ROS2 机器人自主导航

概述：如今，越来越多的场景开始配备配送服务机器人。比如：将物品放入机器人中，然后指定房间编号，机器人就能自主地将物品运送到对应的位置。那么，机器人是如何实现自主移动的呢？

移动机器人的核心在于“自主导航”能力。它需要首先感知当前所处的环境，并在已有地图中准确定位自己，然后根据任务目标和环境信息规划路径，最后控制自身运动完成移动过程。

本章节我们将借助 ROS 2 中强大的开源工具和模块，逐步构建一个具备自主导航能力的机器人系统。

一． 机器人导航概述及 SLAM 基本介绍

假设你是一名仓库配送员，当前的任务是去 A 区货架取一个包裹。首先你需要知道 A 区在哪里，其次要知道从当前位置如何走过去。对于移动机器人来说，自主导航其实也主要解决这两个核心问题：

我现在在哪里？（定位问题）

我应该怎么走？（路径规划问题）

7.1 同步定位与地图构建（SLAM）基本概述

要解决上述两个问题：确定当前位置和知道目标点在哪 -> 定位与建图。

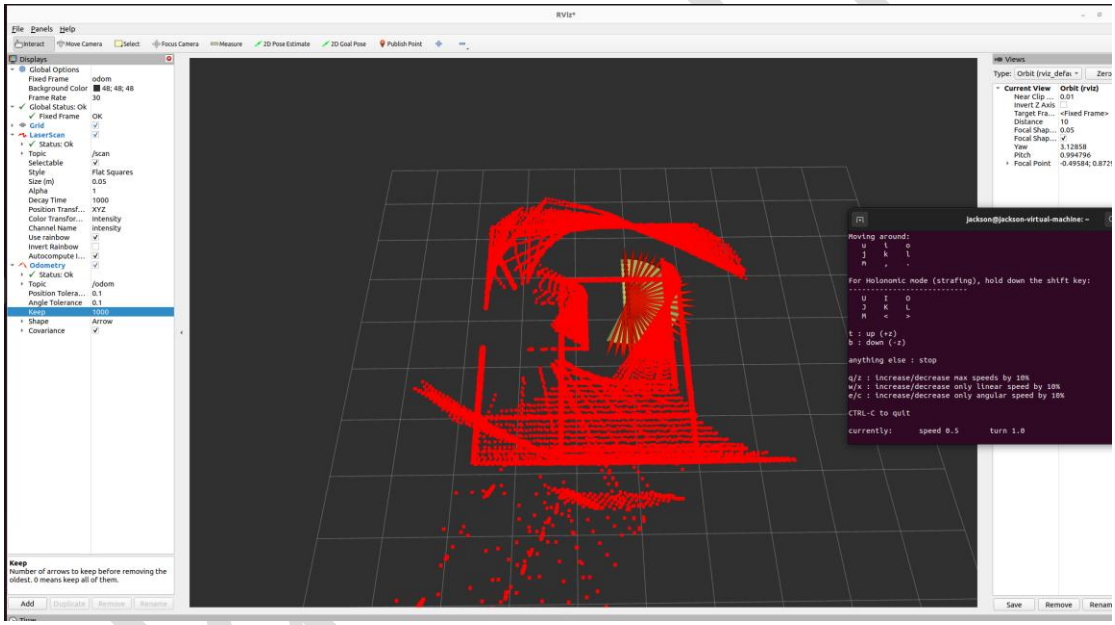
如果使用卫星导航（如 GPS）可以知道自己的经纬度位置，这对户外导航有效。但在室内或地下等无 GPS 信号的环境中，我们就需要依靠传感器（如激光雷达、视觉相机）获取环境信息，来进行同步定位与地图构建（SLAM）。

也就是说，机器人一边移动一边使用激光雷达获取周围障碍物的位置，同时记录沿途的路径数据，从而构建一张可用于导航的地图，并实时推算自身在地图中的位置。这样即使在初始没有地图的环境中，机器人也能实现边走边“画地

图”。比如在工厂或仓库中，机器人每次进入一个新区域时，都可以通过这种方式自动构建地图；而在熟悉的环境中，机器人也可以利用已有地图，直接进行定位和路径规划。

在上一章中，我们已经获取了机器人姿态与激光雷达数据。在本章中，我们将以此为基础，引入 SLAM 技术，实现机器人自主定位与建图功能，为路径规划和导航执行做好准备。

重新启动仿真和 Rviz2 中按左侧调节参数，包括 Decay Time (保留过去 1000s 的雷达数据)，Odometry 下的 Keep（保留过去 1000 个里程计数据）使用键盘方法控制机器人移动，发现轨迹和障碍物信息已经被记录 但转弯出现了较大误差。



7.2 机器人导航

在完成地图构建与定位之后，接下来要解决的关键问题就是：如何规划路径并自主移动到目标位置。通常，路径规划分为两个阶段：全局路径规划与局部路径调整。

a) 全局路径规划

全局路径规划的目标是在已知地图中，从当前位置到目标点生成一条最优路径。这条路径应满足一些基本要求，例如：

避开障碍物；

路径长度尽可能短；

通行代价低、风险小；

考虑通行效率与资源成本（如避开人流量大或地面不平整区域）。

规划完成后，机器人会得到一条整体的行进路线。这条路径一般是静态的，即在没有新障碍物或地图变化的情况下，机器人可以按此路径前进。

b）局部路径规划与动态避障

在实际移动过程中，机器人可能会遇到突发情况，例如：

前方突然出现障碍物；

遇到临时封闭区域或通道堵塞；

识别到动态物体（如行人或其他机器人）阻挡路径。

此时，机器人需要在当前环境感知数据的基础上，动态调整路径，重新局部规划新的可行路线。这一过程称为局部路径规划，也是导航系统中的关键模块之一。

局部路径规划的能力决定了机器人是否具备动态避障、自主决策与连续运行的能力。

c）导航过程中的异常处理：恢复行为

除了常规的避障和路径调整，导航系统还需要应对某些特殊情况，例如：

机器人陷入狭小空间或地图盲区；

无法绕开障碍；

被困在某个位置长时间无法前进。

为了帮助机器人从这些“困境”中脱离，导航系统通常会设计“恢复行为”。例如：

原地旋转尝试脱困；

倒车撤离；

播放语音提示等待人工干预等。

这类机制有助于提高导航系统的鲁棒性和任务完成率。

d）导航行为的整体协调

当前主流的机器人导航系统，通常会将路径规划、障碍检测、避障行为与异常恢复这几个功能模块集成在一起，并结合实时的位置信息进行协调控制。

全局路径规划器：根据起点与目标点，生成静态路线；

局部路径规划器：动态应对障碍，实时调整轨迹；

控制器：接收路径输出，驱动底盘执行动作；

恢复模块：在导航失败或卡顿时介入处理，提升系统鲁棒性。

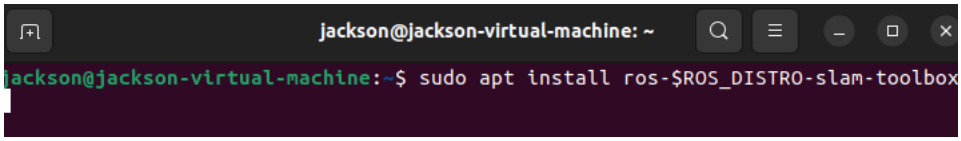
这些模块的协同运行，是实现机器人稳定自主导航的关键。

至此，关于机器人导航的基本概念与技术框架已做初步介绍。在后续内容中，将深入讲解 ROS 2 框架下的实际导航实现——Navigation 2 系统（Nav2），我们将进一步学习其配置方法与模块集成流程。

二. 使用 slam_toolbox 完成建图

SLAM 通过传感器获取周围环境信息然后进行定位建图，ROS2 提供了很多 SLAM 功能包 下面我们用针对二维激光场景的 slam_toolbox 来构建我们第一张地图

安装命令：



7.3 构建第一张导航地图

接着去配置功能包和工作空间 和前面一样 把 firstbot_description 文件复制到

chapt7/chapt7_ws/src 打开 vsCode 后终端三连后启动 launch 文件仿真，随后新开终端启动 slam_toolbox 在线建图 他所产生的地图会通过 /map 话题进行发布，所以我们可以用 rviz 订阅话题进行显示

```
jackson@jackson-virtual-machine:~/chapt7/chapt7_ws$ ros2 launch slam_toolbox online_async_launch.py use_sim_time:=True
```

再新开终端启动 rviz2 并配置如左下图

之前我们说如果视图设置成 map，则会导致机器人显示不正常，现在是正常的，为啥?

因为 slam_toolbox 定位没有直接发布从 map 到 base_footprint，而是发布到了 map 到 odom，再由 odom 到 base_footprint.

分工清晰:

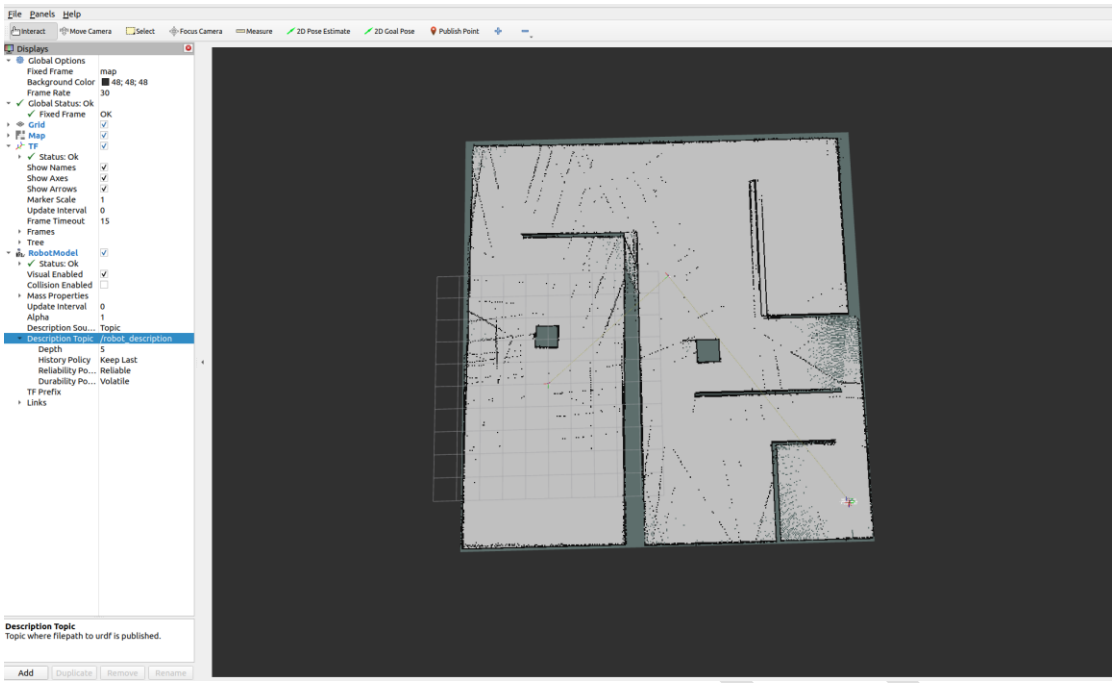
- `map→odom`：纠正里程计累计误差，把机器人"拉回"全局地图。
- `odom→base_footprint`：高频发布，由硬件驱动或里程计节点提供，保持机器人运动的连续性。

