

# IoT lab3: 设备与云平台——基于MQTT的ThingsBoards物联网设备管理

3220102866 陈奕萱

## 1 实验目的和要求

- 掌握 MQTT 协议工作原理；
- 掌握如何搭建、操作、可视化ThingsBoard；
- 掌握如何添加规则引擎；
- 掌握如何数据及结果存储上云；

## 2 实验内容

基于ESP32硬件以及RIOT系统，根据实验手册([https://gitee.com/emnets/emnets\\_experiment/blob/master/part3\\_mqtt\\_thingsboard.md](https://gitee.com/emnets/emnets_experiment/blob/master/part3_mqtt_thingsboard.md))完成以下实验：

1. 设备与云平台管理实验□搭建ThingsBoard物联网平台
2. ESP32设备数据上报到ThingsBoard平台
3. ThingsBoard数据可视化展示
4. 设备数据上云频率可控

## 3 实验背景

- RIOT 操作系统
  - RIOT(<https://github.com/RIOT-OS/RIOT>) 是一个开源的微控制器操作系统，旨在满足物联网(IoT)设备和其他嵌入式设备的需求。它支持一系列通常在物联网(IoT)中发现的设备:8位, 16位和32位微控制器。RIOT基于以下设计原则:节能、实时功能、内存占用小、模块化和统一的API访问，独立于底层硬件(该API提供部分POSIX遵从性)。
- ThingsBoard 介绍
  - ThingsBoard 是一个开源的物联网平台，专注于数据收集、处理、可视化和设备管理。它支持多种行业标准的物联网协议，如 MQTT、CoAP 和 HTTP，能够实现设备的快速连接与管理，并支持云端和本地部署。ThingsBoard 的特点包括：
    - 1)设备管理和数据收集：平台支持多种设备接入协议，方便开发者快速集成和管理各种物联网设备，并提供丰富的数据收集和处理功能。
    - 2)可视化仪表板：ThingsBoard 提供了强大的可视化工具，允许开发者定制各种仪表板，以直观的方式展示设备数据、运行状态和告警信息。
    - 3)规则引擎：通过规则引擎，开发者能够灵活定义业务逻辑，如设备控制、数据转发和告警触发，使物联网应用更加智能和高效。
    - 4)集成和扩展：平台支持与其他系统集成，提供丰富的 API 和插件机制，方便功能扩展和定制。
    - 5)弹性伸缩和高容错性：ThingsBoard 设计考虑到了物联网设备的多样性和数量，能够根据需求进行弹性伸缩，同时采用多种机制保证数据的完整性和可靠性。
    - 6)性能：通过高效的算法和数据处理技术，ThingsBoard 能够快速收集、处理和存储大量物联网数据，确保数据的实时性和准确性。

- 7) 安全性：ThingsBoard 在数据传输和存储方面采取了多种安全措施，确保数据安全和隐私保护，同时支持通过 API 进行身份验证和授权管理。
- ThingsBoard 适用于多种物联网场景，如智能家居、智慧城市、工业自动化、农业智能化等，提供定制化解决方案，帮助用户实现设备的远程监控和管理、数据的可视化展示和分析以及智能化的决策支持，且是完全开源的。

## 4 主要仪器设备

---

- PC
- ESP32-WROOM-32、MPU6050惯性传感器、LED RGB灯。

## 5 实验题目简答

---

### a. 想象一下，MQTT协议在生活中的应用场景有哪些？

#### 1. 智能家居

- **智能灯泡、开关和电器控制：**通过MQTT协议，用户可以使用手机或语音助手远程控制家中的灯光、空调、门锁等设备。设备发送状态信息至MQTT服务器，用户可以在应用程序中实时监控和控制这些设备。
- **智能安防系统：**门窗传感器、监控摄像头、烟雾探测器等设备使用MQTT协议向用户推送安全警报。当门被非法打开时，传感器会通过MQTT服务器推送通知到手机。
- **智能恒温器：**MQTT可以实现远程调整家中的温度设定，恒温器持续发送温度数据并接收来自用户的控制指令，实现节能与智能化管理。

#### 2. 智能农业

- **温湿度监控与调节：**温室大棚中通过传感器检测温度、湿度、土壤湿度等环境数据，使用MQTT协议将数据发送到农民的管理平台，平台根据情况发送调节指令，比如开启喷水系统或通风设备，保持适宜的环境。
- **灌溉系统：**基于MQTT的智能灌溉系统可以根据土壤湿度传感器的数据，自动调节灌溉频率和水量。农民可以远程监控和控制灌溉状态。

#### 3. 智能城市

- **智能交通灯管理：**交通灯与监控摄像头通过MQTT协议实时传输数据，可以根据路况自动调整交通灯的时长，减少交通拥堵。
- **垃圾管理：**垃圾箱内的传感器可以监测垃圾的填充程度，当垃圾箱接近满载时，发送数据到管理平台，通过MQTT通知清洁人员进行处理，实现垃圾回收的自动化和高效化。
- **环境监控：**城市空气质量、噪声水平等数据通过MQTT传输，管理部门可根据实时数据做出决策，比如在污染严重时发布交通限制措施。

#### 4. 医疗健康

- **远程健康监测：**患者的可穿戴设备（如智能手环、血压计、心率监测器等）可以使用MQTT协议将数据传输到医院的监控系统。医生可以实时了解患者的健康状况，并在异常时及时干预。
- **家庭医疗设备：**例如家用血糖仪、血氧仪等设备可以通过MQTT将数据传送到云端，医生和患者都能通过应用程序查看历史数据和实时健康状态。

## 5. 智能汽车

- **车联网：**车辆中的传感器通过MQTT向服务器发送实时信息，如油量、胎压、位置和车速。驾驶员可以通过手机应用监控车辆状况，或者接收车辆的保养提醒和安全警报。
- **共享出行：**共享单车和共享汽车系统通过MQTT协议进行实时的车辆状态更新和位置报告，使用户可以随时通过应用查找附近的可用车辆。

## 6. 工业自动化

- **机器设备状态监测：**工业设备可以通过MQTT协议将设备的温度、压力、振动等信息发送到管理平台，管理者可以实时监控设备运行情况，在发生故障前进行维护。
- **工厂自动化：**在智能工厂中，各个生产设备和传感器通过MQTT实现互联互通，数据传输到中央系统后，系统可以根据生产进度和设备状态自动调整生产节奏，优化资源配置。

## 7. 物流与仓储

- **货物追踪：**通过MQTT协议，物流车辆可以实时传输位置、货物状态、温度和湿度信息。管理系统可以实时更新货物的运输进度，并根据实际情况进行路径优化。
- **智能仓储管理：**仓库中的温湿度传感器可以通过MQTT协议将环境信息发送到仓库管理系统，当条件超出设定范围时，系统可以自动调节或发出警报，确保储存物品安全。

## 8. 智能办公

- **会议室预定与使用监控：**通过MQTT协议，传感器可以监测会议室的使用情况并向服务器发送数据，用户通过手机应用查看会议室是否空闲，系统可以自动更新预定情况。
- **办公环境管理：**空调、照明、遮阳设备等可以通过MQTT实现智能化控制，例如根据办公人员的进出情况自动调节办公环境。

