摘 要

本文讲述了我们小组对于基于树莓派的环境监测和报警系统的计和实现过程。主要包含树莓派上视频和传感器数据采集推送的实现，基于 Vue3 的前端，基于 Flask 的后端三个部分。由于树莓派的性能原因，选择将视频流推流并使用 TensorFlow js 在浏览器端用转换后的模型进行图像检测，有效降低了树莓派端的负载并提高了效率。系统能对温湿度，光照强度以及环境中是否有火焰进行实时检测和通过前端页面显示，并在温湿度、光照强度数据超限以及环境中有火焰时通过远程报警和蜂鸣器通知用户。本小组完成了整个系统的设计和实现，程序能够稳定运行，系统基本功能完善。

关键词：火灾检测，环境监测，Vue，Flask，TensorFlow js

目 录

第一章 复杂工程问题归纳与实施方案可行性研究

1.1 需求分析与复杂工程问题归纳

火灾自动报警系统是由触发装置、火灾报警装置、联动输出装置以及具有其它辅助功能装置组成的，它具有能在火灾初期，将燃烧产生的烟雾、热量、火焰等物理量，通过火灾探测器变成电信号，传输到火灾报警控制器，并同时以声或光的形式通知整个楼层疏散，控制器记录火灾发生的部位、时间等，使人们能够及时发现火灾，并及时采取有效措施，扑灭初期火灾，最大限度的减少因火灾造成的生命和财产的损失，是人们同火灾做斗争的有力工具。有关资料统计表明：凡是安装了火灾自动报警系统的场所，发生了火灾一般地说都能及早报警，不会酿成重大火灾。在我们国家许多重要的办公楼、仓库、变电站、控制中心以及500 总吨以上的各种船舶都根据国家标准和有关条文安装了火灾自动报警系统，在消防安全保卫工作中发挥了重要作用。本项目采用摄像头采集的视频数据和传感器采集的温湿度及光照数据进行处理，检测环境参数以及视频中是否有火焰，并将环境参数和摄像头采集视频在网页上展示，当环境数据超限或者视频中有火焰时，远程报警通知到用户。

1. 需求分析

系统主要分为树莓派服务器端和客户端，其中服务器端包括传感器数据采集、视频采集推流、Web 后端和异常报警；客户端负责对传输到的数据进行深度学习模型的推理和识别火焰、Web 前端和可视化等部分。

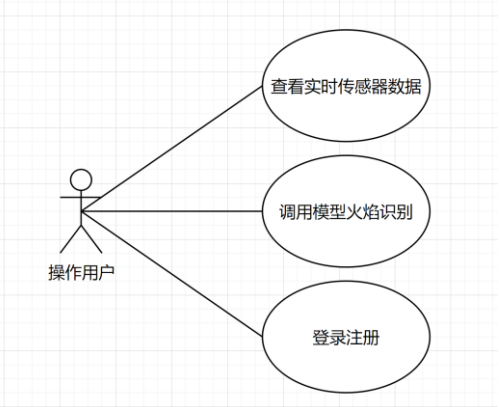

图 1-1 客户端功能需求用例图

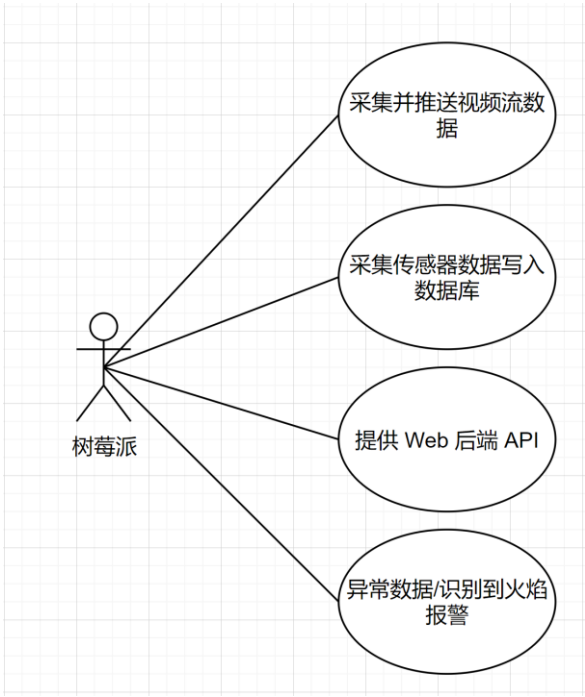

图 1- 2 服务器端功能需求用例图

(1) 传感器数据采集和传输需求分析

对于树莓派服务器端的传感器数据采集和写入部分，主要包括对需要进行检测的数据进行采集分析，检测是否有异常数据的数据处理，以及持久化的数据存储即存放于数据库，如用例图所示：

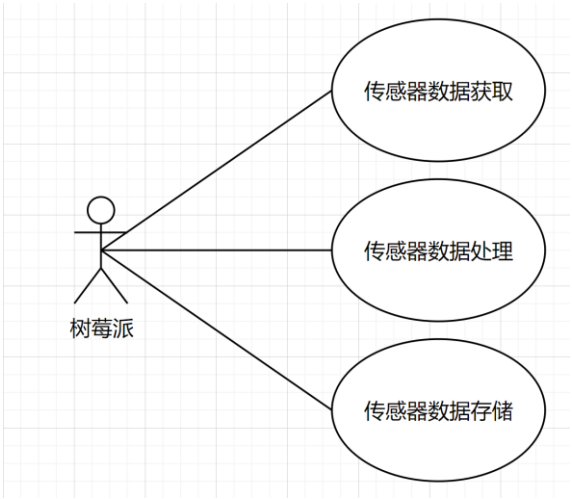

图 1-3 传感器及数据采集用例图

1) 数据采集

数据采集由 ENS210 传感器和 LTR390 传感器组成，ENS210 湿度和温度传感器可以采集目前的温度和湿度，LTR390 传感器可以采集目前的紫外线强度和环境光强度，并将收集到的数据传给后端。

2) 数据预处理和存储

后端对采集到的数据进行初步处理，将异常的数据存储进异常数据库，其他数据存储进有效数据库。

(2) 视频采集和推流需求分析

视频采集由 CreateBlock 树莓派摄像头完成，采集到的图像数据由于不在树莓派端处理，所以直接进行 Websocket 二进制打包推流到浏览器，推流的 Python 脚本内部实际是调用 ffmpeg 将摄像头数据输出到二进制和 Websocket 库传输数据实现，将其一并封装以便之后设置为系统服务自动启动。

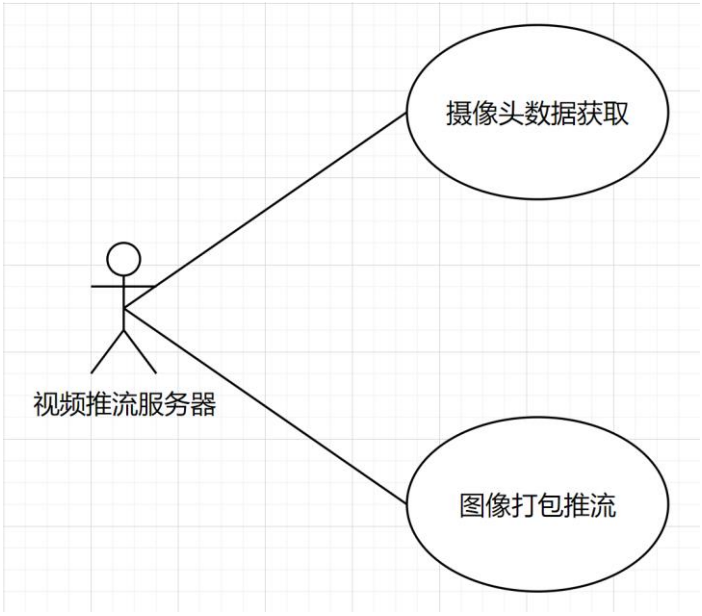

图 1-4 视频采集和推流用例图

(3) 深度学习模型训练需求分析

YOLOv5 默认的模型不含火焰或烟雾的识别能力，仅是一个目标检测框架。为了使其能识别火焰，需要收集并标注火焰的数据集，再在本地进行训练，最后再配置于浏览器端。

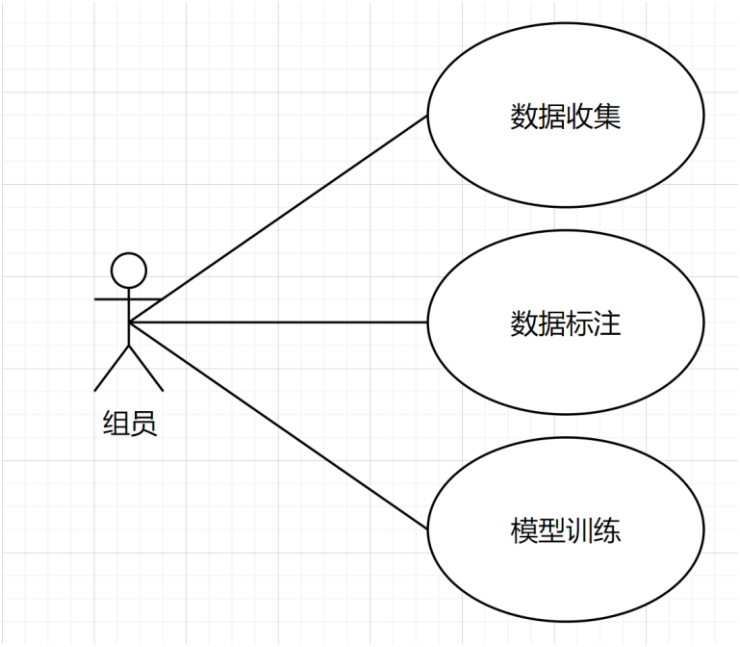

图 1-5 深度学习模型训练用例图

(4) 报警部分需求分析

当检测到异常数据或是前端检测到火焰时，我们的报警系统包括蜂鸣器报警、邮件和微信提醒三个部分，以下为用例图。

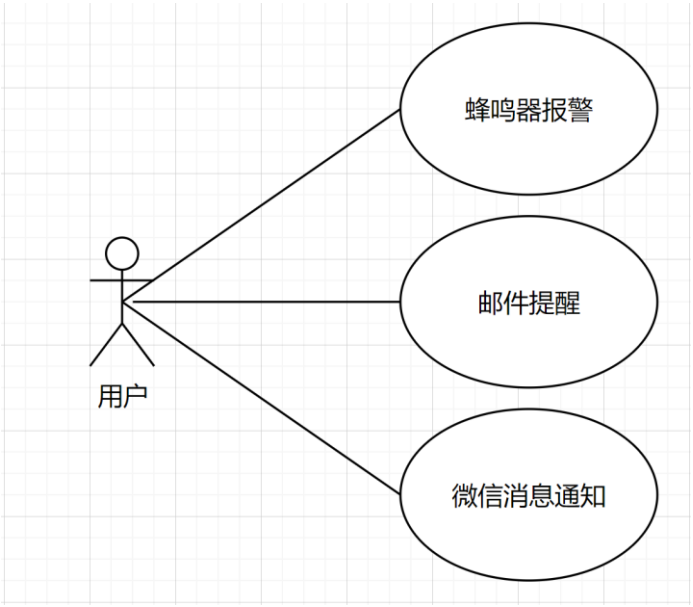

图 1-6 报警系统用例图

该系统通过数据检测和视频火焰识别来判断是否发生火灾，并且二者只要有一个检测到有火灾风险，都会触发提醒机制。

(5) 后端部分需求分析

对于部署在树莓派上的 Flask 后端，我们需要提供用于报警的消息推送的 API 和用于前端显示的传感器数据查询 API，如用例图：

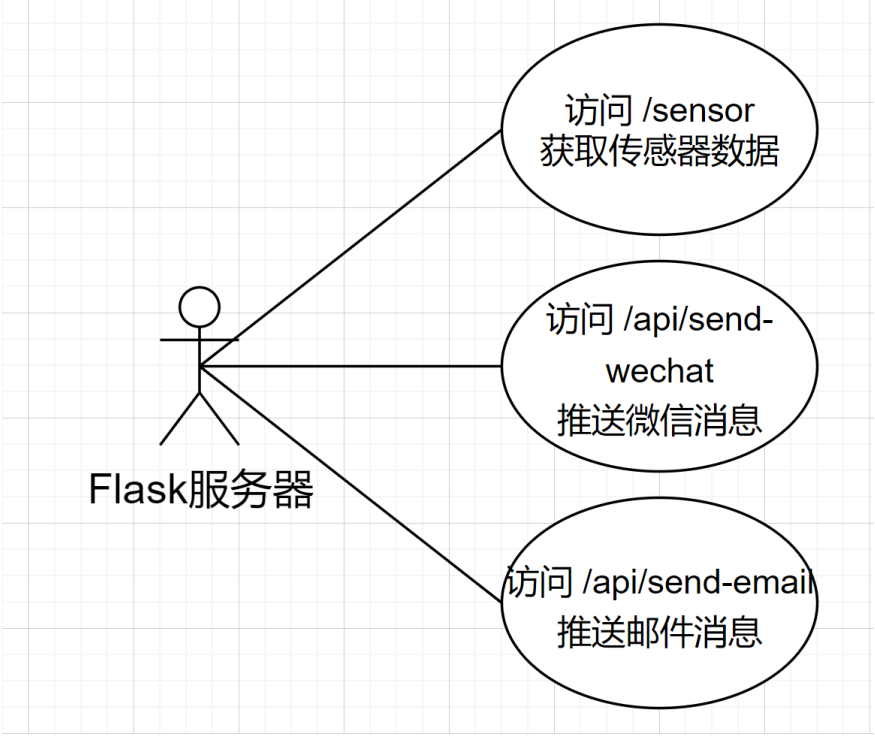

图 1-7 后端部分用例图

(6) 前端用户界面需求分析

为了提高系统的安全性，火灾自动报警系统应具备权限验证的能力，确保敏感信息的保密性。

为了满足上面的要求，我们需要设计用户登录界面。

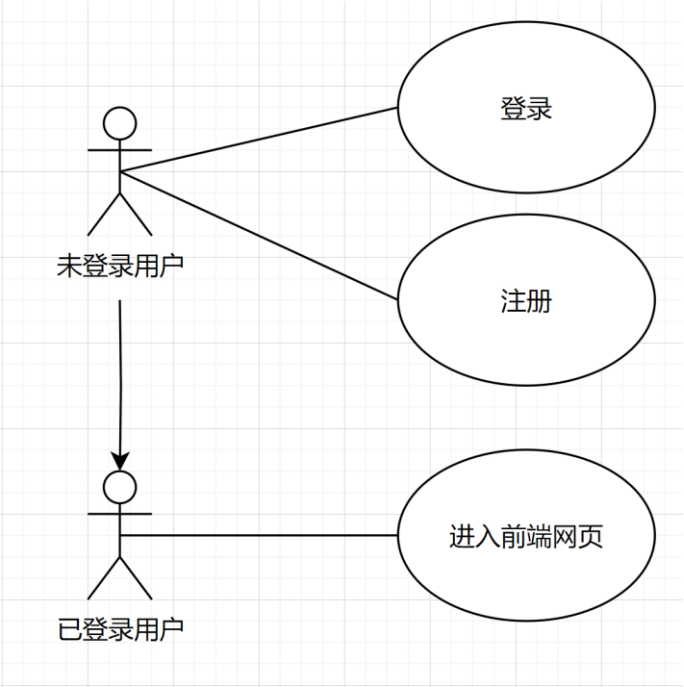

图 1-8 登陆系统用例图

用户需要先进行注册，在后端存储进用户信息后才能使用本系统。用户进入系统界面时会被要求输入用户名和密码。只有符合后端录入的用户才能允许使用本系统。这样可以防止误操作和非法访问，确保系统运行的稳定性和安全性。

(7) 非功能需求分析

除了功能需求外，我们还需对系统的非功能需求进行分析，包括性能，可靠性，安全性，可维护性等。

性能要求：为使火灾检测尽可能即时，应做到视频流传输延迟在秒级内，传感器获取数据时间间隔在秒级内，且长时间运行能避免卡死。

可靠性：在系统意外断电/重启后，应能够及时重启服务并继续记录数据。

安全性：未认证用户无法获取敏感数据，需要登录注册逻辑。

可维护性：系统代码应有足够注释和定义良好的代码规范。

2. 复杂工程问题归纳

1. 系统集成的复杂性

设计一个火灾自动报警系统涉及到多个系统和组件的集成，包括传感器、控制单元、报警装置、通讯接口等。每个组件都有其特定的技术要求和功能，系统集成要求这些组件必须协同工作。处理这种复杂性需要综合考虑各个组件的兼容

性，以及它们之间的交互方式。

2. 数据处理与分析的复杂性

火灾自动报警系统需要处理来自多个传感器的大量数据。这些数据的分析和处理非常关键，因为它们直接关系到火灾的及时检测和报警。复杂性体现在如何准确、快速地处理这些数据，并从中识别出潜在的火灾迹象。

3. 实时性与可靠性的挑战

火灾检测和报警系统必须具备高度的实时性和可靠性。这意味着系统必须能够在火灾发生的第一时间内准确地检测到火源，并且在任何情况下都能稳定运行。这种需求使得设计和实施变得复杂，尤其是在保持系统稳定性和减少误报的同时。

