**Linux系统基础**

**实验报告**

学 号: 2022900479

姓 名: 赵鑫

院 系: 信息工程学院

专 业: 物联网工程

指导教师: 闵海根

日 期: 2024年12月6日

**目 录**

[1 ROS基础实验-虚拟环境中移动机器人控制 1](#_Toc181173344)

[1.1 实验内容 1](#_Toc181173345)

[1.2 实验要求 1](#_Toc181173346)

[2 ROS感知实验-使用激光雷达进行地图构建 2](#_Toc181173347)

[2.1 实验内容 2](#_Toc181173348)

[2.2 实验要求 2](#_Toc181173349)

[3 ROS机器人导航实验-自主导航与路径规划 3](#_Toc181173350)

[3.1 实验内容 3](#_Toc181173351)

[3.2 实验要求 3](#_Toc181173352)

[4 ROS网络通信实验-发布与订阅节点的数据传输 4](#_Toc181173353)

[4.1 实验内容 4](#_Toc181173354)

[4.2 实验步骤 4](#_Toc181173355)

[实验心得 6](#_Toc181173356)

# ROS基础实验-虚拟环境中移动机器人控制

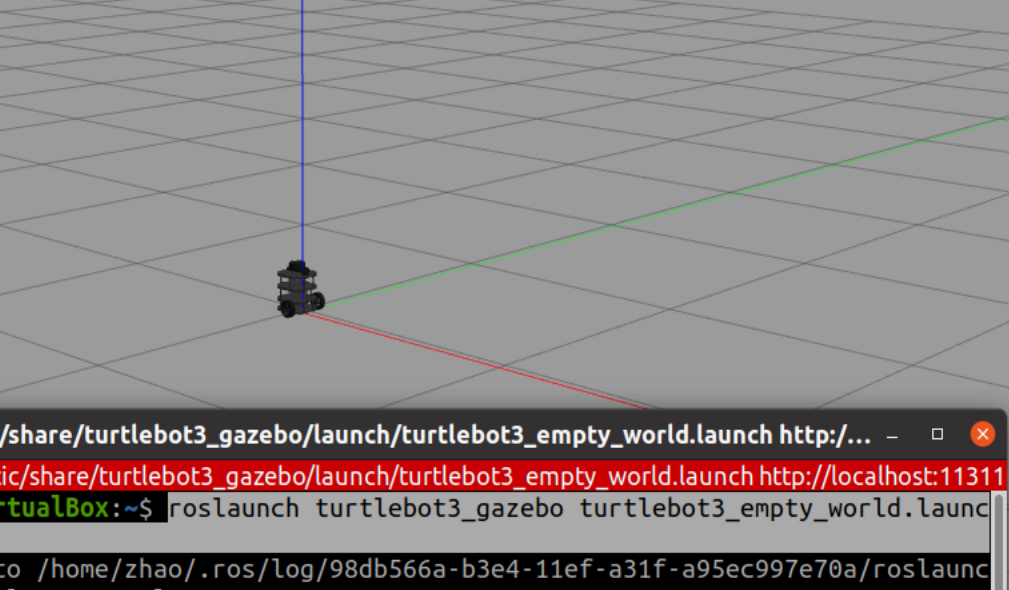
## 实验内容

本实验的目的是学习ROS和Gazebo仿真平台的基本操作，加载并控制一个虚拟机器人在环境中移动。通过本实验，学生可以了解 ROS 中节点、话题、消息等基本概念，并能掌握使用Gazebo仿真器进行虚拟测试的流程。学生将使用turtlebot3仿真模型，在Gazebo中创建虚拟环境，并通过键盘或代码实现机器人移动，最后实现自动导航功能。

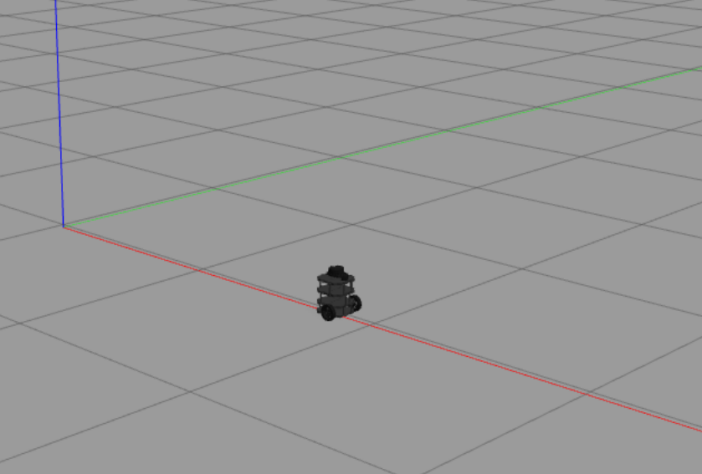
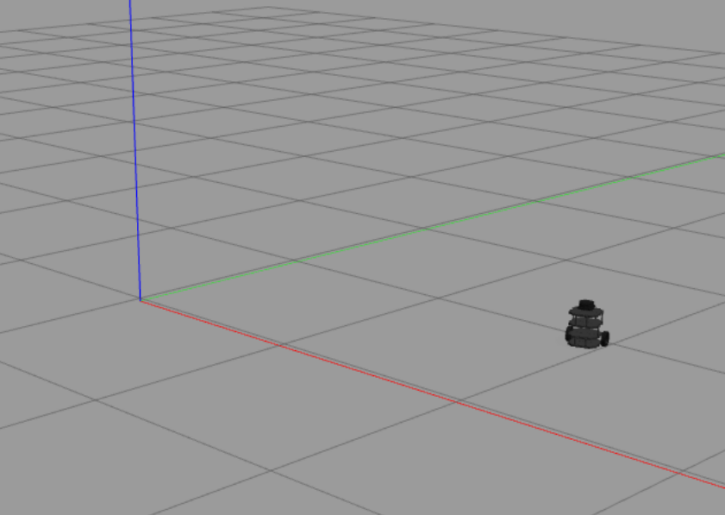
## 实验要求

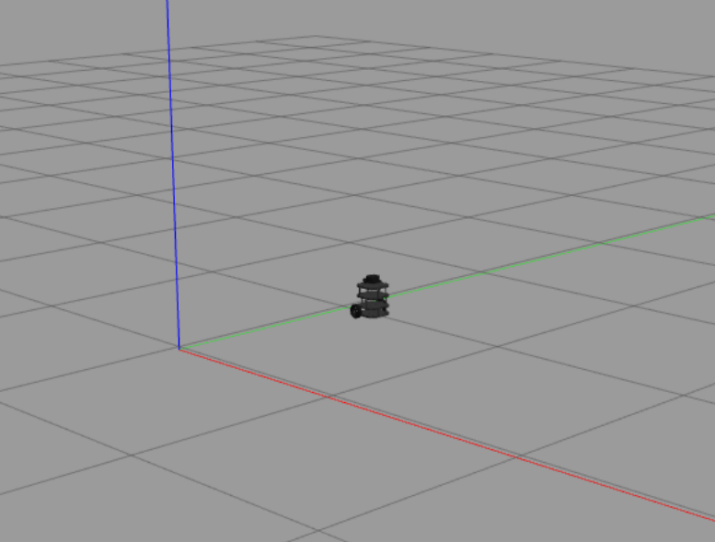
（1）环境搭建

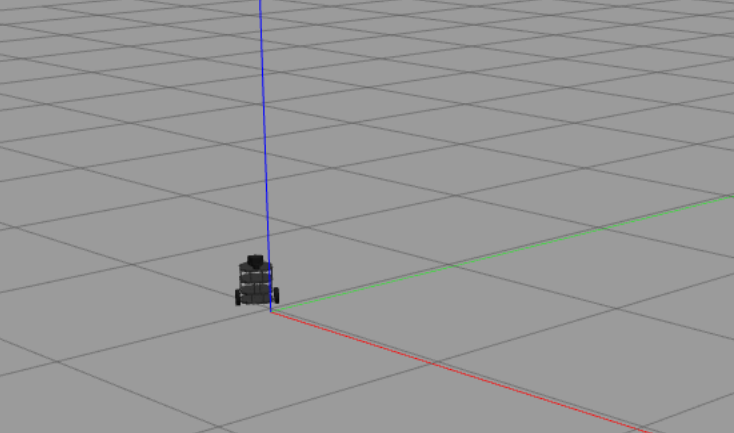
在ROS中安装并配置Gazebo仿真器，确保能加载turtlebot3模型。使用roslaunch turtlebot3\_gazebo turtlebot3\_empty\_world.launch启动一个空白的仿真环境。