这些场景展示了MQTT在各种生活领域的广泛应用，借助其低延迟、低带宽占用等特点，能够有效地提升设备之间的实时通信和控制能力，实现更加智能化和自动化的生活方式。

## b. MQTT协议的特点是什么，请简述并画出其消息传输模型？

### MQTT协议的特点

MQTT (Message Queuing Telemetry Transport) 是一种轻量级、发布/订阅模式的网络协议，广泛应用于物联网 (IoT) 环境中。它的特点包括：

1. **轻量级：**MQTT协议设计用于带宽受限、计算资源有限的环境。它传输数据时开销非常低，非常适合物联网设备。
2. **发布/订阅模型：**客户端通过发布主题 (topics) 和订阅主题来进行消息交换。消息通过中间的服务器 (称为Broker) 传递，不同客户端之间无需直接通信。
3. **低带宽消耗：**通过压缩和简化消息传输协议，MQTT能够在低带宽网络环境下运行。
4. **QoS等级：**提供三种服务质量 (QoS, Quality of Service) 级别，保证消息传递的可靠性：
  - QoS 0：消息至多传递一次，可能丢失。
  - QoS 1：消息至少传递一次，可能重复。
  - QoS 2：消息仅传递一次，确保不丢失也不重复。
5. **保留消息和遗嘱消息：**
  - **保留消息：**客户端可以发布保留消息，Broker会保存该消息，并在新订阅者订阅时发送。

- **遗嘱消息**: 当客户端意外断开连接时, Broker会为该客户端发布一条预先设定的遗嘱消息, 通知其他订阅者。
6. **持久化会话**: 客户端可以设置持久化会话, 使得在重连时不会丢失在离线时发布的消息。
  7. **支持断线重连**: MQTT支持设备断线后重新连接, 并恢复之前的订阅关系。
  8. **安全性**: 通过TLS/SSL加密来确保消息传输的安全性。

## MQTT消息传输模型

MQTT的消息传输模型基于 **发布/订阅 (Publish/Subscribe)** 模式, 模型的核心组件包括:

1. **客户端 (Client)** : 负责发布消息 (Publisher) 或订阅主题 (Subscriber) 。
2. **Broker**: 负责接收客户端发布的消息, 并根据主题分发给相应的订阅者。

## 消息传输流程:

1. **客户端A (发布者)** 发布一条消息到主题 `/sensor/temp`。
2. **Broker** 接收到消息, 并将该消息传递给所有订阅了该主题的客户端 (客户端B和客户端C) 。
3. **客户端B和客户端C (订阅者)** 通过订阅 `/sensor/temp` 主题接收到该消息。

这个模型让发布者和订阅者之间的通信变得更加灵活和解耦, 发布者不需要知道具体有哪些订阅者, Broker自动完成消息的转发。

## c. MQTT协议报文格式, 并简述其连接和发布详细流程?

### MQTT协议报文格式

MQTT协议中的报文 (Message) 有多种类型, 每种报文都有自己的结构和功能。MQTT协议的报文格式非常轻量, 由固定报头、可变报头和有效载荷组成 (并非所有报文都包含可变报头和有效载荷) 。以下是MQTT协议报文的常见结构:

#### 1. 固定报头 (Fixed Header)

- 所有MQTT报文都有固定报头, 包含报文类型和控制标志, 字段如下:
  - **Message Type**: 4位, 用于标识报文的类型 (例如连接、发布、订阅等) 。
  - **DUP (Duplicate Flag)** : 1位, 标识消息是否是重发消息。
  - **QoS (Quality of Service)** : 2位, 表示消息的服务质量等级 (QoS 0、1或2) 。
  - **RETAIN**: 1位, 标识是否需要保留这条消息。
  - **Remaining Length**: 表示整个报文剩余部分的字节长度。

#### 2. 可变报头 (Variable Header)

- 一些报文需要可变报头, 如 `CONNECT` 报文会包含协议名、协议级别、连接标志和保持连接时间等信息。

#### 3. 有效载荷 (Payload)

- 仅在某些类型的报文中使用, 如 `PUBLISH` 报文中的消息内容。有效载荷的内容视报文类型而定, 例如 `PUBLISH` 报文中的实际消息, `SUBSCRIBE` 报文中的订阅主题。

## MQTT常见报文类型及其格式

1. **CONNECT报文**: 用于客户端向Broker发起连接请求。结构如下:
  - **固定报头**: 指定报文类型为 CONNECT。
  - **可变报头**: 包含协议名、协议级别、连接标志、保持连接时间等。
  - **有效载荷**: 包含客户端ID、用户名、密码等信息（视具体情况而定）。
2. **CONNACK报文**: 服务器对 CONNECT 报文的响应，表示连接是否成功。
  - **固定报头**: 指定报文类型为 CONNACK。
  - **可变报头**: 包含连接确认标志和返回码。
3. **PUBLISH报文**: 用于发送消息。格式如下:
  - **固定报头**: 指定报文类型为 PUBLISH，包含QoS、DUP等标志。
  - **可变报头**: 包含主题名和消息标识符（QoS 1和2时）。
  - **有效载荷**: 消息内容。
4. **PUBACK报文**: QoS 1消息的确认。
  - **固定报头**: 指定报文类型为 PUBACK。
  - **可变报头**: 包含消息标识符。

## MQTT连接流程

1. **客户端发送CONNECT报文**
  - 客户端通过 CONNECT 报文请求与Broker建立连接。该报文中包含协议版本、客户端标识符、可选的用户名、密码和会话设置等。
2. **Broker返回CONNACK报文**
  - Broker接收到 CONNECT 报文后，返回 CONNACK 报文，确认连接成功或者失败。CONNACK 报文中包含连接结果的返回码，表示连接成功（0）或失败（例如：认证失败、服务器不可用等）。
3. **客户端连接成功**
  - 如果 CONNACK 返回码为0，表示连接成功，客户端可以进行后续操作，如发布或订阅消息。

## MQTT发布流程

1. **客户端发送PUBLISH报文**
  - 客户端通过 PUBLISH 报文向Broker发布消息。报文中包含发布的主题和消息内容，此外还包括QoS等级和消息标识符（用于QoS 1和2的消息传递保障）。
2. **Broker处理消息并根据QoS等级采取相应措施**
  - **QoS 0**: Broker接收到消息后立即将其分发给订阅者，无需确认。
  - **QoS 1**: 客户端发送消息后，Broker在成功处理消息后，返回 PUBACK 报文，确认消息已经接收到。
  - **QoS 2**: 客户端和Broker需通过四步握手（PUBLISH、PUBREC、PUBREL、PUBCOMP）确保消息不重复传输且不丢失。
3. **消息分发**
  - Broker将收到的消息分发给订阅了相应主题的客户端。订阅者根据其QoS等级来接收消息。
4. **确认消息（仅QoS 1和QoS 2）**