4. 用户界面与交互的复杂性

设计一个直观、易用且功能全面的用户界面同样是一个复杂的工程问题。用户界面不仅需要提供必要的信息和控制选项，而且还要在紧急情况下确保用户能够迅速理解和响应。

5. 安全性与隐私保护

系统的安全性和数据的隐私保护是另一个关键问题。在设计系统时，必须确保数据安全，防止任何形式的非法访问或篡改。同时，保护用户的隐私也是至关重要的。

1.2 实施方案可行性研究

本部分针对上面阐述的复杂问题，提出解决方案，并对方案的可行性进行评估。

1. 系统集成的复杂性

对于这样一个庞大的系统，我们选择将其模块化，分而治之的思想处理，如同软件工程中的 Saas 软件即服务思想，尽量做到高内聚低耦合的系统设计。为此，主要分为了：

传感器部分：目前市场上已有多种类型的传感器（如温湿度、光照、二氧化碳传感器）可用，且它们与树莓派 4B 的兼容性已被验证。这些传感器能够准确捕捉与火灾相关的环境变化。

数据处理部分：利用 Python 及其丰富的数据处理库处理传感器数据是可行的。可以存储海量的数据并以算法分析异常数据，识别潜在的火灾风险。

前后端部分：使用 Flask 和 Vue 等成熟框架进行 Web 后端和前端开发是技术上可行的。

深度学习部分：使用 YOLO 的训练集网络上也能找到很多，前端使用 Tensorflow js 进行客户端推理也是可行的。

我们可以使用单元测试对每个模块单独的可用性进行测试，最后再使用整体测试测试系统的可行性和稳定性，所以，系统集成是可行的。

2. 数据处理与分析的复杂性

对于异常数据，可以通过通过调用数学相关的库函数其对以往平均值或其他统计量的偏移多少判断是否出现火灾；对于图像数据的分析则交给 YOLO 模型。

所以对于数据处理与分析是可行的。

3. 实时性与可靠性的挑战

由于采用了 Websocket 二进制推流，将比传统的 Http 或 rtsp 推流模式降低不少延迟；传感器数值设置为一秒一次的更新，也不会错过异常出现的情况，系统实时性可以得到保证。

为了系统的可靠性，我们设置了多个 Systemd 服务保证各项服务自动重启，若多次重启失败系统管理员可以通过日志查明原因，可靠性也能得到保证。

综上，系统的实时性和可靠性是可行的。

4. 用户页面与交互的复杂性

对于前端设计，我们可以采用成熟且拥有许多开源资源的 Vue 框架，并将控件设计尽量做到简洁美观，同时，我们还实现了自动切换黑夜模式的功能，这样能给用户更舒适的交互体验。

所以用户页面与交互的复杂性是可解决的。

5. 安全性与隐私保护

对于隐私保护，我们设置了用户登陆系统，未注册用户无法通过修改地址进入页面，所有登录验证部分都在前后端均有验证。

所以，系统的安全性和隐私保护也是可行的。

第二章 针对复杂工程问题的方案设计与实现

2.1 针对复杂工程问题的方案设计

系统主要设计思路为：通过树莓派的传感器接收外部数据，并将其保存到数据库中。用户可以通过 Web 服务提供的视频流远程查看环境情况，并通过可视化数据分析变化情况，判断火灾发生的时间和严重程度。此外，如果数据超过正常范围或识别到图像中出现火焰，系统会借助邮箱或即时短信等方式通知远程用户，并使用设备发出警报，提醒周围人员注意危险并采取相应措施。

系统整体架构设计如下

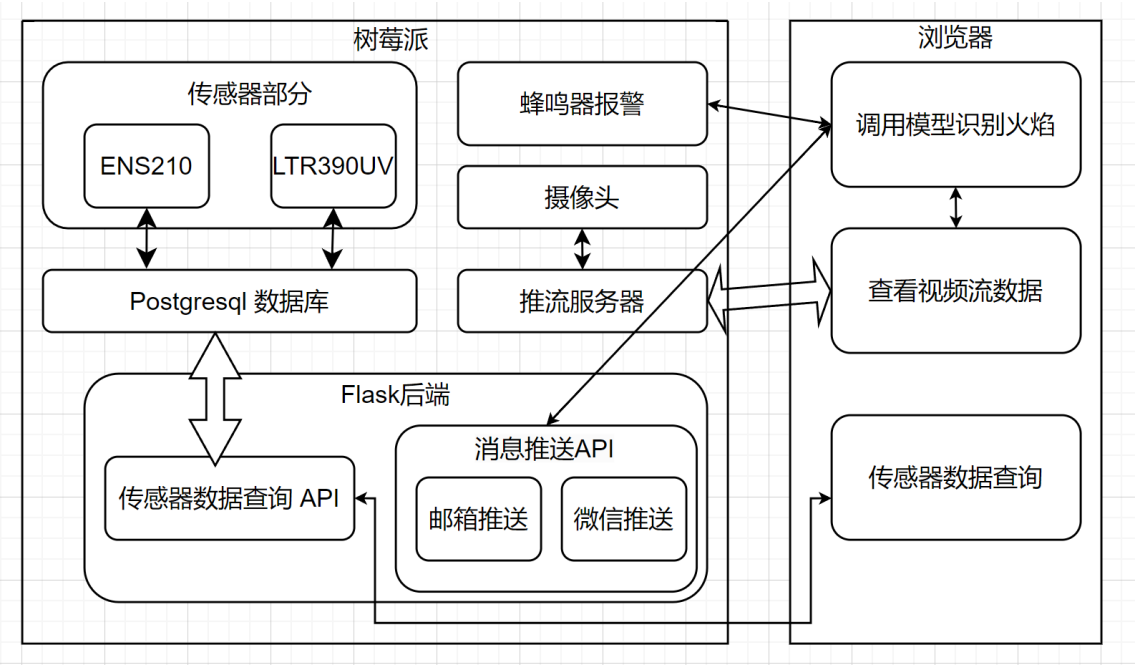

图 2-1 系统总体设计

以下是详细设计

1. 传感器连接方案

由于新的项目需要连接多个传感器，且每个传感器都需要供电及数据交换。如何使用有限的接口还不使电路复杂便成了一个问题。利用面包板可以将多个传感器并联连接到一个接口上面，不仅仅是解决了问题，同时可以随时增加或减少新的传感器，并能够方便接入和调试。由于想要减少电路系统的复杂度，传感器采购上购买了 ENS160 空气质量传感器和 ENS210 温湿度传感器二合一的配

置，可以共用一条 I2C 总线；对于光线传感器 LTR390-UV 的数据读写有 SPI 和 I2C 两种方式，为了简便也采用了 I2C 的连接方式。最终连接结果如下

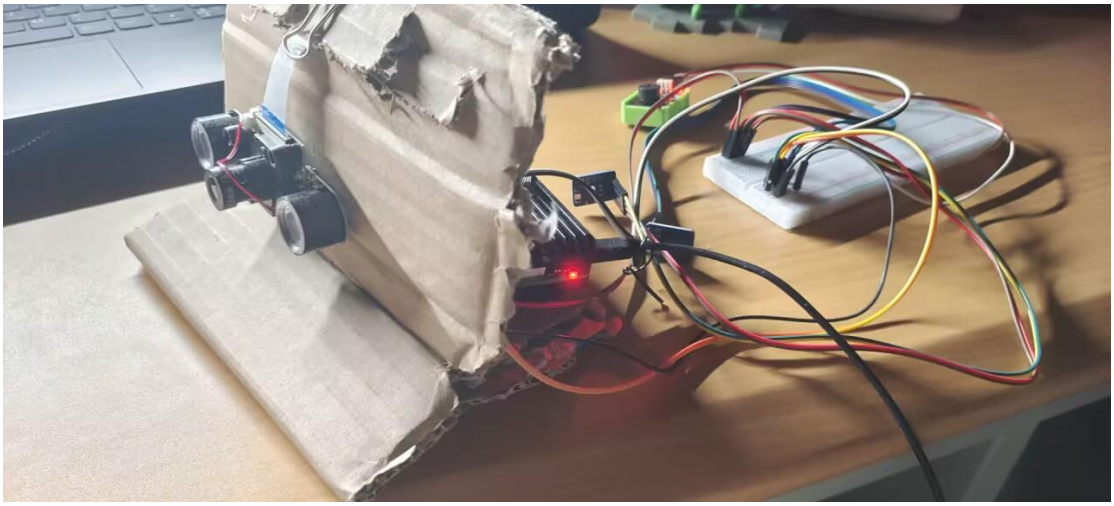

图 2-2 总览

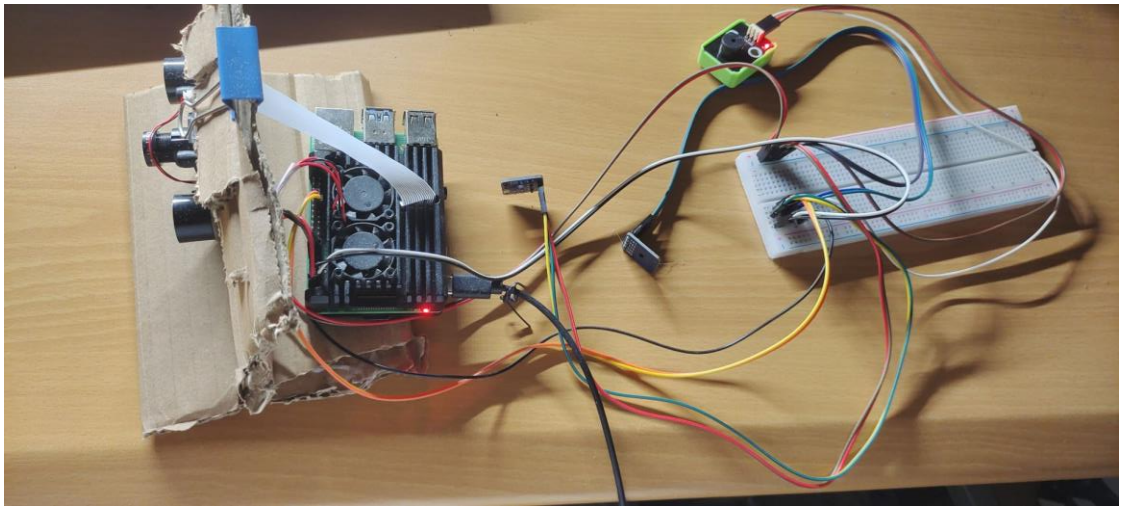

图 2-3 侧视总览

2. 推流系统设计

由于先前直接使用树莓派自带命令 raspivid 推 rtsp 流延迟过高（2~3 秒），不满足延迟需求，寻找网上的开源实现 https://github.com/waveform80/pistreaming 使用 WebSocket 流二进制编码实时推流 MJPEG 解码，将延迟降低到 0.2 秒以内

3. 火焰识别方案设计

由于树莓派无独立显卡，图像处理能力较弱，选择将深度模型识别推理部分

放到浏览器端进行，这又带来图像视频流实时推送的问题，为此选择用 Websocket 二进制进行推流，到了浏览器端再使用 Tensorflow js 库进行推理。

数据集采用网络上已有带标注的火焰图像约 3500 张，使用 Yolo v5s 在个人电脑上进行训练，每次训练大小 batch 等于 16，迭代次数 epoch 等于 10，在 NVIDIA GeForce MX450 上一次 epoch 约 11 分钟。

4. Web 后端设计

Web 后端是一个关键的组件，它承担着将数据库中存储的信息传递给前端并进行渲染的重要任务。在构建这个后端时，我们可以选择使用一些流行的框架，例如 Flask 或 Django。经过仔细考虑，我们最终选择了 Flask 作为 Web 后端的基础，因为它具有简单易用、设计 API 方便、灵活可扩展等许多优点。

为了实现我们的需求，我们设计了 4 个 URL 路由，通过对这些路由进行参数传递，我们可以实现数据查询和发送报警信息的功能。这些 URL 路由提供了灵活性和可定制性，使我们能够根据具体的需求来实现不同的功能。通过对这些路由的调用，我们能够从数据库中检索所需的数据，并将其传递给前端以供展示。此外，我们还实现了发送报警信息的功能，以便及时通知相关人员。

通过采用这种基于 Flask 的 Web 后端架构，我们能够以高效和可靠的方式实现数据传递和功能扩展。这对于我们的项目的成功实施至关重要，并且能够为用户提供出色的体验。我们相信，选择 Flask 作为我们的 Web 后端框架是一个明智的决策，它将为我们提供强大的工具和灵活性，以满足我们的需求。

我们的数据库中包含 3 个表，其中两个表用于存储不同温湿度传感器的数据，而第三个表则用于存储异常数据信息。通常情况下，用户不需要查询异常数据表中的信息。因此，针对这两个传感器表，我们需要构建两个 API 路由，以便能够接收前端传入的起始时间戳和查询条数等数据，并根据这些参数对数据库进行适当的查询，然后将查询到的数据返回给前端页面。另外，为了将报警邮件发送到 QQ 邮箱和发送报警信息到微信，我们需要调用不同的第三方平台接口，因此还需要为这两个功能构建两个额外的 API 路由。

考虑到实时插入的测量数据会导致数据库中记录的数量非常庞大，简单地一次性查询出所有的信息可能会受到数据库性能的限制，导致查询速度变得缓慢。因此，我们需要仔细思考如何编写高性能的 SQL 查询语句，以优化查询效率。为了方便前端处理后端传递的数据，我们将查询到的数据以 JSON 格式返回，这样前端可以更轻松地对数据进行处理和展示。

通过传感器接收的数据包括温度、湿度以及 ALS 和 LUX 两个光照参数，用

于检测环境特征，以判断是否存在火灾的可能性。同时，系统还记录异常数据，以便用户及时察觉环境检测系统的异常情况，并做出必要的调整，从而提高系统检测的准确性。

数据库为了方便管理，加入时间戳字段提供有关数据记录的时间信息，方便后续的数据分析和处理。时间戳可以记录每条数据的创建时间或最后更新时间，这对于数据追溯、监控和分析非常有用。

在数据库的内部实现数据的插入、删除和修改操作过于麻烦，在 Python 代码中使用使用 psycopg2 工具来简化这些操作。

使用 Flask 框架创建 API 接口，以便前端 Web 可以通过发送 HTTP 请求与数据库进行交互，处理前端 Web 的请求。

建立后端接口的整体步骤如下：

(1) 连接树莓派传感器的接口：根据传感器的接口类型和树莓派的 GPIO 口类型进行连接，使用 Python 编写程序读取传感器数据，并将其转换为数字信号。

(2) 使用传感器检测温度、湿度、光照：使用 ENS210 温湿度传感器和 LTR390 光敏传感器读取相应的数据，使用传感器制造商提供的库函数编写程序进行数据读取和处理。

(3) 将数据传入数据库：使用 Python 编写程序将传感器数据传入数据库，可以使用 PostgreSQL 等关系型数据库进行存储。使用 psycopg2 连接数据库，并执行 SQL 语句进行数据的插入和更新。

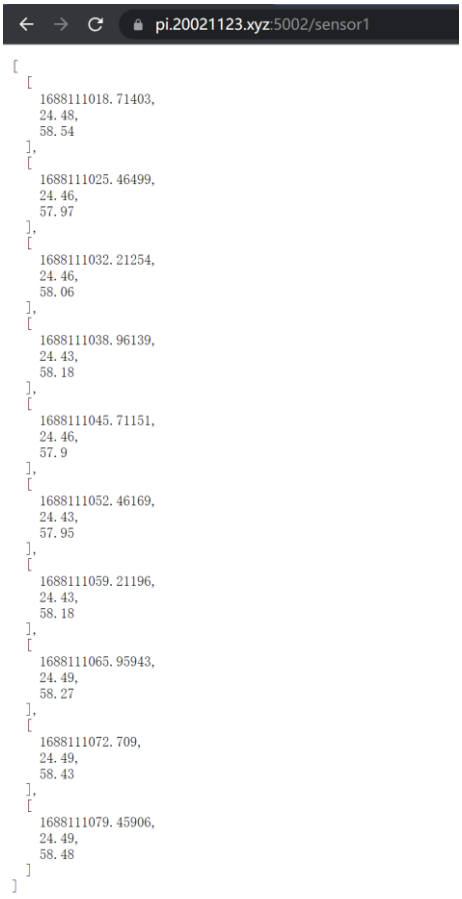

图 2- 4 Json API 示例

(4) 每个传感器的数据传入单独的数据库：为每个传感器创建一个单独的数据库表，将该传感器读取到的数据存储到该表中。可以使用 CREATE TABLE 语句创建表格，使用 INSERT INTO 语句插入数据。

(5) 计算前面数据的平均值：使用 Python 编写程序定期计算每个传感器的数据的平均值，并将结果存储到数据库中。可以使用 Python 的时间模块和定时器，每隔一定时间执行计算程序，并使用 UPDATE 语句更新数据库中的数据。

(6) 碰到异常数据，传入单独的异常数据库：在程序中添加异常数据的处理逻辑，如果检测到异常数据，将其存储到单独的异常数据库中。可以使用 CREATE TABLE 语句创建异常数据表格，使用 INSERT INTO 语句插入数据。

(7) 在 Flask 应用程序中编写相应的 SQL 语句，实现接收起始时间戳与终止时间戳两个参数，用于查询并返回相应的 JSON 格式的数据。

5. 开机自启设计

由于系统要实现无人值守的可用性，综合来看我们选择了 Systemd 作为所有服务的自启守护程序，通过设置 restart=always 能保证遇到错误崩溃时的自动重启。同时也可以使用 Linux 自带的 journalctl 命令查看程序日志。

