



###写在前面###

通过前面的基础学习，本章进入最为激动的机器人自主导航的学习。在前面的学习铺垫后，终于迎来了最大乐趣的时刻，就是赋予我们的 miiboo 机器人能自由行走的生命。本章将围绕机器人 SLAM 建图、导航避障、巡航、监控等内容展开。本章内容：

- 1.在机器人上使用传感器
- 2.google-cartographer 机器人 SLAM 建图
- 3.ros-navigation 机器人自主避障导航
- 4.多目标点导航及任务调度
- 5.机器人巡航与现场监控

###正文###

1.在机器人上使用传感器

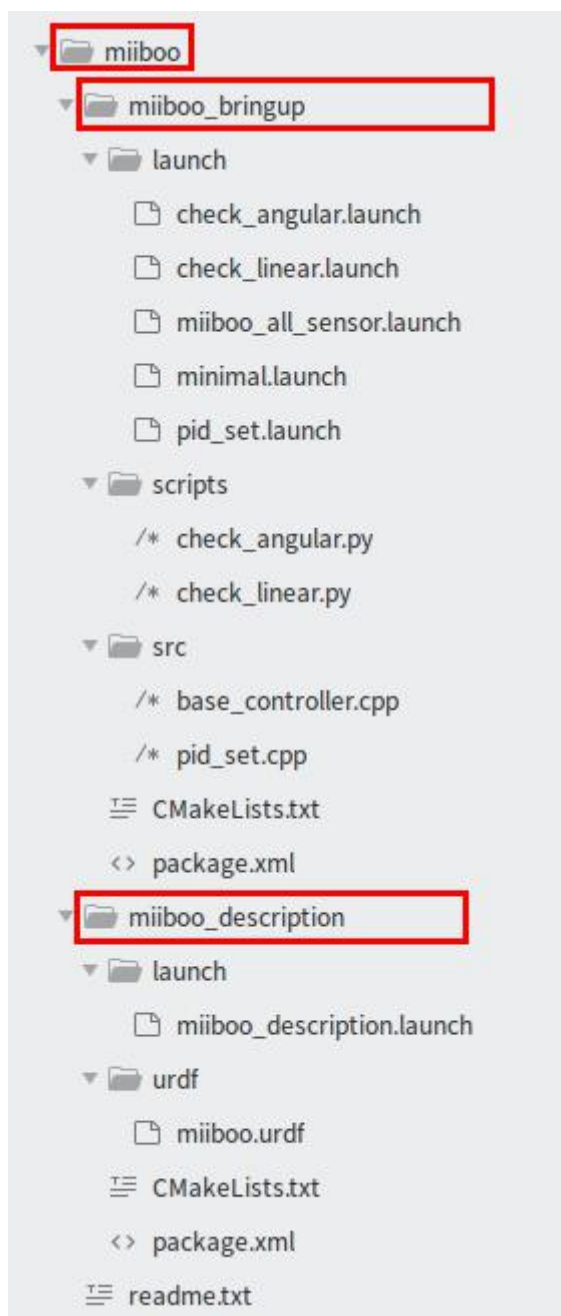
SLAM 建图需要用到底盘、激光雷达和 IMU，所以这里详细介绍如何在机器人上使用这些传感器。要使用这些传感器也很简单，就是在机器人上开启相应传感器的 ROS 驱动节点，在设置合适的可配参数就行了。

需要注意的是，底盘、激光雷达、IMU 这三个传感器都使用串口与树莓派通信，为了防止每次开机这三个设备的串口号发生变动，需要将串口号进行绑定与重映射，操作方法在前面已经介绍过了，如果还没有绑定直接前往前面相关内容参考。

这里将建立一个叫 catkin_ws 的 ROS 工作空间，专门用于存放机器人传感器相关的 ROS 驱动功能包。关于创建 ROS 工作空间的操作，请参考前面相应部分内容，这里就不做讲解。

1.1.使用底盘

在机器人上只需要使用 miiboo 这个驱动包就可以驱动底盘了。将 miiboo 这个驱动包拷贝到 ~/catkin_ws/src/中，编译后就可以使用了。miiboo 驱动包文件结构，如图 1。miiboo 驱动包中含有两个 ROS 功能包 miiboo_bringup 和 miiboo_description，驱动 miiboo 底盘、底盘 PID 整定、里程计标定这些功能包含在 miiboo_bringup 中，miiboo 底盘 urdf 模型包含在 miiboo_description 中。



(图 1) miiboo 驱动包文件结构

底盘控制可配参数:

关于底盘控制可配参数都放在 `miiboo_bringup/launch/minimal.launch` 中, 如图 2。

参数 `com_port` 是底盘控制的串口号, 由于前面已经做了绑定, 所以直接填入绑定好的名称 `/dev/miiboo` 就行了;

参数 `speed_ratio` 是里程计走直线标定值, 这个值通过标定得到。

参数 `wheel_distance` 是里程计转角标定值, 这个值通过标定得到。

其余参数一般不需要修改, 如有需要可以结合阅读源码来了解参数含义和做相应修改。

```
minimal.launch
<launch>
  <node name="miiboo_bringup_node" pkg="miiboo_bringup" type="base_controller" output="screen">
    <!-- serial com set -->
    <param name="com_port" value="/dev/miiboo"/>

    <!-- motor param set -->
    <param name="speed_ratio" value="0.00085"/><!-- unit:m/encode -->
    <param name="wheel_distance" value="0.32326403"/><!-- unit:m -->
    <param name="encode_sampling_time" value="0.04"/><!-- unit:s -->

    <!-- velocity limit -->
    <param name="cmd_vel_linear_max" value="1.5"/><!-- unit:m/s -->
    <param name="cmd_vel_angular_max" value="2.0"/><!-- unit:rad/s -->

    <!-- other -->
    <param name="cmd_vel_topic" value="cmd_vel"/>
    <param name="odom pub topic" value="odom"/>
    <param name="wheel_left_speed_pub_topic" value="wheel_left_speed"/>
    <param name="wheel_right_speed_pub_topic" value="wheel_right_speed"/>
    <param name="odom_frame_id" value="odom"/>
    <param name="odom_child_frame_id" value="base_footprint"/>
  </node>
</launch>
```

(图 2) 底盘控制可配参数

驱动 miiboo 底盘:

其实很简单, 一条命令启动 miiboo 底盘控制。

```
roslaunch miiboo_bringup minimal.launch
```

底盘 PID 整定:

我们的 miiboo 机器人底盘的 stm32 控制板中已经内置了整定好的 PID 参数, 如果选用我们提供的控制板和电机, 一般情况下是不需要整定 PID 的。

对于想体验一下 PID 参数整定过程或将我们的 miiboo 机器人底盘的 stm32 控制板应用到其他地方的朋友, 这里给出了整定 PID 的整个操作过程和思路, 方便大家学习和更深层次的研究。这里主要讲解 PID 整定的操作, 关于原理性的东西可以参考前面相关内容进行了解。

由于底盘 PID 整定是非必须的功能, 所以没有对底盘 PID 整定的串口 (DEBUG-uart1) 做绑定, 需要先手动插入该串口到树莓派 3, 然后手动查看该串口的设备号, 并修改该设备号的读写权限。然后将该设备号填入 miiboo_bringup/launch/pid_set.launch 中的 com_port 参数中。然后, 需要启动底盘控制节点、底盘调试节点、键盘控制节点。键盘控制节点 teleop_twist_keyboard 需要通过 apt-get 命令来安装, rqt_plot 是 ROS 提供的绘图工具。

```
#打开终端, 启动底盘控制节点
roslaunch miiboo_bringup minimal.launch
```

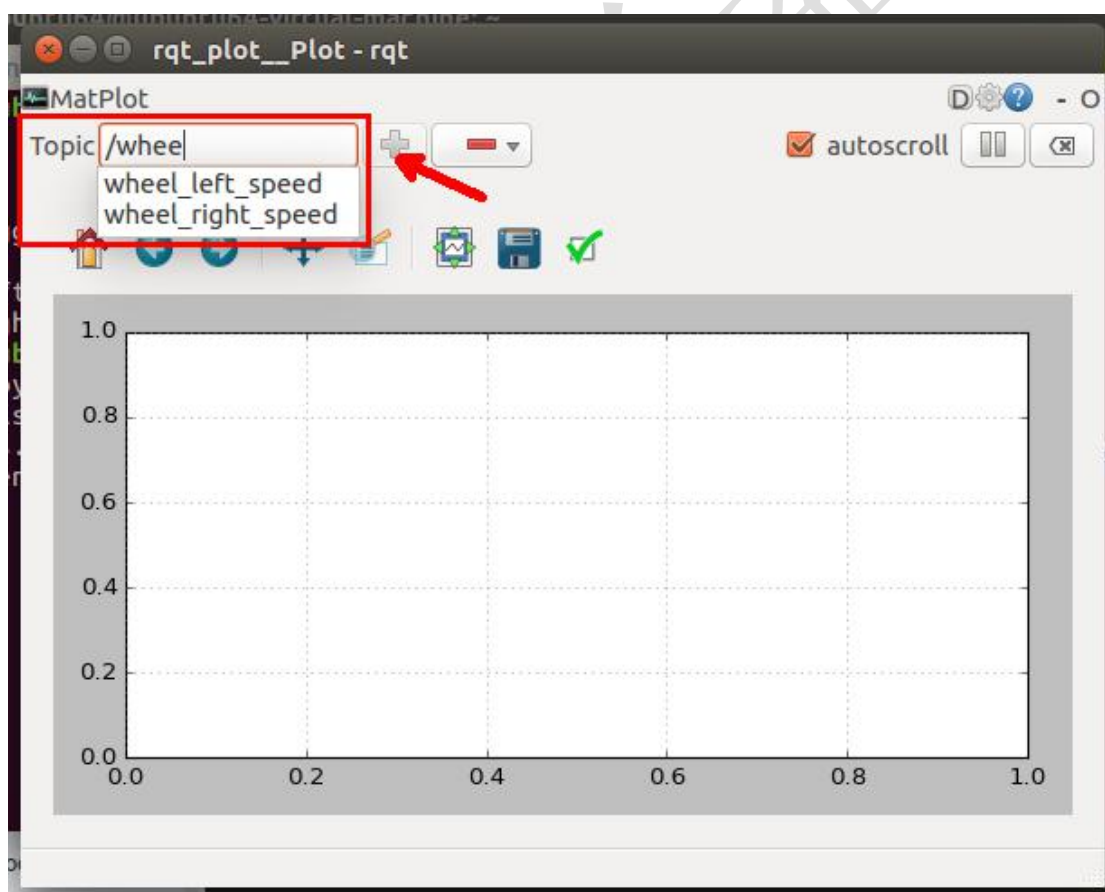
```
#再打开一个终端, 启动底盘调试节点, 按提示输入命令
roslaunch miiboo_bringup pid_set.launch
```

```
#安装键盘控制工具
sudo apt-get install ros-kinetic-teleop-twist-keyboard
```

```
#再打开一个终端, 启动键盘控制节点,
source ~/.bashrc
roslaunch teleop_twist_keyboard teleop_twist_keyboard.py
```

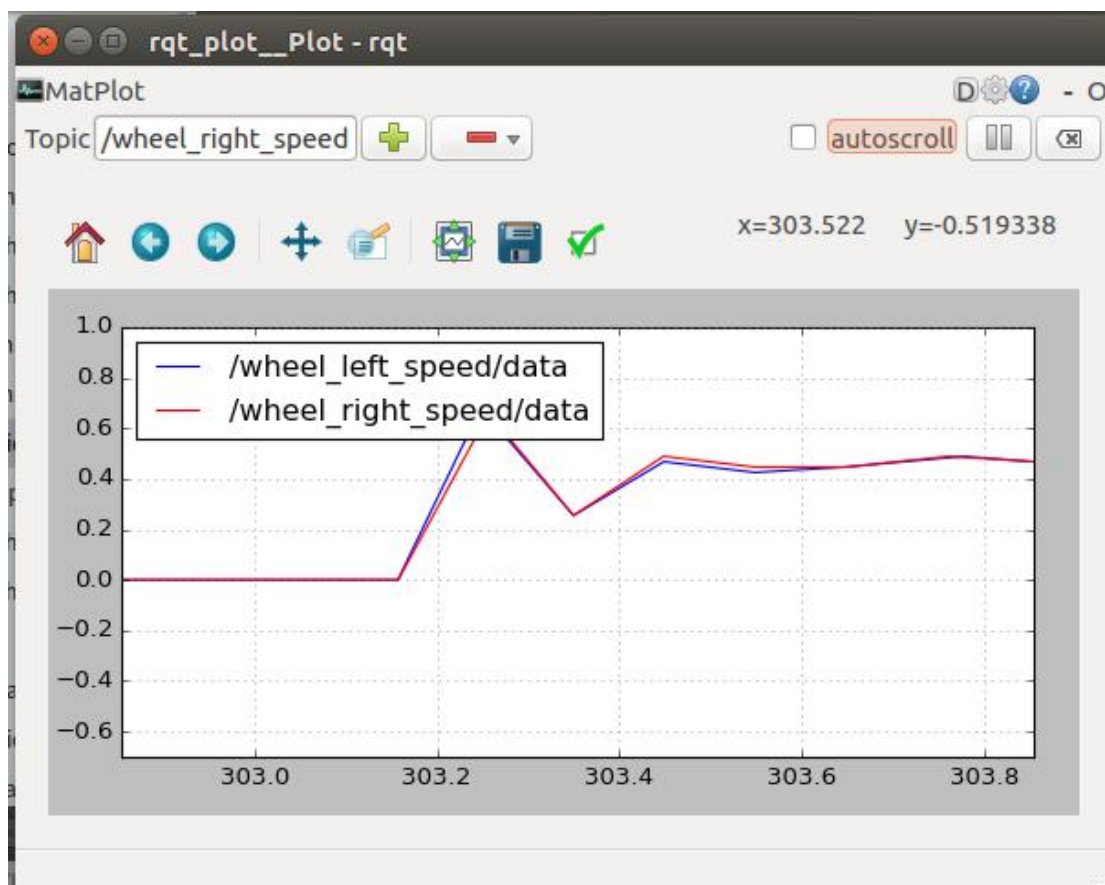
```
#再打开一个终端, 用 rqt_plot 对底盘速度曲线进行绘制, 指定曲线数据来源的 topic
roslaunch rqt_plot rqt_plot
```

ROS 提供的绘图工具 `rqt_plot` 用法很简单，在 `rqt_plot` 界面中，在 `Topic` 栏输入曲线数据来源，我们这里为左、右轮速度，然后点击旁边的“+”将曲线加入绘制界面，人如图 3。



（图 3）用 `rqt_plot` 显示速度曲线

然后，通过在启动 `teleop_twist_keyboard` 节点的终端通过 `I`/`<`/`J`/`L` 四个按键来控制底盘前进/后退/左转/右转控制，并观察速度曲线的变化，根据 `PID` 整定规则对 `PID` 参数进行整定，在启动 `pid_set.launch` 的终端下按相应提示输入 `PID` 参数实现对整定参数的编辑。直到得到一个比较好的速度曲线，就可以结束整定过程了。实时速度曲线显示，如图 4。



(图 4) 实时速度曲线显示

里程计标定:

机器人底盘运行的精度是衡量底盘的重要指标。底盘精度受里程计的走直线误差和转角误差影响。因此，需要对里程计的走直线和转角进行标定，尽量减小误差。miiboo 机器人底盘的 ROS 驱动中已经写好了相应的标定程序，跟里程计标定有关的文件主要有：

.../miiboo_bringup/launch/check_linear.launch 为里程计走直线标定启动文件

.../miiboo_bringup/launch/check angular.launch 为里程计转角标定启动文件

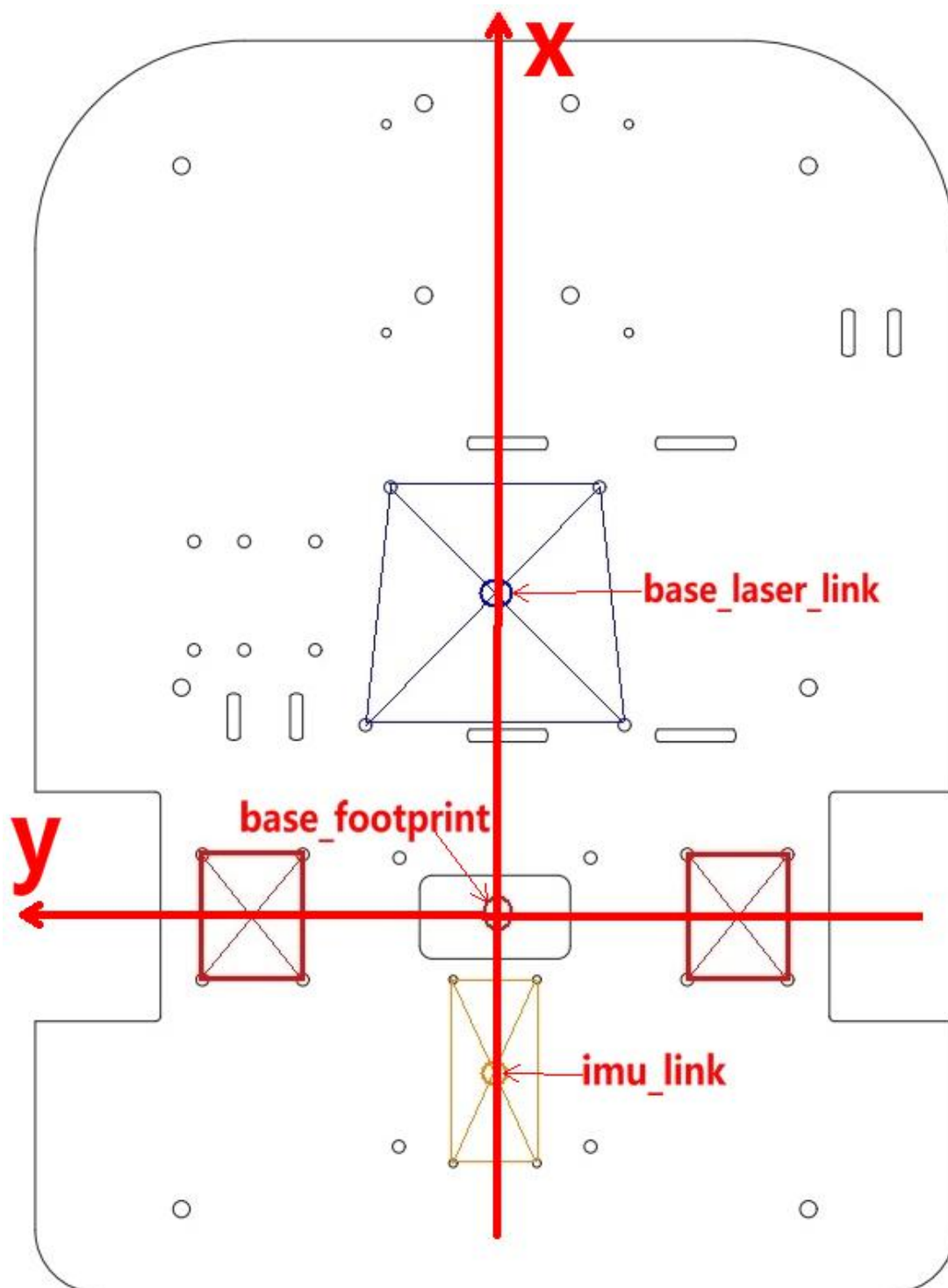
.../miiboo_bringup/launch/minimal.launch 为设置标定参数及底盘控制启动文件