（2）机器人控制

使用teleop工具，通过键盘控制机器人前进、后退、左转和右转，熟悉Gazebo仿真环境与ROS的基本操作。编写一个ROS节点，用于让机器人执行简单的运动序列（例如：沿矩形路径行走）。





**在步骤1的基础上，执行rectangle\_path.cpp文件**

矩形路径代码：**rectangle\_path.cpp**

#include "ros/ros.h"

#include "geometry\_msgs/Twist.h"

int main(int argc, char \*argv[])

{

    ros::init(argc, argv, "turtlebot3\_rectangle\_move");

    ros::NodeHandle n;

    // 创建一个Publisher，发布类型为geometry\_msgs/Twist的消息

    ros::Publisher twist\_pub = n.advertise<geometry\_msgs::Twist>("cmd\_vel", 10);

    ros::Rate loop\_rate(1);  // 设置运行频率  ，每秒1次循环

    geometry\_msgs::Twist move\_cmd;  // 定义速度命令

    // 矩形的边长（单位：米）

    double length = 2.0;  // 长边

    double width = 1.0;   // 短边

    while (ros::ok())  // 控制机器人沿矩形路径行驶

    {

        // 机器人沿长边行驶

        move\_cmd.linear.x = 0.2;  // 前进速度

        move\_cmd.angular.z = 0.0; // 不转弯

        for (double t = 0; t < length; t += 0.2)

 // 每次行驶0.2米，直到走完一条边

        {

            twist\_pub.publish(move\_cmd);

            ros::Duration(1).sleep();  // 等待1秒钟

        }

        // 机器人右转90度

        move\_cmd.linear.x = 0.0;

        move\_cmd.angular.z = 1.57;  // 旋转90度（1.57弧度）

        twist\_pub.publish(move\_cmd);

        ros::Duration(1).sleep();  // 等待1秒钟，完成转弯

        // 机器人沿短边行驶

        move\_cmd.linear.x = 0.2;

        move\_cmd.angular.z = 0.0;

        for (double t = 0; t < width; t += 0.2)

// 每次行驶0.2米，直到走完一条边

        {

            twist\_pub.publish(move\_cmd);

            ros::Duration(1).sleep();  // 等待1秒钟

        }

        // 机器人右转90度

        move\_cmd.linear.x = 0.0;

        move\_cmd.angular.z = 1.57;

        twist\_pub.publish(move\_cmd);

        ros::Duration(1).sleep();

        // 重复上述步骤形成矩形

    }

    return 0;

}

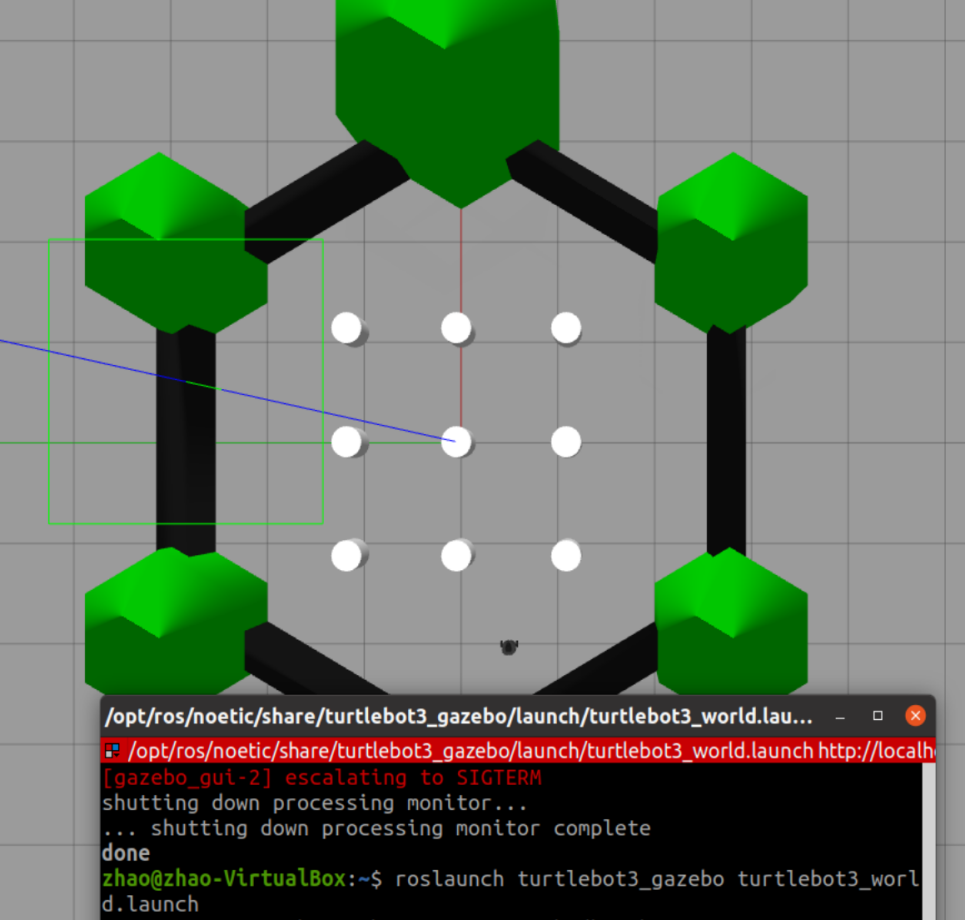
# ROS感知实验-使用激光雷达进行地图构建

## 实验内容

本实验旨在让学生了解如何利用激光雷达传感器进行环境的三维建图。学生将使用turtlebot3机器人和SLAM算法在Gazebo中实现仿真地图构建。通过本实验，学生将掌握激光雷达与SLAM的工作原理，以及如何利用ROS的gmapping包进行实时地图构建。

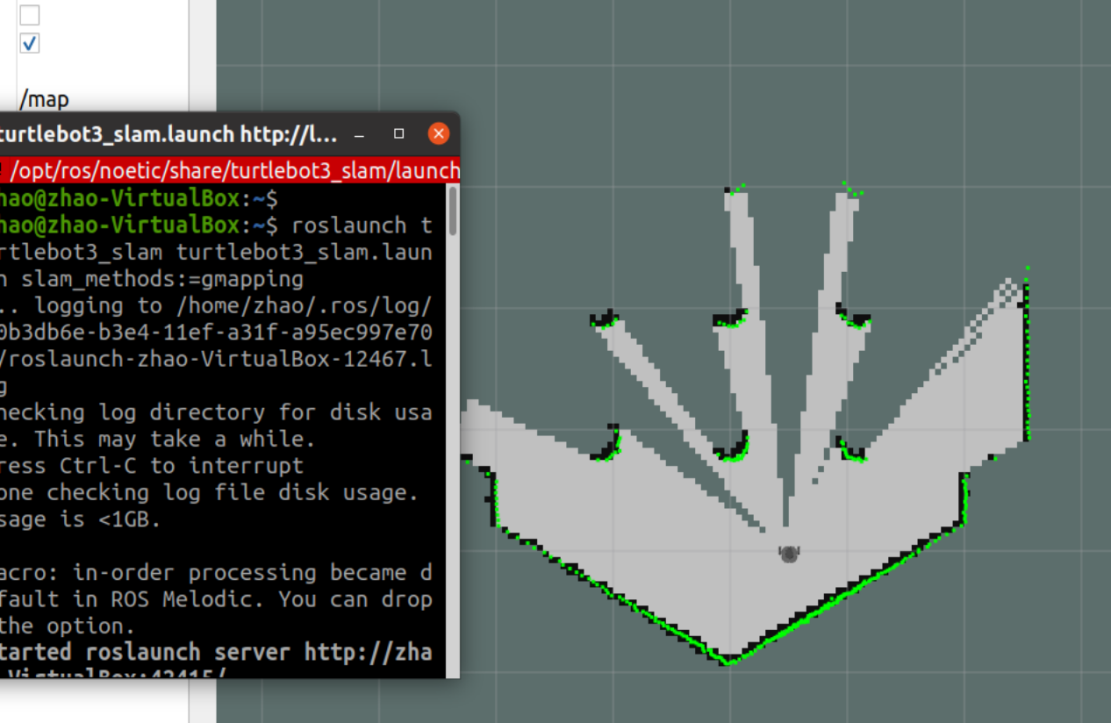
## 实验要求

（1）SLAM 环境搭建

使用 roslaunch turtlebot3\_gazebo turtlebot3\_world.launch 启动一个包含障碍物的仿真环境。安装并配置ROS的gmapping包，确保能够在Gazebo中实现SLAM建图。

