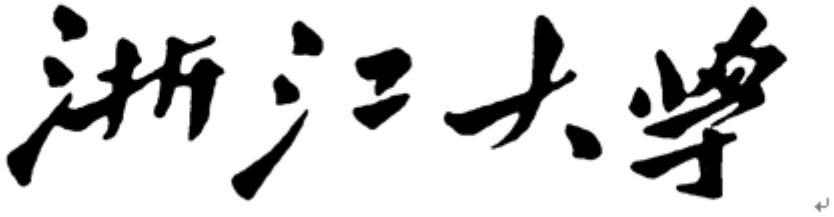


LoT Lab4:基于设备与云平台：基于MQTT的ThingsBoards物联网设备管理

姓名：蔡佳伟

学号：3220104519

一、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；

二、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)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

三、 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。

四、主要仪器设备

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

五、 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?

MQTT (Message Queuing Telemetry Transport) 协议是一种轻量级、发布/订阅模式的消息传输协议，特别适合低带宽、不稳定网络连接和需要低功耗的设备。因此，它在物联网（IoT）和一些场景中非常实用，以下是几个生活中的常见应用场景：

1. 智能家居系统

- **应用细节**：通过MQTT，家中的设备（如灯、空调、门锁等）可以相互通信，并接收来自手机、平板等的指令。
- **场景示例**：用户可以通过手机App或语音助手打开家里的灯光、调节温度，甚至远程监控门锁状态。这些设备通过MQTT服务器（Broker）来接收和发送指令。
- **优势**：低延迟、节省电量；而且，即使网络较差也能保证一定的响应。

2. 智慧城市管理

- **应用细节**：在智慧城市中，很多设施需要互联以提升资源利用率，比如智能路灯、垃圾桶、水电监测等。
- **场景示例**：通过MQTT协议，智慧路灯可以根据车辆流量和时间自动调整亮度，垃圾桶可以定期发送其满溢情况的通知，让环卫部门高效调度清理。
- **优势**：减少数据通信量、实时监测，帮助管理部门在突发状况下快速响应。

3. 车联网和智能交通系统

- **应用细节**：车辆可以通过MQTT和其他设备（如路况监控设备、信号灯等）交换信息，提升交通效率和安全性。
- **场景示例**：车辆可以定期发送位置、速度等信息给MQTT Broker，再由Broker将信息转发至交通管理中心，进行车流监控和调度。还可以实现碰撞预警、自动避让等功能。
- **优势**：高效的数据传输使得车联网应用更加精准，且支持大量设备同时连接。

4. 工业物联网 (IIoT) 监控

- **应用细节**：在工业场景中，设备需要实时监控状态以提高生产效率，降低故障率。MQTT被广泛用于传感器和控制系统的通信。
- **场景示例**：一个工厂里的温度传感器、压力传感器等可以通过MQTT向中央控制系统报告数据，控制系统根据数据及时调节生产参数。
- **优势**：延迟低、实时性强，且可以在工业环境的复杂网络条件下保持连接。

5. 远程健康监护

- **应用细节**：健康监护设备（如心率监测仪、血糖仪等）通过MQTT协议定期将数据发送给监护中心或家属设备。
- **场景示例**：如果患者的血压监控设备检测到异常，设备会通过MQTT协议立即向监护中心发送警报，并通知医护人员采取措施。
- **优势**：延迟低、数据更新及时，且协议轻量适合长时间监控。

6. 农业物联网

- **应用细节**：农场中的传感器（如湿度、土壤温度、光照强度等）可以通过MQTT协议传输数据到中央服务器，实现精准农业管理。
- **场景示例**：自动灌溉系统根据湿度传感器的数据，调整灌溉频率和水量；如果MQTT监测到异常条件（如土壤干燥或温度过高），会及时触发报警或自动启动设备。
- **优势**：降低人工成本，提高农业资源的使用效率。

7. 环境监测

- **应用细节**：在城市和自然保护区，通过布置各种传感器（如空气质量、噪声、水质监测等），定期向中央管理系统汇报数据。
- **场景示例**：环境监测系统可以实时采集空气质量数据，当空气质量下降到一定标准，系统会立即通知相关部门采取措施。
- **优势**：灵活且低成本，支持设备在低网络质量下稳定通信。

MQTT 协议的轻量、低带宽需求和支持发布/订阅的特性，使其在需要实时数据传输、设备众多且网络环境较差的生活场景中极具优势，广泛应用于日常生活的各个方面。

b. MQTT协议的特点是什么，请简述并画出其消息传输模型?

MQTT协议的特点及其消息传输模型如下：

MQTT协议的特点

1. **轻量级**：MQTT协议的设计初衷就是为了在低带宽、高延迟或不可靠的网络环境中运行。因此，它具有较小的数据包开销，适合资源受限的设备。
2. **发布/订阅模式**：基于发布/订阅（Publish/Subscribe）通信模型，使得消息传输更加灵活。设备可以只订阅自己关心的主题（Topic），并从Broker中获取相关信息。
3. **低带宽和低功耗**：MQTT的数据开销小，适合带宽较低的网络环境。此外，它对设备功耗需求也低，适合物联网中需要长时间运行的设备。
4. **消息质量控制**：支持3种消息QoS等级（QoS 0、QoS 1和QoS 2），分别保证“最多一次”、“至少一次”和“仅一次”的消息传递，满足不同的可靠性需求。
5. **断线重连**：支持设备断开连接后重新连接，并可以设置消息保留（Retained）和持久会话（Persistent Session），在断线重连时恢复会话状态。

MQTT消息传输模型

MQTT的消息传输模型包括三个核心组件：

- **发布者（Publisher）**：发送消息的设备或客户端，负责将消息发送到特定的主题（Topic）。
- **订阅者（Subscriber）**：接收消息的设备或客户端，通过订阅某个主题来获取消息。
- **代理（Broker）**：消息的中介，负责接收发布者发送的消息并分发给订阅了该主题的订阅者。

以下是MQTT消息传输模型的简图：

在图中：

1. **Publisher**将消息发布到某个Topic。
2. **Broker**接收Publisher的消息并将其路由到对应Topic的所有订阅者。
3. **Subscriber**从Broker订阅自己感兴趣的Topic并接收相应消息。

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协议的报文结构简单，由固定报头、可变报头和有效载荷三部分组成，每个报文的结构如下：

1. **固定报头**：所有MQTT报文都有一个固定报头，包含两个字节。第一个字节包括报文类型（如CONNECT、PUBLISH等）、标志位；第二字节及后续字节表示剩余长度字段。
2. **可变报头**：根据报文类型的不同，可变报头的内容会有所不同。例如，在连接报文（CONNECT）中，包含协议名称、协议级别、连接标志和保持连接时间等信息。
3. **有效载荷（Payload）**：可变长度的数据区，主要用于传输主题、消息内容等数据。在CONNECT报文中，有效载荷包含客户端标识符、用户名、密码和遗嘱消息等。