下面是标定步骤过程。

由于标定过程在前面已经讲解过了，请直接前往相应内容参考。

miiboo 底盘 urdf 模型:

urdf 模型描述了机器人底盘的形状、传感器之间的安装关系、各个传感器在 tf tree 中的关系。其实，miiboo 底盘 urdf 模型的主要作用是提供各个传感器在 tf tree 中的关系，这些关系将在 SLAM 和导航算法中被使用。



(图 5) miiboo 机器人底盘中各个传感器 tf 关系

图 5 是 miiboo 机器人底盘中各个传感器 tf 关系，`base_footprint` 是底盘的运动中心，`base_laser_link` 是激光雷达的中心，`imu_link` 为 IMU 模块的中心。以 `base_footprint` 为原点，建立机器人底盘的坐标系，坐标系为标准右手系，即底盘正前方为 x 轴、正左方为 y 轴、正上方为 z 轴、以 x 轴起始逆时针方向为 theta 轴。以 `base_footprint` 为父坐标系，建立 `base_footprint->base_laser_link` 关系，建立 `base_footprint->imu_link` 关系，就实现了各个传感器 tf 关系的构建，构建的具体实现在 `miiboo_description/urdf/miiboo.urdf` 中完成。如图 6，为 `miiboo.urdf` 的具体内容。


```
miiboo.urdf x
</material>
<material name="gray">
  <color rgba="0.2 0.2 0.2 1" />
</material>

<link name="base_footprint"/>

<!-- base_link -->
<link name="base_link"/>
<joint name="base_link_joint" type="fixed">
  <parent link="base_footprint" />
  <child link="base_link" />
  <origin xyz="0 0 0.065" rpy="0 0 0.0" />
</joint>

<!-- laser -->
<link name="base_laser_link"/>
<joint name="base_laser_link_joint" type="fixed">
  <origin xyz="0.08 0.00 0.065" rpy="0 0 0.0" />
  <parent link="base_footprint" />
  <child link="base_laser_link" />
</joint>

<!-- imu of hi219 -->
<link name="imu_link"/>
<joint name="imu_link_joint" type="fixed">
  <origin xyz="-0.035 0.00 0.065" rpy="0 0 0" />
  <parent link="base_footprint" />
  <child link="imu_link" />
</joint>
</robot>
```

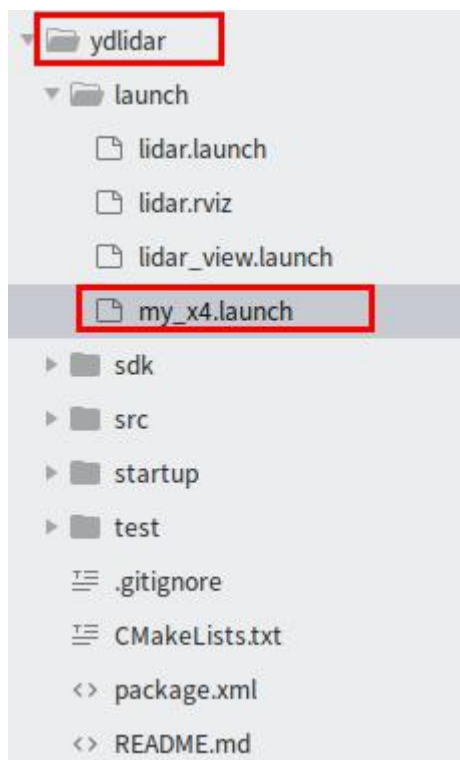
(图 6) miiboo 底盘 urdf 模型描述文件

要使用这个 urdf 模型就很简单了，直接一句命令启动。

```
roslaunch miiboo_description miiboo_description.launch
```

1.2. 使用激光雷达

在机器人上只需要使用 ydlidar 这个驱动包就可以驱动 ydlidar-x4 雷达了。将 ydlidar 这个驱动包拷贝到~/catkin_ws/src/中，编译后就可以使用了。ydlidar 驱动包文件结构，如图 7。ydlidar 驱动包中的其他文件我们不需要关心，这些都是由雷达厂商提供的标准驱动，只需要设置我们自己建立的 ydlidar/launch/my_x4.launch 文件，这个用于启动雷达。



(图 7) ydlidar 驱动包文件结构

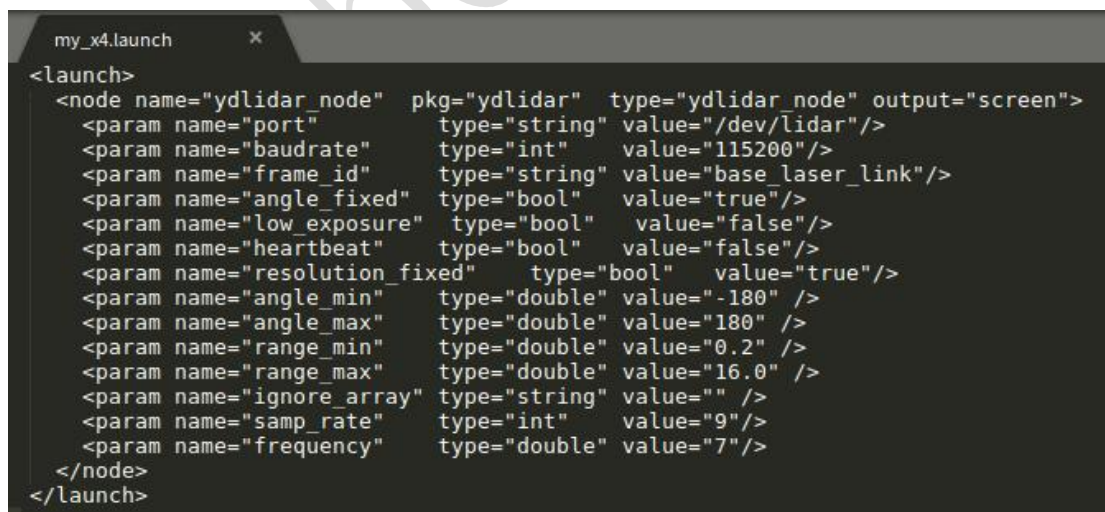
激光雷达数据可配参数:

关于激光雷达数据可配参数都放在 ydlidar/launch/my_x4.launch 中, 如图 8。

参数 port 是激光雷达的串口号, 由于前面已经做了绑定, 所以直接填入绑定好的名称 /dev/lidar 就行了;

参数 range_min 和 range_max 是设置激光雷达数据的有效值区间。

其余参数一般不需要修改, 如有需要可以结合阅读源码来了解参数含义和做相应修改。



(图 8) 激光雷达数据可配参数

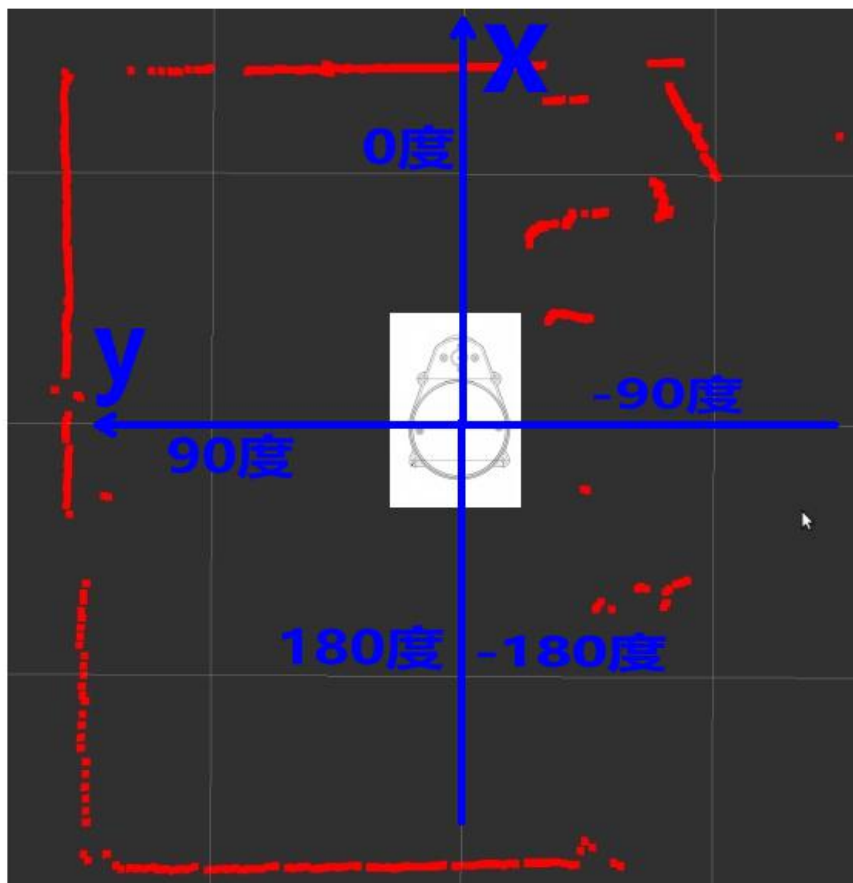
驱动 ydlidar-x4 激光雷达:

其实很简单, 一条命令启动 ydlidar-x4 激光雷达。

```
roslaunch ydlidar my_x4.launch
```


激光雷达数据格式:

激光雷达采用右手坐标系，雷达正前方为 x 轴、正左方为 y 轴、正上方为 z 轴、以 x 轴起始逆时针方向为 θ 轴。激光雷达的扫描数据以极坐标的形式表示，雷达正前方是极坐标 0 度方向、雷达正左方是极坐标 90 度方向，红色点为扫描到的数据点，如图 9 所示。



(图 9) 激光雷达数据格式

激光雷达的数据在 ROS 中是以 `sensor_msgs/LaserScan` 消息类型进行表示，如图 10，`angle_increment` 表示激光数据点的极坐标递增角度，`ranges` 数组存放实际的极坐标点距离值。

`sensor_msgs/LaserScan`

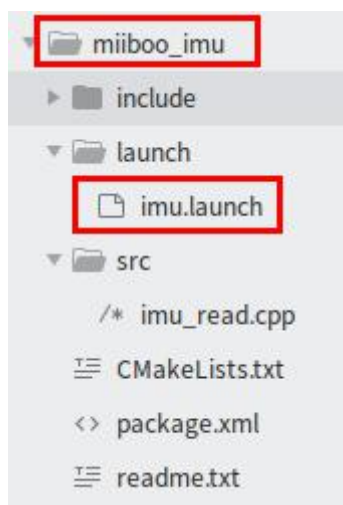
```
std_msgs/Header header
float32 angle_min
float32 angle_max
float32 angle_increment
float32 time_increment
float32 scan_time
float32 range_min
float32 range_max
float32[] ranges
float32[] intensities
```

(图 10) 激光雷达数据 `sensor_msgs/LaserScan` 消息类型

1.3.使用 IMU



在机器人上只需要使用 miiboo_imu 这个驱动包就可以驱动 mpu9250 模块了。将 miiboo_imu 这个驱动包拷贝到~/catkin_ws/src/中，编译后就可以使用了。miiboo_imu 驱动包文件结构，如图 11。ydlidar 驱动包中的其他文件我们不需要关心，只需要设置 ydlidar/launch/my_x4.launch 文件，这个用于启动 IMU。



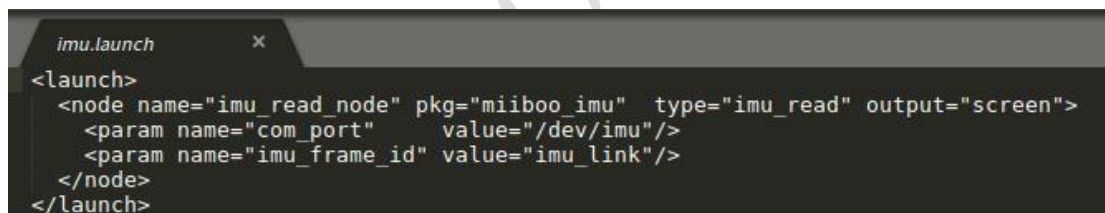
(图 11) miiboo_imu 驱动包文件结构

IMU 数据可配参数:

关于 IMU 数据可配参数都放在 miiboo_imu/launch/imu.launch 中，如图 12。

参数 come_port 是 IMU 的串口号，由于前面已经做了绑定，所以直接填入绑定好的名称 /dev/imu 就行了；

其余参数一般不需要修改，如有需要可以结合阅读源码来了解参数含有和做相应修改。



(图 12) IMU 数据可配参数

驱动 IMU 模块:

其实很简单，一条命令启动 IMU 模块。

```
roslaunch miiboo_imu imu.launch
```

IMU 数据格式:

IMU 模块采用右手坐标系，IMU 模块正前方为 x 轴、正左方为 y 轴、正上方为 z 轴。IMU 模块提供 3 轴加速度、3 轴角速度、3 轴磁力计、经数据融合后用欧拉角表示的姿态。

IMU 数据在 ROS 中是以 sensor_msgs/Imu 消息类型进行表示，如图 13。

sensor_msgs/Imu

```
std_msgs/Header header
geometry_msgs/Quaternion orientation
float64[9] orientation_covariance
geometry_msgs/Vector3 angular_velocity
float64[9] angular_velocity_covariance
geometry_msgs/Vector3 linear_acceleration
float64[9] linear_acceleration_covariance
```

(图 13) IMU 数据 sensor_msgs/Imu 消息类型

1.4.使用摄像头

miiboo 机器人上使用的是 USB 摄像头，用 ROS 驱动 USB 摄像头可以采用以下 3 中方法。

方法 1:

使用 usb_cam 这个 ROS 包直接驱动

方法 2:

使用 gscam 这个 ROS 包直接驱动

方法 3:

自制 OpenCV, cv_bridge, image_transport 驱动 ROS 包

为了方便起见，我采用的是方法 1，直接安装 usb_cam 这个 ROS 包直接驱动。

usb_cam 摄像头驱动安装:

将 usb_cam 下载到~/catkin_ws/src/中，直接编译就行了。

```
cd ~/catkin_ws/src/
git clone https://github.com/ros-drivers/usb_cam.git

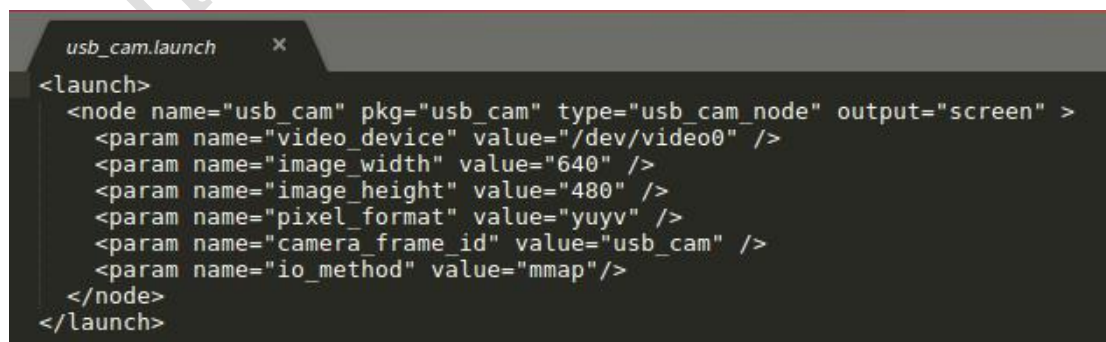
rosdep install usb_cam
cd ~/catkin_ws/
catkin_make
```

摄像头数据可配参数:

关于摄像头数据可配参数都放在 usb_cam/launch/usb_cam.launch 中，如图 14。

参数 video_device 是摄像头的设备号，由于直插了一个 USB 摄像头，所以直接填入名称 /dev/video0 就行了；

其余参数一般不需要修改，如有需要可以结合阅读源码来了解参数含有和做相应修改。



```
usb_cam.launch
<launch>
  <node name="usb_cam" pkg="usb_cam" type="usb_cam_node" output="screen" >
    <param name="video_device" value="/dev/video0" />
    <param name="image_width" value="640" />
    <param name="image_height" value="480" />
    <param name="pixel_format" value="yuyv" />
    <param name="camera_frame_id" value="usb_cam" />
    <param name="io_method" value="mmap" />
  </node>
</launch>
```

(图 14) 摄像头数据可配参数

驱动 USB 摄像头:

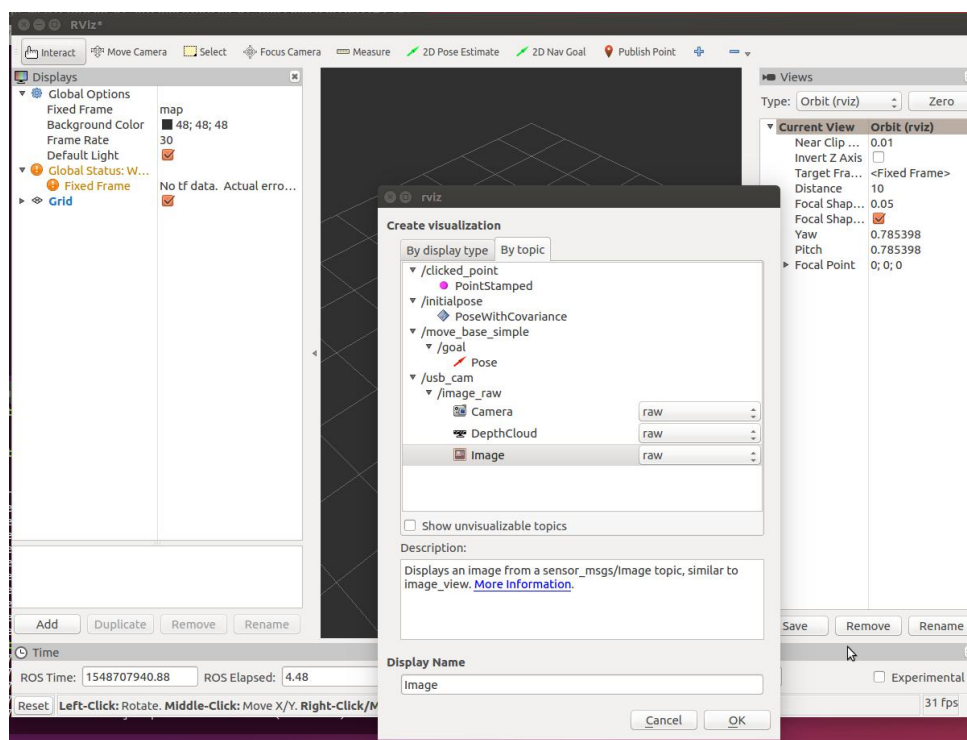
其实很简单，一条命令启动 USB 摄像头。

```
roslaunch usb_cam usb_cam.launch
```