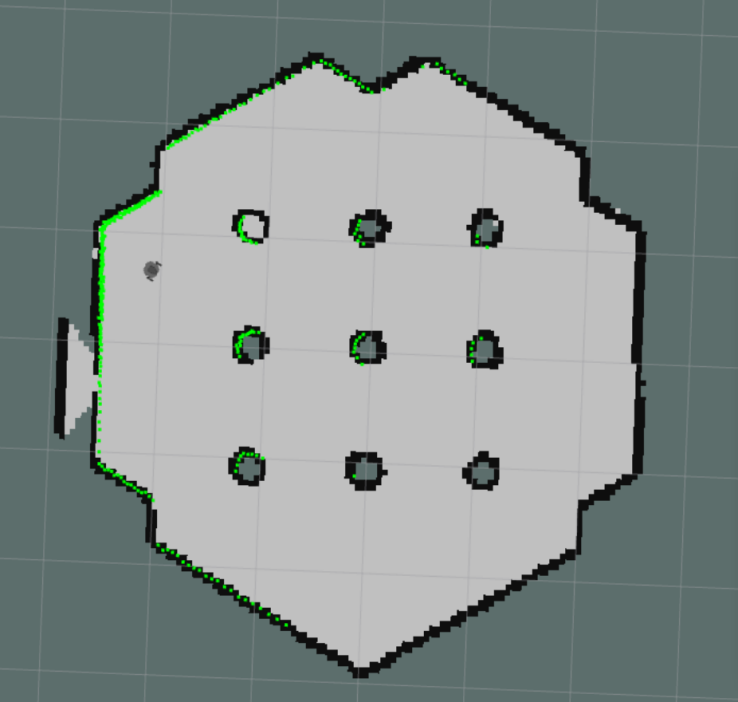
（2）激光雷达传感器使用

了解 turtlebot3 机器人中激光雷达的配置，通过RViz查看激光雷达扫描的数据。使用roslaunch turtlebot3\_slam turtlebot3\_gmapping.launch启动SLAM节点，观察RViz中地图构建的过程。



（3）地图构建

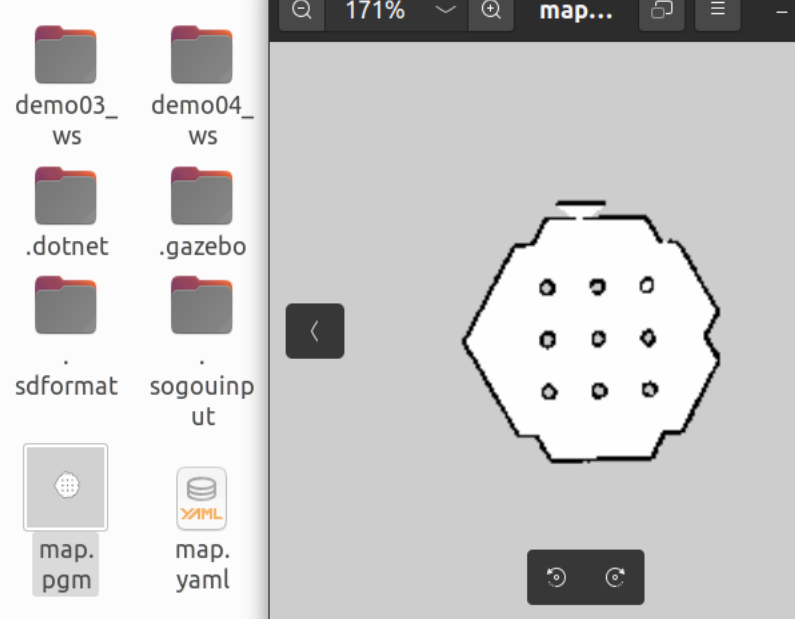
使用teleop工具控制机器人移动，通过激光雷达扫描完成环境的探索和地图的构建。保存生成的地图到本地（例如，保存为map.yaml和map.pgm文件），以便后续实验使用。

**使用roslaunch turtlebot3\_teleop turtlebot3\_teleop\_key.launch控制机器人**

**SLAM建图完成**：

**保存生成的地图到本地（保存为map.yaml和map.pgm文件）**

**使用rosrun map\_server map\_saver -f ~/map 保存地图**



# ROS机器人导航实验-自主导航与路径规划

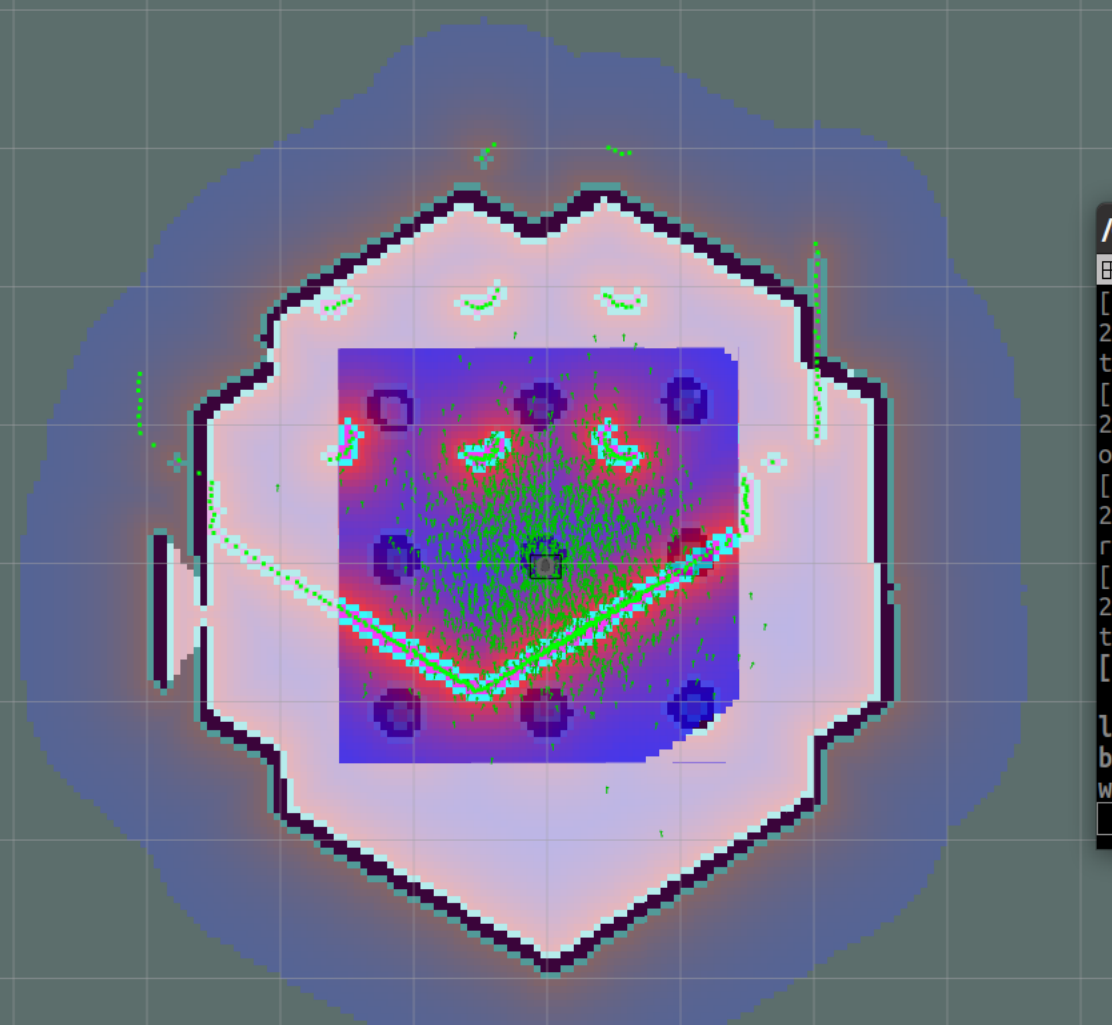
## 实验内容

在本实验中，学生将使用ROS导航栈实现机器人在已知地图中的定位和路径规划。该实验将帮助学生掌握如何结合SLAM构建的地图进行自主导航，实现机器人在已知环境中的自主路径规划、避障及到达目标点。

