Nav2 学习笔记

安装与配置

示例包

使用

```
sudo apt install ros-jazzy-navigation2
sudo apt install ros-jazzy-nav2-bringup
```

可安装jazzy版本（可安装其他版本）下的nav2的示例包，使用

```
sudo apt install ros-jazzy-turtlebot3*
```

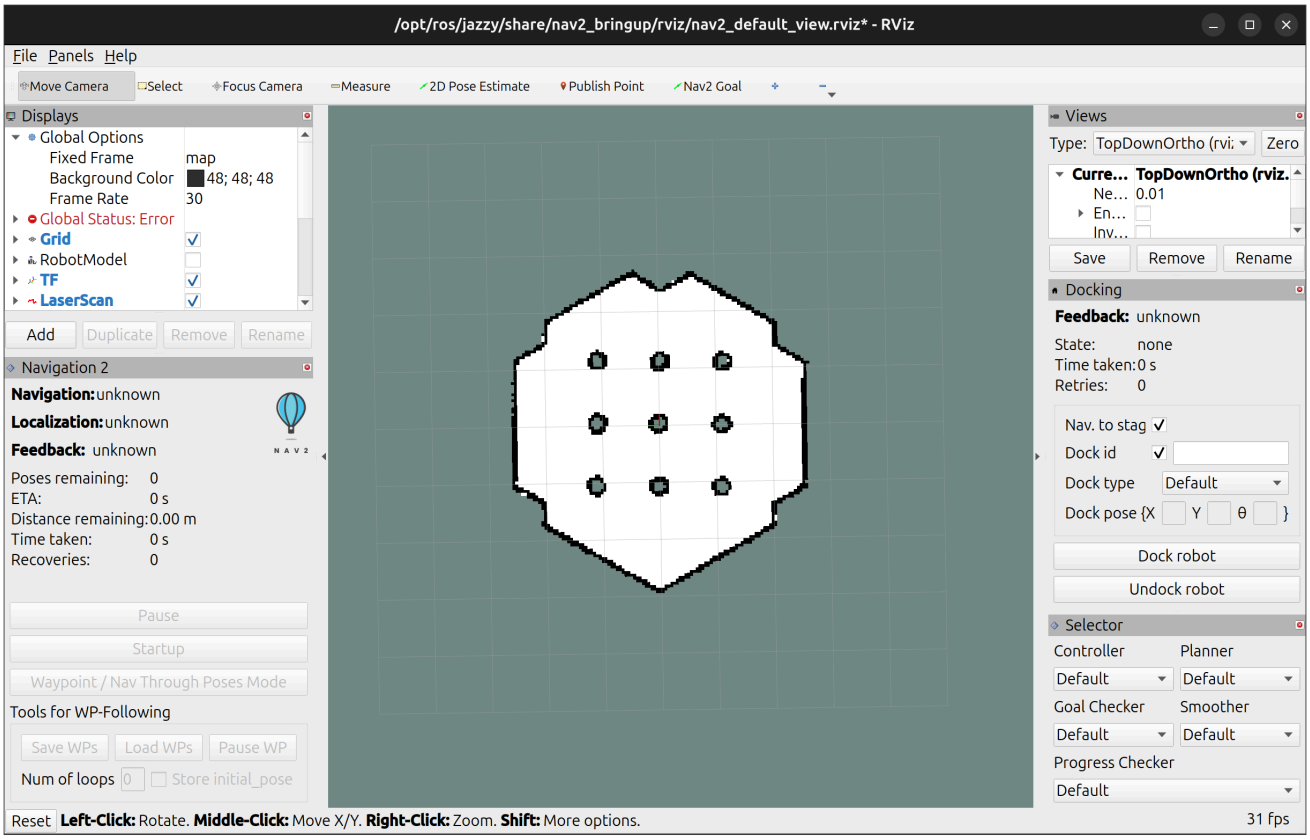
可安装**Turtlebot 3**，然后用

```
source /opt/ros/jazzy/setup.bash
export TURTLEBOT3_MODEL=waffle
export GAZEBO_MODEL_PATH=$GAZEBO_MODEL_PATH:/opt/ros/jazzy/share/turtlebot3_gazebo/models
```