6. 前端页面设计

根据实际需求和考虑到系统的特定要求，我们选择了 Vue 框架和 Element Plus 组件库作为前端开发工具。Vue 提供了清晰的文档和友好的生态系统，使得开发人员可以快速上手并构建出具有良好交互性的用户界面。Element Plus 组件库则为我们提供了丰富的 UI 组件，能够满足系统界面的设计需求，并且与 Vue 框架兼容良好，有助于加快开发进度。使用 vue-element-plus 响应式框架为基础，完成了面板 dashboard、检测 detect、传感器 Operation 三个页面的编写。

(1) 数据可视化

使用开源 echarts 的 js 库实现，仿照官网 https://echarts.apache.org/ 的文档

(2) 浏览器端识别

为减少树莓派的负担，yolo 的识别放在浏览器客户端上实现。为此使用了 TensorFlow js 库并将 yolo 的 .pt 格式模型转化为了 web 模型。

由于 YOLO 模型对于输入图像返回为[boxes, offset, classes]三元组，需要自行在前端用 canvas 画布绘制识别框，流程如图：

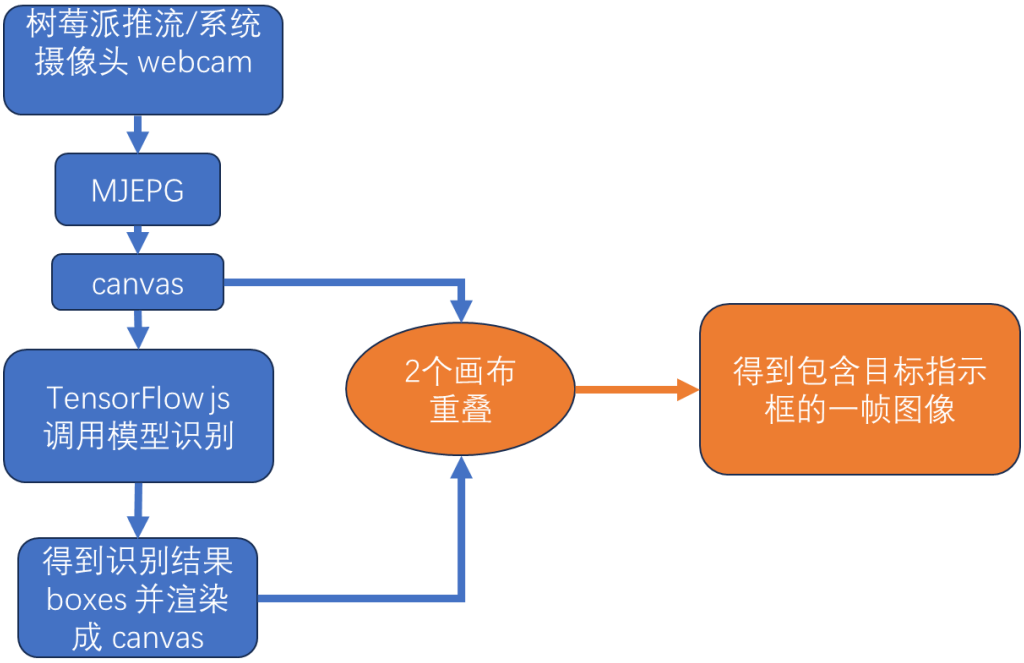

图 2-5 显示预测结果

(3) 注册登录组件

前端使用 Vue 框架和 Element Plus 组件库构建了登录和注册页面，通过 axios 发送请求到后端 API。后端使用 Express 框架搭建了一个简单的 RESTful API 服务器，处理登录和注册的请求。路由配置实现了登录拦截功能，根据 requiresAuth 元信息判断是否需要登录权限。用户登录后，通过 localStorage 保存登录状态和用户信息

图 2-6 注册登录系统

7. 报警系统方案设计

蜂鸣器是一种能够产生可听到声音的电子器件，通常被广泛应用于电子产品、仪器设备、报警器、警报系统、手机、电脑、汽车等领域，用于警报、提示或信号传输。蜂鸣器的工作原理非常简单，通过控制蜂鸣器所接收的电平高低来实现蜂鸣器的发声或停止发声。一般情况下，高电平会使蜂鸣器发出声音，而低电平则会使其停止发声。尽管蜂鸣器是一种简单的电子器件，但在工业生产、通讯、安防等领域中扮演着重要的角色，成为了不可或缺的设备之一。它的广泛应用使得人们能够在需要警报或提示的情况下迅速做出反应，确保安全和有效的信息传递。因此，我们决定采用蜂鸣器作为报警系统的一部分，利用其发出的声音来警示相关人员，以便在火灾或其他紧急情况下能够及时采取行动。这种选择将帮助我们确保警报的可靠性和有效性，为项目的安全性和可靠性提供保障。

图 2-7 蜂鸣器

除了物理警报，在警报发生时还应能对目标用户发送远程信息提醒形式的警报。如邮件、微信推送等。

2.2 针对复杂工程问题的方案实现

1. 传感器数据采集

(1) 传感器数据收集

采用一个 sensor.py 的 Python 文件，调用传感器官方提供的库函数和 postgresql 的库使用，关键代码如下

```
def printData1():
    t_value, h_value = ens210.ENS210_GetMeasure()
    # 温度和湿度
    global temperature1
    global humidity1

    temperature1 =
```

```
float(format(ens210.ENS210_Convert_Temperature(t_value), '.2f'))
    humidity1 = float(format(ens210.ENS210_Convert_Humidity(h_value),
'.2f'))

    print(f"Temp: {temperature1}\tHumi: {humidity1}\t",end='')
def repeat():

def printData3():
    global light_Data1, light_Data2
    if ltr390.LTR390UV_GetDataStatus() == True:
        if ltrmode == ltr390.LTR390UV_MODE_ALS:
            als = ltr390.LTR390UV_GetALS()
            lux = ltr390.LTR390UV_GetLUX(als, 1)
            print("ALS: %d\tLux: %f\r\n" % (als, lux))
            light_Data1 = float(als)
            light_Data2 = float(round(lux,2))
        else:
            uvs = ltr390.LTR390UV_GetUVS()
            uvi = ltr390.LTR390UV_GetUVI(uvs, 1)
            print("UVS: %d\tUVI: %f\r\n" % (uvs, uvi))
            light_Data1 = float(uvs)
            light_Data2 = float(round(uvi,2))
    time.sleep(0.5)

if __name__==’__main__’:
    global temperature1
    #合法数据，正常写入数据库
    if(testDataLegal()):
        #当前时间
        tim = datetime.now()
        # 插入数据到 sensor1 表
        cursor.execute('''INSERT INTO SENSOR1(tistamp, temp, humi) VALUES
(%f,%f,%f)''' %(float(tim.timestamp()),float(temperature1),float(humidity
1)))
        conn.commit()
        print("Records inserted........ SENSOR1")
        #插入数据到 sensor2 表
        cursor.execute('''INSERT INTO SENSOR2(tistamp, als, lux) VALUES
(%f,%f,%f)''' %(float(tim.timestamp()),float(light_Data1),float(light_Dat
a2)))
        conn.commit()
```

```
        print("Records inserted........ SENSOR2")
        # Closing the connection
        #不需要关闭，control + c 就关了
        #conn.close()

    #不合法的数据，写入不合法数据库
    else:
        tim = datetime.now()
        cursor.execute('''INSERT INTO SENSORERRORDATA(tistamp, temp,
humi,als,lux) VALUES
(%f,%f,%f,%f,%f)''' %(float(tim.timestamp()),float(temperature1),float(hu
midity1),float(light_Data1),float(light_Data2)))
        conn.commit()
        print("Records inserted........ SENSORERRORDATA")
    createTimer()
```

(2) 数据库分布式

使用 Postgresql 数据库的 Citus 插件

1) 安装 Citus：首先，在 PostgreSQL 数据库集群中安装了 Citus。

2) 建分布式表：使用 Citus 提供的 create_distributed_table() 函数将现有表转换为分布式表。这个函数会将表的数据分散到多个节点。

3) 选择分布键：在将表转换为分布式表时，选择一个适当的分布键。分布键通常是一个你经常用来过滤查询的列。

安装 Citus

```
#导入公共 GPG 密钥：
wget -qO- https://repos.citusdata.com/community/gpgkey | sudo apt-key add
-
```

添加 Citus 社区版存储库：

```
sudo apt-add-repository "deb
https://repos.citusdata.com/community/debian/ $(lsb_release -cs) main"
```

更新本地包索引并安装 Citus：

```
sudo apt-get update
sudo apt-get install citus
```

数据库加载 Citus 插件

```
CREATE EXTENSION citus;
```

指定要分片的表（sensor1）和要分片的列（tistamp 也是主键）

```
SELECT create_distributed_table('sensor2', 'tistamp');
```

2. 推流部分的实现

对于树莓派视频采集和推流，采用一个 Python 脚本实现，关键代码如下

```
class BroadcastOutput(object):
    def __init__(self, camera):
        print('Spawning background conversion process')
        self.converter = Popen([
            'ffmpeg',
            '-f', 'rawvideo',
            '-pix_fmt', 'yuv420p',
            '-s', '%dx%d' % camera.resolution,
            '-r', str(float(camera.framerate)),
            '-i', '-',
            '-f', 'mpeg1video',
            '-b', '800k',
            '-r', str(float(camera.framerate)),
            '-'],
            stdin=PIPE, stdout=PIPE, stderr=io.open(os.devnull, 'wb'),
            shell=False, close_fds=True)
def main():
    print('Initializing camera')
    with picamera.PiCamera() as camera:
        camera.resolution = (WIDTH, HEIGHT)
        camera.framerate = FRAMERATE
        camera.vflip = VFLIP # flips image rightside up, as needed
        camera.hflip = HFLIP # flips image left-right, as needed
        sleep(1) # camera warm-up time
        print('Initializing websockets server on port %d' % WS_PORT)
        WebSocketWSGIHandler.http_version = '1.1'
        websocket_server = make_server(
            '', WS_PORT,
            server_class=WSGIServer,
            handler_class=WebSocketWSGIRequestHandler,
            app=WebSocketWSGIApplication(handler_cls=StreamingWebSocket))
        websocket_server.initialize_websockets_manager()
        websocket_thread = Thread(target=websocket_server.serve_forever)
        print('Initializing HTTP server on port %d' % HTTP_PORT)
        http_server = StreamingHttpServer()
        http_thread = Thread(target=http_server.serve_forever)
        print('Initializing broadcast thread')
        output = BroadcastOutput(camera)
```

```
        broadcast_thread = BroadcastThread(output.converter,
websocket_server)
        print('Starting recording')
        camera.start_recording(output, 'yuv')
        try:
            print('Starting websockets thread')
            websocket_thread.start()
            print('Starting HTTP server thread')
            http_thread.start()
            print('Starting broadcast thread')
            broadcast_thread.start()
            while True:
                camera.wait_recording(1)
```

3. 火焰识别的实现

模型训练为将标注好的数据按指定格式放入文件夹，然后执行

```
python .\yolov5\train.py –img 640 –batch 8–epochs 10 –
data ./fire_config.yaml –weights yolov5s.pt –workers 0
```

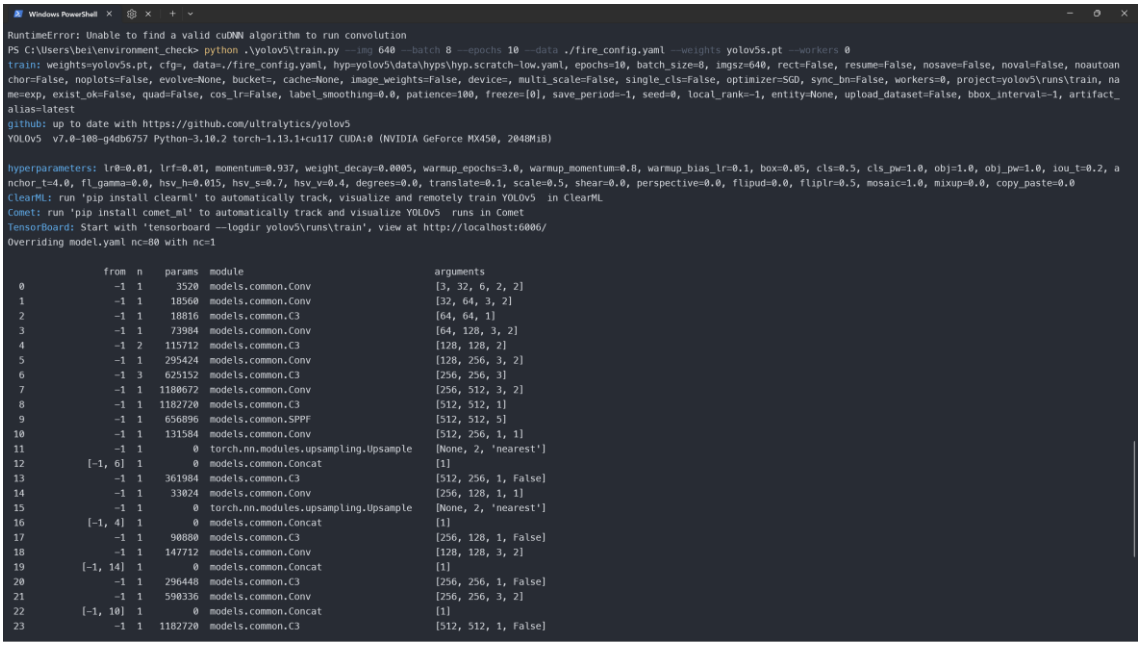

图 2-8 开始训练

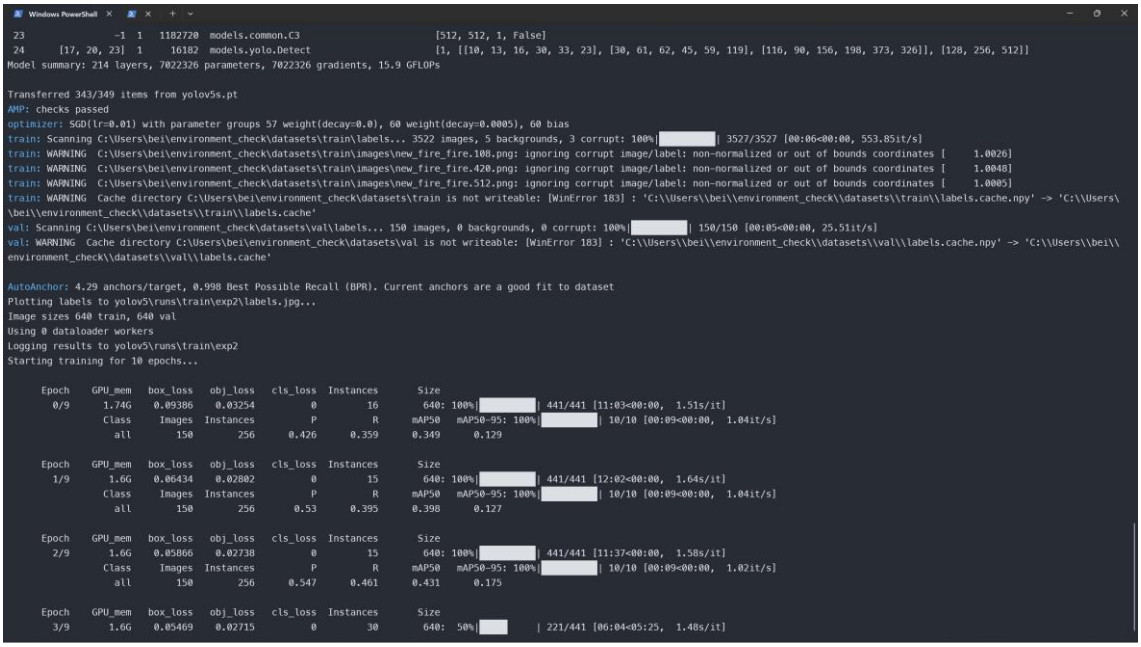

图 2-9 训练过程

为使用 tensorflow js，需先将 pytorch 生成的 .pt 格式的模型文件转化为 tensorflow 格式：

```
python .\export.py --weights ..\models\best.pt --include tfjs
```

