**2025机器人软件工程学课程作业报告**

**项目名称：基于ROS Noetic的服务机器人开发**

**英文名称：Service Robot Software Development Based on ROS Noetic**

姓名：2010920 彭东阳

小组成员：

2012345 张三

2012346 李四

2012347 王五

完成时间：2025年6月

**摘 要**

随着人工智能技术的发展，机器人逐渐从大型，固定的范式向小型移动，智能的方向发展。曾经只能在工厂中见到的机器人设备开始向日常服务的市场拓展，移动服务机器人有着非常广阔的发展前景。本文基于ROS Noetic开发了一种移动服务机器人，通过综合调用Kobuki底盘，Kinect摄像头以及Dynamixel机械臂实现各种任务，特别地我们将所有的任务都语音化适配，更加符合日常使用的场景。

**关键词：VOSK, YOLO, AMCL**

**目 录**

[一、项目背景与相关研究 4](#_Toc1942431084)

[1.1 项目背景 4](#_Toc2131163473)

[1.2 相关技术研究 4](#_Toc586825773)

[二、项目整体介绍 6](#_Toc1977631670)

[2.1 整体功能和框架（图） 6](#_Toc1015833955)

[2.2 各模块之间关系与接口方法 6](#_Toc1387928587)

[三、具体功能原理和实现 7](#_Toc1596276361)

[3.1 闲逛进程的原理和实现 7](#_Toc1920514864)

[3.2 导航模块的原理和实现 8](#_Toc527001263)

[3.2 语音模块的原理和实现 9](#_Toc1968348361)

[3.2 检测跟踪模块的原理和实现 10](#_Toc1198064057)

[3.2 机械臂模块的原理和实现 12](#_Toc889022971)

[四、个人完成的主要工作 13](#_Toc226682441)

[4.1 主要工作一 13](#_Toc851557506)

[4.2 主要工作二 13](#_Toc1979402079)

[五、项目总结与展望 13](#_Toc409292824)

[5.1 项目总结 13](#_Toc902966940)

[5.2 项目展望 14](#_Toc1340188357)

# 一、项目背景与相关研究

## 1.1 项目背景

随着全球人口老龄化加剧和劳动力成本上升，服务机器人在医疗、养老、家政、教育等领域的应用需求持续增长。特别是在医疗和养老领域，机器人能够提供辅助护理、康复训练等服务，有效缓解人力短缺问题。此外，消费升级和智能家居的普及也推动了家用服务机器人的快速发展。全球服务机器人市场规模在2022年已达到数百亿美元，预计到2025年将突破千亿美元，年均复合增长率保持在两位数以上。中国作为全球最大的机器人市场之一，其服务机器人产业规模和技术水平也在快速提升，成为全球市场的重要参与者。

## 1.2 相关技术研究

移动服务机器人的发展离不开人工智能、物联网、大数据等技术的快速进步。尤其是环境识别系统、路径规划技术、多传感器信息融合技术等关键技术的突破，显著提升了机器人的智能化水平和自主移动能力。此外，国际开源机器人平台（如Player/Stage、Willow Garage、TurtleBot）的普及，也为技术研发提供了重要支持。

本文的工作是基于TurtleBot开发。TurtleBot 是一个开源的移动机器人平台，专为教育和研究目的设计，广泛应用于机器人操作系统（ROS）的开发和学习。 TurtleBot由 Melonee Wise 和 Tully Foote 于 2010 年在 Willow Garage 创建，旨在普及 ROS 的使用。第一代 TurtleBot 基于 iRobot 的 Create 平台，与 ROS 同年发布。经过多年发展，TurtleBot 已成为 ROS 的标准平台，最新一代 TurtleBot4 于 2022 年发布。TurtleBot 的硬件套件包括移动基座、2D/3D 距离传感器、笔记本电脑或单板计算机（SBC）以及安装硬件套件。其核心是 ROS，通过与各种传感器（如激光雷达、摄像头、IMU）和执行器的无缝协作，支持自主导航、避障等功能。TurtleBot 还支持扩展接口，允许用户添加自定义传感器以适应特定应用。

为了能够提高服务机器人的便捷性，我们在项目工作中引入了Vosk语音模块用以语音识别。Vosk 是一个基于 Kaldi 的开源语音识别库，具有高效、轻量级和跨平台的特点。它使用深度学习模型来实现语音识别，并且支持多种语言。语音识别的核心任务是将语音信号转换为文本。语音识别流程可以分为几个步骤：信号预处理、特征提取、声学模型处理、语言模型处理、解码输出。Vosk在此基础上进一步对模型进行优化，更加适合于移动设备以及嵌入式设备这类低性能的设备进行推理使用。