## 实验要求

（1）AMCL定位

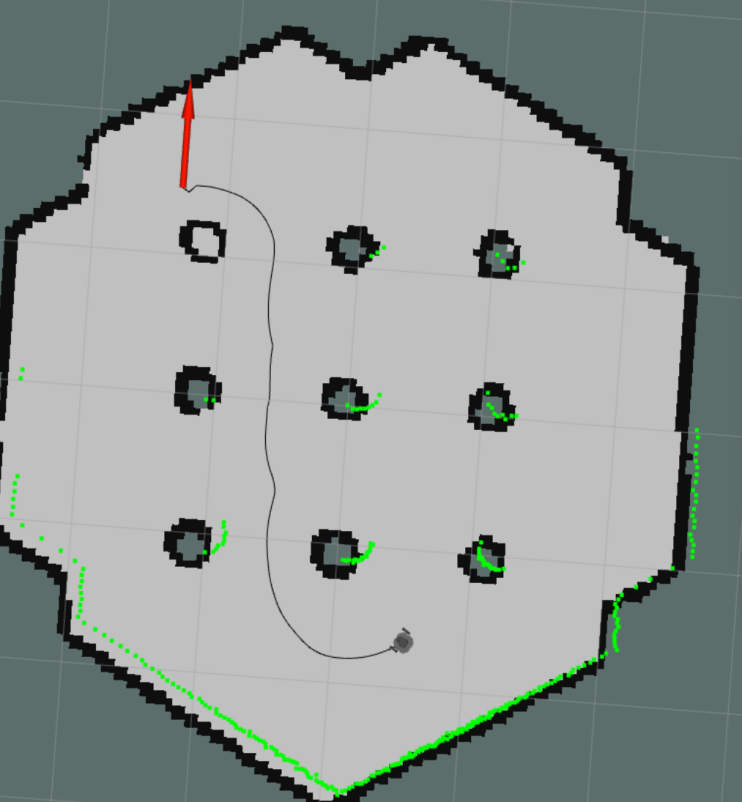
使用上一步实验保存的地图，启动amcl节点，进行自适应蒙特卡洛定位（roslaunch turtlebot3\_navigation turtlebot3\_navigation.launch map\_file:=<path to map.yaml>）。使用RViz确保机器人能够正确地在构建的地图中显示位置，并通过手动控制验证定位精度。



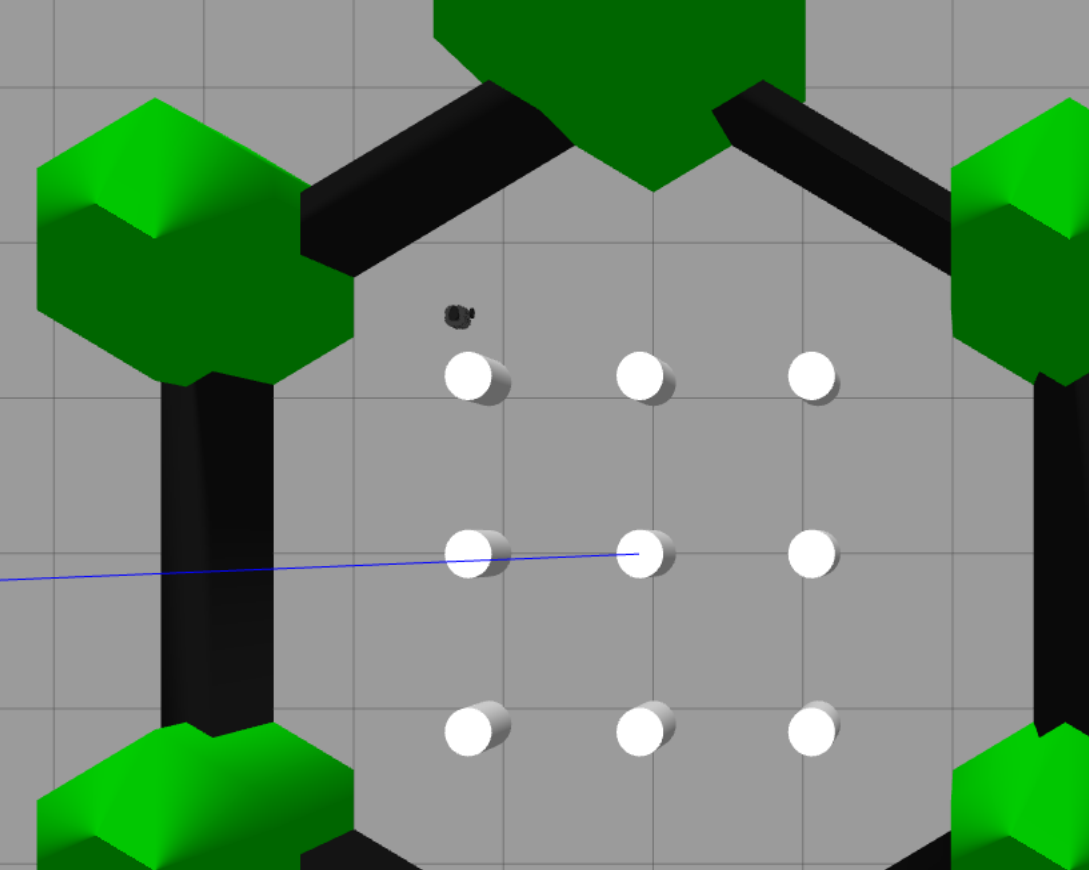
（2）目标设置与路径规划

使用ROS的导航栈move\_base，在RViz中设置目标位置，让机器人自动规划路径到达终点。验证机器人是否能够避开静态障碍物，沿最优路径到达目标位置。

**在RViz中设置目标位置，机器人规划好路径，开始向目标位置移动**



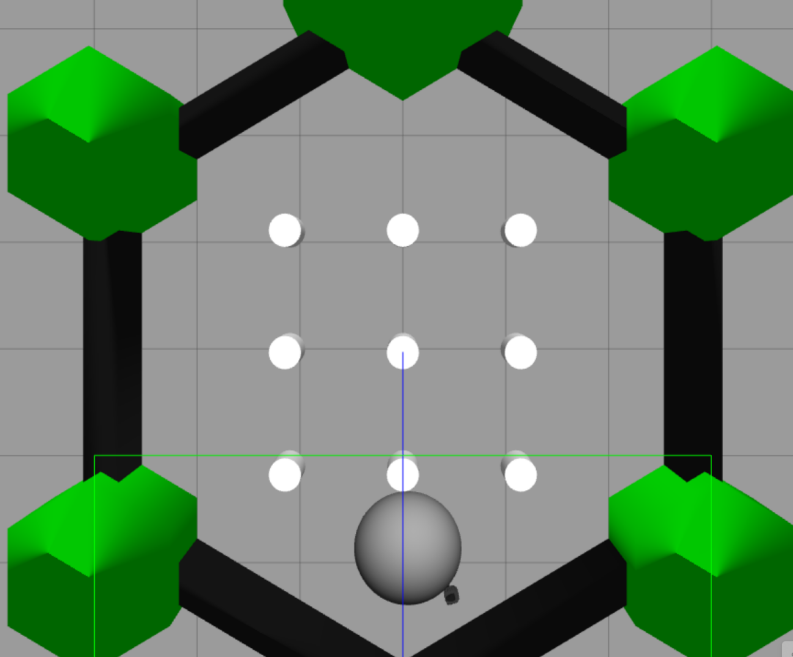
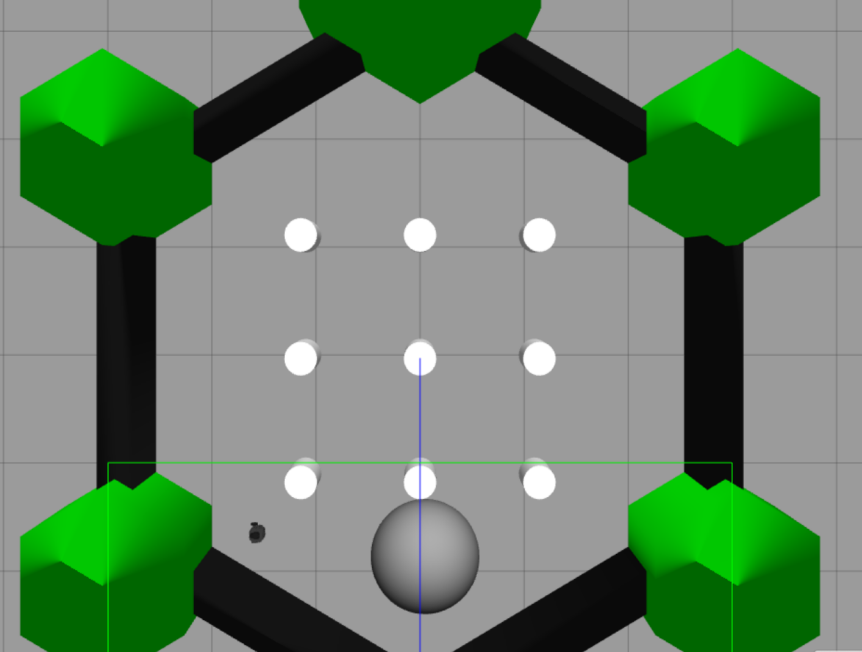
**在Gazebo中，机器人成功到达目标位置**