输出如下

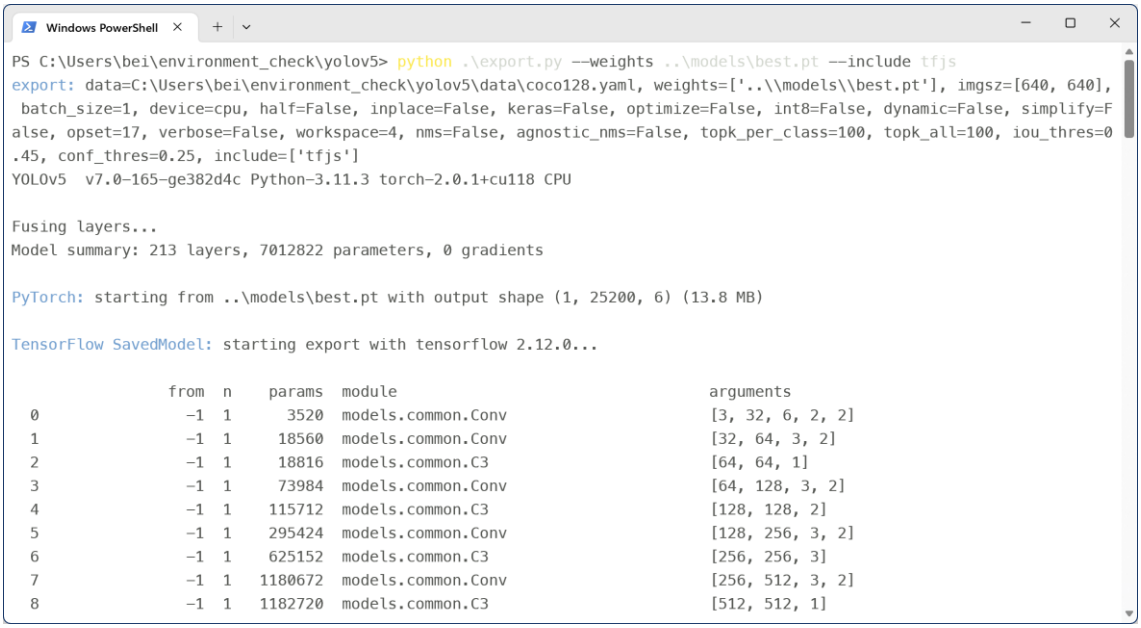

图 2-10 开始转换模型

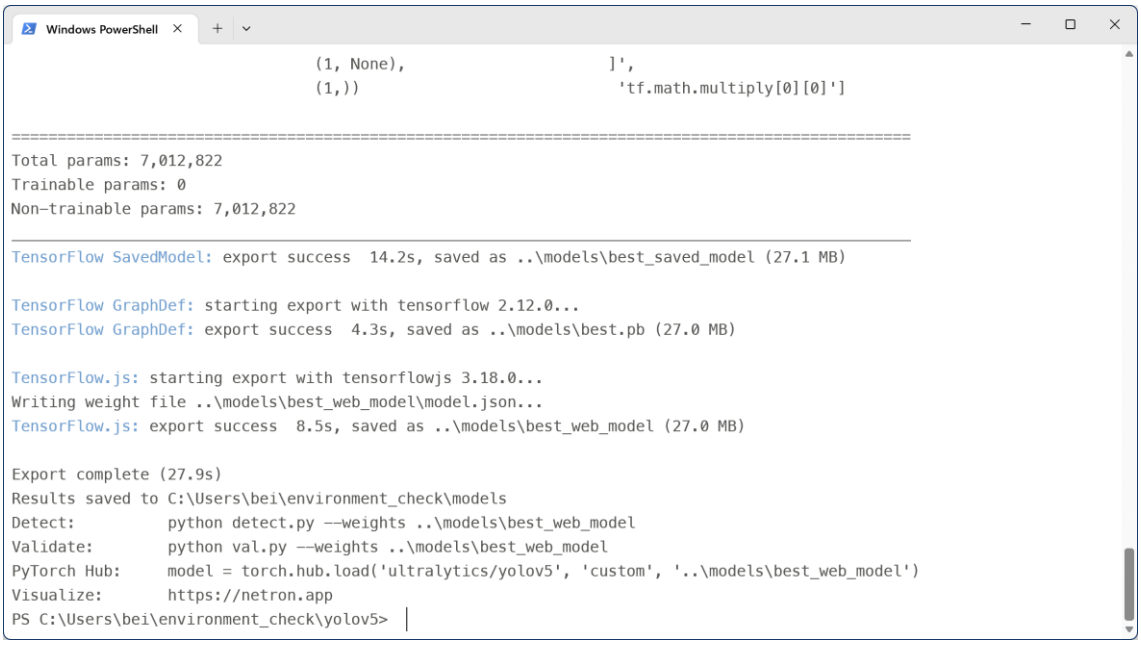

图 2- 11 转换完成

模型格式如图

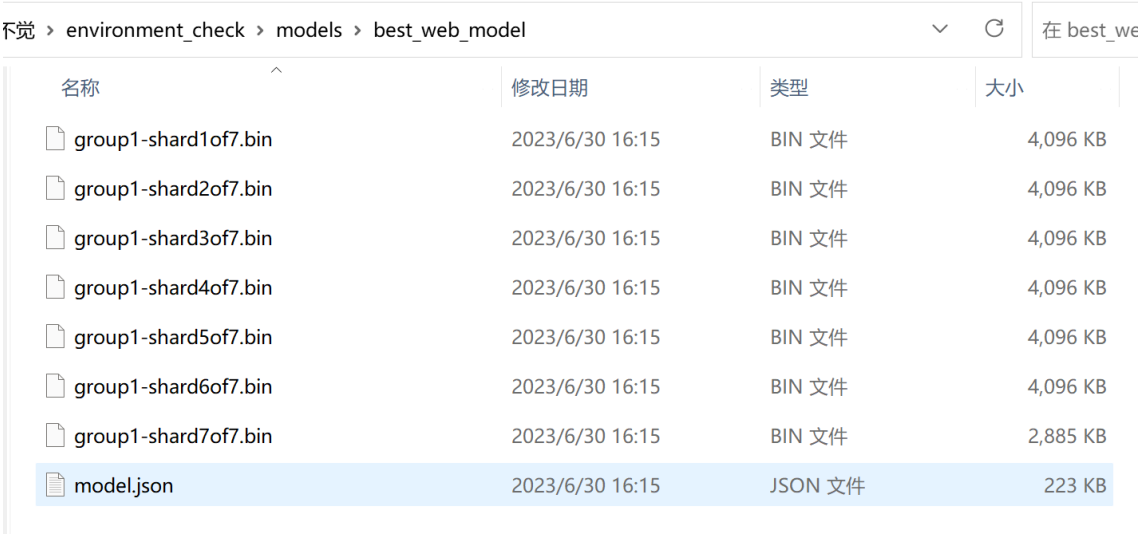

图 2- 12 web model 格式

4. Web 后端的实现

针对与传感器相关的路由，我们可以接收前端传递的 start 、end 和 limit 三个参数，分别用于控制开始时间戳、截至时间戳和查询条数的值，以便进行数据库的查询操作。当这三个参数都没有给出时，默认查询最新的十条记录数据。

如果给出了 start 和 end 的值，但未给出 limit 的值，则会尝试以时间为轴均匀查询从记录开始到当前时间的 200 条数据。最后，我们将查询到的数据以 JSON 格式返回给前端。另外，对于路由 /api/send-wechat 和 /api/send-QQ，我们将分别调用第三方平台的接口和使用 QQ 邮箱来及时向用户发送报警信息。通过以上的设计，我们能够方便地进行数据库查询，并根据用户的需求提供灵活的查询功能。同时，通过与第三方平台的集成，我们能够确保报警信息及时发送给用户，以便他们能够及时采取必要的措施。这些功能的实现将进一步提升我们项目的实用性和用户体验。

代码如下：

```
 1 import os
 2 import requests
 3 import psycopg2
 4 import smtplib
 5 from flask import Flask, render_template,request, url_for,
redirect,jsonify
 6 from email.mime.text import MIMEText
 7 HOST='::'
 8 #HOST='0.0.0.0'
 9 PORT=5001
10 LIMIT = 10
11 app = Flask(__name__)
12
13 def get_db_connection():
14     conn = psycopg2.connect(host='127.0.0.1',
15                             database='sensordata',
16                             user='postgres',
17                             password='3.1415926',port="5432")
18     return conn
19 @app.route('/sensor1')
20 def sensor1():
21     start_tis_tamp = request.args.get('start')
22     end_tis_tamp = request.args.get('end')
23     limit = request.args.get('limit')
24     conn = get_db_connection()
25     sql = 'SELECT * FROM sensor1'
26     if end_tis_tamp and start_tis_tamp:
27         sql += ' WHERE tistamp >= %s AND tistamp <= %s'
28     elif start_tis_tamp:
```

```
 29         sql += ' WHERE tistamp >= %s'
 30     elif end_tis_tamp:
 31         sql += ' WHERE tistamp <= %s'
 32 
 33     sql += ' order by tistamp '
 34     if limit:
 35         sql += f' OFFSET (SELECT COUNT(*) FROM sensor1) - {limit}'
 36         sql += ' LIMIT %s' % (limit)
 37     elif (not end_tis_tamp) and (not start_tis_tamp):
 38         sql += f' OFFSET (SELECT COUNT(*) FROM sensor1) - {LIMIT}'
 39         sql += ' LIMIT %s' % (LIMIT)
 40     elif start_tis_tamp and end_tis_tamp:
 41         sql = 'SELECT * FROM sensor1 WHERE tistamp >= %s AND tistamp
<= %s AND MOD(tistamp::int, (SELECT COUNT(*)/200 FROM sensor1 WHERE
tistamp >= %s AND tistamp <= %s)) = 0 ORDER BY tistamp LIMIT 200'
 42     cur = conn.cursor()
 43     if start_tis_tamp and end_tis_tamp and limit:
 44         cur.execute(sql, (start_tis_tamp,end_tis_tamp))
 45     elif start_tis_tamp and end_tis_tamp:
 46         cur.execute(sql,
(start_tis_tamp,end_tis_tamp,start_tis_tamp,end_tis_tamp))
 47     elif start_tis_tamp:
 48         cur.execute(sql, (start_tis_tamp,))
 49     elif end_tis_tamp:
 50         cur.execute(sql, (end_tis_tamp,))
 51     else:
 52         cur.execute(sql)
 53     measuredatas = cur.fetchall()
 54     cur.close()
 55     conn.close()
 56     response = jsonify(measuredatas)
 57     response.headers.add('Access-Control-Allow-Origin', '*')
 58     return response
 59 @app.route('/sensor2')
 60 def sensor2():
 61     start_tis_tamp = request.args.get('start')
 62     end_tis_tamp = request.args.get('end')
 63     limit = request.args.get('limit')
 64     conn = get_db_connection()
 65     sql = 'SELECT * FROM sensor2'
 66     if end_tis_tamp and start_tis_tamp:
```

```
 67         sql += ' WHERE tistamp >= %s AND tistamp <= %s'
 68     elif start_tis_tamp:
 69         sql += ' WHERE tistamp >= %s'
 70     elif end_tis_tamp:
 71         sql += ' WHERE tistamp <= %s'
 72 
 73     sql += ' order by tistamp '
 74     if limit:
 75         sql += f' OFFSET (SELECT COUNT(*) FROM sensor2) - {limit}'
 76         sql += ' LIMIT %s' % (limit)
 77     elif (not end_tis_tamp) and (not start_tis_tamp):
 78         sql += f' OFFSET (SELECT COUNT(*) FROM sensor2) - {LIMIT}'
 79         sql += ' LIMIT %s' % (LIMIT)
 80     elif start_tis_tamp and start_tis_tamp:
 81        sql = 'SELECT * FROM sensor2 WHERE tistamp >= %s AND tistamp
<= %s AND MOD(tistamp::int, (SELECT COUNT(*)/200 FROM sensor2 WHERE
tistamp >= %s AND tistamp <= %s)) = 0 ORDER BY tistamp LIMIT 200'
 82 
 83     cur = conn.cursor()
 84     if end_tis_tamp and start_tis_tamp and limit:
 85         cur.execute(sql, (start_tis_tamp,end_tis_tamp))
 86     elif end_tis_tamp and start_tis_tamp:
 87         cur.execute(sql,
(start_tis_tamp,end_tis_tamp,start_tis_tamp,end_tis_tamp))
 88     elif start_tis_tamp:
 89         cur.execute(sql, (start_tis_tamp,))
 90     elif end_tis_tamp:
 91         cur.execute(sql, (end_tis_tamp,))
 92     else:
 93         cur.execute(sql)
 94     measuredatas = cur.fetchall()
 95     cur.close()
 96     conn.close()
 97     response = jsonify(measuredatas)
 98     response.headers.add('Access-Control-Allow-Origin', '*')
 99     return response
100 @app.route('/api/send-wechat',methods=['GET'])
101 def send():
102 
url='https://sctapi.ftqq.com/SCT214259TNqTtqelm8r5agYzSDZttV785.send'
103     title_tis_tamp=request.args.get('title')
```

```
104     desp_tis_tamp=request.args.get('desp')
105     if title_tis_tamp:
106         pass
107     else:
108         title_tis_tamp='EXCEPTION OCCURRED'
109     if desp_tis_tamp:
110         pass
111     else:
112         desp_tis_tamp=666
113 
114     myParams={'title':title_tis_tamp,'desp':desp_tis_tamp,'channel':9}
115     res=requests.post(url=url,data=myParams)
116     print('url:',res.request.url)
117     print (res.text)
118     return "Notification sent"
119 @app.route('/api/send-QQ',methods=['GET'])
120 def send2():
121     desp_tis_tamp=request.args.get('desp')
122     title_tis_tamp=request.args.get('title')
123     if title_tis_tamp:
124         pass
125     else:
126         title_tis_tamp='Exception Occurred'
127     if desp_tis_tamp:
128         pass
129     else:
130         desp_tis_tamp='666'
131     HOST='smtp.qq.com'
132     FROM='1968374004@qq.com'
133     TO=request.args.get('to')
134     if TO:
135         pass
136     else:
137         TO='1752862657@qq.com'
138     msg=MIMEText(desp_tis_tamp,'html','utf-8')
139     msg['tltle']=title_tis_tamp
140     msg['From']=FROM
141     msg['To']=TO
142 
143     server=smtplib.SMTP_SSL(HOST,465)
144     server.login(FROM,'lkyvupmrvdgnfccc')
```

```
145     server.sendmail(FROM,[TO],msg.as_string())
146     server.quit()
147     return "Notification sent"
148 if __name__ == '__main__':
149     app.run(host=HOST, port=PORT, debug =True)
```

5. 开机自启动部分

使用 systemd 编写服务的方式实现开机自启动与故障自动重启功能

(1) 开机启动 ENS210 传感器代码

因为 ENS210 传感器开机后第一次启动会固定报错，开机后尝试多次启动

```
1 #!/usr/bin/env python3
 2
 3 # RTrobot ENS210 Sensor Test
 4 # http://rtrobot.org
 5
 6 #开机进行 5 次数据测试
 7
 8 import RTrobot_ENS210
 9 import sys
10 import time
11 import RPi.GPIO as GPIO
12
13 ens = RTrobot_ENS210.RTrobot_ENS210()
14 buf = ens.ENS210_Init()
15
16 """
17 if buf == False:
18     print("ENS210 initialize error.")
19     while True:
20         pass
21 else:
22     print("ENS210 initialize register finished.")
23
24 """
25 ens.ENS210_SetSingleMode(False)
26
27 try:
28     #进行 5 次测试
29     for i in range(5):
30         t_value,h_value=ens.ENS210_GetMeasure()
31
```

```
#print(format(ens.ENS210_Convert_Temperature(t_value),'.2f'),format(ens.E
NS210_Convert_Humidity(h_value),'.2f'),)
32        #time.sleep(1)
33
34
35 except KeyboardInterrupt:
36     pass
37 GPIO.cleanup()
```

(2) 开机启动 ENS210 传感器服务

```
1. [Unit]
2. # 必须执行网络
3. Requires=network-online.target
4. # 在网络启动后启动程序
5. After=network-online.target
6.
7. [Service]
8. Type=simple
9. User=root
10. Group=root
11.
12. # 需要自启动的代码
13.
    ExecStart=/home/pi/environment_check/sensor/ENS210/ENS210_systemctl.p
y
14.
15. [Install]
16. WantedBy=multi-user.target
```

(3) 传感器自启动服务

```
1. [Unit]
2. # 必须执行网络
3. Requires=network-online.target
4. # 在网络启动后启动程序
5. After=network-online.target ENS210_systemd.service
6.
7. [Service]
8. Type=simple
9. User=pi
10. Group=pi
11. Restart=on-failure
12. RestartSec=5s
```

```
WorkingDirectory=/home/pi/environment_check/sensor/
# 需要自启动的代码
ExecStart=/usr/bin/python /home/pi/environment_check/sensor/begin.py

[Install]
WantedBy=multi-user.target
```

(4) Flask 自启动服务

```
[Unit]
# 必须执行网络
Requires=network-online.target
# 在网络启动后启动程序
After=network-online.target ENS210_systemd.service

[Service]
Type=simple
User=pi
Group=pi
Restart=on-failure
RestartSec=5s

# 需要自启动的代码
ExecStart=python /home/pi/environment_check/app/app.py

[Install]
WantedBy=multi-user.target
```

(5) 推流服务器自启动服务

```
[Unit]
# 必须执行网络
Requires=network-online.target
# 在网络启动后启动程序
After=network-online.target ENS210_systemd.service

[Service]
Type=simple
User=pi
Group=pi
WorkingDirectory=/opt/pistreaming
Restart=on-failure
RestartSec=5s

```

```
15. # 需要自启动的代码
16. ExecStart=/opt/pistreaming/server.py
17.
18. [Install]
19. WantedBy=multi-user.target
```

6. 前端部分

采用 vue 响应式框架和 element-plus 的组件库。其中 detect 页面的识别采用 tensorflow js 以达到在浏览器上实时进行火焰识别的功能。

