实践项目: 仓库管理员 (Warehouse Runner)

研习报告

姓名: 卢祥云
学号:1120230944
1120230944@bit.edu.cn

2025 年 12 月 5 日

目录

1 系统概述与环境介绍

本部分将详细介绍本项目所使用的核心技术栈, 包括 Nav2 导航框架、TIAGo 机器人平台、仿真环境构建以及项目启动脚本的功能说明。

1.1 Nav2 导航框架详解

1.1.1 Nav2 概述

Nav2(Navigation2) 是 ROS 2 中的官方导航堆栈, 是 ROS 1 中 Navigation Stack 的完全重写版本。Nav2 为移动机器人提供了一套完整的导航解决方案, 包括感知、定位、路径规划、运动控制以及恢复行为等功能模块。

Nav2 的设计理念强调模块化和可扩展性, 所有核心功能都以插件形式实现, 用户可以根据需要替换默认组件或添加自定义功能。这种架构使得 Nav2 能够适应从简单的室内机器人到复杂的室外自主系统等各种应用场景。

1.1.2 Nav2 系统架构

Nav2 采用分层架构设计, 主要包含以下核心组件:

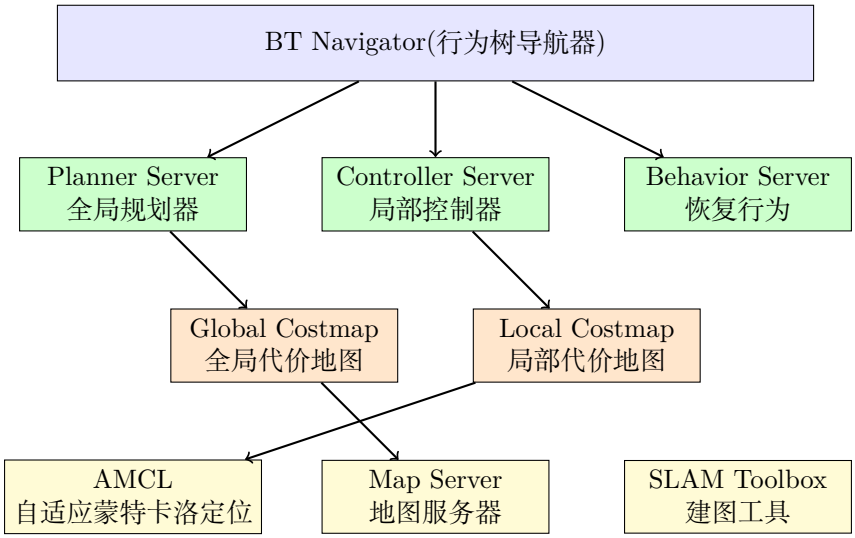

图 1: Nav2 系统架构示意图

1. 行为树导航器 (BT Navigator) BT Navigator 是 Nav2 的核心协调模块, 使用行为树 (Behavior Tree) 来管理导航任务的执行流程。行为树相比传统的有限状态机具有以下优势:

- **模块化**: 每个节点独立封装特定行为
- **可复用性**: 子树可以在不同场景中重复使用
- **易于调试**: 树状结构便于理解和追踪执行流程
- **动态切换**: 支持运行时动态修改行为逻辑

Nav2 默认使用的行为树配置文件为 `navigate_w_replanning_and_recovery.xml`, 该配置实现了带重规划和恢复机制的导航流程。

2. 全局路径规划器 (Planner Server) Planner Server 负责计算从当前位置到目标点的全局路径。Nav2 提供了多种规划算法插件:

表 1: Nav2 全局规划器对比

规划器	算法	特点
NavFn Planner	Dijkstra/A*	经典算法, 稳定可靠
Smac Planner 2D	A* 变体	支持更多启发式
Smac Hybrid-A*	Hybrid A*	考虑运动学约束
Smac Lattice	State Lattice	适用于阿克曼模型
Theta* Planner	Theta*	产生更平滑路径

在本项目中, 默认使用 NavFn Planner, 其配置如下:

```
planner_server:
  ros__parameters:
    planner_plugins: ["GridBased"]
    GridBased:
      plugin: "nav2_navfn_planner/NavfnPlanner"
      tolerance: 0.5
      use_astar: false
      allow_unknown: true
```

3. 局部控制器 (Controller Server) Controller Server 负责跟踪全局路径并生成实时速度指令。本项目涉及的控制器包括:

(a) DWB 控制器 (Dynamic Window Based)

DWB 是基于动态窗口法的控制器, 通过在速度空间中采样可行轨迹并评分来选择最优控制。主要参数包括:

- `max_vel_x`: 最大前向速度 (本项目设置为 1.0 m/s)
- `max_vel_theta`: 最大角速度 (1.0 rad/s)
- `sim_time`: 轨迹仿真时间 (1.7s)
- `vx_samples`, `vtheta_samples`: 速度采样数量