降低延迟与计算量:
slam_toolbox 专注在校准算法，不必处理高频的里程计更新；里程计节点也不需要承担地图匹配的重载。

容错与灵活:
如果 slam 算法短暂失效，里程计链路依然可以让机器人在本地短期自主运动；同样，如果里程计短暂中断，slam 依旧维护全局一致性。

如果你让 SLAM 插件直接发布 `map → base_footprint`：

- **频率低**：SLAM 校正频率通常比里程计低很多，画面会有明显延迟。
- **瞬时跳动**：每次校正都会"拉"动机器人坐标，造成抖动或位置跳跃。
- **丢失里程计连续性**：里程计累积的微小移动无法反映在 TF 树里，显示不连贯。

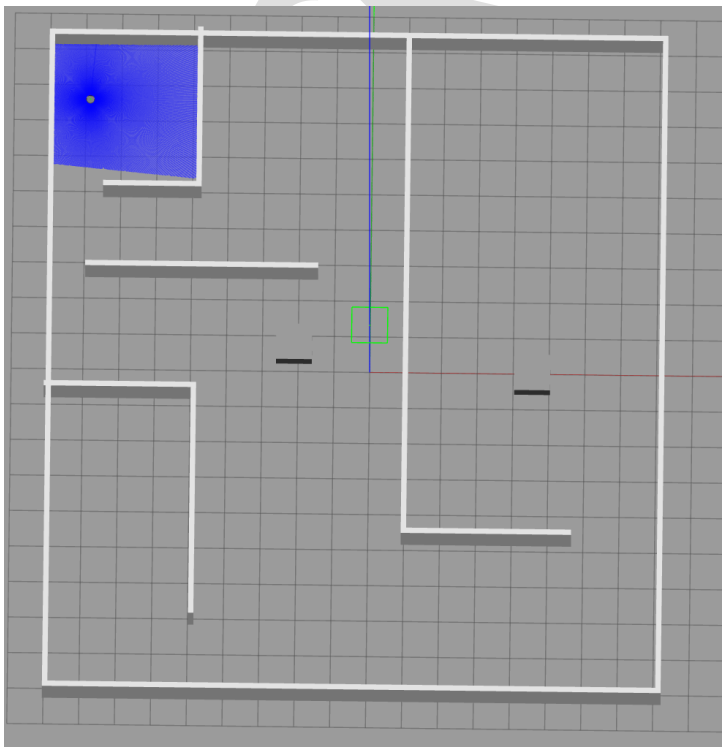


在这个图中，白色表示无障碍物可以行走，黑色代表有障碍物（线条），灰色表示未知（雷达未检测到 大概率是障碍物的内部）

可以看到与下图轮廓高度吻合

这里如果出现漂移等状况总之就是图片不成形

多半都是因为环境太小了，可以把轮廓画的大一点就好了



7.4 将地图保存为文件

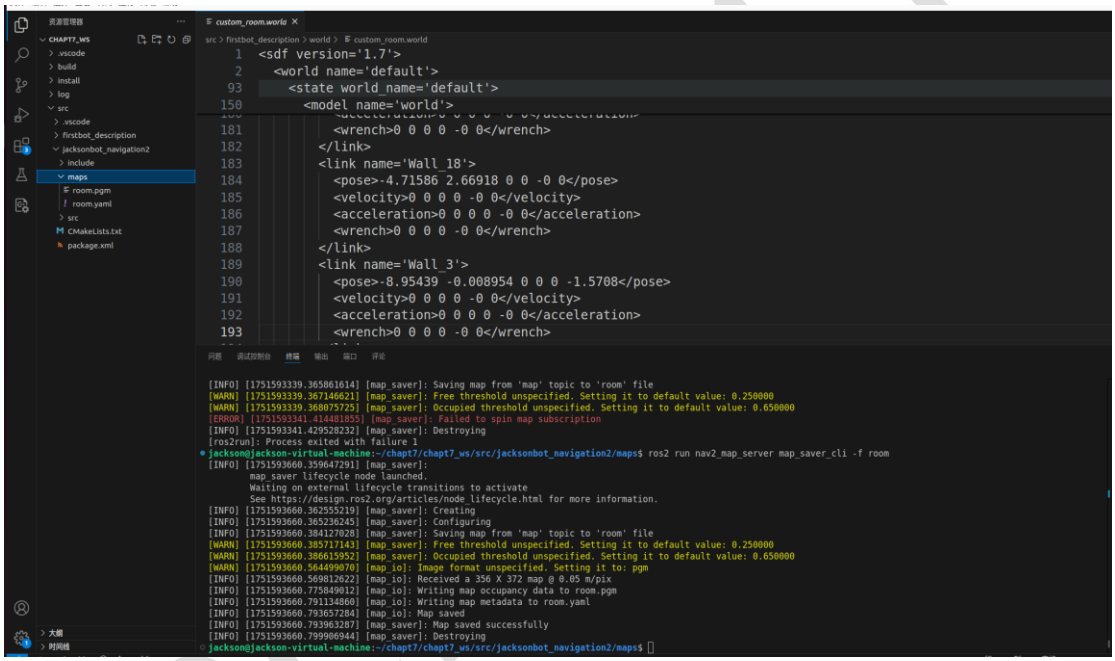
保存地图使用工具：nav2_map_server 需要安装

```
jackson@jackson-virtual-machine:~$ sudo apt install ros-$ROS_DISTRO-nav2-map-ser
ver
```

为了方便使用，我们创建一个导航功能包将地图放到里面以便后续使用

```
jackson@jackson-virtual-machine:~/chapt7/chapt7_ws$ cd src
jackson@jackson-virtual-machine:~/chapt7/chapt7_ws/src$ ros2 pkg create jacksonbot_navigation2
```

运行下面代码存储地图 （注：如果第一次不行就再跑两次，突然就好了）



关于命令：

`ros2 run`	执行一个 ROS 2 可执行文件
`nav2_map_server`	包名，`map_saver_cli` 可执行文件所在的包
`map_saver_cli`	可执行文件，负责将当前 `/map` 话题中的地图保存到本地
`-f room`	指定保存的地图前缀名为 `room`，将生成 `room.pgm` 和 `room.yaml` 两个文件

看下生成的 yaml 文件 （图片的单位是像素）

```yaml
image: room.pgm
mode: trinary
resolution: 0.05
origin: [-14, -11.4, 0]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.25
```

其中 mode: trinary 表示地图中每个像素点有三种状态可能

Black: occupied　　White: free　　Grey: unknown

Resolution: 分辨率，表示每个像素对应的物理尺寸为 0.05m

Negate 表示是否取反地图

剩下两个 thresh 表示设置三个状态的分界线：

< 0.25 空闲　　>0.65 占据　　中间：未知

为啥不设置成有障碍物 1 无 0?

因为地图由传感器观测数据生成，传感器有噪声，所以某个地方的状态不能确定，于是把地图划成一个个小格子，将格子被占的概率与像素映射。

这个地图也可以称为：占据栅格地图

地图讲解到这，下面我们就可以配置导航了！

三． 机器人导航框架：Navigation 2

Navigation 2 是一个开源的机器人导航框架，其核心目标是让机器人能够自主地、安全地从起点 A 移动到目标点 B。它的重点不在于手动控制机器人运动，而是赋予机器人自主导航的能力。为实现这一目标，Navigation 2 提供了诸如路径规划、避障以及自动脱困等基础功能，是实现智能移动的关键组成部分。

7.5 Navigation 2 的介绍与安装

在讲解 Navigation 2 前我们先来看一个新的概念：“行为树”

ROS 2 的 Navigation2 使用行为树来管理机器人的导航流程

行为树是一种用于控制系统行为执行流程的树状结构。它最早被广泛用于游戏 AI，但现在也广泛应用于**机器人自主决策系统**，例如 ROS 2 中的 **Navigation2 框架**。

> **核心目的**：让机器人能灵活、模块化地做出决策，按优先级和状态动态执行行为任务（如导航、避障、重试等）。

🧩 行为树的基本结构

一棵行为树由**节点**组成，分为以下几种类型：

1. 控制节点（Control Nodes）

这些是树的"中间节点"，用来控制流程走向。

- **Sequence（顺序）**：
  - 从左到右依次执行子节点；
  - 遇到第一个失败（`FAILURE`）就返回失败；
  - 所有都成功才返回成功。
- **Selector（选择）**：
  - 从左到右依次尝试子节点；
  - 遇到第一个成功（`SUCCESS`）就返回成功；
  - 所有都失败才返回失败。
- **Parallel（并行）**：
  - 同时执行所有子节点；
  - 成功或失败的判定条件可配置（如"成功数≥N"）。

2. 叶子节点（Leaf Nodes）

这些是最终执行的动作或检查条件。

- **Action Node**：执行具体动作，如"移动到目标"、"发送速度指令"。
- **Condition Node**：检查条件，如"路径是否清晰"、"目标是否已到达"。