(1) 面板（主页）dashboard.vue 主要代码

```
 52 onMounted(() => {
 53   // Show loading notice
 54   var canvas = document.getElementById("videoCanvas");
 55
 56   // Setup the WebSocket connection and start the player
 57   var client = new WebSocket(WS_URL + ":" + WS_PORT);
 58   // @ts-ignore
 59   var player = new jsmpeg(client, { canvas: canvas });
 60   // console.log(player);
 61   getsensor(data_sensor1, "sensor1");
 62   getsensor(data_sensor2, "sensor2");
 63 });
 64
 85 function getsensor(data, sensor: string): void {
 86   fetch("//" + DB_URL + ":" + DB_PORT + "/" + sensor)
 87     .then((res) => res.json())
 88     .then((out) => {
 89       let n = Math.min(DATA_NUM, out.length);
 90       out = out.slice(-n);
 91       for (let i = 0; i < n; i++)
 92         if (data.value.length < n || data.value[n - 1][0] < out[i][0]
* 1000) {
 93           if (data.value.length == n) data.value.shift();
 94           data.value.push([out[i][0] * 1000, out[i][1], out[i][2]]);
 95           //console.log([out[i][0]*1000,out[i][1],out[i][2]]);
 96         }
 97     })
 98     .catch(function (e) {
 99       console.log("error: " + e.toString());
100     });
101 }
```

(2) 传感器页面 Operation.vue 主要函数

```
56 function getsensor(data, sensor: string): void {
 57   fetch("//" + DB_URL + ":" + DB_PORT + "/" + sensor)
 58     .then((res) => res.json())
 59     .then((out) => {
 60       let n = Math.min(DATA_NUM, out.length);
 61       out = out.slice(-n);
 62       for (let i = 0; i < n; i++)
 63         if (data.value.length < n || data.value[n - 1][0] < out[i][0]
* 1000) {
 64           if (data.value.length == n) data.value.shift();
 65           data.value.push([out[i][0] * 1000, out[i][1], out[i][2]]);
 66         }
 67     })
 68     .catch(function (e) {
 69       console.log("error: " + e.toString());
 70     });
 71 }
 72
 73 function getsensorft(data, sensor: string, timerange): void {
 74              if( !timerange||!timerange[0]){
 75              getsensor(data, sensor);
 76              return;
 77              }
 78              console.log(timerange);
 79 //console.log(timerange);
 80   fetch("//" + DB_URL + ":" + DB_PORT + "/" + sensor +
"?start="+parseInt(timerange[0].getTime()/1000) + "&end=" +
parseInt(timerange[1].getTime()/1000))
 81     .then((res) => res.json())
 82     .then((out) => {
 83       let n = out.length;
 84       data.value.splice(0, data.value.length);
 85       out = out.slice(-n);
 86       for (let i = 0; i < n; i++)
 87         if (data.value.length < n || data.value[n - 1][0] < out[i][0]
* 1000) {
 88           if (data.value.length == n) data.value.shift();
 89           data.value.push([out[i][0] * 1000, out[i][1], out[i][2]]);
 90         }
```

```
 91     })
 92     .catch(function (e) {
 93       console.log("error: " + e.toString());
 94     });
 95 }
```

(3) 识别页面 detect.vue 关键部分

```
324 const detectObjects = async () => {
325   if (!isModelReady.value){
326     return;
327   }
328   if(inputsource.value == 1 && !isVideoStreamReady.value) {
329     /*
330     console.log("start gg");
331     console.log(isModelReady.value);
332     console.log(inputsource.value);
333     console.log(isVideoStreamReady.value);
334     */
335
336     requestAnimationFrame(detectObjects);
337     return;
338 // get another frame
339
340   }
341
342   var inputShape = model.inputs[0].shape;
343   const [modelWidth, modelHeight] = inputShape.slice(1, 3);
344
345  // tf.engine().startScope(); // start scoping tf engine
346   //console.log(videoRef.value);
347   const [input, xRatio, yRatio] = preprocess(inputsource.value == 0 ?
player.canvas : videoRef.value, modelWidth, modelHeight);
348 try{
349   await model.executeAsync(input).then((res) => {
350     const [boxes, scores, classes] = res.slice(0, 3);
351     const boxes_data = boxes.dataSync();
352     const scores_data = scores.dataSync();
353     const classes_data = classes.dataSync();
354     renderBoxes(canvasRef.value, classThreshold, boxes_data,
scores_data, classes_data, [
355       xRatio,
```

```
356       yRatio,
357     ]); // render boxes
358
359     tf.dispose(res); // clear memory
360   }) .catch(error => {
361       console.log('Run detect failed ', error);
362       throw (error);
363     });
364   }catch(error){
365       console.log('Run detect failed 2', error);
366       throw (error);
367   }
368   animateid=requestAnimationFrame(detectObjects); // get another
frame
369  // tf.engine().endScope(); // end of scoping
370 };
```

(4) 登录页面设计实现

登录组件：

- 登录组件设计如下：
- 使用 Element Plus 组件库构建登录页面，包括一个卡片容器、表单和按钮。
- 使用 data 属性定义了一个 ruleForm 对象，包含用户名和密码，用于绑定表单输入框的值。
- 使用 rules 属性定义了表单的校验规则，包括用户名和密码不能为空。
- 使用 methods 属性定义了 submitForm、resetForm 和 register 方法。
- submitForm 方法用于提交登录表单，通过 axios 发送 POST 请求到后端 API，验证用户名和密码。
- register 方法用于注册新用户，通过 axios 发送 POST 请求到后端 API，将新用户信息保存到文件中。
- 如果登录或注册成功，将返回的 token 保存到 localStorage 中，并跳转到之前访问的页面或默认页面。
- 如果登录或注册失败，根据错误类型显示相应的错误信息。

由于登录注册是在原有框架基础基础之上进行，导致在登录注册的时候也会有侧边框出现。我们将侧边框组件绑定 showSidebar 方法。

```
<BaseSide v-if="showSidebar" />
export default {
```

```
  computed: {
    showSidebar() {
      // 如果当前路由的元信息中 requiresAuth 为 true，则显示侧边栏
      return this.$route.meta.requiresAuth;
    }
  }
};
```

接着在路由配置内部设置元信息，如果元信息为 true，展示侧边框，登录组件设置为 flase。

```
{
    path: '/login',
    component: login,
    name: 'login',
    props:{
      msg: "登录"
    },
    meta: { requiresAuth: false }
  },
```

效果如下：

点击注册按钮，触发 register 方法，通过 axios 发送 POST 请求到后端 API。

```
async register() {
...
      try {
        const response = await axios.post(
          "http://pi.20021123.xyz:3000/api/register",
          {
            username: this.ruleForm.uname,
            password: this.ruleForm.password,
          }
        );
    ...
}
```

之后进入后端 API。后端 API 的逻辑是从文件中读取现有账户信息，检查账号是否已存在，添加新账号到数组，将更新后的账户信息保存到文件。

由于账号密码总量不大，所以没有使用数据库，而是使用 json 文件进行储存。

```
// 存储账户信息的文件路径
const accountsFilePath = path.join(__dirname, "accounts.json");
```

登录逻辑 submitForm 方法首先通过 axios 发送 POST 请求到后端 API。后端登录逻辑代码如下，逻辑是先从我们创立的文件中读取现有账户信息，之后查找是否有匹配的用户，成功则返回 tocken 以及用户信息。

```
app.post("/api/login", (req, res) => {
  const { username, password } = req.body;
  // 从文件中读取现有账户信息
  const accounts = readAccountsFromFile();
  // 查找匹配的账户
  const account = accounts.find(
    (acc) => acc.username === username && acc.password === password
  );
  if (account) {
    // 以下是一个示例 token 字符串
    const token = "example-token-string";
    // 登录成功，返回 token 和用户信息
    res.status(200).json({
      message: "登录成功",
      token: token,
      user: {
        username: account.username,
        // 其他用户信息可以在这里添加
      },
    });
  } else {
    // 登录失败，返回一个错误信息
    res.status(401).json({ message: "用户名或密码错误" });
  }
});
```

路由方面实现判断

实现登录前不允许查看某些路径的内容，需要在 Vue Router 中使用全局前置守卫（beforeEach）来进行权限控制。全局前置守卫可以在路由跳转之前进行检查，判断用户是否已经登录，从而决定是否允许访问特定路由。

下面是如何在路由配置中添加全局前置守卫来阻止未登录用户访问受保护的路由：

首先，isLoggedIn()来监测用户是否登录。

```
function isLoggedIn() {
  return !!localStorage.getItem('userToken');
}
```

```
然后，在路由配置中添加 beforeEach 守卫：
router.beforeEach((to, from, next) => {
  if (to.matched.some(record => record.meta.requiresAuth)) {
    // 判断该路由是否需要登录权限
    if (!isLoggedIn()) {
      // 如果未登录，则跳转到登录页面
      next({ path: '/login', query: { redirect: to.fullPath } });
    } else {
      // 如果已登录，在这里可以进行其他的权限验证步骤
      next();
    }
  } else {
    // 对于不需要登录权限的路由，直接放行
    next();
  }
});
```

最后，在登录组件 login.vue 里面，需要设置一个标志，使用 localStorage 中的 userToken 来表示用户已登录。

```
    submitForm(formName) {
      this.$refs[formName].validate(async (valid) => {
        if (valid) {

           // ...用户验证成功
            // 登录成功
            // 假设后端返回的响应中包含了 token 和用户信息
            const { token, user } = response.data;
            localStorage.setItem("userToken", token);

            // 跳转到之前想要访问的页面或默认页面
            const redirect = this.$route.query.redirect || "/digram";
            this.$router.push(redirect);
            this.$message.success("登录成功！");

        } else {
          //  验证失败逻辑
        }
      });
}
```

7. 对于信息警报通知模块的实现

(1) 微信和邮箱推送

我们决定分别在微信和邮箱上实现警报的推送。我们可以通过设置参数控制输出的内容和主题，然后，构造请求参数，发送 POST 请求，将参数通过 URL 编码的形式传递给指定的 URL。最后，打印请求的 URL 和服务器返回的响应结果，完成微信推送。根据这个学期我们学习的计算机网络知识，我们可以创建 SMTP 连接，使用 QQ 邮箱的 SMTP 服务器地址和端口号 465。登录发件人邮箱，调用 sendmail 方法发送邮件，并调用 quit 方法关闭 SMTP 连接。

下面是代码实现：

```
1 @app.route('/api/send-wechat',methods=['GET'])
 2 def send():
 3 
url='https://sctapi.ftqq.com/SCT214259TNqTtqelm8r5agYzSDZttV785.send'
 4     title_tis_tamp=request.args.get('title')
 5     desp_tis_tamp=request.args.get('desp')
 6     if title_tis_tamp:
 7         pass
 8     else:
 9         title_tis_tamp='EXCEPTION OCCURRED'
10     if desp_tis_tamp:
11         pass
12     else:
13         desp_tis_tamp=666
14 
15     myParams={'title':title_tis_tamp,'desp':desp_tis_tamp,'channel':9}
16     res=requests.post(url=url,data=myParams)
17     print('url:',res.request.url)
18     print (res.text)
19     return "Notification sent"
20 
21 @app.route('/api/send-email',methods=['GET'])
22 def send2():
23     desp_tis_tamp=request.args.get('desp')
24     title_tis_tamp=request.args.get('title')
25     if title_tis_tamp:
26         pass
27     else:
28         title_tis_tamp='Exception Occurred'
29     if desp_tis_tamp:
30         pass
```

```
    else:
        desp_tis_tamp='666'
    HOST='smtp.qq.com'
    FROM='1968374004@qq.com'
    TO=request.args.get('to')
    if TO:
        pass
    else:
        TO='1752862657@qq.com'
    msg=MIMEText(desp_tis_tamp,'html','utf-8')
    msg['tltle']=title_tis_tamp
    msg['From']=FROM
    msg['To']=TO

    server=smtplib.SMTP_SSL(HOST,465)
    server.login(FROM,'lkyvupmrvdgnfccc')
    server.sendmail(FROM,[TO],msg.as_string())
    server.quit()
    return "Notification sent"
```

(2) 对于物理警报通知模块的实现

我们可以利用树莓派支持的 GPIO 接口来方便地实现蜂鸣器发声并充当警报的功能。通过控制引脚的状态、时间、频率、脉冲和占空比等参数值，我们能够轻松地控制蜂鸣器的工作方式。这样，我们可以根据需要在适当的时候触发蜂鸣器，让其发出相应的声音，提醒相关人员注意。通过树莓派的 GPIO 接口与蜂鸣器的结合，我们能够简单而有效地实现警报功能，为项目的安全性和实用性增添一层保障。这种灵活的控制方式使得我们能够根据具体的需求和场景来定制警报的音频特性，提供更加个性化和可定制化的警报体验。

```
import time
import RPi.GPIO as GPIO

def alert():

    alert_fm = 13 # 设置引脚
    alert_time = 1 # 设置发出警报的时间
    alert_Hz = 4978 # 声音频率

    GPIO.setmode(GPIO.BOARD)
    GPIO.setwarnings(False)
```

```
12      GPIO.setup(alert_fm, GPIO.OUT) # 设置 GPIO 13 为输出
13      pwm = GPIO.PWM(alert_fm, alert_Hz) # 设置 GPIO 13 为 PWM 输出，设置脉
冲
14
15      pwm.start(0)    # 初始化占空比为 0%
16      pwm.ChangeDutyCycle(50)     # 改变占空比为 50%
17      time.sleep(alert_time)  # 开始发声
18
19      pwm.stop()
20      GPIO.cleanup()
21
22  if __name__ == '__main__':
23      alert()
```

第三章 测试环境构建与测试驱动开发

3.1 传感器测试

传感器数据实时更新，如图所示

```
python

pi in environment_check/sensor at raspberrypi on  main [↑$✘!?] took 10.1s …
➜ python measure.py
Firware:5.4.6
ENS210 initialize register finished.
ENS160 initialize register finished.
LTR390UV initialize register finished.
mode: ALS

02:13:07        Temp: 33.38     Humi: 28.04     AQI: 3  TVOC: 323ppb    eCO2: 809ppm    ALS: 87 Lux: 17.400000

02:13:10        Temp: 33.30     Humi: 28.18     AQI: 3  TVOC: 320ppb    eCO2: 808ppm    ALS: 88 Lux: 17.600000

02:13:13        Temp: 33.21     Humi: 28.02     AQI: 3  TVOC: 301ppb    eCO2: 801ppm    ALS: 91 Lux: 18.200000

02:13:15        Temp: 33.13     Humi: 28.03     AQI: 3  TVOC: 309ppb    eCO2: 804ppm    ALS: 87 Lux: 17.400000

02:13:18        Temp: 33.10     Humi: 28.16     AQI: 3  TVOC: 301ppb    eCO2: 801ppm    ALS: 87 Lux: 17.400000

02:13:21        Temp: 33.10     Humi: 28.22     AQI: 3  TVOC: 320ppb    eCO2: 808ppm    ALS: 87 Lux: 17.400000

02:13:24        Temp: 33.05     Humi: 28.11     AQI: 3  TVOC: 315ppb    eCO2: 806ppm    ALS: 87 Lux: 17.400000

02:13:26        Temp: 33.01     Humi: 28.13     AQI: 3  TVOC: 314ppb    eCO2: 806ppm    ALS: 88 Lux: 17.600000

02:13:29        Temp: 32.96     Humi: 28.14     AQI: 3  TVOC: 294ppb    eCO2: 798ppm    ALS: 88 Lux: 17.600000

02:13:32        Temp: 32.94     Humi: 28.25     AQI: 3  TVOC: 317ppb    eCO2: 807ppm    ALS: 87 Lux: 17.400000

02:13:35        Temp: 32.93     Humi: 28.36     AQI: 3  TVOC: 299ppb    eCO2: 801ppm    ALS: 90 Lux: 18.000000

02:13:37        Temp: 32.93     Humi: 28.44     AQI: 3  TVOC: 304ppb    eCO2: 802ppm    ALS: 88 Lux: 17.600000
```

图 3-1 传感器实时数据

数据库列表

名称	拥有者	字元编码	校对规则	Ctype	存取权限
postgres	postgres	UTF8	zh_CN.UTF-8	zh_CN.UTF-8	
sensordata	postgres	UTF8	zh_CN.UTF-8	zh_CN.UTF-8	
template0	postgres	UTF8	zh_CN.UTF-8	zh_CN.UTF-8	=c/postgres + postgres=CTc/postgres

图 3-2 传感器数据库

```
 架构模式 |      名称       |  类型  |  拥有者
----------+-----------------+--------+----------
 public   | sensor1         | 数据表 | postgres
 public   | sensor2         | 数据表 | postgres
 public   | sensorerrordata | 数据表 | postgres
(3 行记录)
```

图 3-3 传感器数据表

传感器数据记录为 Unix 时间戳，方便不同时区的人们使用

传感器数据连接数据库，将数据插入对应的表里面

```
1.  import psycopg2 ##导入
2.
3.  ## 通过 connect 方法，创建连接对象 conn
4.  ## 这里连接的是本地的数据库
5.  conn = psycopg2.connect(database="sensordata", user="postgres",
6.   password="3.1415926", host="127.0.0.1", port="5432")
7.
8.  #Setting auto commit false
9.  conn.autocommit = True
10.
11. #Creating a cursor object using the cursor() method
12. cursor = conn.cursor()
13.
14.         tim = datetime.now()
15.         # 插入数据到 sensor1 表
16.         cursor.execute('''INSERT INTO SENSOR1(tistamp, temp, humi)
VALUES
(%f,%f,%f)''' %(float(tim.timestamp()),float(temperature1),float(humidity
1)))
17.
18.         conn.commit()
19.         print("Records inserted........ SENSOR1")
20.
21.         #插入数据到 sensor2 表
22.         cursor.execute('''INSERT INTO SENSOR2(tistamp, als, lux)
VALUES
(%f,%f,%f)''' %(float(tim.timestamp()),float(light_Data1),float(light_Dat
a2)))
```

```
23.
24.         conn.commit()
25.         print("Records inserted........ SENSOR2")
```

```
1683804879.402145       Temp: 23.83     Humi: 48.23     ALS: 0  Lux: 0.000000

Records inserted........ SENSOR1
Records inserted........ SENSOR2

1683804886.156849       Temp: 23.87     Humi: 48.08     ALS: 0  Lux: 0.000000

Records inserted........ SENSOR1
Records inserted........ SENSOR2
```

