查看脚本是否构建成功。

脚本构建成功如下图所示：

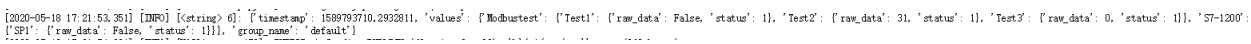
```
[2020-05-18 17:18:27,460] [INFO] [MqttProxy.py 175]: Build module: main, type: publish
[2020-05-18 17:18:27,469] [INFO] [MqttProxy.py 191]: Build OK.
```

脚本构建失败如下图所示：

```
[2020-05-18 17:22:04,612] [ERROR] [MqttProxy.py 195]: Build module error. 'No main entry method found. main1'
```

查看 App 的云服务输出是否正确

您可以使用使用 `logger` 和 `logging` 输出重要日志。下图是在运行脚本中的第 6 行使用了 `logging.info` 方法，在日志中可以通过搜索`<string> 6` 查看输出结果是否符合预期。



1.3 阿里云 IoT 使用说明

越来越多的企业选择将 IoT 设备迁移上云。为此，阿里云提供企业从自建 MQTT 集群迁移到阿里云物联网平台（以下简称阿里云 IoT）的解决方案，为设备提供安全可靠的连接通信能力，向下连接海量设备，支撑设备数据采集上云；向上提供云端 API，服务端通过调用云端 API 将指令下发至设备端，实现远程控制。

为便于用户实现设备与阿里云 IoT 的对接，边缘计算网关 InGateway902（以下简称 IG902）提供 Device Supervisor App（以下简称 Device Supervisor）对接阿里云 IoT。本文档将以 IG902 为例为你说明如何实现 Device Supervisor 与阿里云 IoT 的业务数据上报和配置数据下发。

- 先决条件
- *1.* 环境准备
  - *1.1* 阿里云配置
    - *1.1.1* 创建产品
    - *1.1.2* 创建设备
  - *1.2* 边缘计算网关配置
    - *1.2.1* 基础配置
    - *1.2.2* 配置数据采集
- *2.* 发布和订阅消息
  - *2.1* 连接阿里云 *IoT*
    - *2.1.1* 一机一密
    - *2.1.2* 一型一密
  - *2.2* 发布消息到阿里云 *IoT*
    - *2.2.1* 自定义 *Topic*
    - *2.2.2* 属性上报
    - *2.2.3* 事件上报
  - *2.3* 订阅阿里云 *IoT* 的消息
    - *2.3.1* 自定义 *Topic*
    - *2.3.2* 服务调用

1.3.1 先决条件

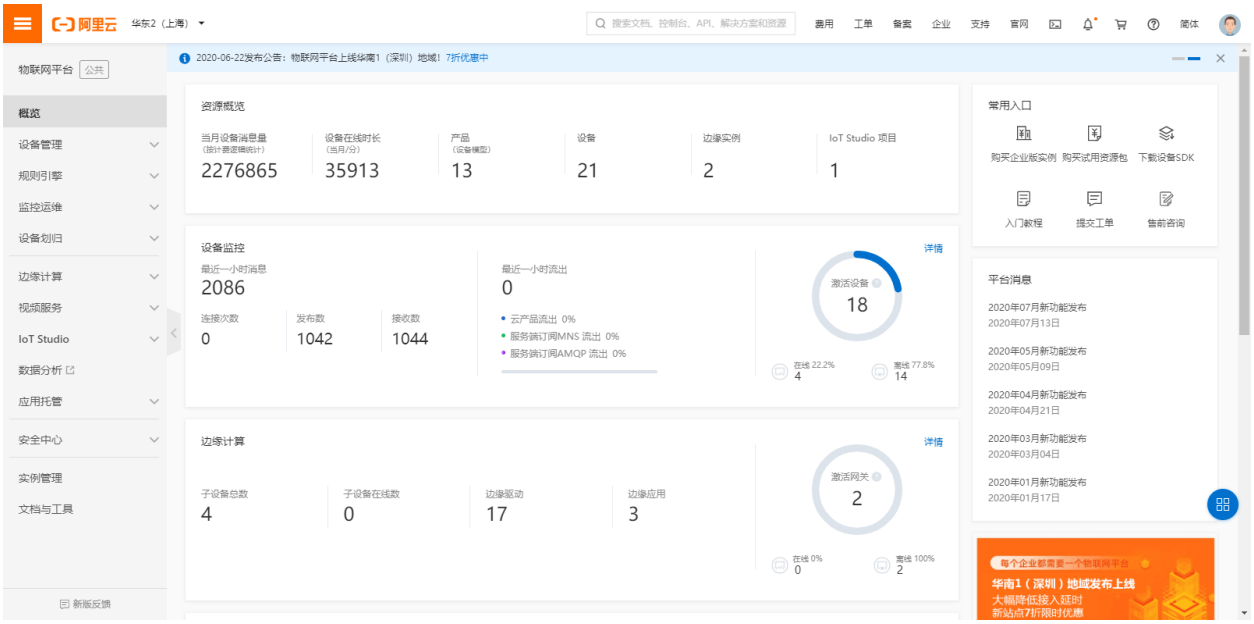
- 阿里云账号
- 边缘计算网关 IG501/IG902
  - 固件版本
    * IG902：`IG9-V2.0.0.r12754` 及以上
    * IG501：`IG5-V2.0.0.r12884` 及以上
  - SDK 版本
    * IG902：`py3sdk-V1.4.0_Edge-IG9` 及以上
    * IG501：`py3sdk-V1.4.0_Edge-IG5` 及以上
  - App 版本：`device_supervisor-V1.2.4` 及以上

1.3.2 1. 环境准备

1.1 阿里云配置

如果你已经在阿里云的物联网云平台中配置了相应的产品和设备，可以直接查看下一节 *1.2* 边缘计算网关配置。否则请按照如下流程配置物联网云平台中的产品和设备。访问阿里云官网 https://www.aliyun.com 并登录，登录后选择“物联网平台”。

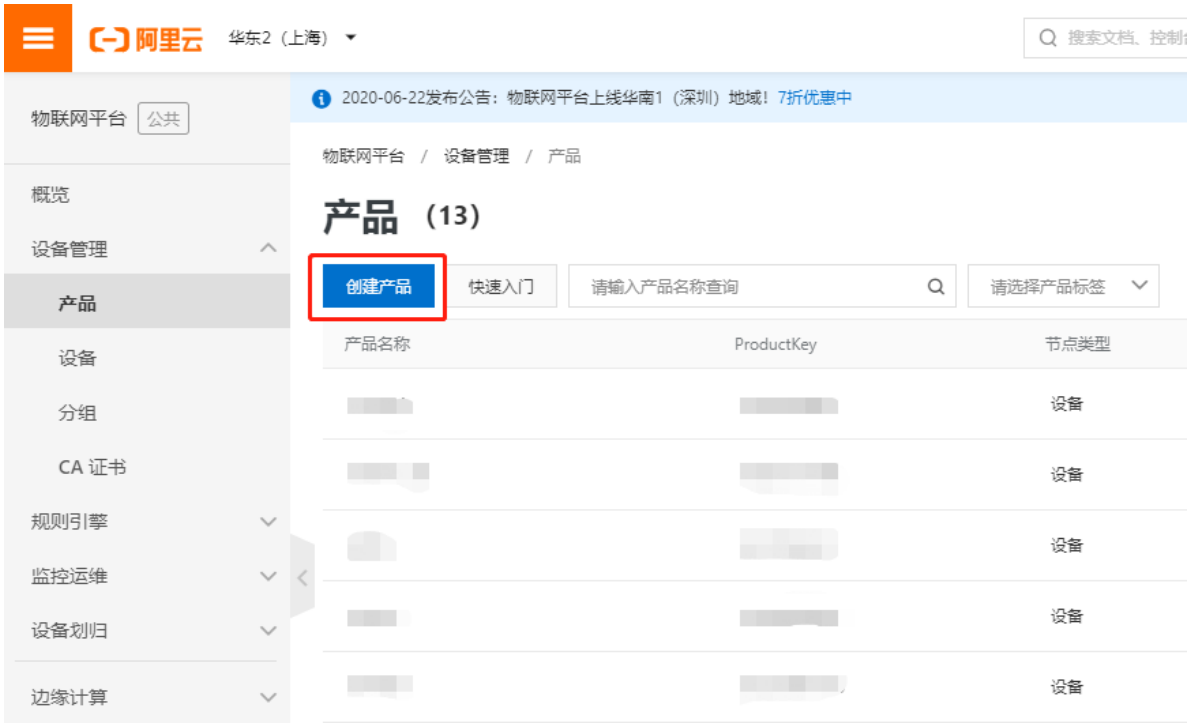
“物联网平台”页面如下所示：



1.1.1 创建产品

- 步骤 1：创建产品

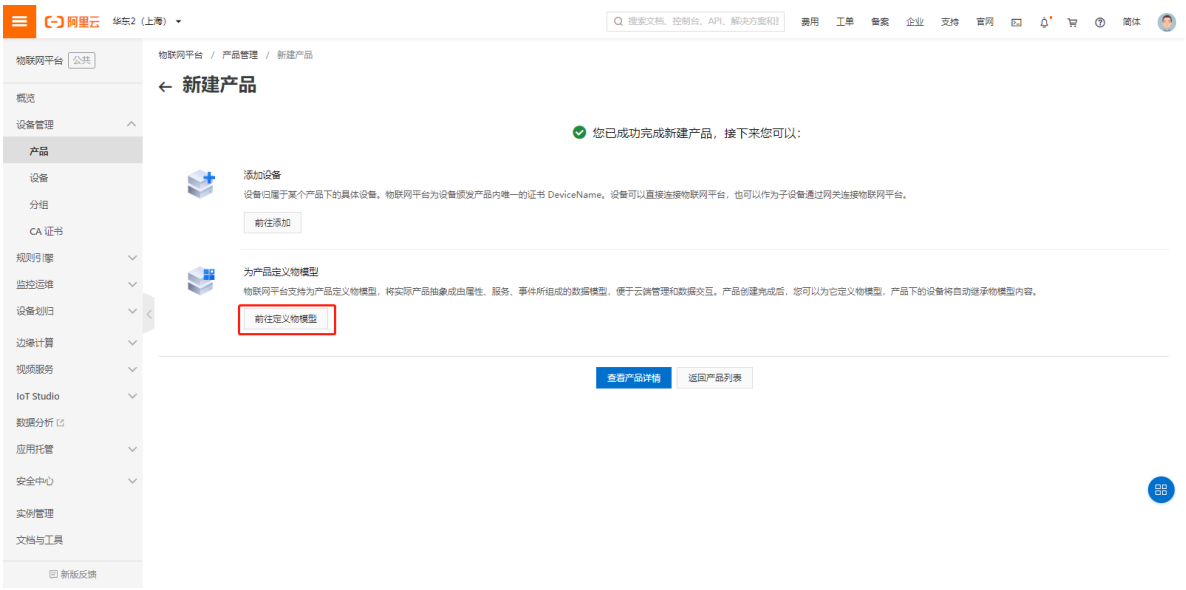
  进入“设备管理 > 产品”页面，点击“创建产品”。



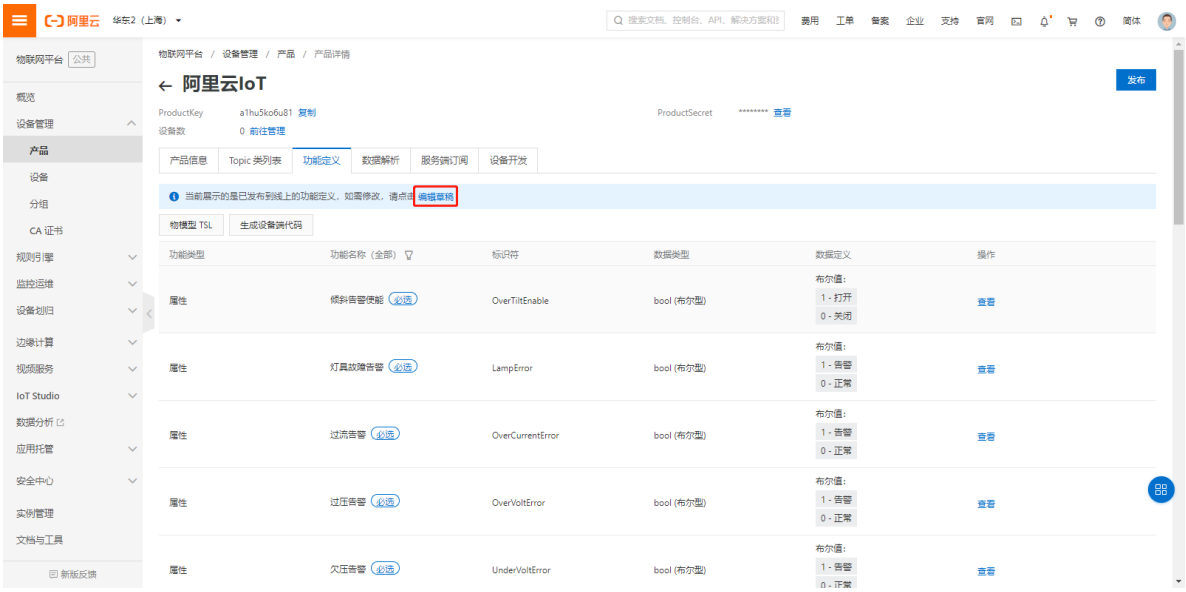
  以下为添加一个路灯照明产品的示例。注意：数据格式仅支持“ICA 标准数据格式”，认证方式仅支持“设备密钥”。

- 步骤 2：为产品定义物模型（可选）

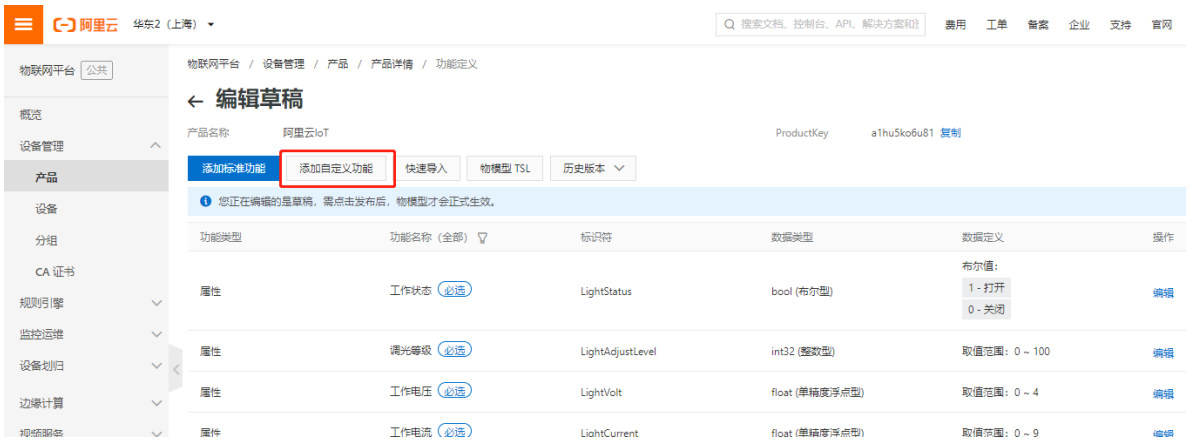
  创建完成后，点击“前往定义物模型”。



点击“编辑草稿”以配置产品的功能定义。

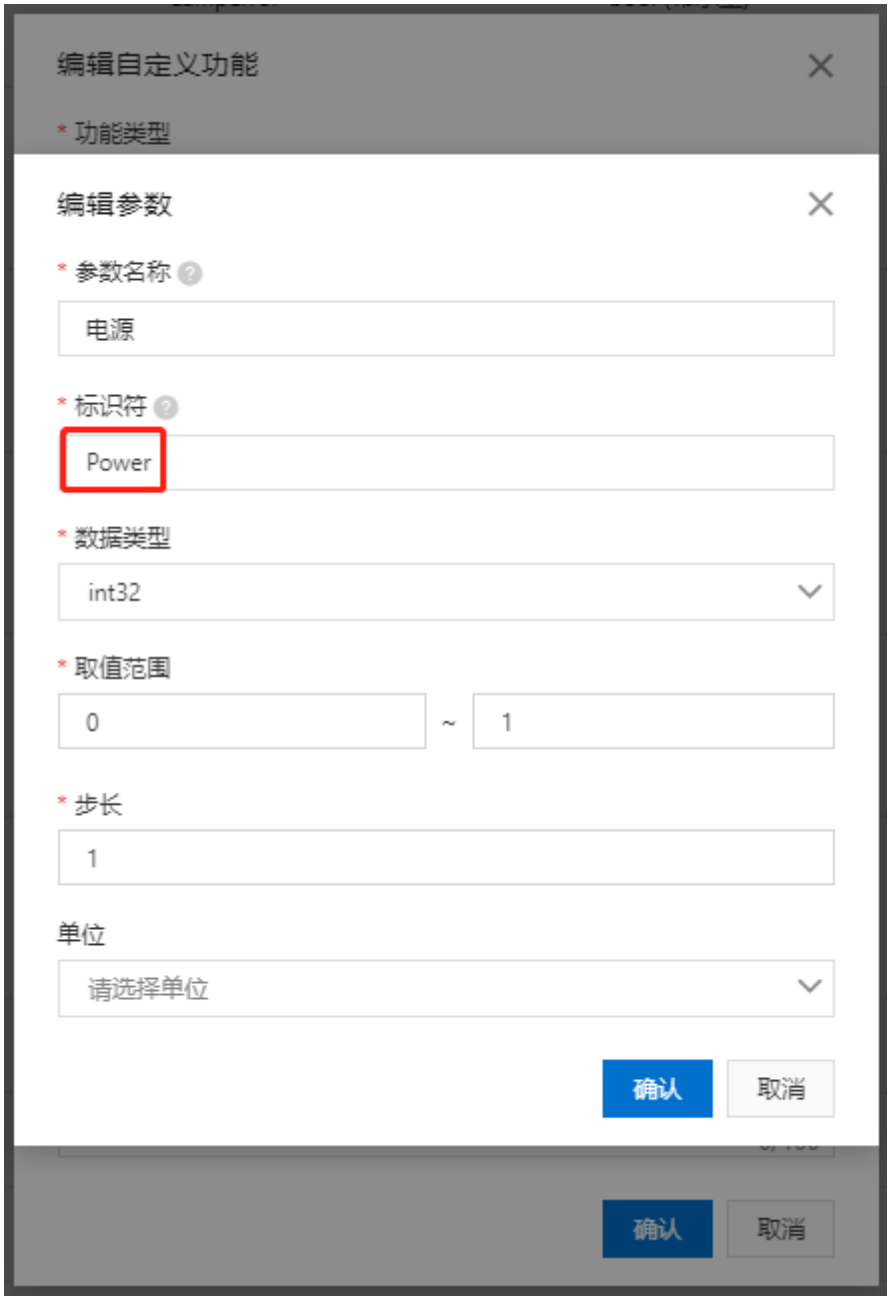


点击“添加自定义功能”并配置相应的“服务”和“事件”，关于服务和事件的介绍请参考添加单个物模型。注意：配置服务时，调用方式仅支持异步。



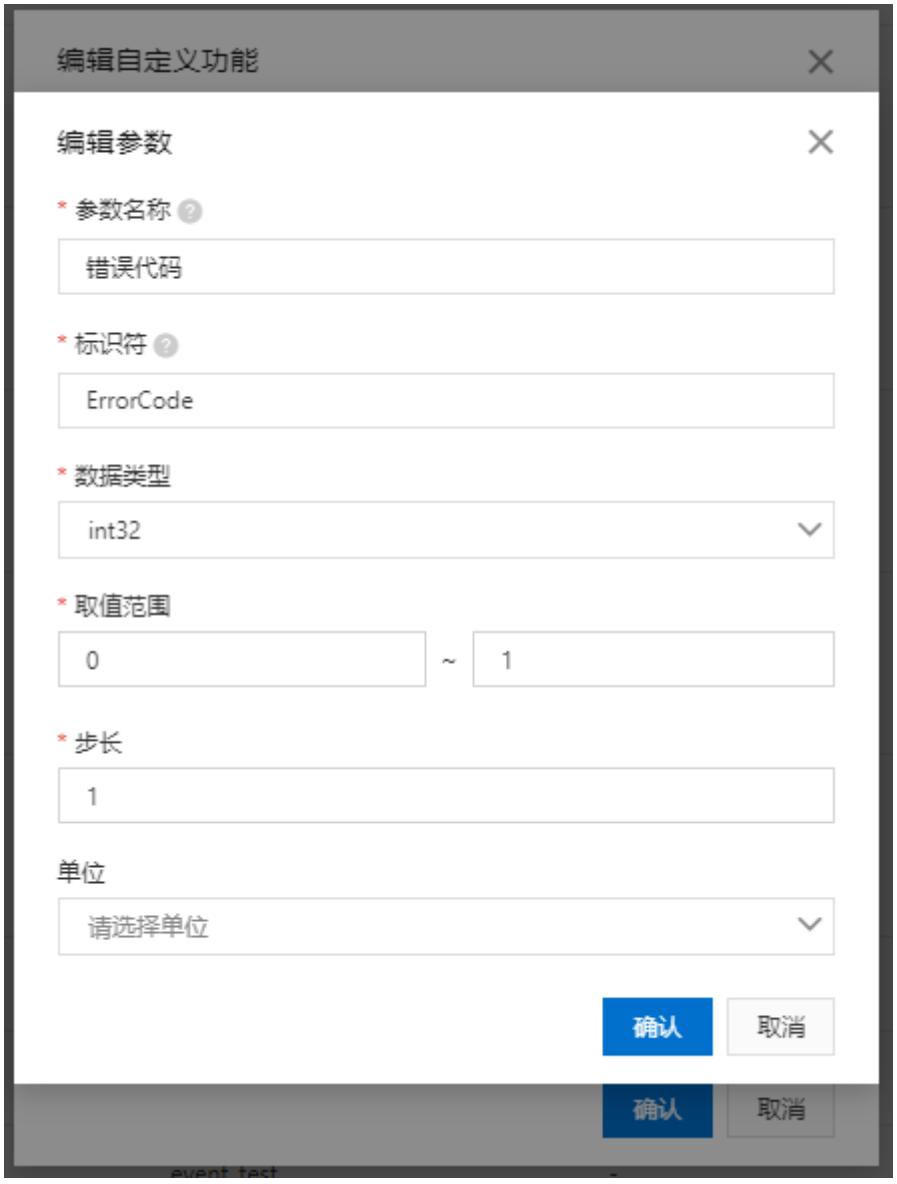
- 配置服务 (使用自定义 Topic 时无需配置此项)

在“输入参数”下点击“增加参数”，并配置相应的输入参数。

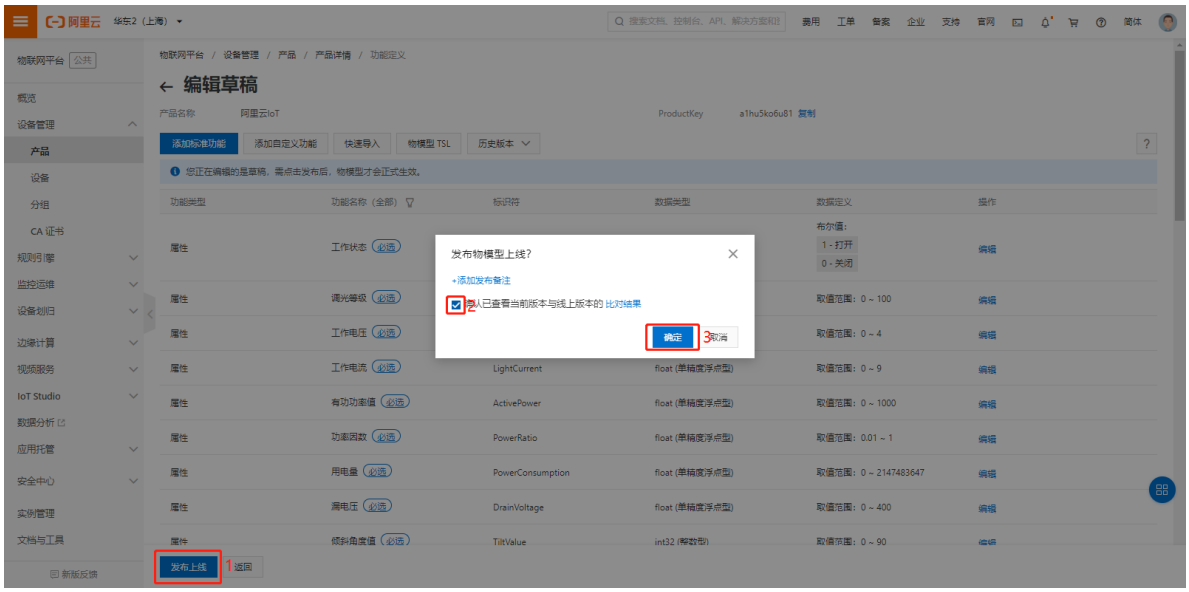


- 配置事件 (使用自定义 Topic 时无需配置此项)

在“输出参数”下点击“增加参数”，并配置相应的输出参数。

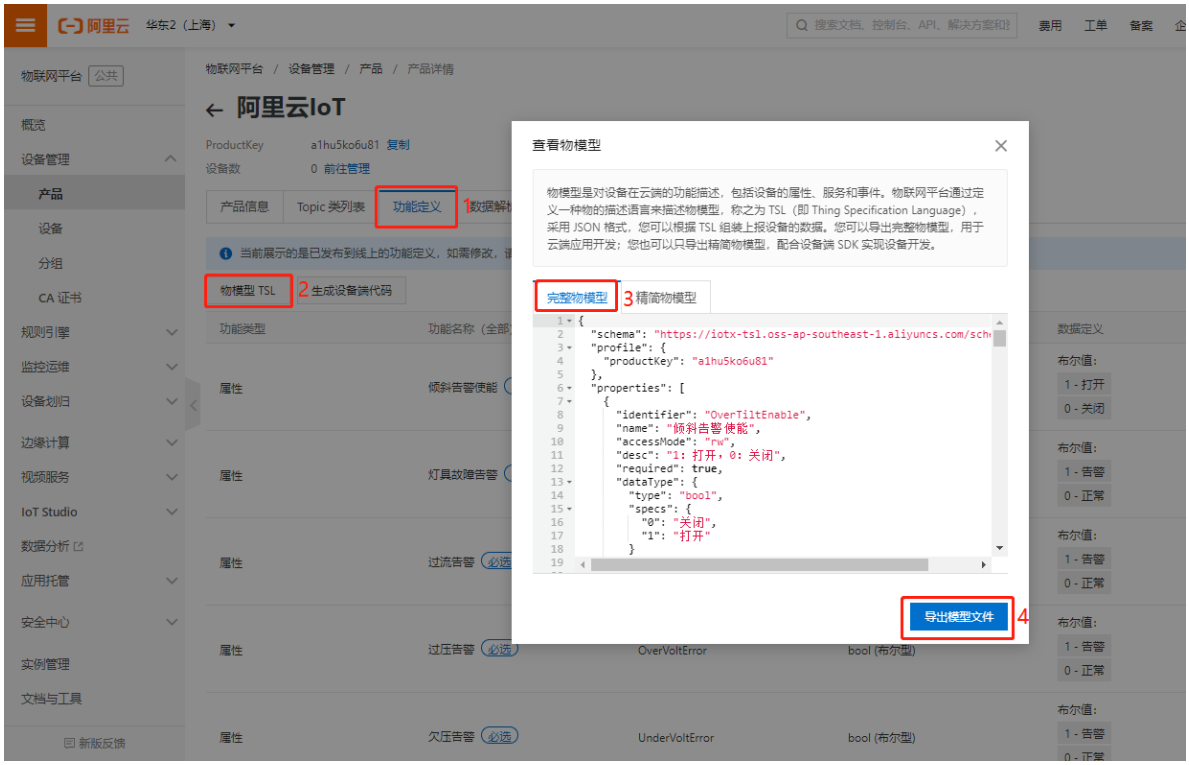


配置完成后点击“发布上线”以提交配置。



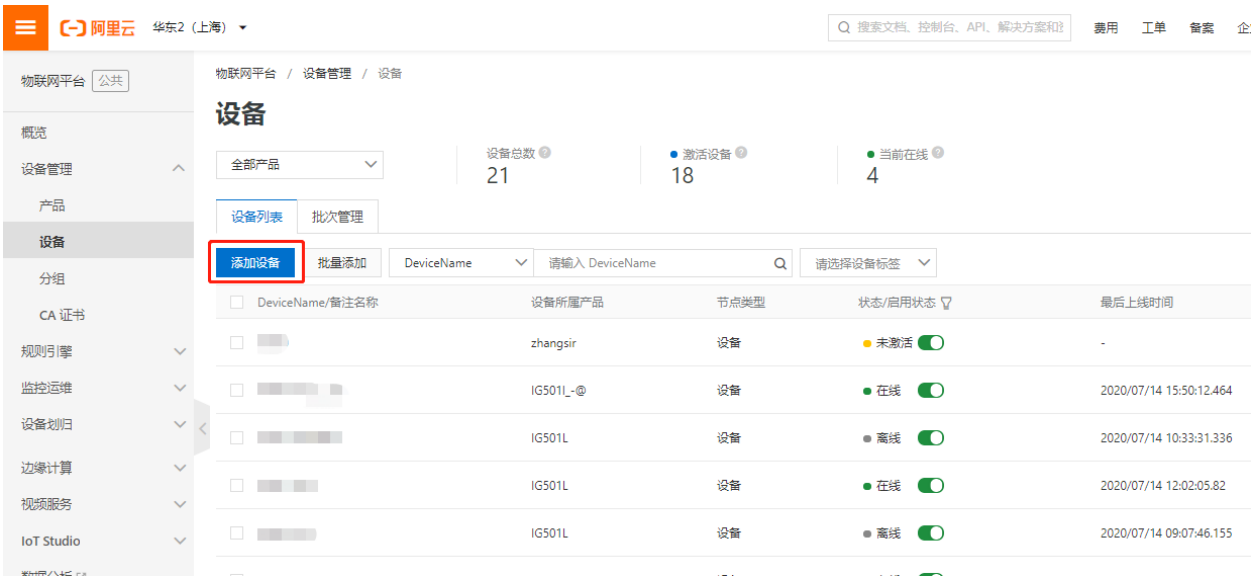
- 步骤 3：导出物模型 TSL

  产品配置完成后，在“功能定义”页面点击“物模型 TSL”，随后选择“完整物模型”并点击“导出模型文件”以备后续使用。注意：导出物模型时不要选择“精简物模型”，否则可能导致数据上报异常。