DWB 使用多个评价函数 (Critics) 对轨迹进行综合评分:

```
critics: ["RotateToGoal", "Oscillation", "BaseObstacle",
          "GoalAlign", "PathAlign", "PathDist", "GoalDist"]
```

(b) RPP 控制器 (Regulated Pure Pursuit)

RPP 是一种改进的纯追踪控制器, 增加了速度调节机制:

- 根据路径曲率自动降速
- 接近障碍物时减速
- 接近目标点时平滑减速
- 前瞻距离 (Lookahead Distance) 可配置

(c) MPPI 控制器 (Model Predictive Path Integral)

MPPI 是基于模型预测和路径积分的高级控制器:

- 使用蒙特卡洛采样预测多条轨迹
- 根据代价函数对轨迹加权平均
- 支持复杂的代价函数设计
- 计算量较大但效果更优

4. 代价地图 (Costmap) 代价地图是 Nav2 进行路径规划和避障的基础。系统维护两种代价地图:

全局代价地图 (Global Costmap):

- 覆盖整个已知环境
- 用于全局路径规划
- 包含静态地图层、障碍物层、膨胀层

局部代价地图 (Local Costmap):

- 以机器人为中心的滚动窗口
- 用于局部避障和控制
- 实时更新传感器观测

关键参数——膨胀层 (Inflation Layer):

```
inflation_layer:
  plugin: "nav2_costmap_2d::InflationLayer"
  cost_scaling_factor: 3.0  # 代价衰减因子
  inflation_radius: 0.55    # 膨胀半径
```

5. 定位系统 (AMCL) AMCL(Adaptive Monte Carlo Localization) 使用粒子滤波算法进行机器人定位。核心原理是:

1. 初始化: 在地图上分布大量粒子 (假设位姿)

2. 预测: 根据里程计信息移动粒子
3. 更新: 根据激光扫描与地图匹配程度为粒子赋权重
4. 重采样: 保留高权重粒子, 淘汰低权重粒子

主要配置参数:

```
amcl:
  ros__parameters:
    min_particles: 500
    max_particles: 2000
    laser_model_type: "likelihood_field"
    robot_model_type: "nav2_amcl::DifferentialMotionModel"
```

6. 恢复行为 (Behavior Server) 当导航陷入困境时, 恢复行为模块会尝试使机器人脱困:

- `Spin`: 原地旋转, 扫描周围环境
- `BackUp`: 后退指定距离
- `DriveOnHeading`: 沿当前方向行驶
- `Wait`: 等待障碍物移开

1.2 TIAGo 机器人详解

1.2.1 TIAGo 机器人概述

TIAGo 是由 PAL Robotics 公司开发的服务机器人平台, 专为研究和商业应用设计。该机器人具有高度模块化的设计, 可根据需求配置不同的硬件组件。

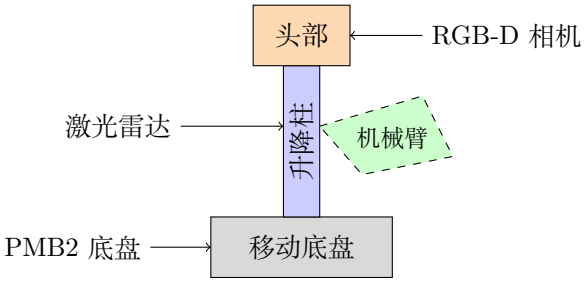

图 2: TIAGo 机器人结构示意图

1.2.2 TIAGo 硬件组成

1. 移动底盘 (PMB2 Base) TIAGo 采用 PMB2 差速驱动底盘, 具有以下特性:

- **驱动方式**: 差速驱动 (Differential Drive)
- **最大速度**: 前进 1.0 m/s, 旋转 1.0 rad/s
- **尺寸**: 直径约 54cm, 高度约 98cm(不含手臂)
- **机器人半径**:0.275m(用于导航避障)

差速驱动运动学模型:

$$\begin{cases} v = \frac{v_r + v_l}{2} \\ \omega = \frac{v_r - v_l}{L} \end{cases} \tag{1}$$

其中 v_r 和 v_l 分别为左右轮速度,L 为轮距。

2. 激光雷达 (LiDAR) TIAGo 配备 2D 激光雷达用于导航和避障:

- 安装位置: 底盘前部 (base_laser_link)
- 相对底盘偏移:x=0.202m, z=-0.004m
- 扫描范围: 通常为 270 度或 360 度
- 最大探测距离:20m
- 数据话题:`/scan`

3. RGB-D 相机 头部配备深度相机用于 3D 感知:

- 可选型号:Orbbec Astra、Intel RealSense 等
- 功能: 物体识别、3D 建图、人机交互

4. 机械臂 (可选) TIAGo 可配置 7 自由度机械臂:

- 末端执行器:PAL Gripper、Hey5 灵巧手、Robotiq 夹爪等
- 运动规划: 集成 MoveIt2 框架
- 本项目使用无臂版本 (no-arm) 以简化导航测试

