

第八届

全国大学生集成电路创新创业大赛

报告类型*: 作品设计报告

参赛杯赛*: 飞腾杯

作品名称*: 智慧之眼：视障人士的户外安全出行伴侣

队伍编号*: CICC1085

团队名称*: 飞腾风驰队

目 录

1 系统概述	1
1.1 项目背景与功能介绍.....	1
1.1.1 项目背景.....	1
1.1.2 功能介绍.....	1
1.2 系统体系结构设计.....	2
1.2.1 子系统划分与层次划分.....	3
1.2.2 结构选择.....	5
1.2.3 模块划分.....	5
1.3 开发板硬件连接与技术规格.....	7
1.3.1 硬件连接关系.....	7
1.3.2 技术规格.....	8
2 各系统程序架构	10
2.1 系统多任务布局与核间通信分配.....	10
2.1.1 Master 端多任务架构	11
2.1.2 Slave 端多任务架构.....	11
2.2 源代码层次结构.....	12
2.2.1 Master 端源代码层次结构	12
2.2.2 Slave 端源代码层次结构.....	13
3 各模块技术细节	13
3.1 机器视觉处理模块.....	13
3.1.1 适用场景概述.....	14
3.1.2 障碍物识别神经网络：YOLOv8n	20
3.1.3 角度与距离计算方法.....	22
3.1.4 监护人 GUI 端小地图显示功能.....	24

3.1.5 人行楼梯级数识别算法.....	25
3.1.6 斑马线方向识别算法.....	32
3.1.7 交通灯识别算法.....	35
3.1.8 将所有检测结果打包为字符串，传输至 Slave 端.....	37
3.2 系统状态与感知模块.....	39
3.2.1 GPS 北斗双模定位模块数据解析	39
3.2.2 人行道方向判断功能.....	44
3.2.3 陀螺仪传感器数据解析.....	47
3.2.4 摔倒与视角异常检测.....	50
3.2.5 温湿度环境信息数据解析.....	51
3.3 监护客户端显示模块.....	52
3.3.1 监护客户端 GUI 界面设计	53
3.3.2 监护客户端 GUI 功能详述	57
3.3.3 监护客户端的远程访问实现.....	59
3.3.4 系统性能监控测量环节及显示.....	60
3.4 传感器处理与语音反馈模块.....	62
3.4.1 TTS 语音合成模块通信协议	62
3.4.2 面向视障者的实时语音输出交互逻辑.....	64
3.5 OpenAMP 跨系统核间通信模块	68
3.5.1 OpenAMP rpmsg 核间通信机制工作原理概述	69
3.5.2 Master 端核间通信方式	74
3.5.3 Slave 端核间通信方式.....	75
4 主要创新点	76
4.1 人行楼梯级数检测算法的迭代.....	76
4.2 基于 Multiprocessing 的多进程并行框架的实现.....	78

4.3 Master 端 Linux 内核的 rpmsg 多静态端点实现多通道并行核间通信	81
4.4 Slave 端非阻塞 platform_poll 的实现.....	82
4.5 自主训练目标检测网络设计与评价.....	83
4.5.1 数据集设计.....	83
4.5.2 模型训练结果.....	84
5 测试与验证	87
5.1 功能运行测试.....	87
5.1.1 测试场景.....	87
5.1.2 人行道模式测试结果.....	88
5.1.3 路口模式测试结果.....	89
5.1.4 人行天桥模式测试结果.....	91
5.1.5 未沿道路方向行走功能测试结果.....	92
5.2 性能验证与分析.....	93
5.2.1 目标检测量化耗时.....	93
5.2.2 楼梯级数识别算法耗时分析.....	95
5.2.3 传感器数据采集耗时.....	97
5.2.4 rpmsg 核间通信用耗时	97
6 总结	99
6.1 2 个系统.....	99
6.2 5 个模块.....	100
6.3 8 个创新点.....	100
7 参考文献	101
附 录	102

1 系统概述

1.1 项目背景与功能介绍

1.1.1 项目背景

在全球范围内，视障人士的数量逐年增加，据世界卫生组织(WHO)于 2020 年的报告显示，目前全球有至少 10 亿人口存在某种形式的视觉障碍，其中约 4300 万人处于全盲状态^[1]。这些视障人士在生活的各个层面，特别是在户外出行方面，面临着极大的挑战。在城市化迅速发展的今天，视障者的出行需求更是日益增长，他们渴望能够像正常视力者一样，自由、安全地行走在城市的每一个角落。然而，现有的解决方案未能完全满足这一需求，存在明显的服务缺口。

传统的视障辅助工具虽多，但各有不足。

- **探路杖：**作为最常见的辅助工具之一，虽然便携且使用简单，但其只能感知到杖尖可触及的局部障碍物，对于头部高度的障碍物、地面的不平等情况无能为力，无法提供主动避障功能。
- **导盲犬：**作为另一种普遍的辅助方式，能够有效引导视障者避开路上的障碍，但导盲犬由于生理限制，如色盲，不能识别红绿灯，这在交通繁忙的城市环境中构成了非常大的安全隐患。此外，导盲犬无法提供人类语言反馈，其指示往往需要视障者经过长时间的训练和适应才能理解。
- **监护人陪伴：**虽然是最直接的帮助方式，但这种依赖性极大地限制了视障者追求独立自主生活的可能性，且不是所有视障者都有条件随时得到家人或朋友的陪同。

1.1.2 功能介绍

针对现有解决方案的局限性，我们团队开发了“智慧之眼”——一种创新的视障人士户外安全出行系统。这一系统通过集成先进的人工智能视觉识别技术，能够全面感知户外环境中的各种障碍物、斑马线走向、交通灯状态及梯级信息等多个维度，实现了一个从未有过的全方位辅助系统。

与传统探路杖和导盲犬相比，智慧之眼的 AI 智能模块可以准确识别并解读交通灯颜色和状态，有效避免由导盲犬色盲问题带来的风险；通过实时感知当前位置与周边道路的信息，能够判别视障人士是否沿道路行驶，以免因为无法判别道路前进方向而进入危险的机动车道中。

同时，它可以通过语音提示直接使视障者得以理解，提供清晰易懂的指导信息，帮助他们做出准确的行走决策，极大地提高了出行的安全性和独立性。

此外，智慧之眼还包含了监护人实时监控功能，这一功能不仅允许视障者独立自由地出行，还能通过一个特定的监控端口让家人或朋友远程了解视障者的实时位置和周边环境状况，从而提供必要时的远程帮助，保障双重安全。这种独立而全面的解决方案，填补了目前视障辅助领域的重大缺口，为视障者提供了一个真正意义上的户外行走“眼睛”，让他们可以更加自信和安全地探索世界。

1.2 系统体系结构设计

作品的完整体系结构如下图所示：

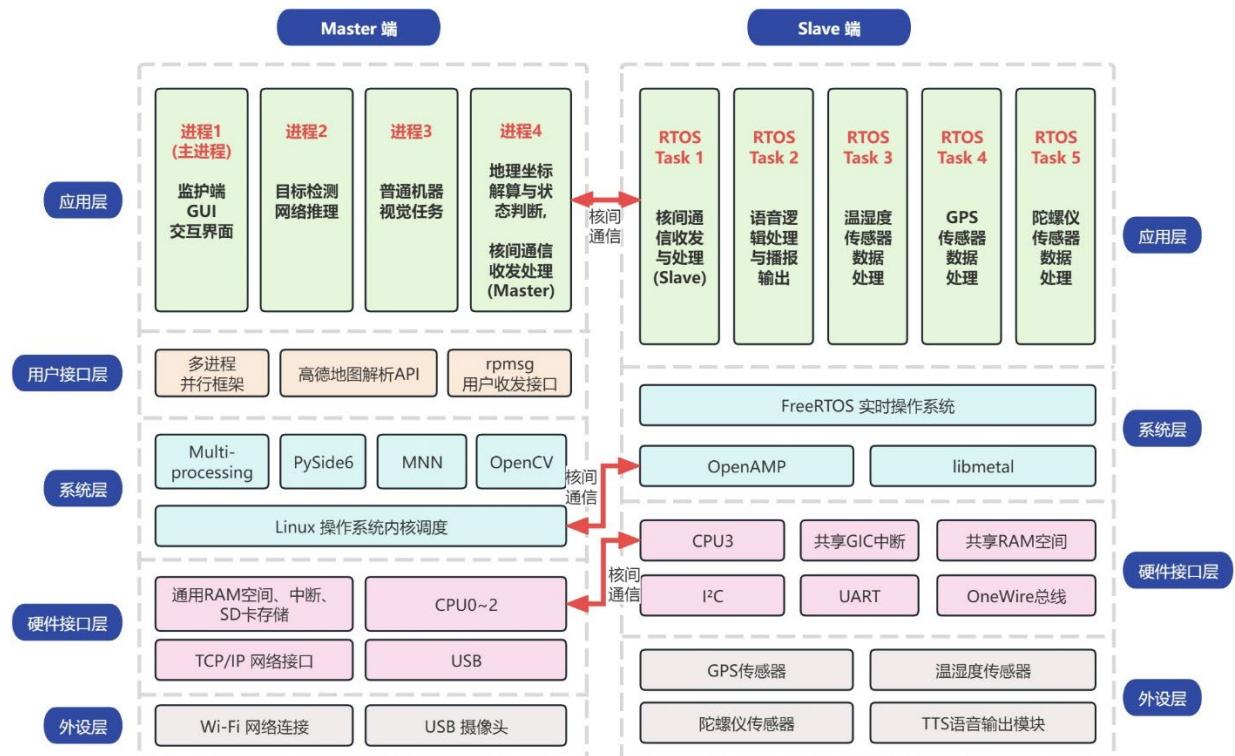


图 1 比赛作品的完整体系结构

1.2.1 子系统划分与层次划分

为了高效地处理复杂的视障辅助任务，我们将整个系统基于 OpenAMP 框架划分为主子系统和 Slave 子系统，这种划分充分利用了硬件资源，优化了任务执行的效率和响应时间。

一、子系统划分

Master 子系统：

系统设计基于飞腾 E2000Q 四核 CPU，其中的 CPU0、CPU1 和 CPU2 三个核心被选用来运行 Ubuntu Linux 操作系统。这一子系统拥有自己的根文件系统和绝大部分的 RAM 空间，为运行高性能应用程序提供了充足的资源。Master 子系统主要负责处理那些对计算性能要求较高的任务，如图形用户界面(GUI)的显示、通过 USB 摄像头进行的目标检测、以及网络推理等机器视觉任务。这些任务虽对性能要求极高，但对实时性的要求相对较低，适合在资源丰富且操作系统功能完善的 Linux 环境下运行。

Slave 子系统：

相比于 Master 子系统，Slave 子系统配置在四核 CPU 中的剩余一个 CPU3 核心上，运行着 Baremetal 环境下的 FreeRTOS 实时操作系统。这一子系统通过 OpenAMP 框架与 Master 子系统共享内存空间和核间中断，确保了资源的高效利用。Slave 子系统主要负责执行对实时性要求高、但对性能要求不是特别高的任务，包括通过 I2C、OneWire 和 UART 低速总线接入的 GPS 定位、温湿度监测、陀螺仪传感器数据的采集、处理和解析，以及 TTS 语音输出模块的控制和语音播报逻辑。这些功能是系统与用户交互的核心部分，需要快速响应，以确保视障人士能实时获取环境信息并作出反应。

两个子系统之间的联系通过 OpenAMP 的 rpmsg（远程消息传递）机制实现，这种机制支持多通道并行通信，非常适合于数据量大且需频繁交换的应用场景。通过这种高效的通信方式，Master 子系统可以快速地将图像识别结果和其他处理信息传送给 Slave 子系统，后者则可以根据这些信息进行快速的决策和反馈，如进行语音提示，并将新采集到的传感器数据信息及时返回到 Master 子系统中^[2,3]。

二、层次划分

外设层：

外设层是系统的基础，包括各种外接传感器（如 GPS、摄像头等），TTS（文字转语音）输出模块以及网络连接模块。这一层的主要功能是实时采集外部环境数据，进行语音交互，并保证系统的联网功能，以便实时接收和发送数据。

硬件接口层：

硬件接口层作为硬件资源与上层软件之间的桥梁，包括主控端（Master）和从控端（Slave）的 CPU、内存及外存，并且管理外设总线接口。这一层确保所有硬件资源的有效分配和调度，为系统层提供底层硬件支持。

系统层：

系统层包括操作系统和中间件组件，分别为 Master 端的 Linux 内核和 Slave 端的 FreeRTOS。此外，此层还包含了各种中间件库，如 ONNX Runtime 用于深度学习模型的推理，OpenCV 用于图像处理，PySide 用于图形用户界面的开发，以及 OpenAMP 和 libmetal 用于多核处理和低级硬件交互。系统层提供了一个稳定的运行时环境，支持复杂的数据处理和系统管理任务。

用户接口层：

用户接口层定义了应用程序与用户交互的通用接口，这包括自行研发的多线程并行框架和 rpmsg 用户 IO 读写接口等。通过这一层，应用程序能够以模块化的方式实现用户交互，提高了系统的交互效率和用户体验^[4]。

应用层：

应用层位于系统架构的最顶层，主要负责实现具体的应用功能。这一层包括 Master 端运行的多个进程和 Slave 端的多个 RTOS 任务，以及这些进程和任务之间的交互机制。应用层通过调用下层提供的各种服务和接口，实现了视障人士户外安全出行伴侣的核心功能，如环境感知、路径规划和动态障碍物避让等。

1.2.2 结构选择

我们采纳了一种结合分模块和分层次的系统结构设计策略。这种设计方法在提供系统功能与服务的同时，优化了开发流程、提升了系统的可维护性和扩展性，并确保了高效的资源管理和更优的用户体验。

分模块的设计优势：

1. 专业性和优化性能：各模块专注核心任务，性能强的子系统处理复杂模块，实时子系统负责传感器和交互，提高硬件和软件优化，确保功能模块适宜运行。
2. 系统稳定性和安全性：模块独立运作，故障局限于特定模块，不干扰整体系统，使用 OpenAMP 进行安全的核间通信。
3. 易于维护与升级：模块独立，维护升级便捷，减少系统架构调整需求，降低复杂度。

分层次的设计优势：

1. 明确的职责分工：逐层设计从外设层到应用层，每层有明确功能，低耦合设计提高系统稳定性。
2. 灵活性和可扩展性：分层架构便于未来升级和功能拓展，减少对底层硬件的依赖，降低成本和风险。
3. 提升开发效率：分层使多个团队成员同时独立工作，提高效率，便于单元和层间测试，确保软件质量。

1.2.3 模块划分

本系统的模块分为以下五个部分：机器视觉处理模块、系统状态与感知模块、监护客户端显示模块、传感器处理与语音交互模块、OpenAMP 跨系统核间通信模块。

其中，机器视觉处理模块、监护客户端显示模块由 Master 子系统实现；系统状态与感知模块、传感器处理与语音交互模块由 Slave 子系统实现；OpenAMP 跨系统核间通信模块由两个子系统共同完成。

具体模块划分的功能描述如下所示：

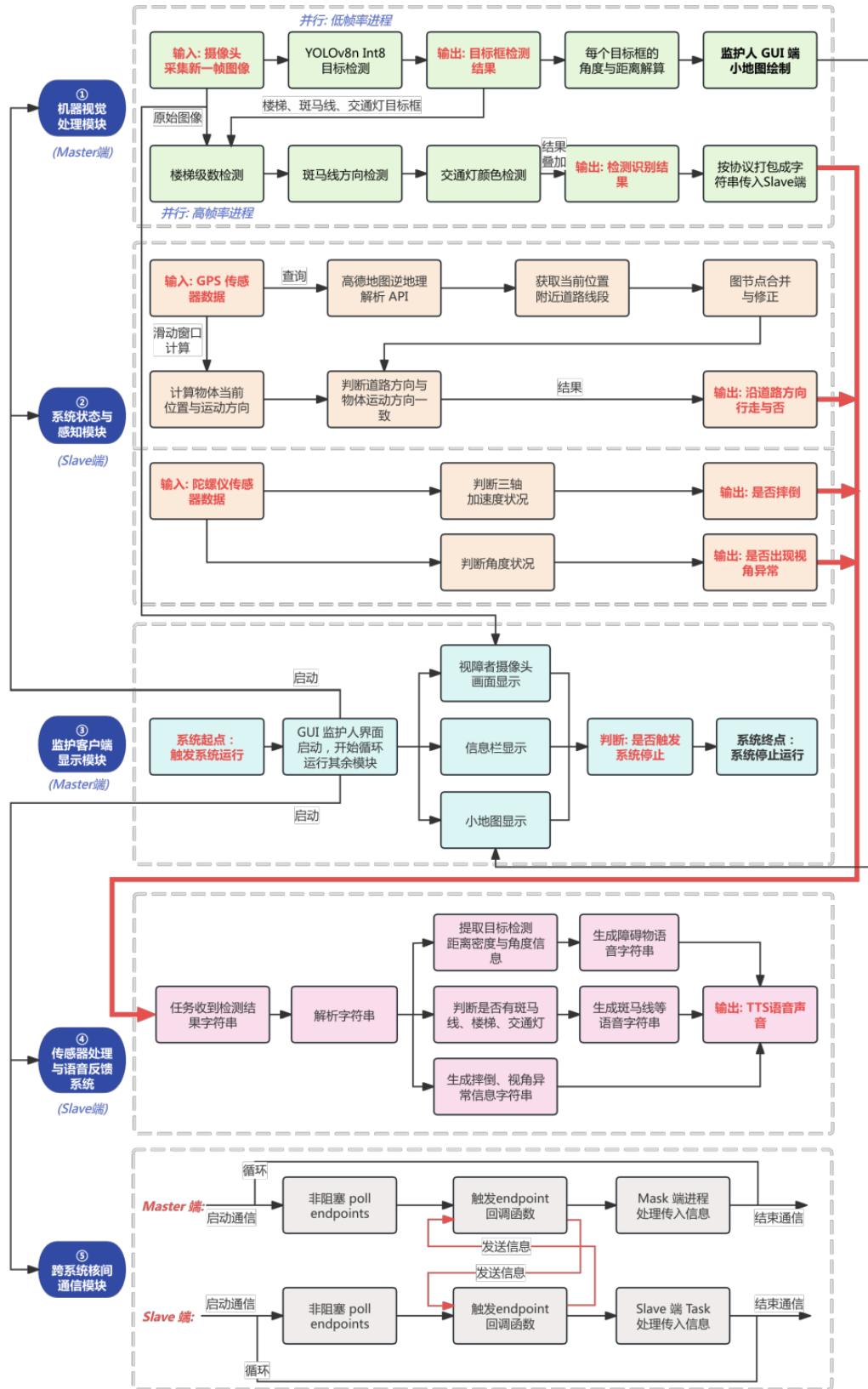


图 2 作品的模块划分

1.3 开发板硬件连接与技术规格

1.3.1 硬件连接关系

默认状况下，开发板硬件连接有 5 个外接模块，分别为：USB 摄像头、温湿度传感器、六轴姿态传感器、GPS 北斗双模定位模块、语音合成模块。其中，USB 摄像头在 USB 接口相连接，而剩余四个传感器由

另外，在调试过程中，我们额外将鼠标键盘、显示器与开发板上的 USB 接口相连接。在电源连接上，我们在实际应用过程中将开发板连接到具有 12V、3A DC 供电能力的外接电池。

总体硬件连接关系如下图所示：

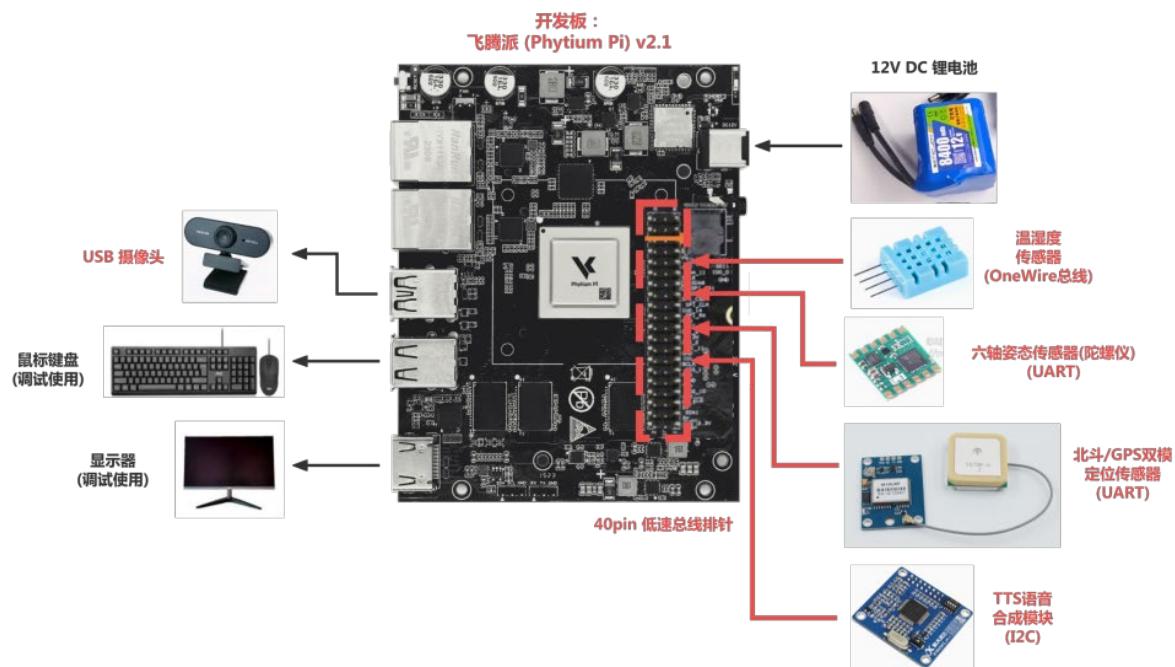


图 3 总体硬件连接关系

其中，开发板上的 40pin 低速总线接口，所连接上的是温湿度传感器、六轴姿态传感器、GPS 北斗双模定位模块、语音合成模块共 4 个模块。这一部分的接线关系如下所示。

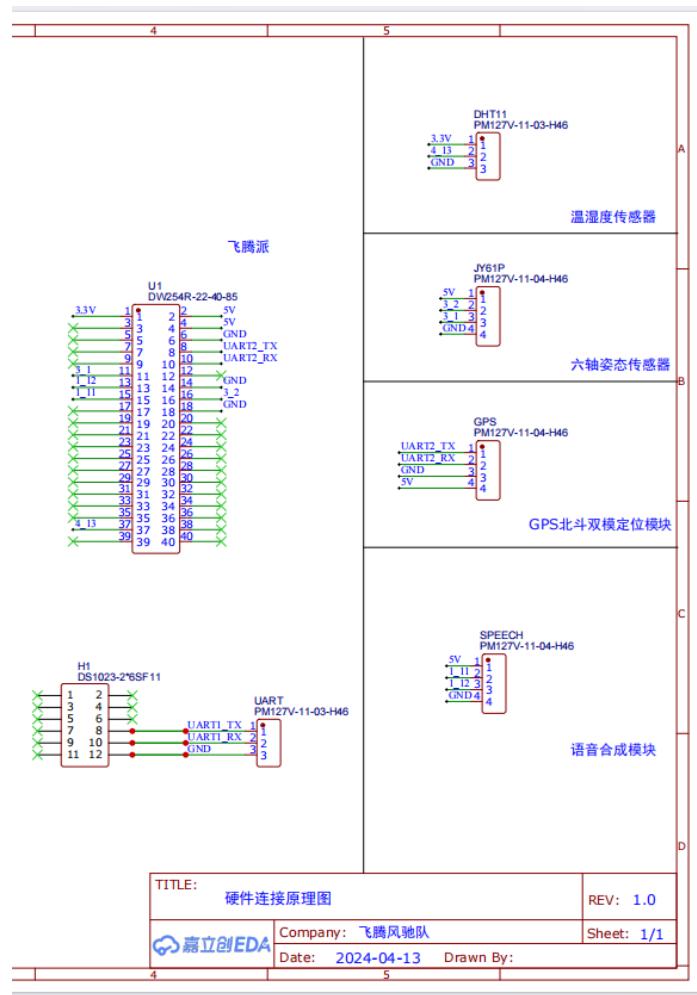


图 4 低速总线接口硬件连接关系接线图

1.3.2 技术规格

系统的技术规格如下所示：

开发平台	飞腾派 (Phytium Pi) v2.1 版本 (萤火工场制造)
板载 CPU	飞腾 E2000Q ARMv8 架构, 2*FTC664@1.8GHz +2*FTC310@1.5GHz
运行内存	4GB
板载显卡、AI 加速器等	无, 本项目不使用任何外加显卡、硬件加速器
Master 端操作系统	内核: Linux 5.10 (修改后) 根文件系统: Ubuntu 20.04 LTS
Slave 端操作系统	FreeRTOS 实时操作系统

OpenAMP 处理器核心分配	CPU0, 1, 2: Master 端 CPU3: Slave 端
Master 端开发环境	Python 3.8 opencv-python == 4.9.0.80 onnxruntime == 1.7.1 numpy == 1.24.4 scipy == 1.10.0 matplotlib == 3.7.5 pyside6 == 6.6.2 ioctl_opt == 1.3.0
Slave 端开发环境	phytium-freertos-sdk (经过修改)
目标检测网络模型规格	YOLOv8n, Float16 量化, MNN 格式 (正式使用版本)
外接 USB 摄像头硬件型号及协议	乐视 LeTMC-520, USB 3.0 (Master 端)
外接温湿度传感器型号及协议	DHT11, OneWire 总线 (Slave 端)
外接六轴姿态传感器型号	JY61P, UART(TTL-232) (Slave 端)
外接北斗 GPS 双模传感器型号	中科微 ATGM336H-5N, UART(TTL-232) (Slave 端)
外接 TTS 语音合成模块型号	科大讯飞 XFS5152CE, I2C (Slave 端)

2 各系统程序架构

2.1 系统多任务布局与核间通信分配

如图 5 所示为系统在 Master 与 Slave 端的多任务与核间通信分配示意图。

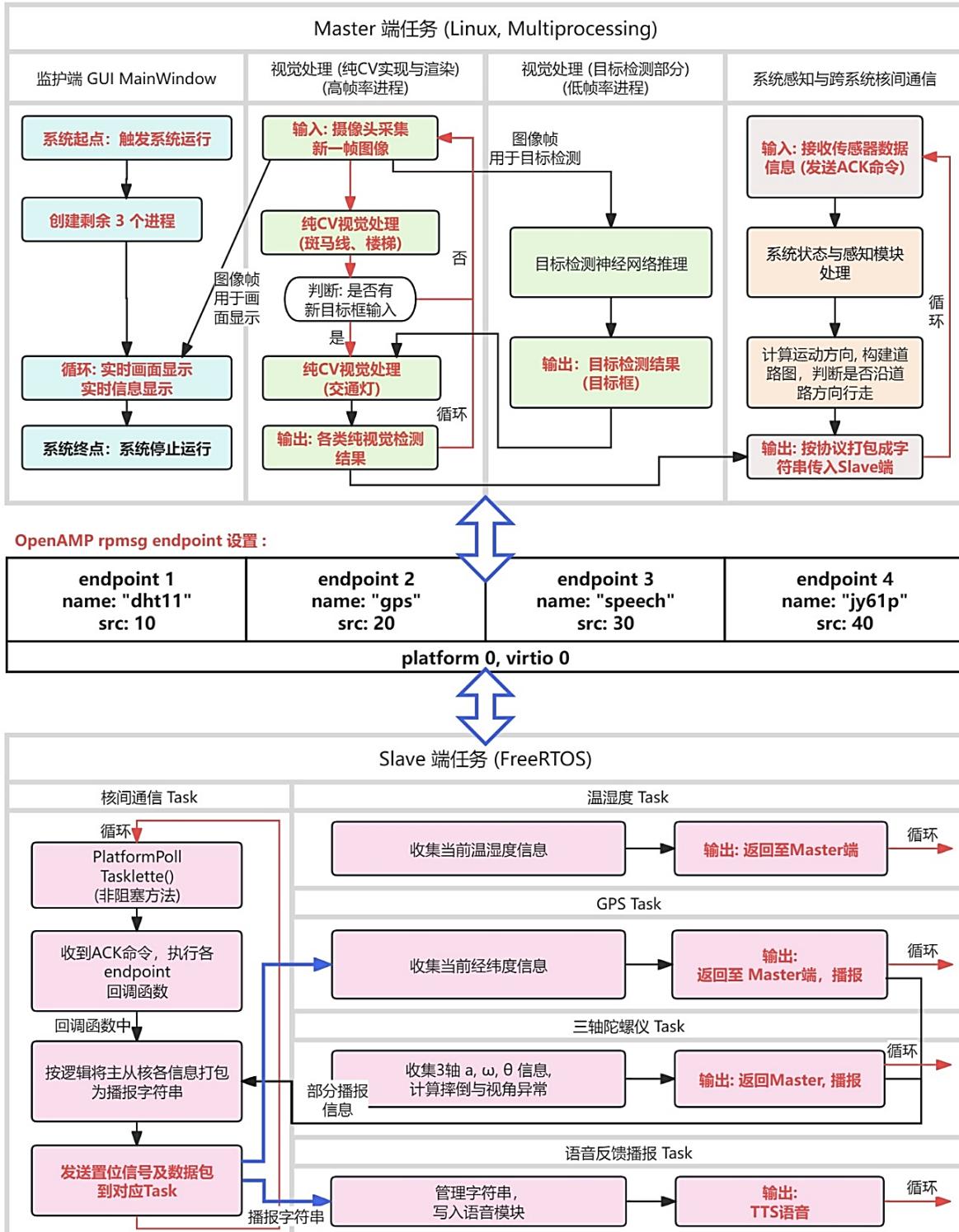


图 5 本项目多任务与核间通信分配示意图

2.1.1 Master 端多任务架构

在 Master 端，使用 QThread 与 Multiprocessing 以实现多任务架构。

监护端 GUI 界面的 MainWindow 为主任务，由 QThread 实现；而剩余三个子任务：纯 CV 实现视觉处理与渲染、目标检测部分视觉处理、系统感知与核间通信，由 Multiprocessing 完成子进程的实现。

具体而言，将三个 Multiprocessing 实现的子进程共同组成类 videoDetection，该类继承类 QObject，由主任务进行生命周期管理。

2.1.2 Slave 端多任务架构

在 Slave 端，使用 FreeRTOS 的多个任务（Task）来实现多任务架构。

1. **初始化 FreeRTOS:** 初始化 FreeRTOS 内核，配置 FreeRTOS 并启动调度器，实现多任务并行处理。
2. **创建不同的 Task:** 使用 FreeRTOS 提供的 API 来创建不同的任务，每个任务都有一个唯一的任务句柄 Task Handle 用于后续操作。使用 xTaskCreate() 函数来创建任务。根据各个模块的功能分为：通信与传感器处理任务 OpenAMPTask，以及负责各传感器的任务：SpeechTask、DHT11Task、GPSTask、Jy61pTask。
3. **任务间通信:** 任务之间通过多种方式进行通信，其中包括队列、信号量、事件组等。如 OpenAMP 通信任务中向各传感器 Task 间发送数据，以及新数据包 flag 置位信号。各传感器 Task 轮询检测到新数据包 flag 被置位后，对所收到的数据包进行处理，实现 Slave 端的数据解析功能与播报功能。
4. **任务调度:** FreeRTOS 将根据任务的优先级和调度算法来决定哪个任务应该被执行。通过设置任务的优先级来控制任务的执行顺序。在 FreeRTOS 中，任务的优先级范围通常是 0 到 configMAX_PRIORITIES - 1，其中 configMAX_PRIORITIES 是 FreeRTOS 中定义的最大优先级数。
5. **任务执行:** 启动 FreeRTOS 的调度器后，任务将开始按照优先级和调度算法执行。在任务函数中，包含实际的任务逻辑和操作，例如传感器数据采集、语音播报数据解析、与主核的通信。

2.2 源代码层次结构

2.2.1 Master 端源代码层次结构

以下呈现的是 Master 端系统正式使用的源代码及其简要解释：

```
/ 根目录
main.py          主程序
home.ui          客户端 UI 文件
resources.qrc    UI 资源配置文件
slaveThread1.py  从核任务调度
UIFunctions.py   主窗口拓展功能
videoThread0.py  主核任务调度
custom_grips.py  主窗口边缘调整功能

/ classes
coordination.py      角度与距离解算、小地图功能
corelogicdef.py       多进程并行框架内核
crossroadsDetector.py 斑马线走向与红绿灯颜色检测功能
gpsdecode.py          解析高德逆地理解析 API 功能
roadDeviationDetector.py 人行道方向判断与行进方向偏移对比功能
test.py               目标检测神经网络推理功能
stairsDetector.py    楼梯级数检测功能
yolov5v8.py           目标检测神经网络推理功能

/ config
fold.json           用于记忆上一次打开的文件夹路径

/ drivers
rpmsg.py            OpenAMP rpmsg 用户收发接口
detectResultPackUp.py 所有视觉检测结果打包为 bytearray
speakout.py          仅 Debug 环节使用，解析视觉检测结果 byte array 并生成播报字符串

/ models
haizhuv8nint8.onnx  YOLOv8n INT8 量化 ONNX 格式 目标检测模型（自行训练）
hzv8nfp16.mnn        YOLOv8n FLOAT16 量化 MNN 格式 目标检测模型（自行训练）

/ img  监护端 GUI 控件图像

/ ui
resource_rc.py       由 resources.qrc 编译得 ui 资源配置文件
test_ui.py           由 home.ui 编译得 ui 文件

/ utils
capnums.py          视频源输入接口
```

