

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 7 日

报告概览

本报告基于 ROS 2 Humble 和 Nav2 导航框架, 在 TIAGo 机器人仿真平台上开展导航性能优化研究。

- **第一部分:**Nav2 框架架构、TIAGo 平台、Gazebo 仿真环境、遥控 UI 与自动化测试脚本。
- **第二部分:**6 组参数配置对比实验,baseline 以 100% 成功率和 0.453 m/s 取得最佳表现。
- **第三部分:**DWB/RPP/MPPI/Stanley 四种控制器对比,RPP 效率最高,MPPI 精度最优; 详述 Stanley 插件开发与参数设计。
- **第四部分:** 动态多控制器行为树设计, 通过 Blackboard 实现三阶段控制器切换。
- **第五部分:** 主要结论与未来展望。

目录

1 系统概述与环境介绍	3
1.1 Nav2 导航框架	3
1.1.1 核心组件	3
1.2 TIAGo 机器人平台	3
1.2.1 硬件组成与运动学模型	4
1.2.2 坐标系与软件架构	4
1.3 仿真环境与工具链	4
1.3.1 仓库仿真环境与 SLAM 建图	4
1.3.2 遥控 UI 与项目脚本	4
1.4 系统整体架构	5
1.4.1 仿真与通信层	6
1.4.2 ROS 2 通信机制与 TF 系统	6
1.4.3 Nav2 导航数据流与系统启动	6
2 Nav2 导航参数优化	7
2.1 实验设计与测试框架	7
2.1.1 实验方法与测试路线	7
2.1.2 评价指标体系	7
2.1.3 自动化测试框架	7
2.2 参数配置与理论分析	8
2.2.1 测试配置设计	8
2.2.2 参数影响机理	8
2.3 实验结果与分析	9

2.3.1	总体性能对比	9
2.3.2	结果分析与关键发现	9
2.3.3	配置推荐	9
3	控制器对比分析	9
3.1	控制器算法原理	9
3.1.1	DWB 控制器 (Dynamic Window Approach)	9
3.1.2	RPP 控制器 (Regulated Pure Pursuit)	10
3.1.3	MPPI 控制器 (Model Predictive Path Integral)	10
3.1.4	Stanley 控制器 (自定义实现)	10
3.2	Stanley 控制器插件开发	10
3.2.1	插件架构与接口实现	10
3.2.2	核心算法实现	10
3.2.3	插件注册与编译	10
3.2.4	关键参数说明与设计决策	11
3.3	控制器对比测试	12
3.3.1	测试配置	12
3.3.2	测试结果	12
3.3.3	图表化分析	12
3.3.4	结果分析	13
3.3.5	控制器选型建议	13
4	行为树子任务设计	13
4.1	行为树基础	14
4.1.1	行为树概述	14
4.1.2	Blackboard 与动态参数传递	14
4.2	动态多控制器行为树设计	14
4.2.1	三种控制器配置	14
4.2.2	行为树整体结构	14
4.2.3	阶段一: 快速接近 (FollowPath 控制器)	15
4.2.4	阶段二: 慢速精调 (SlowFollowPath 控制器)	15
4.2.5	阶段三: 精确对齐 (StanleyFollowPath 控制器)	15
4.2.6	动态控制器切换机制总结	16
4.3	行为树配置与部署	16
5	总结与展望	17
5.1	主要结论	17
5.2	未来展望	17

1 系统概述与环境介绍

本部分介绍项目的核心技术栈, 包括 Nav2 导航框架、TIAGo 机器人平台、仿真环境以及系统整体架构。

1.1 Nav2 导航框架

Nav2(Navigation2) 是 ROS 2 的官方导航堆栈, 为移动机器人提供完整的自主导航解决方案。其设计强调模块化和可扩展性——所有核心功能以插件形式实现, 用户可根据需要替换默认组件或添加自定义功能。Nav2 的核心架构如图 1 所示, 采用分层设计, 顶层由行为树导航器 (BT Navigator) 协调, 下层包含规划、控制、感知等功能模块。

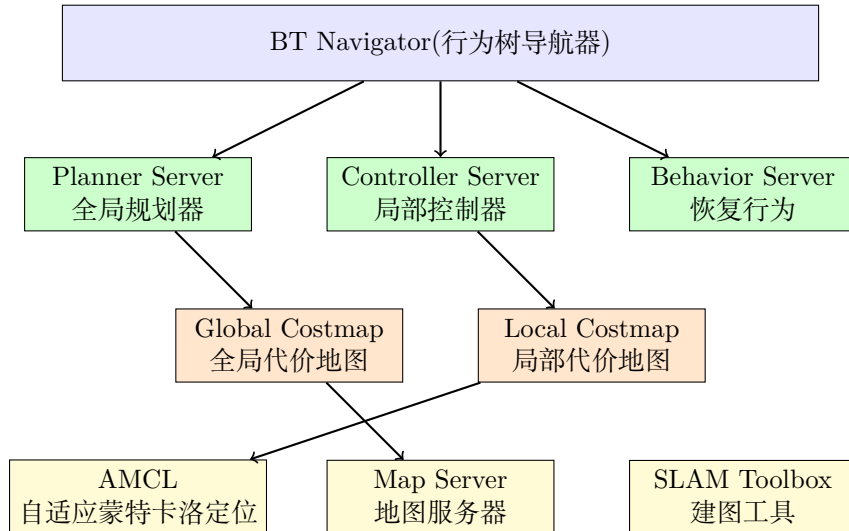


图 1: Nav2 系统架构示意图

1.1.1 核心组件

行为树导航器 (BT Navigator) 是 Nav2 的核心协调模块, 使用行为树管理导航任务的执行流程。相比传统有限状态机, 行为树具有模块化、可复用、易调试和支持动态切换等优势。Nav2 默认使用 `navigate_w_replanning_and_recovery` 配置, 实现带重规划和恢复机制的导航流程。

全局路径规划器 (Planner Server) 负责计算从当前位置到目标点的全局路径。Nav2 提供多种规划算法: NavFn Planner 采用经典的 Dijkstra/A* 算法, 稳定可靠; Smac Planner 系列支持 2D、Hybrid-A* 和 State Lattice 等变体, 可考虑运动学约束; Theta* Planner 能产生更平滑的路径。本项目默认使用 NavFn Planner。

局部控制器 (Controller Server) 负责跟踪全局路径并生成实时速度指令。本项目涉及三种控制器: (1) **DWB 控制器** 基于动态窗口法, 在速度空间中采样可行轨迹并通过多个评价函数 (Critics) 综合评分选择最优控制, 主要参数包括最大速度、轨迹仿真时间和速度采样数量; (2) **RPP 控制器** (Regulated Pure Pursuit) 是改进的纯追踪控制器, 增加了根据路径曲率、障碍物距离和目标点距离自动调节速度的机制; (3) **MPPI 控制器** (Model Predictive Path Integral) 基于模型预测和路径积分, 使用蒙特卡洛采样预测多条轨迹并加权平均, 计算量较大但效果更优。

代价地图 (Costmap) 是路径规划和避障的基础。系统维护两种代价地图: 全局代价地图覆盖整个已知环境, 包含静态地图层、障碍物层和膨胀层, 用于全局路径规划; 局部代价地图是以机器人为中心的滚动窗口, 实时更新传感器观测, 用于局部避障和控制。

定位系统 (AMCL) 使用自适应蒙特卡洛定位算法, 通过粒子滤波实现机器人定位。其工作流程为: 初始化时在地图上分布大量粒子表示假设位姿, 根据里程计信息移动粒子 (预测), 根据激光扫描与地图匹配程度为粒子赋权重 (更新), 最后通过重采样保留高权重粒子。

恢复行为 (Behavior Server) 在导航陷入困境时尝试使机器人脱困, 包括原地旋转 (Spin) 扫描周围环境、后退指定距离 (BackUp)、沿当前方向行驶 (DriveOnHeading) 以及等待障碍物移开 (Wait) 等行为。

1.2 TIAGo 机器人平台

TIAGo 是 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。该机器人采用高度模块化设计, 可根据需求配置不同硬件组件。