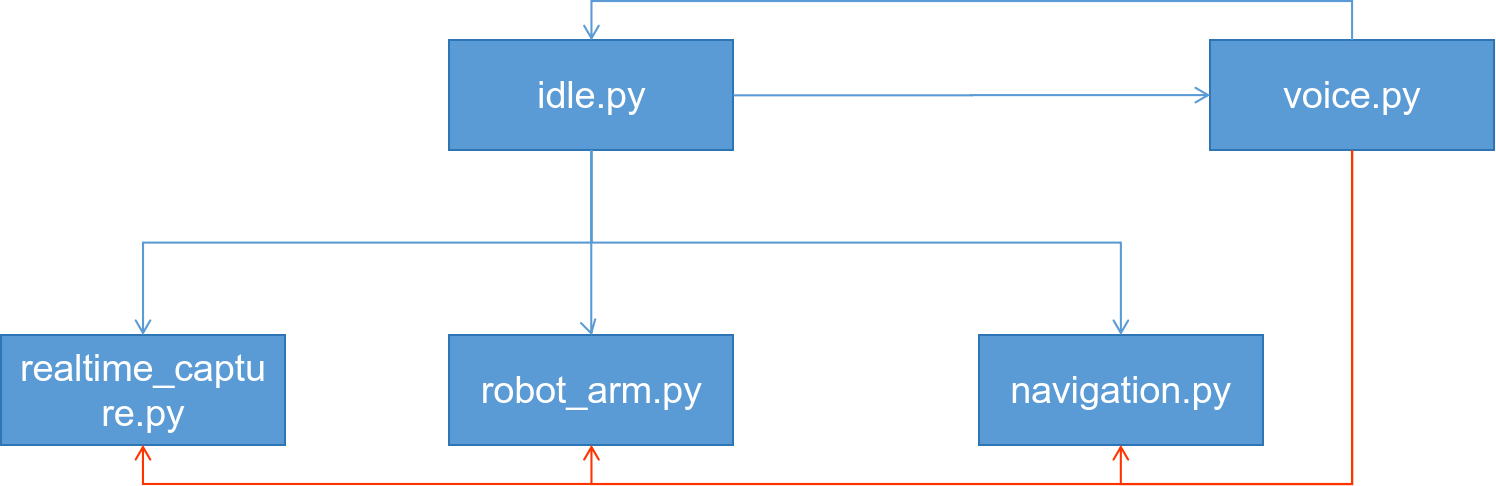
机器人的视觉是机器人接收环境信息的重要部分。视觉处理方面，目标检测是许多功能的上游任务。YOLO（You Only Look Once）是一种基于深度学习的实时目标检测算法，其核心思想是将目标检测任务转化为一个回归问题，通过单次前向传播即可完成目标的定位和分类。相较于传统的目标检测方法，YOLO具有速度快、精度高的优势，特别适合实时应用场景。YOLO算法的核心思想是将输入图像划分为S×S的网格，每个网格负责预测中心点落在该网格内的物体。每个网格预测B个边界框（Bounding Box），并为每个边界框预测置信度（Confidence Score）和类别概率（Class Probability）。置信度表示边界框包含物体的概率以及预测的准确度，类别概率则表示物体属于各个类别的概率。

YOLO的组成包括两个部分：特征提取网络、检测层。特征提取网络采用DarkNet作为骨干，通过多层卷积提取图像的特征；检测层则在特征图的基础上进行预测，输出边界框的坐标，置信度和类别概率。每个边界框的坐标包括中心点（x, y）和宽高（w, h），置信度表示边界框包含物体的概率，类别概率则表示物体属于各个类别的概率。最终，模型通过非极大值抑制（NMS）算法筛选出最优的边界框。

YOLO算法自提出以来，经历了多个版本的改进，如YOLOv2、YOLOv3、YOLOv4和YOLOv8等。这些改进主要集中在网络结构设计、损失函数优化、特征提取能力提升等方面。例如，YOLOv8引入了新的网络架构，结合了CSP技术和深度可分离卷积，显著提高了检测精度和速度。

# 二、项目整体介绍

## 2.1 整体功能和框架（图）



## 2.2 各模块之间关系与接口方法

本项目按照操作系统的设计方式，初始启动时只有一个闲逛进程idle.py，这个脚本中的主函数是一个写死的循环，每个循环都从终端中读取一个指令，根据指令选择合适的服务，首先声明一个新的线程并绑定到对应的功能函数，随后启动功能函数线程再返回到闲逛进程的主函数中。

realtime\_capture.py是一个实时检测并跟踪的一个模块，在该模块中注册了一个ROS Service名为start\_capture，该服务是一个SetBool类型的服务，通过传入一个布尔值来调用对应的处理函数进行对应的动作，当传入值为True时，处理函数会启动新的检测跟踪线程；反之传入False时，处理函数会将对应的状态值self.running\_state直接置为False，那么线程就会自动终止退出循环。

navigate.py是导航模块，该模块同样注册了一个ROS Service名为goto，与上面的start\_capture服务不同的是，goto服务并非SetBool类型，而是使用的一个自定义类型的服务，需要手动编写一个navigate\_msg.srv的服务类型定义文件。这里我们简单地将消息的内容定义为string类型，闲逛进程主函数通过不同的地点名称来告诉navigate.py模块去导航到对应的位置去。

robot\_arm.py是机械臂控制模块，该模块也注册了两个ROS Service名为seize和release，用于控制机械臂的抓取和松开。其调用方式与start\_capture类似。

voice.py则是语音代理模块，该模块注册了两个ROS Service分别名为start\_voice和stop\_voice，分别用于控制语音代理的开始和关闭。值得一提的是，语音模块打开之后会禁用终端输入，输入的语音直接用于判断执行的服务。

# 三、具体功能原理和实现

## 3.1 闲逛进程的原理和实现

该模块是一个基于ROS（机器人操作系统）的主控制节点，用于管理和执行多种机器人任务，包括目标检测、语音控制、导航和机械臂操作等。其核心原理是通过订阅主题（Topic）和调用服务（Service）来实现与机器人各个模块的交互，从而完成用户输入的命令或语音指令。