1.2.3 TIAGo 软件架构

本项目中 TIAGo 相关的 ROS 2 软件包结构如下:

```
robot/
|-- tiago_robot/                    # TIAGo核心包
|   |-- tiago_description/          # 模型描述
|   |   |-- models/tiago/           # Gazebo SDF模型
|   |   |-- models/tiago_no_arm/    # 无臂版SDF模型
|   |   |-- urdf/                   # URDF描述文件
|   |   |-- meshes/                 # 3D网格文件
|   |   |-- robots/                 # 机器人配置
|   |-- tiago_bringup/              # 启动文件
|   |-- tiago_controller_configuration/
|-- tiago_navigation/               # 导航相关包
|   |-- tiago_2dnav/                # 2D导航配置
|   |-- tiago_laser_sensors/        # 激光传感器配置
|   |-- tiago_rgbd_sensors/         # RGBD传感器配置
|-- tiago_simulation/               # 仿真相关包
|   |-- tiago_gazebo/               # Gazebo仿真配置
|   |-- tiago_multi/                # 多机器人仿真
|-- tiago_moveit_config/            # MoveIt运动规划配置
|-- pal_navigation_cfg_public/      # PAL导航参数
|   |-- pal_navigation_cfg_params/
|       |-- params/tiago_nav2.yaml  # Nav2参数文件
```

1.2.4 TIAGo 坐标系 (TF Tree)

TIAGo 的坐标系结构遵循 REP-105 标准:

```
map
  |-- odom
        |-- base_footprint
              |-- base_link
                    |-- base_laser_link
                    |-- torso_lift_link
                          |-- head_*_link
                          |-- arm_*_link (如果有)
```

各坐标系说明:

- `map`: 全局固定坐标系, 由 AMCL 发布
- `odom`: 里程计坐标系, 连续但会漂移
- `base_footprint`: 机器人底盘投影到地面
- `base_link`: 底盘中心
- `base_laser_link`: 激光雷达坐标系

1.2.5 在本项目中使用 TIAGo

本项目通过以下方式使用 TIAGo 机器人:

1. 模型加载 在 Gazebo 中加载 TIAGo 的 SDF 模型:

```
ros2 run ros_gz_sim create \
  -name tiago \
  -file tiago_description/models/tiago/model.sdf \
  -x 0 -y 0 -z 0.05
```

2. 话题桥接 通过 ros_gz_bridge 将 Gazebo 话题桥接到 ROS 2:

```
ros2 run ros_gz_bridge parameter_bridge \
  /clock@rosgraph_msgs/msg/Clock[ignition.msgs.Clock \
  /cmd_vel@geometry_msgs/msg/Twist]ignition.msgs.Twist \
  /odom@nav_msgs/msg/Odometry[ignition.msgs.Odometry \
```

```
/scan@sensor_msgs/msg/LaserScan[ignition.msgs.LaserScan
```

3. 控制接口

- 速度指令话题:/cmd_vel
- 里程计数据:/odom
- 激光扫描数据:/scan
- 关节状态:/joint_states

1.3 仿真环境与地图构建

1.3.1 仓库仿真环境

本项目使用 Gazebo Sim(Ignition Gazebo) 构建仓库仿真环境。仿真世界定义文件位于:

```
map/turtlebot4_gz_bringup/worlds/warehouse.sdf
```

环境构成 仓库环境包含以下主要元素:

表 2: 仓库环境元素

元素类型	数量	来源
仓库主体	1	OpenRobotics/Warehouse
大型货架 (shelf_big)	5	MovAi/shelf_big
小型货架 (shelf)	8	MovAi/shelf
护栏 (barrier)	4	OpenRobotics/Jersey Barrier
椅子等障碍物	若干	OpenRobotics/Chair

物理引擎配置

```
<physics type="ode">
  <max_step_size>0.003</max_step_size>
  <real_time_factor>1.0</real_time_factor>
</physics>
```

1.3.2 SLAM 建图过程

SLAM Toolbox 简介 本项目使用 SLAM Toolbox 进行 2D 激光 SLAM 建图。SLAM Toolbox 是 ROS 2 中功能强大的 2D SLAM 解决方案, 支持:

- 在线同步/异步建图
- 离线地图编辑
- 地图序列化与继续建图
- 回环检测与图优化

建图流程

1. **启动仿真环境**: 加载 Gazebo 世界和 TIAGo 机器人
2. **启动 SLAM 节点**: 运行 slam_toolbox 的 sync_slam_toolbox_node
3. **遥控建图**: 使用 teleop_ui.py 控制机器人遍历环境
4. **保存地图**: 调用 save_map.sh 保存建图结果