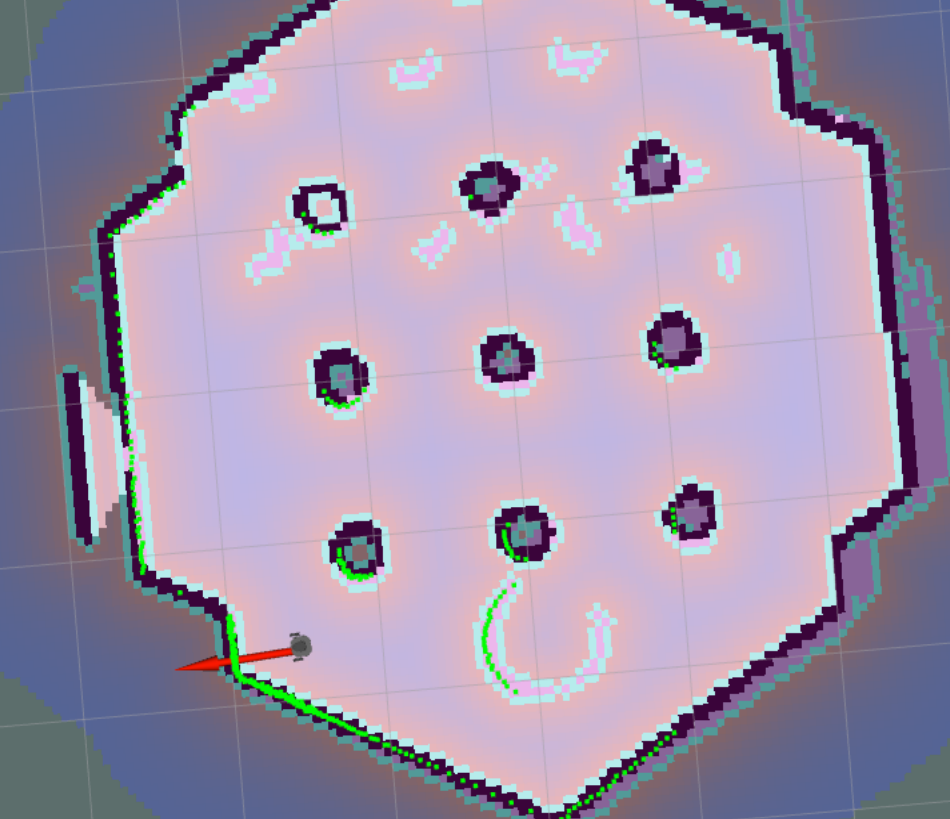
（3）动态障碍物避让

在仿真环境中添加动态障碍物，测试机器人能否实时避开动态障碍物，并继续前往目标。通过调整局部规划器的参数，优化机器人的避障效果和路径规划速度。

**机器人运动过程中在Gazebo中添加障碍物**



**在RViz中，机器人成功绕过障碍物到达目标位置**



# ROS网络通信实验-发布与订阅节点的数据传输

## 实验内容

本实验旨在帮助学生理解ROS中的发布-订阅机制，通过创建多个ROS节点，实现数据的发布与订阅，探索不同节点之间如何通过话题进行通信。学生将编写简单的发布者和订阅者节点，观察消息的传递过程，并使用工具可视化ROS系统的拓扑结构。

## 实验步骤

**（1）环境准备**

确保 ROS 系统（如 ROS Noetic）已成功安装，并已创建工作空间。

创建一个新的ROS包，用于存放本实验的节点代码。

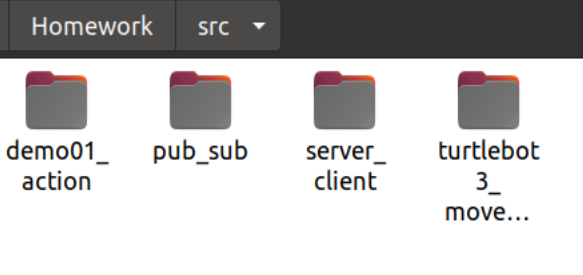
mkdir -p Homework/src #创建工作空间Homework

cd Homework #进入工作空间

catkin\_make #编译

cd src #进入src

catkin\_create\_pkg pub\_sub roscpp rospy std\_msgs #创建pub\_sub功能包



**（2）编写发布者节点**

创建一个发布者节点，该节点定期发布一个整数计数器（从0开始，每秒增加1），将其发布到特定的话题上（自己命名）。

**发布者节点代码：homework\_pub1.cpp**

/\* 1.包含头文件 \*/

#include "ros/ros.h"

#include "std\_msgs/Int32.h"

int main(int argc, char \*argv[])

{

    //防止中文乱码

    setlocale(LC\_ALL,"");

    /\* 2.初始化ROS节点 (命名唯一) \*/

    ros::init(argc,argv,"talker1");

    /\* 3.实例化ROS节点句柄 \*/

    ros::NodeHandle nh;

    /\* 4.实例化发布者对象 \*/

    ros::Publisher pub = nh.advertise<std\_msgs::Int32>("Counter",10);//共享话题，命名唯一

    /\* 5.组织被发布的数据，并编写逻辑发布数据 \*/

    /\* 创建一个Int32类型的消息 \*/

    std\_msgs::Int32 msg;

    int count = 1;//给消息计数器赋初值

    /\* 频率：一秒一次 \*/

    ros::Rate r(1);

    /\* 循环发布 \*/

    while(ros::ok())//节点不死

    {

        /\* 给msg.data赋值 \*/

        msg.data = count;

        /\* 发布消息 \*/

        pub.publish(msg);

        /\* 设置文本的前缀 \*/

        ROS\_INFO("%d", msg.data);

        /\* 自动休眠 休眠时间=1/频率 \*/

        r.sleep();

        count++;//循环结束前，让 count 自增

        /\* 暂无应用 \*/

        ros::spinOnce();

    }

    return 0;

}

（3）**编写订阅者节点**

创建一个订阅者节点，订阅话题，接收来自发布者的整数消息，并将其打印到控制台。

**订阅者节点代码：homework\_sub.cpp**

// 1.包含头文件

#include "ros/ros.h"

#include "std\_msgs/String.h"

#include "std\_msgs/Int32.h"

/\* 定义回调函数doMsg1（）接收整数消息 \*/

void doMsg1(const std\_msgs::Int32::ConstPtr &msg\_p1)

{

    ROS\_INFO("这是订阅方收到的整数消息：%d",msg\_p1->data);//设置文本的前缀

}

/\* 定义回调函数doMsg2（）接受字符串消息 \*/

void doMsg2(const std\_msgs::String::ConstPtr &msg\_p2)

{

    ROS\_INFO("这是订阅方收到的消息：%s",msg\_p2->data.c\_str());//设置文本的前缀

}

int main(int argc, char \*argv[])

{

    //防止中文乱码

    setlocale(LC\_ALL,"");

    /\* 2.初始化ROS节点 (命名唯一) \*/

    ros::init(argc,argv,"listener");

    /\* 3.实例化ROS节点句柄 \*/

    ros::NodeHandle nh;

    /\* 4.实例化订阅者对象 \*/

    ros::Subscriber sub1 = nh.subscribe<std\_msgs::Int32>("Counter",10,doMsg1);//接收整数消息

    ros::Subscriber sub2 = nh.subscribe<std\_msgs::String>("String",10,doMsg2);//接收字符串消息

    /\* 5.设置循环调用doMsg() \*/

    ros::spin();//循环读取接收的数据，并调用回调函数doMsg()处理

    return 0;