首先，初始化了一个名为FindOS 的ROS节点，并订阅了/voice/voice\_command主题，用于接收语音指令。语音指令通过回调函数voice\_command\_callback进行处理，并将其存储在一个全局的命令缓冲区command\_buffer中。为了确保主线程能够及时处理语音指令，代码使用了threading.Semaphore来实现线程同步，当缓冲区中有新指令时，信号量buffer\_full会被释放，主线程随后从缓冲区中取出指令进行处理。

在主循环中，代码通过用户输入或语音指令来执行不同的任务。用户可以通过键盘输入命令，也可以通过语音控制模式输入指令。根据命令内容，代码会调用相应的ROS服务来执行任务。例如，当命令中包含follow时，代码会调用/detector/start\_capture服务，启动目标检测模块；当命令中包含stop时，代码会调用同样的服务，但传递False参数以停止检测。语音控制模式通过调用/voice/start\_voice和/voice/stop\_voice服务来启用或禁用语音输入。

此外，代码还支持机械臂操作和导航任务。当命令中包含hold或release时，代码会分别调用/arm/seize和/arm/release服务，控制机械臂抓取或释放物体。当命令中包含go to时，代码会调用/goto服务，并传递目标位置作为参数，启动导航任务。导航服务的请求对象navigate\_msgRequest被用来封装目标位置，并通过navigate服务进行调用。

此外还有一些异常情况，例如服务调用失败时会记录错误日志，并在程序结束时打印堆栈信息以帮助调试。最终，当用户输入quit命令时，主循环退出，节点停止运行。

总体而言，通过结合ROS的消息传递机制和服务调用功能，实现了一个灵活且可扩展的机器人控制框架。它能够处理多种输入方式（键盘和语音），并根据不同的命令调用相应的服务来执行任务。这种设计使得代码易于维护和扩展，适合用于复杂的机器人应用场景。

## 3.2 导航模块的原理和实现

该模块实现了一个基于ROS的导航代理节点，主要用于控制机器人在已知地图中移动到指定目标位置。其核心原理是通过调用ROS的move\_base动作服务器来实现导航任务，并通过自定义服务/goto来接收外部请求，触发导航行为，能够高效地处理导航任务并对外提供服务。

首先，初始化一个名为navigate\_agent的ROS节点，并创建了一个Navigation类的实例。在Navigation类的构造函数中，代码初始化了与move\_base动作服务器的连接，并等待服务器启动。move\_base是ROS中用于实现导航的核心模块，负责路径规划和执行。为了确保导航任务能够顺利进行，代码设置了120秒的超时时间，如果在此期间未能连接到服务器，程序会记录错误日志。

在Navigation类中，定义了一个目标位置字典target\_dict，其中包含了多个预设的目标位置及其对应的坐标和朝向。这些目标位置可以是地图中的关键点，例如“教室”、“家”或“点”。当外部请求通过/goto服务传入时，代码会根据请求中的目标名称在字典中查找对应的坐标，并启动导航任务。

导航任务的核心逻辑封装在goto方法中。该方法首先将目标位置的朝向从角度转换为弧度，并进一步转换为四元数，以便用于表示机器人的姿态。然后，代码创建了一个PoseStamped对象，用于描述目标位置在全局地图中的位姿。接着，代码创建了一个MoveBaseGoal对象，并将其发送给move\_base动作服务器以启动导航任务。如果blocking参数为True，代码会等待导航任务完成，并在任务结束后记录日志。

为了支持非阻塞的导航任务，代码使用了多线程技术。当外部请求通过/goto服务传入时，handle\_navigation方法会创建一个新的线程来执行goto方法，从而避免阻塞主线程。这种设计使得导航任务能够独立运行，而不会影响其他任务的执行。

在程序的主逻辑中，代码注册了/goto服务，并将handle\_navigation方法绑定为该服务的回调函数。这样，当外部节点调用/goto服务时，handle\_navigation方法会被触发，并根据请求中的目标名称启动导航任务。此外，代码还实现了clean\_up方法，用于在节点关闭时执行清理操作，例如记录日志。

总体而言，这一部分通过结合ROS的动作服务器和服务机制，实现了一个灵活且高效的导航代理节点。它能够根据外部请求启动导航任务，并支持多线程执行以避免阻塞。通过预设的目标位置字典，代码能够快速定位目标并完成导航任务。这种设计使得代码易于扩展和维护，适合用于复杂的机器人应用场景。

## 3.2 语音模块的原理和实现

该模块实现了一个基于ROS的语音控制节点，其核心功能是通过麦克风录制音频，并使用Vosk语音识别模型将音频转换为文本指令，然后将这些指令发布到ROS系统中。代码的设计体现了模块化和多线程的思想，能够高效地处理语音输入和识别任务，并通过ROS服务提供启动和停止语音控制的功能。

首先，代码初始化了一个名为voice\_control的ROS节点，并创建了一个VoiceAgent类的实例。在VoiceAgent类的构造函数中，代码加载了Vosk语音识别模型，并初始化了音频输入设备。Vosk是一个基于Kaldi的轻量级语音识别库，支持多种语言和高效的实时识别。音频输入设备通过pyaudio库进行配置，包括采样率、通道数和帧大小等参数。为了管理音频数据的处理，代码使用了两个线程信号量list\_full和list\_empty，分别用于控制音频数据的生产和消费。