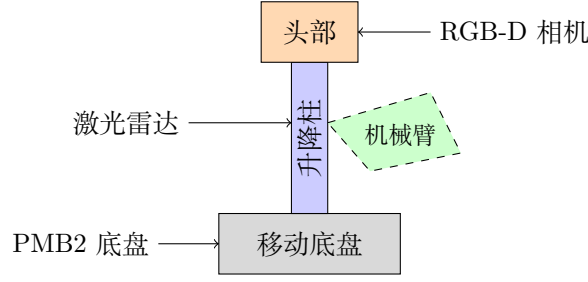


图 2: TIAGo 机器人结构示意图

1.2.1 硬件组成与运动学模型

TIAGo 采用 PMB2 差速驱动底盘, 最大前进速度 1.0 m/s, 最大旋转速度 1.0 rad/s, 底盘直径约 54cm, 机器人半径 0.275m(用于导航避障)。差速驱动的运动学模型为:

$$v = \frac{v_r + v_l}{2}, \quad \omega = \frac{v_r - v_l}{L} \quad (1)$$

其中 v_r 和 v_l 分别为左右轮速度, L 为轮距。

传感器方面, 底盘前部安装 2D 激光雷达 (base_laser_link), 相对底盘偏移 $x=0.202\text{m}$ 、 $z=-0.004\text{m}$, 扫描范围 270-360 度, 最大探测距离 20m, 数据发布到 /scan 话题。头部配备 RGB-D 深度相机 (如 Orbbec Astra 或 Intel RealSense), 用于物体识别和 3D 感知。TIAGo 还可选配 7 自由度机械臂, 集成 MoveIt2 运动规划框架, 本项目使用无臂版本 (no-arm) 以简化导航测试。

1.2.2 坐标系与软件架构

TIAGo 的坐标结构遵循 REP-105 标准, 形成如下 TF 树: 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(底盘投影到地面) → base_link(底盘中心) → base_laser_link(激光雷达) 等。

本项目的 TIAGo 软件包结构包括: 核心包 tiago_robot(模型描述、启动文件、控制器配置)、导航包 tiago_navigation(导航、激光和 RGBD 传感器配置)、仿真包 tiago_simulation(Gazebo 仿真配置) 以及 PAL 导航参数包 (包含 tiago_nav2.yaml 配置文件)。

在 Gazebo 中通过 `ros2 run ros_gz_sim create` 命令加载 TIAGo 的 SDF 模型, 使用 `ros_gz_bridge` 将 Gazebo 话题桥接到 ROS 2, 主要控制接口包括速度指令 (/cmd_vel)、里程计 (/odom)、激光扫描 (/scan) 和关节状态 (/joint_states)。

1.3 仿真环境与工具链

1.3.1 仓库仿真环境与 SLAM 建图

本项目使用 Gazebo Sim 构建仓库仿真环境, 世界定义文件位于 `map/turtlebot4_gz_bringup/worlds/warehouse.sdf`。仓库环境包含: 仓库主体 (OpenRobotics/Warehouse)、5 个大型货架和 8 个小型货架 (MovAi)、4 个护栏 (Jersey Barrier) 以及椅子等障碍物。物理引擎采用 ODE, 步长 0.003 秒, 实时因子 1.0。

地图构建使用 SLAM Toolbox 进行 2D 激光 SLAM, 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。建图流程为: 启动 Gazebo 仿真和 TIAGo 机器人 → 运行 `slam_toolbox` 的 `sync_slam_toolbox_node` → 使用遥控 UI 控制机器人遍历环境 → 调用 `save_map.sh` 保存结果。关键 SLAM 参数包括: 地图分辨率 0.05m/像素、最大激光探测距离 20m、触发扫描匹配的最小移动距离 0.3m。建图完成后生成 PGM 灰度地图、YAML 元数据文件和 SLAM 序列化文件。

1.3.2 遥控 UI 与项目脚本

本项目开发了基于 Tkinter 的图形化遥控界面 (`teleop_ui.py`), 支持键盘 (WASD+ 空格) 和按钮两种控制方式。技术特点包括: 多话题发布 (同时向 /cmd_vel、/mobile_base_controller/cmd_vel_unstamped 和 /model/tiago/cmd_vel 发布, 确保兼容不同配置)、以 100ms 间隔持续发送速度指令、渐进式速度调节 (每次 ± 0.1 m/s) 以及实时状态显示。



图 3: TIAGo 遥控 UI 界面

项目在 `map/` 目录下提供六个 Shell 脚本, 分为基础仿真、导航运行和自动化测试三类。

基础仿真与建图脚本 `launch_tiago.sh` 是基础仿真启动脚本, 一键启动 Gazebo 仿真环境、加载 TIAGo 模型、启动 ROS-Gazebo 话题桥接和遥控 UI。支持通过环境变量配置世界名称、无头模式和机器人初始位姿。`launch_tiago_slam.sh` 在基础仿真之上增加 Robot State Publisher、SLAM Toolbox 节点和 RViz 可视化, 用于建图。`save_map.sh` 将 SLAM 建图结果保存为标准导航地图 (PGM+YAML)、PNG 格式和 SLAM 序列化文件。

导航与自动化测试脚本 `launch_tiago_nav2.sh` 在已保存的地图上启动完整 Nav2 导航堆栈, 包括 Gazebo 仿真、话题桥接、TF 发布、Nav2 各服务节点和 RViz 可视化。支持通过环境变量指定地图文件 (MAP_FILE)、Nav2 参数 (NAV2_PARAMS) 和机械臂类型 (ARM_TYPE) 等。

`launch_tiago_test.sh` 是单次导航性能测试脚本, 在 Nav2 环境基础上集成自动化测试流程: 自动设置初始位姿、依次导航到预设 waypoints、记录每段导航的时间/距离/误差等性能数据, 最终输出 YAML 和 CSV 格式的测试报告。核心特性包括: 支持参数文件增量合并 (基础参数 + 参数覆盖 + 控制器覆盖)、每个 waypoint 独立超时控制 (WAYPOINT_TIMEOUT)、自动初始位姿设置 (AUTO_INIT_POSE)。

`batch_nav_test.sh` 是批量测试调度脚本, 自动遍历多组参数配置和控制器组合, 依次调用 `launch_tiago_test.sh` 执行测试。支持三种测试模式: 仅测试参数配置、仅测试控制器、参数与控制器组合测试。具备断点续测 (SKIP_EXISTING)、测试间隔控制 (PAUSE_BETWEEN)、信号捕获清理等容错机制。测试结果按批次组织, 自动生成 `batch_summary.yaml` 汇总文件。

表 1: 项目脚本功能概览

脚本	功能
<code>launch_tiago.sh</code>	基础仿真: Gazebo + TIAGo + 话题桥接 + 遥控 UI
<code>launch_tiago_slam.sh</code>	SLAM 建图: 基础仿真 + SLAM Toolbox + RViz
<code>save_map.sh</code>	保存 SLAM 地图为 PGM/YAML/PNG 格式
<code>launch_tiago_nav2.sh</code>	Nav2 导航: 完整导航堆栈 + 可视化
<code>launch_tiago_test.sh</code>	单次测试: 自动 waypoint 导航 + 性能数据采集
<code>batch_nav_test.sh</code>	批量测试: 多配置/控制器遍历 + 结果汇总

1.4 系统整体架构

整个仓库管理员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。

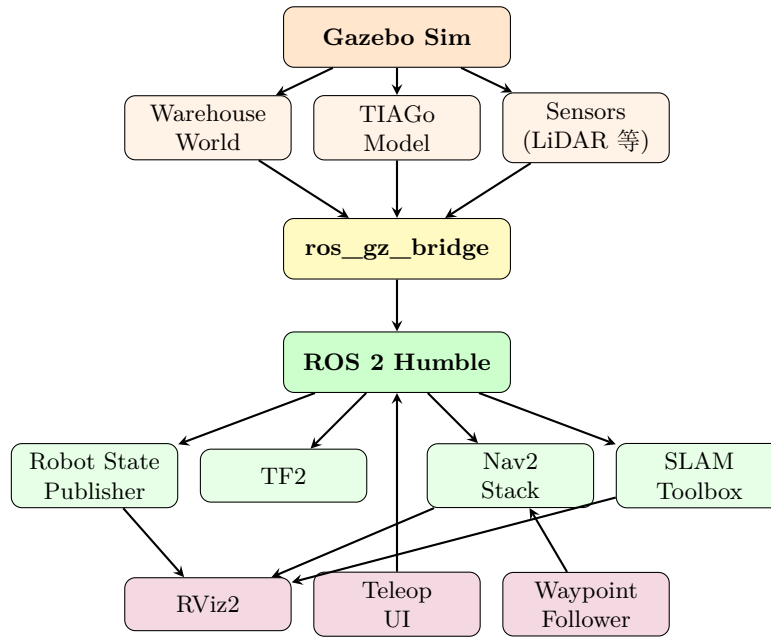


图 4: 系统整体架构图

1.4.1 仿真与通信层

Gazebo Sim 作为物理仿真引擎, 承担世界仿真 (加载 warehouse.sdf 定义的仓库环境)、机器人仿真 (加载 TIAGo 的 SDF 模型并模拟其物理特性)、传感器仿真 (模拟激光雷达扫描) 和运动仿真 (根据速度指令计算机器人轨迹) 等职责。Gazebo 内部使用 ignition.msgs 消息系统, 与 ROS 2 不同, 因此需要桥接。

ros_gz_bridge 是连接 Gazebo 和 ROS 2 的关键组件, 负责双向消息转换。主要桥接话题包括:/clock(仿真时间同步,Gz→ROS)、/cmd_vel(速度控制,ROS→Gz)、/odom(里程计,Gz→ROS)、/scan(激光扫描,Gz→ROS)、/tf(坐标变换,Gz→ROS) 和/joint_states(关节状态,Gz→ROS)。桥接配置中,[表示 Gazebo 到 ROS 的单向桥接,]表示 ROS 到 Gazebo 的单向桥接。