}

1. **运行ROS节点**

roscore #打开ros核心

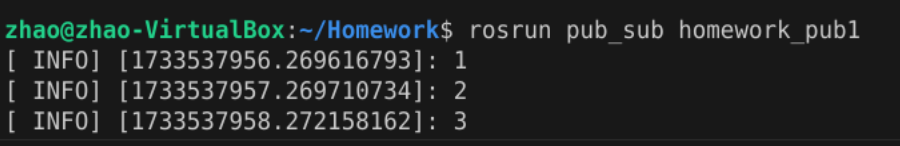
rosrun pub\_sub homework\_pub1 #运行发布者节点

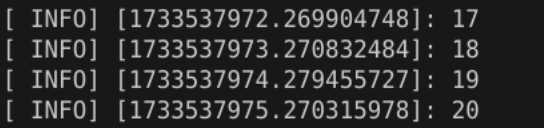
rosrun pub\_sub homework\_sub #运行订阅者节点

（5）**观察数据传输**

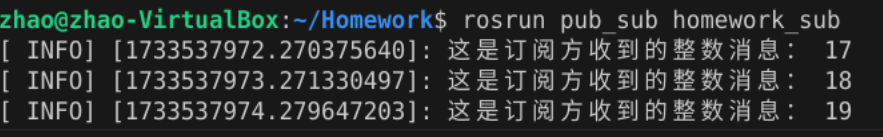
在订阅者的控制台中，观察到来自发布者的整数计数消息被成功接收并打印。

发布者节点发布整数计数消息：



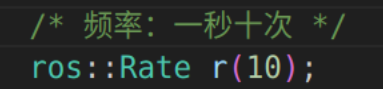


订阅者成功接收整数计数消息并打印：

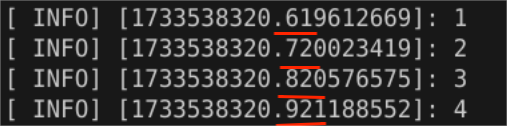


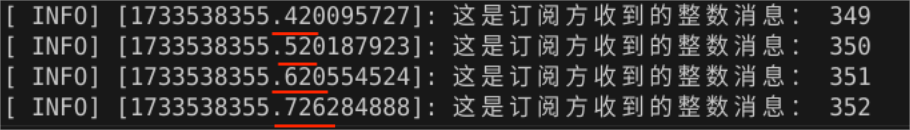
通过调整发布频率或改变发布的数据类型，观察对消息传递的影响。

在**homework\_pub1.cpp**中修改发布频率为10（之前是1）



观察发布消息和接收消息的时间，每0.1秒发送或接收一次消息

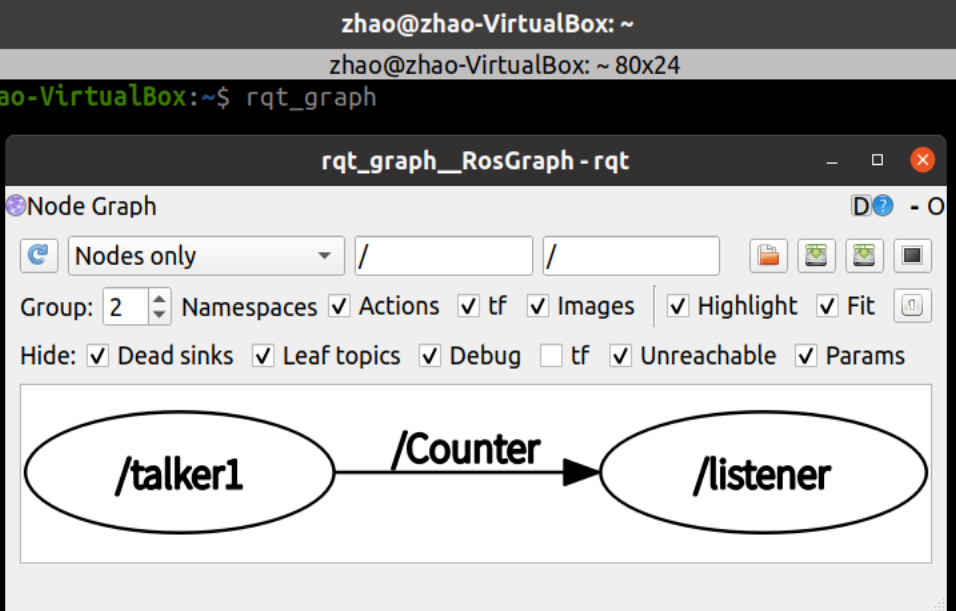




**（6）可视化与调试**

使用rqt\_graph工具可视化ROS节点之间的关系。

rqt\_graph #打开计算图



可以清楚地看到，发布者节点talker1发布了Counter话题，订阅者节点listener订阅了该话题。

**（7）扩展任务**

添加更多的发布者和订阅者节点，形成复杂的消息传递结构。例如，创建一个新的发布者节点发布字符串消息，观察订阅者如何接收不同类型的消息。

字符串发布节点代码：homework\_pub2.cpp

/\* 1.包含头文件 \*/

#include "ros/ros.h"

#include "std\_msgs/String.h" //普通文本类型的消息

#include <sstream>

int main(int argc, char \*argv[])

{

    //防止中文乱码

    setlocale(LC\_ALL,"");

    /\* 2.初始化ROS节点 (命名唯一) \*/

    ros::init(argc,argv,"talker2");

    /\* 3.实例化ROS节点句柄 \*/

    ros::NodeHandle nh;

    /\* 4.实例化发布者对象 \*/

    ros::Publisher pub = nh.advertise<std\_msgs::String>("String",10);

    /\* 5.组织被发布的数据，并编写逻辑发布数据 \*/

    /\* 创建一个字符串类型的消息 \*/

    std\_msgs::String msg;

    std::string msg\_string = "字符串消息abcde";//定义一个字符串变量储存字符串消息

    int count = 1;//给消息计数器赋初值

    /\* 频率：一秒一次 \*/

    ros::Rate r(1);

    /\* 循环发布 \*/

    while(ros::ok())//节点不死

    {

        //使用 stringstream 拼接字符串与编号

        std::stringstream ss;

        ss << msg\_string << " "<< count;

        msg.data = ss.str();

        /\* 发布消息 \*/

        pub.publish(msg);

        /\* 设置文本的前缀 \*/

        ROS\_INFO("这是发布者2发送的消息:%s",msg.data.c\_str());

        /\* 自动休眠 休眠时间=1/频率 \*/

        r.sleep();

        count++;//循环结束前，让 count 自增

        //暂无应用

        ros::spinOnce();

    }

    return 0;

}

订阅者节点已经编写接受字符串消息的代码。

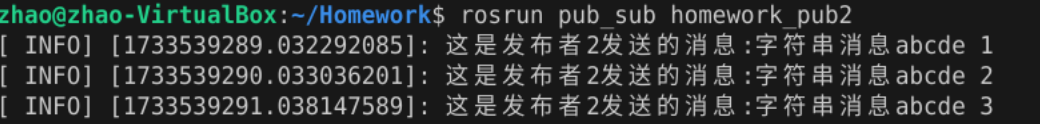
**运行ROS节点**：

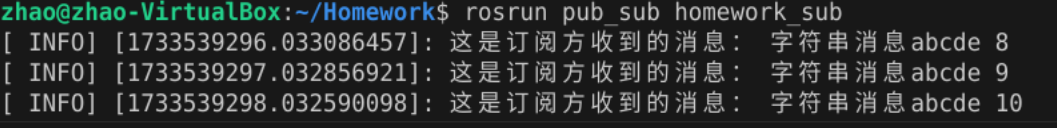
roscore #打开ros核心

rosrun pub\_sub homework\_pub2 #运行字符串发布者节点

rosrun pub\_sub homework\_sub #运行订阅者节点

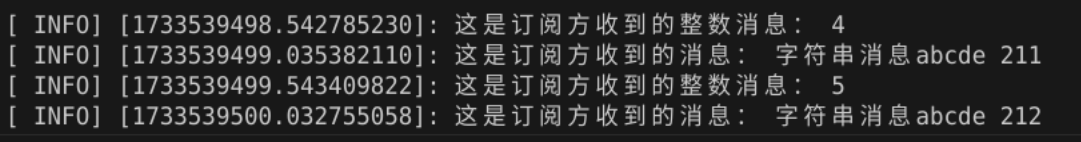
**观察数据传输：**

****

****

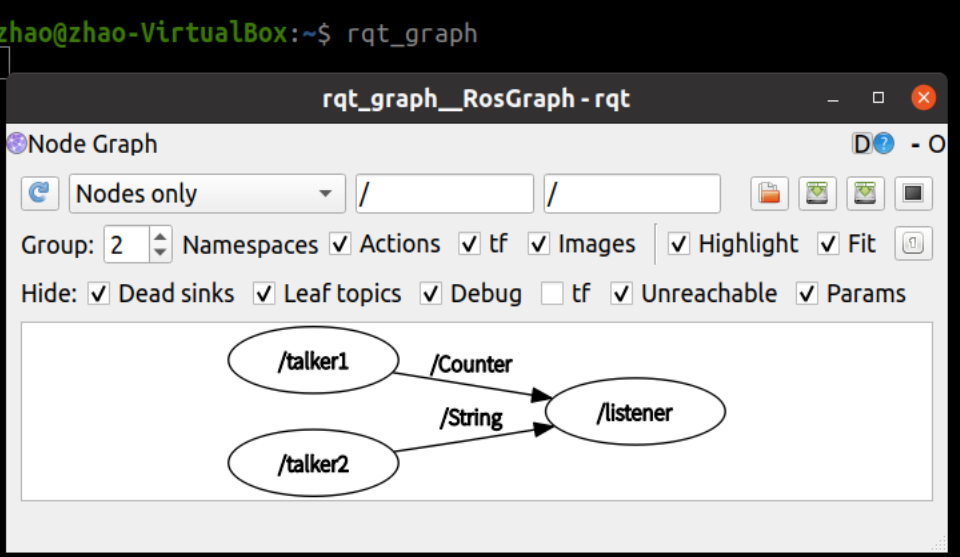
可以看到，发布者节点和订阅者节点成功发布和接收了字符串消息。

**再次运行整数发布者节点，观察订阅者节点接收的数据：**

****

可以看到，订阅者节点成功接收了字符串消息和整数消息。

**再次打开计算图**：



可以清楚地看到，整数发布者talker1节点发布了Counter话题，字符串发布者节点talker2发布了String话题，而订阅者节点listener订阅了这两个话题。