- 对于QoS 1，订阅者向Broker发送 PUBACK 以确认接收消息。
- 对于QoS 2，订阅者和Broker通过多个报文交互确保消息仅被接收一次。

## d. 请对HTTP、CoAP以及MQTT三个协议进行对比，写出相同点和区别？

### HTTP、CoAP 和 MQTT 三个协议的对比

#### 相同点

1. **应用层协议**：HTTP、CoAP 和 MQTT 都是运行在应用层的协议，用于不同设备之间的通信。
2. **基于TCP/IP或UDP网络**：这三种协议都是基于TCP/IP或UDP网络栈的，尽管它们的具体传输方式有所不同。
3. **消息传输**：它们都用于在客户端与服务器或设备之间进行消息传递，适合物联网等应用场景。
4. **可用于物联网（IoT）**：尽管HTTP主要为Web服务设计，但它也被用于一些简单的物联网场景，而CoAP和MQTT则专门为物联网设计，更加轻量化。

#### 详细比较

##### 1. 消息传输模式

- **HTTP**：采用请求/响应模式，客户端主动向服务器发起请求，服务器响应消息。适合用于客户端/服务器的交互式通信，较为同步。
- **CoAP**：同样采用请求/响应模式，但基于UDP传输，可以采用异步的方式。其消息非常简洁，适用于资源受限的设备。
- **MQTT**：使用发布/订阅模式，客户端通过Broker进行消息转发。客户端通过订阅主题接收消息，发布者向Broker发送消息，这种模式下客户端彼此解耦。

##### 2. 资源占用与效率

- **HTTP**：较为重量级，消息头较大，需要较高的带宽和计算资源，不适合小型物联网设备和低功耗环境。
- **CoAP**：设计为轻量化协议，消息头小且基于UDP传输，占用带宽和计算资源极少，非常适合低功耗、低带宽环境。
- **MQTT**：轻量级协议，消息头比HTTP小，通过TCP传输，适合不可靠网络，且有QoS选项可控制消息的传输可靠性。

##### 3. 安全性

- **HTTP**：可以通过HTTPS（SSL/TLS）来确保通信安全。
- **CoAP**：使用DTLS来确保UDP通信的安全性，适合在受限环境中使用。
- **MQTT**：支持TLS加密，确保通信安全性。

##### 4. 使用场景

- **HTTP**：广泛应用于Web应用程序，适用于需要较高交互和复杂数据传输的场景，如Web服务、浏览器通信等。对于物联网，HTTP常用于简单的请求/响应场景。
- **CoAP**：适用于受限设备和网络环境，如传感器、智能家居设备等低带宽、低功耗的IoT设备通信。
- **MQTT**：适用于大规模物联网设备通信，特别是需要低延迟、低带宽的应用场景，如远程监控、智能农业、智能家居、传感器网络等。

## 总结

HTTP适合成熟的Web应用和复杂数据传输，而CoAP和MQTT专为物联网设计，CoAP更适合小型设备和低功耗场景，而MQTT则凭借发布/订阅机制和QoS控制在大规模物联网环境中具有优势。

## e. 请描绘ThingsBoard系统管理员、租户管理员以及用户之间的关系及作用。

### 1. 系统管理员 (System Administrator)

- **最高权限**: 系统管理员是ThingsBoard平台的最高管理者，拥有对整个系统的完全控制权限。
- 主要职责
  - **创建和管理租户**: 系统管理员可以创建和管理不同的租户 (Tenant)，为每个租户分配资源和权限。
  - **管理平台资源**: 系统管理员可以配置平台级别的设置，例如系统资源、数据存储、安全策略、集群配置等。
  - **监控系统运行状态**: 负责监控平台的整体运行，保证系统稳定性和性能，并可以对系统进行维护和升级。
  - **管理用户访问控制**: 系统管理员负责为租户分配租户管理员账户，进行权限的控制和分配。
  - **配置集成与扩展**: 可以集成其他服务和扩展功能，如规则引擎、集群管理等。

### 2. 租户管理员 (Tenant Administrator)

- **中间权限**: 租户管理员是由系统管理员创建的，负责管理其所在租户中的所有资源和用户。
- 主要职责
  - **管理租户下的设备和资产**: 租户管理员可以创建和管理属于该租户的设备、资产、仪表板和规则链。
  - **用户管理**: 租户管理员可以为租户下创建多个用户，并根据需要分配不同的角色和权限（如普通用户、客户用户等）。
  - **监控与分析**: 租户管理员可以监控设备状态、分析数据，并根据业务需求配置仪表板和警报。
  - **管理租户级别的资源**: 租户管理员可以管理该租户的API密钥、设备配置、传感器数据等。
  - **客户分配与管理**: 租户管理员还可以将设备分配给具体的客户，管理客户对设备的访问权限。

### 3. 普通用户 (Customer/User)

- **有限权限**: 普通用户是租户管理员为客户创建的，权限通常受限于特定设备或资产的查看和操作。
- 主要职责
  - **访问特定设备或仪表板**: 普通用户只能访问租户管理员授权的设备或仪表板，通常用来查看监控数据、执行简单操作或配置设备的部分功能。
  - **监控设备数据**: 普通用户可以查看分配给他的设备数据，并可能通过仪表板查看关键的传感器信息。
  - **操作设备 (如果有权限)**: 根据租户管理员分配的权限，普通用户可能可以执行简单的设备控制操作（如开关设备、调整参数等）。
  - **无需平台管理权限**: 普通用户无权访问平台的管理配置功能，只能使用被授权的资源和功能。

## 关系与作用：

1. **系统管理员创建并管理租户**，可以对多个租户进行管理，同时分配租户管理员，管理整个 ThingsBoard 平台的全局配置。
2. **租户管理员管理租户内的资源**，例如设备、资产、仪表板和用户。同时，租户管理员可以为客户（即普通用户）分配访问权限，控制他们对租户资源的访问。
3. **普通用户只拥有有限的访问权限**，他们可以基于租户管理员的授权，查看或操作特定的设备或数据，但无权管理平台配置。

## f. 请简述规则引擎的作用，有什么应用场景？

### 规则引擎的作用

规则引擎（Rule Engine）是用于根据预定义的规则自动处理数据和事件的系统。在物联网（IoT）平台中，规则引擎可以根据传感器数据、设备状态等触发相应的操作或报警。其核心作用是**简化和自动化决策流程**，从而减少人工干预，提高响应效率。

具体来说，规则引擎的主要作用包括：

1. **事件处理与决策**：根据特定的条件（例如传感器值、设备状态）自动触发相应的动作或通知，帮助系统自动化决策。
2. **数据处理与转换**：可以根据规则对数据进行过滤、转换和处理，将原始数据转化为有用的信息或触发进一步操作。
3. **报警与通知**：当满足特定条件时，规则引擎可以触发报警或通知相关人员（例如设备故障时向维护人员发送警告信息）。
4. **工作流程自动化**：规则引擎可以根据特定事件自动执行一系列操作，如启动设备、关闭设备、发送报告等。
5. **集成与扩展**：规则引擎可以与其他系统或服务进行集成，如数据库、Web 服务、云平台等，实现复杂的自动化操作和跨平台数据处理。