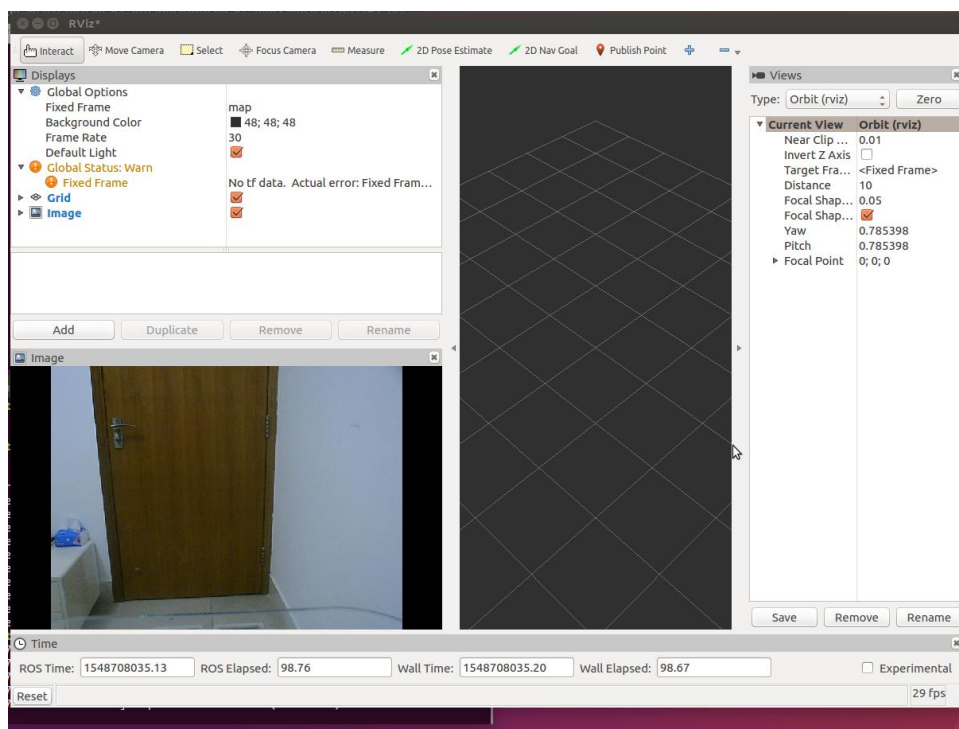
摄像头远程显示:

摄像头数据远程显示的方法有两种，方法一是在 PC 端 rviz 中订阅摄像头发布的图像 topic，方法二是用 Android 手机上 miiboo 机器人 APP 直接显示。

先说方法一，在 PC 端打开 rviz，在 rviz 中添加需要显示的 Topic，这样就可以看到图像了。如图 15 和 16。

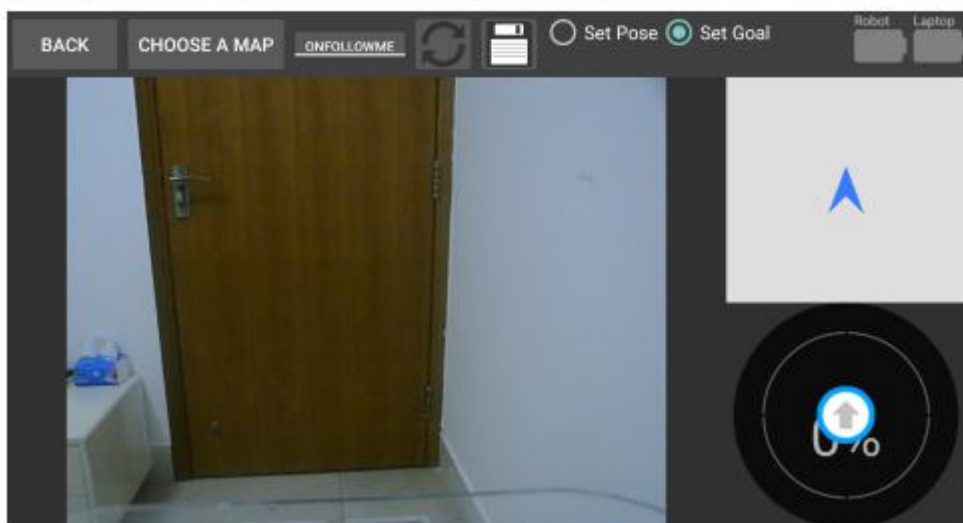


(图 15) 在 rviz 中添加需要显示的图像 Topic



(图 16) 在 rviz 中显示图像 Topic

方法二，就很简单了，只要 Android 手机上 miiboo 机器人 APP 连接到机器人端成功后，就能自动显示图像了。如图 17。



(图 17) 在 Android 手机的 miiboo 机器人 APP 中显示图像 Topic

1.5.局域网内广播机器人自己的 IP

这个很简单，由 `broadcast_ip` 这个功能包实现，我已经写好放入 `~/catkin_ws/src/` 并编译了。只需要一句命令启动就行了。

```
roslaunch broadcast_ip broadcast_udp.launch
```

2.google-cartographer 机器人 SLAM 建图

主流的激光 SLAM 算法有 `hector`、`gmapping`、`karto`、`cartographer`。

`hector` 是一种结合了鲁棒性较好的扫描匹配方法 2D_SLAM 方法和使用惯性传感系统的导航技术。传感器的要求较高，高更新频率小测量噪声的激光扫描仪，不需要里程计。使空中无人机与地面小车在不平坦区域运行存在运用的可能性。作者利用现代激光雷达的高更新率和低

距离测量噪声,通过扫描匹配实时地对机器人运动进行估计。所以当只有低更新率的激光传感器时,即便测距估计很精确,对该系统都会出现一定的问题。

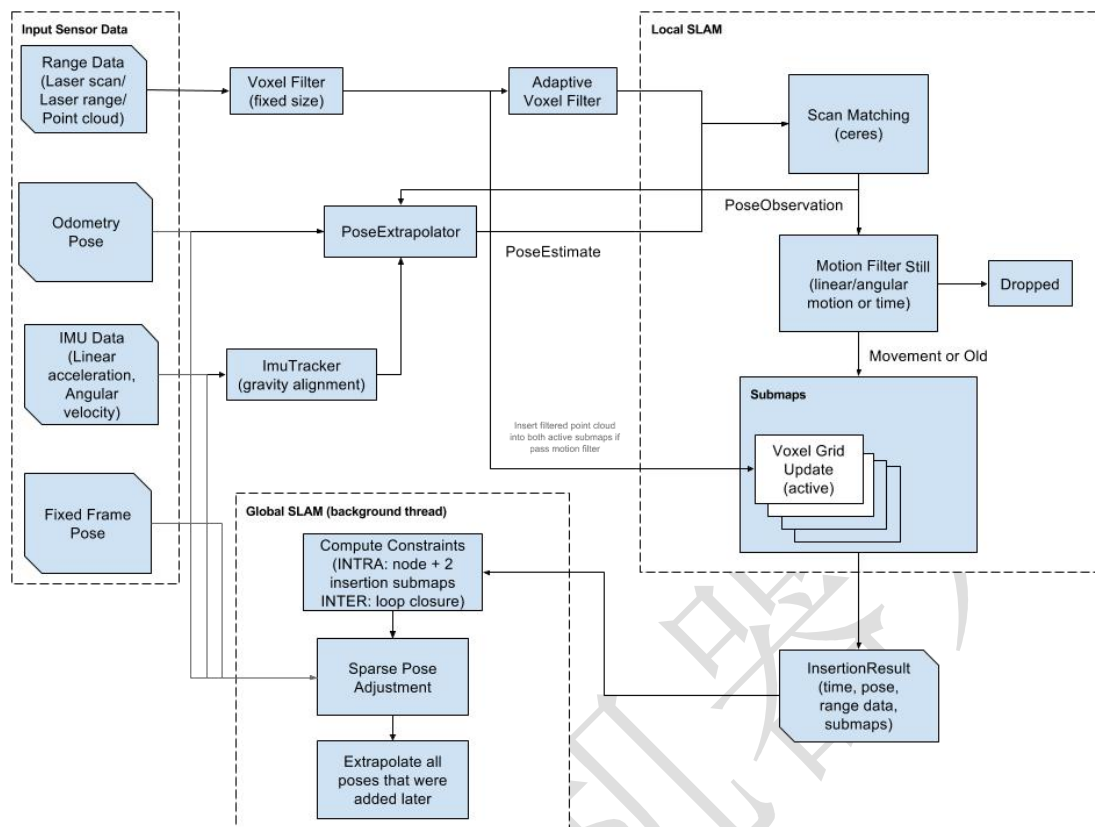
gmapping 是一种基于粒子滤波的激光 **SLAM** 算法,它已经集成在 **ROS** 中,是移动机器人中使用最多的 **SLAM** 算法。基于粒子滤波的算法用许多加权粒子表示路径的后验概率,每个粒子都给出一个重要性因子。但是,它们通常需要大量的粒子才能获得比较好的结果,从而增加该算法的计算复杂性。此外,与 **PF** 重采样过程相关的粒子退化耗尽问题也降低了算法的准确性。

karto 是基于图优化的 **SLAM** 算法,用高度优化和非迭代 **cholesky** 矩阵进行稀疏系统解耦作为解。图优化方法利用图的均值表示地图,每个节点表示机器人轨迹的一个位置点和传感器测量数据集,箭头的指向的连接表示连续机器人位置点的运动,每个新节点加入,地图就会依据空间中的节点箭头的约束进行计算更新。路标 **landmark** 越多,内存需求越大,然而图优化方式相比其他方法在大环境下制图优势更大。

cartographer 是 **google** 开发的实时室内 **SLAM** 项目, **cartographer** 采用基于 **google** 自家开发的 **ceres** 非线性优化的方法, **cartographer** 的量点在于代码规范与工程化,非常适合于商业应用和再开发。并且 **cartographer** 基于 **submap** 子图构建全局地图的思想,能有效的避免建图过程中环境中移动物体的干扰。并且 **cartographer** 支持多传感器数据 (**odometry**、**IMU**、**LaserScan** 等) 建图,支持 **2D_SLAM** 和 **3D_SLAM** 建图。所以,我果断采用 **cartographer** 来建图,我的树莓派 3 主板跑 **cartographer** 实时建图是十分的流畅,这一点很欣慰^_^

2.1.google-cartographer 建图算法原理分析

cartographer 采用的是主流的 **SLAM** 框架,也就是特征提取、闭环检测、后端优化的三段式。由一定数量的 **LaserScan** 组成一个 **submap** 子图,一系列的 **submap** 子图构成了全局地图。用 **LaserScan** 构建 **submap** 的短时间过程累计误差不大,但是用 **submap** 构建全局地图的长时间过程就会存在很大的累计误差,所以需要利用闭环检测来修正这些 **submap** 的位置,闭环检测的基本单元是 **submap**,闭环检测采用 **scan_match** 策略。**cartographer** 的重点内容就是融合多传感器数据 (**odometry**、**IMU**、**LaserScan** 等) 的 **submap** 子图创建以及用于闭环检测的 **scan_match** 策略的实现。



(图 18) cartographer 算法系统框图

2.2.cartographer_ros 安装

我们直接参考 google-cartographer 官方教程安装就行，官方教程分为 cartographer 和 cartographer_ros，其实 cartographer 就是核心算法层、cartographer_ros 是核心算法层的 ros 调用层。官方教程如下：

<https://google-cartographer.readthedocs.io/en/latest/index.html#>

<https://google-cartographer-ros.readthedocs.io/en/latest/index.html#>

直接按照第二个链接 cartographer_ros 的安装教程，就可将 cartographer_ros、cartographer、以及各种依赖都安装了。不过特别说明一点，为了解决从官网下载 ceres-solver 速度慢的问题，我将 ceres-solver 的下载地址换到了 github 源；我需要将官方教程中生成的 src/.rosinstall 替换成了自己的内容，如图 19。其余安装过程和官方教程一模一样。

(1) 安装编译工具

我来编译 cartographer_ros，我们需要用到 wstool 和 rosdep。为了加快编译，我们使用 ninja 工具进行编译。

```
sudo apt-get update
sudo apt-get install -y python-wstool python-rosdep ninja-build
```

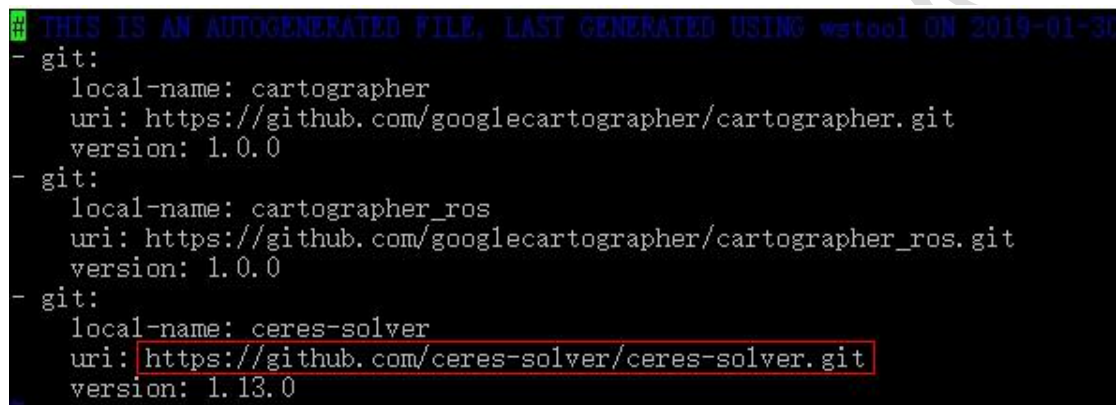
(2) 创建存放 cartographer_ros 的专门工作空间

```
mkdir catkin_ws_carto
cd catkin_ws_carto
wstool init src

wstool merge -t src
https://raw.githubusercontent.com/googlecartographer/cartographer_ros/master/cartographer_ros.rosinstall

wstool update -t src
```

特别说明，在执行 `wstool update -t src` 之前，需要将 `src/.rosinstall` 文件修改成以下内容，以解决 `ceres-solver` 下载不了的问题，如图 19。



```
THIS IS AN AUTOGENERATED FILE, LAST GENERATED USING wstool ON 2019-01-30
- git:
  local-name: cartographer
  uri: https://github.com/googlecartographer/cartographer.git
  version: 1.0.0
- git:
  local-name: cartographer_ros
  uri: https://github.com/googlecartographer/cartographer_ros.git
  version: 1.0.0
- git:
  local-name: ceres-solver
  uri: https://github.com/ceres-solver/ceres-solver.git
  version: 1.13.0
```

（图 19）我修改后的 `src/.rosinstall` 文件内容

（3）安装依赖项

安装 `cartographer_ros` 的依赖项 `proto3`、`deb` 包等。如果执行 `sudo rosdep init` 报错，可以直接忽略。

```
src/cartographer/scripts/install_proto3.sh
sudo rosdep init
rosdep update

rosdep install --from-paths src --ignore-src
--rosdistro=${ROS_DISTRO} -y
```

（4）编译和安装

上面的配置和依赖都完成后，就可以开始编译和安装 `cartographer_ros` 整个项目工程了。

```
catkin_make_isolated --install --use-ninja
```

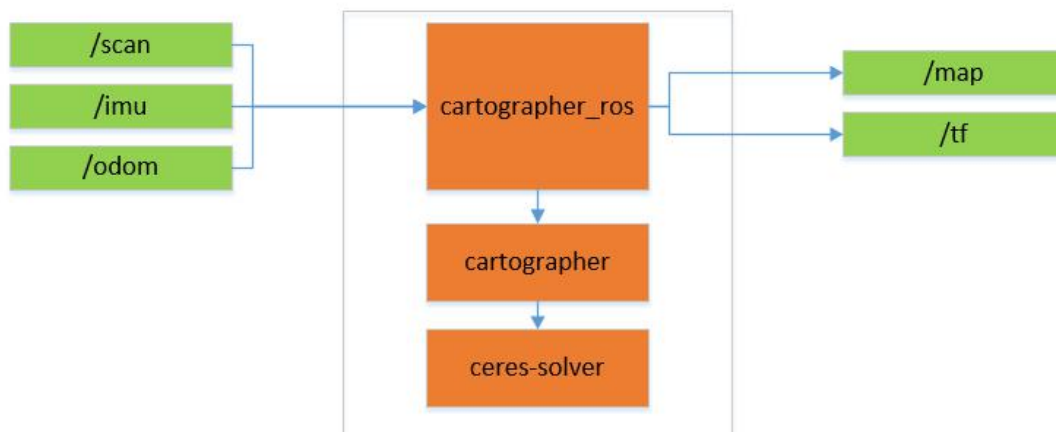
特别提醒，以后对 `cartographer_ros` 中的配置文件或源码有改动时，都需要执行这个编译命令使修改生效。

2.3.cartographer_ros 使用

cartographer_ros 整体代码结构分析：

最顶层的是 `cartographer_ros`，作为 `rosl` 接口调用层，通过调用 `cartographer` 核心算法，订阅多传感器数据（`/scan`、`/imu`、`/odom` 等），并发布地图、机器人位置信息（`/map`、`/tf` 等）；其次是 `cartographer`，作为 SLAM 算法的核心实现，特征提取、子图构建、闭环检测、全局

优化都在这里实现，其中优化过程需要调用 `ceres-solver` 非线性优化库；最后是 `ceres-solver`，是非线性优化库，用于求解 SLAM 中的优化问题。



(图 20) `cartographer_ros` 整体代码结构

在 miiboo 机器人上用 `cartographer_ros` 多传感器建图进行配置：

经过前面对 `cartographer_ros` 进行安装后，我们肯定迫不及待想在实际的 miiboo 机器人上使用 `cartographer_ros` 进行 SLAM 建图了。为了最大限度的提高 SLAM 建图的性能，我们的 miiboo 机器人提供了激光雷达、IMU、轮式里程计（`/scan`、`/imu`、`/odom`）这三种传感器的数据，所以我们需要先将 `cartographer_ros` 配置成对应的工作模式。

`cartographer` 算法是一个非常通用和适应不同平台的开放框架算法，所以支持多种配置与工作模式。我们就来看看 `cartographer_ros` 如何进行配置。配置文件由 `*.lua` 书写被放在路径 `cartographer_ros/configuration_files/`，我们需要建立一个我们自己的配置文件，取名就叫 `miiboo_mapbuild.lua` 吧，具体内容如图 21。由于我们的 miiboo 机器人采用激光雷达、IMU、轮式里程计三种传感器融合建图，所以以下参数一定要设置正确：

参数 `tracking_frame` 设置为 `imu_link`，因为我们使用 `/imu` 的数据；

参数 `published_frame` 设置为 `odom`，因为我们使用 `/odom` 的数据；

参数 `provide_odom_frame` 设置为 `false`，因为我们使用外部 `/odom`，所以这里不需要内部提供；

参数 `use_odometry` 设置为 `true`，因为我们使用外部 `/odom` 的数据；

参数 `use_imu_data` 设置为 `true`，因为我们使用 `/imu` 的数据；

参数 `imu_gravity_time_constant` 设置为 10，这个是 IMU 的重力加速度常数。

其余参数根据需要自行调整，由于 `cartographer` 是发展很迅速的算法，所以很多代码和文档一直在更新，所以参考官方文档来解读这些参数的含义是最好的选择，官方文档连接地址我贴在下面了。