---

3. 装饰节点（Decorator）

用来修改子节点的返回状态，例如：

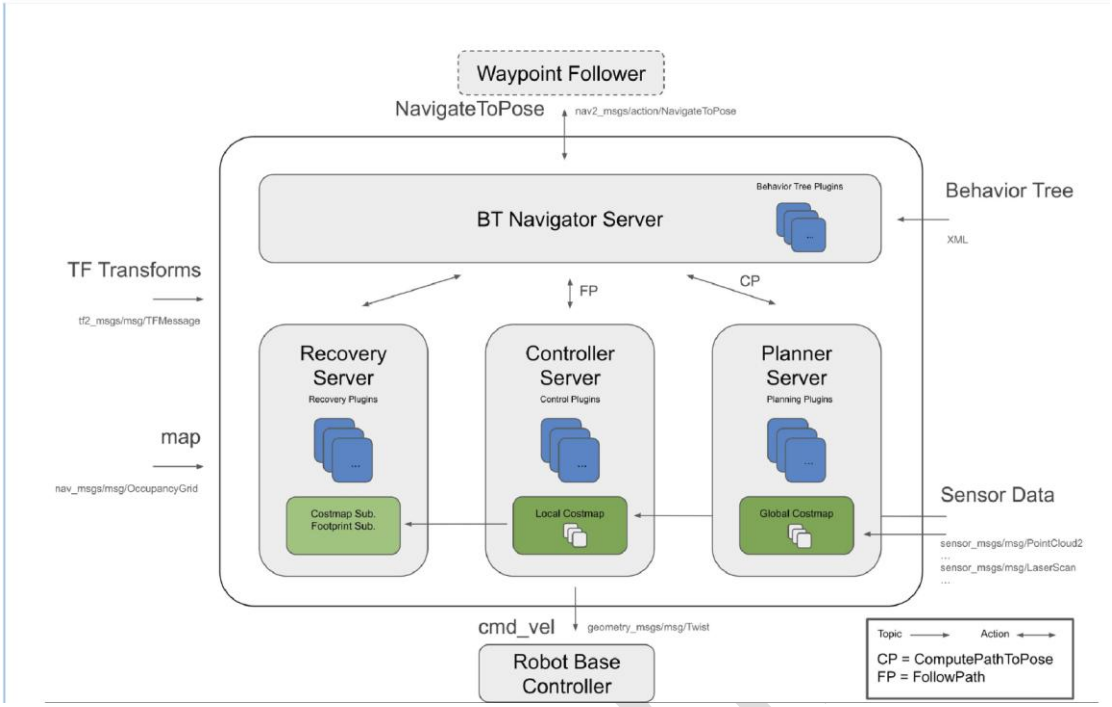
- **RetryUntilSuccessful**：不断尝试直到成功。
- **Timeout**：为子节点设置时间限制。

🔄 节点执行的返回值

每个节点在执行后，返回以下三种状态之一：

状态	含义
`SUCCESS`	动作成功完成
`FAILURE`	动作失败，需尝试其他分支
`RUNNING`	动作正在进行中（比如正在移动）

Navigation 2 导航系统框架：



这里我们可以看到：

输入：TF Transforms ; map ; Behaviour Tree ; Sensor Data

输出: cmd_vel

1. TF Transforms

- **数据类型**：`tf2_msgs/msg/TFMessage`
- **作用**：提供坐标系之间的变换关系，例如 `map → odom → base_link`；
- **用途**：用于规划、避障和机器人的姿态判断；
- **来源**：通常由 `robot_state_publisher` 和 SLAM 系统发布。

2. map（地图）

- **数据类型**：`nav_msgs/msg/OccupancyGrid`
- **作用**：提供静态全局地图（Global Costmap）；
- **来源**：由 `map_server` 或 `slam_toolbox` 提供；
- **用途**：规划路径时使用，标记障碍物和自由空间。

3. Sensor Data（传感器数据）

- **数据类型**：如 `sensor_msgs/msg/LaserScan`、`PointCloud2` 等；
- **来源**：激光雷达、深度相机等；
- **作用**：用于生成代价地图（Costmap）；
- **分支**：
  - **Local Costmap**：用于局部避障；
  - **Global Costmap**：用于全局路径规划。

4. cmd_vel

- **数据类型**：`geometry_msgs/msg/Twist`
- **功能**：机器人底盘控制的速度指令；
- **接收方**：`Robot Base Controller`；
- **来源**：`Controller Server` 产生的控制指令；
- **方向**：从系统内部输出到机器人硬件。

内部结构介绍：

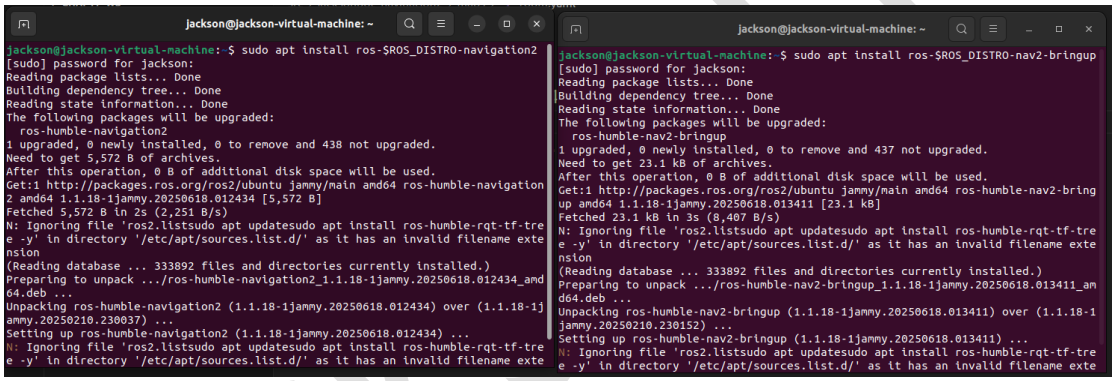
1） BT 导航服务器：该服务器接受 XML 格式的行为树描述文件后调用下面三个服务器完成对机器人的控制

- **核心任务**：是整个 Nav2 的**调度中心**，控制导航行为的执行流程；
- **机制**：使用 **Behavior Tree（行为树）插件**来编排任务顺序；
- **输入**：接受上层任务（如 NavigateToPose）；
- **输出**：
  - 调用 **Planner Server** → 计算路径；
  - 调用 **Controller Server** → 执行跟随；
  - 调用 **Recovery Server** → 异常处理；
- **文件配置**：行为树的定义来自 `.xml` 文件，例如 `navigate_w_replanning_and_recovery.xml`。

右边 Planner Server 规划器服务器：全局路径规划

中间 Controller Server 控制器服务器：根据全局路径，结合实时障碍物和局部代价地图完成对机器人的控制。

左边 Recovery Server 恢复器服务器：加载不同的恢复行为完成机器人的脱困。



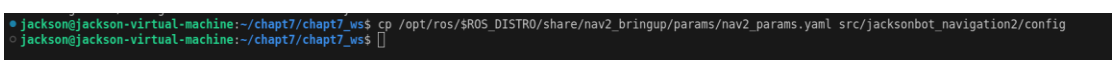
输入安装命令安装 Navigation 2 同时 Navigation 2 还提供了启动示例功能包的 nav2_bringup 也可以进行安装。

7.5 配置 Navigation 2 参数

可以将 Navigation 2 当作一个模块，只要给他输入正确的数据他就可以正常工作，我们要让他适配我们的仿真机器人，就需要进行一些调整。

参数： 话题名称、坐标系名称、机器人描述

我们只需要在 nav2__bringup 上修改即可，在 jacksonbot_navigation2 下创建 config 目录 将 nav2__bringup 默认参数复制到 config 目录下，键入以下命令



可以看到里面有一堆参数 大致可以分为：

带 topic 的基本关于话题的设置　　带 frame 的基本关于坐标系名称设置

除了关注话题坐标系，进行路径规划时，如果机器人半径参数设置过大：过不去，过小：发生碰撞。所以在全局代价地图和局部代价地图的 robot_radius 改成 0.12。

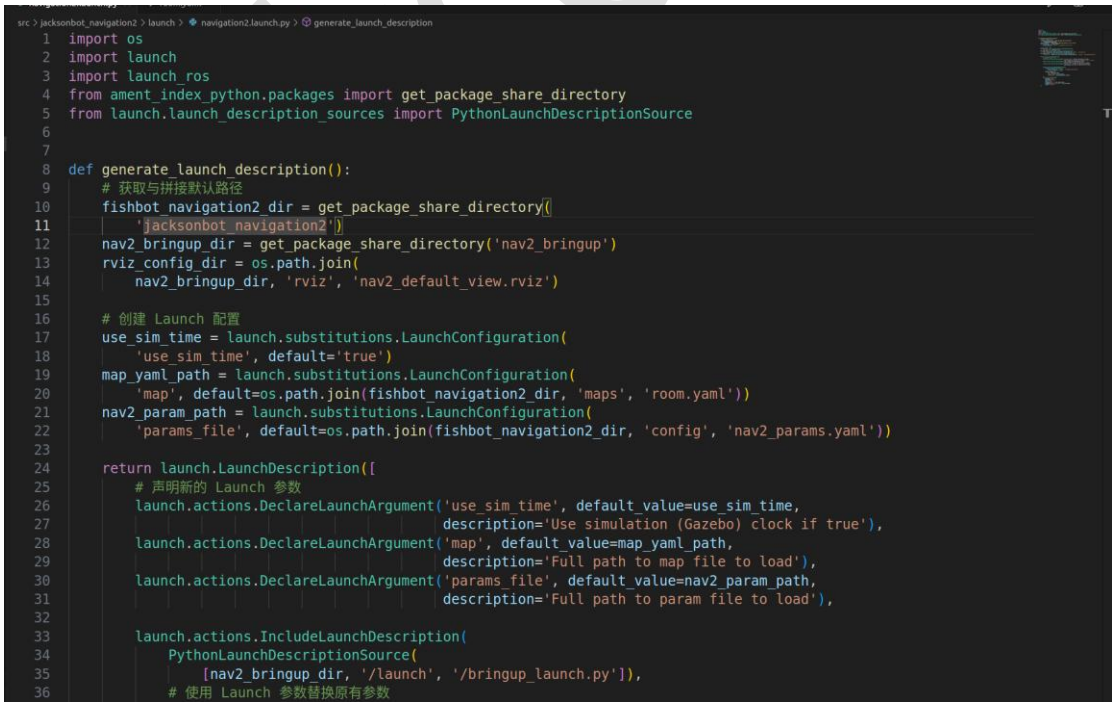
注意这里的参数很有意思，它不可以直接读取之前的 URDF，所以这里半径需要我们自己手动配置，另外两个半径可以不一样，一般为了防止避障的误差行，会比正常机器人的大小写大一丢丢，这里鱼香 ROS 没有写，我也就先这样吧

参数配置就先设置到这里！

7.6 编写 launch 文件并启动导航

在 jacksonbot_navigation2 功能包下新建 navigation2.launch.py

键入以下代码：



```
            launch_arguments={
                'map': map_yaml_path,
                'use_sim_time': use_sim_time,
                'params_file': nav2_param_path}.items(),
        ),
        launch_ros.actions.Node(
            package='rviz2',
            executable='rviz2',
            name='rviz2',
            arguments=['-d', rviz_config_dir],
            parameters=[{'use_sim_time': use_sim_time}],
            output='screen'),
    ])
```