图 3-4 插入传感器数据

效果如上图，一共五个数据，时间戳，Temp 是温度，Humi 是湿度，ALS，LUX 是光照数据的两个参数，如果插入成功，就会打印出已经插入，Records inserted..... SENSOR2 表示成功插入 SENSOR2 的表

对于异常数据的处理，我们使用一般常见的数据作为异常数据的识别，要是数据异常，打印出哪个数据异常，并且插入数据异常表，效果展示如下

```
1683806444.338795       Temp: 23.91     Humi: 48.37     ALS: 0  Lux: 0.000000

-2.0
temperature ERROR
Records inserted........ SENSORERRORDATA
```

图 3-5 错误数据处理

当前探测器测出温度为-2.0 摄氏度，打印出了”temporature ERROR”的信息，方便管理员知道哪里出错

下面展示异常数据表的数据

```
sensordata=# select * from sensorerrordata
;
     tistamp      | temp | humi  | als | lux
------------------+------+-------+-----+-----
  1683806365.8619 |   -1 | 48.25 |   0 |   0
 1683806446.08323 |   -2 | 48.37 |   0 |   0
 1683806452.83377 |   -2 | 48.41 |   0 |   0
 1683806459.58208 |   -2 | 48.52 |   0 |   0
(4 行记录)
```

图 3-6 错误数据存放位置

可以看到四个时间戳的数据有异常。

3.2 推流服务测试

一开始使用 rtsp 推流模式，延迟较高，如图 3-7：

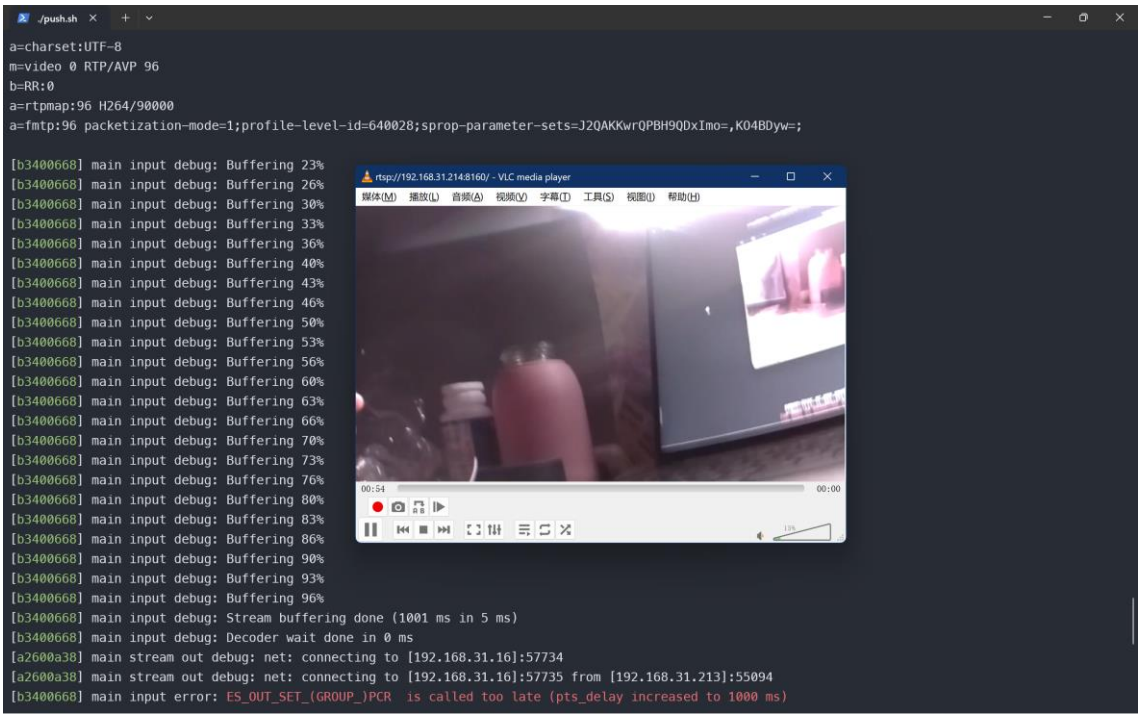

图 3- 7 rtsp 推流

之后采用 websocket 二进制推流，可在浏览器 F12 工具中查看，如图 3-8：

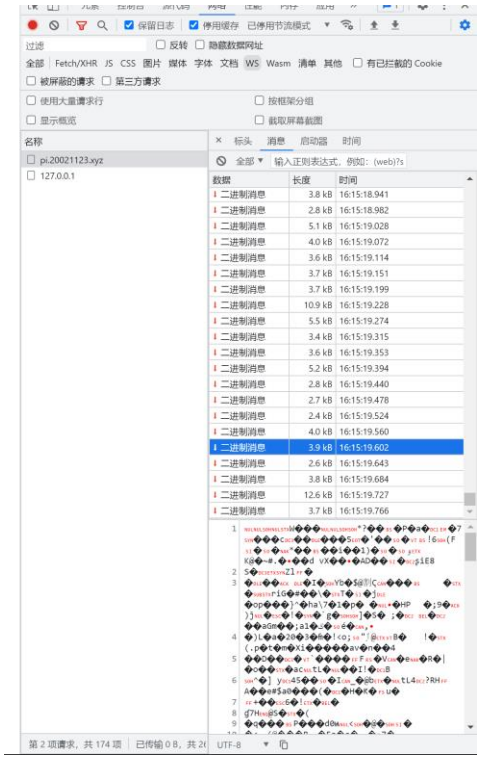

图 3-8 推流数据查看

延迟降低为 0.2s 以内，如图 3-9：

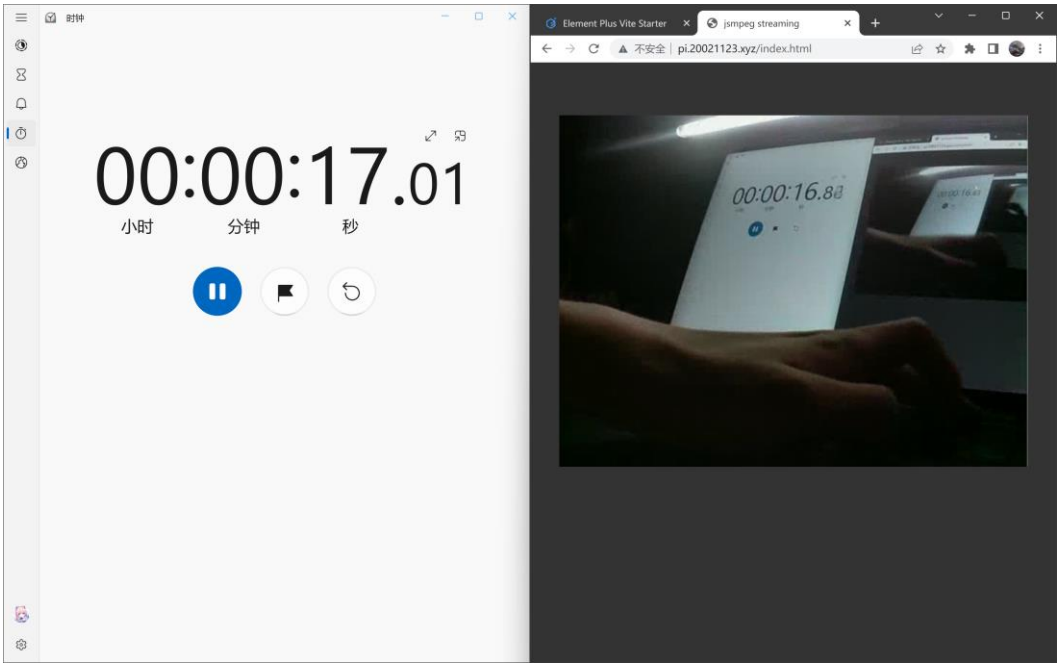

图 3-9 推流延迟测试

3.3 火焰识别测试

对于训练好的模型，直接从摄像头获取数据并测试识别，如图 3-10：

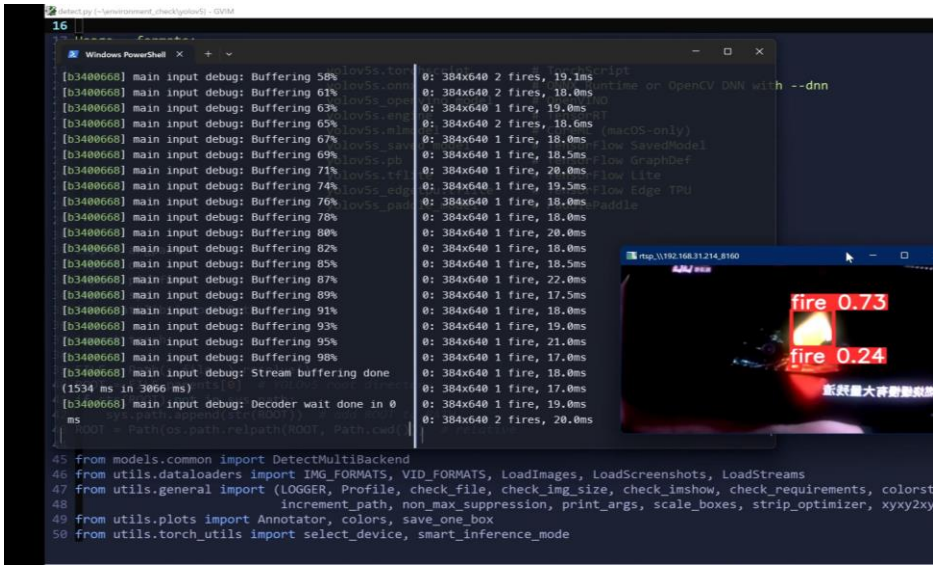

图 3-10 火焰识别测试

运行 predict 后会在目录下生成测试集的识别情况文件，包括带检测框的 prediction 和 loss、召回率、准确率等数据。如图 3-11，3-12 所示。

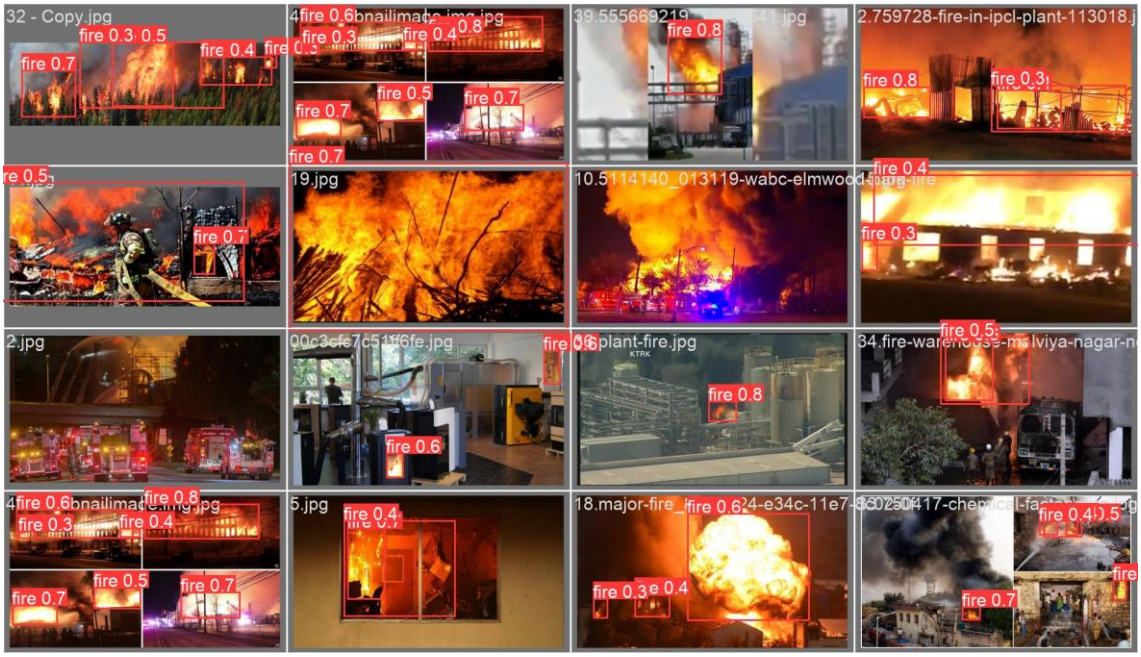

图 3-11 目录下 prediction 示例

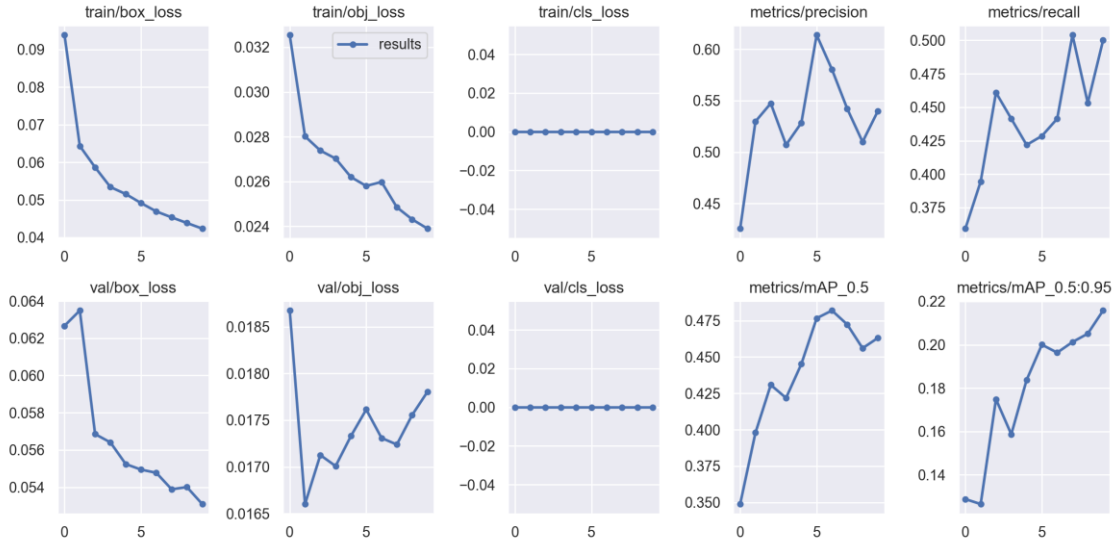

图 3- 12 loss，准确率，精确率，召回率等数据

3.4 后端测试

直接访问后端服务器 API，得到传感器数据（消息推送 API 在另一部分），如图 3-13 所示。

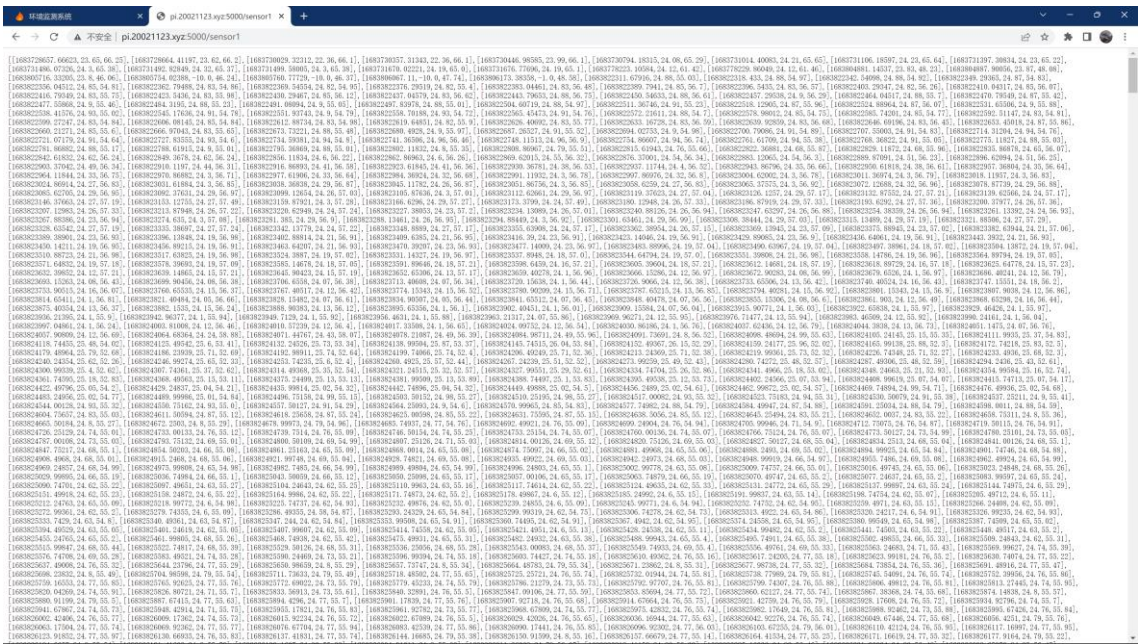

图 3- 13 后端测试

3.5 服务自启动测试

1. 传感器自启动

我们的服务分为了传感器数据获取 sensor.service、Flask 后端 flask.service 和推流服务器 stream.service 三个服务，相互独立，在系统重启后使用 Systemctl 命令可以查看运行状态：

```
➜sudo systemctl status sensor.service
● sensor.service
   Loaded: loaded (/lib/systemd/system/sensor.service; enabled; vendor preset: en
   Active: active (running) since Thu 2023-06-29 20:58:04 CST; 12h ago
 Main PID: 500 (python)
    Tasks: 2 (limit: 4915)
   CGroup: /system.slice/sensor.service
           └─500 /usr/bin/python /home/pi/environment_check/sensor/begin.py

6月 30 09:34:58 raspberrypi python[500]: 1688088876.288493         Temp: 27.54
6月 30 09:34:58 raspberrypi python[500]: Records inserted........ SENSOR1
6月 30 09:34:58 raspberrypi python[500]: Records inserted........ SENSOR2
6月 30 09:34:58 raspberrypi python[500]: 1688088883.038025         Temp: 27.49
6月 30 09:34:58 raspberrypi python[500]: Records inserted........ SENSOR1
6月 30 09:34:58 raspberrypi python[500]: Records inserted........ SENSOR2
6月 30 09:34:58 raspberrypi python[500]: 1688088889.787408         Temp: 27.43
6月 30 09:34:58 raspberrypi python[500]: Records inserted........ SENSOR1
6月 30 09:34:58 raspberrypi python[500]: Records inserted........ SENSOR2
6月 30 09:34:58 raspberrypi python[500]: 1688088896.536827         Temp: 27.51
```