在RecordSingle方法中，代码通过麦克风录制一段音频，并将其存储为二进制数据流。录制的音频数据随后被添加到processing\_list中，供后续的语音识别任务使用。为了确保音频数据的连续性和高效处理，代码使用了多线程技术。在RecordLoop方法中，代码通过一个循环不断录制音频，并将录制的数据添加到processing\_list中。同时，信号量list\_empty和list\_full被用来控制音频数据的生产和消费，以避免数据溢出或不足。

在AnalysisSingle方法中，代码从processing\_list中取出音频数据，并使用Vosk模型进行语音识别。识别结果被转换为文本，并通过ROS的command\_pub发布到voice\_command主题中。为了确保识别的准确性和连续性，代码使用了AnalysisLoop方法，通过一个循环不断处理音频数据，并将识别结果发布到ROS系统中。

代码还提供了启动和停止语音控制的功能。在start\_voice方法中，代码启动了两个线程，分别用于录制音频和处理音频数据。在stop\_voice方法中，代码停止音频录制和处理，并清空processing\_list和信号量。这些功能通过ROS服务start\_voice和stop\_voice对外提供，外部节点可以通过调用这些服务来启动或停止语音控制。

总体而言，这段代码通过结合Vosk语音识别库和ROS的消息传递机制，实现了一个灵活且高效的语音控制节点。它能够实时录制和处理音频数据，并将识别结果发布到ROS系统中，供其他节点使用。通过多线程和信号量技术，代码能够高效地管理音频数据的生产和消费，确保语音控制的连续性和准确性。这种设计使得代码易于扩展和维护，适合用于复杂的机器人应用场景。

## 3.2 检测跟踪模块的原理和实现

这段代码实现了一个基于ROS和YOLOv11模型的实时目标检测与跟踪系统，主要用于机器人对特定目标（如人或其他物体）的识别、定位和跟随。其核心原理是通过订阅RGB和深度图像数据，利用YOLOv11模型进行目标检测，并根据检测结果调整机器人的运动，使其能够跟随目标或搜索目标。代码的设计体现了模块化和多线程的思想，能够高效地处理图像数据和机器人控制任务，并通过ROS服务提供启动和停止检测的功能。

首先，代码初始化了一个名为yolov11\_realtime\_detector的ROS节点，并创建了一个YOLOv11RealtimeDetector类的实例。在YOLOv11RealtimeDetector类的构造函数中，代码加载了预训练的YOLOv11模型，并初始化了用于图像转换的CvBridge对象。YOLOv11是一个高效的目标检测模型，能够在实时场景中快速识别多种物体。代码还订阅了/camera/rgb/image\_raw和/camera/depth/image\_raw主题，分别用于获取RGB图像和深度图像数据。RGB图像用于目标检测，而深度图像用于计算目标与机器人之间的距离。

在rgb\_callback和depth\_callback方法中，代码将ROS图像消息转换为OpenCV格式，并分别存储在rgb\_image和depth\_image变量中。这些图像数据随后被传递到process\_images方法中进行处理。在process\_images方法中，代码使用YOLOv11模型对RGB图像进行目标检测，并将检测结果可视化后发布到/detection\_result主题中，供其他节点使用。检测结果包括目标物体的类别、位置和边界框信息。

如果系统处于搜索模式（searching为True），代码会调用search方法，使机器人原地旋转，直到检测到目标物体。搜索模式通常用于目标丢失后的重新定位。如果系统处于跟踪模式（searching为False），代码会调用follow方法，根据目标物体的位置调整机器人的朝向和运动。具体来说，代码会计算目标物体与图像中心的偏差，并根据偏差调整机器人的旋转速度。如果目标物体位于图像中心附近，代码会根据深度图像中的距离值控制机器人的前进或停止。

为了支持多任务处理，代码使用了多线程技术。在start\_capture方法中，代码启动了一个新的线程来处理图像数据和机器人控制任务。该线程通过一个循环不断调用process\_images方法，并根据检测结果调整机器人的运动。当外部节点调用start\_capture服务时，代码会根据请求参数启动或停止检测任务。

总体而言，这段代码通过结合YOLOv11目标检测模型和ROS的消息传递机制，实现了一个灵活且高效的实时目标检测与跟踪系统。它能够根据图像数据识别目标物体，并通过调整机器人的运动实现目标跟随或搜索。通过多线程技术，代码能够高效地处理图像数据和机器人控制任务，确保系统的实时性和响应性。这种设计使得代码易于扩展和维护，适合用于复杂的机器人应用场景。

## 3.2 机械臂模块的原理和实现

这段代码实现了一个基于ROS的机器人手臂控制节点，其核心功能是通过发布关节控制指令来控制机器人手臂的运动，包括抓取、释放和复位等操作。代码的设计简洁明了，通过ROS服务提供对外接口，使得其他节点可以方便地调用这些功能。

首先，代码初始化了一个名为robot\_arm\_agent的ROS节点，并创建了一个RobotArmAgent类的实例。在RobotArmAgent类的构造函数中，代码初始化了多个关节控制器的发布器，分别用于控制机器人手臂的倾斜关节（tilt\_joint）和四个旋转关节（joint1\_joint到joint4\_joint）。这些发布器将关节角度指令发布到相应的ROS主题中，从而驱动机器人手臂的运动。代码还定义了一些关键关节角度值，例如zero\_point（初始位置）、horizon\_point（水平位置）和seized\_point（抓取位置），这些值用于控制手臂的具体运动。