配置好环境变量（第一行如果配置ros时用过就不需要再用了），然后用

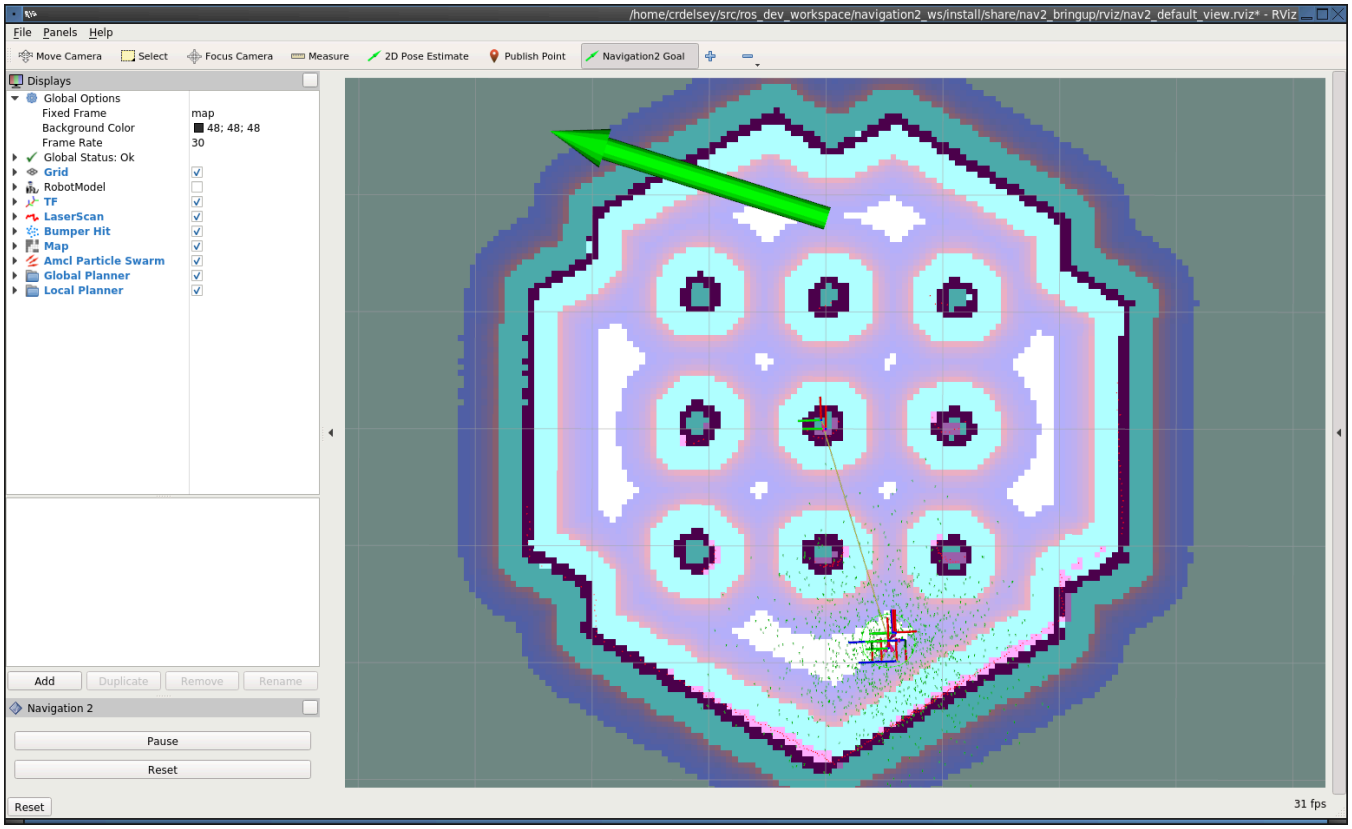
```
ros2 launch nav2_bringup tb3_simulation_launch.py
```

就能启动运行了
启动后应该能看到这样的画面



一般来说选择左下角的`Startup`就能启动显示这个画面

然后依次选择上方的`2D Pose Estimate`进行放置指定位置的机器人并扫描整个平面，然后点击`Nav2 Goal`（也有`Navigation2 Goal`的版本）指定接下来机器人移动的目标和最终朝向



途中的一团复杂的彩色线段堆砌的位置就是机器人所在处，巨大的绿色箭头的起点是机器人移动的目标点，方向是机器人最终朝向，松手后机器人就会自动寻路移动

安装与编译

安装

在**Ubuntu20.04**以上的版本，使用

```
sudo apt install ros-jazzy-navigation2 ros-jazzy-nav2-bringup ros-jazzy-turtlebot3*
```