图 3- 14 sensor 服务状态

2. Flask 后端自启动

```
● flask.service
   Loaded: loaded (/lib/systemd/system/flask.service; enabled; vendor preset: ena
   Active: active (running) since Thu 2023-06-29 23:49:06 CST; 9h ago
 Main PID: 13960 (python)
    Tasks: 3 (limit: 4915)
   CGroup: /system.slice/flask.service
           ├─13960 /usr/bin/python /home/pi/environment_check/app/app.py
           └─14400 /usr/bin/python /home/pi/environment_check/app/app.py

6月 29 23:49:07 raspberrypi python[13960]:    WARNING: Do not use the development
6月 29 23:49:07 raspberrypi python[13960]:    Use a production WSGI server instea
6月 29 23:49:07 raspberrypi python[13960]:  * Debug mode: on
6月 29 23:49:07 raspberrypi python[13960]:  * Running on http://[::]:5001/ (Press
6月 29 23:49:07 raspberrypi python[13960]:  * Restarting with stat
6月 29 23:49:08 raspberrypi python[13960]:  * Debugger is active!
6月 29 23:49:08 raspberrypi python[13960]:  * Debugger PIN: 111-237-949
6月 29 23:49:10 raspberrypi python[13960]: ::ffff:192.168.31.69 - - [29/Jun/2023
6月 29 23:49:11 raspberrypi python[13960]: ::ffff:192.168.31.69 - - [29/Jun/2023
6月 29 23:53:29 raspberrypi python[13960]: ::ffff:192.168.31.69 - - [29/Jun/2023
lines 1-19
```

图 3- 15 flask 服务状态

3. 推流服务器自启动

```
pi in environment_check at raspberrypi on  main [$!?] took 1m 6.2s …
➜ sudo systemctl status stream.service
● stream.service
   Loaded: loaded (/lib/systemd/system/stream.service; enabled; vendor preset: en
   Active: active (running) since Thu 2023-06-29 20:58:04 CST; 12h ago
 Main PID: 506 (python)
    Tasks: 21 (limit: 4915)
   CGroup: /system.slice/stream.service
           ├─506 python /opt/pistreaming/server.py
           └─650 ffmpeg -f rawvideo -pix_fmt yuv420p -s 640x480 -r 24.0 -i - -f m

6月  29 20:58:04 raspberrypi systemd[1]: Started stream.service.
6月  29 23:27:27 raspberrypi server.py[506]: Error when terminating the connection
6月  29 23:27:27 raspberrypi server.py[506]: Error when terminating the connection
6月  29 23:27:27 raspberrypi server.py[506]: Error when terminating the connection
lines 1-13/13 (END)
```

图 3- 16 stream 服务状态

3.6 前端网页测试

1. 主页 dashboard

主页包括树莓派摄像头实时画面，和两个传感器最近 10 条数据的折线图，如图 3-17，3-18 所示：

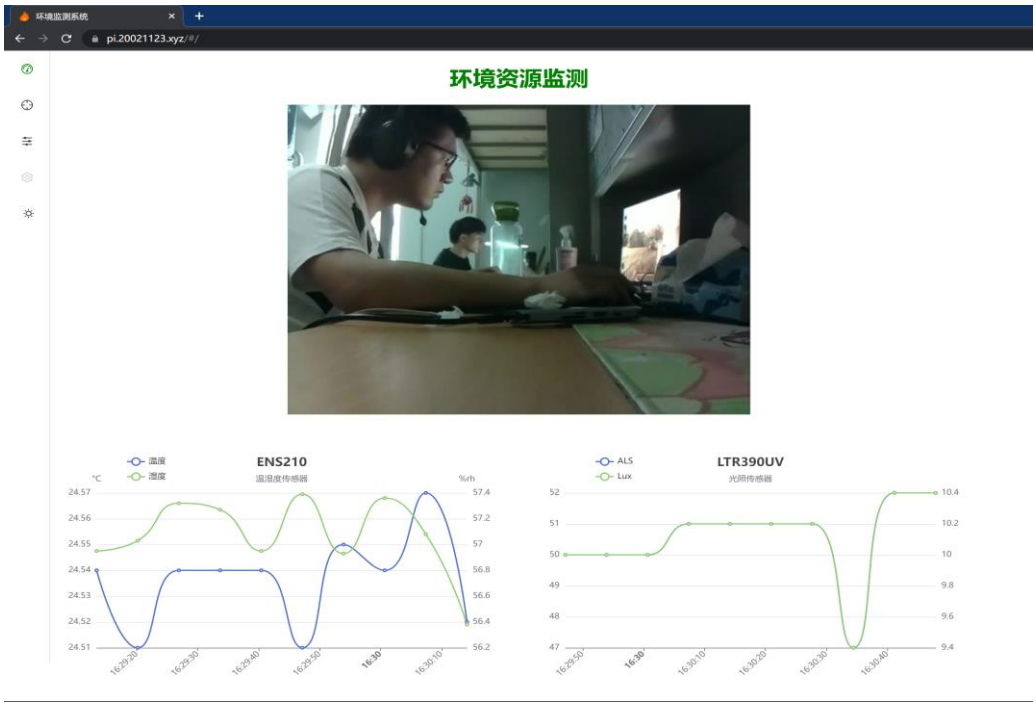

图 3- 17 主页

图 3- 18 夜间模式

2. 传感器页 Operation

传感器页可以查看某传感器的所有数据，默认可选最近 1 分钟、1 小时、1 天或一周，还可以自行选择数据起始时间，如图 3-19~3-21 所示：

图 3- 19 传感器页

图 3- 20 最近一天温度变化

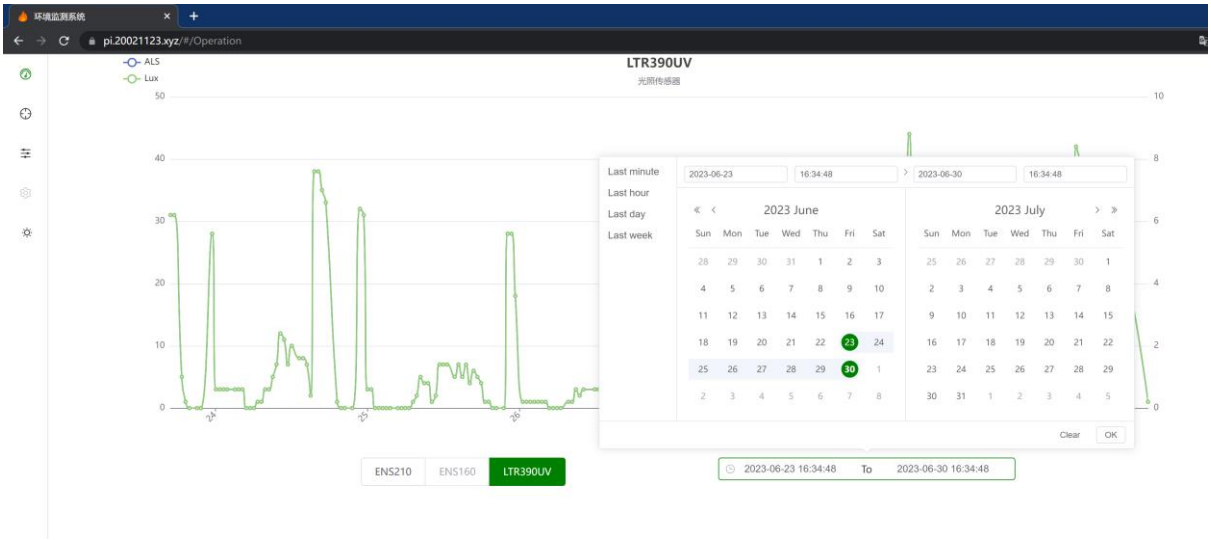

图 3- 21 自由选择起止时间

3. 检测页 detect

检测页调用了我们自行训练的火焰检测模型和 Yolov5n 自带的 80 个常见物品的识别，可以选择输入端为树莓派摄像头还是 webcam 系统摄像头，以方便调试。