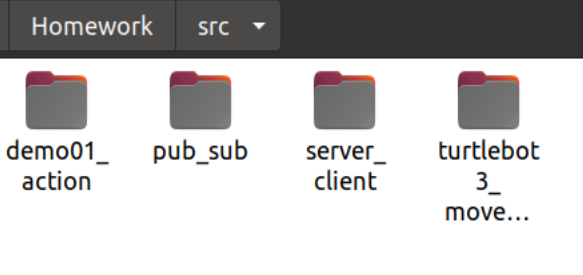
**探索ROS的服务（Service）和动作（Action）机制，与发布-订阅模型进行对比，理解不同通信方式的适用场景。**

1. **服务（Service）**

cd src #进入src

#创建server\_client功能包

catkin\_create\_pkg server\_client roscpp rospy std\_msgs



在server\_client功能包下建立srv目录，编写自定义srv文件：

**AddInts.srv**

# 客户端请求时发送的两个数字

int32 num1

int32 num2

--- #分割

# 服务器响应发送的数据

int32 sum

**服务端节点**：homework\_server.cpp

/\*

    需求:

        编写两个节点实现服务通信，客户端节点需要提交两个整数到服务器

        服务器需要解析客户端提交的数据，相加后，将结果响应回客户端，

        客户端再解析

\*/

//1.包含头文件

#include "ros/ros.h"

#include "server\_client/AddInts.h"

//回调函数doReq()

//返回的布尔值标志着是否处理成功

bool doReq(server\_client::AddInts::Request &req,//声明了一个名为 req 的变量，它是 Request 类型的一个引用。

          server\_client::AddInts::Response &resp)

{

    //处理请求

    int num1 = req.num1;

    int num2 = req.num2;

    ROS\_INFO("服务器接收到的请求数据为:num1 = %d, num2 = %d",num1, num2);

    //逻辑处理

    if (num1 < 0 || num2 < 0)

    {

        ROS\_ERROR("提交的数据异常:数据不可以为负数");

        return false;

    }

    //如果没有异常，那么相加并将结果赋值给 resp

    int sum = num1 + num2;

    resp.sum = sum;

    return true;

}

int main(int argc, char \*argv[])

{

    //防止中文乱码

    setlocale(LC\_ALL,"");

    // 2.初始化 ROS 节点

    ros::init(argc,argv,"AddInts\_Server");

    // 3.创建 ROS 句柄

    ros::NodeHandle nh;

    // 4.创建 服务 对象

    ros::ServiceServer server = nh.advertiseService("AddInts",doReq);//共享服务AddInts，回调函数doReq

    ROS\_INFO("服务已经启动....");

    //     5.回调函数处理请求并产生响应

    //     6.由于请求有多个，需要调用 ros::spin()

    ros::spin();

    return 0;

}

**客户端节点**：homework\_client.cpp

/\*

    需求:

        编写两个节点实现服务通信，客户端节点需要提交两个整数到服务器

        服务器需要解析客户端提交的数据，相加后，将结果响应回客户端，

        客户端再解析

\*/

// 1.包含头文件

#include "ros/ros.h"

#include "server\_client/AddInts.h"

int main(int argc, char \*argv[])//char \*argv[]是一个指针数组，每个元素都存储着前面传入参数的地址

{

    //argv[0]存放的是程序的路径，我理解的就是执行命令中的 包名+文件名

    //argv[1]和argv[2]存放的是要提交的两个整数

    //防止中文乱码

    setlocale(LC\_ALL,"");

    // 调用时动态传值,如果通过 launch 的 args 传参，需要传递的参数个数 +3

    if (argc != 3)

    // if (argc != 5)//launch 传参(0-文件路径 1传入的参数 2传入的参数 3节点名称 4日志路径)

    {

        ROS\_ERROR("请提交两个整数");

        return 1;

    }

    //执行命令，提交100和200两个整数

    //rosrun server\_client homework\_client 100 200

    // 2.初始化 ROS 节点

    ros::init(argc,argv,"AddInts\_Client");

    // 3.创建 ROS 句柄

    ros::NodeHandle nh;

    // 4.创建 客户端 对象

    ros::ServiceClient client = nh.serviceClient<server\_client::AddInts>("AddInts");//共享服务AddInts

    //如果先启动客户端再启动服务端，会导致运行失败，解决方法：先挂起客户端，然后等待服务端启动

    //方式1

    ros::service::waitForService("AddInts");

    //方式2

    // client.waitForExistence();

    // 5.组织请求数据

    server\_client::AddInts ai;

    ai.request.num1 = atoi(argv[1]);//将字符串转换成整型

    ai.request.num2 = atoi(argv[2]);

    // ai.request.num1 = 100;

    // ai.request.num2 = 200;

    // 6.发送请求,返回布尔值，标记是否成功

    bool flag = client.call(ai);

    // 7.处理响应

    if (flag)

    {

        ROS\_INFO("请求正常处理,响应结果:%d",ai.response.sum);

    }

    else

    {

        ROS\_ERROR("请求处理失败....");

        return -1;

    }

    return 0;

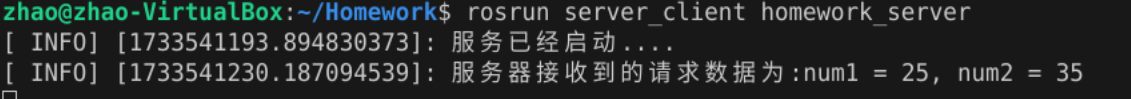
}

**运行ROS节点：**

roscore #打开ros核心

rosrun server\_client homework\_server #运行服务端节点

rosrun server\_client homework\_client 25 35 #运行客户端节点



Snipaste_2024-12-07_11-14-26

可以看到，服务端先启动服务，然后客户端提交两个整数25和35给服务端，服务端接收数据后进行相加把结果60返回给客户端。

1. **动作（Action）**

cd src #进入src

#创建demo01\_action功能包

catkin\_create\_pkg demo01\_action roscpp rospy std\_msgs actionlib actionlib\_msgs

在demo01\_action功能包下创建action目录，编写action文件

**AddInts.action**

#目标值

int32 num

---

#最终结果

int32 result

---

#连续反馈

float64 progress\_bar

**Action服务端节点：action01\_server.cpp**

#include "ros/ros.h"

#include "actionlib/server/simple\_action\_server.h"

#include "demo01\_action/AddIntsAction.h"

/\*

需求:

创建两个ROS节点，服务端和客户端，

客户端可以向服务端发送目标数据N（一个整型数据）

服务端会计算1到N之间所有整数的和，这是一个循环累加的过程，返回给客户端，

这是基于请求响应模式的，

又已知服务端从接收到请求到产生响应是一个耗时操作，每累加一次耗时0.1s，

为了良好的用户体验，需要服务端在计算过程中，

每累加一次，就给客户端响应一次百分比格式的执行进度，使用action实现。

流程:

1.包含头文件;

2.初始化ROS节点;

3.创建NodeHandle;

4.创建action服务对象;

5.处理请求,产生反馈与响应;

6.spin().