1.4.2 ROS 2 通信机制与 TF 系统

系统中各节点通过三种 ROS 2 通信机制协作: **话题通信**用于持续数据流传输, 如激光扫描 (10-20Hz)、里程计 (50Hz)、速度指令 (20Hz)、栅格地图 (按需更新) 和坐标变换 (高频更新); **服务通信**用于请求-响应模式, 如加载地图、保存 SLAM 地图和清除代价地图; **动作通信**用于长时间运行的任务, 如导航到目标点、跟随路径点序列和计算路径。

TF2 负责维护各坐标系之间的变换关系, 是导航系统的核心。变换链为:map $\xrightarrow{\text{AMCL}}$ odom $\xrightarrow{\text{odom_to_tf.py}}$ base_footprint $\xrightarrow{\text{robot_state_publisher}}$ robot_footprint等。其中 map→odom 变换由 AMCL 根据定位结果发布以校正里程计漂移,odom→base_footprint 由 odom_to_tf.py 从/odom 话题提取并发布,base_footprint 到其他坐标系由 robot_state_publisher 根据 URDF 发布。

1.4.3 Nav2 导航数据流与系统启动

Nav2 导航过程中的数据流向为: 感知输入 (激光扫描 → 代价地图更新,TF 变换 → 机器人位姿获取)→ 定位 (AMCL 接收激光和地图, 输出 map→odom 变换)→ 路径规划 (Planner Server 接收目标点和全局代价地图, 输出全局路径)→ 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)→ 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)。

RViz2 作为可视化工具, 订阅/map、/scan、/tf、/plan、/local_plan 和代价地图等数据进行显示, 同时提供 2D Pose Estimate(设置初始位姿) 和 2D Goal Pose(设置导航目标) 交互功能。

为确保系统正常运行, 各组件需按顺序启动:Gazebo 仿真 →ros_gz_bridge→robot_state_publisher→ 静态 TF/odom_to_tf→SLAM 或 Map Server→Nav2 Stack→RViz→Teleop UI。这个启动顺序在 launch_tiago_nav2.sh 脚本中已正确实现。

2 Nav2 导航参数优化

本部分介绍 Nav2 导航参数优化的实验设计、测试框架、参数配置以及实验结果分析。

2.1 实验设计与测试框架

2.1.1 实验方法与测试路线

为科学评估不同 Nav2 参数配置对导航性能的影响, 本项目采用**控制变量法**进行实验设计: 设计一条固定的测试路线, 在相同的仿真环境和初始条件下, 仅改变待测试的参数配置, 从而获得可对比的性能数据。这种设计确保了公平性 (消除路径差异影响)、可重复性 (便于验证结果一致性) 和可对比性 (支持定量比较)。

测试路线通过自主开发的 `record_waypoints.py` 工具获取, 支持在 RViz 中点击地图捕获目标点或按“r”键记录机器人当前位置两种方式。经过多次调整验证, 最终确定了包含 **12 个目标点** 的测试路线, 覆盖仓库的货架通道、开阔区域和角落位置。路线设计遵循以下原则: 覆盖直线、转弯、狭窄通道等典型场景; 包含不同难度路段; 目标朝向多样化 (-163.5° 至 115.2°); 最后一点接近起点形成闭环。

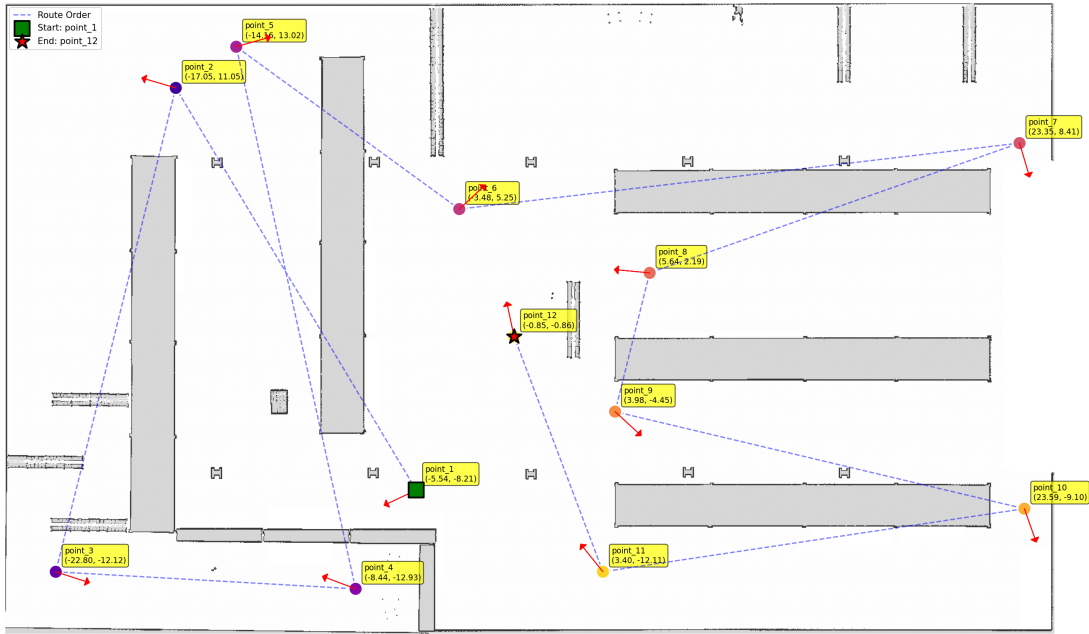


图 5: 测试路线可视化: 绿色方块为起点, 红色星形为终点, 蓝色虚线表示导航顺序

2.1.2 评价指标体系

为全面评估导航性能, 本项目通过 `nav_performance_test.py` 采集以下指标:

单段导航指标 (每个 waypoint): 导航是否成功 (success)、导航耗时 (time_seconds)、实际行驶距离 (distance_meters, 通过里程计累积计算)、位置误差 (position_error, 到达位置与目标位置的欧氏距离)、朝向误差 (yaw_error, 到达朝向与目标朝向的角度差)。

整体性能指标: 成功率 (success_rate, 成功到达的 waypoint 占比)、总耗时 (total_time, 所有成功导航的累计时间)、总距离 (total_distance, 所有成功导航的累计里程)、平均速度 (avg_velocity, 总距离除以总耗时)、平均位置误差 (avg_position_error)、最大位置误差 (max_position_error)、平均朝向误差 (avg_yaw_error)。

这些指标从效率 (时间、速度)、可靠性 (成功率) 和精度 (位置/朝向误差) 三个维度全面刻画导航性能, 其中成功率是最关键的指标——导航失败意味着任务无法完成, 其他指标仅在成功的前提下才有意义。

2.1.3 自动化测试框架

为系统性评估不同参数配置, 本项目实现了自动化批量测试框架, 由调度层 (`batch_nav_test.sh`) 和执行层 (`launch_tiago_test.sh`) 组成。

测试框架采用**增量覆盖**机制管理参数: 各配置文件存放在 `config/nav2_params/` 目录下, 只需指定与基准不同的参数, 系统通过 `merge_nav2_params.py` 自动合并生成完整配置。单次测试流程为: 启动 Gazebo 仿真 → 生成 TIAGo 机器人 → 启动 Nav2 导航堆栈 → 自动设置初始位姿 → 依次执行 waypoint 导航 → 采集性能数据 → 输出 YAML 和 CSV 格式报告。主要控制参数包括 `WAYPOINT_TIMEOUT` (单点超时, 默认 120 秒)、`HEADLESS` (无头模式)、`SKIP_EXISTING` (断点续测)。