## 应用场景

### 1. 智能家居

- **自动化控制**：基于环境传感器（如温度、湿度、光线）数据，规则引擎可以自动控制家中的灯光、空调、窗帘等设备。例如，当温度高于一定阈值时，自动启动空调。
- **安防系统**：当门窗传感器检测到非法入侵时，规则引擎可以触发报警器，并向用户发送通知。

### 2. 智能农业

- **自动灌溉系统**：规则引擎可以根据土壤湿度传感器的反馈自动启动或停止灌溉系统，确保植物获得适量的水分。
- **环境监控与报警**：当温室中的温度或湿度超过设定的范围时，规则引擎会触发报警，并通知农场主采取相应措施。

### 3. 工业自动化

- **设备监控与维护**：规则引擎可以根据设备的传感器数据实时监控设备状态。例如，当振动传感器检测到异常时，规则引擎可以自动生成维护任务或关闭设备以避免损坏。
- **能效管理**：根据工厂中的用电情况，规则引擎可以在用电高峰期自动降低某些设备的功率，以减少电力消耗。

### 4. 智能交通

- **交通灯管理**: 在智慧城市中，交通灯可以通过规则引擎根据实时的交通流量数据自动调整，减少车辆拥堵。
- **停车管理**: 基于停车场传感器数据，规则引擎可以自动更新停车位的可用状态，并引导车辆进入空余车位。

## 5. 智能物流

- **货物追踪与调度**: 在物流领域，规则引擎可以根据GPS数据和货物状态自动优化配送路径，提升运输效率。
- **冷链物流监控**: 在冷链运输中，规则引擎可以根据温度传感器数据触发警报或调整冷藏设备，以确保货物的质量。

## 6. 医疗监控

- **远程健康监控**: 规则引擎可以实时监测病人的生理参数（如心率、血压），当参数超出正常范围时，自动通知医生或家属。
- **自动诊断**: 结合医疗设备的数据，规则引擎可以根据预定义的医学规则进行自动初步诊断，并提示医生进一步检查。

## 7. 金融领域

- **实时风控**: 规则引擎可以在金融交易中检测异常交易行为，实时触发风控策略，如暂停账户、发送报警等。
- **信用评估**: 根据用户的财务历史、行为数据等，规则引擎可以自动评估用户的信用等级，并决定是否通过贷款申请。

## 总结

规则引擎的核心作用在于**基于事件自动触发动作**，从而提高业务流程的自动化水平，减少人工干预。它适用于**智能家居、工业自动化、农业、医疗、金融**等多个领域，帮助不同场景下的系统实时决策、处理数据、触发操作和发出警报，极大地提升了系统的智能化和自动化水平。

# 6 实验数据记录和处理

---

## 6.1 基础设备控制实验

### a. 代码截图

#### 1. ledcontroller.cpp

同前面的 lab:

```
ledcontroller.cpp > ⚙ change_led_color(uint8_t)
23 /**
24 */
25 void LEDController::change_led_color(uint8_t color){
26     // input your code
27     switch (color)
28     {
29     case COLOR_NONE:
30         gpio_write(led_gpio[0],0); //close red
31         gpio_write(led_gpio[1],0); //close green
32         gpio_write(led_gpio[2],0); //close blue
33         break;
34     case COLOR_RED:
35         gpio_write(led_gpio[0],1); //open red
36         gpio_write(led_gpio[1],0); //close green
37         gpio_write(led_gpio[2],0); //close blue
38         break;
39     case COLOR_GREEN:
40         gpio_write(led_gpio[0],0); //close red
41         gpio_write(led_gpio[1],1); //open green
42         gpio_write(led_gpio[2],0); //close blue
43         break;
44     case COLOR_BLUE:
45         gpio_write(led_gpio[0],0); //close red
46         gpio_write(led_gpio[1],0); //close green
47         gpio_write(led_gpio[2],1); //open blue
48         break;
49     default:
50         break;
51     }
52 }
```

## 2. main\_function.cc

在第一次报错 201 时修改了参数：

```
29 // Globals, used for compatibility with Arduino-style sketches.
30 namespace {
31     const tflite::Model* model = nullptr;
32     tflite::MicroInterpreter* interpreter = nullptr;
33     TfLiteTensor* input = nullptr;
34     TfLiteTensor* output = nullptr;
35
36     // Create an area of memory to use for input, output, and intermediate arrays.
37     // Finding the minimum value for your model may require some trial and error.
38     constexpr int kTensorArenaSize = 9 * 1024;
39     uint8_t tensor_arena[kTensorArenaSize];
40 } // namespace
```

(以及在此次实验中我选择了 mlp 模型)

## 3. main.cpp

修改了连接的热点的账号密码，和新的 ip 地址：

```
58 #define BUF_SIZE 1024
59 #define MQTT_VERSION_V311 4 /* MQTT v3.1.1 version is 4 */
60 #define COMMAND_TIMEOUT_MS 4000
61
62 string DEFAULT_MQTT_CLIENT_ID = "esp32_test";
63 string DEFAULT_MQTT_USER = "esp32";
64 string DEFAULT_MQTT_PWD = "esp32";
65 // Please enter the IP of the computer on which you have ThingsBoard installed.
66 string DEFAULT_IPV4 = "192.168.43.59";
67 string DEFAULT_TOPIC = "v1/devices/me/telemetry";
68
```

在报错 exception 和 overflow 后修改了 `led_thread` 和 `motion_thread` 的栈大小，并且为了方面后续参数上传将变量 `mpu` 改为全局静态变量：

```
103 #define g_acc (9.8)
104 #define class_num (4)
105 #define SAMPLES_PER_GESTURE (10)
106 #define THREAD_STACKSIZE (THREAD_STACKSIZE_IDLE)
107 static char stack_for_led_thread[800];
108 static char stack_for_motion_thread[2000];
109 static int current_motion = 0;
110 float threshold = 0.7;
111 static MPU6050 mpu;
```

led 线程设计同 lab3 无修改：

```
129 void *_led_thread(void *arg)
130 {
131     (void) arg;
132     LEDController led(LED_GPIO_R, LED_GPIO_G, LED_GPIO_B);
133     led.change_led_color(0);
134     while(1){
135         // Input your codes
136         // Wait for a message to control the LED
137         // Display different light colors based on the motion state of the device.
138         msg_t msg;
139         msg_receive(&msg);
140         led_state = msg.content.value;
141         if(led_always == 5){ // change led according to moving state
142             if (msg.content.value == Stationary) {
143                 led.change_led_color(COLOR_NONE);
144                 printf("[led_thread]: led turn off!\n");
145             } else if (msg.content.value == Tilted) {
146                 led.change_led_color(COLOR_RED);
147                 printf("[led_thread]: led turn red!\n");
148             } else if (msg.content.value == Rotating) {
149                 led.change_led_color(COLOR_BLUE);
150                 printf("[led_thread]: led turn blue!\n");
151             } else if (msg.content.value == Moving) {
152                 led.change_led_color(COLOR_GREEN);
153                 printf("[led_thread]: led turn green!\n");
154             } else {
155                 led.change_led_color(COLOR_YELLOW);
156                 printf("[led_thread]: led turn yellow!\n");
157             }
158         }
159         else if(led_always == 0){
160             led.change_led_color(COLOR_NONE);
161             printf("[led_thread]: led stay none\n");
162         }
163         else if(led_always == 1){
164             led.change_led_color(COLOR_RED);
165             printf("[led_thread]: led stay red\n");
166         }
167     }
168 }
```