MQTT协议连接流程（CONNECT流程）

MQTT的连接流程主要分为以下几个步骤：

1. **客户端发起连接（CONNECT）**：
   - 客户端向Broker发送一个CONNECT报文，包含协议级别、客户端ID、用户名、密码、遗嘱（Will）消息等信息。
   - 客户端可以选择在CONNECT报文中指定一个“保持连接”时间（Keep Alive），Broker在该时间内没有收到客户端消息则会断开连接。
2. **Broker响应（CONNACK）**：
   - Broker接收到CONNECT报文后，验证客户端信息，如果通过验证，则返回一个CONNACK报文表示连接成功，否则返回失败代码。
   - 如果客户端和Broker之间启用了保持会话（Session Persistency），则在客户端重新连接时，Broker会恢复上次的会话状态（订阅列表等）。

MQTT协议消息发布流程（PUBLISH流程）

MQTT的消息发布和订阅流程可以总结为以下几步：

1. **客户端发布消息（PUBLISH）**：
   - Publisher客户端发送一个PUBLISH报文至Broker。该报文包含主题（Topic）、消息内容和QoS（服务质量）等级等信息。
   - QoS等级可以是0（最多一次）、1（至少一次）或2（只有一次），根据QoS等级的不同，Broker和客户端会进行不同的确认操作。
2. **Broker处理消息并分发**：
   - Broker收到PUBLISH报文后，将消息存储并根据主题转发至所有订阅了该主题的订阅者。
   - 如果消息的QoS等级较高（如1或2），则Broker需要向发布者确认，以确保消息传递的可靠性。
3. **订阅者接收消息（SUBSCRIBE流程）**：
   - 订阅者发送一个SUBSCRIBE报文至Broker，声明自己订阅的主题。
   - Broker根据SUBSCRIBE报文将相应主题的消息分发至该订阅者。

以下是每个报文在流程中的关键作用：

- **CONNECT/CONNACK**：用于客户端连接Broker。
- **PUBLISH**：用于发布者发送消息到指定的主题。
- **SUBSCRIBE**：用于订阅者声明自己感兴趣的主题。
- **PUBACK**（QoS 1）和**PUBREC/PUBREL/PUBCOMP**（QoS 2）：用于确认QoS等级消息的传输情况。

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是三种常见的网络通信协议，它们各自适用于不同的应用场景，但也有一些相似之处。以下是这三者的对比，包括相同点和主要区别：

相同点

1. **网络传输协议**：三者都使用TCP/IP协议栈进行通信，其中HTTP和MQTT基于TCP，CoAP基于UDP。
2. **面向应用层**：三种协议都工作在应用层，通常用于客户端与服务器或设备之间的通信。
3. **数据传输方式**：三者都支持小规模数据的传输，尤其适合物联网中的设备之间的通信。
4. **资源受限环境应用**：虽然HTTP相对较重，但三者都可以在资源受限的环境中使用（如物联网设备通信）。

区别

特性	HTTP	CoAP	MQTT
协议架构	请求/响应模型，基于客户端-服务器	请求/响应模型，基于客户端-服务器	发布/订阅模型，基于客户端-Broker
传输协议	TCP	UDP	TCP
消息大小	较大，HTTP头部数据较多	轻量级，头部数据小	轻量级，头部数据少

特性	HTTP	CoAP	MQTT
适用场景	Web应用、数据交换、API接口	资源受限的物联网设备	低带宽、低功耗的物联网场景
QoS支持	无原生QoS机制，可靠性通过TCP保障	提供简易的可靠传输机制（重传、确认）	原生支持QoS，提供QoS 0（最多一次）、1（至少一次）、2（仅一次）
消息格式	通常为文本（如JSON、XML等），头部较长	二进制编码（如CBOR），头部简洁	二进制格式，头部简洁
传输效率	较低（较大的消息头），适合非实时应用	高（轻量化设计，支持小数据包，适合实时应用）	高（轻量级，适合实时通信和低带宽场景）
安全性	基于HTTPS/TLS	支持DTLS（Datagram Transport Layer Security）	基于TLS
适用性	更适合传统Web应用或需要完整功能的服务	适合低带宽、低功耗的物联网应用	适合低功耗、低带宽且有消息推送需求的物联网应用
连接管理	每次请求响应会新建或保持连接	无状态，保持轻量	长连接，支持持久会话
设备电量需求	较高	较低	较低

总结

- **HTTP**适合Web应用、API调用和数据交互，功能较全但较为耗资源。
- **CoAP**适合低带宽、高延迟的物联网设备，适应资源受限环境，传输效率高。
- **MQTT**适合低功耗、低带宽的物联网场景，尤其是需要消息推送和实时通信的场景，如远程监控、智能家居等。

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。

在ThingsBoard中，系统管理员、租户管理员和普通用户三者之间有明确的角色和权限划分，各自负责不同的管理任务，并协同实现整个物联网管理平台的运作。以下是这三个角色之间的关系及作用：

1. 系统管理员（System Administrator）

- **角色定位**：系统管理员是ThingsBoard系统中权限最高的角色，负责平台级别的设置和管理。
- **主要职责**：
  - 创建和管理租户（Tenant）。
  - 配置平台的全局设置，包括设备类型、数据存储配置和系统监控等。
  - 监控系统性能，确保平台的安全性和稳定性。

    - 为租户管理员提供技术支持和问题解决。
- **关系**：
    - 系统管理员主要面向租户管理员，提供支持和管理权限，但不会直接管理普通用户。
    - 为每个租户创建一个租户管理员，并将租户的管理权限交由租户管理员。

2. 租户管理员（Tenant Administrator）

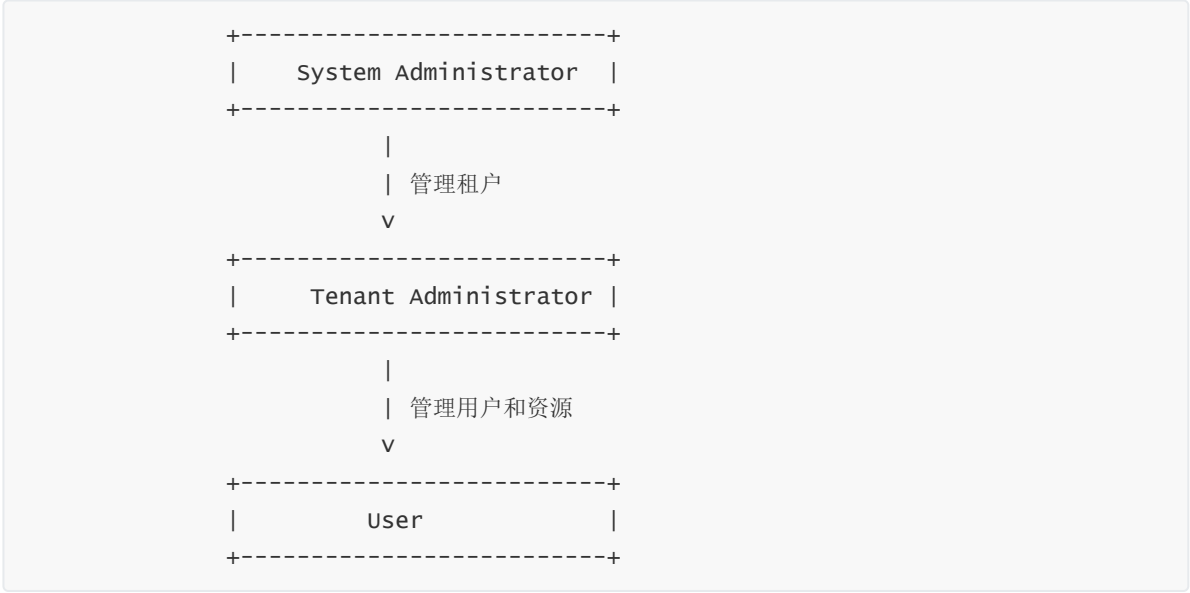
- **角色定位**：租户管理员是每个租户的顶级管理角色，负责管理租户内部的资源和用户。
- **主要职责**：
    - 管理租户内的设备、资产、仪表板、规则链等资源。
    - 创建和管理普通用户账号，分配权限。
    - 监控租户范围内的数据流和设备状态，确保租户内系统的正常运转。
    - 自定义租户内部的仪表板和应用，优化业务流程。
- **关系**：
    - 租户管理员直接接受系统管理员的管理，为该租户内部的用户提供管理支持。
    - 租户管理员可以根据业务需求创建和管理普通用户，分配资源和权限。