2.2.2 Slave 端源代码层次结构

以下呈现的是 Slave 端系统正式使用的源代码及其简要解释：

/ 根目录	
Kconfig	Makefile GUI
main.c	主入口
makefile	主函数, 程序入口
sdkconfig	编译时使用
Makefile	配置记录
/ configs	SDK 配置文件
phytiumpi_aarch64_firefly_gpio.config	修改后的 SDK 资源配置文件
/ inc	头文件
dht11.h	温湿度传感器模块函数
gpio_init.h	模拟 I2C 方法函数
gps.h	GPS 模块函数
jy61p.h	六轴陀螺仪模块（串口）函数
jy61p_i2c.h	六轴陀螺仪模块（I2C）函数
REG.h	六轴陀螺仪模块寄存器定位表
rpmsg-echo.h	OpenAMP rpmsg 核间通信函数与任务实现
speech_i2c.h	语音输出模块（新选型）函数
syn6288.h	语音输出模块（旧选型, 已弃用）函数
uart.h	UART 串口传输方法函数
wit_c_sdk.h	六轴陀螺仪模块数据解析函数
/ src	源文件
dht11.c	温湿度传感器模块函数
gpio_init.c	模拟 I2C 方法函数
gps.c	GPS 模块函数
jy61p.c	六轴陀螺仪模块（串口）函数
jy61p_i2c.c	六轴陀螺仪模块（I2C）函数
rpmsg-echo.c	OpenAMP rpmsg 核间通信函数与任务实现
speech_i2c.c	语音输出模块（新选型）函数
syn6288.c	语音输出模块（旧选型, 已弃用）函数
uart.c	UART 串口传输方法函数
wit_c_sdk.c	六轴陀螺仪模块数据解析函数

3 各模块技术细节

3.1 机器视觉处理模块

机器视觉处理模块的作用在于，为视障人士在户外出行过程中，捕获前方视觉信息。在通过 USB 摄像头采集画面信息后，输入 YOLOv8 目标检测以及基于 OpenCV 方法的、针对户外人行道出行最常见的斑马线方向、交通灯颜色、人行天桥与隧道楼

梯级数检测算法。最后输出检测的识别结果并传入到 Slave 端；将所有识别到的障碍物目标解算角度与距离，并输入到位于 Master 端的监护人 GUI，进行小地图绘制。

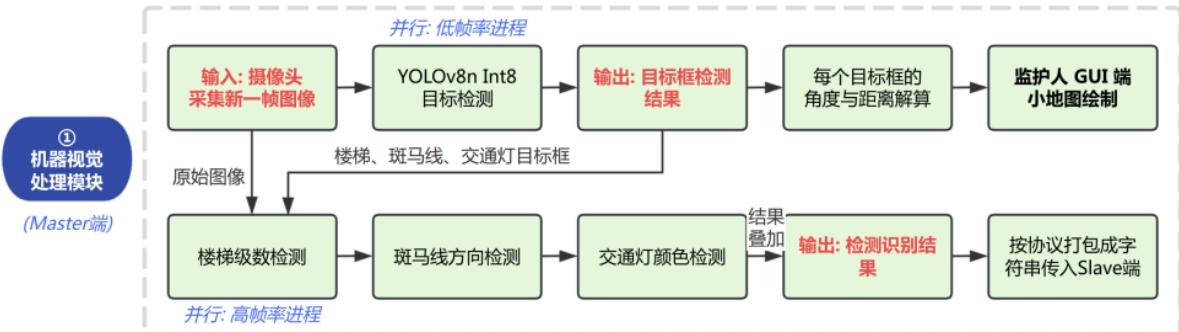


图 6 机器视觉模块总框图

3.1.1 适用场景概述

机器视觉处理模块目前针对下图所示的、视障人士户外出行的核心场景进行适配。

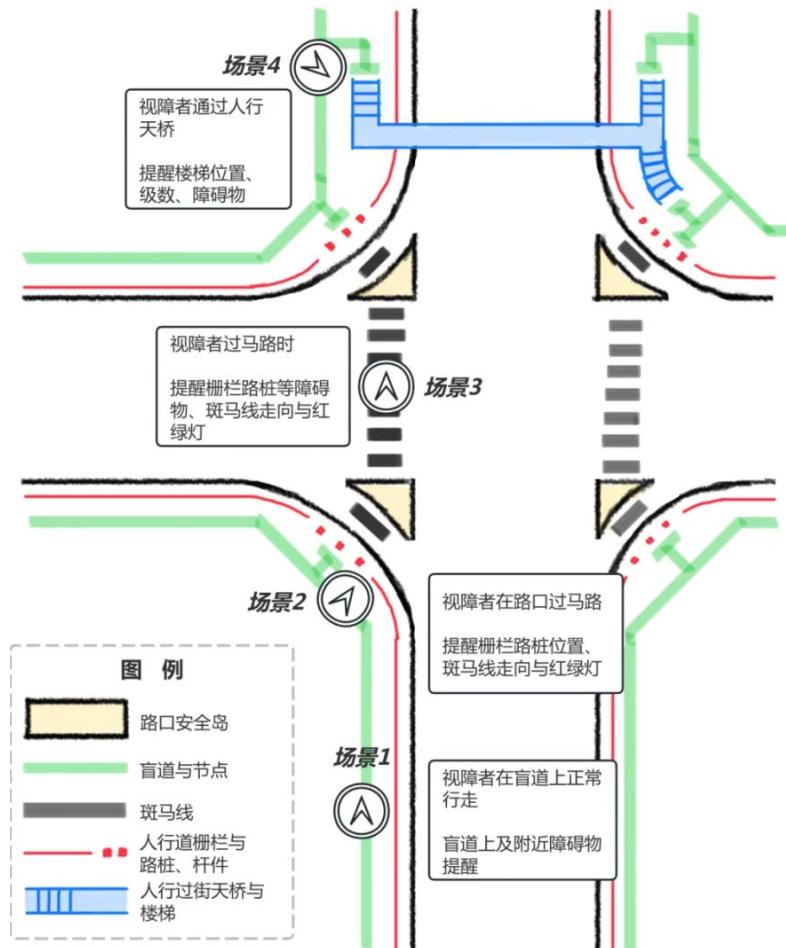


图 7 视障人士户外出行的核心场景

具体场景描述如下所示：

一、场景 1：盲道行走

当视障人士于盲道上行走时，系统通过目标检测识别到使用者摄像头视野内的障碍物信息后，通过语音播报在用户接近障碍物前提醒用户注意避让，避免磕碰。

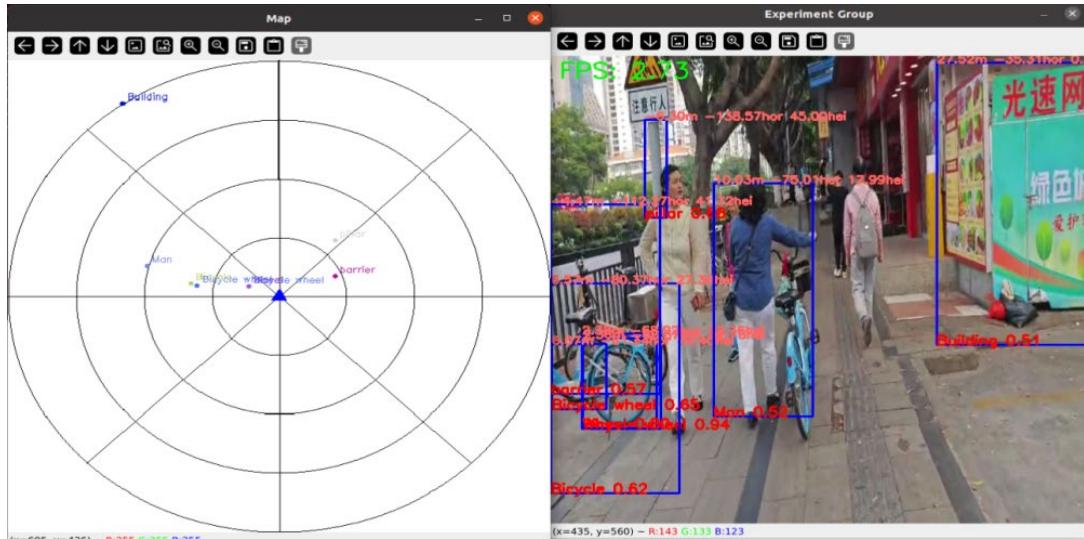


图 8 盲道行走场景 1

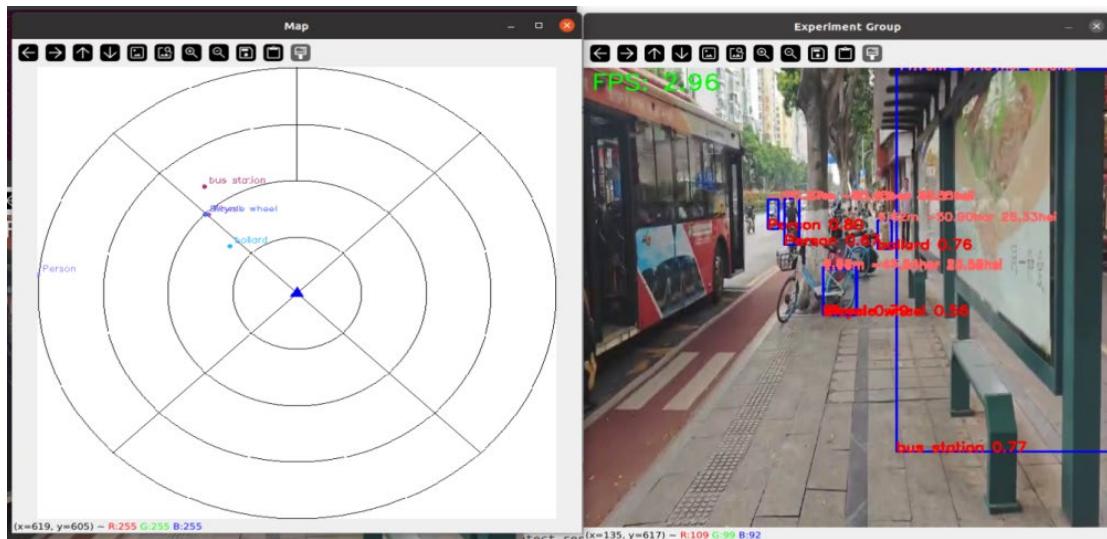


图 9 盲道行走场景 2

二、场景 2：接近十字路口

在行走至十字路口时，用户将要离开盲道走斑马线过街，系统在检测到斑马线后对斑马线走向进行判断并提醒用户；考虑人行横道起点和终点均设有较多的石柱、指示牌铁杆等易于造成磕碰受伤，系统使用的目地检测网络将在检测到这些障碍物后，

进行方位角和距离判断，并将结果反馈至客户端，通过小地图显示并提醒用户；另外，考虑十字路口红绿灯大多数无声提示，当检测网络检测到红绿灯时，系统将对最近的红绿灯进行颜色判断，当指示灯为绿灯时发出语音提醒用户可以过马路。

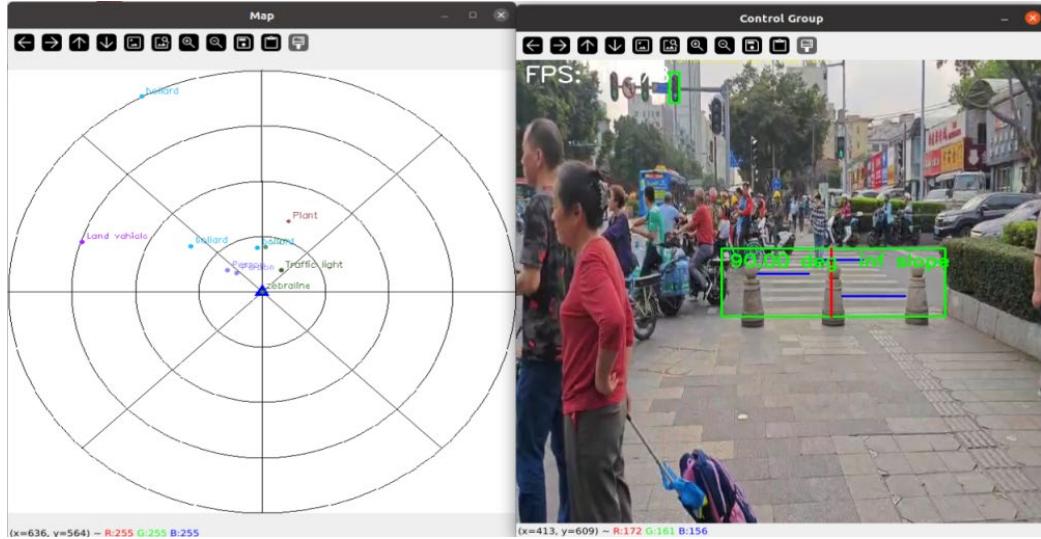


图 10 接近十字路口场景 1

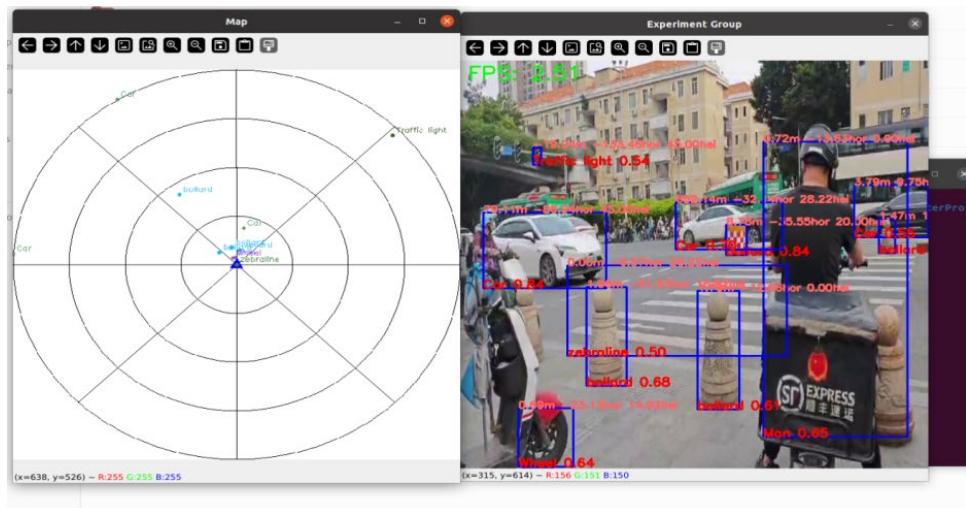
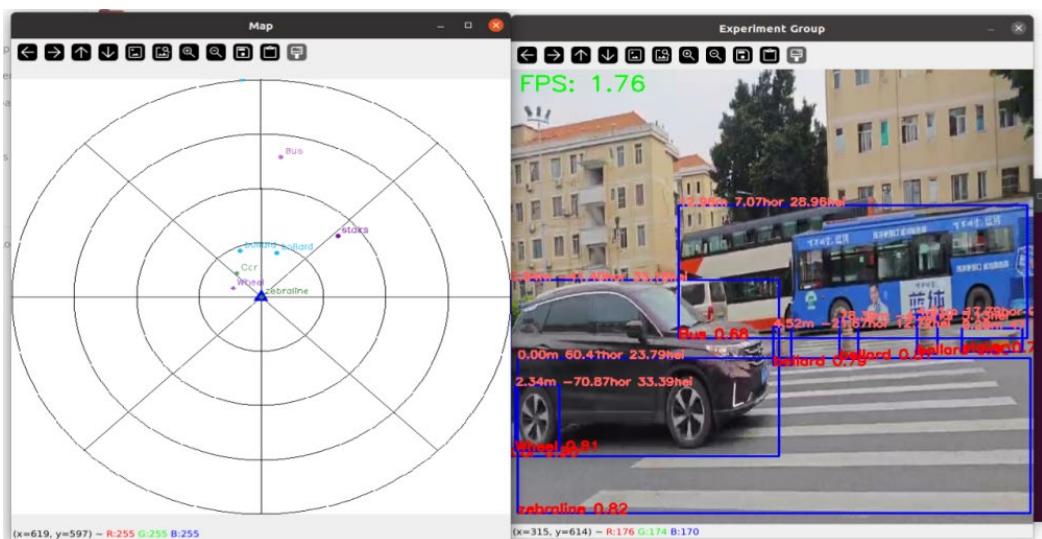
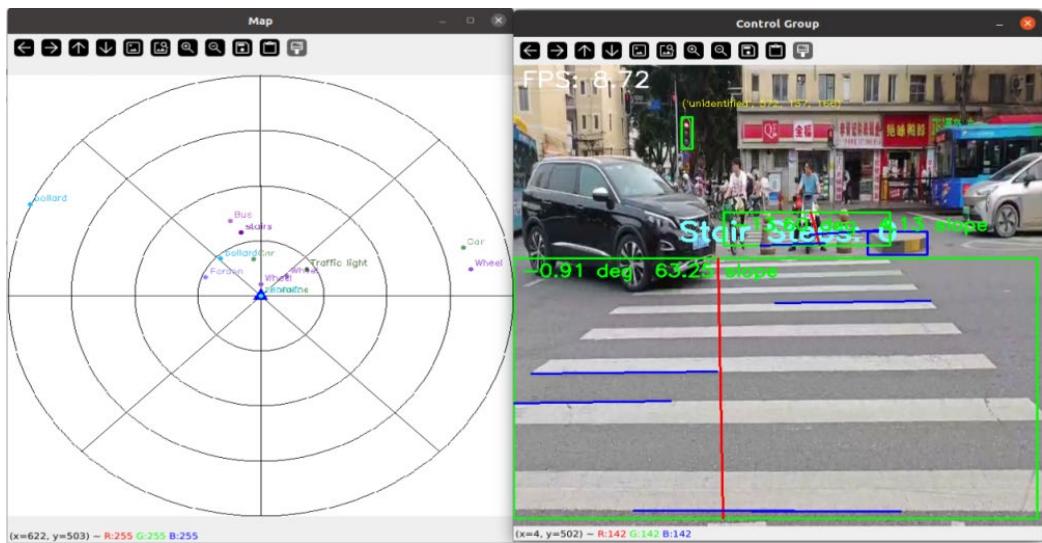


图 11 接近十字路口场景 2

三、场景 3：人行横道过街

在过马路时，由于地面平坦，视障人士无法触知道路走向，系统将实时反馈视野内的斑马线延伸方向提醒用户。在步行至马路中央需要等待下一个红绿灯的位置时，系统将反馈马路对面红绿灯颜色，提醒用户是否可以继续向前。另外通过检测等待区域的石柱及绿化坛边缘台阶以提醒使用者避免摔倒。



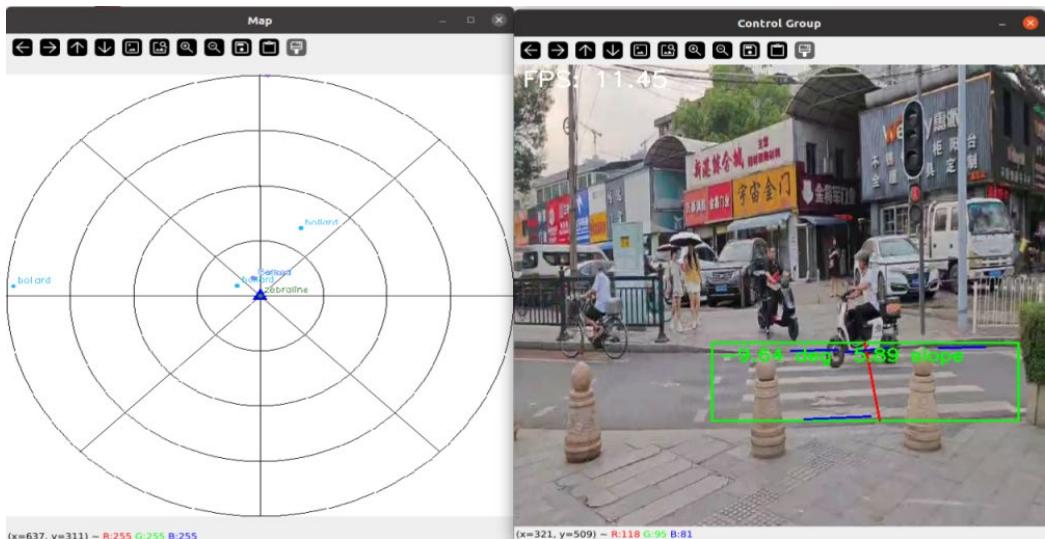


图 12~15 人行横道过街场景

四、场景 4：人行天桥过街

行走过程中，系统的网络检测到人行天桥时将反馈给使用者，若用户需通过人行天桥过马路，可在系统的协助下步行至人行天桥。当视野中出现人行天桥台阶时，系统将进行楼梯阶数检测并进行反馈，沿人行天桥行走的过程中，系统将结合对障碍物的检测和反馈，帮助视障人士安全行走。

由于人行隧道与人行天桥对于楼梯检测识别的原理相似，因此合并两者，命名为“人行天桥过街”场景。



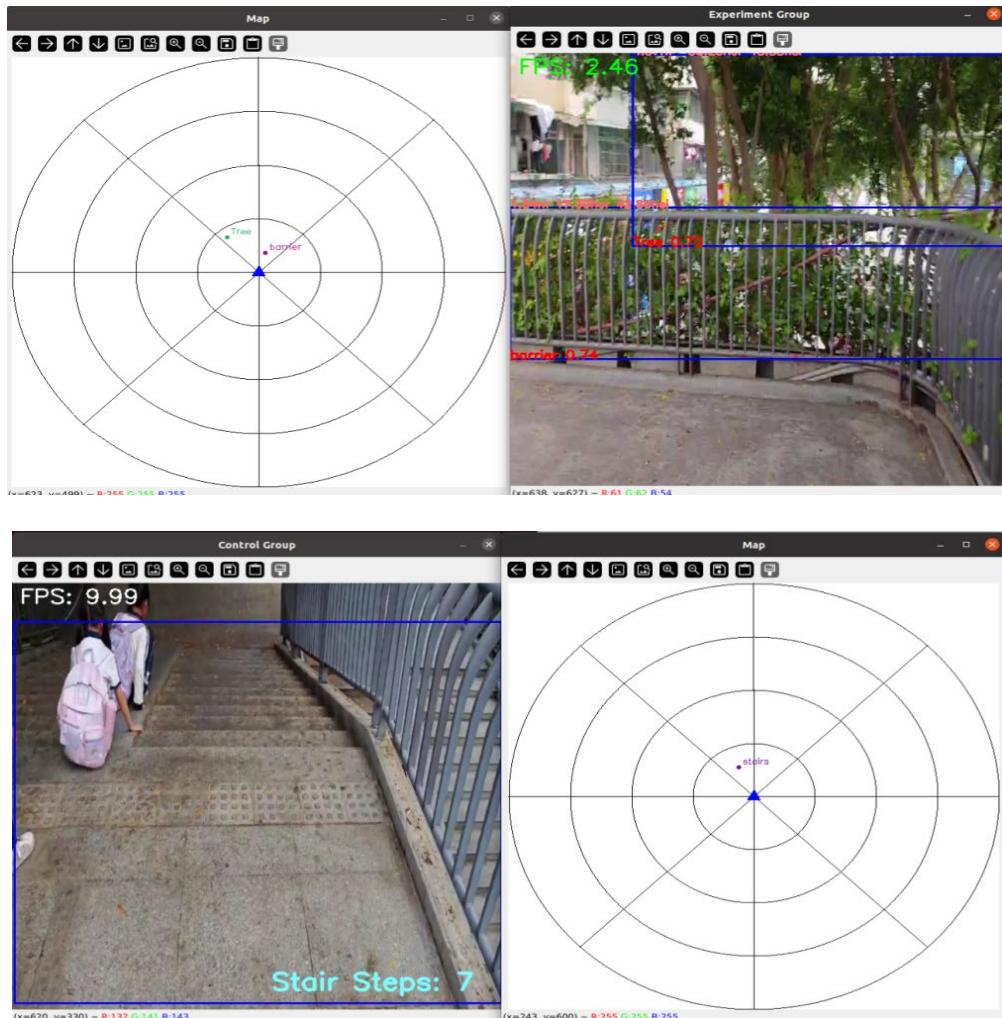


图 16~18 人行天桥过街场景

在理想情况下，本系统应当在硬件层面上设计为头戴式设备，其中摄像头的视角与视障人士眼睛所处的视角一致，便于模拟正常人的视觉效果。而本作品由于所使用的开发板的体积和模块布局的限制，原型机暂且设置为简易手持设备。理想状况下的设备设计示意如下图所示：



图 19 (1~2) 理想状况下的“智慧之眼”设备设计（想象示意图）

实际搭建的原型机设备如下所示：

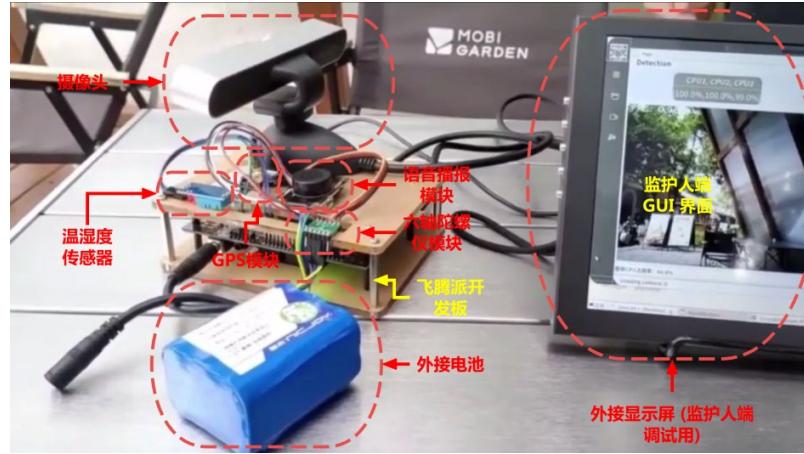


图 20 实际的原型机设备

3.1.2 障碍物识别神经网络：YOLOv8n

一、YOLOv8n 简述

YOLOv8n 的网络结构简述

YOLOv8n，是新一代的 YOLO（You Only Look Once）实时目标检测系统的变体之一，专为在硬件资源受限的环境中运行而设计。该网络通过优化卷积层、激活函数和批量归一化层的结构，实现了对模型大小和计算需求的显著减少，而不牺牲检测准确性。YOLOv8n 采用轻量级的神经网络架构，包括多尺度特征提取器和精简的卷积层，使其非常适合嵌入式系统和边缘计算设备^[5]。具体而言，YOLOv8 系列的网络结构如下图所示。

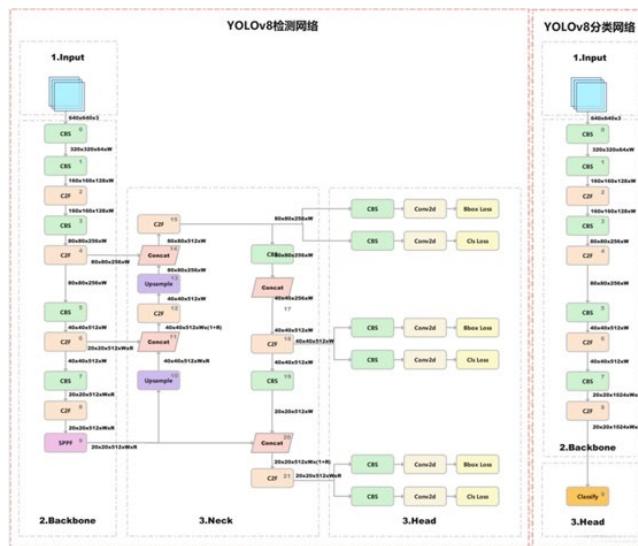


图 21 YOLOv8 系列目标检测网络的基本网络结构

YOLOv8n 的选用优势

YOLOv8n 的最大优势在于其轻量化设计，使得模型即便在计算能力较低的设备上也能保持高速的推理性能。特别是对于飞腾派这样的轻量化嵌入式硬件，YOLOv8n 能够提供快速且高效的图像处理能力。这种设计允许设备在不依赖云计算或外部数据处理中心的情况下，进行高效的实时目标检测，极大地扩展了其应用场景，包括但不限于移动机器人、无人驾驶车辆和智能视频监控等。

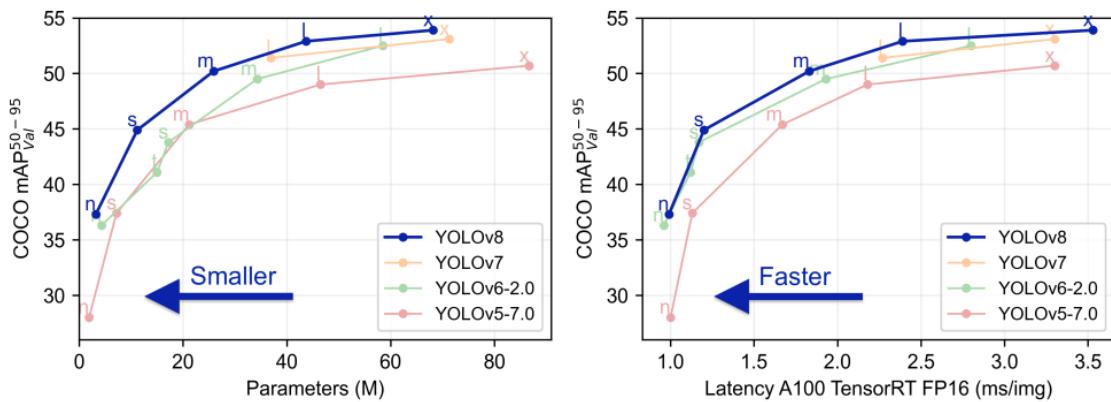


图 22 YOLOv8 与不同代 YOLO 算法相比，在更轻量化、更快速的同时，达到更好的推理性能

二、YOLOv8n 神经网络的不同量化方式与不同框架下的移植

我们通过神经网络方法实现，采用常用的 YOLOv8n 网络结构，具有轻量化与推理效果并重的特征。为了适应飞腾派硬件的性能限制，我们对 YOLOv8n 网络结构进行优化。在量化方式上，我们将模型从 FP16 转换为 INT8，进一步提升推理速度；同时，我们将模型移植到 ONNX Runtime 与 MNN 两种推理框架中，以求在飞腾派具有硬件性能限制的纯 CPU 推理环境中实现硬件加速。

三、选用 YOLOv8n 网络实现目标检测

在本系统中，目标检测算法的作用是读取从进程 3 中，输出到共享队列的摄像头画面帧，并在画面中识别在户外行走的视障者眼前的障碍物信息，以目标种类 (class) 与边界框在视图中的位置的输出结果呈现。其中，摄像头反映了视障者正前方的画面信息，类似人眼观察到的效果。

在本系统中目标检测的分类上，我们确定了总共 50 个 class，其中高频常用的 class 分类如下所示（以 class 名称英文字母顺序排列）：

ID 编号	class 名称	解释	ID 编号	class 名称	解释
14	Barrier	人行道栅栏	3	Motorcycle	摩托车、电动摩托车
28	Bench	长椅	2, 16, 18	Person, Man, Woman	行人
4	Bicycle	自行车	7	Pillar	各类立柱（包括但不限于交通灯、监控摄像头、路灯等）
8	Bollard	常见于行人安全岛的短柱、短桩	9	Plant	绿化隔离带
20	Bus	公交车	10, 30	Stairs	楼梯，或人行道路缘台阶
26	Bus station	公交车站亭	31	Traffic light	交通灯
21	Cabinet	人行道常见的变电箱或通信设备箱	11	Tree	行道树
0	Car	小轿车	24	Truck	卡车
34	Chair	椅子	29	Van	面包车
23	Fire hydrant	消防栓	1	Wheel	各种车辆的车轮
22	Land vehicle	三轮车、运送快递的电瓶车	6	Zebraline	斑马线

3.1.3 角度与距离计算方法

在进程 2 中，每轮输出一帧画面中的所有边界框后，根据每个边界框的种类与位置信息，获取到物体相对于摄像机的详细空间位置信息，包括物体与摄像头的距离、水平偏角和垂直偏角^[6,7]。以下是对每个计算步骤的详细解释：

边界框尺寸提取: 从边界框 (box) 中提取像素坐标 top, left, right, bottom 来计算像素宽度 (pixwidth) 和高度 (pixheight)。

实际尺寸推断: 根据目标的 class 类型, 确定实际宽度或高度 (actualwidth 或 actualheight)。考虑到不同物体宽度或高度的普遍分布, 我们假设其中的一些 class 目标框的高度与实际物体高度值的经验值一致; 另一些 class 的目标框宽度与实际宽度值经验值一致, 以反映不同视角下不同物体的目标框的视角长度 (宽度) 发生改变时, 对应的视角宽度 (长度) 不变的情形。例如, 我们设置 "Car" 类型每个目标框的高度, 均对应实际小轿车的高度经验值 1.4m, 而 "Car" 在视角中水平旋转时, 虽然边界框的长宽比例发生变化, 但总能保证小轿车的高度恒定不变, 而小轿车在视角中投影的宽度随边界框长宽比例的变化而发生变化。这样也保证了计算不同边界框对应的物体距离摄像头的距离尽可能地准确。

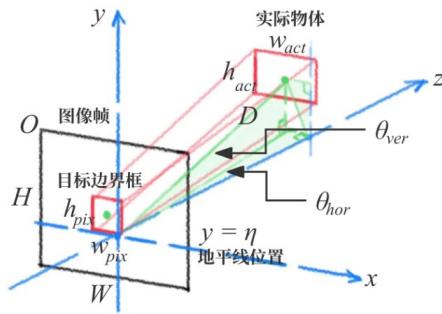


图 23 目标距离角度解析示意图

距离计算:

当识别到的目标种类取高度为恒定不变的经验值时, h_{act} 是物体的实际高度(m), h_{pix} 是边界框的像素高度, 满足以下计算公式:

$$D = \frac{h_{act}}{h_{pix}} \times \frac{f}{\cos(\text{abs}(\theta_{hor}) \times \cos(\text{abs}(\theta_{ver})))} \quad (1)$$