IMU 传感器数据处理同 lab1 无修改：

```
180     float gyro_fs_convert = 1.0;
181     float accel_fs_convert;
182     static int collect_interval_ms = 20;
183     static int predict_interval_ms = 500;
184
185     void get_imu_data(MPU6050 mpu, float *imu_data){
186         int16_t ax, ay, az, gx, gy, gz;
187         for(int i = 0; i < SAMPLES_PER_GESTURE; ++i)
188     {
189         /* code */
190         delay_ms(collect_interval_ms);
191         mpu.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);
192         imu_data[i*6 + 0] = ax / accel_fs_convert;
193         imu_data[i*6 + 1] = ay / accel_fs_convert;
194         imu_data[i*6 + 2] = az / accel_fs_convert;
195         imu_data[i*6 + 3] = gx / gyro_fs_convert;
196         imu_data[i*6 + 4] = gy / gyro_fs_convert;
197         imu_data[i*6 + 5] = gz / gyro_fs_convert;
198     }
199 }
200 }
```

motion 线程设计在 lab3 的基础上添加 log 记录每次 `mqtt_interval_ms` 数据:

```
201     void *_motion_thread(void *arg)
202 {
203     // Calculate accelerometer conversion factor
204     accel_fs_convert = 32768.0 / accel_fs_real;
205     float imu_data[SAMPLES_PER_GESTURE * 6] = {0};
206     int data_len = SAMPLES_PER_GESTURE * 6;
207     delay_ms(200);
208     // Main loop
209     int ret = 0;
210     string motions[class_num] = {"Stationary", "Tilted", "Rotating", "Moving"};
211     while (1) {
212         delay_ms(predict_interval_ms);
213         // Log the delay between predictions
214         LOG_INFO("[MOTION_THREAD] Prediction gap: %d ms\n", predict_interval_ms);
215         // Log the mqtt interval data
216         LOG_INFO("[MQTT] mqtt interval data: %d ms\n", mqtt_interval_ms);
217         // Read sensor data
218         get_imu_data(mpu, imu_data);
219         // Log the current threshold before prediction
220         LOG_INFO("[MOTION THREAD] Using threshold: %.2f\n", threshold);
221         ret = predict(imu_data, data_len, threshold, class_num);
222         // Send message to LED thread based on motion
223         msg_t msg;
224         msg.content.value = ret;
225         msg_send(&msg, _led_pid); // Send message to LED control thread
226         // Log the prediction result
227         LOG_INFO("[MOTION_THREAD] Prediction result: %d, %s\n", ret, motions[ret].c_str());
228         // Update global motion state
229         current_motion = ret;
230         LOG_INFO("Predict: %d, %s\n", ret, motions[ret].c_str());
231     }
232     return NULL;
233 }
```

uuid 设计在 lab3 的基础上添加一条特性，实现 `mqtt_interval_ms` 的读取和写入：

```

271 // input your code, 自定义想要的UUID
272 /* UUID = 1bcce38b3-d137-48ff-a13e-033e14c7a335 */
273 static const ble_uuid128_t gatt_svr_svc_rw_demo_uuid
274 = {{128}, {0x15, 0xa3, 0xc7, 0x14, 0x3e, 0x03, 0x3e, 0xa1, 0xff,
275 0x48, 0x37, 0xd1, 0xb3, 0x38, 0xce, 0x1b}};
276 //35f21ed readwrite
277 static const ble_uuid128_t gatt_svr_chr_led_uuid
278 = {{128}, {0x62, 0x17, 0x99, 0x7e, 0x50, 0x27, 0x38, 0xba, 0x3b,
279 0x4f, 0x70, 0x30, 0x86, 0x83, 0xf2, 0x35}};
280 //0effmotion read
281 static const ble_uuid128_t gatt_svr_chr_motion_uuid
282 = {{128}, {0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88, 0x99,
283 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff, 0x00}};
284 //0201threshold readwrite
285 static const ble_uuid128_t gatt_svr_chr_threshold_uuid
286 = {{128}, {0x13, 0x24, 0x35, 0x46, 0x57, 0x68, 0x79, 0x8a, 0x91,
287 0xac, 0xbd, 0xce, 0xdf, 0xf0, 0x01, 0x02}};
288 //0302frequency readwrite
289 static const ble_uuid128_t gatt_svr_chr_frequency_uuid
290 = {{128}, {0x14, 0x25, 0x36, 0x47, 0x58, 0x69, 0x7a, 0x8b, 0x92,
291 0xad, 0xbe, 0xcf, 0xe0, 0xf1, 0x02, 0x03}};
292 //0301mqtt readwrite
293 static const ble_uuid128_t gatt_svr_chr_mqtt_uuid
294 = {{128}, {0x14, 0x25, 0x36, 0x47, 0x58, 0x69, 0x7a, 0x8b, 0x92,
295 0xad, 0xbe, 0xcf, 0xe0, 0xf1, 0x01, 0x03}};
296

```

此部分同 uuid 的增加而修改：

```

302 static const struct ble_gatt_svc_def gatt_svr_svcs[] = {
303 /*
304 {
305     /* Service: Read/Write Demo */
306     .type = BLE_GATT_SVC_TYPE_PRIMARY,
307     .uuid = (ble_uuid_t*) &gatt_svr_svc_rw_demo_uuid.u,
308     .characteristics = (struct ble_gatt_chr_def[]) {
309         /* Characteristic: Read/Write Demo write */
310         .uuid = (ble_uuid_t*) &gatt_svr_chr_led_uuid.u,
311         .access_cb = gatt_svr_chr_access_rw_demo,
312         .flags = BLE_GATT_CHR_F_READ | BLE_GATT_CHR_F_WRITE,
313         // .flags = BLE_GATT_CHR_F_READ | BLE_GATT_CHR_F_WRITE_NO_RSP,
314     }, {
315         .uuid = (ble_uuid_t*) &gatt_svr_chr_motion_uuid.u,
316         .access_cb = gatt_svr_chr_access_rw_demo,
317         .flags = BLE_GATT_CHR_F_READ,
318     }, {
319         .uuid = (ble_uuid_t*) &gatt_svr_chr_threshold_uuid.u,
320         .access_cb = gatt_svr_chr_access_rw_demo,
321         .flags = BLE_GATT_CHR_F_READ | BLE_GATT_CHR_F_WRITE,
322     }, {
323         .uuid = (ble_uuid_t*) &gatt_svr_chr_frequency_uuid.u,
324         .access_cb = gatt_svr_chr_access_rw_demo,
325         .flags = BLE_GATT_CHR_F_READ | BLE_GATT_CHR_F_WRITE,
326     }, {
327         .uuid = (ble_uuid_t*) &gatt_svr_chr_mqtt_uuid.u,
328         .access_cb = gatt_svr_chr_access_rw_demo,
329         .flags = BLE_GATT_CHR_F_READ | BLE_GATT_CHR_F_WRITE,
330     }, {
331         0, /* No more characteristics in this service */
332     },
333     0, /* No more services */
334 },
335 {
336     0,
337 },
338 },
339 },
340 };

