

序言

1.课程内容

2.项目产出

3.答疑

4.写在最后

第1章 ROS概述与环境搭建

1.1 ROS简介

1.1.1ROS概念

1.1.2ROS设计目标

1.1.3ROS发展历程

1.2 ROS安装

1.2.1 安装虚拟机软件

1.下载virtualbox

2.安装virtualbox

1.2.2 虚拟一台主机

1.2.3 安装ubuntu

1.ubuntu安装

2.使用优化

①安装虚拟机工具

②启动文件交换模式

③安装扩展插件

④其他

1.2.4 安装 ROS

1.配置ubuntu的软件和更新

2.设置安装源

3.设置key

4.安装

5.配置环境变量

6.卸载

后记

6.安装构建依赖

1.2.5 测试 ROS

1.2.6 资料:其他ROS版本安装

1.配置ubuntu的软件和更新

2.安装源

3.设置key

4.安装

5.环境设置

6.安装构建依赖

1.3 ROS快速体验

1.3.1 HelloWorld实现简介

1.创建工作空间并初始化

2.进入 src 创建 ros 包并添加依赖

1.3.2 HelloWorld(C++版)

1.进入 ros 包的 src 目录编辑源文件

2.编辑 ros 包下的 Cmakelist.txt文件

3.进入工作空间目录并编译

4.执行

1.3.3 HelloWorld(Python版)

1.进入 ros 包添加 scripts 目录并编辑 python 文件

2.为 python 文件添加可执行权限

3.编辑 ros 包下的 CamkeList.txt 文件

4.进入工作空间目录并编译

5.进入工作空间目录并执行

1.4 ROS集成开发环境搭建

1.4.1 安装终端

1.安装

2.添加到收藏夹

3.Terminator 常用快捷键

1.4.2 安装VScode

1.下载

2.vscode 安装与卸载

2.1 安装

2.2 卸载

3.vscode 集成 ROS 插件

4.vscode 使用_基本配置

4.1 创建 ROS 工作空间

4.2 启动 vscode

- 4.3 vscode 中编译 ros
- 4.4 创建 ROS 功能包
- 4.5 C++ 实现
- 4.6 python 实现
- 4.7 配置 CMakeLists.txt
- 4.8 编译执行

5.其他 IDE

1.4.3 launch文件演示

- 1.需求
- 2.实现

1.5 ROS架构

- 1.设计者
- 2.维护者
- 3.系统架构
- 4.自身结构

1.5.1 ROS文件系统

- 1.package.xml
- 2.CMakelists.txt

1.5.2 ROS文件系统相关命令

- 1.增
- 2.删
- 3.查
- 4.改
- 5.执行
 - 5.1roscore
 - 5.2rosrun
 - 5.3roslaunch

1.5.3 ROS计算图

- 1.计算图简介
- 2.计算图安装
- 3.计算图演示

1.6 本章小结

第2章 ROS通信机制

2.1 话题通信

2.1.1 理论模型

- 0.Talker注册
- 1.Listener注册
- 2.ROS Master实现信息匹配
- 3.Listener向Talker发送请求
- 4.Talker确认请求
- 5.Listener与Talker件里连接
- 6.Talker向Listener发送消息

2.1.2 话题通信基本操作A(C++)

- 1.发布方
- 2.订阅方
- 3.配置 CMakeLists.txt
- 4.执行
- 5.注意

2.1.3 话题通信基本操作B(Python)

- 1.发布方
- 2.订阅方
- 3.添加可执行权限
- 4.配置 CMakeLists.txt
- 5.执行

2.1.4 话题通信自定义msg

- 1.定义msg文件
- 2.编辑配置文件
- 3.编译

2.1.5 话题通信自定义msg调用A(C++)

- 0.vscode 配置
- 1.发布方
- 2.订阅方
- 3.配置 CMakeLists.txt
- 4.执行

2.1.6 话题通信自定义msg调用B(Python)

- 0.vscode配置
- 1.发布方
- 2.订阅方

3.权限设置

4.配置 CMakeLists.txt

5.执行

2.2 服务通信

2.2.1 服务通信理论模型

0.Server注册

1.Client注册

2.ROS Master实现信息匹配

3.Client发送请求

4.Server发送响应

2.2.2 服务通信自定义srv

1.定义srv文件

2.编辑配置文件

3.编译

2.2.3 服务通信自定义srv调用A(C++)

0.vscode配置

1.服务端

2.客户端

3.配置 CMakeLists.txt

4.执行

2.2.4 服务通信自定义srv调用B(Python)

0.vscode配置

1.服务端

2.客户端

3.设置权限

4.配置 CMakeLists.txt

5.执行

2.3 参数服务器

2.3.1 参数服务器理论模型

1.Talker 设置参数

2.Listener 获取参数

3.ROS Master 向 Listener 发送参数值

2.3.2 参数操作A(C++)

1.参数服务器新增(修改)参数

2.参数服务器获取参数

3.参数服务器删除参数

2.3.3 参数操作B(Python)

1.参数服务器新增(修改)参数

2.参数服务器获取参数

3.参数服务器删除参数

2.4 常用命令

2.4.1 rosnode

2.4.2 rostopic

2.4.3 rosmsg

2.4.4 rosservice

2.4.5 rossrv

2.4.6 rosparam

2.5 通信机制实操

2.5.1 实操01_话题发布

1.话题与消息获取

1.1话题获取

1.2消息获取

2.实现发布节点

3.运行

2.5.2 实操02_话题订阅

1.话题与消息获取

2.实现订阅节点

3.运行

2.5.3 实操03_服务调用

1.服务名称与服务消息获取

2.服务客户端实现

3.运行

2.5.4 实操04_参数设置

1.参数名获取

2.参数修改

3.运行

4.其他设置方式

2.6 通信机制比较

2.7 本章小结

第3章 ROS通信机制进阶

3.1 常用API

3.1.1 初始化

C++

初始化

Python

初始化

3.1.2 话题与服务相关对象

C++

1.发布对象

对象获取:

消息发布函数:

2.订阅对象

对象获取:

3.服务对象

对象获取:

4.客户端对象

对象获取:

请求发送函数:

等待服务函数1:

等待服务函数2:

Python

1.发布对象

对象获取:

消息发布函数:

2.订阅对象

对象获取:

3.服务对象

对象获取:

4.客户端对象

对象获取:

请求发送函数:

等待服务函数:

3.1.3 回旋函数

C++

1.spinOnce()

2.spin()

3.二者比较

Python

3.1.4 时间

C++

1.时刻

2.持续时间

3.持续时间与时刻运算

4.设置运行频率

5.定时器

Python

1.时刻

2.持续时间

3.持续时间与时刻运算

4.设置运行频率

5.定时器

3.1.5 其他函数

C++

Python

3.2 ROS中的头文件与源文件

3.2.1 自定义头文件调用

1.头文件

2.可执行文件

3.配置文件

3.2.2 自定义源文件调用

1.头文件

2.源文件

3.可执行文件

4.配置文件

3.3 Python模块导入

- 1.新建两个Python文件并使用import导入
- 2.添加可执行权限，编辑配置文件并执行

第4章 ROS运行管理

4.1 ROS元功能包

4.2 ROS节点运行管理launch文件

1.新建launch文件

2.调用 launch 文件

4.2.1 launch文件标签之launch

1.属性

2.子级标签

4.2.2 launch文件标签之node

1.属性

2.子级标签

4.2.3 launch文件标签之include

1.属性

2.子级标签

4.2.4 launch文件标签之remap

1.属性

2.子级标签

4.2.5 launch文件标签之param

1.属性

2.子级标签

4.2.6 launch文件标签之rosparam

1.属性

2.子级标签

4.2.7 launch文件标签之group

1.属性

2.子级标签

4.2.8 launch文件标签之arg

1.属性

2.子级标签

3.示例

4.3 ROS工作空间覆盖

4.4 ROS节点名称重名

4.4.1 rosrun设置命名空间与重映射

1.rosrun设置命名空间

1.1设置命名空间演示

1.2运行结果

2.rosrun名称重映射

2.1为节点起别名

2.2运行结果

3.rosrun命名空间与名称重映射叠加

3.1设置命名空间同时名称重映射

3.2运行结果

4.4.2 launch文件设置命名空间与重映射

1.launch文件

2.运行

4.4.3 编码设置命名空间与重映射

1.C++ 实现:重映射

1.1名称别名设置

1.2执行

2.C++ 实现:命名空间

2.1命名空间设置

2.2执行

3.Python 实现:重映射

3.1名称别名设置

3.2执行

4.5 ROS话题名称设置

4.5.1 rosrun设置话题重映射

1.方案1

2.方案2

4.5.2 launch文件设置话题重映射

1.方案1

2.方案2

4.5.3 编码设置话题名称

1.C++ 实现

1.1全局名称

1.2相对名称

1.3 私有名称

2. Python 实现

2.1 全局名称

2.2 相对名称

2.3 私有名称

4.6 ROS参数名称设置

4.6.1 rosrun设置参数

1. 设置参数

2. 运行

4.6.2 launch文件设置参数

1. 设置参数

2. 运行

4.6.3 编码设置参数

1. C++实现

1.1 ros::param设置参数

1.2 ros::NodeHandle设置参数

2. python实现

4.7 ROS分布式通信

4.8 本章小结

第5章 ROS常用组件

5.1 TF坐标变换

5.1.1 坐标msg消息

1. geometry_msgs/TransformStamped

2. geometry_msgs/PointStamped

5.1.2 静态坐标变换

1. 创建功能包

2. 发布方

3. 订阅方

4. 执行

1. 创建功能包

2. 发布方

3. 订阅方

4. 执行

补充1:

补充2:

5.1.3 动态坐标变换

1. 创建功能包

2. 发布方

3. 订阅方

4. 执行

1. 创建功能包

2. 发布方

3. 订阅方

4. 执行

5.1.4 多坐标变换

1. 创建功能包

2. 发布方

3. 订阅方

4. 执行

1. 创建功能包

2. 发布方

3. 订阅方

4. 执行

5.1.5 坐标系关系查看

6.1 准备

6.2 使用

6.2.1 生成 pdf 文件

6.2.2 查看文件

5.1.6 TF坐标变换实操

1. 创建功能包

2. 服务客户端(生成乌龟)

3. 发布方(发布两只乌龟的坐标信息)

4. 订阅方(解析坐标信息并生成速度信息)

5. 运行

1. 创建功能包

2. 服务客户端(生成乌龟)

3. 发布方(发布两只乌龟的坐标信息)

4. 订阅方(解析坐标信息并生成速度信息)

5.运行

5.1.7 TF2与TF

1.TF2与TF比较_简介

2.TF2与TF比较_静态坐标变换演示

2.1启动 TF2 与 TF 两个版本的静态坐标变换

2.2运行结果比对

2.3结论

5.1.8 小结

5.2 rosbag

5.2.1 rosbag使用_命令行

5.2.2 rosbag使用_编码

1.写 bag

2.读bag

1.写 bag

2.读bag

5.3 rqt工具箱

5.3.1 rqt安装启动与基本使用

1.安装

2.启动

3.基本使用

5.3.2 rqt常用插件:rqt_graph

5.3.3 rqt常用插件:rqt_console

5.3.4 rqt常用插件:rqt_plot

5.3.5 rqt常用插件:rqt_bag

5.4 本章小结

第6章 机器人系统仿真

6.1 概述

1.概念

2.作用

2.1仿真优势:

2.2仿真缺陷:

3.相关组件

3.1URDF

3.2rviz

3.3gazebo

6.2 URDF集成Rviz基本流程

1. 创建功能包，导入依赖
2. 编写 URDF 文件
3. 在 launch 文件中集成 URDF 与 Rviz
4. 在 Rviz 中显示机器人模型
5. 优化 rviz 启动

6.3 URDF语法详解

6.3.1 URDF语法详解01_robot

robot

1. 属性
2. 子标签

6.3.2 URDF语法详解02_link

link

1. 属性
2. 子标签
3. 案例

6.3.3 URDF语法详解03_joint

joint

1. 属性
2. 子标签
3. 案例
4. base_footprint优化urdf
5. 遇到问题以及解决

6.3.4 URDF练习

1. 新建urdf以及launch文件
2. 底盘搭建
3. 添加驱动轮
4. 添加万向轮

6.3.5 URDF工具

1. check_urdf 语法检查
2. urdf_to_graphiz 结构查看

6.4 URDF优化_xacro

6.4.1 Xacro_快速体验

- 1.Xacro文件编写
- 2.Xacro文件转换成 urdf 文件
- 6.4.2 Xacro_ 语法详解
 - 1.属性与算数运算
 - 2.宏
 - 3.文件包含
- 6.4.3 Xacro_ 完整使用流程示例
 - 1.编写 Xacro 文件
 - 2.集成launch文件
- 6.4.4 Xacro_ 实操
 - 1.摄像头和雷达 Xacro 文件实现
 - 2.组合底盘摄像头与雷达的 xacro 文件
 - 3.launch 文件
- 6.5 Rviz中控制机器人模型运动
 - 6.5.1 Arbotix使用流程
 - 1.安装 Arbotix
 - 2.创建新功能包，准备机器人 urdf、xacro
 - 3.添加 arbotix 所需的配置文件
 - 4.launch 文件中配置 arbotix 节点
 - 5.启动 launch 文件并控制机器人模型运动
- 6.6 URDF集成Gazebo
 - 6.6.1 URDF与Gazebo基本集成流程
 - 1.创建功能包
 - 2.编写URDF文件
 - 3.启动Gazebo并显示模型
 - 6.6.2 URDF集成Gazebo相关设置
 - 1.collision
 - 2.inertial
 - 3.颜色设置
 - 6.6.3 URDF集成Gazebo实操
 - 1.编写封装惯性矩阵算法的 xacro 文件
 - 2.复制相关 xacro 文件，并设置 collision inertial 以及 color 等参数
- A.底盘 Xacro 文件

B.摄像头 Xacro 文件

C.雷达 Xacro 文件

D.组合底盘、摄像头与雷达的 Xacro 文件

3.在 gazebo 中执行

6.6.4 Gazebo 仿真环境搭建

1.添加内置组件创建仿真环境

1.1 启动 Gazebo 并添加组件

1.2 保存仿真环境

1.3 启动

2.自定义仿真环境

2.1 启动 gazebo 打开构建面板，绘制仿真环境

2.2 保存构建的环境

2.3 保存为 world 文件

2.4 启动

3.使用官方提供的插件

3.1 下载官方模型库

3.2 将模型库复制进 gazebo

3.3 应用

6.7 URDF、Gazebo 与 Rviz 综合应用

6.7.1 机器人运动控制以及里程计信息显示

1.ros_control 简介

2.运动控制实现流程(Gazebo)

2.1 为 joint 添加传动装置以及控制器

2.2 xacro 文件集成

2.3 启动 gazebo 并控制机器人运动

3.Rviz 查看里程计信息

3.1 启动 Rviz

3.2 添加组件

6.7.2 雷达信息仿真以及显示

1.Gazebo 仿真雷达

1.1 新建 Xacro 文件，配置雷达传感器信息

1.2 xacro 文件集成

1.3 启动仿真环境

2.Rviz 显示雷达数据

6.7.3 摄像头信息仿真以及显示

1.Gazebo 仿真摄像头

1.1 新建 Xacro 文件，配置摄像头传感器信息

1.2 xacro 文件集成

1.3启动仿真环境

2.Rviz 显示摄像头数据

6.7.4 kinect信息仿真以及显示

1.Gazebo仿真Kinect

1.1 新建 Xacro 文件，配置 kinetic传感器信息

1.2 xacro 文件集成

1.3启动仿真环境

2 Rviz 显示 Kinect 数据

补充:kinect 点云数据显示

6.8 本章小结

第 7 章 机器人导航(仿真)

7.1 概述

7.1.1 导航模块简介

1.全局地图

2.自身定位

3.路径规划

4.运动控制

5.环境感知

7.1.2 导航之坐标系

1.简介

2.特点

3.坐标系变换

7.1.3 导航条件说明

1.硬件

2.软件

7.2 导航实现

7.2.1 导航实现01_SLAM建图

1.gmapping简介

2.gmapping节点说明

2.1订阅的Topic

2.2发布的Topic

2.3服务

2.4参数

2.5所需的坐标变换

2.6发布的坐标变换

3.gmapping使用

3.1编写gmapping节点相关launch文件

3.2执行

7.2.2 导航实现02_地图服务

1.map_server简介

2.map_server使用之地图保存节点(map_saver)

2.1map_saver节点说明

2.2地图保存launch文件

2.3 保存结果解释

3.map_server使用之地图服务(map_server)

3.1map_server节点说明

3.2地图读取

3.3地图显示

7.2.3 导航实现03_定位

1.amcl简介

2.amcl节点说明

3.1订阅的Topic

3.2发布的Topic

3.3服务

3.4调用的服务

3.5参数

3.6坐标变换

3.amcl使用

3.1编写amcl节点相关的launch文件

3.2编写测试launch文件

3.3执行

7.2.4 导航实现04_路径规划

1.move_base简介

2.move_base节点说明

2.1动作

2.2订阅的Topic

2.3发布的Topic

2.4服务

2.5参数

3.move_base与代价地图

3.1概念

3.2组成

3.3碰撞算法

4.move_base使用

4.1launch文件

4.2配置文件

4.2.1costmap_common_params.yaml

4.2.2global_costmap_params.yaml

4.2.3local_costmap_params.yaml

4.2.4base_local_planner_params

4.2.5参数配置技巧

4.3launch文件集成

4.4测试

7.2.5 导航与SLAM建图

1.编写launc文件

2.测试

7.3 导航相关消息

7.3.1 导航之地图

1.nav_msgs/MapMetaData

2.nav_msgs/OccupancyGrid

7.3.2 导航之里程计

7.3.3 导航之坐标变换

7.3.4 导航之定位

7.3.5 导航之目标点与路径规划

7.3.6 导航之激光雷达

7.3.7 导航之相机

7.3.8 深度图像转激光数据

1.depthimage_to_laserscan简介

1.1原理

1.2优缺点

1.3安装

2.depthimage_to_laserscan节点说明

2.1订阅的Topic

2.2发布的Topic

2.3参数

3.depthimage_to_laserscan使用

3.1编写launch文件

3.2修改URDF文件

3.3执行

4.SLAM应用

7.4 本章小结

第8章 机器人平台设计

8.1 概述

1.传感系统

2.控制系统

3.驱动系统

4.执行机构

8.2 机器人平台设计之arduino基础

8.2.1 arduino 开发环境搭建

1.Arduino 连接 Ubuntu

2.安装 Arduino IDE

1.下载arduino ide安装包

2.使用tar命令对压缩包解压

3.将解压后的文件移动到/opt下

4.进入安装目录,对install.sh添加可执行权限,并执行安装

5.启动并配置 Arduino IDE

3.Hello World实现

1.案例调用

2.编译及上传

3.运行结果

4.代码解释

8.2.2 arduino 基本语法概述

1.程序结构

2.常量

3.通信_Serial

4.函数_数字IO

5.函数_模拟IO

6.函数_时间

7.函数_中断

8.2.3 arduino 基本语法演示

1.通信实现01

2.通信实现02

8.2.4 arduino 基本语法演示02

1.数字IO操作

2.模拟IO操作

8.2.5 arduino 基本语法演示03

8.3 机器人平台设计之电机驱动

8.3.1 硬件_电机与电机驱动板

1.直流减速电机

2.电机驱动板

3.准备工作

8.3.2 电机基本控制实现

1.编码

2.运行

8.3.3 电机测速01_理论

1.概念

2.测速原理

3.测速举例

8.3.4 电机测速02_实现

1.编码实现脉冲统计

2.转速计算

3.测试

8.3.5 电机调速01_PID控制理论

PID简介

1.P

2.I

3.D

8.3.6 电机调速02_PID控制实现

1.添加Arduino-PID-Library

2.编码

4.调试

8.4 机器人平台设计之底盘实现

8.4.1 底盘实现_概述

1.ros_arduino_bridge 简介

2.ros_arduino_bridge 架构

3.案例实现

8.4.2 底盘实现_01Arduino端入口

1.串口命令

2.启用基座控制器

8.4.3 底盘实现_02Arduino端编码器驱动

1.定义编码器驱动

2.修改encoder_driver.h文件

3.修改encoder_driver.ino 文件

4.ROSArduinoBridge.ino 实现初始化

5.测试

8.4.4 底盘实现_03Arduino端电机驱动

1.定义电机驱动

2.修改motor_driver.h文件

3.修改motor_driver.ino 文件

4.测试

8.4.5 底盘实现_04Arduino端PID控制

1.ros_arduino_bridge中PID调试源码分析

2.PID调试

8.5 机器人平台设计之控制系统

8.5.1 控制系统实现_树莓派概述

概念

结构

配件

接线以及使用

8.5.2 控制系统实现_分布式框架

8.5.3 控制系统实现_ssh远程连接

概念

实现

1.安装SSH客户端与服务端

2.服务端启动SSH服务

3.客户端远程登陆服务端

4.实现数据传输

使用优化

1.生成密钥对

2.将公钥上传至树莓派

8.5.4 控制系统实现_安装ros_arduino_bridge

1.系统准备

2.程序修改

3.程序上传

4.测试

资料:控制系统实现_树莓派安装ROS

1.Ubuntu安装

1.1硬件准备

1.2软件准备

1.3系统烧录

1.4系统安装

2.ROS安装

1.配置软件与更新

2.设置安装源

3.设置key

4.安装

5.环境配置

6.构建软件包的依赖关系

8.6 机器人平台设计之传感器

8.6.1 传感器_激光雷达简介

概念

原理

优点

缺点

分类

8.6.2 传感器_雷达使用

1.硬件准备

1.雷达连接上位机

2.确认当前的 USB 转串口终端并修改权限

2.软件安装

3.启动并测试

1.rplidar.launch文件准备

2.终端中执行 launch 文件

3.rviz中订阅雷达相关消息

8.6.3 传感器_相机简介

1.单目相机

2.双目相机

3.深度相机

8.6.4 传感器_相机使用

1.硬件准备

2.软件准备

3.测试

1.launch文件准备

2.启动launch文件

3.rviz显示

8.6.5 传感器_集成

1.launch文件

2.坐标变换

3.测试

1.树莓派

2.PC端

3.结果显示

8.7 本章小结

第9章 机器人导航(实体)

9.1 概述

9.2 VScode远程开发

- 1.准备工作
- 2.为VScode安装远程开发插件
- 3.配置远程连接
- 4.使用

9.3 导航实现

9.3.1 导航实现01_准备工作

- 1.1分布式架构
- 1.2功能包安装
- 1.3机器人模型以及坐标变换
 - 1.3.1 创建机器人模型相关的功能包
 - 1.3.2 准备机器人模型文件
 - 1.3.3 在launch文件加载机器人模型
- 1.4结果演示

9.3.2 导航实现02_SLAM建图

- 2.1编写gmapping节点相关launch文件
- 2.2执行

9.3.3 导航实现03_地图服务

- 3.1地图保存launch文件
- 3.2地图读取

9.3.4 导航实现04_定位

- 4.1编写amcl节点相关的launch文件
- 4.2编写测试launch文件
- 4.3执行

9.3.5 导航实现05_路径规划

- 5.1编写launch文件
- 5.2编写配置文件
 - 1.costmap_common_params.yaml
 - 2.global_costmap_params.yaml
 - 3.local_costmap_params.yaml
 - 4.base_local_planner_params
- 5.3launch文件集成
- 5.4测试

9.3.6 导航与SLAM建图

6.1编写launch文件

6.2测试

9.4 本章小结

第 10 章 ROS进阶

10.1 action通信

10.1.1 action通信自定义action文件

1.定义action文件

2.编辑配置文件

3.编译

10.1.2 action通信自定义action文件调用A(C++)

0.vscode配置

1.服务端

2.客户端

3.编译配置文件

4.执行

10.1.3 action通信自定义action文件调用(Python)

0.vscode配置

1.服务端

2.客户端

3.编辑配置文件

4.执行

10.2 动态参数

10.2.1 动态参数客户端

1.新建功能包

2.添加.cfg文件

3.配置 CMakeLists.txt

4.编译

10.2.2 动态参数服务端A(C++)

0.vscode配置

1.服务器代码实现

2.编译配置文件

3.执行

10.2.3 动态参数服务端B(Python)

0.vscode配置

- 1.服务器代码实现
- 2.编辑配置文件
- 3.执行

10.3 pluginlib

10.3.1 pluginlib使用

- 1.准备
- 2.创建基类
- 3.创建插件
- 4.注册插件
- 5.构建插件库
- 6.使插件可用于ROS工具链
- 7.使用插件
- 8.执行

10.4 nodelet

10.4.1 使用演示

- 1.案例简介
- 2.nodelet 基本使用语法
- 3.内置案例调用
 - 1.启动roscore
 - 2.启动manager
 - 3.添加nodelet节点
 - 4.执行

10.4.2 nodelet实现

- 1.准备
- 2.创建插件类并注册插件
- 3.构建插件库
- 4.使插件可用于ROS工具链
 - 4.1配置xml
 - 4.2导出插件
- 5.执行

10.5 本章小结

序言

ROS(机器人操作系统)近几年发展迅速，国内也有相当一部分开发人员有意向涉足ROS，但是苦于没有低门槛的系统性教程，只能望之兴叹，基于此我们设计了一套**免费、零基础、理论与实践相结合**的教程，以帮助有志于机器人开发的童鞋方便快捷的上手ROS，继而推动整个行业的进步。

1. 课程内容

本教程主要由理论篇与实践篇组成，理论篇对应的是第1到第5章，实践篇对应的是第6章以及以后，具体内容如下：

理论篇

章节	内容
第1章 ROS概述与环境搭建	旨在了解ROS并搭建开发环境
第2章 ROS通信机制	ROS核心实现
第3章 ROS通信机制进阶	ROS核心实现
第4章 ROS运行管理	ROS中零散但又常用的知识点
第5章 ROS常用组件	ROS中比较实用的功能模块

实践篇

章节	内容
第6章 机器人系统仿真	机器人模型的创建，仿真环境的创建以及使用
第7章 机器人导航(仿真)	仿真环境下实现导航功能
第8章 机器人平台设计	从0到1手把手教你DIY一台机器人
第9章 机器人导航(实体)	将导航功能从仿真环境移植到实体机器人

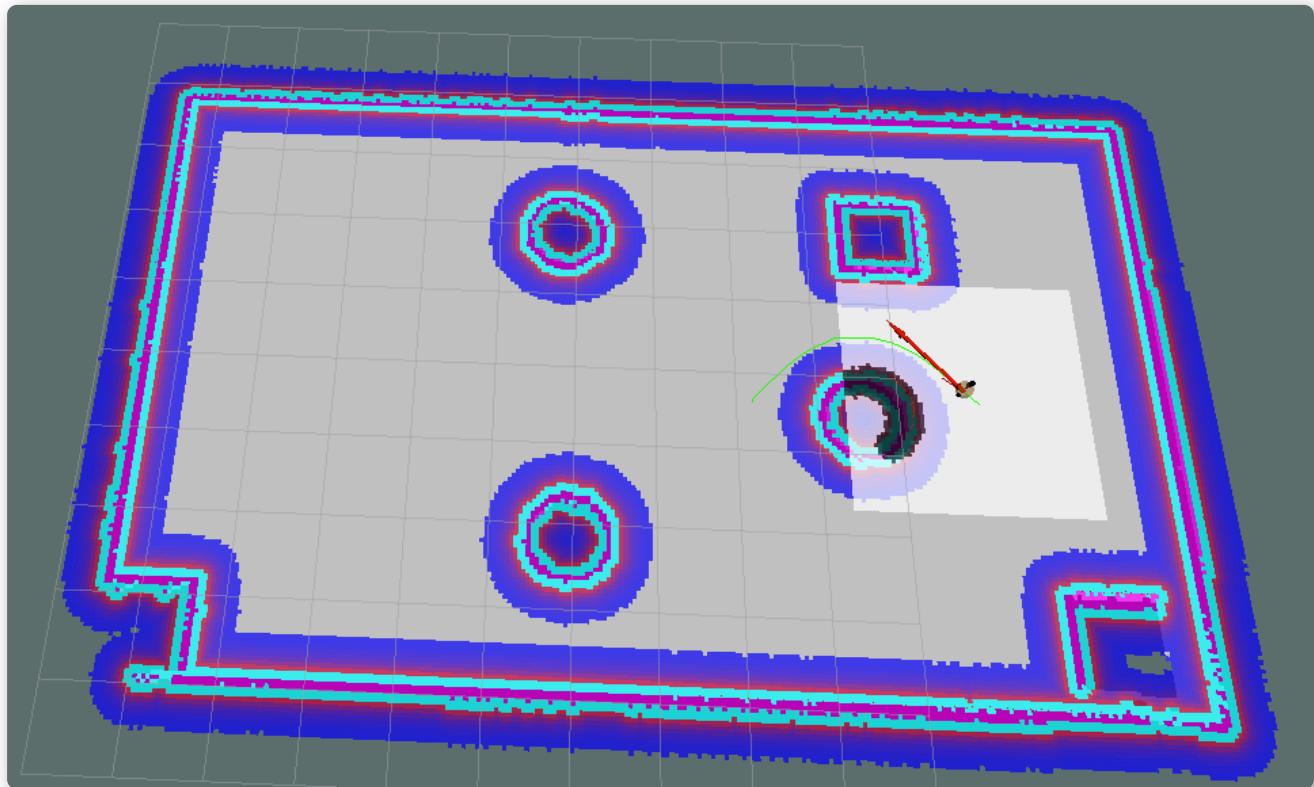
章节	内容
第10章 ROS进阶	ROS中的高级应用
第11章 ROS项目	公司内部一些ROS项目
...	...

整体而言，理论篇侧重于理论的介绍，是整个教程的基石，实践篇侧重于“可见”的应用，会通过一些案例将理论加以整合。

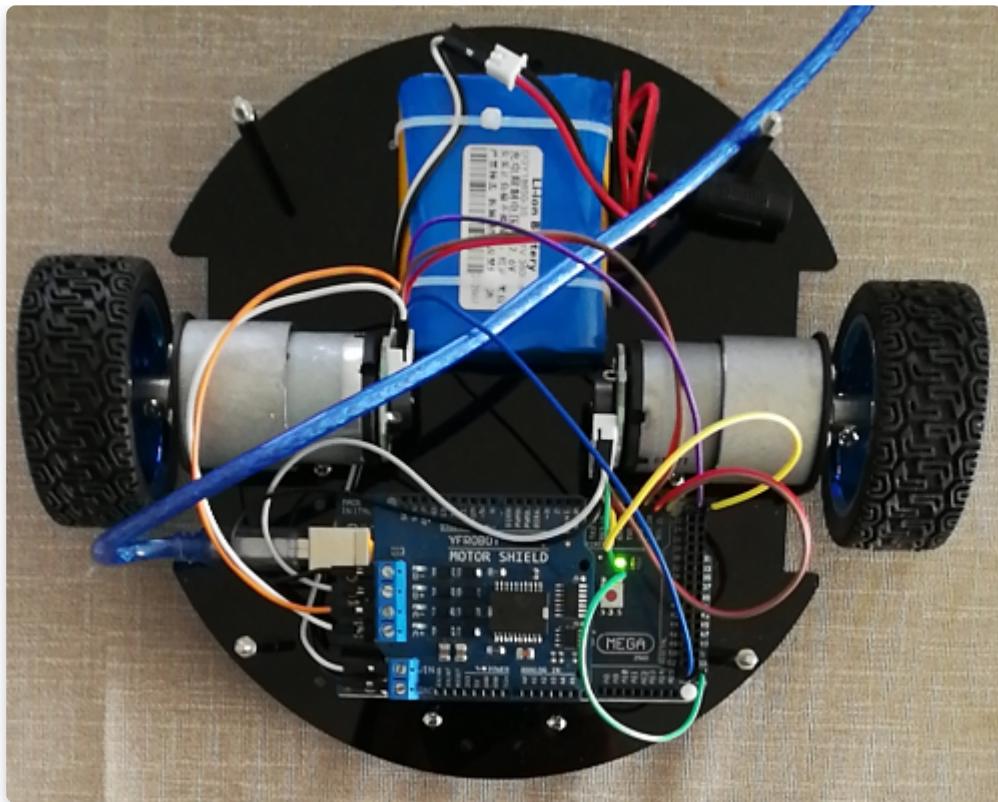
2.项目产出

部分演示如下

演示1:仿真环境下的导航实现



演示2:DIY的机器人





演示3:机器人SLAM

<https://www.bilibili.com/video/BV15z4y1672p>

演示4:机器人多点导航

<https://www.bilibili.com/video/BV1j5411n7Nc>

演示5:ROS模拟器

<https://www.bilibili.com/video/BV1bx411E7SC>

3. 答疑

1. 教程是完全免费的吗?