SLAM 配置参数 关键 SLAM 参数配置:

```
slam_toolbox:
  ros__parameters:
    odom_frame: odom
    map_frame: map
    base_frame: base_footprint
    scan_topic: /scan
    resolution: 0.05          # 地图分辨率 5cm/像素
    max_laser_range: 20.0     # 最大激光探测距离
    minimum_travel_distance: 0.3  # 触发扫描匹配的最小移动距离
    minimum_travel_heading: 0.3   # 触发扫描匹配的最小旋转角度
    do_loop_closing: true     # 启用回环检测
```

地图文件格式 SLAM 建图完成后生成以下文件:

- `*.pgm`: 灰度图像格式的占据栅格地图
- `*.yaml`: 地图元数据文件, 包含分辨率、原点等信息
- `*_slam.posegraph`:SLAM 序列化文件, 可用于继续编辑

地图 YAML 文件示例:

```yaml
image: warehouse.pgm
resolution: 0.050000    # 每像素代表0.05米
origin: [-50.0, -50.0, 0.0]  # 地图原点在世界坐标系中的位置
negate: 0
occupied_thresh: 0.65   # 占据概率阈值
free_thresh: 0.196      # 空闲概率阈值
```

1.4 遥控 UI 控制方法

本项目开发了一个基于 Tkinter 的图形化遥控界面 (teleop_ui.py), 用于在 SLAM 建图和导航测试过程中手动控制 TIAGo 机器人。

1.4.1 UI 界面

图 3: TIAGo 遥控 UI 界面

1.4.2 控制方式

UI 提供键盘和按钮两种等效的控制方式:

表 3: 遥控按键说明

按键	按钮	功能
W	W	加速 (线速度 +0.1 m/s, 最大 1.2 m/s)
S	S	减速/后退 (线速度-0.1 m/s, 最小-0.6 m/s)
A	A	左转 (角速度 1.2 rad/s, 松开停止转向)
D	D	右转 (角速度-1.2 rad/s, 松开停止转向)
空格	停	紧急停车 (线速度和角速度归零)

1.4.3 技术实现

遥控 UI 的核心技术特点:

- **多话题发布**: 同时向`/cmd_vel`、`/mobile_base_controller/cmd_vel_unstamped` 和`/model/tiago/cmd_vel` 发布速度指令, 确保兼容不同的仿真配置
- **持续发送**: 以 100ms 间隔持续发送当前速度指令, 保证机器人运动的连续性
- **渐进式速度控制**: 线速度采用步进式调节 (每次 ±0.1 m/s), 便于精细控制
- **实时状态显示**: 界面底部实时显示当前线速度和角速度

1.5 项目脚本功能说明

本项目在 map/目录下提供了四个 Shell 脚本, 用于启动不同功能的仿真环境。

1.5.1 launch_tiago.sh - 基础仿真启动

功能概述 一键启动 Gazebo 仿真环境并加载 TIAGo 机器人模型, 提供基本的遥控功能。

主要功能

1. 启动 Gazebo 仿真器, 加载指定的世界文件
2. 在仿真环境中生成 TIAGo 机器人
3. 启动 ROS-Gazebo 话题桥接
4. 启动遥控 UI 界面 (teleop_ui.py)

使用方法

```
# 默认加载warehouse世界
./launch_tiago.sh

# 指定其他世界
./launch_tiago.sh custom_world

# 无头模式运行
HEADLESS=1 ./launch_tiago.sh

# 指定机器人初始位置
TIAGO_X=1.0 TIAGO_Y=2.0 TIAGO_YAW=1.57 ./launch_tiago.sh
```

表 4: launch_tiago.sh 环境变量

变量名	默认值	说明
WORLD_NAME	warehouse	Gazebo 世界名称
HEADLESS	0	是否无头运行 (0=GUI,1= 无头)
TIAGO_X/Y/Z	0/0/0.05	机器人初始位置
TIAGO_YAW	0	机器人初始朝向
START_BRIDGE	1	是否启动话题桥接
START_UI	1	是否启动遥控 UI

环境变量

1.5.2 launch_tiago_slam.sh - SLAM 建图启动

功能概述 启动完整的 SLAM 建图环境, 包括仿真、传感器数据处理、SLAM 算法和可视化。

启动流程

1. **Gazebo 仿真**: 加载仓库世界和 TIAGo 机器人
2. **话题桥接**: 桥接 clock、cmd_vel、odom、scan、tf 等话题
3. **Robot State Publisher**: 发布机器人 URDF 模型的 TF 变换
4. **静态 TF 发布**: 补充必要的坐标系变换
5. **SLAM Toolbox**: 启动同步 SLAM 节点进行建图
6. **RViz 可视化**: 显示地图、激光扫描和机器人位姿
7. **遥控 UI**: 提供键盘控制界面

使用方法

```
# 启动SLAM建图环境
./launch_tiago_slam.sh

# 使用WASD键控制机器人移动,RViz中实时显示建图结果
# 完成后在另一终端运行save_map.sh保存地图
```