框架通过信号捕获 (trap) 确保中断时正确清理所有进程, 测试状态实时记录到 `batch_summary.yaml`, 单次失败不会中断批量测试。测试结果按批次组织, 每批次生成独立目录, 包含批量摘要和各配置の詳細结果文件。

2.2 参数配置与理论分析

2.2.1 测试配置设计

本项目设计了 6 组参数配置, 每组配置涉及控制器、代价地图、行为服务器等多个模块的参数协调调整。

baseline(基准配置): 采用中等速度 (0.5 m/s) 和标准安全边距, 各参数平衡设置。DWB 控制器仿真时间 1.2s, 路径对齐权重 (PathAlign/PathDist) 与目标对齐权重 (GoalAlign/GoalDist) 分别为 32 和 24, 障碍物权重 0.03。RotateToGoal 权重 48, 减速因子 3.0。代价地图膨胀半径 0.55m, 衰减因子 3.0。作为对照基准验证其他配置的改进效果。

better_rotation(优化转向): 针对原始配置转向迟缓的问题进行优化。提高最小转向速度 (min_speed_theta: 0.3) 避免缓慢转向, 增加角速度采样数 (vtheta_samples: 25) 获得更细腻的转向选择, 提高 RotateToGoal 权重 (48) 并启用前瞻转向 (lookahead_time: 1.0) 实现提前转向。同时提高角加速度限制 (3.5 rad/s²) 加快转向响应。最大速度提升至 0.6 m/s。

high_speed(高速配置): 追求最快导航速度, 适合开阔环境。最大线速度提升至 0.8 m/s, 角速度 1.5 rad/s, 加速度 2.5 m/s²。缩短仿真时间 (0.8s) 加快决策, 减小膨胀半径 (全局 0.50m, 局部 0.45m) 允许更激进的路径, 扩大局部代价地图 (4m×4m) 适应高速感知需求。提高目标权重 (GoalAlign/GoalDist: 28) 加速到达。

aggressive(激进配置): 追求快速响应, 适合时间敏感场景。采用最高加速度 (3.0 m/s² 线性, 4.5 rad/s² 角度) 实现快速启停, 缩短进度检查时间 (movement_time_allowance: 8.0s) 要求快速进度。降低障碍物权重 (BaseObstacle.scale: 0.015) 减少避障导致的减速, 显著提高目标权重 (GoalAlign/GoalDist: 32) 和 RotateToGoal 权重 (52) 强化目标导向, 降低减速因子 (2.0) 加快转向。膨胀半径减至 0.45m/0.40m。

safe_narrow(安全狭窄通道): 专为复杂狭窄环境设计。采用低速 (0.35 m/s) 和低加速度 (1.5 m/s²) 确保稳定, 延长进度检查时间 (15.0s) 和放宽目标容差 (0.30m) 适应困难路段。显著提高障碍物权重 (BaseObstacle.scale: 0.08) 增强避障, 减小膨胀半径 (0.45m/0.40m) 允许通过狭窄区域, 降低衰减因子 (2.5) 保持警惕。延长仿真时间 (1.5s) 实现更谨慎的规划。

smooth_path(平滑路径): 追求平稳运动, 适合对舒适性要求高的场景。采用最低加速度 (1.2 m/s² 线性, 2.0 rad/s² 角度) 实现平滑启停, 延长仿真时间 (2.0s) 规划更平滑轨迹。提高路径对齐权重 (PathAlign/PathDist: 40) 加强路径跟随, 提高 RotateToGoal 减速因子 (5.0) 和前瞻时间 (1.5s) 实现平滑转向。收紧目标容差 (0.20m) 提高到达精度。扩大局部地图 (4m×4m) 获得更平滑规划。

表 2: 各配置核心参数对比

参数	baseline	better_rot	high_speed	aggressive	safe_narrow	smooth
max_vel_x (m/s)	0.5	0.6	0.8	0.7	0.35	0.5
max_vel_theta (rad/s)	1.0	1.2	1.5	1.4	0.8	0.9
acc_lim_x (m/s ²)	2.0	2.0	2.5	3.0	1.5	1.2
acc_lim_theta (rad/s ²)	3.2	3.5	4.0	4.5	2.5	2.0
sim_time (s)	1.2	1.0	0.8	0.8	1.5	2.0
vtheta_samples	20	25	25	25	20	25
PathAlign.scale	32	32	32	28	32	40
GoalDist.scale	24	24	28	32	20	20
BaseObstacle.scale	0.03	0.02	0.02	0.015	0.08	0.03
RotateToGoal.scale	48	48	48	52	48	48
slowing_factor	3.0	3.0	2.5	2.0	4.0	5.0
inflation_radius (m)	0.55	0.55	0.50	0.45	0.45	0.55
xy_goal_tolerance (m)	0.25	0.25	0.25	0.25	0.30	0.20

2.2.2 参数影响机理

速度与加速度参数: max_vel_x 和 max_vel_theta 直接决定导航效率上限, 但高速需要配合足够的感知范围 (局部代价地图尺寸) 和决策时间 (sim_time)。加速度参数影响响应速度和运动平滑度——aggressive 的高加速度 (3.0 m/s²) 实现快速响应但可能产生抖动, smooth_path 的低加速度 (1.2 m/s²) 运动平稳但启停耗时增加。

轨迹仿真与采样参数: sim_time 决定 DWB 控制器预测轨迹的时间窗口, 较长的仿真时间 (smooth_path: 2.0s) 能规划更平滑路径但响应变慢, 较短的仿真时间 (high_speed: 0.8s) 响应快但可能产生短视行为。vtheta_samples 增加 (25 vs 20) 可获得更细腻的转向控制, 对转向优化配置尤为重要。

评价函数权重: DWB 通过多个 Critics 评分选择最优轨迹。PathAlign/PathDist 控制路径跟随紧密度, smooth_path

设为 40 强化路径跟随;GoalAlign/GoalDist 控制目标导向性,aggressive 设为 32 加速到达;BaseObstacle 控制避障倾向,safe_narrow 设为 0.08 显著增强避障。RotateToGoal.scale 和 slowing_factor 共同影响转向行为——高 scale 使机器人更倾向转向目标,高 slowing_factor 使转向更平滑但较慢。

代价地图参数:inflation_radius 决定机器人与障碍物的安全距离,aggressive 和 safe_narrow 采用较小值 (0.45m) 以通过狭窄通道,但 aggressive 因高速可能有碰撞风险。cost_scaling_factor 控制代价衰减速度,较高值使代价快速衰减允许更靠近障碍物。局部地图尺寸影响感知范围,高速配置扩大至 4m×4m 以适应更远的前瞻需求。

2.3 实验结果与分析

2.3.1 总体性能对比

在 12 个目标点的测试路线上,各配置的导航性能如下表所示。

表 3: 导航性能总体对比

配置	成功率 (%)	总时间 (s)	总距离 (m)	平均速度 (m/s)	位置误差 (m)	朝向误差 (°)
baseline	100.0	548.2	248.4	0.453	0.234	17.2
better_rotation	100.0	562.6	251.5	0.447	0.213	19.1
smooth_path	100.0	573.0	249.1	0.435	0.213	14.4
safe_narrow	91.7	742.1	204.0	0.275	0.287	20.7
aggressive	75.0	541.9	193.5	0.357	0.177	20.2
high_speed	8.3	331.5	10.1	0.030	0.059	23.6

2.3.2 结果分析与关键发现

成功率分析:baseline、better_rotation 和 smooth_path 均达到 100% 成功率,证明中等速度配合适当参数调优是最稳定的选择。safe_narrow(91.7%) 在 point_5 超时失败——虽设计为安全配置,但过低速度 (0.35 m/s) 导致长距离路段无法按时到达。aggressive(75%) 失败 3 个点,高加速度和小膨胀半径导致复杂区域控制不稳定。high_speed(8.3%) 几乎全部失败,仅完成第一个点,过高速度 (0.8 m/s) 和过短仿真时间 (0.8s) 导致控制器无法稳定跟踪路径。

效率与精度分析:在完成全部 12 点的配置中,baseline 最快 (548.2s, 平均 0.453 m/s),better_rotation 次之 (562.6s),smooth_path 最慢 (573.0s) 但朝向精度最优 (14.4°)。其长仿真时间 (2.0s) 和高 RotateToGoal 减速因子使转向更精确。位置精度各配置相近 (0.21-0.29m),均在目标容差附近。