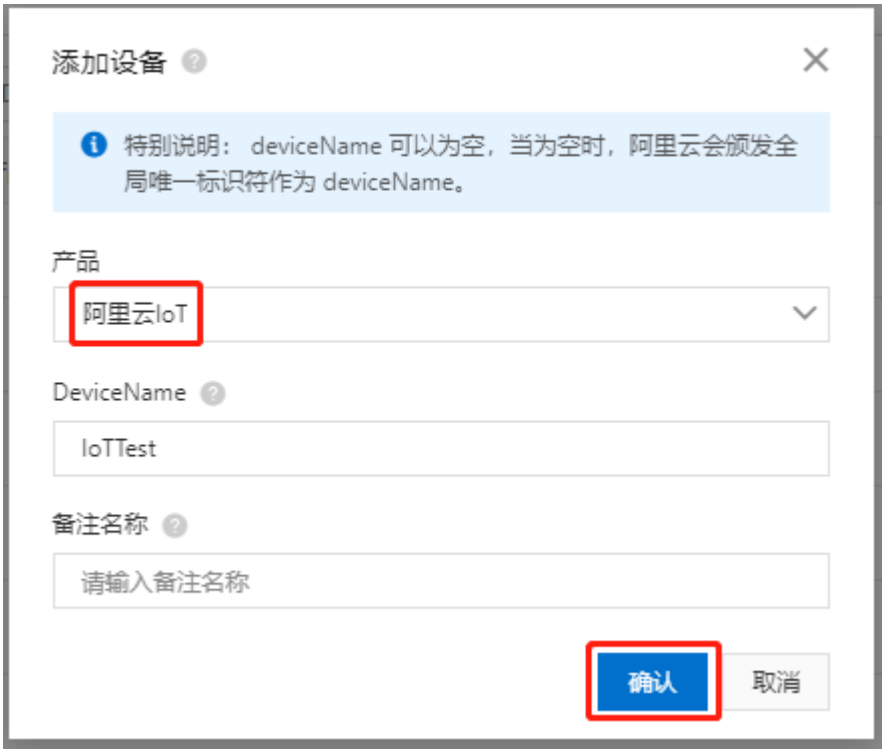


1.1.2 创建设备

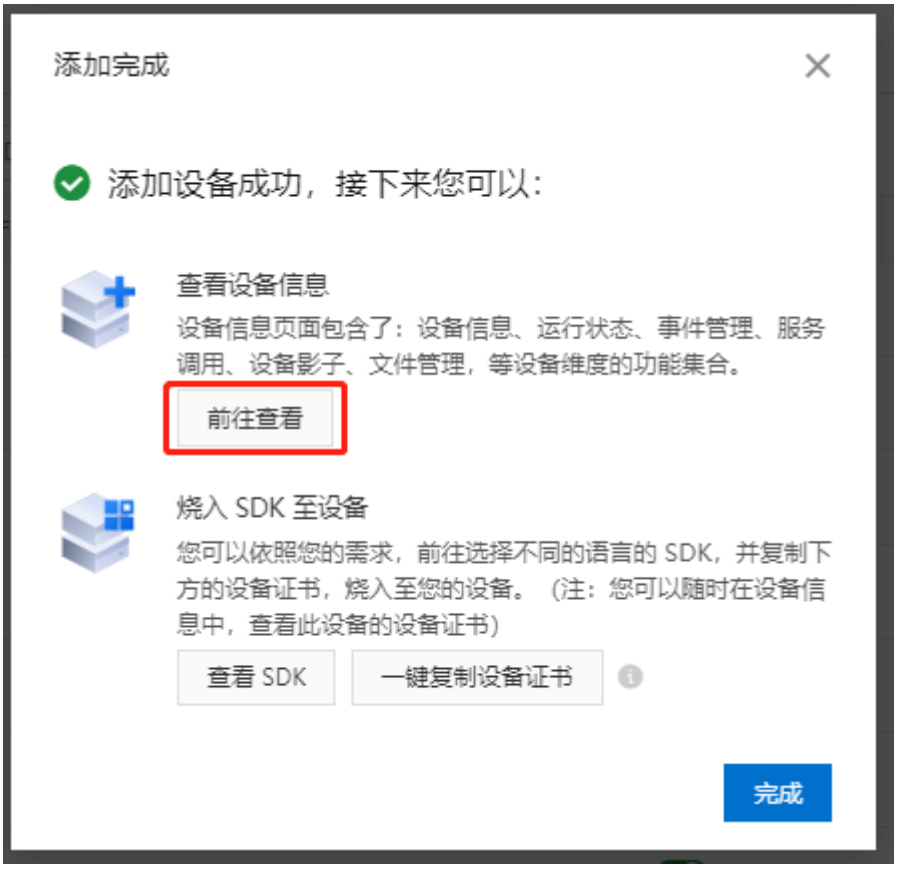
进入“设备管理 > 设备 > 设备列表”页面，点击“添加设备”。



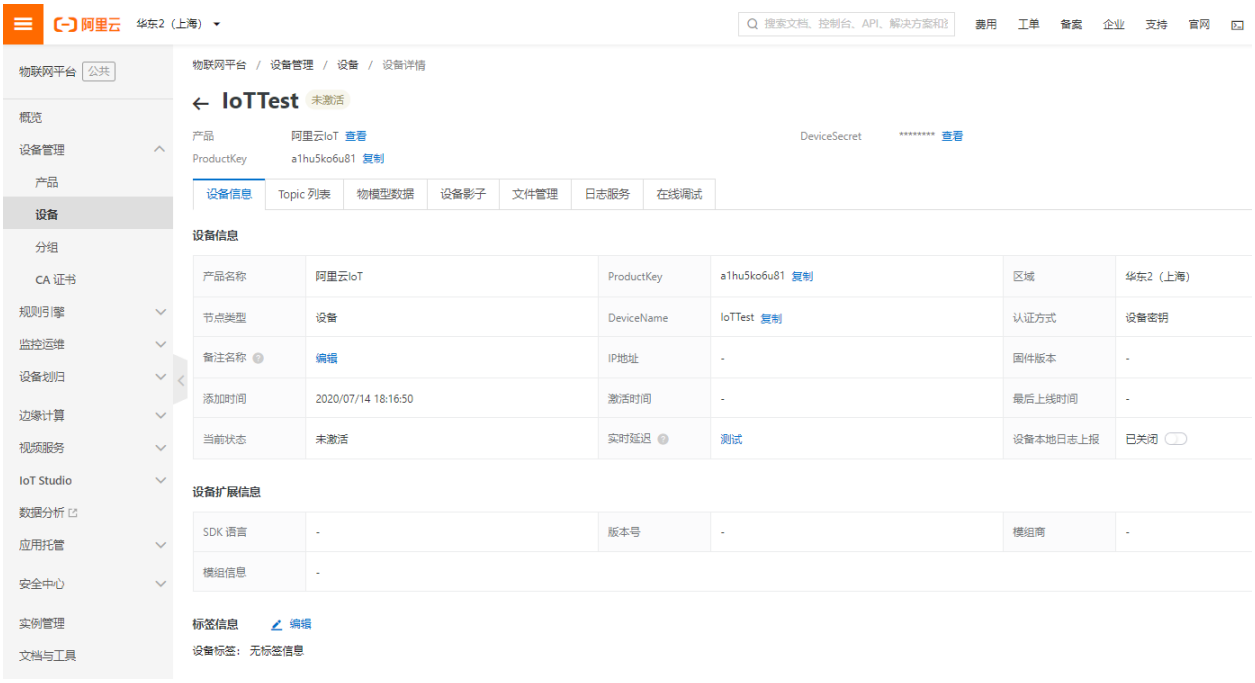
选择上一步中创建的产品并配置其他参数。



添加成功后点击“前往查看”查看设备的详细信息。



“设备详情”页面如下图所示：



至此，完成了在阿里云的物联网云平台中配置了相应的产品和设备。

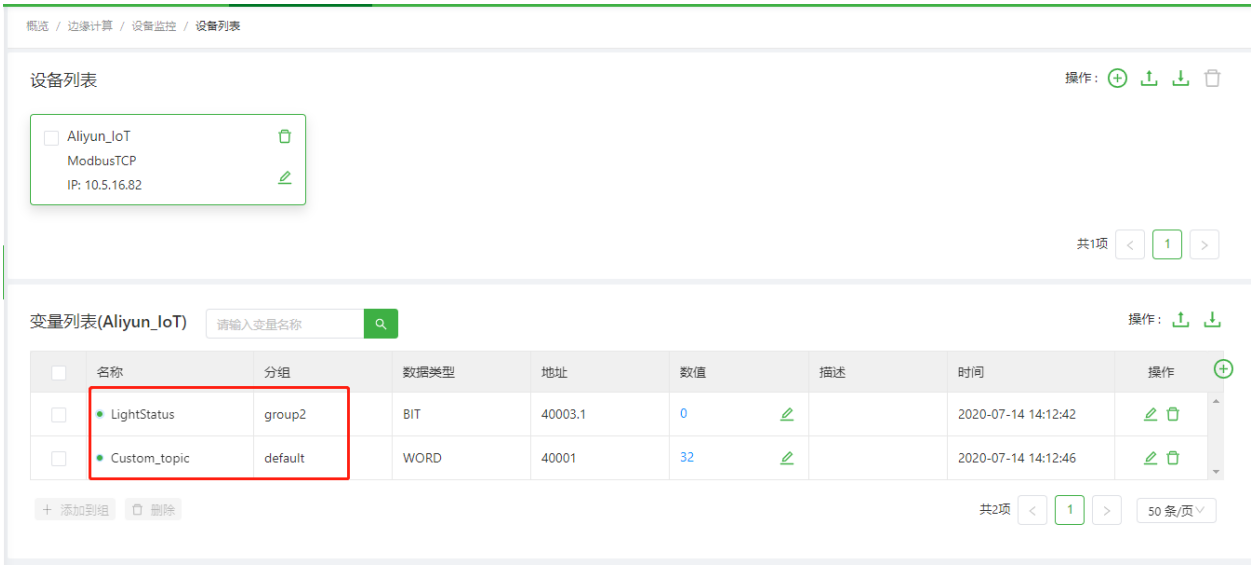
1.2 边缘计算网关配置

1.2.1 基础配置

- 如何配置 IG902 联网、更新软件版本等操作请参考IG902 快速使用手册。
- 如何配置 IG501 联网、更新软件版本等操作请参考IG501 快速使用手册。

1.2.2 配置数据采集

Device Supervisor 详细的基础数据采集配置见Device Supervisor App 用户手册。本文档配置采集了 `Custom_topic` 和 `LightStatus` 变量以及一条告警，分别用于自定义 *Topic*、属性上报和事件上报。



1.3.3 2. 发布和订阅消息

- *2.1 连接阿里云 IoT*
- *2.2 发布消息到阿里云 IoT*
- *2.3 订阅阿里云 IoT 的消息*

2.1 连接阿里云 IoT

- *2.1.1 一机一密*
- *2.1.2 一型一密*

进入 IG902 的“边缘计算 > 设备监控 > 云服务”页面，勾选“启用云服务”并选择类型为“阿里云 IoT”。阿里云 IoT 的认证方式有两种，一机一密和一型一密。关于认证方式的详细介绍请参考阿里云设备安全认证。

2.1.1 一机一密

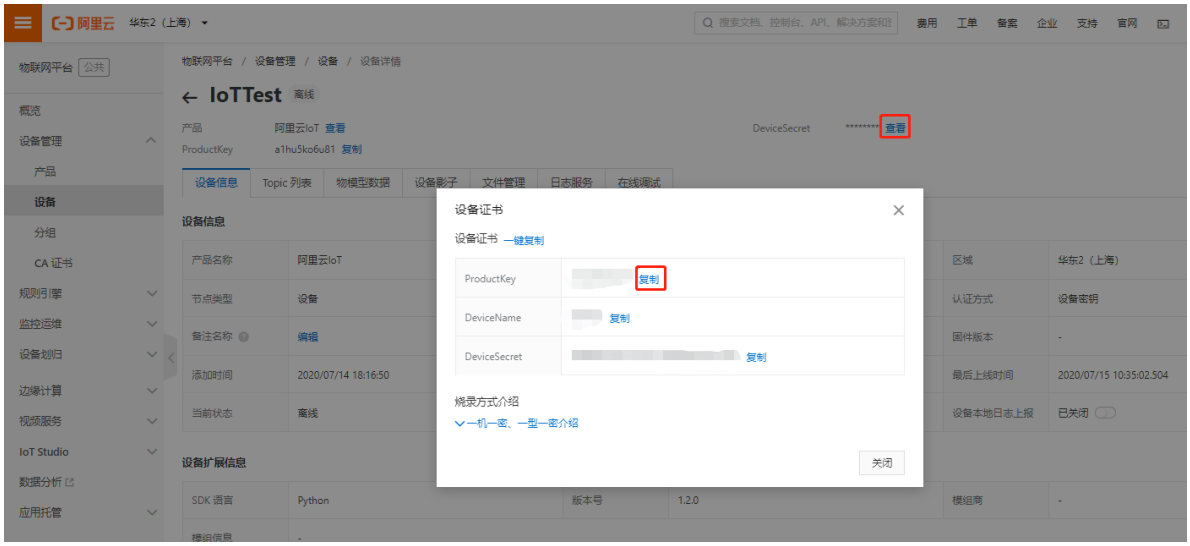
使用“一机一密”的认证方式与阿里云 IoT 建立连接时，选择认证方式为“一机一密”。示例配置如下：

各项参数说明如下：

- 类型：连接阿里云 IoT 时，选择“阿里云 IoT”
- 云端域名：云端域名的配置方式请参考地域和可用区。文档中的地域名称为华东 2，因此使用 `cn-shanghai`。

地域名称	所在城市	Region ID	可用区数量
华北 1	青岛	cn-qingdao	2
华北 2	北京	cn-beijing	8
华北 3	张家口	cn-zhangjiakou	3
华北 5	呼和浩特	cn-huhehaote	2
华北 6	乌兰察布	cn-wulanchabu	2
华东 1	杭州	cn-hangzhou	8
华东 2	上海	cn-shanghai	7
华南 1	深圳	cn-shenzhen	5
华南 2	河源	cn-heyuan	2
西南 1	成都	cn-chengdu	2

- 认证方式：根据实际情况选择认证方式，本次使用一机一密。
- `ProductKey`：进入“设备管理 > 设备 > 设备详情”页面，点击“DeviceSecret”后的“查看”以复制认证信息。



- `DeviceName`：获取方法同一机一密的 ProductKey。
- `DeviceSecret`：获取方法同一机一密的 ProductKey。
- 物模型 `TSL`：导入创建产品时保存的物模型文件。说明：“消息管理”中仅使用“自定义 Topic”类型的消息时可以不导入物模型。

连接阿里云 IoT 成功后如下图所示：

2.1.2 一型一密

使用“一型一密”的认证方式与阿里云 IoT 建立连接时，选择认证方式为“一型一密”。示例配置如下：

注意:

- 认证方式为“一型一密”时，需要开启“动态注册”。
- 仅未激活的设备支持通过一型一密认证方式与阿里云 IoT 建立连接且在连接建立后设备自动激活，激活后 IG902 切换为一机一密与阿里云 IoT 连接。你可以在“设备详情”页面查看设备是否有激活时间来判断设备是否已激活。

各项参数说明如下:

- 类型：连接阿里云 IoT 时，选择“阿里云 IoT”

- 云端域名：云端域名的配置方式请参考地域和可用区。文档中的地域名称为华东 2，因此使用 `cn-shanghai`。

- 认证方式：根据实际情况选择认证方式，本次使用一型一密。

- `ProductKey`：进入“设备管理 > 产品 > 产品详情”页面，点击“ProductSecret”后的“查看”以复制认证信息。

- `DeviceName`：设备名称，获取方法同一机一密的 ProductKey。
- `ProductSecret`：获取方法同一型一密的 ProductKey。
- 物模型 `TSL`：导入创建产品时保存的物模型文件。说明：“消息管理”中仅使用“自定义 Topic”类型的消息时可以不导入物模型。

连接阿里云 IoT 成功后如下图所示：

2.2 发布消息到阿里云 IoT

- *2.2.1* 自定义 *Topic*
- *2.2.2* 属性上报
- *2.2.3* 事件上报

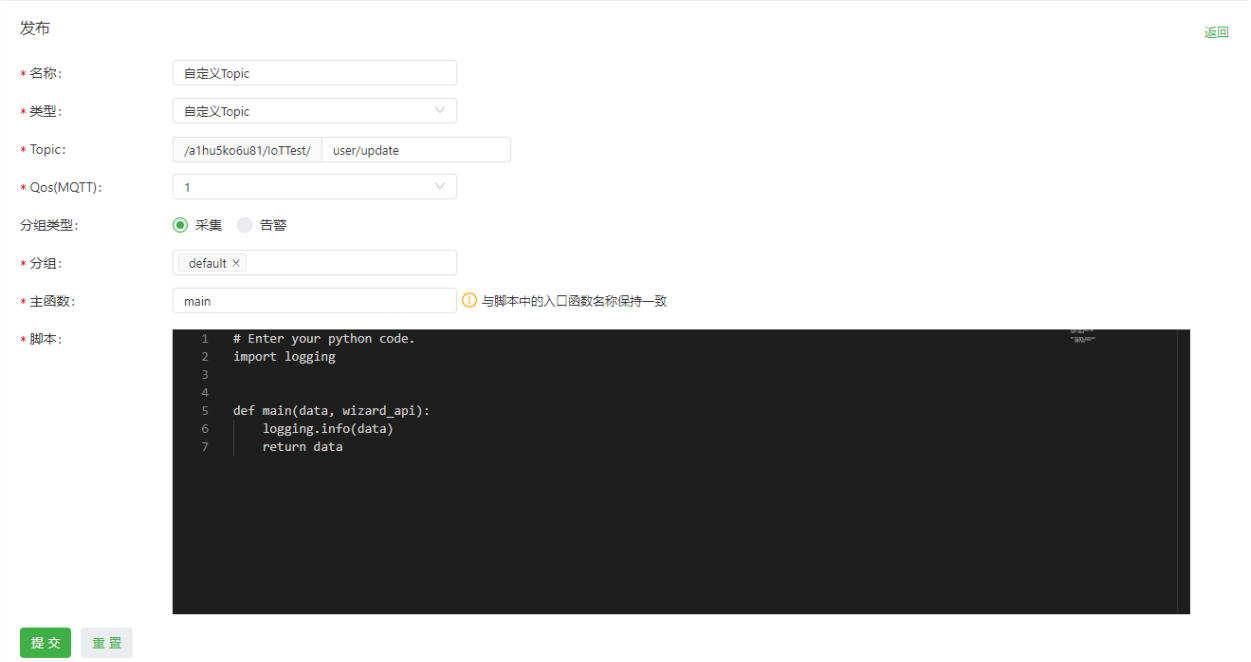
进入 IG902 的“边缘计算 > 设备监控 > 云服务”页面，在“消息管理 > 发布”中添加发布消息。发布消息支持三种类型的发布消息：自定义 `topic`、属性上报和事件上报。

数据上报时的数据格式要求见物模型开发。注意：使用属性上报和事件上报时，请确保 IG902 中的“物模型

TSL”与阿里云 IoT 中产品的物模型一致。

2.2.1 自定义 Topic

发布消息类型为自定义 `topic` 的示例配置如下：



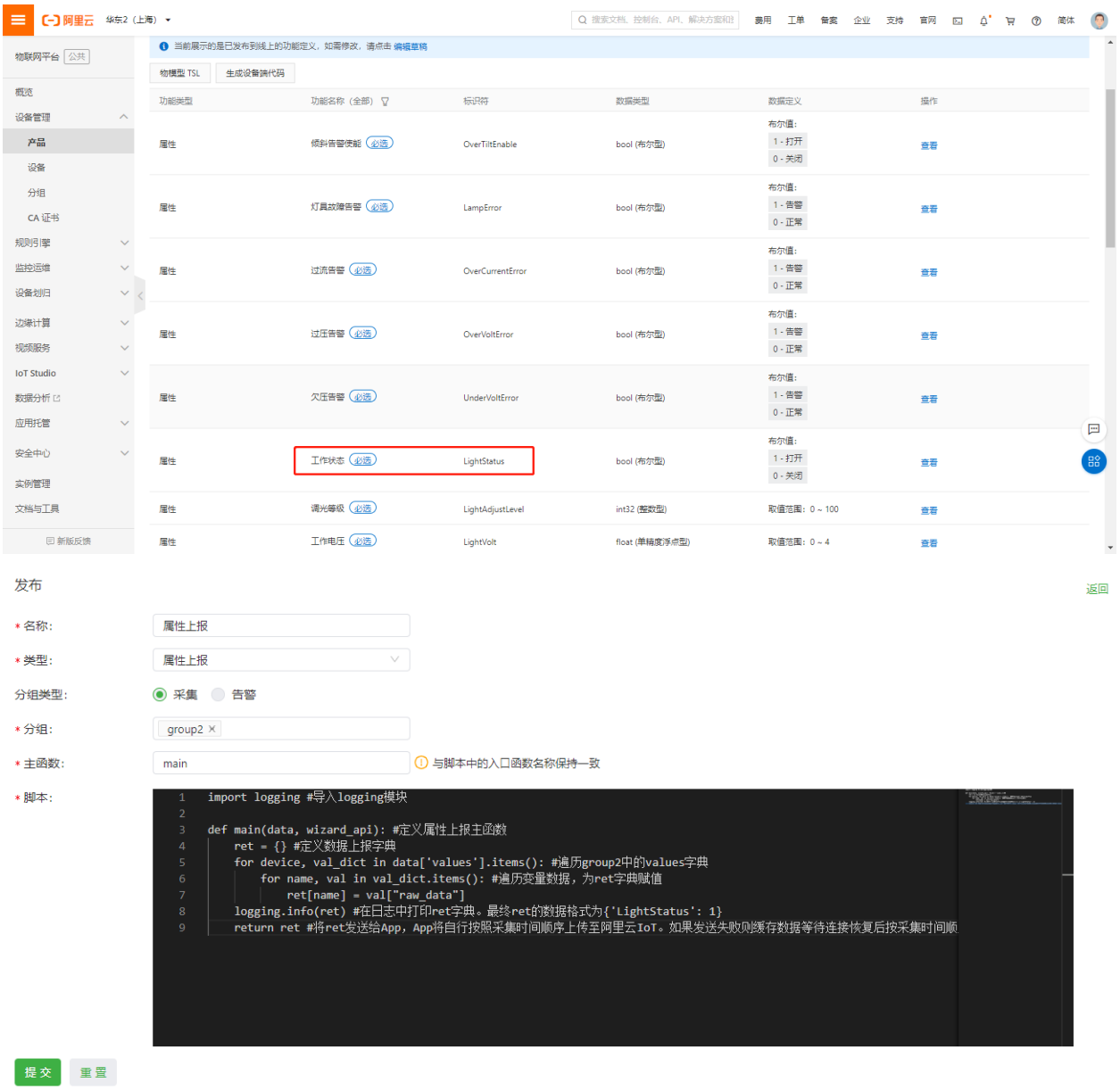
发布消息配置参数说明如下：

- 名称：用户自定义发布名称
- 类型：发布消息的类型
- 主题：发布主题，与阿里云 IoT 的“设备管理 > 产品 > 产品详情 > Topic 类列表 > 自定义 Topic”页面中“操作权限”为“发布”的 Topic 类保持一致。说明：主题中的 ProductKey 和 DeviceName 信息会自动补全，配置主题时复制 `${deviceName}/`后的主题信息即可。

- Qos(MQTT)：发布 Qos，建议与 MQTT 服务器的 Qos 保持一致
  - 0：只发送一次消息，不进行重试
  - 1：最少发送一次消息，确保消息到达 MQTT 服务器
- 分组类型：发布变量数据时请选择“采集”，随后在分组中仅能选择“采集组”；发布告警数据时请选择“告警”，随后在分组中仅能选择“告警组”
- 分组：选择相应的分组后，分组下所有变量通过该发布配置将数据上传至 MQTT 服务器；可选择多个分组，当选择多个分组时，按照分组的采集间隔分别对各分组下的变量执行发布中的脚本逻辑。分组中必须包含变量，否则不会执行发布中的脚本逻辑
- 主函数：主函数名称，即入口函数名称，与脚本中的入口函数名称保持一致
- 脚本：使用 Python 代码自定义组包和处理逻辑，主函数参数包括：
  - 参数 1：同标准 MQTT-发布主函数中的参数 1
  - 参数 2：Device Supervisor 的阿里云 IoT api 接口，参数说明见 *Device Supervisor 的阿里云 IoT api 接口说明*

2.2.2 属性上报

你可以进入阿里云 IoT 的“设备管理 > 产品 > 产品详情 > 功能定义”页面查看产品的属性、事件等信息。示例的产品具备一个名称为“工作状态”的属性，该属性的标识符为“LightStatus”。将配置数据采集中配置的“LightStatus”变量通过属性上报上传至阿里云 IoT，配置如下。



发布消息配置参数说明请参考发布-自定义 *Topic*。脚本代码及说明如下：

```
import logging #导入logging模块

def main(data, wizard_api): #定义属性上报主函数
    ret = {} #定义数据上报字典
```

(续下页)

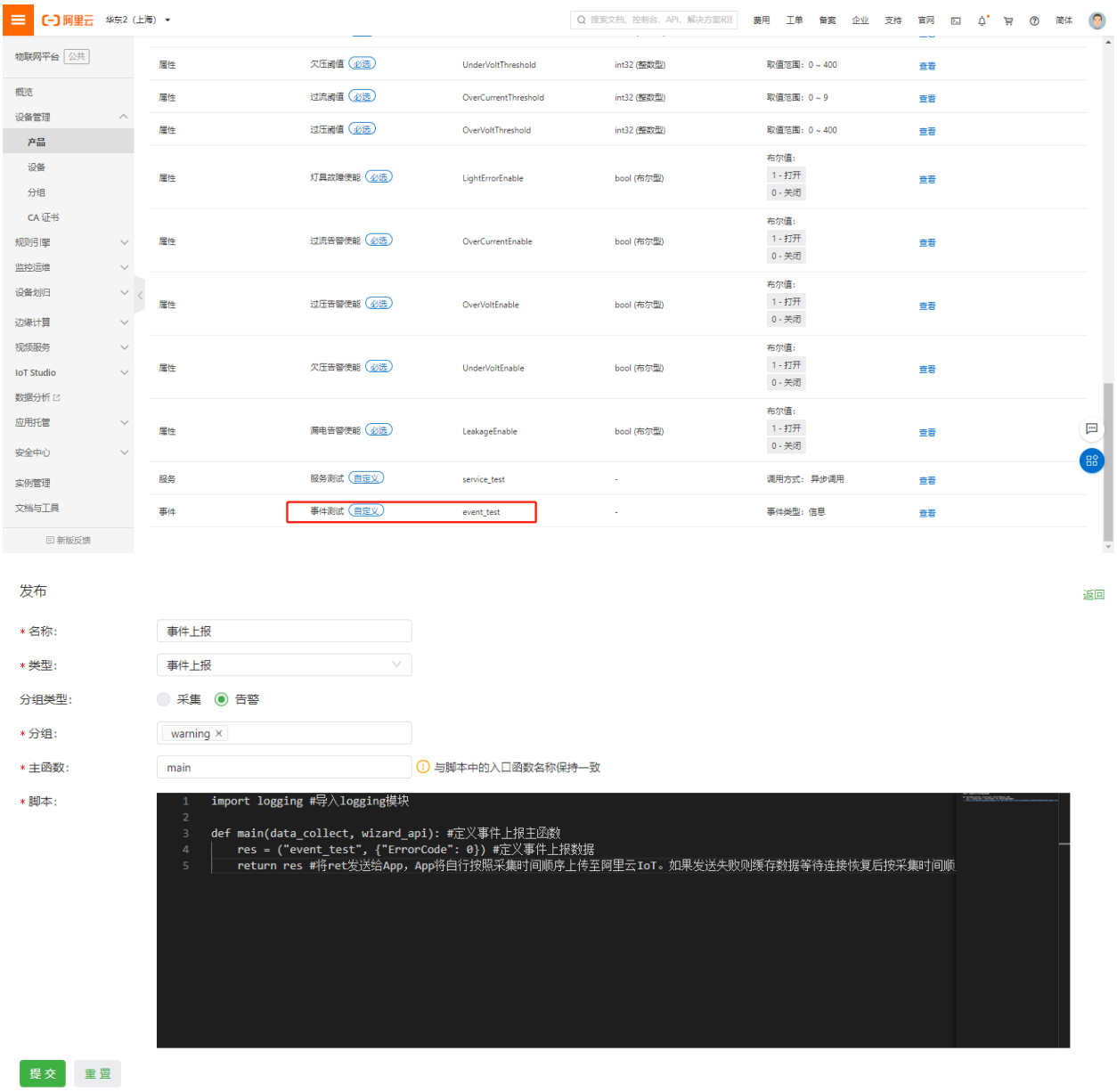
(接上页)

```
    for device, val_dict in data['values'].items(): #遍历group2中的values字典
        for name, val in val_dict.items(): #遍历变量数据，为ret字典赋值
            ret[name] = val["raw_data"]
    logging.info(ret) #在日志中打印ret字典。最终ret的数据格式为{'LightStatus': 1}
    return ret
↪#将ret发送给App，App将自行按照采集时间顺序上传至阿里云IoT。如果发送失败则缓存数据等待连接恢复后按采
```

可在阿里云 IoT 的“设备管理 > 设备 > 设备详情 > 物模型数据 > 运行状态”页面查看属性状态。

2.2.3 事件上报

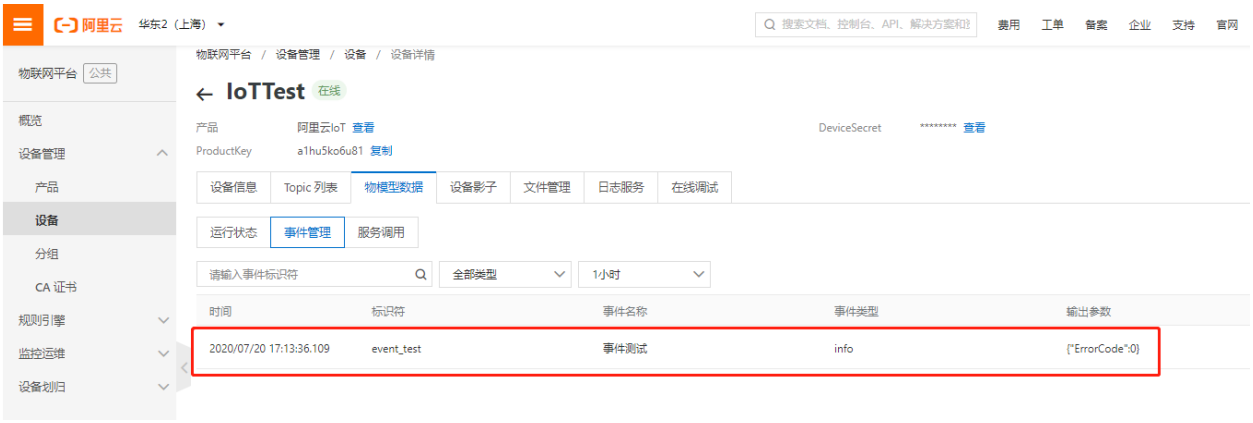
你可以进入阿里云 IoT 的“设备管理 > 产品 > 产品详情 > 功能定义”页面查看产品的属性、事件等信息。示例的产品具备一个名称为事件测试的事件，该事件的标识符为“event_test”，输出参数的标识符为“ErrorCode”。当配置数据采集中配置的“event_test”告警触发时，通过事件上报上传至阿里云 IoT，配置如下。



发布消息配置参数说明请参考发布-自定义 *Topic*。脚本代码及说明如下：

```
import logging #导入logging模块

def main(data_collect, wizard_api): #定义事件上报主函数
    res = ("event_test", {"ErrorCode": 0}) #定义事件上报数据
    return res
↪#将ret发送给App，App将自行按照采集时间顺序上传至阿里云IoT。如果发送失败则缓存数据等待连接恢复后按采
```

可在阿里云 IoT 的“设备管理 > 设备 > 设备详情 > 物模型数据 > 事件管理”页面查看事件信息。



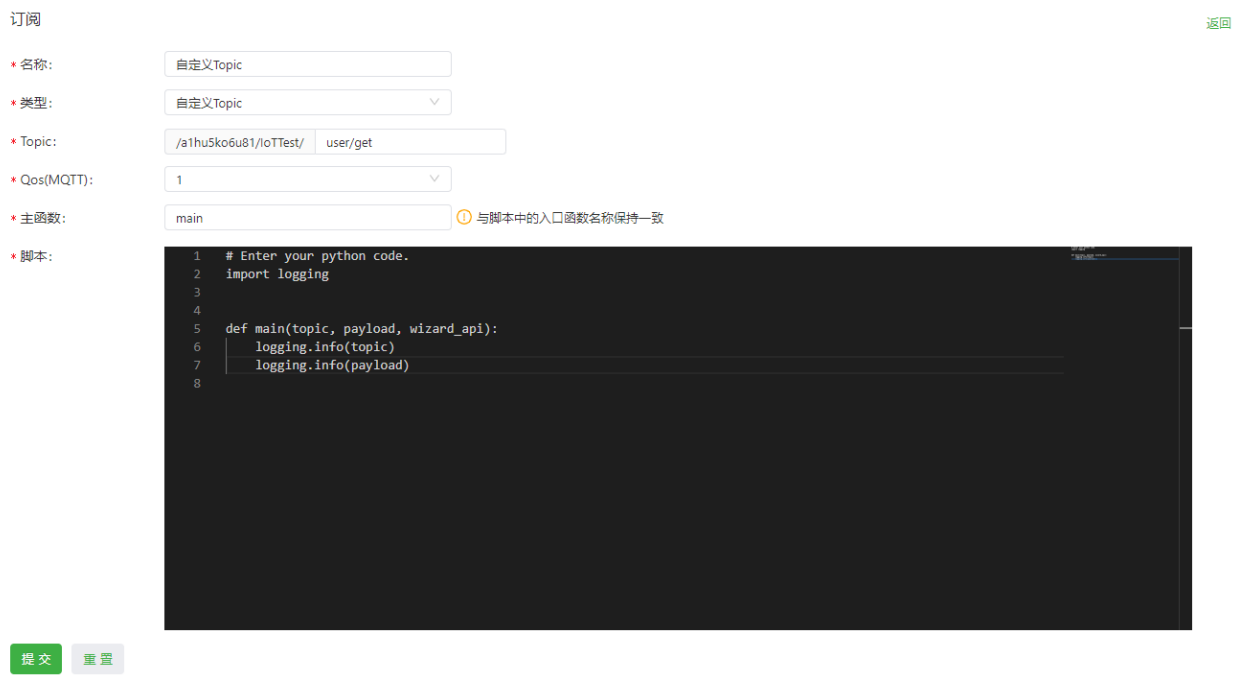
2.3 订阅阿里云 IoT 的消息

- *2.3.1 自定义 Topic*
- *2.3.2 服务调用*
- *2.3.3 属性设置*

进入 IG902 的“边缘计算 > 设备监控 > 云服务”页面，在“消息管理 > 订阅”中添加订阅消息。订阅消息支持三种类型的订阅消息：自定义 topic、服务调用和属性设置。

