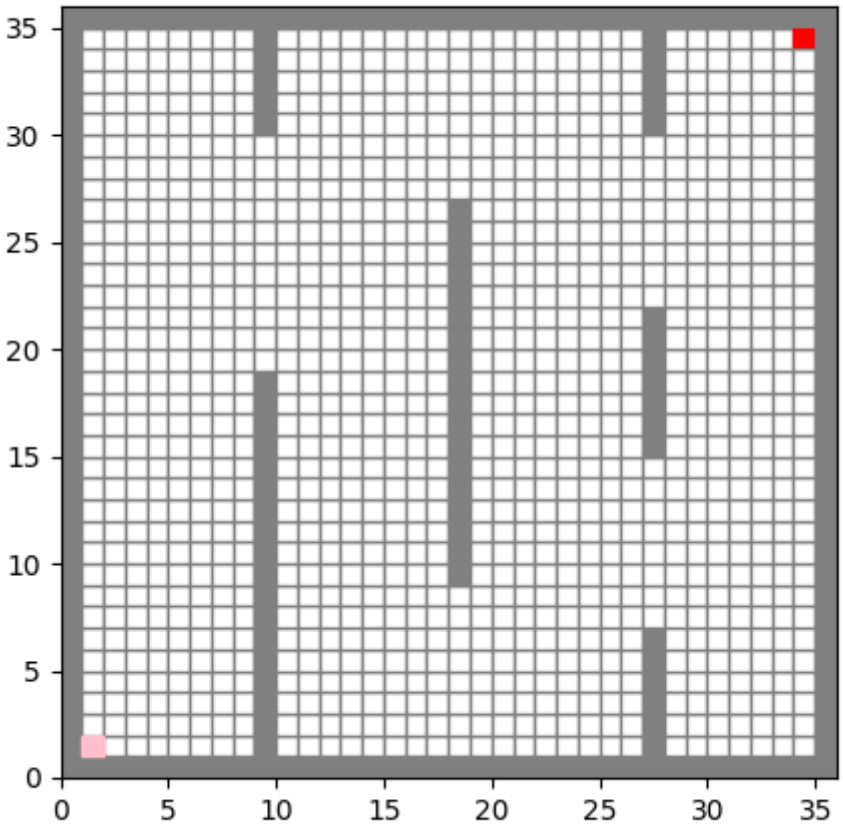
A* 寻路算法

栅格地图

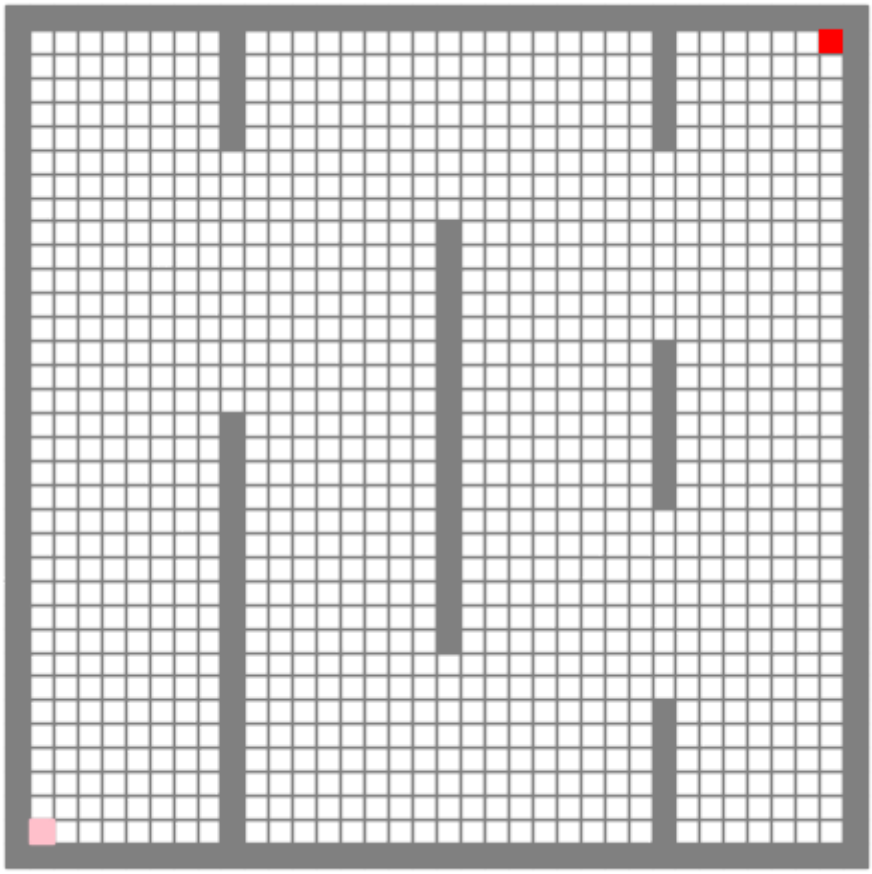
栅格地图是把环境划分成一系列栅格，其中每一栅格给定一个可能值，表示该栅格被占据的概率。