再记得修改下 CMakeList，每一次创建一个新的功能包都要再写进 CMake 里

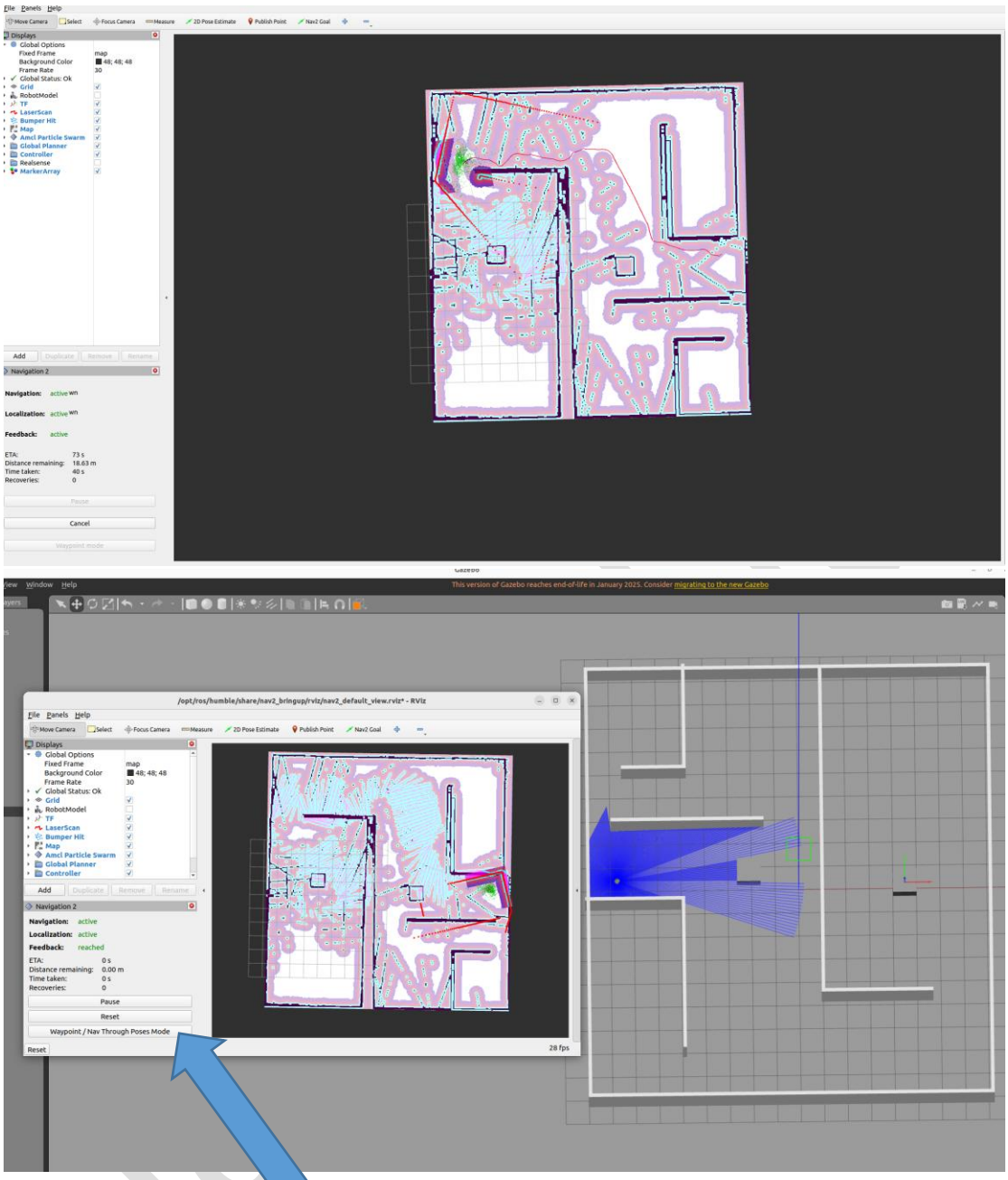
终端启动仿真，你可能看到也可能看不到地图已经显示在上面了，但是不管你看不看的到我们都发现 vsCode 给我们返回错误说：Timeout 一大堆，TF 相关错误。这是因为我们还没有设置机器人初始位置 点击上面的 2D Pose Estimate 选定大致位置朝向即可发现问题解决。

我们可以看到膨胀图层已经显示出来，这个图层的意义在于防止机器人与障碍物贴的太近导致碰撞 他是按照一定半径进行膨胀的

我们在 Global Planner 配置中取消候选全局代价地图即可看到局部代价地图。

7.7 单点与路点导航

在上一个页面点击 Nav2 Goal 随便选择一个位置就可以实现单点导航



接着是指定路点导航即经过特定位置的导航

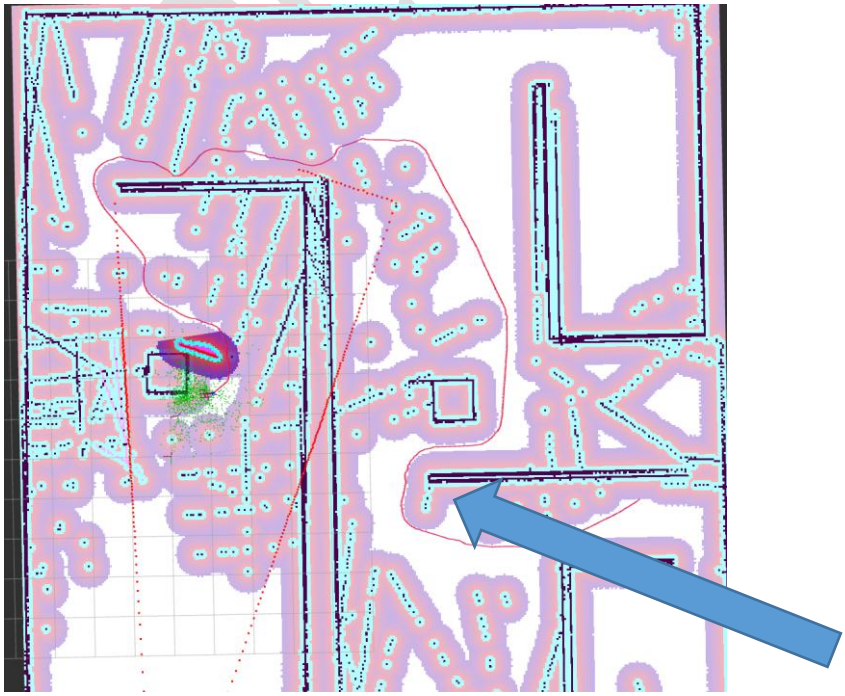
通过下面这个可以实现　　同时用 Nav2 Goal 点击多个位置 （在下图里显示为绿色小点 见红圈） 这些表示路点（需要经过的点）