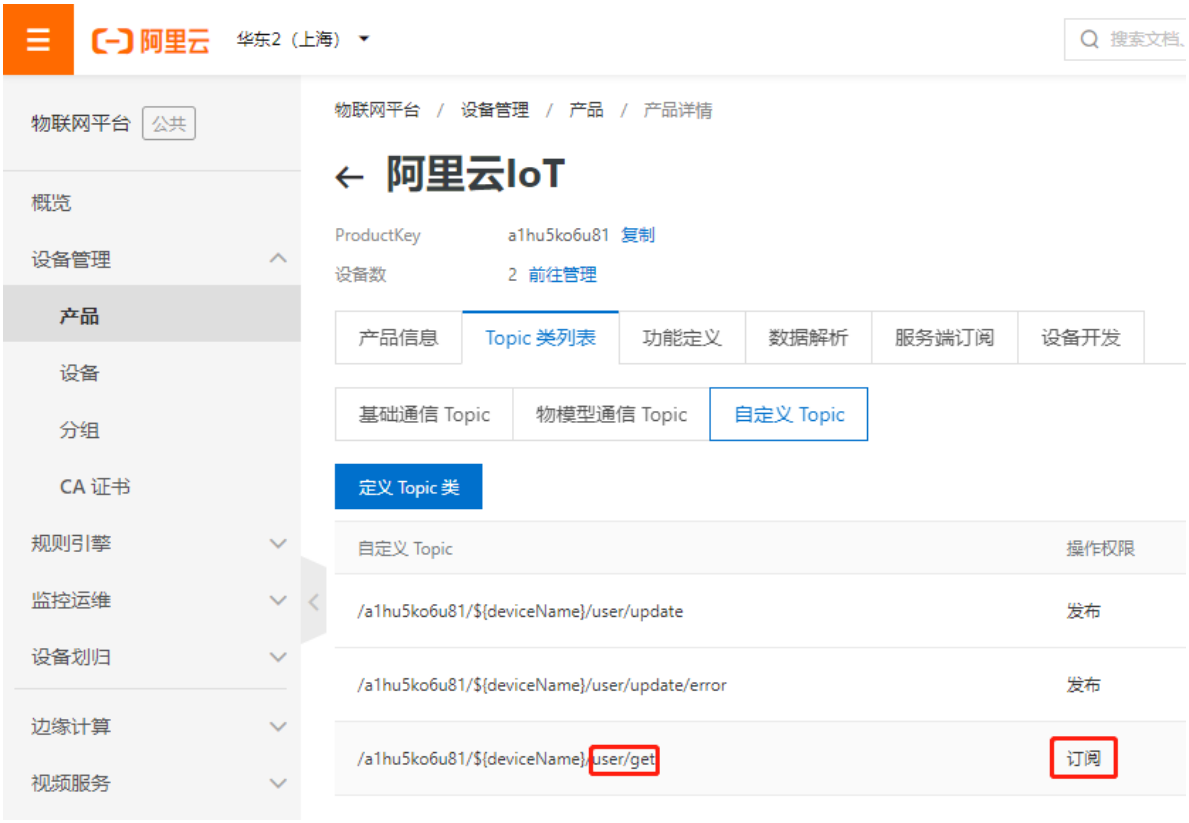
2.3.1 自定义 Topic

订阅消息类型为自定义 topic 的示例配置如下：



订阅消息配置参数说明如下：

- 名称：自定义订阅名称
- 主题：订阅主题，与阿里云 IoT 的“设备管理 > 产品 > 产品详情 > Topic 类列表 > 自定义 Topic”页面中“操作权限”为“订阅”的 Topic 类保持一致。说明：主题中的 ProductKey 和 DeviceName 信息会自动补全，配置主题时复制 `${deviceName}/`后的主题信息即可。



- `Qos(MQTT)`：订阅 Qos，默认为 1
- 主函数：主函数名称，即入口函数名称，与脚本中的入口函数名称保持一致
- 脚本：使用 Python 代码自定义组包和处理逻辑，自定义 Topic 的订阅主函数参数包括：
  - 参数 1：该参数为接收到的主题，数据类型为 `string`
  - 参数 2：该参数为接收到的数据，数据类型为 `string`
  - 参数 3：Device Supervisor 的阿里云 IoT api 接口，参数说明见 *Device Supervisor 的阿里云 IoT api 接口说明*

2.3.2 服务调用

以下是服务调用的示例，该示例可以接受阿里云 IoT 下发的“服务测试”设置指令。注意：使用服务测试时，请确保 IG902 中的“物模型 TSL”与阿里云 IoT 中产品的物模型一致。

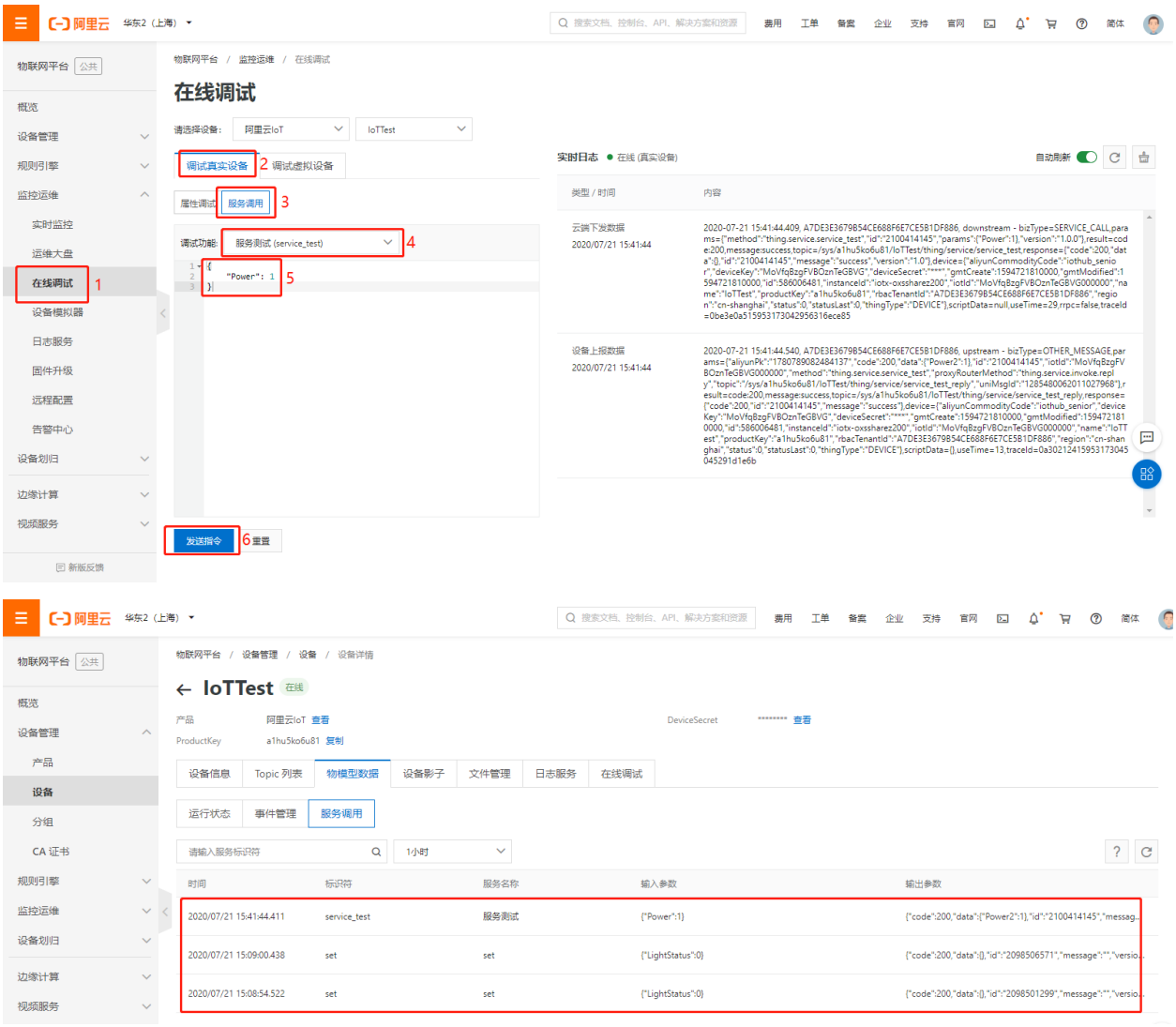
订阅消息配置参数说明请参考订阅-自定义 *Topic*，服务调用的订阅主函数参数包括：

- 参数 1：服务标识符
- 参数 2：请求 ID
- 参数 3：服务输入参数
- 参数 4：预留参数
- 参数 5：Device Supervisor 的阿里云 IoT api 接口，参数说明见 *Device Supervisor* 的阿里云 *IoT api* 接口说明

脚本代码及说明如下：

```
import logging #导入logging模块

def main(identifier, request_id, params, userdata, wizard_api): #定义属性设置主函数
    logging.info(identifier) #打印服务标识符
    logging.info(request_id) #打印请求ID
    logging.info(params) #打印服务参数
    wizard_api.thing_answer_service(identifier, request_id, 200, {"Power2": 1})
↪#调用wizard_api模块中的thing_answer_service方法，返回服务响应信息
```

进入阿里云 IoT 的“监控运维 > 在线调试 > 调试真实设备 > 服务调用”页面选择相应的产品和设备进行服务调用测试；在“设备管理 > 设备 > 设备详情 > 物模型数据 > 服务调用”页面查看服务调用信息。



2.3.3 属性设置

以下是属性设置的示例，该示例可以接受阿里云 IoT 下发的“工作状态”指令。注意：使用属性设置时，请确保 IG902 中的“物模型 TSL”与阿里云 IoT 中产品的物模型一致。

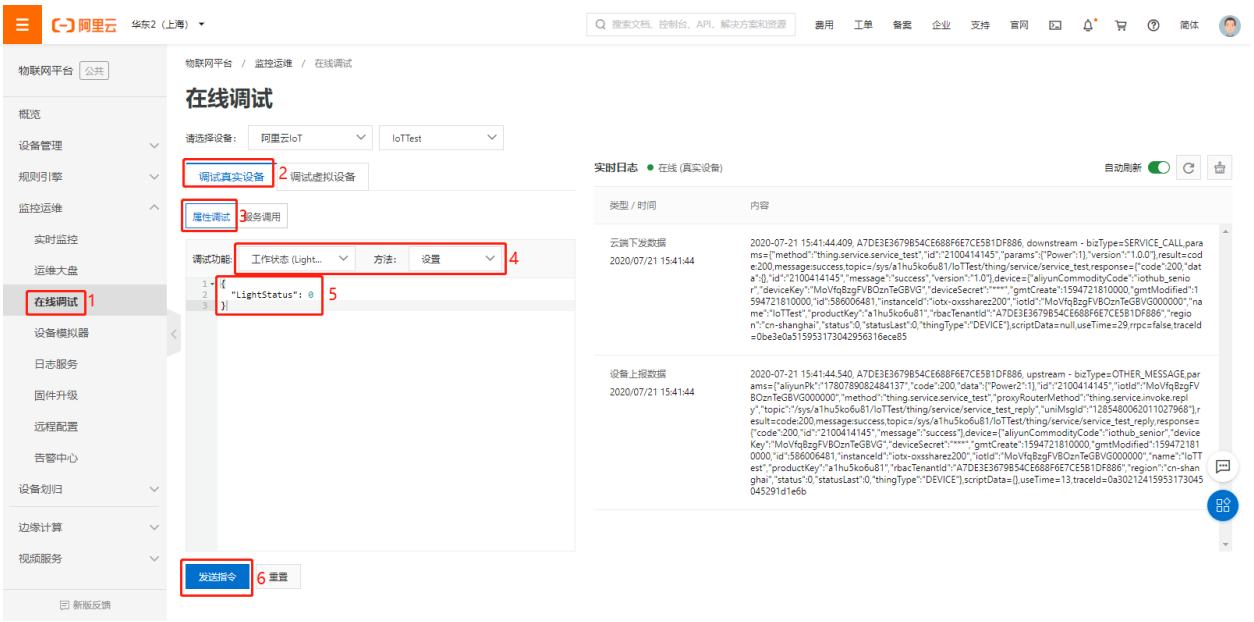
订阅消息配置参数说明请参考订阅-自定义 *Topic*，属性设置的订阅主函数参数包括：

- 参数 1：下发的属性数据
- 参数 2：预留参数
- 参数 3：Device Supervisor 的阿里云 IoT api 接口，参数说明见 *Device Supervisor* 的阿里云 *IoT api* 接口说明

脚本代码及说明如下：

```
import logging #导入logging模块

def main(params, userdata, wizard_api): #定义属性设置主函数
    logging.info(params) #打印params数据
    logging.info(userdata) #打印userdata数据
    wizard_api.write_plc_values({"LightStatus":params["LightStatus"]}) #调用wizard_
↪api模块中的write_plc_values方法，将数据下发至LightStatus变量
```

进入阿里云 IoT 的“监控运维 > 在线调试 > 调试真实设备 > 属性调试”页面选择相应的产品、设备、调试功能和方法进行属性设置测试；在“设备管理 > 设备 > 设备详情 > 物模型数据 > 运行状态”页面查看设备运行状态。


阿里云
华东2（上海）
物联网平台
公共
概览
设备管理
规则引擎
监控运维
实时监控
运维大盘
在线调试
设备模拟器
日志服务
固件升级
远程配置
告警中心
设备划归
边缘计算
视频服务
新版反馈
物联网平台 / 监控运维 / 在线调试
在线调试
请选择设备：
阿里云IoT
IoTTest
调试真实设备
2 调试虚拟设备
属性调试
3 服务调用
调试功能：
工作状态 (Light...
方法：
设置
4
{
"LightStatus": 0
}
5
发送指令
6 重置
实时日志
在线 (真实设备)
自动刷新
类型 / 时间
内容
云端下发数据
2020/07/21 15:41:44
设备上报数据
2020/07/21 15:41:44


阿里云
华东2（上海）
搜索文档、控制台、API、
物联网平台
公共
概览
设备管理
产品
设备
分组
CA 证书
规则引擎
监控运维
设备划归
边缘计算
视频服务
IoT Studio
数据分析
物联网平台 / 设备管理 / 设备 / 设备详情
← IoTTest
在线
产品
阿里云IoT 查看
ProductKey
a1hu5ko6u81 复制
设备信息
Topic 列表
物模型数据
设备影子
文件管理
日志服务
在线调试
运行状态
事件管理
服务调用
实时刷新
工作状态
查看数据
0 (关闭)
2020/07/21 15:58:53.270
倾斜告警使能
查看数据
0 (关闭)
2020/07/21 10:58:22.126
过压告警
查看数据
--
欠压告警
查看数据
--

1.3.4 附录

Device Supervisor 的阿里云 IoT api 接口说明

wizard_api 的基础配置方法请参考Device Supervisor 的 api 接口说明。当云服务类型为阿里云 IoT 时，wizard_api 额外提供以下方法，各方法的说明和格式要求见物模型开发。

- thing_post_property(prop_data)
  - 方法说明：属性上报方法
  - 参数：prop_data，需要上报的属性数据
  - 使用示例：

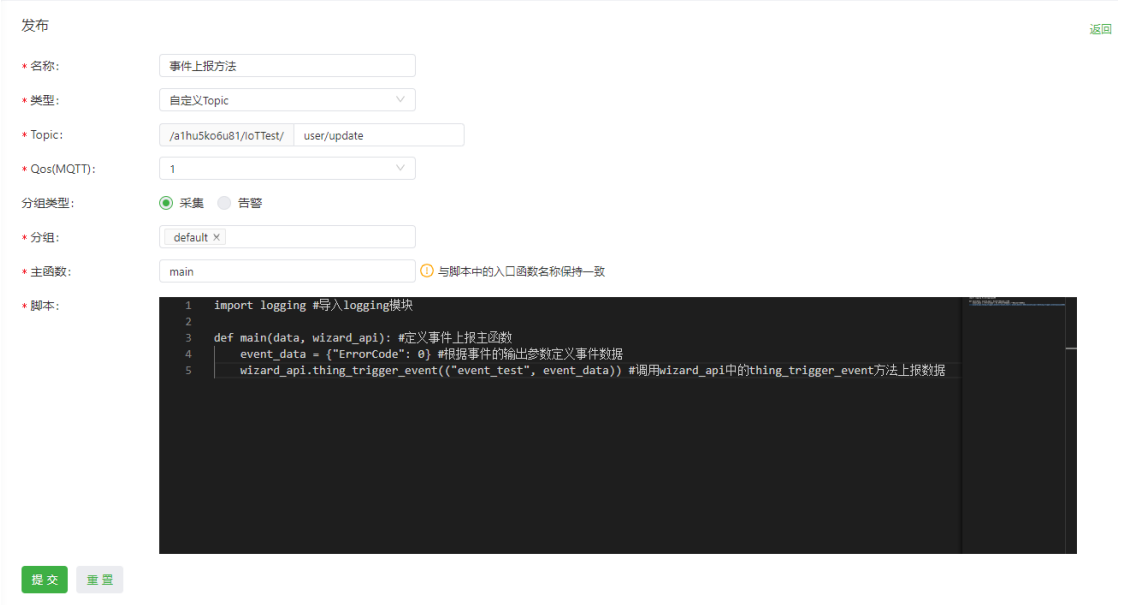

```
import logging #导入logging模块

def main(data, wizard_api): #定义属性上报主函数
    prop_data = { #定义上报数据
        "abs_speed": 11,
        "power_stage": 10
    }
    wizard_api.thing_post_property(prop_data) #调用wizard_api中的thing_post_
↪property方法上报数据
```

- thing_trigger_event((identifier, event_data))
  - 方法说明：事件上报方法
  - 参数：元组数据，包含以下信息

    * identifier：阿里云 IoT 事件标识符

    * event_data：告警数据

  - 使用示例：




```
import logging #导入logging模块

def main(data, wizard_api): #定义事件上报主函数
    event_data = {"ErrorCode": 0} #根据事件的输出参数定义事件数据
    wizard_api.thing_trigger_event(("event_test", event_data)) #调用wizard_
↪api中的thing_trigger_event方法上报数据
```

- thing_answer_service(identifier, request_id, code, params)

  - 方法说明：服务响应方法

  - 参数：

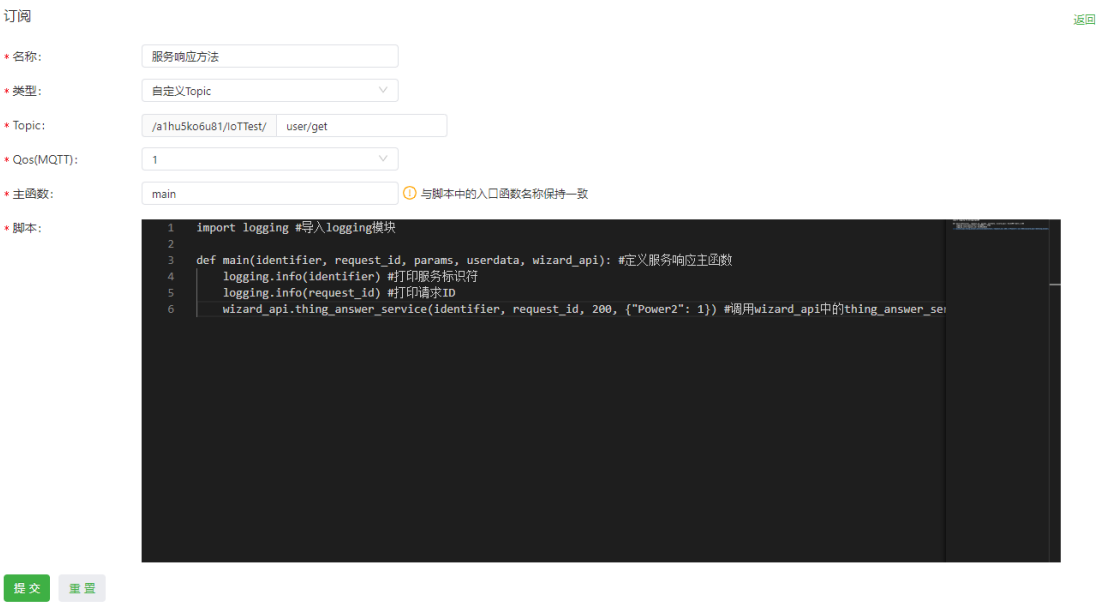
    * identifier：服务标识符

    * request_id：请求 ID

    * code：响应代码，代码说明见设备端通用 code

    * params：服务调用参数

  - 使用示例：




```
import logging #导入logging模块

def main(identifier, request_id, params, userdata, wizard_api):
↪#定义服务响应主函数
    logging.info(identifier) #打印服务标识符
    logging.info(request_id) #打印请求ID
    wizard_api.thing_answer_service(identifier, request_id, 200, {"Power2":␣
↪1}) #调用wizard_api中的thing_answer_service方法响应服务
```

1.4 AWS IoT 使用说明

AWS IoT 可在连接了 Internet 的设备（如传感器、致动器、嵌入式微控制器或智能设备）与 AWS 云之间提供安全的双向通信。这样，您便能从多台设备收集遥测数据，然后存储和分析数据。

为便于用户实现设备与 AWS IoT 的对接，边缘计算网关 InGateway902（以下简称 IG902）提供 Device Supervisor App（以下简称 Device Supervisor）对接 AWS IoT。本文档将以 IG902 为例为你说明如何实现 Device Supervisor 与 AWS IoT 的业务数据上报和配置数据下发，关于 AWS 的使用限制请参考AWS 服务配额。

- 先决条件
- *1.* 环境准备
  - *1.1 AWS IoT* 配置
    - *1.1.1* 创建事物
    - *1.1.2* 创建策略
    - *1.1.3* 配置证书

1.4.1 先决条件

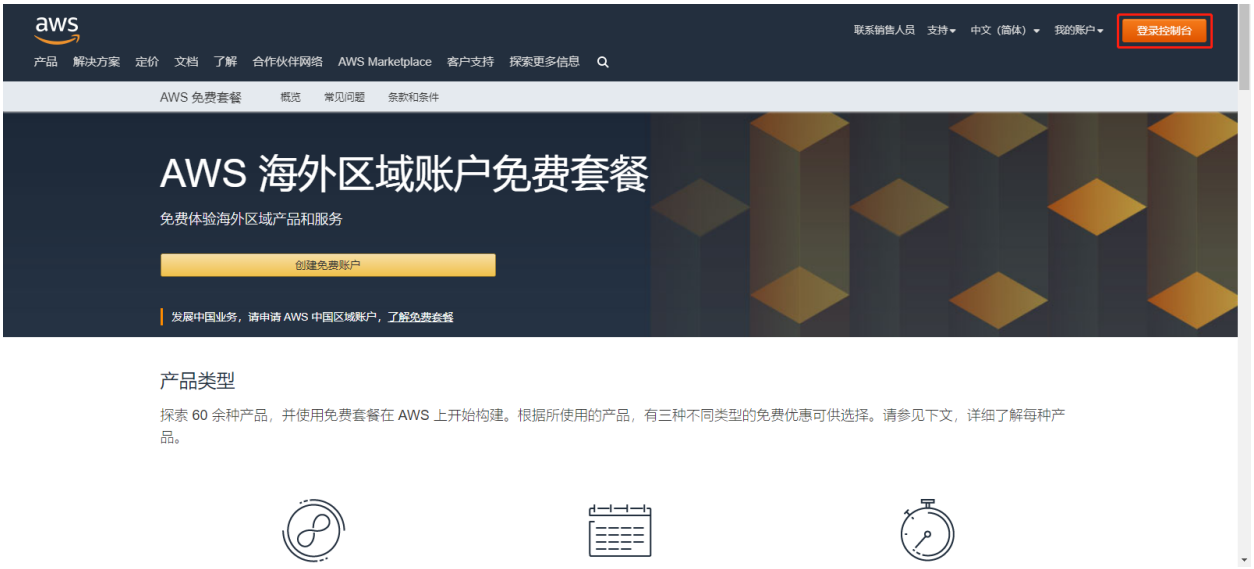
- AWS 云平台账号
- 边缘计算网关 IG501/IG902
  - 固件版本
    - IG902：`IG9-V2.0.0.r12754` 及以上
    - IG501：`IG5-V2.0.0.r12884` 及以上
  - SDK 版本
    - IG902：`py3sdk-V1.4.0_Edge-IG9` 及以上
    - IG501：`py3sdk-V1.4.0_Edge-IG5` 及以上
  - App 版本：`device_supervisor-V1.2.5` 及以上

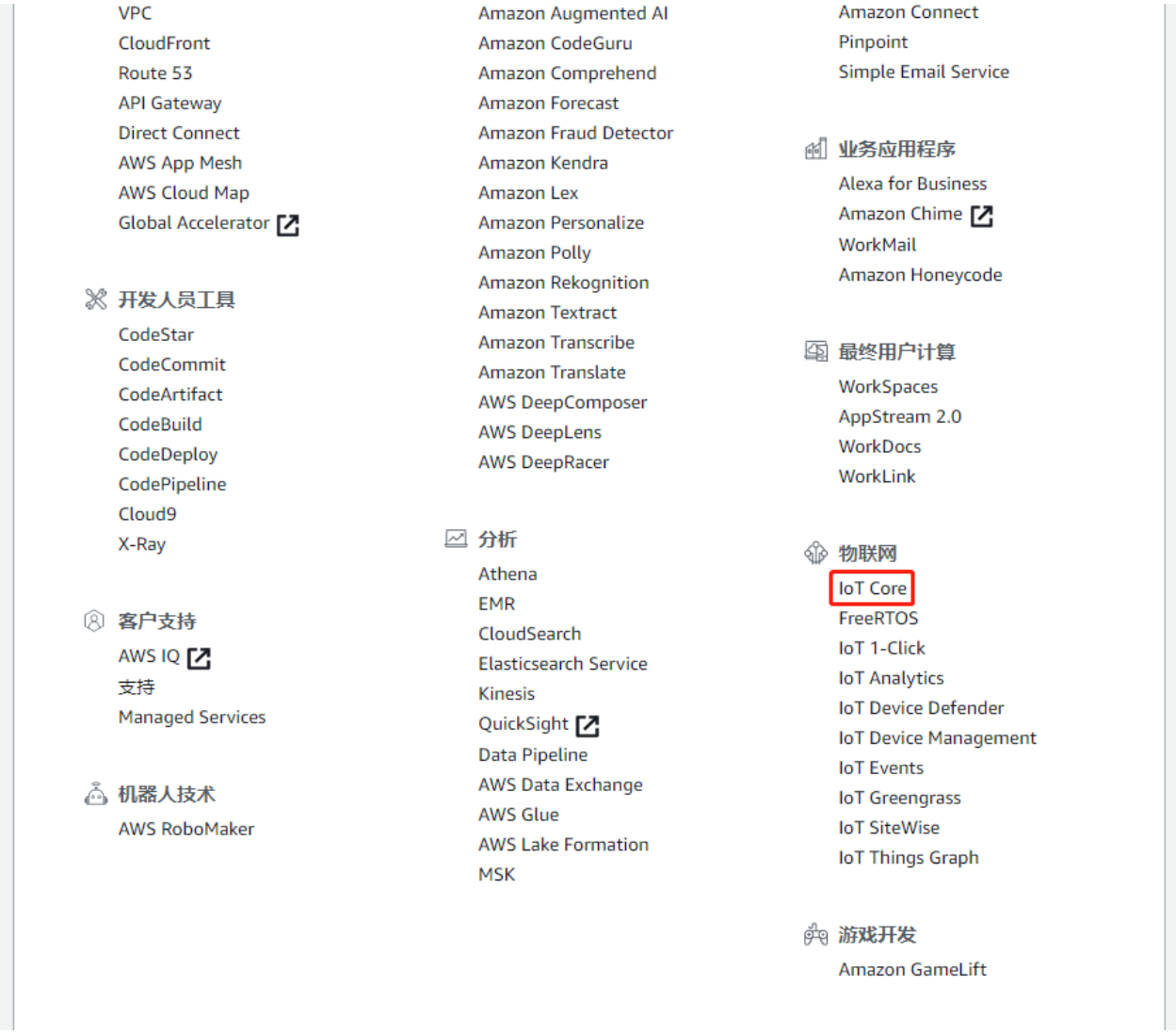
1.4.2 1. 环境准备

- *1.1 AWS IoT* 配置
- *1.2* 边缘计算网关配置

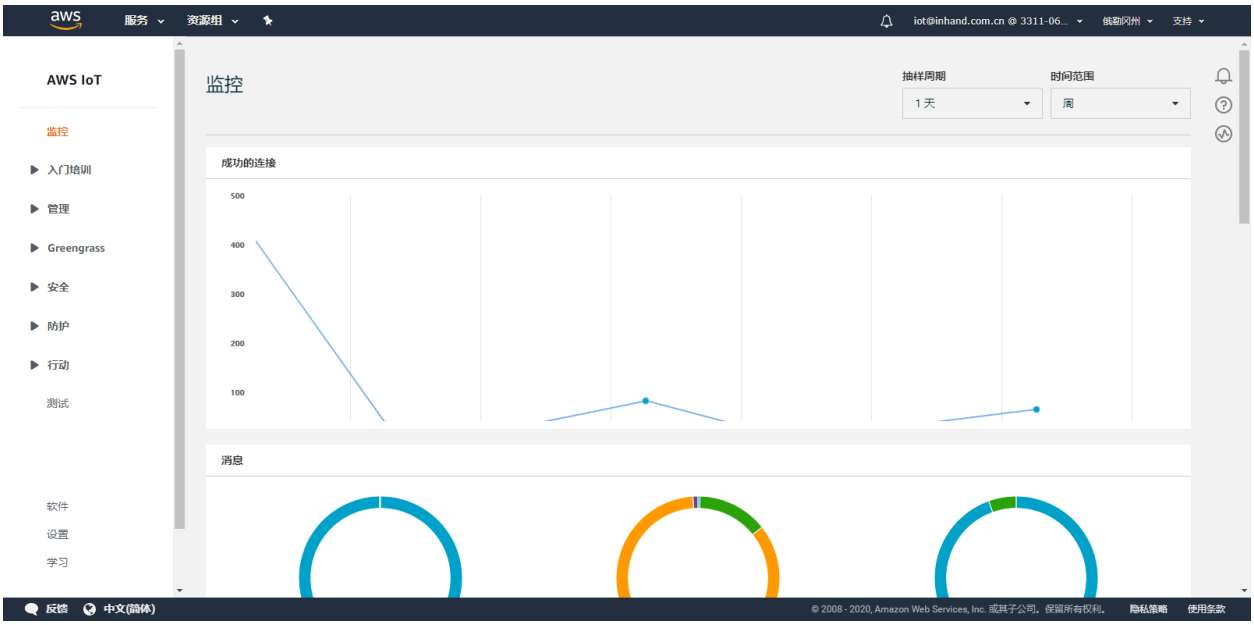
1.1 AWS IoT 配置

如果你已经在 AWS 的 IoT 控制台中配置了相应的事物、策略和证书，可以直接查看下一节 *1.2* 边缘计算网关配置。否则请按照如下流程配置 IoT 控制台。访问 AWS 官网 https://aws.amazon.com/ 并登录控制台，中国区域的用户可以使用 AWS 中国区域的 AWS 服务，地址为 https://cn-northwest-1.console.amazonaws.cn。需要注意的是，两者的数据相互独立。登录控制台后选择“IoT Core”，使用中国 AWS 服务时选择“AWS IoT”。