3. 普通用户（User）

- **角色定位**：普通用户是ThingsBoard中的最终使用者，具有最少的权限，只能访问特定资源。
- **主要职责**：
    - 通过ThingsBoard平台访问、监控和操作租户分配的设备和资产。
    - 查看数据和仪表板，并可能通过规则链触发某些自动化任务。
- **关系**：
    - 普通用户直接接受租户管理员的管理，由租户管理员分配权限和资源。
    - 普通用户不能创建或管理其他用户，仅有权限访问被分配到的设备和数据。

角色关系图

以下是系统管理员、租户管理员和普通用户三者之间的关系图：



总结

- **系统管理员**负责平台整体的配置和租户管理，是全局管理角色。
- **租户管理员**负责租户内部的资源和用户管理，是租户级别的管理角色。
- **普通用户**负责监控和操作设备，直接使用平台提供的功能。

f. 请简述规则引擎的作用，有什么应用场景?

规则引擎在物联网平台（如ThingsBoard）中起着重要作用，主要用于实现数据处理和自动化控制。它能够基于定义的条件或规则，自动响应和处理来自物联网设备的数据，帮助用户实现灵活的业务逻辑和流程自动化。

规则引擎的作用

1. **实时数据处理**：规则引擎可以实时处理设备上传的数据，并根据预定义规则对数据进行筛选、转换和聚合等操作。例如，将温度数据进行阈值过滤，如果超出范围则生成报警事件。
2. **事件响应和通知**：基于事件触发规则引擎可以自动生成报警、发送通知或执行控制命令。例如，当检测到设备故障时，可以通过电子邮件或短信通知管理员。
3. **自动化控制**：规则引擎可以基于特定条件自动控制其他设备。例如，当环境传感器检测到光照度过低时，规则引擎可以触发命令自动打开照明系统。
4. **数据转发与集成**：规则引擎支持将数据转发到第三方系统（如数据库、云服务、REST API等），实现与外部服务或应用的集成，扩展物联网平台的数据处理能力。

应用场景

1. **工业物联网监控**：在生产车间中，规则引擎可以监测设备状态和生产数据，自动触发报警或控制设备。比如，当设备温度超过安全阈值时，可以自动关闭设备并通知维修人员。
2. **智慧城市管理**：在智慧路灯或智能交通系统中，规则引擎可以根据时间、天气和流量条件动态调整路灯亮度或信号灯模式，实现城市资源的节约和智能管理。
3. **智能家居**：在家庭环境中，规则引擎可以实现自动化控制，比如当门窗传感器检测到有人入侵时，自动触发报警器并通知用户；或者在温度过高时自动开启空调。
4. **环境监测**：在环境监测系统中，规则引擎可以实时监控空气质量、湿度、温度等数据，并在数据超标时自动报警或生成报告，帮助管理机构做出快速响应。
5. **健康监测**：在医疗设备监控中，规则引擎可以实时分析病人的健康数据，如心率、血氧等，一旦检测到异常情况可以自动通知医护人员进行及时处理。

总结

规则引擎通过实时的数据处理、事件响应、自动控制和集成能力，实现了物联网设备间的联动和自动化，广泛应用于各类智能化场景。

六、实验数据记录和处理

6.1 基础设备控制实验

a. 代码截图

makefile:

WIFI_SSID和WIFI_PASS 修改成自己的热点的名字和密码，注意名字不能含有中文字符

```
WIFI_SSID ?= "jklhonorMagic4"
WIFI_PASS ?= "06784588"
# DEVICE NAME 也可自定义
CFLAGS += -DCONFIG_NIMBLE_AUTOADV_DEVICE_NAME='"NimBLE GATT Example"'
CFLAGS += -DCONFIG_NIMBLE_AUTOADV_START_MANUALLY=1
```

main_function.cc:

修改了kTensorArenaSize，这里改成了9*1024，因为太大了的话连接网络那一步会报错。

```
namespace {
    const tflite::Model* model = nullptr;
    tflite::MicroInterpreter* interpreter = nullptr;
    TfLiteTensor* input = nullptr;
    TfLiteTensor* output = nullptr;

    // Create an area of memory to use for input, output, and intermediate arrays.
    // Finding the minimum value for your model may require some trial and error.
    constexpr int kTensorArenaSize =  9 * 1024;
    uint8_t tensor_arena[kTensorArenaSize];
}  // namespace
```

led_contronller.cpp:

直接复制粘贴了之前的文件

main.cpp:

修改DEFAULT_IPV4地址为本机ip地址，可以通过ipconfig查看

如果修改了MQTT_CLIENT_ID，这些也需要修改。

```
main.cpp > DEFAULT_TOPIC
51  using namespace std;
52  void setup();
53  int predict(float *imu_data, int data_len, float threashold, int class_num);
54
55  #define MAIN_QUEUE_SIZE     (8)
56  static msg_t _main_msg_queue[MAIN_QUEUE_SIZE];
57
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
69
70
```