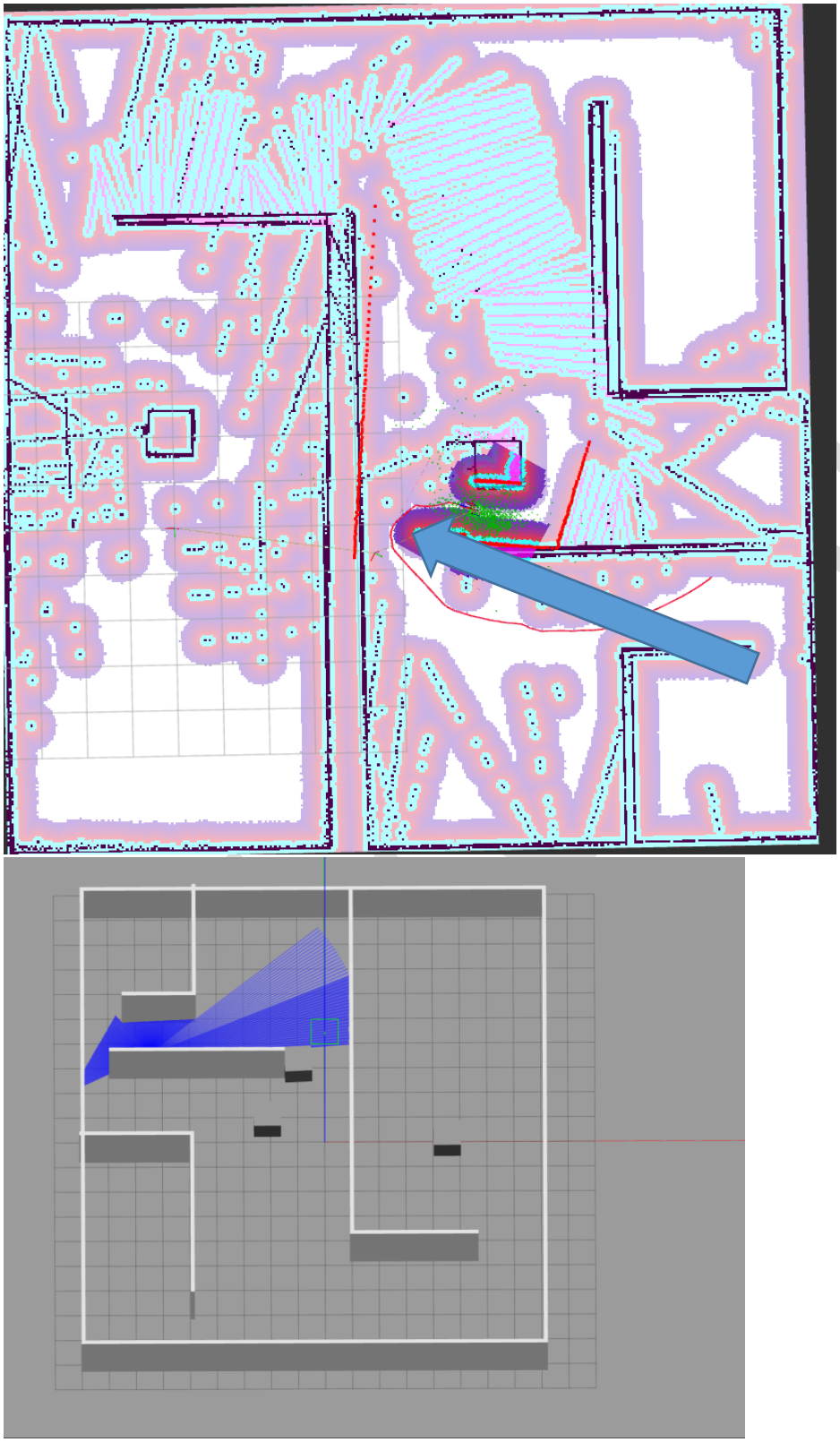
点击最下面的 Start Waypoint Following 即可开始导航

7.8 导航过程中的动态避障

在机器人按照之前规划的全局路径行进过程中，突然我们在中间放上一个正方体，他还会按照原来方式行进吗，还是会通过雷达实时探测实现动态避障？接下来我们可以实验一下：

我们可以重点关注下下方图即是路径规划出来的最短路线（因为误识别障碍物的原因，可以暂时认为它没问题）看右下角那个紧贴墙壁的地方





在我们放完正方体后可以看到红线外凸了，这反映了在机器人行进过程中，雷达会不断检测周围的距离信息，一旦发现新的障碍物就会添加到代价地图中进行重新规划！

7.9 优化导航速度和膨胀半径

设置的雷达扫描频率只有 5HZ ——> 如果机器人旋转过快就容易导致定位不准

——>调整配置文件中的速度参数

分析：Navigation2 框架三个器：控制器、规划器、恢复器

规划完全局路径后由控制器控制机器人运动，所以我们应该去控制器里调整参数，打开 nav2_param.yaml 文件找到 controller_server

```
max_vel_theta: 0.8
min_speed_xy: 0.0
max_speed_xy: 0.26
min_speed_theta: 0.0
# Add high threshold velocity for turtlebot
# https://github.com/ROBOTIS-GIT/turtlebot3
acc_lim_x: 2.5
acc_lim_y: 0.0
acc_lim_theta: 2.0
```

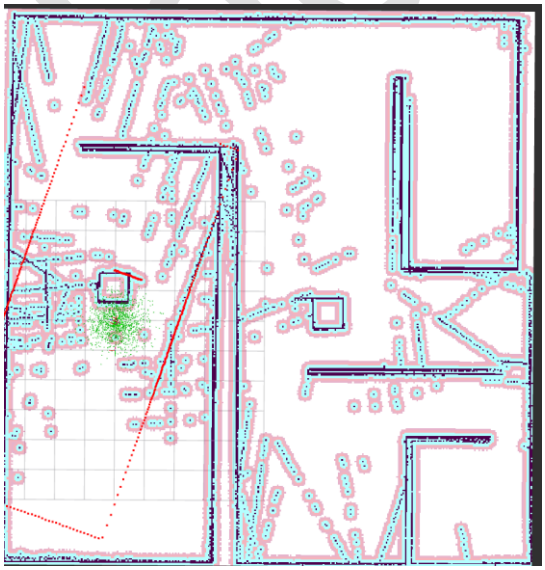
调整两个带 theta 的值

另外如果你建图时和我一样粗糙，那么我们会在操控时发现几乎地图显示中没有空白的地方，但是机器人完全可以通过，这就是膨胀半径不合理，太大了

我们一般把膨胀半径设置成机器人的直径

自己去找吧 有个叫 inflation_radius 的设置成 0.24 （上面我们也看到了机器人的半径是 0.12）

重新构建启动



可以发现膨胀半径小很多了！可以重新试试之前学的，发现最短路径发生变化

7.10 优化机器人到点精度

比如我让机器人到 A 点，但是机器人到 A 点距离 1 米的位置停下了，这就是到点精度不够，在控制器配置中找到 xy_tolerance ；yaw_goal_tolerance; xy_goal_tolerance 改成 0.15 米表示接受误差 0.15 米。

注意：也不可以把范围定的太小，太小会导致机器人在目标点徘徊反而不利于导航！

导航的配置及基本参数调整到此结束，下面进入第四部分导航应用技巧！加油！

四． 机器人导航应用开发指南

回顾我们是怎样启动机器人导航的：

启动 gazebo 得到仿真机器人和仿真地图 -> 启动 Navigation2 Rviz 加载灰度地图 -> 在地图中使用 2D Position 手动标注机器人位置

现在我们要创建一个自动巡检机器人，穿梭在不同位置中并在一定位置处拍照，此时还要专门雇人在用 Rviz 确认位置，指定目标地点并拍照太麻烦了！

在实际机器人项目中，导航只是一个模块，我们通过接口进行调用即可。本章我们来讲解机器人导航常用方法。

7.11 使用话题初始化机器人位姿

“使用 2D Position 手动标注机器人位置” -> amcl 节点根据地图和传感器数据实时计算

所以我们要给 amcl 节点发布话题让他订阅，告诉它机器人大致位置。

启动仿真导航后，键入以下代码可以看到以下信息：

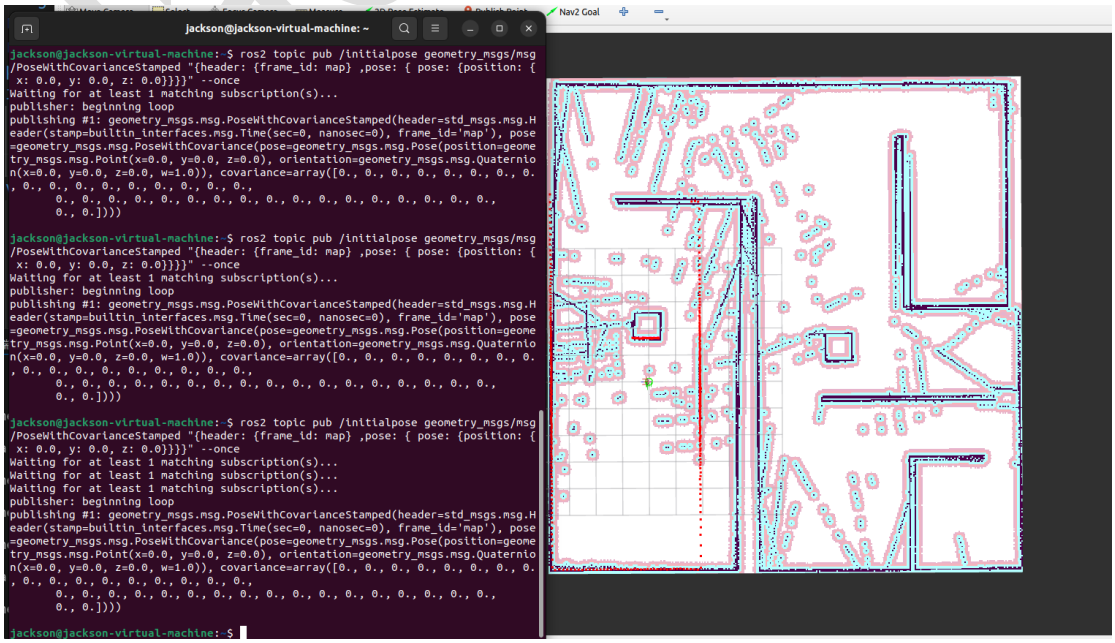
```
jackson@jackson-virtual-machine: ~
jackson@jackson-virtual-machine:~$ ros2 node info /amcl
/amcl
  Subscribers:
    /bond: bond/msg/Status
    /clock: rosgraph_msgs/msg/Clock
    /initialpose: geometry_msgs/msg/PoseWithCovarianceStamped
    /map: nav_msgs/msg/OccupancyGrid
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /scan: sensor_msgs/msg/LaserScan
  Publishers:
    /amcl/transition_event: lifecycle_msgs/msg/TransitionEvent
    /amcl_pose: geometry_msgs/msg/PoseWithCovarianceStamped
    /bond: bond/msg/Status
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /particle_cloud: nav2_msgs/msg/ParticleCloud
    /rosout: rcl_interfaces/msg/Log
    /tf: tf2_msgs/msg/TFMessage
  Service Servers:
    /amcl/change_state: lifecycle_msgs/srv/ChangeState
    /amcl/describe_parameters: rcl_interfaces/srv/DescribeParameters
    /amcl/get_available_states: lifecycle_msgs/srv/GetAvailableStates
    /amcl/get_available_transitions: lifecycle_msgs/srv/GetAvailableTransitions
    /amcl/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
    /amcl/get_parameters: rcl_interfaces/srv/GetParameters
    /amcl/get_state: lifecycle_msgs/srv/GetState
    /amcl/get_transition_graph: lifecycle_msgs/srv/GetAvailableTransitions
    /amcl/list_parameters: rcl_interfaces/srv/ListParameters
    /amcl/set_parameters: rcl_interfaces/srv/SetParameters
    /amcl/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
    /reinitialize_global_localization: std_srvs/srv/Empty
    /request_nomotion_update: std_srvs/srv/Empty
    /set_initial_pose: nav2_msgs/srv/SetInitialPose
  Service Clients:

  Action Servers:

  Action Clients:
```

该节点订阅了两个话题：雷达和地图 并且订阅了 initialpose 话题用于接收初始化位姿

使用命令行发布初始位姿



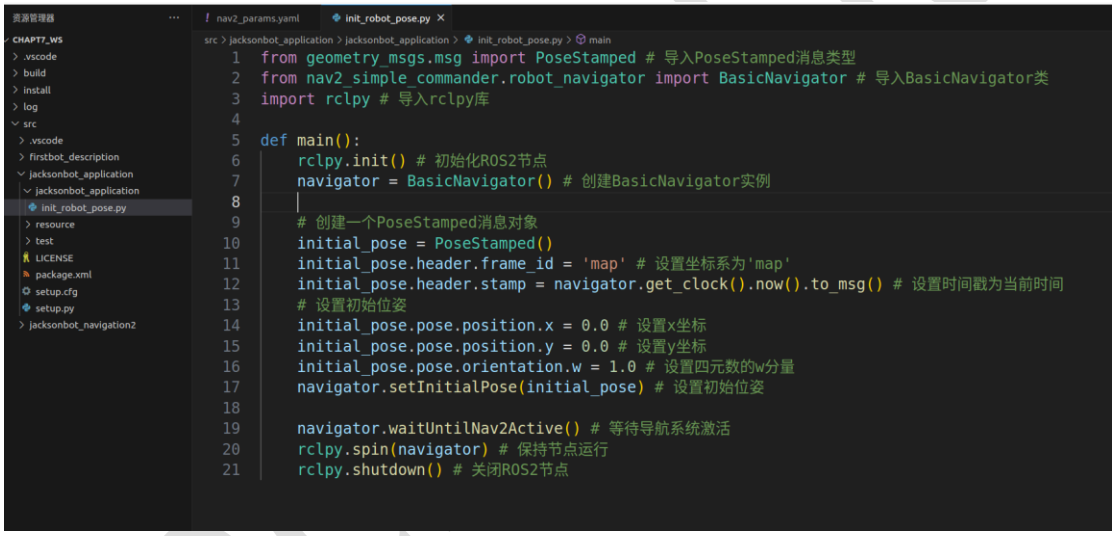
这里我们通过话题告知 amcl 机器人当前位置在地图坐标系原点附近，看 Rviz 发现机器人位姿已经出现在地图上了，