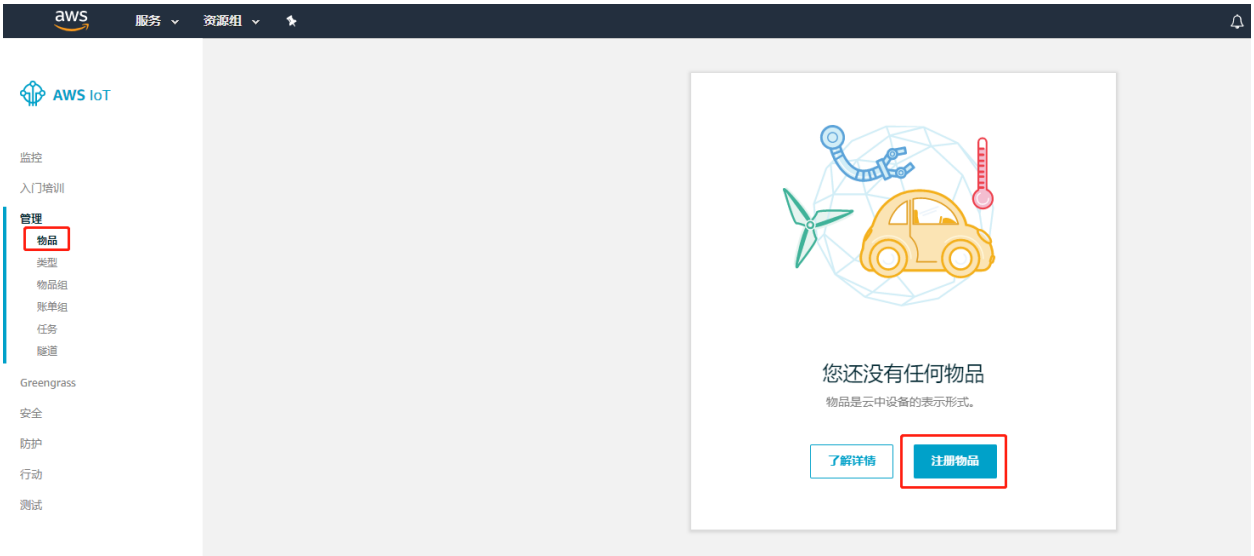


登录“IoT Core”后页面如下所示：



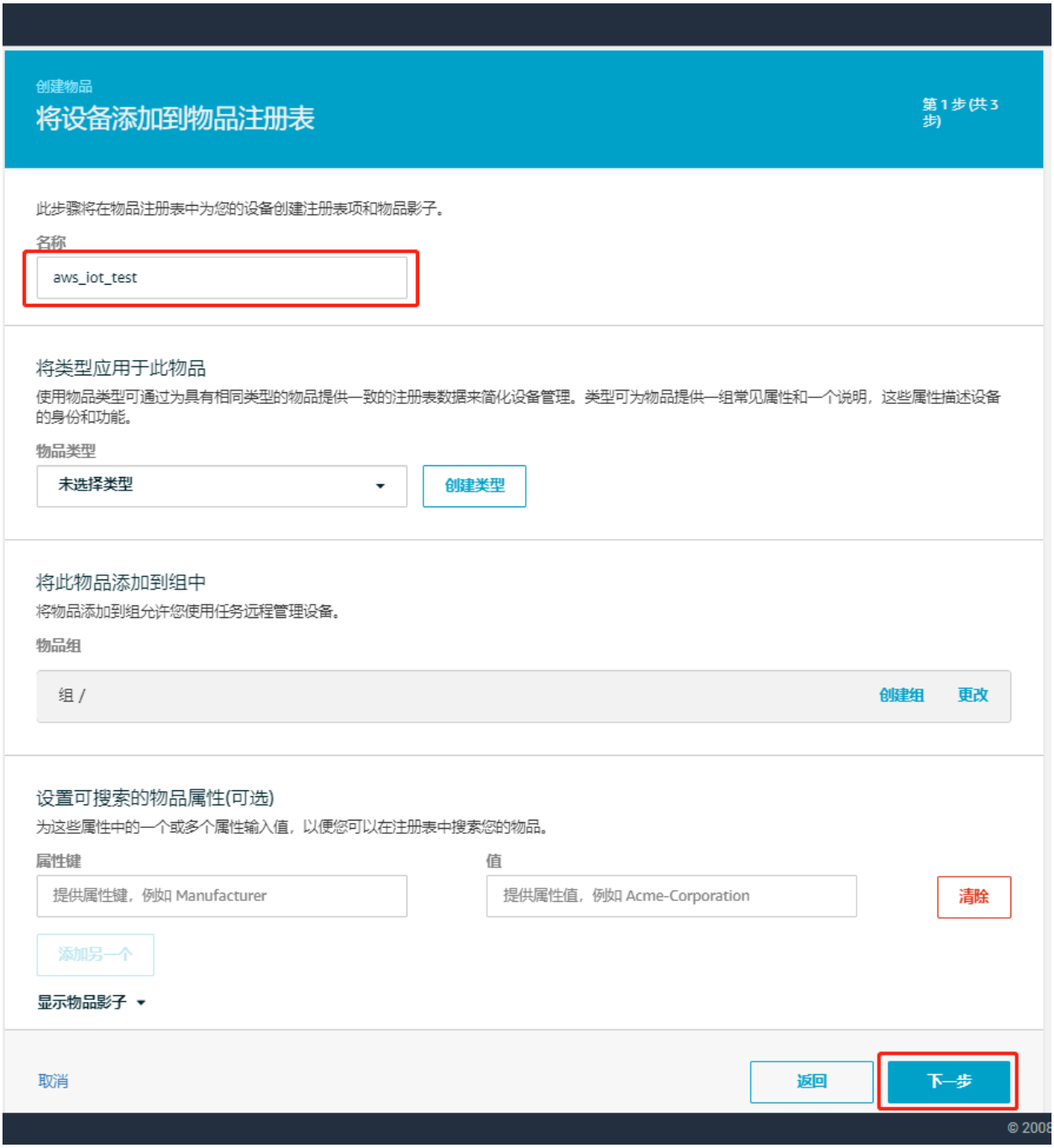
1.1.1 创建事物

选择“管理 > 物品”进入“物品”页面点击“注册物品”或“创建”以创建物品。

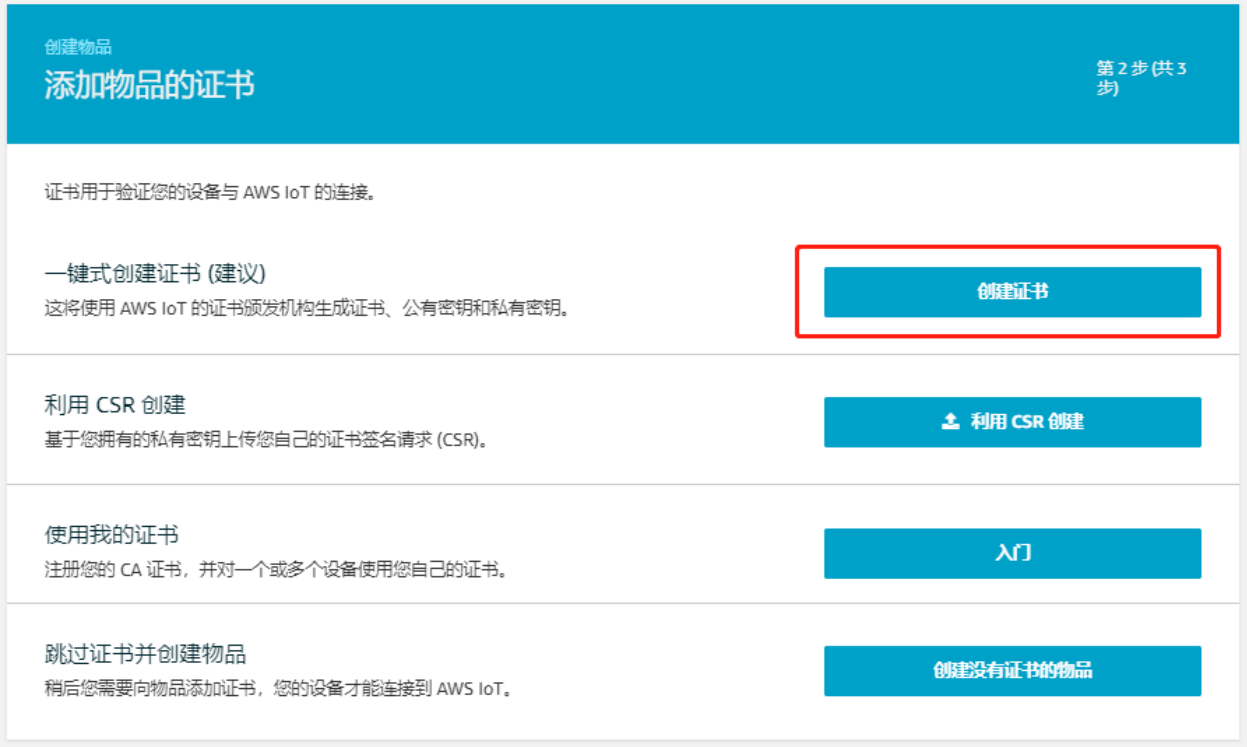


选择“创建单个物品”。

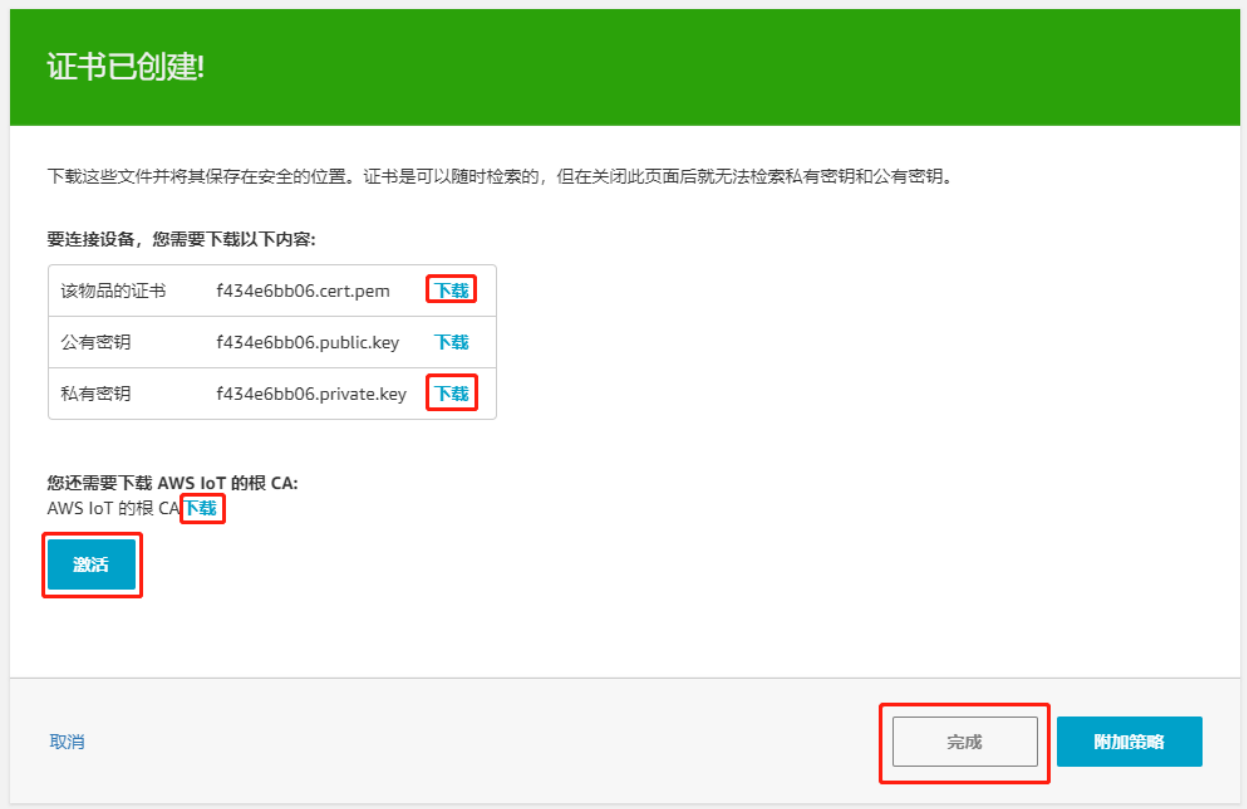
随后为物品设置名称，如 `aws_iot_test`，其余项使用默认配置即可，配置完成后点击“下一步”。


创建物品
将设备添加到物品注册表
第 1 步 (共 3 步)
此步骤将在物品注册表中为您的设备创建注册表项和物品影子。
名称
aws_iot_test
将类型应用于此物品
使用物品类型可通过为具有相同类型的物品提供一致的注册表数据来简化设备管理。类型可为物品提供一组常见属性和一个说明，这些属性描述设备的身份和功能。
物品类型
未选择类型
创建类型
将此物品添加到组中
将物品添加到组允许您使用任务远程管理设备。
物品组
组 /
创建组
更改
设置可搜索的物品属性(可选)
为这些属性中的一个或多个属性输入值，以便您可以在注册表中搜索您的物品。
属性键
提供属性键，例如 Manufacturer
值
提供属性值，例如 Acme-Corporation
清除
添加另一个
显示物品影子
取消
返回
下一步

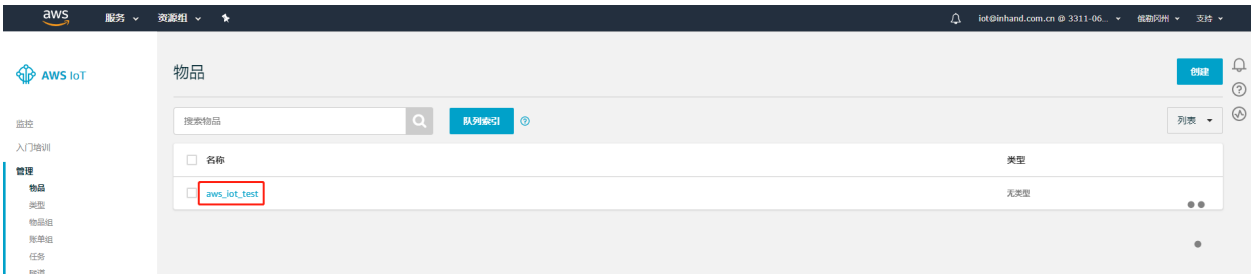
点击“创建证书”。



证书创建完成后需要下载物品的证书、私有秘钥和 AWS IoT 的根 CA（下载根 CA 证书时建议下载 `Amazon Root CA 1` 或 `Starfield 根 CA 证书`，目前不支持“Amazon Root CA 3”证书）并激活证书，之后点击“完成”以完成创建物品。

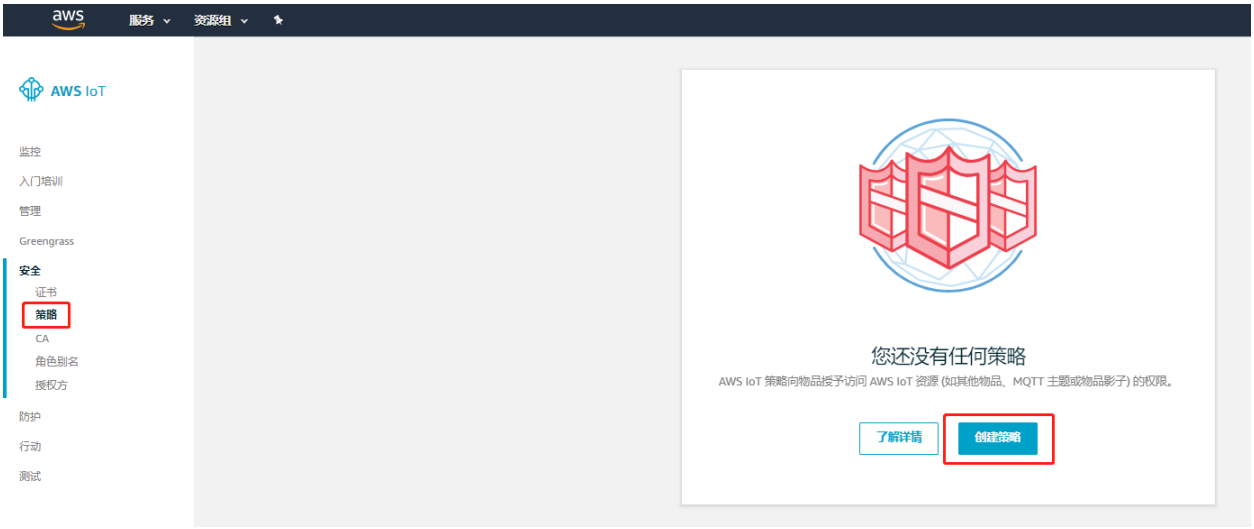

证书已创建!
下载这些文件并将其保存在安全的位置。证书是可以随时检索的，但在关闭此页面后就无法检索私有密钥和公有密钥。
要连接设备，您需要下载以下内容:
该物品的证书
f434e6bb06.cert.pem
下载
公有密钥
f434e6bb06.public.key
下载
私有密钥
f434e6bb06.private.key
下载
您还需要下载 AWS IoT 的根 CA:
AWS IoT 的根 CA
下载
激活
取消
完成
附加策略

物品创建成功后如下图所示：



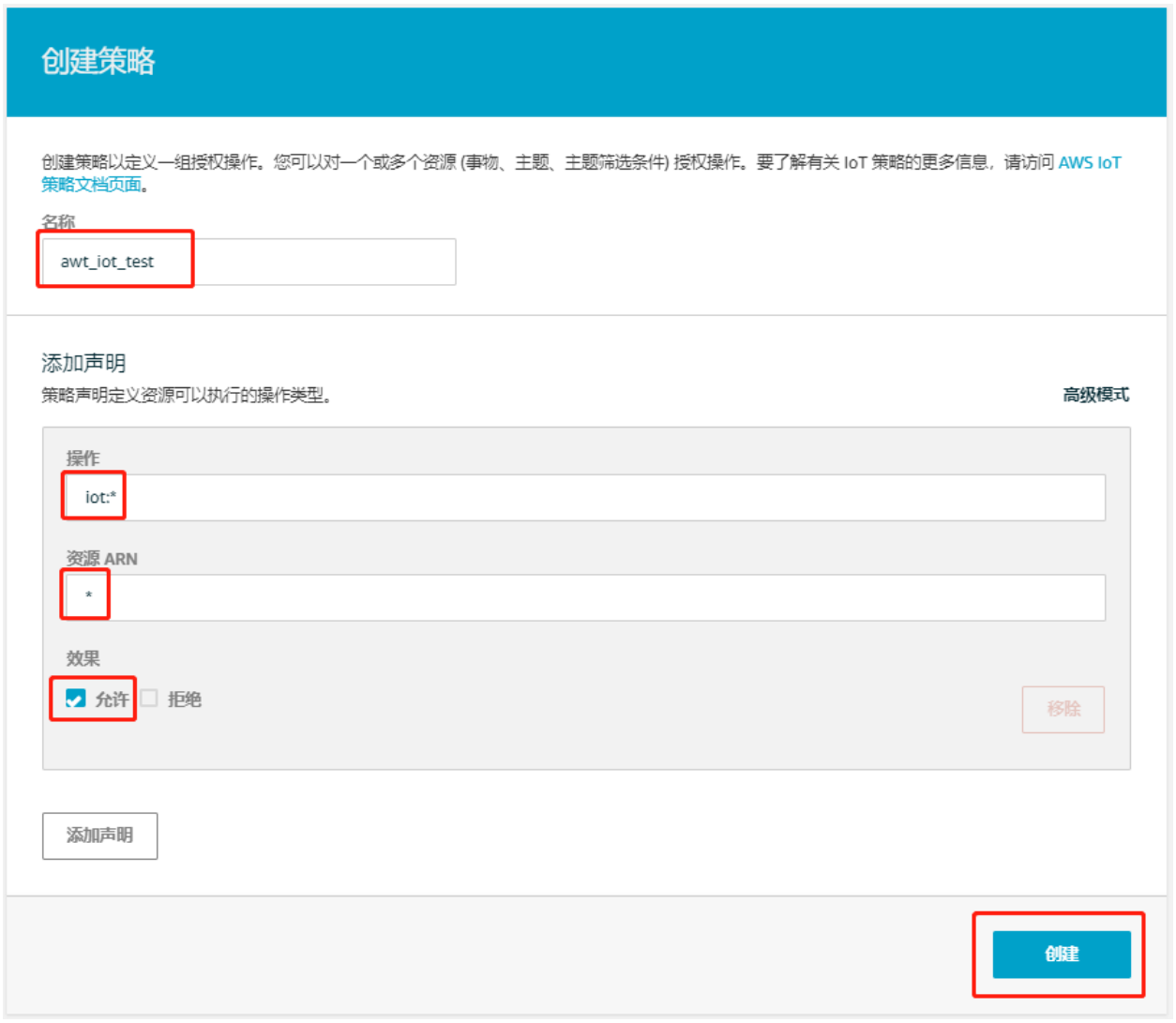
1.1.2 创建策略

选择“安全 > 策略”进入“策略”页面点击“创建策略”或“创建”以创建策略。

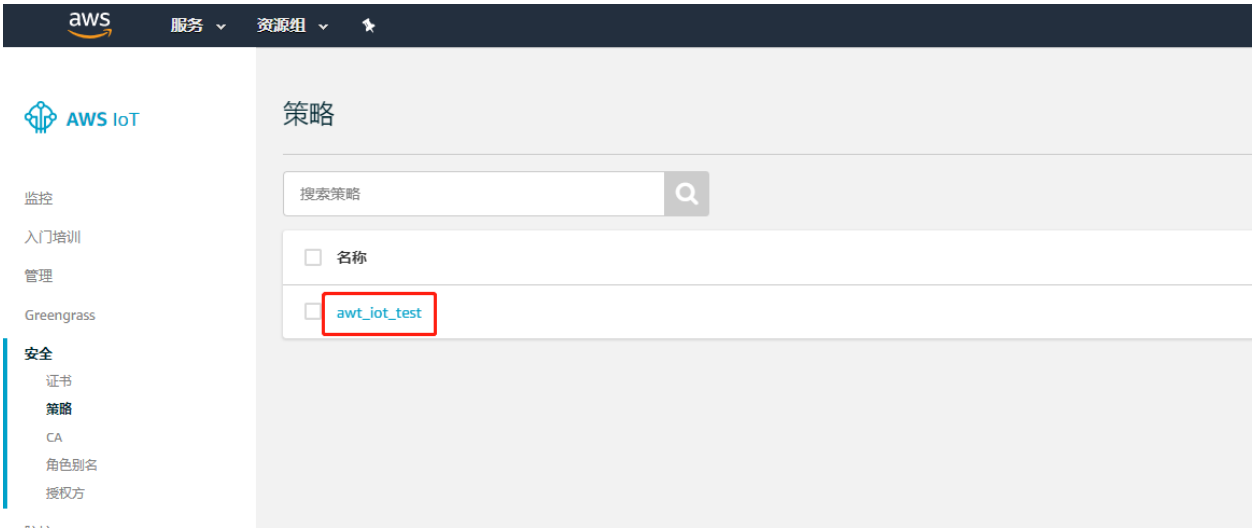


在“创建策略”页面输入策略名称并参考下列配置来配置策略（该策略使所有客户端都能连接到 AWS IoT），配置完成后点击“创建”完成策略创建。

- “操作”中填入 `iot:*`
- “资源 ARN”中填入 `*`
- “效果”选择“允许”

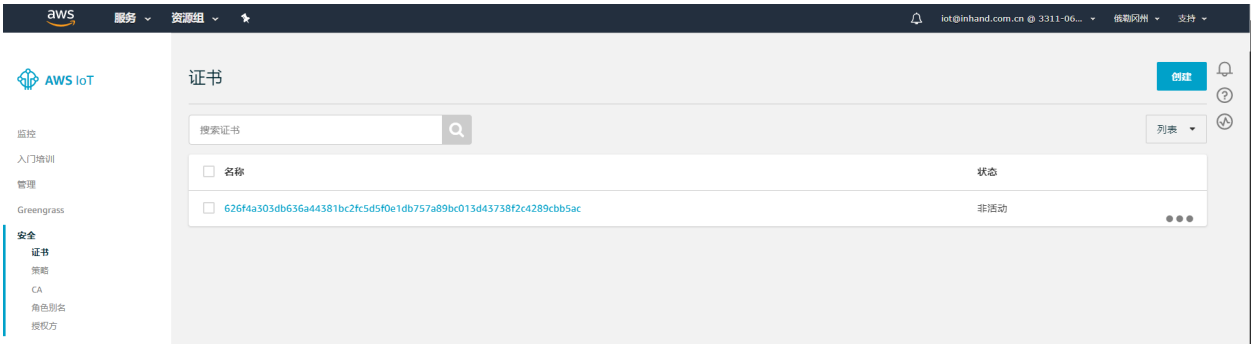


策略创建成功后如下图所示：



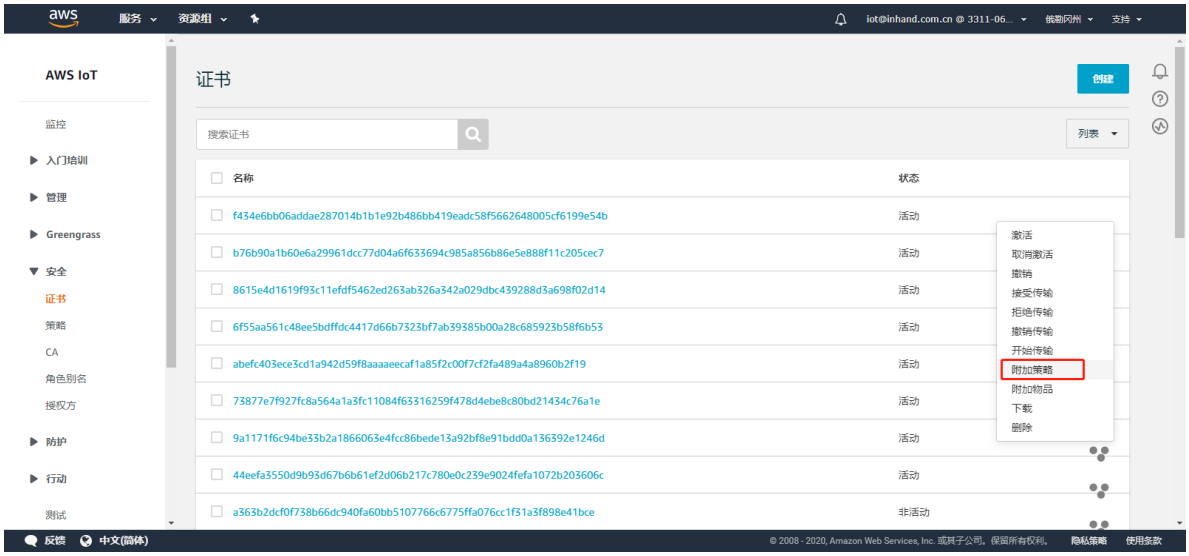
1.1.3 配置证书

选择“安全 > 证书”进入“证书”页面，如下图所示：



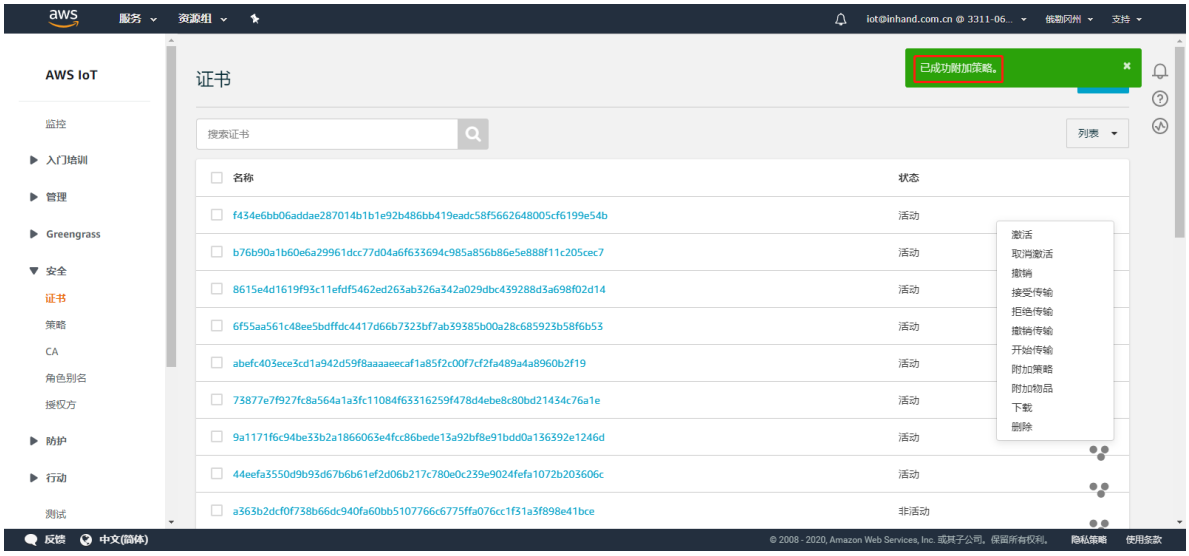
- 附加策略

  点击证书右侧的“…”并选择“附加策略”为证书附加策略。



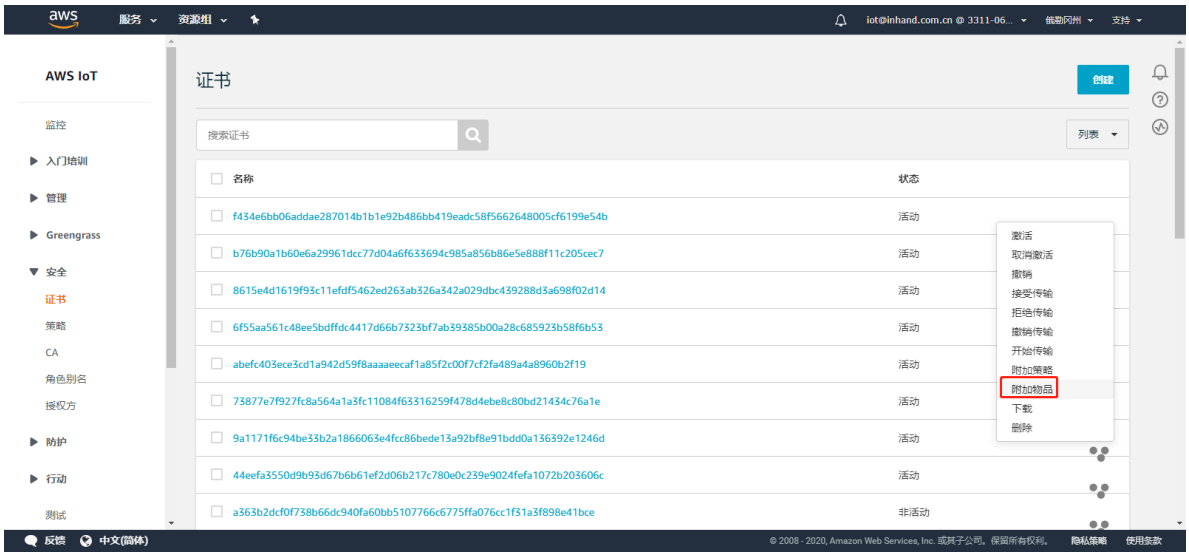
随后选择相应的策略并点击“附加”。

策略附加成功如下图所示：



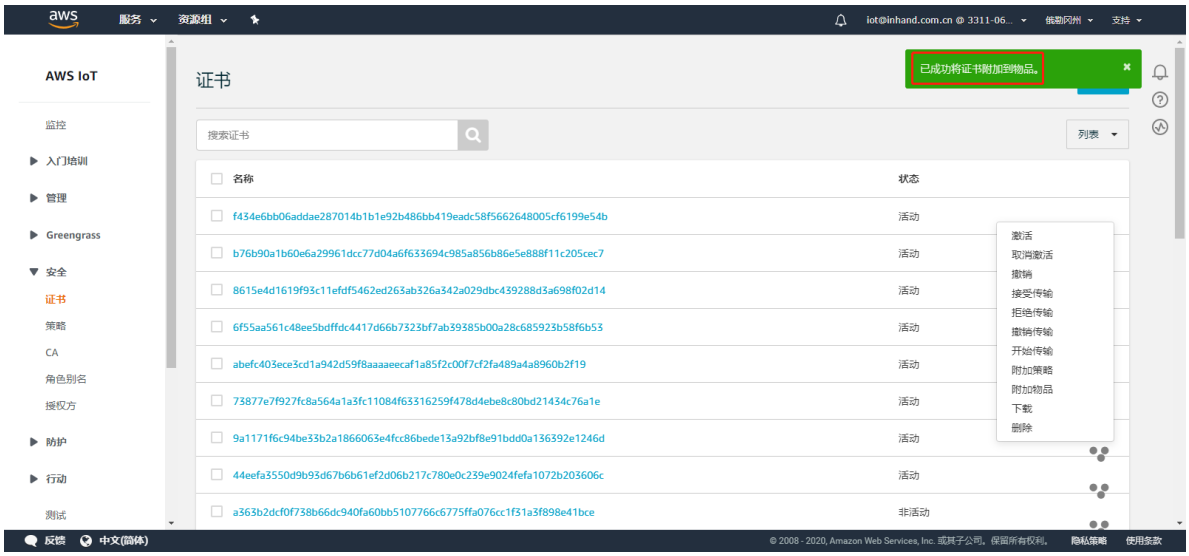
- 附加物品

  点击证书右侧的“…”并选择“附加物品”为证书附加物品。



随后选择相应的物品并点击“附加”。

物品附加成功如下图所示：



至此，完成了 AWS IoT 环境准备。

1.2 边缘计算网关配置

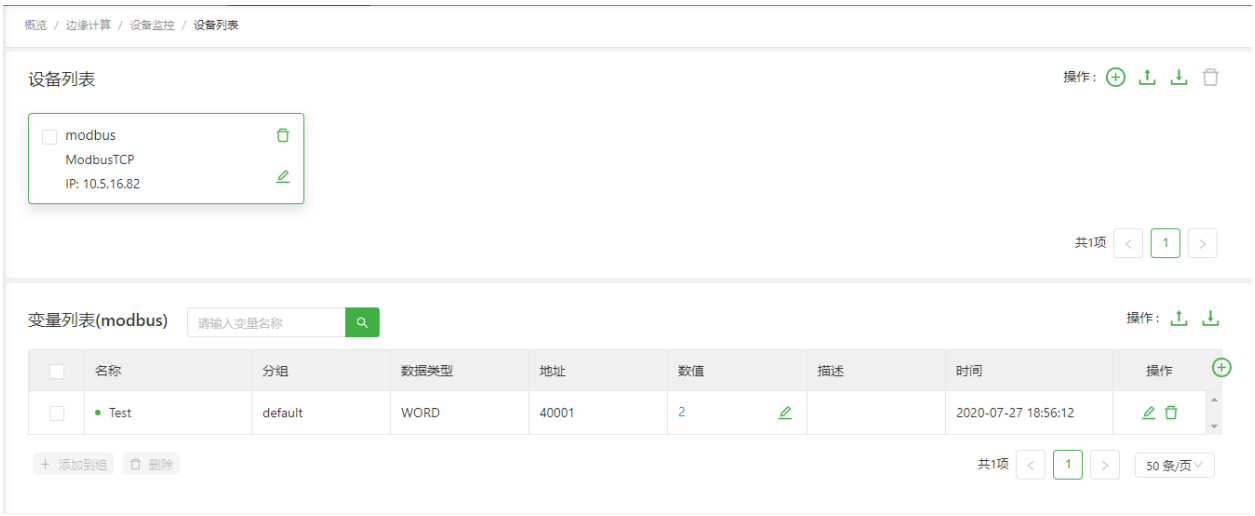
- *1.2.1* 基础配置
- *1.2.2* 配置数据采集

1.2.1 基础配置

- 如何配置 IG902 联网、更新软件版本等操作请参考IG902 快速使用手册。
- 如何配置 IG501 联网、更新软件版本等操作请参考IG501 快速使用手册。