而当识别到的目标种类取宽度为恒定不变的经验值时, w_{act} 是物体的实际宽度(m), w_{pix} 是边界框的像素宽度, 满足以下计算公式:

$$D = \frac{w_{act}}{w_{pix}} \times \frac{f}{\cos(\text{abs}(\theta_{hor}) \times \cos(\text{abs}(\theta_{ver})))} \quad (2)$$

其中 D 是计算出的物体距离摄像头的距离， f 是焦距系数， θ_{hor} 是物体相对于摄像头正前方的水平角， θ_{ver} 是物体相对于摄像头水平线的仰角。

水平角度计算：

$$\theta_{hor} = \left(\frac{\Delta_{pix}}{W/2} \times \theta_{hormax} + \theta_{horbias} \right) \times k_{\theta_{hormax}} \quad (3)$$

其中 Δ_{pix} 是边界框中心相对图像中心的像素偏移量， W 是图像帧的水平宽度（典型值为 640）， θ_{hormax} 为视角范围内，图像左右边缘对应的相对于摄像头正前方的水平角最大幅值（恒为正值）， $\theta_{horbias}$ 为水平角根据摄像头实际情形的校正经验增量因子， $k_{\theta_{hormax}}$ 为水平角根据摄像头实际情形的校正经验乘积因子^[8]。

仰角计算：

$$\theta_{ver} = \theta_{vermax} \times \left(\frac{H \times \eta - y_{center}}{H} \right) \times k_{\theta_{ver}} \quad (4)$$

其中 θ_{vermax} 为视角范围内，图像上边缘对应的相对于摄像头水平线的仰角最大幅值（恒为正值）， H 为图像帧的垂直高度（典型值为 640）， η 是正常水平视角下地平线在图像 y 坐标的对应值， y_{center} 是边界框垂直中心的像素位置， $k_{\theta_{vermax}}$ 为仰角根据摄像头实际情形的校正经验乘积因子。

3.1.4 监护人 GUI 端小地图显示功能

在后续提及使用的 GUI 监护人客户端界面中，对从摄像头读取的每帧数据的各个物体的边界框得到的物体距离、水平角度，以“小地图”的形式呈现，类似于现实世界中的雷达系统。小地图中，辐射状线可以视为经线，帮助确定物体相对于摄像头视角前方偏左或偏右的角度方向，每个刻度设置为 45°。环状线则类似于纬线，表示距离，每个刻度设置为 2.5m，最大的显示范围距离摄像头 10m。每个物体基于其距离和方向在地图上的位置不同，通过转换极坐标（距离和角度）到笛卡尔坐标（ x, y 坐标）来确定并表示^[9]。

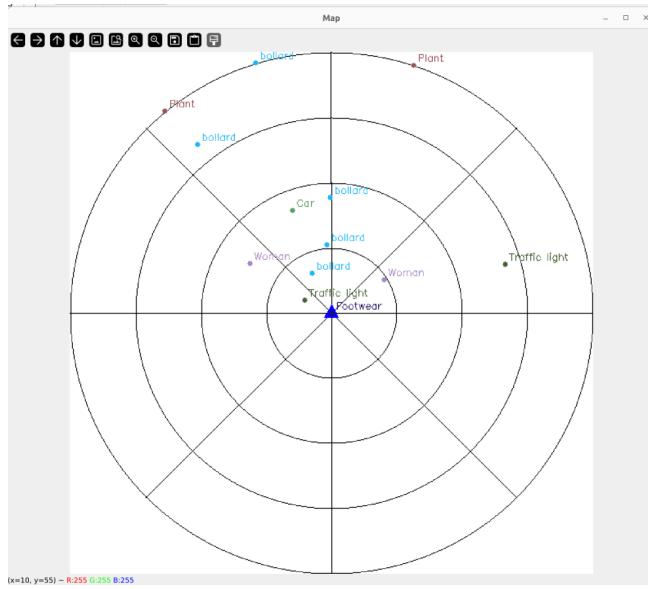


图 24 小地图效果显示

3.1.5 人行楼梯级数识别算法

人行楼梯级数识别算法用于识别人行天桥和人行隧道登上楼梯、走下楼梯时，前方或脚下的楼梯级数。该算法有助于使户外独自出行的视障人士，可以直接获取脚下的楼梯数量，以增加上下楼梯时的安全性与便利性。

目前人行楼梯级数识别算法已经迭代了两个版本，这里展示现行的最新版本。两个版本间的改进与对比将在后续“创新点”环节中展现。

该算法的总体概括如下所示：

算法：楼梯级数识别

输入：一系列带有“**Stairs**”目标框的图像帧

输出：每个目标框内的楼梯阶数

过程：

1：对每一帧图像进行处理：

2： 图像预处理：

3： 灰度转换：将图像转换为灰度图

4： 高斯模糊：对灰度图应用高斯模糊以减少噪声

5： 边缘检测：使用 **Canny** 算法进行边缘检测

6： 直线线段检测：

7： 对“**Stairs**”目标框使用霍夫变换检测直线线段

8： 记录每个线段的两个端点 (x_1, y_1) , (x_2, y_2)

9： 斜率直方图分析与提取：

10： 计算每个线段的斜率 k 和截距 b

```

11: 使用 histogram() 方法对斜率  $k$  进行频率分布分析
12: 筛选出斜率  $k$  接近  $\theta$  的线段作为后续处理的基础
13: 水平截距直方图分析与楼梯结构阶数提取:
14: 对筛选出的线段基于截距  $b$  进行频率分布直方图分析
15: 设置阈值并识别频率分布直方图中的波峰个数
16: 根据波峰个数估计楼梯的阶数
17: 返回每个目标框的楼梯阶数

```

具体代码实现如附录 (a) 所示。

一、图像与处理：边缘检测

灰度转换： 图像首先被转换为灰度图，便于简化后续处理。

高斯模糊： 应用高斯模糊是为了减少图像噪声，提高后续边缘检测的效果。

高斯模糊的原理基于高斯函数，该函数在统计学中用于描述正态分布，其在图像处理中的应用形式表现为二维高斯函数。高斯模糊通过与每个像素及其邻近像素的加权平均实现模糊效果，权重由高斯函数确定。

具体来说，高斯模糊可以表示为：

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \quad (5)$$

其中， $G(x, y)$ 是二维高斯函数， x 和 y 表示距离中心像素的横纵坐标距离， σ 是高斯分布的标准差，控制模糊的幅度。

在本系统的实际应用中，高斯模糊应用一个大小为 $m \times n$ （例如 5×5 ）的卷积核进行图像的卷积处理。卷积核中的每个元素值由上述高斯函数计算得出，所有元素值之和通常归一化为 1。

应用于图像 I 的每个像素 $I(x, y)$ ，经过高斯模糊处理后的像素 $I'(x, y)$ 的值可通过下式计算：

$$I'(x, y) = \sum_{u=-k}^k \sum_{v=-k}^k G(u, v) \cdot I(x+u, y+v) \quad (6)$$

其中， k 是卷积核半径，此处为 2， $G(u, v)$ 是卷积核中位于 u, v 的权重，该权重由高斯分布函数计算得出。

边缘检测：使用 Canny 算法检测图像中的边缘，得到黑白（无灰度）的图像通道层。Canny 算法主要包括以下几个步骤：噪声降低、计算图像梯度、非最大抑制、双阈值处理以及边缘连接。

噪声降低：由于边缘检测算法对噪声非常敏感，因此首步使用高斯滤波器降低图像噪声。高斯滤波器基于二维高斯函数，可表达为：

$$G(x, y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right) \quad (7)$$

其中， x 和 y 分别代表横纵坐标距离中心点的距离， σ 为高斯分布的标准差，控制模糊的程度。通过与图像卷积，使用该函数生成的核减少图像中的高频噪声，为后续处理步骤提供更平滑的图像基础。

计算图像梯度：边缘通常出现在图像灰度级明显变化的地方。为找出这些地方，Canny 算法计算图像中每一点的梯度大小和方向。梯度大小可以用以下公式表示：

$$G = \sqrt{G_x^2 + G_y^2} \quad (8)$$

梯度方向则为：

$$\theta = \arctan\left(\frac{G_y}{G_x}\right) \quad (9)$$

其中， G_x 和 G_y 分别是图像在水平和垂直方向上的一阶导数，通常通过预定义的 Sobel 算子来近似计算。

非最大抑制：此步骤旨在“瘦身”边缘。算法遍历图像中每个像素的梯度大小，去除那些非边缘上的点。具体做法是，如果当前像素的梯度大小不是在梯度方向上局部最大，则该像素点被置为 0，这样只留下了构成边缘的细线。

双阈值处理：为区分真实边缘与噪声，引入了高低两个阈值。强边缘（梯度值高于高阈值的）被立即选为真实边缘，弱边缘（梯度值处于两个阈值之间的）则在与强边缘连接时才被考虑。低于低阈值的点则被忽略。

边缘连接：通过追踪边缘连接，最终确定图像中的边缘。若一个像素位于边缘上，它会被标记，否则将被抑制。

二、目标框下直线线段检测

通过霍夫变换检测图像中各个“Stairs”目标框中的直线线段，输出每个“Stairs”目标框所得到的线条信息（线段在 OpenCV 图像中的两个端点： $(x_1, y_1), (x_2, y_2)$ ）。这一方法的具体环节如下文所述：

霍夫变换的数学基础

每条直线可以用以下两种参数方程来表示：

- a. 斜率-截距形式： $y = mx + b$
- b. 极坐标形式： $\rho = x \cos \theta + y \sin \theta$

其中， ρ 是原点到直线的垂直距离， θ 是该垂线与 x 轴的夹角。在霍夫变换中，我们使用后者（极坐标形式）来表示直线，以便更好地处理垂直线段的情况。

霍夫线变换的算法描述

霍夫变换通过以下步骤来检测图像中的直线：

- a. **输入边缘检测：**首先输入使用 Canny 算法处理图像所获得的图像的边缘映射。
- b. **霍夫空间映射：**将图像空间中的每个点（特别是边缘点）映射到霍夫参数空间 (ρ, θ) 空间）。在此空间中，每个点对应一组可能的直线，这些直线将在对应的 ρ, θ 值上产生交点。
- c. **参数空间和投票机制：**在霍夫变换中，我们构造一个参数空间 (ρ, θ) 的二维数组（也称为累加器），用于记录每一个可能的 (ρ, θ) 对的票数。每个像素点如果是边缘点，就会对所有可能通过该点的直线的 (ρ, θ) 投票。
- d. **投票过程：**
 - i. 对于图像中的每一个边缘点 (x, y) ，遍历所有可能的角度 θ 从 0 到 180 度）。
 - ii. 对于每一个 θ ，根据 $\rho = x \cos \theta + y \sin \theta$ 计算对应的 ρ 值。
 - iii. 在累加器数组中对应的 (ρ, θ) 位置增加票数。

e. x_j 检测直线：投票完成后，直线的检测就转变成了寻找累加器中票数较高的 (ρ, θ) 值。票数高意味着有较多的边缘点共线，这表明存在一条直线。

在实际应用中，由于累加器大部分元素可能为零，可以使用稀疏矩阵技术来存储和处理这些数据，以节省存储空间和计算资源。这涉及到：

- a) 只存储非零票数的位置和值。
- b) 使用高效的数据结构（如哈希表）来实现这一点。

上述边缘检测与线段检测两个步骤，除了本文中提及的人行楼梯级数识别算法外，在斑马线方向识别算法中同样使用，后续的步骤基于这两个步骤所输出的结果，根据不同任务的不同需求，所完成的变换存在差异。

三、斜率 (k) 直方图分析与提取

1. 获取直方图

此步骤首先将所有的线段的端点表示方式，转换为在 OpenCV 图像中的斜率与 $x = 0$ 坐标下 y 轴的偏移截距 (k, b) 的表示方式。接下来，提取所有线段的斜率，通过 `histogram()` 方法，对各线段的斜率 k 值的频率分布进行分析，具体过程如下：

考虑一个斜率的数据集 $\{s_i\}_{i=1}^n$ ，其中 s_i 表示第 i 个斜率值。我们的目标是根据这些数据构建一个频率直方图。首先，定义自动分箱的直方图函数 $H(s, b)$ ，其中 s 是斜率数据集， b 是分箱策略。

对于直方图的边界 $B = \{b_0, b_1, \dots, b_m\}$ ，其中 m 是由自动分箱策略确定的边界数量，每个斜率值 s_i 被归类到相应的边界区间 $\{b_j, b_{j+1}\}$ 中。直方图 H 的每个条目 h_j 表示区间 $\{b_j, b_{j+1}\}$ 中的数据点数量，即：

$$h_j = \#\{s_i \mid b_j \leq s_i < b_{j+1}\} \quad (10)$$

直方图中条的中心位置由下式给出：

$$x_j = \frac{b_j + b_{j+1}}{2} \quad (11)$$

2. 获取波峰

经过大量数据的分析结果，楼梯的轮廓在摄像头的图像帧中，通常体现为斜率相近且近似水平的大量直线分布；而在关于 k 的频率分布直方图上，通常显示为 $k = 0$ 附近的、显著的波峰。

使用波峰检测方法 $P(h)$ 识别直方图中的所有局部最大值，即波峰。设 P 为检测到的波峰索引集合，每个索引对应 h 中的一个波峰。主要波峰的索引 k 通过寻找 h 在 P 中的最大值确定：

$$k = \arg \max_{j \in P} h_j \quad (12)$$

主要波峰的中心位置 x^* 由下式确定：

$$x^* = x_k \quad (13)$$

3. 定义主波峰周围的范围

为了进一步分析主波峰附近的数据，在系统中，我们定义一个围绕 x^* 的范围 $[x^* - \delta, x^* + \delta]$ ，其中 $\delta = 0.25$ 是我们根据实际经验选取的一个小范围值。这个区间用于过滤和分析靠近主波峰的斜率值。

4. 过滤斜率 $k = 0$ 附近的线段

这一步骤通过滤波方法，筛选出 $k = 0$ 附近波峰对应的线段，作为下一步骤的输入。

具体而言，基于上述定义的范围，过滤出直线集合中斜率在主波峰附近的直线。设直线集合为 $\{l_i\}_{i=1}^p$ ，其中每个直线由其斜率 l_i^0 定义。筛选后的直线集合 l'_i 定义为：

$$\{l'_i\} = \{l_i \mid x^* - \delta \leq l_i^0 \leq x^* + \delta\} \quad (14)$$

四、水平截距 (b) 直方图分析与楼梯结构阶数提取

此步骤将上一步骤中所提取得到的 $k = 0$ 附近的波峰对应的线段，根据它们的截距 b 值的频率分布直方图进行分析。根据经验，在设置了一定峰值阈值、波峰宽度阈值、直方图的 bar 个数之下，截取目标框内频率分布直方图中波峰的个数。这些波峰的个数可近似获得目标框内的楼梯阶数数量值。具体方法如下：

1. 计算截距 **b** 频率分布直方图

首先，从多个直线数据中提取截距值形成数据集 $\{b_i\}_{i=1}^n$ ，其中 b_i 表示第 i 个直线的截距。接着，计算这些截距值的频率直方图 $H(b, 60)$ ，其中使用了 60 个固定宽度的分箱，且计算了每个分箱的密度而非频数。根据直方图的分箱边界，计算每个条的中心位置。

2. 曲线拟合与平滑化

在直方图的基础上，曲线拟合与平滑化过程的实现通过数学方法中的样条插值来完成。此过程主要依赖于单变量样条函数 S ，它是一个灵活的数学工具，用于生成一个平滑的函数，该函数通过一组离散数据点进行定义。

具体而言，给定一组数据点，其中 x_i 表示每个直方图分箱的中心位置 c_i 表示对应分箱的计数，样条插值函数 $S(x)$ 可以定义为：

$$S(x) = \sum_{j=1}^m B_j(x) \cdot \beta_j \quad (15)$$

其中， $B_j(x)$ 是基函数，通常选用的是 B 样条基函数，而 β_j 是控制点（或称为系数），其值通过最小化以下目标函数获得，以确保平滑性和数据适配性：

$$\min \left\{ \sum_{i=1}^n [c_i - S(x_i)]^2 + s \int [S''(x)]^2 dx \right\} \quad (16)$$

其中，第一项是数据适配项，确保样条曲线尽可能接近实际数据点；第二项是平滑项，其中 s 是非负的平滑参数，控制曲线的平滑度。在我们的实现中， $s = 0$ 表示完全通过数据点进行插值，不加入额外的平滑。

为了提高曲线的解析度，生成一个新的更细的 x 值集合 x_{smooth} ，范围覆盖 x 的最小值到最大值，并在这个新的定义域上评估样条函数 $S(x)$ ，即：

$$\begin{cases} x_{\text{smooth}} = \text{linspace}(x_{\text{min}}, x_{\text{max}}, 1000) \\ y_{\text{smooth}} = S(x_{\text{smooth}}) \end{cases} \quad (17)$$

这样得到的 y_{smooth} 代表了平滑处理后的曲线，能够减少因原始采样不足造成的误差和噪声影响。

3. 从截距 b 曲线中寻找波峰，根据波峰数量输出楼梯级数的近似值

关于波峰检测，假设 y_{smooth} 为拟合的样条函数 $S(x_{smooth})$ 的结果，使用波峰检测函数 `signal.find_peaks()` 来寻找满足特定间隔 $d = 25$ 和最小高度 $h = 0.005$ 的波峰。最后，根据波峰数量，输出楼梯级数的近似值。

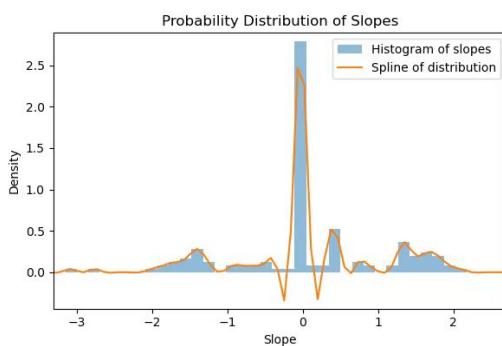


图 25 斜率直方图分析与提取效果

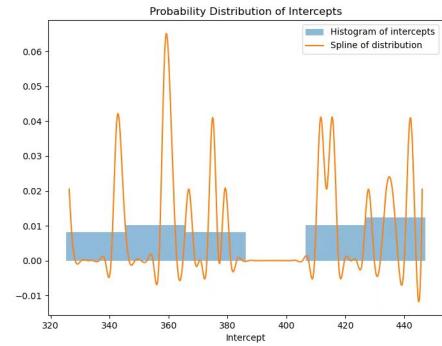


图 26 水平间距直方图分析与提取效果

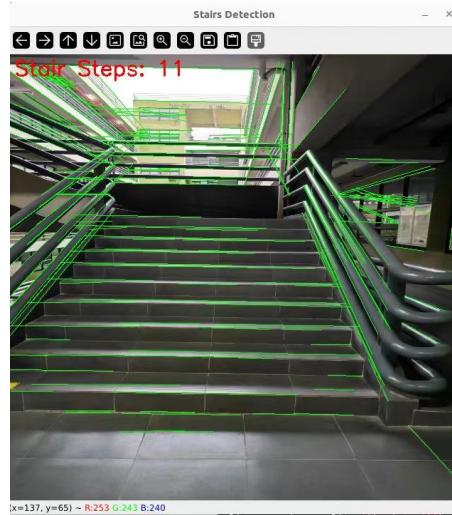


图 27 楼梯结构阶数提取效果（测试界面）

3.1.6 斑马线方向识别算法

斑马线走向方位检测的方法与人行天桥楼梯级数识别的方法相比，图像预处理、直线线段检测两个步骤的方法完全相同。而不同之处在于，由于斑马线的走向方位需要体现行人沿着斑马线行走时的方向，这与斑马线的线条轮廓中较长的线条在摄像头

图像帧的体现，在视角中呈现相垂直的关系。因此，下面阐述斑马线走向方位检测在经过了图像预处理、直线线段检测后的步骤：

算法：斑马线走向方位检测

输入：一系列带有“ZebraLine”边界框的图像帧

输出：每个边界框的斑马线真实走向角度

过程：

1：对每一帧图像进行处理：

2： 图像预处理与直线线段检测（与楼梯级数识别相同）

3： 较长线段的筛选与提取：

4： 计算每个线段的长度 l

5： 构造长度 l 的频率分布直方图，bar 数量设为 8

6： 通过三次样条曲线拟合波峰，选择最大波峰确定斑马线中较长的线段

7： 选取波峰周围的线段作为后续处理的输入

8： 筛除过于密集的线条：

9： 设置密集度阈值（例如 55 像素）

10： 如果两线条间距小于阈值，则仅保留后一条

11： 更新线段集合以减少误差

12： 取垂直线并做修正以得到斑马线真实走向角度：

13： 计算线段平均斜率的平均值，求得垂直斜率

14： 根据摄像头视角调整垂直斜率，缩放变换至地平线位置

15： 计算视角角度差，输出斑马线的真实走向和偏移角度

16： 返回每个目标框的斑马线走向角度

具体代码实现如附录 (b) 所示。

一、较长线段的筛选与提取

首先，在经过图像预处理与直线线段检测两个步骤后，得到“ZebraLine”边界框内所读取的各个线段，以端点格式表示：（线段在 OpenCV 图像中的两个端点： $(x_1, y_1), (x_2, y_2)$ ）。

通过三角函数公式得到每个线段的长度 l ，接着得到关于 l 的频率分布直方图。具体地，为了滤除目标框内其他混杂线段（噪声）的影响，这里将频率分布直方图的 bar 个数设置为 8，再使用三次样条曲线的方法进行拟合。为了得到斑马线线条轮廓中较长的线段，取拟合后三次样条曲线得到的所有波峰中，长度 l 最大的一个。

从该波峰起，低至前一个波谷、高至下一个波谷或最大值的 1 范围内的线段被筛选出来，作为下一步骤的输入。

二、筛除过于密集的线条

我们发现，过于密集的线条会导致斑马线走向实时判断误差的大幅提升。根据经验，我们设置了 55 像素值为线条密集程度的阈值，只要两线条间的距离小于该阈值，那么将仅保留两线段中的后一条。筛除后的新的线段集合将会作为下一步骤的输入。

三、取垂直线并做修正，以得到斑马线真实走向角度

对上一步骤输出的线条集合，取线条平均斜率的平均值。在该平均值的基础上，计算与它垂直的斜线。由于摄像头的视角并非俯视，为了能在摄像头图像帧中显示真实的斑马线走向，以及计算斑马线走向与摄像头正前方的偏移角度，将上述的垂直斜线根据地平线的位置做缩放变换，再根据缩放变换后的视角角度差得到真实的走向和偏移角度，作为最终的输出结果。具体步骤如下：

对于斑马线走向角度的计算，首先对上一步骤输出的线条集合，计算线条平均斜率的平均值：

$$\bar{m} = \frac{1}{n} \sum_{i=1}^n m_i \quad (18)$$

其中， m_i 为每条线的斜率， n 为线条的总数。

接下来，计算与该平均斜率垂直的斜率：

$$m_{\perp} = -\frac{1}{\bar{m}} \quad (19)$$

为了修正垂直斜率以反映真实的斑马线走向角度，需要考虑摄像头的视角。假设摄像头的视角与地平线之间的便宜系数为 θ ，则垂直线在图像中显示的斜率 $m_{\text{corrected}}$ 为：

$$m_{\text{corrected}} = m_{\perp} \cdot \theta \quad (20)$$

最后，根据修正后的斜率 $m_{\text{corrected}}$ ，计算出斑马线真实的走向角度 α ：

$$\alpha = \arctan(m_{\text{corrected}}) \quad (21)$$

3.1.7 交通灯识别算法

由于本系统面向视障者行人步行出行的情景，因此交通灯颜色检测仅分为红灯（“red”）、绿灯（“green”）与非亮灯状态（“unidentified”）三种判断结果。在得到摄像头图像帧与该帧的目标检测推理结果后，执行以下步骤，以实现对交通灯信号颜色检测：

算法：交通灯颜色检测

输入：带有“Traffic light”边界框的图像帧

输出：每个交通灯的颜色状态（红灯、绿灯或非亮灯）

过程：

- 1: 对每个“Traffic light”边界框进行处理：
- 2: 目标框网格分割：
- 3: 划分边界框为 6×8 网格
- 4: 提取每个网格中 R、G、B 通道的最大值
- 5: 检测亮灯情况：
- 6: 初始化红灯单元数和绿灯单元数为 0
- 7: 对每个网格单元进行检测：
 - 8: 如果 R 最大值 > 200 且 G、B 最大值中至少一项 < 145 :
 红灯单元数增加
 - 9: 如果 G 最大值 > 200 且 R 最大值 $< G$ 最大值 - 50:
 绿灯单元数增加
 - 10: 根据红灯和绿灯单元数决定交通灯颜色：
 - 11: 如果红灯单元数 $>$ 绿灯单元数：
 输出红灯（“red”）
 - 12: 否则如果绿灯单元数 $>$ 红灯单元数：
 输出绿灯（“green”）
 - 13: 否则：
 输出非亮灯状态（“unidentified”）
- 14: 返回每个交通灯的颜色状态

具体代码实现如附录 (c) 所示。

一、目标框网格分割

一般而言，在目标检测得到交通灯（“Traffic light”）的边界框内，交通灯的信号通常在黑色背景下呈现较亮的、具有特征颜色值的像素点。为了避免逐个像素点判断所产生的噪声信号干扰与性能问题，对于每个“Traffic light”边界框，划分为 6×8 的网格（x 坐标划分为 6 份，y 坐标划分为 8 份）。对于每个网格单元的所有像素，提取对应的图像帧的 R、G、B 三个通道的最大值，并保存至一个三通道的三维数组中。

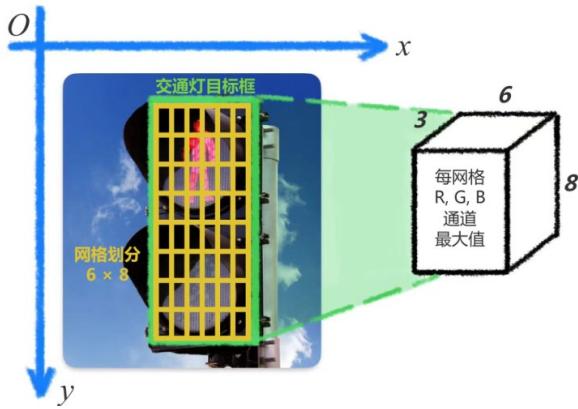


图 28 交通灯目标框网格划分示意图

二、检测亮灯情况

对于 6×8 网格的每个网格单元，记录红灯与绿灯的单元数。具体逻辑如下：

1. 红灯单元：该单元的 R 通道最大值大于 200，并且 G、B 通道最大值至少一项小于 145。
2. 绿灯单元：该单元的 G 通道最大值大于 200，并且 R 通道最大值小于 G 通道最大值减去 50 的差。

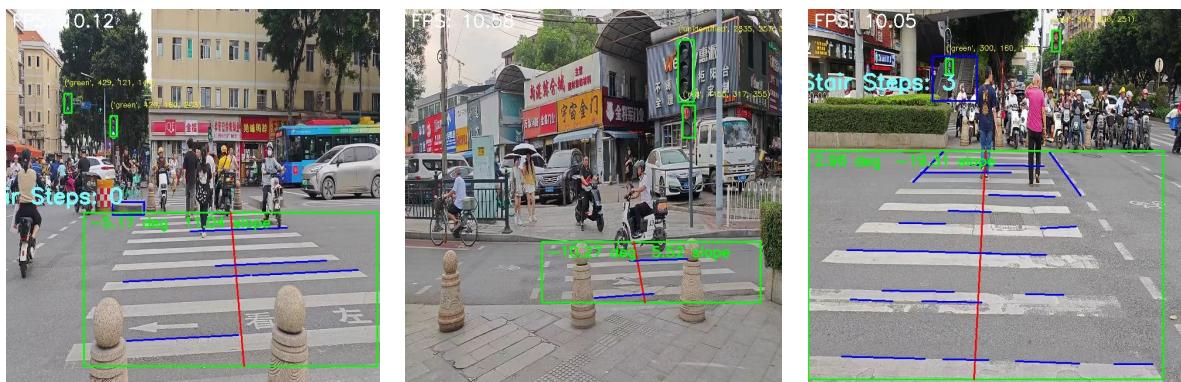


图 29 (1~3) 斑马线与交通灯在不同路口场景下的识别检测效果

若出现红灯或绿灯单元，且红灯单元数量大于绿灯单元数量，则判断该交通灯为红灯（“red”）；而若绿灯单元数量大于红灯单元数量，则判断该交通灯为绿灯（“green”）；否则，若所有网格均没有出现红灯或绿灯单元，则判断为非亮灯状态（“unidentified”）。

3.1.8 将所有检测结果打包为字符串，传输至 Slave 端

由于 Master 端与 Slave 端使用的 rpmsg 通信协议，实际上的传输负载为 bytarray 的形式，且长度有限，因此，为了将所检测的结果以规整、方便的形式，传输至 Slave 端，在完成不同进程下异步的目标检测（进程 2）与其他 CV 算法（进程 3）下，按照以下格式打包为 bytarray。一个 bytarray 没有固定长度，这是因为不同帧的目标检测、斑马线、红绿灯、楼梯段的数量存在差异：

帧头：

[0] uint8 0xAA 字头
[1] uint8 物体数量（除了 zebracline, traffic light, stairs 以及非必要物体外）， max = 78
[2] uint8 斑马线数量（zebracline）， max = 5
[3] uint8 红绿灯数量（traffic light）， max = 10
[4] uint8 楼梯数量（stairs）， max = 5

目标检测段（同一画面不可超过 78 个）：

[0] uint8 0x1A 字头
[1~2] uint16 中心 x
[3~4] uint16 中心 y
[5] uint8 class 类别号
[6] uint8 distance (实际距离*10, 保留 uint8)
[7] int8 verticalangle (垂直角度, -127~127, 保留 uint8 取整)
[8~9] int16 horizontalangle (水平角度, 实际角度*10, 保留 int16 取整)
[10] uint8 0x1B 字尾

斑马线段（同一画面不可超过 5 个）：

[0] uint8 0x2A 字头

[1~2] uint16 中心 x

[3~4] uint16 中心 y

[5~6] int16 deg (实际角度*10, 保留 int16 取整) (注意: 90deg 和 0deg 没有特别区别)

[7] uint8 0x2B 字尾

红绿灯段（同一画面不可超过 10 个）：

[0] uint8 0x3A 字头

[1~2] uint16 中心 x

[3~4] uint16 中心 y

[5] uint8 亮灯状态: undefined:0, red:1, green:2

[6] uint8 0x3B 字尾

楼梯段（同一画面不可超过 5 个）：

[0] uint8 0x4A 字头

[1~2] uint16 中心 x

[3~4] uint16 中心 y

[5] uint8 楼梯级数

[6] uint8 0x4B 字尾

帧尾：

[0] uint8 0xBB 字尾

注意：每帧的长度不能超过 1004 字节，符合 1024 字节的负载限制。

与此同时，上述每帧的传输频率，与进程 2、进程 3 中较慢的一个（目标检测进程，即进程 2）一致。

3.2 系统状态与感知模块

系统状态与感知模块包括两个部分，一是关于 GPS 传感器数据与逆地理解析的应用，包括地理位置获取与判断视障者是否沿道路方向行走功能；二是关于陀螺仪传感器数据的应用，包括对三轴加速度的判断以检测视障者是否摔倒，以及对视角角度的判断检测系统摄像头是否出现视角异常。

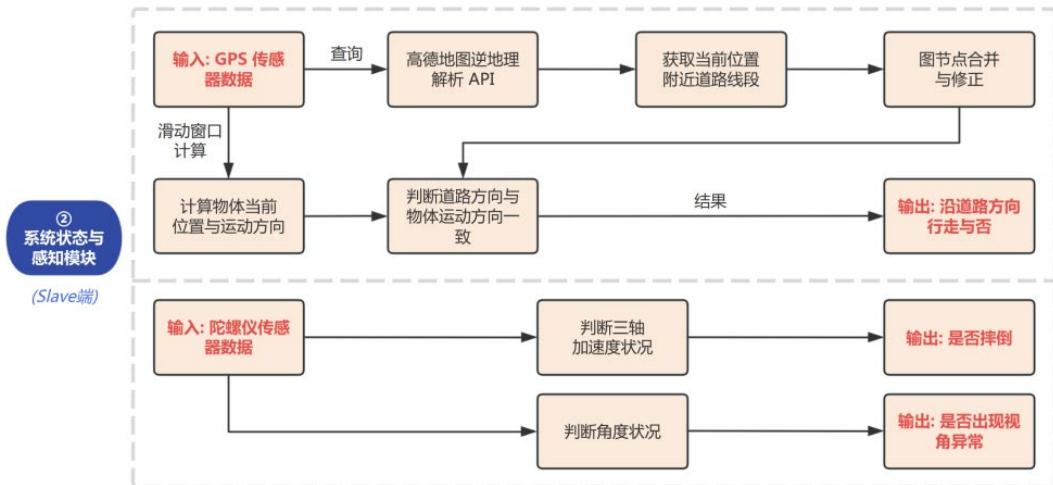


图 30 系统状态与感知模块总框图

3.2.1 GPS 北斗双模定位模块数据解析

本系统采用 GPS 北斗双模定位模块采集使用者的位置信息数据，并将定位信息通过 OpenAMP 传递到主核中。GPS 北斗双模定位模块是一个基于 ATGM336H-5N 的高性能 BDS/GNSS 定位导航模块，能够同时接收多个卫星的信号。在本系统中，GPS 模块使用串口与主控设备飞腾派连接。通过 CASIC 协议实现模块与主机通信。