答:是。

2. 课程学习需要储备哪些知识?

答:操作系统Linux, 编程语言C++或Python, 其它如果不会, 遇到现学。

3. 学习需要硬件支持吗？

答：仿真环境可以实现大多数需求，如有需要就买，买我们(<http://www.autolabor.com.cn/>)的。

4. 写在最后

技术交流可以加非官方QQ群：869643967。

另外，课程内容如有不当，请多指正。

公司官网：<http://www.autolabor.com.cn/>

课程链接：<https://www.bilibili.com/video/BV1Ci4y1L7ZZ>

讲义链接：<http://www.autolabor.com.cn/book/ROSTutorials/index.html>

第1章 ROS概述与环境搭建

学习是一个循序渐进的过程，具体到计算机领域的软件开发层面，每当接触一个新的知识模块时，按照一般的步骤，我们会先去了解该模块的相关概念，然后再安装官方软件包，接下来再搭建其集成的开发环境...这些准备工作完毕之后，才算是叩开了新领域的大门。

学习ROS，我们也是遵循这一流程，本章作为ROS体系的开篇主要内容如下：

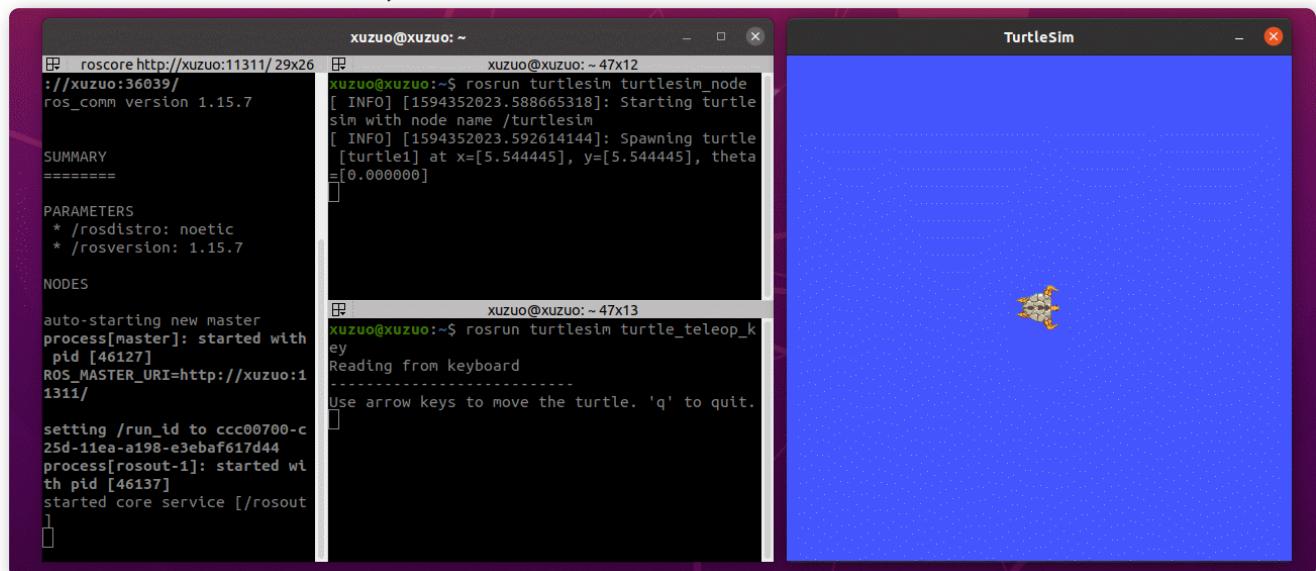
- ROS的相关概念
- 怎样安装ROS
- 如何搭建ROS的集成开发环境

该章内容学习完毕预期达成的目标如下：

- 了解 ROS 概念、设计目标以及发展历程
- 能够独立安装并运行 ROS
- 能够使用 C++ 或 Python 实现 ROS 版本的 HelloWorld
- 能够搭建 ROS 的集成开发环境
- 了解 ROS 架构设计

案例演示：

1.ROS安装成功后,可以运行内置案例:该案例是通过键盘控制乌龟运动



2.集成开发环境使用了VScode，可以提高开发效率

```
src > demo_helloworld > src > hello_pub.cpp
1  #include "ros/ros.h"
2
3
4
5
```

1.1 ROS简介

ROS诞生背景



机器人是一种高度复杂的系统性实现，机器人设计包含了机械加工、机械结构设计、硬件设计、嵌入式软件设计、上层软件设计....是各种硬件与软件集成，甚至可以说机器人系统是当今工业体系的集大成者。



机器人体系是相当庞大的，其复杂度之高，以至于没有任何个人、组织甚至公司能够独立完成系统性的机器人研发工作。

一种更合适的策略是：让机器人研发者专注于自己擅长的领域，其他模块则直接复用相关领域更专业研发团队的实现，当然自身的研究也可以被他人继续复用。这种基于“复用”的分工协作，遵循了不重复发明轮子的原则，显然是可以大大提高机器人的研发效率的，尤其是随着机器人硬件越来越丰富，软件库越来越庞大，这种复用性和模块化开发需求也愈发强烈。

在此大背景下，于 2007 年，一家名为 **柳树车库 (Willow Garage)** 的机器人公司发布了 **ROS**(机器人操作系统)，ROS是一套机器人通用软件框架，可以提升功能模块的复用性，并且随着该系统的不断迭代与完善，如今 ROS 已经成为机器人领域的事实标准。

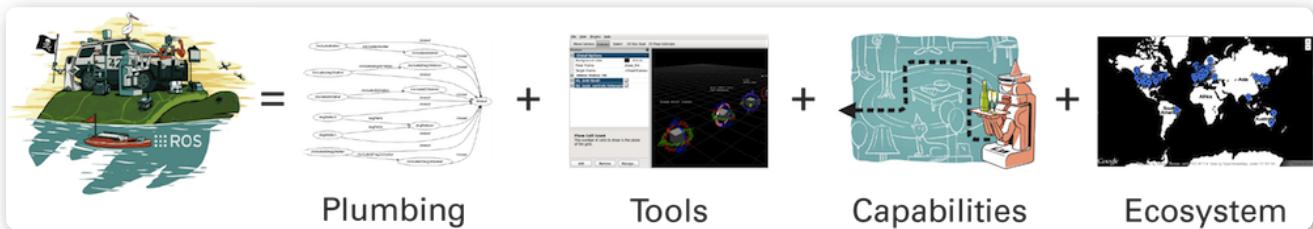


1.1.1ROS概念

ROS全称Robot Operating System(机器人操作系统)

- ROS是适用于机器人的开源元操作系统
- ROS集成了大量的工具，库，协议，提供类似OS所提供的功能，简化对机器人的控制
- 还提供了用于在多台计算机上获取，构建，编写和运行代码的工具和库，ROS在某些方面类似于“机器人框架”

- ROS设计者将ROS表述为“ROS = Plumbing + Tools + Capabilities + Ecosystem”，即ROS是通讯机制、工具软件包、机器人高层技能以及机器人生态系统的集合体



1.1.2 ROS设计目标

机器人开发的分工思想，实现了不同研发团队间的共享和协作，提升了机器人的研发效率，为了服务“分工”，ROS主要设计了如下目标：

- **代码复用:** ROS的目标不是成为具有最多功能的框架，ROS的主要目标是支持机器人技术研发中的代码重用。
- **分布式:** ROS是进程（也称为*Nodes*）的分布式框架，ROS中的进程可分布于不同主机，不同主机协同工作，从而分散计算压力
- **松耦合:** ROS中功能模块封装于独立的功能包或元功能包，便于分享，功能包内的模块以节点为单位运行，以ROS标准的IO作为接口，开发者不需要关注模块内部实现，只要了解接口规则就能实现复用，实现了模块间点对点的松耦合连接
- **精简:** ROS被设计为尽可能精简，以便为ROS编写的代码可以与其他机器人软件框架一起使用。ROS易于与其他机器人软件框架集成：ROS已与OpenRAVE，Orocos和Player集成。
- **语言独立性:** 包括Java，C++，Python等。为了支持更多应用开发和移植，ROS设计为一种语言弱相关的框架结构，使用简洁，中立的定义语言描述模块间的消息接口，在编译中再产生所使用语言的目标文件，为消息交互提供支持，同时允许消息接口的嵌套使用
- **易于测试:** ROS具有称为[rostest](#)的内置单元/集成测试框架，可轻松安装和拆卸测试工具。
- **大型应用:** ROS适用于大型运行时系统和大型开发流程。
- **丰富的组件化工具包:** ROS可采用组件化方式集成一些工具和软件到系统中并作为一个组件直接使用，如RVIZ（3D可视化工具），开发者根据ROS定义的接口在其中显示机器人模型等，组件还包括仿真环境和消息查看工具等
- **免费且开源:** 开发者众多，功能包多

1.1.3ROS发展历程

- ROS是一个由来已久、贡献者众多的大型软件项目。在ROS诞生之前，很多学者认为，机器人研究需要一个开放式的协作框架，并且已经有不少类似的项目致力于实现这样的框架。在这些工作中，斯坦福大学在2000年年中开展了一系列相关研究项目，如斯坦福人工智能机器人（STanford AI Robot, STAIR）项目、个人机器人（Personal Robots, PR）项目等，在上述项目中，在研究具有代表性、集成式人工智能系统的过程中，创立了用于室内场景的高灵活性、动态软件系统，其可以用于机器人学研究。
- 2007年，柳树车库（Willow Garage）提供了大量资源，用于将斯坦福大学机器人项目中的软件系统进行扩展与完善，同时，在无数研究人员的共同努力下，ROS的核心思想和基本软件包逐渐得到完善。
- ROS的发行版本（ROS distribution）指ROS软件包的版本，其与Linux的发行版本（如Ubuntu）的概念类似。推出ROS发行版本的目的在于使开发人员可以使用相对稳定的代码库，直到其准备好将所有内容进行版本升级为止。因此，每个发行版本推出后，ROS开发者通常仅对这一版本的bug进行修复，同时提供少量针对核心软件包的改进。
- 版本特点: 按照英文字母顺序命名，ROS目前已经发布了ROS1的终极版本: noetic，并建议后期过渡至ROS2版本。noetic版本之前默认使用的是Python2，noetic支持Python3。
建议版本: noetic 或 melodic 或 kinetic

Distro	Release date	Poster	Tuturtle, turtle in tutorial	EOL date
ROS Noetic Ninjemy's (Recommended)	May 23rd, 2020			May, 2025 (Focal EOL)
ROS Melodic Morenia	May 23rd, 2018			May, 2023 (Bionic EOL)
ROS Lunar Loggerhead	May 23rd, 2017			May, 2019
ROS Kinetic Kame	May 23rd, 2016			April, 2021 (Xenial EOL)
ROS Jade Turtle	May 23rd, 2015			May, 2017
ROS Indigo Igloo	July 22nd, 2014			April, 2019 (Trusty EOL)
ROS Hydro Medusa	September 4th, 2013			May, 2015
ROS Groovy Galapagos	December 31, 2012			July, 2014
ROS Fuerte Turtle	April 23, 2012			--
ROS Electric Emys	August 30, 2011			--
ROS Diamondback	March 2, 2011			--
ROS C Turtle	August 2, 2010			--
ROS Box Turtle	March 2, 2010			--

另请参考：

- <https://www.ros.org/about-ros/>
- <http://wiki.ros.org/ROS/Introduction>
- <http://wiki.ros.org/Distributions>

1.2 ROS安装

我们使用的是 ROS 版本是 Noetic，那么可以在 ubuntu20.04、Mac 或 windows10 系统上安装，虽然一般用户平时使用的操作系统以 windows 居多，但是 ROS 之前的版本基本都不支持 windows，所以当前我们选用的操作系统是 ubuntu，以方便向历史版本过渡。ubuntu 安装常用方式有两种：

- 实体机安装 ubuntu (较为常用的是使用双系统，windows 与 ubuntu 并存)；
- 虚拟机安装 ubuntu。

两种方式比较，各有优缺点：

- 方案1可以保证性能，且不需要考虑硬件兼容性问题，但是和 windows 系统交互不便；
- 方案2可以方便的实现 windows 与 ubuntu 交互，不过性能稍差，且与硬件交互不便。

在 ROS 中，一些仿真操作是比较耗费系统资源的，且经常需要和一些硬件(雷达、摄像头、imu、STM32、arduino....)交互，因此，原则上建议采用方案1，不过如果只是出于学习目的，那么方案2也基本够用，且方案2在 windows 与 ubuntu 的交互上更为方便，对于学习者更为友好，因此本教程在此选用的是方案2。当然，具体采用哪种实现方案，请按需选择。

如果采用虚拟机安装 ubuntu，再安装 ROS 的话，大致流程如下：

1. 安装虚拟机软件(比如：virtualbox 或 VMware)；
2. 使用虚拟机软件虚拟一台主机；
3. 在虚拟主机上安装 ubuntu 20.04；
4. 在 ubuntu 上安装 ROS；
5. 测试 ROS 环境是否可以正常运行。

虚拟机软件选择上，对于我们学习而言 virtualbox 和 VMware 都可以满足需求，二者比较，前者免费，后者收费，所以本教程选用 virtualbox。

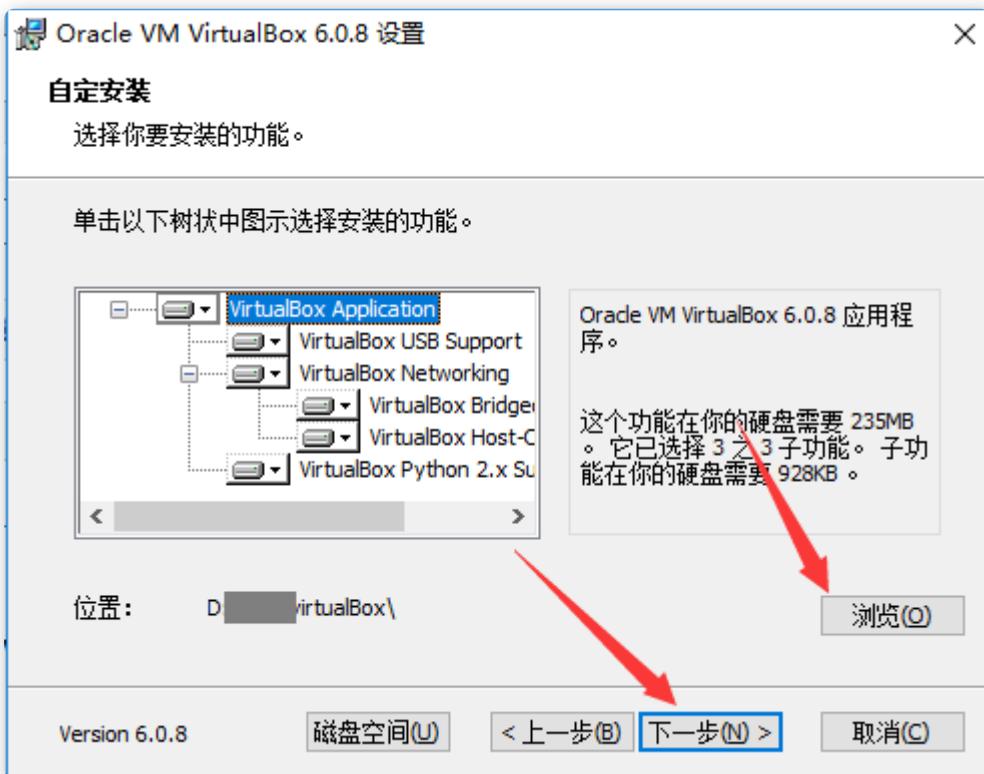
1.2.1 安装虚拟机软件

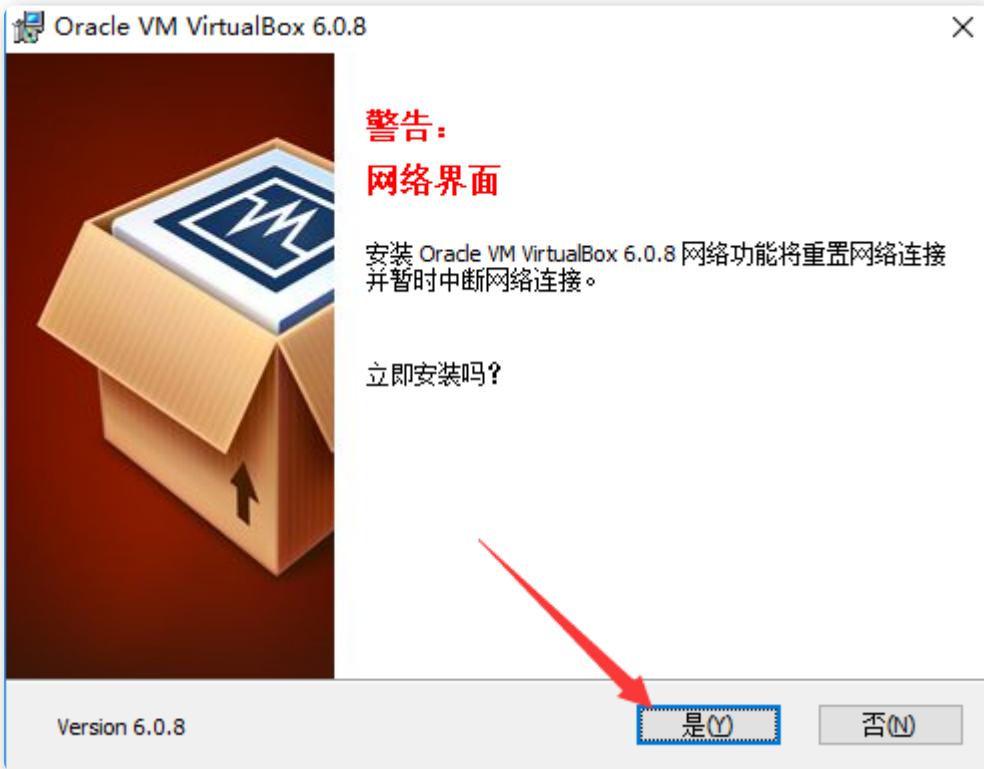
1. 下载virtualbox

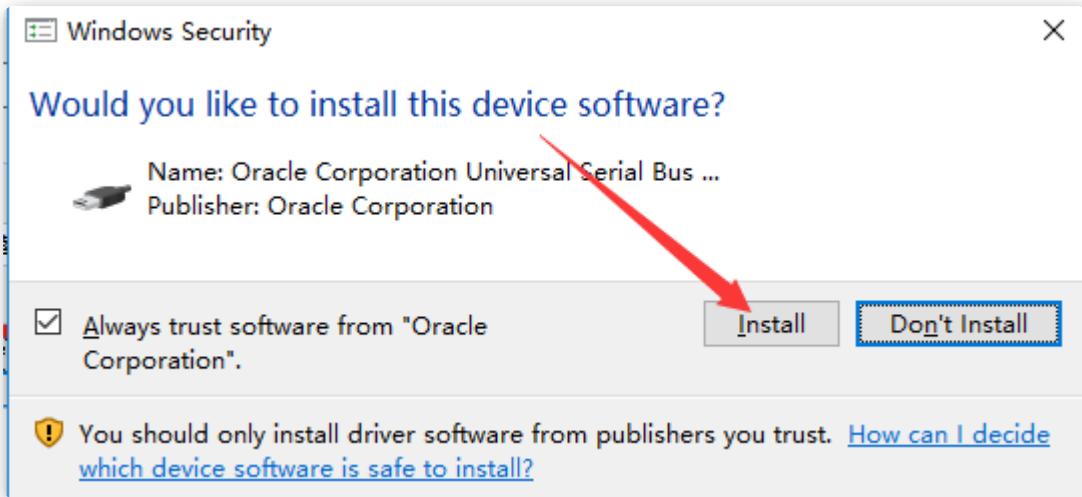
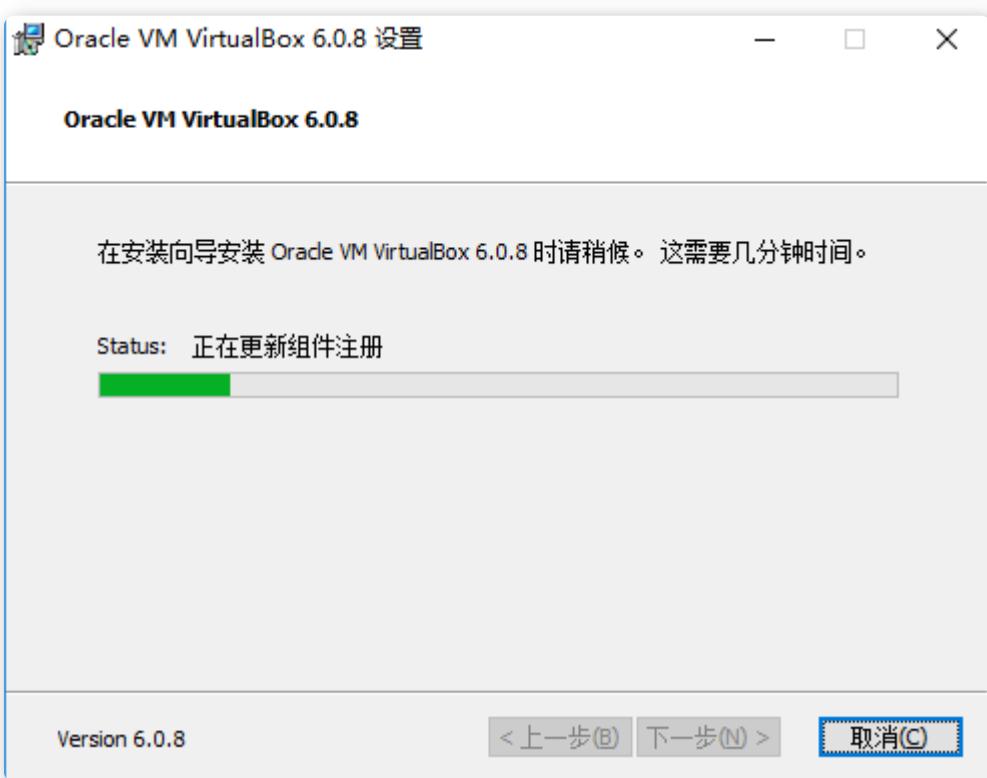
安装 virtualbox 需要先访问官网，下载安装包，官网下载地址:<https://www.virtualbox.org/wiki/Downloads>

2. 安装virtualbox

virtualbox 安装比较简单，如果没有特殊需求，双击安装文件，一直 "下一步" 即可。





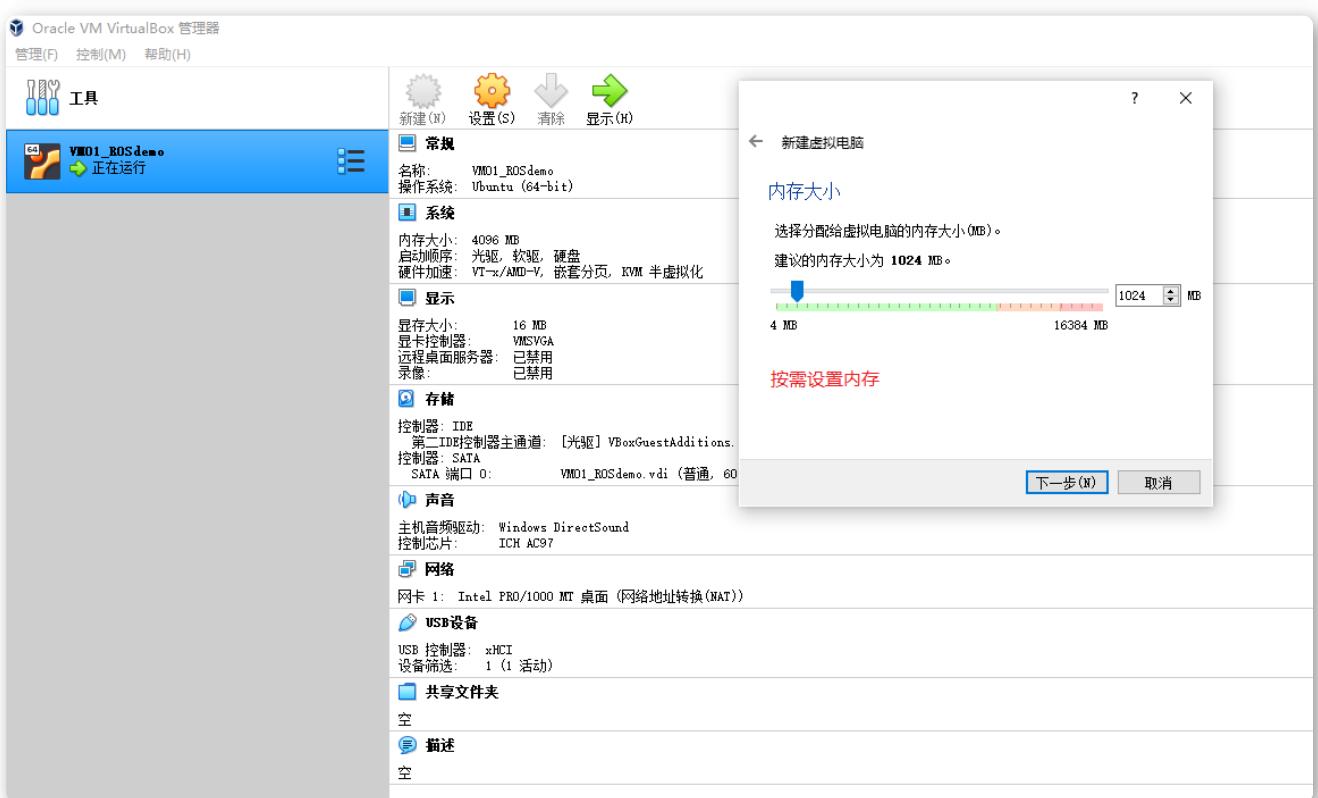
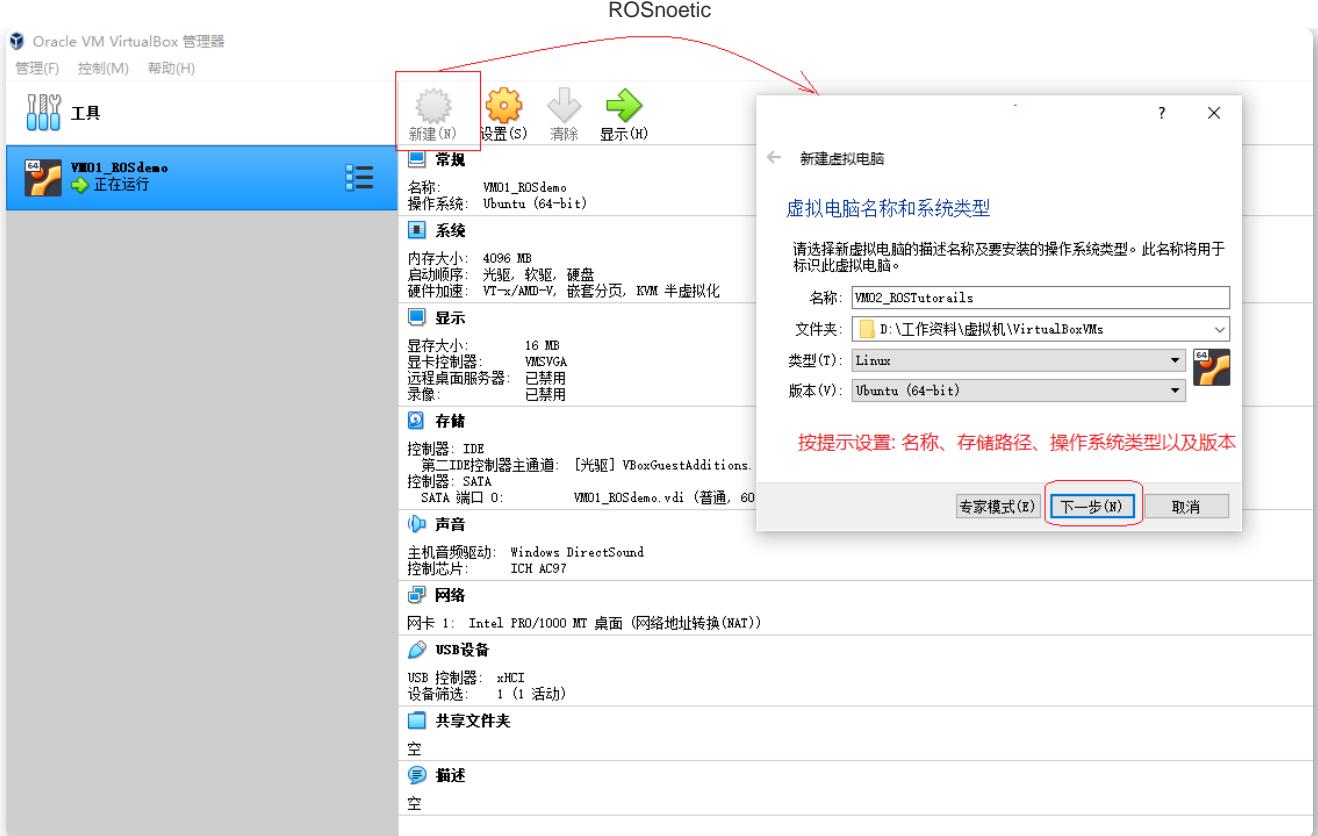