1.2.2 配置数据采集

Device Supervisor 详细的基础数据采集配置见Device Supervisor App 用户手册。本文档的数据采集配置如下：



1.4.3 2. 发布和订阅消息

- *2.1 连接 AWS IoT*
- *2.2 发布消息到 AWS IoT*
- *2.3 订阅 AWS IoT 的消息*

以美元符号 ($) 开头的主题保留供 AWS IoT 使用。你可以在允许的情况下订阅和发布到这些保留的主题；但是，你不能创建以美元符号开头的新主题。对保留的主题执行不受支持的发布或订阅操作可能会导致连接终止。AWS IoT 的保留主题见保留的主题。

2.1 连接 AWS IoT

进入 IG902 的“边缘计算 > 设备监控 > 云服务”页面，勾选“启用云服务”并选择类型为“AWS IoT”。示例配置如下：

各项参数说明如下：

- 类型：连接 AWS IoT 时，选择“AWS IoT”
- 服务器地址：AWS IoT 的终端节点地址，可从 AWS IoT 的“设置”页面获取。使用“VeriSign Class 3 Public Primary G5 根 CA 证书”时，请删除地址中的“-ats”

- MQTT 客户端 ID：任一唯一 ID
- 物品的证书：导入创建物品时下载的物品证书或自定义证书
- 私有密钥：导入创建物品时下载的私有秘钥或自定义私有秘钥
- CA 证书：导入用于服务器身份验证的 CA 证书，你可以从这里下载相应的 CA 证书（建议使用 Amazon Root CA 1 或 Starfield 根 CA 证书）。目前不支持“Amazon Root CA 3”证书
- 其余项使用默认配置即可

2.2 发布消息到 AWS IoT

- 步骤 1：配置发布消息

  在“云服务 > 消息管理”中添加一条发布消息，配置如下：

脚本如下：

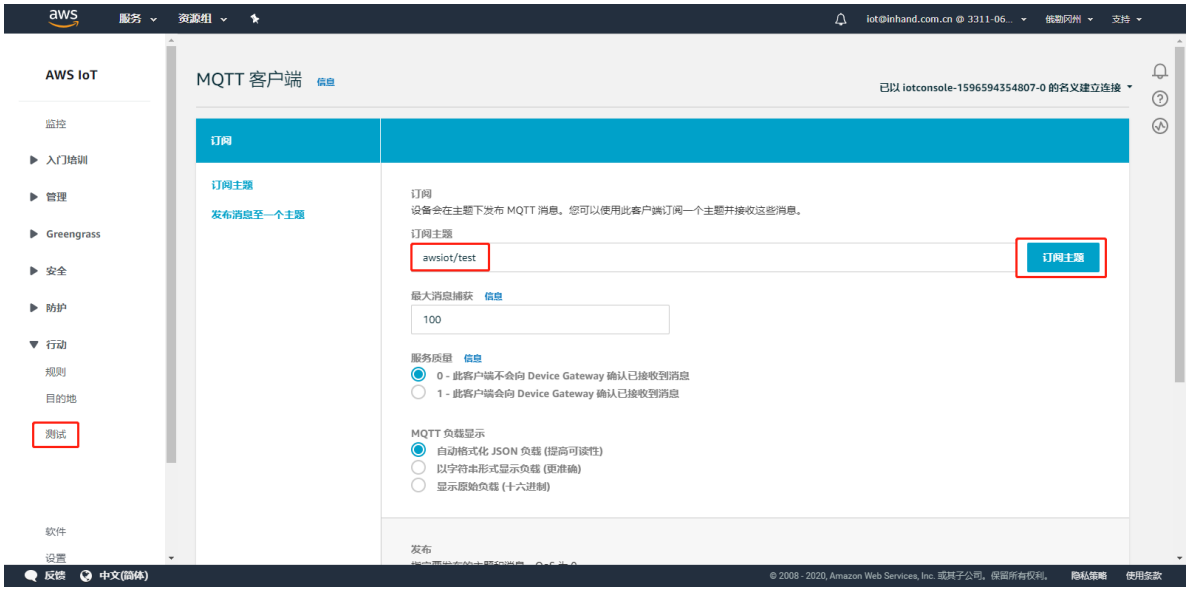
```
import logging
from datetime import datetime
"""
在网关中打印日志通常有两种办法。
1.import logging: 使用logging.
↪info(XXX)打印日志，该方法的日志显示不受全局参数页面中的日志等级参数控制。
2.from common.Logger import logger: 使用logger.
↪info(XXX)打印日志，该方法的日志显示受全局参数页面中的日志等级参数控制。
"""

def vars_upload_test(data_collect, wizard_api): #定义发布主函数
    value_list = [] #定义数据列表
    for device, val_dict in data_collect['values'].items():
↪#遍历values字典，该字典中包含设备名称和设备下的变量数据
        value_dict = { #自定义数据字典
                       "Device": device,
                       "timestamp": data_collect["timestamp"],
                       "Data": {}
                       }
        for id, val in val_dict.items(): #遍历变量数据，为Data字典赋值
            value_dict["Data"][id] = val["raw_data"]
        value_list.append(value_dict) #依次将value_dict添加到value_list中
    logging.info(value_list) #在App日志中打印value_list，数据格式为[{'Device':
↪'S7-1200', 'timestamp': 1589538347.5604711, 'Data': {'Test1': False, 'Test2':␣
↪12}}]
    return value_list #将value_
↪list发送给App，App将自行按照采集时间顺序上传至MQTT服务器。如果发送失败则缓存数据等待连接恢复后
```

发布消息配置参数说明如下：

- 名称：用户自定义发布名称
- 主题：发布主题，与 MQTT 服务器订阅的主题保持一致
- Qos(MQTT)：发布 Qos，建议与 MQTT 服务器的 Qos 保持一致
  - 0：只发送一次消息，不进行重试
  - 1：最少发送一次消息，确保消息到达 MQTT 服务器
- 分组类型：发布变量数据时请选择“采集”，随后在分组中仅能选择“采集组”；发布告警数据时请选择“告警”，随后在分组中仅能选择“告警组”
- 分组：选择相应的分组后，分组下所有变量通过该发布配置将数据上传至 MQTT 服务器；可选择多个分组，当选择多个分组时，按照分组的采集间隔分别对各分组下的变量执行发布中的脚本逻辑。分组中必须包含变量，否则不会执行发布中的脚本逻辑
- 主函数：主函数名称，即入口函数名称，与脚本中的入口函数名称保持一致
- 脚本：使用 Python 代码自定义组包和处理逻辑，主函数参数包括：
  - 参数 1：同标准 MQTT-发布主函数中的参数 1
  - 参数 2：Device Supervisor 的 AWS IoT api 接口，参数说明见 *Device Supervisor 的 AWS IoT api 接口说明*

- 步骤 2：在 AWS IoT 中订阅消息

进入 AWS IoT 的“测试”页面，在“订阅主题”中输入 IG902 的发布主题，本文档为 awsiot/test。



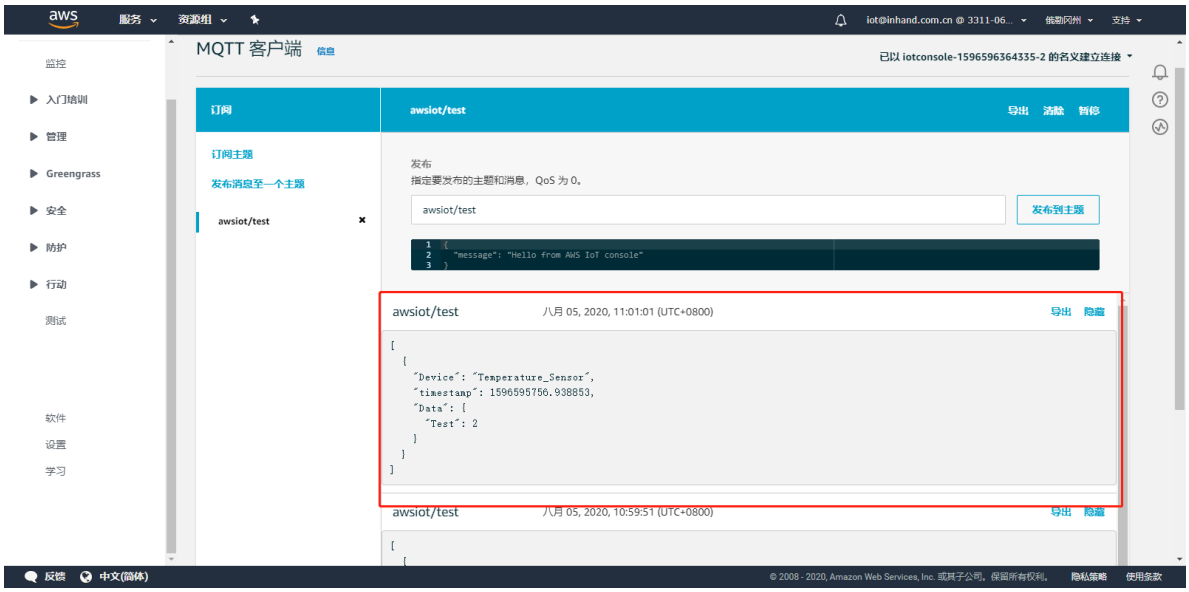
消息管理

发布

名称	类型	Topic	Qos(MQTT)	分组	主函数	操作
上报数据		awsiot/test	1	default	vars_upload_test	

- 步骤 3：查看 AWS IoT 接收的消息

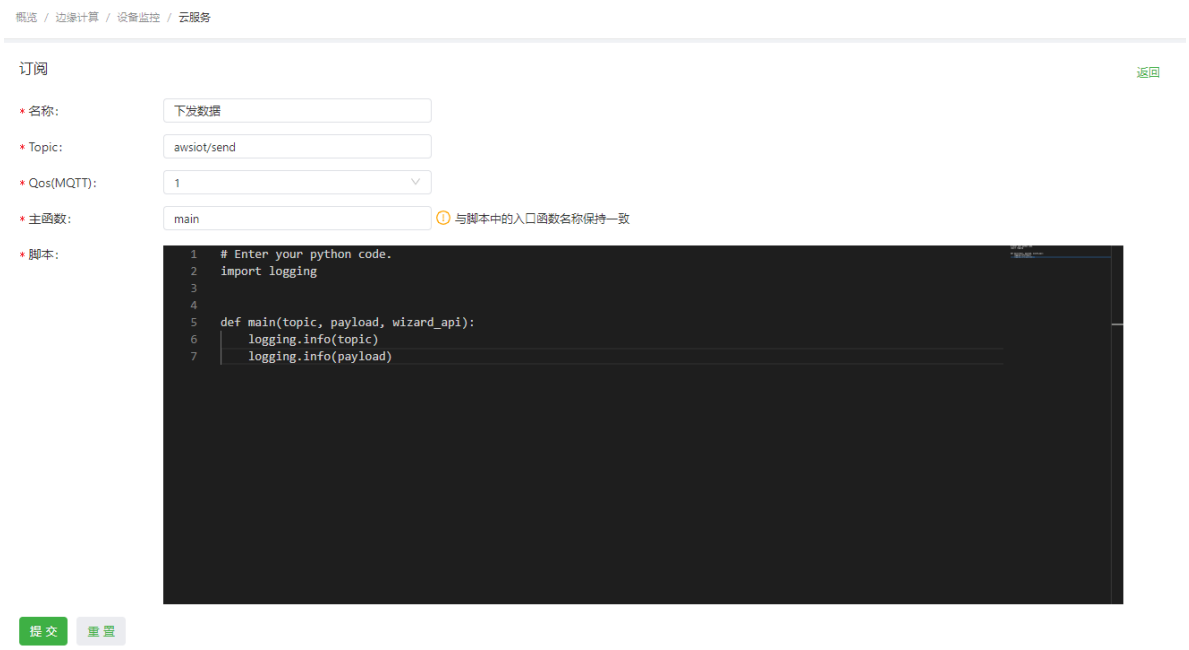
  订阅主题后，可在订阅主题页面查看收到的消息内容。



2.3 订阅 AWS IoT 的消息

- 步骤 1：配置订阅消息

  在“云服务 > 消息管理”中添加一条订阅消息，配置如下：

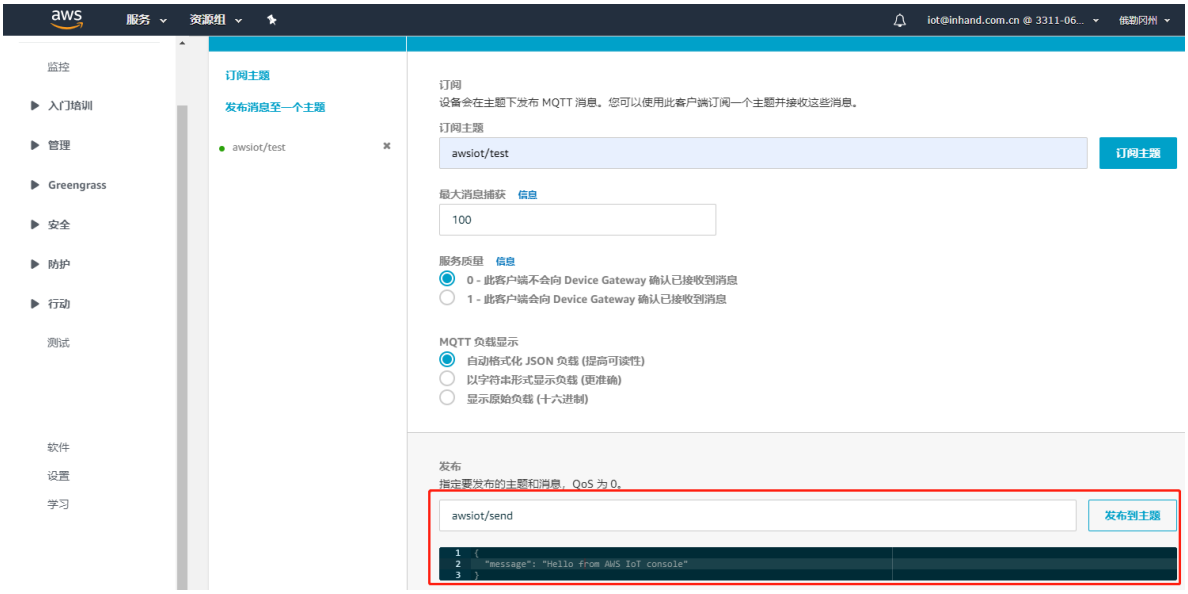


订阅消息配置参数说明如下：

- 名称：自定义订阅名称
- 主题：订阅主题，与 MQTT 服务器发布的数据主题保持一致
- Qos(MQTT)：订阅 Qos，默认为 0
- 主函数：主函数名称，即入口函数名称，与脚本中的入口函数名称保持一致
- 脚本：使用 Python 代码自定义组包和处理逻辑，自定义 Topic 的订阅主函数参数包括：
  - 参数 1：该参数为接收到的主题，数据类型为 `string`
  - 参数 2：该参数为接收到的数据，数据类型为 `string`
  - 参数 3：Device Supervisor 的 AWS IoT api 接口，参数说明见 *Device Supervisor 的 AWS IoT api 接口说明*

- 步骤 2：在 AWS IoT 中发布消息

  进入 AWS IoT 的“测试”页面，在“发布消息至一个主题”中输入 IG902 的订阅主题，本文档为 `awsiot/send`。



消息管理

发布

名称	类型	Topic	Qos(MQTT)	分组	主函数	操作
上报数据		awsiot/test	1	default	vars_upload_test	

订阅

名称	类型	Topic	Qos(MQTT)	主函数	操作
下发数据		awsiot/send	1	main	

- 步骤 3：查看 AWS IoT 发布的消息

  AWS IoT 发布消息后，可在 App 运行日志中查看收到的消息内容。

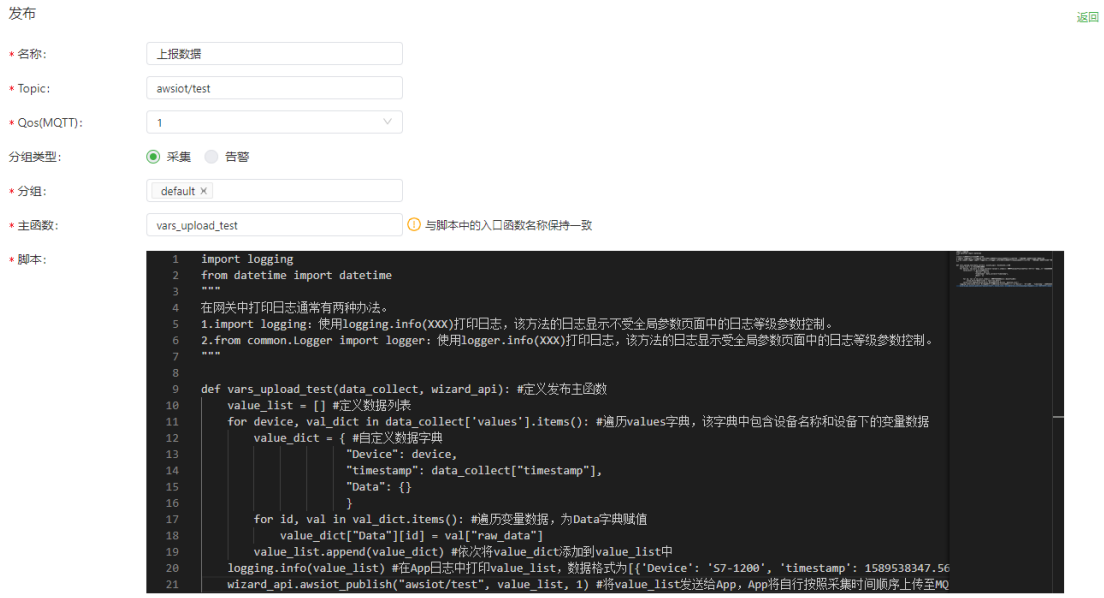
```
[2020-08-05 11:19:27,789] [INFO] [AWSIoT.py 197]: [AWSIoT]: on_cloud_subscribe. topic: awsiot/send, payload: b'{\n  "message": "Hello from AWS IoT console"\n}', qos: 0
[2020-08-05 11:19:27,791] [INFO] [<string> 6]: awsiot/send
[2020-08-05 11:19:27,792] [INFO] [<string> 7]: b'{\n  "message": "Hello from AWS IoT console"\n}'
[2020-08-05 11:19:27,793] [INFO] [AWSIoT.py 139]: [AWSIoT]: receive message, topic: awsiot/send, payload: b'{\n  "message": "Hello from AWS IoT console"\n}'
```

1.4.4 附录

Device Supervisor 的 AWS IoT api 接口说明

`wizard_api` 的基础配置方法请参考Device Supervisor 的 api 接口说明。当云服务类型为 `AWS IoT` 时，`wizard_api` 额外提供以下方法：

- `awsiot_publish(topic, payload, qos)`
  - 方法说明：数据上报方法
  - 参数
    - 参数 1：MQTT 主题，数据类型为 `string`。通过至该主题发送数据到 MQTT 服务器
    - 参数 2：需要发送的数据
    - 参数 3：qos 等级（包括 0/1 两种等级）
  - 使用示例：




```
import logging
from datetime import datetime
"""
在网关中打印日志通常有两种办法。
1.import logging: 使用logging.
↪info(XXX)打印日志，该方法的日志显示不受全局参数页面中的日志等级参数控制。
2.from common.Logger import logger: 使用logger.
↪info(XXX)打印日志，该方法的日志显示受全局参数页面中的日志等级参数控制。
"""

def vars_upload_test(data_collect, wizard_api): #定义发布主函数
    value_list = [] #定义数据列表
```

(续下页)

(接上页)

```
    for device, val_dict in data_collect['values'].items():
↪#遍历values字典，该字典中包含设备名称和设备下的变量数据
        value_dict = { #自定义数据字典
                        "Device": device,
                        "timestamp": data_collect["timestamp"],
                        "Data": {}
                        }
        for id, val in val_dict.items(): #遍历变量数据，为Data字典赋值
            value_dict["Data"][id] = val["raw_data"]
        value_list.append(value_dict) #依次将value_dict添加到value_list中
    logging.info(value_list) #在App日志中打印value_list，数据格式为[{'Device
↪': 'S7-1200', 'timestamp': 1589538347.5604711, 'Data': {'Test1': False,
↪'Test2': 12}}]
    wizard_api.awsiot_publish("awsiot/test", value_list, 1) #将value_
↪list发送给App，App将自行按照采集时间顺序上传至MQTT服务器。如果发送失败则缓存数据等待连接恢
↪
```

1.5 Azure IoT 使用说明

Azure IoT Hub（以下简称 Azure IoT）托管服务在云中进行托管，充当中央消息中心，用于 IoT 应用程序与其管理的设备之间的双向通信。可以使用 Azure IoT，将数百万 IoT 设备和云托管解决方案后端之间建立可靠又安全的通信，生成 IoT 解决方案。几乎可以将任何设备连接到 IoT Hub。

为便于用户实现设备与 Azure IoT 的对接，边缘计算网关 InGateway902（以下简称 IG902）提供 Device Supervisor App（以下简称 Device Supervisor）对接 Azure IoT。本文档将以 IG902 为例为你说明如何实现 Device Supervisor 与 Azure IoT 的业务数据上报和配置数据下发。

- 先决条件
- *1.* 环境准备
  - *1.1 Azure IoT* 配置
    - *1.1.1* 添加 *IoT Hub*
    - *1.1.2* 添加 *IoT Device*
  - *1.2* 边缘计算网关配置
    - *1.2.1* 基础配置
    - *1.2.2* 配置数据采集
- *2.* 发布和订阅消息
  - *2.1* 连接 *Azure IoT*
  - *2.2* 发布消息到 *Azure IoT*

1.5.1 先决条件

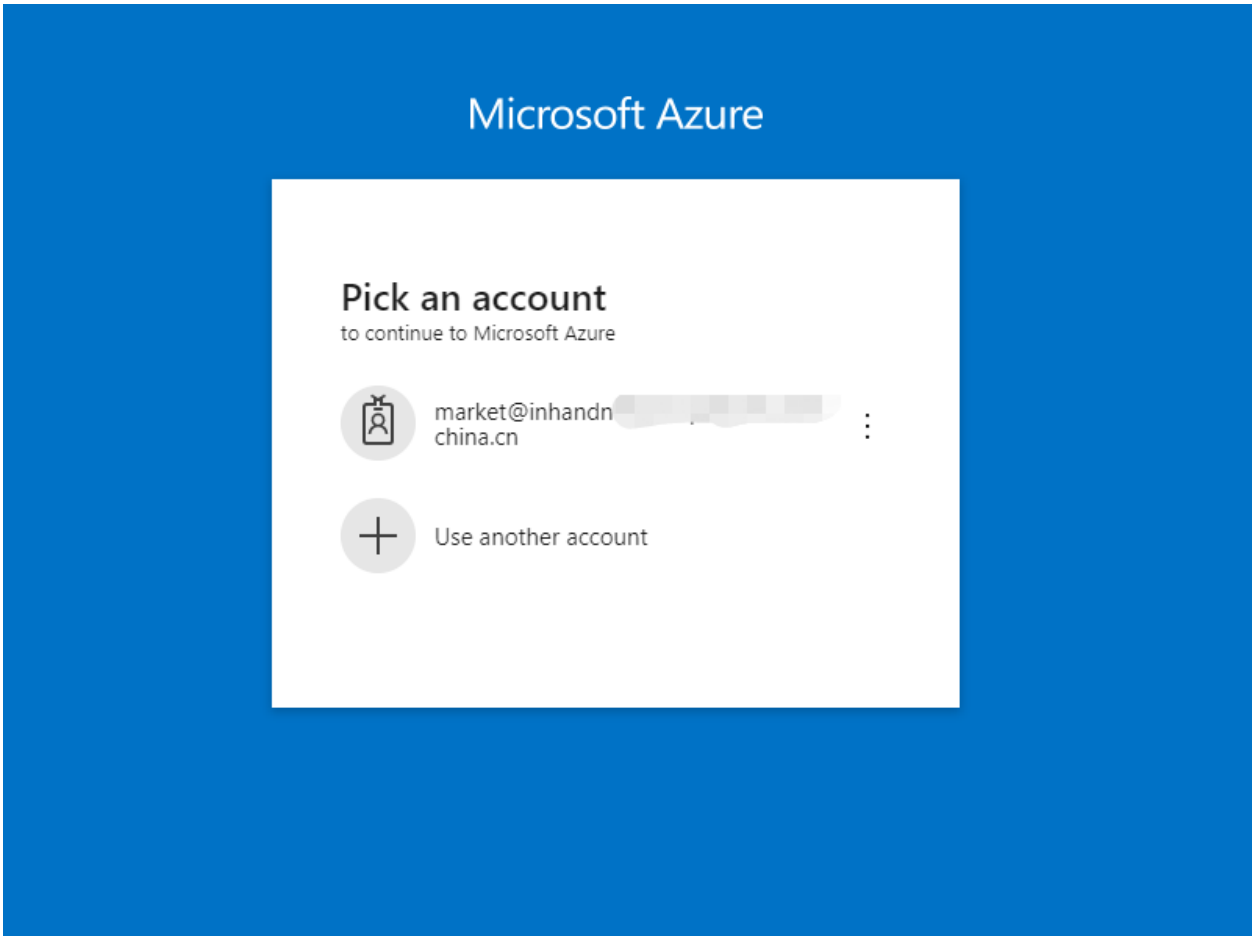
- Azure 云平台账号
- 边缘计算网关 IG501/IG902
  - 固件版本
    - IG902：`IG9-V2.0.0.r12754` 及以上
    - IG501：`IG5-V2.0.0.r12884` 及以上
  - SDK 版本
    - IG902：`py3sdk-V1.4.0_Edge-IG9` 及以上
    - IG501：`py3sdk-V1.4.0_Edge-IG5` 及以上
  - App 版本：`device_supervisor-V1.2.5` 及以上

1.5.2 1. 环境准备

- *1.1 Azure IoT* 配置
- *1.2* 边缘计算网关配置

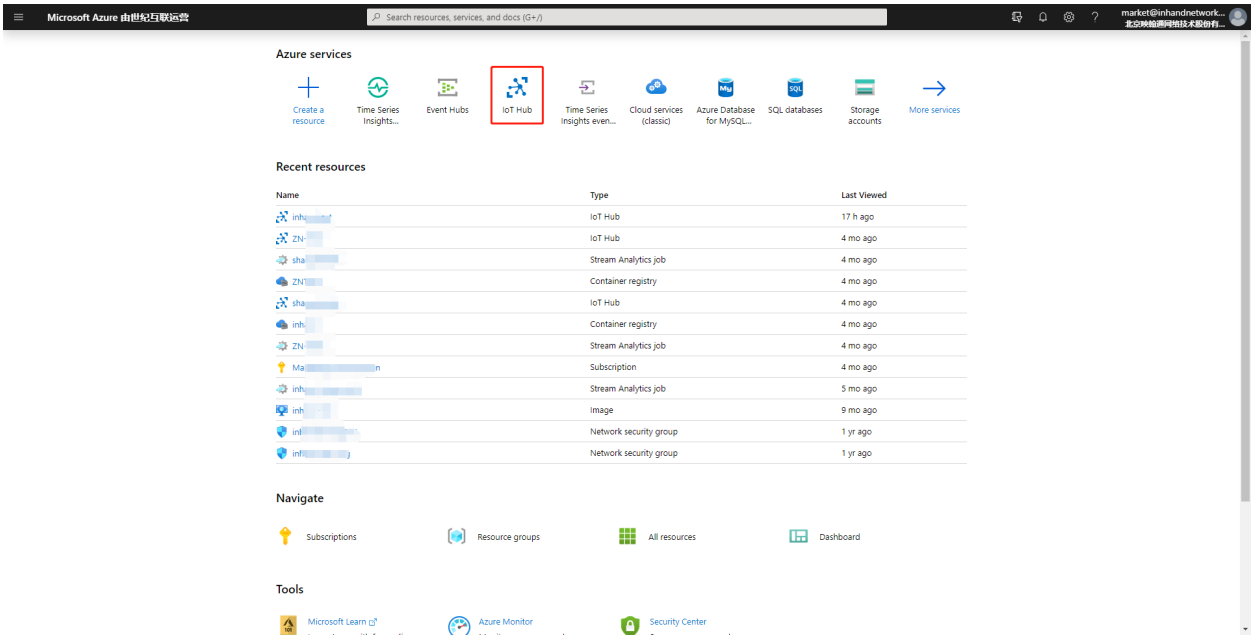
1.1 Azure IoT 配置

如果你已经在 Azure IoT 中配置了相应的 IoT Hub 和 IoT device，可以直接查看下一节 *1.2 边缘计算网关配置*。否则请按照如下流程配置 Azure IoT。访问https://portal.azure.cn/登录 Azure。