（这里我的机器人在 gazebo 的位置不在原点，也可以设置成自己的 我是为了复刻鱼香 ROS 的）

注：这里如果运行一遍终端代码不行，就多运行几次，我是运行 3 次出来的

我们也可以使用 Python 或 C++来初始化， Navigation2 提供了 nav2_simple_commander 的 Python 包，我们可以直接调用。

在 src 下命令行创建功能包：jacksonbot_application 新建 Python 文件键入以下代码：



导入的 BasicNavigator 表示节点类 用于导航操作

实例化消息接口 initial_pose 方便传入初始数据

在 setup 里注册（python 是 setup, c/c++是 CMake）

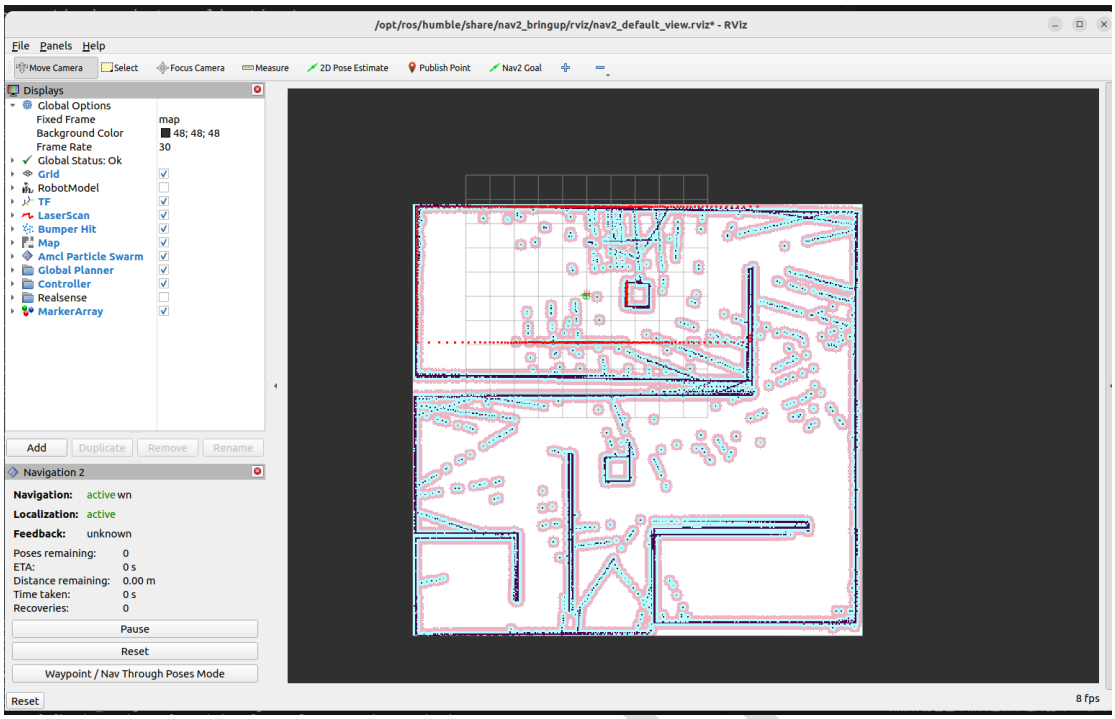
注意！！！我这里上面图片有个错误，在我创建 python 功能包时误删除了__init__.py 导致找不到可执行功能包，然后如果你也误删了记得不是_而是__两个下划线！然后每次改动功能包和可执行文件这些都需要两件套！

另外这里运行的是 python 不是 launch 文件，所以运行命令

Ros2 run jacksonbot_application init_robot_pose

注意这里没有 init_robot_pose.py !!!!

运行后可看到以下效果：



7.12 使用 TF 获取机器人实时位置

当 amcl 正常运行后，首先计算机器人在地图中位置，结合里程计发布 map 到 base 之间的 TF 变化。那么如果想获取机器人实时位置我们只需要使用 TF 监听即可。

在上一节课同目录下创建 get_robot_pose.py 键入以下代码：

这个代码和 5.2.3 代码及其相似 目的是为了用于监听 map 到 base_footprint 的坐标变化并输出。 Setup 注册重构功能包，再次运行仿真导航，初始化位姿后启动该节点，新建终端看输出。

```python
import rclpy
from rclpy.node import Node
from tf2_ros import TransformListener, Buffer # 坐标变换监听器
from transforms3d.euler import quat2euler  # 四元数转欧拉角

# 定义一个ROS2节点，用于监听和查询坐标变换
class TFListener(Node):
    def __init__(self):
        # 初始化节点，设置节点名称为'tf2_listener'
        super().__init__('tf2_listener')
        self.buffer = Buffer() # 创建坐标变换缓冲区
        # 创建一个坐标变换监听器，监听缓冲区中的坐标变换
        self.listener = TransformListener(self.buffer, self)
        # 创建一个定时器，每隔1秒调用一次`get_transform`方法
        self.timer = self.create_timer(1, self.get_transform)

    def get_transform(self):

        '''
        实时查询坐标关系 buffer
        '''
        try:
            # 从缓冲区中查询两个坐标系之间的变换关系
            # 'map' 是源坐标系，'base_footprint' 是目标坐标系
            # rclpy.time.Time(seconds=0.0) 表示查询最新的变换
            # rclpy.time.Duration(seconds=1.0) 表示查询超时时间为1秒
            result = self.buffer.lookup_transform('map', 'base_footprint',
                    rclpy.time.Time(seconds=0.0), rclpy.time.Duration(seconds=1.0))
            transform = result.transform # 获取变换关系中的坐标变换部分

            # 将四元数转换为欧拉角（RPY）
            rotation_euler = quat2euler([
                transform.rotation.x, transform.rotation.y,
                transform.rotation.z, transform.rotation.w])

            self.get_logger().info(
                f'平移: {transform.translation.x},旋转四元素: {transform.rotation}, 旋转欧拉角:{rotation_euler}')
        except Exception as e:
            self.get_logger().error(f'获取坐标变换失败: 原因{str(e)}')

def main():
    rclpy.init()  # 初始化ROS2
    node = TFListener()  # 创建静态坐标广播器节点
    rclpy.spin(node)  # 保持节点运行
    rclpy.shutdown()  # 关闭ROS2
```

```
[INFO] [1752025315.415362539] [tf2_listener]: 平移: 0.0,旋转四元素: geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=0.0, w=1.0), 旋转欧拉角:(0.0, -0.0, 3.141592653589793)
[INFO] [1752025316.416398558] [tf2_listener]: 平移: 0.0,旋转四元素: geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=0.0, w=1.0), 旋转欧拉角:(0.0, -0.0, 3.141592653589793)
[INFO] [1752025317.415410808] [tf2_listener]: 平移: 0.0,旋转四元素: geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=0.0, w=1.0), 旋转欧拉角:(0.0, -0.0, 3.141592653589793)
[INFO] [1752025318.415306397] [tf2_listener]: 平移: 0.0,旋转四元素: geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=0.0, w=1.0), 旋转欧拉角:(0.0, -0.0, 3.141592653589793)
[INFO] [1752025319.415145494] [tf2_listener]: 平移: 0.0,旋转四元素: geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=0.0, w=1.0), 旋转欧拉角:(0.0, -0.0, 3.141592653589793)
[INFO] [1752025320.415120973] [tf2_listener]: 平移: 0.0,旋转四元素: geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=0.0, w=1.0), 旋转欧拉角:(0.0, -0.0, 3.141592653589793)
[INFO] [1752025321.415289072] [tf2_listener]: 平移: 0.0,旋转四元素: geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=0.0, w=1.0), 旋转欧拉角:(0.0, -0.0, 3.141592653589793)
[INFO] [1752025322.414975865] [tf2_listener]: 平移: 0.0,旋转四元素: geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=0.0, w=1.0), 旋转欧拉角:(0.0, -0.0, 3.141592653589793)
[INFO] [1752025323.415499429] [tf2_listener]: 平移: 0.0,旋转四元素: geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=0.0, w=1.0), 旋转欧拉角:(0.0, -0.0, 3.141592653589793)
[INFO] [1752025324.415080421] [tf2_listener]: 平移: 0.0,旋转四元素: geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=0.0, w=1.0), 旋转欧拉角:(0.0, -0.0, 3.141592653589793)
[INFO] [1752025325.414921160] [tf2_listener]: 平移: 0.0,旋转四元素: geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=0.0, w=1.0), 旋转欧拉角:(0.0, -0.0, 3.141592653589793)
[INFO] [1752025326.415447956] [tf2_listener]: 平移: 0.0,旋转四元素: geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=0.0, w=1.0), 旋转欧拉角:(0.0, -0.0, 3.141592653589793)
[INFO] [1752025327.415270000] [tf2_listener]: 平移: 0.0,旋转四元素: geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=0.0, w=1.0), 旋转欧拉角:(0.0, -0.0, 3.141592653589793)
[INFO] [1752025328.415363336] [tf2_listener]: 平移: 0.0,旋转四元素: geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=0.0, w=1.0), 旋转欧拉角:(0.0, -0.0, 3.141592653589793)
[INFO] [1752025329.414942953] [tf2_listener]: 平移: 0.0,旋转四元素: geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=0.0, w=1.0), 旋转欧拉角:(0.0, -0.0, 3.141592653589793)
[INFO] [1752025330.414638090] [tf2_listener]: 平移: 0.0,旋转四元素: geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=0.0, w=1.0), 旋转欧拉角:(0.0, -0.0, 3.141592653589793)
[INFO] [1752025331.416154588] [tf2_listener]: 平移: 0.0,旋转四元素: geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=0.0, w=1.0), 旋转欧拉角:(0.0, -0.0, 3.141592653589793)
[INFO] [1752025332.415492218] [tf2_listener]: 平移: 0.0,旋转四元素: geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=0.0, w=1.0), 旋转欧拉角:(0.0, -0.0, 3.141592653589793)
[INFO] [1752025333.415161202] [tf2_listener]: 平移: 0.0,旋转四元素: geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=0.0, w=1.0), 旋转欧拉角:(0.0, -0.0, 3.141592653589793)
[INFO] [1752025334.415361031] [tf2_listener]: 平移: 0.0,旋转四元素: geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=0.0, w=1.0), 旋转欧拉角:(0.0, -0.0, 3.141592653589793)
[INFO] [1752025335.415209138] [tf2_listener]: 平移: 0.0,旋转四元素: geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=0.0, w=1.0), 旋转欧拉角:(0.0, -0.0, 3.141592653589793)
[INFO] [1752025336.415283009] [tf2_listener]: 平移: 0.0,旋转四元素: geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=0.0, w=1.0), 旋转欧拉角:(0.0, -0.0, 3.141592653589793)
[INFO] [1752025337.415386868] [tf2_listener]: 平移: 0.0,旋转四元素: geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=0.0, w=1.0), 旋转欧拉角:(0.0, -0.0, 3.141592653589793)
[INFO] [1752025338.416129586] [tf2_listener]: 平移: 0.0006247034155856677,旋转四元素: geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=-0.3107671125398227 5, w=0.9504860870963137), 旋转欧拉角:(-0.6319999999999888, -0.0, 3.141592653589793)
[INFO] [1752025339.415596305] [tf2_listener]: 平移: 0.024537317317383 13,旋转四元素: geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=-0.6555884221730636, w=0.7551184150268307), 旋转欧拉角:(-1.4299232040781744, -0.0, 3.141592653589793)
[INFO] [1752025340.415331184] [tf2_listener]: 平移: -0.019979756159839927,旋转四元素: geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=0.8791062472145235, w=-0.4766258554761766), 旋转欧拉角:(-2.1479675578828057, 0.0, 3.141592653589793)
[INFO] [1752025341.415507480] [tf2_listener]: 平移: -0.011196192242441 13,旋转四元素: geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=0.9097890671707368, w=-0.4150709014814222), 旋转欧拉角:(-2.285551186853376, 0.0, 3.141592653589793)
[INFO] [1752025342.415444635] [tf2_listener]: 平移: -0.005900893595291249,旋转四元素: geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=-0.7228560732033721, w=0.6909986233220737), 旋转欧拉角:(-1.6158533757167548, -0.0, 3.141592653589793)
[INFO] [1752025343.415195423] [tf2_listener]: 平移: 0.022300220299026015,旋转四元素: geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=-0.30540882457256024, w=0.9522213239962688), 旋转欧拉角:(-0.6207354749948039, -0.0, 3.141592653589793)
[INFO] [1752025344.415887150] [tf2_listener]: 平移: 0.057608194522835 69,旋转四元素: geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=-0.4927088465123705, w=0.8701942269220414), 旋转欧拉角:(-1.0303998822875637, -0.0, 3.141592653589793)
[INFO] [1752025345.415668916] [tf2_listener]: 平移: 0.084035315636852 11,旋转四元素: geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=-0.7883642352686802, w=0.6152087715151898), 旋转欧拉角:(-1.8162911431619606, -0.0, 3.141592653589793)
[INFO] [1752025346.415480117] [tf2_listener]: 平移: -0.063680007110498 1,旋转四元素: geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=0.9485208634647391, w=-0.3167146532323788), 旋转欧拉角:(-2.497065008560233 5, 0.0, 3.141592653589793)
```