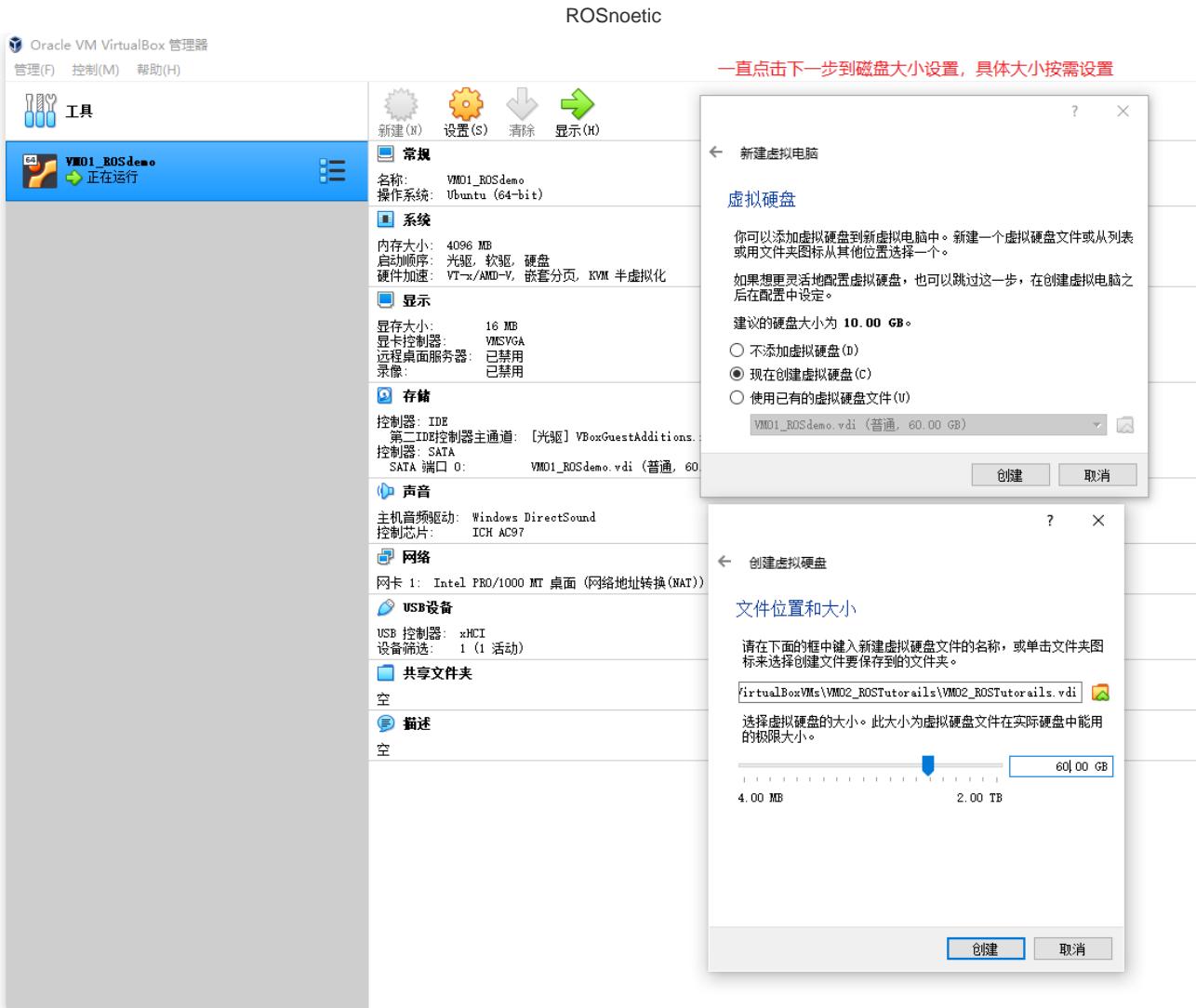


安装完毕后，虚拟机已经可以正常启动了，接下来需要使用其虚拟出一台计算机

1.2.2 虚拟一台主机

使用 virtual 虚拟计算机的过程也不算复杂，只需要按照提示配置其相关参数即可



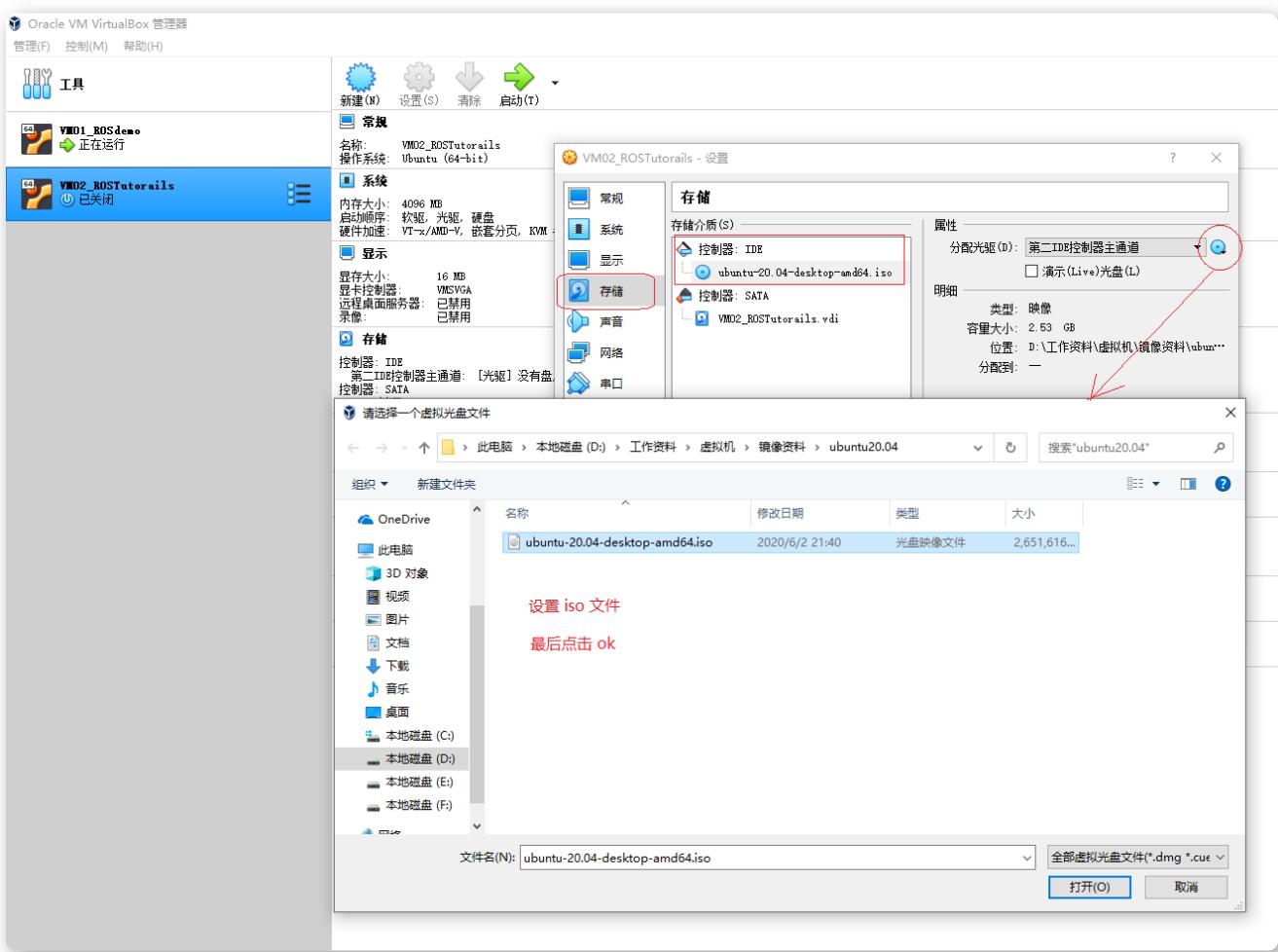
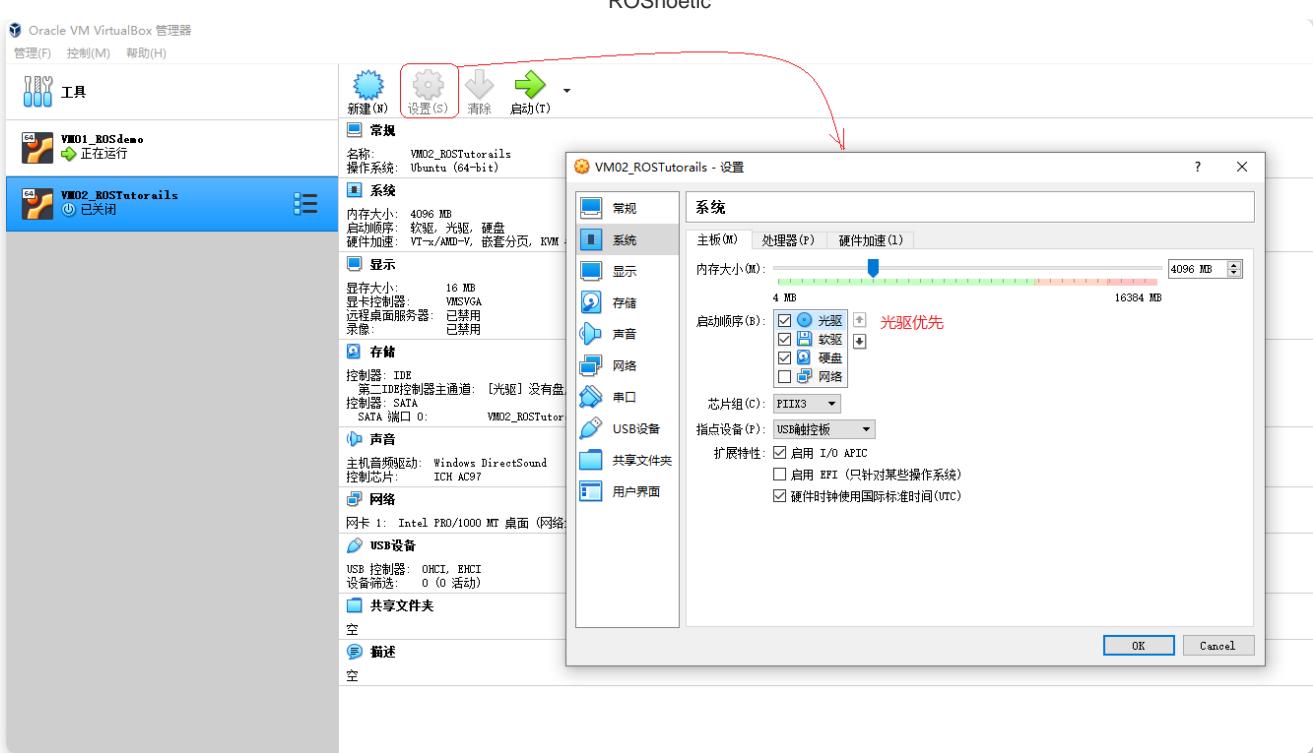


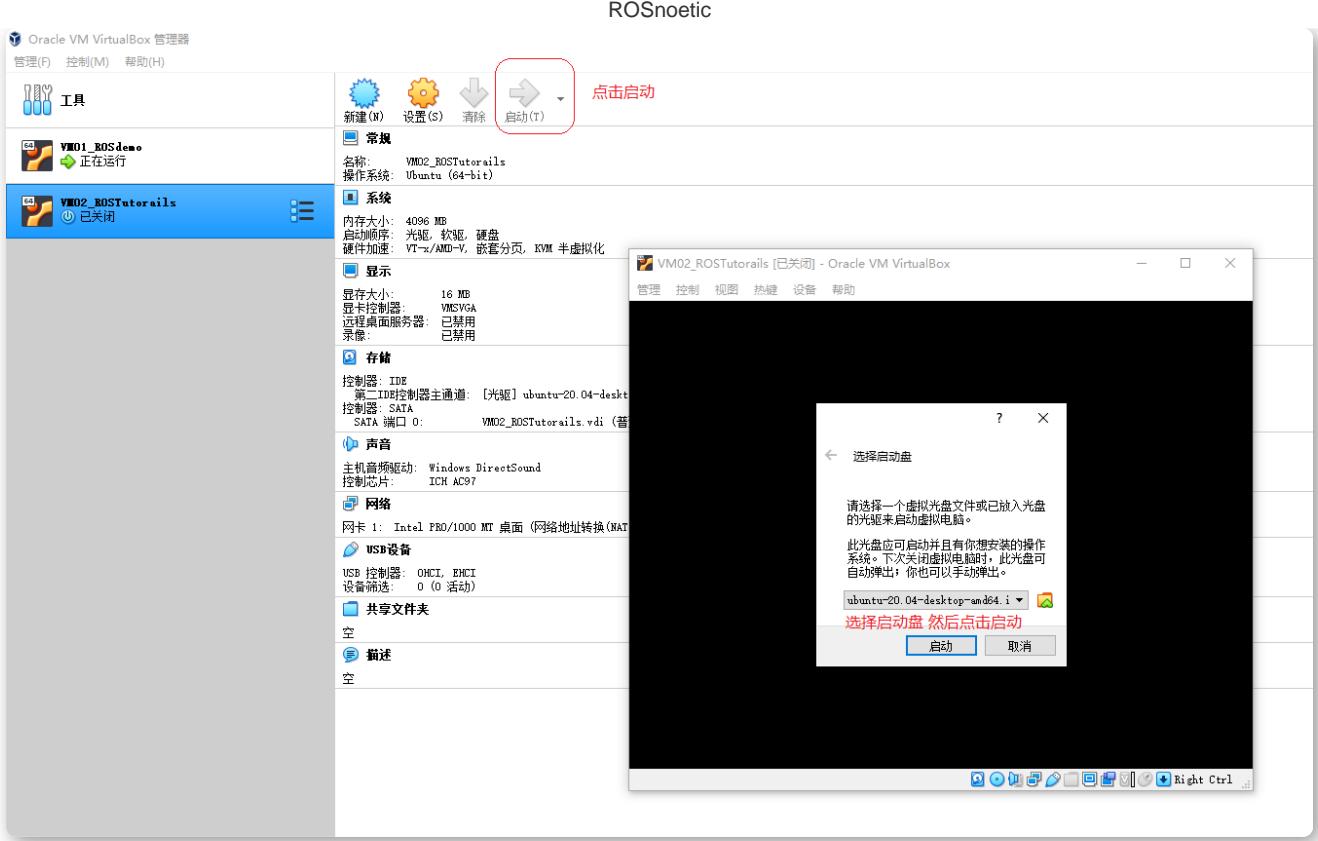
1.2.3 安装ubuntu

1.ubuntu安装

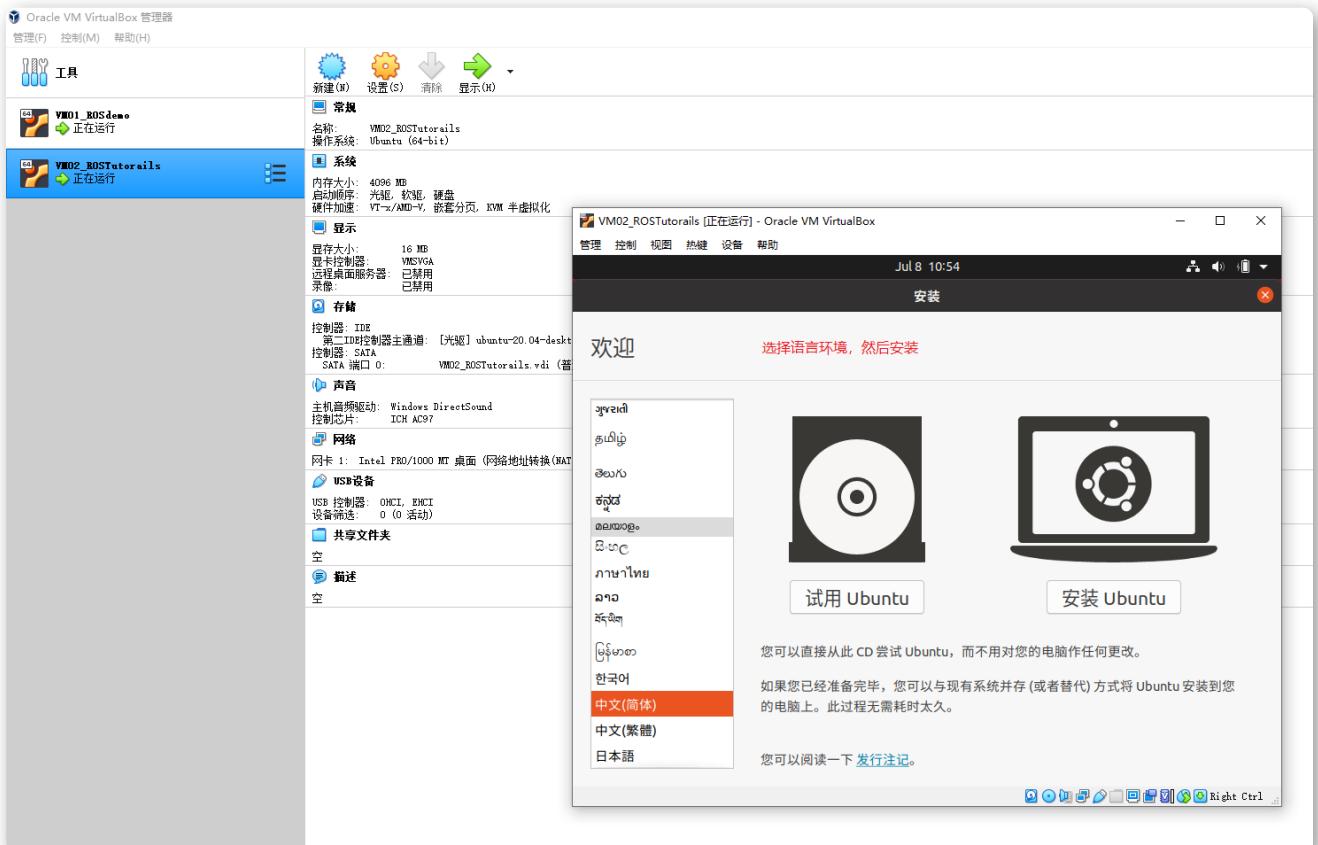
首先下载 Ubuntu 的镜像文件，链接如下:<http://mirrors.aliyun.com/ubuntu-releases/20.04/>；

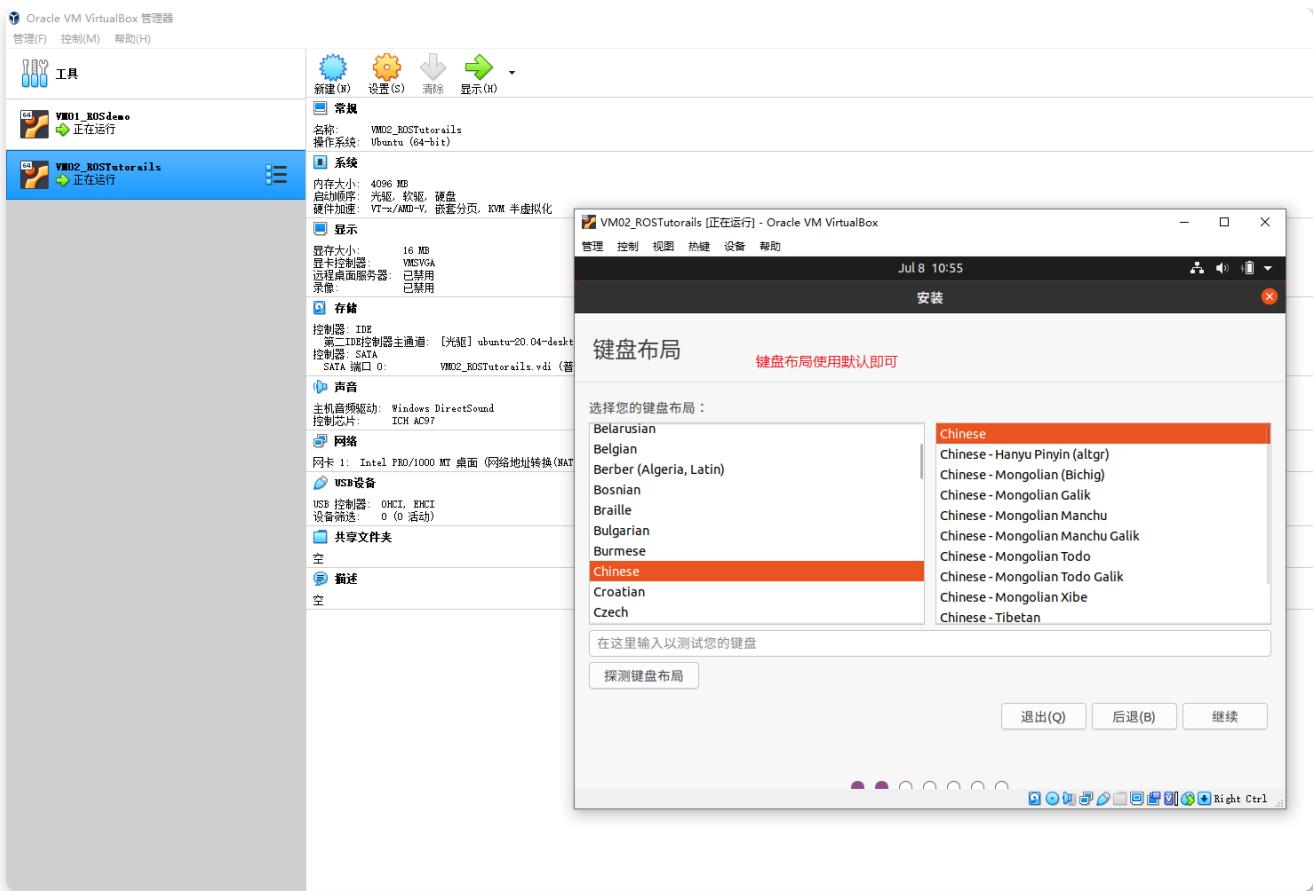
然后，配置虚拟主机，关联 Ubuntu 镜像文件：



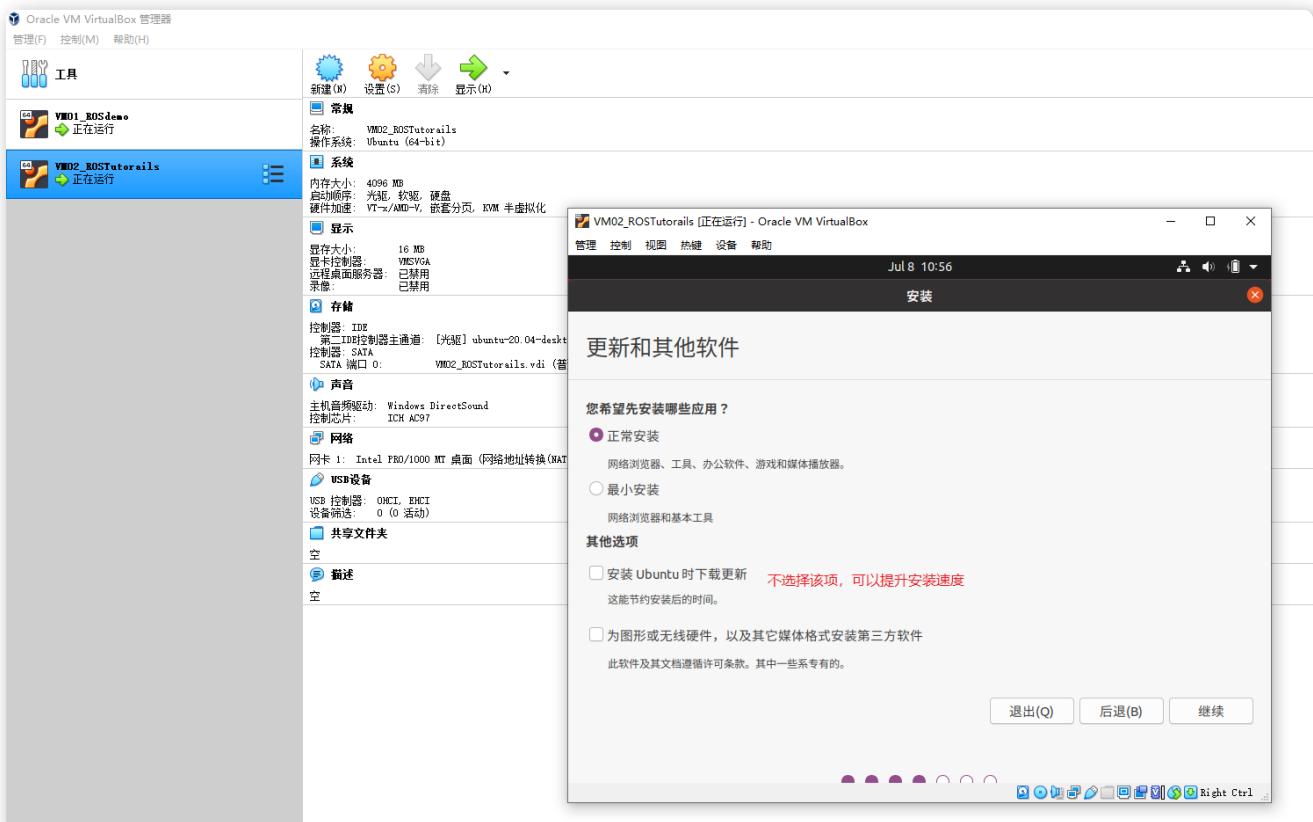


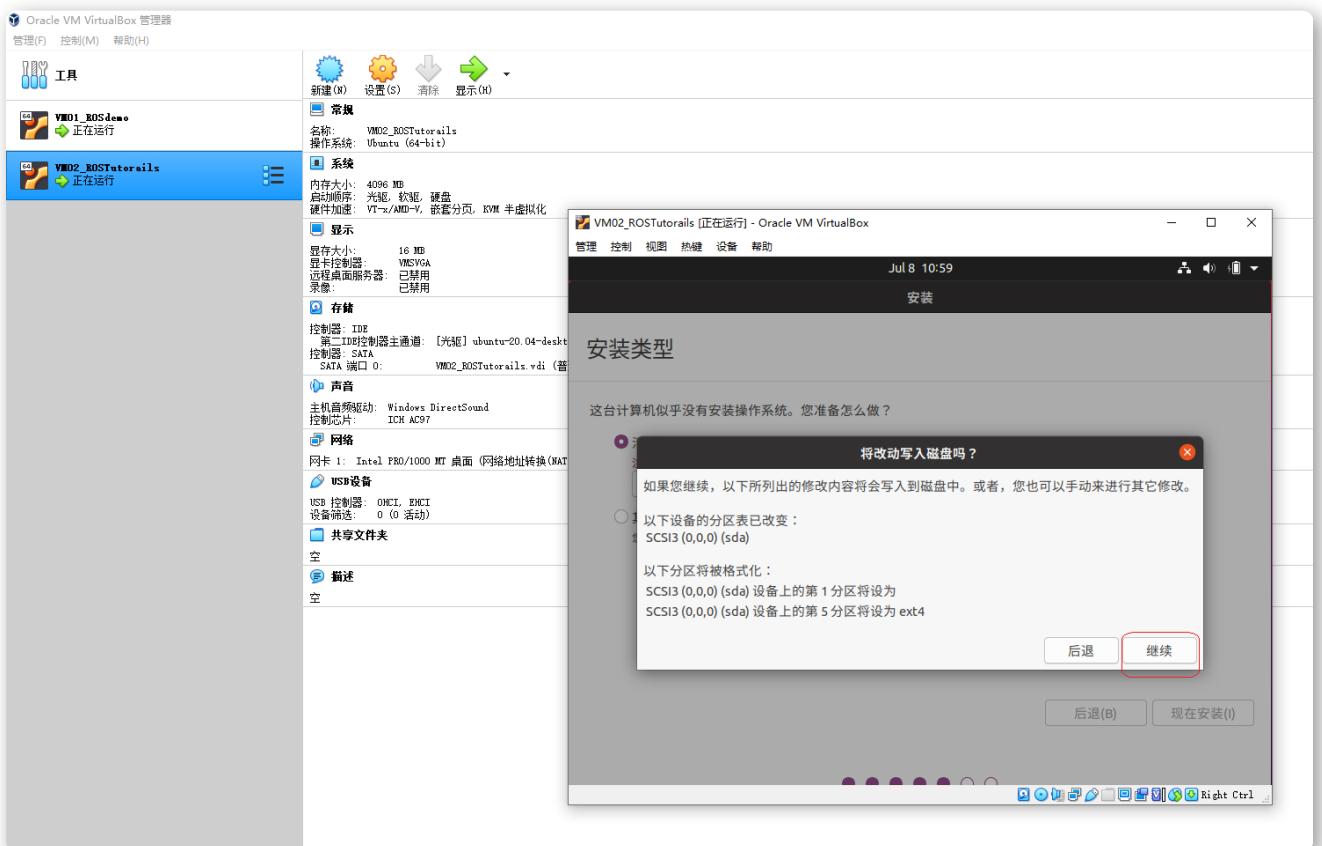
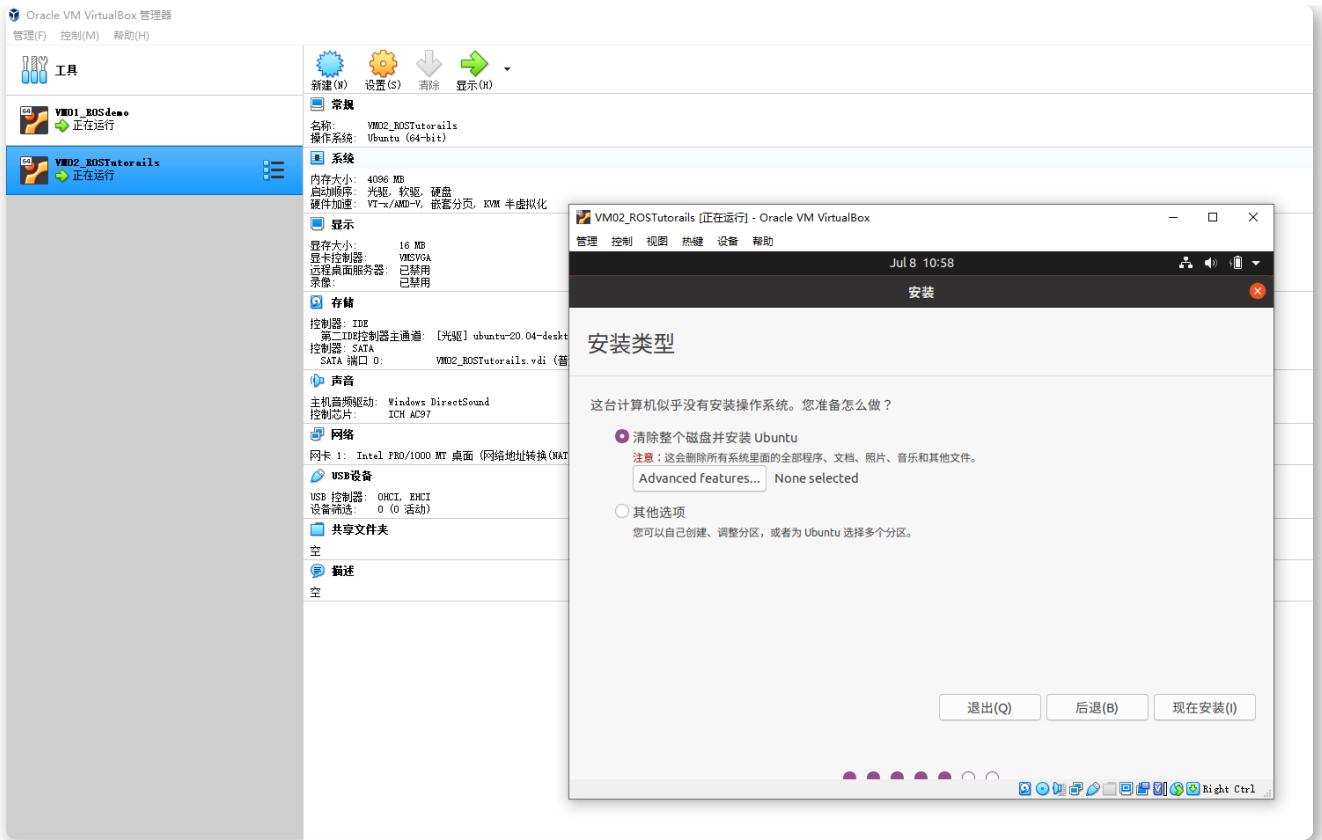
启动后，开始配置 ubuntu 操作系统：