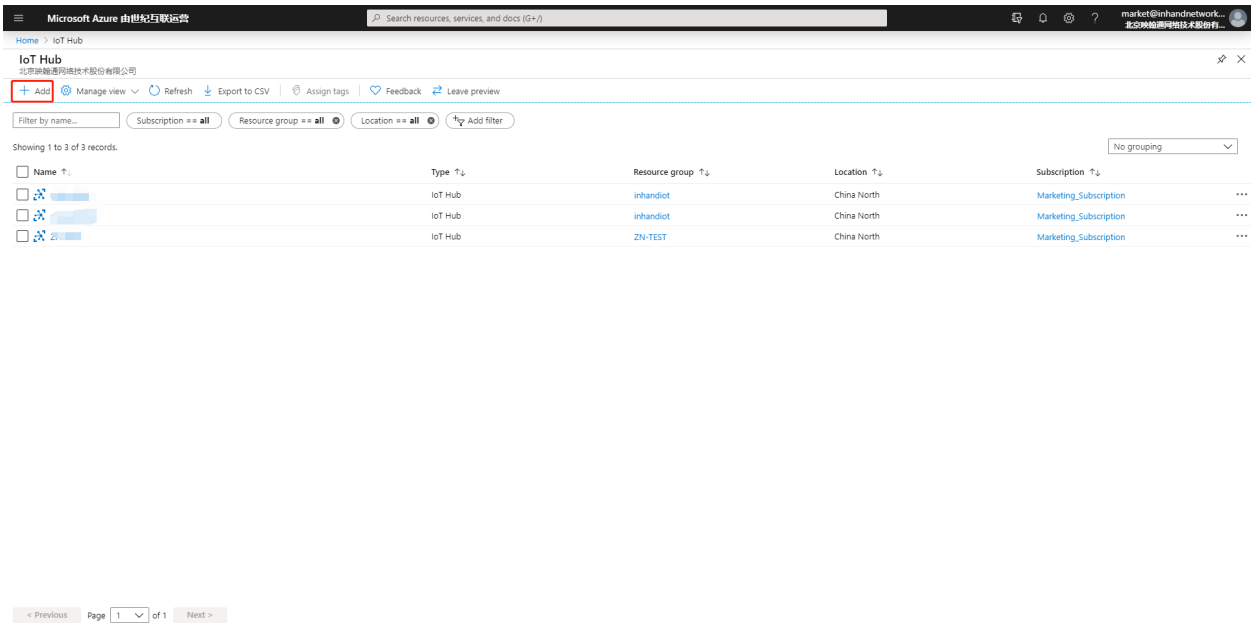


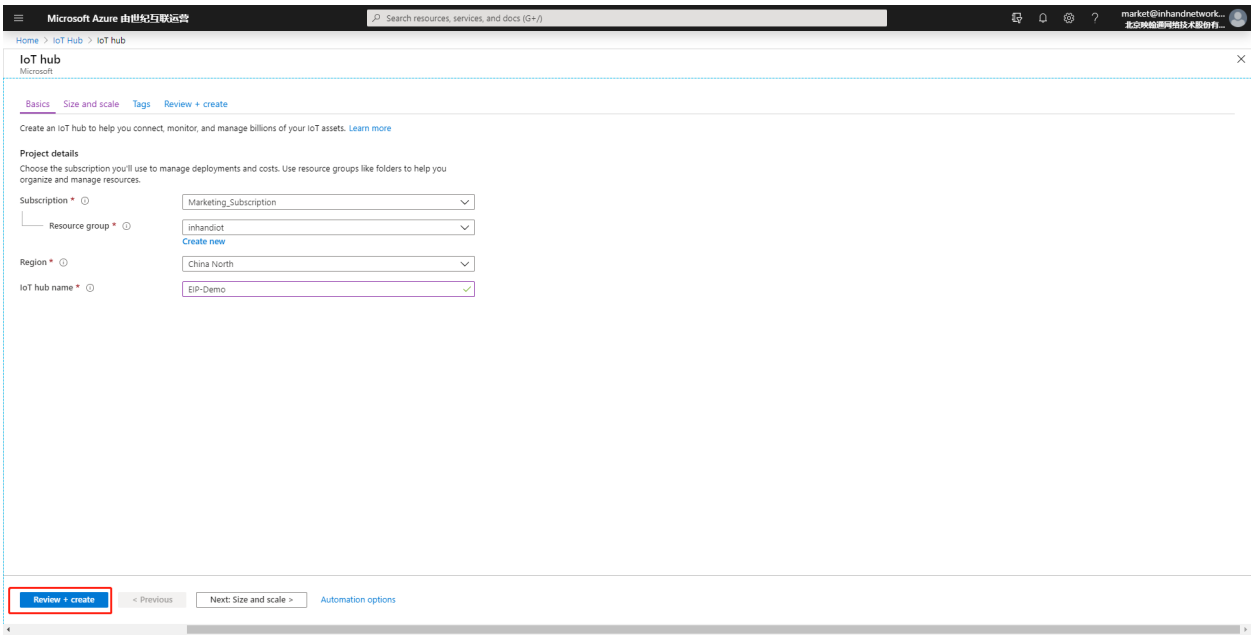
1.1.1 添加 IoT Hub

登录成功后如下图所示，选择“IoT Hub”。

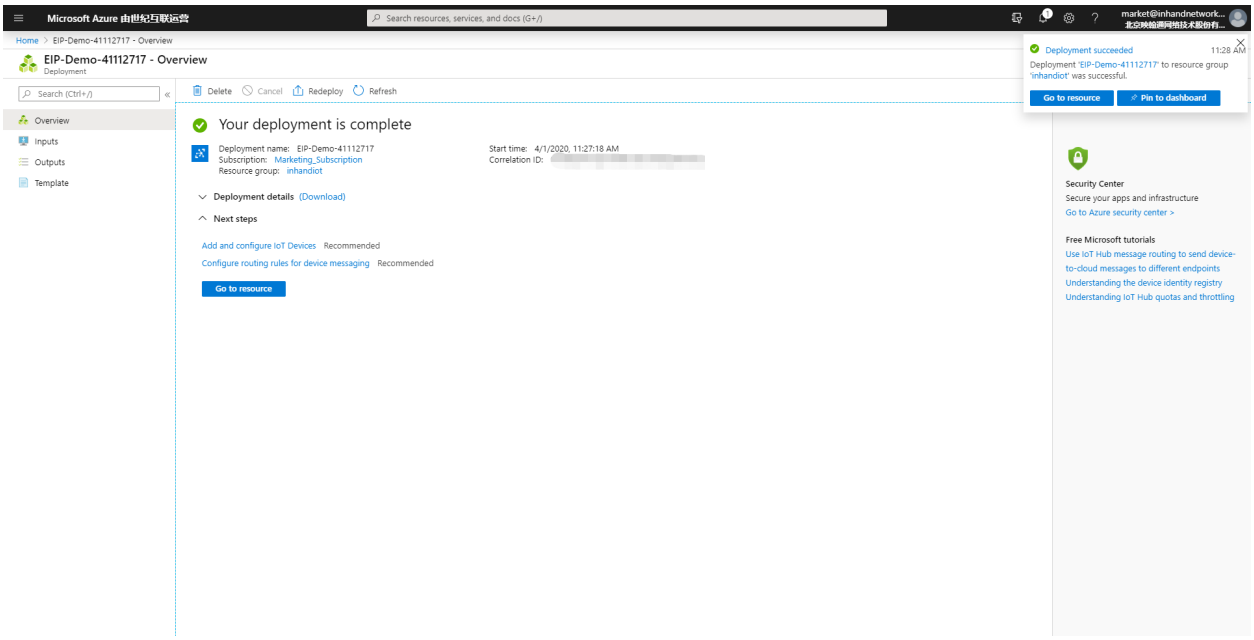


点击“Add”创建一个 IoT Hub。



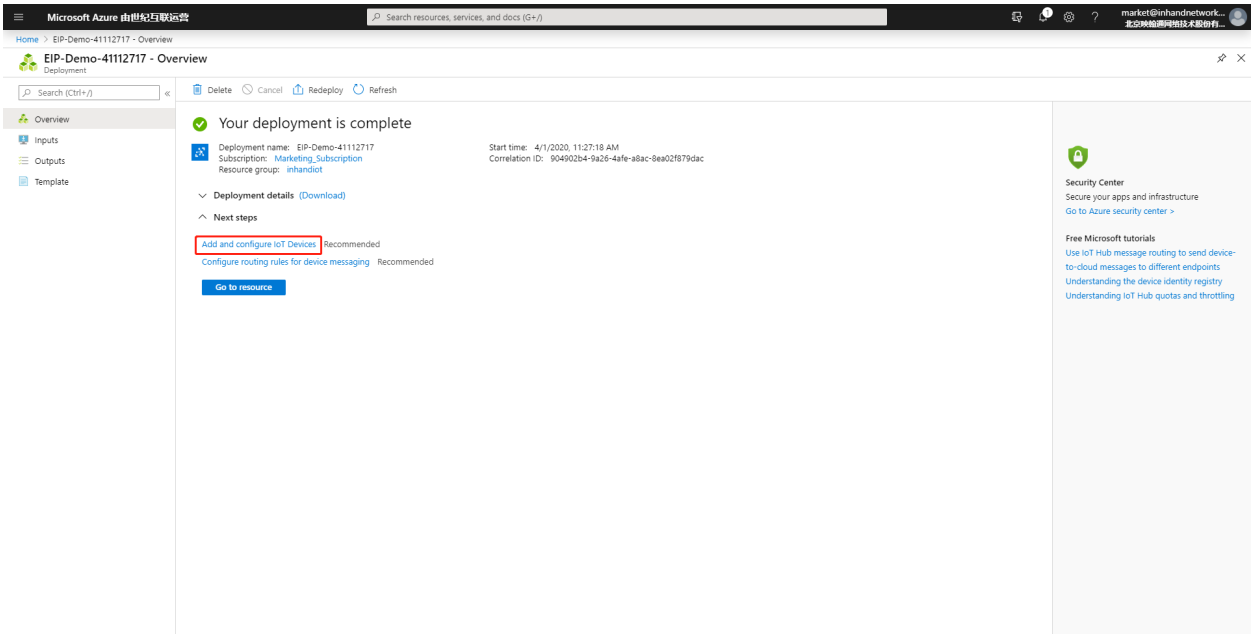

Microsoft Azure 由世纪互联运营
Search resources, services, and docs (G+/)
market@inhandnetwork...
Home > IoT Hub > IoT hub
IoT hub
Microsoft
Basics
Size and scale
Tags
Review + create
Create an IoT hub to help you connect, monitor, and manage billions of your IoT assets. Learn more
Project details
Choose the subscription you'll use to manage deployments and costs. Use resource groups like folders to help you organize and manage resources.
Subscription *
Marketing_Subscription
Resource group *
inhandiot
Create new
Region *
China North
IoT hub name *
EIP-Demo
Review + create
< Previous
Next: Size and scale >
Automation options

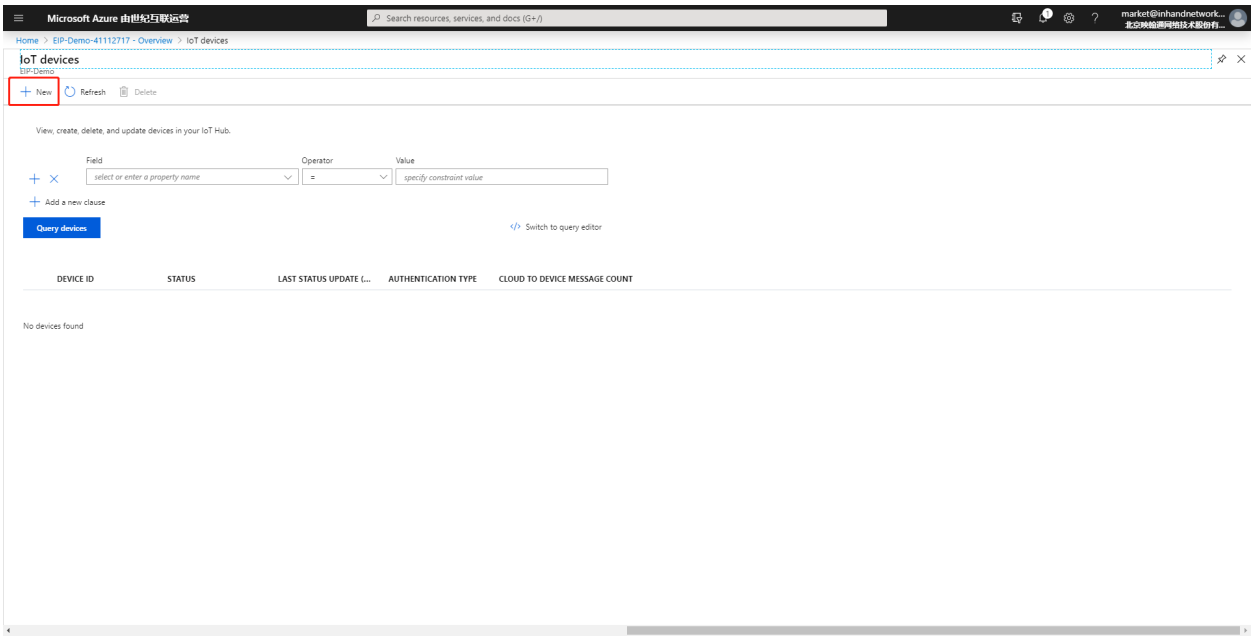
创建成功后如下图所示：

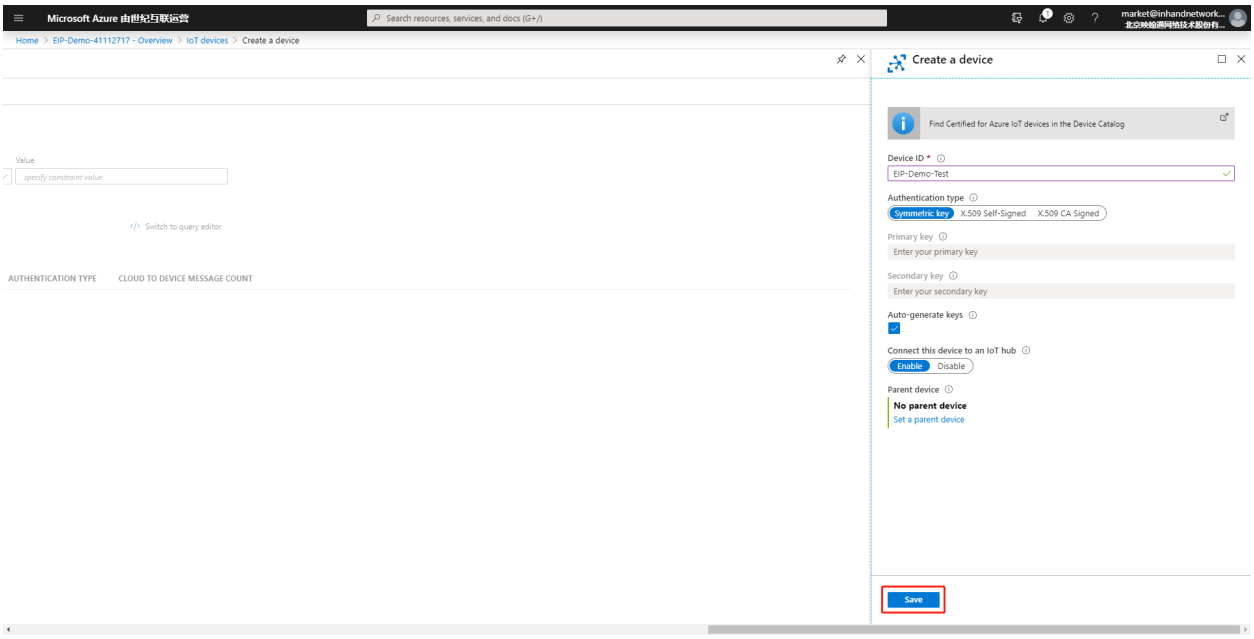

Microsoft Azure 由世纪互联运营
Search resources, services, and docs (G+/)
market@inhandnetwork...
Home > EIP-Demo-41112717 - Overview
EIP-Demo-41112717 - Overview
Deployment
Search (Ctrl+/)
Overview
Inputs
Outputs
Template
Delete
Cancel
Redeploy
Refresh
Your deployment is complete
Deployment name: EIP-Demo-41112717
Subscription: Marketing_Subscription
Resource group: inhandiot
Start time: 4/1/2020, 11:27:18 AM
Correlation ID:
Deployment details (Download)
Next steps
Add and configure IoT Devices Recommended
Configure routing rules for device messaging Recommended
Go to resource
Deployment succeeded
11:28 AM
Deployment 'EIP-Demo-41112717' to resource group 'inhandiot' was successful.
Go to resource
Pin to dashboard
Security Center
Secure your apps and infrastructure
Go to Azure security center >
Free Microsoft tutorials
Use IoT Hub message routing to send device-to-cloud messages to different endpoints
Understanding the device identity registry
Understanding IoT Hub quotas and throttling

1.1.2 添加 IoT Device

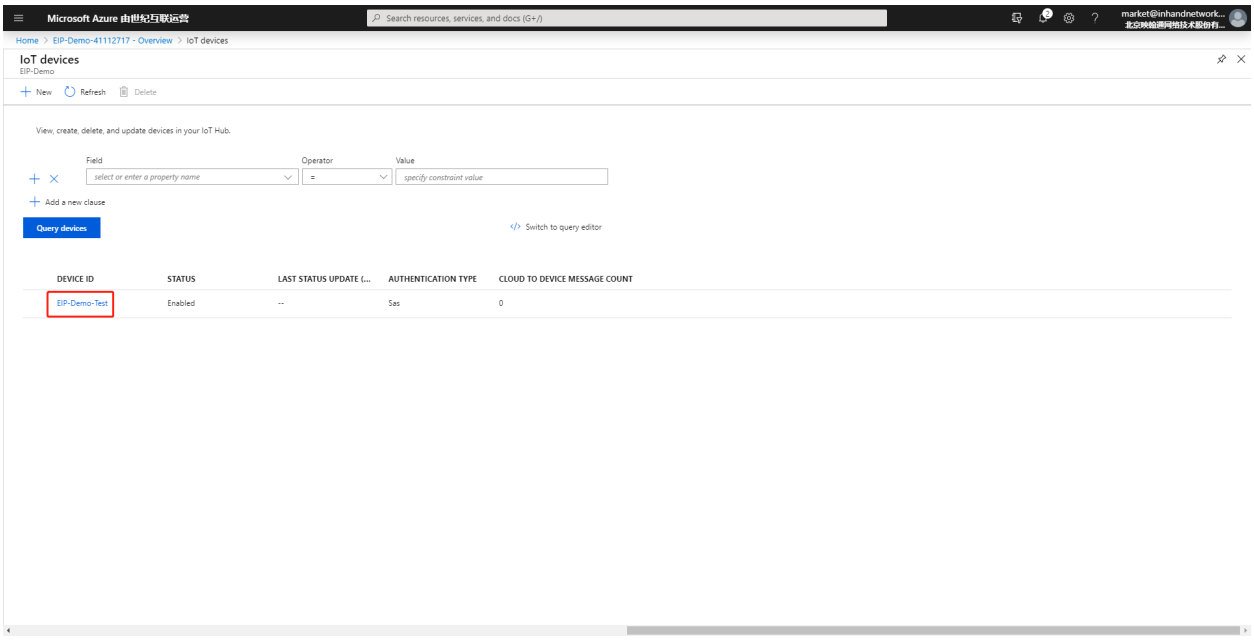
在 IoT Hub 中创建一个 IoT Device。






Microsoft Azure 由世纪互联运营
Search resources, services, and docs (G+/)
market@inhandnetwork...
北京映翰通网络技术股份有...
Home > EIP-Demo-41112717 - Overview > IoT devices > Create a device
Value
specify constraint value
Switch to query editor
AUTHENTICATION TYPE
CLOUD TO DEVICE MESSAGE COUNT
Create a device
Find Certified for Azure IoT devices in the Device Catalog
Device ID *
EIP-Demo-Test
Authentication type
Symmetric key
X.509 Self-Signed
X.509 CA Signed
Primary key
Enter your primary key
Secondary key
Enter your secondary key
Auto-generate keys
Connect this device to an IoT hub
Enable
Disable
Parent device
No parent device
Set a parent device
Save

创建成功后如下图所示：


Microsoft Azure 由世纪互联运营
Search resources, services, and docs (G+/)
market@inhandnetwork...
Home > EIP-Demo-41112717 - Overview > IoT devices
IoT devices
EIP-Demo
New
Refresh
Delete
View, create, delete, and update devices in your IoT Hub.
Field
Operator
Value
select or enter a property name
=
specify constraint value
Add a new clause
Query devices
Switch to query editor
DEVICE ID
STATUS
LAST STATUS UPDATE (...
AUTHENTICATION TYPE
CLOUD TO DEVICE MESSAGE COUNT
EIP-Demo-Test
Enabled
--
Sas
0

1.2 边缘计算网关配置

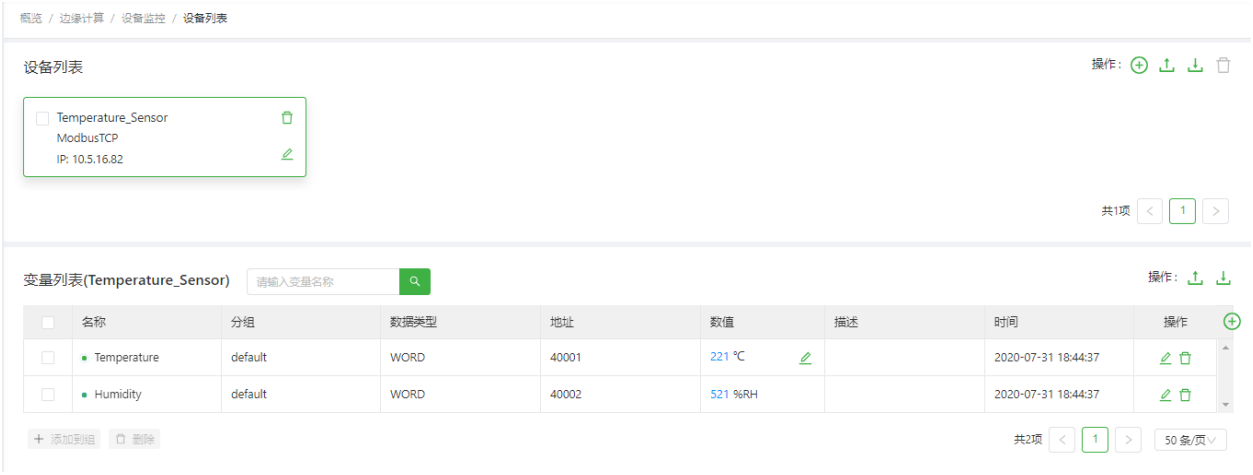
- *1.2.1* 基础配置
- *1.2.2* 配置数据采集

1.2.1 基础配置

- 如何配置 IG902 联网、更新软件版本等操作请参考IG902 快速使用手册。
- 如何配置 IG501 联网、更新软件版本等操作请参考IG501 快速使用手册。

1.2.2 配置数据采集

Device Supervisor 详细的基础数据采集配置见Device Supervisor App 用户手册。本文档的数据采集配置如下：



1.5.3 2. 发布和订阅消息

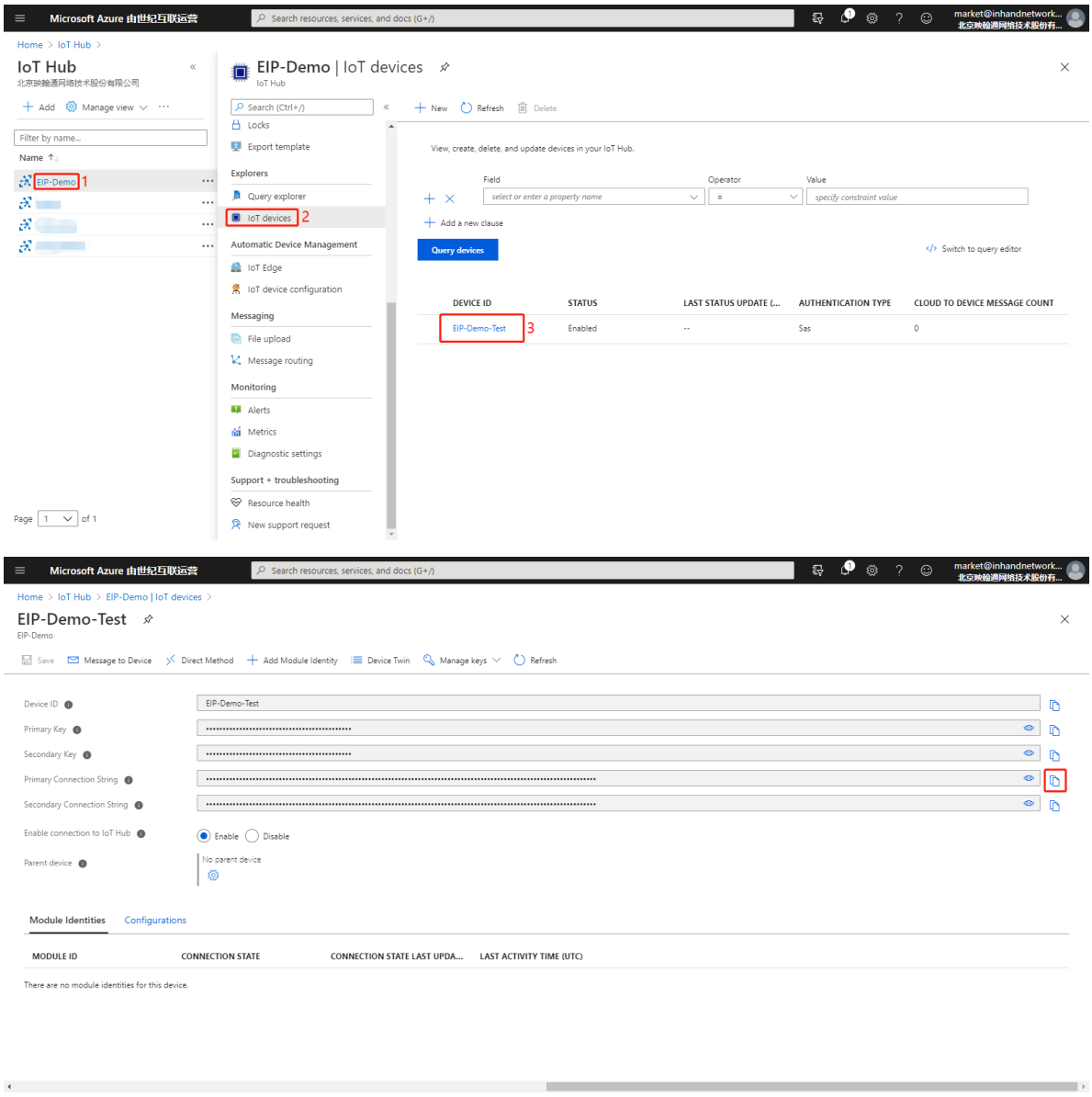
- *2.1* 连接 *Azure IoT*
- *2.2* 发布消息到 *Azure IoT*
- *2.3* 订阅 *Azure IoT* 的消息

2.1 连接 Azure IoT

进入 IG902 的“边缘计算 > 设备监控 > 云服务”页面，勾选“启用云服务”并选择类型为“Azure IoT”。示例配置如下：

各项参数说明如下：

- 类型：连接 Azure IoT 时，选择“Azure IoT”
- 连接字符串：Azure IoT Device 的主连接字符串，你可以从 Azure IoT 的 IoT Hub 中选择相应的设备并复制主连接字符串到此处



2.2 发布消息到 Azure IoT

- 步骤 1：配置发布消息

  在“云服务 > 消息管理”中添加一条发布消息，配置如下：


```
import logging
"""
在网关中打印日志通常有两种办法。
1.import logging: 使用logging.
↪info(XXX)打印日志，该方法的日志显示不受全局参数页面中的日志等级参数控制。
2.from common.Logger import logger: 使用logger.
↪info(XXX)打印日志，该方法的日志显示受全局参数页面中的日志等级参数控制。
"""

def vars_upload_test(data_collect, wizard_api): #定义发布主函数
    value_list = [] #定义数据列表
    for device, val_dict in data_collect['values'].items():
↪#遍历values字典，该字典中包含设备名称和设备下的变量数据
        value_dict = { #自定义数据字典
                       "Device": device,
                       "timestamp": data_collect["timestamp"],
                       "Data": {}
                       }
        for id, val in val_dict.items(): #遍历变量数据，为Data字典赋值
            value_dict["Data"][id] = val["raw_data"]
        value_list.append(value_dict) #依次将value_dict添加到value_list中
    logging.info(value_list) #在App日志中打印value_list，数据格式为[{'Device':
↪'S7-1200', 'timestamp': 1589538347.5604711, 'Data': {'Test1': False, 'Test2':␣
↪12}}]
    return value_list #将value_
↪list发送给App，App将自行按照采集时间顺序上传至MQTT服务器。如果发送失败则缓存数据等待连接恢复后
```

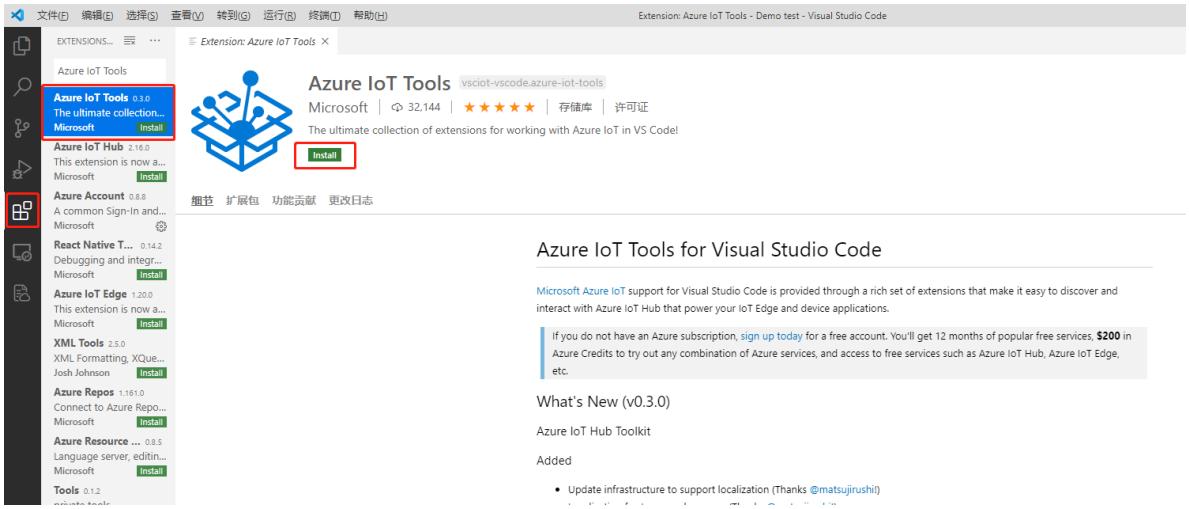
发布消息配置参数说明如下：

- 名称：用户自定义发布名称

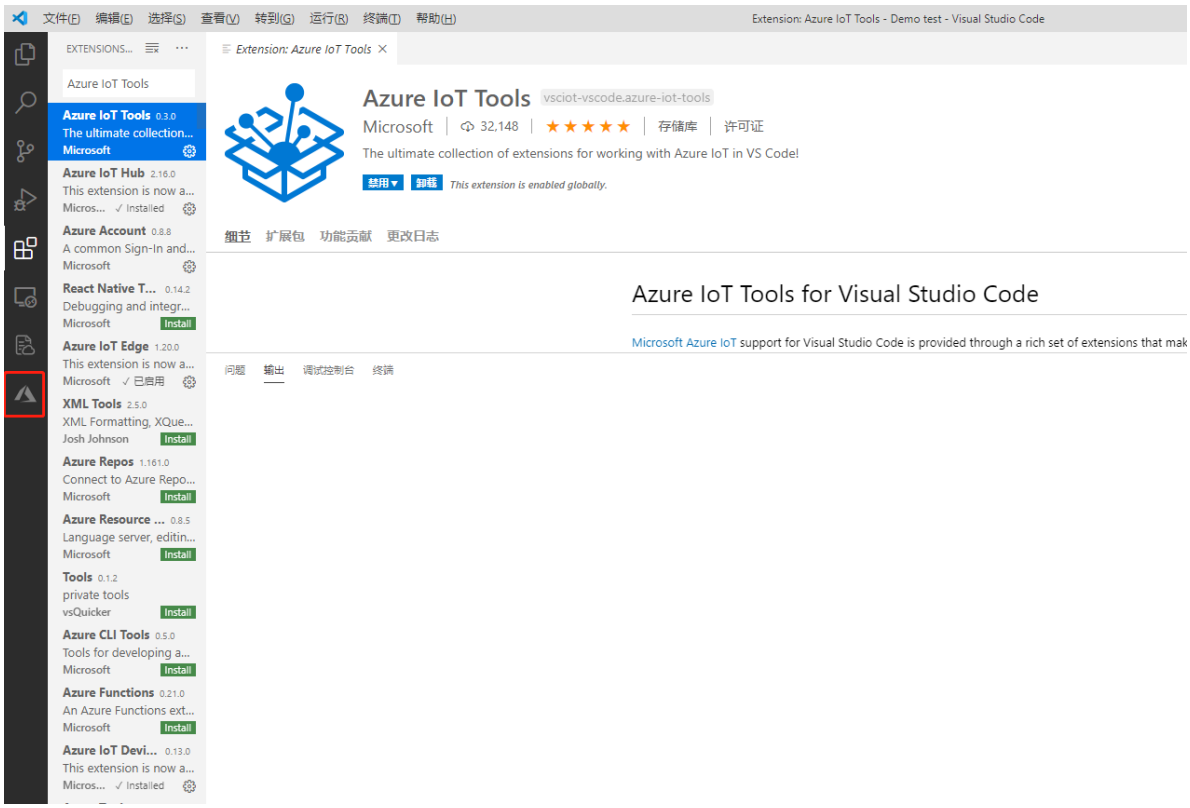
  - 分组类型：发布变量数据时请选择“采集”，随后在分组中仅能选择“采集组”；发布告警数据时请选择“告警”，随后在分组中仅能选择“告警组”
  - 分组：选择相应的分组后，分组下所有变量通过该发布配置将数据上传至 MQTT 服务器；可选择多个分组，当选择多个分组时，按照分组的采集间隔分别对各分组下的变量执行发布中的脚本逻辑。分组中必须包含变量，否则不会执行发布中的脚本逻辑
  - 主函数：主函数名称，即入口函数名称，与脚本中的入口函数名称保持一致
  - 脚本：使用 Python 代码自定义组包和处理逻辑，主函数参数包括：
    * 参数 1：同标准 MQTT-发布主函数中的参数 1
    * 参数 2：Device Supervisor 的 Azure IoT api 接口，参数说明见 *Device Supervisor 的 Azure IoT api 接口说明*

- 步骤 2：使用 VS Code 的 Azure IoT Tools 插件建立与 IoT Hub 的连接

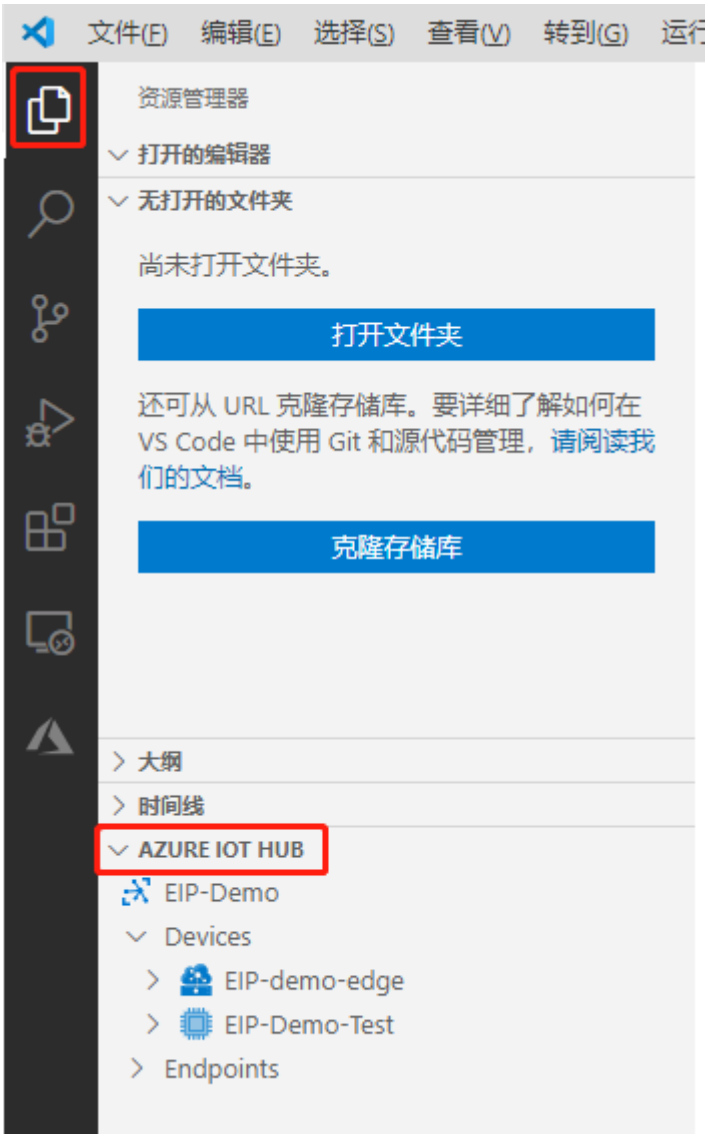
  提交后发布消息后，使用 Visual Studio Code 软件（以下简称 VS Code）的 Azure IoT Tools 插件查看发送到 Azure IoT 的消息。你可以访问：https://code.visualstudio.com/Download 获取相应的 Visual Studio Code 软件（以下简称 VS Code）。VS Code 安装完毕后，打开 VS Code 并点击“Extensions”，在搜索框中输入 `Azure IoT Tools` 并安装 `Azure IoT Tools` 插件。