生成的 RViz 配置 脚本自动生成优化的 RViz 配置, 显示:

- 2D 栅格地图 (/map 话题)
- 激光扫描点云 (/scan 话题)
- 坐标系 TF 树
- 俯视正交视角

1.5.3 launch_tiago_nav2.sh - Nav2 导航启动

功能概述 在已保存的地图上启动完整的 Nav2 导航堆栈, 支持自主导航到目标点。

启动组件

1. Gazebo 仿真环境
2. TIAGo 机器人 (支持有臂/无臂版本)
3. ROS-Gazebo 话题桥接
4. Robot State Publisher
5. 里程计 TF 发布节点 (odom_to_tf.py)
6. 激光 Frame 转换器 (laser_frame_remapper.py)
7. Nav2 导航堆栈 (nav2_bringup)
8. RViz 可视化
9. 可选: 自动路径点跟随

关键配置文件

- **地图文件**:`map/nav_maps/warehouse.yaml`
- **Nav2 参数**:`tiago_nav2.yaml`
- **路径点配置**:`config/waypoints.yaml`

使用方法

```
# 基本启动
./launch_tiago_nav2.sh

# 指定地图文件
MAP_FILE=/path/to/map.yaml ./launch_tiago_nav2.sh

# 自动运行路径点
RUN_WAYPOINTS=1 ./launch_tiago_nav2.sh

# 使用无臂版TIAGo
ARM_TYPE=no-arm ./launch_tiago_nav2.sh

# 自定义Nav2参数
NAV2_PARAMS=/path/to/custom_nav2.yaml ./launch_tiago_nav2.sh
```

表 5: launch_tiago_nav2.sh 环境变量

变量名	默认值	说明
MAP_FILE	nav_maps/warehouse.yaml	导航地图路径
NAV2_PARAMS	tiago_nav2.yaml	Nav2 参数文件
WORLD_NAME	warehouse	Gazebo 世界名称
RUN_RVIZ	1	是否启动 RViz
START_UI	1	是否启动遥控 UI
RUN_WAYPOINTS	0	是否自动发送路径点
ARM_TYPE	no-arm	机械臂类型

环境变量

1.5.4 save_map.sh - 地图保存

功能概述 保存 SLAM 建图结果为多种格式, 便于后续导航使用。

保存内容

1. **标准导航地图** (PGM + YAML):Nav2 Map Server 使用
2. **PNG 格式地图**: 便于查看和文档使用
3. **SLAM 序列化文件**: 可用于继续编辑地图

使用方法

```
# 自动命名保存到默认目录
./save_map.sh

# 指定地图名称
./save_map.sh my_warehouse_map

```

```
# 指定名称和保存目录
./save_map.sh my_map /home/user/maps
```

输出示例

```
生成的文件:
  - my_map.pgm         : 灰度地图图像
  - my_map.yaml        : 地图元数据
  - my_map_png.png     : PNG格式地图
  - my_map_slam.*      : SLAM序列化文件
```

1.6 系统整体架构

本节详细介绍仓库管理员系统的整体架构, 包括各组件的角色、通信机制以及数据流向。

1.6.1 系统架构概览

整个系统由多个相互协作的组件构成, 通过 ROS 2 的话题 (Topic)、服务 (Service) 和动作 (Action) 机制进行通信:

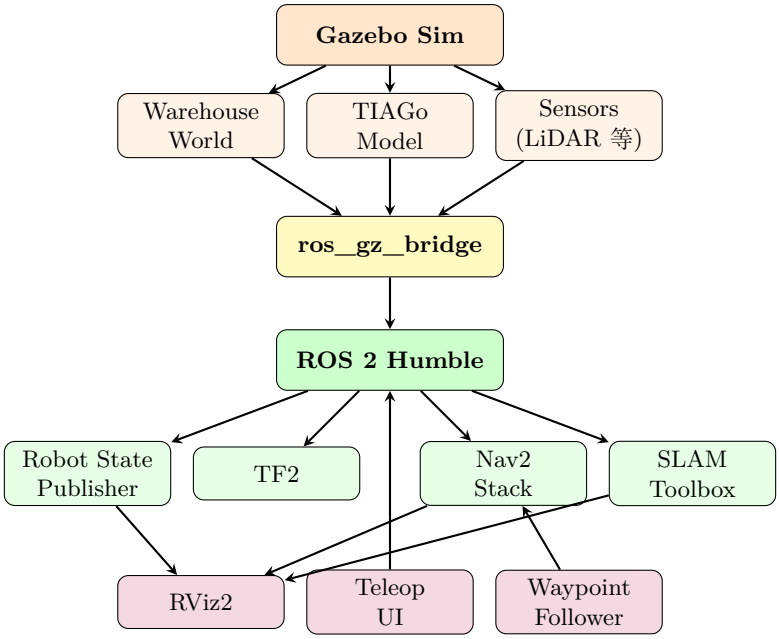

图 4: 系统整体架构图

1.6.2 Gazebo 仿真环境

Gazebo Sim(原 Ignition Gazebo) 作为物理仿真引擎, 承担以下职责:

- **世界仿真**: 加载 warehouse.sdf 定义的仓库环境, 包括地面、货架、护栏等静态物体
- **机器人仿真**: 加载 TIAGo 的 SDF 模型, 模拟其物理特性 (质量、惯性、碰撞)
- **传感器仿真**: 模拟激光雷达扫描, 生成符合物理规律的传感器数据
- **运动仿真**: 根据接收的速度指令, 计算机器人的运动轨迹和位姿变化

Gazebo 内部使用自己的消息系统 (ignition.msgs), 与 ROS 2 的消息系统不同, 因此需要桥接。

1.6.3 ros_gz_bridge 通信桥接

ros_gz_bridge 是连接 Gazebo 和 ROS 2 的关键组件, 负责双向消息转换:

表 6: 主要桥接话题

ROS 2 话题	Gazebo 话题	方向	用途
/clock	/clock	Gz→ROS	仿真时间同步
/cmd_vel	/model/tiago/cmd_vel	ROS→Gz	速度控制指令
/odom	/odom	Gz→ROS	里程计数据
/scan	/scan	Gz→ROS	激光扫描数据
/tf	/tf	Gz→ROS	坐标变换
/joint_states	/joint_states	Gz→ROS	关节状态

桥接配置示例:

```
ros2 run ros_gz_bridge parameter_bridge \
  /clock@rosgraph_msgs/msg/Clock[ignition.msgs.Clock \
  /cmd_vel@geometry_msgs/msg/Twist]ignition.msgs.Twist \
  /scan@sensor_msgs/msg/LaserScan[ignition.msgs.LaserScan
```

其中 [表示 Gazebo 到 ROS 的单向桥接,] 表示 ROS 到 Gazebo 的单向桥接,@ 分隔 ROS 消息类型和 Gazebo 消息类型。

1.6.4 ROS 2 通信机制

系统中各节点通过以下 ROS 2 通信机制协作:

话题通信 (Topic) 用于持续的数据流传输:

- `/scan`: 激光扫描数据 (10-20Hz)
- `/odom`: 里程计数据 (50Hz)
- `/cmd_vel`: 速度指令 (20Hz)
- `/map`: 栅格地图 (按需更新)
- `/tf`: 坐标变换 (高频更新)

服务通信 (Service) 用于请求-响应模式:

- `/map_server/load_map`: 加载地图
- `/slam_toolbox/serialize_map`: 保存 SLAM 地图
- `/global_costmap/clear_entirely_global_costmap`: 清除代价地图

动作通信 (Action) 用于长时间运行的任务:

- `/navigate_to_pose`: 导航到目标点
- `/follow_waypoints`: 跟随路径点序列
- `/compute_path_to_pose`: 计算路径

1.6.5 TF 坐标变换系统

TF2 负责维护各坐标系之间的变换关系, 是导航系统的核心:

```
map (全局固定坐标系)
 |-- [AMCL发布] map->odom变换 (定位校正)
 |
 +-> odom (里程计坐标系)
      |-- [odom_to_tf.py发布] odom->base_footprint变换
      |
      +-> base_footprint (机器人底盘投影)
           |-- [robot_state_publisher发布]
           |
           +-> base_link -> base_laser_link (激光雷达)
                         -> torso_lift_link -> ...
```

各变换的发布者:

- **map→odom**: 由 AMCL 根据定位结果发布, 校正里程计漂移
- **odom→base_footprint**: 由 odom_to_tf.py 从/odom 话题提取并发布
- **base_footprint→ 其他**: 由 robot_state_publisher 根据 URDF 发布

1.6.6 Nav2 导航数据流

Nav2 导航过程中的数据流向:

1. **感知输入**:
   - 激光扫描数据 (/scan_remapped)→ 代价地图更新
   - TF 变换 → 机器人位姿获取
2. **定位**:
   - AMCL 接收/scan_remapped 和/map
   - 输出 map→odom 变换
3. **路径规划**:

- Planner Server 接收目标点和全局代价地图
- 输出全局路径 (/plan)

4. **路径跟踪**:
   - Controller Server 接收全局路径和局部代价地图
   - 输出速度指令 (/cmd_vel)
5. **执行**:
   - 速度指令通过 Bridge 传递给 Gazebo
   - Gazebo 更新机器人位姿
   - 新的传感器数据反馈回 ROS 2

1.6.7 RViz 可视化

RViz2 作为可视化工具, 订阅以下数据进行显示:

- **/map**: 显示 2D 栅格地图
- **/scan**: 显示激光扫描点云
- **/tf**: 显示坐标系关系
- **/plan**: 显示全局规划路径
- **/local_plan**: 显示局部规划路径
- **/global_costmap/costmap**: 显示全局代价地图
- **/local_costmap/costmap**: 显示局部代价地图