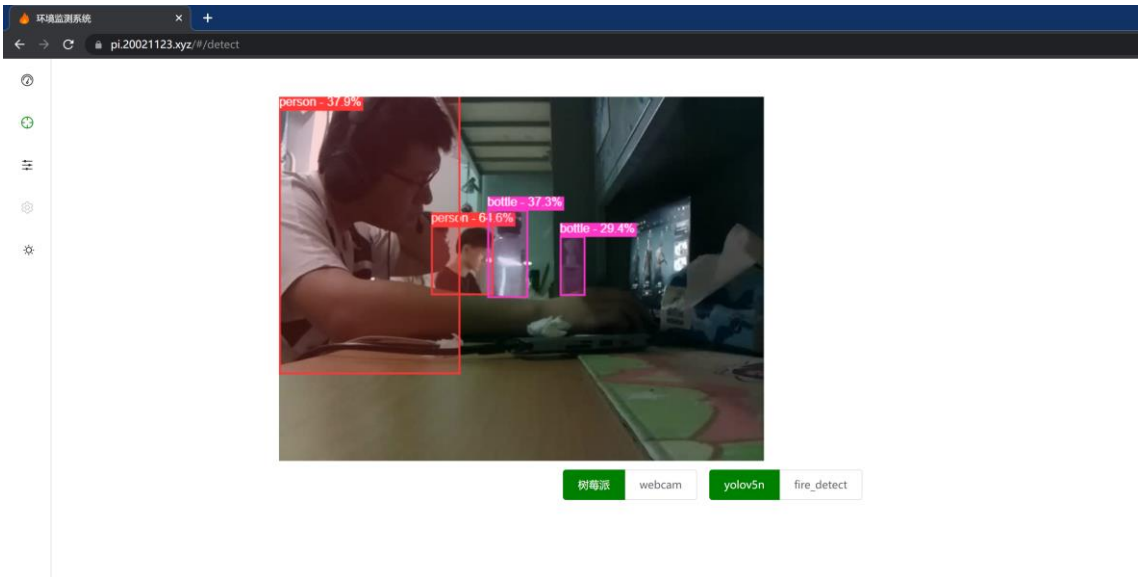

图 3- 22 检测页（默认使用 yolov5n 官方模型）

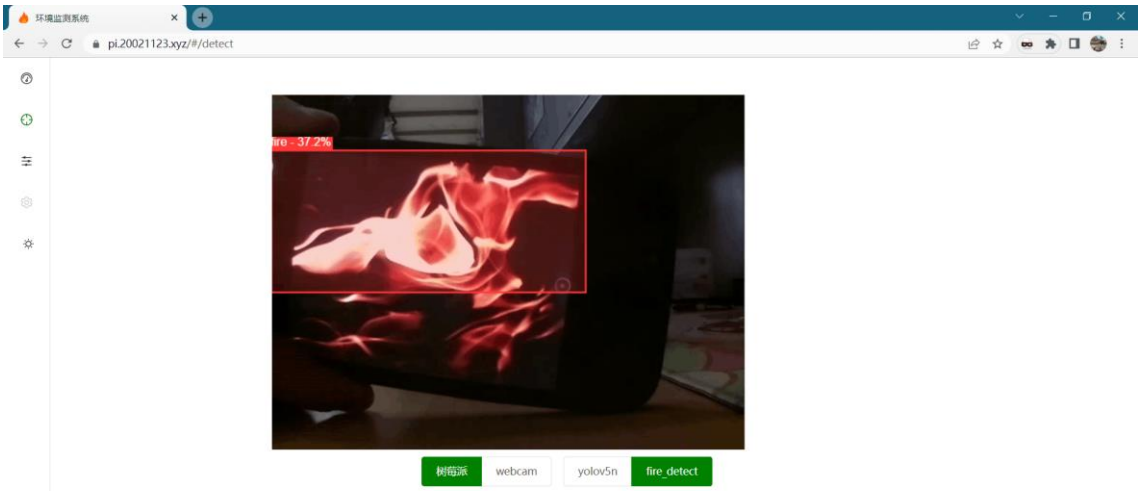

图 3- 23 使用火焰识别模型

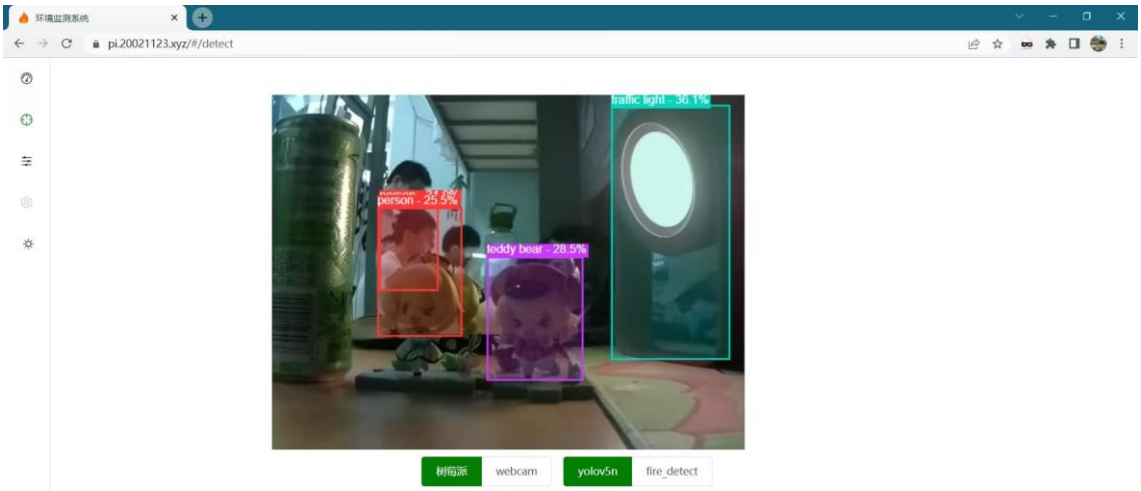

图 3- 24 多类别目标检测

当使用 HTTPS 时，可以支持 webcam 调用系统摄像头，如图 3-25，3-26。

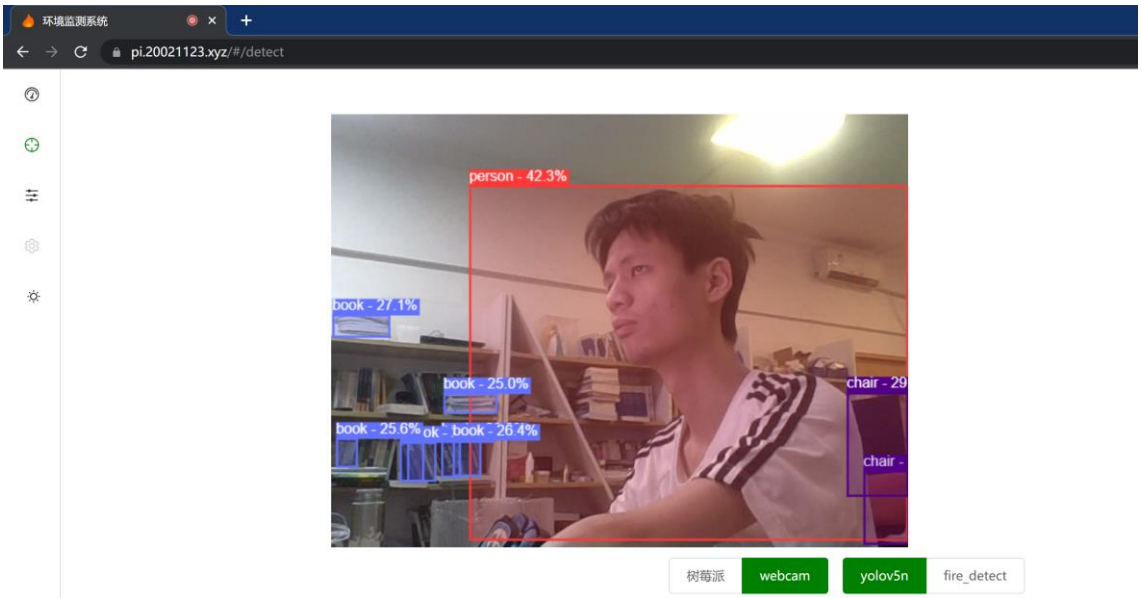

图 3- 25 系统摄像头检测

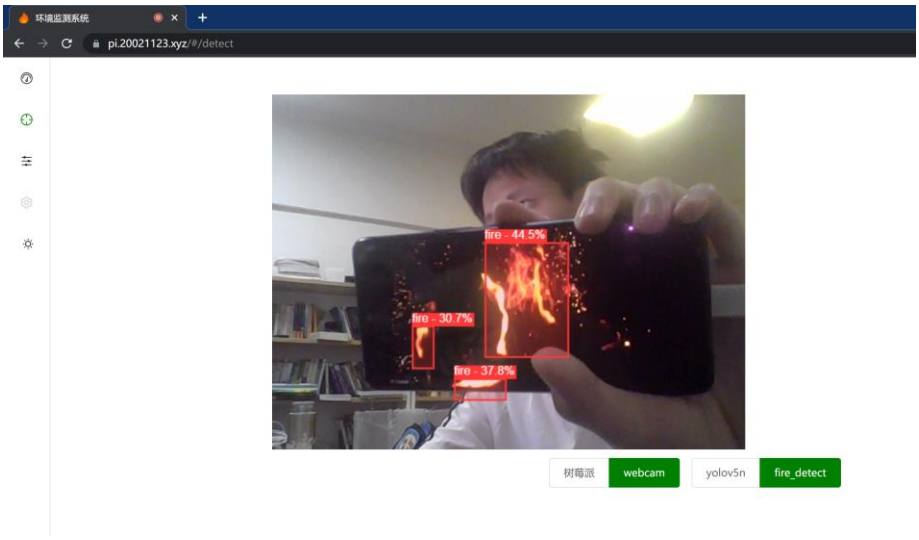

图 3- 26 webcam 检测火焰

4. 登陆测试

登录注册功能展示：

(1) 拦截功能：当用户未登录时，不能直接访问数据页面，如图 3-27：

图 3- 27 登录页面

当用户没有登录尝试直接访问传感器数据，如图 3-28：

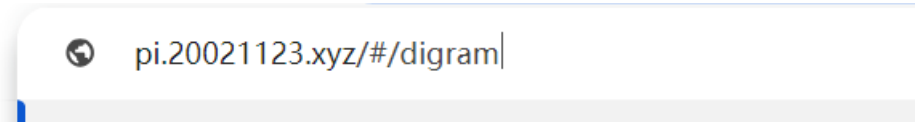

图 3- 28 尝试直接访问

此时访问会被拦截，如图 3-29。

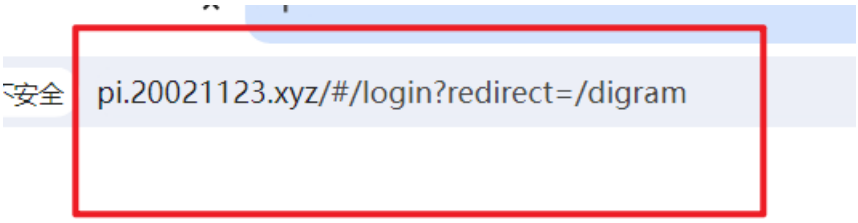

图 3- 29 访问被拦截

访问被拦截时自动跳转页面如图 3-30。

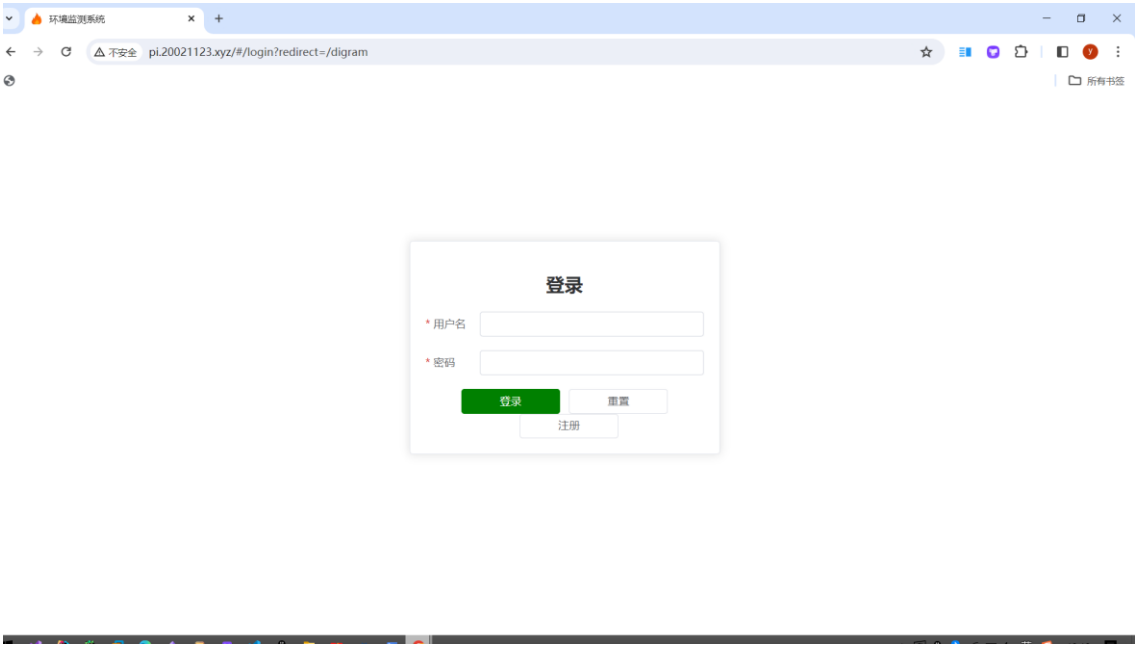

图 3- 30 访问被拦截

(1) 注册功能：

当用户完成注册，信息会被保存在 accounts.json 文件里面，下面是目前的用户。

st > backend > {...} accounts.json > ...

```
[
  {
    "username": "大胡子",
    "password": "111"
  },
  {
    "username": "大胡子测试",
    "password": "111"
  },
  {
    "username": "小明",
    "password": "1234"
  }
]
```

图 3- 31 已注册用户

现在注册新用户，如图 3-32：

图 3- 32 注册页面

后端保存账号密码文件正确添加，保存文件如图 3-33。

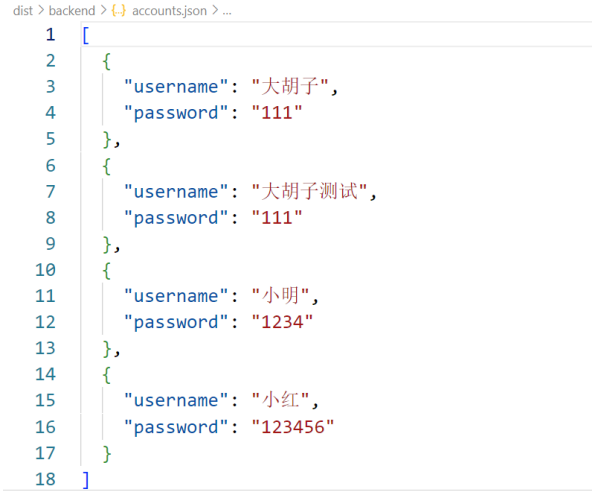

图 3- 33 保存成功

(2) 登录功能

使用新账号登录，如图 3-34：

图 3- 34 新用户登录

登录成功如图 3-35 所示。

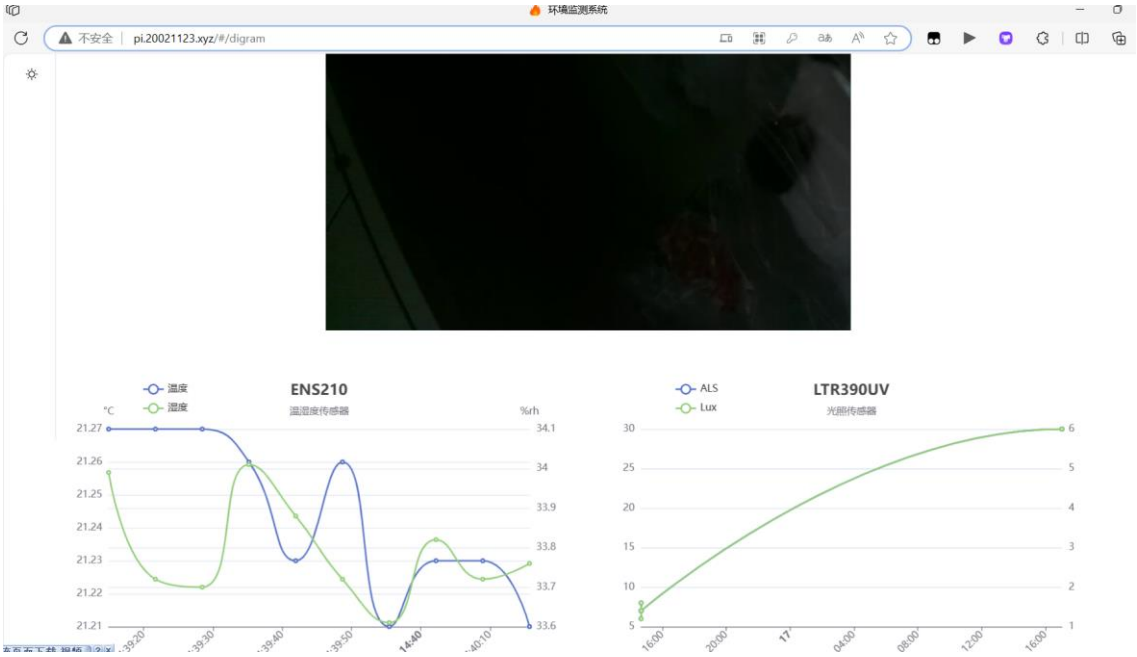

图 3- 35 登陆成功

(3) 重置功能

清零输入框数据

图 3- 36 重置前

图 3- 37 重置后

3.7 消息推送功能测试

在设计中，对微信推送和邮箱推送情况进行测试。

1. 微信推送测试（方糖酱）

若没有设置推送内容，则发送默认内容

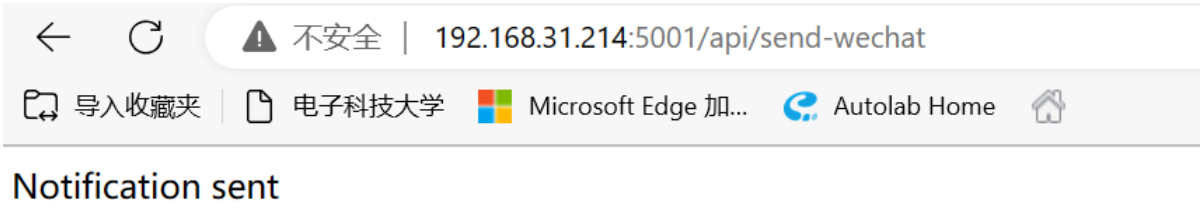

图 3- 38 无推送内容

发送结果如图 3-39 所示

图 3- 39 无推送内容发送默认值

若设置发送内容，则将设置的内容发送到指定公众号，如图 3-40

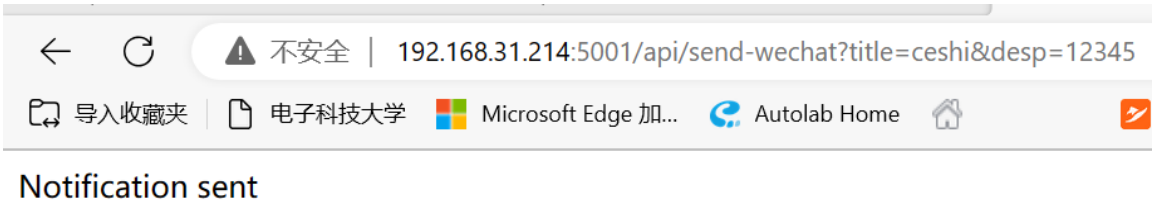

图 3- 40 指定推送内容

发送结果如图 3-41

图 3- 41 指定推送结果

方糖酱公众号的总体情况如图 3-42

图 3- 42 总体情况

2. 邮箱推送测试

若没有设置推送内容，则向默认邮箱发送默认内容，如图 3-43，3-44

图 3- 43 无内容设置

图 3- 44 默认推送结果

若指定对象和发送内容，则对目标对象发送指定内容，如图 3-45

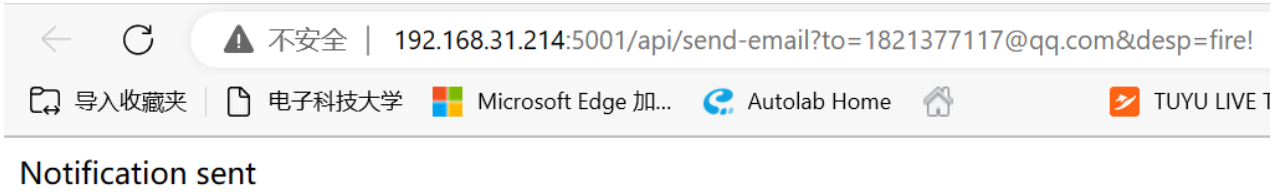

图 3- 45 指定推送内容及对象

对象邮箱接受结果如图 3-46

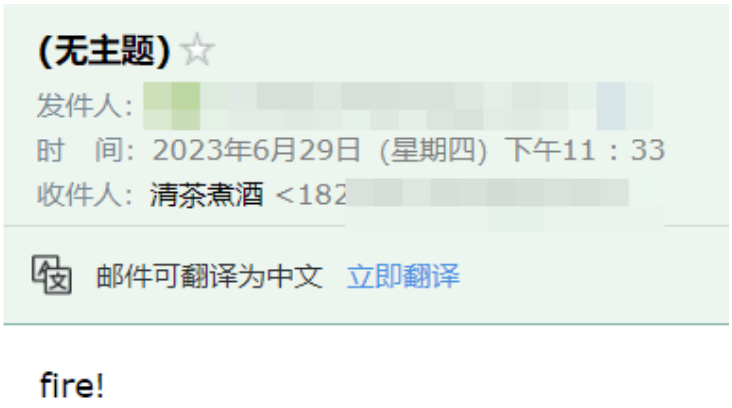

图 3- 46 推送结果

若发送对象错误，则返回错误信息，如图 3-47

图 3- 47 错误处理

第四章 知识技能学习情况

Systemd 是 Linux 系统工具，用来启动守护进程，使用 Systemd 来实现自定义服务的自启动[1][2]。

通过仔细研读 Flask 官方文档[3]和 Psycopg 官方文档[4]，我们不仅学习了它们的基础用法，而且能够熟练运用 Flask 框架编写适当的路由，并且通过 Psycopg 实现与数据库的连接。此外，我们还研究了 Postgresql 相关文档[5][6]以及教程，并尝试编写高性能的 SQL 查询语句，以优化数据库查询的效率。

通过学习方糖酱自带的教学文档，我们学习到了如何使用 post 请求发送推送通知。同时这个学期的计网学习，我们知道了如何利用 QQ 邮箱的 smtp/imap 服务器发送邮件通知。

TensorFlow.js 是一个用于使用 JavaScript 进行机器学习开发的库[7]，它支持使用 JavaScript 开发机器学习模型，并直接在浏览器或 Node.js 中使用机器学习模型。

Vue 是一款用于构建用户界面的 JavaScript 框架[8]。它基于标准 HTML、CSS 和 JavaScript 构建，并提供了一套声明式的、组件化的编程模型，帮助用户高效地开发用户界面。

第五章 分工协作与交流情况

分工协作

刘贝：作为组长，统筹与计划安排小组成员的具体工作内容；负责 AI 部分火焰识别的代码和整理汇总以及相关论文部分编写；前端网页部分的代码设计实现完成；识别模型的准备与转换，同时掌握项目总体的设计方向。

刘为丰：负责蜂鸣器部分的代码编写，数据库部分的数据库和对接的 api 以及相关论文部分编写。

雷镒鸣：负责传感器数据测量写入，判断异常数据库部分的数据库和对接的 api 以及相关论文部分编写，登录注册界面及拦截功能实现，数据库分布式功能实现。

钱鑫：负责硬件连接与数据库推送通知的代码编写，数据文件整理以及相关论文部分编写。

交流情况

在本学期的项目开展中，小组同学交流高效密切，团队配合与分工合理。每个人选择并完成自己部分的代码，然后上转到 https://github.com/buzhibujuelb/environment_check 处。同时方便项目合作的交流。小组成员建立了 QQ 群以及随时进行线下集会，共同交流目前出现的问题以及解决方法。同时使用了 IPV6 和内网穿透技术，现在在校园网下可以直连，在公网可以内网穿透连接树莓派，这样每个人可以直接使用自己的电脑完成代码调试，提高了项目的完成效率

参考文献

[1] https://blog.csdn.net/qq_32526087/article/details/119749810

[2] https://ruanyifeng.com/blog/2016/03/systemd-tutorial-commands.html

[3] https://flask.palletsprojects.com/en/2.3.x/

[4] https://www.psycopg.org/docs/

[5] https://www.postgresqltutorial.com/

[6] https://sc.ftqq.com/?c=wechat&a=bind

[7] https://www.tensorflow.org/js/tutorials

[8] https://cn.vuejs.org/

致谢

本报告的工作是在指导教师易黎老师的悉心指导下完成的，我们在此感谢老师的指导和帮助。