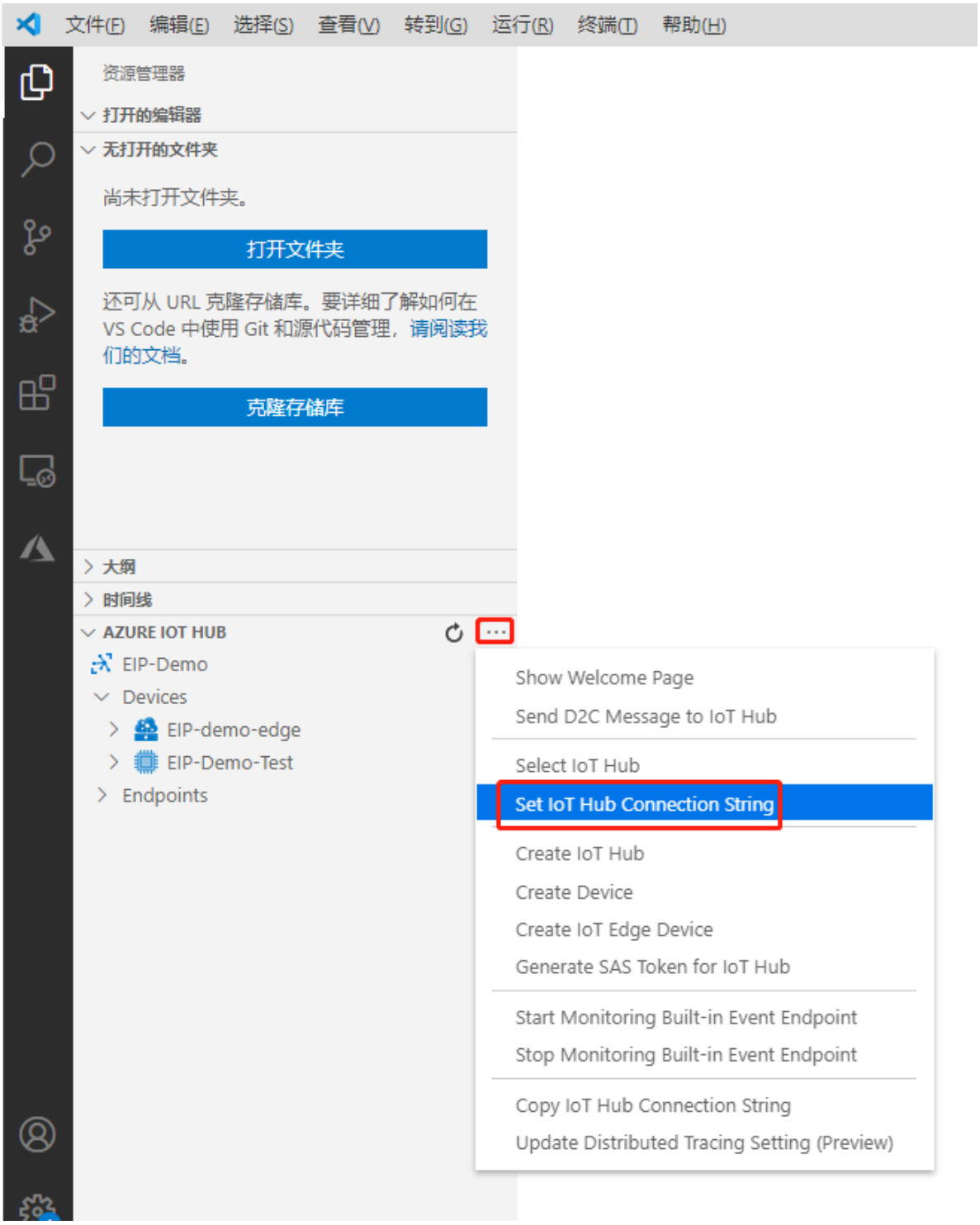
  安装成功后在左侧可以看到 `Azure` 模块。



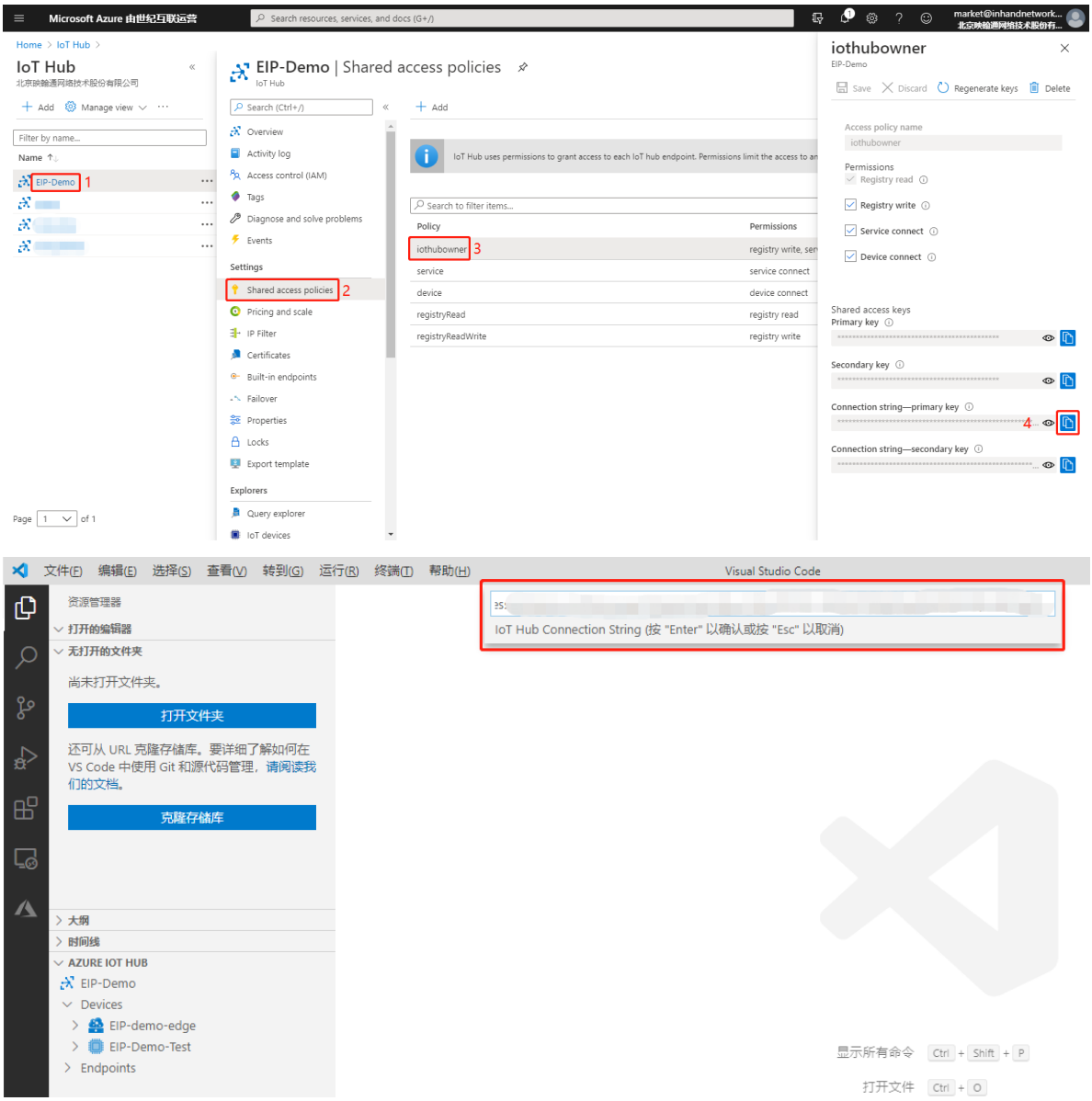
点击“资源管理器 > AZURE IOT HUB”，展开 `AZURE IOT HUB`。



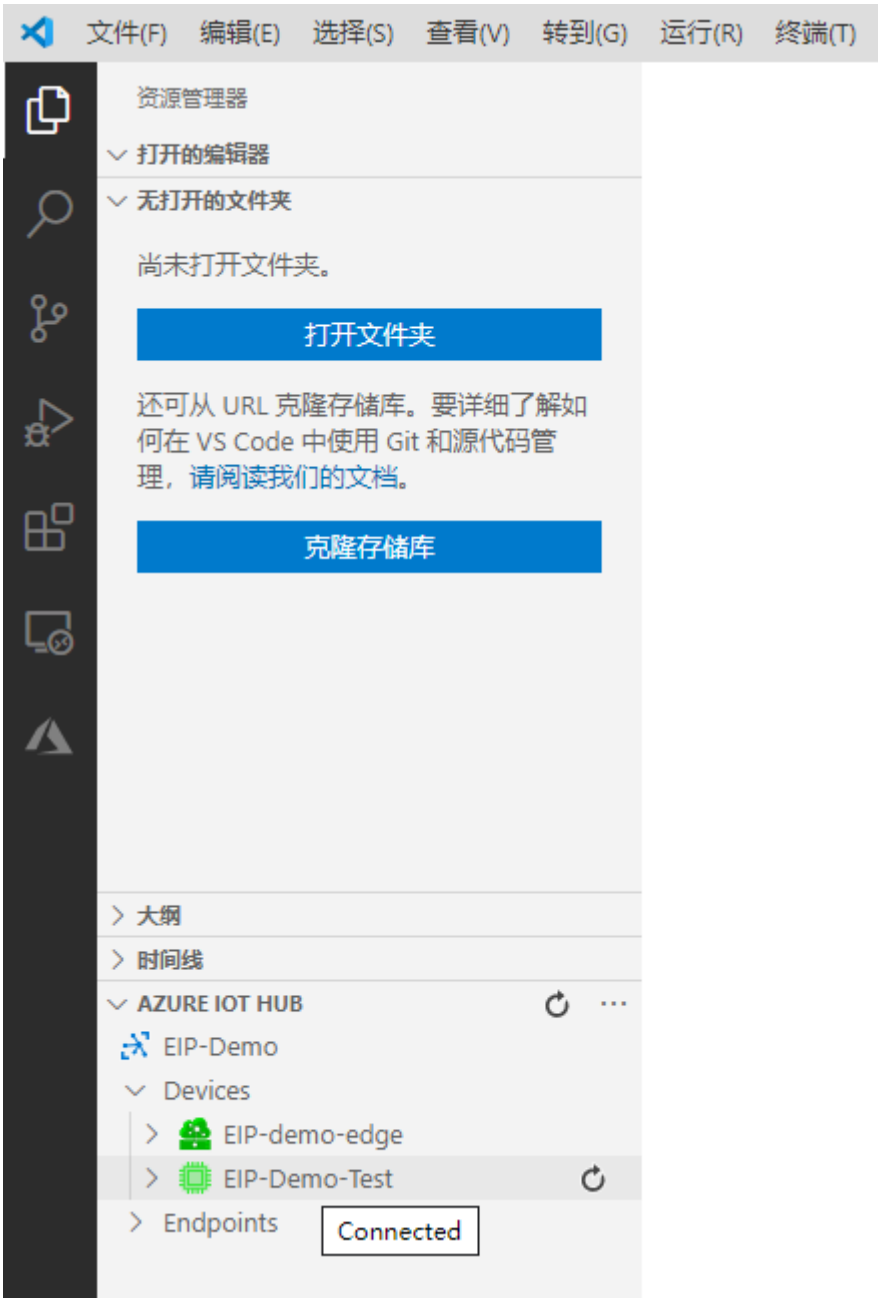
点击“···> Set IoT Hub Connection String”设置 IoT Hub 的连接字符串。



在提示框中输入 IoT Hub 的连接字符串。IoT Hub 的连接字符串可从指定 IoT Hub 的“Shared access policies > iothubowner”页面获取。

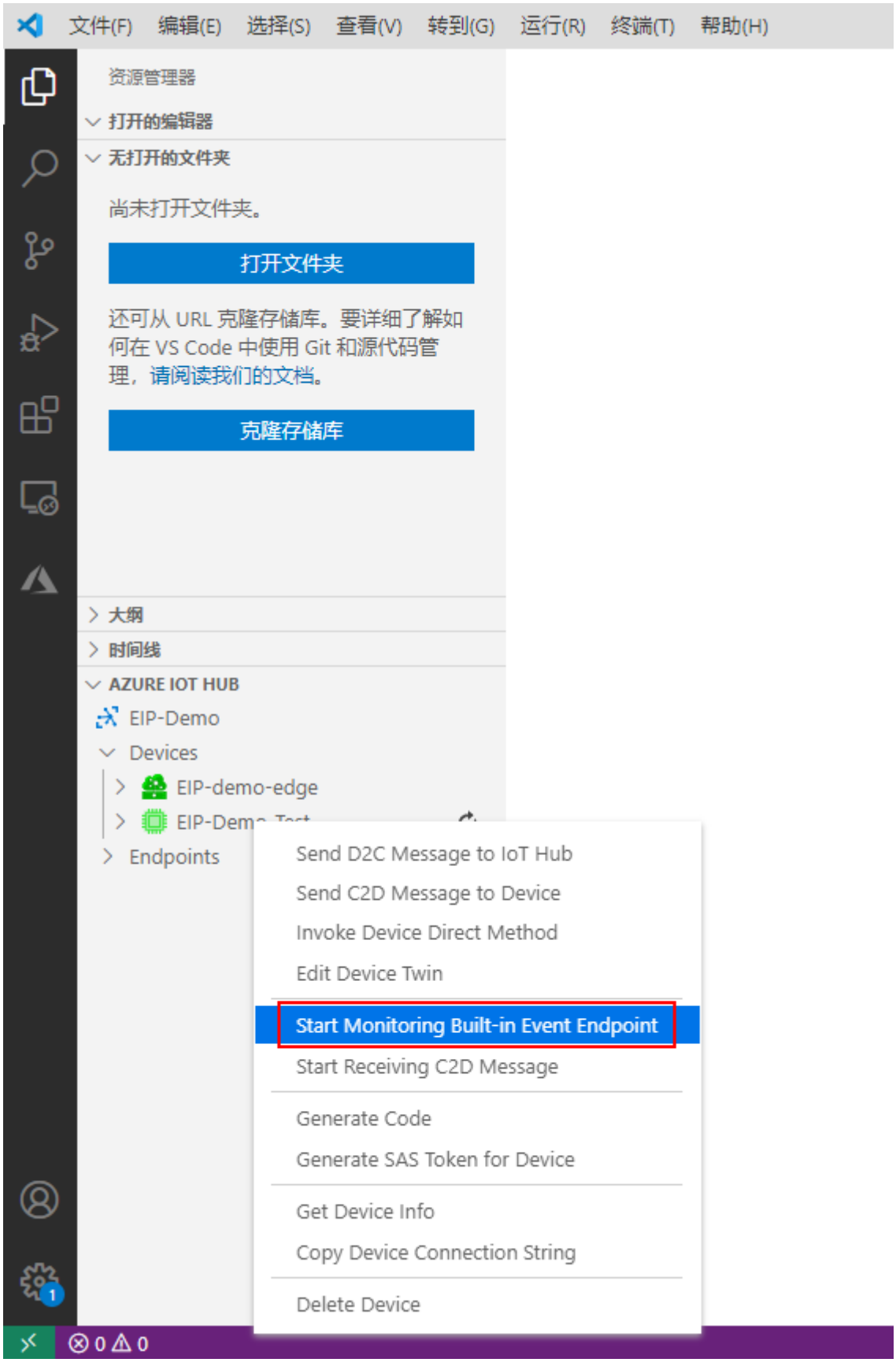


输入连接字符串并回车后在“AZURE IOT HUB”下可以看到已创建的 Azure IoT Device 和连接状态。

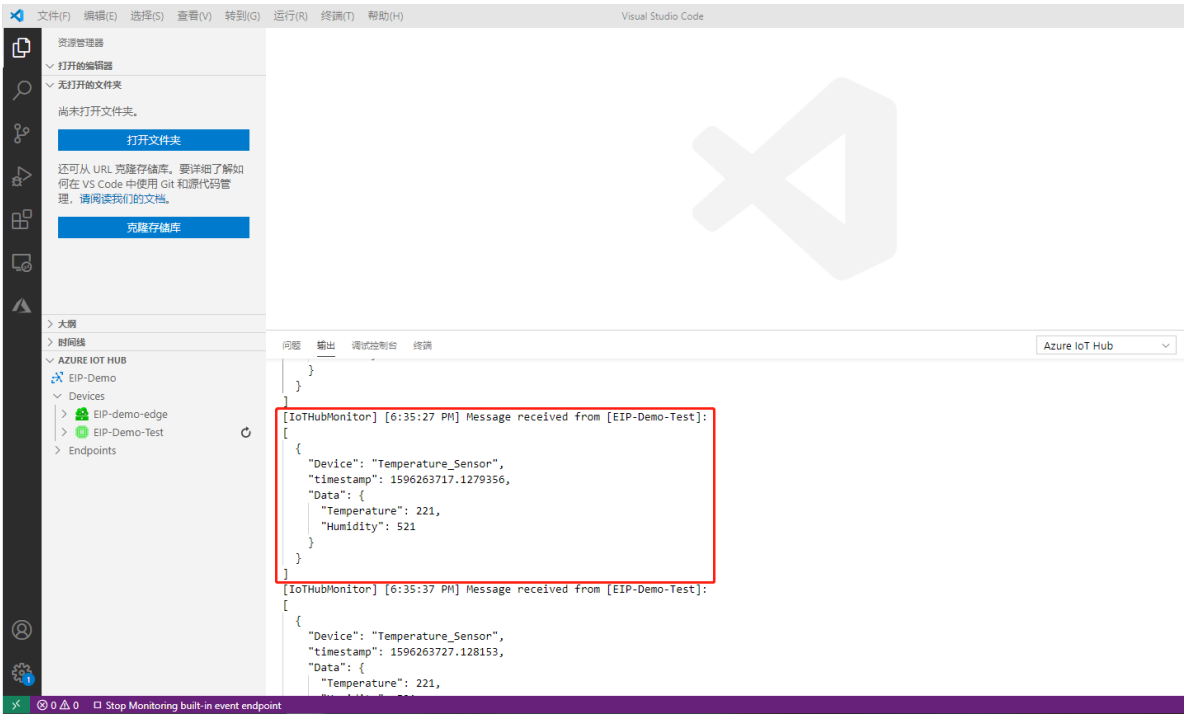


- 步骤 3：查看 IG902 上传至 Azure IoT 的消息

  右击指定的 Azure IoT Device 并在菜单中选择 `Start Monitoring Built-in Event Endpoint` 以查看 IG902 推送到 IoT Hub 的数据。


文件(F)
编辑(E)
选择(S)
查看(V)
转到(G)
运行(R)
终端(T)
帮助(H)
资源管理器
打开的编辑器
无打开的文件夹
尚未打开文件夹。
打开文件夹
还可从 URL 克隆存储库。要详细了解如何在 VS Code 中使用 Git 和源代码管理，请阅读我们的文档。
克隆存储库
大纲
时间线
AZURE IOT HUB
EIP-Demo
Devices
EIP-demo-edge
Endpoints
Send D2C Message to IoT Hub
Send C2D Message to Device
Invoke Device Direct Method
Edit Device Twin
Start Monitoring Built-in Event Endpoint
Start Receiving C2D Message
Generate Code
Generate SAS Token for Device
Get Device Info
Copy Device Connection String
Delete Device
0 0

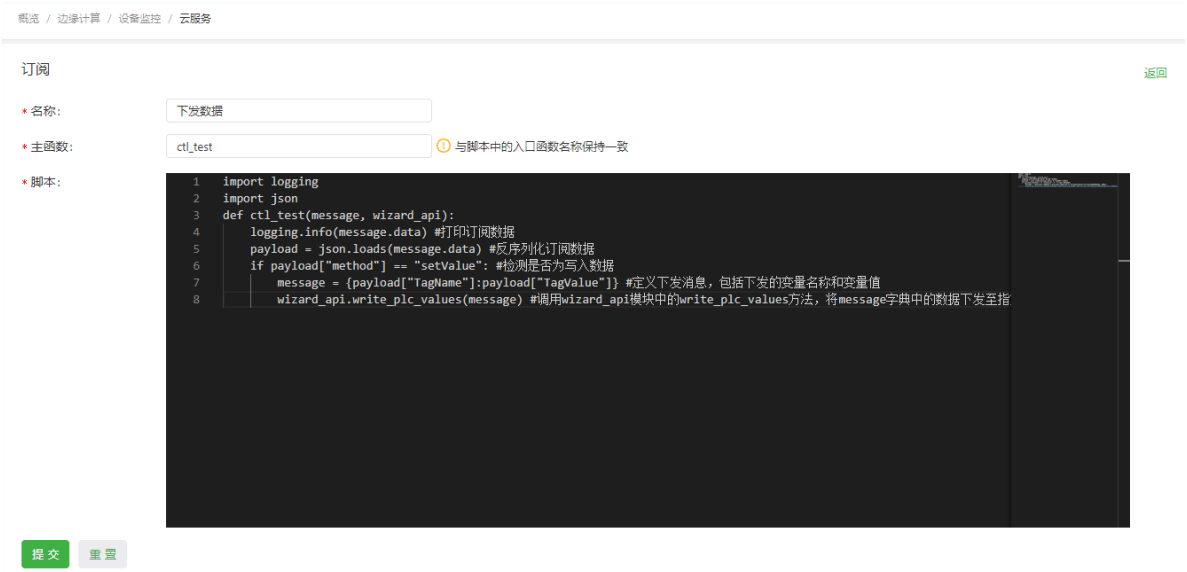
随后在“输出”窗口中可以看到 IoT Hub 接收到的消息内容。



2.3 订阅 Azure IoT 的消息

- 步骤 1：配置订阅消息

  在“云服务 > 消息管理”中添加一条订阅消息，配置如下：



```
import logging
import json
def ctl_test(message, wizard_api):
    logging.info(message.data) #打印订阅数据,假定payload数据为{"method":"setValue
↪", "TagName":"SP1", "TagValue":12.3}
    payload = json.loads(message.data) #反序列化订阅数据
    if payload["method"] == "setValue": #检测是否为写入数据
        message = {payload["TagName"]:payload["TagValue"]}
↪#定义下发消息，包括下发的变量名称和变量值
        wizard_api.write_plc_values(message) #调用wizard_api模块中的write_plc_
↪values方法，将message字典中的数据下发至指定变量
```

订阅消息配置参数说明如下：

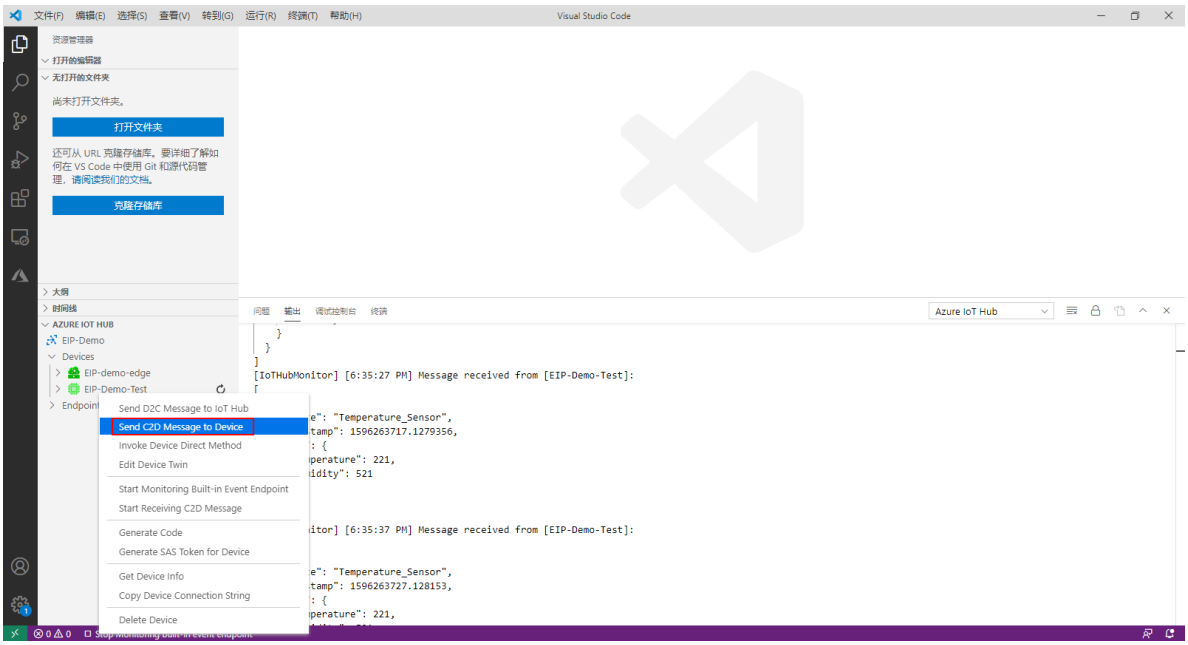
- 名称：用户自定义订阅名称
- 主函数：主函数名称，即入口函数名称，与脚本中的入口函数名称保持一致
- 脚本：使用 Python 代码自定义组包和处理逻辑，主函数参数包括：
  - 参数 1：Azure IoT 下发的消息类，支持 `data` 和 `custom_properties` 方法，使用示例见订阅 *Azure IoT* 消息示例
  - 参数 2：Device Supervisor 的 Azure IoT api 接口，参数说明见 *Device Supervisor* 的 *Azure IoT api* 接口说明

- 步骤 2：使用 VS Code 的 Azure IoT Tools 插件建立与 IoT Hub 的连接

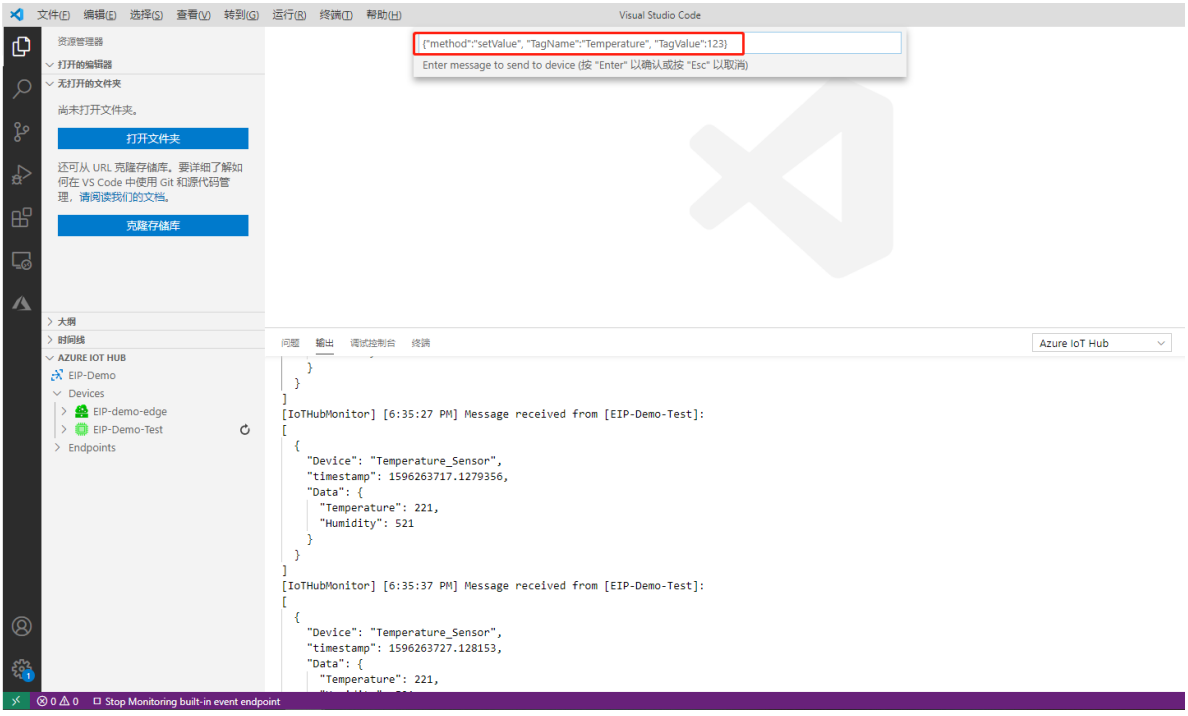
  同“发布消息到 *Azure IoT*”的步骤 2。

- 步骤 3：使用 Azure IoT Tools 下发数据至 IG902

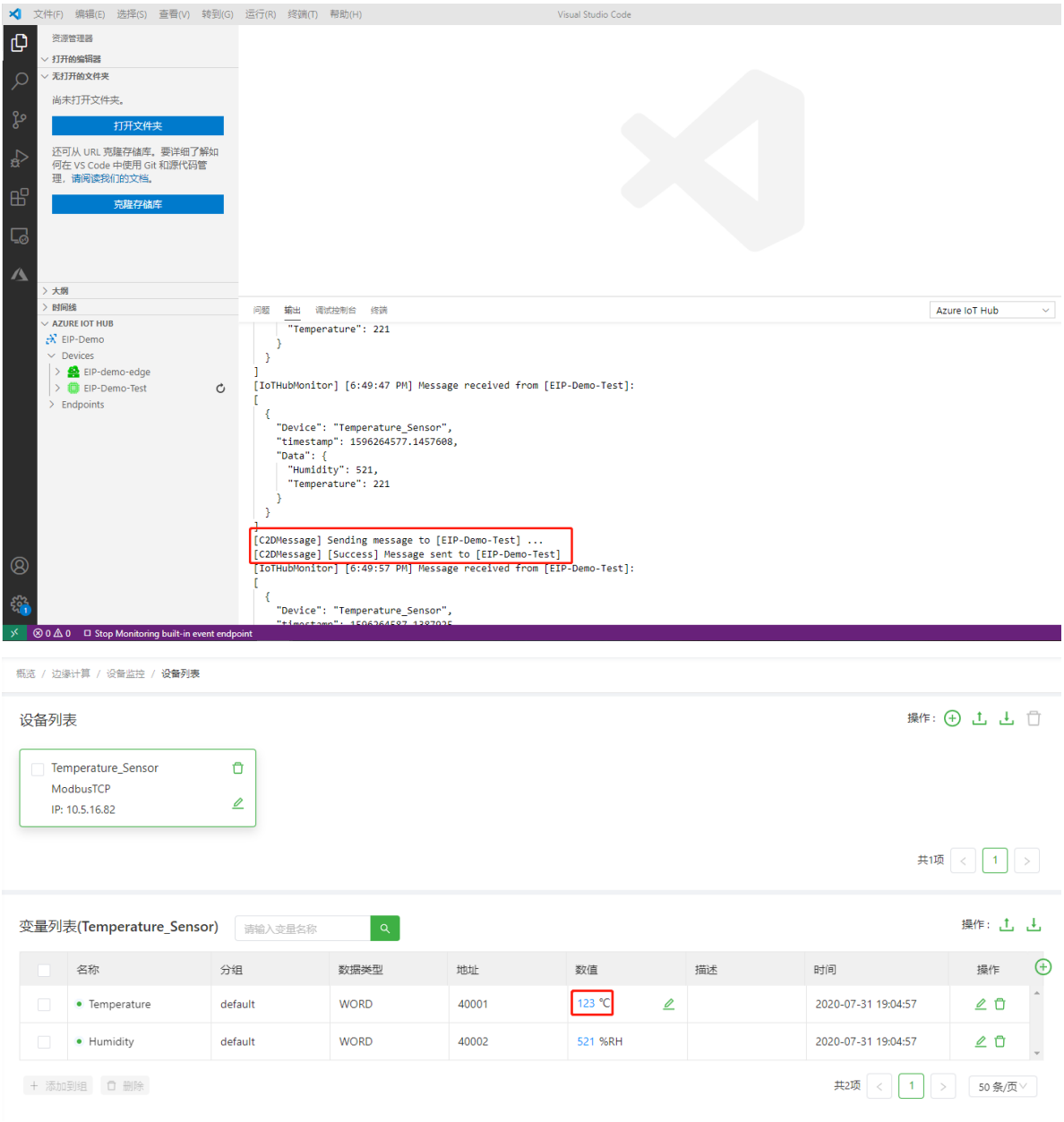
  右击指定的 Azure IoT Device 并在菜单中选择 `Send C2D Message to Device` 以下发数据至 IG902。