一、定位模块通信原理

以下以 GPS 信号为例，简要描述该模块如何接收 GPS 定位信息，并转换为数字电路可识别的 CASIC 协议。

1. 射频信号接收转换为基带信号

GPS 信号主要是 L1 载波（1575.42 MHz）上的 C/A 码（Coarse/Acquisition code），它是一种调制在射频载波上的伪随机噪声信号。接收过程可以通过以下步骤和公式描述：

接收: GPS 接收机首先捕捉到由 GPS 卫星传输的射频信号。这些信号包含了卫星的位置、时间信息以及用户的位置信息。

下变频转换: 接收到的射频信号首先经过一个混频器（Mixer）与本地振荡信号混合，进行频率下转换，以降低信号的频率到一个更易于处理的频率范围内。数学上，这可以表示为：

$$s_{IF}(t) = s_{RF}(t) \cdot \cos(2\pi f_{LO}t) \quad (22)$$

其中， $s_{RF}(t)$ 是接收到的射频信号， f_{LO} 是本地振荡器的频率， $s_{IF}(t)$ 是下变频后的中频信号。

滤波与放大: 中频信号通过带通滤波器以滤除不需要的频率分量，然后被放大以准备进一步处理。

基带转换: 中频信号再次被混频处理，完全转换到基带。这涉及到解调过程，通常使用相干解调技术，将载波和调制信号分离出来。这可以通过下式表示：

$$s_{BB}(t) = s_{IF}(t) \cdot \cos(2\pi f_{IF}t) + \hat{s}_{IF}(t) \cdot \sin(2\pi f_{IF}t) \quad (23)$$

其中， $s_{BB}(t)$ 是基带信号， $\hat{s}_{IF}(t)$ 是 $s_{IF}(t)$ 的希尔伯特变换，提供正交分量以支持调制过程。

2. 基带信号的数字化与 CASIC 协议转换

模数转换: 将基带模拟信号转换为数字信号，通过模数转换器（ADC）实现。在这个过程中，模拟信号被采样和量化，转换为数字形式，以便进一步的数字信号处理。

数字信号处理: 数字信号经过进一步的滤波、解调和解码，提取出有效的导航数据，如时间、位置和速度等信息。

数据接口与通信: 最后，提取的数据通过串口通信接口（如 UART）以 CASIC 协议格式发送给主核。CASIC 协议涉及数据的封装，包括数据帧的开始、结束标识符、校验和等，确保数据传输的正确性和完整性。

通过上述两个主要步骤，GPS 定位模块能够将卫星的射频信号转换为可以被主控设备利用的数字信号，从而实现高精度的位置定位功能。

二、定位模块通信协议

主控设备飞腾派在读取数据后，根据 CASIC 协议将封装好的十六进制数据解析为字符串。

名字	类型	描述
NAV	0x01	导航结果：位置、速度、时间
TIM	0x02	定时消息：时间脉冲输出、时间标记结果
RXM	0x03	接收机输出的测量信息（伪距、载波相位等）
ACK	0x05	ACK/NAK 消息：对 CFG 消息的应答消息
CFG	0x06	输入配置消息：配置导航模式、波特率等
MSG	0x08	接收机输出的卫星电文信息
MON	0x0A	监控消息：通信状态、CPU 载荷、堆栈利用等
AID	0x0B	辅助消息：星历、历书和其它 A-GPS 数据

其中，CASIC 协议框架如下图所示：

字段 1	字段 2	字段 3	字段 4	字段 5	字段 6
消息头	有效载荷长度	消息类	消息编号	有效载荷	校验值
0xBA,0xCE	无符号短整型 2个字节	1个字节	1个字节	<2k 字节	无符号整型 4个字节

字段 1：消息头（0xBA, 0xCE）：四个十六进制字符作为消息起始定界字符（消息头），占用两个字节。

字段 2：有效载荷长度（len）：消息长度（两个字节）表示有效载荷（字段 5）占用的字节数，不包括消息头、消息类型、消息编号、长度以及校验和字段。

字段 3：消息类（class）：占一个字节，表示当前消息所属的基本子集。

字段 4：消息编号（id）：消息类后为一个字节的消息编号。

字段 5：有效载荷（payload）：有效载荷是数据包传送的具体内容，其长度（字节数）可变，且为 4 的整数倍。

字段 6：校验值（ckSum）：校验和是从字段 2 到字段 5 之间（包括字段 2 和字段 5）的所有数据的按字（1 个字包括 4 个字节）累加和，占用 4 个字节。

校验值的计算遵循如下算法：

```

ckSum = (id << 24) + (class << 16) + len;
for (i = 0; i < (len / 4); i++)
{
    ckSum = ckSum + payload [i];
}

```

式中，payload 包含了字段 5 的全部信息。在计算过程中，首先将字段 2 到字段 4 的部分进行组装（4 个字节组成一个字），再将字段 5 的数据按 4 个字节一组的顺序（先接收的在低位）进行累加。

NMEA 协议用于从字符串中提取 GPS 定位信息。

NMEA协议框架

检验和的计算范围				
\$	<地址>	{,<数值>}	*<校验和>	<CR><LF>
起始符	地址段	数据段	校验和段	结束序列
每条语句都是以'\$'开始	分为两部分：发送器标识符和语句类型	以'>'开始，后面的数据长度是可变的，也有定长的	对'\$'和'*'之间的数据（不包括这两个字符）按字节进行异或运算的结果，用十六进制数值表示	每条语句都是以<CR><LF>结束

在串口助手中输出实时的位置信息(注：第一行为原始数据，第二至六行为解析数据)：

```

Save_Data.N_S = N
Save_Data.longitude = 113.389227 E
Save_Data.E_W = E
*****
$GNRMC,135755.000,A,2302.57328,N,11323.35362,E,0.00,2.75,110324,,,A*7E
Save_Data.UTCTime = 135755.000
Save_Data.latitude = 23.042888 N
Save_Data.N_S = N
Save_Data.longitude = 113.389227 E
Save_Data.E_W = E
*****
$GNRMC,135756.000,A,2302.57323,N,11323.35362,E,0.00,2.75,110324,,,A*76
Save_Data.UTCTime = 135756.000
Save_Data.latitude = 23.042887 N
Save_Data.N_S = N
Save_Data.longitude = 113.389227 E
Save_Data.E_W = E

```

图 31 串口助手实时位置信息输出（调试环节）

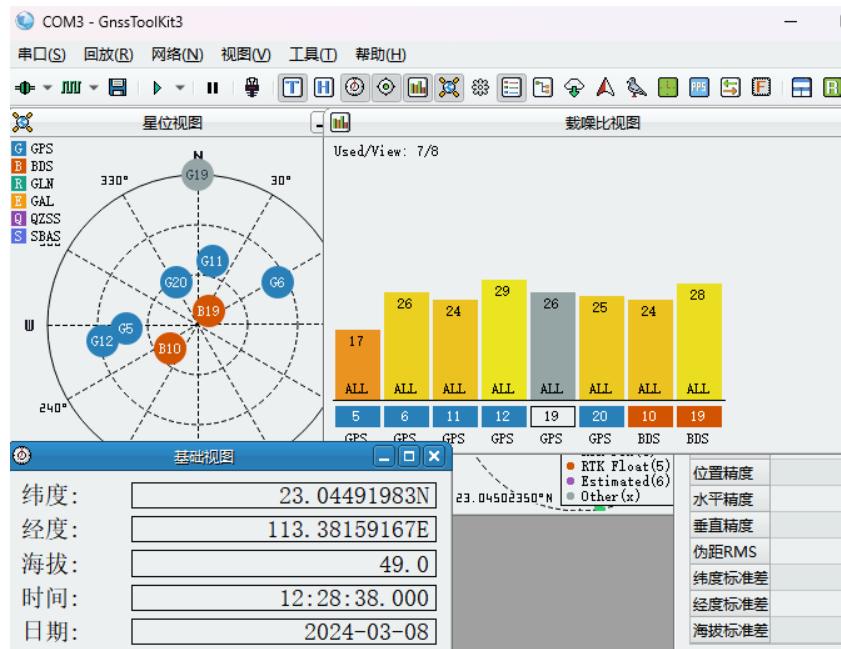


图 32 调试环节下，定位传感器卫星接收结果

对于 GPS 定位模块，在回调函数中读取一次定位信息，打包成数据包，具体格式如下：

包头	8 位数据	8 位数据	8 位数据	8 位数据	8 位数据
0xA0	纬度[0:7]	纬度[8:15]	纬度[16:23]	经度[0:7]	经度[8:15]
8 位数据	校验码	校验码	校验码	包尾	8 位数据
经度[15:24]	校验码[0:7]	校验码[8:15]	校验码[16:23]	0x5A	经度[15:24]

其中校验码计算函数如下：

```
int GPSCalculateCheck(int latitude, int longitude) {
    int code = latitude + longitude;
    if(latitude >= -90*1000000 && latitude <= 90*1000000)
        if(latitude > 0)
            code += 100;
    if(longitude >= -180*1000000 && longitude <= 180*1000000)
        if(longitude > 0)
            code += 10;
    return code;
}
```

3.2.2 人行道方向判断功能

一、高德地图 API 逆地理数据解析

通过 rpmsg 从 Slave 端获取实时的经纬度坐标 (x, y) ，传递给地理信息解析模块进行解析。解析时模块调用高德地图逆地理编码服务，将经纬度坐标作为参数发送给高德地图的 API，获取相应的地理位置信息。

高德地图返回的数据格式为 XML 格式，首先将 XML 数据解析为 Python 字典，字典中包括当前位置的街道走向，城市编码等内容，从字典中提取出 `formatted_address` 键对应的值，即为解析后得到的地理位置信息。

传入参数示例如下：

```
https://restapi.amap.com/v3/geocode/regeo?output=xml  
&location=116.310003,39.991957 // 输入的经纬度坐标（示例）  
&key=<用户的 key> // API key  
&radius=1000 // 查询 POI 的半径范围，单位：米  
&extensions=all // 返回结果控制
```

传出参数示例如下：

```
<building>  
  <name>建筑名称</name> <type>科教文化服务；学校；高等院校</type>  
</building>  
<roads>  
  <road>  
    <id>道路 id</id> <name>道路名称</name> <distance>道路到请求坐标的距离  
</distance>  
    <direction>点和此路的相对方位</direction> <location>坐标点</location>  
</road>  
</roads>
```

其中，获取大量当前位置附近的每一条道路的坐标点 location 信息 (x_1, y_1, x_2, y_2) 并输入到 Slave 端，可以在 Slave 端中分析得到人行道方向的判断结果。

二、Slave 端：周边道路图 (Graph) 的建构

周边道路图 (Graph) 的建构通过函数 `MergePoints()` 实现。在构建和维护所在位置周边城市道路网络的数字映射中，准确地处理和表达道路图 (Graph) 至关重要。该函数用于处理和优化道路网络图中的节点数据，特别是针对接近彼此的坐标点进行合

并。该函数的核心功能在于减少图中冗余节点，从而简化道路数据的处理和提升道路图的精度。

初始化和数据准备: 函数开始时，首先将道路列表（每条道路由一对坐标点组成，标识道路的起点和终点）转换成列表形式，以便于修改数据。此列表包含多条道路，每条道路由两个坐标点（起点和终点）定义。

每条道路的定义形式如下：

$$(x_1, y_1, x_2, y_2)$$

上述 x_1 与 x_2 指道路线段起终点的经度值；而 y_1 与 y_2 指道路线段起终点的纬度值。

设置阈值: 设定一个合并阈值（threshold），该阈值是用于判断两个点是否足够接近以至于可以合并。在此函数中，阈值设为 0.0004（40m），此值是基于经验或特定的准确度需求设定的。

建立点的索引: 创建一个字典来索引每个点的初始坐标，确保每个点都被唯一标识和追踪。

点的合并操作: 函数通过嵌套循环对每对点进行比较。如果两点之间的距离小于或等于设定的阈值，这两个点将被视为重复或足够接近，应该合并。合并点的新坐标通过一个名为 UpdatePoint 的辅助函数计算，该函数取两点的平均位置作为新位置。

更新引用: 更新所有引用这两个坐标点的道路数据，确保道路图中的每条道路都引用正确的节点位置。

三、Slave 端：当前运动方向计算

当前运动方向的计算通过函数 GetCurrentLocationAndDirection() 实现。在 Slave 端，我们维持一个具有 6 个坐标点长度的 Queue，每当更新一个数据点时，最旧的数据点将会出栈，而最新的数据点将会入栈处理。在计算过程中，为了充分利用所有的数据点，我们将会按照以下公式，得到偏移角度：

$$\begin{cases} \Delta y = \frac{(y_1 - y_4) + (y_2 - y_5) + (y_3 - y_6)}{3} \\ \Delta x = \frac{(x_1 - x_4) + (x_2 - x_5) + (x_3 - x_6)}{3} \\ \theta = \arctan\left(\frac{\Delta y}{\Delta x}\right) - 90 \text{ deg} \end{cases} \quad (24)$$

其中， y 为各项纬度值（北纬为正值，南纬为负值）； x 为各项经度值（东经为正值，西经为负值）； θ 为角度，单位为 deg，以正北为 0 deg，顺时针为正方向。

四、找到最近的道路，计算角度偏移，判断是否沿道路行走

1. 找到最近的道路

首先，系统从当前位置出发，搜索与当前图内所有道路边（路径）的最近距离。这个过程通过计算点到线段的最小距离来实现。如果最近的道路距离小于特定阈值（例如 80 米），则认为当前位置靠近某条道路。如果两条或更多的道路距离都小于较小的阈值（如 30 米），则认为当前位置靠近路口。具体计算如下：

$$d = \min \left(\frac{|(y_2 - y_1)x - (x_2 - x_1)y + x_2y_1 - y_2x_1|}{\sqrt{(y_2 - y_1)^2 + (x_2 - x_1)^2}} \right) \quad (25)$$

其中， (x, y) 是当前位置坐标，而 (x_1, y_1) 与 (x_2, y_2) 是道路的两端点坐标。

2. 计算角度偏移

一旦确定了最近的道路，下一步是计算当前运动方向与该道路方向的角度偏移。道路方向可以通过计算从道路一端到另一端的向量方向得到，而当前方向通常由位置跟踪设备提供。角度偏移计算如下：

道路方向计算：

$$\theta_{\text{road}} = \tan^{-1} \left(\frac{y_2 - y_1}{x_2 - x_1} \right) \quad (26)$$

角度偏移计算：

$$\Delta\theta = \theta_{\text{road}} - \theta_{\text{current}} \quad (27)$$

其中， θ_{current} 是当前运动方向的角度。

为了处理超过 360° （数值越界）的情况，角度偏移进行了调整，使其保持在 $(-180^\circ, 180^\circ]$ 范围内。

3. 判断是否沿道路行走

最后，通过比较角度偏移与预设的阈值（如 $\pm 30^\circ$ ），确定是否沿道路方向行走。如果当前方向与道路方向的偏差在 $\pm 30^\circ$ 范围内，或者接近正反方向 ($\pm 180^\circ$)，则认为是沿道路行走。这一逻辑反映在以下判断中：

$$\text{isAlongRoad} = (\Delta\theta \in [-30^\circ, 30^\circ]) \vee (\Delta\theta \in [-180^\circ, -150^\circ]) \vee (\Delta\theta \in [150^\circ, 180^\circ])$$

通过上述方法，系统能够准确判断视障者当前位置是否在道路上，并且确保视障者的行走方向是否与道路方向一致。

3.2.3 陀螺仪传感器数据解析

本系统采用 JY61P 加速度计作为使用者姿态数据的采集模块。JY61P 是一款常用的惯性测量单元 (IMU)，通常用于测量和跟踪物体的姿态、加速度和角速度等信息。它由加速度计、陀螺仪和磁力计等传感器组成，能够提供多种惯性参数的测量数据。

一、陀螺仪传感器原理

加速度测量

加速度计使用微机电系统 (MEMS) 技术，其中包括一个微型质量块、弹簧和电容传感器。当传感器沿一个轴（如 X 轴）加速时，内部的微型质量块由于惯性会在相反方向移动，导致弹簧形变。这种形变改变了与质量块相连的电容器的电容值，从而可以通过测量电容值的变化来计算出加速度。加速度测量通过以下数学模型来表示：

$$a_x = \frac{\Delta C_x}{S_x} k \quad (28)$$

其中， a_x 是 X 轴的加速度， ΔC_x 是电容变化量， S_x 为灵敏度系数， k 为根据加速度的量程调整的系数。类似的公式也适用于 Y 轴与 Z 轴。

角速度测量

陀螺仪用来测量角速度，同样基于 MEMS 技术。它工作的原理是基于科里奥利效应，即当一个旋转的物体在另一个方向上被加速时，会感受到一个垂直于旋转和加速度方向的力。在 MEMS 陀螺仪中，振动的质量会因为其旋转而受到微小的力的作用，这种力的变化会被转换为电压信号，从而测量角速度。数学表达式为：

$$\omega_x = \frac{V_x}{K_x} \quad (29)$$

其中， ω_x 是绕 X 轴的角速度， V_x 是由于科里奥利力引起的电压变化， K_x 是灵敏度系数。Y 轴和 Z 轴同理。

角度测量

角度的测量通常通过集成陀螺仪输出的角速度来得到。具体来说，可以通过对角速度信号进行时间积分来获得旋转角度。例如，绕 X 轴的角度可以表示为：

$$\theta_x(t) = \int_0^t \omega_x(\tau) d\tau + \theta_x(0) \quad (30)$$

其中， $\theta_x(t)$ 是时间 t 时的角度， $\omega_x(\tau)$ 是时间 τ 时的角速度， $\theta_x(0)$ 是初始角度。对于 Y 轴和 Z 轴，计算方式相同。

二、陀螺仪传感器通信协议

由于系统高实时性的要求，我们使用 I²C 串行总线读取该姿态传感器的加速度、陀螺仪数据，最高采集速率可达到 400KHz。

读取数据时，主控设备向姿态传感器发送一个 Start 信号，再将模块的 I²C 地址 IICAddr 写入寄存器地址 RegAddr，然后向模块发送设备地址信号 ($IICAddr \ll 1$)|1，姿态传感器接收到主机的信号后开始输出数据。接下来主机开始接收数据，每当主机一个字节后，就拉低 SDA 总线，向模块发出一个应答信号。直到接收完指定数量的数据，主机向模块再发送一个停止信号，结束一次数据读取。

我们将读取到的数据根据以下方式进行解析。

加速度数据包解析方式:

0x55	0x51	AxL	AxH	AyL	AyH	AzL	AzH	TL	TH	SUM
------	------	-----	-----	-----	-----	-----	-----	----	----	-----

计算方法：

$$a_x = ((AxH << 8) | AxL) / 32768 * 16g \quad (g \text{ 为重力加速度, 可取 } 9.8m/s^2)$$

$$a_y = ((AyH << 8) | AyL) / 32768 * 16g \quad (g \text{ 为重力加速度, 可取 } 9.8m/s^2)$$

$$a_z = ((AzH << 8) | AzL) / 32768 * 16g \quad (g \text{ 为重力加速度, 可取 } 9.8m/s^2)$$

温度计算公式：

$$T = ((TH << 8) | TL) / 100 \text{ } ^\circ\text{C}$$

校验和：

$$\text{Sum} = 0x55 + 0x51 + AxH + AxL + AyH + AyL + AzH + AzL + TH + TL$$

角速度数据包解析方式:

0x55	0x52	wxL	wxH	wyL	wyH	wzL	wzH	TL	TH	SUM
------	------	-----	-----	-----	-----	-----	-----	----	----	-----

计算方法：

$$wx = ((wxH << 8) | wxL) / 32768 * 2000(^{\circ}/s)$$

$$wy = ((wyH << 8) | wyL) / 32768 * 2000(^{\circ}/s)$$

$$wz = ((wzH << 8) | wzL) / 32768 * 2000(^{\circ}/s)$$

温度计算公式：

$$T = ((TH << 8) | TL) / 100 \text{ } ^\circ\text{C}$$

校验和：

$$\text{Sum} = 0x55 + 0x52 + wxH + wxL + wyH + wyL + wzH + wzL + TH + TL$$

角度数据包解析方式:

0x55	0x53	RollL	RollH	PitchL	PitchH	YawL	YawH	VL	VH	SUM
------	------	-------	-------	--------	--------	------	------	----	----	-----

计算方法：

$$\text{滚转角 (x 轴)} \text{ Roll} = ((RollH << 8) | RollL) / 32768 * 180(^{\circ})$$

$$\text{俯仰角 (y 轴)} \text{ Pitch} = ((PitchH << 8) | PitchL) / 32768 * 180(^{\circ})$$

$$\text{偏航角 (z 轴)} \text{ Yaw} = ((YawH << 8) | YawL) / 32768 * 180(^{\circ})$$

固件版本计算公式：

$$\text{Version} = (VH << 8) | VL$$

校验和：

$$\text{Sum} = 0x55 + 0x53 + RollH + RollL + PitchH + PitchL + YawH + YawL + VH + VL$$

三、陀螺仪传感器的核间通信帧

Slave 端在完成数据采集后，将采集到的数据打包发送到 Master 端，实现核间通信。该 Slave 端的 endpoint 命名为 jy61p。在 rpmsg endpoint 回调函数中调用采集函数，采集一次六轴数据数据，并将读取到的数据打包成三个数据包，具体格式如下：

包头	8 位数据	8 位数据	8 位数据	数据代码	包尾
160 (十进制)	roll	pitch	yaw	加速度: 1 角速度: 2 角度: 3	90 (十进制)

由于每个数据包已经在硬件通信协议中有相应的校验和计算，因此在核间通信过程中，不单独在传输过程中设置独立的校验和。

在一次回调函数中，分三次将数据包发送到端点中，避免了一次发送导致的数据溢出。

3. 2. 4 摔倒与视角异常检测

在本系统中，摔倒与视角异常检测均在 Slave 端检测，并将结果通过异常情形下的语音输出，反馈给语音播报模块，并返回到 Master 端，显示在 GUI 界面上。

一、摔倒检测

系统通过加速度计检测使用者的行走状态，当传感器检测到的加速度轴数据超过设定的阈值并且角速度异常时，判定使用者行走时发生摔倒，并且在客户端记录显示，同时发出语音警报，防止周围行人碰撞对视障人士造成进一步伤害。

在我们的系统中，这一阈值设定为：pitch、yaw、roll 任一轴的加速度大于 3m/s^2 ，即：

$$\exists |a_{pitch}|, |a_{yaw}|, |a_{roll}| \geq 3 \text{ m/s}^2 \quad (31)$$

二、视角异常检测

为了保证系统的正常工作，用户需要保证相机的视野向前。在我们的系统硬件中，相机和加速度计的相对位置需要保持不变。检测到异常的阈值设置为：

$$\exists |\omega_{pitch}| \geq 40 \text{ deg}, |\omega_{roll}| \geq 20 \text{ deg} \quad (32)$$

当加速度计测得的偏移角度异常时，如：系统将判定此时相机的视角异常，可能过度向下、向上倾斜或向 roll 左倾或右倾方向倾斜，此时将提示用户相机视角异常，需要调整后使用。

3.2.5 温湿度环境信息数据解析

本系统选用 DHT11 作为温湿度传感器，用于采集环境的温度和湿度。DHT11 传感器采用单一的数字信号线进行数据传输，通过 OneWire 总线协议对模块写时序和读时序的操作，实现温度和湿度的数据采集。

一、DHT11 传感器的工作原理

1. 温度数据采集原理

DHT11 传感器中温度测量是通过一个热敏电阻（或称为负温度系数（NTC）热敏电阻）来实现的。热敏电阻的电阻值随温度变化而变化，通常温度升高，电阻值降低。传感器内部的微处理器将电阻值转换为温度读数。数学表达式如下：

$$R = R_0 e^{B \left(\frac{1}{T} - \frac{1}{T_0} \right)} \quad (33)$$

其中， R 是热敏电阻在温度 T 下的电阻值（欧姆）。 R_0 是参考温度 T_0 （通常是 298 K）时的电阻值。 B 是材料的特性常数。 T 绝对温度（开尔文）。

2. 湿度数据采集原理

湿度测量则通过一个电容式湿度传感器来实现。该传感器利用空气湿度对电容的影响来测量相对湿度。空气中的水分子会改变电容器两板之间的介电常数，进而改变电容值。这个电容值的变化与空气的湿度成正比。传感器电路将这种电容的变化转换为数字输出，反映当前的相对湿度。数学表达式如下：

$$C = \epsilon_r \epsilon_0 \frac{A}{d} \quad (34)$$

其中， C 是电容值（法拉）。 ϵ_r 是材料的相对介电常数，该值随湿度变化。 ϵ_0 是真空的介电常数（约等于 $8.85 \times 10^{-12} \text{ F/m}$ ）。 A 是电容器板的面积（平方米）。 d 是电容器板之间的距离（米）。

二、DHT11 传感器的 OneWire 总线协议

通过飞腾派向该传感器发送启动信号，当传感器接收到正确的信号后，DHT11 模块会周期性地发送 40 位数据，包括温度和湿度信息以及校验位。在飞腾派读取数据的

数字信号后，如下图所示，根据 DHT11 的通信协议进行解析，得到环境的温度和湿度值。

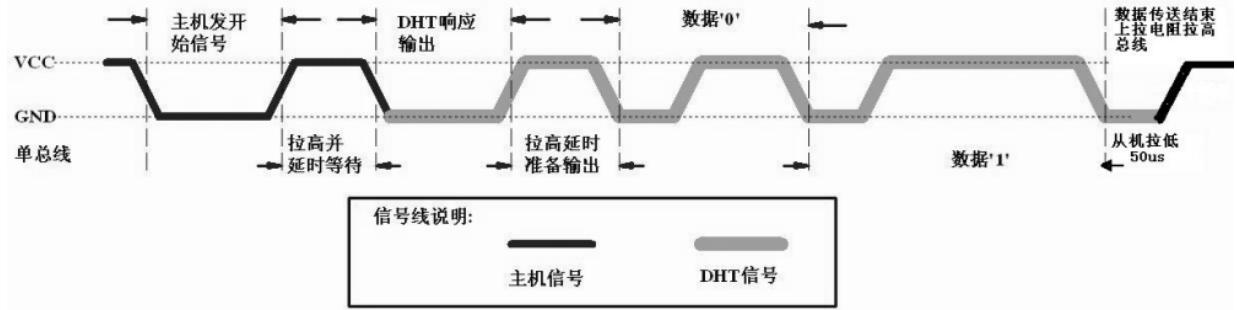


图 33 DHT11 传感器的 OneWire 总线协议时序

三、DHT11 传感器的核间通信帧

DHT11 传感器的数据采集工作在 Slave 端的 `rpmmsg endpoint` 回调函数中执行，该 Slave 端的 `endpoint` 被命名为 `dht11`。每次 Master 端向此 `endpoint` 发送任意指令数据，Slave 端将及时响应此请求，执行数据采集，并返回一帧数据。Master 端接收到数据帧后，将根据以下的校验码计算步骤重新计算校验码。如果重新计算得出的校验码与 Slave 端传送过来的校验码一致，则接受该数据帧；如果不一致，则丢弃该数据帧。

数据帧的结构如下，帧由五个字节组成，每个字节均为 `uint8` 类型：

<code>byte[0]</code>	<code>byte[1]</code>	<code>byte[2]</code>	<code>byte[3]</code>	<code>byte[4]</code>
字头, <code>0xA0</code>	湿度数据（单位：%）	温度数据（单位：摄氏度）	校验码	字尾, <code>0x5A</code>

校验码的计算公式为：

$$x = (h + t) \bmod 256 \quad (35)$$

其中， x 代表校验码（取低 8 位）， h 为湿度值（整数）， t 为温度值（整数）， \bmod 表示求余运算。

3.3 监护客户端显示模块

监护客户端显示模块使用 PySide6 实现 GUI 设计，该模块面向视障者的监护人设计。除了充当系统的总开关外，同时为监护人实时显示摄像头的画面信息，以及斑马

线方向、交通灯亮灯状况、行人楼梯级数、目标检测结果信息；特色小地图功能以俯视图的“雷达”状画面，显示视障者前方 10m 内的所有目标检测物体的位置、角度信息；通过信息栏显示用户的位置信息、摔倒状态、视角状态、温湿度状态等。

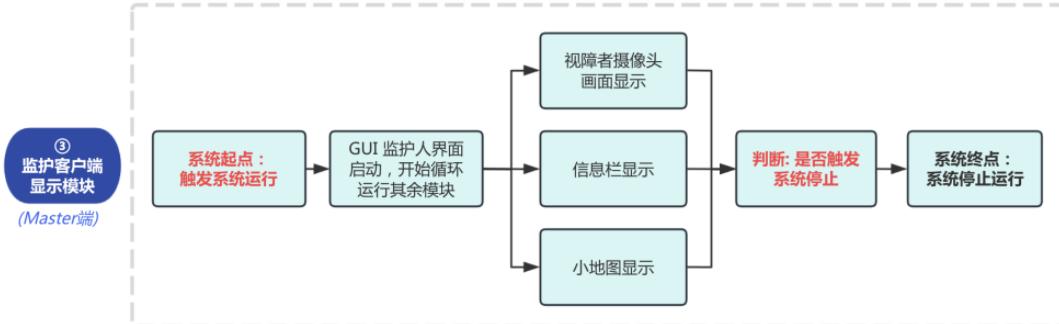


图 34 监护客户端显示模块总框图

3.3.1 监护客户端 GUI 界面设计

一、界面布局

客户端界面总体采用灰色调色彩，界面左上角添加自设计的队伍 LOGO，并隐藏了窗口的标准边框和标题栏。通过自定义方法实现界面平滑切换及边缘调整，配合阴影显示构建了简洁美观的主界面。界面布局设计旨在实现直观操作与信息高效传达，主要由以下几部分构成：

- 左侧边栏：**整合了快速访问工具，包括展开菜单、视频文件、摄像头和网络摄像头输入选项，便于用户根据实际需要灵活选择数据源。
- 顶部工具栏：**简洁明了，集成了窗口最小化、最大化/恢复及关闭按钮，满足基本的窗口管理需求。
- 底部工具栏：**底部工具栏分布于地理位置及天气标签两侧，提供了一键启动/停止系统运行的便捷操作，增强了用户交互的灵活性。

主界面分为两大板块：

- 首页：**集中展示了监护的核心信息，包括 GPS 精确定位（经纬度）、摔倒检测状态、温湿度监测数据，以及基于视觉处理的障碍物小地图，为监护人提供全方位的环境感知。

b) 系统监控页：转向后台性能监控，清晰罗列了 Master 端 CPU 使用状况、系统及通信响应时间，辅以 CPU 综合负载视图和视频效果对比（用户侧与神经网络处理侧），为用户提供了详尽的性能评估工具。