如下图，我们用灰色表示灰色表示障碍物，用白色白色表示可以行走区域。

其中左下角我们放置一个粉色表示起始的位置，右上角红色表示我们想要去的目标位置



A* 算法



代价计算

这里我们允许机器人在栅格地图中沿着八邻域行走，所以我们使用对角距离来计算代价。

代价计算公式如下：

$$f(n) = g(n) + h(n)$$

g(n) 表示当前节点n距离起点的距离

h(n) 表示当前节点n距离终点的距离

f(n) 表示当前节点距离起点和终点累积代价

实现步骤

1. 初始化两个集合：
    1. 一个存放要处理点的集合open_set
    2. 另一个存放已经处理过点的集合close_set
2. 将起点放入open_set，并设置代价为最高代价0
3. 开始遍历
    1. 如果open_set不为空，则从中选取代价最高的点n
    2. 若点n为终点：
        1. 从终点开始逐渐追踪parent节点，一直到起点
        2. 返回结果路径，算法结束
    3. 若点n不为终点
        1. 将点n从open_set中删除，放到close_set中
        2. 遍历节点n的8邻域节点：
            1. 如果邻域节点m在close_set中，则直接调过
            2. 如果邻域节点m不在open_set中：
                1. 设置节点m的parent为节点n
                2. 计算节点m的cost代价
                3. 将节点m加入open_set中