关键发现:(1) 速度并非越快越好——high_speed 设置最高速度却因控制不稳定几乎全部失败,实际有效速度反而最低;(2) 参数需要协调配合——单独提高速度而不调整仿真时间、采样数等参数会导致系统不稳定,baseline 的成功在于各参数平衡;(3) sim_time 是关键参数——过短 (0.8s) 导致控制器“短视”无法有效规划,过长 (2.0s) 虽路径更优但响应变慢,1.0-1.2s 是较好平衡点;(4) 安全与效率需权衡——safe_narrow 过于保守反而导致超时失败;(5) 转向优化效果显著——better_rotation 通过提高 min_speed_theta 和 RotateToGoal 权重使转向更果断。

2.3.3 配置推荐

基于实验结果,针对不同场景推荐:通用场景使用 baseline(平衡的速度与稳定性,100% 成功率);精确停靠使用 smooth_path(最小朝向误差 14.4°,平滑运动);快速响应使用 better_rotation(转向果断,效率略优于 baseline);狭窄通道建议在 safe_narrow 基础上适当提高速度以避免超时。

3 控制器对比分析

本部分对 Nav2 支持的三种主流控制器 (DWB、RPP、MPPI) 以及自主开发的 Stanley 控制器进行对比测试与分析。

3.1 控制器算法原理

Nav2 框架通过 Controller Server 提供可插拔的控制器接口,允许用户根据应用场景选择或开发合适的路径跟踪算法。本项目测试了四种控制器,各有不同的算法原理和适用场景。

3.1.1 DWB 控制器 (Dynamic Window Approach)

DWB 控制器基于动态窗口法,是 Nav2 的默认控制器。其核心思想是在速度空间 (v, ω) 中采样多条候选轨迹,通过多个评价函数 (Critics) 对每条轨迹评分,选择得分最高的轨迹对应的速度指令执行。主要参数包括:仿真时间 sim_time(决定轨迹预测长度)、采样数量 vx/vtheta_samples(影响搜索精度)、以及 PathAlign、GoalDist、

BaseObstacle 等评分权重 (控制行为偏好)。DWB 的优势在于通用性强、参数可调性高, 适合大多数场景; 缺点是计算量随采样数增加而增长。

3.1.2 RPP 控制器 (Regulated Pure Pursuit)

RPP 控制器是对经典纯追踪 (Pure Pursuit) 算法的改进。纯追踪算法通过在路径前方选取一个 lookahead 点, 计算机器人转向该点所需的曲率来生成转向指令。RPP 在此基础上增加了三种速度调节机制: 根据路径曲率调节 (弯道减速)、根据障碍物距离调节 (靠近障碍物减速)、根据目标点距离调节 (接近目标减速)。关键参数包括 lookahead_dist (前瞻距离, 影响路径跟踪平滑度)、desired_linear_vel (期望速度)、rotate_to_heading_angular_vel (原地转向角速度)。RPP 的优势在于计算量小、响应快速, 适合开阔环境和高速行驶; 缺点是在复杂狭窄环境中表现不如 DWB。

3.1.3 MPPI 控制器 (Model Predictive Path Integral)

MPPI 控制器基于模型预测控制 (MPC) 和路径积分理论, 是四种控制器中最复杂的。其工作原理为: 首先根据运动模型预测大量 (batch_size, 本配置为 2000 条) 带噪声的候选轨迹, 然后使用多个 Critics 计算每条轨迹的代价, 最后通过指数加权平均得到最优控制输入。MPPI 使用的 Critics 包括: ConstraintCritic (约束满足)、ObstaclesCritic (障碍物避免)、GoalCritic (目标接近)、PathAlignCritic (路径对齐)、PreferForwardCritic (前进偏好) 等。主要参数包括 time_steps (预测步数, 本配置 56 步)、model_dt (时间步长 0.05s)、temperature (控制探索程度)。MPPI 的优势在于轨迹质量高、避障能力强; 缺点是计算量大, 建议使用 GPU 加速。

3.1.4 Stanley 控制器 (自定义实现)

Stanley 控制器最初由斯坦福大学开发用于 DARPA 无人驾驶挑战赛, 是一种基于横向误差的路径跟踪算法。与纯追踪不同, Stanley 控制器同时考虑航向误差 (heading error) 和横向偏差 (cross-track error), 转向角计算公式为:

$$\delta = \theta_e + \arctan \left(\frac{k \cdot e_{ct}}{v + v_{soft}} \right) \quad (2)$$

其中 θ_e 为航向误差 (机器人朝向与路径切线方向的夹角), e_{ct} 为横向偏差 (机器人到路径的垂直距离), k 为增益系数, v 为当前速度, v_{soft} 为软化速度 (防止低速时分母过小)。这种设计使得机器人在偏离路径时能够同时修正航向和位置。

3.2 Stanley 控制器插件开发

本项目基于 nav2_core::Controller 接口开发了 Stanley 控制器插件, 实现了完整的 Nav2 集成。

3.2.1 插件架构与接口实现

Nav2 控制器插件需要继承 nav2_core::Controller 基类并实现以下核心方法:

configure(): 初始化控制器, 获取节点句柄、TF 缓冲区和代价地图, 并从参数服务器加载配置参数。本实现声明并获取 k_gain (横向误差增益)、softening_speed (软化速度)、max_steer_rad (最大转向角)、lookahead_dist (前瞻距离) 等 10 个参数。

setPlan(): 接收全局路径规划器生成的路径, 存储到成员变量供后续使用。使用互斥锁保护路径访问以确保线程安全。

computeVelocityCommands(): 核心控制逻辑, 根据当前位姿和速度计算下一时刻的速度指令。实现流程为: 首先检查是否到达目标 (位置误差 $< \text{goal_tolerance_lin}$ 且朝向误差 $< \text{goal_tolerance_yaw}$), 若到达则返回零速度; 否则在路径上寻找前瞻点, 计算航向误差和横向偏差, 应用 Stanley 公式得到转向角, 最后根据转向角大小调节线速度 (转向越大速度越低)。

setSpeedLimit(): 支持外部速度限制接口, 可以是绝对值或相对比例。

3.2.2 核心算法实现

控制器的核心计算在 computeVelocityCommands() 中实现。首先通过 findTargetPoint() 在路径上搜索第一个距离机器人超过 lookahead_dist 的点作为目标点; 然后计算路径在该点附近的切线方向 (通过相邻两点连线确定); 接着计算横向偏差 e_{ct} —— 使用向量叉积确定机器人相对路径的左右偏移及其符号; 最后应用 Stanley 公式计算转向角并限幅。线速度采用指数衰减策略: $v = v_{max} \cdot e^{-k_{slow} \cdot |\delta|}$, 即转向角越大速度越低, 确保弯道安全。接近目标时额外降低速度并增强转向以精确停靠。

3.2.3 插件注册与编译

控制器通过 pluginlib 机制注册为 Nav2 插件。plugin.xml 定义插件元信息:

```

1 <library path="stanley_controller">
2   <class type="custom_controller::StanleyController"
3     base_class_type="nav2_core::Controller">
4     <description>Stanley-based path follower</description>
5   </class>
6 </library>

```

CMakeLists.txt 配置编译规则, 使用 ament_cmake 构建系统, 链接 nav2_core、pluginlib、tf2 等依赖库, 最终生成 libstanley_controller.so 共享库。编译安装后, 可在 Nav2 参数文件中通过 plugin: "custom_controller::StanleyController" 指定使用该控制器。

3.2.4 关键参数说明与设计决策

Stanley 控制器共有 10 个可配置参数, 分为控制算法参数、速度限制参数和目标判定参数三类。

表 4: Stanley 控制器参数详解

参数名	类型	默认值	说明
控制算法参数			
k_gain	double	1.8	横向误差增益, 控制对路径偏离的响应强度
softening_speed	double	0.1	软化速度 (m/s), 防止低速时除零
max_steer_rad	double	0.7	最大转向角 (rad), 约 40°
lookahead_dist	double	0.7	前瞻距离 (m), 寻找目标点的距离阈值
slow_down_gain	double	2.5	减速增益, 控制转向时的速度衰减程度
速度限制参数			
max_linear_speed	double	0.35	最大线速度 (m/s)
min_linear_speed	double	0.05	最小线速度 (m/s), 防止完全停止
max_angular_speed	double	1.0	最大角速度 (rad/s)
目标判定参数			
goal_tolerance_lin	double	0.05	位置容差 (m), 判定到达目标的距离阈值
goal_tolerance_yaw	double	0.08	朝向容差 (rad), 约 4.6°