添加全局变量，类似lab3，led_always状态设为5

main.cpp > DEFAULT_TOPIC

```
#define class_num (4)
#define SAMPLES_PER_GESTURE (10)
#define THREAD_STACKSIZE        (THREAD_STACKSIZE_IDLE)
static char stack_for_led_thread[800];
static char stack_for_motion_thread[2000];
static int current_motion = 0;
float threshold = 0.7;
static MPU6050 mpu;
enum MoveState{Stationary, Tilted, Rotating, Moving};
static int led_always = 5; // 5 means normal 0 1 2 3 means different colors
// the pid of led thread
static kernel_pid_t _led_pid;
// static kernel_pid_t _main_pid;
static kernel_pid_t _motion_pid;
int mqtt_interval_ms = 5000;
```

led_thread函数：复制上次lab3的部分，修改led颜色

```
void *_led_thread(void *arg)
{
    (void) arg;
    LEDController led(LED_GPIO_R, LED_GPIO_G, LED_GPIO_B);
    led.change_led_color(0);
    while(1){
        // Input your codes
        // Wait for a message to control the LED
        // Display different light colors based on the motion state of the device.
        msg_t msg;
        msg_receive(&msg);
        led_state = msg.content.value;
        if(led_always == 5){ // change led according to moving state
            if (msg.content.value == Stationary) {
            led.change_led_color(COLOR_NONE);
            printf("[led_thread]: led turn off!\n");
            } else if (msg.content.value == Tilted) {
            led.change_led_color(COLOR_RED);
            printf("[led_thread]: led turn red!\n");
            } else if (msg.content.value == Rotating) {
            led.change_led_color(COLOR_BLUE);
            printf("[led_thread]: led turn blue!\n");
            } else if (msg.content.value == Moving) {
            led.change_led_color(COLOR_GREEN);
            printf("[led_thread]: led turn green!\n");
            } else {
            led.change_led_color(COLOR_YELLO);
            printf("[led_thread]: led turn yellow!\n");
            }
        }
```

```
        else if(led_always == 0){
            led.change_led_color(COLOR_NONE);
            printf("[led_thread]: led stay none\n");
        }
        else if(led_always == 1){
            led.change_led_color(COLOR_RED);
            printf("[led_thread]: led stay red\n");
        }
        else if(led_always == 2){
            led.change_led_color(COLOR_BLUE);
            printf("[led_thread]: led stay blue\n");
        }
        else if(led_always == 3){
            led.change_led_color(COLOR_GREEN);
            printf("[led_thread]: led stay green\n");
        }
        delay_ms(10);
    }
    return NULL;
}
```

get_imu_data函数：复制lab3部分，获得传感器数值并进行处理

```
void get_imu_data(MPU6050 mpu, float *imu_data){
    int16_t ax, ay, az, gx, gy, gz;
    for(int i = 0; i < SAMPLES_PER_GESTURE; ++i)
    {
        /* code */
        delay_ms(collect_interval_ms);
        mpu.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);
        imu_data[i*6 + 0] = ax / accel_fs_convert;
        imu_data[i*6 + 1] = ay / accel_fs_convert;
        imu_data[i*6 + 2] = az / accel_fs_convert;
        imu_data[i*6 + 3] = gx / gyro_fs_convert;
        imu_data[i*6 + 4] = gy / gyro_fs_convert;
        imu_data[i*6 + 5] = gz / gyro_fs_convert;
    }
}
```

_motion_thread函数：复制lab3部分，调整参数，

```c
void *_motion_thread(void *arg)
{
    (void) arg;
    // Initialize MPU6050 sensor

    // get mpu6050 device id
    uint8_t device_id = mpu.getDeviceID();
    printf("[IMU_THREAD] DEVICE_ID:0x%x\n", device_id);
    mpu.initialize();
    // Configure gyroscope and accelerometer full scale ranges
    uint8_t gyro_fs = mpu.getFullScaleGyroRange();
    uint8_t accel_fs_g = mpu.getFullScaleAccelRange();
    uint16_t accel_fs_real = 1;

    // Convert gyroscope full scale range to conversion factor
    if (gyro_fs == MPU6050_GYRO_FS_250)
        gyro_fs_convert = 131.0;
    else if (gyro_fs == MPU6050_GYRO_FS_500)
        gyro_fs_convert = 65.5;
    else if (gyro_fs == MPU6050_GYRO_FS_1000)
        gyro_fs_convert = 32.8;
    else if (gyro_fs == MPU6050_GYRO_FS_2000)
        gyro_fs_convert = 16.4;
    else
        printf("[IMU_THREAD] Unknown GYRO_FS: 0x%x\n", gyro_fs);

    // Convert accelerometer full scale range to real value
    if (accel_fs_g == MPU6050_ACCEL_FS_2)
        accel_fs_real = g_acc * 2;
```

```c

    // Convert accelerometer full scale range to real value
    if (accel_fs_g == MPU6050_ACCEL_FS_2)
        accel_fs_real = g_acc * 2;
    else if (accel_fs_g == MPU6050_ACCEL_FS_4)
        accel_fs_real = g_acc * 4;
    else if (accel_fs_g == MPU6050_ACCEL_FS_8)
        accel_fs_real = g_acc * 8;
    else if (accel_fs_g == MPU6050_ACCEL_FS_16)
        accel_fs_real = g_acc * 16;
    else
        printf("[IMU_THREAD] Unknown ACCEL_FS: 0x%x\n", accel_fs_g);

    // Calculate accelerometer conversion factor
    accel_fs_convert = 32768.0 / accel_fs_real;
    float imu_data[SAMPLES_PER_GESTURE * 6] = {0};
    int data_len = SAMPLES_PER_GESTURE * 6;
    delay_ms(200);
```

```
    // Calculate accelerometer conversion factor
    accel_fs_convert = 32768.0 / accel_fs_real;
    float imu_data[SAMPLES_PER_GESTURE * 6] = {0};
    int data_len = SAMPLES_PER_GESTURE * 6;
    delay_ms(200);
    // Main loop
    int ret = 0;
    string motions[class_num] = {"Stationary", "Tilted", "Rotating", "Moving"};
    while (1) {
        delay_ms(predict_interval_ms);
        // Log the delay between predictions
        LOG_INFO("[MOTION_THREAD] Prediction gap: %d ms\n", predict_interval_ms);
        // Log the mqtt interval data
        LOG_INFO("[MQTT] mqtt interval data: %d ms\n", mqtt_interval_ms);
        // Read sensor data
        get_imu_data(mpu, imu_data);
        // Log the current threshold before prediction
        LOG_INFO("[MOTION_THREAD] Using threshold: %.2f\n", threshold);
        ret = predict(imu_data, data_len, threshold, class_num);
        // Send message to LED thread based on motion
        msg_t msg;
        msg.content.value = ret;
        msg_send(&msg, _led_pid); // Send message to LED control thread
        // Log the prediction result
        LOG_INFO("[MOTION_THREAD] Prediction result: %d, %s\n", ret, motions[ret].c_str());
        // Update global motion state
        current_motion = ret;
        LOG_INFO("Predict: %d, %s\n", ret, motions[ret].c_str());
    }
    return NULL;
```