同时 RViz 提供交互功能:

- **2D Pose Estimate**: 发布/initialpose 设置初始位姿
- **2D Goal Pose**: 发布/goal_pose 设置导航目标

1.6.8 系统启动顺序

为确保系统正常运行, 各组件需按以下顺序启动:

1. **Gazebo 仿真**: 加载世界和机器人模型
2. **ros_gz_bridge**: 建立 Gazebo-ROS 通信
3. **robot_state_publisher**: 发布机器人 TF
4. **静态 TF/odom_to_tf**: 补充坐标变换
5. **SLAM 或 Map Server**: 提供地图
6. **Nav2 Stack**: 启动导航服务
7. **RViz**: 可视化界面
8. **Teleop UI**: 用户控制接口

这个启动顺序在 launch_tiago_nav2.sh 脚本中已经正确实现。

1.7 本章小结

本章详细介绍了仓库管理员项目的技术基础:

1. **Nav2 导航框架**: 包括行为树导航、全局规划、局部控制、代价地图、定位系统等核心模块, 为机器人自主导航提供完整解决方案。
2. **TIAGo 机器人**: 介绍了其硬件组成 (PMB2 底盘、激光雷达、深度相机等)、软件架构和坐标系结构, 以及在本项目中的使用方式。
3. **仿真环境**: 描述了基于 Gazebo 的仓库仿真环境构建, 以及使用 SLAM Toolbox 进行地图构建的流程。
4. **遥控 UI**: 介绍了图形化遥控界面的使用方法和技术实现。
5. **项目脚本**: 说明了四个核心脚本的功能和使用方法, 涵盖从基础仿真到 SLAM 建图再到 Nav2 导航的完整工作流程。
6. **系统架构**: 详细阐述了 Gazebo 仿真、ros_gz_bridge 桥接、ROS 2 通信、TF 坐标系统、Nav2 数据流和 RViz 可视化之间的协作关系。

这些技术基础为后续章节中的导航参数优化、控制器对比实验和自定义功能开发奠定了基础。

2 Nav2 导航参数优化

本部分将详细介绍 Nav2 导航参数优化的测试策略、实验设计以及批量测试框架的实现。

2.1 测试策略与实验设计

2.1.1 测试框架概述

为了系统性地评估不同 Nav2 参数配置对导航性能的影响, 本项目设计并实现了一套**自动化批量测试框架**。该框架采用分层架构, 由调度层和执行层两部分组成:

表 7: 测试框架层级结构

层级	脚本	职责
调度层	batch_nav_test.sh	遍历参数配置、管理测试生命周期、汇总结果
执行层	launch_tiago_test.sh	启动仿真环境、运行单次测试、采集性能数据

这种分层设计使得测试框架具有良好的可扩展性和可维护性。调度层负责测试的编排和协调, 执行层专注于单次测试的完整执行, 两者职责分明、相互独立。

2.1.2 参数化测试策略

测试框架采用**参数覆盖机制**进行参数化管理。所有待测试的 Nav2 参数配置以 YAML 文件形式存放在 `config/nav2_params/`目录下:

```
config/nav2_params/
|-- baseline.yaml         # 基准配置
|-- high_speed.yaml       # 高速导航配置
|-- conservative.yaml     # 保守配置
|-- low_inflation.yaml    # 低膨胀半径配置
|-- ...
```

每个配置文件采用**增量覆盖**的方式, 仅需指定与基准参数不同的部分。系统通过 `merge_nav2_params.py` 工具自动将覆盖参数与基准参数合并, 生成完整的 Nav2 配置文件。这种设计具有以下优势:

- **简洁性**: 配置文件只包含差异部分, 易于阅读和维护
- **一致性**: 所有测试共享相同的基准配置, 确保对比的公平性
- **可追溯性**: 清晰记录每次测试的参数变化

参数合并示例:

```
# high_speed.yaml - 仅包含需要覆盖的参数
controller_server:
  ros__parameters:
    FollowPath:
      max_vel_x: 1.5        # 提高最大速度
      max_vel_theta: 1.5    # 提高最大角速度
```

2.1.3 单次测试执行流程

每次测试由 `launch_tiago_test.sh` 脚本执行, 完整流程如下:

图 5: 单次测试执行流程

测试脚本会自动统计 waypoints 配置文件中的目标点数量, 并在测试开始前显示测试概要信息, 包括测试名称、参数文件路径、输出目录等。

2.1.4 测试控制参数

批量测试框架提供了丰富的控制参数, 可通过环境变量进行配置:

表 8: 批量测试控制参数

参数	默认值	说明
WAYPOINT_TIMEOUT	120s	单个目标点的导航超时时间
PAUSE_BETWEEN	10s	测试间隔时间, 确保进程完全清理
HEADLESS	0	无头模式 (1= 无 GUI), 适用于服务器环境
RUN_RVIZ	1	是否启动 RViz 可视化
SKIP_EXISTING	0	跳过已有结果的测试, 支持断点续测
OUTPUT_DIR	test_results/	测试结果输出目录