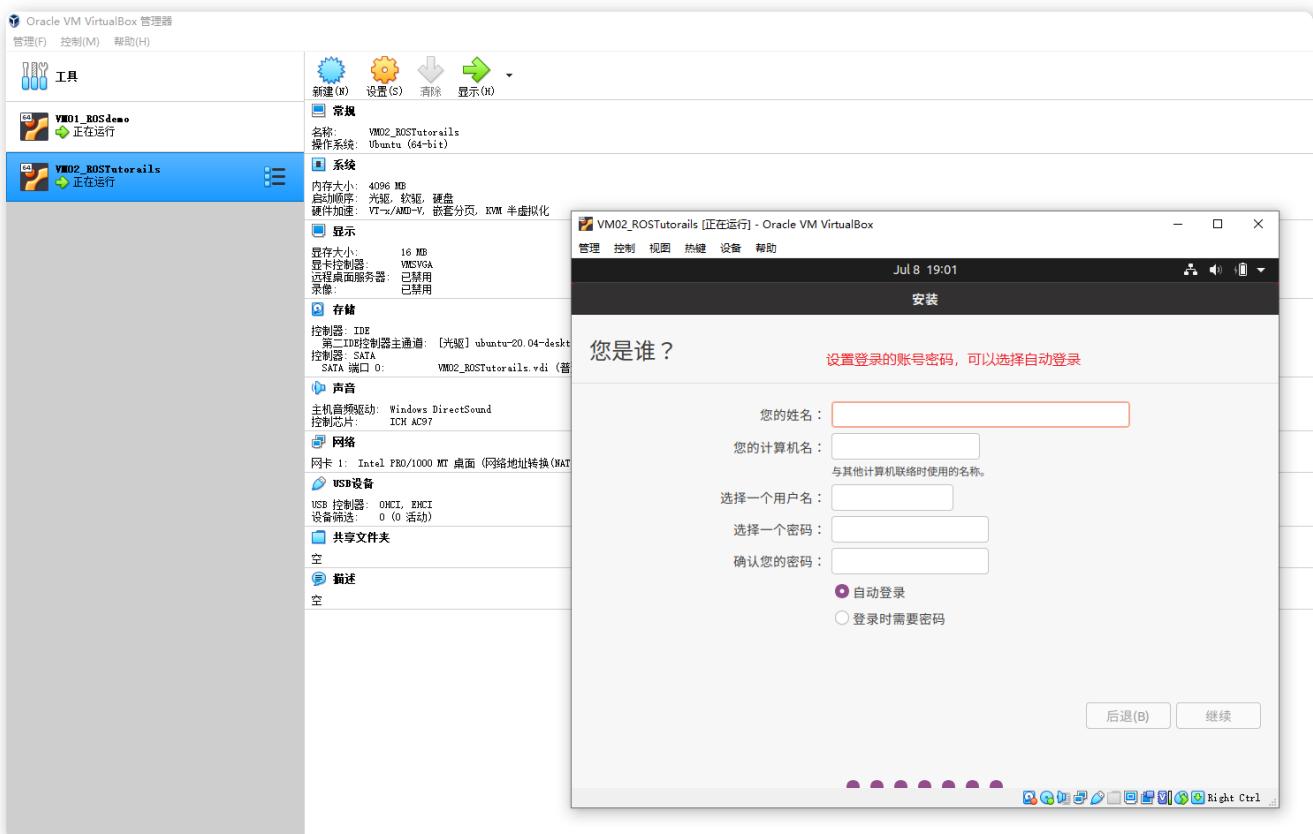
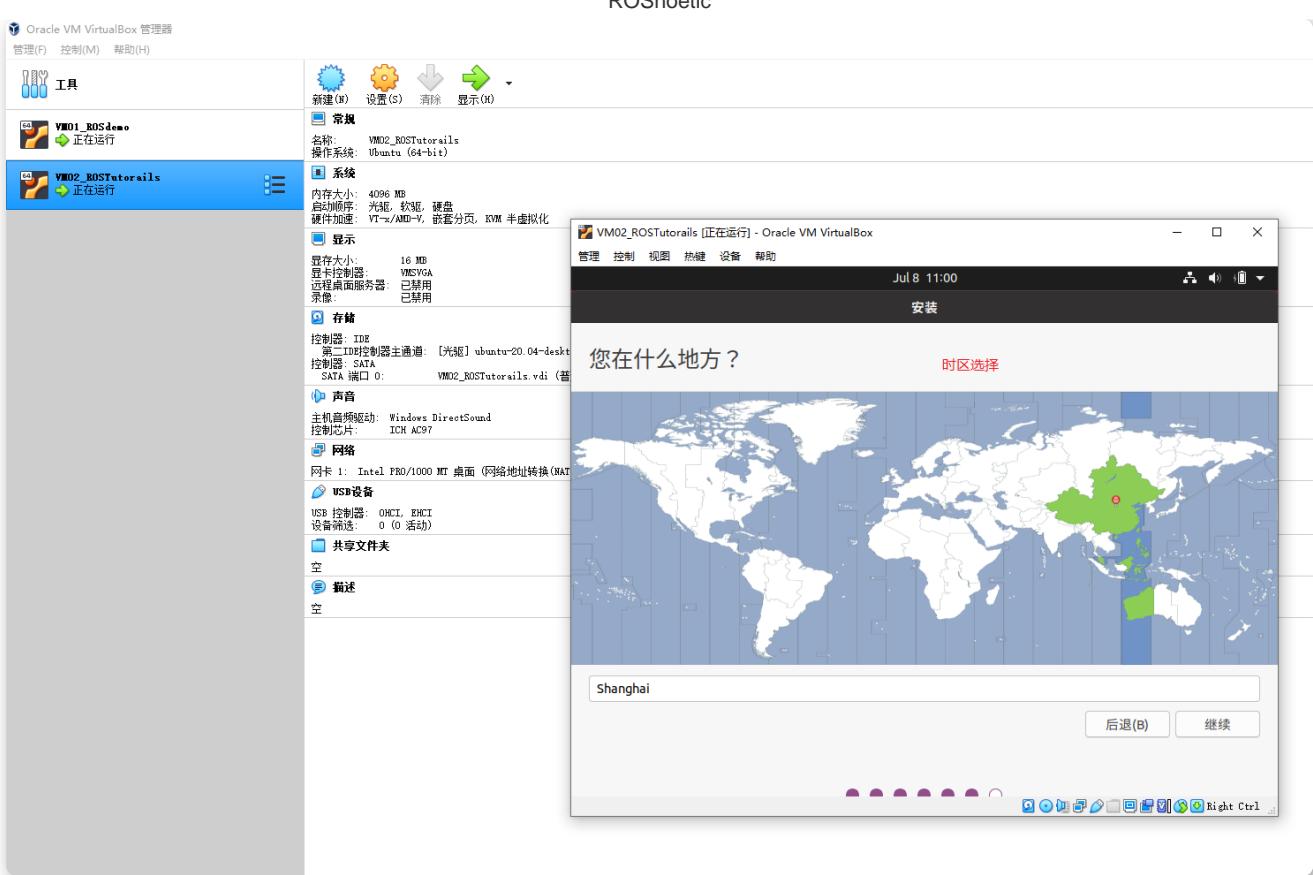


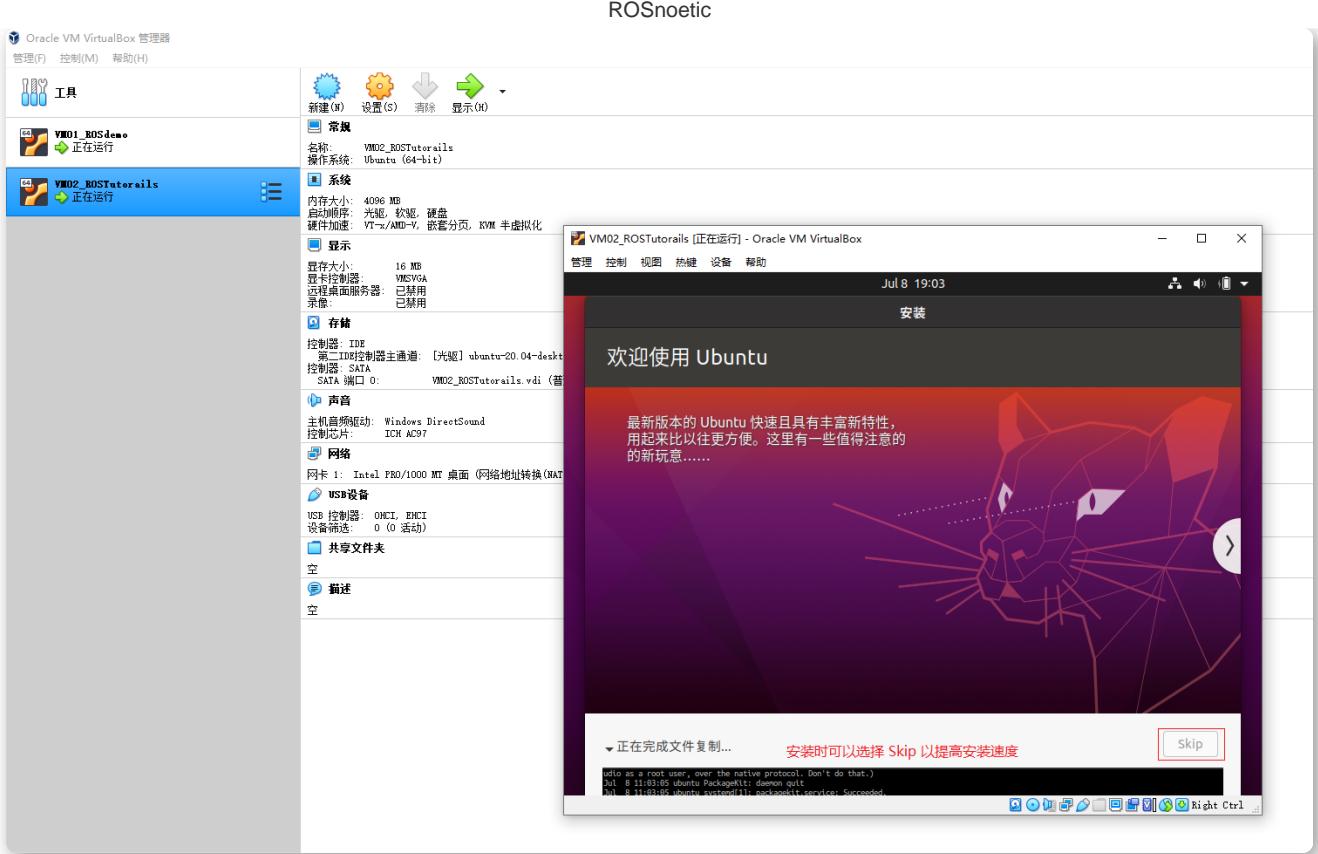


安装过程中，断开网络连接，可以提升安装速度：

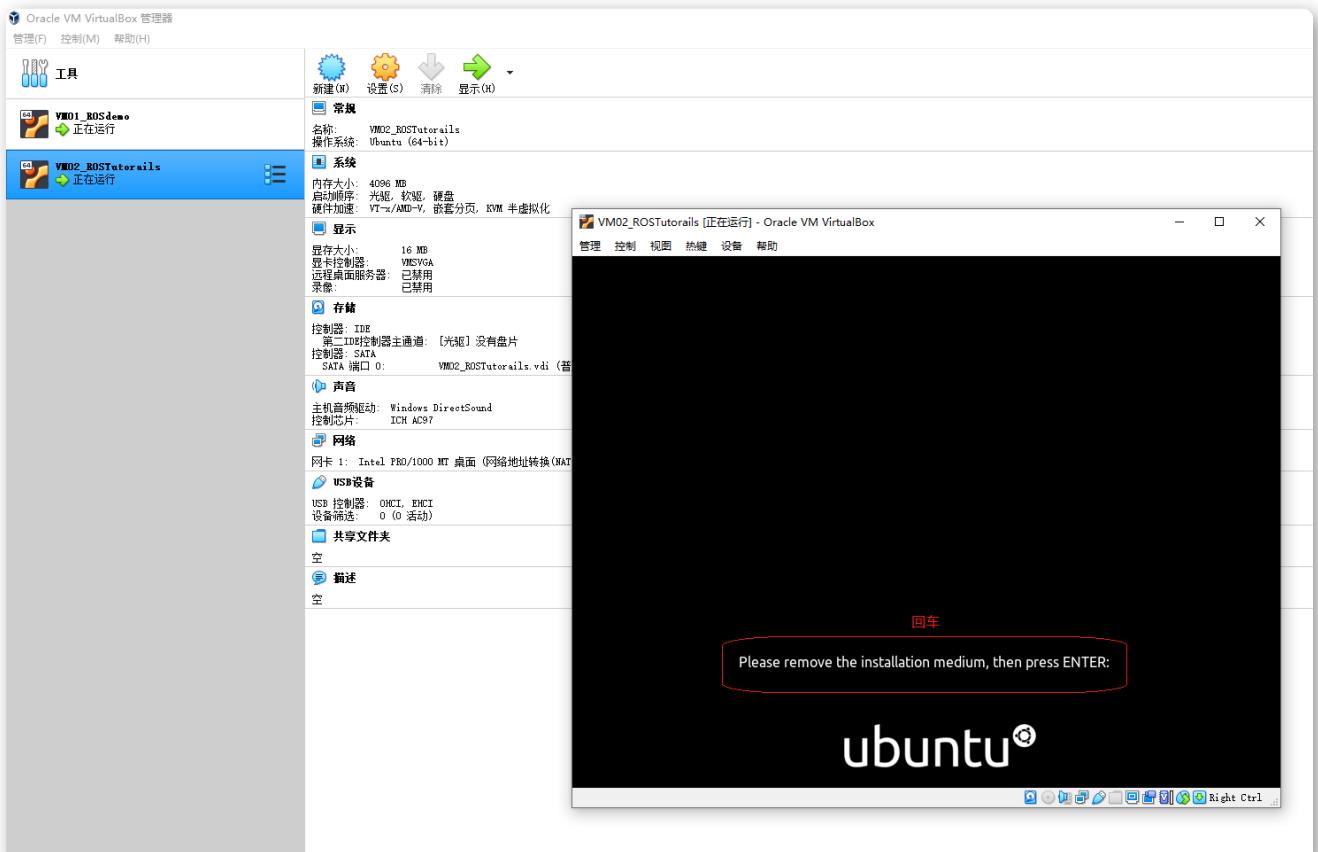








安装完毕后，会给出重启提示，点击重启确定按钮即可：

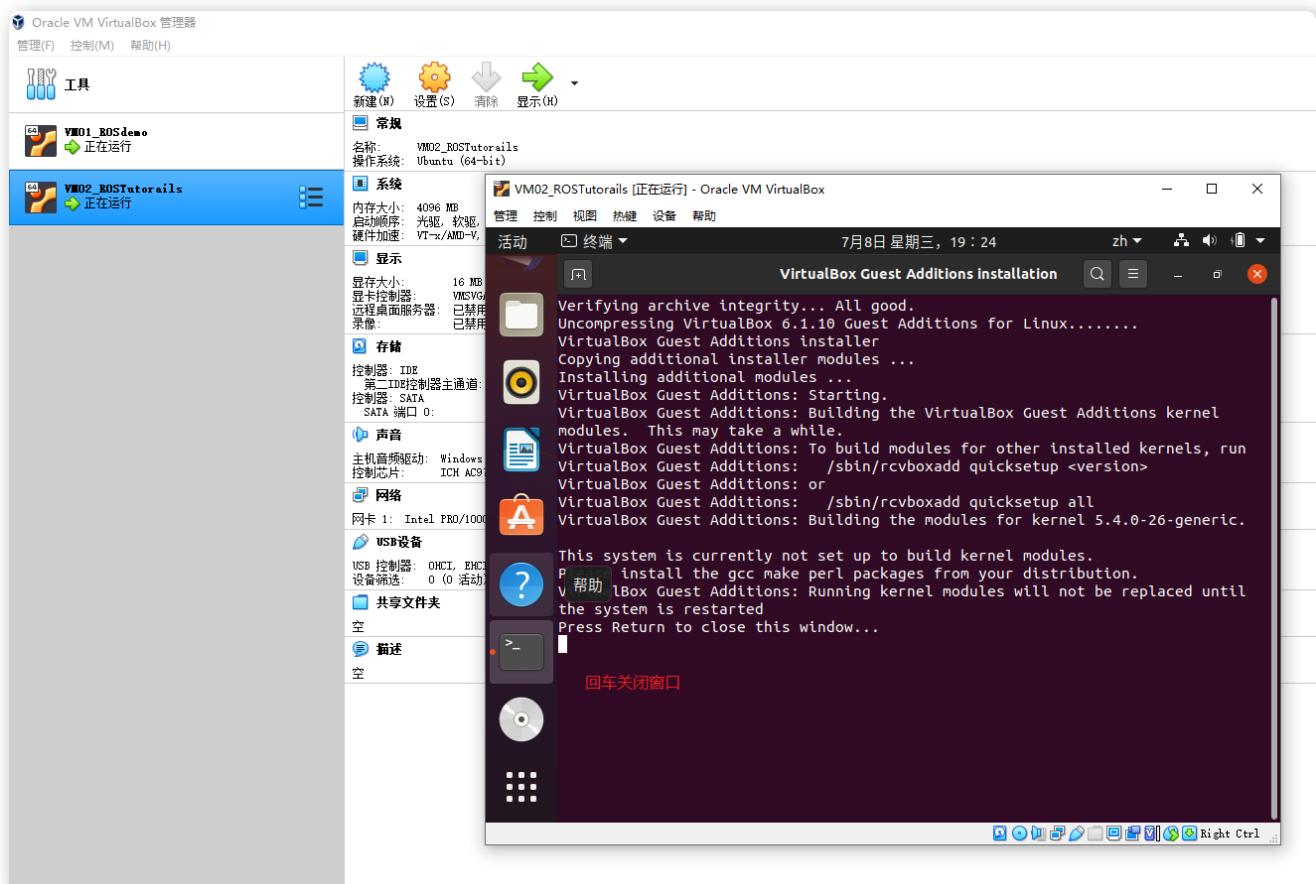
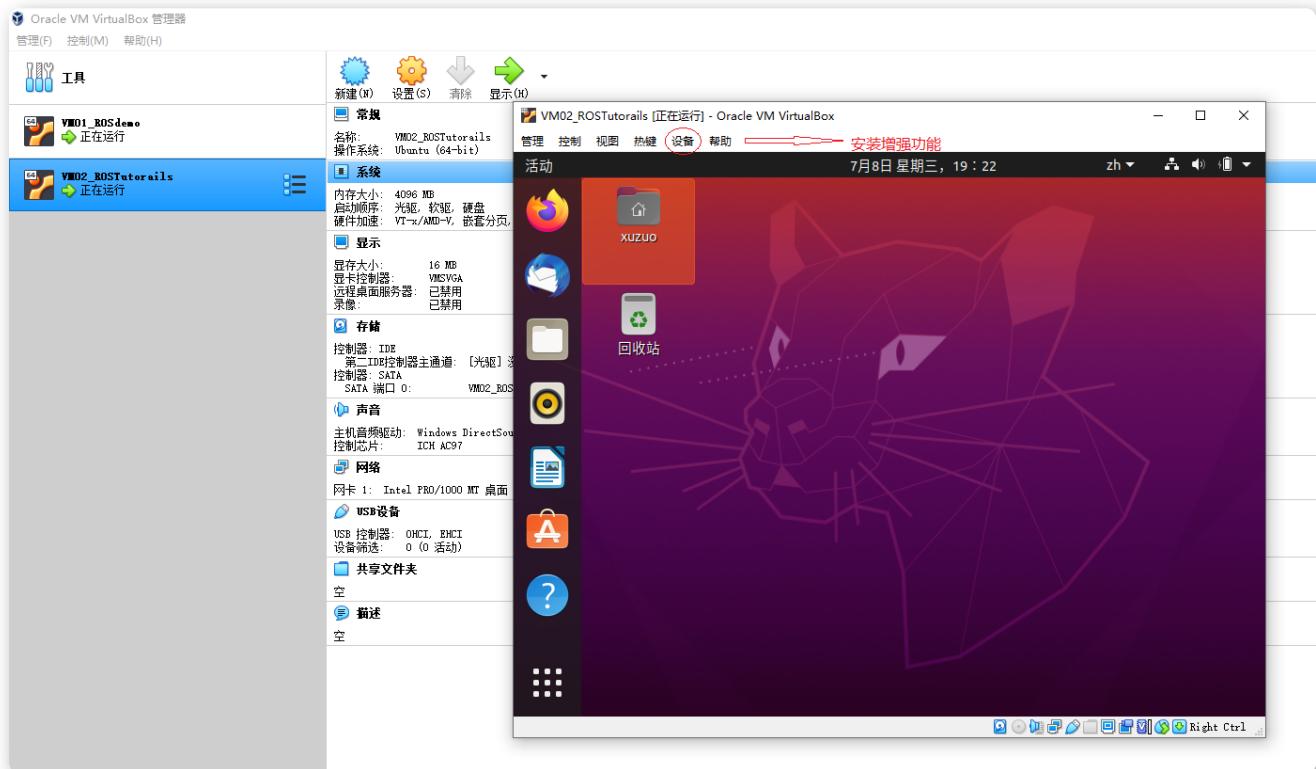


到目前为止 VirtualBox 已经正常安装了 ubuntu，并启动成功。

2. 使用优化

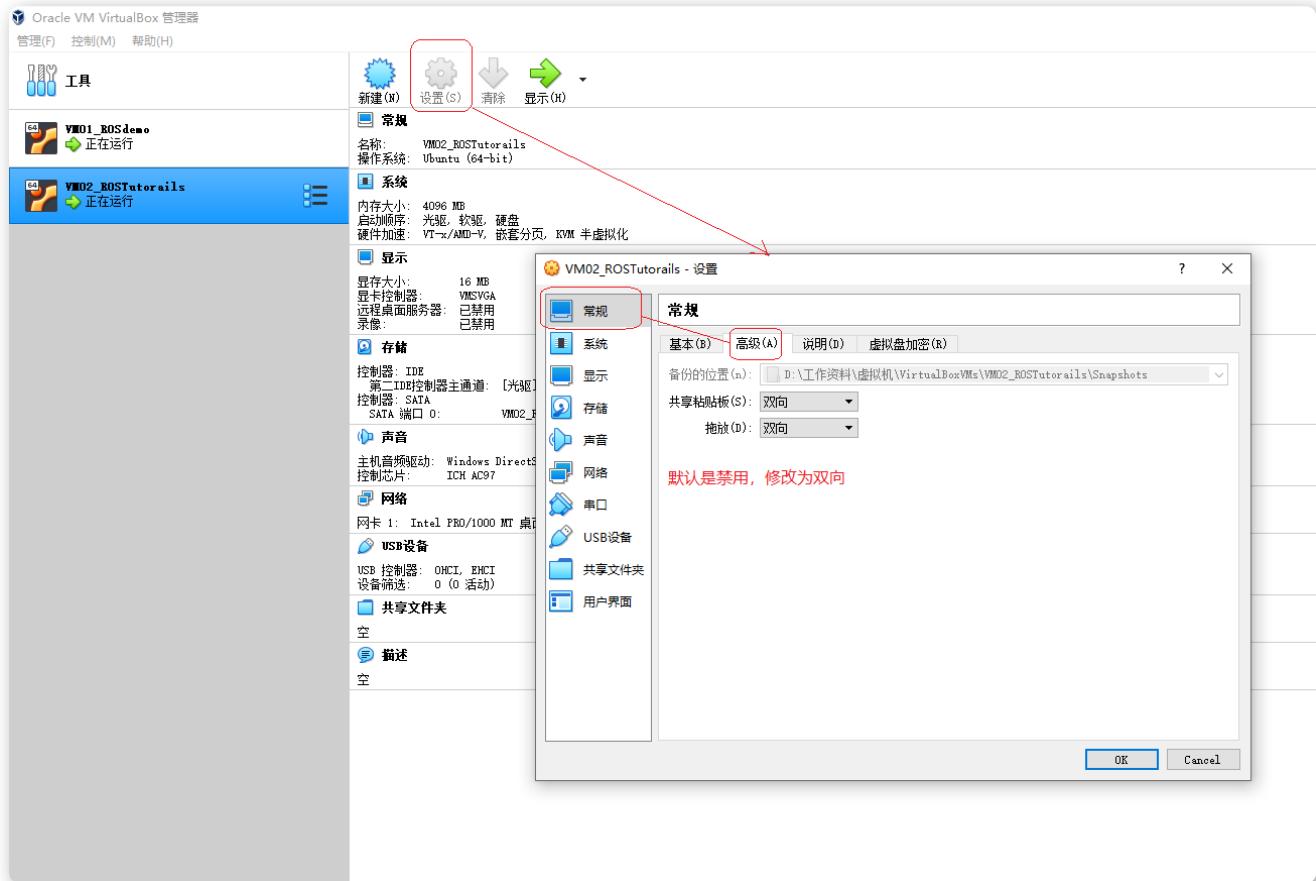
为了优化 ubuntu 操作的用户体验，方便虚拟机与宿主机的文件交换以及 USB 设备的正常使用，还需做如下操作：

① 安装虚拟机工具



重启使之生效，选择菜单栏的 **自动调整窗口大小**，然后ubuntu 桌面会自动使用窗口大小：右 **ctrl + F** 全屏。

②启动文件交换模式



③安装扩展插件

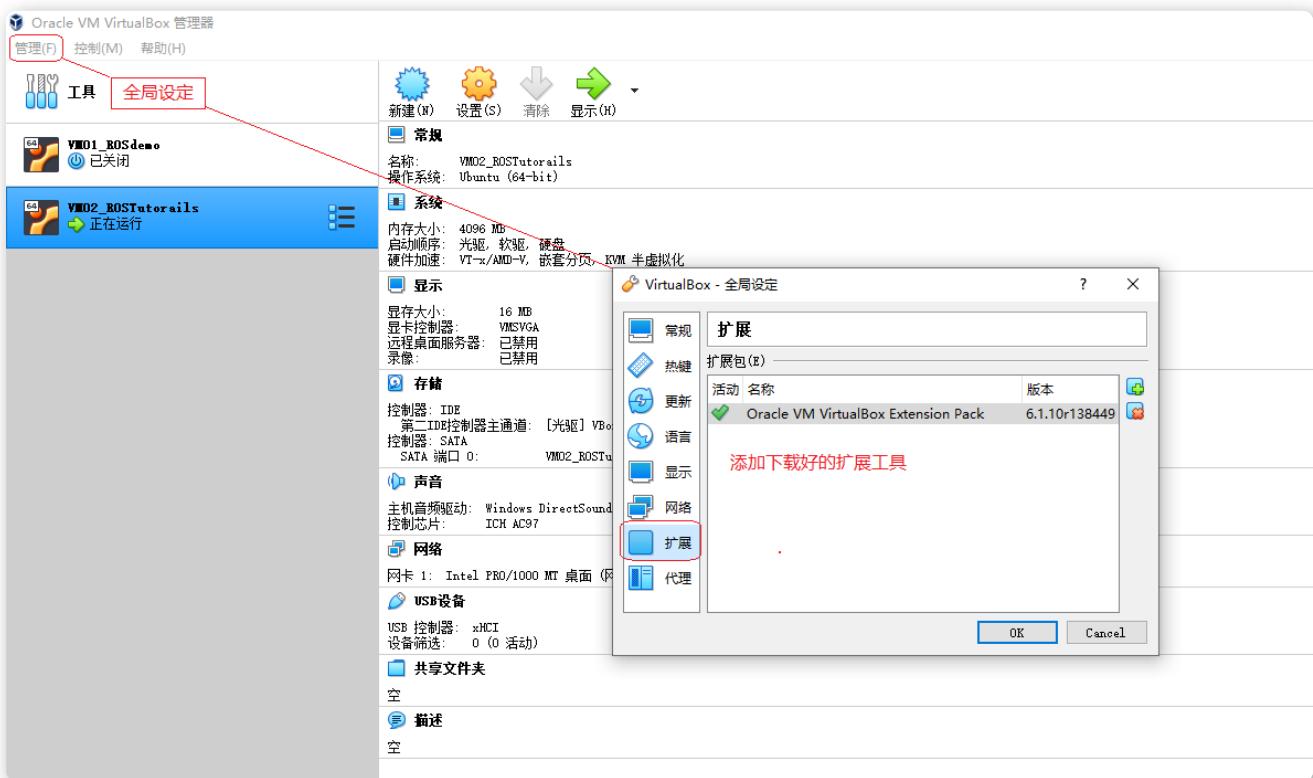
先去 [virtualbox 官网](#) 下载扩展包

virtualbox.org/wiki/Downloads



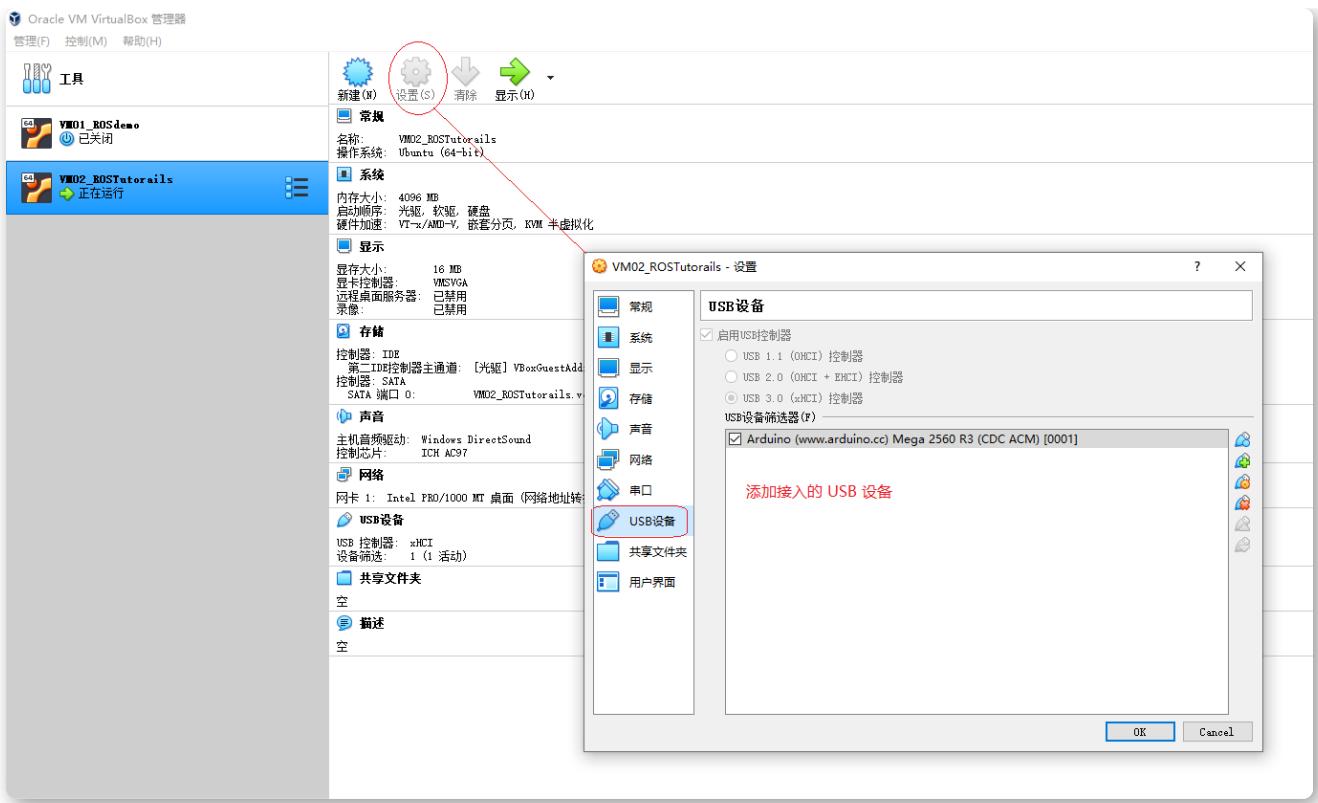
The screenshot shows the VirtualBox download page. The left sidebar includes links for About, Screenshots, Downloads (which is highlighted with a red box), Documentation, End-user docs, Technical docs, Contribute, and Community. The main content area features the VirtualBox logo and a section titled 'VirtualBox binaries'. It includes links for Windows hosts, OS X hosts, Linux distributions, and Solaris hosts. A note states: 'The binaries are released under the terms of the GPL version 2.' Below this is a 'changelog' link. A note about checksums follows, with a link to 'SHA256 checksums, MD5 checksums'. A 'Note' section advises upgrading guest additions after an upgrade. A 'VirtualBox 6.1.10 Oracle VM VirtualBox Extension Pack' section is highlighted with a red box, showing a link to 'All supported platforms'. A note next to it says '扩展插件也一并下载, 后期会使用' (Extensions will also be downloaded, used later). A note at the bottom states: 'Support for USB 2.0 and USB 3.0 devices, VirtualBox RDP, disk encryption, NVMe and PXE boot for Intel cards. See this chapter from the User Manual for a VirtualBox Personal Use and Evaluation License (PUEL). Please install the same version extension pack as your installed version of VirtualBox.' A 'VirtualBox 6.1.10 Software Developer Kit (SDK)' section is also present.