<https://google-cartographer-ros.readthedocs.io/en/latest/index.html>


```
miiboo_mapbuild.lua x
include "map_builder.lua"
include "trajectory_builder.lua"

options = {
  map_builder = MAP_BUILDER,
  trajectory_builder = TRAJECTORY_BUILDER,
  map_frame = "map",
  tracking_frame = "imu_link", --default:"base_link","imu_link" if use imu
  published_frame = "odom", --default:"base_link","odom" if use odometry
  odom_frame = "odom", --default:"odom",no-use if provide_odom_frame = false
  provide_odom_frame = false, --default:true
  publish_frame_projected_to_2d = false,
  use_odometry = true, --default:false
  use_nav_sat = false,
  use_landmarks = false,
  num_laser_scans = 1, --default:0
  num_multi_echo_laser_scans = 0, --default:1
  num_subdivisions_per_laser_scan = 10,
  num_point_clouds = 0,
  lookup_transform_timeout_sec = 0.2, --default:0.2
  submap_publish_period_sec = 0.3,
  pose_publish_period_sec = 5e-3,
  trajectory_publish_period_sec = 30e-3,
  rangefinder_sampling_ratio = 1.,
  odometry_sampling_ratio = 1.,
  fixed_frame_pose_sampling_ratio = 1.,
  imu_sampling_ratio = 1.,
  landmarks_sampling_ratio = 1.,
}

MAP_BUILDER.use_trajectory_builder_2d = true
TRAJECTORY_BUILDER_2D.num_accumulated_range_data = 10 --default:10,72
--new params add
TRAJECTORY_BUILDER_2D.min_range = 0.20 --lidar:ydlidar-x4
TRAJECTORY_BUILDER_2D.max_range = 16.0 --lidar:ydlidar-x4
TRAJECTORY_BUILDER_2D.submaps.num_range_data = 50
TRAJECTORY_BUILDER_2D.use_imu_data = true
TRAJECTORY_BUILDER_2D.imu_gravity_time_constant = 10.0
TRAJECTORY_BUILDER_2D.use_online_correlative_scan_matching = false
TRAJECTORY_BUILDER_2D.ceres_scan_matcher.translation_weight = 10
TRAJECTORY_BUILDER_2D.ceres_scan_matcher.rotation_weight = 40
POSE_GRAPH.constraint_builder.max_constraint_distance = 4.

return options
```

(图 21) 我们 miiboo 机器人的建图配置文件 miiboo_mapbuild.lua

然后需要配置*.launch 文件，我们给 miiboo 机器人建立启动文件取名叫 miiboo_mapbuild.launch，存放路径在 cartographer_ros/launch/里面，具体内容如图 22。不难发现 launch 文件中包含三个 node 启动项，即 urdf 模型启动项、cartographer_node 启动项、cartographer_occupancy_grid_node 启动项。第一个启动项是启动 urdf 模型，这个接口是提供给那些只使用 cartographer 单独建图的应用场景，由于我们 miiboo 机器人建立完地图后还需要继续进行自动导航任务，所以我们使用 miiboo 底盘提供的 urdf 模型，而不使用这里的 urdf 模型，所以这个启动项被注释掉了，这样建图和导航就更容易管理。第二个启动项是启动 cartographer_node 建图节点，这个是 SLAM 建图主节点，我们建立的配置 miiboo_mapbuild.lua 将在这里被载入，同时这里可以对建图输入数据 scan、imu、odom 的 topic 名称做重映射。第三个启动项是启动 cartographer_occupancy_grid_node 地图格式转换节点，由于 cartographer_node 建图节点提供的地图是 submapList 格式的，需要转换成 GridMap 格式才

能在 ROS 中显示和使用。这里面有两个可配参数，`resolution` 用来设置 GridMap 地图的分辨率，`publish_period_sec` 用来设置 GridMap 地图发布的频率。

```
miiboo_mapbuild.launch X
<launch>
  <!-- param name="robot_description"
    textfile="$(find cartographer_ros)/urdf/backpack_2d.urdf" /-->
  <!-- node name="robot_state_publisher" pkg="robot_state_publisher"
    type="robot_state_publisher" /-->

  <node name="cartographer_node" pkg="cartographer_ros"
    type="cartographer_node" args="
      -configuration_directory $(find cartographer_ros)/configuration_files
      -configuration_basename miiboo_mapbuild.lua"
    output="screen">
    <remap from="scan" to="/scan" />
    <remap from="imu" to="/imu" />
    <remap from="odom" to="/odom" />
  </node>

  <node name="cartographer_occupancy_grid_node" pkg="cartographer_ros"
    type="cartographer_occupancy_grid_node" args="
      -resolution 0.05
      -publish_period_sec 1.0" />
</launch>
```

(图 22) 我们 miiboo 机器人的建图启动文件 `miiboo_mapbuild.launch` 配置参数修改好后，不要忘了再编译一次整个 `catkin_ws_carto` 工作空间，切换到 `catkin_ws_carto` 目录，执行下面的编译命令。

```
catkin_make_isolated --install --use-ninja
```

启动 `cartographer_ros` 建图：

要在 miiboo 机器人上，启动 `cartographer_ros` 建图，分为这几个步骤：启动机器人上的各个传感器、启动 `cartographer_ros`、在 PC 端启动键盘控制机器人运动并启动 `rviz` 观察地图（或者在 Android 手机端用 miiboo 机器人 APP 控制机器人运动和观察地图）。

首先，启动机器人上的各个传感器，为了操作方便，我已经将要启动的传感器都写入 `miiboo_bringup/launch/miiboo_all_sensor.launch` 这个启动文件了，文件内容如图 23。这个启动文件包含机器人 `urdf` 启动项、miiboo 底盘启动项、激光雷达启动项、IMU 启动项、摄像头启动项、广播 IP 启动项。

```
miiboo_all_sensor.launch X
<launch>
  <!-- robot model -->
  <include file="$(find miiboo_description)/launch/miiboo_description.launch"/>

  <!-- miiboo bring up -->
  <include file="$(find miiboo_bringup)/launch/minimal.launch"/>

  <!-- launch laser -->
  <include file="$(find ydlidar)/launch/my_x4.launch" />

  <!-- launch imu -->
  <include file="$(find miiboo_imu)/launch/imu.launch" />

  <!-- launch usb cam -->
  <include file="$(find usb_cam)/launch/usb_cam.launch" />

  <!-- launch broadcast ip -->
  <include file="$(find broadcast_ip)/launch/broadcast_udp.launch" />
</launch>
```

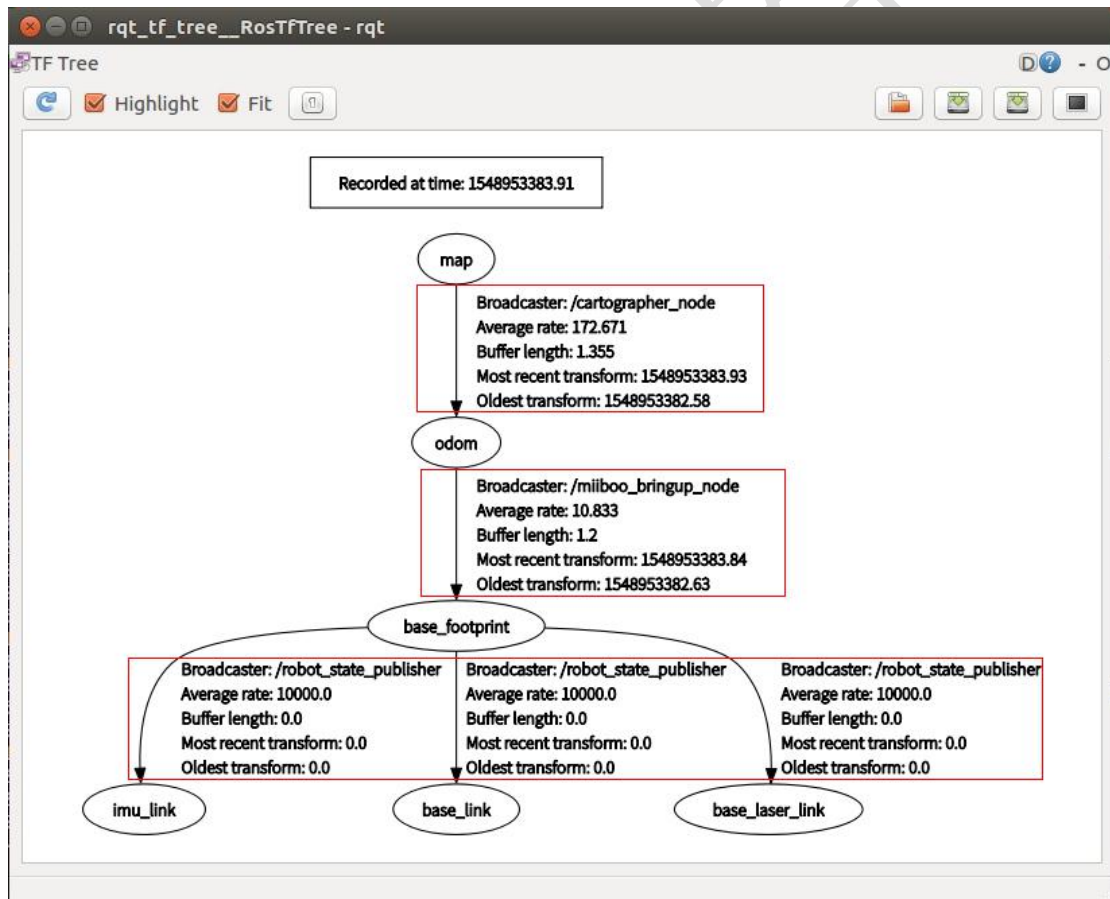
(图 23) 各个传感器启动文件 `miiboo_all_sensor.launch` 打开终端，通过下面的命令直接启动就行了。

```
source ~/catkin_ws/devel/setup.bash
roslaunch miiboo_bringup miiboo_all_sensor.launch
```

然后，启动 `cartographer_ros`，由于前面已经做好了相应的配置，所以直接使用命令启动就行了。

```
source ~/catkin_ws_carto/install_isolated/setup.bash
roslaunch cartographer_ros miiboo_mapbuild.launch
```

这里给个小提示，为了查看 `cartographer_ros` 建图算法有没有正常开始工作，我们可以用 `roslaunch rqt_tf_tree rqt_tf_tree` 查看整个 `tf` 树的结构，正常的 `tf` 树如图 24。`map`->`odom` 之间的关系由 `cartographer` 建图节点提供，`odom`->`base_footprint` 之间的关系由 `miiboo` 底盘的轮式里程计提供，`base_footprint`->`imu_link` 和 `base_link` 和 `base_laser_link` 之间的关系由 `miiboo` 机器人的 `urdf` 模型提供。从 `tf` 树不难看出整个建图过程中机器人定位的实现原理，`cartographer` 建图节点通过维护 `map`->`odom` 之间的关系最终实现全局定位，`miiboo` 底盘的轮式里程计通过维护 `odom`->`base_footprint` 之间的关系来实现局部定位，传感器之间的安装关系由 `urdf` 模型提供，这个静态关系主要用于多传感器数据融合。

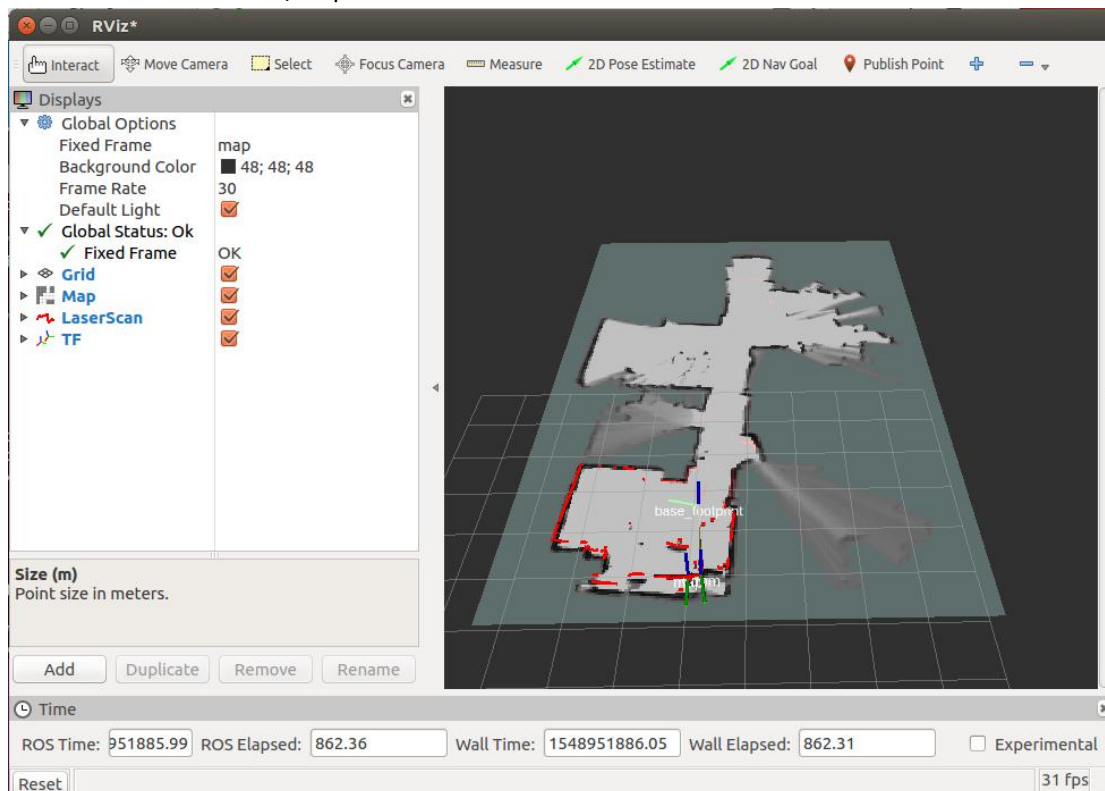


(图 24) `cartographer` 运行时正常的 `tf` 树

最后，在 PC 端启动键盘控制机器人运动并启动 `rviz` 观察地图(或者在 Android 手机端用 `miiboo` 机器人 APP 控制机器人运动和观察地图)。如果用 PC 端控制和观察，启动命令如下。在 PC 端打开一个新终端，运行 `rviz` 启动命令。

```
roslaunch rviz rviz
```


在 rviz 窗口中添加订阅/map，就可以看到建图效果了，如图 25。



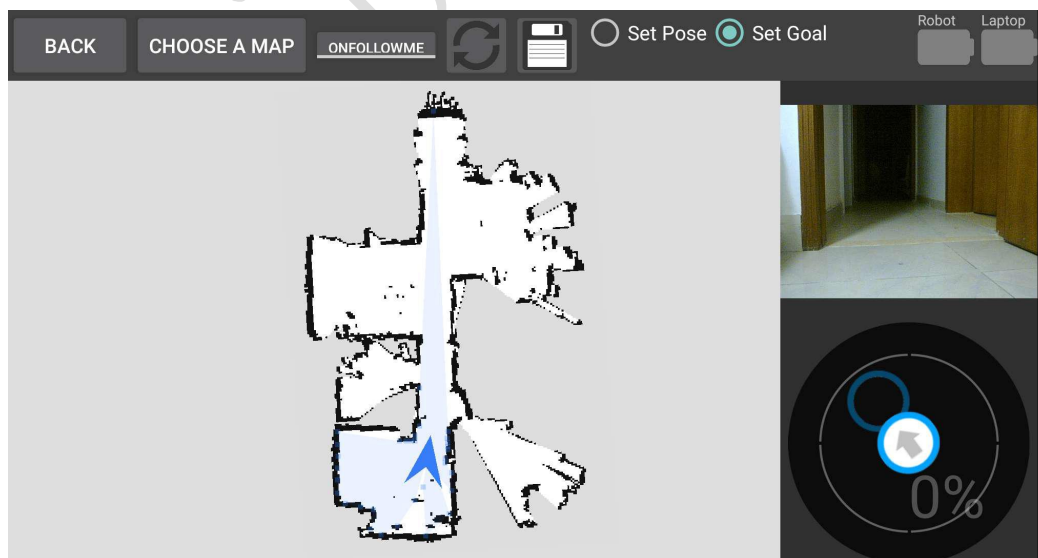
（图 25）在 PC 端用 rviz 观察地图

在 PC 端再打开一个新终端，运行键盘控制启动命令。

```
rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

在该终端下，用键盘就可以控制机器人前进、后退、左转、右转了。

如果是在 Android 手机端用 miiboo 机器人 APP 控制机器人运动和观察地图，直接就能使用，如图 26。



（图 26）在 Android 手机端用 miiboo 机器人 APP 控制机器人运动和观察地图
保存 cartographer_ros 建图结果：

当我们在房间里面扫描一圈，地图建立的差不多了，就可以将建图结果保存下来了，

cartographer_ros 提供了将建图结果保存为*.pbstream 专门的方法，其实就是一条命令。

```
source ~/catkin_ws_carto/install_isolated/setup.bash
rosservice call /write_state /home/ubuntu/map/carto_map.pbstream
```

其实就是调用 cartographer_ros 提供的叫/write_state 这个名字的服务，服务传入参数 /home/ubuntu/map/carto_map.pbstream 为地图的保存路径。保存成功后，会返回相应的状态信息，如图 27。

```
ubuntu@ubuntu-desktop:~$ rosservice call /write_state /home/ubuntu/map/carto_map.pbstream
status:
  code: 0
  message: "State written to '/home/ubuntu/map/carto_map.pbstream'."
ubuntu@ubuntu-desktop:~$
```

（图 27）调用/write_state 服务保存建图结果

地图格式转换：

由于用 cartographer_ros 提供的/write_state 方法保存的地图是*.pbstream 的格式，而要在后续的自主导航中使用这个地图，我们需要将其转换为 ROS 中通用的 GridMap 格式。其实很简单，cartographer_ros 已经跟我们提供了 cartographer_pbstream_to_ros_map 这个节点用于转换的实现。所以，我们只需要写一个启动文件启动这个节点就行了，我给这个启动文件取名 miiboo_pbstream2rosmmap.launch，存放路径是 cartographer_ros/launch/，启动文件的内容如图 28。在使用这个启动文件进行启动时，需要从外部传入两个参数，参数 pbstream_filename 为待转换的*.pbstream 文件路径，参数 map_filestem 为转换后存放结果的文件路径。