示例代码

```
import sys
import time
import numpy as np
from matplotlib.patches import Rectangle

from point import Point

class AStar:
    def __init__(self, map):
        self.map=map
        # 待遍历的点
        self.open_set = []
```

```
        # 已经遍历的节点
        self.close_set = []

    # 计算距离起点的对角距离
    def BaseCost(self, p):
        x_dis = abs(p.x - self.startPoint.x)
        y_dis = abs(p.y - self.startPoint.y)
        # Distance to start point
        return x_dis + y_dis + (np.sqrt(2) - 2) * min(x_dis,
y_dis)

    # 计算距离终点的对角距离
    def HeuristicCost(self, p):
        x_dis = abs(self.endPoint.x - p.x)
        y_dis = abs(self.endPoint.y - p.y)
        # Distance to end point
        return x_dis + y_dis + (np.sqrt(2) - 2) * min(x_dis,
y_dis)

    # 总的代价
    def TotalCost(self, p):
        return self.BaseCost(p) + self.HeuristicCost(p)

    def IsValidPoint(self, x, y):
        if x < 0 or y < 0:
            return False
        if x >= self.map.size or y >= self.map.size:
            return False
        return not self.map.IsObstacle(x, y)

    def IsInPointList(self, p, point_list):
        for point in point_list:
            if point.x == p.x and point.y == p.y:
                return True
        return False

    def IsInOpenList(self, p):
        return self.IsInPointList(p, self.open_set)

    def IsInCloseList(self, p):
        return self.IsInPointList(p, self.close_set)

    def IsStartPoint(self, p):
        return p.x == self.startPoint.x and p.y ==
self.startPoint.y
```

```
    def IsEndPoint(self, p):
        return p.x == self.endPoint.x and p.y == self.endPoint.y

    def SaveImage(self, plt):
        millis = int(round(time.time() * 1000))
        filename = './' + str(millis) + '.png'
        plt.savefig(filename)
        #plt.pause(0.00001)
        pass

    def ProcessPoint(self, x, y, parent):
        # 若点是无效的点，则直接跳过
        if not self.IsValidPoint(x, y):
            return # Do nothing for invalid point

        # 判断点是否已经处理过了
        p = Point(x, y)
        if self.IsInCloseList(p):
            return # Do nothing for visited point

        # 若点不在待处理的点里面
        if not self.IsInOpenList(p):
            p.parent = parent
            p.cost = self.TotalCost(p)
            self.open_set.append(p)

            print('计算点：  [', p.x, ',', p.y, ']', ', cost: ',
p.cost)

    def SelectPointInOpenList(self):
        """
            找出代价最小的点
        """
        index = 0
        selected_index = -1
        min_cost = sys.maxsize
        for p in self.open_set:
            # cost = self.TotalCost(p)
            cost = p.cost
            if cost < min_cost:
                min_cost = cost
                selected_index = index
            index += 1
        return selected_index
```

```
def BuildPath(self, p, ax, plt, start_time):
    """
    构建路径
    :param p:
    :param ax:
    :param plt:
    :param start_time:
    :return:
    """
    path = []
    # 根据parent反向将所有的点添加到路径中
    while True:
        # 始终将点插到队列的前面
        path.insert(0, p)
        if self.IsStartPoint(p):
            break
        else:
            p = p.parent

    # 找到的路径绘制出来
    for p in path:
        rec = Rectangle((p.x, p.y), 1, 1, color='g')
        ax.add_patch(rec)
        plt.draw()
        self.SaveImage(plt)

    end_time = time.time()
    print('算法执行完成，耗时：', int(end_time-start_time), ' 秒')

def run(self, ax, plt,startPoint,endPoint):
    self.startPoint = startPoint
    self.endPoint = endPoint


    start_time = time.time()

    self.startPoint.cost = 0
    self.open_set.append(self.startPoint)

    while True:
        index = self.SelectPointInOpenList()
        if index < 0:
            print('没有找到路径，算法执行失败!!!')
```

```
            return
        # 取出点的信息
        p = self.open_set[index]
        # 在图上绘制出要处理的点的信息
        rec = Rectangle((p.x, p.y), 1, 1, color='pink')
        ax.add_patch(rec)
        self.SaveImage(plt)

        # 判断是否已经是终点，若为终点，则反向构建路径
        if self.IsEndPoint(p):
            return self.BuildPath(p, ax, plt, start_time)

        # 将已经处理的点从open_set中删除
        del self.open_set[index]
        # 将已处理的点存到close_set中
        self.close_set.append(p)

        # 处理8个方向相邻的点
        x = p.x
        y = p.y
        self.ProcessPoint(x-1, y+1, p)
        self.ProcessPoint(x-1, y, p)
        self.ProcessPoint(x-1, y-1, p)
        self.ProcessPoint(x, y-1, p)
        self.ProcessPoint(x+1, y-1, p)
        self.ProcessPoint(x+1, y, p)
        self.ProcessPoint(x+1, y+1, p)
        self.ProcessPoint(x, y+1, p)
```

SLAM介绍

SLAM (simultaneous localization and mapping)即时定位与地图构建。

希望机器人从未知环境的未知地点出发，在运动过程中通过重复观测得到的地图特征（比如，墙角，柱子等）定位自身位置和姿态，再根据自身位置增量式的构建地图，从而达到同时定位和地图构建的目的。

运用SLAM技术，我们可以将一个机器人放到未知环境中的未知位置，然后机器人可以一边移动一边创建环境的地图，当地图创建好之后，我们就可以控制机器人更加准确移动到地图中的任意角落啦！

我们学习SLAM技术需要思考三大问题: where am I ? 我在哪 where am I going ? 我要去哪里 How do I get there? 怎么去