使用示例:

```
# 无头模式批量测试,跳过已完成的测试
HEADLESS=1 RUN_RVIZ=0 SKIP_EXISTING=1 ./map/batch_nav_test.sh

# 只测试指定配置
./map/batch_nav_test.sh baseline high_speed conservative

# 设置每个waypoint的超时时间为60秒
WAYPOINT_TIMEOUT=60 ./map/batch_nav_test.sh
```

2.1.5 容错与进程管理

为确保批量测试的稳定性和可靠性, 测试框架实现了完善的容错机制:

1. 信号捕获与优雅退出 通过 `trap` 命令捕获 INT 和 TERM 信号, 确保测试中断时能够正确清理所有相关进程:

```
trap batch_cleanup INT TERM
```

清理函数会依次终止以下进程组:

- Gazebo/Ignition 相关进程 (gz sim, gzserver, gzclient)

- Nav2 导航相关进程 (bt_navigator, controller_server, planner_server 等)
- 可视化和桥接进程 (rviz2, parameter_bridge, robot_state_publisher)
- 测试辅助进程 (nav_performance_test, odom_to_tf, laser_frame_remapper)

2. 进程隔离 每次测试结束后, 无论成功与否, 都会执行完整的进程清理流程:

1. 首先发送 SIGTERM 信号, 允许进程优雅退出
2. 等待 2 秒后, 对残留的 Gazebo 进程发送 SIGKILL 强制终止
3. 在测试间隔期间等待指定时间 (默认 10 秒), 确保系统资源完全释放

3. 状态记录与断点续测 测试框架实时记录每个测试的执行状态到 `batch_summary.yaml` 文件:

```yaml
results:
  - name: baseline
    status: completed
    exit_code: 0
    duration_seconds: 245
  - name: high_speed
    status: error
    exit_code: 1
    duration_seconds: 180
  - name: conservative
    status: skipped
    reason: result_exists
```

通过设置 `SKIP_EXISTING=1`, 可以跳过已存在结果的测试, 实现断点续测功能。

4. 退出码传递 单次测试的失败不会中断整个批量测试流程。框架会统计所有测试的通过/失败/跳过数量, 最终根据是否有失败的测试返回相应的退出码。

2.1.6 输出与结果分析

测试结果按批次组织, 采用统一的目录结构:

```
test_results/
|-- batch_20241205_143000/
    |-- batch_summary.yaml           # 批量测试摘要
    |-- nav_test_baseline.yaml       # baseline配置详细结果
    |-- nav_test_baseline.csv        # baseline配置CSV格式结果
    |-- nav_test_high_speed.yaml     # high_speed配置详细结果
    |-- nav_test_high_speed.csv
    |-- nav_test_conservative.yaml
    |-- nav_test_conservative.csv
    |-- comparison_report.html       # 自动生成的对比报告
```

批量测试摘要 `batch_summary.yaml` 文件记录整个批量测试的元信息和汇总结果:

```yaml
batch_id: "batch_20241205_143000"
start_time: "2024-12-05T14:30:00+08:00"
end_time: "2024-12-05T15:45:00+08:00"
configs_tested: 5
config_names:
  - baseline
  - high_speed
  - conservative
  - low_inflation
  - high_inflation
summary:
  passed: 4
  failed: 1
  skipped: 0
  total: 5
```

自动对比报告生成 批量测试完成后, 框架自动调用 `compare_nav_results.py` 脚本生成跨配置的性能对比报告。该报告以 HTML 格式输出, 包含:

- 各配置的导航成功率对比
- 平均导航时间和路径长度对比
- 速度和加速度统计对比

- 可视化图表 (柱状图、折线图等)

2.1.7 测试设计原则

本测试框架的设计遵循以下原则:

1. **可重复性**: 相同的配置在相同的环境下应产生一致的结果。通过固定随机种子、使用仿真时间等方式确保测试的可重复性。
2. **可对比性**: 所有测试使用相同的 waypoints 路线、相同的仿真环境、相同的初始条件, 仅改变待测试的 Nav2 参数, 确保对比的公平性。
3. **可扩展性**: 添加新的测试配置只需在 `config/nav2_params/`目录下创建新的 YAML 文件, 无需修改测试框架代码。
4. **自动化**: 从测试启动到结果生成全程自动化, 支持无人值守的批量测试, 适合在 CI/CD 环境中运行。
5. **健壮性**: 完善的错误处理和进程管理机制, 确保单个测试的失败不会影响整体测试流程。

这套测试策略为后续的 Nav2 参数优化实验提供了可靠的数据基础, 使我们能够系统性地评估不同参数配置对导航性能的影响。

3 控制器对比分析

4 行为树子任务设计

5 自定义控制器插件开发

6 总结与展望