结果可见：机器人的实时位姿已经被输出了。

7.13 调用接口进行单点导航

Navigation2 对外提供了用于导航调用的动作服务——动作通信

动作通信主要用于控制场景，优点：反馈机制

当客户段发送目标给服务端后，除了傻等着，他还能收到来自服务端的处理进度反馈。启动导航后在终端使用动作相关命令，我们能看到：

```
jackson@jackson-virtual-machine:~$ ros2 action list
/assisted_teleop
/backup
/compute_path_through_poses
/compute_path_to_pose
/drive_on_heading
/follow_path
/follow_waypoints
/navigate_through_poses
/navigate_to_pose
/smooth_path
/spin
/wait
```

其中 navigate_to_pose 就是用于处理导航到点请求的动作

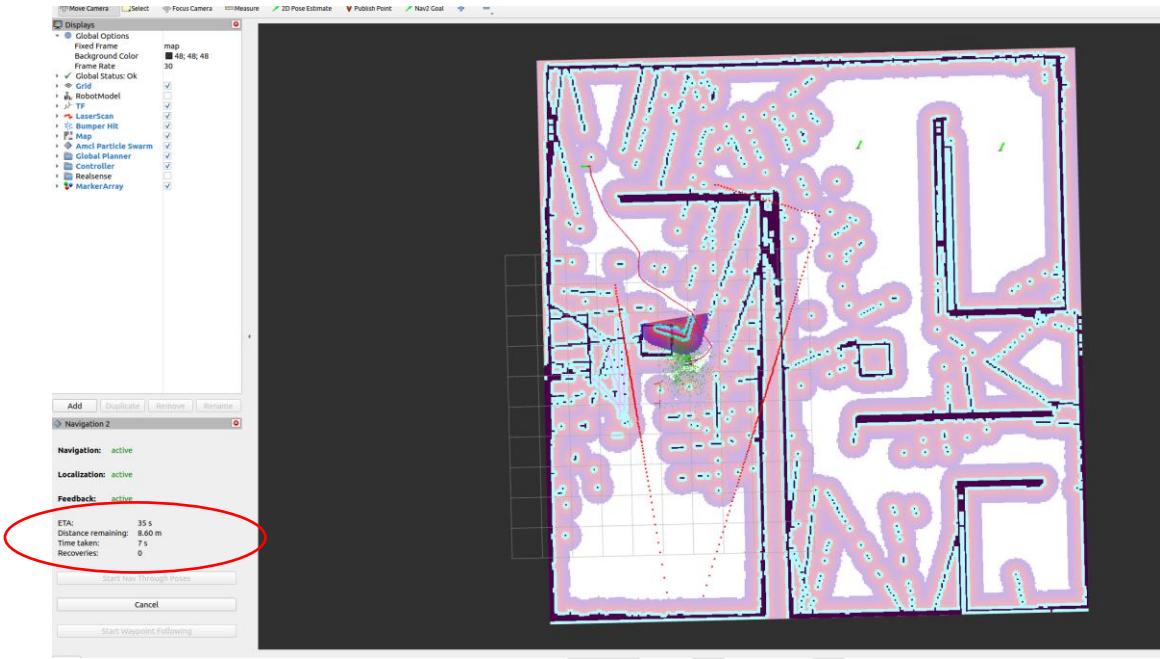
```
jackson@jackson-virtual-machine:~$ ros2 action info /navigate_to_pose -t
Action: /navigate_to_pose
Action clients: 4
    /bt_navigator [nav2_msgs/action/NavigateToPose]
    /waypoint_follower [nav2_msgs/action/NavigateToPose]
    /rviz2 [nav2_msgs/action/NavigateToPose]
    /rviz_navigation_dialog_action_client [nav2_msgs/action/NavigateToPose]
Action servers: 1
    /bt_navigator [nav2_msgs/action/NavigateToPose]
```

查看接口情况

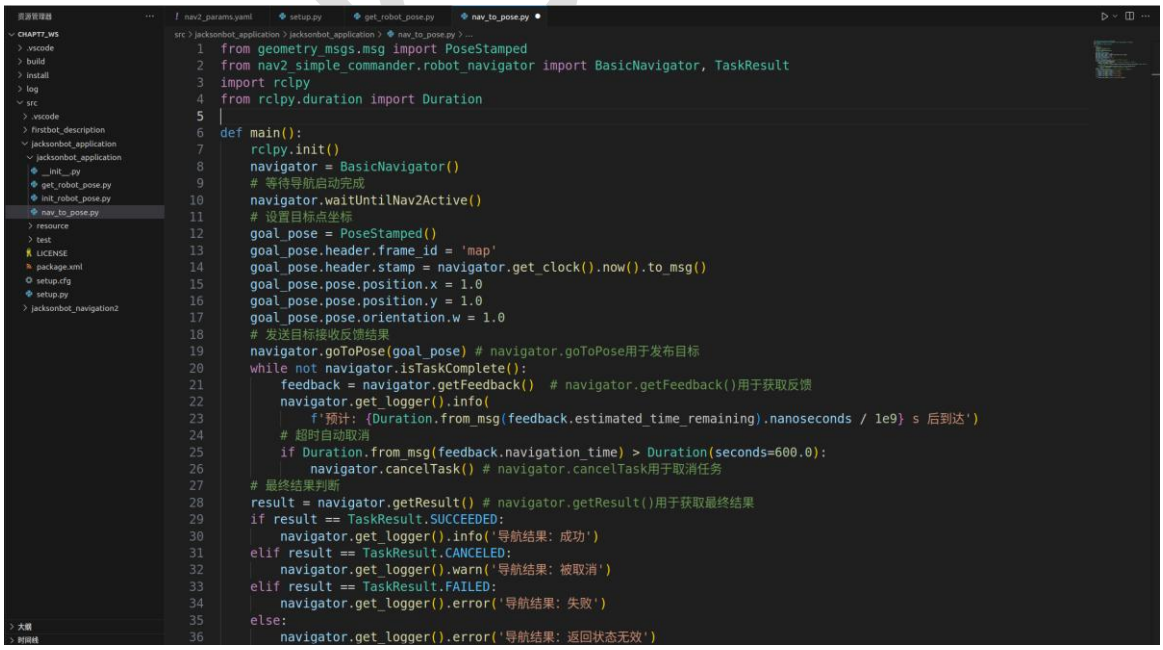
```
jackson@jackson-virtual-machine:~$ ros2 interface show nav2_msgs/action/Navigate
ToPose
#goal definition
geometry_msgs/PoseStamped pose
        std_msgs/Header header
                builtin_interfaces/Time stamp
                        int32 sec
                        uint32 nanosec
                string frame_id
        Pose pose
                Point position
                        float64 x
                        float64 y
                        float64 z
                Quaternion orientation
                        float64 x 0
                        float64 y 0
                        float64 z 0
                        float64 w 1
string behavior_tree
---
#result definition
std_msgs/Empty result
---
#feedback definition
geometry_msgs/PoseStamped current_pose
        std_msgs/Header header
                builtin_interfaces/Time stamp
                        int32 sec
                        uint32 nanosec
                string frame_id
        Pose pose
                Point position
                        float64 x
                        float64 y
                        float64 z
                Quaternion orientation
                        float64 x 0
                        float64 y 0
                        float64 z 0
                        float64 w 1
builtin_interfaces/Duration navigation_time
        int32 sec
        uint32 nanosec
builtin_interfaces/Duration estimated_time_remaining
        int32 sec
        uint32 nanosec
int16 number_of_recoveries
float32 distance_remaining
```