这里新添加了一个mqtt_uid，自定义uuid，下方也进行了修改，补全相关部分。

```
void *_motion_thread(void *arg)

/// input your code, 自定义想要的UUID
/* UUID = 1bce38b3-d137-48ff-a13e-033e14c7a335 */
static const ble_uuid128_t gatt_svr_svc_rw_demo_uuid
= {{128}, {0x15, 0xa3, 0xc7, 0x14, 0x3e, 0x03, 0x3e, 0xa1, 0xff,
0x48, 0x37, 0xd1, 0xb3, 0x38, 0xce, 0x1b}};
//35f2led readwrite
static const ble_uuid128_t gatt_svr_chr_led_uuid
= {{128}, {0x62, 0x17, 0x99, 0x7e, 0x50, 0x27, 0x38, 0xba, 0x3b,
0x4f, 0x70, 0x30, 0x86, 0x83, 0xf2, 0x35}};
//00ffmotion read
static const ble_uuid128_t gatt_svr_chr_motion_uuid
= {{128}, {0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88, 0x99,
0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF, 0x00}};
//0201threshold readwrite
static const ble_uuid128_t gatt_svr_chr_threshold_uuid
= {{128}, {0x13, 0x24, 0x35, 0x46, 0x57, 0x68, 0x79, 0x8A, 0x91,
0xAC, 0xBD, 0xCE, 0xDF, 0xF0, 0x01, 0x02}};
//0302frequency readwrite
static const ble_uuid128_t gatt_svr_chr_frequency_uuid
= {{128}, {0x14, 0x25, 0x36, 0x47, 0x58, 0x69, 0x7A, 0x8B, 0x92,
0xAD, 0xBE, 0xCF, 0xE0, 0xF1, 0x02, 0x03}};
//0301mqtt readwrite
static const ble_uuid128_t gatt_svr_chr_mqtt_uuid
= {{128}, {0x14, 0x25, 0x36, 0x47, 0x58, 0x69, 0x7A, 0x8B, 0x92,
0xAD, 0xBE, 0xCF, 0xE0, 0xF1, 0x04, 0x03}};

static int gatt_svr_chr_access_rw_demo(
        uint16_t conn_handle, uint16_t attr_handle,
        struct ble_gatt_access_ctxt *ctxt, void *arg);
```

```c
static const struct ble_gatt_svc_def gatt_svr_svcs[] = {
    /*
     * given characteristics
     */
    // input your code, 请按需求更改。
    {
        /* Service: Read/Write Demo */
        .type = BLE_GATT_SVC_TYPE_PRIMARY,
        .uuid = (ble_uuid_t*) &gatt_svr_svc_rw_demo_uuid.u,
        .characteristics = (struct ble_gatt_chr_def[]) { {
        /* Characteristic: Read/Write Demo write */
        .uuid = (ble_uuid_t*) &gatt_svr_chr_led_uuid.u,
        .access_cb = gatt_svr_chr_access_rw_demo,
        .flags = BLE_GATT_CHR_F_READ | BLE_GATT_CHR_F_WRITE,
        // .flags = BLE_GATT_CHR_F_READ | BLE_GATT_CHR_F_WRITE_NO_RSP,
        }, {
        .uuid = (ble_uuid_t*) &gatt_svr_chr_motion_uuid.u,
        .access_cb = gatt_svr_chr_access_rw_demo,
        .flags = BLE_GATT_CHR_F_READ,
        }, {
        .uuid = (ble_uuid_t*) &gatt_svr_chr_threshold_uuid.u,
        .access_cb = gatt_svr_chr_access_rw_demo,
        .flags = BLE_GATT_CHR_F_READ | BLE_GATT_CHR_F_WRITE,
        }, {
        .uuid = (ble_uuid_t*) &gatt_svr_chr_frequency_uuid.u,
        .access_cb = gatt_svr_chr_access_rw_demo,
        .flags = BLE_GATT_CHR_F_READ | BLE_GATT_CHR_F_WRITE,
        }, {
        .uuid = (ble_uuid_t*) &gatt_svr_chr_mqtt_uuid.u,
        .access_cb = gatt_svr_chr_access_rw_demo,
        .flags = BLE_GATT_CHR_F_READ | BLE_GATT_CHR_F_WRITE,
        }, {
            0, /* No more characteristics in this service */
        }, }
    },
    {
        0, /* No more services */
    },
};
```

```
    struct ble_gatt_access_ctxt *ctxt, void *arg)
switch(ctxt->op){
case BLE_GATT_ACCESS_OP_READ_CHR:
    if (ble_uuid_cmp(ctxt->chr->uuid, &gatt_svr_chr_motion_uuid.u) == 0)
    {
        // 读取当前预测的运动状态
        rc = os_mbuf_append(ctxt->om, &current_motion, sizeof(current_motion));
        LOG_INFO("[READ] current_motion = %d\n", current_motion);
        return rc;
    }
    else if (ble_uuid_cmp(ctxt->chr->uuid, &gatt_svr_chr_led_uuid.u) == 0){
        //读取当前的led灯状态
        rc = os_mbuf_append(ctxt->om, &current_motion,
        sizeof(current_motion));
        LOG_INFO("[READ] led_status = %d\n", current_motion);
        return rc;
    }
    else if (ble_uuid_cmp(ctxt->chr->uuid, &gatt_svr_chr_threshold_uuid.u) == 0) {
        // 读取当前阈值
        rc = os_mbuf_append(ctxt->om, &threshold, sizeof(threshold));
        LOG_INFO("[READ] threshold = %.2f\n", threshold);
        return rc;
    }
    else if (ble_uuid_cmp(ctxt->chr->uuid, &gatt_svr_chr_frequency_uuid.u) == 0) {
    // 读取数据采集频率
        rc = os_mbuf_append(ctxt->om, &collect_interval_ms,
        sizeof(collect_interval_ms));
        LOG_INFO("[READ] collect_interval_ms = %d\n",
        collect_interval_ms);
        return rc;
    }
    else if (ble_uuid_cmp(ctxt->chr->uuid, &gatt_svr_chr_mqtt_uuid.u) == 0) {
    // 读取mqtt频率
        rc = os_mbuf_append(ctxt->om, &mqtt_interval_ms,
        sizeof(mqtt_interval_ms));
        LOG_INFO("[READ] mqtt_interval_ms = %d\n",
        mqtt_interval_ms);
        return rc;
    }
```

mqtt_connect和mqtt_disconnect函数不需修改。

mqtt_pub函数增加代码，首先是根据led状态转化三个led_r,led_g,led_b的值。并补充了输出内容。

```
int mqtt_pub(void)
{
    int16_t ax, ay, az, gx, gy, gz;
    mpu.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);

    int16_t led_r = 0, led_g = 0, led_b = 0;
    if(led_state == 0){
        led_r = 0;
        led_g = 0;
        led_b = 0;
    }else if(led_state == 1){
        led_r = 1;
        led_g = 0;
        led_b = 0;
    }else if(led_state == 2){
        led_r = 0;
        led_g = 0;
        led_b = 1;
    }else if(led_state == 3){
        led_r = 0;
        led_g = 1;
        led_b = 0;
    }

    enum QoS qos = QOS0;

    MQTTMessage message;
    message.qos = qos;
    message.retained = IS_RETAINED_MSG;
    led_state = !led_state;
    char json[200];
    snprintf(json, 200, "{\"ax\":%.02f, \"ay\":%.02f, \"az\":%.02f, \"gx\":%.02f, \"gy\":%.02f, \"gz\":%.02f, \"current_motion\":\"%d\", \"led_r\":%d, \"led_g\
            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,
    printf("[Send] Message:%s\n", json);
    message.payload = json;
    message.payloadlen = strlen((char *)message.payload);
```