```

特性设计在 lab3 基础上添加了 `mqtt_interval_ms` 的读和写：

```
343     struct ble_gatt_access_ctxt *ctxt, void *arg)
350     switch(ctxt->op){
352         if (ble_uuid_cmp(ctxt->chr->uuid, &gatt_svr_chr_motion_uuid.u) == 0)
356             LOG_INFO("[READ] current_motion = %d\n", current_motion);
357             return rc;
358     }
359     else if (ble_uuid_cmp(ctxt->chr->uuid, &gatt_svr_chr_led_uuid.u) == 0){
360         //读取当前的led灯状态
361         rc = os_mbuf_append(ctxt->om, &current_motion,
362                             sizeof(current_motion));
363         LOG_INFO("[READ] led_status = %d\n", current_motion);
364         return rc;
365     }
366     else if (ble_uuid_cmp(ctxt->chr->uuid, &gatt_svr_chr_threshold_uuid.u) == 0) {
367         // 读取当前阈值
368         rc = os_mbuf_append(ctxt->om, &threshold, sizeof(threshold));
369         LOG_INFO("[READ] threshold = %.2f\n", threshold);
370         return rc;
371     }
372     else if (ble_uuid_cmp(ctxt->chr->uuid, &gatt_svr_chr_frequency_uuid.u) == 0) {
373         // 读取数据采集频率
374         rc = os_mbuf_append(ctxt->om, &collect_interval_ms,
375                             sizeof(collect_interval_ms));
376         LOG_INFO("[READ] collect_interval_ms = %d\n",
377                 collect_interval_ms);
378         return rc;
379     }
380     else if (ble_uuid_cmp(ctxt->chr->uuid, &gatt_svr_chr_mqtt_uuid.u) == 0) {
381         // 读取mqtt频率
382         rc = os_mbuf_append(ctxt->om, &mqtt_interval_ms,
383                             sizeof(mqtt_interval_ms));
384         LOG_INFO("[READ] mqtt_interval_ms = %d\n",
385                 mqtt_interval_ms);
386         return rc;
387     }
388     break;
```

```

343     struct ble_gatt_access_ctxt *ctxt, void *arg)
350     switch(ctxt->op){
389         case BLE_GATT_ACCESS_OP_WRITE_CHR:
390             uint16_t om_len;
391             om_len = OS_MBUF_PKTLEN(ctxt->om);
392             if (ble_uuid_cmp(ctxt->chr->uuid, &gatt_svr_chr_threshold_uuid.u) == 0) {
393                 // 写入新的阈值
394                 rc = ble_hs_mbuf_to_flat(ctxt->om, &threshold, sizeof(threshold),
395                 &om_len);
396                 LOG_INFO("[WRITE] new value of threshold: %.2f\n", threshold);
397                 //predict_interval_ms[om_len] = '\0';
398             } else if (ble_uuid_cmp(ctxt->chr->uuid, &gatt_svr_chr_frequency_uuid.u) == 0) {
399                 // 写入新的采集频率
400                 rc = ble_hs_mbuf_to_flat(ctxt->om, &collect_interval_ms,
401                 sizeof(collect_interval_ms), &om_len);
402                 LOG_INFO("[WRITE] new value of collect_interval_ms: %d\n",
403                 collect_interval_ms);
404                 //collect_interval_ms[om_len] = '\0';
405             } else if (ble_uuid_cmp(ctxt->chr->uuid, &gatt_svr_chr_mqtt_uuid.u) == 0) {
406                 // 写入新的mqtt频率
407                 rc = ble_hs_mbuf_to_flat(ctxt->om, &mqtt_interval_ms,
408                 sizeof(mqtt_interval_ms), &om_len);
409                 LOG_INFO("[WRITE] new value of mqtt_interval_ms: %d\n",
410                 mqtt_interval_ms);
411                 //collect_interval_ms[om_len] = '\0';
412             }else if (ble_uuid_cmp(ctxt->chr->uuid, &gatt_svr_chr_led_uuid.u) == 0) {
413                 // 写led灯常亮
414                 rc = ble_hs_mbuf_to_flat(ctxt->om, &led_always,
415                 sizeof(led_always), &om_len);
416                 if(led_always != 5){
417                     msg_t msg;
418                     msg.content.value = led_always;
419                     msg_send(&msg, _led_pid);
420                 } else{
421                     LOG_INFO("[WRITE] led continuing prediction: \n");
422                 }
423             }