显示效果如下所示：



图 35 GUI 初始化界面

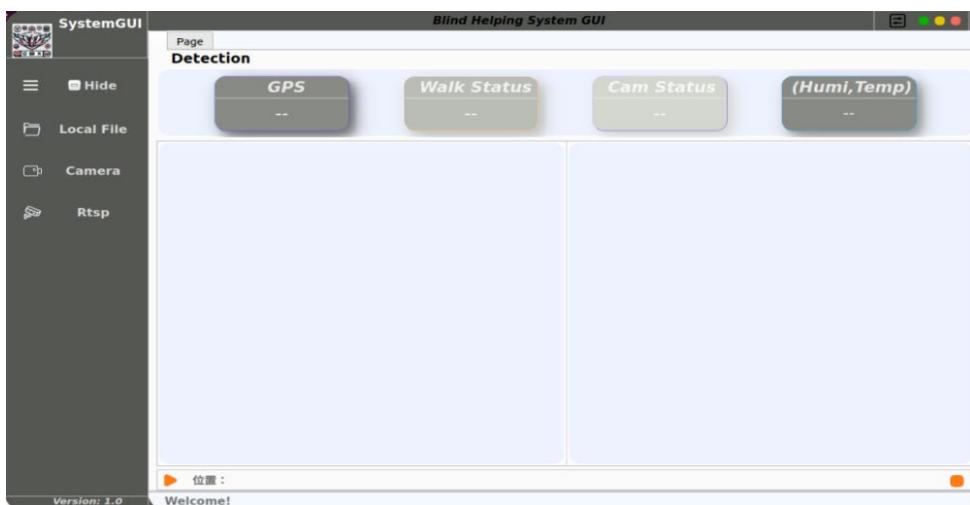


图 36 GUI 界面的侧边栏显示

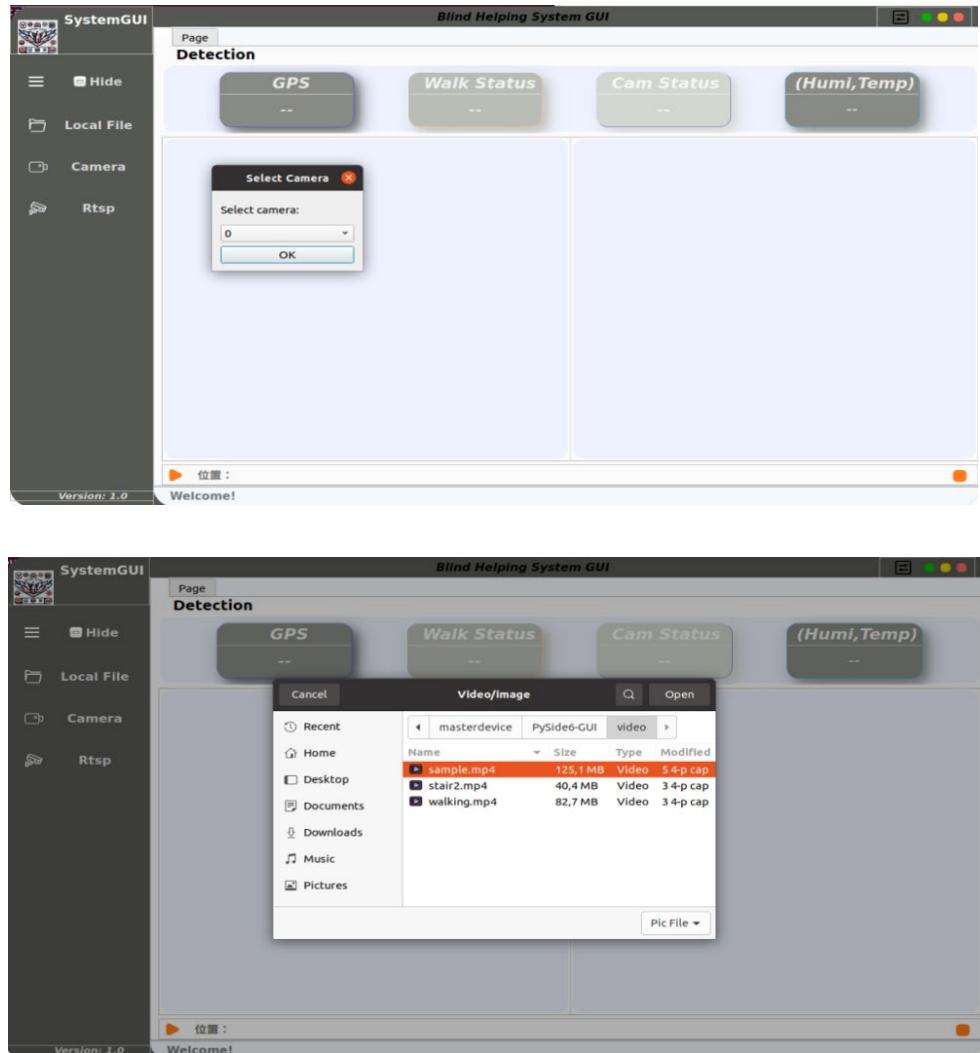


图 37~38 GUI 视频输入与相机输入选择界面

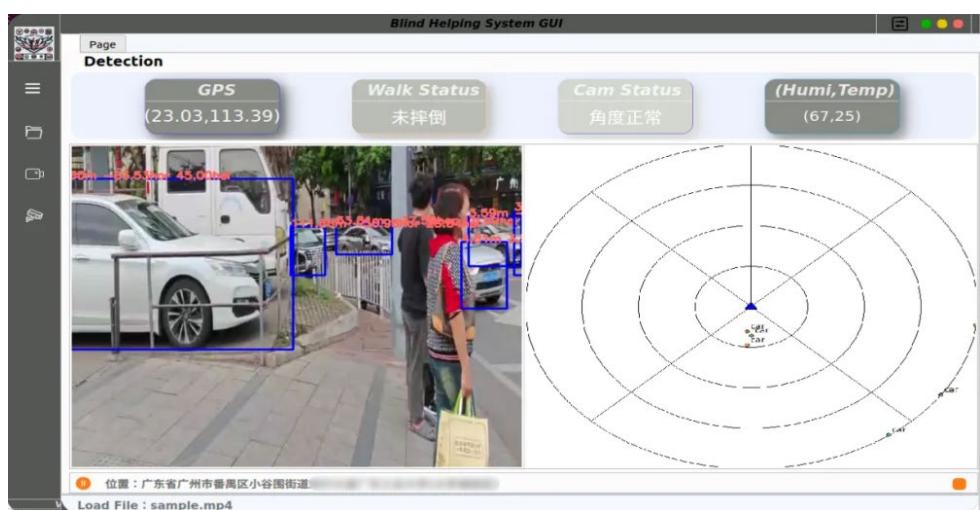




图 39~40 客户端 GUI 首页与资源监控页

二、程序设计

(一) 主程序设计

在 GUI 客户端主程序设计中，我们通过多进程和多线程的综合应用，确保了多任务的独立执行和实时数据的快速响应，同时兼顾了多线程的实时性和多进程的高效能。我们使用 Multiprocessing 框架的多进程方法来处理各项任务，实现真正的并行处理，主要分为传感器信息处理进程、视觉处理进程和界面显示进程，在每个进程中都有多个子线程处理不同的数据。

界面显示进程是继承自 QMainWindow 类的首个启动进程，负责激活传感器信息处理进程和视觉处理进程，并初始化信号及其对应的槽函数，以确保信息能及时更新至 GUI 客户端的界面显示。传感器信息处理进程和视觉处理进程都是继承自 QObject 类的，这样的继承关系允许在相应的进程中使用 PySide6 框架的信号和槽机制进行通信。但是，在独立的进程中，不同的任务由不同的子线程处理，而信号和槽机制要求所有信号的发送都必须在主线程中进行，以确保线程安全。因此，我们采用队列来存储每一组需要传递至 GUI 界面显示的数据，并设置定时器在主线程中定期触发，发送信号到界面显示进程，确保界面端能够正常接收数据。

(二) 主界面功能扩展

在客户端的图形用户界面(GUI)设计中，我们不仅追求功能的全面集成，还注重用户体验的优化与界面的直观性。通过自定义的 UIFuncitons 类实现以下界面扩展功能：

1. 左侧菜单展开与收起 (toggleMenu 方法): 允许用户通过动画效果切换左侧菜单栏的宽度，从而展示或隐藏菜单内容。
2. 窗口最大化与还原 (maximize_restore 方法): 控制窗口在最大化和正常状态之间切换。当最大化时，会隐藏窗口的自定义边缘调整把手，反之则显示这些把手。
3. 窗口控制与交互定义 (uiDefinitions 方法):
 - a) 双击标题栏实现窗口的最大化/还原。
 - b) 当窗口处于非最大化状态时，可以通过点击并拖动标题栏移动窗口位置。
 - c) 添加了四个角落的自定义边缘调整把手 (CustomGrip 实例)，允许用户调整窗口大小。
 - d) 窗口最小化、最大化/还原、关闭按钮的信号连接。
4. 调整边缘把手的位置 (resize_grips 方法): 根据窗口的实时大小调整四个边缘调整把手的位置。
5. 窗口阴影效果 (shadow_style 方法): 为指定的窗口部件添加图形阴影效果，增加视觉上的深度感。

综上所述，客户端 GUI 设计不仅考虑了功能的完整性，还深度融合了用户体验优化策略，从信息展示逻辑到辅助功能的每一个细节，都力求实现直观、高效与人性化的操作体验。

3.3.2 监护客户端 GUI 功能详述

一、数据显示

系统首页设计上，我们采用了现代化的扁平化风格，确保视觉元素简洁而不失丰富信息量。界面顶部，GPS 模块实时反馈的经纬度坐标以醒目的形式展现，辅以底部显示的逆地理编码得到的位置信息，直观反映视障用户的当前位置。紧随其后，相机状态、摔倒检测、温湿度数据显示区域以醒目文字呈现，迅速引起注意。在使用前需要调整相机时，也可通过客户端界面查看当前的相机视野进行辅助调整。

页面中段，视觉处理的成果——障碍物小地图，由 yolov5v8.py 神经网络推理功能驱动，结合 coordination.py 实现的距离与角度解算，以动态极坐标图形式描绘出用户

周围环境，图中原点代表用户所在位置，每一点代表潜在的障碍物，图表上的距离表示接近程度，为用户提供即时的环境感知。

底部，地理定位结合的天气情况展示，利用高德地图逆地理编码(gpsdecode.py)与天气 API 接口，为监护人提供全面的户外信息概览，确保用户安全。

系统监控页则以数据可视化为核心，将多进程的运行效率和系统资源使用情况直观展示，展现 Master 端 CPU 使用率、系统响应延迟等关键性能指标，以及 Slave 端的核间通信效率，使用户对系统状态了如指掌。右侧的视频效果对比区，直观对比原始视频流与经过神经网络处理后的差异，验证算法的有效性，同时提供调试便利。

二、状态播报

除了以文字呈现相机状态及行走状态外，系统定时调用语言播报模块播报当前相机状态及行走状态，句式如“摔倒，相机视角异常”，提醒用户扶正相机，同时通过语音提醒旁人有人摔倒需要避让，避免踩踏视障者造成进一步伤害。

当视野内检测到障碍物时，系统除了会在 GUI 界面以小地图的形式展示实时的障碍物分布外，还会通过预设的语音播报逻辑调用传感器信息进程下的对应逻辑启动语音播报，用语音播报的方式告知视障用户周围的障碍物分布情况，以及离用户所在位置最近的 5 个障碍物。

当视野内检测到斑马线或红绿灯时，系统除了会在 GUI 界面显示的视频上进行标注外，还会通过预设的语音播报逻辑告知通过语音播报的方式告知视障用户斑马线的走向、斑马线的方位和距离用户最近的红绿灯的颜色，用户需要过马路时可根据语音播报的反馈在绿灯时沿斑马线通过。

当行走过程中视野内检测到阶梯时，系统除了会在首页显示的视频上标注外，同样会通过预设的语音逻辑进行播报，告知视障用户阶梯的所在方位以及阶梯数量。人行道上行走时，用户可根据阶梯数提示避免摔倒。除此以外，当用户需要通过人行天桥过街时，可根据语音播报的反馈向阶梯所在方位移动沿人行天桥完成过街。

三、辅助功能与细节优化

界面调整工具 (custom_grips.py): 为了让用户根据个人偏好调整界面布局，我们集成了自定义窗口边缘调整功能，使得窗口大小调整更加灵活自如。

资源与配置管理：通过 resources.qrc 和 fold.json，我们不仅统一管理 UI 资源，还实现了用户习惯的记忆功能，自动加载上次使用的文件夹路径，提升软件的易用性。

驱动层整合：在 drivers 目录下，我们通过 rpmsg.py 实现了 OpenAMP rpmsg 协议的高效数据交换，确保跨平台通信的实时性与稳定性。同时，detectResultPackUp.py 和 speakout.py 模块提供了灵活的视觉检测结果打包与语音播报功能。

3. 3. 3 监护客户端的远程访问实现

一、VNC 远程桌面实现

在监护客户端的设计与部署中，实现高效、低延迟的远程访问功能至关重要，尤其是在基于 Ubuntu 系统环境下。选择合适的远程访问技术至关重要。常见的选择包括 VNC（如 TightVNC 讨论的）、RDP（Remote Desktop Protocol）、SSH（Secure Shell）隧道搭配 X11 forwarding（针对 Linux），以及现代的云原生解决方案如 TeamViewer、AnyDesk 等。对于监护客户端，考虑其易用性、安全性、跨平台兼容性和资源消耗，TightVNC 结合 SSH 隧道或使用 TeamViewer 这类商业解决方案可能是较优选择。

概述：VNC (Virtual Network Computing) 是一种远程桌面协议，允许用户通过网络控制另一台计算机的桌面。TightVNC 和 TurboVNC 是 VNC 的优化版本，专为减少带宽使用和提高画质设计。我们安装 VNC 服务端在 Ubuntu 上，客户端安装对应的 VNC 查看器，配置适当的压缩级别和帧率以优化性能，实现在另一设备上远程控制 Ubuntu 的 xfe 桌面。

配置步骤：我们在 Ubuntu 系统上安装 tightvncserver，首次运行 TightVNC Server 以进行初始配置，完成登录密码配置，设置的密码将用于远程访问时的身份验证。用户在 VNC 客户端中输入 Ubuntu 系统的 IP 地址（或通过 SSH 隧道映射的本地地址）和之前设置的显示号（例如，如果之前是:1，端口号是 5901）。输入密码后，应该能看到 Ubuntu 的桌面环境，从而可以远程控制应用。使用 VNC 协议传输我们需要确保网络环境允许两台设备之间的连接，如果不在同一局域网内，我们可以设置端口转发或使用 DDNS 服务建立连接。

显示效果：



图 41 远程桌面显示

二、SSH 与远程 Qt 后端实现

使用 SSH 与远程 Qt 后端相比于 VNC 远程桌面显示对远程开发板的资源占用更小。我们在本地主机中安装 X server（如 Xming 或 VcXsrv），并启用 SSH X11 转发，即在 SSH 配置文件中启用“X11Forwarding yes”配置项。

在具体操作过程中，首先启动本地主机上的 X server；用户在本地终端 SSH 程序中登录远程 Ubuntu 主机，并启用远程端的 X11 转发；登陆成功后运行远程端的 Qt 程序即可。

3.3.4 系统性能监控测量环节及显示

一、功能描述

本模块用于周期性地监测系统的 CPU 使用状况，包括每个单独 CPU 核心的使用率以及整个系统的总 CPU 使用率，并将这些信息实时反馈至用户界面，为系统管理员或监控者提供即时的性能指标。

二、技术实现

1. **获取每个 CPU 核心使用率:** 利用 psutil 的 `cpu_percent` 方法, 通过设定 `interval=1` 秒的参数来获取每个 CPU 核心在过去一秒的平均使用率。此操作为用户提供了详细的 CPU 负载信息, 每个核心的使用情况被收集到列表 `cpu_usage` 中。之后, 使用 `map` 函数将列表中的每个使用率转换为字符串, 通过连接后加上括号形成符合显示格式的字符串 `cpu_str`, 并通过信号 `self.cpusig.emit(cpu_str)` 发送至 UI 界面显示。

2. **系统整体 CPU 使用率:** 同样使用 `psutil.cpu_percent(interval=1)`, 但不指定 `percpu=True` 来获取整个系统 CPU 的平均使用率。这代表了系统所有 CPU 核心的综合负载情况。该值被格式化为字符串, 并通过 `self.cputotal.emit(f"total_cpu_usage{total_cpu_usage}%")` 发送至界面, 用于显示系统整体 CPU 使用百分比。

3. **获取系统响应时间:** 利用 `time` 方法记录系统启动时间 `time_start` 及系统响应时的时间 `time_right`, 通过计算两者时间差得到系统响应时间 `system time`, 同样通过信号与槽机制发送至界面显示。

4. **获取核间通信响应时间:** 同样利用 `time` 方法记录调用 `writeEptDevice` 方法从 Master 传递信号到 Slave 端开始写入数据的时间 `time_start` 及 Slave 端写入数据完成的时间 `time_right`, 计算两者时间差得到 Master 端到 Slave 端核间通信的响应时间 (Master to Slave), 格式化后再通过信号与槽机制发送到界面。

三、显示效果

CPU 核心使用率展示: 在界面上, 每个 CPU 核心的使用率将以“CPU X : Y%”的形式列出, 其中 X 为 CPU 核心编号, Y 为使用率百分比。这为用户提供了直观的各核心负载视图, 便于发现可能的瓶颈或不平衡问题。

系统整体 CPU 使用率: 系统整体的 CPU 使用率直接以百分比形式展示, 例如“系统整体 CPU 使用率: Z%”, Z 代表总使用率。此信息有助于快速判断系统是否处于高负载状态。

响应时间展示：根据返回的数值大小，系统响应时间最终以"X s"的格式显示，系统核间通信响应时间以"X ms"的格式显示。



图 42 系统的响应效果

3.4 传感器处理与语音反馈模块

传感器处理与语音反馈模块位于 Slave 端实现，该系统处理经过 Master 端的视觉模块、地理解析模块处理后数据，以及 Slave 端的陀螺仪传感器数据，通过特定的逻辑解析，最后通过 TTS 语音方式为视障者提供实时的语音播报信息。

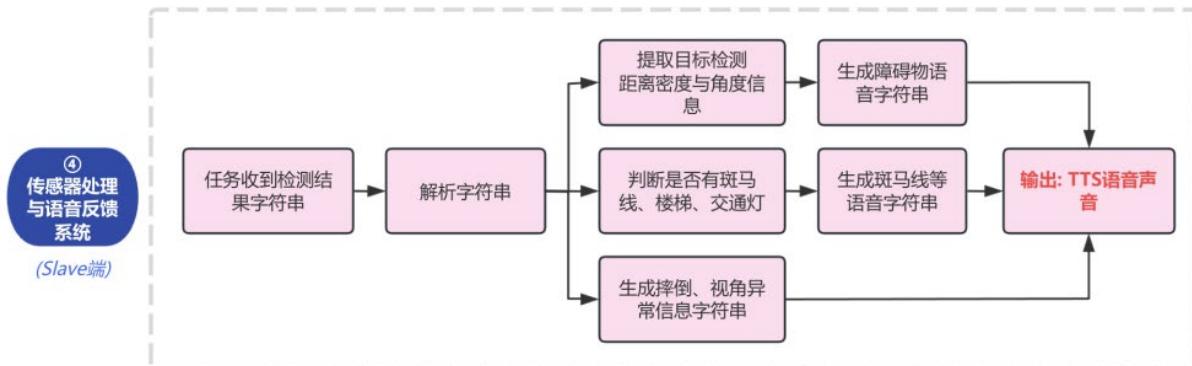


图 43 传感器处理与语音反馈系统总框图

3.4.1 TTS 语音合成模块通信协议

本系统采用 XFS5152CE TTS (Text To Speech) 语音合成模块，通过该模块将外界提示信息以语音的方式实时反馈给使用者。该模块使用 I²C 通讯协议，作为 I²C 的从设备使用。

命令帧的封装：上位机发送给语音识别模块的所有语音播报命令都需要用“帧”的方式进行封装后传输。帧结构由帧头标志、数据区长度和数据区三部分组成。具体的命令说明如下所示：

帧头	数据区长度	数据区
0xFD	0xXX, 0XX	Data

名称	长度	说明
帧头	1 Byte	定义为十六进制 “0xFD”
数据区长度	2 Bytes	用两个字节表示，高字节在前，低字节在后
数据区	小于 4k Bytes	命令字和命令参数，长度和“数据区长度”一致

数据区是由命令字和命令参数组成的，上位机使用命令字来实现语音播报芯片的各种功能。不同命令字有对应不同的命令参数。

名称	发送的数据	说明	
命令字	0x01	语音合成命令	
	0x02	停止合成命令，没有参数	
	0x03	暂停合成命令，没有参数	
	0x04	恢复合成命令，没有参数	
	0x21	芯片状态查询命令	

语音播报命令输入：语音播报通过 I2C 将帧头固定值为 0xFD，数据区长度分为两个字节，数据区长度为带合成文本长度+2（命令字和文本格式各为一个字节）。其中文本长度表示的是文本的字节长度。具体命令说明如下所示：

名称	发送的数据	说明			
命令字	0x01	带文本编码设置的文本播放命令			
参数列表	0Xxx	1Byte 表示文本的编码格式，取值为 0~3	参数取值	文本编码格式	
			0x00	GB2312	
			0x01	GBK	
			0x02	BIG5	
			0x03	UNICODE	
命令帧格式结构	待合成文本的二进制内容				
	帧头	数据区长度		数据区	
	0xFD	高字节	低字节	命令字	文本编码格式

3.4.2 面向视障者的实时语音输出交互逻辑

一、模式触发逻辑

1. 默认逻辑:

默认情况下，仅运行目标检测播报逻辑，播报目标检测字符串。任何情况下，目标检测播报逻辑均处于运行状态。是否沿道路行走播报逻辑也默认开启，但播报频率相比于目标检测播报逻辑更低。

2. 增量逻辑:

当视角画面出现斑马线与红绿灯时，增加斑马线及其方向、红绿灯播报逻辑字符串，添加在目标检测播报逻辑字符串的后方。

当视角画面出现楼梯时，增加楼梯及级数的播报逻辑字符串，添加在目标检测播报逻辑字符串的后方。

若斑马线与红绿灯播报逻辑，与楼梯与级数播报逻辑同时出现时，楼梯与级数播报逻辑位于斑马线与红绿灯播报逻辑的后方。

3. 异常事件逻辑:

若出现视角异常与摔倒状况时，上述的所有默认逻辑、增量逻辑全部暂停运行，只频繁播报视角异常与摔倒状况下的字符串。即：“发生摔倒”与“相机视角异常”。

二、目标检测播报逻辑

该逻辑通过分析视障者前方不同距离区间的障碍物数量，判断整体是否拥挤，并指出障碍物密集的区间，最后根据近处对象的类型生成语音输出文本。

算法：目标检测播报逻辑

输入：

objects: 一个列表，包含大量的 (`x, y, class, distance, ver, hor`) 元组，每个元组描述一个对象的位置、类别、距离及垂直和水平角度。

输出：

output_text: 一段文本，描述前方环境的拥挤情况，距离区间内障碍物最多的位置，以及最近的几个物体的类别。

过程：

1. 初始化变量：

- `dist1, dist2, dist3, dist4`: 四个整数变量，用于统计不同距离范围内的对象数量。
 - `closest_class`: 列表，用于存储最近的几个对象的类别。
 - `is_total_crowded`: 布尔变量，表示是否拥挤。
 - `most_crowded`: 整数变量，表示哪个距离区间内最多障碍物。
2. 对列表中元组按 `distance` 升序排序。
3. 遍历排序后的列表：
- 若 `closest_class` 长度小于 5 且该对象的 `ver` 绝对值小于 45，将该对象的类别添加至 `closest_class`。
 - 根据 `distance` 的值增加相应距离区间的计数 (`dist1, dist2, dist3, dist4`)。
4. 判断总体拥挤情况：
- 如果所有距离区间的对象总数大于或等于 20，则设置 `is_total_crowded` 为 `True`，否则为 `False`。
5. 确定最拥挤的距离区间：
- 使用 `max` 函数比较 `dist1, dist2, dist3, dist4`，并根据最大值确定 `most_crowded` 的值。
6. 构建输出文本：
- `str1`: 如果 `is_total_crowded` 为 `True`，则设为“前方拥挤”，否则为“前方宽松”。
 - `str2`: 根据 `most_crowded` 的值，分别设为“近距离多障碍”，“中距离多障碍”，“远距离多障碍”，“极远处多障碍”。
 - `str3`: 将 `closest_class` 中的元素组合成一个字符串。
7. 输出组合后的字符串 `output_text = str1 + str2 + str3`。

该部分播报的完整句如下：

“前方拥挤/ 宽松”	“近距离/中距离/远距离/极 远处多障碍”	“1, 2, 3, 4, 5” (此处仅作举例，实际上是距离最近的 5 个物体 的物体 class 编号)
---------------	--------------------------	---

解释：

str1: 若总目标检测物体个数 > 20，那么判断“拥挤”，否则“宽松”。展示的是视障人士前方的拥挤状况

str2: 由所有物体的距离分布直方图的最值所在区间决定。“近距离”：最值区间为 [0,2.5m]；“中距离”：最值区间为 [2.5m,5m)；“远距离”：[5m,7.5m)；“极远距”：[7.5m,10m)。

str3: 先筛选所有目标检测物体的水平角为视障人士正前方 ±45deg 以内，再根据目标距离从近到远进行排序，取距离最近的 5 个物体的 class 编号。这里播报编号而不是物体类型名称的原因有二，一是直接播报编号时间更短、效率更高；二是我们假设视障人士在使用系统时，能够熟练掌握 class 编号与物体实际类型的对应关系。

三、斑马线及其方向与红绿灯播报逻辑

该逻辑负责解析视障者前方的斑马线及交通灯信息，输出描述斑马线位置及方向的字符串和交通灯状态的字符串，以辅助视障者安全出行。

算法：斑马线及其方向与红绿灯播报逻辑

输入：

zebralist: 由大量的(*x*, *y*, *deg*)元组组成
trafficlist: 由大量的(*x*, *y*, *lightstatus*)元组组成

输出：

zebrastring: 描述斑马线位置与方向的字符串
trafficstring: 描述交通灯状态与位置的字符串

过程：

1. 初始化变量：

- **is_zebraline_on_left**: 布尔变量，标识斑马线是否在左侧
- **is_zebraline_goes_left**: 布尔变量，标识斑马线方向是否向左
- **zebra_x**, **zebra_y**: 整数变量，存储斑马线的 *x* 和 *y* 坐标
- **traffic_number**: 整数变量，记录交通灯的数量
- **nearest_traffic_signal**: 整数变量，记录最近交通灯的状态

2. 处理斑马线列表：

- 若 **zebralist** 非空：
 - 根据 *y* 坐标从大到小排序 **zebralist**
 - 判断第一个元组的 *x* 坐标是否小于 320，以确定斑马线是否在左侧
 - 判断第一个元组的 **deg** 是否小于 0，以确定斑马线的方向
 - 构建描述斑马线的字符串：包括斑马线数量、位置（左侧或右侧）、方向（左前方或右前方）
 - 否则：**zebrastring** 为空字符串

3. 处理交通灯列表：

- 若 **trafficlist** 非空：
 - 计算交通灯总数并赋值给 **traffic_number**
 - 根据 *y* 坐标从大到小排序 **trafficlist**
 - 删除所有 **lightstatus** 为 0 的元组
 - 若 **zebralist** 非空且经过删除后 **trafficlist** 仍非空：
 - 寻找与 **zebralist** 中第一个元组坐标最近的交通灯，记录其 **lightstatus**
 - 构建描述交通灯的字符串：包括交通灯数量和最近交通灯的状态（红灯或绿灯）
 - 否则：**trafficstring** 为空字符串

该部分播报的完整句如下：

斑马线部分：

“识别到...条斑马线”	“脚下斑马线在您左/右侧”	“指向左/右前方”
--------------	---------------	-----------

交通灯部分：

“识别到...个交通灯”	“最近距离交通灯为红/绿灯”
--------------	----------------

四、楼梯及其级数播报逻辑

该逻辑通过分析楼梯的位置和数量，生成描述最近楼梯的位置（左侧、前方、右侧）及其阶梯数的文本。

算法：楼梯及其级数播报逻辑

输入：

`stairs_list`: 一个列表，包含所有楼梯的信息，每个元组格式为 `(x, y, stairsnumbers)`

输出：

`output_string`: 描述最近的楼梯位置和阶梯数的字符串

过程：

1. 初始化变量：

- `nearest_stair_number`: 整数变量，存储最近楼梯的阶梯数
- `nearest_stair_x`: 整数变量，存储最近楼梯的 x 坐标

2. 处理楼梯列表：

- 若 `stairs_list` 非空：
 - 根据 `y` 坐标从大到小排序 `stairs_list`
 - 从排序后的列表中取第一个元组，将其 `x` 坐标和阶梯数分别赋值给 `nearest_stair_x` 和 `nearest_stair_number`
 - 构建描述楼梯的字符串：
 - 字符串 `str1`: 表示识别到的楼梯总数
 - 字符串 `str2`: 根据 `nearest_stair_x` 的值，判断最近的楼梯是在左侧、前方还是右侧
 - 字符串 `str3`: 描述最近楼梯的阶梯数
 - 输出字符串 `output_string` 为 `str1`, `str2` 和 `str3` 的组合
 - 否则：
 - `output_string` 为空字符串

该部分播报的完整句如下：

“识别到...处阶梯”	“最近阶梯在您左侧/前方/右侧”	“阶梯数.....”
-------------	------------------	------------

五、是否沿道路行走播报逻辑

该逻辑通过判断用户是否在道路附近、是否在交汇口附近以及是否沿道路行走，来生成适当的语音输出，指导视障者正确地沿道路行走或警告其道路偏离情况。

该部分播报的完整句如以下逻辑所述：

算法：是否沿道路行走播报逻辑

输入：

`isalongroad`: 布尔变量，指示是否沿着道路行走（30 度阈值以内）

`isnearroad`: 布尔变量，指示是否在道路附近

`isnearcrossing`: 布尔变量，指示是否位于道路交汇口附近

输出：

`output_str`: 描述用户相对于道路位置的字符串

过程：

1. 检查是否在道路附近 (`isnearroad`) :

- 如果不在道路附近:

- `output_str` 设置为空字符串

- 如果在道路附近:

- 检查是否在道路交汇口附近 (`isnearcrossing`) :

- 如果在交汇口附近:

- `output_str` = "您位于道路交汇路口附近"

- 如果不在交汇口附近:

- 检查是否沿着道路行走 (`isalongroad`) :

- 如果是沿道路行走:

- `output_str` = "识别到道路，您正在沿路方向 30 度内行走"

- 如果不是沿道路行走:

- `output_str` = "识别到道路，您正在偏移道路方向行走"

所有经过 Master 端实时传入 Slave 端的视觉检测信息，以及 Slave 端各传感器实时采集的信息，均通过以上五个逻辑进行文字整理，最终在 Slave 端的语音播报任务 SpeechTask 中处理、打包成 bytestream，写入至 TTS 语音播报模块中，实现实时向视障者的语音交互。

3.5 OpenAMP 跨系统核间通信模块

该模块与前文所述的四个模块不同，前文所述的四个模块主要强调面向用户的功能的具体实现，该模块强调支撑上述用户功能的 Master 端与 Slave 端应用程序利用 OpenAMP 与 rpmsg 核间通信机制进行信息传输的工作。由于此模块起到了衔接的重要作用，因此单独详细阐述。

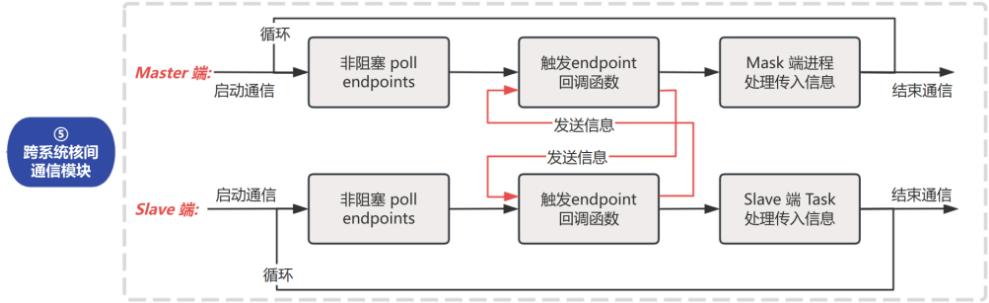


图 44 OpenAMP 跨系统核间通信模块总框图

3.5.1 OpenAMP rpmsg 核间通信机制工作原理概述

一、基本工作原理

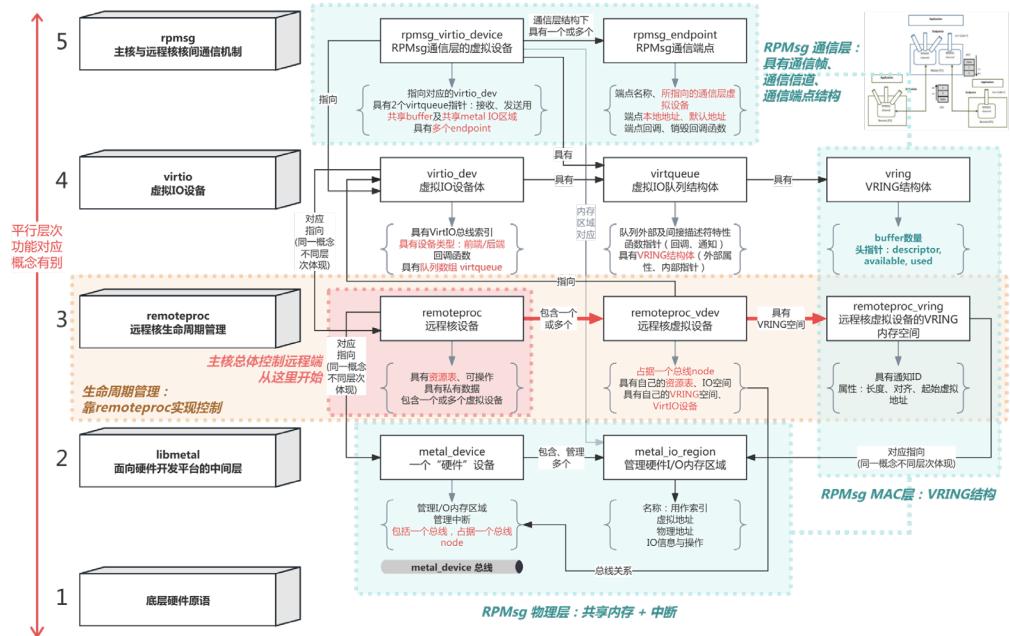


图 45 OpenAMP 在飞腾 phytiump-freeRTOS-sdk SDK 中组件的详细结构图

OpenAMP (Open Asymmetric Multi-Processing) 是一种框架, 用作异构处理器系统中实现处理器之间的通信和资源管理。rpmsg (Remote Processor Messaging) 是 OpenAMP 框架中的一部分, 专门用于不同处理器间的消息传递。OpenAMP 和 rpmsg 的工作原理可以从以下三个方面进行描述:

(一) 平行层次结构

OpenAMP 的体系结构从下到上划分为五个层次, 每个层次负责特定的功能:

1. **底层硬件原语**: 这是最底层的物理硬件部分，包括处理器和通信通道（例如共享内存或串行通信）。
2. **libmetal**: 这是一个面向硬件开发的平台中间层，提供了对底层硬件的抽象和管理，使上层软件能够更方便地与硬件交互。
3. **remoteproc**: 远程核生命周期管理层，负责启动、停止和重启远程处理器。它实现了对从属处理器生命周期的全面控制。
4. **virtio**: 虚拟 I/O 设备层，提供了虚拟化的设备接口，使得处理器之间能够进行标准化的 I/O 操作。
5. **RPMsg**: 主核与远程核的通信机制层，通过 RPMsg 提供消息传递的具体机制，确保不同处理器之间的消息能够高效传递。

(二) RPMsg 各层

RPMsg 通信机制由以下几个层次组成：

1. **物理层 (Physical Layer)** : 在此层次上，RPMsg 利用共享内存或串行通信等物理媒介进行数据传输，确保不同处理器之间的数据包传递。
2. **MAC 层 (MAC Layer)** : 此层负责管理虚拟 I/O 队列 (virtqueue) 和虚拟环 (vring) 的结构，确保数据包在虚拟通道上的可靠传输。它包括对数据包头的描述符、可用性和使用情况的管理。
3. **RPMsg 通信层 (RPMsg Communication Layer)** : 这是 RPMsg 的核心部分，提供了数据包的创建、发送和接收机制。它包含 rpmsg_virtio_device 和 rpmsg_endpoint 等组件，分别负责通信设备和通信端点的管理。