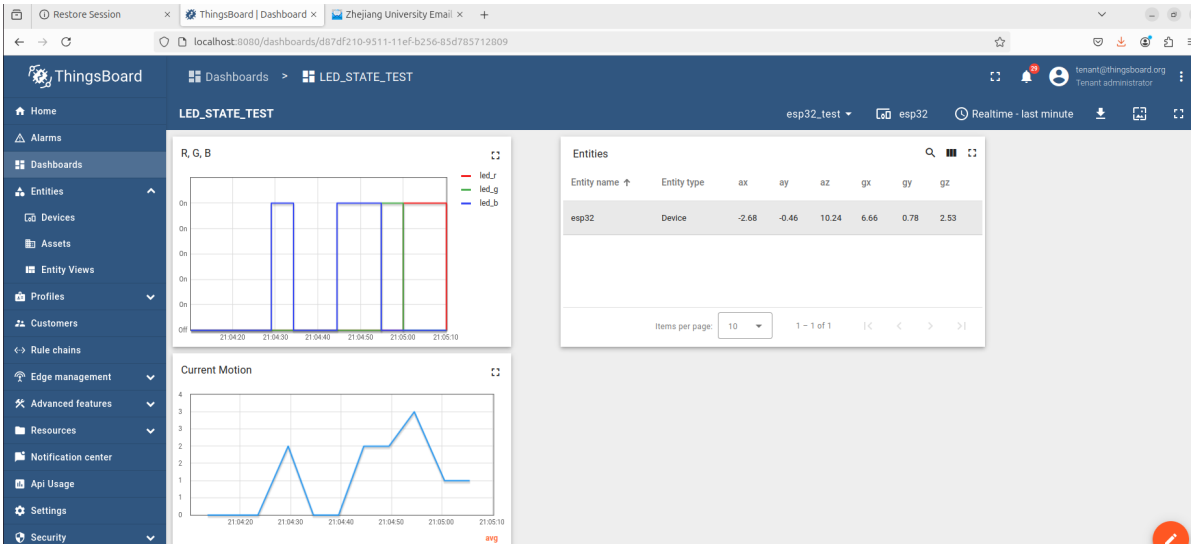
```
    int rc;
    if ((rc = MQTTPublish(&client, DEFAULT_TOPIC.c_str(), &message)) < 0) {
        printf("mqtt_example: Unable to publish (%d)\n", rc);
    }
    else {
        printf("mqtt_example: Message (%s) has been published to topic %s"
               "with QOS %d\n",
               (char *)message.payload, DEFAULT_TOPIC.c_str(), (int)message.qos);
    }

    return rc;
}
```

main函数不需要修改。过程中遇到报错代码，把current_motion修改成全局变量即可解决问题。

b. ThingBoard 可视化数据展示结果图

LED颜色RGB结果、Current Motion和传感器数据结果图



附终端界面：

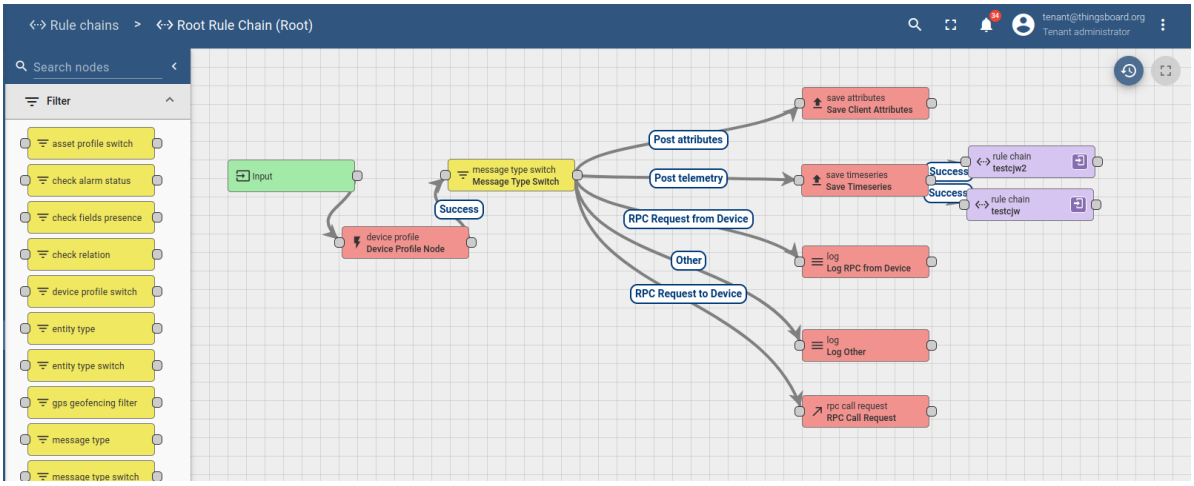
led状态预测和mqtt终端输出展示。

```
2024-10-28 21:08:18,222 # [MOTION_THREAD] Prediction result: 0, Stationary
2024-10-28 21:08:18,223 # Predict: 0, Stationary
2024-10-28 21:08:18,225 # [led_thread]: led turn off!
2024-10-28 21:08:18,633 # mqtt_example: Connection successfully
2024-10-28 21:08:18,644 # [Send] Message:{"ax":-2.02, "ay":-0.39, "az":10.47, "gx":6.78, "gy":0.56, "gz":2.47, "current_motion":"0", "led_r":0, "led_g":0, "led_b":0}
2024-10-28 21:08:18,663 # mqtt_example: Message ({"ax":-2.02, "ay":-0.39, "az":10.47, "gx":6.78, "gy":0.56, "gz":2.47, "current_motion":"0", "led_r":0, "led_g":0, "led_b":0}) has been published to topic v1/devices/me/telemetrywith QOS 0
2024-10-28 21:08:18,665 # mqtt_example: Disconnect successful
2024-10-28 21:08:18,715 # [MOTION_THREAD] Prediction gap: 500 ms
2024-10-28 21:08:18,717 # [MQTT] mqtt interval data: 5000 ms
2024-10-28 21:08:18,920 # [MOTION_THREAD] Using threshold: 0.70
2024-10-28 21:08:18,925 # ------------------------------
```