```

6个传感器数据、当前运动状态和当前 r, g, b 状态上传:

```

snprintf(json, 200, "{\"ax\":%.02f, \"ay\":%.02f, \"az\":%.02f, \"gx\":%.02f,
\"gy\":%.02f, \"gz\":%.02f, \"current_motion\":\"%d\", \"led_r\":%d,
\"led_g\":%d, \"led_b\":%d}",
           ax/accel_fs_convert, ay/accel_fs_convert, az/accel_fs_convert,
gx/gyro_fs_convert, gy/gyro_fs_convert, gz/gyro_fs_convert, current_motion,
led_r, led_g, led_b);

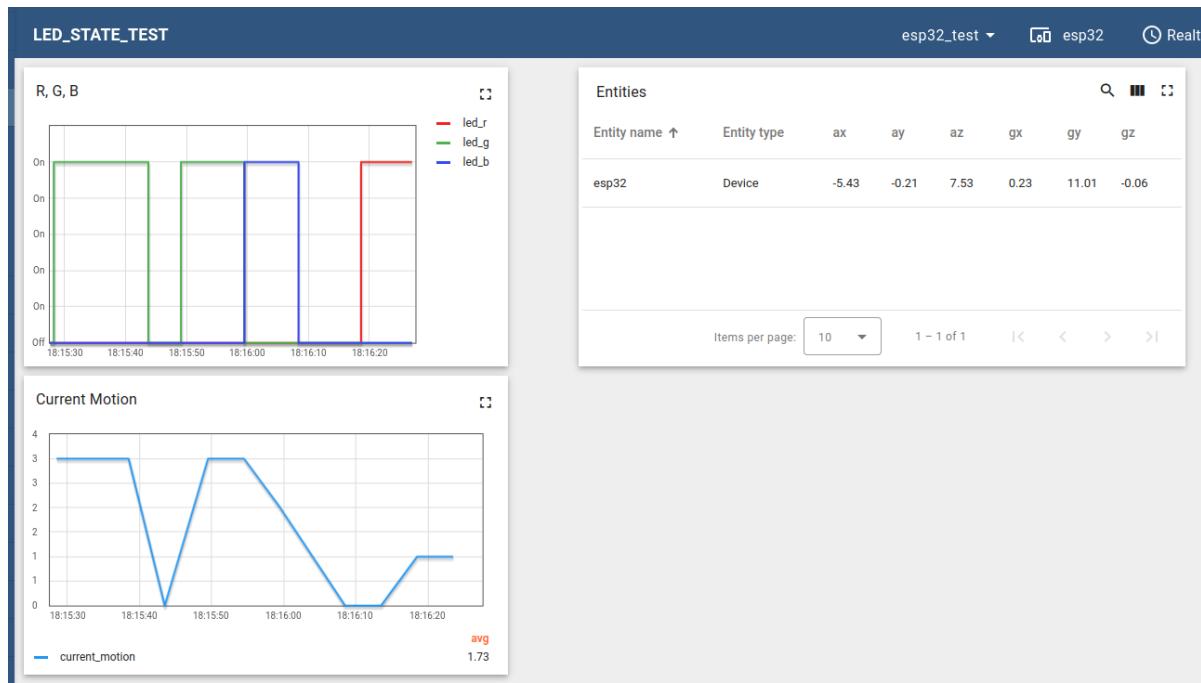
```

```

483 int mqtt_pub(void)
484 {
485     int16_t ax, ay, az, gx, gy, gz;
486     mpu.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);
487     int16_t led_r = 0, led_g = 0, led_b = 0;
488     if(led_state == 0){
489         led_r = 0;
490         led_g = 0;
491         led_b = 0;
492     }else if(led_state == 1){
493         led_r = 1;
494         led_g = 0;
495         led_b = 0;
496     }else if(led_state == 2){
497         led_r = 0;
498         led_g = 0;
499         led_b = 1;
500     }else if(led_state == 3){
501         led_r = 0;
502         led_g = 1;
503         led_b = 0;
504     }
505
506     enum QoS qos = QOS0;
507     MQTTMessage message;
508     message.qos = qos;
509     message.retained = IS_RETAINED_MSG;
510     led_state = !led_state;
511     char json[200];
512     sprintf(json, 200, "{\"ax\":%.02f, \"ay\":%.02f, \"az\":%.02f, \"gx\":%.02f, \"gy\":%.02f}",
513             | | | | ax/accel_fs_convert, ay/accel_fs_convert, az/accel_fs_convert, gx/gyro_fs_convert,
514             printf("[Send] Message:%s\n", json);
515     message.payload = json;
516     message.payloadlen = strlen((char *)message.payload);
517 }
```

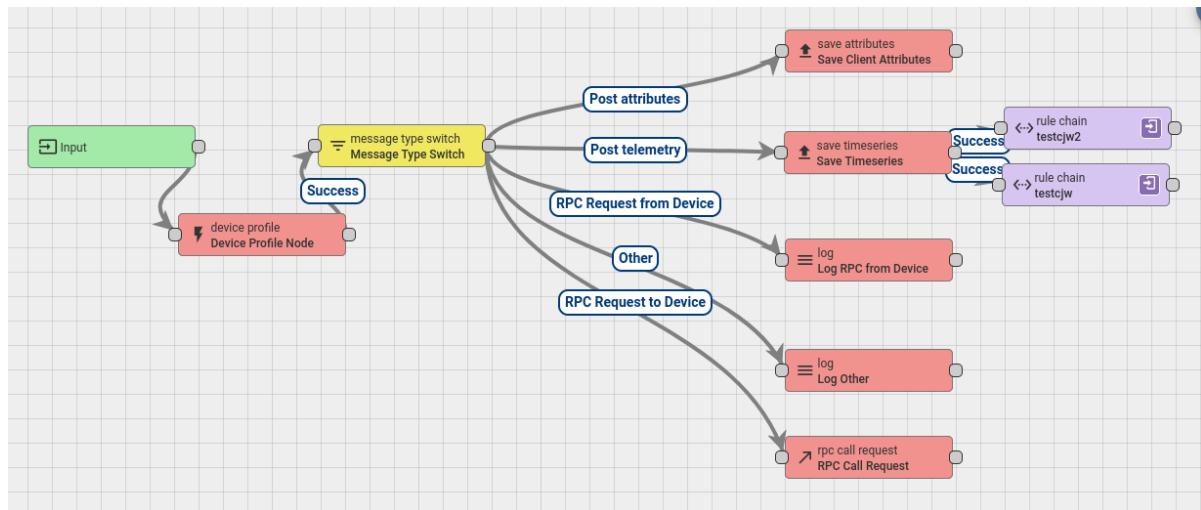
## b. ThingBoard 可视化数据展示结果图

如图，可以看到，在改变设备状态时 led 灯 RGB 数据的改变、对当前设置的 mqtt\_interval 下最新状态的 IMU 传感器数据和设备当前状态的预测结果：



c. 若添加规则引擎，需要展示RGB原始数据和RGB代码两个结果或者设备姿态和警报？否则，则忽略。

总规则链设计：



RGB 代码：

```
var rgbCode="";
if(msg.current motion === 0){
    rgbCode+="000000";
} else if(msg.current motion === 1){
    rqbcode+="FF0000";
} else if(msg.current motion === 2){
    rqbCode+="00FF00";
} else{
    rgbCode+="0000FF";
}
msg.rgbCode=rgbcode;
return {msg: msg, metadata: metadata, msgType: msgType};
```

RGB 原始数据：

# esp32

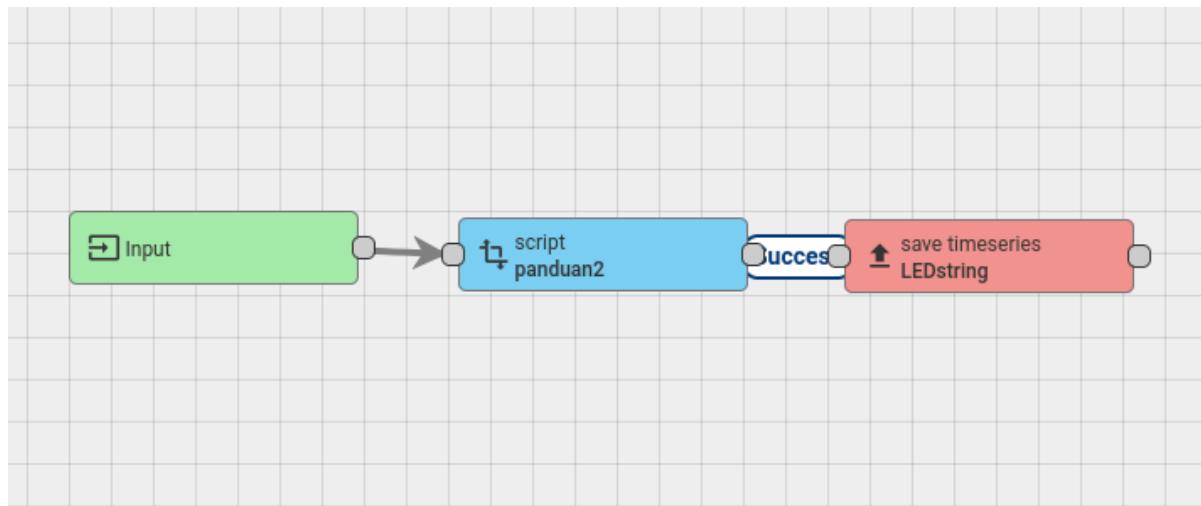
## Device details

Details    Attributes    **Latest telemetry**    Alarms    Events    Relations    Audit Logs    Version control