(三) 生命周期管理

生命周期管理是 OpenAMP 框架中的关键部分，通过 remoteproc 组件实现。它负责控制从属处理器的整个生命周期，包括：

启动 (Start) : Master 处理器通过 remoteproc 启动 Slave 处理器，加载其固件并初始化通信通道。

停止 (Stop) : 当需要时，Master 处理器可以通过 remoteproc 停止 Slave 处理器的运行。

重启（Restart）：在特定情况下，Master 处理器可以通过 remoteproc 重新启动 Slave 处理器，重新加载固件和初始化通信。

具体而言，remoteproc 管理包括以下组件：

1. **remoteproc 设备（remoteproc device）**：表示远程处理器设备，管理其资源和操作。
2. **remoteproc 虚拟设备（remoteproc vdev）**：对应于 virtio 设备，管理虚拟 I/O 设备的生命周期。
3. **remoteproc VRING（remoteproc vring）**：管理虚拟 I/O 环的内存空间，确保数据在不同处理器之间的传输。

通过这些组件，Master 处理器可以实现对 Slave 处理器的全面控制和管理，确保处理器之间的协调和高效通信。

二、Master 端启动 Slave 端全流程概述

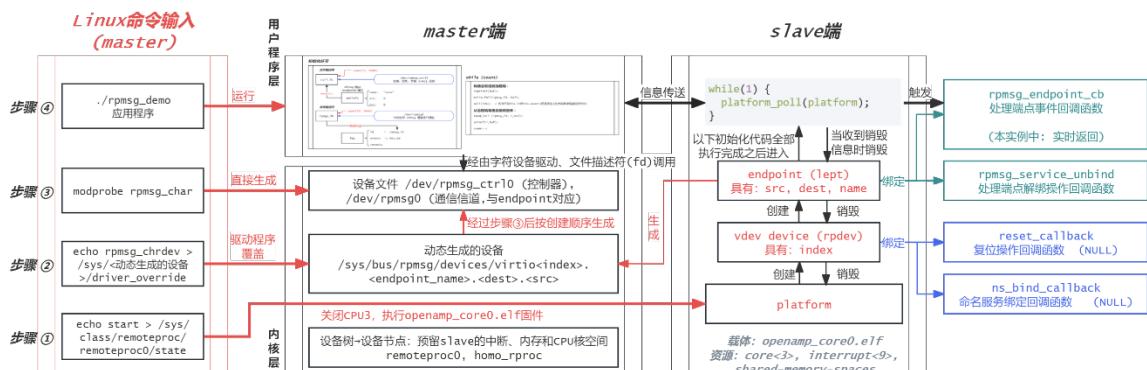


图 46 Master 端启动 Slave 端全过程

在使用飞腾的 E2000Q SoC 平台时，Master 端运行 Linux 系统，Slave 端运行 baremetal（或 FreeRTOS）。在这种配置下，Master 端通过一系列步骤启动 Slave 端系统。以下是该过程的简要解释。

(一) Linux 命令输入

在 Master 端的 Linux 系统中，通过输入一系列命令来启动和配置 Slave 端。这些命令按照顺序执行，确保正确的初始化和通信设置。

步骤 1：启动 remoteproc

```
echo start > /sys/class/remoteproc/remoteproc0/state
```

通过写入“start”命令到 remoteproc 的状态文件，启动 Slave 处理器。这一操作将关闭 CPU3，并执行 **openamp_core0.elf** 固件。

在我们的项目中，remoteproc 在设备树节点中，包括以下资源：

- a) 为 Slave 端分配的 CPU 核心：core<3>
- b) 为 Slave 端分配的核间共享中断：interrupts<9>。

步骤 2：覆盖驱动程序

```
echo rpmsg_chrdev > /sys/<动态生成的设备>/driver_override
```

该命令将 **rpmsg_chrdev** 驱动程序覆盖到动态生成的设备上，准备 rpmsg 字符设备的创建。

步骤 3：加载 rpmsg 字符设备模块

```
modprobe rpmsg
```

使用 **modprobe** 命令加载 rpmsg 字符设备模块，使系统能够识别并使用 rpmsg 字符设备。

步骤 4：运行项目中读取应用程序，启动应用层的通信功能。

(二) Master 端响应

在 Master 端执行上述命令后，系统将作出一系列响应，设置和启动相应的通信设备和端点。

步骤 1：设备文件的创建

设备文件 **/dev/rpmsg_ctrl****（控制器）和 **/dev/rpmsg****（通信信道，与 endpoint 对应）会被创建。它们允许后续的字符设备驱动操作。

步骤 2：动态生成的设备

动态生成的设备位于 `/sys/bus/rpmsg/devices/virtio<index>. <endpoint_name>. <dest> . <src>` 路径下，用于描述和管理具体的 virtio 设备和通信端点。

(三) Slave 端响应

Slave 端在接收到启动命令和初始化代码后，执行相应的初始化和通信设置。

步骤 1：初始化代码执行

Slave 端的初始化代码执行完毕后，进入主循环，等待通信指令。

```
int FRpmmsgCommunication (void) {
    rpmsg_create_ept(); // 创建所有的 4 个端点
    ...
    while(1) {
        PlatformPollTasklette(platform); // 等待通信指令，非阻塞方式
        if (shutdown_req) break; // 当收到关闭请求时退出程序
    }
    rpmsg_destroy_ept(); // 销毁端点
    ...
}
```

通过修改后的 PlatformPollTasklette() 函数，Slave 端等待和处理来自 Master 端的通信请求。

步骤 2：端点和设备创建

在 Slave 端，rpmsg 通信端点（endpoint）和虚拟设备（vdev device）被创建和绑定。每个端点具有源地址、目标地址和名称等属性，虚拟设备具有索引值。

(四) 函数调用和回调

在整个过程中，涉及一系列函数调用和回调函数，以处理通信事件和状态变化：

rpmsg_endpoint_cb: 处理端点事件的回调函数，例如消息接收。

rpmsg_service_unbind: 处理端点解绑操作的回调函数。

reset_callback 和 **ns_bind_callback:** 分别处理复位操作和命名服务绑定操作的回调函数。

3.5.2 Master 端核间通信方式

一、设备树预留资源分配

由于在 Master 与 Slave 系统运行的过程中，需要在运行 Linux 系统的 Master 端为 Slave 端预留资源空间。由设备树中为 homo_rproc 预留的节点数据显示：

预留内存空间地址	0xb010 0000 ~ 0x1990 0000
预留处理器编号	CPU 3
预留共享中断号	<9>

二、用户端 rpmsg 收发接口

核间通信应用模块主要围绕 RPMsg 类对象的方法调用得以实现。RPMsg 类由我们自行设计，这是关于每个面向 Slave 端的 rpmsg 通信链路的一套抽象接口。它调用了 /dev/rpmsg_ctrl* 设备文件，在此节点上，利用下文所述的内核空间驱动程序以创建 /dev/rpmsg_endpoint* 设备文件，并提供了完整的设备文件开启、关闭、读写的抽象方法，便于应用程序的其他部分进行调用与信息互通。

以下是该类中各个方法的功能概述：

- a) **__init__**: 初始化 RPMsg 类，设置控制设备路径、端点设备路径、端点信息（名称、源地址、目标地址）以及相关的宏定义。
- b) **openCtrlDevice**: 打开 rpmsg 控制设备，获取文件描述符。
- c) **createEndpoint**: 创建通信端点，如果设备文件不存在，则使用 ioctl 系统调用创建。
- d) **openEptDevice**: 打开 rpmsg 端点设备，获取文件描述符并注册用于轮询的 poll 对象。
- e) **writeEptDevice**: 向 rpmsg 端点设备写入数据。
- f) **pollEptDeviceWithReadEvent**: 轮询端点设备的事件，等待数据读取事件的发生。
- g) **readEptDevice**: 从 rpmsg 端点设备读取数据。

h) **closeEptDevice**: 关闭 rpmsg 端点设备的文件描述符。

i) **closeCtrlDevice**: 关闭 rpmsg 控制设备的文件描述符。

在 Master 端应用程序进行核间通信时，首先需要通过 `openCtrlDevice` 打开对应 `endpoint` 的控制设备，并在 `/dev` 中利用 `createEndpoint` 创建这个设备相对应的通信端点，并通过 `openEptDevice` 打开。在进行通信时，需要通过 `pollEptDeviceWithReadEvent` 不断轮询端点设备的事件，再从 Slave 端读取数据、写入数据。在完成所有的通信过程后，通过 `closeEptDevice` 与 `closeCtrlDevice` 关闭。

3. 5. 3 Slave 端核间通信方式

一、FreeRTOS 中分别为核间通信与各传感器处理预留单独的 Task

在 FreeRTOS 中，我们为 OpenAMP 通信预留了一个独立的任务，该任务主要负责处理核间通信，而其他任务则用于处理不同的传感器或语音数据。在项目中，创建 **OpenAMPTask** 任务用于处理 OpenAMP 的通信，而剩余的任务用于处理通信以外 Slave 端的工作：**SpeechTask** 用于获取语音逻辑并输出语音信息；**DHT11Task** 和 **GPSTask** 分别用于读取和处理 DHT11 传感器和 GPS 传感器的数据；**Jy61pTask** 用于处理陀螺仪传感器的数据。这种任务分配方式保证了各个模块的独立性和实时性。

二、初始化每个 endpoint 的过程

初始化每个 endpoint 涉及以下几个步骤：

1. **平台初始化**: 通过 `platform_init` 函数初始化通信平台，设置必要的资源。
2. **创建虚拟设备**: 使用 `platform_create_rpmsg_vdev` 函数创建 rpmsg 虚拟设备（virtio 设备），该设备负责处理实际的消息传递。
3. **创建端点**: 通过 `rpmsg_create_ept` 函数为每个传感器和模块创建相应的通信端点。每个端点包含名称、源地址、目标地址、消息处理回调函数和解绑回调函数。

这些步骤确保了每个端点可以正确地发送和接收消息，并且在需要时可以被安全地销毁。

三、OpenAMPTask 轮询并响应回调函数，各传感器 Task 响应并处理相关任务

该过程涉及到以下几个步骤，便于实现 OpenAMPTask 在核间通信过程中的高效处理：

1. **轮询事件：**在 OpenAMPTask 主循环中，使用 **PlatformPollTasklette** 函数定时轮询 endpoint 事件，检查是否有新的数据到达。轮询间隔由 FreeRTOS 的时间片机制，将 CPU 占用交由其他 task 处理。
2. **处理事件：**当 OpenAMPTask 检测到 endpoint 事件时，调用相应的回调函数，首先识别所接收到的数据应当交由哪个负责传感器的 task 处理，通过任务间通信机制，将对应传感器的 task 内置的新数据 flag 置位；然后接收数据包，并实时发送到对应传感器的 task。
3. **传感器任务响应：**各传感器 task 在循环过程中，判断新数据 flag 的置位情况。若该位被置位，则检测并处理从 OpenAMPTask 传来的数据包，实现 Slave 端的应用功能和传感器数据处理功能。

4 主要创新点

4.1 人行楼梯级数检测算法的迭代

1. 改进前：

首先通过目标检测网络识别到楼梯，接着在楼梯的目标框内，以楼梯目标框的中点为中心，上至目标框顶端，下至目标框底端，绘制平行于 y 轴的线段。接着，以线段每个 y 轴点为中心，x 轴左右 5 个像素点（每个 y 轴点对应 11 个 x 轴像素点）取 R、G、B 三个通道值的亮度值均值；最后对目标框内的每个 y 轴点，沿着 y 轴的正方向，绘制 R、G、B 三个通道的均值的分布图，并进行波峰提取，以获得楼梯的级数。

下图为改进前楼梯检测算法波峰提取环节的效果图。



图 47 改进前楼梯检测算法波峰提取环节效果图

改进前算法的缺点有三点：

- 不同亮度、不同纹理的楼梯，R、G、B 三个通道的亮度值的波峰形态（包括波峰 - 波谷高度差、绝对高度等）各异，这为复杂状况下滤波的阈值设置造成了较大困难；**
- 由于采样线段的方向固定为平行于 y 轴的方向，在侧视楼梯的视角下，楼梯级数判断效果不佳。**
- 数据量过大，性能低。**

2. 改进后：

同样地，我们通过目标检测网络识别到楼梯，而所改进的步骤在于，我们基于目标框框定的范围进行图像变换及轮廓线条检测。轮廓线条检测相比于 RGB 通道检测的方法，对于不同亮度和纹理的楼梯画面的适应性更优；通过对轮廓线条进行斜率 k 频率分布分析，可以改善侧视楼梯视角下的识别效果。



图 48 改进后楼梯检测算法效果图

改进前后关键步骤对比如下：

序号	改进前	改进后
1	目标框内线段提取	目标框内线段提取
2	根据线条斜率均值 k 取垂线，判断楼梯走向	分析斜率 k 直方图，提取 $k \approx 0$ 波峰附近线段
3	过框中点沿走向作线段 1，每 y 值左右 11 像素点平均采样，利用 RGB 通道亮度值直方图波峰分析	根据经验值设置阈值，分析截距 b 直方图峰数

改进前后运行时间测试及时间复杂度分析将在第 5 章详细描述。

4.2 基于 Multiprocessing 的多进程并行框架的实现

相对于单进程的方法，通过将不同的任务分配给不同的进程，可以同时执行图像采集、边缘检测、目标检测等多个任务，提高了系统的处理效率。设计过程中，我们尝试了两种实现方法：

1. 改进前：基于 QThread 实现

为了和 PySide6 框架通信，系统将基于 Multiprocessing 实现的并行机制迁移至 Qt 框架下的 QThread 实现。程序将视频读取、目标网络推理和检测结果再处理分别置于三个独立线程下进行，通过 PySide 的信号与槽机制在不同线程间通信。当视频读取线程开启时，程序将读入的帧通过发送一个 `np.ndarray` 类型的信号将捕获到的帧传递至网络推理线程。网络推理线程接收到帧后将其放入栈进行缓存，推理时取出栈顶的帧，利用栈的后进先出（LIFO）特性，以保证处理的是最新收到的视频帧，使得输出画面和实时画面同步。

优点：

- a. 轻量级：QThread 是 Qt 框架提供的线程类，相对于操作系统的原生线程来说，它更轻量级，创建和销毁的开销较小。
- b. 方便事件循环：QThread 内置了事件循环，可以方便地处理事件和信号，与 PySide 框架的其他组件集成更加方便。

c. 易于 UI 编程：对于 GUI 应用程序，使用 QThread 可以更方便地与 PySide6 的 UI 组件进行交互，因为它们都在同一个事件循环中。

缺点：

a. 共享数据复杂：如果多个线程需要共享数据，需要使用互斥锁等机制来保证数据的一致性，否则可能会出现竞态条件等问题。

b. 受 GIL 限制：尽管 QThread 是 C++ 实现的，但是在 Python 中使用时仍受到全局解释器锁的限制，Python 中的全局解释器锁限制了多线程并发执行 Python 代码的效果，因此对于 CPU 计算密集型任务，多线程模型并不会带来真正的并行处理。

在我们的项目中，使用基于 QThread 的实现的最大缺陷在于，QThread 所受到的全局解释器锁限制，无法实现在目标检测推理过程中 OpenCV 处理及高帧率线程中图像帧的实时输出的环节。为了实现在网络推理的同时能够实现并行的 OpenCV 处理与高帧率线程中图像帧的实时输出，我们基于 Multiprocessing 对机器视觉处理模块的实现进行代码适配与改写。

2. 改进后：基于 Multiprocessing 实现

程序通过一个主进程进行视频流读取，将读取的视频帧发送至子进程，从图像队列中获取帧，进行目标检测和推理处理，然后将检测结果放入另一个队列中缓存，以供使用。当存放结果的队列非空时，主进程从缓存队列中取出检测结果，针对目标框做进一步处理。在目标检测的结果中，程序对楼梯处理得阶梯数量，对斑马线处理得走向信息，对红绿灯处理得颜色信息。并将结果输出到视频流。

优点：

a. 真正的并行处理：使用多进程可以在多个 CPU 核心上同时执行任务，从而实现真正的并行处理，提高了系统的利用率。且考虑网络推理需要占用较多的 CPU 资源，此方法下的并行处理可以根据不同进程的任务量对不同进程做资源分配，使得最终的显示结果和实时的视频达到画面同步。

b. 资源隔离：多进程的方法可以做到资源隔离，每个进程拥有独立的内存空间和资源，相互之间不会产生影响，提高了系统的稳定性和可靠性。

c. 更适合 CPU 密集型任务：对于 CPU 密集型的任务，多进程可以更有效地利用系统资源，因为它们在不同的进程中运行，不会受到 GIL（全局解释器锁）的限制。

缺点：

- a. 资源消耗较多：每个进程都有独立的内存空间和系统资源，因此多进程模型消耗更多的内存和系统资源。
- b. 启动和销毁进程耗时：创建和销毁进程的开销相对较大，因此需要频繁启动和销毁的任务时，可能会影响性能。

图 49 为 Multiprocessing 框架下，同时运行目标检测和纯 OpenCV 算法的线程中，并行帧率所满足的矛盾关系。此图的硬件背景是 SoC 为 E2000Q 的飞腾派，CPU0~2 开启，CPU3 关闭，推理网络使用自行训练的 YOLOv8n 网络（Float16 量化，MNN 格式）。

由于目标检测进程循环与纯 CV 算法与渲染进程循环的时间较固定，因此通过调节目标检测循环后的延时，测试两者帧率的差异。结果显示，为了保证 CV&显示进程达到准实时显示级别，即 10FPS，同时并行运行的目标检测推理进程帧率设置为 0.4FPS，可以达到较平衡的效果。而实际测试中，CV&显示进程为 6FPS，目标检测推理进程为 0.7FPS。

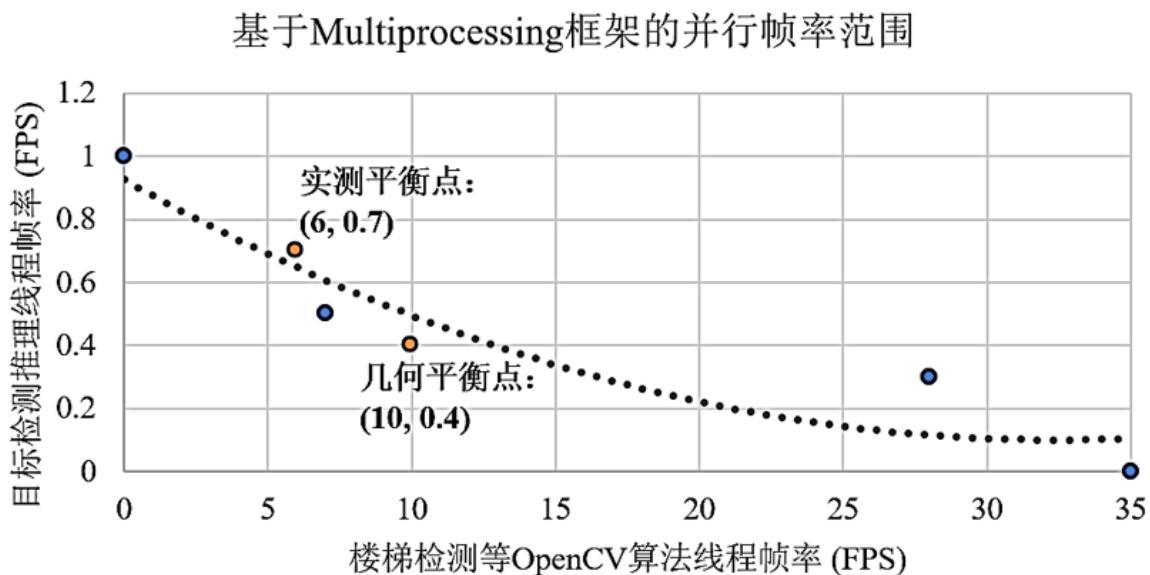


图 49 Multiprocessing 框架下并行帧率的矛盾关系与最优平衡点示意

4.3 Master 端 Linux 内核的 rpmsg 多静态端点实现多通道并行核间通信

相比于飞腾官方例程中提供的 Linux 内核中的 rpmsg 字符设备驱动程序，我们在此基础上，修复了多个 rpmsg 终端生成时，由于设备节点/dev/rpmsg_ctrl*与 /dev/rpmsg*名称生成规律造成设备读写混乱的问题。与此同时，我们结合了重新编译的 Linux 5.10 内核，实现了一个修改后的 rpmsg 字符设备驱动程序版本。

具体而言，我们根据经验，设定每一个 rpmsg_ctrl 设备文件对应唯一一个 rpmsg_endpoint 终端设备文件，因此在命名上，将编号*，由内核驱动自动分配的设备号决定，修改为根据/sys/bus/rpmsg/devices 的每个终端的 src 编号决定，从而解决了这一设备读写混乱的问题，实现了在 Master 端与 Slave 端间多个 endpoint 的并行通信。最终，每一个 endpoint 恰好与 Slave 端的每一个传感器或模块对应，既能实现高效通信的同时，也能保证传感器与模块的数据读写不发生混乱或错误。

修改内核驱动程序 /drivers/rpmsg/rpmsg.c 如下：

函数 rpmsg_eptdev_create():

```
static int rpmsg_eptdev_create(struct rpmsg_ctrldev *ctrldev,
struct rpmsg_channel_info chinfo)
{
    ...
    u32 destnum; //用于存储终端的 src 编号
    ...
    destnum = ctrldev->rpdev->dst;
    dev_set_name(dev, "rpmsg_%d", destnum); //设置名称
    ...
}
```

函数 rpmsg_chrdev_probe():

```
static int rpmsg_chrdev_probe(struct rpmsg_device *rpdev)
{
    ...
    u32 destnum; //用于存储终端的 src 编号
    ...
    destnum = rpdev->dst;
    dev_set_name(&ctrldev->dev, "rpmsg_ctrl_%d", destnum); //设置名称
    ...
}
```

通过这种设置方法，可以使得 rpmsg_ctrl 节点名称与 rpmsg 节点名称一一对应，且与 endpoint 的 src 字段绑定，可以避免终端编号混乱所致的节点读写错误问题。

4.4 Slave 端非阻塞 platform_poll 的实现

由于飞腾 phytiun-freertos-sdk 中 /third-party/openamp/ports/platform_info.c 的轮询机制只有阻塞实现，实际运行时也只能在 baremetal 环境中运行，而无法在具有 FreeRTOS 的环境下运行，这是因为它会阻塞 FreeRTOS 的任务调度机制。修改后的 platform_poll 程序如下所示：

```
int PlatformPollTasklette(void *priv)
{
    struct remoteproc *rproc = priv;      //传参传入的是 platform, remoteproc 平台的指针
    struct remoteproc_priv *prproc;
    unsigned int flags;
    int ret;

    prproc = rproc->priv;
    while (1) {
        if (metal_io_read32(prproc->kick_io, 0) & 0x2) { //RPROC_M2S_SHIFT
            ret = remoteproc_get_notification(rproc, RSC_NOTIFY_ID_ANY);
            if (ret) {
                return ret;
            }
            break;
        }
        (void)flags;
        vTaskDelay(100 / portTICK_PERIOD_MS); //延时 100ms, 这个需放在 while(1)
    内部
    }
    return 0;
}
```

在增加了 FreeRTOS 下的延时机制后，可实现轮询机制在多 Task 非阻塞的环境下执行，也保障了 Slave 端通过 FreeRTOS 得以实现多任务运行。

4.5 自主训练目标检测网络设计与评价

4.5.1 数据集设计

一、数据集原始采集

针对目标检测网络在实际场景下的适用性，我们在广州市海珠区赤岗街道的多条市政道路人行道与路口中拍摄自己的原始数据集，数据集大小为 36.4GB。我们使用这一数据集标注得到的数据进行 YOLOv8n 网络的训练。

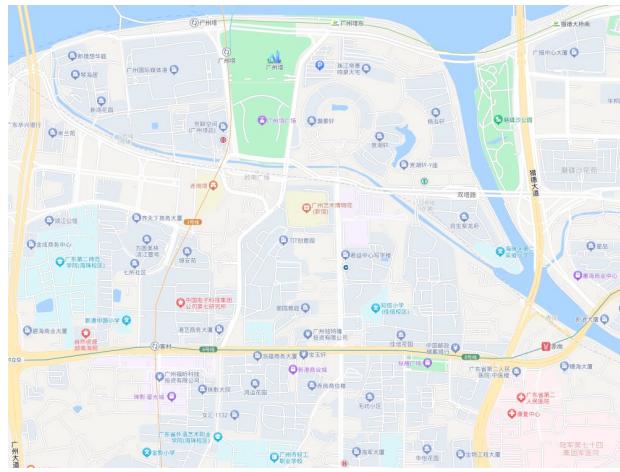


图 50 数据集采集地点（广州市海珠区赤岗街道）附近街道图示

二、利用预训练模型进行预标注

为了提高标注效率，我们使用了基于在 OpenImagesv7 数据集上的 YOLOv8x 预训练模型，针对原始数据集进行部分标签的预标注，包括 Bench、Person 等。这一预标注过程在服务器主机中运行，使用 YOLOv8x 对原始数据进行推理，得到推理后的结果。这些结果将会作为下述步骤的 Ground Truth 数据的一部分。

三、添加符合项目应用需求的标签

在此基础上，我们在数据集中添加了预训练模型中所没有的、而符合我们应用需求的 Barrier、Cabinet 等标签，使得检测出的目标更符合视障人士的避障需求。添加后的标签通过 Python 的 fiftyone 工具进行数据集管理，以便提高效率。

四、数据增强

通过 imgaug 方法，对模型进行了 5 种不同方式的数据增强。数据增强方法如下所示：

```

seq = iaa.Sequential([
    iaa.Flipud(0.5), # 垂直翻转
    iaa.Fliplr(0.5), # 水平翻转
    iaa.Multiply((1.2, 1.5)), # 更改亮度
    iaa.GaussianBlur(sigma=(0, 3.0)), # 高斯模糊
    iaa.Affine(
        translate_px={"x": 15, "y": 15},
        scale=(0.8, 0.95),
        rotate=(-30, 30)
    ) # 缩放并旋转
])

```

标注生成的数据集被用于 YOLOv8n 模型训练，筛选后的总数据量为 4800 张图片，共 8 万个 labels。

4.5.2 模型训练结果

一、关键指标评价

模型在装载有 NVIDIA RTX 3060 显卡的服务器上，经过 175 epoch 轮次训练，`imgsz` 为 640×640 ，得到以下结果，如下图所示：

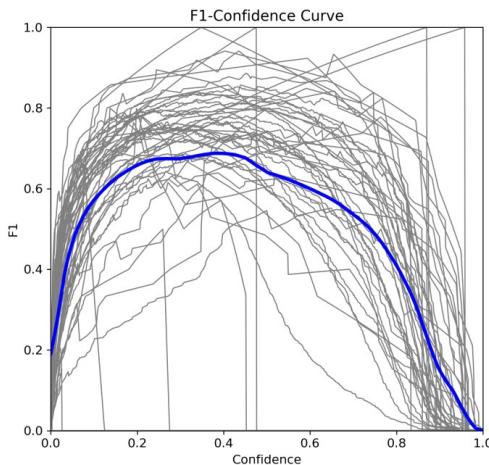


图 51 F1 曲线结果

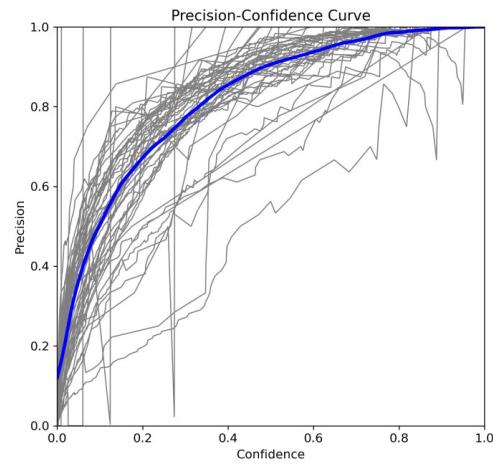


图 52 P-置信度曲线结果

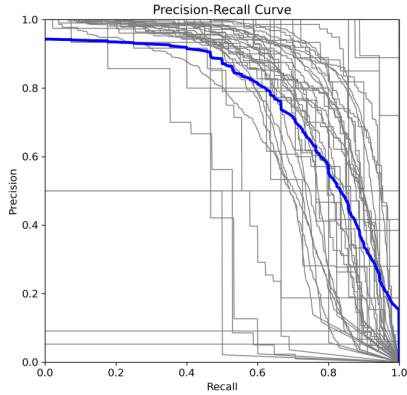


图 53 P-R 曲线结果

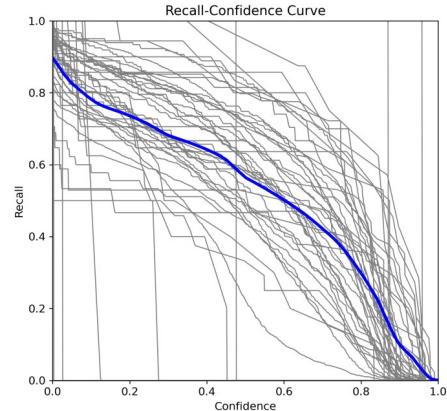


图 54 R-置信度曲线结果

以下通过几个关键指标衡量模型的训练结果：

- **精确率 (Precision, P)**：也称为“查准率”，它表示正确检测结果在所有检测结果中的百分比。换句话说，精确率是模型在所有预测为正的样本中，实际为正样本的比例。
- **召回率 (Recall, R)**：也称为“查全率”，它表示正确检测结果在所有实际正样本中的百分比。即召回率是模型在所有实际为正的样本中，成功预测为正样本的比例。

而 F1 曲线是衡量目标检测结果的重要指标，它由精确率 (Precision) 和召回率 (Recall) 的调和平均数构成。

- **F1 分数 (F1 Score)**：是精确率和召回率的调和平均数，计算公式为：

$$F1 = \frac{2 \times P \times R}{P + R} \quad (36)$$

由上述图像可知，图像的左侧部分：偏向于显示精确率 (P) 的情况；而图像的右侧部分：偏向于显示召回率 (R) 的情况。可以观察到，图像顶部偏右部分表现出明显的倾斜，这表明在这些区域内，召回率相较于精确率效果较差。

然而，由于不同类别 (class) 的训练数据样本数量存在较大差异，因此实际应用中，主要的、起到关键作用的障碍物的平均 F1 值会更高。这意味着尽管在某些类别上的召回率表现不佳，但整体系统在检测关键障碍物时仍具有较高的可靠性和有效性。

二、数据集评价

在训练过程中，我们设置 14% 的数据作为测试集，86% 的数据作为训练集。

以下是标准化后的混淆矩阵：

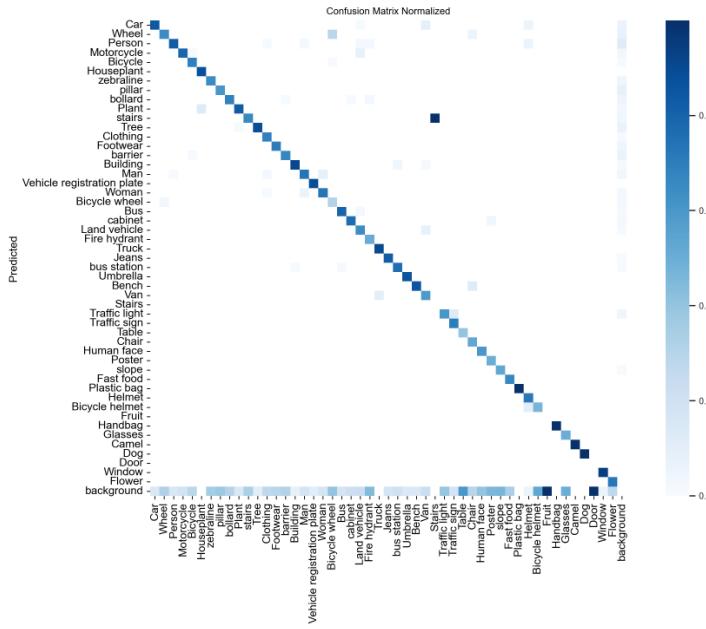


图 55 标准化后的混淆矩阵

上述矩阵显示，绝大多数关键的 class，如 Person, Bicycle, Motorcycle 等，TP 值的大小均超过 0.8；而有部分数据集较少的 class，TP 值大小稍低，但总体均超过 0.6，显示了模型较高的预测结果。而值得注意的是，Stairs 类别与 stairs 类别的混淆矩阵发生异常，由名称“Stairs”与“stairs”可知，由于数据集名称的处理问题，所有 stairs 类别均被识别为 Stairs 类别，但这并未对模型的效果造成显著影响。

以下为典型的测试集的标注结果与训练结果：



图 56 测试集标注结果



图 57 测试集测试结果

由上述结果可知，该模型在复杂路况下的识别能力较高，即使不少 class 的目标框置信度较低，但仍能准确识别到复杂路况下画面中的所有障碍物等关键目标。

5 测试与验证

5.1 功能运行测试

5.1.1 测试场景

如下图所示，实地测试环节位于广州市海珠区赤岗街道的多条道路与十字路口，时间位于工作日（周一）下午 16:30~17:30，以模拟交通繁忙的城市道路场景。

由于需要实时观察用户端 GUI 界面的结果，因此整个测试过程需要 2 人配合完成，一人手持设备模拟视障人士行走场景，摄像头高度与正常成人高度相同；另一人实时通过显示屏观察监护人界面显示效果。