控制算法参数设计决策 k_gain(横向误差增益): 该参数决定了 Stanley 公式中横向误差项的权重。根据 Stanley 原始论文, 增益值通常在 1.0-3.0 之间。本实现选择 1.8 的原因是:(1) 过小的增益 (如 0.5) 会导致路径跟踪不紧密, 机器人偏离路径后收敛缓慢;(2) 过大的增益 (如 5.0) 在差速驱动机器人上容易引发振荡, 因为差速底盘的转向响应与 Ackermann 模型不同;(3) 1.8 在仿真测试中表现出较好的跟踪精度与稳定性平衡。

softening_speed(软化速度): Stanley 公式包含 $v + v_{soft}$ 作为分母, 当机器人速度接近零时, 若无软化项会导致转向角趋于无穷大。设置 0.1 m/s 的软化速度确保:(1) 静止启动时不会产生剧烈转向;(2) 低速精确停靠时控制平稳。该值应略大于 min_linear_speed 以确保有效性。

lookahead_dist(前瞻距离): 前瞻距离决定了控制器“看多远”来确定目标点。设置 0.7m 的考量:(1) 过短 (如 0.2m) 会导致控制器过于“近视”, 在弯道处反应滞后;(2) 过长 (如 2.0m) 会使控制器忽略近处的路径细节, 切弯过大;(3) 0.7m 约为机器人直径的 2.5 倍, 在仓库通道环境中提供足够的前瞻同时保持路径跟踪精度。

slow_down_gain(减速增益): 线速度通过 $v = v_{max} \cdot e^{-k_{slow} \cdot |\delta|}$ 计算, 该参数控制转向角对速度的影响程度。设置 2.5 意味着:(1) 转向角为 0.4rad(≈23°) 时, 速度降至最大值的 37%;(2) 转向角为 0.7rad(最大) 时, 速度降至最大值的 17%;(3) 这种指数衰减策略确保弯道安全同时避免不必要的减速。

速度限制参数设计决策 max_linear_speed 设为 0.35 m/s, 显著低于 DWB 的 0.5 m/s。这是考虑到:(1) Stanley 控制器设计用于精确对齐阶段, 不需要高速;(2) 差速驱动底盘在高速下执行 Stanley 算法的转向角映射可能产生不稳定;(3) 较低速度有助于提高停靠精度。

min_linear_speed 设为 0.05 m/s, 确保机器人始终保持微小前进, 避免陷入纯转向状态。这对于差速驱动机器人尤为重要, 因为 Stanley 算法假设车辆持续前进。

目标判定参数设计决策 goal_tolerance_lin 设为 0.05m(5cm), 远小于 DWB 的 0.25m。这是精确停车场景的核心需求——在货架对接、充电桩停靠等应用中, 5cm 的精度通常是必要的。

goal_tolerance_yaw 设为 0.08rad(约 4.6°), 确保机器人停靠后朝向正确。对于需要后续操作 (如机械臂抓取) 的场景, 正确的朝向与位置同样重要。

参数调优建议 针对不同应用场景, 建议调整以下参数:(1) **开阔环境**: 可适当增大 max_linear_speed 至 0.5m/s, 减小 k_gain 至 1.2 以获得更平滑的轨迹;(2) **狭窄通道**: 增大 k_gain 至 2.5 提高路径跟踪紧密度, 减小 look-ahead_dist 至 0.4m 避免切弯;(3) **高精度停靠**: 减小 goal_tolerance_lin 至 0.03m, 增大 slow_down_gain 至 3.0 确保低速接近。

3.3 控制器对比测试

3.3.1 测试配置

四种控制器均使用 baseline 参数配置, 在相同的 12 点测试路线上进行评估。控制器特定参数配置如下:

表 5: 控制器核心参数配置

参数类别	DWB	RPP	MPPI	Stanley
控制频率 (Hz)	20	20	30	20
最大线速度 (m/s)	0.5	0.5	0.5	0.35
最大角速度 (rad/s)	1.0	1.8	1.9	1.0
前瞻距离/仿真时间	1.7s	0.6m	56 步 × 0.05s	0.7m
采样/批次	20 × 20	-	2000	-
目标容差 (m)	0.25	0.25	0.25	0.05

3.3.2 测试结果

表 6: 控制器性能对比

控制器	成功率 (%)	总时间 (s)	总距离 (m)	平均速度 (m/s)	位置误差 (m)	朝向误差 (°)
DWB	100.0	554.1	248.8	0.449	0.255	16.9
RPP	100.0	535.4	248.2	0.464	0.252	18.6
MPPI	100.0	615.1	248.8	0.405	0.240	17.9
Stanley	8.3	1158.7	10.0	0.009	0.115*	17.2*

*Stanley 仅成功 1 个点, 误差数据不具统计意义

3.3.3 图表化分析

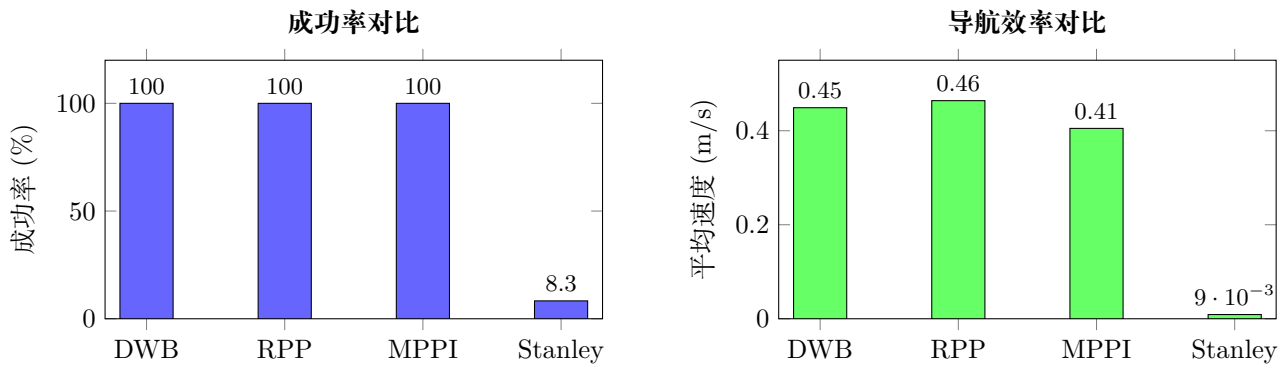


图 6: 控制器成功率与导航效率对比 (Stanley 因大量失败导致有效速度极低)

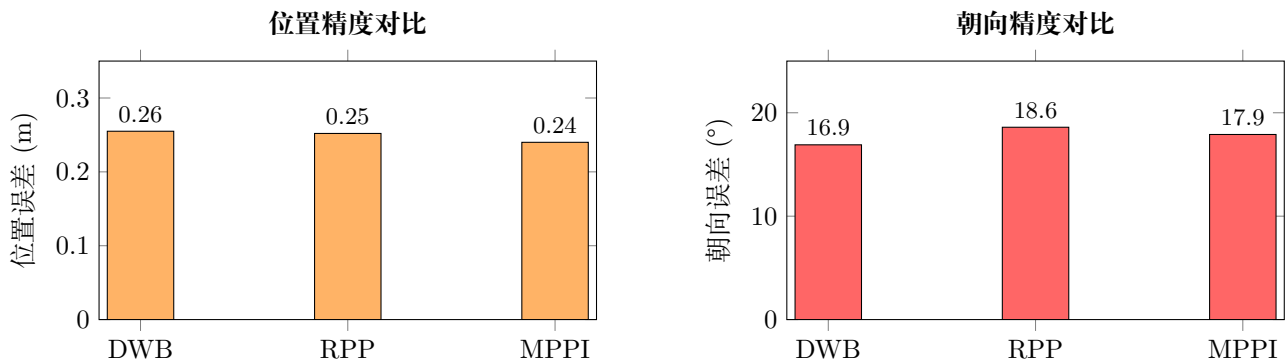


图 7: 三种成功控制器的精度对比 (Stanley 数据因样本不足未纳入)