在Seize方法中，代码通过发布seized\_point角度值来控制joint4\_joint关节，使机器人手臂执行抓取动作。如果手臂当前未处于抓取状态（seized为False），代码会发布抓取指令，并等待1秒以确保动作完成，然后将 `seized` 状态设置为True。在Release方法中，代码通过发布zero\_point角度值来控制joint4\_joint关节，使机器人手臂执行释放动作。如果手臂当前处于抓取状态（seized为True），代码会发布释放指令，并等待1秒以确保动作完成，然后将seized状态设置为False。

在Reset方法中，代码通过发布zero\_point和horizon\_point角度值来控制所有关节，使机器人手臂恢复到初始状态。这一过程包括将倾斜关节和旋转关节的角度设置为初始值，并等待3秒以确保所有动作完成。`Reset` 方法通常在节点启动时调用，以确保机器人手臂处于已知的初始位置。

代码还提供了两个ROS服务，分别用于处理抓取和释放请求。在handle\_seize方法中，代码根据请求参数调用Seize方法，并返回操作结果。在handle\_release方法中，代码根据请求参数调用Release方法，并返回操作结果。这些服务通过seize和release主题对外提供，其他节点可以通过调用这些服务来控制机器人手臂的抓取和释放动作。

在程序的主逻辑中，代码初始化了RobotArmAgent实例，并注册了seize和release服务。随后，代码调用Reset方法将机器人手臂复位到初始位置，并记录日志信息以表明服务已准备就绪。最后，代码进入ROS的主循环rospy.spin()，等待外部请求并处理相应的操作。

总体而言，通过结合ROS的消息传递机制和服务机制，实现了一个灵活且高效的机器人手臂控制节点。它能够根据外部请求执行抓取、释放和复位等操作，并通过ROS服务提供对外接口。这种设计使得代码易于扩展和维护，适合用于复杂的机器人应用场景。

# 四、个人完成的主要工作

## 4.1 主要工作一：完成realtime\_capture.py的编写

realtime\_capture.py最初的设计是要做一个实时目标检测的模块。实时目标检测的功能是这样实现的：首先通过调用cvbridge来获取摄像机节点捕捉的信息，也就是一帧一帧的图像，然后使用YOLO进行目标检测，YOLO模型会返回检测框的位置数据，在原图像的基础上画出检测框，就能实现基本的物体检测效果。

不过考虑到仅仅让摄像机进行实时目标检测似乎并没有什么直接的意义遂在此基础上进行了改进，我们希望机器人能够根据视觉检测结果来跟踪一个目标物体，但是由于摄像头的参数测定比较麻烦，如果希望通过视觉图像来估计物体的位置进行导航，那么必然要进行大量的实验进行几何计算，例如当前物体位于机器人的哪个方位，多少角度，这些都与摄像头的参数有关，为了避免这个问题，我们将跟踪分为了两步：首先原地转向直至物体落于摄像框的正中央，这一步是很容易办到的，只要让机器人在原地按固定方向固定速度进行旋转直到目标框的几何中心落在整个图像的中央即可停止旋转；当确定目标物体已经落在摄像框的正中央时，那么此时机器人一定是正对着物体，此时机器人只需要直行就可以抵达目标，而正对着目标的距离可以通过调用深度摄像头来获取。先旋转再直行的跟踪好处有两个，一是不需要通过摄像机参数的几何计算来估计目标物体的方位，二是也不需要考虑深度摄像机的激光发射角度。

下面我们结合具体的代码进行分析

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | **def** \_\_init\_\_(self):      """      初始化 YOLOv11 实时检测器      """      self.model\_path **=** '/home/ubuntuber/ros\_se/yolo11n.pt'      self.rgb\_topic **=** '/camera/rgb/image\_raw'      self.depth\_topic **=** '/camera/depth/image\_raw'      self.bridge **=** CvBridge()      self.image\_sub **=** None      self.depth\_sub **=** None      self.model **=** YOLO(self.model\_path)      self.cmd\_vel\_pub **=** rospy.Publisher('/cmd\_vel\_mux/input/navi', Twist, queue\_size**=**10)  # 发布机器人运动指令      self.target\_object **=** 'person'  # 默认目标物体      self.forward **=** True      self.searching **=** False  # 是否正在寻找目标      self.target\_lost **=** False  # 目标是否丢失      self.running\_state **=** False      self.follow\_thread **=** None      self.default\_target **=** False      self.image\_pub **=** rospy.Publisher('/detection\_result', Image, queue\_size**=**10)      rospy.loginfo("YOLOv11 Realtime Detector initialized.")    **def** rgb\_callback(self, data):      """      RGB 图像回调函数      :param data: ROS 图像消息      """  **try**:          # 将 ROS 图像消息转换为 OpenCV 图像          self.rgb\_image **=** self.bridge.imgmsg\_to\_cv2(data, "bgr8")  **except** CvBridgeError as e:          rospy.logerr("CvBridge Error: {}".format(e))    **def** depth\_callback(self, data):      """      深度图像回调函数      :param data: ROS 深度图像消息      """  **try**:          # 将 ROS 深度图像消息转换为 OpenCV 图像          self.depth\_image **=** self.bridge.imgmsg\_to\_cv2(data, "passthrough")  **except** CvBridgeError as e:          rospy.logerr("CvBridge Error: {}".format(e)) |