在 virtual box 中添加扩展工具



The screenshot shows the Oracle VM VirtualBox Manager interface. On the left, there are two virtual machines: 'VM01_ROSdemo' (已关闭) and 'VM02_ROSTutorials' (正在运行). The 'VM02_ROSTutorials' tab is selected. A red box highlights the '工具' (Tools) icon in the toolbar. A red arrow points from this icon to the '全局设定' (Global Settings) button in the toolbar. The '全局设定' button is also highlighted with a red box. The main window displays the settings for 'VM02_ROSTutorials'. A red arrow points from the 'VM02_ROSTutorials' tab to the '扩展' (Extensions) tab in the 'VirtualBox - 全局设定' dialog box, which is also highlighted with a red box. The dialog box lists the 'Oracle VM VirtualBox Extension Pack' (版本 6.1.10r138449) and has a button labeled '添加下载好的扩展工具' (Add downloaded extension tools).

在虚拟机中添加 USB 设备



重启后，使用 `ll /dev/ttyUSB*` 或 `ll /dev/ttyACM*` 即可查看新接入的设备。

④其他

其他设置，比如输入法可以根据喜好自行下载安装。

ubuntu 20.04 鼠标右击没有创建文件选项，如果想要设置此选项，可以进入 `主目录` 下的 `模板` 目录，使用 `gedit` 创建一个空文本文档，以后，鼠标右击就可以添加新建文档选项，并且创建的文档与当前自定义的文档名称一致

....

1.2.4 安装 ROS

Ubuntu 安装完毕后，就可以安装 ROS 操作系统了，大致步骤如下：

1. 配置ubuntu的软件和更新；
2. 设置安装源；
3. 设置key；
4. 安装；
5. 配置环境变量。

1. 配置ubuntu的软件和更新

配置ubuntu的软件和更新，允许安装不经认证的软件。

首先打开“软件和更新”对话框，具体可以在Ubuntu搜索按钮中搜索。

打开后按照下图进行配置（确保勾选了“restricted”，“universe”，“multiverse.”）



2. 设置安装源

官方默认安装源：

```
1 sudo sh -c 'echo "deb
http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```

或来自国内清华的安装源

```
1 sudo sh -c '.
 /etc/lsb-release && echo "deb
 http://mirrors.tuna.tsinghua.edu.cn/ros/ubuntu/
 `lsb_release -cs` main" >
 /etc/apt/sources.list.d/ros-latest.list'
```

或来自国内中科大的安装源

```
1 sudo sh -c '.
 /etc/lsb-release && echo "deb
 http://mirrors.ustc.edu.cn/ros/ubuntu/ `lsb_release
 -cs` main" > /etc/apt/sources.list.d/ros-
 latest.list'
```

PS:

1. 回车后,可能需要输入管理员密码
2. 建议使用国内资源, 安装速度更快。

3.设置key

```
1 sudo apt-key adv --keyserver
 'hkp://keyserver.ubuntu.com:80' --recv-key
 C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
```

4.安装

首先需要更新 apt(以前是 apt-get, 官方建议使用 apt 而非 apt-get), apt 是用于从互联网仓库搜索、安装、升级、卸载软件或操作系统的工具。

```
1 sudo apt update
```

等待...

然后, 再安装所需类型的 ROS:ROS 多个类型:Desktop-Full、Desktop、ROS-Base。这里介绍较为常用的Desktop-Full(官方推荐)安装: ROS, rqt, rviz, robot-generic libraries, 2D/3D simulators, navigation and 2D/3D perception

```
1 sudo apt install ros-noetic-desktop-full
```

等待.....(比较耗时)

友情提示: 由于网络原因, 导致连接超时, 可能会安装失败, 如下所示:

```
错误:321 http://cn.archive.ubuntu.com/ubuntu focal/main amd64 libfreetype-dev amd64 2.10.1-2
 504 Gateway Timeout [IP: 117.128.6.16 80]

错误:522 http://cn.archive.ubuntu.com/ubuntu focal/main amd64 libpcre32-3 amd64 2:8.39-12build1
 无法发起与 cn.archive.ubuntu.com:80 (2001:67c:1562::18) 的连接 - connect (101: 网络不可达) 无法发起与

E: 无法下载 http://cn.archive.ubuntu.com/ubuntu/pool/universe/libv/libva/va-driver-all_2.7.0-2_amd64.deb
  的连接 - connect (101: 网络不可达) 无法发起与 cn.archive.ubuntu.com:80 (2001:67c:1562::15) 的连接 - co
E: 无法下载 http://cn.archive.ubuntu.com/ubuntu/pool/main/libv/libvdpau/vdpau-driver-all_1.3-1ubuntu2_amd64.deb
  的连接 - connect (101: 网络不可达) 无法发起与 cn.archive.ubuntu.com:80 (2001:67c:1562::15) 的连接 - co
E: 无法下载 http://cn.archive.ubuntu.com/ubuntu/pool/universe/p/proj/proj-bin_6.3.1-1_amd64.deb
  connect (101: 网络不可达) 无法发起与 cn.archive.ubuntu.com:80 (2001:67c:1562::15) 的连接 - connect (101: 网络不可达) 无法发起与
E: 有几个软件包无法下载, 要不运行 apt-get update 或者加上 --fix-missing 的选项再试试?
```

可以多次重复调用 `更新` 和 `安装`命令, 直至成功。

5.配置环境变量

配置环境变量, 方便在任意 终端中使用 ROS。

```
1 echo "source /opt/ros/noetic/setup.bash" >>
  ~/.bashrc
2 source ~/.bashrc
```

6.卸载

如果需要卸载ROS可以调用如下命令:

```
1 sudo apt remove ros-noetic-*
```

注意: 在 ROS 版本 noetic 中无需构建软件包的依赖关系, 没有 `rosdep` 的相关安装与配置。

另请参考: <http://wiki.ros.org/noetic/Installation/Ubuntu>。

后记

6. 安装构建依赖

在 noetic 最初发布时，和其他历史版本稍有差异的是：没有安装构建依赖这一步骤。随着 noetic 不断完善，官方补齐了这一操作。

首先安装构建依赖的相关工具

```
1 sudo apt install python3-rosdep python3-rosinstall  
python3-rosinstall-generator python3-wstool build-  
essential
```

ROS中使用许多工具前，要求需要初始化rosdep(可以安装系统依赖) -- 上一步实现已经安装过了。

```
1 sudo apt install python3-rosdep
```

初始化rosdep

```
1 sudo rosdep init  
2 rosdep update
```

如果一切顺利的话，rosdep 初始化与更新的打印结果如下：

```
rosnoetic@rosnoetic-VirtualBox:~$ sudo rosdep init  
Wrote /etc/ros/rosdep/sources.list.d/20-default.list  
Recommended: please run  
  
rosdep update
```

```
rosnoetic@rosnoetic-VirtualBox:~$ rosdep update
reading in sources list data from /etc/ros/rosdep/sources.list.d
Hit https://gitee.com/zhao-xuzuo/rosdistro/raw/master/rosdep/osx-homebrew.yaml
Hit https://gitee.com/zhao-xuzuo/rosdistro/raw/master/rosdep/base.yaml
Hit https://gitee.com/zhao-xuzuo/rosdistro/raw/master/rosdep/python.yaml
Hit https://gitee.com/zhao-xuzuo/rosdistro/raw/master/rosdep/ruby.yaml
Hit https://gitee.com/zhao-xuzuo/rosdistro/raw/master/releases/fuerte.yaml
Query rosdistro index https://gitee.com/zhao-xuzuo/rosdistro/raw/master/index-v4.yaml
Skip end-of-life distro "ardent"
Skip end-of-life distro "bouncy"
Skip end-of-life distro "crystal"
Add distro "dashing"
Skip end-of-life distro "eloquent"
Add distro "foxy"
Skip end-of-life distro "groovy"
Skip end-of-life distro "hydro"
Skip end-of-life distro "indigo"
Skip end-of-life distro "jade"
Add distro "kinetic"
Skip end-of-life distro "lunar"
Add distro "melodic"
Add distro "noetic"
Add distro "rolling"
updated cache in /home/rosnoetic/.ros/rosdep/sources.cache
```

但是，在 rosdep 初始化时，多半会抛出异常。

问题:

```
rosnoetic@rosnoetic-VirtualBox:~$ sudo rosdep init
[sudo] rosnoetic 的密码:
ERROR: cannot download default sources list from:
https://raw.githubusercontent.com/ros/rosdistro/master/rosdep/sources.list.d/20-default.list
Website may be down.
```

原因:

境外资源被屏蔽。

解决:

百度或google搜索，解决方式有多种(<https://github.com/ros/rosdistro/issues/9721>)，可惜在 ubuntu20.04 下，集体失效。

新思路:将相关资源备份到 gitee,修改 rosdep 源码,重新定位资源。

实现:

1.先打开资源备份路径:<https://gitee.com/zhao-xuzuo/rosdistro>，打开 rosdistro/rosdep/sources.list.d/20-default.list 文件留作备用(主要是复用 URL 的部分内容:gitee.com/zhao-xuzuo/rosdistro/raw/master)。

```

1 # os-specific listings first
2 yaml https://gitee.com/zhao-xuzuo/rosdistro/raw/master/rosdep/osx-homebrew.yaml osx
3
4 # generic
5 yaml https://gitee.com/zhao-xuzuo/rosdistro/raw/master/rosdep/base.yaml
6 yaml https://gitee.com/zhao-xuzuo/rosdistro/raw/master/rosdep/python.yaml
7 yaml https://gitee.com/zhao-xuzuo/rosdistro/raw/master/rosdep/ruby.yaml
8 gbpdistro https://gitee.com/zhao-xuzuo/rosdistro/raw/master/releases/fuerte.yaml fuerte
9
10 # newer distributions (Groovy, Hydro, ...) must not be listed anymore, they are being fetched from the rosdistro index.yaml instead

```

2. 进入"/usr/lib/python3/dist-packages/" 查找rosdep中和 raw.githubusercontent.com 相关的内容，调用命令：

```
1 find . -type f | xargs grep "raw.githubusercontent"
```

```

rosnoetic@rosnoetic-VirtualBox:/usr/lib/python3/dist-packages$ find . -type f | xargs grep "raw.githubusercontent"
grep: ./setuptools/command/launcher: 没有那个文件或目录
grep: manifest.xml: 没有那个文件或目录
grep: ./setuptools/script: 没有那个文件或目录
grep: (dev).tmp: 没有那个文件或目录
./requests-2.22.0.egg-info/PKG-INFO:      [![image](https://raw.githubusercontent.com/requests/requests/master/docs/_static/requests-logo-small.png)](http://docs.python-requests.org/)
匹配到二进制文件 ./rosdistro/manifest_provider/_pycache_/_github.cpython-38.pyc
./rosdistro/manifest_provider/github.py:    url = 'https://raw.githubusercontent.com/%s/%s/package.xml' % (path, release_tag)
./rosdistro/manifest_provider/github.py:    url = 'https://raw.githubusercontent.com/%s/%s/%s' % \
./rosdistro/_init__.py:DEFAULT_INDEX_URL = 'https://raw.githubusercontent.com/ros/rosdistro/master/index-v4.yaml'
匹配到二进制文件 ./rosdistro/_pycache_/_init__.cpython-38.pyc
匹配到二进制文件 ./rosdep2/_pycache_/_rep3.cpython-38.pyc
匹配到二进制文件 ./rosdep2/_pycache_/_sources_list.cpython-38.pyc
匹配到二进制文件 ./rosdep2/_pycache_/_gbpdistro_support.cpython-38.pyc
./rosdep2/gbpdistro_support.py:FUERTE_GBPDISTRO_URL = 'https://raw.githubusercontent.com/ros/rosdistro/' \
./rosdep2/sources_list.py:DEFAULT_SOURCES_LIST_URL = 'https://raw.githubusercontent.com/ros/rosdistro/master/rosdep/sources.list.d/20-default.list'
./rosdep2/rep3.py:REP3_TARGETS_URL = 'https://raw.githubusercontent.com/ros/rosdistro/master/releases/targets.yaml'

```

3. 修改相关文件，主要有：

./rosdistro/init.py、./rosdep2/gbpdistro_support.py、./rosdep2/sources_list.py、./rosdep2/rep3.py。可以使用 sudo gedit 命令修改文件：

文件中涉及的 URL 内容，如果是： raw.githubusercontent.com/ros/rosdistro/master 都替换成步骤1中准备的 gitee.com/zhao-xuzuo/rosdistro/raw/master 即可。

修改完毕，再重新执行命令：

```

1 sudo rosdep init
2 rosdep update

```

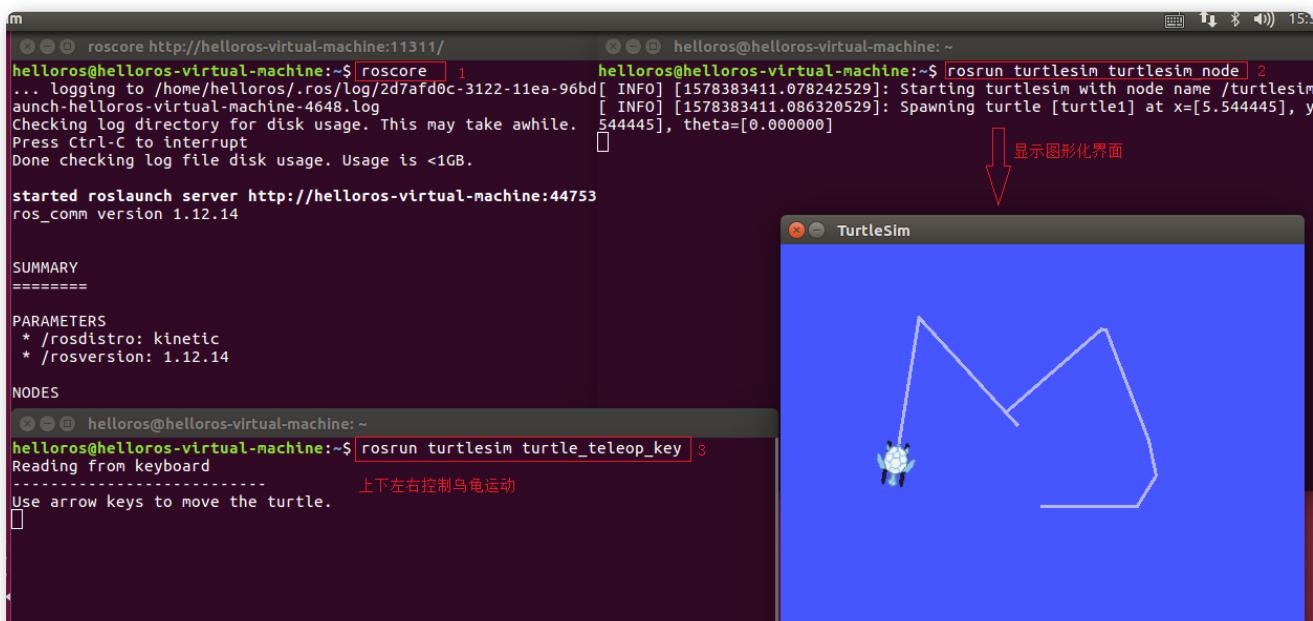
就可以正常实现 rosdep 的初始化与更新了。

1.2.5 测试 ROS

ROS 内置了一些小程序，可以通过运行这些小程序以检测 ROS 环境是否可以正常运行

1. 首先启动三个命令行(ctrl + alt + T)
2. 命令行1键入:roscore
3. 命令行2键入:rosrun turtlesim turtlesim_node(此时会弹出图形化界面)
4. 命令行3键入:rosrun turtlesim turtle_teleop_key(在3中可以通过上下左右控制2中乌龟的运动)

最终结果如下所示:



注意：光标必须聚焦在键盘控制窗口，否则无法控制乌龟运动。

1.2.6 资料:其他ROS版本安装

我们的教程采用的是ROS的最新版本noetic，不过noetic较之于之前的ROS版本变动较大且部分功能包还未更新，因此如果有需要(比如到后期实践阶段，由于部分重要的功能包还未更新，需要ROS降级)，也会安装之前版本的ROS，在此，建议选用的版本是melodic或kinetic。

接下来，就以melodic为例演示ROS历史版本安装(当然先要准备与melodic对应的Ubuntu18.04):

1.配置ubuntu的软件和更新

首先打开“软件和更新”对话框，打开后按照下图进行配置（确保你的"restricted", "universe, " 和 "multiverse."前是打上勾的）

00ROS安装之ubuntu准备

2.安装源

官方默认安装源：

```
1 sudo sh -c 'echo "deb
http://packages.ros.org/ros/ubuntu $(lsb_release -
sc) main" > /etc/apt/sources.list.d/roslatest.list'
```

或来自国内中科大的安装源

```
1 sudo sh -c '. /etc/lsb-release && echo "deb
http://mirrors.ustc.edu.cn/ros/ubuntu/ `lsb_release
-cs` main" > /etc/apt/sources.list.d/roslatest.list'
```

或来自国内清华的安装源

```
1 sudo sh -c '. /etc/lsb-release && echo "deb
http://mirrors.tuna.tsinghua.edu.cn/ros/ubuntu/
`lsb_release -cs` main" >
/etc/apt/sources.list.d/roslatest.list'
```

PS:回车后,可能需要输入管理员密码

3.设置key

```
1 sudo apt-key adv --keyserver
'hkp://keyserver.ubuntu.com:80' --recv-key
C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
```

4. 安装

首先需要更新 apt(以前是 apt-get, 官方建议使用 apt 而非 apt-get), apt 是用于从互联网仓库搜索、安装、升级、卸载软件或操作系统的工具。

```
1 sudo apt update
```

等待...

然后, 再安装所需类型的 ROS:ROS 多个类型:**Desktop-Full**、**Desktop**、**ROS-Base**。这里介绍较为常用的**Desktop-Full**(官方推荐)安装: ROS, rqt, rviz, robot-generic libraries, 2D/3D simulators, navigation and 2D/3D perception

```
1 sudo apt install ros-melodic-desktop-full
```

等待...

5. 环境设置

配置环境变量, 方便在任意 终端中使用 ROS。

```
1 echo "source /opt/ros/melodic/setup.bash" >>
~/ .bashrc
2 source ~/ .bashrc
```

6. 安装构建依赖

首先安装构建依赖的相关工具

```
1 sudo apt install python-rosdep python-rosinstall
python-rosinstall-generator python-wstool build-
essential
```

然后安装rosdep(可以安装系统依赖)

```
1 sudo apt install python-rosdep
```

初始化rosdep

```
1 sudo rosdep init  
2 rosdep update
```

注意:

当执行到最后 sudo rosdep init 时，可能会抛出异常；

错误提示:

ERROR: cannot download default sources list from:

<https://raw.githubusercontent.com/ros/rosdistro/master/rosdep/sources.list.d/20-default.list>

Website may be down.

原因:

境外资源被屏蔽

解决思路:

查询错误提示中域名的IP地址，然后修改 /etc/hosts 文件，添加域名与IP映射

实现:

1. 访问域名查询网址:<https://site.ip138.com/>

2. 查询域名ip，搜索框中输入: raw.githubusercontent.com，自由复制一个查询到的IP

IP或域名查询

raw.githubusercontent.com X 查询 访问

域名注册com42元 top10元 cn18元 xyz12元 广告QQ:1073353388

 **买卖域名, 网站, 自媒体
上中介网!** 广告

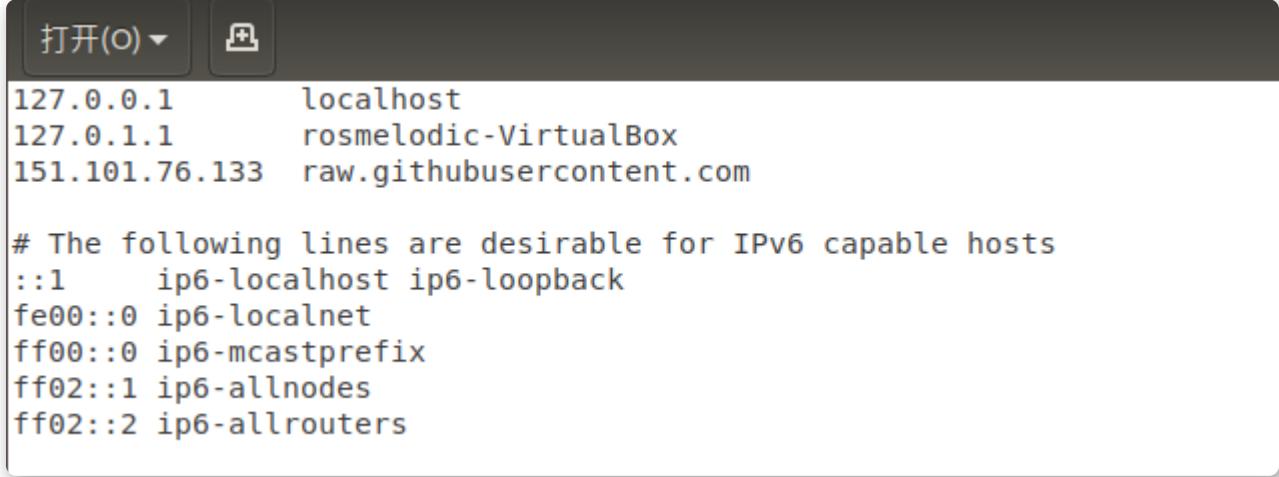
**域名被劫持? DNS被污染?
全面检测解决DNS劫持** 广告

IP	子域名	备案	Whois	快照
raw.githubusercontent.com服务器IP:				
当前解析:				
151.101.108.133			日本 东京	
151.101.0.133			美国	
0.0.0.0			保留地址	
151.101.228.133			日本 东京	
151.101.196.133			美国 加利福尼亚 洛杉矶	
151.101.8.133			新加坡	
151.101.76.133			中国 香港	
151.101.24.133			美国 加利福尼亚 洛杉矶	

3.修改 /etc/hosts 文件，命令:

```
1 sudo gedit /etc/hosts
```

添加内容:151.101.76.133 raw.githubusercontent.com (查询到的ip与域名), 保存并退出。



```

127.0.0.1      localhost
127.0.1.1      rosmelodic-VirtualBox
151.101.76.133 raw.githubusercontent.com

# The following lines are desirable for IPv6 capable hosts
::1      ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters

```

或者，也可以使用 vi 或 vim 修改。

4.重新执行rosdep初始化与更新命令，如果rosdep update 抛出异常，基本都是网络原因导致的(建议使用手机热点)，多尝试几次即可。

```

rosmelodic@rosmelodic-VirtualBox:~$ sudo rosdep init
Wrote /etc/ros/rosdep/sources.list.d/20-default.list
Recommended: please run

rosdep update

```

```

rosmelodic@rosmelodic-VirtualBox:~$ rosdep update
reading in sources list data from /etc/ros/rosdep/sources.list.d
Hit https://raw.githubusercontent.com/ros/rosdistro/master/rosdep/osx-homebrew.yaml
Hit https://raw.githubusercontent.com/ros/rosdistro/master/rosdep/base.yaml
Hit https://raw.githubusercontent.com/ros/rosdistro/master/rosdep/python.yaml
Hit https://raw.githubusercontent.com/ros/rosdistro/master/rosdep/ruby.yaml
Hit https://raw.githubusercontent.com/ros/rosdistro/master/releases/fuerte.yaml
Query rosdistro index https://raw.githubusercontent.com/ros/rosdistro/master/index-v4.yaml
Skip end-of-life distro "ardent"
Skip end-of-life distro "bouncy"
Skip end-of-life distro "crystal"
Add distro "dashing"
Add distro "eloquent"
Add distro "foxy"
Skip end-of-life distro "groovy"
Skip end-of-life distro "hydro"
Skip end-of-life distro "indigo"
Skip end-of-life distro "jade"
Add distro "kinetic"
Skip end-of-life distro "lunar"
Add distro "melodic"
Add distro "noetic"
Add distro "rolling"
updated cache in /home/rosmelodic/.ros/rosdep/sources.cache

```

综上，历史版本的安装与noetic流程类似，只是多出了“构建功能包依赖关系”的步骤。

另请参考：<http://wiki.ros.org/melodic/Installation/Ubuntu>

1.3 ROS快速体验

编写 ROS 程序，在控制台输出文本: Hello World，分别使用 C++ 和 Python 实现。

1.3.1 HelloWorld实现简介

ROS中涉及的编程语言以C++和Python为主，ROS中的大多数程序两者都可以实现，在本系列教程中，每一个案例也都会分别使用C++和Python两种方案演示，大家可以根据自身情况选择合适的实现方案。

ROS中的程序即便使用不同的编程语言，实现流程也大致类似，以当前 HelloWorld程序为例，实现流程大致如下：

1. 先创建一个工作空间；
2. 再创建一个功能包；
3. 编辑源文件；
4. 编辑配置文件；
5. 编译并执行。

上述流程中，C++和Python只是在步骤3和步骤4的实现细节上存在差异，其他流程基本一致。本节先实现C++和Python程序编写的通用部分步骤1与步骤2，1.3.2节和1.3.3节再分别使用C++和Python编写HelloWorld。

1.创建工作空间并初始化

```
1 mkdir -p 自定义空间名称/src
2 cd 自定义空间名称
3 catkin_make
```

上述命令，首先会创建一个工作空间以及一个 src 子目录，然后再进入工作空间调用 catkin_make命令编译。

2.进入 src 创建 ros 包并添加依赖

```
1 cd src
2 catkin_create_pkg 自定义ROS包名 roscpp rospy std_msgs
```

上述命令，会在工作空间下生成一个功能包，该功能包依赖于 roscpp、rospy 与 std_msgs，其中roscpp是使用C++实现的库，而rospy则是使用python实现的库，std_msgs是标准消息库，创建ROS功能包时，一般都会依赖这三个库实现。

注意: 在ROS中，虽然实现同一功能时，C++和Python可以互换，但是具体选择哪种语言，需要视需求而定，因为两种语言相较而言:C++运行效率高但是编码效率低，而Python则反之，基于二者互补的特点，ROS设计者分别设计了roscpp与rospy库，前者旨在成为ROS的高性能库，而后者则一般用于对性能无要求的场景，旨在提高开发效率。

1.3.2 HelloWorld(C++版)

本节内容基于1.3.1，假设你已经创建了ROS的工作空间，并且创建了ROS的功能包，那么就可以进入核心步骤了，使用C++编写程序实现：

1.进入 ros 包的 src 目录编辑源文件

```
1 cd 自定义的包
```

C++源码实现(文件名自定义)

```

1 #include "ros/ros.h"
2
3 int main(int argc, char *argv[])
4 {
5     //执行 ros 节点初始化
6     ros::init(argc,argv,"hello");
7     //创建 ros 节点句柄(非必须)
8     ros::NodeHandle n;
9     //控制台输出 hello world
10    ROS_INFO("hello world!");
11
12    return 0;
13 }
```

2. 编辑 ros 包下的 Cmakelist.txt 文件

```

1 add_executable(步骤3的源文件名
2     src/步骤3的源文件名.cpp
3 )
4 target_link_libraries(步骤3的源文件名
5     ${catkin_LIBRARIES}
6 )
```

3. 进入工作空间目录并编译

```

1 cd 自定义空间名称
2 catkin_make
```

生成 build devel

4. 执行

先启动命令行1：

```
1 roscore
```

再启动命令行2:

```
1 cd 工作空间
2 source ./devel/setup.bash
3 rosrun 包名 C++节点
```

命令行输出: HelloWorld!

PS: `source ~/工作空间/devel/setup.bash` 可以添加进 `.bashrc` 文件, 使用上更方便

添加方式1: 直接使用 gedit 或 vi 编辑 `.bashrc` 文件, 最后添加该内容

添加方式2: `echo "source ~/工作空间/devel/setup.bash" >> ~/ .bashrc`

1.3.3 HelloWorld(Python版)

本节内容基于1.3.1, 假设你已经创建了ROS的工作空间, 并且创建了ROS的功能包, 那么就可以进入核心步骤了, 使用Python编写程序实现:

1.进入 ros 包添加 scripts 目录并编辑 python 文件

```
1 cd ros包
2 mkdir scripts
```

新建 python 文件: (文件名自定义)

```

1  #! /usr/bin/env python
2
3  """
4      Python 版 HelloWorld
5
6  """
7  import rospy
8
9  if __name__ == "__main__":
10      rospy.init_node("Hello")
11      rospy.loginfo("Hello World!!!!")

```

2.为 python 文件添加可执行权限

```
1 chmod +x 自定义文件名.py
```

3.编辑 ros 包下的 CamkeList.txt 文件

```

1 catkin_install_python(PROGRAMS scripts/自定义文件名.py
2   DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
3 )

```

4.进入工作空间目录并编译

```

1 cd 自定义空间名称
2 catkin_make

```

5.进入工作空间目录并执行

先启动命令行1:

```
1 roscore
```

再启动命令行2:

```
1 cd 工作空间  
2 source ./devel/setup.bash  
3 rosrun 包名 自定义文件名.py
```

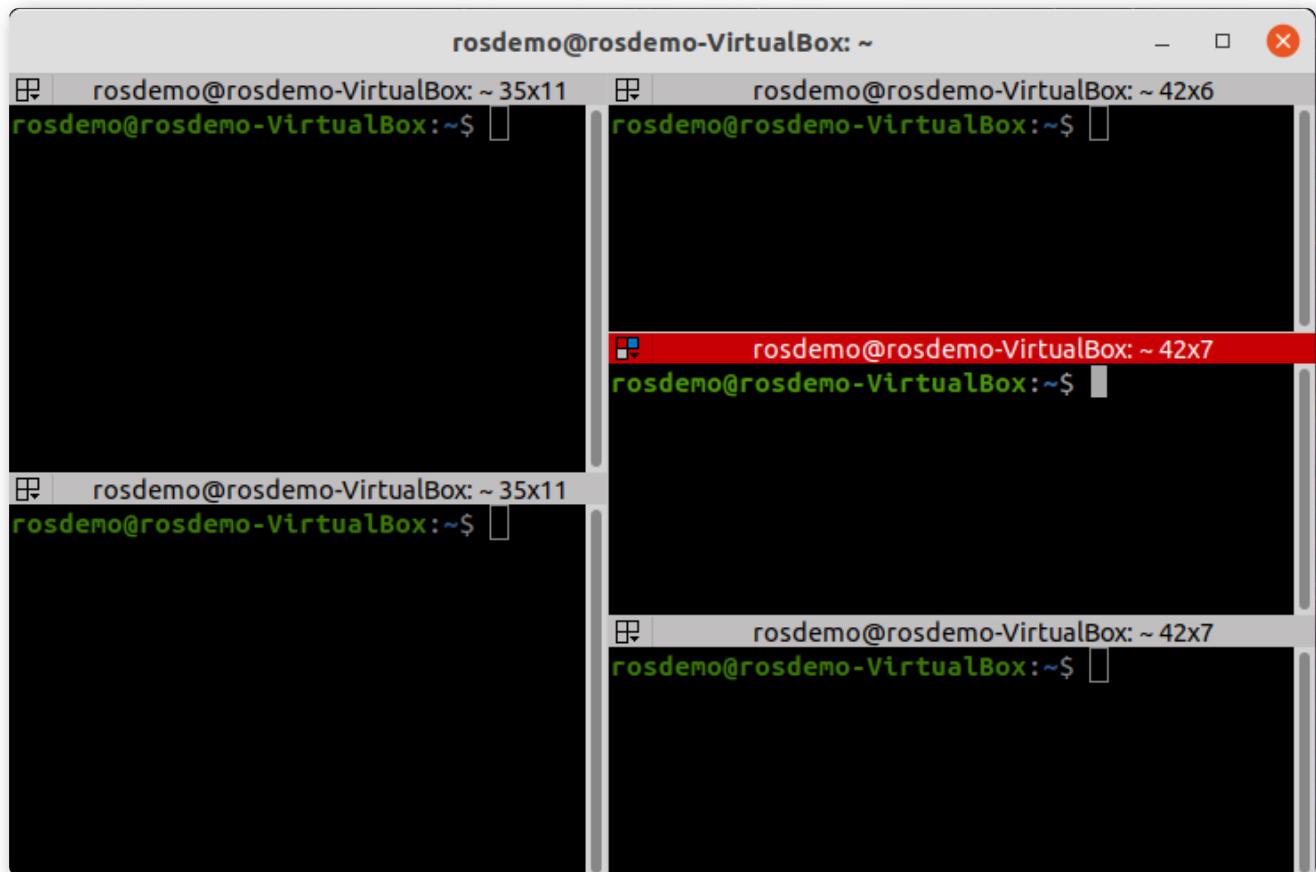
输出结果: Hello World!!!!

1.4 ROS集成开发环境搭建

和大多数开发环境一样，理论上，在 ROS 中，只需要记事本就可以编写基本的 ROS 程序，但是工欲善其事必先利其器，为了提高开发效率，可以先安装集成开发工具和使用方便的工具:终端、IDE....

1.4.1 安装终端

在 ROS 中，需要频繁的使用到终端，且可能需要同时开启多个窗口，推荐一款较为好用的终端:Terminator。效果如下:



1. 安装

```
1 sudo apt install terminator
```

2. 添加到收藏夹

显示应用程序 ---> 搜索 terminator ---> 右击 选择 添加到收藏夹

3. Terminator 常用快捷键

第一部份：关于在同一个标签内的操作

1 Alt+Up	// 移动到上面的终端
2 Alt+Down	// 移动到下面的终端
3 Alt+Left	// 移动到左边的终端
4 Alt+Right	// 移动到右边的终端
5 Ctrl+Shift+0	// 水平分割终端
6 Ctrl+Shift+E	// 垂直分割终端
7 Ctrl+Shift+Right	// 在垂直分割的终端中将分割条向右移动
8 Ctrl+Shift+Left	// 在垂直分割的终端中将分割条向左移动
9 Ctrl+Shift+Up	// 在水平分割的终端中将分割条向上移动
10 Ctrl+Shift+Down	// 在水平分割的终端中将分割条向下移动
11 Ctrl+Shift+S	// 隐藏/显示滚动条
12 Ctrl+Shift+F	// 搜索
13 Ctrl+Shift+C	// 复制选中的内容到剪贴板
14 Ctrl+Shift+V	// 粘贴剪贴板的内容到此处
15 Ctrl+Shift+W	// 关闭当前终端
16 Ctrl+Shift+Q	// 退出当前窗口，当前窗口的所有终端都将被关闭
17 Ctrl+Shift+X	// 最大化显示当前终端
18 Ctrl+Shift+Z	// 最大化显示当前终端并使字体放大

```

19 Ctrl+Shift+N or Ctrl+Tab      //移动到下一个终端
20 Ctrl+Shift+P or Ctrl+Shift+Tab //Ctrl+Shift+Tab 移动到之前的一个终端

```

第二部份：有关各个标签之间的操作

1 F11	//全屏开关
2 Ctrl+Shift+T	//打开一个新的标签
3 Ctrl+PageDown	//移动到下一个标签
4 Ctrl+PageUp	//移动到上一个标签
5 Ctrl+Shift+PageDown	//将当前标签与其后一个标签交换位置
6 Ctrl+Shift+PageUp	//将当前标签与其前一个标签交换位置
7 Ctrl+Plus (+)	//增大字体
8 Ctrl+Minus (-)	//减小字体
9 Ctrl+Zero (0)	//恢复字体到原始大小
10 Ctrl+Shift+R	//重置终端状态
11 Ctrl+Shift+G	//重置终端状态并clear屏幕
12 Super+g	//绑定所有的终端，以便向一个输入能够输入到所有的终端
13 Super+Shift+G	//解除绑定
14 Super+t	//绑定当前标签的所有终端，向一个终端输入的内容会自动输入到其他终端
15 Super+Shift+T	//解除绑定
16 Ctrl+Shift+I	//打开一个窗口，新窗口与原来的窗口使用同一个进程
17 Super+i	//打开一个新窗口，新窗口与原来的窗口使用不同的进程

1.4.2 安装VScode

VSCode 全称 Visual Studio Code，是微软出的一款轻量级代码编辑器，免费、开源而且功能强大。它支持几乎所有主流的程序语言的语法高亮、智能代码补全、自定义热键、括号匹配、代码片段、代码对比 Diff、GIT 等特性，支持插件扩展，并针对网页开发和云端应用开发做了优化。软件跨平台支持

Win、Mac 以及 Linux。

1. 下载

vscode 下载: <https://code.visualstudio.com/docs?start=true>

历史版本下载链接: <https://code.visualstudio.com/updates>

2. vscode 安装与卸载

2.1 安装

方式1: 双击安装即可(或右击选择安装)

方式2: `sudo dpkg -i xxxx.deb`

2.2 卸载

```
1 sudo dpkg --purge code
```

3. vscode 集成 ROS 插件

使用 VScode 开发 ROS 程序, 需要先安装一些插件, 常用插件如下:



4. vscode 使用_基本配置

4.1 创建 ROS 工作空间

```
1 mkdir -p xxx_ws/src (必须得有 src)  
2 cd xxx_ws  
3 catkin_make
```

4.2 启动 vscode

进入 `xxx_ws` 启动 vscode

```

1 cd xxx_ws
2 code .

```

4.3 vscode 中编译 ros

快捷键 **ctrl + shift + B** 调用编译，选择: `catkin_make:build`

可以点击配置设置为默认，修改`.vscode/tasks.json`文件

```

1  {
2 // 有关 tasks.json 格式的文档, 请参见
3 // https://go.microsoft.com/fwlink/?LinkId=733558
4   "version": "2.0.0",
5   "tasks": [
6     {
7       "label": "catkin_make:debug", //代表提示
8         "type": "shell", //可以选择shell或者
9         process,如果是shell代码是在shell里面运行一个命令,如果是
10        process代表作为一个进程来运行
11         "command": "catkin_make", //这个是我们需要
12         运行的命令
13         "args": [], //如果需要在命令后面加一些后缀,
14         可以写在这里,比如-
15         DCATKIN_WHITELIST_PACKAGES="pac1;pac2"
16         "group": {
17           {"kind": "build", "isDefault": true},
18           "presentation": {
19             "reveal": "always" //可选always或者
20             silence, 代表是否输出信息
21           },
22           "problemMatcher": "$msCompile"
23         }
24       ]
25     }
26   }

```

4.4 创建 ROS 功能包

选定 src 右击 ---> create catkin package

设置包名 添加依赖

07vscode_新建ROS包

4.5 C++ 实现

在功能包的 src 下新建 cpp 文件

```

1  /*
2   * 控制台输出 HelloVSCode !!!
3
4  */
5 #include "ros/ros.h"
6
7 int main(int argc, char *argv[])
8 {
9     setlocale(LC_ALL, "");
10    //执行节点初始化
11    ros::init(argc, argv, "HelloVSCode");
12
13    //输出日志
14    ROS_INFO("Hello VSCode!!!哈哈哈哈哈哈哈哈哈哈");
15    return 0;
16 }
```

PS1: 如果没有代码提示

修改 .vscode/c_cpp_properties.json

设置 "cppStandard": "c++17"

PS2: main 函数的参数不可以被 const 修饰

PS3: 当ROS__INFO 终端输出有中文时，会出现乱码

INFO: ?????????????????????????????

解决办法：在函数开头加入下面代码的任意一句

```
1 setlocale(LC_CTYPE, "zh_CN.utf8");
2 setlocale(LC_ALL, "");
```

4.6 python 实现

在 功能包 下新建 scripts 文件夹，添加 python 文件，并添加可执行权限

```
1 #! /usr/bin/env python
2 """
3     Python 版本的 HelloVScode，执行在控制台输出
4     HelloVScode
5     实现：
6     1. 导包
7     2. 初始化 ROS 节点
8     3. 日志输出 HelloWorld
9
10 """
11
12 import rospy # 1. 导包
13
14 if __name__ == "__main__":
15
16     rospy.init_node("Hello_Vscode_p") # 2. 初始化
17     ROS 节点
18     rospy.loginfo("Hello VScode, 我是 Python ....")
19     #3. 日志输出 HelloWorld
```

4.7 配置 CMakeLists.txt

C++ 配置：

```

1 add_executable(节点名称
2   src/C++源文件名.cpp
3 )
4 target_link_libraries(节点名称
5   ${catkin_LIBRARIES}
6 )

```

Python 配置:

```

1 catkin_install_python(PROGRAMS scripts/自定义文件名.py
2   DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
3 )

```

4.8 编译执行

编译: **ctrl + shift + B**

执行: 和之前一致, 只是可以在 VScode 中添加终端, 首先执行: `source ./devel/setup.bash`

 08vscode_执行

PS:

如果不编译直接执行 python 文件, 会抛出异常。

1. 第一行解释器声明, 可以使用绝对路径定位到 python3 的安装路径 `#!/usr/bin/python3`, 但是不建议

2. 建议使用 `#!/usr/bin/env python` 但是会抛出异常: `/usr/bin/env: “python” : 没有那个文件或目录`

3. 解决1: `#!/usr/bin/env python3` 直接使用 python3 但存在问题: 不兼容之前的 ROS 相关 python 实现

4. 解决2: 创建一个链接符号到 python 命令: `sudo ln -s /usr/bin/python3 /usr/bin/python`

5.其他 IDE

ROS 开发可以使用的 IDE 还是比较多人的，除了上述的 VScode，还有 Eclipse、QT、PyCharm、Roboware, 详情可以参考官网介绍: <http://wiki.ros.org/IDEs>

QT Creator Plugin for ROS，参考教程: <https://ros-qt-c-plugin.readthedocs.io/en/latest/>

Roboware 参考: [http://www.roboware.me/#/\(PS](http://www.roboware.me/#/(PS): Roboware 已经停更了，可惜....)

1.4.3 launch文件演示

1.需求



一个程序中可能需要启动多个节点，比如:ROS 内置的小乌龟案例，如果要控制乌龟运动，要启动多个窗口，分别启动 roscore、乌龟界面节点、键盘控制节点。如果每次都调用 rosrun 逐一启动，显然效率低下，如何优化？

官方给出的优化策略是使用 launch 文件，可以一次性启动多个 ROS 节点。

2.实现

1. 选定功能包右击 ---> 添加 launch 文件夹
2. 选定 launch 文件夹右击 ---> 添加 launch 文件
3. 编辑 launch 文件内容

```

1 <launch>
2   <node pkg="helloworld" type="demo_hello"
3     name="hello" output="screen" />
4   <node pkg="turtlesim"
5     type="turtlesim_node" name="t1"/>
6   <node pkg="turtlesim"
7     type="turtle_teleop_key" name="key1" />
8 </launch>

```

- node ---> 包含的某个节点
- pkg -----> 功能包
- type -----> 被运行的节点文件
- name --> 为节点命名
- output-> 设置日志的输出目标

运行 launch 文件

```
1 rosrun 包名 launch文件名
```

运行结果: 一次性启动了多个节点

1.5 ROS架构

到目前为止，我们已经安装了ROS，运行了ROS中内置的小乌龟案例，并且也编写了ROS小程序，对ROS也有了一个大概的认知，当然这个认知可能还是比较模糊并不清晰的，接下来，我们要从宏观上来介绍一下ROS的架构设计。

立足不同的角度，对ROS架构的描述也是不同的，一般我们可以从设计者、维护者、系统结构与自身结构4个角度来描述ROS结构：

1.设计者

ROS设计者将ROS表述为“ROS = Plumbing + Tools + Capabilities + Ecosystem”

- Plumbing: 通讯机制(实现ROS不同节点之间的交互)

- Tools :工具软件包(ROS中的开发和调试工具)
- Capabilities :机器人高层技能(ROS中某些功能的集合, 比如:导航)
- Ecosystem:机器人生态系统(跨地域、跨软件与硬件的ROS联盟)

2. 维护者

立足维护者的角度: ROS 架构可划分为两大部分

- main: 核心部分, 主要由Willow Garage 和一些开发者设计、提供以及维护。它提供了一些分布式计算的基本工具, 以及整个ROS的核心部分的程序编写。
- universe: 全球范围的代码, 有不同国家的ROS社区组织开发和维护。一种是库的代码, 如OpenCV、PCL等; 库的上一层是从功能角度提供的代码, 如人脸识别, 他们调用下层的库; 最上层的代码是应用级的代码, 让机器人完成某一确定的功能。

3. 系统架构

立足系统架构: ROS 可以划分为三层

- OS 层, 也即经典意义的操作系统

ROS 只是元操作系统, 需要依托真正意义的操作系统, 目前兼容性最好的是 Linux 的 Ubuntu, Mac、Windows 也支持 ROS 的较新版本

- 中间层

是 ROS 封装的关于机器人开发的中间件, 比如:

- 基于 TCP/UDP 继续封装的 TCPROS/UDPROS 通信系统
- 用于进程间通信 Nodelet, 为数据的实时性传输提供支持
- 另外, 还提供了大量的机器人开发实现库, 如: 数据类型定义、坐标变换、运动控制....

- 应用层

功能包, 以及功能包内的节点, 比如: master、turtlesim的控制与运动节点...

4. 自身结构

就 ROS 自身实现而言: 也可以划分为三层

- 文件系统

ROS文件系统级指的是在硬盘上面查看的ROS源代码的组织形式

- 计算图

ROS 分布式系统中不同进程需要进行数据交互, 计算图可以以点对点的网络形式表现数据交互过程, 计算图中的重要概念: 节点(Node)、消息(message)、通信机制主题(*topic*)、通信机制服务(service)

- 开源社区

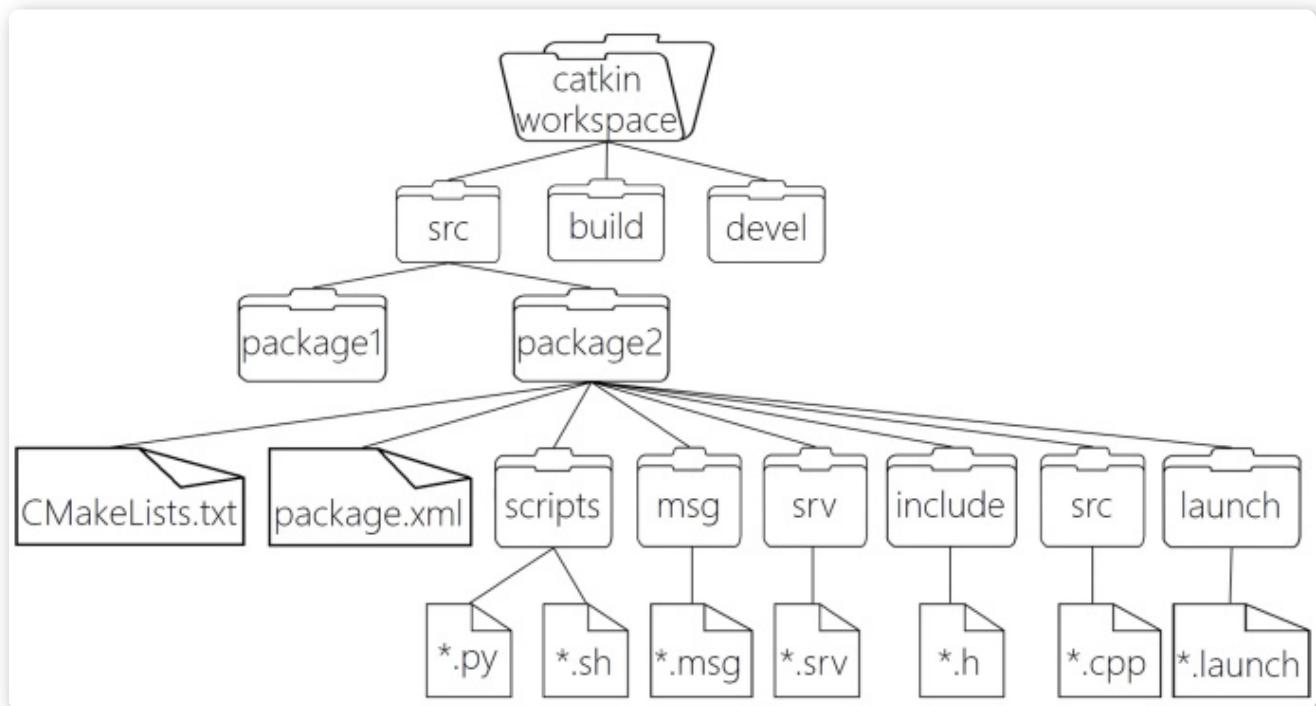
ROS的社区级概念是ROS网络上进行代码发布的一种表现形式

- 发行版 (Distribution) ROS发行版是可以独立安装、带有版本号的一系列综合功能包。ROS发行版像Linux发行版一样发挥类似的作用。这使得ROS软件安装更加容易, 而且能够通过一个软件集合维持一致的版本。
- 软件库 (Repository) ROS依赖于共享开源代码与软件库的网站或主机服务, 在这里不同的机构能够发布和分享各自的机器人软件与程序。
- ROS维基 (ROS Wiki) ROS Wiki是用于记录有关ROS系统信息的主要论坛。任何人都可以注册账户、贡献自己的文件、提供更正或更新、编写教程以及其他行为。网址是<http://wiki.ros.org/>。
- Bug提交系统 (Bug Ticket System) 如果你发现问题或者想提出一个新功能, ROS提供这个资源去做这些。
- 邮件列表 (Mailing list) ROS用户邮件列表是关于ROS的主要交流渠道, 能够像论坛一样交流从ROS软件更新到ROS软件使用中的各种疑问或信息。网址是<http://lists.ros.org/>。
- ROS问答 (ROS Answer) 用户可以使用这个资源去提问题。网址是<https://answers.ros.org/questions/>。
- 博客 (Blog) 你可以看到定期更新、照片和新闻。网址是<https://www.ros.org/news/>, 不过博客系统已经退休, ROS社区取而代之, 网址是<https://discourse.ros.org/>。

现在处于学习的初级阶段，只是运行了ROS的内置案例，编写了简单的ROS实现，因此，受限于当前进度，不会详细介绍所有设计架构中的所有模块，当前只介绍文件系统与计算图，下一章会介绍ROS的通信机制，这也是ROS的核心实现之一。

1.5.1 ROS文件系统

ROS文件系统级指的是在硬盘上ROS源代码的组织形式，其结构大致可以如下图所示：



1 WorkSpace --- 自定义的工作空间

2

3 |--- build: 编译空间，用于存放CMake和catkin的缓存信息、配置信息和其他中间文件。

4

5 |--- devel: 开发空间，用于存放编译后生成的目标文件，包括头文件、动态&静态链接库、可执行文件等。

6

7 |--- src: 源码

8

9 |-- package: 功能包(ROS基本单元)包含多个节点、库与配置文件，包名所有字母小写，只能由字母、数字与下划线组成

10

```

11          |-- CMakeLists.txt 配置编译规则, 比如源文
12         件、依赖项、目标文件
13          |-- package.xml 包信息, 比如: 包名、版本、作
14         者、依赖项... (以前版本是 manifest.xml)
15          |-- scripts 存储python文件
16
17          |-- src 存储C++源文件
18
19          |-- include 头文件
20
21          |-- msg 消息通信格式文件
22
23          |-- srv 服务通信格式文件
24
25          |-- action 动作格式文件
26
27          |-- launch 可一次性运行多个节点
28
29          |-- config 配置信息
30
31          |-- CMakeLists.txt: 编译的基本配置

```

ROS 文件系统中部分目录和文件前面编程中已经有所涉及, 比如功能包的创建、src目录下cpp文件的编写、scripts目录下python文件的编写、launch目录下launch文件的编写, 并且也配置了 package.xml 与 CMakeLists.txt 文件。其他目录下的内容后面教程将会再行介绍, 当前我们主要介绍: package.xml 与 CMakeLists.txt 这两个配置文件。

1.package.xml

该文件定义有关软件包的属性, 例如软件包名称, 版本号, 作者, 维护者以及对其他catkin软件包的依赖性。请注意, 该概念类似于旧版 rosbuild 构建系统中使用的 *manifest.xml* 文件。

```

1 <?xml version="1.0"?>
2 <!-- 格式: 以前是 1, 推荐使用格式 2 -->

```

```
3 <package format="2">
4   <!-- 包名 -->
5   <name>demo01_hello_vscode</name>
6   <!-- 版本 -->
7   <version>0.0.0</version>
8   <!-- 描述信息 -->
9   <description>The demo01_hello_vscode
  package</description>
10
11   <!-- One maintainer tag required, multiple
  allowed, one person per tag -->
12   <!-- Example: -->
13   <!-- <maintainer
  email="jane.doe@example.com">Jane Doe</maintainer>
  -->
14   <!-- 维护人员 -->
15   <maintainer
  email="xuzuo@todo.todo">xuzuo</maintainer>
16
17
18   <!-- One license tag required, multiple allowed,
  one license per tag -->
19   <!-- Commonly used license strings: -->
20   <!-- BSD, MIT, Boost Software License, GPLv2,
  GPLv3, LGPLv2.1, LGPLv3 -->
21   <!-- 许可证信息, ROS核心组件默认 BSD -->
22   <license>TODO</license>
23
24
25   <!-- Url tags are optional, but multiple are
  allowed, one per tag -->
26   <!-- Optional attribute type can be: website,
  bugtracker, or repository -->
27   <!-- Example: -->
28   <!-- <url
  type="website">http://wiki.ros.org/demo01_hello_vs
  code</url> -->
29
30
```

```
31  <!-- Author tags are optional, multiple are
32  allowed, one per tag -->
33  <!-- Authors do not have to be maintainers, but
34  could be -->
35
36
37  <!-- The *depend tags are used to specify
38  dependencies -->
39  <!-- Dependencies can be catkin packages or
40  system dependencies -->
41  <!-- Examples: -->
42  <!-- Use depend as a shortcut for packages that
43  are both build and exec dependencies -->
44  <!-- <depend>roscpp</depend> -->
45  <!-- Note that this is equivalent to the
46  following: -->
47  <!-- <build_depend>roscpp</build_depend> -->
48  <!-- <exec_depend>roscpp</exec_depend> -->
49  <!-- Use build_depend for packages you need at
50  compile time: -->
51
52  <!--
53  <build_depend>message_generation</build_depend> --
54  >
55  <!-- Use build_export_depend for packages you
56  need in order to build against this package: -->
57  <!--
58  <build_export_depend>message_generation</build_exp
59  ort_depend> -->
60  <!-- Use buildtool_depend for build tool
61  packages: -->
62  <!--
63  <buildtool_depend>catkin</buildtool_depend> -->
64  <!-- Use exec_depend for packages you need at
65  runtime: -->
66  <!--
67  <exec_depend>message_runtime</exec_depend> -->
```

```
53  <!-- Use test_depend for packages you need only
  for testing: -->
54  <!-- <test_depend>gtest</test_depend> -->
55  <!-- Use doc_depend for packages you need only
  for building documentation: -->
56  <!-- <doc_depend>doxygen</doc_depend> -->
57  <!-- 依赖的构建工具, 这是必须的 -->
58  <buildtool_depend>catkin</buildtool_depend>
59
60  <!-- 指定构建此软件包所需的软件包 -->
61  <build_depend>roscpp</build_depend>
62  <build_depend>rospy</build_depend>
63  <build_depend>std_msgs</build_depend>
64
65  <!-- 指定根据这个包构建库所需要的包 -->
66  <build_export_depend>roscpp</build_export_depend>
67  <build_export_depend>rospy</build_export_depend>
68  <build_export_depend>std_msgs</build_export_depend>
69
70  <!-- 运行该程序包中的代码所需的程序包 -->
71  <exec_depend>roscpp</exec_depend>
72  <exec_depend>rospy</exec_depend>
73  <exec_depend>std_msgs</exec_depend>
74
75
76  <!-- The export tag contains other, unspecified,
  tags -->
77  <export>
78      <!-- Other tools can request additional
  information be placed here -->
79
80  </export>
81 </package>
```

2.CMakeLists.txt

文件CMakeLists.txt是CMake构建系统的输入，用于构建软件包。任何兼容CMake的软件包都包含一个或多个CMakeLists.txt文件，这些文件描述了如何构建代码以及将代码安装到何处。

```
1 cmake_minimum_required(VERSION 3.0.2) #所需 cmake
2 #版本
3
4 ## Compile as C++11, supported in ROS Kinetic and
5 ## newer
6
7 ## Find catkin macros and libraries
8 ## if COMPONENTS list like find_package(catkin
9 ## REQUIRED COMPONENTS xyz)
10 ## is used, also find other catkin packages
11 #设置构建所需要的软件包
12 find_package(catkin REQUIRED COMPONENTS
13   roscpp
14   rospy
15   std_msgs
16 )
17 ## System dependencies are found with CMake's
18 ## conventions
19 #默认添加系统依赖
20 # find_package(Boost REQUIRED COMPONENTS system)
21
22 ## Uncomment this if the package has a setup.py.
23 ## This macro ensures
24 ## modules and global scripts declared therein get
25 ## installed
```

```
24 ## See
25   http://ros.org/doc/api/catkin/html/user\_guide/setup\_dot\_py.html
26 #启动 python 模块支持
27 # catkin_python_setup()
28 #####
29 ## Declare ROS messages, services and actions ##
30 ## 声明 ROS 消息、服务、动作... ##
31 #####
32
33 ## To declare and build messages, services or
34 actions from within this
35 ## package, follow these steps:
36 ## * Let MSG_DEP_SET be the set of packages whose
37 message types you use in
38 ## your messages/services/actions (e.g.
39 std_msgs, actionlib_msgs, ...).
40 ## * In the file package.xml:
41 ##   * add a build_depend tag for
42 "message_generation"
43 ##   * add a build_depend and a exec_depend tag
44 for each package in MSG_DEP_SET
45 ##   * If MSG_DEP_SET isn't empty the following
46 dependency has been pulled in
47 ##     but can be declared for certainty
48 nonetheless:
49 ##   * add a exec_depend tag for
50 "message_runtime"
51 ## * In this file (CMakeLists.txt):
52 ##   * add "message_generation" and every package
53 in MSG_DEP_SET to
54 ##     find_package(catkin REQUIRED COMPONENTS
55 ##     ...)
56 ##   * add "message_runtime" and every package in
57 MSG_DEP_SET to
58 ##     catkin_package(CATKIN_DEPENDS ...)
59 ##   * uncomment the add_*_files sections below as
60 needed
```

```
49 ##      and list every .msg/.srv/.action file to be
50 ##      * uncomment the generate_messages entry below
51 ##      * add every package in MSG_DEP_SET to
52 ##          generate_messages(DEPENDENCIES ...)
53
54 ## Generate messages in the 'msg' folder
55 # add_message_files(
56 #   FILES
57 #   Message1.msg
58 #   Message2.msg
59 #
60 ## Generate services in the 'srv' folder
61 # add_service_files(
62 #   FILES
63 #   Service1.srv
64 #   Service2.srv
65 #
66
67 ## Generate actions in the 'action' folder
68 # add_action_files(
69 #   FILES
70 #   Action1.action
71 #   Action2.action
72 #
73
74 ## Generate added messages and services with any
75 ## dependencies listed here
76 # 生成消息、服务时的依赖包
77 # generate_messages(
78 #   DEPENDENCIES
79 #   std_msgs
80 #
81 ######
82 ## Declare ROS dynamic reconfigure parameters ##
83 ## 声明 ROS 动态参数配置 ##
84 ######
```

```

85
86 ## To declare and build dynamic reconfigure
87 ## parameters within this
88 ## package, follow these steps:
89 ## * add a build_depend and a exec_depend tag
90 ## for "dynamic_reconfigure"
91 ## * In this file (CMakeLists.txt):
92 ##     * add "dynamic_reconfigure" to
93 ##         find_package(catkin REQUIRED COMPONENTS
94 ##             ...)
95 ##     * uncomment the
96 ##         "generate_dynamic_reconfigure_options" section
97 ##         below
98 ##     and list every .cfg file to be processed
99
100 ## Generate dynamic reconfigure parameters in the
101 ## 'cfg' folder
102 # generate_dynamic_reconfigure_options(
103 #     cfg/DynReconf1.cfg
104 #     cfg/DynReconf2.cfg
105 # )
106
107 ######
108 ## catkin specific configuration ##
109 ## catkin 特定配置##
110 ## The catkin_package macro generates cmake config
111 ## files for your package
112 ## Declare things to be passed to dependent
113 ## projects
114 ## INCLUDE_DIRS: uncomment this if your package
115 ## contains header files
116 ## LIBRARIES: libraries you create in this project
117 ## that dependent projects also need
118 ## CATKIN_DEPENDS: catkin_packages dependent
119 ## projects also need
120 ## DEPENDS: system dependencies of this project
121 ## that dependent projects also need

```

```
112 # 运行时依赖
113 catkin_package(
114 #   INCLUDE_DIRS include
115 #   LIBRARIES demo01_hello_vscode
116 #   CATKIN_DEPENDS roscpp rospy std_msgs
117 #   DEPENDS system_lib
118 )
119
120 ######
121 ## Build ##
122 #####
123
124 ## Specify additional locations of header files
125 ## Your package locations should be listed before
126 ## other locations
127 # 添加头文件路径, 当前程序包的头文件路径位于其他文件路径之前
128 include_directories(
129 #   include
130 #     ${catkin_INCLUDE_DIRS}
131 )
132
133 ## Declare a C++ library
134 # 声明 C++ 库
135 #   add_library(${PROJECT_NAME}
136 #     src/${PROJECT_NAME}/demo01_hello_vscode.cpp
137 #   )
138
139 ## Add cmake target dependencies of the library
140 ## as an example, code may need to be generated
141 ## before libraries
142 ## either from message generation or dynamic
143 ## reconfigure
144 # 添加库的 cmake 目标依赖
145 #   add_dependencies(${PROJECT_NAME}
146 #     ${${PROJECT_NAME}_EXPORTED_TARGETS}
147 #     ${catkin_EXPORTED_TARGETS})
148
149 ## Declare a C++ executable
```