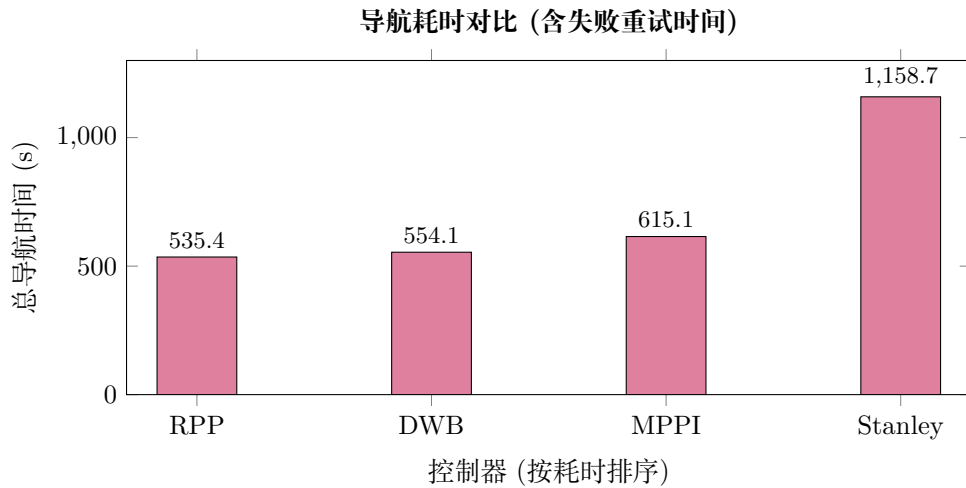


图 8: 四种控制器总耗时对比:RPP 最快,Stanley 因多次超时导致总耗时最长

从图表可以直观看出:(1) DWB、RPP、MPPI 三种 Nav2 原生控制器均达到 100% 成功率,Stanley 仅 8.3%;(2) RPP 平均速度最高 (0.464 m/s),MPPI 最低 (0.405 m/s);(3) 位置精度 MPPI 最优 (0.240m), 朝向精度 DWB 最优 (16.9°);(4) 总耗时 RPP 最短 (535.4s),Stanley 因大量超时导致总耗时超过 1150 秒。

3.3.4 结果分析

成功率分析:DWB、RPP 和 MPPI 三种 Nav2 原生控制器均达到 100% 成功率,表现稳定可靠。自定义的 Stanley 控制器仅完成 12 个目标点中的 1 个 (8.3% 成功率),其余 11 个均超时失败。分析失败原因:Stanley 控制器设计用于车辆模型 (Ackermann 转向),其假设前轮可以独立转向,但 TIAGo 采用差速驱动底盘,只能通过左右轮差速实现转向。虽然本实现将转向角映射为角速度指令,但在需要大角度转向的场景 (如狭窄通道调头) 表现不佳,容易陷入振荡或无法及时转向导致超时。

效率分析:在三个成功的控制器中,RPP 最快 (535.4s, 平均 0.464 m/s),DWB 次之 (554.1s, 0.449 m/s),MPPI 最慢 (615.1s, 0.405 m/s)。RPP 的高效率源于其简单的几何计算和较高的目标速度设置;MPPI 较慢的原因是其控制频率虽高 (30Hz) 但每次决策需要评估 2000 条轨迹,计算开销大,导致实际控制响应略有延迟。

精度分析:位置精度方面,MPPI 最优 (0.240m),略优于 RPP (0.252m) 和 DWB (0.255m);朝向精度方面,DWB 最优 (16.9°),优于 MPPI (17.9°) 和 RPP (18.6°)。MPPI 通过大量轨迹采样能够找到更优的到达路径,位置精度最高;DWB 的 RotateToGoal 评分函数专门优化到达朝向,因此朝向精度最好;RPP 采用 rotate_to_heading 策略在到达前进行原地转向调整,可能因时间不足导致朝向误差略大。

3.3.5 控制器选型建议

基于测试结果,针对不同应用场景推荐如下:

通用场景:推荐 DWB 控制器。作为 Nav2 默认控制器,DWB 在效率和精度间取得良好平衡,参数可调性强,适合大多数室内导航任务。

追求效率:推荐 RPP 控制器。RPP 计算量最小,响应最快,适合开阔环境、对实时性要求高或计算资源受限的场景。

追求精度:推荐 MPPI 控制器。MPPI 通过大规模采样获得最优轨迹,位置精度最高,适合对停靠精度要求严格的场景,但需要较强的计算能力。

Stanley 控制器:当前实现不适合差速驱动机器人,需要进一步优化以适配 TIAGo 的运动学模型,或改用专为差速驱动设计的控制算法。

4 行为树子任务设计

本部分介绍 Nav2 行为树的基本原理,以及本项目设计的动态多控制器行为树方案。该行为树的核心特点是集成不少于三种控制器,并具备控制器 ID 的动态切换能力。

4.1 行为树基础

4.1.1 行为树概述

行为树 (Behavior Tree, BT) 是一种用于控制决策流程的树状结构, 最初在游戏 AI 中广泛应用, 近年来被引入机器人领域。与传统有限状态机 (FSM) 相比, 行为树具有模块化程度高、可复用性强、易于调试和扩展等优势。Nav2 采用 BehaviorTree.CPP 库实现行为树, 通过 XML 文件定义导航任务的执行流程。

行为树由多种类型的节点组成, 每个节点执行后返回三种状态之一: SUCCESS(成功)、FAILURE(失败) 或 RUNNING(运行中)。主要节点类型包括:

控制节点 (Control Nodes): 决定子节点的执行顺序和逻辑。**Sequence**(序列节点) 按顺序执行子节点, 任一子节点失败则整体失败, 全部成功才返回成功——适合表达”依次完成 A、B、C”的逻辑。**Fallback**(回退节点, 又称 Selector) 依次尝试子节点直到某个成功, 全部失败才返回失败——适合表达”尝试 A, 不行就试 B”的容错逻辑。

动作节点 (Action Nodes): 执行具体操作并返回结果。Nav2 提供丰富的动作节点, 如 **ComputePathToPose**(计算路径)、**FollowPath**(跟随路径)、**BackUp**(后退)、**Wait**(等待)、**ClearEntireCostmap**(清除代价地图) 等。其中 **FollowPath** 节点支持通过 **controller_id** 参数指定使用的控制器, 这是实现动态控制器切换的基础。

条件节点 (Condition Nodes): 检查某个条件是否满足。如 **GoalReached** 检查是否到达目标, 是精确停车判断的关键节点。

装饰节点 (Decorator Nodes): 修饰单个子节点的行为。**SpeedController** 限制导航速度, 通过 **min_rate** 和 **max_rate** 参数控制速度范围, 是实现分阶段减速的核心节点。

4.1.2 Blackboard 与动态参数传递

Blackboard(黑板) 是行为树的共享数据存储机制, 允许节点之间传递数据。通过 **SetBlackboard** 节点可以在运行时修改黑板变量, 而其他节点可以通过 **{variable_name}** 语法引用这些变量。这种机制使得**同一个 FollowPath 节点可以在不同阶段调用不同的控制器**——只需在执行前通过 **SetBlackboard** 修改 **controller_id** 变量即可。

4.2 动态多控制器行为树设计

本项目设计的 **dynamic_multi_controller.xml** 行为树实现了精确停车功能, 其核心特点是**集成三种不同的控制器**, 并通过 **Blackboard** 机制在运行时**动态切换控制器 ID**, 充分发挥各控制器在不同阶段的优势。

4.2.1 三种控制器配置

行为树使用的三种控制器需要在 Nav2 参数文件中预先配置:

表 7: 动态多控制器行为树使用的三种控制器

控制器 ID	控制器类型	特点与用途
FollowPath	DWB(快速配置)	高速响应,max_vel_x=0.5m/s, 用于快速接近阶段
SlowFollowPath	DWB(慢速配置)	平稳跟踪,max_vel_x=0.3m/s, 用于缓行接近阶段
StanleyFollowPath	自定义 Stanley	横向误差校正,max_vel_x=0.15m/s, 用于精确对齐阶段

4.2.2 行为树整体结构

行为树采用三阶段顺序执行的 **Sequence** 结构, 每个阶段开始前通过 **SetBlackboard** 切换控制器:

```
1 <root main_tree_to_execute="MainTree">
2   <BehaviorTree ID="MainTree">
3     <Sequence name="MultiControllerNav">
4       <!-- 阶段1: 快速接近 (FollowPath) -->
5       <!-- 阶段2: 慢速精调 (SlowFollowPath) -->
6       <!-- 阶段3: 精确对齐 (StanleyFollowPath) -->
7     </Sequence>
8   </BehaviorTree>
9 </root>
```

4.2.3 阶段一：快速接近 (FollowPath 控制器)