来安装nav2

编译

安装好后需要进行编译
先创建一个名为`nav2_ws`的工作空间，进到其下的`src`文件夹内，随后将nav2分支功能包克隆到这个目录下：（在humble或foxy版本中需要将`main`改为对应的`<ros2-distro>-devel`）

```
git clone https://github.com/ros-planning/navigation2.git --branch main
```

然后退回到`nav2_ws`目录下，用rosdep获取所有依赖项

```
rosdep install -y -r -q --from-paths src --ignore-src --rosdistro jazzy
```

然后使用

```
colcon build --symlink-install
```

进行软链接编译
（注意，直接进行编译可能会导致并行编译强制跑满所有内存和cpu直接卡死重启，为了避免这种情况，可以考虑设置交换内存为16~32G，同时进行限制性编译（即限制编译时只能一次编译一个项目等）例如：

```
MAKEFLAGS="-j1" colcon build --symlink-install --executor sequential --parallel-workers 1
```

这意味着一次进行一个项目编译，绝不多占用，这样不会导致卡死，但是仍有一个问题就是可能会出现编译器无法识别到`.hpp`文件的情况（明明存在且找到），这时候最好直接安装二进制版，不再折腾此编译，这里我们以二进制版本继续）
输入

```
rm -rf ~/nav2_ws/src/navigation2
```

可以删除之前编译的源代码，使用

```
sudo apt install ros-jazzy-navigation2 ros-jazzy-nav2-bringup ros-jazzy-turtlebot3-gazebo
```

安装二进制版
同时再使用

```
sudo apt install ros-jazzy-nav2-loopback-sim ros-jazzy-nav2-bringup ros-jazzy-turtlebot3-g
```

安装`nav2_loopback_sim`，因为jazzy将nav2拆分的更细，光靠上面的无法安装全部
（如果是在折腾全部编译中途传过来二进制版，则很可能遇到环境污染，此时需要删除编译产物

```
cd ~/nav2_ws
rm -rf build/ install/ log/
```

记得设置环境变来嗯

```
export TURTLEBOT3_MODEL=burger
export GAZEBO_MODEL_PATH=$GAZEBO_MODEL_PATH:/opt/ros/jazzy/share/turtlebot3_gazebo/models
```

）
随后使用`ros2 launch nav2_bringup tb3_simulation_launch.py headless:=False`就能启动仿真了
如果不走二进制安装，且实在无法编译源码，可以使用**Docker**容器，这里不详细展开

导航相关概念

生命周期节点

该节点为ros2独有的节点，和普通节点（`rclcpp::Node`）不一样，生命周期节点（`rclcpp_lifecycle::LifecycleNode`）引入了状态机，它被划分为

- 未配置—**Unconfigured**
- 未激活—**Inactive**
- 激活—**Active**
- 已结束—**Finalized**

四个状态
同时必须通过外部指令（或者launch）手动触发它的状态转换

	普通节点	生命周期节点
逻辑执行时间	构造函数完成	进入Active状态
配置灵活性	运行时修改参数较麻烦	可在Inactive状态下完成配置再激活
确定性	低（节点启动顺序不确定等）	高

在nav2中，我们可以使用`nav2_util LifecycleNode`这个包装器，它统一了原有的复杂逻辑并简化了操作，同时它还拥有一个`bond`，这个东西用于连接**生命周期管理器**与各个导航服务，如

果`bond`断开，管理器认为服务器不一定处于Active状态，无法信任，会直接**向下过渡**，让所有的导航节点都变为Inactive或者Unconfigured状态

有限状态机(Finite State Machine, FSM)

状态机是一种数学模型，描述了一个对象在其生命周期内所经历的状态，以及由于触发事件而导致的状态转换
有限状态机-FSM就是有限个状态，包含

- 有限个状态
- 转换条件
- 状态动作

它的局限性是当逻辑变得复杂，异常情况会增多，难以维护

行为树(Behaviour Tree, BT)