在下发框中输入需要下发的数据，如 `{"method":"setValue", "TagName":"Temperature", "TagValue":123}`



回车发送数据后，在“输出”窗口可以看到消息下发成功的日志，同时在“设备列表”页面可以看到变量 `Temperature` 的数据已修改为 123。



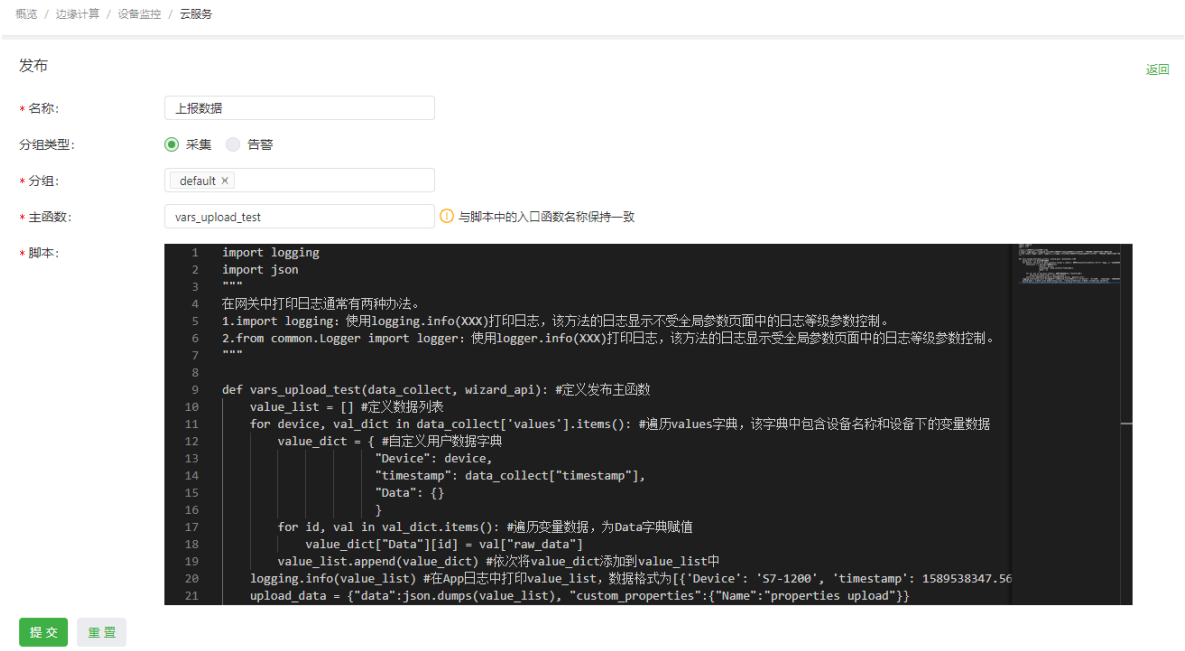
至此，实现了 Device Supervisor 与 Azure IoT 的业务数据上报和配置数据下发。

1.5.4 附录

- 发布 *Azure IoT* 消息示例
- 订阅 *Azure IoT* 消息示例
- *Device Supervisor* 的 *Azure IoT api* 接口说明

发布 Azure IoT 消息示例

- 发布示例 1：使用 return 发布用户数据和属性数据



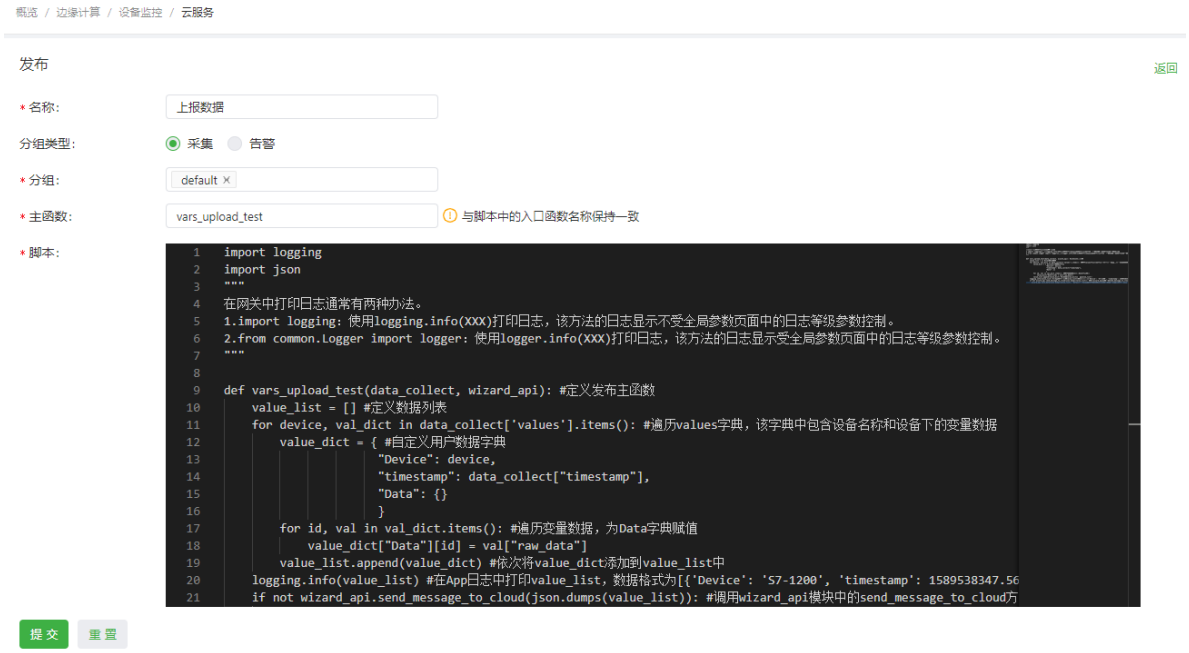

```
import logging
import json
"""
在网关中打印日志通常有两种办法。
1.import logging: 使用logging.
↪info(XXX)打印日志，该方法的日志显示不受全局参数页面中的日志等级参数控制。
2.from common.Logger import logger: 使用logger.
↪info(XXX)打印日志，该方法的日志显示受全局参数页面中的日志等级参数控制。
"""

def vars_upload_test(data_collect, wizard_api): #定义发布主函数
    value_list = [] #定义数据列表
    for device, val_dict in data_collect['values'].items():
↪#遍历values字典，该字典中包含设备名称和设备下的变量数据
```

(续下页)

(接上页)

```
        value_dict = { #自定义用户数据字典
                    "Device": device,
                    "timestamp": data_collect["timestamp"],
                    "Data": {}
                    }
        for id, val in val_dict.items(): #遍历变量数据，为Data字典赋值
            value_dict["Data"][id] = val["raw_data"]
        value_list.append(value_dict) #依次将value_dict添加到value_list中
    logging.info(value_list) #在App日志中打印value_list，数据格式为[{'Device':
↪'S7-1200', 'timestamp': 1589538347.5604711, 'Data': {'Test1': False, 'Test2':␣
↪12}}]
    upload_data = {"data":json.dumps(value_list), "custom_properties":{"Name":
↪"properties upload"}} #定义上报数据，数据类型为字典。用户数据为"data
↪"的值，数据类型为字符串；属性数据为"custom_properties"的值，数据类型为字典
    return(upload_data) #将upload_
↪data发送给App，App将自行按照采集时间顺序上传至Azure␣
↪IoT。如果发送失败则缓存数据等待连接恢复后按采集时间顺序上传至Azure IoT
```

- 发布示例 2：使用 `send_message_to_cloud` 发送用户数据并使用 `save_data` 存储上传失败的变量




```
import logging
import json
"""
```

(续下页)

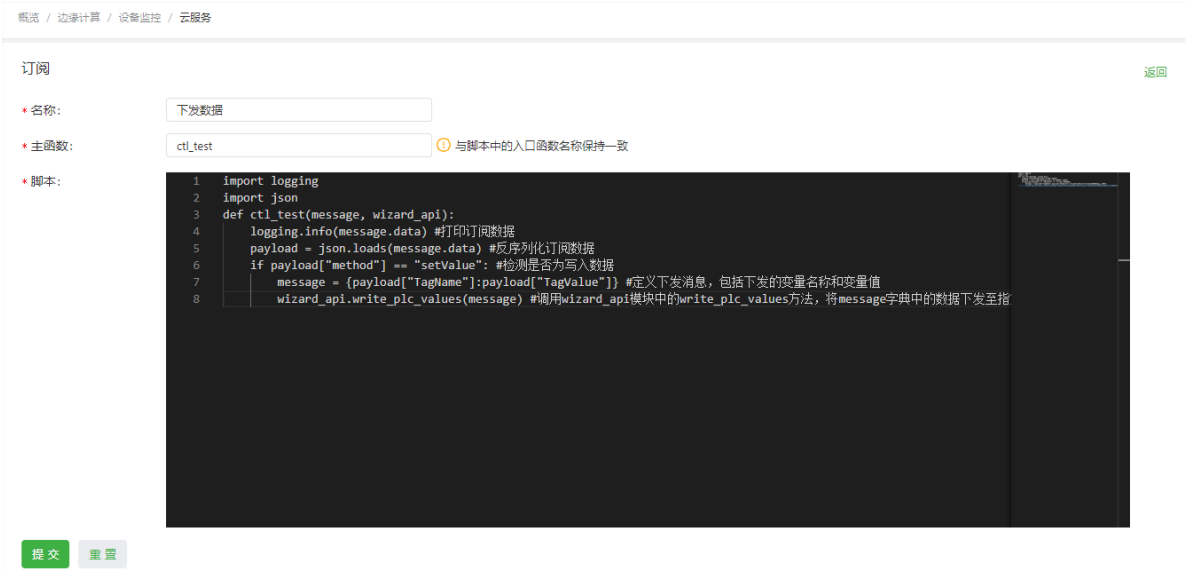
(接上页)

```
在网关中打印日志通常有两种办法。
1.import logging: 使用logging.
↪info(XXX)打印日志，该方法的日志显示不受全局参数页面中的日志等级参数控制。
2.from common.Logger import logger: 使用logger.
↪info(XXX)打印日志，该方法的日志显示受全局参数页面中的日志等级参数控制。
"""

def vars_upload_test(data_collect, wizard_api): #定义发布主函数
    value_list = [] #定义数据列表
    for device, val_dict in data_collect['values'].items():
↪#遍历values字典，该字典中包含设备名称和设备下的变量数据
        value_dict = { #自定义用户数据字典
                        "Device": device,
                        "timestamp": data_collect["timestamp"],
                        "Data": {}
                        }
        for id, val in val_dict.items(): #遍历变量数据，为Data字典赋值
            value_dict["Data"][id] = val["raw_data"]
        value_list.append(value_dict) #依次将value_dict添加到value_list中
    logging.info(value_list) #在App日志中打印value_list，数据格式为[{'Device':
↪'S7-1200', 'timestamp': 1589538347.5604711, 'Data': {'Test1': False, 'Test2':␣
↪12}}]
    if not wizard_api.send_message_to_cloud(json.dumps(value_list)): #调用wizard_
↪api模块中的send_message_to_cloud方法将value_list发送给Azure␣
↪IoT并检测是否发送成功
      wizard_api.save_data(json.dumps(value_list), 'default')
↪#发送失败则存储数据，等待连接恢复后将按时间顺序上传存储数据
```

订阅 Azure IoT 消息示例

- 订阅用户数据

  示例配置如下：

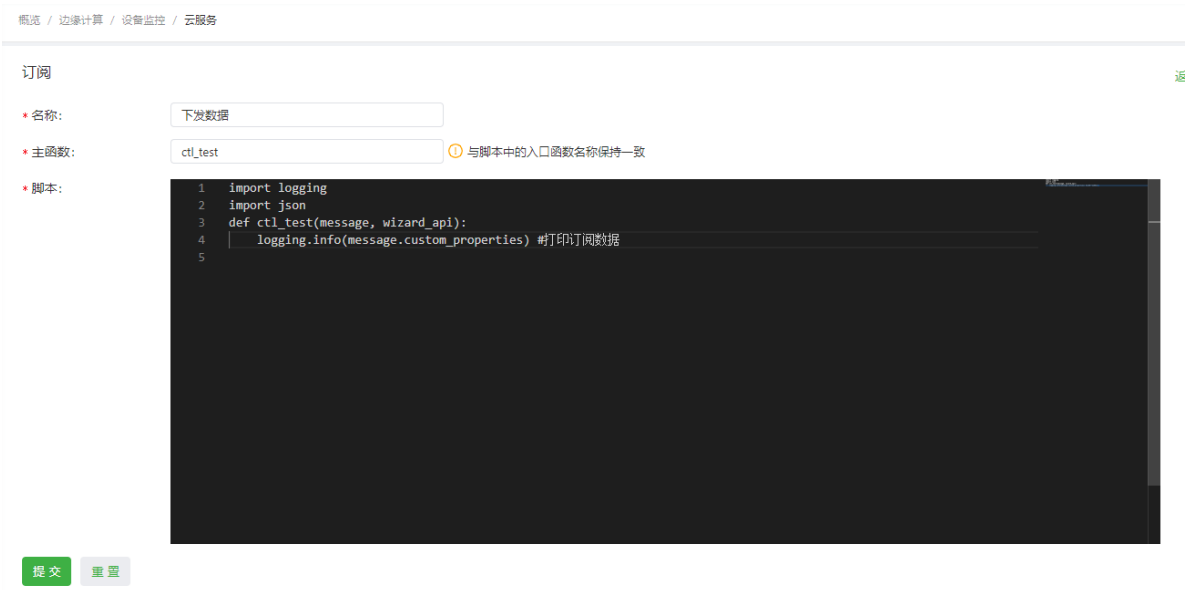


脚本如下：

```
import logging
import json
def ctl_test(message, wizard_api):
    logging.info(message.data) #打印订阅数据，假定payload数据为{"method":"setValue
↪", "TagName":"SP1", "TagValue":12.3}
    payload = json.loads(message.data) #反序列化订阅数据
    if payload["method"] == "setValue": #检测是否为写入数据
        message = {payload["TagName"]:payload["TagValue"]}
↪#定义下发消息，包括下发的变量名称和变量值
        wizard_api.write_plc_values(message) #调用wizard_api模块中的write_plc_
↪values方法，将message字典中的数据下发至指定变量
```

- 订阅属性数据

  示例配置如下：



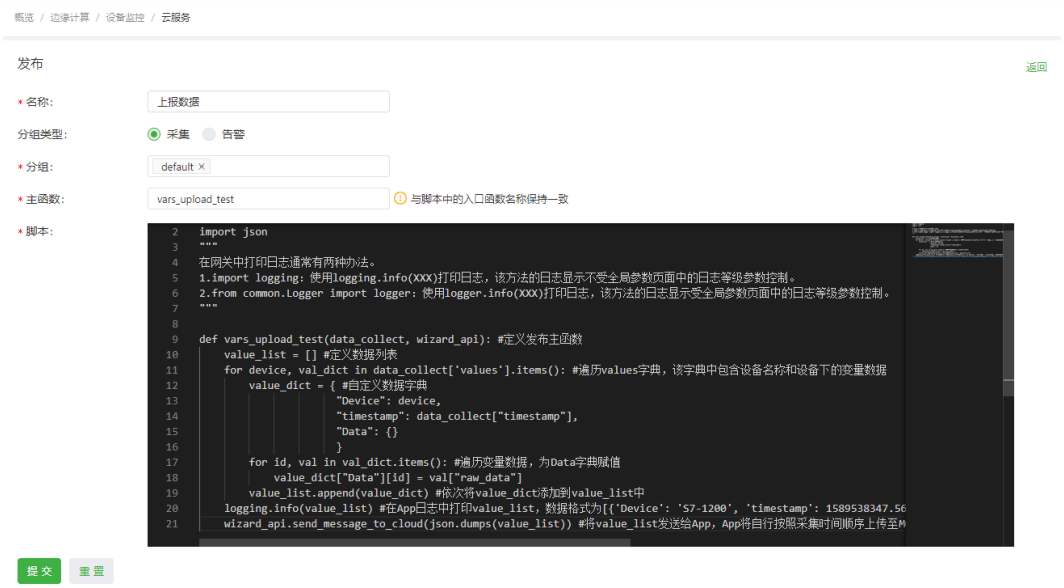
脚本如下：

```
import logging
import json
def ctl_test(message, wizard_api):
    logging.info(message.custom_properties) #打印订阅数据
```

Device Supervisor 的 Azure IoT api 接口说明

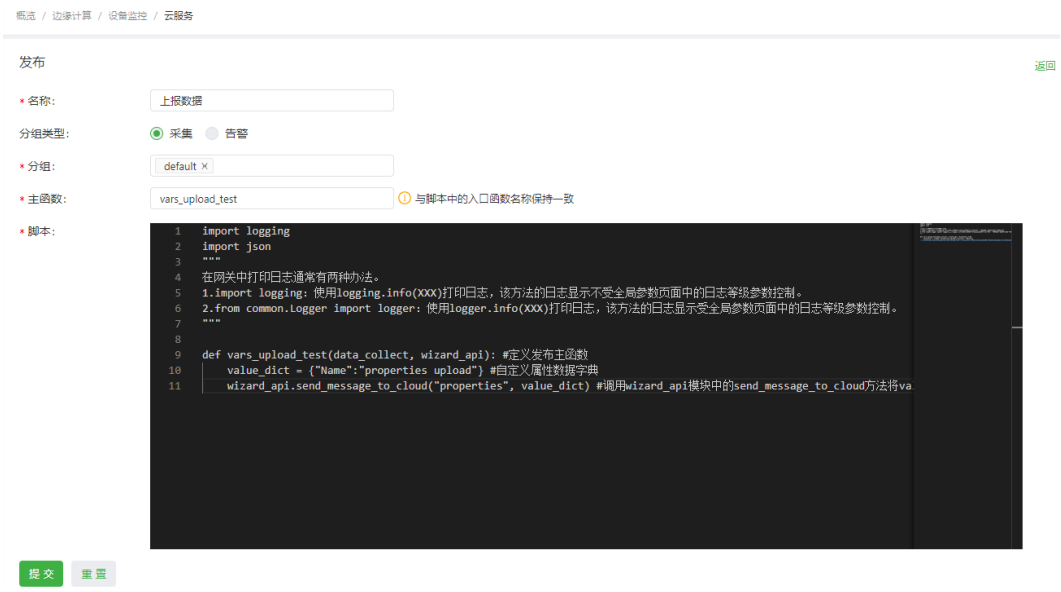
`wizard_api` 的基础配置方法请参考Device Supervisor 的 api 接口说明（注意：如需存储数据，只能参考发布示例 2 即通过组名存储数据）。当云服务类型为 `Azure IoT` 时，`wizard_api` 额外提供以下方法：

- `send_message_to_cloud`
  - 方法说明：数据上报方法
  - 参数
    - `data`：需要上报的用户数据。该参数不能为空且数据类型须为字符串，单次上报的数据大小不能超过 256KB
    - `custom_properties`：需要上报的属性数据。属性数据的数据类型须为字典，字典中的值的数据类型仅支持整数、浮点数和字符串。单次上报的数据不能超过 81KB
  - 使用示例
    - 上报用户数据




```
import logging
import json
"""
在网关中打印日志通常有两种办法。
1.import logging: 使用logging.
↪info(XXX)打印日志，该方法的日志显示不受全局参数页面中的日志等级参数控制。
2.from common.Logger import logger: 使用logger.
↪info(XXX)打印日志，该方法的日志显示受全局参数页面中的日志等级参数控制。
"""

def vars_upload_test(data_collect, wizard_api): #定义发布主函数
    value_list = [] #定义数据列表
    for device, val_dict in data_collect['values'].items():
↪#遍历values字典，该字典中包含设备名称和设备下的变量数据
        value_dict = { #自定义用户数据字典
                        "Device": device,
                        "timestamp": data_collect["timestamp"],
                        "Data": {}
                        }
        for id, val in val_dict.items(): #遍历变量数据，为Data字典赋值
            value_dict["Data"][id] = val["raw_data"]
        value_list.append(value_dict) #依次将value_dict添加到value_list中
    logging.info(value_list) #在App日志中打印value_list，数据格式为[{
↪'Device': 'S7-1200', 'timestamp': 1589538347.5604711, 'Data': {'Test1
↪': False, 'Test2': 12}}]
    wizard_api.send_message_to_cloud(json.dumps(value_list)) #调用wizard_
↪api模块中的send_message_to_cloud方法将value_
↪list（用户数据）发送给Azure IoT
```

* 上报属性数据




```
import logging
import json
"""
在网关中打印日志通常有两种办法。
1.import logging: 使用logging.
↪info(XXX)打印日志，该方法的日志显示不受全局参数页面中的日志等级参数控制。
2.from common.Logger import logger: 使用logger.
↪info(XXX)打印日志，该方法的日志显示受全局参数页面中的日志等级参数控制。
"""

def vars_upload_test(data_collect, wizard_api): #定义发布主函数
    value_dict = {"Name":"properties upload"} #自定义属性数据字典
    wizard_api.send_message_to_cloud("properties", value_dict)
↪#调用wizard_api模块中的send_message_to_cloud方法将value_
↪dict（属性数据）发送给Azure␣
↪IoT。注意：发送属性数据时，参数1不能为空且数据类型为字符串
```

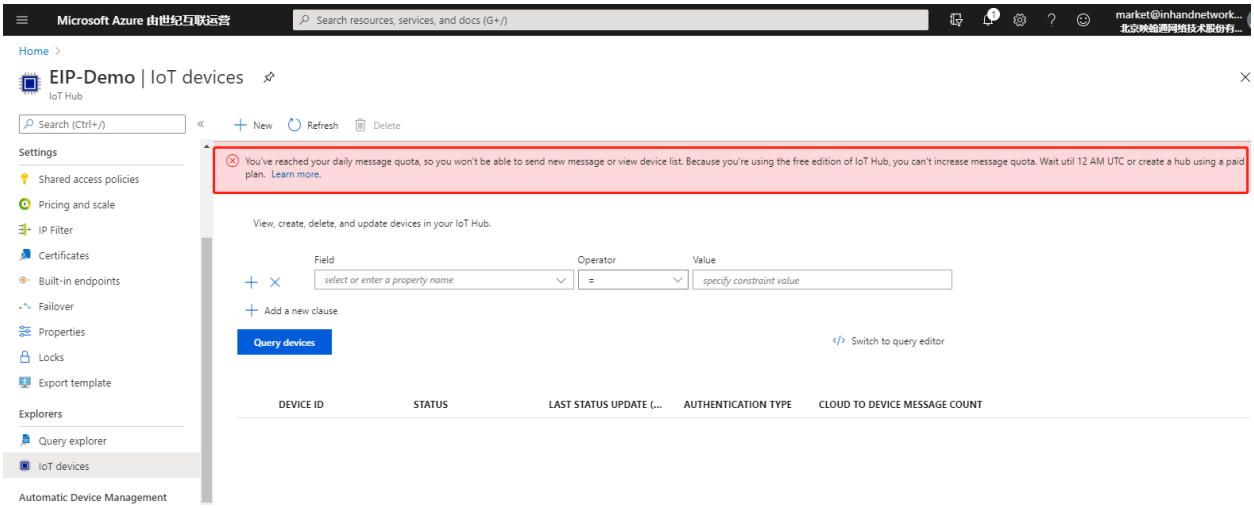
1.5.5 FAQ

Q1：连接 Azure IoT 时，出现频繁连接正常一段时间后又断开

A1：查看 App 运行日志，发现日志中有 `Paho returned rc==1`。

```
[2020-08-10 14:55:48,622] [INFO] [pipeline_stages_mqtt.py 274]: _on_mqtt_connected called
[2020-08-10 14:55:48,624] [DEBUG] [pipeline_stages_base.py 253]: PipelineRootStage: ConnectedEvent received. Calling on_connected_handler
[2020-08-10 14:55:48,626] [DEBUG] [pipeline_thread.py 108]: Starting _on_connected in callback thread
[2020-08-10 14:55:48,634] [ERROR] [handle_exceptions.py 28]: Exception caught in background thread.  Unable to handle.
[2020-08-10 14:55:48,636] [ERROR] [handle_exceptions.py 29]: ["TypeError: 'NoneType' object is not callable\n"]
[2020-08-10 14:55:48,641] [DEBUG] [pipeline_stages_mqtt.py 280]: completing connect op
[2020-08-10 14:55:48,646] [DEBUG] [pipeline_stages_mqtt.py 103]: MQTTTransportStage(ConnectOperation): cancelling watchdog
[2020-08-10 14:55:48,650] [DEBUG] [pipeline_ops_base.py 109]: ConnectOperation: completing without error
[2020-08-10 14:55:48,654] [DEBUG] [pipeline_stages_base.py 525]: ConnectionLockStage(ConnectOperation): op succeeded.  Unblocking queue
[2020-08-10 14:55:48,655] [DEBUG] [pipeline_stages_base.py 550]: ConnectionLockStage(ConnectOperation): unblocking and releasing queued ops.
[2020-08-10 14:55:48,657] [INFO] [pipeline_stages_base.py 553]: ConnectionLockStage(ConnectOperation): processing 0 items in queue
[2020-08-10 14:55:48,662] [DEBUG] [pipeline_stages_base.py 1046]: ReconnectStage(ConnectOperation): on_connect_complete error=None state=LOGICALLY_CONNECTED never_connected=False connected=True
[2020-08-10 14:55:48,666] [INFO] [pipeline_stages_base.py 1141]: ReconnectStage: completing waiting ops with error=None
[2020-08-10 14:55:48,670] [DEBUG] [pipeline_stages_base.py 1095]: ReconnectStage: Reconnect timer expired. State is WAITING_TO_RECONNECT Connected is False.
[2020-08-10 14:55:48,674] [DEBUG] [pipeline_stages_base.py 1071]: ReconnectStage: sending new connect op down
[2020-08-10 14:55:48,678] [DEBUG] [pipeline_stages_base.py 541]: ConnectionLockStage(ConnectOperation): blocking
[2020-08-10 14:55:48,682] [INFO] [pipeline_stages_mqtt.py 170]: MQTTTransportStage(ConnectOperation): connecting
[2020-08-10 14:55:48,686] [DEBUG] [pipeline_stages_mqtt.py 68]: MQTTTransportStage(ConnectOperation): Starting watchdog
[2020-08-10 14:55:48,691] [INFO] [mqtt_transport.py 376]: connecting to mqtt broker
[2020-08-10 14:55:48,694] [INFO] [mqtt_transport.py 387]: Connect using port 8883 (TCP)
[2020-08-10 14:55:48,900] [DEBUG] [client.py 2165]: Sending CONNECT (u1, p1, wr0, wq0, wf0, c0, k60) client_id=b'IG902-55'
[2020-08-10 14:55:48,909] [DEBUG] [mqtt_transport.py 428]: _mqtt_client.connect returned rc=0
[2020-08-10 14:55:48,915] [INFO] [pipeline_stages_mqtt.py 321]: MQTTTransportStage: _on_mqtt_disconnect called: ConnectionDroppedError('Paho returned rc==1')
[2020-08-10 14:55:48,919] [DEBUG] [pipeline_stages_base.py 1002]: ReconnectStage(DisconnectedEvent): State is WAITING_TO_RECONNECT Connected is False.
[2020-08-10 14:55:48,923] [DEBUG] [pipeline_stages_base.py 261]: PipelineRootStage: DisconnectedEvent received. Calling on_disconnected_handler
[2020-08-10 14:55:48,924] [DEBUG] [pipeline_thread.py 108]: Starting _on_disconnected in callback thread
[2020-08-10 14:55:48,929] [INFO] [pipeline_stages_mqtt.py 359]: MQTTTransportStage: disconnection was unexpected
[2020-08-10 14:55:48,931] [INFO] [sync_clients.py 90]: Connection State - Disconnected
[2020-08-10 14:55:48,936] [INFO] [sync_clients.py 92]: Cleared all pending method requests due to disconnect
[2020-08-10 14:55:48,935] [INFO] [handle_exceptions.py 52]: Unexpected disconnection.  Safe to ignore since other stages will reconnect.
[2020-08-10 14:55:48,942] [INFO] [handle_exceptions.py 53]: azure.iot.device.common.transport_exceptions.ConnectionDroppedError: ConnectionDroppedError('Paho returned rc==1')

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "/var/user/app/device_supervisorbak/device_supervisor/lib/azure/iot/device/common/handle_exceptions.py", line 43, in swallow_unraised_exception
azure.iot.device.common.transport_exceptions.ConnectionDroppedError: ConnectionDroppedError(None) caused by ConnectionDroppedError('Paho returned rc==1')
```

登录 Azure IoT，进入连接字符串所属 IoT devices 所在的“IoT Hub > IoT devices”页面，发现每日消息配额已到上限，将无法发送新消息或查看设备列表。出现该问题时，需要等待刷新每日消息配额或者提高每日消息配额上限。



1.6 DeviceSupervisor 2.0 升级注意事项

以下将版本号为 **1.2.X** 的 DeviceSupervisor 简称为 **DS 1.0**，将版本号为 **2.X.X** 的 DeviceSupervisor 简称为 **DS 2.0**，将 DS 1.0 升级到 DS 2.0 需要注意以下事项。

1.6.1 全局说明

1. 仅支持 1.2.9 及以上的 DS 1.0 版本平滑升级至 DS 2.0；
2. 升级后 DS 1.0 的配置信息将会被翻译为 DS 2.0 的配置，升级后不能再回退到 DS 1.0 版本；
3. 平滑升级后, 调用 DS 1.0 **get_tag_config** API 获取到的配置格式有变更；
4. DS 2.0 的配置和 DS 1.0 不一致，DS 1.0 的配置文件无法导入 DS 2.0。

1.6.2 升级后的名词解释

- 设备–> 控制器
- 变量–> 测点
- 告警策略–> 告警规则
- 分组 (轮询间隔) –> 分组 (上报间隔)
- 新增控制器“轮询间隔”

1.6.3 各功能模块更新说明

测点监控

1. 升级后 OPCUA/EtherNetIP 的测点配置信息将被丢弃；
2. 数据类型 **BOOL** 将被转换成 **BIT** 类型。DS2.2 新增 BIT 映射, 可以将 BIT 的测点值 0/1 显示成 False/True；
3. DS 1.0 支持多个相同控制器，DS 2.0 只支持一个；
4. DS 1.0 的**只写**模式变量升级后将变成**可读可写**模式；
5. 平滑升级后，modbus 地址变化：**20000->110000,40000->310000,50000->410000**；
6. 平滑升级后，测点上传模式，realtime -> periodic。

告警

1. 升级后 DS 1.0 存储的历史告警和离线缓存数据将会被清除；
2. DS2 中的告警策略不再支持” 直接使用地址” 策略，会直接在相应控制器下手动添加相应的测点，上传类型是 never；
3. 升级后 DS 1.0 的告警分组被删除，DS 2.0 仅按照告警名称来区分不同的告警，需要注意 DS 1.0 云服务脚本如果引用了空的告警分组，则升级后此脚本可能因为没有告警分组无法运行，请在云服务脚本，手动选择触发源类型。

云服务

1. 升级后 DS 1.0 的阿里云自定义 RRPC 脚本，topic 的响应会失效，但脚本功能可以正常执行，建议使用 DS 2.0 的 API 修改脚本；
2. 平滑升级不支持 GreenGrass Core 相关配置迁移；
3. 平滑升级后，云服务所用证书名称会变成 DS 2.0 默认的证书名称，不影响功能使用；
4. 平滑升级后, 调用 DS1” 获取配置 API” 获取到的配置格式有变更，DS1.0 获取配置的方式将不可用，建议使用 2.0 的 api 重新修改脚本；5、logging 日志输出方式不支持，建议更换为 logger 日志输出方式；6、DS1 的 `save_data` API 接口不再支持。

参数设置

1. 平滑升级后，自定义参数新增 SN 和 MAC 两个参数，DS 1.0 内置的参数 gateway_sn 也会显示添加出来，如果脚本中使用了 gateway_sn，请谨慎删除 gateway_sn，可以使用 DS 2.0 新增的 SN 代替 gateway_sn；
2. 平滑升级后，参数设置取消了历史数据最大条数这一项。

1.6.4 DS2.0.1 及 2.1.1 升级到 DS2.2 注意事项说明

测点监控

1、三菱 3C 协议，地址类型 T-》TN，数据类型 C(16bit)-> CN，数据类型 C(32bit)-> CN 2、三菱编程口协议，地址类型 T-》TN，数据类型 C(16bit)-> CN 3、三菱 3E 协议，地址类型 T-》TN，数据类型 C(16bit)-> CN，数据类型 C(32bit)-> CN

云服务

1、iSCADA 更换了域名，升级后默认域名从 iscada.com.cn->iscada.inhandcloud.cn，如果升级前配置了 iSCADA，会保留之前的域名 2、云服务的 Python 脚本兼容了 DS1.0 和 DS2.0.1 及 2.1.1 的 API，但是我们更建议客户使用 DS2.2 提供的 API