第一阶段使用 DWB 快速配置控制器, 以较高速度完成大部分路程:

```
1 <!-- 动态设置控制器ID为FollowPath -->
2 <SetBlackboard output_key="controller_id" value="FollowPath"/>
3
4 <!-- 路径规划(带Fallback恢复机制) -->
5 <Fallback name="PlanFast">
6   <ComputePathToPose goal="{goal}" path="{fast_path}"
7     planner_id="GridBased"/>
8   <Sequence>
9     <ClearEntireCostmap service_name="global_costmap/..." />
10    <Wait wait_duration="0.5"/>
11    <ComputePathToPose goal="{goal}" path="{fast_path}"
12      planner_id="GridBased"/>
13  </Sequence>
14 </Fallback>
15
16 <!-- 路径跟随(使用黑板中的controller_id) -->
17 <Fallback name="FollowFast">
18   <FollowPath path="{fast_path}" controller_id="{controller_id}"/>
19   <Sequence>
20     <ClearEntireCostmap service_name="local_costmap/..." />
21     <Wait wait_duration="0.5"/>
22     <FollowPath path="{fast_path}" controller_id="{controller_id}"/>
23   </Sequence>
24 </Fallback>
```

关键设计点:(1) SetBlackboard 将 controller_id 设为"FollowPath", 后续 FollowPath 节点通过 {controller_id} 引用该值;(2) 路径规划和跟随都使用 Fallback 包装, 失败时自动清除代价地图并重试, 增强鲁棒性;(3) 此阶段使用 DWB 快速配置, 以 0.5m/s 的速度高效完成长距离导航。

4.2.4 阶段二：慢速精调 (SlowFollowPath 控制器)

第二阶段切换到 DWB 慢速配置, 配合 SpeedController 进一步限速:

```
1 <!-- 动态切换控制器ID为SlowFollowPath -->
2 <SetBlackboard output_key="controller_id" value="SlowFollowPath"/>
3 <Wait wait_duration="0.3"/>
4
5 <!-- 重新规划路径 -->
6 <ComputePathToPose goal="{goal}" path="{slow_path}"
7   planner_id="GridBased"/>
8
9 <!-- 使用SpeedController限制速度范围 -->
10 <SpeedController min_rate="0.1" max_rate="0.3" filter_duration="0.5">
11   <FollowPath path="{slow_path}" controller_id="{controller_id}"/>
12 </SpeedController>
```

关键设计点:(1) 通过 SetBlackboard 将 controller_id 切换为"SlowFollowPath", 同一个 FollowPath 节点现在调用的是慢速 DWB 控制器;(2) Wait 节点确保控制器切换后系统稳定;(3) SpeedController 装饰节点将速度限制在 0.1-0.3m/s 范围内,filter_duration=0.5s 实现平滑过渡;(4) 此阶段重新规划路径, 确保从当前位置到目标的最优路径。

4.2.5 阶段三：精确对齐 (StanleyFollowPath 控制器)

第三阶段切换到自定义 Stanley 控制器, 利用其横向误差校正能力实现精确对齐:

```
1 <!-- 动态切换控制器ID为StanleyFollowPath -->
2 <SetBlackboard output_key="controller_id" value="StanleyFollowPath"/>
3 <Wait wait_duration="0.3"/>
4
5 <Fallback name="FinalAdjust">
6   <!-- 如果已到达目标,直接成功 -->
7   <GoalReached/>
8
9   <!-- 否则执行微调序列 -->
10   <Sequence>
11     <!-- 后退0.12m获得更好的接近角度 -->
```

```

12 <BackUp backup_dist="0.12" backup_speed="0.03"
13       time_allowance="6.0"/>
14 <Wait wait_duration="0.3"/>
15
16 <!-- 重新规划最终路径 -->
17 <ComputePathToPose goal="{goal}" path="{final_path}"
18                   planner_id="GridBased"/>
19
20 <!-- 超低速精确跟随 -->
21 <SpeedController min_rate="0.05" max_rate="0.15"
22                 filter_duration="0.3">
23   <FollowPath path="{final_path}" controller_id="{controller_id}"/>
24 </SpeedController>
25 </Sequence>
26 </Fallback>

```

关键设计点:(1) 切换到 Stanley 控制器, 利用其 $\delta = \theta_e + \arctan(k \cdot e_{ct}/v)$ 公式同时校正航向误差和横向偏差;(2) Fallback 结构首先检查 GoalReached, 若已满足目标容差则跳过微调;(3) 若未到达, 执行 BackUp 后退 0.12m——这是精确停车的关键技巧, 后退可以让机器人退出局部最优位置, 获得更好的重新接近角度;(4) SpeedController 将速度限制在 0.05-0.15m/s 的超低速范围, 确保最终阶段的精确控制。

4.2.6 动态控制器切换机制总结

本行为树的核心点在于通过 Blackboard 实现控制器 ID 的动态切换:

1. **统一的 FollowPath 节点:** 整个行为树中的所有 FollowPath 节点都使用 controller_id="{controller_id}" 引用黑板变量, 而非硬编码控制器名称。
2. **阶段性切换:** 每个阶段开始前通过 SetBlackboard output_key="controller_id" value="XXX" 修改黑板中的控制器 ID。
3. **三种控制器协同:** FollowPath(快速)→SlowFollowPath(平稳)→StanleyFollowPath(精确), 各控制器在其擅长的阶段发挥作用。
4. **运行时决策:** 控制器切换发生在行为树执行过程中, 无需重启导航系统, 实现了真正的动态切换。

这种设计思想是“物尽其用”——快速阶段使用响应快、效率高的 DWB 控制器; 接近目标时切换到慢速配置确保平稳; 最终对齐使用具有横向误差校正能力的 Stanley 控制器提高精度。通过行为树的模块化设计, 这三种控制器被有机地组合在一起, 形成完整的精确停车解决方案。

4.3 行为树配置与部署

在 Nav2 参数文件中配置三种控制器和行为树:

```

1 controller_server:
2   ros__parameters:
3     controller_plugins: ["FollowPath", "SlowFollowPath",
4                         "StanleyFollowPath"]
5
6   FollowPath:
7     plugin: "dwb_core::DWBLocalPlanner"
8     max_vel_x: 0.5
9     # ... 快速配置参数
10
11   SlowFollowPath:
12     plugin: "dwb_core::DWBLocalPlanner"
13     max_vel_x: 0.3
14     # ... 慢速配置参数
15
16   StanleyFollowPath:
17     plugin: "custom_controller::StanleyController"
18     max_linear_speed: 0.15
19     # ... Stanley 配置参数
20
21 bt_navigator:
22   ros__parameters:
23     default_bt_xml_filename: "dynamic_multi_controller.xml"

```

5 总结与展望

5.1 主要结论

通过本项目的研究,得出以下主要结论:

1. **参数优化需要系统性方法。**Nav2 涉及数十个相互关联的参数,孤立调整某个参数往往无法达到预期效果。本项目采用的控制变量法和自动化批量测试框架,是进行系统性参数优化的有效途径。
2. **控制器选择应匹配应用场景。**DWB 适合通用场景,RPP 适合追求效率的开阔环境,MPPI 适合追求精度的复杂场景。没有”最优”控制器,只有”最适合”的控制器。
3. **行为树是实现复杂导航任务的利器。**通过行为树的模块化设计,可以将多种控制器、恢复行为、条件判断有机组合,实现比单一控制器更强大的导航能力。
4. **差速驱动机器人需要专门的控制算法。**将为 Ackermann 模型设计的 Stanley 算法直接应用于差速驱动底盘效果不佳,说明控制算法与机器人运动学模型的匹配至关重要。

5.2 未来展望

本项目仍有若干方面值得进一步研究和改进:

(1) **Stanley 控制器适配优化。**当前 Stanley 实现将转向角直接映射为角速度,未充分考虑差速驱动的运动学特性。后续可研究专门针对差速底盘的 Stanley 变体,或引入模型预测控制 (MPC) 框架提高适应性。

(2) **动态障碍物避障。**当前测试环境为静态场景,实际仓库中存在移动的叉车、行人等动态障碍物。可结合 MPPI 控制器的预测能力和动态代价地图,提升动态环境下的导航安全性。

(3) **多机器人协同调度。**单机器人导航是多机器人仓库系统的基础。后续可研究基于 Nav2 的多机器人路径规划、冲突检测与解决、任务调度等问题。

(4) **实机验证与部署。**本项目所有实验均在仿真环境中进行。将优化后的参数配置和行为树部署到真实 TIAGo 机器人上,验证仿真结果的可迁移性,是走向实际应用的必经之路。

(5) **深度学习增强。**探索将深度强化学习 (DRL) 用于控制器参数自适应调整,或使用神经网络替代传统的评价函数,可能进一步提升导航性能。