初始化阶段，首先加载预训练的YOLOv11模型（模型文件路径为/home/ubuntuber/ros\_se/yolo11n.pt），并创建ROS节点作为核心通信单元。通过订阅/camera/rgb/image\_raw和/camera/depth/image\_raw话题，分别接收RGB彩色图像和深度图像数据。同时，使用CvBridge工具将ROS图像消息转换为OpenCV格式，以便后续处理。机器人运动控制指令通过/cmd\_vel\_mux/input/navi话题发布，其消息类型为geometry\_msgs/Twist，包含线速度和角速度参数。目标物体的默认设置为“person”，但系统支持运行时动态修改。为确保实时性，图像处理逻辑以10Hz的频率在主循环中运行，并支持多线程处理外部控制请求。

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65 | **def** start\_capture(self):      """      开始实时检测      """      rospy.loginfo('Start capture, subscribing...')  **try**:          self.image\_sub **=** rospy.Subscriber(self.rgb\_topic, Image, self.rgb\_callback)          self.depth\_sub **=** rospy.Subscriber(self.depth\_topic, Image, self.depth\_callback)          rospy.loginfo('Subscribed successfully.')  **except** Exception as e:          rospy.logerr('Failed to subscribe: {}'.format(e))  **return**      # 等待用户输入目标物体名称  **if** self.default\_target:          self.target\_object **=** input('Enter the name of the object to follow (or press Enter to skip): ')  **else**: rospy.sleep(2)  **if** self.target\_object:          rospy.loginfo(f'Tracking object: {self.target\_object}')      # 开始寻找目标物体      self.searching **=** True      # 主循环      rate **=** rospy.Rate(10)  # 10 Hz      rospy.loginfo("rospy status: %s" **%**rospy.is\_shutdown())  **while** **not** rospy.is\_shutdown():          self.process\_images()          rate.sleep()  **if** **not** self.running\_state:  **break**      # 停止订阅并关闭窗口  **if** self.image\_sub:          self.image\_sub.unregister()  **if** self.depth\_sub:          self.depth\_sub.unregister()      cv2.destroyAllWindows()      rospy.loginfo('Capture stopped and windows closed.')  **def** process\_images(self):      """      处理 RGB 和深度图像      """    **if** self.rgb\_image **is** None **or** self.depth\_image **is** None:          rospy.logerr("No image")  **return**      # 显示实时 RGB 图像        # cv2.imshow("Live RGB Camera Feed", self.rgb\_image)      # rospy.loginfo('here')      # 使用 YOLOv8 模型进行检测      results **=** self.model(self.rgb\_image, verbose**=**False)      annotated\_image **=** results[0].plot()      # 显示检测结果      # cv2.imshow("Matching Result", annotated\_image)  **try**:          ros\_image **=** self.bridge.cv2\_to\_imgmsg(annotated\_image, "bgr8")          self.image\_pub.publish(ros\_image)  **except** CvBridgeError as e:          rospy.logerr("CvBridge Error: {}".format(e))      # 如果正在寻找目标，调用 search 方法  **if** self.searching:          self.search(results)      # 如果指定了目标物体，则调用 follow 方法  **elif** self.target\_object:          self.follow(results)      # 等待 1 毫秒以刷新图像窗口      # cv2.waitKey(1) |

当RGB图像到达时，rgb\_callback函数将ROS图像消息转换为OpenCV的BGR格式矩阵，并存储为类属性。深度图像通过类似的depth\_callback转换为浮点型矩阵保存。两者共同构成机器人的视觉输入。在process\_images方法中，系统首先检查图像的有效性，随后调用YOLOv11模型对RGB图像进行目标检测。检测结果通过results[0].plot()生成带有边界框和标签的标注图像，再通过OpenCV转换为ROS图像消息并发布到/detection\_result话题，供其他节点可视化使用。此过程中，YOLOv11模型的检测结果包括目标的类别名称、边界框坐标和置信度，为后续跟踪逻辑提供基础数据。