```

145 ## With catkin_make all packages are built within
146 ## a single CMake context
147 ## The recommended prefix ensures that target
148 ## names across packages don't collide
149 # 声明 C++ 可执行文件
150 add_executable(Hello_VSCode src/Hello_VSCode.cpp)
151
152 ## Rename C++ executable without prefix
153 ## The above recommended prefix causes long target
154 ## names, the following renames the
155 ## target back to the shorter version for ease of
156 ## user use
157 ## e.g. "rosrun someones_pkg node" instead of
158 ## "rosrun someones_pkg someones_pkg_node"
159 #重命名c++可执行文件
160 # set_target_properties(${PROJECT_NAME}_node
161 ## PROPERTIES OUTPUT_NAME node PREFIX "")
162
163 ## Add cmake target dependencies of the executable
164 ## same as for the library above
165 #添加可执行文件的 cmake 目标依赖
166 add_dependencies(Hello_VSCode
167 ${${PROJECT_NAME}_EXPORTED_TARGETS}
168 ${catkin_EXPORTED_TARGETS})
169
170 ## Specify libraries to link a library or
171 ## executable target against
172 #指定库、可执行文件的链接库
173 target_link_libraries(Hello_VSCode
174 ${catkin_LIBRARIES})
175
176 )
177
178 ######
179 ## Install ##
180 ## 安装 ##
181 ######
182
183 # all install targets should use catkin
184 DESTINATION variables

```

```
174 # See
175   http://ros.org/doc/api/catkin/html/adv\_user\_guide/variables.html
176 ## Mark executable scripts (Python etc.) for
177 ## installation
178 ## in contrast to setup.py, you can choose the
179 ## destination
180 #设置用于安装的可执行脚本
181 catkin_install_python(PROGRAMS
182   scripts/Hi.py
183   DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
184 )
185
186 ## Mark executables for installation
187 ## See
188   http://docs.ros.org/melodic/api/catkin/html/howto/format1/building\_executables.html
189 # install(TARGETS ${PROJECT_NAME}_node
190 #   RUNTIME DESTINATION
191 #     ${CATKIN_PACKAGE_BIN_DESTINATION}
192 #   )
193
194 ## Mark libraries for installation
195 ## See
196   http://docs.ros.org/melodic/api/catkin/html/howto/format1/building\_libraries.html
197 # install(TARGETS ${PROJECT_NAME}
198 #   ARCHIVE DESTINATION
199 #     ${CATKIN_PACKAGE_LIB_DESTINATION}
200 #   LIBRARY DESTINATION
201 #     ${CATKIN_PACKAGE_LIB_DESTINATION}
202 #   RUNTIME DESTINATION
203 #     ${CATKIN_GLOBAL_BIN_DESTINATION}
204 #   )
205
206 ## Mark cpp header files for installation
207 # install(DIRECTORY include/${PROJECT_NAME}/
```

```

200 # DESTINATION
      ${CATKIN_PACKAGE_INCLUDE_DESTINATION}
201 # FILES_MATCHING PATTERN "*.h"
202 # PATTERN ".svn" EXCLUDE
203 #
204
205 ## Mark other files for installation (e.g. launch
   and bag files, etc.)
206 # install(FILES
207 #   # myfile1
208 #   # myfile2
209 # DESTINATION
      ${CATKIN_PACKAGE_SHARE_DESTINATION}
210 #
211
212 #####
213 ## Testing ##
214 #####
215
216 ## Add gtest based cpp test target and link
   libraries
217 # catkin_add_gtest(${PROJECT_NAME}-test
   test/test_demo01_hello_vscode.cpp)
218 # if(TARGET ${PROJECT_NAME}-test)
219 #   target_link_libraries(${PROJECT_NAME}-test
   ${PROJECT_NAME})
220 # endif()
221
222 ## Add folders to be run by python nosetests
223 # catkin_add_nosetests(test)

```

1.5.2 ROS文件系统相关命令

ROS 的文件系统本质上都还是操作系统文件，我们可以使用Linux命令来操作这些文件，不过，在ROS中为了更好的用户体验，ROS专门提供了一些类似于Linux的命令，这些命令较之于Linux原生命令，更为简介、高效。文件操作，无外乎就是增删改查与执行等操作，接下来，我们就从这五个维度，来介绍ROS文件系统的一些常用命令。

1. 增

`catkin_create_pkg` 自定义包名 依赖包 === 创建新的ROS功能包

`sudo apt install xxx` === 安装 ROS功能包

2. 删

`sudo apt purge xxx` ==== 删除某个功能包

3. 查

`rospack list` === 列出所有功能包

`rospack find 包名` === 查找某个功能包是否存在，如果存在返回安装路径

`roscd 包名` === 进入某个功能包

`rosfs 包名` === 列出某个包下的文件

`apt search xxx` === 搜索某个功能包

4. 改

`rosed 包名 文件名` === 修改功能包文件

需要安装 vim

比如: `rosed turtlesim Color.msg`

5. 执行

5.1 `roscore`

`roscore` === 是 ROS 的系统先决条件节点和程序的集合，必须运行 `roscore` 才能使 ROS 节点进行通信。

roscore 将启动:

- `ros master`

- ros 参数服务器
- rosout 日志节点

用法:

```
1 roscore
```

或(指定端口号)

```
1 roscore -p xxxx
```

5.2rosrun

rosrun 包名 可执行文件名 === 运行指定的ROS节点

比如: `rosrun turtlesim turtlesim_node`

5.3roslaunch

roslaunch 包名 launch文件名 === 执行某个包下的 launch 文件

1.5.3 ROS计算图

1.计算图简介

前面介绍的是ROS文件结构，是磁盘上 ROS 程序的存储结构，是静态的，而 ros 程序运行之后，不同的节点之间是错综复杂的，ROS 中提供了一个实用的工具:rqt_graph。

rqt_graph能够创建一个显示当前系统运行情况的动态图形。ROS 分布式系统中不同进程需要进行数据交互，计算图可以以点对点的网络形式表现数据交互过程。rqt_graph是rqt程序包中的一部分。

2.计算图安装

如果前期把所有的功能包 (package) 都已经安装完成，则直接在终端窗口中输入

```
rosrun rqt_graph rqt_graph
```

如果未安装则在终端 (terminal) 中输入

```
1 $ sudo apt install ros-<distro>-rqt
2 $ sudo apt install ros-<distro>-rqt-common-plugins
```

请使用你的ROS版本名称 (比如:kinetic、melodic、Noetic等) 来替换掉。

例如当前版本是 Noetic,就在终端窗口中输入

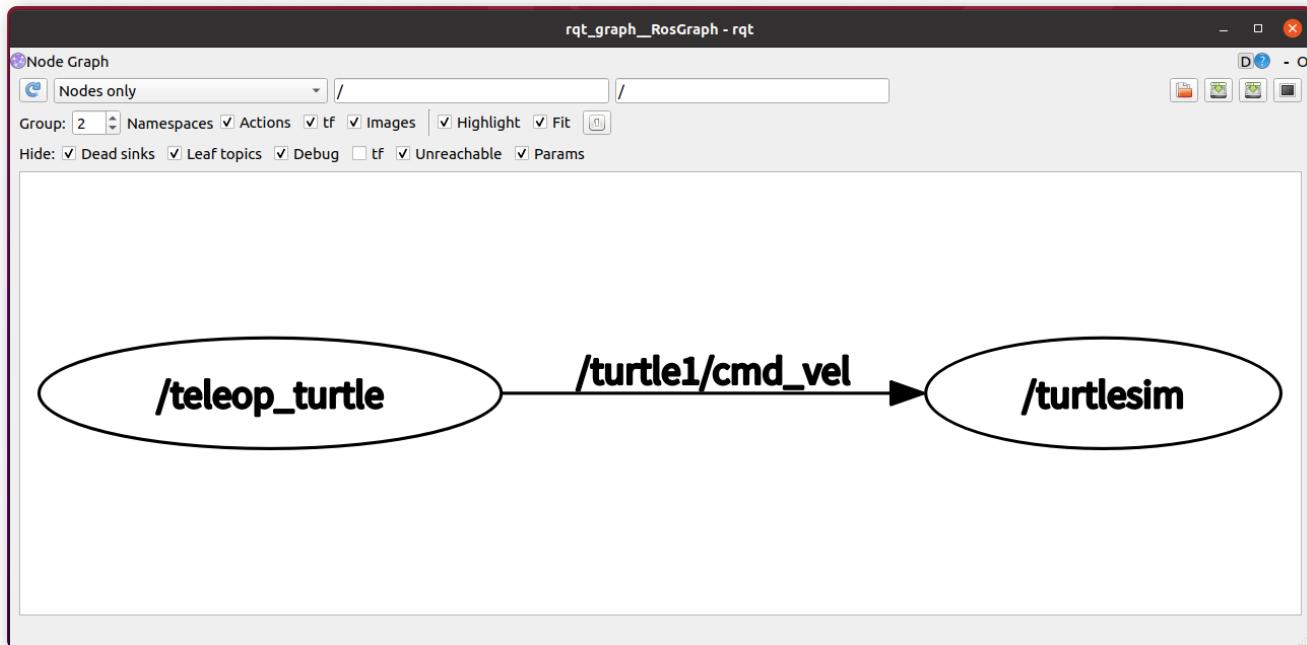
```
1 $ sudo apt install ros-noetic-rqt
2 $ sudo apt install ros-noetic-rqt-common-plugins
```

3.计算图演示

接下来以 ROS 内置的小乌龟案例来演示计算图

首先，按照前面所示，运行案例

然后，启动新终端，键入: rqt_graph 或 rosrun rqt_graph rqt_graph，可以看到类似下图的网络拓扑图，该图可以显示不同节点之间的关系。



1.6 本章小结

本章内容主要介绍了ROS的相关概念、设计目标、发展历程等理论知识，安装了ROS并搭建了ROS的集成开发环境，编写了第一个ROS小程序，对ROS实现架构也有了宏观的认识。ROS的大门已经敞开，接下来就要步入新的征程了。

第2章 ROS通信机制

机器人是一种高度复杂的系统性实现，在机器人上可能集成各种传感器(雷达、摄像头、GPS...)以及运动控制实现，为了解耦合，在ROS中每一个功能点都是一个单独的进程，每一个进程都是独立运行的。更确切的讲，**ROS是进程(也称为*Nodes*)的分布式框架**。因为这些进程甚至还可分布于不同主机，不同主机协同工作，从而分散计算压力。不过随之也有一个问题：不同的进程是如何通信的？也即不同进程间如何实现数据交换的？在此我们就需要介绍一下ROS中的通信机制了。

ROS 中的基本通信机制主要有如下三种实现策略：

- 话题通信(发布订阅模式)
- 服务通信(请求响应模式)
- 参数服务器(参数共享模式)

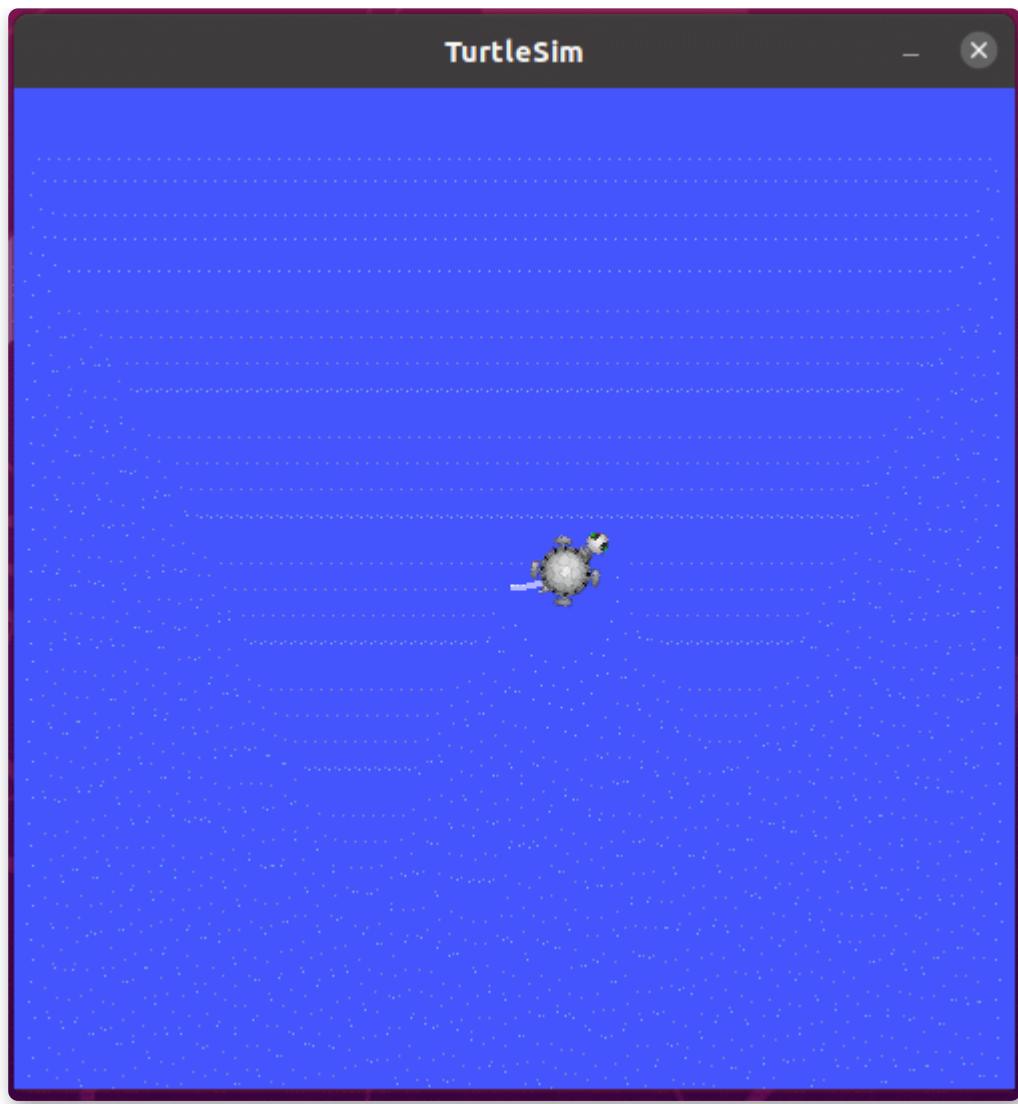
本章的主要内容就是是介绍各个通信机制的应用场景、理论模型、代码实现以及相关操作命令。本章预期达成学习目标如下：

- 能够熟练介绍ROS中常用的通信机制
- 能够理解ROS中每种通信机制的理论模型
- 能够以代码的方式实现各种通信机制对应的案例
- 能够熟练使用ROS中的一些操作命令
- 能够独立完成相关实操案例

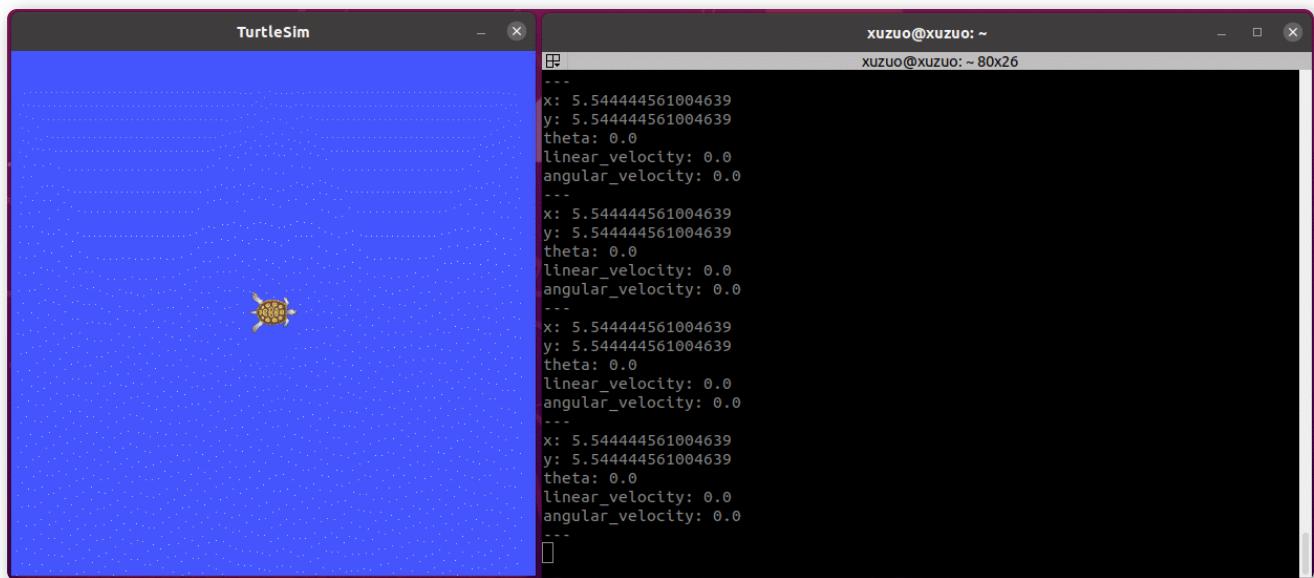
案例演示：

1. 话题演示案例：

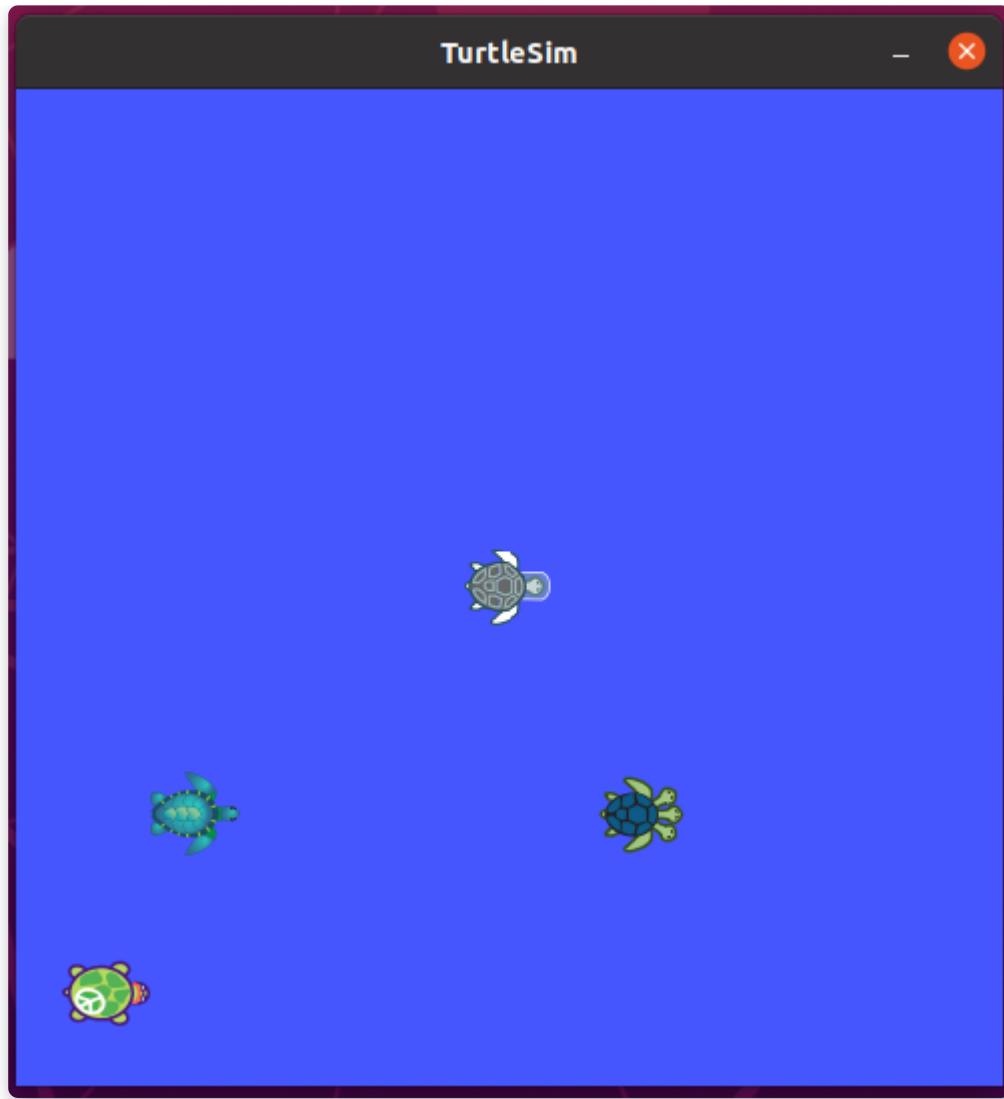
控制小乌龟做圆周运动



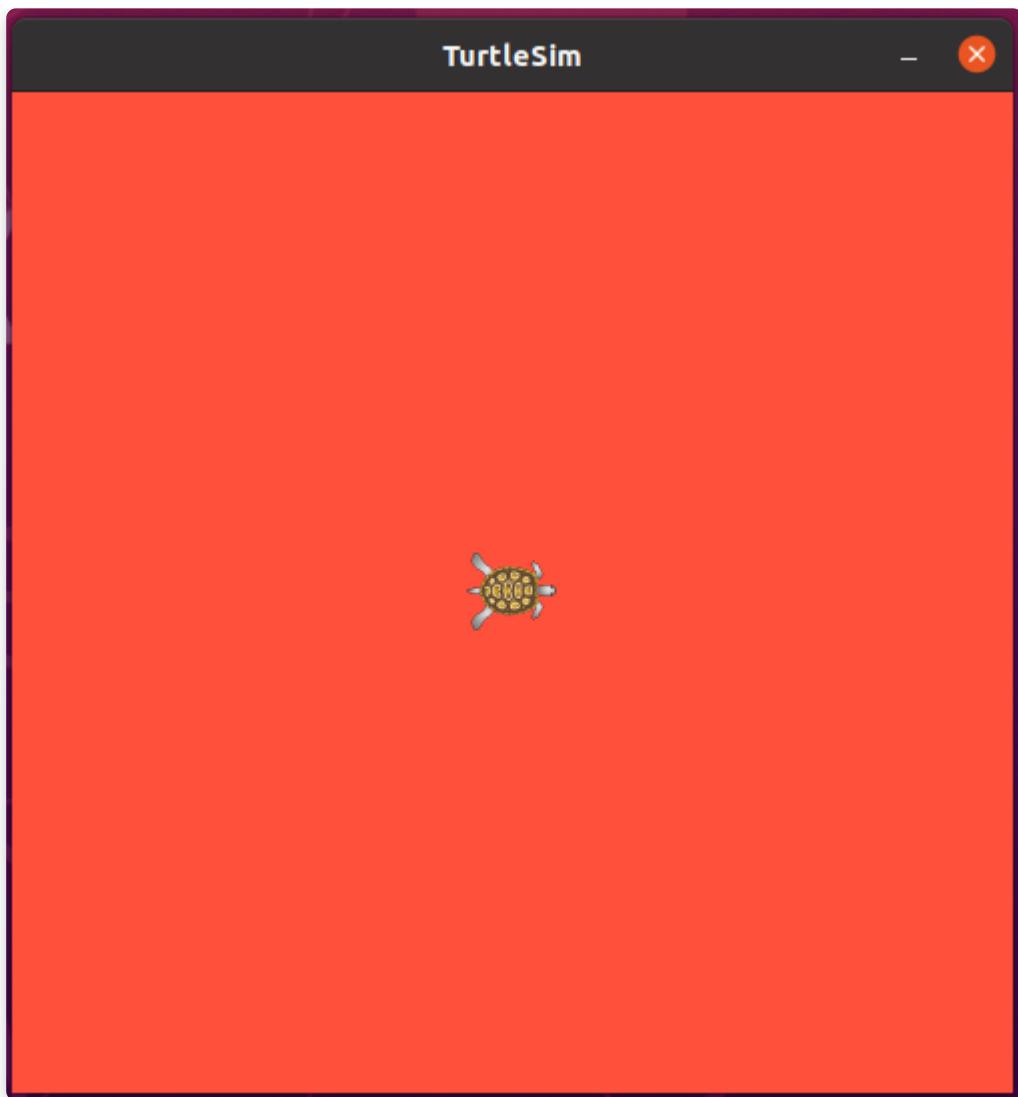
获取乌龟位姿



2.服务演示案例:在指定位置生成乌龟



3.参数演示案例:改变乌龟窗口的背景颜色



2.1 话题通信

话题通信是ROS中使用频率最高的一种通信模式，话题通信是基于发布订阅模式的，也即:一个节点发布消息，另一个节点订阅该消息。话题通信的应用场景也极其广泛，比如下面一个常见场景：



机器人在执行导航功能，使用的传感器是激光雷达，机器人会采集激光雷达感知到的信息并计算，然后生成运动控制信息驱动机器人底盘运动。

在上述场景中，就不止一次使用到了话题通信。

- 以激光雷达信息的采集处理为例，在ROS中有一个节点需要时时的发布当前雷达采集到的数据，导航模块中也有节点会订阅并解析雷达数据。

- 再以运动消息的发布为例，导航模块会根据传感器采集的数据时时的计算出运动控制信息并发布给底盘，底盘也可以有一个节点订阅运动信息并最终转换成控制电机的脉冲信号。

以此类推，像雷达、摄像头、GPS.... 等等一些传感器数据的采集，也都是使用了话题通信，换言之，话题通信适用于不断更新的数据传输相关的应用场景。

概念

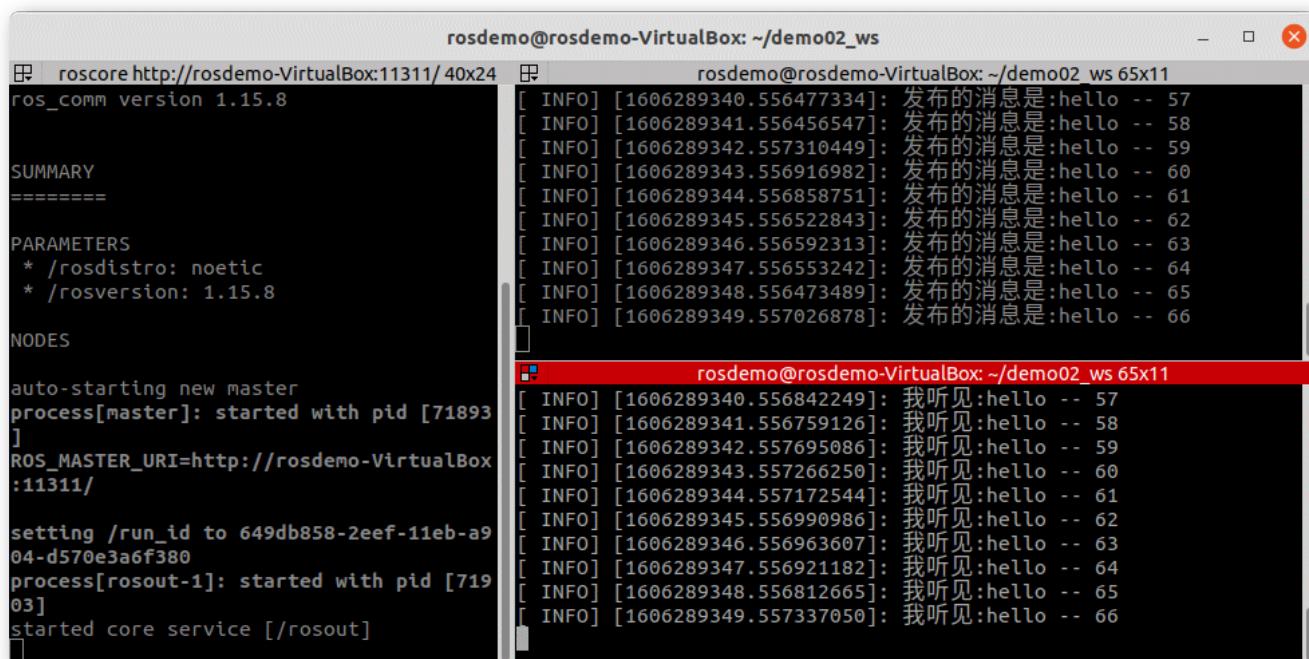
以发布订阅的方式实现不同节点之间数据交互的通信模式。

作用

用于不断更新的、少逻辑处理的数据传输场景。

案例

- 实现最基本的发布订阅模型，发布方以固定频率发送一段文本，订阅方接收文本并输出。(2.1.2 -- 2.1.3)

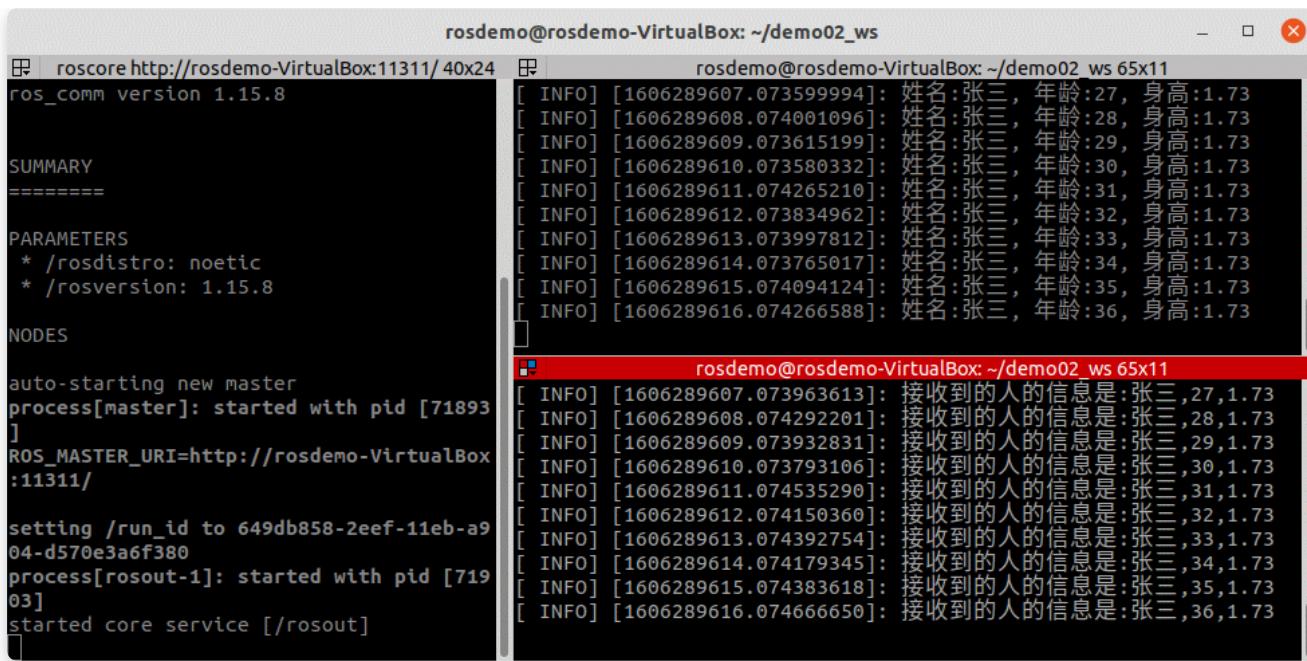


```
roscore http://rosdemo-VirtualBox:11311/ 40x24
ros_comm version 1.15.8
SUMMARY
=====
PARAMETERS
  * /rosdistro: noetic
  * /rosversion: 1.15.8
NODES
auto-starting new master
process[master]: started with pid [71893]
ROS_MASTER_URI=http://rosdemo-VirtualBox:11311/
setting /run_id to 649db858-2eef-11eb-a904-d570e3a6f380
process[rosout-1]: started with pid [71903]
started core service [/rosout]
```

```
[ INFO] [1606289340.556477334]: 发布的消息是:hello -- 57
[ INFO] [1606289341.556456547]: 发布的消息是:hello -- 58
[ INFO] [1606289342.557310449]: 发布的消息是:hello -- 59
[ INFO] [1606289343.556916982]: 发布的消息是:hello -- 60
[ INFO] [1606289344.556858751]: 发布的消息是:hello -- 61
[ INFO] [1606289345.556522843]: 发布的消息是:hello -- 62
[ INFO] [1606289346.556592313]: 发布的消息是:hello -- 63
[ INFO] [1606289347.556553242]: 发布的消息是:hello -- 64
[ INFO] [1606289348.556473489]: 发布的消息是:hello -- 65
[ INFO] [1606289349.557026878]: 发布的消息是:hello -- 66
```

```
[ INFO] [1606289340.556842249]: 我听见:hello -- 57
[ INFO] [1606289341.556759126]: 我听见:hello -- 58
[ INFO] [1606289342.557695086]: 我听见:hello -- 59
[ INFO] [1606289343.557266250]: 我听见:hello -- 60
[ INFO] [1606289344.557172544]: 我听见:hello -- 61
[ INFO] [1606289345.556990986]: 我听见:hello -- 62
[ INFO] [1606289346.556963607]: 我听见:hello -- 63
[ INFO] [1606289347.556921182]: 我听见:hello -- 64
[ INFO] [1606289348.556812665]: 我听见:hello -- 65
[ INFO] [1606289349.557337050]: 我听见:hello -- 66
```

- 实现对自定义消息的发布与订阅。(2.1.4 -- 2.1.6)



The terminal window displays two nodes: 'roscore' and 'rosdemo@rosdemo-VirtualBox: ~/demo02_ws'. The 'roscore' node shows system information and node details. The 'rosdemo' node shows a series of INFO log messages indicating the reception of data from a publisher, with each message containing a name, age, and height.