```
miiboo_pbstream2rosmmap.launch x
<launch>
  <node name="cartographer_pbstream_to_ros_map_node" pkg="cartographer_ros"
    type="cartographer_pbstream_to_ros_map" args="
      -pbstream_filename $(arg pbstream_filename)
      -map_filestem $(arg map_filestem)"
    output="screen">
  </node>
</launch>
```

（图 28）pbstream 转 GridMap 启动文件

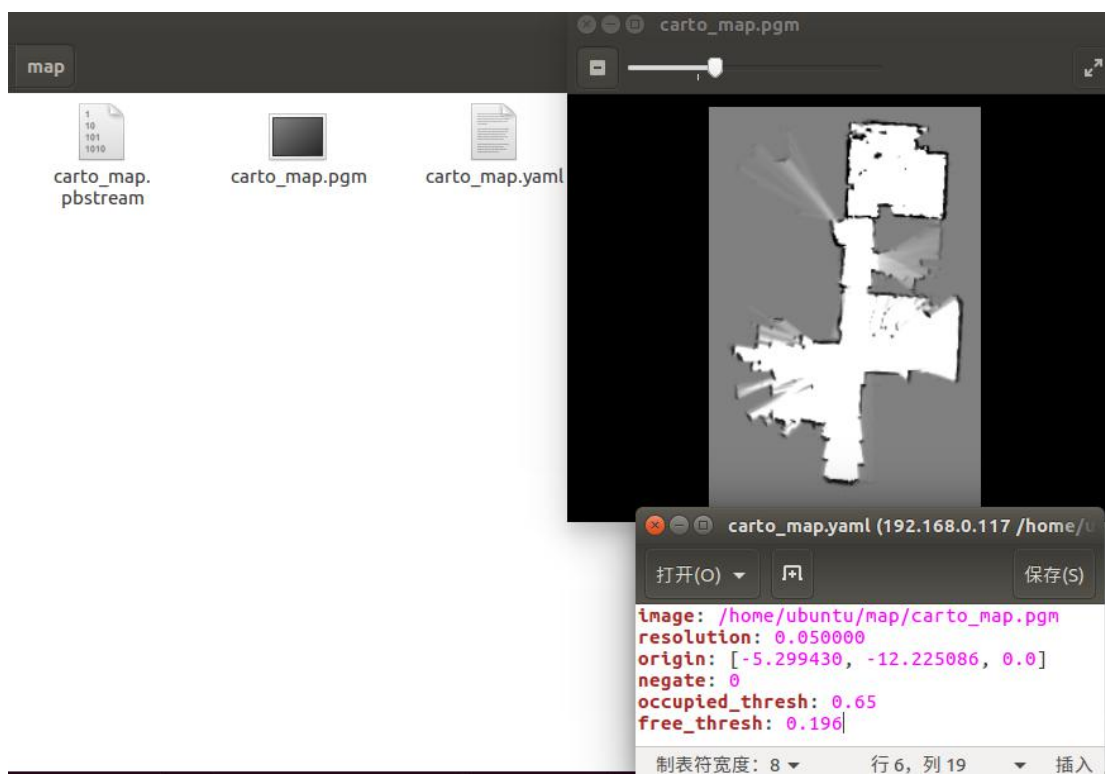
配置参数修改好后，不要忘了再编译一次整个 catkin_ws_carto 工作空间，切换到 catkin_ws_carto 目录，执行下面的编译命令。

```
catkin_make_isolated --install --use-ninja
```

最后，就可以打开终端，使用启动这个启动文件，对地图格式进行转换了，命令如下。

```
roslaunch cartographer_ros miiboo_pbstream2rosmmap.launch
pbstream_filename:=/home/ubuntu/map/carto_map.pbstream
map_filestem:=/home/ubuntu/map/carto_map
```

保存结束后，节点会自动退出，这时我们可以得到转换后的地图，转换后的 GridMap 地图由*.pgm 和*.yaml 两部分构成，这时标准的 ROS 格式地图，可以被 ROS 导航框架中的 map_server 节点直接调用，转换后的地图结果如图 29。



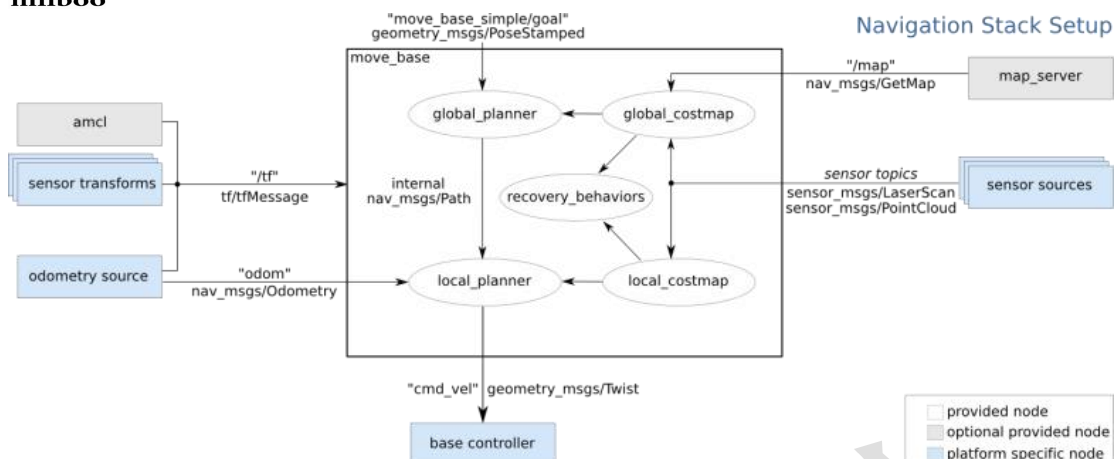
(图 29) 地图格式转换后的结果

3.ros-navigation 机器人自主避障导航

前面的学习教程打好了必须的基础，现在就正式开始探讨 ROS 系统最强大的特性之一，让我们的机器人能自主导航和避障。这都得益于开源社区和共享代码，使 ROS 拥有大量可用的导航算法。将这些导航算法集大成者，便是 ros-navigation 导航功能包集。了解更多 ros-navigation 的信息，请参考官方 wiki 教程：<http://wiki.ros.org/navigation/>。

3.1.机器人自主避障导航原理分析

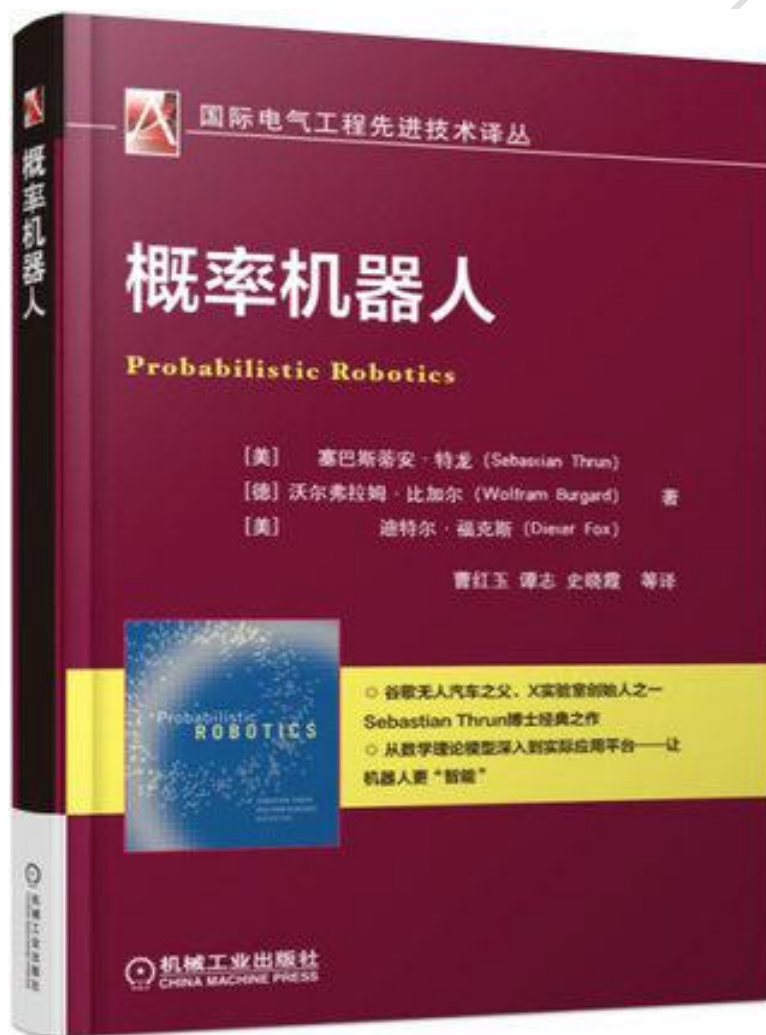
要分析导航功能包的原理，肯定需要先参考下面这张 ros-navigation 官方给出的系统框图，如图 30。最中心的是 move_base 节点，是导航过程运动控制的最终执行机构，move_base 订阅用户发布的导航目标 move_base_simple/goal，并将实时运动控制信号 cmd_vel 下发给底盘以实现最终运动控制，move_base 中的各种导航算法模块都是以插件的形式进行调用的，这样可以很方便的替换不同的算法以适应不同的应用，其中 global_planner 用于全局路径规划、local_planner 用于局部路径规划、global_costmap 是全局代价地图用于描述全局环境信息、local_costmap 是局部代价地图用于描述局部环境信息、recovery_behaviors 是恢复策略用于机器人碰到障碍后自动进行逃离恢复。然后是 amcl 节点，amcl 节点利用粒子滤波算法实现机器人的全局定位，为机器人导航提供全局位置信息。再然后是 map_server 节点，map_server 节点通过调用前面 SLAM 建图得到的地图为导航提供环境地图信息。最后就是要提供机器人模型相关的 tf 信息、里程计 odom 信息、激光雷达信息 scan。



(图 30) ros-navigation 导航功能包集系统框图

机器人全局定位 amcl 粒子滤波算法:

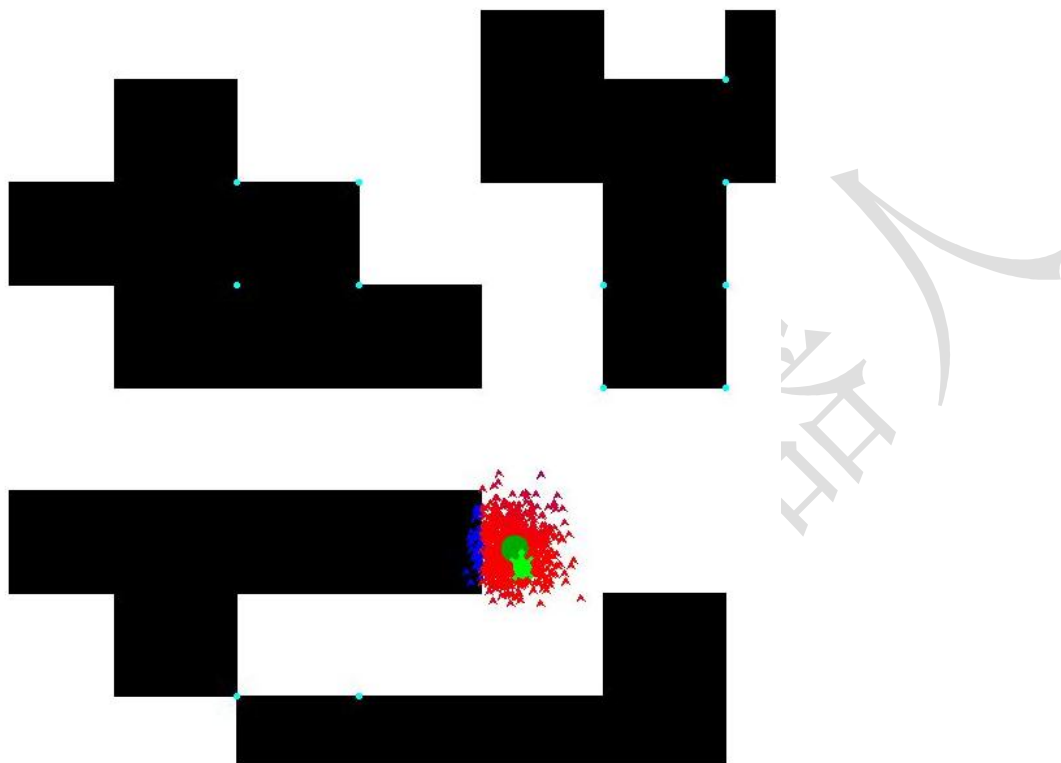
amcl 的全称是自适应蒙特卡洛粒子滤波，这里通过讲解粒子滤波、重要性采样、机器人绑定、自适应蒙特卡洛这几个概念来说明机器人全局定位的原理。由于 amcl 的数学理论比较复杂，限于篇幅这里不展开讲解，感兴趣的朋友可以参考《Probabilistic Robotics》这本书，里面有详细的推导过程，如图 31。



(图 31) 概率机器人

粒子滤波，是一种思想，比如要计算一个矩形里面一个不规则形状的面积，这个问题不好直

接计算，但是可以拿一把豆子均匀撒到矩形中，统计落在不规则形状中豆子的占比就能算出其面积了。在机器人定位问题中，我们在地图的任意位置撒上许多粒子点，然后通过传感器观测数据按照一定的评价方法对每个粒子点进行打分，评分高的粒子点表示机器人有更大的可能在此位置；在下一轮撒点时，就在评分高的粒子点附近多撒一些点，这样通过不断的迭代，粒子点就会聚拢到一个地方。这个粒子点聚集的地方，就是机器人位置的最优估计点。如图 32，红色的粒子点慢慢聚拢到一团。



（图 32）粒子滤波

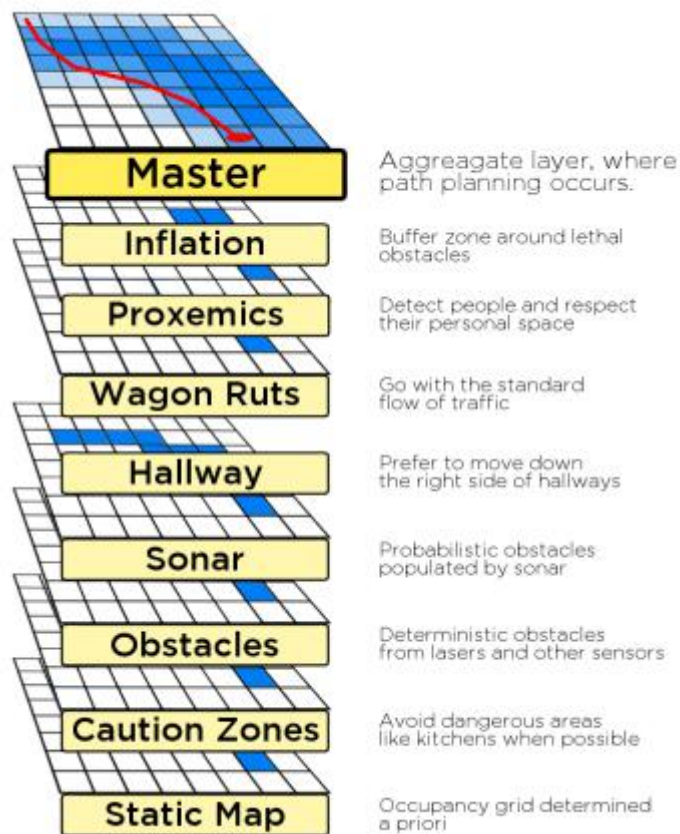
重要性采样，在粒子滤波的迭代过程中，评分高的粒子点会被下一轮迭代时更加看重，这样不断迭代真实估计值附近的粒子点会越来越多。

机器人绑架，当机器人被突然从一个地方抱走到另一个地方，这个时候前一轮迭代得到的粒子点完全不能在新的位置上试用，这样继续迭代下去就会发生位置估计的错误。

自适应蒙特卡洛，自适应主要体现在两个方面。通过判断粒子点的平均分突变来识别机器人绑架问题，并在全局重新撒点来解决机器人绑架问题；通过判断粒子点的聚集程度来确定位置估计是否准确，在估计比较准确的时候降低需要维护的粒子点数目，这样来降低算法的计算开销。

代价地图 costmap:

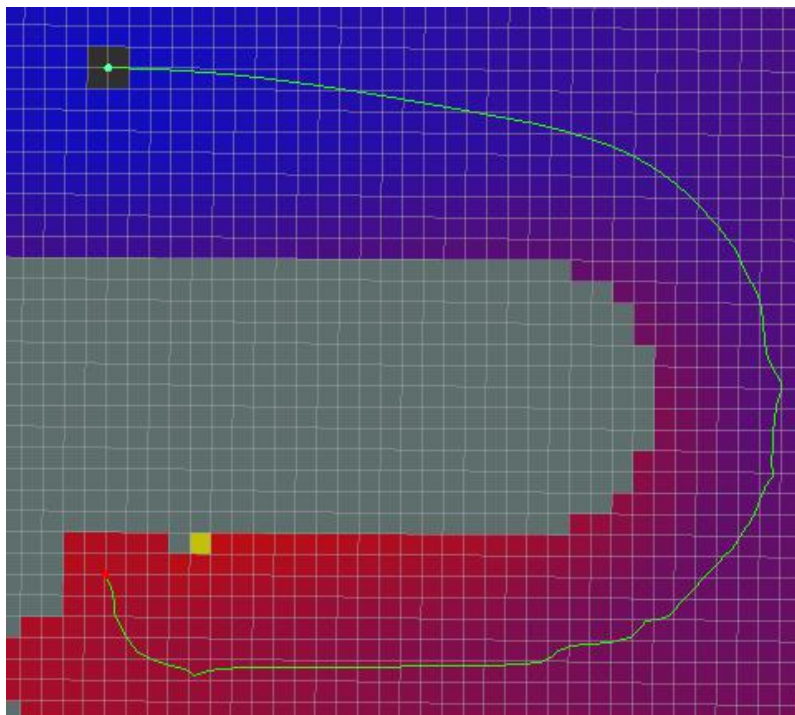
代价地图用于描述环境中的障碍物信息，代价地图利用激光雷达、声呐、红外测距等探测传感器的数据来生成，大致的原理是通过建立不同的图层 Layer 然后叠在一起，被填充的栅格点表示有障碍物，如图 33，想要了解代价地图的更多细节可以阅读《Layered Costmaps for Context-Sensitive Navigation》这篇论文。机器人导航中用到了两个代价地图，全局代价地图 global_costmap 和局部代价地图 local_costmap。



(图 33) 代价地图

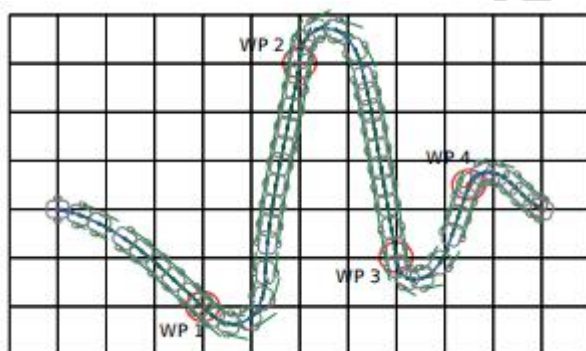
路径规划:

机器人导航中核心的就是路径规划, 路径规划是通过利用环境障碍物信息找到一条到达目标并且开销小的路劲。导航中会用到两种路径规划, 即全局路径规划 `global_planner` 和局部路径规划 `local_planner`。全局路径规划更像是一种战略性策略, 需要考虑全局, 规划处一条尽量短并且易于执行的路径。在全局路径的指导下, 机器人在实际行走时还需要考虑周围实时的障碍物并制定避让策略, 这就是局部路径规划要完成的事, 可以说机器人的自主导航最终是由局部路径规划一步步完成的。ROS 中推荐的全局路径规划器是 `global_planner`, 基于 A* 算法, 如图 34。



(图 34) 基于 A* 算法的全局路径规划

由于 ROS 中的局部路径规划器比较老了，所以我推荐大家使用比较新的局部路径规划器 `teb_local_planner`。`teb_local_planner` 是基于弹性时间带碰撞约束的算法，算法将动态障碍物、运行时效、路径平滑性等约束做综合考虑，在复杂环境下有更优秀的表现，如图 35。



(图 35) 基于弹性时间带碰撞算法的局部路径规划

3.2.ros-navigation 安装

`ros-navigation` 的安装有两种方法，方法一是直接通过 `apt-get` 安装编译好的 `ros-navigation` 库到系统中，方法二是下载 `ros-navigation` 源码手动编译安装。由于后续可能需要对 `ros-navigation` 中的算法做修改和改进，所以我采用方法二进行安装。

这里将建立一个叫 `catkin_ws_nav` 的 ROS 工作空间，专门用于存放机器人导航相关的功能包。关于创建 ROS 工作空间的操作，请参考前面相应部分内容，这里就不做讲解。

安装命令很简单，执行下面的命令编译安装就行了。

首先前往 <https://github.com/ros-planning/navigation>，将分支切换到 `kinetic-devel` 然后下载源码到本地，将 `navigation-kinetic-devel.zip` 解压备用。将解压好的 `navigation-kinetic-devel` 文件夹拷贝到 `~/catkin_ws_nav/src/`后，就可编译安装了。