### Latest telemetry

Last update time	Key ↑	Value
2024-10-28 23:59:41	gz	2.29
2024-10-28 23:59:41	led_b	1
2024-10-28 23:59:41	led_g	0
2024-10-28 23:59:41	led_r	0
2024-10-28 00:28:59	r_led_state	0
2024-10-28 23:59:41	rgbCode	00FF00
2024-10-26 18:45:57	temperature	25

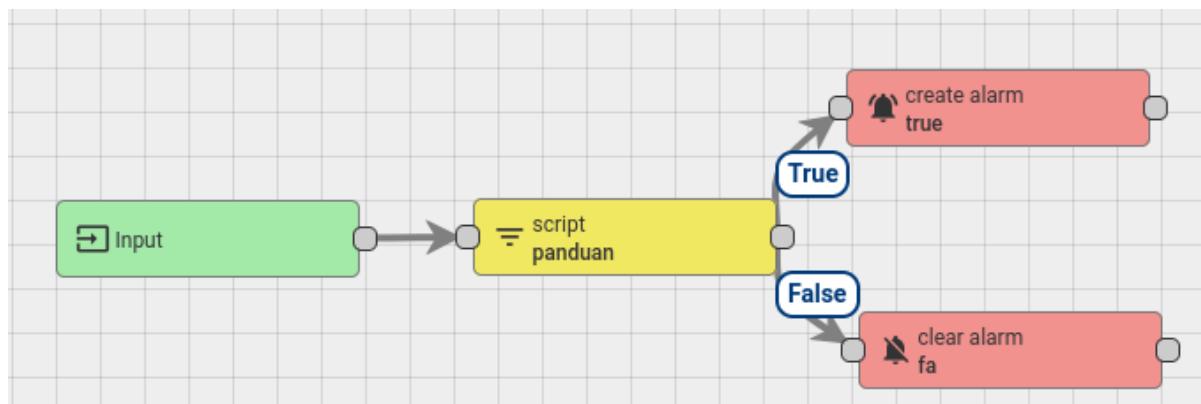
RGB 规则链设计：



tilted 状态警报代码：

```
return msg.current motion === 1;
```

tilted 状态警报规则链设计：



tilted 状态发生警报：

Filter: Active		For all time					
Created time ↓	Originator	Type	Severity	Assignee	Status	Details	
2024-10-28 23:15:30	esp32	General Alarm	Critical	Unassigned	Active Unacknowledged	...	
2024-10-28 23:15:29	esp32	General Alarm	Critical	Unassigned	Active Unacknowledged	...	

## 7 实验结果与分析

### 基于该实验，总结设备数据上云平台以及数据可视化的基本流程。

在物联网（IoT）应用中，设备数据上云平台以及实现数据可视化的基本流程通常包括以下几个步骤：

#### 1. 设备连接与数据采集

- 连接设备**：首先，需要将物理设备（如传感器、智能设备等）连接到网络。设备通过WIFI、以太网、4G/5G或其他通信方式接入网络。
- 数据采集**：设备通过内置传感器或外部传感器采集环境数据（如温度、湿度、光照强度、设备状态等）。这一步需要设备具备一定的计算能力来收集并处理原始数据。

#### 2. 设备数据传输

- 选择通信协议**：  
设备将采集到的数据传输到云平台，常用的通信协议包括MQTT、CoAP、HTTP等。协议选择取决于数据的传输频率、网络条件以及设备资源的限制。
  - MQTT**：轻量级，支持发布/订阅模式，适合低带宽、资源受限的环境。
  - HTTP**：广泛应用，适合请求/响应模式的数据传输。
  - CoAP**：用于低带宽和功耗受限的设备，基于UDP传输。
- 加密与认证**：数据传输过程中，需要使用安全机制（如TLS/SSL）来确保数据的安全性，同时设备需要通过认证才能向云平台发送数据。

#### 3. 云平台数据接收与处理

- 数据接收**：云平台通过接入层（例如MQTT Broker或HTTP服务器）接收设备发送的数据。此时，设备的数据已经通过网络发送至云端。
- 数据存储**：接收到的数据会被存储在云平台的数据库中，通常会采用时序数据库（如InfluxDB）来存储时间相关的数据，或者关系型数据库（如MySQL、PostgreSQL）来存储其他结构化数据。
- 数据处理**：云平台会对接收到的数据进行初步处理，如清洗、格式转换、异常检测等。部分场景下还可能使用规则引擎进行自动化处理和响应（如发送报警、触发动作等）。

#### 4. 数据可视化

- 选择可视化工具**：  
常用的可视化工具包括ThingsBoard、Grafana、Kibana等。这些工具通过连接云平台的数据存储，能够提供丰富的图表、仪表盘等数据展示形式。
  - 仪表板设计**：创建仪表板来显示数据，选择适合的图表类型（如折线图、柱状图、饼图等）以展示设备状态、历史数据或实时监控信息。
  - 实时数据展示**：通过WebSocket或其他实时数据传输技术，可以实现数据的实时更新，动态展示设备的运行状态。

- **报警与通知**: 如果数据触发了某些预设的规则(如传感器数据超出设定阈值), 可视化平台会实时显示报警, 并通过邮件、短信或其他渠道通知相关人员。

## 5. 控制与反馈(可选)

- **远程控制**: 通过可视化平台, 用户可以对设备进行远程控制。例如, 可以通过界面发送命令控制设备的开关、调节参数等。控制指令通常通过云平台再发送回设备, 设备执行指令并返回执行结果。
- **反馈数据**: 设备执行完远程指令后, 可能会反馈执行结果或状态数据回到云端, 再通过可视化界面展示给用户。

## 6. 数据分析与优化

- **历史数据分析**: 用户可以查看设备历史数据, 分析趋势、设备使用情况、能效管理等。分析可以帮助优化设备性能、预测故障或进行设备维护。
- **数据导出与报告生成**: 可视化平台通常支持数据导出功能, 用户可以将特定时间段的数据导出用于报告生成或进一步分析。

## 总结

1. **设备连接与数据采集**: 通过传感器采集数据并将其传输至云平台。
2. **数据传输**: 使用适当的通信协议将数据发送至云端, 保证数据安全性和可靠性。
3. **数据接收与处理**: 云平台接收数据并对其进行处理和存储。
4. **数据可视化**: 通过可视化工具展示设备数据, 支持实时监控、历史数据查询和报警通知。
5. **控制与反馈**: 可视化平台支持对设备的远程控制和反馈。
6. **数据分析**: 利用历史数据进行趋势分析、优化设备性能。

通过这一流程, 物联网设备的数据可以方便地上传到云平台, 并实现实时监控、远程控制和数据分析,从而提高设备管理的智能化水平。