\*/

typedef actionlib::SimpleActionServer<demo01\_action::AddIntsAction> Server;//重命名为Server

//回调函数

void cb(const demo01\_action::AddIntsGoalConstPtr &goal,Server\* server){

//获取目标值

int num = goal->num;

ROS\_INFO("客户端提交的目标值:%d",num);

//累加并响应连续反馈

int result = 0;

demo01\_action::AddIntsFeedback feedback;//连续反馈

ros::Rate rate(10);//通过频率设置休眠时间

for (int i = 1; i <= num; i++)

{

result += i;

//组织连续数据并发布

feedback.progress\_bar = i / (double)num;//百分比

server->publishFeedback(feedback);

rate.sleep();

}

//设置最终结果

demo01\_action::AddIntsResult r;

r.result = result;

server->setSucceeded(r);

ROS\_INFO("最终结果:%d",r.result);

}

int main(int argc, char \*argv[])

{

//防止中文乱码

setlocale(LC\_ALL,"");

ROS\_INFO("action服务端实现");

// 2.初始化ROS节点;

ros::init(argc,argv,"AddInts\_server");

// 3.创建NodeHandle;

ros::NodeHandle nh;

// 4.创建action服务对象;

/\*SimpleActionServer(ros::NodeHandle n,

std::string name,

boost::function<void (const demo01\_action::AddIntsGoalConstPtr &)> execute\_callback,

bool auto\_start)

\*/

// actionlib::SimpleActionServer<demo01\_action::AddIntsAction> server(....);

Server server(nh,"addInts",boost::bind(&cb,\_1,&server),false);

server.start();//如果auto\_start为false，需要手动调用该函数启动

// 5.处理请求,产生反馈与响应;

// 6.spin().

ros::spin();

return 0;

}

**Action客户端节点：action02\_client.cpp**

#include "ros/ros.h"

#include "actionlib/client/simple\_action\_client.h"

#include "demo01\_action/AddIntsAction.h"

/\*

需求:

创建两个ROS节点，服务器和客户端，

客户端可以向服务器发送目标数据N（一个整型数据）

服务器会计算1到N之间所有整数的和，这是一个循环累加的过程，返回给客户端，

这是基于请求响应模式的，

又已知服务器从接收到请求到产生响应是一个耗时操作，每累加一次耗时0.1s，

为了良好的用户体验，需要服务器在计算过程中，

每累加一次，就给客户端响应一次百分比格式的执行进度，使用action实现。

流程:

1.包含头文件;

2.初始化ROS节点;

3.创建NodeHandle;

4.创建action客户端对象;

5.发送目标，处理反馈以及最终结果;

6.spin().

\*/

//重命名为Client

typedef actionlib::SimpleActionClient<demo01\_action::AddIntsAction> Client;

/\* 三个回调函数 \*/

//处理最终结果

void done\_cb(const actionlib::SimpleClientGoalState &state, const demo01\_action::AddIntsResultConstPtr &result){

if (state.state\_ == state.SUCCEEDED)

{

ROS\_INFO("最终结果:%d",result->result);

} else {

ROS\_INFO("任务失败！");

}

}

//服务已经激活

void active\_cb(){

ROS\_INFO("服务已经被激活....");

}

//处理连续反馈

void feedback\_cb(const demo01\_action::AddIntsFeedbackConstPtr &feedback){

ROS\_INFO("当前进度:%.2f",feedback->progress\_bar);

}

int main(int argc, char \*argv[])

{

//防止中文乱码

setlocale(LC\_ALL,"");

// 2.初始化ROS节点;

ros::init(argc,argv,"AddInts\_client");

// 3.创建NodeHandle;

ros::NodeHandle nh;

// 4.创建action客户端对象;

// SimpleActionClient(ros::NodeHandle & n, const std::string & name, bool spin\_thread = true)

// actionlib::SimpleActionClient<demo01\_action::AddIntsAction> client(nh,"addInts");

Client client(nh,"addInts",true);

//等待服务启动

client.waitForServer();

// 5.发送目标，处理反馈以及最终结果;

/\*

void sendGoal(const demo01\_action::AddIntsGoal &goal,

boost::function<void (const actionlib::SimpleClientGoalState &state, const demo01\_action::AddIntsResultConstPtr &result)> done\_cb,

boost::function<void ()> active\_cb,

boost::function<void (const demo01\_action::AddIntsFeedbackConstPtr &feedback)> feedback\_cb)

\*/

demo01\_action::AddIntsGoal goal;

goal.num = 10;

client.sendGoal(goal,&done\_cb,&active\_cb,&feedback\_cb);

// 6.spin().

ros::spin();

return 0;

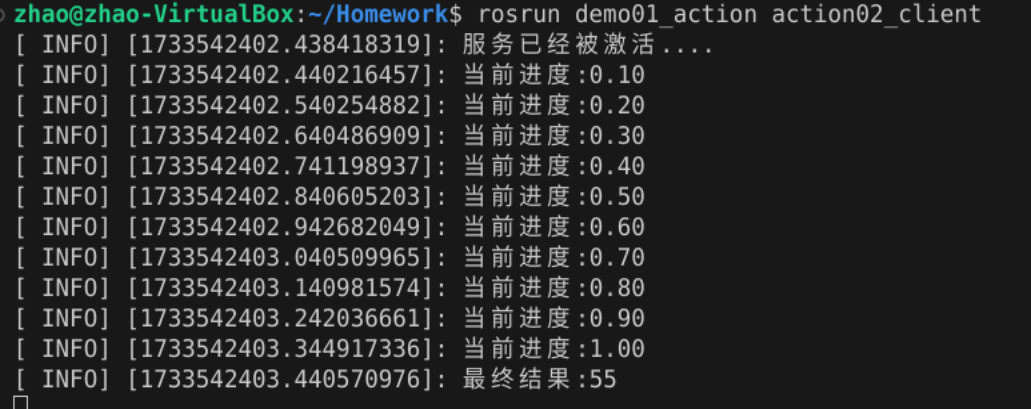
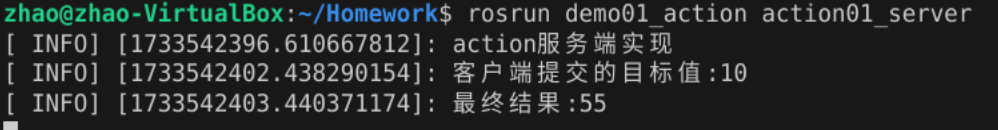
}

**运行ROS节点**：

roscore #打开ros核心

rosrun demo01\_action action01\_server #运行Action服务端节点

rosrun demo01\_action action02\_client #运行Action客户端节点



可以看到，Action服务端先启动服务，然后再启动Action客户端，客户端在代码中提交了目标值num为10，服务端接收到num后进行处理，将连续反馈 progress\_bar的值和最终结果result（从1加到10）返回给客户端。

**总结：ROS中话题通信、服务通信、Action通信的区别**

**一、话题通信**

模型：话题通信采用发布/订阅模型，即一个节点可以发布消息，其他节点可以订阅这些消息。

通信方式：异步通信，消息发送后，发布者不需要等待接收者的响应。

适用场景：适合用于需要频繁更新数据的情况，比如传感器数据（如激光雷达、相机图像等）。在这种场景下，数据流的更新频率较高，且数据通常是由多个节点共享或处理的。

**二、服务通信**

模型：服务通信使用请求/响应机制，即一个节点可以发送请求，另一个节点处理请求后返回响应。

通信方式：同步通信，请求方需要等待服务方的处理结果。

适用场景：适合处理单次请求和响应的任务，比如获取某个数据的状态或执行某个动作。在这种场景下，通常有一个明确的请求者和一个服务提供者，且请求者需要等待服务提供者的响应来完成任务。

**三、Action通信**

模型：Action通信是基于服务的扩展，支持长时间运行的任务。它结合了话题和服务的特点，提供了一个更加灵活和强大的方式来控制和管理机器人的复杂行为。

通信方式：异步的请求/反馈模型，调用方可以获得执行进度的反馈，也可以在执行过程中取消动作。

特点：

目标（Goal）：由Action客户端发送给Action服务器的请求，指示服务器需要执行的具体任务。

反馈（Feedback）：在执行目标的过程中，Action服务器会定期向客户端发送反馈，以报告当前的执行状态和进度。

结果（Result）：当目标执行完成后，Action服务器会向客户端发送一个结果，表明任务的最终执行状态。

可抢占性：与服务不同，Action是可抢占的。这意味着在执行过程中，客户端可以随时取消当前的目标，或者发送一个新的目标来覆盖旧的目标。

适用场景：适合执行需要较长时间的任务，比如移动到某个位置或执行复杂的操作。在这种场景下，任务通常涉及多个步骤或阶段，且需要实时反馈和可能的取消操作。

**总结**

话题通信适合频繁更新的数据流，服务通信适合单次请求和响应的任务，而Action通信则用于需要较长时间且需要反馈和取消操作的任务。在ROS系统中，这三种通信机制可以根据实际需求进行选择和组合使用，以实现高效、灵活和可靠的机器人系统通信。

# 实验心得

在本次《Linux系统基础》实验中，我将理论知识运用于实践，学会了如何使用Gazebo和Rviz来生成机器人模型和模拟仿真环境，使用SLAM建图并实现了机器人的动态规划路径以及自动避障功能。除此之外，我还深刻认识到ROS中不同的节点是如何进行通信的，了解到话题通信、服务通信、Action通信的区别，并掌握了相关节点文件的编写。