```
#编译过程可能会遇到找不到 Bullet 等依赖问题，解决这些依赖简便办法
#先 apt-get 安装 ros-navigation 包，这样依赖会被自动装上
sudo apt-get install ros-kinetic-navigation*
#再 apt-get 卸载掉 ros-navigation 包
sudo apt-get remove ros-kinetic-navigation ros-kinetic-navigation-experimental

cd ~/catkin_ws_nav/
catkin_make
```

然后需要安装 `teb_local_planner`，前往 https://github.com/rst-tu-dortmund/teb_local_planner，将分支切换到 `kinetic-devel` 然后下载源码到本地，并解压源码包到 `~/catkin_ws_nav/src/` 后，就可以编译安装了。

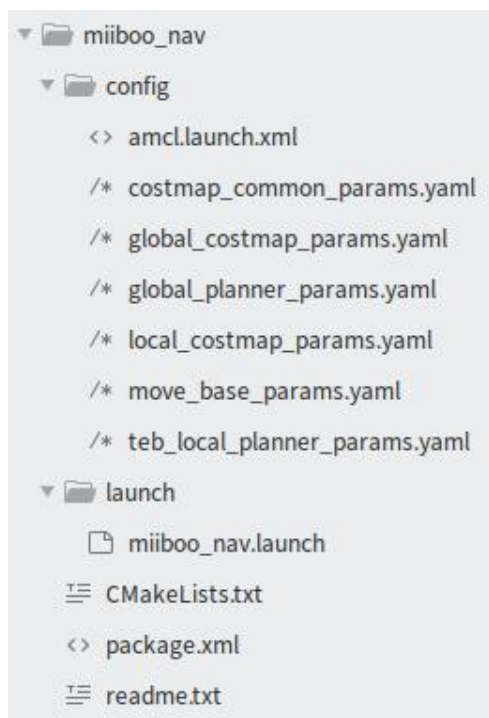
```
#安装依赖
source ~/catkin_ws_nav/devel/setup.bash
rosdep install teb_local_planner

cd ~/catkin_ws_nav/
catkin_make -DCATKIN_WHITELIST_PACKAGES="teb_local_planner"
```

值得一提的是，`teb_local_planner` 源码中关于 `plugin` 的配置文件均已写好，直接编译源码就能完成 `plugin` 的注册及插入，非常方便。

3.3.ros-navigation 使用

`ros-navigation` 功能包集是一个很强大的导航框架，支持几乎所有的移动机器人。我们只需要按照适当的配置就可以将 `ros-navigation` 应用到我们自己的机器人。为了更好的管理配置文件和启动节点，我们需要先为 `miiboo` 机器人建立一个功能包 `miiboo_nav`，用于专门存放我们机器人的导航配置与启动文件。`miiboo_nav` 功能包文件结构，如图 36。不难发现，`miiboo_nav` 是一个不包含任何可执行源码的功能包，里面只包含 `config` 和 `launch`。`config` 中存放导航中各算法模块的参数配置，`launch` 中存放启动导航所需的各种节点的启动文件。下面依次介绍这些文件的作用。



(图 36) miiboo_nav 功能包文件结构

机器人全局定位 AMCL 算法配置 `amcl.launch.xml`:

导航中用到的全局定位 AMCL 算法功能包包含很多可以配置的参数，这里建立配置文件 `amcl.launch.xml` 对这些参数进行配置，内容如图 37。配置参数分为 3 类：粒子滤波参数、雷达模型参数、里程计模型参数。由于参数比较多，关于参数配置的具体讲解就不展开，请直接参考 wiki 官方教程：<http://wiki.ros.org/amcl>。

```

amcl.launch.xml
<launch>
  <arg name="initial_pose_x" default="0.0"/>
  <arg name="initial_pose_y" default="0.0"/>
  <arg name="initial_pose_a" default="0.0"/>
  <node pkg="amcl" type="amcl" name="amcl" output="screen">
    <!--Overall filter parameters-->
    <param name="min_particles" value="2000"/><!--default:100-->
    <param name="max_particles" value="5000"/><!--default:5000-->
    <param name="kld_err" value="0.05"/><!--default:0.01-->
    <param name="kld_z" value="0.99"/><!--default:0.99-->
    <param name="update_min_d" value="0.25"/><!--default:0.2-->
    <param name="update_min_a" value="0.2"/><!--default:π/6.0-->
    <param name="resample_interval" value="1"/><!--default:2-->
    <param name="transform_tolerance" value="2.0"/><!--default:0.1-->
    <param name="recovery_alpha_slow" value="0.0"/><!--default:0.0-->
    <param name="recovery_alpha_fast" value="0.0"/><!--default:0.0-->
    <param name="initial_pose_x" value="$(arg initial_pose_x)"/>
    <param name="initial_pose_y" value="$(arg initial_pose_y)"/>
    <param name="initial_pose_a" value="$(arg initial_pose_a)"/>
    <!--param name="initial_cov_xx" value="0.5*0.5"/>
    <!--param name="initial_cov_yy" value="0.5*0.5"/>
    <!--param name="initial_cov_aa" value="(π/12)*(π/12)"/>
    <param name="gui_publish_rate" value="10.0"/><!--default:-1.0-->
    <param name="save_pose_rate" value="0.5"/><!--default:0.5-->
    <param name="use_map_topic" value="false"/><!--default:false-->
    <param name="first_map_only" value="false"/><!--default:false-->

    <!--Laser model parameters-->
    <param name="laser_min_range" value="-1.0"/><!--default:-1.0-->
    <param name="laser_max_range" value="-1.0"/><!--default:-1.0-->
    <param name="laser_max_beams" value="60"/><!--default:30-->
    <param name="laser_z_hit" value="0.5"/><!--default:0.95-->
    <param name="laser_z_short" value="0.05"/><!--default:0.1-->
    <param name="laser_z_max" value="0.05"/><!--default:0.05-->
    <param name="laser_z_rand" value="0.5"/><!--default:0.05-->
    <param name="laser_sigma_hit" value="0.2"/><!--default:0.2-->
    <param name="laser_lambda_short" value="0.1"/><!--default:0.1-->
    <param name="laser_likelihood_max_dist" value="2.0"/><!--default:2.0-->
    <param name="laser_model_type" value="likelihood_field"/><!--default:likelihood_field-->

    <!--Odometry model parameters-->
    <param name="odom_model_type" value="diff"/><!--default:diff-->
    <param name="odom_alpha1" value="0.2"/><!--default:0.2-->
    <param name="odom_alpha2" value="0.2"/><!--default:0.2-->
    <param name="odom_alpha3" value="0.2"/><!--default:0.2-->
    <param name="odom_alpha4" value="0.2"/><!--default:0.2-->
    <!--param name="odom_alpha5" value="0.2"/><!--only used if model is "omni"-->
    <param name="odom_frame_id" value="odom"/><!--default:odom-->
    <param name="base_frame_id" value="base_footprint"/><!--default:base_link-->
    <param name="global_frame_id" value="map"/><!--default:map-->
    <param name="tf_broadcast" value="true"/><!--default:true-->

    <remap from="scan" to="/scan"/>
  </node>
</launch>

```

(图 37) 全局定位 AMCL 算法配置 amcl.launch.xml

代价地图公有参数配置 costmap_common_params.yaml:

全局代价地图和局部代价地图公有的参数被放置在 costmap_common_params.yaml 这里面，这样只需要载入这个配置文件就能完成共有参数的配置，避免了在全局代价地图和局部代价地图配置文件中重复配置一些相同参数的冗余，内容如图 38。配置参数分为 2 类：机器人形状和代价地图各 Layer 图层。机器人形状可以用多边形或圆形描述，我们的 miiboo 机器人是矩形的，所以选多边形描述。代价地图各 Layer 图层包括：静态层 static_layer（由 SLAM 建立得到的地图提供数据）、障碍层 obstacle_layer（由激光雷达等障碍扫描传感器提供实时数据）、全局膨胀层 global_inflation_layer（为全局代价地图提供膨胀效果）、局部膨胀层 global_inflation_layer（为局部代价地图提供膨胀效果）。关于参数配置的具体讲解就不展开，请直接参考 wiki 官方教程：http://wiki.ros.org/costmap_2d。

```
costmap_common_params.yaml x
#robot footprint shape
footprint: [[0.2, 0.11], [-0.1, 0.11], [-0.1, -0.11], [0.2, -0.11]]
#robot_radius: 0.22

#plugins layers list
static_layer:
  enabled: true
  unknown_cost_value: -1
  lethal_cost_threshold: 100
  map_topic: /map
  first_map_only: false
  subscribe_to_updates: true #default:false
  track_unknown_space: true
  use_maximum: false
  trinary_costmap: true

obstacle_layer:
  enabled: true
  #Sensor management parameters
  observation_sources: laser_scan_sensor #point_cloud_sensor
  laser_scan_sensor:
    topic: /scan
    sensor_frame: /base_laser_link
    #observation_persistence: 0.0
    #expected_update_rate: 0.0
    data_type: LaserScan #alternatives: LaserScan, PointCloud, PointCloud2
    clearing: true #true, modify by cabin in 03.02
    marking: true #true, modify by cabin in 03.02
    #max_obstacle_height: 0.35 #2.0
    #min_obstacle_height: 0.25 #0.0
    #obstacle_range: 2.5
    #raytrace_range: 3.0
    #inf is valid: false
  #Global Filtering Parameters
  #max_obstacle_height: 0.6 #2.0
  obstacle_range: 2.0 #2.5
  raytrace_range: 3.0 #3.0

  #ObstacleCostmapPlugin
  track_unknown_space: true #false
  #footprint_clearing_enabled: true

  #VoxelCostmapPlugin
  #origin z: 0.0
  #z_resolution: 0.2
  #z_voxels: 10
  #unknown_threshold: 10
  #mark threshold: 0
  #publish_voxel_map: false
  #footprint_clearing_enabled: true

global_inflation_layer:
  enabled: true
  inflation_radius: 1.0 #0.15
  cost_scaling_factor: 2.0 #10.0

local_inflation_layer:
  enabled: true
  inflation_radius: 0.15 #0.15
  cost_scaling_factor: 5.0 #10.0

#sonar layer:
# topics: ["/sonar/s1", "/sonar/s2", "/sonar/s3", "/sonar/s4"]
# no_readings_timeout: 0.0
# clear_threshold: 0.2 #0.4
# mark_threshold: 0.5 #0.5
```

(图 38) 代价地图共有参数配置 costmap_common_params.yaml

全局代价地图参数配置 global_costmap_params.yaml:

全局代价地图是以插件的形式组建的，在 costmap_common_params.yaml 中定义的各 Layer 图层都可以通过插件的形式放入全局代价地图中，各 Layer 图层的组合可以根据需求自由选择，如图 39。我们的 miiboo 机器人的全局代价地图只用了 static_layer 和 global_inflation_layer 两个图层。

```
global_costmap_params.yaml X
global_costmap:
  #Coordinate frame and tf parameters
  global_frame: /map #default:/map
  robot_base_frame: /base_footprint #default:/base_link
  transform_tolerance: 2.0 #default:0.2

  #Rate parameters
  update_frequency: 1.0 #default:5.0
  publish_frequency: 0.0 #default:0.0

  #map params
  static_map: true #default:false
  rolling_window: false

  plugins:
    - {name: static_layer, type: "costmap_2d::StaticLayer"}
    #- {name: sonar_layer, type: "range_sensor_layer::RangeSensorLayer"}
    #- {name: obstacle_layer, type: "costmap_2d::ObstacleLayer"}
    - {name: global_inflation_layer, type: "costmap_2d::InflationLayer"}
```

(图 39) 全局代价地图参数配置 global_costmap_params.yaml

局部代价地图参数配置 local_costmap_params.yaml:

局部代价地图和全局代价地图类似，就不展开讲解了，如图 40。我们的 miiboo 机器人的局部代价地图只用了 obstacle_layer 和 local_inflation_layer 两个图层。

```
local_costmap_params.yaml X
local_costmap:
  #Coordinate frame and tf parameters
  global_frame: /odom #default:/odom
  robot_base_frame: /base_footprint #default:/base_link
  transform_tolerance: 2.0 #default:0.2

  #Rate parameters
  update_frequency: 5.0 #default:5.0
  publish_frequency: 5.0 #default:5.0

  #map params
  static_map: false
  rolling_window: true
  width: 4.0 #default:6.0
  height: 4.0 #default:6.0
  resolution: 0.05 #default:0.05
  #origin_x: 0.0 #default:0.0
  #origin_y: 0.0 #default:0.0

  #robot model
  inscribed_radius: 0.22 #default:0.325
  circumscribed_radius: 0.22 #default:0.46

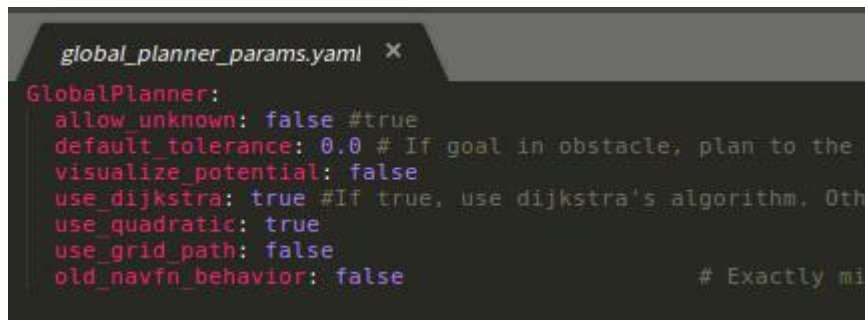
  plugins:
    #- {name: sonar_layer, type: "range_sensor_layer::RangeSensorLayer"}
    - {name: obstacle_layer, type: "costmap_2d::ObstacleLayer"}
    - {name: local_inflation_layer, type: "costmap_2d::InflationLayer"}
```

(图 40) 局部代价地图参数配置 local_costmap_params.yaml

全局路径规划器参数配置 global_planner_params.yaml:

全局路径规划器也是以插件的形式被使用，这里主要对全局路径规划器 GlobalPlanner 插件及参数进行申明，在后面的 move_base_params.yaml 中将会调用这个插件。配置文件内容如图 41。关于参数配置的具体讲解就不展开，请直接参考 wiki 官方教程：

http://wiki.ros.org/global_planner。



(图 41) 全局路径规划器参数配置 global_planner_params.yaml

局部路径规划器参数配置 teb_local_planner_params.yaml:

局部路径规划器也是以插件的形式被使用，这里主要对局部路径规划器 TebLocalPlannerROS 插件及参数进行申明，在后面的 move_base_params.yaml 中将会调用这个插件。配置文件内容如图 42。关于参数配置的具体讲解就不展开，请直接参考 wiki 官方教程：

http://wiki.ros.org/teb_local_planner。

```

teb_local_planner_params.yaml x
TebLocalPlannerROS:

  odom_topic: odom
  map_frame: map #default:odom, map if in case of a static map

  #####Trajectory#####
  teb_autosize: True #default:True
  dt_ref: 0.3 #default:0.3
  dt_hysteresis: 0.1 #default:0.1
  #min_samples: 3 #default:3
  #max_samples: 500 #default:500
  global_plan_overwrite_orientation: True #default:True
  allow_init_with_backwards_motion: False #default:False
  #global_plan_via_point_sep: -1 #default:-1
  #via_points_ordered: False #default:False
  max_global_plan_lookahead_dist: 3.0 #default:3.0
  #exact_arc_length: False #default:False
  #force_reinit_new_goal_dist: 1.0 #default:1.0
  feasibility_check_no_poses: 5 #default:5
  #publish_feedback: False #default:False

  #####Robot#####
  max_vel_x: 0.30 #default:0.4
  max_vel_x_backwards: 0.00 #default:0.05
  max_vel_y: 0.0 #default:0.0
  max_vel_theta: 1.2 #default:1.0
  acc_lim_x: 0.5 #default:0.5
  #acc_lim_y: 0.5 #default:0.5
  acc_lim_theta: 1.0 #default:0.5
  #min_turning_radius: 0.0 #default:0.0,only for carlike robots
  #wheelbase: 1.0 #default:1.0,only required if cmd_angle_instead_rotvel
  #cmd_angle_instead_rotvel: False #default:False
  #is_footprint_dynamic: False #default:False
  footprint_model:
    type: "point"
  #####GoalTolerance#####
  xy_goal_tolerance: 0.2 #default:0.2
  yaw_goal_tolerance: 0.2 #default:0.2
  free_goal_vel: False #default:False

  #####Obstacles#####
  min_obstacle_dist: 0.36 #footprint model/type=point
  inflation_dist: 0.38 #default:0.35
  dynamic_obstacle_inflation_dist: 0.40 #default:0.6
  #include_dynamic_obstacles: True #default:True
  #include_costmap_obstacles: True #default:True
  costmap_obstacles_behind_robot_dist: 1.5 #default:1.5
  obstacle_poses_affected: 30 #default:30
  #legacy_obstacle_association: False #default:False
  #obstacle_association_force_inclusion_factor: 1.5 #default:1.5,use
  #obstacle_association_cutoff_factor: 5 #default:5,used only if le
  costmap_converter_plugin: "" #default:""
  costmap_converter_spin_thread: True #default:True
  costmap_converter_rate: 5.0 #default:5.0