在这里仅简要介绍，具体可阅读《Behaviour Tree in Robotics and AI》（https://arxiv.org/abs/1709.00084）
假如有若干个状态，FSM就像是网状结构，状态就是各个网络中的节点，自由切换，但是逻辑复杂起来就异常庞大，难以维护
而BT就是一棵树，状态就是各个节点，例如有四个状态A,B,C,D，其中B，D是A的子节点，C是B的子节点
当切换状态时，A会每秒发出多次信号脉冲（**Tick**），这个tick会一直往下深探，直到到达某个节点后它返回**成功**或者**失败**，如果失败则往回走，如果成功则停下来，若这个节点只是一个**策略**，那么就直接继续走，在这里它经过B到达C，此时B根据C返回的信号决定是否回到A还是继续走或者停止继续
举个例子：
假如A是一个选择节点，B是走路，D是原地自旋，C是前进
A先发射tick到B，经过B到达C，如果确实可以前进，那就在这里停止，否则就返回失败信号，此时A知道B这个路走不通，就去D，再重复以上逻辑
Nav2使用`Behaviour CPP V3`作为行为树库，它可以加载子树，也就是说可以将一个nav2项目的BT加载到另一个更高级别的BT中，从而将其作为插件，同时它还提供了一个NavigationToPoseAction插件，可以从客户端应用程序通过通常的动作接口调用Nav2软件堆栈

导航服务器

规划器和控制器是导航任务的核心，恢复器用于使机器人摆脱不良状态或尝试处理各种形式的问题，以使系统具有容错能力

规划器、控制器和恢复器

在ros2中，这三个服务器以Action Server形式存在，并由BT统一管理调用
它们都托管于一个**地图算法插件**，即**代价地图—Costmap 2D**（被实现为pluginlib插件）及其相关插件层

- 地图算法插件
  所有的导航服务器都托管于代价地图上，规划器托管于**全局代价地图**（关注整张地图），控制器托管于**局部代价地图**（关注机器人周边局部地图）
  这个插件则是代价地图的层级结构，分为
  - **静态层**
    提供自地图的`.yaml`文件，记录的是死物
  - **障碍物层**
    来自激光雷达或者摄像机，捕捉的是活物或动物
  - **膨胀层**
    给障碍物画一个警戒圈，警示机器人与墙的距离

规划器和控制器都需要在运行时配置一个配置名称（任务别名）和使用的算法类型（已注册的插件库的名称）
例如使用名为`FollowPath`的DWB控制器，此时DWB的所有参数都会放置在该命名空间中，例`FollowPath.<param>`
然后这两个服务器会公开任务对应的操作接口，以便调用所选算法
对于恢复器，它类似于备选方案，即用于给机器人不同方案执行后出现问题后的备选方案，其公共接口一般是原地旋转、等待、后退、清理代价图层等
它主要由BT监控并调用，而不是用户

航点跟随

航点跟随是导航系统的基本功能之一，它会告知系统如何使用导航程序到达多个目的地
`nav2_waypoint_follower`软件包含一个航路点和跟踪程序，它一般有两种主要的实现思想，即哑机器人应用程序+智能集中式调度器和智能机器人应用程序+哑集中式调度器
区别在于统一由调度器考虑所有信息还是分给各部分分别考虑

状态估计

在导航项目中，需要提供两个主要的坐标转换：

- map 到 odom 的坐标变换由定位系统（定位，建图，SLAM）提供
- odom 到 base_link 的坐标转换由里程计系统提供

REP – 105(https://www.ros.org/reps/rep-0105.html)表示至少必须为机器人建造一个包含`map` -> `odom` -> `base_link` -> `[sensor frames]/base_laser` 的完整的TF树，TF2是ROS 2中的时变坐标变换库，Nav2使用TF2来表达和获取时间同步的坐标变换
全球定位系统 (GPS、SLAM-同步定位与建图、动作捕捉Motion Capture) 的工作是至少要提供 `map` -> `odom` 的坐标转换
然后，里程计系统的作用是提供 `odom` -> `base_link` 的坐标转化
关于 base_link 的其余坐标转换应该是静态的，并应在 URDF 中定义
`map`是全局地图坐标系，`odom`是里程计坐标系，`base_link`是机器人中心坐标系，`base_laser`是传感器坐标系
这个`map` -> `odom` -> `base_link` -> `base_laser`转换实际上是`base_laser`传感获取数据，处理为`base_link`坐标系下的数据，再处理为给`odom`坐标系下使用的数据，最后处理为`map`即整张地图的数据（类似于一个坐标）
这样的处理就是为了给导航服务器等其他东西使用的，用于规划和执行机器人运动方案
为什么要多一个odom？为什么不直接把转换到odom这一步的数据直接当成map处理？
因为里程计有误差，从odom到map这一步转换通过AMCL-**自适应蒙特卡洛定位**插件得出，即一个微小的数据修正处理
最后得到的map才是精准的数据