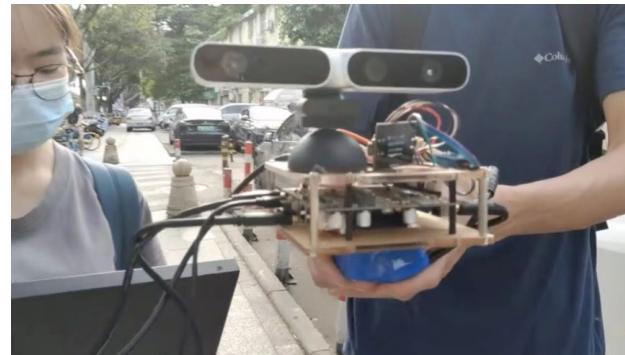




图 58~59 测试场景

5.1.2 人行道模式测试结果

一、一般人行道场景

地点：广州市海珠区新港中路北侧，西往东方向。

时间：工作日（周一）下午 16: 40

图示：



图 60 一般人行道场景

系统性能：

CPU0, 1, 2 占用率	单帧平均循环时间	Master 与 Slave 通信响应时间
97.0%, 99.0%, 100.0%	0.103s	0.09ms

播报完整语音：

- “前方宽松，中距多障碍”
- “11, 0, 2, 0, 1”（正前方物体从近到远编号对应：树木，小轿车，行人，小轿车，车轮）

没有提示视角异常、摔倒异常。

偏离道路方向位于沿道路行走模式，默认不播报。

二、无交通灯支路路口场景

地点：广州市海珠区新港中路北侧，西往东方向。

时间：工作日（周一）下午 16: 45

图示：



图 61 无交通灯支路路口场景

系统性能：

CPU0, 1, 2 占用率	Master 与 Slave 通信响应时间
97.0%, 99.0%, 100.0%	0.09ms

播报完整语音：

- “前方宽松，近距多障碍”
- “6, 8, 8, 0, 9”（正前方物体从近到远编号对应：斑马线，短桩，短桩，小轿车，绿化隔离带）
- “识别到 1 条斑马线，脚下斑马线在您右侧，指向右前方”

没有提示视角异常、摔倒异常。

偏离道路方向位于沿道路行走模式，默认不播报。

5.1.3 路口模式测试结果

一、测试场景 1

地点：广州市海珠区新港中路与赤岗北路十字路口西北侧，西北往东南方向

时间：工作日（周一）下午 17: 00，交通高峰期

图示：（调试用 GUI 界面显示结果，左侧界面仅显示斑马线、交通灯结果）



图 62 正常过马路场景（场景 1）

播报完整语音：

- “前方宽松，近距多障碍”
- “6, 3, 1, 3, 18, 8”（正前方物体从近到远编号对应：斑马线，摩托车(左前方私人车辆)，车轮，摩托车(右前方外卖车辆)，行人，短桩）
- “识别到 1 条斑马线，脚下斑马线在您右侧，指向左前方”
- “识别到 3 个交通灯，最近距离交通灯为红灯”

没有提示视角异常、摔倒异常。

偏离道路方向位于路口模式，默认不播报。

二、测试场景 2

地点：广州市海珠区新港中路与赤岗北路十字路口西北侧，西往东方向

时间：工作日（周一）下午 17: 05，交通高峰期

图示：



图 63 正常过马路场景（场景 2）

系统性能：

CPU0, 1, 2 占用率	单帧平均循环时间	Master 与 Slave 通信响应时间
97.0%, 99.0%, 100.0%	0.78s	0.09ms

播报完整语音：

- “前方拥挤，远距多障碍”
- “6, 6, 1, 3, 13, 11”（正前方物体从近到远编号对应：斑马线，斑马线，车轮，摩托车，鞋子，树木）
- “识别到 2 条斑马线，脚下斑马线在您左侧，指向左前方”
- “识别到 4 个交通灯，最近距离交通灯为绿灯”

没有提示视角异常、摔倒异常。

偏离道路方向位于路口模式，默认不播报。

5.1.4 人行天桥模式测试结果

一、人行天桥（上楼梯）

地点：广州市海珠区新港中路与赤岗北路十字路口东北侧，正东方向

时间：工作日（周一）下午 17: 15

图示：（调试用 GUI 界面显示结果，左侧界面仅显示楼梯结果）



图 64 人行天桥（上楼梯）

播报完整语音：

- “前方宽松，近距多障碍”
- “10, 3, 13, 13, 18”（正前方物体从近到远编号对应：楼梯，摩托车，鞋子，鞋子，行人）
- “识别到 1 个阶梯，在您右侧，阶梯数 21”

没有提示视角异常、摔倒异常。

偏离道路方向位于路口模式，默认不播报。

二、人行天桥（下楼梯）

地点： 广州市海珠区新港中路人行天桥，南侧下天桥口

时间： 工作日（周一）下午 17: 20

图示：



图 65 人行天桥（下楼梯）

此时成功识别到该段楼梯级数为 14。

5.1.5 未沿道路方向行走功能测试结果

地点： 由于赤岗路沿线附近车辆环境复杂，易发生交通事故，该场景测试在较安全的无车双向二线内部道路中完成。

时间： 工作日（周三）下午 14: 30

当视障者在远离路口的街道上偏离道路方向 $\pm 30^\circ$ 角度行走时，在每轮循环中播报声音“偏移道路”，以提醒视障者调整方向，回到沿道路行走的状态。

播报完整语音： “偏移道路”。

图示：



图 66 未沿道路方向行走

5.2 性能验证与分析

5.2.1 目标检测量化耗时

我们使用了自行训练的 YOLOv8n 目标检测模型，分别使用 Float16 量化与 Int8 量化的 ONNX 模型进行推理，耗时结果如下表所示。

初始状态：使用 FLOAT16 量化，ONNX 格式：

模块（单帧图像）	执行时间 (ms)
目标检测	1775 (仅使用 CPU0~2, 共 3 个 CPU)
楼梯级数识别*	5.05e-2
红绿灯识别	4.17
斑马线识别	29.57

控制变量 1：使用 INT8 量化，保持使用 ONNX 格式：

模块（单帧图像）	执行时间 (ms)
目标检测	375 (仅使用 CPU0~2, 共 3 个 CPU)
楼梯级数识别*	5.25e-2
红绿灯识别	4.24

斑马线识别	30.70
-------	-------

由上述表格可知，使用 INT8 进行量化后，目标检测执行时间大幅下降，使空闲时间增多，也引致其他模块的平均帧率同步上升。

这一差异主要源于 INT8 与 FLOAT16 在计算效率和内存带宽占用上的本质区别。INT8 量化将模型权重和激活值压缩到 8 位整数，这大幅减少了计算复杂度和内存访问开销。由于 INT8 数据占用的内存带宽仅为 FLOAT16 的 1/2，每个内存单元可以存储更多的数据，从而提高了缓存命中率并减少了内存瓶颈。

相比之下，FLOAT16 尽管也减少了浮点数的精度，但其计算复杂度仍高于整数运算，且内存带宽需求仍较高。结果是，在相同硬件条件下，INT8 量化模型在计算和数据传输方面都更高效，从而使得其执行时间几乎是 FLOAT16 量化的 1/3。

另外，在将模型量化至 INT8 的过程中，在训练后量化（Post-Training Quantization, PTQ）方法与量化感知训练（Quantization-Aware Training, QAT）方法中，我们选择了 QAT 方法处理。这个方法相比于 PTQ 方法直接对推理时的浮点数转换为更低精度的整数，针对我们的自有数据集，在模型训练阶段中即已将量化过程引入到训练中，使得模型在训练时“感知”到量化的影响，其权重和激活值会模拟量化的效果，从而使得模型能够在保持更高精度的同时，享有与 PTQ 方法相似的推理速度与资源占用。

控制变量 2：使用 MNN 格式，保持使用 FLOAT16 量化：

模块（单帧图像）	执行时间 (ms)
目标检测	768 (仅使用 CPU0~2，共 3 个 CPU)
楼梯级数识别*	5.05e-2
红绿灯识别	4.22
斑马线识别	32.13

由上述表格可知，使用 MNN 格式的目标检测时间相比于 ONNX 格式下降一半以上，但由于仍然保持使用 FLOAT16 量化，相比于 ONNX 格式的 INT8 量化的目标检测时间增加一倍。这与 MNN 针对 64 位 ARMv8 处理器的特定优化密切相关。

首先，MNN 专为移动设备和嵌入式系统设计，特别针对飞腾派的 ARMv8 架构进行了深入优化。它充分利用了 ARMv8 的 64 位特性和高级矢量扩展（NEON）指令集，这使得 MNN 能够更高效地执行并行计算任务。通过利用 NEON 指令集，MNN 在处理 Float16 数据时实现了更高的计算吞吐量和更低的延迟，从而显著加快了推理速度。

其次，MNN 对 ARMv8 平台的内存管理进行了专门优化。ARMv8 处理器的内存层次结构和缓存系统被 MNN 充分利用，通过优化数据加载和缓存策略，减少了内存带宽的占用和缓存未命中率。这种针对性的内存优化有助于加快处理速度，特别是在处理高维度的神经网络模型时。

此外，MNN 在 ARMv8 平台上对常用的深度学习操作符进行了高度优化。这些操作符被重新实现，以更好地匹配 ARMv8 处理器的指令集和计算单元。相比之下，ONNX Runtime 在 ARMv8 平台上的优化更为通用，可能无法充分发挥出 ARMv8 架构的全部潜力。

最后，MNN 的计算图优化策略也充分考虑了 ARMv8 平台的特性。通过子图融合、操作符重排序等技术，MNN 能够在 ARMv8 平台上以更高的效率执行神经网络推理，进一步缩短了推理时间。

*注意：所有测试环节均在楼梯级数识别环节使用改进后的楼梯级数识别算法。

5.2.2 楼梯级数识别算法耗时分析

一、耗时结果

项目	执行时间 (ms)
改进前：楼梯走向 + RGB 直方图法	77.8
改进后：线段 k, b 直方图波峰分析法	5.05e-2

表格显示，楼梯级数识别算法的执行时间在执行前与执行后的执行时间大幅减少，共减少了 4 个数量级。

二、改进前后时间复杂度对比

由于改进前后均使用 Canny 算子进行边缘检测，以及 Hough 变换提取线段，设楼梯目标框横轴像素数为 x ，纵轴像素数为 y ，所提取出的线段数为 n 。因此有：

1. 改进前后共同的时间复杂度：

提取线段环节： $O(2xy)$

其中，Canny 算子边缘检测耗时 $O(xy)$ ，而 Hough 变换提取线段耗时 $O(xy)$ 。两种算法均循环对目标框的每一个像素点进行扫描，因此各有 $x \times y$ 的耗时结果。而 Canny 算子与 Hough 变换在 OpenCV 库中使用优化的数据结构与运算方法进行处理，因此耗时极短。

2. 改进前算法独有：

第 2 步骤通过获得所有线段斜率的均值，并求其垂直线，以获取楼梯的大致走向： $O(n)$ 。

第 3 步骤使用了 RGB 直方图分析方法： $O(y \times (4x + 3xy)) = O(4xy + 3xy^2)$

其中，第一重循环对每个 y 值遍历： $O(y)$ ；第二重循环内包括两项：对 x 值的筛选，以及 R、G、B 共 3 个通道的循环，总共四重对 x 值的循环： $O(4x)$ 。再加上滤波前对平均亮度的计算过程，针对目标框内每个像素点完成 R、G、B 共 3 个通道的循环： $O(3xy)$ 。

3. 改进后算法独有：改进后算法的第 2、3 步骤使用了对 k, b 直方图的波峰分析： $O(2n)$ 。其中，分别对线段进行两次直方图分析，各 $O(n)$ 。

4. 共计对比：

改进前： $O(2xy + 4xy + 3xy^2 + n) = O(6xy + 3xy^2 + n)$

改进后： $O(2xy + 2n)$

三、对比：斑马线识别算法的时间复杂度

斑马线识别算法的时间复杂度为： $O(n^2 + 2n)$ 。

其中， $O(n^2)$ 用于计算所有线段间的距离，以便筛选过于密集的线段；而 $O(2n)$ 分别用作中点计算（ $O(n)$ ），以及保留筛选后的稀疏线段（此过程需要遍历，因此耗时 $O(n)$ ）。

5.2.3 传感器数据采集耗时

1. 耗时测量结果：

传感器	有效数据个数	平均采集时间(ms)
DHT11 温湿度传感器	147	23.63
GPS 模块	147	924.5
JY61P 三轴陀螺仪传感器	185	983.86

2. 测量方法：

在从核使用 `xTaskGetTickCount()` 函数读取 SysTick，将采集前和采集后的计数值相减，得到采集数据所需要的时间，多次采样后取平均值。

5.2.4 rpmsg 核间通信耗时

1. 耗时测量结果：

项目	执行时间(μs)
Master 端向 Slave 端写入数据 (4 个 endpoint，并行执行)	135
Master 端从 Slave 端获取数据 (4 个 endpoint，并行执行)	390
Master 端向 Slave 端写入数据 (1 个 endpoint，等效串行执行任务)	136

2. 测量方法代码实现（以单个 endpoint 为例）：

a. 测试脚本主函数：

```
if __name__ == "__main__":
    time.sleep(1) # 睡眠 1 秒
    write_start=time.time()
    rpmsg0.writeEptDevice1() # 将已有的数据写入到端点里面去
    write_end=time.time()
    print("write time:",write_end-write_start)
    #rpmsg0.writeEptDevice()
    print("[OPENAMP] WAIT data ")
    rpmsg0.pollEptDeviceWithReadEvent()
    time.sleep(1) # 睡眠 1 秒
    print("[OPENAMP] read data ")
    # 以下是数据解析
    read_start=time.time()
    rpmsg0data = rpmsg0.readEptDevice()
    read_end=time.time()
    print("read time:",read_end-read_start)
```

b. 测试 Master 端向 Slave 端写入函数：

```
def writeEptDevice1(self):
    # todo: demo only, 后续需要加入业务代码, 如 JSON 之类的
    string =
"aa070000001a64015b01003311fa010b1a6901740101180b12011b1afa000e01081d21f9fe1b1
aac0100021500056021b1a7d018801013f0774011b1a1f012501" # 测试使用
    buf0 = string.encode('ascii') # 为字符串编码为 ASCII
    buf0 = buf0.ljust(32, b'\x00') # 确保不超过 32 字节
    # 开始写入数据
    self.ret = os.write(self.eptfd, buf0)
    if self.ret < 0:
        print("[OPENAMP] Write endpoint '%s' failed" % self.ept_device_path)
```

c. 测试 Master 端从 Slave 端获取函数：

```
def readEptDevice(self):
```

```

if not self.eptpollevents: # 没有初始化
    print("[OPENAMP] Endpoint eptpoller not initialized")
    return None

else: # POLLIN 事件发生了，意味着可以读取数据了
    for _, event in self.eptpollevents:
        if event & select.POLLIN:
            try:
                while True:
                    self.r_buf0 = os.read(self.eptfd, 32)
                    if len(self.r_buf0) == 0:
                        print("[OPENAMP] No data read from %s" %
self.ept_device_path)
                    else:
                        print("[OPENAMP] Read data: {self.r_buf0.hex()}")
                    return self.r_buf0.hex() # 将数据以 16 进制的形式返回
            except OSError as e:
                print(f"[OPENAMP] Reading failed: {e}")
            else:
                # todo 后续如果需要增加更多的那就加入
                return None

```

3. 解析：

在实际系统中，我们使用了 4 个 endpoint 并行通信，每个 endpoint 对应一个传感器或语音输出模块。尽管有多个 endpoint 同时进行通信，但得益于 OpenAMP rpmsg 的高效设计和良好的资源管理，由上述耗时测量结果对比表明，多个 endpoint 并行下的通信耗时相比于单个 endpoint 通信耗时，在写入数据过程中基本一致，而在读取数据过程中大幅减小。这意味着，在并行工作情况下，系统可以同时处理多个数据传输任务，不仅不会显著增加通信延迟，同时达到了加快通信带宽的效果。

6 总结

6.1 2 个系统

- **Master 端：** Linux 系统，通过 Multiprocessing 实现 4 个进程并行，实现高性能任务：目标检测与 CV 机器视觉处理、系统状态与感知、监护客户端显示。

- **Slave 端:** FreeRTOS 系统，共 2 个 Task 并行，实现高实时性任务：多传感器采集、语音逻辑播报输出

6.2 5 个模块

- **机器视觉处理模块:** 目标检测网络识别障碍物、楼梯级数检测、斑马线走向检测、交通灯颜色检测
- **系统状态与感知模块:** GPS 地理数据解析道路信息，获取视障者沿道路方向；陀螺仪传感器数据判断是否摔倒或视角异常
- **监护客户端显示模块:** 视障者摄像头画面显示、信息栏显示、小地图显示、系统性能信息显示
- **传感器处理与语音反馈模块:** 生成障碍物、道路要素、系统状态字符串，并通过 TTS 语音播报输出
- **OpenAMP 跨系统核间通信模块:** 通过 4 个 endpoint 并行通信，采用 OpenAMP rpmsg 机制实现核间高效通信

6.3 8 个创新点

功能应用创新点：

- **人行楼梯级数检测算法（共迭代 2 个版本）**
改进前：线段提取 + 走向 + RGB 直方图；改进后：线段提取 + k, b 直方图。
- **自主训练目标检测网络与轻量化处理**
自有数据集、自有标注，场景定制；INT8 轻量化，无外加硬件，CPU 高效推理。
- **原创斑马线方向与交通灯识别算法**
- **拓展型单目距离、角度结算方法，特色小地图显示**
- **利用周边道路图解构与运动方向滑动窗口法，判断视障人士是否沿道路行走**

系统应用创新点：

- **Master 端多进程并行框架（共迭代 2 个版本）**

改进前：QThread 库，间歇卡顿；改进后：Multiprocessing 库，性能均衡。

- **Master 端 Linux 内核的 rpmsg 多静态端点实现多通道并行核间通信**

修改原生 Linux 内核驱动程序，解决 rpmsg 设备文件命名混乱，实现多信道并行通信。

- **Slave 端非阻塞 platform_poll 的实现**

修改飞腾 SDK 仅有的阻塞方案，实现核间通信与 FreeRTOS Task 同时运行。

7 参考文献

[1] Global Estimates of Vision Loss[EB/OL].<https://www.iapb.org/learn/vision-atlas>.

[2] Introduction to OpenAMP Library
(v1.1a)[EB/OL].<https://static.linaro.org/connect/hkg18/presentations/hkg18-411.pdf>.

[3] RPMsg Messaging Protocol — OpenAMP documentation[EB/OL].https://openamp.readthedocs.io/en/latest/protocol_details/rpmsg.html.

[4] RPMsg in Linux and OpenAMP Project[EB/OL].<https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/2423717889/RPMsg+in+Linux+and+OpenAMP+Project>.

[5] Chen Z, Zhang Z, Dai F, et al. Monocular vision-based underwater object detection[J]. Sensors (Switzerland), 2017, 17 (8).

[6] Wu F, Zhu S, Ye W. A Single Image 3D Reconstruction Method Based on a Novel Monocular Vision System[J], 2020, 20 (24): 7045.

[7] Dong H, Fu Q, Zhao X, et al. Practical rotation angle measurement method by monocular vision[J]. Applied Optics, 2015, 54 (3): 425-435.

[8] Jin J, Lingna Z, Xu S. High-precision rotation angle measurement method based on monocular vision[J]. Journal of the Optical Society of America A, 2014, 31.

[9] Xu Y, Liu C, Zhang J, et al. Design and recognition of monocular visual artificial landmark based on arc angle information coding[C]. Proceedings - 2018 33rd Youth Academic Annual Conference of Chinese Association of Automation, YAC 2018, 2018: 722-727.

附录

(a) 楼梯级数识别算法代码实现（包括改进前与改进后两种实现）：

```
1. import multiprocessing
2. import cv2
3. import numpy as np
4. import random
5. import matplotlib
6. matplotlib.use('QtAgg')
7. import matplotlib.pyplot as plt
8. from scipy.interpolate import UnivariateSpline
9. from scipy.interpolate import make_interp_spline
10. from scipy.signal import find_peaks
11. import os
12. from classes import yolov5v8
13. # 输入相对于当前脚本的位置，输出绝对位置
14. # 输入: filepath 相对目录位置字符串
15. # 输出: relative_path 绝对目录位置字符串
16. def toAbsolutePath(current_script_path, filepath):
17.     current_dir = os.path.dirname(os.path.abspath(current_script_path)) # 获取当前脚本的绝对路径
18.     relative_path = os.path.join(current_dir, filepath)
19.     return relative_path
20. # 连线检测器
21. class LineDetector:
22.     def __init__(self):
23.         # 固定随机数种子，以保证每次运行结果一致
24.         random.seed(42)
25.     # 完整的检测流程
26.     # 输入: 1. image, for debug only
27.     def detectLines(self, image):
28.         # step 1 检测线条
29.         lines = cv2.HoughLinesP(image, 1, np.pi / 180, threshold=50, minLineLength=50, maxLineGap=10)
30.         if lines is not None:
31.             # step 2 计算线条方程
32.             line_equations = self.calculateLineEquations(lines)
33.             # step 3 分析斜率k 分布
34.             self.filtered_lines, max_peak_x = self.analysisFilterLinesAroundPeak(line_equations)
35.             # step 4 分析截距b 分布
36.             self.rough_lines = self.analysisInterceptDistribution(self.filtered_lines, max_peak_x)
37.             # 返回: 粗处理后的线条
38.             return self.rough_lines, lines
39.     else:
40.         return None
41.     # 完整的单个楼梯检测流程（包括标签框内）（旧版）
42.     # 输入: 1. original_lines: 所有检测到的边缘
```

```

43.     # 2. singlefinalbox: 单个标签框, 不是一组标签框, 包括[x1, y1, x2, y2]
44.     def processLinesWithinBox(self, original_lines, singlefinalbox):
45.         # step 1 检测框内线条
46.         box_x1, box_y1, box_x2, box_y2 = singlefinalbox
47.         inside_box_lines = []
48.         if original_lines is None:
49.             return [], []
50.         if len(original_lines) <= 0:
51.             return [], []
52.         for line in original_lines:
53.             for x1, y1, x2, y2 in line:
54.                 # 检查线条的任一端点是否在框内
55.                 if (box_x1 <= x1 <= box_x2 and box_y1 <= y1 <= box_y2) or
56.                     (box_x1 <= x2 <= box_x2 and box_y1 <= y2 <= box_y2):
57.                     inside_box_lines.append(line)
58.                     break
59.         if inside_box_lines is not None:
60.             # step 2 计算线条方程
61.             line_equations = self.calculateLineEquations(inside_box_lines)
62.             # step 3 分析斜率k 分布
63.             self.filtered_lines, max_peak_x = self.analysisFilterLinesArounPeak(line_equations)
64.             # step 4 分析截距b 分布
65.             if self.filtered_lines is None: # 如果存在这种峰值分布
66.                 return [], [] # 需要返回两个空list, 保持返回形式一致
67.             else:
68.                 self.rough_lines = self.analysisInterceptDistribution(self
69. .filtered_lines, max_peak_x)
70.             # 返回: 粗处理后的线条
71.             return self.rough_lines, inside_box_lines
72.         else:
73.             return [], [] # 需要返回两个空list, 保持返回形式一致
74.         # 完整的单个楼梯检测流程 (包括标签框内) (仅输出斜率)
75.         # 输入: 1. original_lines: 所有检测到的边缘
76.         # 2. singlefinalbox: 单个标签框, 不是一组标签框, 包括[x1, y1, x2, y2]
77.         # 输出: 斜率
78.         def processLinesWithinBoxOutputKOnly(self, original_lines, singlefinal
79. box):
80.             # step 1 检测框内线条
81.             box_x1, box_y1, box_x2, box_y2 = singlefinalbox
82.             inside_box_lines = []
83.             if original_lines is None:
84.                 return 0
85.             if len(original_lines) <= 0:
86.                 return 0
87.             for line in original_lines:
88.                 for x1, y1, x2, y2 in line:
89.                     # 检查线条的任一端点是否在框内

```

```

88.         if (box_x1 <= x1 <= box_x2 and box_y1 <= y1 <= box_y2) or
(
89.             box_x1 <= x2 <= box_x2 and box_y1 <= y2 <= box_y2)
:
90.             inside_box_lines.append(line)
91.             break
92.         if inside_box_lines is not None:
93.             # step 2 计算线条方程
94.             line_equations = self.calculateLineEquations(inside_box_lines)
95.             # step 3 分析斜率k 分布
96.             _, max_peak_x = self.analysisFilterLinesAroundPeak(line_equations)
97.             # 返回: 最可能斜率
98.             return max_peak_x
99.         else:
100.             return 0
101.     # 计算线条方程
102.     # 输出一个List, 由大量的(斜率, 截距)tuple 组成。
103.     def calculateLineEquations(self, lines):
104.         line_equations = []
105.         for line in lines:
106.             for x1, y1, x2, y2 in line:
107.                 if x2 != x1:
108.                     # 计算斜率和截距
109.                     m = (y2 - y1) / (x2 - x1)
110.                     b = y1 - m * x1
111.                     line_equations.append((m, b))
112.                 else:
113.                     # 处理垂直线段的情况
114.                     line_equations.append((float('inf'), x1)) # 使用 'inf' 表示斜率无穷大, 存储x 坐标作为截距
115.         return line_equations
116.     # 分析斜率k 分布
117.     # 输入:
118.     # Lines, 一个List, 由大量(斜率, 截距)tuple 组成;
119.     # isplot, 是否绘制图像
120.     # 返回值:
121.     # filtered_lines, 一个List, 由大量(斜率, 截距)tuple 组成;
122.     # max_peak_x, 峰值的x 坐标
123.     def analysisFilterLinesAroundPeak(self, lines, isplot = False):
124.         slopes = [slope for slope, _ in lines]
125.         slopes = np.array(slopes, dtype=float)
126.         finite_slopes = slopes[np.isfinite(slopes)]
127.         # Calculate the histogram of the slopes
128.         # count, bins = np.histogram(slopes, bins='auto', density=False)
129.         count, bins = np.histogram(finite_slopes, bins='auto', density=False)
130.         # Use the middle of each bin for the x values
131.         x = (bins[:-1] + bins[1:]) / 2
132.         # Find the peaks in the histogram

```

```

133.         peaks, _ = find_peaks(count)
134.         # Find the highest peak
135.         if len(count[peaks]) == 0:
136.             return None, 0
137.         else:
138.             max_peak_idx = np.argmax(count[peaks])
139.             max_peak_x = x[peaks][max_peak_idx]
140.             # Define the range around the peak
141.             range_min, range_max = max_peak_x - 0.25, max_peak_x + 0.2
5 # todo: range 是经验值
142.             # Filter the lines based on the slope being within the range of the main peak +- 3
143.             filtered_lines = [line for line in lines if range_min <= line[0] <= range_max]
144.             # 当且仅当绘制图像时, 为true
145.             if isplot is True:
146.                 # Create a spline to fit the histogram
147.                 spline = UnivariateSpline(x, count, s=0)
148.                 # Generate more points to create a smooth line
149.                 x_smooth = np.linspace(x.min(), x.max(), 1000)
150.                 y_smooth = spline(x_smooth)
151.                 # Plot the results
152.                 plt.figure(figsize=(8, 3))
153.                 plt.hist(slopes, bins='auto', density=True, alpha=0.5, label='Histogram of slopes')
154.                 plt.plot(x_smooth, y_smooth, label='Spline of distribution')
155.                 plt.xlabel('Slope')
156.                 plt.ylabel('Density')
157.                 plt.title('Probability Distribution of Slopes')
158.                 plt.legend()
159.                 plt.show()
160.             return filtered_lines, max_peak_x
161.         # 分析截距b 分布
162.         # 输入:
163.         # Lines, 一个List, 由大量(斜率, 截距)tuple 组成:
164.         # max_peak_x, 峰值的x 坐标
165.         # isplot, 是否绘制图像
166.         # 输出:
167.         # rough_lines, 所有得到的楼梯的粗处理(不是精处理)线条, 一个List, 由大量(斜率, 截距)tuple 组成:
168.         def analysisInterceptDistribution(self, lines, max_peak_x, isplot
= False):
169.             # Extract the intercepts from the list of tuples
170.             intercepts = [intercept for _, intercept in lines]
171.             # Calculate the histogram of the intercepts
172.             count, bins = np.histogram(intercepts, bins=60, density=True)
173.             # Use the middle of each bin for the x values
174.             x = (bins[:-1] + bins[1:]) / 2
175.             # Create a spline to fit the histogram
176.             spline = UnivariateSpline(x, count, s=0)

```

```

177.         # Generate more points to create a smooth line
178.         x_smooth = np.linspace(x.min(), x.max(), 1000)
179.         y_smooth = spline(x_smooth)
180.         # 从截距曲线中寻找波峰
181.         # distance: 波层之间的最小距离
182.         # height: 波峰的阈值高度
183.         y_peaks, _ = find_peaks(y_smooth, distance=25, height=0.0050)
    # todo: 需要进一步调节最小距离、阈值高度
184.         # 将拟合好的线条提取出来
185.         rough_lines = []
186.         for y_peak in y_peaks:
187.             rough_lines.append((max_peak_x, y_peak))    # 得到了新的初步
分析的楼梯的线条，但仍需进一步筛选，以确认属于楼梯
188.         # 当且仅当绘制图像时，设置为True
189.         if isplot is True:
190.             plt.figure(figsize=(8, 3))
191.             plt.hist(intercepts, bins='auto', density=True, alpha=0.5,
label='Histogram of intercepts')
192.             plt.plot(x_smooth, y_smooth, label='Spline of distribution
')
193.             plt.xlabel('Intercept')
194.             plt.ylabel('Density')
195.             plt.title('Probability Distribution of Intercepts')
196.             plt.legend()
197.             plt.show()
198.         return rough_lines
199.     # 楼梯检测器
200.     class StairsDetector:
201.         # 构造函数
202.         def __init__(self):
203.             self.original_image = None    # 原始图像
204.             self.output_image = None
205.             self.linedetector = LineDetector()  # 线条检测器
206.             random.seed(42)
207.         # 工具函数：检测边缘
208.         def detectEdges(self, image):
209.             temp_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)  # 转换为灰
度图
210.             temp_image = cv2.GaussianBlur(temp_image, (5, 5), 0) # 应用高斯
模糊减少噪声
211.             temp_image = cv2.Canny(temp_image, 50, 150) # 使用Canny 算法检测
边缘
212.             return temp_image
213.         # 工具函数：检测线条
214.         def detectLines(self, image):
215.             lines = cv2.HoughLinesP(image,
216.                                     1, np.pi / 180,
217.                                     threshold=50, minLineLength=50, m
axLineGap=10)
218.             return lines
219.         # 工具函数：将每一条线条延伸到边缘

```

```

220.     def detectLinesToImageEdge(self, image):
221.         lines = cv2.HoughLinesP(image, 1, np.pi / 180, 50, minLineLength=50, maxLineGap=10)
222.         if lines is not None:
223.             extended_lines = []
224.             for line in lines:
225.                 for x1, y1, x2, y2 in line:
226.                     # 计算线段的斜率和截距
227.                     if x2 - x1 == 0: # 垂直线处理
228.                         extended_lines.append([x1, 0, x1, image.shape[0]])
229.                     else:
230.                         slope = (y2 - y1) / (x2 - x1)
231.                         intercept = y1 - slope * x1
232.                         # 计算与图像边缘的交点
233.                         y_at_x0 = int(intercept)
234.                         y_at_xmax = int(slope * image.shape[1] + intercept)
235.                         if slope != 0:
236.                             x_at_y0 = int(-intercept / slope)
237.                             x_at_ymax = int((image.shape[0] - intercept) / slope)
238.                         else:
239.                             x_at_y0 = 0
240.                             x_at_ymax = image.shape[1]
241.                         # 确定线段的新端点
242.                         new_x1, new_y1, new_x2, new_y2 = x1, y1, x2, y2
243.                         if 0 <= y_at_x0 <= image.shape[0]: # y = 0 交点
244.                             new_x1, new_y1 = 0, y_at_x0
245.                         if 0 <= y_at_xmax <= image.shape[0]: # y = ym
246.                             new_x2, new_y2 = image.shape[1], y_at_xmax
247.                         if 0 <= x_at_y0 <= image.shape[1]: # x = 0 交点
248.                             new_x1, new_y1 = x_at_y0, 0
249.                         if 0 <= x_at_ymax <= image.shape[1]: # x = xm
250.                             new_x2, new_y2 = x_at_ymax, image.shape[0]
251.                         extended_lines.append([new_x1, new_y1, new_x2, new_y2])
252.             return np.array(extended_lines)
253.         return lines
254.     # 工具函数: 绘制线条 (当且仅当元素为Numpy64 时使用)
255.     def drawLinesNumpy64(self, image, lines):
256.         if lines is not None:
257.             for line in lines:
258.                 cv2.line(image, (line[0], line[1]),
259.                         (line[2], line[3]), (0, 255, 0), 1)
260.         return image