我在哪

Localization: 定位问题 帮助机器人知道他在什么位置 手段: gps , 高精度gps ,激光雷达, 照相机, 超声波传感器, 蓝牙,nfc,二维码等。

我要去哪里

mapping: 构建地图过程 机器人需要识别地图, 认识他已经移动的位置

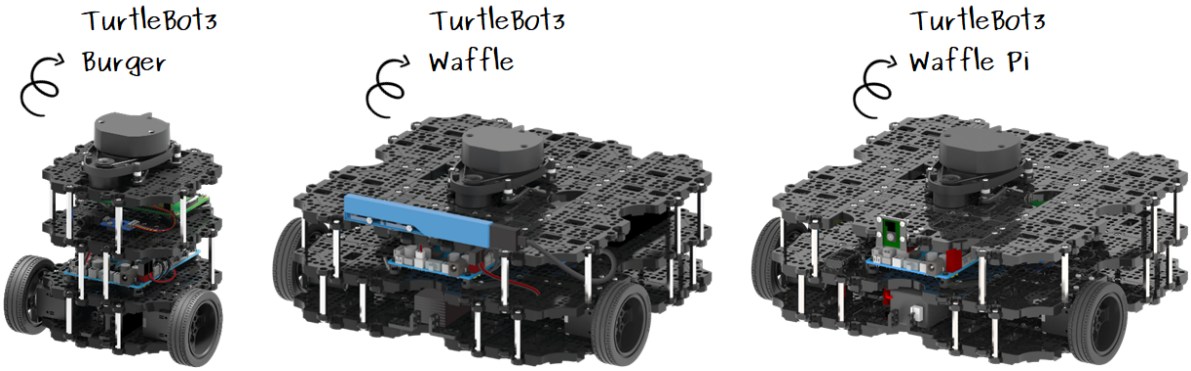
目标: 经度,维度, 房间编号, 几何中心, 或者xyz坐标系 map可以是实现定义好, 也可以边走边建立

怎么去

motion planning and path planning: 运动规划 规划出到达目标的路径, 避免碰撞, 避免兜圈子

目标: 生成一个路径, 一套坐标系的点

tutlebot3仿真环境



1.安装tutlebot3

```
sudo apt-get install ros-melodic-turtlebot3-*
# 安装地图算法依赖
apt-get install ros-melodic-gmapping
# 安装dwa路径规划算法
apt-get install ros-melodic-dwa-local-planner


sudo apt-get install ros-melodic-turtlebot3-*
# 安装地图算法依赖
apt-get install ros-melodic-gmapping
# 安装dwa路径规划算法
apt-get install ros-melodic-dwa-local-planner
```

2. 控制小车运动

首先我们可以在仿真环境中启动tutlebot3小车

```
export TURTLEBOT3_MODEL=waffle_pi
roslaunch turtlebot3_gazebo turtlebot3_empty_world.launch
```

然后执行下面的命令，根据输出提示即可控制小车运动

```
rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

若小车不能运动，则有可能是缺少依赖

```
sudo apt-get install ros-melodic-gazebo-ros-pkgs ros-melodic-
gazebo-ros-control
sudo apt-get install ros-melodic-teleop-twist-keyboard
```

若安装出现`[Err] [REST.cc:205] Error in REST request`则需要进行如下配置:

```
sudo gedit ~/.ignition/fuel/config.yaml

将文件中https://api.ignitionfuel.org修改为
https://api.ignitionrobotics.org
```

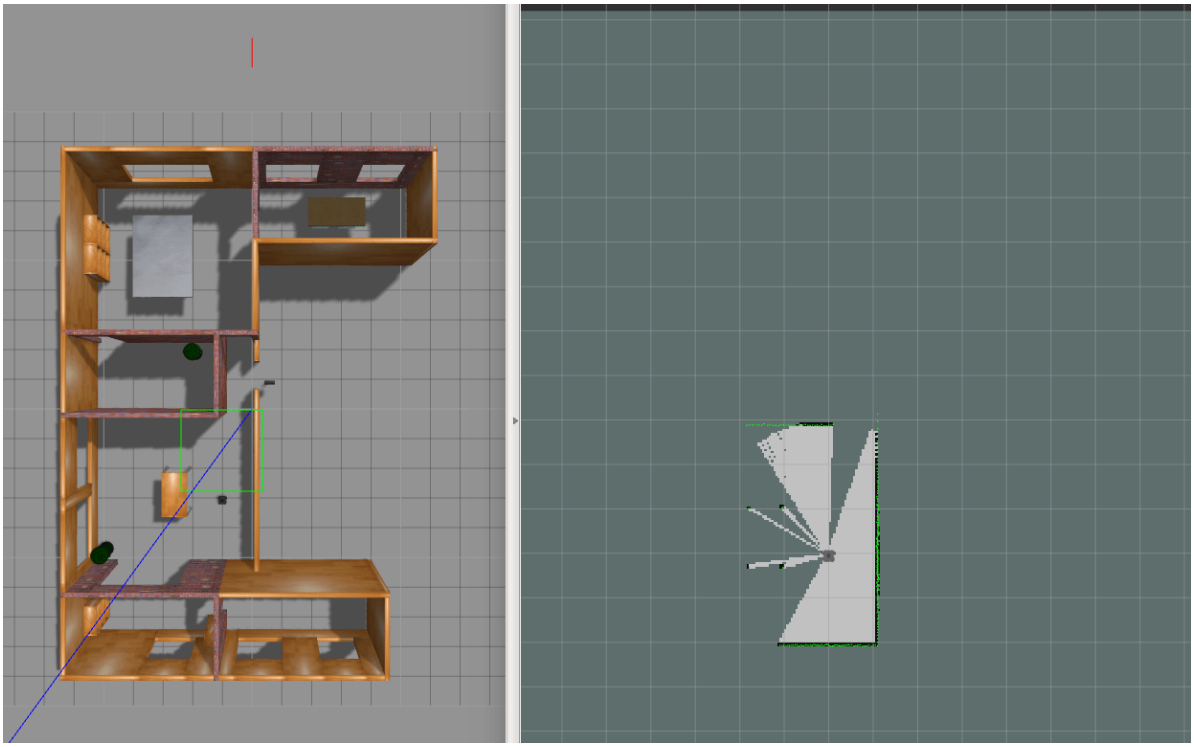
若经过上述步骤还是无法打开gazebo ,如果是虚拟机的话,就需要关闭3D加速.

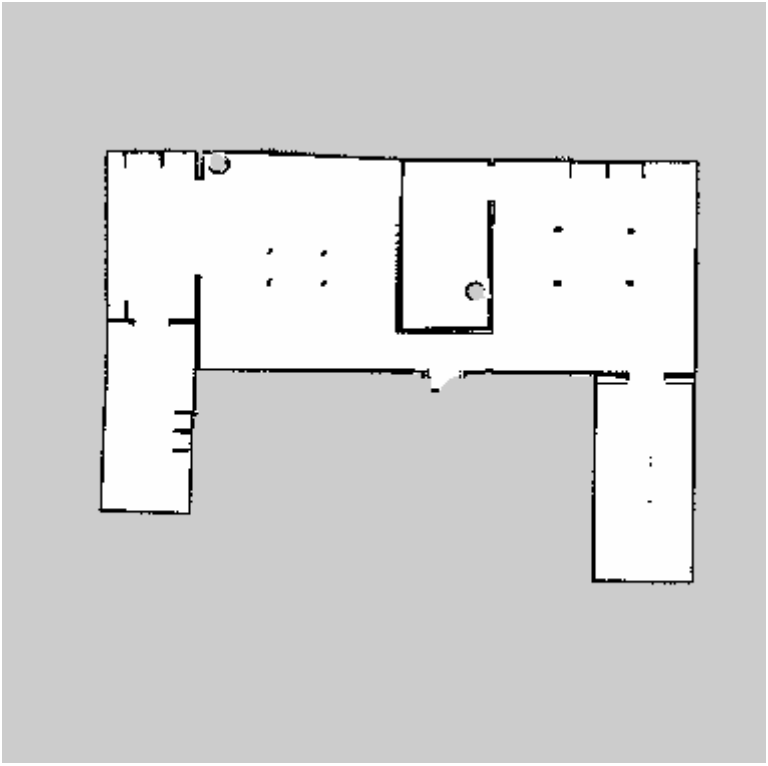
先执行如下命令