地图过滤器

我们可以对地图文件进行**注释**，即可以将某些区域排除在外避免在这些区域内进行路径规划，这种带注释的地图称为**Filter mask**
这个地图过滤器基于成本地图插件实现

里程计系统

我们可以从各种传感器硬件中获取里程计信息，如IMU(**Inertial Measurement Unit**)-惯性测量单元、LIDAR、RADAR等，odom框架与之相关
里程计系统主要用于解决机器人局部的运动问题（局部的轮胎打滑、运动漂移、运动不稳等）
这个`odom` -> `base_link`的转换一般由tf2或诸如`robot_localization`等的框架发布

这个`robot_localization`会订阅各个有关机器人位姿和运动参数（传感器IMU、车轮编码器等），通过一系列矫正算法（EKF、UKF等）计算出最合适的转换并发布，和`map -> odom`不同的是，这个转换着重于在整个地图上的精确位置，而这个`robot_localization`则是关注于机器人的局部

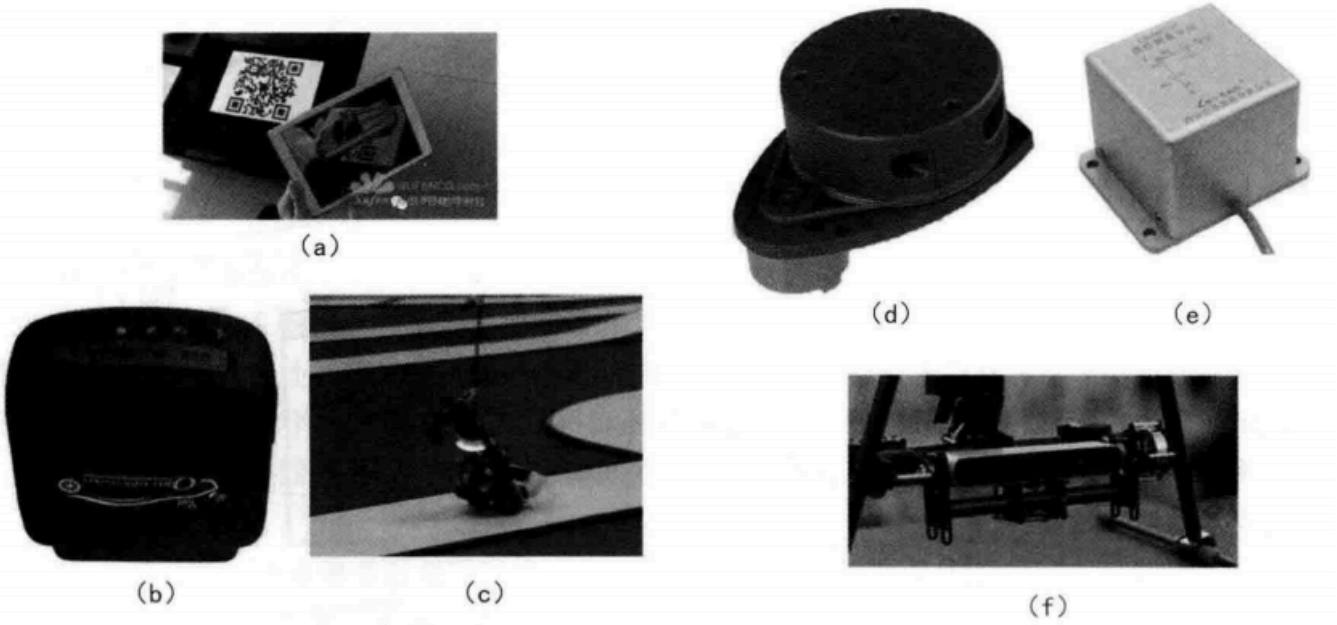


图 2-2　一些传感器。(a) 利用二维码进行定位的增强现实软件；(b) GPS 定位装置；(c) 铺设导轨的小车；(d) 激光雷达；(e) IMU 单元；(f) 双目相机