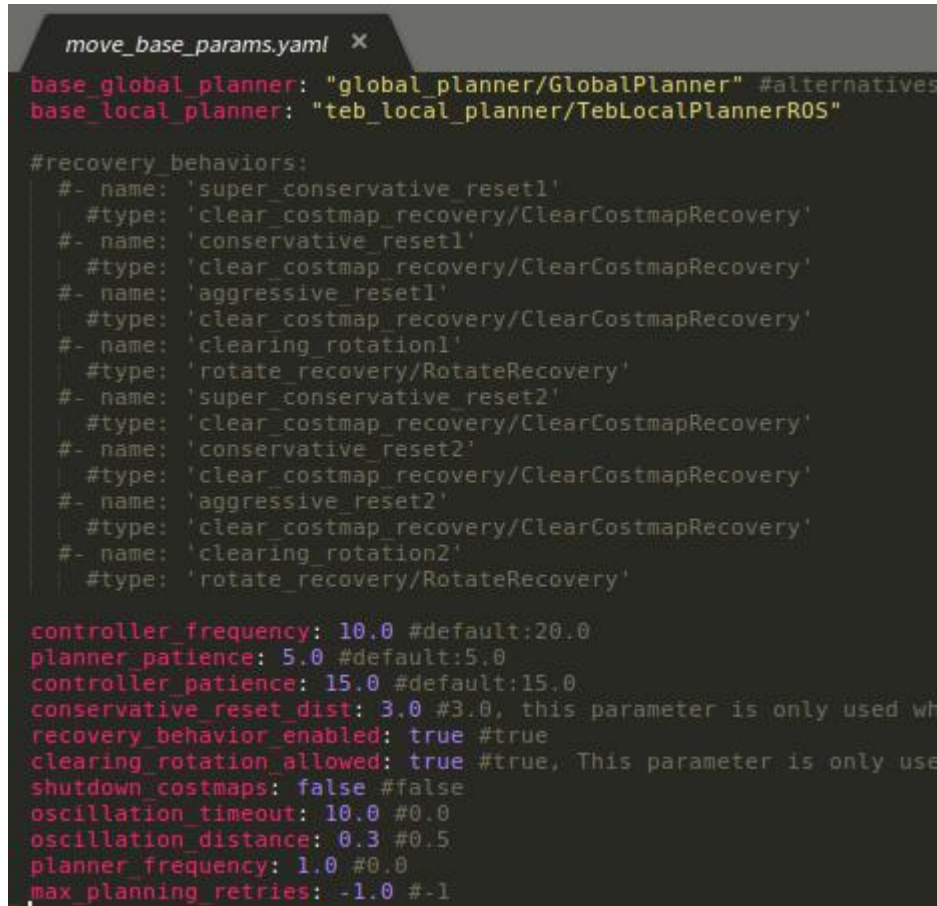
  #####Optimization#####
  no_inner_iterations: 2 #default:5
  no_outer_iterations: 1 #default:4
  optimization_activate: True #default:True
  optimization_verbose: False #default:False
  penalty_epsilon: 0.001 #default:0.1
  weight_max_vel_x: 100 #default:2
  #weight_max_vel_y: 2 #default:2
  weight_max_vel_theta: 100 #default:1
  weight_acc_lim_x: 1 #default:1
  #weight_acc_lim_y: 1 #default:1

```

(图 42) 局部路径规划器参数配置 teb_local_planner_params.yaml

导航核心控制参数配置 move_base_params.yaml:

导航核心控制由 move_base 节点最终实现, move_base 在整个导航框架中处于核心地位, 全局代价地图、局部代价地图、全局路径规划、局部路径规划、恢复策略等等都将在这里得到具体实现的调用。配置内容如图 43。不难发现, 最开始两行便对调用的全局路径规划器和局部路径规划器进行了申明, 然后就是一些运动控制方面的参数。关于参数配置的具体讲解就不展开, 请直接参考 wiki 官方教程: http://wiki.ros.org/move_base。



(图 43) 导航核心控制参数配置 move_base_params.yaml

启动导航所需的各节点的启动文件 miiboo_nav.launch:

在 config 目录中将各个导航模块的参数配置好后, 就可以在一个启动文件中将导航所需的各节点进行启动, 并载入相关配置。启动文件 miiboo_nav.launch 就是来完成这个任务的, 如图 44。不难发现, 第一个启动项是 map_server, 将 SLAM 建图中得到的地图进行载入后发布到 ROS 中供导航中需要的模块使用; 第二个启动项是 amcl, amcl 开始运行后将为导航中的模块提供机器人在地图中的全局位置; 第三个启动项是 move_base, 在这里将载入代价地图公有参数、全局代价地图参数、局部代价地图参数、move_base 参数、全局路径规划参数、局部路径规划参数。

```
miiboo_nav.launch
<launch>
  <!-- Map server -->
  <arg name="map_path" default="/home/ubuntu/map/carto_map.yaml" />
  <node name="map_server" pkg="map_server" type="map_server" args="$(arg map_path)"/>

  <!-- Run AMCL -->
  <arg name="initial_pose_x" default="0.0"/>
  <arg name="initial_pose_y" default="0.0"/>
  <arg name="initial_pose_a" default="0.0"/>
  <include file="$(find miiboo_nav)/config/amcl.launch.xml">
    <arg name="initial_pose_x" value="$(arg initial_pose_x)"/>
    <arg name="initial_pose_y" value="$(arg initial_pose_y)"/>
    <arg name="initial_pose_a" value="$(arg initial_pose_a)"/>
  </include>

  <!-- Run move_base -->
  <node pkg="move_base" type="move_base" respawn="false" name="move_base" output="screen" clear_params="true">
    <rosparam file="$(find miiboo_nav)/config/costmap_common_params.yaml" command="load" ns="global_costmap" />
    <rosparam file="$(find miiboo_nav)/config/costmap_common_params.yaml" command="load" ns="local_costmap" />
    <rosparam file="$(find miiboo_nav)/config/global_costmap_params.yaml" command="load" />
    <rosparam file="$(find miiboo_nav)/config/local_costmap_params.yaml" command="load" />
    <rosparam file="$(find miiboo_nav)/config/move_base_params.yaml" command="load" />
    <rosparam file="$(find miiboo_nav)/config/global_planner_params.yaml" command="load" />
    <rosparam file="$(find miiboo_nav)/config/teb_local_planner_params.yaml" command="load" />
  </node>
</launch>
```

(图 44) 启动导航所需的各种节点的启动文件 miiboo_nav.launch

到这里，我们已经准备好了导航的各个配置文件和启动文件，现在就正式来进行导航。这里需要特别提醒，我们假设已经在 SLAM 建图过程得到了一张比较好的环境地图，并保存到了正确的路径。启动导航分为 3 步：启动机器人上所有传感器、启动导航所需各个节点、发送导航目标点。

首先，启动机器人上所有传感器，打开终端，通过下面的命令直接启动就行了。

```
source ~/catkin_ws/devel/setup.bash
roslaunch miiboo_bringup miiboo_all_sensor.launch
```

然后，启动导航所需各个节点，打开终端，通过下面的命令直接启动就行了。

```
source ~/catkin_ws_nav/devel/setup.bash
roslaunch miiboo_nav miiboo_nav.launch
```

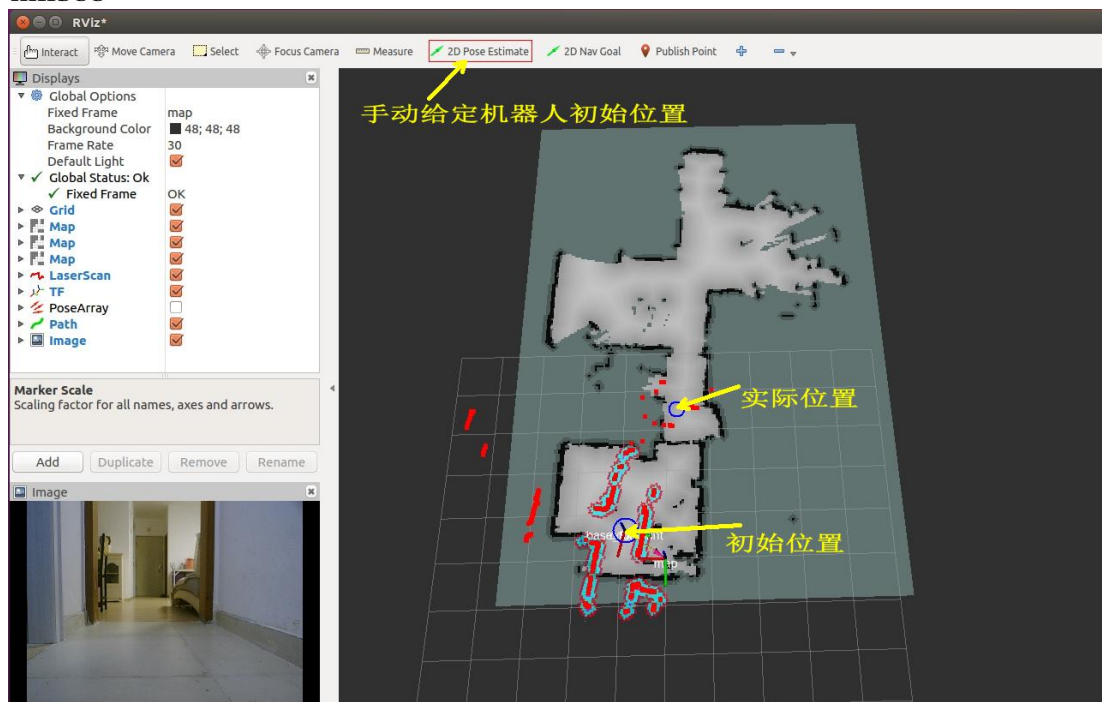
最后，发送导航目标点。这里有两种方式来下发导航目标点，方法一是通过 rviz 交互界面，方法二是通过手机 APP 交互界面。下面对两种方法分别介绍。

方法一：

在 PC 端打开 rviz，打开一个新终端，运行 rviz 启动命令。

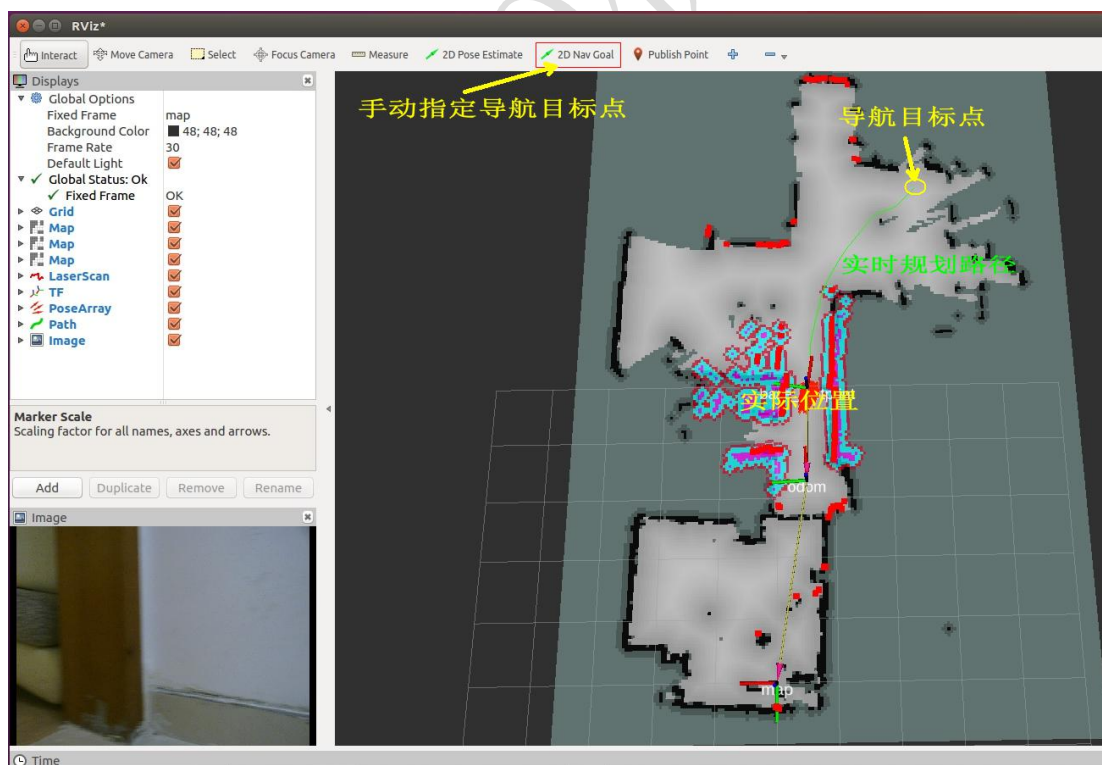
```
roslaunch rviz rviz
```

订阅/map、/scan、/tf 等信息，并观察机器人的初始位置是否正确，如果机器人的初始位置不正确，需要用[2D Pose Estimate]按钮手动给定一个正确的初始位置，如图 45。操作方法很简单，先点击[2D Pose Estimate]按钮，然后将鼠标放置到机器人在地图中实际应该的位置，最后按住鼠标并拖动鼠标来完成机器人朝向的设置。



（图 45）在 rviz 中手动给定机器人初始位置

初始位置设置正确后，就可以用[2D Nav Goal]按钮手动指定导航目标点了，如图 46。操作方法很简单，先点击[2D Nav Goal]按钮，然后将鼠标放置到地图中任意的想让机器人到达的空白位置，最后按住鼠标并拖动鼠标来完成机器人朝向的设置。这样机器人就会开始规划路径并自动导航到该指定目标点。

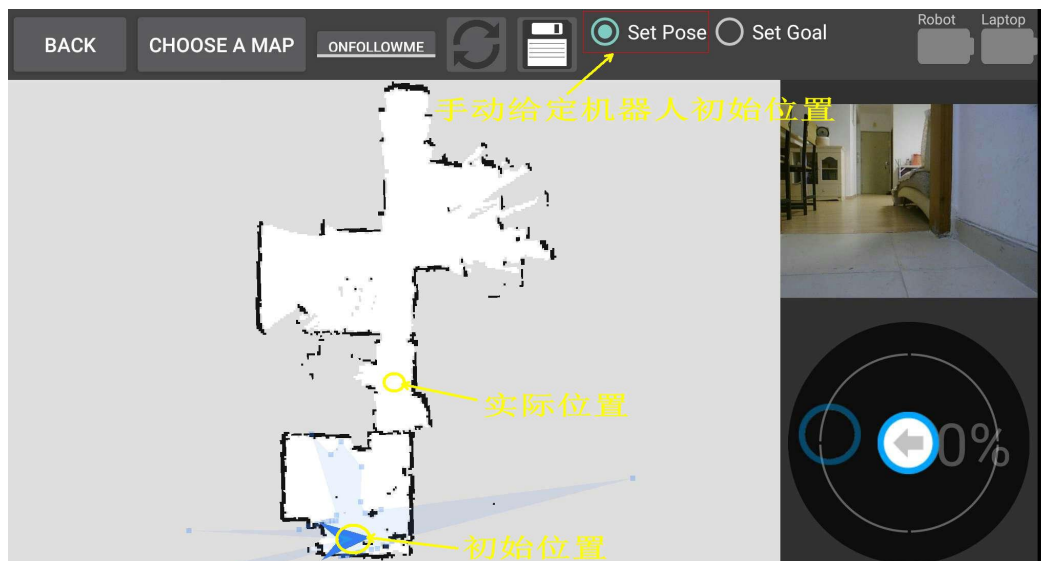


（图 46）在 rviz 中手动指定导航目标点

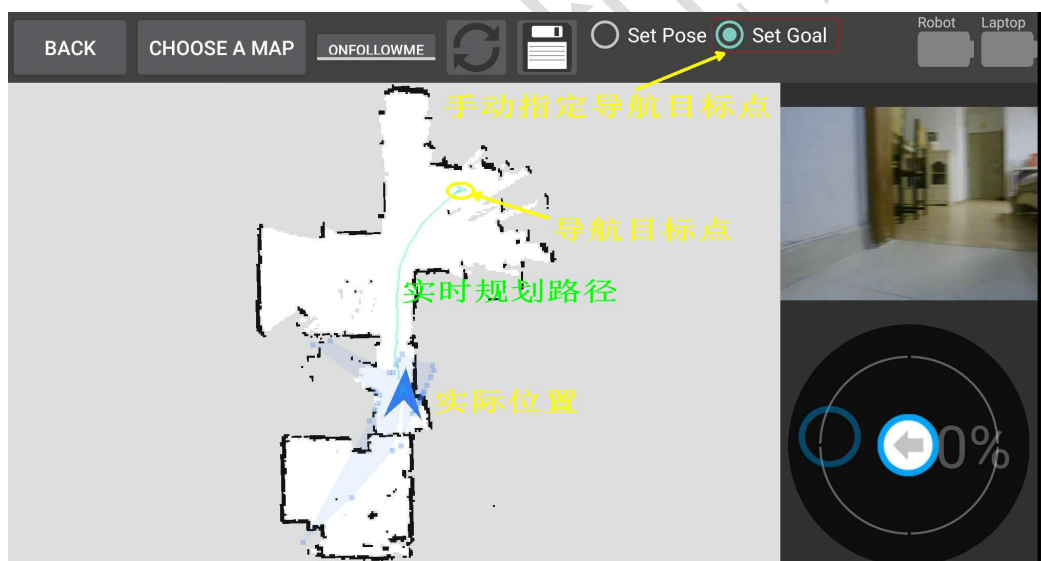
方法二：

直接打开手机上的 miiboo 机器人 APP，就可以看到地图和机器人在地图中的位置了。和方

法一中的一样，需要观察机器人的初始位置是否正确，如果机器人的初始位置不正确，用[Set Pose]按钮手动给定一个正确的初始位置，如图 47。操作过程和方法一类似，就不展开了。



(图 47) 在 miiboo 机器人 APP 中手动给定机器人初始位置
初始位置设置正确后，就可以用[Set Goal]按钮手动指定导航目标点了，如图 48。操作过程和方法一类似，就不展开了。



(图 48) 在 miiboo 机器人 APP 中手动指定导航目标点

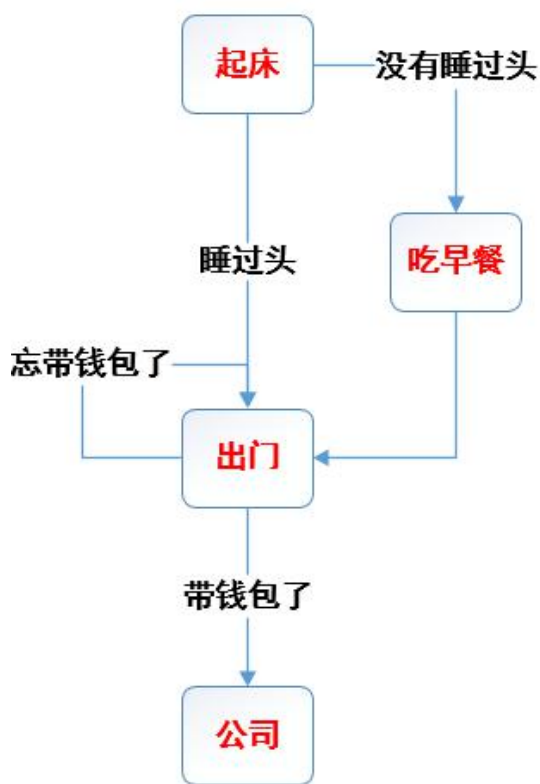
4.多目标点导航及任务调度

通过前面的学习，我们已经可以通过点击地图的方式来命令机器人运动到目标点。其实，ros-navigation 导航框架就是为我们提供了一个最基本的机器人自动导航接口，即单点导航。然而，在实际的机器人应用中，机器人往往要完成复杂的任务，这些复杂的任务都是由一个个基本的任务组合而成的。一般的，机器人通过状态机的形式将一个个基本任务组合在一起来进行复杂任务的调度实现。

4.1.状态机

这里我们只讨论有限状态机,也称为 FSM(Finite State Machine), 其在任意时刻都处于有限状态集合中的某一状态。当其获得一个输入条件时, 将从当前状态转换到另一个状态, 或者仍然保持在当前状态。任何一个 FSM 都可以用状态转换图来描述, 图中的节点表示 FSM 中的

一个状态，有向加权边表示输入条件时状态的变化。如图 49，以一个上班族的生活场景来举例说明状态机的状态转换图。矩形框表示 FSM 中的一个状态，有向边表示在输入条件下的状态转换过程。



(图 49) 有限状态机 FSM 举例

4.2.多目标点巡航

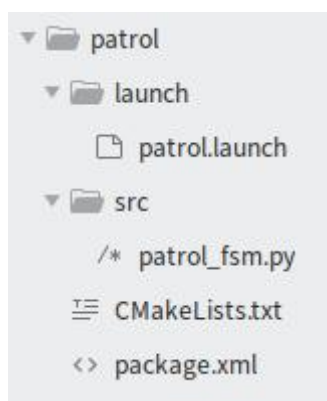
机器人多目标点巡航，特别是按特定巡逻路径进行巡航是很实用的功能。这里将利用前面学到的 `ros-navigation` 单点导航、状态机、状态机任务调度的知识。我们来编写一个应用功能包实现机器人多目标点巡航。

到这里，我们慢慢清楚了 `miiboo` 机器人编程的框架思路，我们将传感器相关的底层驱动包放在 `~/catkin_ws/` 工作空间统一管理，将基于 `google-cartographer` 的 `SLAM` 建图程序包放在 `~/catkin_ws_carto/` 工作空间统一管理，将基于 `ros-navigation` 的导航程序包放在 `~/catkin_ws_nav/` 工作空间统一管理，将高层应用功能包放在 `~/catkin_ws_apps/` 工作空间统一管理。`miiboo` 机器人编程的框架思路，如图 50。



(图 50) miiboo 机器人编程的框架思路

这里将建立一个叫 `catkin_ws_apps` 的 ROS 工作空间，专门用于放置日后开发的各种应用层功能包。关于创建 ROS 工作空间的操作，请参考前面相应部分内容，这里就不做讲解。在 `~/catkin_ws_apps/src/` 中建立一个叫 `patrol` 的功能包，建好后的 `patrol` 功能包文件结构，如图 51。



(图 51) patrol 功能包文件结构

关于功能包的文件结构，大家已经很熟悉了，就不啰嗦了。这里重点讲解一下 `patrol_fsm.py` 这个文件，文件内容如图 52。