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79 | **def** follow(self, results):      """      跟踪指定物体并调整机器人朝向，然后向前移动      :param results: YOLOv8 检测结果      """      target\_found **=** False  # 当前帧是否检测到目标物体  **for** result **in** results:          boxes **=** result.boxes  # 获取检测框  **for** box **in** boxes:              class\_id **=** int(box.cls)  # 类别 ID              class\_name **=** result.names[class\_id]  # 类别名称              # 如果检测到目标物体  **if** class\_name **==** self.target\_object:                  target\_found **=** True                  x1, y1, x2, y2 **=** map(int, box.xyxy[0])  # 获取检测框坐标                  center\_x **=** (x1 **+** x2) **//** 2  # 计算物体中心 X 坐标                  center\_y **=** (x1 **+** y2) **//** 2  # 计算物体中心 Y 坐标                  # 计算物体中心与图像中心的偏差                  image\_center\_x **=** self.rgb\_image.shape[1] **/** 2                  offset\_x **=** center\_x **-** image\_center\_x                  # 如果偏差较大，调整机器人朝向  **if** abs(offset\_x) > 50:                      twist **=** Twist()                      twist.angular.z **=** **-**0.5 **if** offset\_x > 0 **else** 0.5  # 物体在右侧则右转，左侧则左转                      self.cmd\_vel\_pub.publish(twist)                      rospy.loginfo(f"Turning to align with {self.target\_object}")  **return**  # 退出函数，继续调整朝向                  # 如果偏差较小，开始向前移动  **else**:                      # 获取深度图像中物体中心的深度值                      depth\_value **=** self.depth\_image[center\_y, center\_x]                      rospy.loginfo(f"Distance to {self.target\_object}: {depth\_value} meters")  **if** self.forward:                          # 如果距离大于阈值，向前移动  **if** depth\_value > 0.5:  # 假设距离阈值为 0.5 米                              twist **=** Twist()                              twist.linear.x **=** 0.15  # 设置线速度                              self.cmd\_vel\_pub.publish(twist)                              rospy.loginfo(f"Moving towards {self.target\_object}")  **else**:                              # 如果距离小于阈值，停止移动                              twist **=** Twist()                              self.cmd\_vel\_pub.publish(twist)                              rospy.loginfo(f"Reached close to {self.target\_object}")  **return**      # 如果未检测到目标物体  **if** **not** target\_found:  **if** **not** self.target\_lost:              rospy.loginfo(f"{self.target\_object} lost, starting search...")              self.target\_lost **=** True              self.searching **=** True  # 重新开始寻找  **else**:          self.target\_lost **=** False  # 目标重新找到      # 如果未检测到目标物体，停止机器人      twist **=** Twist()      self.cmd\_vel\_pub.publish(twist)      rospy.loginfo("No target object detected.")    **def** search(self, results):      """      寻找目标物体：原地旋转并检查是否存在目标物体      :param results: YOLOv8 检测结果      """  **for** result **in** results:          boxes **=** result.boxes  # 获取检测框  **for** box **in** boxes:              class\_id **=** int(box.cls)  # 类别 ID              class\_name **=** result.names[class\_id]  # 类别名称              # 如果检测到目标物体  **if** class\_name **==** self.target\_object:                  # rospy.loginfo(f"Found {self.target\_object}, stopping rotation.")                  self.searching **=** False  # 停止寻找                  self.target\_lost **=** False  # 目标重新找到  **return**  # 退出函数，进入跟踪模式      # 如果未检测到目标物体，继续旋转      twist **=** Twist()      twist.angular.z **=** 0.3  # 设置旋转速度（正值为逆时针，负值为顺时针）      self.cmd\_vel\_pub.publish(twist)      rospy.loginfo("Rotating to search for target object...") |

在search方法中，若目标丢失，机器人将进入持续旋转状态，通过发布固定的角速度（0.3rad/s）使其原地转动。旋转过程中，每帧的YOLO检测结果会被持续检查，一旦目标重新进入视野，立即终止旋转并恢复跟踪模式。搜索逻辑确保机器人在目标暂时移出视野时能够自主重新定位，避免陷入静止等待状态。

此外我们还引入了多线程管理的方式。系统通过ROS服务start\_capture（服务类型为std\_srvs/SetBool）实现外部启停控制。当服务请求的data字段为True时，handle\_start\_capture函数启动新线程运行start\_capture方法，开启图像订阅与处理循环；反之则停止线程并释放资源。线程管理通过threading模块实现，确保主节点在rospy.spin()中保持响应性，同时处理服务请求。用户还可通过动态修改target\_object属性切换跟踪目标，扩展了系统的灵活性。

在代码主函数中，ROS节点yolov11\_realtime\_detector被初始化，并创建YOLOv11RealtimeDetector实例。服务start\_capture被注册后，节点进入等待状态，监听外部服务请求。用户可通过ROS命令行工具（如rosservice call）或与其他ROS节点集成的方式触发跟踪任务。整个生命周期中，节点的日志输出（如目标检测状态、运动指令详情）通过rospy.loginfo和rospy.logerr记录，便于调试与监控。

## 4.2 主要工作二：代码调试与实机测试

由于本人主要负责realtime\_capture.py的代码编写，有关代码的调试也基本由本人负责。实际调试的过程，遇到了如下问题：

首先是动画显示动画更新速度过慢，经过分析之后发现是因为ultralytics包的YOLO在每一帧检测的时候都会在终端输出大量调试信息，而终端的输出系统调用实际上是一个非常复杂的系统调用，当输出信息过多时就会导致大量占用cpu，画面检测速度低下，修改YOLO模型的属性使其不输出调试信息即可。

其次是跟踪算法的设计，最初我们并没有考虑目标从摄像框消失的情况，直接消失会导致机器直接原地发呆直到目标重新出现。为了避免这种情况，我又重新设计了一个标志位target\_lost用来表示目标突然消失，当该标志置位时，机器人会重新执行search函数开始原地旋转，直到目标重新出现在视野内。

第三个遇到的问题是最初我们始终是使用freenect系列控件来控制摄像头，当我们尝试为项目引入amcl导航的时候，发现这两个包会由于竞争kinect摄像头资源而发生冲突，进而导致整个程序无法执行。根据我们反复的尝试和实验，发现amcl的包其实也可以启动摄像头，但是深度摄像机话题订阅要从/camera/depth-registered/image变为/camera/depth/image-raw，经过这样的修正之后我们就不需要重复启动freenect的相关包去竞争摄像头资源了。

最后还遇到一个问题是opencv窗口的问题，经过我们实验，发现opencv的imshow窗口在Linux平台下，如果调用了cv2.DestroyAllWindows函数之后，再次调用imshow函数窗口无法正常启动，网上查阅资料发现这是一个历史遗留的bug，所以这里我们将原来所有的cv2窗口都去掉了，改为直接只用image-view节点来窗口显示。