```

```

261.     # 工具函数：绘制线条（当且仅当元素为 Iterator 时使用）
262.     def drawLines(self, image, lines):
263.         if lines is not None:
264.             for line in lines:
265.                 for x1, y1, x2, y2 in line:
266.                     cv2.line(image, (x1, y1), (x2, y2), (0, 255, 0), 1
267. )
268.         return image
269.     # 绘制粗处理后的线条，image 为图片，rough_lines 是粗处理后的线条，由大量的(斜率，截距)tuple 组成
270.     # singlefinalbox 为单独的一个标签框
271.     # 返回值：该标签框内的线条数量
272.     def extractRoughLinesOnImage(self, image, rough_lines, singlefinal
273. box):
274.         x1, y1, x2, y2 = int(singlefinalbox[0]), int(singlefinalbox[1]
275. ), int(singlefinalbox[2]), int(singlefinalbox[3])
276.         # 绘制标签框，并在图像上显示直线方程的数量
277.         cv2.rectangle(image, (x1, y1), (x2, y2), (255, 0, 0), 2) # 目
278.         # 将识别结果、识别分数、识别距离显示在目标框旁
279.         num_lines_text = f"Stair Steps: {len(rough_lines)}"
280.         # print(num_lines_text) #for debug only
281.         cv2.putText(image, '{0}'.format(num_lines_text),
282.                     (x2 - 300, y2 - 20),
283.                     cv2.FONT_HERSHEY_SIMPLEX,
284.                     1.2, (255, 255, 120), 3)
285.         #cv2.putText(image, num_lines_text, (10, 30), cv2.FONT_HERSHEY
286. _SIMPLEX, 1, (0, 0, 255), 2)
287.         return len(rough_lines) # 如果有需要将识别到的阶梯级数显示出来
288.     # 绘制粗处理后的线条，image 为图片，rough_lines 是粗处理后的线条，由大量的(斜率，截距)tuple 组成
289.     # singlefinalbox 为单独的一个标签框
290.     # 返回值：该标签框内的线条数量
291.     def plotFinalBoxAndNumbers(self, image, linenumbers, singlefinalbo
292. x):
293.         x1, y1, x2, y2 = int(singlefinalbox[0]), int(singlefinalbox[1]
294. ), int(singlefinalbox[2]), int(
295.             singlefinalbox[3])
296.         # 绘制标签框，并在图像上显示直线方程的数量
297.         cv2.rectangle(image, (x1, y1), (x2, y2), (255, 0, 0), 2) # 目
298.         # 将识别结果、识别分数、识别距离显示在目标框旁
299.         num_lines_text = f"Stair Steps: {linenumbers}"
300.         # print(num_lines_text) # for debug only
301.         cv2.putText(image, '{0}'.format(num_lines_text),
302.                     (x2 - 300, y2 - 20),
303.                     cv2.FONT_HERSHEY_SIMPLEX,
304.                     1.2, (255, 255, 120), 3)
305.         # cv2.putText(image, num_lines_text, (10, 30), cv2.FONT_HERSHEY
306. _SIMPLEX, 1, (0, 0, 255), 2)
307.         return linenumbers # 如果有需要将识别到的阶梯级数显示出来

```

```

301.      # 提取图像的RGB与亮度分布
302.      # 输入: image: OpenCV 图像; bbox: 目标框,[x1, y1, x2, y2], 需要保证都是整数且被clipped;
303.      # isplot: 是否使用matplotlib 绘图, 仅做 debug 使用
304.      def extractRGBBrightnessDistribution(self, image, bbox, isplot=True,
305.                                              contrast_factor = 1.2):    #
306.          contrast_factor: 对比度因子
307.          # 请确保所有的bbox 的值都是整数, 且被clipped 到0~640 范围内。
308.          # 1. 增加图像对比度
309.          image = np.clip((image.astype(np.float32) * contrast_factor),
310.                            0, 255).astype(np.uint8)
311.          # 2. 计算目标框的中心点, 计算垂直斜率k'
312.          center_x = (bbox[0] + bbox[2]) // 2
313.          center_y = (bbox[1] + bbox[3]) // 2
314.          # 3. 初始化RGB 和亮度的列表
315.          r_values, g_values, b_values, brightness_values = [], [], [], []
316.          # 4. 遍历选定范围内的像素点
317.          for y in range(bbox[1], bbox[3] + 1):
318.              # 计算水平范围内的RGB 平均值
319.              region = image[y, center_x - 5:center_x + 6] # 取中心点左右5 个像素, 共11 个像素
320.              avg_r = np.mean(region[:, 0])
321.              avg_g = np.mean(region[:, 1])
322.              avg_b = np.mean(region[:, 2])
323.              avg_brightness = np.mean(region) # 计算亮度的平均值
324.              # 将计算结果添加到列表中
325.              r_values.append(avg_r)
326.              g_values.append(avg_g)
327.              b_values.append(avg_b)
328.              brightness_values.append(avg_brightness)
329.          # 5. y 轴的值, 即图像的高度范围
330.          y_values = range(bbox[1], bbox[3] + 1)
331.          # 6. 使用spline 曲线平滑数据
332.          y_new = np.linspace(min(y_values), max(y_values), 300) # 生成更密集的y 值以便绘制平滑曲线
333.          spl_r = make_interp_spline(y_values, r_values, k=2) # 创建插值
334.          # 曲线
335.          spl_g = make_interp_spline(y_values, g_values, k=2)
336.          spl_b = make_interp_spline(y_values, b_values, k=2)
337.          spl_brightness = make_interp_spline(y_values, brightness_value
338.                                              s, k=2)
339.          r_smooth = spl_r(y_new)
340.          g_smooth = spl_g(y_new)
341.          b_smooth = spl_b(y_new)
342.          brightness_smooth = spl_brightness(y_new)

```

```

343.         plt.plot(y_new, g_smooth, label='Green', color='green')
344.         plt.plot(y_new, b_smooth, label='Blue', color='blue')
345.         plt.plot(y_new, brightness_smooth, label='Brightness', col
or='black')
346.         plt.title('RGB and Brightness distribution along y-axis')
347.         plt.xlabel('y-axis')
348.         plt.ylabel('Value')
349.         plt.legend()
350.         plt.show()
351.         return 0 # todo: 新增
352.     # 提取图像的RGB与亮度分布
353.     # 输入: image: OpenCV 图像; bbox: 目标框,[x1, y1, x2, y2], 需要保证都
是整数且被clipped; possible_m: 斜率
354.     # isplot: 是否使用matplotlib 绘图, 仅做 debug 使用
355.     def extractRGBBrightnessDistributionNew(self, image, bbox, possibl
e_m, isplot=False,
356.                                                 contrast_factor = 1.2, # cont
rast_factor: 对比度因子
357.                                                 window_size = 5,      # 采样滑动
窗口大小
358.                                                 k_value = 3):       # 拟合spline
曲线阶数
359.     # 请确保所有的bbox 的值都是整数, 且被clipped 到0~640 范围内。
360.     # 1. 增加图像对比度
361.     image = np.clip((image.astype(np.float32) * contrast_factor),
0, 255).astype(np.uint8)
362.     # 2. 计算目标框的中心点, 计算垂直斜率k'
363.     center_x = (bbox[0] + bbox[2]) // 2
364.     center_y = (bbox[1] + bbox[3]) // 2
365.     if possible_m != 0:
366.         m_perpendicular = -1 / possible_m
367.     else:
368.         m_perpendicular = float('inf')    # 处理原斜率为0 的情况
369.     # 3. 初始化RGB 和亮度的列表
370.     r_values, g_values, b_values, brightness_values, y_values = []
, [], [], []
371.     # 4. 遍历y 范围
372.     for y in range(bbox[1], bbox[3] + 1):
373.         # 5. 对于每个y 值, 计算相应的x 值
374.         if m_perpendicular != float('inf'):
375.             x = int((y - center_y) / m_perpendicular + center_x)
376.             x_range = range(max(0, x - 5), min(639, x + 5)) # clip
, 防止出错! 很不错!
377.             rgb_values = [image[y, x_val] for x_val in x_range if
0 <= x_val < 640]
378.         else:
379.             # 6. 对于垂直线, x 值不变
380.             x = int(center_x)
381.             rgb_values = [image[y, x] for _ in range(11)] # 重复
11 次, 模拟左右5 个像素的取值
382.         # 7. 计算RGB 和亮度平均值

```

```

383.         if rgb_values:
384.             avg_r = np.mean([rgb[0] for rgb in rgb_values])
385.             avg_g = np.mean([rgb[1] for rgb in rgb_values])
386.             avg_b = np.mean([rgb[2] for rgb in rgb_values])
387.             avg_brightness = np.mean([np.mean(rgb) for rgb in rgb_
values])
388.             r_values.append(avg_r)
389.             g_values.append(avg_g)
390.             b_values.append(avg_b)
391.             brightness_values.append(avg_brightness)
392.             y_values.append(y)
393.         # 8. 使用spline 曲线平滑数据
394.         y_values = np.array(y_values)
395.         y_new = np.linspace(y_values.min(), y_values.max(), 300) # 生
成更密集的y 值以便绘制平滑曲线
396.         r_smooth = make_interp_spline(y_values, r_values, k=2)(y_new)
397.         g_smooth = make_interp_spline(y_values, g_values, k=2)(y_new)
398.         b_smooth = make_interp_spline(y_values, b_values, k=2)(y_new)
399.         brightness_smooth = make_interp_spline(y_values, brightness_va
lues, k=2)(y_new)
400.         # 调参时间到
401.         # maxmin = max(brightness_smooth) - min(brightness_smooth)
402.         # stds = np.std(brightness_smooth)
403.         prominence = max(max(brightness_smooth) - min(brightness_smoot
h), 0) * 0.18709 + 2.68322
404.         # 9. 取brightness_smooth 的波峰
405.         peaks, _ = find_peaks(brightness_smooth, prominence=prominence
, width=(1, 20))
406.         peaknumbers = len(peaks)
407.         # Last. 绘制RGB 和亮度的分布图
408.         if isplot is True:
409.             plt.figure(figsize=(10, 6))
410.             plt.plot(y_new, r_smooth, label='Red', color='red')
411.             plt.plot(y_new, g_smooth, label='Green', color='green')
412.             plt.plot(y_new, b_smooth, label='Blue', color='blue')
413.             plt.plot(y_new, brightness_smooth, label='Brightness', col
or='black')
414.             plt.title('RGB and Brightness distribution along y-axis')
415.             plt.xlabel('y-axis')
416.             plt.ylabel('Value')
417.             plt.legend()
418.             plt.show()
419.         return peaknumbers # todo
420.     # 将上述所使用的工具函数组合得到的完整检测流程
421.     # 输入: image: 原始OpenCV 图像, 640*640
422.     # bounding_boxes: 原始目标检测网络(通用/楼梯专用)的检测集合, 需要是
YOLOV5V8 类生成的
423.     # target_class: 需要检测的编号
424.     def TotalDetection(self, image, bounding_boxes=None, target_class=
0):
425.         self.original_image = image

```

```

426.         # step 1, 保留特定的楼梯标签框
427.         finalboxes, finalscores = processDetectionResults(bounding_boxes, target_class)
428.             # finalboxes: 被保留的楼梯标签框
429.             # finalscores: 每个框的得分
430.             if not finalboxes or not finalscores:
431.                 return image
432.             self.edged_image = self.detectEdges(self.original_image)      #
433.                 # edged_image 是灰度图, 带有边缘的灰度图
434.                 self.original_lines = self.detectLines(self.edged_image)    #
435.                     step 3, 检测线条
436.                     for box in finalboxes: # 遍历所有的finalboxes
437.                         # step 4, 检测可能为楼梯线条的粗处理后的线条
438.                         self.rough_lines, self.insidebox_lines = self.linedetector.
processLinesWithinBox(self.original_lines, box)
439.                             # 注释: 由于检测楼梯线条的算法不成熟, 后续还需要借由目标检测、图
像分割等算法辅助完成。
440.                             if len(self.rough_lines) > 0: # 要检测到有楼梯级数
441.                                 self.output_image = self.drawLine(self.original_image
, self.insidebox_lines) # step 5, 将原始识别出的线条绘制在图像上
442.                                 self.extractRoughLinesOnImage(self.output_image, self.
rough_lines, box)      # step 6, 将粗处理后线条提取出来
443.                                 if len(self.rough_lines) > 0:
444.                                     return self.output_image
445.                                 else:
446.                                     return image
447. # 将上述所使用的工具函数组合得到的完整检测流程
448. # 输入: image: 原始OpenCV 图像, 640*640
449. # bounding_boxes: 原始目标检测网络(通用/楼梯专用) 的检测集合, 需要是
YOLOV5V8 类生成的
450. # target_class: 需要检测的编号
451. def TotalDetection2(self, image, bounding_boxes=None, target_class
=10):
452.     self.original_image = image
453.     # step 1, 保留特定的楼梯标签框
454.     finalboxes, finalscores = processDetectionResults(bounding_boxes, target_class)
455.         # finalboxes: 被保留的楼梯标签框
456.         # finalscores: 每个框的得分
457.         if not finalboxes or not finalscores:
458.             return image
459.             self.edged_image = self.detectEdges(self.original_image)      #
460.                 # edged_image 是灰度图, 带有边缘的灰度图
461.                 self.original_lines = self.detectLines(self.edged_image)    #
462.                     step 3, 检测线条
463.                     for box in finalboxes: # 遍历所有的finalboxes
464.                         # 注意: 每个finalbox 包括一个list, 内容为[x1 y1 x2 y2]这样。
box = box.astype(int)
box = np.clip(box, 0, 639)

```

```

465.             [x1, y1, x2, y2] = box
466.             possible_k = self.linedetector.processLinesWithinBoxOutput
KOnly(self.original_lines, box)
467.             line_numbers = self.extractRGBBrightnessDistributionNew(se
lf.original_image, box, possible_k) # 分析RGB与亮度分布状况
468.             self.plotFinalBoxAndNumbers(self.original_image, line_numb
ers, box)
469.         return self.original_image
470.     # 将上述所使用的工具函数组合得到的完整检测流程
471.     # 输入: image: 原始OpenCV图像, 640*640
472.     # bounding_boxes: 原始目标检测网络(通用/楼梯专用)的检测集合, 需要是
YOLOV5V8类生成的
473.     # target_class: 需要检测的编号
474.     # 输出: outputdata, 格式为: tuple([x1, y1, x2, y2], line_numbers).
475.     def TotalDetectionWithOutputData(self, image, bounding_boxes=None,
target_class=10, out_stairs_list=None):
476.         if not isinstance(out_stairs_list, list): # 为了消除一个bug而
使用
477.             out_stairs_list = []
478.             self.original_image = image
479.             # step 1, 保留特定的楼梯标签框
480.             finalboxes, finalscores = processDetectionResults(bounding_box
es, target_class)
481.             # finalboxes: 被保留的楼梯标签框
482.             # finalscores: 每个框的得分
483.             if not finalboxes or not finalscores:
484.                 return image
485.             self.edged_image = self.detectEdges(self.original_image) # st
ep 2, 检测边缘
486.             # edged_image 是灰度图, 带有边缘的灰度图
487.             self.original_lines = self.detectLines(self.edged_image) # st
ep 3, 检测线条
488.             for box in finalboxes: # 遍历所有的finalboxes
489.                 # 注意: 每个finalbox 包括一个list, 内容为[x1 y1 x2 y2]这样。
490.                 box = box.astype(int)
491.                 box = np.clip(box, 0, 639)
492.                 [x1, y1, x2, y2] = box
493.                 possible_k = self.linedetector.processLinesWithinBoxOutput
KOnly(self.original_lines, box)
494.                 line_numbers = self.extractRGBBrightnessDistributionNew(se
lf.original_image, box,
495.                                         possible_k) # 分析RGB与亮度分布状况
496.                                         self.plotFinalBoxAndNumbers(self.original_image, line_numb
ers, box)
497.                                         out_stairs_list.append((int((x1 + x2)/2), int((y1 + y2)/2)
, line_numbers))
498.         return self.original_image # 最后输出了这个!
499. ###### 功能函数
500. # 拉伸cv2图像
501. def ResizeImage(image, target_width, target_height):

```

```

502.     # 获取图像的宽度和高度
503.     height, width = image.shape[:2]
504.     # 根据拉伸比例进行拉伸
505.     resized_image = cv2.resize(image, (target_width, target_height))
506.     return resized_image
507. def isOverLap(box1, box2):
508.     """
509.     判断两个 bounding box 是否重叠。
510.     box 的格式为 [x1, y1, x2, y2], 其中 (x1, y1) 是左上角的坐标, (x2, y2) 是右
      下角的坐标。
511.     """
512.     if box1[2] < box2[0] or box1[0] > box2[2] or box1[3] < box2[1] or
      box1[1] > box2[3]:
513.         return False
514.     return True
515. def processDetectionResults(input_data, target_class):
516.     """
517.     处理检测结果, 保留特定 class 编号的对象, 对于重叠的 bounding boxes 只保留
      得分最高的。
518.     """
519.     boxes, scores, classes = input_data
520.     filtered_boxes = []
521.     filtered_scores = []
522.     # 根据 class 编号筛选
523.     for box, score, cls in zip(boxes, scores, classes):
524.         if cls == target_class:
525.             filtered_boxes.append(box)
526.             filtered_scores.append(score)
527.     # 检查并处理重叠的 bounding boxes
528.     final_boxes = []
529.     final_scores = []
530.     for i in range(len(filtered_boxes)):
531.         overlap = False
532.         for j in range(len(filtered_boxes)):
533.             if i != j and isOverLap(filtered_boxes[i], filtered_boxes[j]):
534.                 overlap = True
535.                 # 保留得分更高的 bounding box
536.                 if filtered_scores[i] < filtered_scores[j]:
537.                     break
538.             else:
539.                 # 如果没有发现重叠或者当前 box 得分最高, 则加入最终结果
540.                 if not overlap or filtered_scores[i] >= max(filtered_scores):
541.                     final_boxes.append(filtered_boxes[i])
542.                     final_scores.append(filtered_scores[i])
543.     return final_boxes, final_scores
544. # 主程序, for debug only
545. if __name__ == "__main__":
546.     imagePath = "/home/lawrence/Desktop/stair2.jpg"
547.     image = cv2.imread(imagePath)

```

```

548.     image = ResizeImage(image, 640, 640)
549.     # 初始化
550.     current_script_path = os.path.abspath(__file__) # 当前python脚本
    目录，后续迁移可以不需要改动
551.     stairsmodel_path = toAbsolutePath(current_script_path, "../models/
    haizhuv8nint8.onnx") # 相对当前脚本位置
552.     stairsobject_detector = yolov5v8.YOLOV5V8(stairsmodel_path, isType
    ='TEST') # 创建一个YOLOv8n对象
553.     stairs_detector = StairsDetector() # 创建一个楼梯检测对象
554.     # 楼梯目标检测
555.     stairsboxes, output_img = stairsobject_detector.inference(image)
556.     # 纯楼梯检测
557.     # target_class 字段:
558.     # 如果使用的是楼梯目标检测网络，那么填0
559.     # 如果使用的是yolov8n-oiv7 网络，那么填489
560.     image1 = stairs_detector.TotalDetection2(image, stairsboxes, 6)
561.     cv2.imshow("Stairs Detection", image1)
562.     # 循环等待按键事件，直到按下'q'键退出
563.     while True:
564.         key = cv2.waitKey(1) & 0xFF
565.         if key == ord('q'):
566.             break
567.     cv2.destroyAllWindows()

```

(b) 斑马线方向识别算法代码实现

```

1. # 斑马线检测工具类
2. # 新的ZebraLineDetector，由于与stairsDetector.py 的LineDetector 在用法上有类
似之处，因此直接继承处理
3. class ZebraLineDetector(LineDetector):
4.     # 新的构造函数
5.     def __init__(self):
6.         super(ZebraLineDetector)
7.         super().__init__()
8.     # ----- 工具函数
9.     # step 1 专用函数：将某个目标框内的特定的线条过滤出来
10.    def filterLinesWithinSingleBox(self, original_lines, singlefinalbox):
11.        ...
12.        将某个目标框内的特定的线条过滤出来
13.        :param original_lines: 原始的未过滤的线条
14.        :param singlefinalbox: 单个目标检测框
15.        :return: 一个list，如果存在线条，那么list中有线条；如果不存在，那么
list为空
16.        ...
17.        box_x1, box_y1, box_x2, box_y2 = singlefinalbox
18.        inside_box_lines = []
19.        if original_lines is None:
20.            return [], []
21.        if len(original_lines) <= 0:
22.            return [], []

```

```

23.     for line in original_lines:
24.         for x1, y1, x2, y2 in line:
25.             # 检查线条的所有端点是否在框内（不是任意端点在框内）
26.             if (box_x1 <= x1 <= box_x2 and box_y1 <= y1 <= box_y2) and
27.                 box_x1 <= x2 <= box_x2 and box_y1 <= y2 <= box_y2):
28.                 inside_box_lines.append(line)
29.                 break
30.             if inside_box_lines is not None:
31.                 return inside_box_lines
32.             else:
33.                 return []
34.     # step 2 专用函数：分析框内的长度分布，并将属于最长长度的线段输出出来
35.     def analysisFilterLengthOfLines(self, lines, isPlot=False):
36.         # 计算每个线段的长度
37.         # 注意：lines 需要这样提取：[[x1, y1, x2, y2]]！别忘了！
38.         lengths = [np.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2) for [x1, y1,
39.             x2, y2] in lines]
40.         # 得到长度的频率分布直方图
41.         n, bins, patches = plt.hist(lengths, bins=8, density=True, visible
42.             =isPlot)
43.         # 注意，bins=12 是不错的选择！# todo: 12 这个参数是可行的！
44.         # 使用spline 进行拟合
45.         bin_centers = 0.5 * (bins[:-1] + bins[1:])
46.         spline = UnivariateSpline(bin_centers, n, s=0, k=3) # 保证它是三次
样条曲线，而不报错
47.         # 绘制曲线
48.         if isPlot is True:
49.             plt.xlabel('Length')
50.             plt.ylabel('Frequency')
51.             plt.title('Frequency Distribution of Line Lengths')
52.             xs = np.linspace(min(lengths), max(lengths), 1000)
53.             plt.plot(xs, spline(xs), 'r-', label='Spline Fit')
54.             plt.legend()
55.             plt.show()
56.         # 找到波谷和波峰
57.         # n 是直方图的值, bin_centers 是直方图中每个桶的中心值
58.         n_derivative = np.diff(n) # 计算一阶导数
59.         n_derivative2 = np.diff(n_derivative) # 计算二阶导数
60.         # 寻找波峰和波谷
61.         peaks = [] # 波峰：一阶导数符号由正转负，二阶导数为负
62.         troughs = [] # 波谷：一阶导数符号由负转正，二阶导数为正
63.         for i in range(1, len(n_derivative)):
64.             if n_derivative[i - 1] > 0 > n_derivative[i] and n_derivative2
65.             [i - 1] < 0:
66.                 peaks.append(bin_centers[i])
67.             elif n_derivative[i - 1] < 0 < n_derivative[i] and n_derivative2
68.             [i - 1] > 0:
69.                 troughs.append(bin_centers[i])
70.         # 确定波峰

```

```

67.     if peaks:
68.         peak = max(peaks)
69.     else:
70.         # print("No peaks found.")
71.         peak = None
72.     # 确定波谷
73.     if not troughs:
74.         left_trough, right_trough = min(bin_centers), max(bin_centers)
75.     elif len(troughs) == 1:
76.         if peak and peak > troughs[0]:
77.             left_trough, right_trough = troughs[0], max(bin_centers)
78.         else:
79.             left_trough, right_trough = min(bin_centers), troughs[0]
80.     else:
81.         if peak:
82.             left_trough = max([trough for trough in troughs if trough
83. < peak], default=min(bin_centers))
84.             right_trough = min([trough for trough in troughs if trough
85. > peak], default=max(bin_centers))
86.         else:
87.             left_trough, right_trough = min(troughs), max(troughs)
88.     # 筛选符合条件的线段
89.     filtered_lines = [line for line in lines if
90.                         left_trough <= np.sqrt(
91.                             (line[0][2] - line[0][0]) ** 2 + (line[0][3]
92. - line[0][1]) ** 2) <= right_trough]
93.     return filtered_lines
94. # step 3 专用函数: 去除过于密集的线
95. def removeDenseLines(self, lines, threshold=55):
96. """
97. 移除过于密集的线段。
98. :param lines: 原始线段列表, 每个线段的格式为[[x1, y1, x2, y2]]
99. :param threshold: 距离阈值, 用于判断线段是否过于密集 # TODO: 调整了
100. :return: 稀疏化后的线段列表
101. """
102.     midpoints =[((line[0][0] + line[0][2]) / 2, (line[0][1] + line[0]
103. [3]) / 2) for line in lines]
104.     keep = [True] * len(lines) # 初始化所有线段的保留状态为True
105.     for i in range(len(midpoints)):
106.         for j in range(i + 1, len(midpoints)):
107.             if keep[i] and keep[j]:
108.                 # 计算两中点之间的距离
109.                 dist = np.sqrt((midpoints[i][0] - midpoints[j][0])
110. ** 2 + (midpoints[i][1] - midpoints[j][1]) ** 2)
111.                 if dist < threshold:
112.                     # 如果两线段过于靠近, 则不保留后一条线段
113.                     keep[j] = False
114.     # 只保留未被标记为过于密集的线段
115.     return [line for k, line in zip(keep, lines) if k]
116. # step 4 专用函数: 计算斜率, 取斜率的平均值, 然后取这个平均值的垂直线

```

```

112.      # 返回值: direction, 是(x5, y5, x6, y6), 这是一条新的线段, 该线段在目
标框内并且通过目标框的中点, 指示这个平均值的垂直线
113.      # degree: 这个名为“direction”的新线段的斜率相对于图像垂直线的角度。这个
角度在-90 到 90 度之间, 顺时针为正, 逆时针为负。
114.      def analysisAveOfSlopeAndReturnByVerticalLine(self, densed_lines,
singlefinalbox):
115.          # 计算所有线段的斜率
116.          # 注意: line 中需要中间嵌套一层
117.          slopes = [
118.              (line[0][3] - line[0][1]) / (line[0][2] - line[0][0]) if (
line[0][2] - line[0][0]) != 0 else float('inf')
119.              for line in
120.                  densed_lines]
121.          # 计算斜率的平均值
122.          avg_slope = np.mean([slope for slope in slopes if slope != flo
at('inf')])
123.          # 计算垂直斜率 k1
124.          if avg_slope == 0:
125.              k1 = float('inf') # 防止除以0
126.          else:
127.              k1 = -1 / avg_slope
128.          # k1 不是绝对的垂直, 而是和斑马线框的比例系数有关系
129.          x_min, y_min, x_max, y_max = singlefinalbox
130.          # if abs(k1) > 3:
131.          k1 = k1 * ((y_max - y_min) / (x_max - x_min)) # 进一步修
正 todo
132.          # 线段中点
133.          # mid_point = [(singlefinalbox[0] + singlefinalbox[2]) / 2, (s
inglefinalbox[1] + singlefinalbox[3]) / 2]
134.          midpoints = [((line[0][0] + line[0][2]) / 2, (line[0][1] + lin
e[0][3]) / 2) for line in densed_lines]
135.          if len(midpoints) > 0:
136.              avg_x = sum(point[0] for point in midpoints) / len(midpoi
nts)
137.              avg_y = sum(point[1] for point in midpoints) / len(midpoi
nts)
138.          else:
139.              avg_x = np.int32(320)
140.              avg_y = np.int32(320)
141.          mid_point = [np.int32(avg_x), np.int32(avg_y)]
142.          # 根据斜率和中点计算direction 线段的两个端点
143.          # 由于线段需要在目标框内, 我们可以将x 设置为x3 和x4, 然后计算对应的
y
144.          if k1 != float('inf'):
145.              x5, x6 = singlefinalbox[0], singlefinalbox[2]
146.              y6 = k1 * (x6 - mid_point[0]) + mid_point[1] # 不要调换
了!
147.              y5 = k1 * (x5 - mid_point[0]) + mid_point[1] # 不要调换
了!
148.          else:
149.              # 垂直线段的情况

```

```

150.         y5, y6 = singlefinalbox[1], singlefinalbox[3]
151.         x5 = x6 = mid_point[0]
152.         # 检查并调整线段的端点，确保它们在目标框内
153.         def adjust_point(x, y, k1, x_min, x_max, y_min, y_max):
154.             # 如果线段垂直
155.             if k1 == float('inf'):
156.                 return x, max(min(y, y_max), y_min)
157.             # 如果线段水平
158.             elif k1 == 0:
159.                 return max(min(x, x_max), x_min), y
160.             else:
161.                 # 检查并调整y5
162.                 if y < y_min:
163.                     x = x + ((y_min - y) / k1) # 这一行和下面一行不要调
换顺序！
164.                     y = y_min
165.                 elif y > y_max:
166.                     x = x - ((y - y_max) / k1) # 这一行和下面一行不要调
换顺序！
167.                     y = y_max
168.                     # 确保x在边界内
169.                     # x = max(min(x, x_max), x_min)
170.                     return x, y
171.         x5_new, y5_new = adjust_point(x5, y5, k1, x_min, x_max, y_min,
y_max)
172.         x6_new, y6_new = adjust_point(x6, y6, k1, x_min, x_max, y_min,
y_max)
173.         direction = [np.int32(x5_new), np.int32(y5_new), np.int32(x6_n
ew), np.int32(y6_new)]
174.         # 计算角度值
175.         # 对于垂直于x轴的线，我们定义角度为90或-90
176.         if k1 == 0:
177.             degree = 0
178.         elif k1 == float('inf'):
179.             degree = 90
180.         else:
181.             degree = np.arctan(1 / -k1) * 180 / np.pi # -k1不要搞错哦
182.             # 调整角度范围到-90到90
183.             if degree > 90:
184.                 degree -= 180
185.             elif degree < -90:
186.                 degree += 180
187.         return direction, degree, k1
188.         # ----- 流程函数
189.         # 为斑马线设计的输出流程，适用于单个斑马线box，而不是所有的斑马线boxes
190.         def zebraProcessLinesWithinBox(self, original_lines, singlefinalbo
x):
191.             # step 1 检测框内线条
192.             inside_box_lines = self.filterLinesWithinSingleBox(original_li
nes, singlefinalbox)

```

```

193.         # step 2 分析框内的长度分布, 然后取最长的那部分线条出来, 通常能够代
表斑马线
194.         # 最长的那部分线, 位于densed_lines 中。
195.         densed_lines = self.analysisFilterLengthOfLines(inside_box_lin
es)
196.         if len(densed_lines) <= 0:
197.             return None, None, None, None
198.         # step 3 去除过于密集的线
199.         densed_lines_new = self.removeDenseLines(densed_lines)
200.         if len(densed_lines) <= 0:
201.             return None, None, None, None
202.         # step 4 对最长的那几条线, 计算它们的斜率, 然后取这些斜率的平均值,
最后取这个平均值的垂直线, 得到对应的斜率
203.         direction, degree, k1 = self.analysisAveOfSlopeAndReturnByVert
icalLine(densed_lines_new, singlefinalbox)
204.         return direction, degree, densed_lines_new, k1

```

(c) 交通灯颜色检测算法代码实现:

```

1. # 交通灯识别工具类
2. class TrafficLightIdentifier:
3.     def __init__(self):
4.         pass
5.     # 检测单个交通灯目标框内的颜色, 判定是红灯还是绿灯
6.     def trafficLightColor(self, image, singlebox, isPlot=False):
7.         color = "invalid"
8.         if singlebox is None:
9.             return color, 0
10.            x1, y1, x2, y2 = singlebox
11.            width = x2 - x1
12.            height = y2 - y1
13.            size = width * height    # 红绿灯的面积大小, 可供参考
14.            # 调整目标框尺寸
15.            if width >= 4 and width % 4 != 0:
16.                width -= width % 4
17.            if height >= 8 and height % 8 != 0:
18.                height -= height % 8
19.            # 如果目标框太小, 不分网格
20.            # 函数需要输出的量: grid_rgb_max
21.            if width <= 0 or height <= 0:
22.                return color, size
23.            if width < 6 or height < 8:
24.                color_values = np.mean(image[y1:y2, x1:x2], axis=(0, 1))
25.                grid_rgb_val = np.tile(color_values.reshape(3, 1, 1), (1, 6, 8
)))
26.            else:    # 分为4*8 的网格
27.                grid_width = width // 6
28.                grid_height = height // 8
29.                grid_rgb_val = np.zeros((3, 6, 8), dtype=np.uint8)
30.                for i in range(6):

```

```

31.         for j in range(8):
32.             grid_x1 = x1 + i * grid_width
33.             grid_y1 = y1 + j * grid_height
34.             grid_x2 = grid_x1 + grid_width
35.             grid_y2 = grid_y1 + grid_height
36.             # 提取并保存每个网格中R, G, B 通道的最大值
37.             grid_rgb_val[:, i, j] = np.mean(image[grid_y1:grid_y2,
38.                                         grid_x1:grid_x2], axis=(0, 1))
39.             # 注意: 通道1 为B/G/R, 不是R/G/B。通道1, [0]为B, [1]为
40.             # G, [2]为R。
41.             # 检测亮灯情况:
42.             # 如果为红灯, 那么必有R>200 且G,B 至少一项小于145。
43.             # 如果为绿灯, 那么必有G>200 且R 小于G-50。
44.             # 接着, 记录红灯或绿灯的块数。红灯优先。如果红灯/绿灯块数>1, 那么直接判
45.             redcount = 0
46.             greencount = 0
47.             for i in range(6):
48.                 for j in range(8):
49.                     # OpenCV 中R 通道是第2 个维度的最后一个, 即[2]
50.                     if grid_rgb_val[2, i, j] > 200 and (grid_rgb_val[0, i, j]
51. < 145 or grid_rgb_val[1, i, j] < 145):
52.                         redcount += 1
53.                     # OpenCV 中G 通道是第2 个维度的中间一个, 即[1]
54.                     if grid_rgb_val[1, i, j] > 200 and grid_rgb_val[2, i, j] <
55. grid_rgb_val[1, i, j] - 50:
56.                         greencount += 1
57.             if redcount <= 0 and greencount <= 0:
58.                 color = "unidentified"
59.             elif redcount >= greencount - 1:    # 测试结果
60.                 color = "red"
61.             else:
62.                 color = "green"
63.             # for debug only
64.             if isPlot is True:
65.                 fig, axs = plt.subplots(1, 3, figsize=(15, 5))
66.                 for i, color in enumerate(['Reds', 'Greens', 'Blues']):
67.                     axs[i].imshow(grid_rgb_val[i], cmap=color, interpolation='nearest')
68.                     axs[i].set_title(f"[{'R', 'G', 'B'}[{i}]] channel")
69.                     for j in range(4):
70.                         for k in range(8):
71.                             axs[i].text(j, k, f'{grid_rgb_val[i, j, k]}', ha='center', va='center', color='w')
72.             plt.title("Rect {x1}, {y1} to {x2}, {y2}")
73.             plt.tight_layout()
74.             plt.show()
75.             return color, size

```