```

patrol_fsm.py x
#!/usr/bin/env python

import rospy
from smach import StateMachine
from smach_ros import SimpleActionState
from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal

waypoints = [
    ['one', (2.1, 2.2), (0.0, 0.0, 0.0, 1.0)],
    ['two', (6.5, 4.43), (0.0, 0.0, -0.984047240305, 0.177907360295)],
    ['three', (1.5, 4.0), (0.0, 0.0, 0.0, 1.0)]
]

if __name__ == '__main__':
    rospy.init_node('patrol')
    patrol = StateMachine(['succeeded', 'aborted', 'preempted'])
    with patrol:
        for i, w in enumerate(waypoints):
            goal_pose = MoveBaseGoal()
            goal_pose.target_pose.header.frame_id = 'map'
            goal_pose.target_pose.pose.position.x = w[1][0]
            goal_pose.target_pose.pose.position.y = w[1][1]
            goal_pose.target_pose.pose.position.z = 0.0
            goal_pose.target_pose.pose.orientation.x = w[2][0]
            goal_pose.target_pose.pose.orientation.y = w[2][1]
            goal_pose.target_pose.pose.orientation.z = w[2][2]
            goal_pose.target_pose.pose.orientation.w = w[2][3]

            StateMachine.add(
                w[0],
                SimpleActionState('move_base', MoveBaseAction, goal=goal_pose),
                transitions={'succeeded': waypoints[(i + 1) % len(waypoints)][0]}
            )

    patrol.execute()

```

(图 52) patrol_fsm.py 文件内容

这里采用 python 来编写多目标点巡航的逻辑，python 开发 ROS 节点的优点是简洁高效。代码中 waypoints 数组里面存放的是要巡航的各个目标点，大家可以根据需要进行相应的替换和增减；with patrol 代码块里面实现状态机的构建；最后调用状态机的执行函数，状态机就开始工作了，也就是开始执行巡航了。

启动多目标点巡航分为 3 步：启动机器人上所有传感器、启动导航所需各个节点、启动多目标点巡航节点。

首先，启动机器人上所有传感器，打开终端，通过下面的命令直接启动就行了。

```

source ~/catkin_ws/devel/setup.bash
roslaunch miiboo_bringup miiboo_all_sensor.launch

```

然后，启动导航所需各个节点，打开终端，通过下面的命令直接启动就行了。

```

source ~/catkin_ws_nav/devel/setup.bash
roslaunch miiboo_nav miiboo_nav.launch

```

最后，启动多目标点巡航节点，打开终端，通过下面的命令直接启动就行了。

```

source ~/catkin_ws_apps/devel/setup.bash
roslaunch patrol patrol.launch

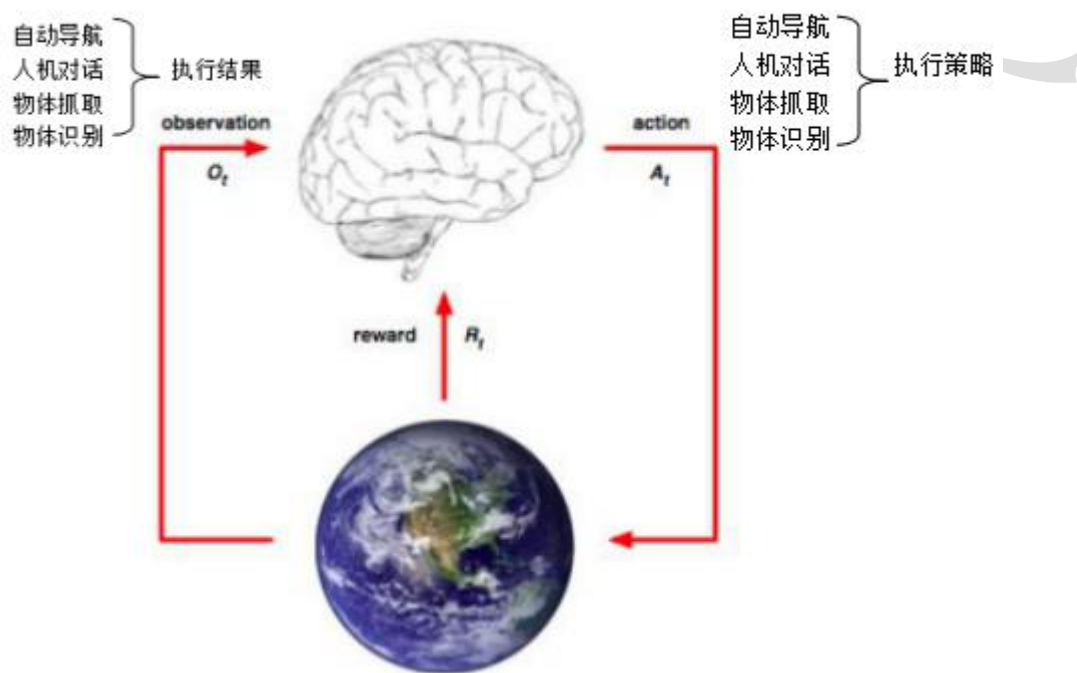
```

4.3.复杂多任务机器人未来展望

机器人可以进行自动导航、人机对话、用机械臂抓取物体、物体识别等。将这些任务结合起

来，利用机器人强大的大脑推理机制能完成更为复杂和智能化的任务。如果说基于状态机的复杂任务调度是 1.0 版本的智能，那么基于大脑推理机制的复杂任务调度将是 2.0 版本的智能。

我的设想是利用强化学习神经网络作为大脑推理机制的实现实体，如图 53。自动导航、人机对话、用机械臂抓取物体、物体识别等任务组合的整体作为机器人与外界环境交互的动作空间，动作空间的的状态分为两种形态：执行结果、执行策略。执行结果作为强化学习神经网络的输入，而执行策略作为强化学习神经网络的输出。我们不断通过各种复杂的实际场景的粒子来训练机器人，让机器人能在复杂场景下能做正确的事情。比如说，当机器人收到主人“我渴了”的语音信息后，自动导航到桌子边，然后识别桌上的可乐，并用机械臂抓取，最后递给主人，并提醒主人“你的可乐来了”。



(图 53) 强化学习神经网络作为大脑推理机制

哈哈！这样的想法很炫酷，不过以目前的技术实现难度还比较大，所以作为未来展望分享给大家。希望和大家一起努力，在不远的将来能实现这个梦想。

5. 机器人巡航与现场监控

机器人在实际生产生活中能给人类带来很大的帮助，比如一款能巡视和现场监控的机器人。商场、机场、家里等需要监控和安保的场景，拍一个机器人去是再合适不过的了。结合前面所学的知识，我们可以让 miiboo 机器人执行巡视与现场监控的任务。这里介绍两种工作方式：手动遥控式现场监控、自动巡航式现场监控。

5.1. 手动遥控式现场监控

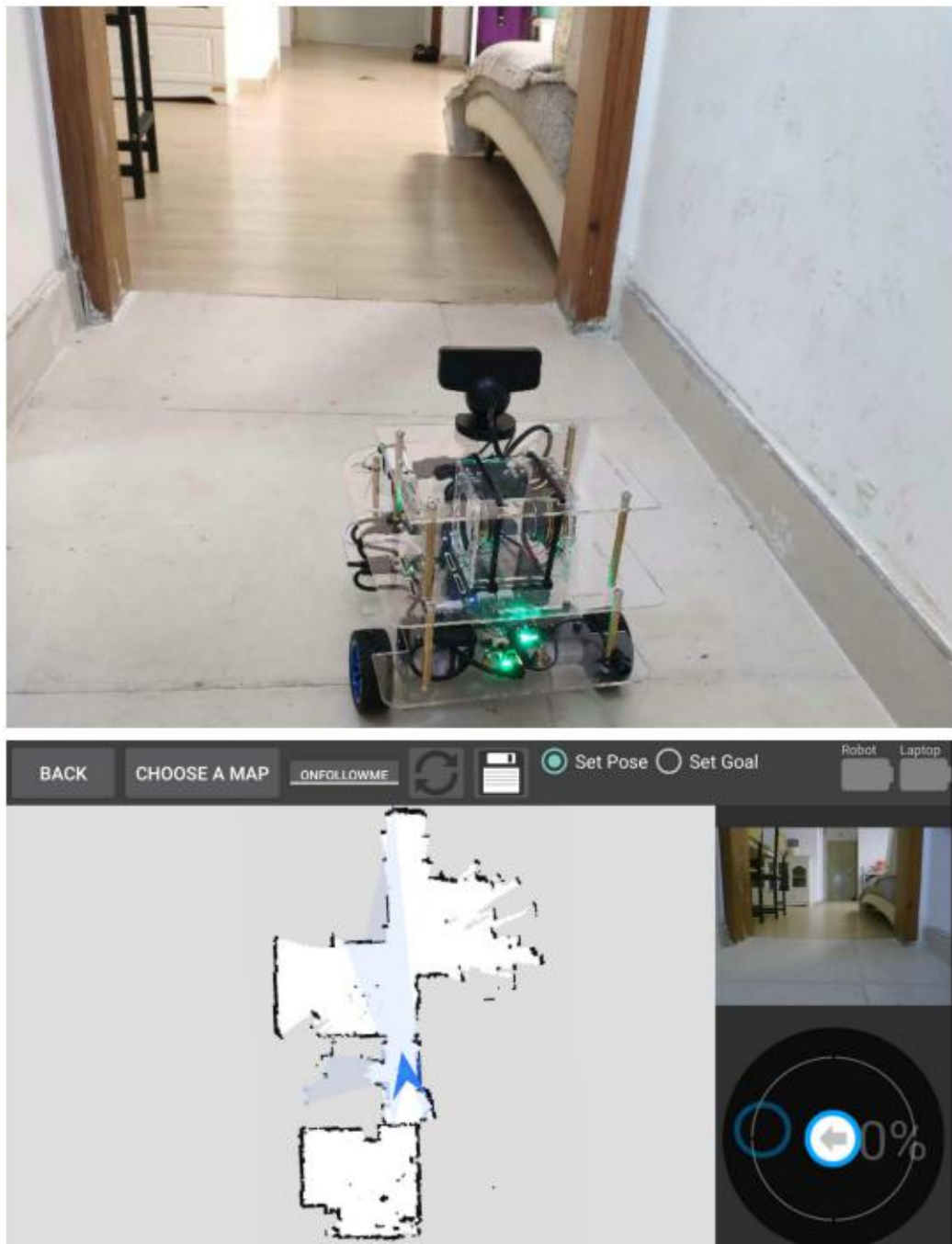
首先，启动机器人上所有传感器，这样机器人上的摄像头就可以将图像发布到 ROS 中，供远程订阅和显示。打开终端，通过下面的命令直接启动就行了。

```
source ~/catkin_ws/devel/setup.bash
roslaunch miiboo_bringup miiboo_all_sensor.launch
```

然后，启动导航所需各个节点，这样就可以实时查看机器人所在的具体位置。打开终端，通过下面的命令直接启动就行了。


```
source ~/catkin_ws_nav/devel/setup.bash
roslaunch miiboo_nav miiboo_nav.launch
```

最后，在 Android 手机端用 miiboo 机器人 APP 控制机器人运动和视频监控周围环境，如图 54。



（图 54）miiboo 机器人 APP 控制机器人运动和视频监控周围环境

5.2. 自动巡航式现场监控

与手动遥控式现场监控的区别是，自动巡航式现场监控不需要人员值守和遥控机器人运动。机器人能根据设定的巡逻路线巡航并实时回传视频监控，结合人脸识别、事件识别等人工智能及视频分析算法，可以对异常事件进行自动的监控。

首先，启动机器人上所有传感器，这样机器人上的摄像头就可以将图像发布到 ROS 中，供



远程订阅和显示。打开终端，通过下面的命令直接启动就行了。

```
source ~/catkin_ws/devel/setup.bash
roslaunch miiboo_bringup miiboo_all_sensor.launch
```

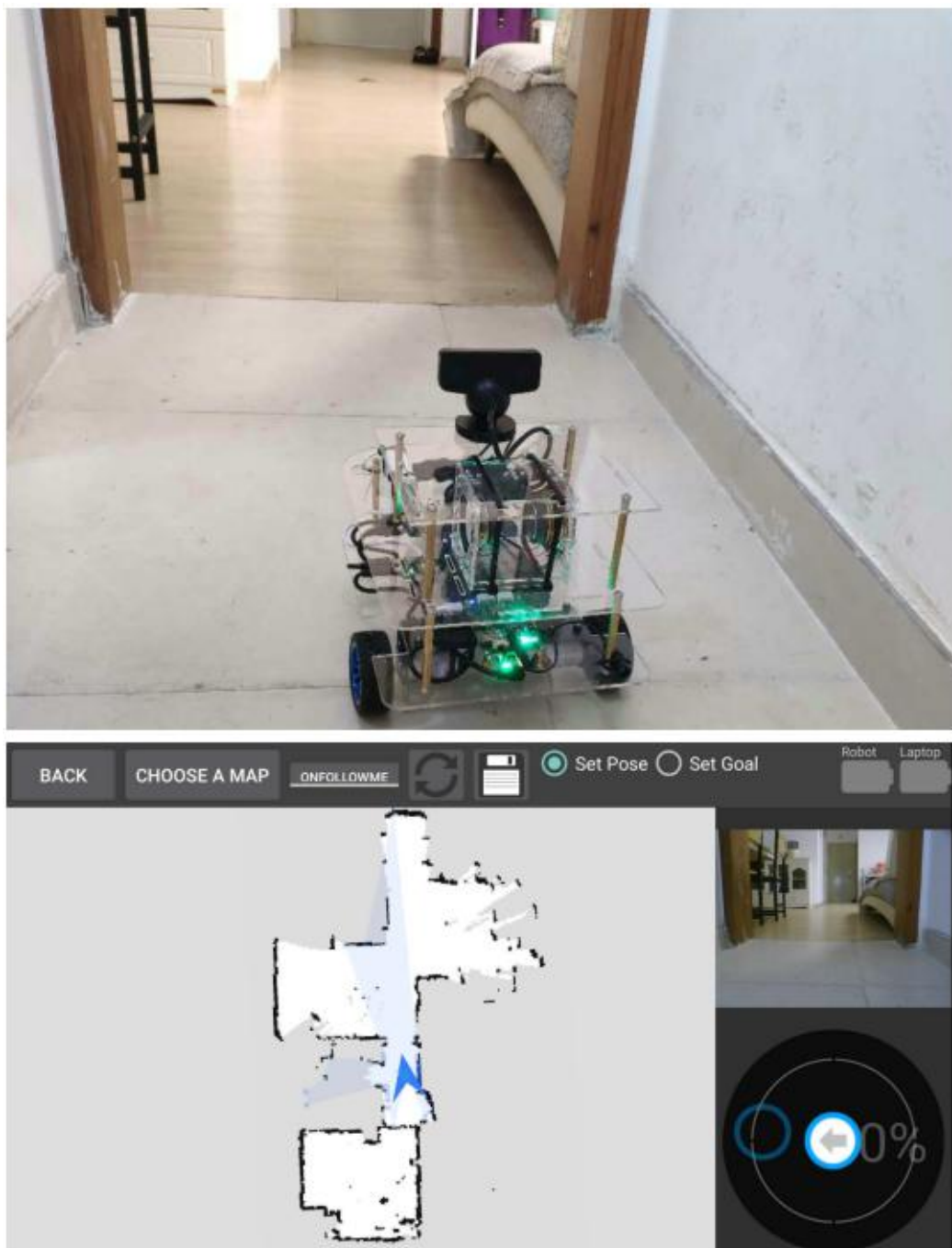
然后，启动导航所需各个节点，这样就可以实时查看机器人所在的具体位置。打开终端，通过下面的命令直接启动就行了。

```
source ~/catkin_ws_nav/devel/setup.bash
roslaunch miiboo_nav miiboo_nav.launch
```

接下来，设定好巡逻路线后，启动多目标点巡航节点，打开终端，通过下面的命令直接启动就行了。

```
source ~/catkin_ws_apps/devel/setup.bash
roslaunch patrol patrol.launch
```

最后，在 Android 手机端用 miiboo 机器人 APP 视频监控周围环境，或者将订阅的视频放入人工智能及视频分析算法，如图 55。



(图 55) miiboo 机器人 APP 视频监控周围环境