```
export SVGA_VGPU10=0
```

然后在虚拟机设置的显示中关闭3D加速

3. slam构建地图





1. 开启仿真环境

```
export TURTLEBOT3_MODEL=waffle_pi
roslaunch turtlebot3_gazebo turtlebot3_house.launch
```

2. 开启实时建图功能,算法选择gmapping, 还有多种slam算法, 如cartographer, hector_slam

```
export TURTLEBOT3_MODEL=waffle_pi
roslaunch turtlebot3_slam turtlebot3_slam.launch
slam_methods:=gmapping
```

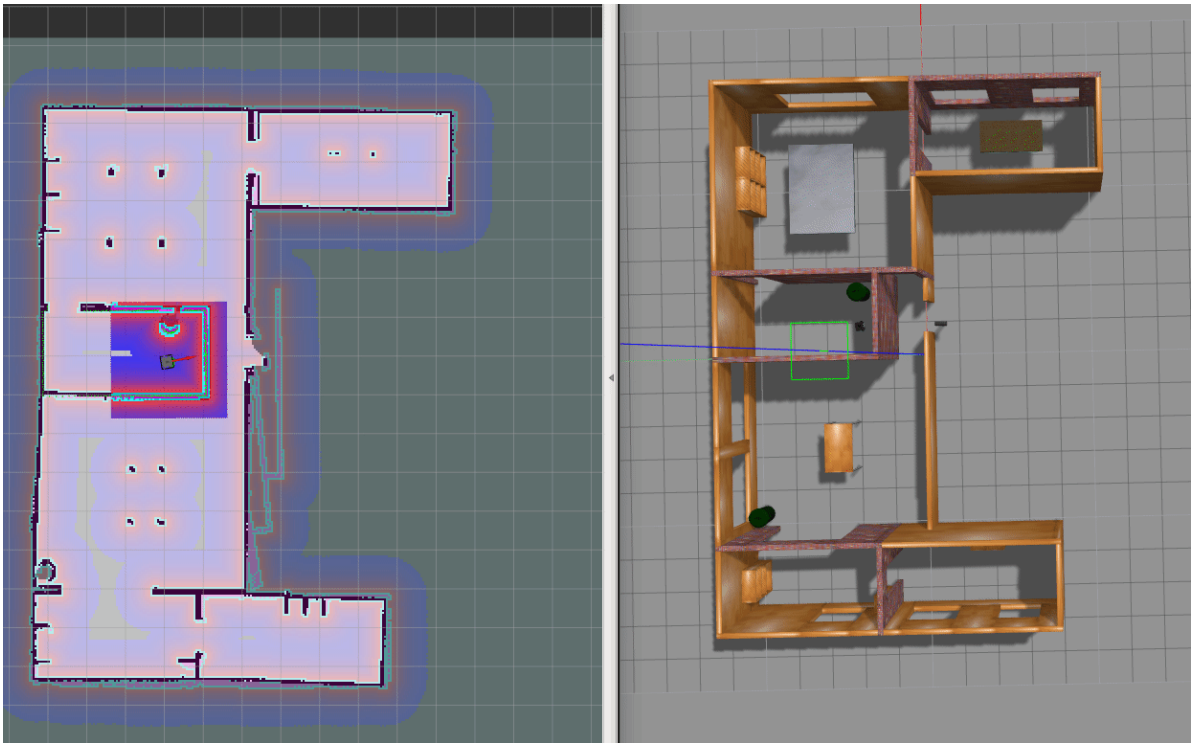
3. 控制turtlebot3