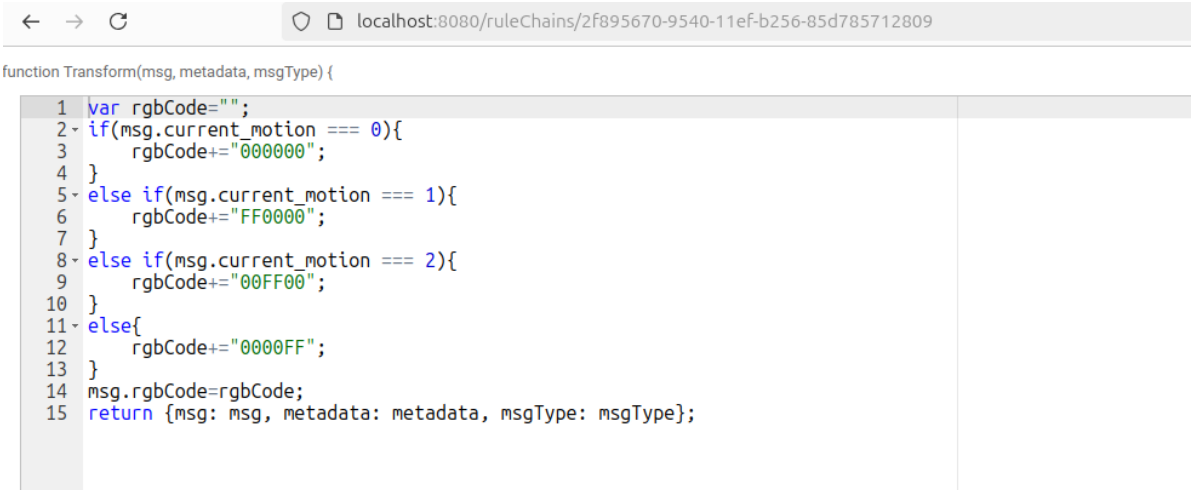
```
2024-10-28 21:03:02,039 # ------------------------------
2024-10-28 21:03:02,043 # [0] value: 0.00
2024-10-28 21:03:02,045 # [1] value: 0.00
2024-10-28 21:03:02,046 # [2] value: 1.00
2024-10-28 21:03:02,047 # [3] value: 0.00
2024-10-28 21:03:02,050 # Motion prediction: 2
2024-10-28 21:03:02,054 # [MOTION_THREAD] Prediction result: 2, Rotating
2024-10-28 21:03:02,056 # Predict: 2, Rotating
2024-10-28 21:03:02,059 # [led_thread]: led turn blue!
2024-10-28 21:03:02,546 # [MOTION_THREAD] Prediction gap: 500 ms
2024-10-28 21:03:02,548 # [MQTT] mqtt interval data: 5000 ms
2024-10-28 21:03:02,750 # [MOTION_THREAD] Using threshold: 0.70
2024-10-28 21:03:02,756 # ------------------------------
2024-10-28 21:03:02,758 # [0] value: 0.00
2024-10-28 21:03:02,759 # [1] value: 0.00
2024-10-28 21:03:02,760 # [2] value: 0.03
2024-10-28 21:03:02,761 # [3] value: 0.97
2024-10-28 21:03:02,763 # Motion prediction: 3
2024-10-28 21:03:02,768 # [MOTION_THREAD] Prediction result: 3, Moving
2024-10-28 21:03:02,770 # Predict: 3, Moving
```

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。

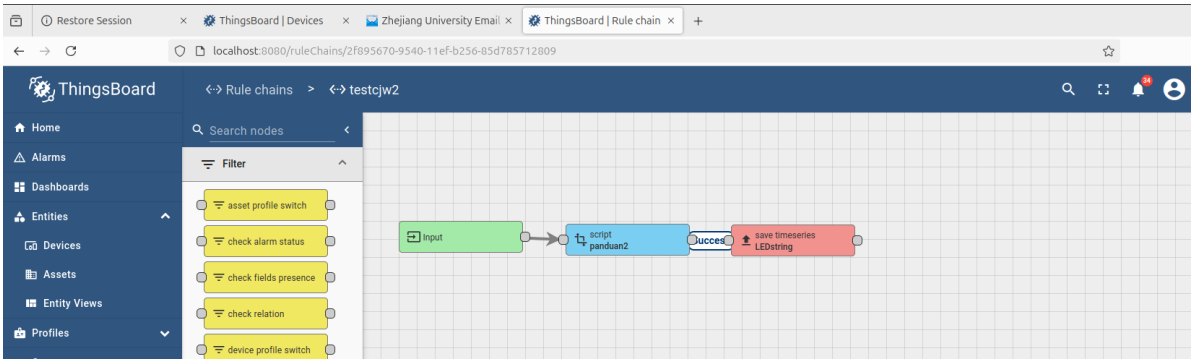
root规则链修改后内容：



bonus1：展示规则链中判断代码如下：



规则链结构如下：



倾斜状态时，current_motion数值和led相关值如下：

Latest telemetry

Last update time	Key	Value
2024-10-28 23:47:44	ax	-0.99
2024-10-28 23:47:44	ay	7.84
2024-10-28 23:47:44	az	6.51
2024-10-28 23:47:44	current_motion	1
2024-10-28 23:47:44	gx	6.76
2024-10-28 23:47:44	gy	0.89
2024-10-28 23:47:44	gz	2.45
2024-10-28 23:51:25	led_b	0
2024-10-28 23:51:25	led_g	0
2024-10-28 23:51:25	led_r	1

rgbCode代码如下：

Latest telemetry

Last update time	Key	Value
2024-10-28 00:28:59	r_led_state	0
2024-10-28 23:48:40	rgbCode	FF0000
2024-10-26 18:45:57	temperature	25

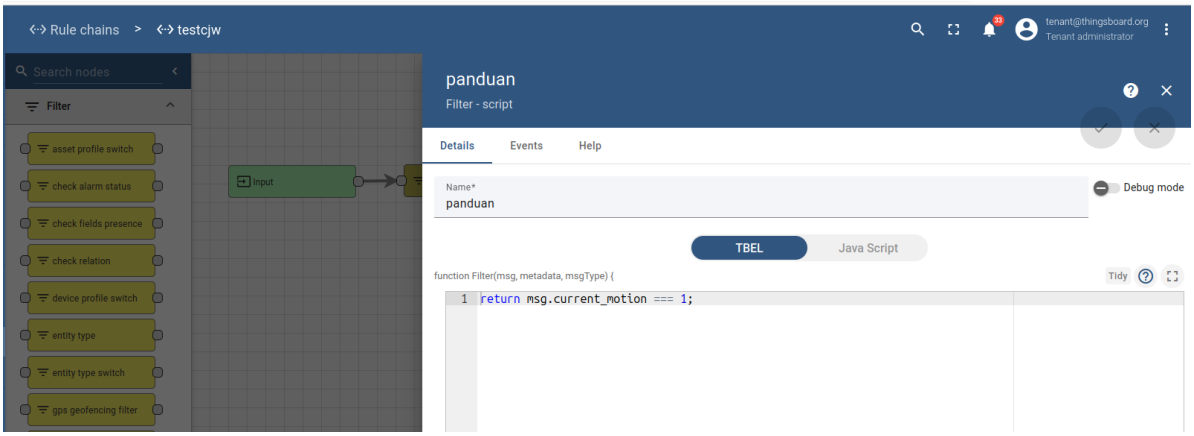
旋转状态时相关结果如下：说明rgbCode正常变化。

2024-10-28 23:54:08	current_motion	2
2024-10-28 23:54:08	gx	6.63
2024-10-28 23:54:08	gy	0.6
2024-10-28 23:54:08	gz	2.53
2024-10-28 23:54:08	led_b	1
2024-10-28 23:54:08	led_g	0
2024-10-28 23:54:08	led_r	0