可以看到动作消息接口分为：goal / result / feedback 三个部分

（翻翻之前的图片）我们可以看到一些实时反馈的数据（下图红圈内）



同上一节一样我们可以使用命令行来控制机器人到达目标点，但是同样我们也可以编写 python 文件来实现这一目标，使用 nav2_simple_commander 库就能实现。在 application 目录下新建 nav_to_pose.py 键入以下代码：

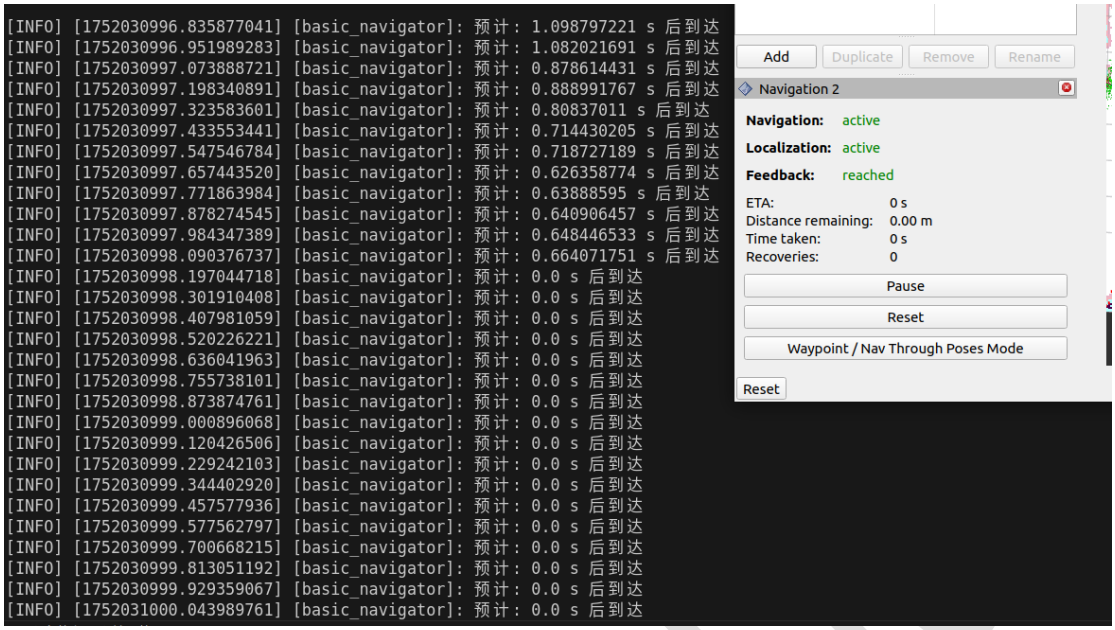


```python
from geometry_msgs.msg import PoseStamped
from nav2_simple_commander.robot_navigator import BasicNavigator, TaskResult
import rclpy
from rclpy.duration import Duration

def main():
    rclpy.init()
    navigator = BasicNavigator()
    # 等待导航启动完成
    navigator.waitUntilNav2Active()
    # 设置目标点坐标
    goal_pose = PoseStamped()
    goal_pose.header.frame_id = 'map'
    goal_pose.header.stamp = navigator.get_clock().now().to_msg()
    goal_pose.pose.position.x = 1.0
    goal_pose.pose.position.y = 1.0
    goal_pose.pose.orientation.w = 1.0
    # 发送目标接收反馈结果
    navigator.goToPose(goal_pose) # navigator.goToPose用于发布目标
    while not navigator.isTaskComplete():
        feedback = navigator.getFeedback()  # navigator.getFeedback()用于获取反馈
        navigator.get_logger().info(
            f'预计: {Duration.from_msg(feedback.estimated_time_remaining).nanoseconds / 1e9} s 后到达')
        # 超时自动取消
        if Duration.from_msg(feedback.navigation_time) > Duration(seconds=600.0):
            navigator.cancelTask() # navigator.cancelTask用于取消任务
    # 最终结果判断
    result = navigator.getResult() # navigator.getResult()用于获取最终结果
    if result == TaskResult.SUCCEEDED:
        navigator.get_logger().info('导航结果: 成功')
    elif result == TaskResult.CANCELED:
        navigator.get_logger().warn('导航结果: 被取消')
    elif result == TaskResult.FAILED:
        navigator.get_logger().error('导航结果: 失败')
    else:
        navigator.get_logger().error('导航结果: 返回状态无效')
```

重新构建注册运行仿真导航启动节点

得到下面这幅图：



图中显示预计到达时间在逐渐减少，最后显示为导航成功！

至此我们完成了单个目标点导航代码调用的学习，接下来我们进入本章节最后一课，多个目标点导航代码调用的学习！

7.14 调用接口进行路点导航

路点导航和单点都差不多，服务名是 follow_waypoints

（很奇怪，我这里跑出来就是 1 个 client，鱼香 ROS 那里是两个一样的）

```
jackson@jackson-virtual-machine:~$ ros2 action info /follow_waypoints -t
Action: /follow_waypoints
Action clients: 1
    /rviz_navigation_dialog_action_client [nav2_msgs/action/FollowWaypoints]
Action servers: 1
    /waypoint_follower [nav2_msgs/action/FollowWaypoints]
```

接着我们可以看到消息接口信息：

```
jackson@jackson-virtual-machine:~$ ros2 interface show nav2_msgs/action/FollowWaypoints
#goal definition
geometry_msgs/PoseStamped[] poses
        std_msgs/Header header
                builtin_interfaces/Time stamp
                        int32 sec
                        uint32 nanosec
                string frame_id
        Pose pose
                Point position
                        float64 x
                        float64 y
                        float64 z
                Quaternion orientation
                        float64 x 0
                        float64 y 0
                        float64 z 0
                        float64 w 1
---
#result definition
int32[] missed_waypoints
---
#feedback definition
uint32 current_waypoint
```

目标：目标点数组　　结果：没有导航到的点编号　　反馈：当前目标路点编号

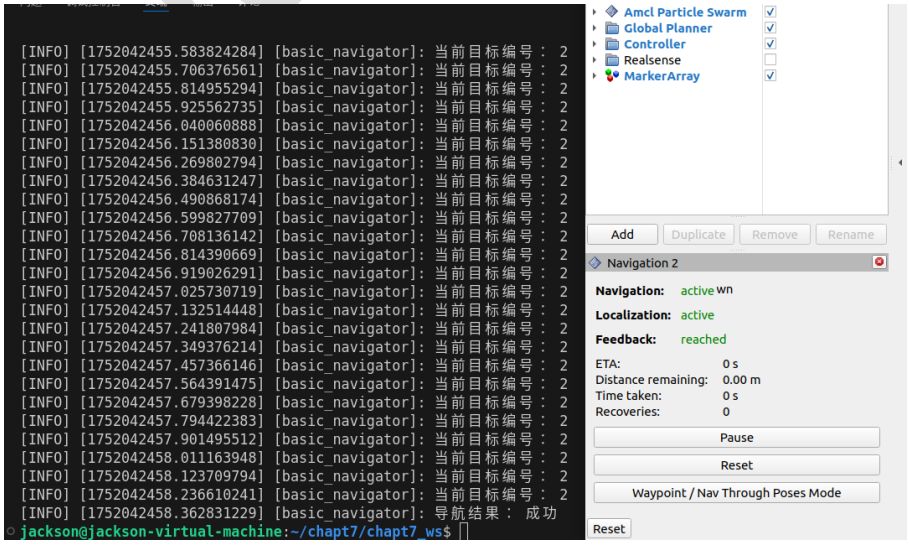
Application 目录下新建 waypoint_follower.py 键入以下代码：

```python
from geometry_msgs.msg import PoseStamped
from nav2_simple_commander.robot_navigator import BasicNavigator, TaskResult
import rclpy
from rclpy.duration import Duration

def main():
    rclpy.init()
    navigator = BasicNavigator()
    navigator.waitUntilNav2Active()
    # 创建点集
    # 这里创建了三个目标点，分别位于 (0,0), (2,0), (2,2)
    goal_poses = []
    goal_pose1 = PoseStamped()
    goal_pose1.header.frame_id = 'map'
    goal_pose1.header.stamp = navigator.get_clock().now().to_msg()
    goal_pose1.pose.position.x = 0.0
    goal_pose1.pose.position.y = 0.0
    goal_pose1.pose.orientation.w = 1.0
    goal_poses.append(goal_pose1)
    goal_pose2 = PoseStamped()
    goal_pose2.header.frame_id = 'map'
    goal_pose2.header.stamp = navigator.get_clock().now().to_msg()
    goal_pose2.pose.position.x = 2.0
    goal_pose2.pose.position.y = 0.0
    goal_pose2.pose.orientation.w = 1.0
    goal_poses.append(goal_pose2)
    goal_pose3 = PoseStamped()
    goal_pose3.header.frame_id = 'map'
    goal_pose3.header.stamp = navigator.get_clock().now().to_msg()
    goal_pose3.pose.position.x = 2.0
    goal_pose3.pose.position.y = 2.0
    goal_pose3.pose.orientation.w = 1.0
    goal_poses.append(goal_pose3)
    # 调用路点导航服务
    navigator.followWaypoints(goal_poses)
    # 判断结束及获取反馈
    while not navigator.isTaskComplete():
        feedback = navigator.getFeedback()
        navigator.get_logger().info(
            f'当前目标编号: {feedback.current_waypoint}')
    # 最终结果判断
    result = navigator.getResult()
    if result == TaskResult.SUCCEEDED:
        navigator.get_logger().info('导航结果: 成功')
    elif result == TaskResult.CANCELED:
        navigator.get_logger().warn('导航结果: 被取消')
    elif result == TaskResult.FAILED:
        navigator.get_logger().error('导航结果: 失败')
    else:
        navigator.get_logger().error('导航结果: 返回状态无效')
```

和单点一样的操作，重新构建注册运行仿真导航启动节点

得到下面这幅图：



可以看到已经成功实现了，接下来我们将基于本章所有学习内容来做一个机器人导航的实践项目——自动巡检机器人！

后续内容可以见下一个 WORD 文件，同时 Jackson 会把它上传至 Github