```
rosrun teleop_twist_keyboard teleop_twist_keyboard.py
或者
export TURTLEBOT3_MODEL=waffle_pi
roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

4. 尽可能保证地图闭合,然后保存地图为文件

```
rosrun map_server map_saver -f ~/map
```

4. slam自动导航



```
# 打开仿真环境
export TURTLEBOT3_MODEL=waffle_pi
roslaunch turtlebot3_gazebo turtlebot3_house.launch
# 打开自动导航功能
export TURTLEBOT3_MODEL=waffle_pi
roslaunch turtlebot3_navigation turtlebot3_navigation.launch
map_file:=/home/kaijun/map.yaml
```

在rviz界面上，设定目标点，小车就会自动运行到目标位置去。

5. 代码实现导航

这一块的内容和我们刚才讲过的内容很类似，这里我们通过代码的方式来实现小车的导航功能

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import rospy
import actionlib
from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal
from actionlib_msgs.msg import *
from geometry_msgs.msg import Point

# 导航移动api
def move_to_goal(xGoal,yGoal):
    #simpleactionclient
    ac = actionlib.SimpleActionClient("move_base",
MoveBaseAction)
    #等待5秒 ,actionserver启动
    while(not ac.wait_for_server(rospy.Duration.from_sec(5.0))):
        rospy.loginfo("等待move_base actionserver启动")
    goal = MoveBaseGoal()
    #指定地图参考系
    goal.target_pose.header.frame_id = "map"
    goal.target_pose.header.stamp = rospy.Time.now()
    #移动目标设定位姿 xyz和四元数
    goal.target_pose.pose.position = Point(xGoal,yGoal,0)
    goal.target_pose.pose.orientation.x = 0.0
    goal.target_pose.pose.orientation.y = 0.0
    goal.target_pose.pose.orientation.z = 0.0
    goal.target_pose.pose.orientation.w = 1.0
    rospy.loginfo("发送目标到actionserver ...")
    ac.send_goal(goal)
```

```
    #设置超时时间为60s
    ac.wait_for_result(rospy.Duration(60))
    if(ac.get_state() == GoalStatus.SUCCEEDED):
        rospy.loginfo("成功到达")
        return True
    else:
        rospy.loginfo("未在规定时间内到达目的地,失败了")
        return False

if __name__ == '__main__':
    rospy.init_node('map_navigation', anonymous=False)
    x_goal = 1
    y_goal = 0
    print'start go to goal'
    move_to_goal(x_goal,y_goal)
    rospy.spin()
```

执行如下代码即可让小车运动到指定的地图坐标:

```
kaijun@kaijun-pc:~/Documents/slam/slam_ws$ rosrun heima_slam
slam_demo.py
```

真实小车

配置本机的环境变量

```
vim ~/.bashrc
# 将下面的内容加入其中
export BASE_TYPE=4WD
export LIDAR_TYPE=ydlidar
#export BASE_TYPE=4WD_OMNI
#export LIDAR_TYPE=rplidar

export ROS_IP=`hostname -I | awk '{print $1}'`
export ROS_HOSTNAME=`hostname -I | awk '{print $1}'`
export ROS_MASTER_URI=http://master_ip:11311
```

地图创建

在小车树莓派上执行：

```
roslaunch robot_navigation robot_slam_laser.launch
```

在监控电脑上执行rviz:

```
roslaunch robot_navigation slam_rviz.launch
```

遥控小车进行地图探索(注意必须选中当前命令行窗口)：

```
rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

注意：在小车移动的过程中，一定小车不能发生严重碰撞，否则里程计会发生错误，然后建出来的地图会有问题

等地图边缘黑线闭合之后，我们就可以在树莓派小车上保存建立好的地图

```
roscd robot_navigation/maps
rosrun map_server map_saver -f map
```

自动导航

在小车树莓派上执行：

```
roslaunch robot_navigation robot_navigation.launch
```

在监控电脑上执行：

```
roslaunch robot_navigation navigation_rviz.launch
```

操作步骤：

1. 给小车设定初始位置
2. 给小车输入指定目标点，小车即可自动运行到目标点中