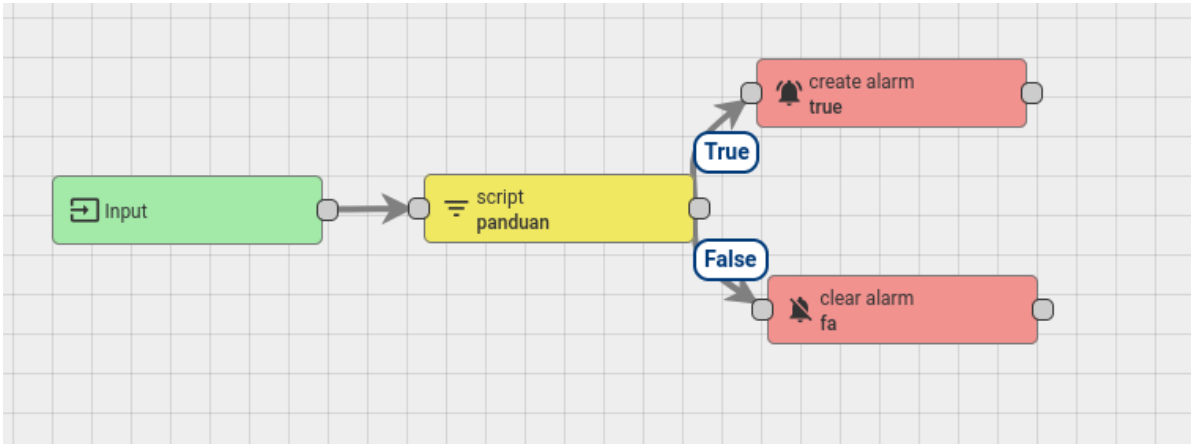
Latest telemetry

Last update time	Key	Value
2024-10-28 00:28:59	r_led_state	0
2024-10-28 23:53:33	rgbCode	00FF00
2024-10-26 18:45:57	temperature	25

bonus2：展示规则链中代码判断如下：



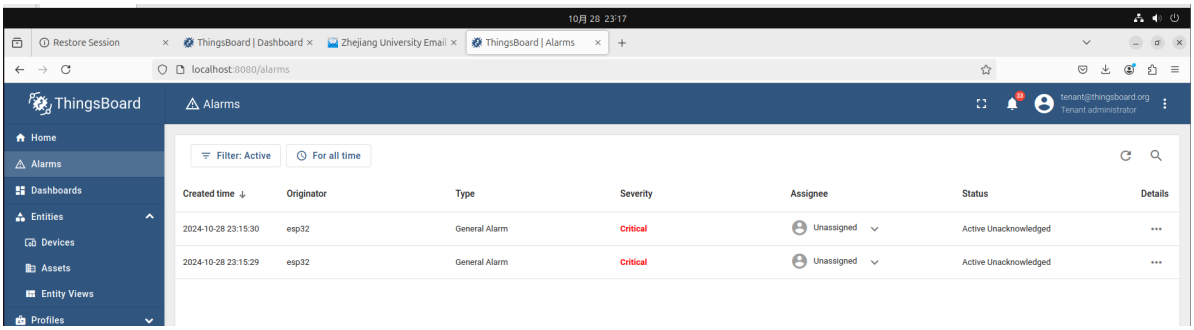
规则链结构如下：



倾斜状态时，current_motion和led相关值结果如下：

Last update time	Key	Value
2024-10-28 23:52:05	current_motion	1
2024-10-28 23:52:05	gx	6.78
2024-10-28 23:52:05	gy	0.61
2024-10-28 23:52:05	gz	2.45
2024-10-28 23:52:05	led_b	0
2024-10-28 23:51:45	led_b	0
2024-10-28 23:51:45	led_g	0
2024-10-28 23:51:45	led_r	1

出现警报结果如下：



其他状态不出现警报。

七、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。

基于设备数据上云平台和数据可视化的实验，可以总结出以下基本流程：

1. 设备数据采集

- **数据采集**：物联网设备通过传感器采集各类数据（如温度、湿度、位置等）。
- **数据预处理**：设备端可以对原始数据进行初步处理，比如过滤噪声、去除异常值或数据压缩，以便更高效地上传至云平台。

2. 数据传输到云平台

- **选择传输协议**：通常通过 MQTT、HTTP 或 CoAP 协议将设备数据上传至云平台。协议的选择取决于应用场景和需求：
  - MQTT适合低功耗、实时性强的场景；
  - HTTP适合请求响应式的传输需求；
  - CoAP适合轻量化通信。
- **数据传输**：通过预设的规则或数据传输策略将数据定期或实时发送到云平台的接收端口。

3. 云平台数据接收和存储

- **数据接收**：云平台通过配置的API接口、MQTT Broker或其他通信节点接收设备发送的数据。
- **数据存储**：将接收到的数据存入云端数据库或时序数据库中，通常选择可扩展的数据库（如InfluxDB、TimescaleDB）以适应高频数据写入。
- **数据持久化**：对数据进行备份或持久化存储，方便后续的数据处理和分析。

4. 数据处理与规则引擎触发

- **规则引擎处理**：云平台中的规则引擎根据预定义的规则实时处理数据。可以执行数据过滤、聚合、事件触发、数据转发等操作。
- **事件响应**：规则引擎可以生成报警、通知，或触发其他设备响应，比如在数据超出阈值时发送提醒或控制设备行为。

5. 数据分析与可视化

- **数据处理与分析**：可以对存储的数据进行统计分析、机器学习建模等，以提取更有价值的信息。例如，可以分析趋势、异常检测等。
- **可视化仪表板**：利用可视化工具（如ThingsBoard、Grafana等）创建数据仪表板，以图表、图形的形式展示数据，帮助用户更直观地了解设备运行状态和数据趋势。
- **自定义视图**：用户可以在平台上自定义视图布局、选择不同的图表（如折线图、饼图、热力图等），满足不同业务需求的展示需求。

6. 数据监控与反馈

- **实时监控**：通过可视化仪表板实时监控设备状态，便于及时发现异常并采取相应措施。
- **反馈机制**：平台的监控数据可以反馈给设备，或者通过消息通知给相关人员，实现闭环控制。例如，设备接收到平台的控制指令后调整工作参数。

流程图总结

```
1. 设备数据采集 → 2. 数据传输到云平台 → 3. 云平台数据接收和存储
         ↓                                  ↓                                  ↓
   6. 数据监控与反馈 ← 5. 数据分析与可视化 ← 4. 数据处理与规则引擎触发
```

总结

设备数据上云平台以及数据可视化的基本流程涵盖了从数据采集、传输、存储、处理到展示和监控的全过程。通过云平台和可视化工具的结合，可以实现数据的高效处理与展示，帮助用户实时掌握设备运行情况，并根据数据驱动的决策进一步优化管理流程。