# 五、项目总结与展望

## 5.1 项目总结

本项目基于ROS（机器人操作系统）构建了一个多功能机器人控制系统，涵盖了闲逛进程、导航模块、语音模块、检测跟踪模块和机械臂模块等多个功能模块，实现了机器人从自主移动、目标检测与跟踪、语音控制到机械臂操作的完整功能闭环。通过模块化设计和多线程技术的应用，系统不仅具备高效的任务处理能力，还具有良好的扩展性和可维护性，能够适应复杂的机器人应用场景。

在闲逛进程中，系统通过订阅主题和调用服务的方式，实现了对机器人多种任务的管理与执行。无论是通过键盘输入还是语音指令，系统都能够根据命令内容调用相应的ROS服务，例如启动目标检测、控制机械臂抓取物体或执行导航任务。语音控制模块通过Vosk语音识别模型将音频转换为文本指令，并将其发布到ROS系统中，进一步丰富了人机交互的方式。导航模块则利用ROS的`move\_base`动作服务器，实现了机器人在已知地图中的自主导航功能。通过预设的目标位置字典，系统能够快速定位目标并完成导航任务，同时支持多线程技术以提升任务执行的效率。

检测跟踪模块是系统的核心功能之一，它基于YOLOv11目标检测模型，结合RGB和深度图像数据，实现了对特定目标的实时识别与跟踪。系统能够根据目标物体的位置调整机器人的朝向和运动，确保目标始终处于视野范围内。当目标丢失时，系统会自动进入搜索模式，通过原地旋转重新定位目标。机械臂模块则通过发布关节控制指令，实现了机器人手臂的抓取、释放和复位等操作。通过ROS服务提供的对外接口，其他节点可以方便地调用这些功能，进一步增强了系统的灵活性和实用性。

整个系统在设计和实现过程中，充分考虑了模块之间的协作与数据交互。例如，语音模块识别的指令可以直接触发导航模块或机械臂模块的任务执行，而检测跟踪模块的识别结果则可以为导航模块提供目标位置信息。这种模块化的设计不仅降低了系统的耦合度，还使得每个模块可以独立开发和测试，极大地提高了开发效率和系统的稳定性。

总体而言，本项目通过结合ROS的消息传递机制、服务机制和动作服务器，构建了一个功能丰富、性能高效的机器人控制系统。无论是从技术实现的深度，还是从功能覆盖的广度来看，该系统都展现了较高的实用性和创新性。通过进一步优化和扩展，该系统有望在更多实际场景中得到应用，例如智能家居、工业自动化以及服务机器人等领域。

## 5.2 项目展望

展望未来，本项目所构建的机器人控制系统具有广阔的应用前景和发展潜力。随着人工智能、机器人技术和物联网的快速发展，该系统可以在多个领域进一步优化和扩展，以满足日益复杂和多样化的需求。

首先，在智能家居领域，该系统可以作为家庭服务机器人的核心控制平台。通过进一步集成环境感知、语音交互和自主导航功能，机器人可以承担更多的家务任务，例如清洁、物品搬运和安防监控。结合智能家居设备，机器人还可以实现与其他设备的联动，例如根据用户指令调节灯光、温度或播放音乐，从而为用户提供更加便捷和智能的生活体验。

其次，在工业自动化领域，该系统可以应用于生产线上的物料搬运、设备巡检和质量检测等任务。通过优化目标检测和机械臂控制算法，机器人可以更高效地完成复杂的操作任务，例如抓取不规则物体或进行高精度装配。此外，结合5G通信技术和边缘计算，机器人可以实现实时数据传输和远程控制，进一步提升工业生产的自动化水平和效率。

在服务机器人领域，该系统可以扩展为商场导购、酒店服务和医疗辅助等应用场景。通过增强语音交互和情感识别功能，机器人可以更好地理解用户需求并提供个性化的服务。例如，在商场中，机器人可以为顾客提供商品推荐和路线导航；在酒店中，机器人可以承担客房服务、行李搬运和前台接待等任务；在医疗场景中，机器人可以协助医护人员进行药品配送、病人护理和远程会诊，从而减轻医护人员的工作负担。

此外，随着深度学习技术和计算机视觉算法的不断进步，系统的目标检测和跟踪能力可以进一步提升。例如，引入更先进的神经网络模型和多传感器融合技术，可以提高目标识别的准确性和鲁棒性，尤其是在复杂环境和动态场景中。同时，通过强化学习算法，机器人可以逐步优化其决策和控制策略，从而在未知环境中实现更加自主和智能的行为。

最后，系统的模块化设计和开放接口为未来的功能扩展提供了便利。开发者可以根据具体需求，快速集成新的功能模块或替换现有模块，例如增加手势识别、情感分析或多机器人协作功能。这种灵活性和可扩展性使得系统能够适应不同应用场景的需求，并为未来的技术创新提供了坚实的基础。

综上所述，本项目所构建的机器人控制系统不仅具有广泛的应用前景，还可以通过持续的技术优化和功能扩展，为智能家居、工业自动化、服务机器人等领域带来更多的创新和价值。未来，随着技术的不断进步和应用场景的不断拓展，该系统有望成为智能化社会的重要组成部分，为人们的生活和工作带来更多便利和可能性。