```

roscore http://rosdemo-VirtualBox:11311/ 40x24
ros_comm version 1.15.8

SUMMARY
=====
PARAMETERS
  * /rosdistro: noetic
  * /rosversion: 1.15.8

NODES
auto-starting new master
process[master]: started with pid [71893]
ROS_MASTER_URI=http://rosdemo-VirtualBox:11311/
setting /run_id to 649db858-2eef-11eb-a904-d570e3a6f380
process[rosout-1]: started with pid [71903]
started core service [/rosout]

[ INFO] [1606289607.073599994]: 姓名:张三, 年龄:27, 身高:1.73
[ INFO] [1606289608.074001096]: 姓名:张三, 年龄:28, 身高:1.73
[ INFO] [1606289609.073615199]: 姓名:张三, 年龄:29, 身高:1.73
[ INFO] [1606289610.073580332]: 姓名:张三, 年龄:30, 身高:1.73
[ INFO] [1606289611.074265210]: 姓名:张三, 年龄:31, 身高:1.73
[ INFO] [1606289612.073834962]: 姓名:张三, 年龄:32, 身高:1.73
[ INFO] [1606289613.073997812]: 姓名:张三, 年龄:33, 身高:1.73
[ INFO] [1606289614.073765017]: 姓名:张三, 年龄:34, 身高:1.73
[ INFO] [1606289615.074094124]: 姓名:张三, 年龄:35, 身高:1.73
[ INFO] [1606289616.074266588]: 姓名:张三, 年龄:36, 身高:1.73

```

另请参考:

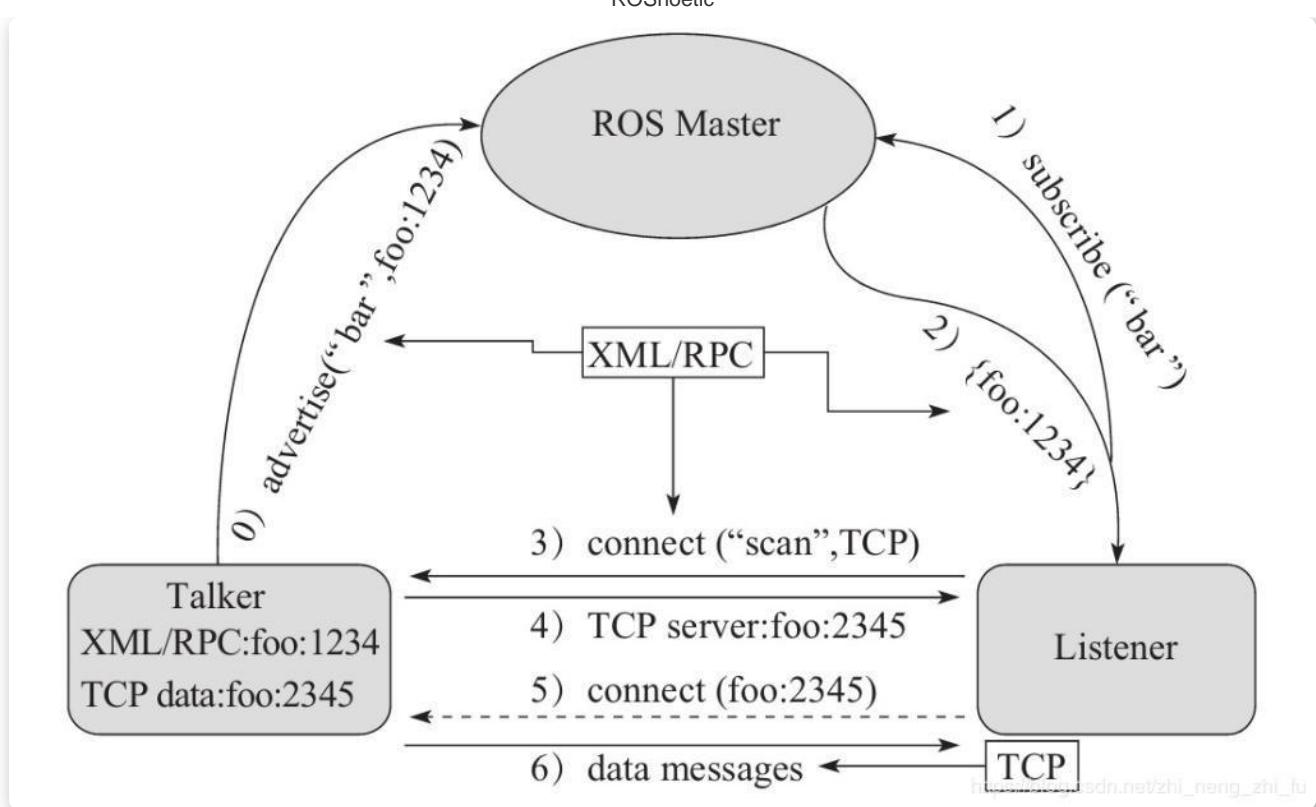
- <http://wiki.ros.org/ROS/Tutorials/CreatingMsgAndSrv>
- <http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28c%2B%2B%29>
- <http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29>

2.1.1 理论模型

话题通信实现模型是比较复杂的，该模型如下图所示,该模型中涉及到三个角色:

- ROS Master (管理者)
- Talker (发布者)
- Listener (订阅者)

ROS Master 负责保管 Talker 和 Listener 注册的信息，并匹配话题相同的 Talker 与 Listener，帮助 Talker 与 Listener 建立连接，连接建立后，Talker 可以发布消息，且发布的消息会被 Listener 订阅。



整个流程由以下步骤实现:

0.Talker注册

Talker启动后，会通过RPC在 ROS Master 中注册自身信息，其中包含所发布消息的话题名称。ROS Master 会将节点的注册信息加入到注册表中。

1.Listener注册

Listener启动后，也会通过RPC在 ROS Master 中注册自身信息，包含需要订阅消息的话题名。ROS Master 会将节点的注册信息加入到注册表中。

2.ROS Master实现信息匹配

ROS Master 会根据注册表中的信息匹配Talker 和 Listener，并通过 RPC 向 Listener 发送 Talker 的 RPC 地址信息。

3.Listener向Talker发送请求

Listener 根据接收到的 RPC 地址，通过 RPC 向 Talker 发送连接请求，传输订阅的话题名称、消息类型以及通信协议(TCP/UDP)。

4.Talker确认请求

Talker 接收到 Listener 的请求后，也是通过 RPC 向 Listener 确认连接信息，并发送自身的 TCP 地址信息。

5.Listener与Talker件里连接

Listener 根据步骤4 返回的消息使用 TCP 与 Talker 建立网络连接。

6.Talker向Listener发送消息

连接建立后，Talker 开始向 Listener 发布消息。



注意1:上述实现流程中，前五步使用的 RPC 协议，最后两步使用的是 TCP 协议

注意2: Talker 与 Listener 的启动无先后顺序要求

注意3: Talker 与 Listener 都可以有多个

注意4: Talker 与 Listener 连接建立后，不再需要 ROS Master。也即，即便关闭ROS Master，Talker 与 Listener 照常通信。

2.1.2 话题通信基本操作A(C++)

需求:



编写发布订阅实现，要求发布方以10HZ(每秒10次)的频率发布文本消息，订阅方订阅消息并将消息内容打印输出。

分析:

在模型实现中，ROS master 不需要实现，而连接的建立也已经被封装了，需要关注的关键点有三个：

1. 发布方
2. 接收方
3. 数据(此处为普通文本)

流程:

1. 编写发布方实现;
2. 编写订阅方实现;
3. 编辑配置文件;
4. 编译并执行。

1.发布方

```

1  /*
2      需求: 实现基本的话题通信, 一方发布数据, 一方接收数据,
3          实现的关键点:
4          1.发送方
5          2.接收方
6          3.数据(此处为普通文本)
7
8      PS: 二者需要设置相同的话题
9
10
11     消息发布方:
12         循环发布信息:HelloWorld 后缀数字编号
13
14     实现流程:
15         1.包含头文件
16         2.初始化 ROS 节点:命名(唯一)
17         3.实例化 ROS 句柄
18         4.实例化 发布者 对象
19         5.组织被发布的数据, 并编写逻辑发布数据
20
21 */
22 // 1.包含头文件
23 #include "ros/ros.h"
24 #include "std_msgs/String.h" //普通文本类型的消息
25 #include <iostream>
26

```

```

27 int main(int argc, char *argv[])
28 {
29     //设置编码
30     setlocale(LC_ALL, "");
31
32     //2. 初始化 ROS 节点:命名(唯一)
33     // 参数1和参数2 后期为节点传值会使用
34     // 参数3 是节点名称, 是一个标识符, 需要保证运行后, 在
35     // ROS 网络拓扑中唯一
36     ros::init(argc, argv, "talker");
37     //3. 实例化 ROS 句柄
38     ros::NodeHandle nh; //该类封装了 ROS 中的一些常用功
39     能
40
41     //4. 实例化 发布者 对象
42     //泛型: 发布的消息类型
43     //参数1: 要发布到的话题
44     //参数2: 队列中最大保存的消息数, 超出此阈值时, 先进的先
45     //销毁(时间早的先销毁)
46     ros::Publisher pub =
47     nh.advertise<std_msgs::String>("chatter", 10);
48
49     //5. 组织被发布的数据, 并编写逻辑发布数据
50     //数据(动态组织)
51     std_msgs::String msg;
52     // msg.data = "你好啊！！！！";
53     std::string msg_front = "Hello 你好! "; //消息前
54     缀
55     int count = 0; //消息计数器
56
57     //逻辑(一秒10次)
58     ros::Rate r(1);
59
60     //节点不死
61     while (ros::ok())
62     {
63         //使用 stringstream 拼接字符串与编号
64         std::stringstream ss;
65         ss << msg_front << count;

```

```

61     msg.data = ss.str();
62     //发布消息
63     pub.publish(msg);
64     //加入调试，打印发送的消息
65     ROS_INFO("发送的消息:%s",msg.data.c_str());
66
67     //根据前面制定的发送频率自动休眠 休眠时间 = 1/
68     //频率:
69     r.sleep();
70     count++; //循环结束前，让 count 自增
71     //暂无应用
72     ros::spinOnce();
73
74
75     return 0;
76 }
```

2.订阅方

```

1  /*
2  需求： 实现基本的话题通信，一方发布数据，一方接收数据，  

3  实现的关键点：  

4  1.发送方  

5  2.接收方  

6  3.数据(此处为普通文本)  

7
8
9  消息订阅方：  

10  订阅话题并打印接收到的消息  

11
12  实现流程：  

13  1.包含头文件  

14  2.初始化 ROS 节点：命名(唯一)  

15  3.实例化 ROS 句柄  

16  4.实例化 订阅者 对象  

17  5.处理订阅的消息(回调函数)  

18  6.设置循环调用回调函数
```

```

19
20 */
21 // 1. 包含头文件
22 #include "ros/ros.h"
23 #include "std_msgs/String.h"
24
25 void doMsg(const std_msgs::String::ConstPtr&
26             msg_p){
27     ROS_INFO("我听见:%s",msg_p->data.c_str());
28     // ROS_INFO("我听见:%s",(*msg_p).data.c_str());
29 }
30
31 int main(int argc, char *argv[])
32 {
33     setlocale(LC_ALL,"");
34     //2. 初始化 ROS 节点:命名(唯一)
35     ros::init(argc,argv,"listener");
36     //3. 实例化 ROS 句柄
37     ros::NodeHandle nh;
38
39     //4. 实例化 订阅者 对象
40     ros::Subscriber sub =
41     nh.subscribe<std_msgs::String>
42     ("chatter",10,doMsg);
43     //5. 处理订阅的消息(回调函数)
44
45     //      6. 设置循环调用回调函数
46     ros::spin(); //循环读取接收的数据, 并调用回调函数处理
47
48     return 0;
49 }

```

3. 配置 CMakeLists.txt

```

1 add_executable(Hello_pub
2   src/Hello_pub.cpp
3 )
4 add_executable(Hello_sub
5   src/Hello_sub.cpp
6 )
7
8 target_link_libraries(Hello_pub
9   ${catkin_LIBRARIES}
10 )
11 target_link_libraries(Hello_sub
12   ${catkin_LIBRARIES}
13 )

```

4.执行

1.启动 roscore;

2.启动发布节点;

3.启动订阅节点。

运行结果与引言部分的演示案例1类似。

5.注意

补充0:

vscode 中的 main 函数 声明 int main(int argc, char const *argv[]){},
默认生成 argv 被 const 修饰，需要去除该修饰符

补充1:

ros/ros.h No such file or directory

检查 CMakeList.txt find_package 出现重复,删除内容少的即可

参考资料:<https://answers.ros.org/question/237494/fatal-error-rosrosesh-no-such-file-or-directory/>

补充2:

find_package 不添加一些包，也可以运行啊， ros.wiki 答案如下

1 You may notice that sometimes your project builds fine even if you did not call find_package with all dependencies. This is because catkin combines all your projects into one, so if an earlier project calls find_package, yours is configured with the same values. But forgetting the call means your project can easily break when built in isolation.

补充3:

订阅时，第一条数据丢失

原因: 发送第一条数据时， publisher 还未在 roscore 注册完毕

解决: 注册后，加入休眠 `ros::Duration(3.0).sleep();` 延迟第一条数据的发送

PS: 可以使用 `rqt_graph` 查看节点关系。

2.1.3 话题通信基本操作B(Python)

需求:



编写发布订阅实现，要求发布方以10HZ(每秒10次)的频率发布文本消息，订阅方订阅消息并将消息内容打印输出。

分析:

在模型实现中，ROS master 不需要实现，而连接的建立也已经被封装了，需要关注的关键点有三个：

1. 发布方
2. 接收方
3. 数据(此处为普通文本)

流程:

1. 编写发布方实现；
2. 编写订阅方实现；
3. 为python文件添加可执行权限；
4. 编辑配置文件；
5. 编译并执行。

1.发布方

```

1  #! /usr/bin/env python
2  """
3      需求： 实现基本的话题通信，一方发布数据，一方接收数据，  

4          实现的关键点：  

5              1.发送方  

6              2.接收方  

7              3.数据(此处为普通文本)  

8
9      PS： 二者需要设置相同的话题
10
11
12      消息发布方：  

13          循环发布信息：HelloWorld 后缀数字编号
14
15      实现流程：  

16          1.导包  

17          2.初始化 ROS 节点：命名(唯一)  

18          3.实例化 发布者 对象  

19          4.组织被发布的数据，并编写逻辑发布数据
20
21
22  """
23  #1. 导包
24  import rospy
25  from std_msgs.msg import String

```

```

26
27 if __name__ == "__main__":
28     #2. 初始化 ROS 节点:命名(唯一)
29     rospy.init_node("talker_p")
30     #3. 实例化 发布者 对象
31     pub =
32         rospy.Publisher("chatter",String,queue_size=10)
33     #4. 组织被发布的数据, 并编写逻辑发布数据
34     msg = String() #创建 msg 对象
35     msg_front = "hello 你好"
36     count = 0 #计数器
37     # 设置循环频率
38     rate = rospy.Rate(1)
39     while not rospy.is_shutdown():
40
41         #拼接字符串
42         msg.data = msg_front + str(count)
43
44         pub.publish(msg)
45         rate.sleep()
46         rospy.loginfo("写出的数据:%s",msg.data)
47         count += 1

```

2.订阅方

```

1 #! /usr/bin/env python
2 """
3     需求: 实现基本的话题通信, 一方发布数据, 一方接收数据,
4     实现的关键点:
5         1.发送方
6         2.接收方
7         3.数据(此处为普通文本)
8
9
10    消息订阅方:
11        订阅话题并打印接收到的消息
12
13    实现流程:

```

```

14     1. 导包
15     2. 初始化 ROS 节点:命名(唯一)
16     3. 实例化 订阅者 对象
17     4. 处理订阅的消息(回调函数)
18     5. 设置循环调用回调函数
19
20
21
22 """
23 #1. 导包
24 import rospy
25 from std_msgs.msg import String
26
27 def doMsg(msg):
28     rospy.loginfo("I heard:%s",msg.data)
29
30 if __name__ == "__main__":
31     #2. 初始化 ROS 节点:命名(唯一)
32     rospy.init_node("listener_p")
33     #3. 实例化 订阅者 对象
34     sub =
35         rospy.Subscriber("chatter",String,doMsg,queue_size
36         =10)
37         #4. 处理订阅的消息(回调函数)
38         #5. 设置循环调用回调函数
39         rospy.spin()

```

3.添加可执行权限

终端下进入 scripts 执行: chmod +x *.py

4.配置 CMakeLists.txt

```

1 catkin_install_python(PROGRAMS
2     scripts/talker_p.py
3     scripts/listener_p.py
4     DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
5 )

```

5. 执行

1. 启动 roscore;

2. 启动发布节点;

3. 启动订阅节点。

运行结果与引言部分的演示案例1类似。

PS：可以使用 rqt_graph 查看节点关系。

2.1.4 话题通信自定义msg

在 ROS 通信协议中，数据载体是一个较为重要组成部分，ROS 中通过 std_msgs 封装了一些原生的数据类型，比如：String、Int32、Int64、Char、Bool、Empty.... 但是，这些数据一般只包含一个 data 字段，结构的单一意味着功能上的局限性，当传输一些复杂的数据，比如：激光雷达的信息... std_msgs 由于描述性较差而显得力不从心，这种场景下可以使用自定义的消息类型

msgs只是简单的文本文件，每行具有字段类型和字段名称，可以使用的字段类型有：

- int8, int16, int32, int64 (或者无符号类型: uint*)
- float32, float64
- string
- time, duration
- other msg files
- variable-length array[] and fixed-length array[C]

ROS中还有一种特殊类型： Header，标头包含时间戳和ROS中常用的坐标帧信息。会经常看到msg文件的第一行具有 Header 标头。

需求：创建自定义消息，该消息包含人的信息：姓名、身高、年龄等。

流程:

1. 按照固定格式创建 msg 文件
2. 编辑配置文件
3. 编译生成可以被 Python 或 C++ 调用的中间文件

1. 定义msg文件

功能包下新建 msg 目录，添加文件 Person.msg

```

1 string name
2 uint16 age
3 float64 height

```

2. 编辑配置文件

package.xml 中添加编译依赖与执行依赖

```

1 <build_depend>message_generation</build_depend>
2 <exec_depend>message_runtime</exec_depend>
3 <!--
4 exce_depend 以前对应的是 run_depend 现在非法
5 -->

```

CMakeLists.txt 编辑 msg 相关配置

```

1 find_package(catkin REQUIRED COMPONENTS
2   roscpp
3   rospy
4   std_msgs
5   message_generation
6 )
7 # 需要加入 message_generation, 必须有 std_msgs
8 ## 配置 msg 源文件
9 add_message_files(
10   FILES
11   Person.msg
12 )

```

```

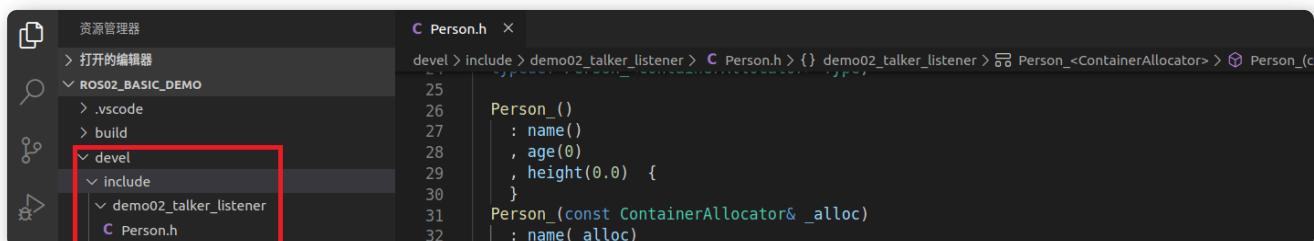
13 # 生成消息时依赖于 std_msgs
14 generate_messages(
15     DEPENDENCIES
16     std_msgs
17 )
18 #执行时依赖
19 catkin_package(
20     INCLUDE_DIRS include
21     LIBRARIES demo02_talker_listener
22     CATKIN_DEPENDS roscpp rospy std_msgs
23     message_runtime
24     DEPENDS system_lib
25 )

```

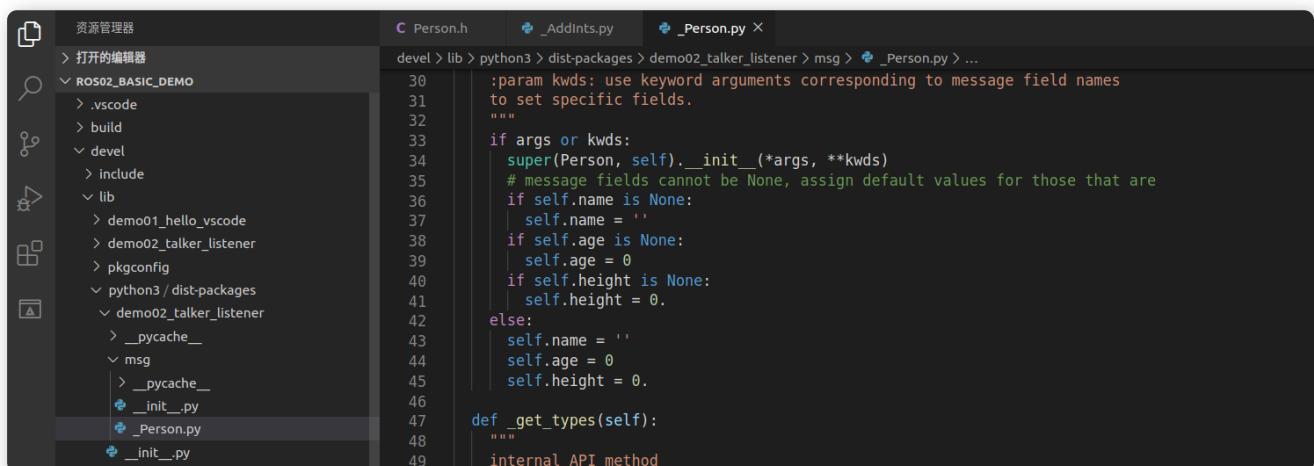
3. 编译

编译后的中间文件查看：

C++ 需要调用的中间文件(.../工作空间/devel/include/包名/xxx.h)



Python 需要调用的中间文件(.../工作空间/devel/lib/python3/dist-packages/包名/msg)



后续调用相关 msg 时，是从这些中间文件调用的

2.1.5 话题通信自定义msg调用A(C++)

需求:



编写发布订阅实现，要求发布方以10HZ(每秒10次)的频率发布自定义消息，订阅方订阅自定义消息并将消息内容打印输出。

分析:

在模型实现中，ROS master 不需要实现，而连接的建立也已经被封装了，需要关注的关键点有三个：

1. 发布方
2. 接收方
3. 数据(此处为自定义消息)

流程:

1. 编写发布方实现；
2. 编写订阅方实现；
3. 编辑配置文件；
4. 编译并执行。

0.vscode 配置

为了方便代码提示以及避免误抛异常，需要先配置 vscode，将前面生成的 head 文件路径配置进 c_cpp_properties.json 的 includepath 属性：

```

1  {
2      "configurations": [
3          {
4              "browse": {
5                  "databaseFilename": "",
6                  "limitSymbolsToIncludedHeaders": true
7              },
8              "includePath": [

```

```

9           "/opt/ros/noetic/include/**",
10          "/usr/include/**",
11          "/xxx/yyy工作空间/devel/include/**"
12      ],
13      "name": "ROS",
14      "intelliSenseMode": "gcc-x64",
15      "compilerPath": "/usr/bin/gcc",
16      "cStandard": "c11",
17      "cppStandard": "c++17"
18  }
19 ],
20 "version": 4
21 }

```

1.发布方

```

1  /*
2   * 需求：循环发布人的信息
3
4  */
5
6 #include "ros/ros.h"
7 #include "demo02_talker_listener/Person.h"
8
9 int main(int argc, char *argv[])
10 {
11     setlocale(LC_ALL, "");
12
13     //1.初始化 ROS 节点
14     ros::init(argc, argv, "talker_person");
15
16     //2.创建 ROS 句柄
17     ros::NodeHandle nh;
18
19     //3.创建发布者对象

```

```

20     ros::Publisher pub =
21         nh.advertise<demo02_talker_listener::Person>
22             ("chatter_person", 1000);
23
24     //4.组织被发布的消息，编写发布逻辑并发布消息
25     demo02_talker_listener::Person p;
26     p.name = "sunwukong";
27     p.age = 2000;
28     p.height = 1.45;
29
30     ros::Rate r(1);
31     while (ros::ok())
32     {
33         pub.publish(p);
34         p.age += 1;
35         ROS_INFO("我叫:%s,今年%d岁,高%.2f米",
36             p.name.c_str(), p.age, p.height);
37         r.sleep();
38         ros::spinOnce();
39     }
40
41     return 0;
42 }
```

2.订阅方

```

1  /*
2   * 需求：订阅人的信息
3
4  */
5
6 #include "ros/ros.h"
7 #include "demo02_talker_listener/Person.h"
8
```

```

9 void doPerson(const
  demo02_talker_listener::Person::ConstPtr&
  person_p){
10   ROS_INFO("订阅的人信息:%s, %d, %.2f", person_p-
  >name.c_str(), person_p->age, person_p->height);
11 }
12
13 int main(int argc, char *argv[])
14 {
15   setlocale(LC_ALL, "");
16
17   //1. 初始化 ROS 节点
18   ros::init(argc, argv, "listener_person");
19   //2. 创建 ROS 句柄
20   ros::NodeHandle nh;
21   //3. 创建订阅对象
22   ros::Subscriber sub =
23     nh.subscribe<demo02_talker_listener::Person>
24     ("chatter_person", 10, doPerson);
25
26   //4. 回调函数中处理 person
27
28   //5. ros::spin();
29   ros::spin();
30   return 0;
31 }
```

3. 配置 CMakeLists.txt

需要添加 `add_dependencies` 用以设置所依赖的消息相关的中间文件。

```

1 add_executable(person_talker
  src/person_talker.cpp)
2 add_executable(person_listener
  src/person_listener.cpp)
3
4
5
```

```

6 add_dependencies(person_talker
                  ${PROJECT_NAME}_generate_messages_cpp)
7 add_dependencies(person_listener
                  ${PROJECT_NAME}_generate_messages_cpp)
8
9
10 target_link_libraries(person_talker
11   ${catkin_LIBRARIES}
12 )
13 target_link_libraries(person_listener
14   ${catkin_LIBRARIES}
15 )

```

4.执行

1.启动 roscore;

2.启动发布节点;

3.启动订阅节点。

运行结果与引言部分的演示案例2类似。

PS：可以使用 rqt_graph 查看节点关系。

2.1.6 话题通信自定义msg调用B(Python)

需求:



编写发布订阅实现，要求发布方以1HZ(每秒1次)的频率发布自定义消息，订阅方订阅自定义消息并将消息内容打印输出。

分析:

在模型实现中，ROS master 不需要实现，而连接的建立也已经被封装了，需要关注的关键点有三个：

1. 发布方
2. 接收方
3. 数据(此处为自定义消息)

流程：

1. 编写发布方实现；
2. 编写订阅方实现；
3. 为python文件添加可执行权限；
4. 编辑配置文件；
5. 编译并执行。

0.vscode配置

为了方便代码提示以及误抛异常，需要先配置 vscode，将前面生成的 python 文件路径配置进 settings.json

```

1  {
2      "python.autoComplete.extraPaths": [
3          "/opt/ros/noetic/lib/python3/dist-packages",
4          "/xxx/yyy工作空间/devel/lib/python3/dist-
5          packages"
6      ]
7  }

```

1.发布方

```

1  #! /usr/bin/env python
2  """
3      发布方：
4          循环发送消息
5
6  """
7  import rospy
8  from demo02_talker_listener.msg import Person

```

```

9
10
11 if __name__ == "__main__":
12     #1. 初始化 ROS 节点
13     rospy.init_node("talker_person_p")
14     #2. 创建发布者对象
15     pub =
16         rospy.Publisher("chatter_person",Person,queue_size
17 =10)
18         #3. 组织消息
19         p = Person()
20         p.name = "葫芦瓦"
21         p.age = 18
22         p.height = 0.75
23
24         #4. 编写消息发布逻辑
25         rate = rospy.Rate(1)
26         while not rospy.is_shutdown():
27             pub.publish(p) #发布消息
28             rate.sleep() #休眠
29             rospy.loginfo("姓名:%s, 年龄:%d, 身
30 高:%.2f",p.name, p.age, p.height)

```

2.订阅方

```

1  #! /usr/bin/env python
2  """
3      订阅方：
4          订阅消息
5
6  """
7  import rospy
8  from demo02_talker_listener.msg import Person
9
10 def doPerson(p):
11     rospy.loginfo("接收到人的信息:%s, %d,
12 %.2f",p.name, p.age, p.height)

```

```

13
14 if __name__ == "__main__":
15     #1. 初始化节点
16     rospy.init_node("listener_person_p")
17     #2. 创建订阅者对象
18     sub =
19         rospy.Subscriber("chatter_person",Person,doPerson,
queue_size=10)
20         rospy.spin() #4. 循环

```

3.权限设置

终端下进入 scripts 执行: chmod +x *.py

4.配置 CMakeLists.txt

```

1 catkin_install_python(PROGRAMS
2     scripts/talker_p.py
3     scripts/listener_p.py
4     scripts/person_talker.py
5     scripts/person_listener.py
6     DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
7 )

```

5.执行

1.启动 roscore;

2.启动发布节点;

3.启动订阅节点。

运行结果与引言部分的演示案例2类似。

PS: 可以使用 rqt_graph 查看节点关系。

2.2 服务通信

服务通信也是ROS中一种极其常用的通信模式，服务通信是基于请求响应模式的，是一种应答机制。也即：一个节点A向另一个节点B发送请求，B接收处理请求并产生响应结果返回给A。比如如下场景：



机器人巡逻过程中，控制系统分析传感器数据发现可疑物体或人... 此时需要拍摄照片并留存。

在上述场景中，就使用到了服务通信。

- 一个节点需要向相机节点发送拍照请求，相机节点处理请求，并返回处理结果

与上述应用类似的，服务通信更适用于对时效性有要求、具有一定逻辑处理的应用场景。

概念

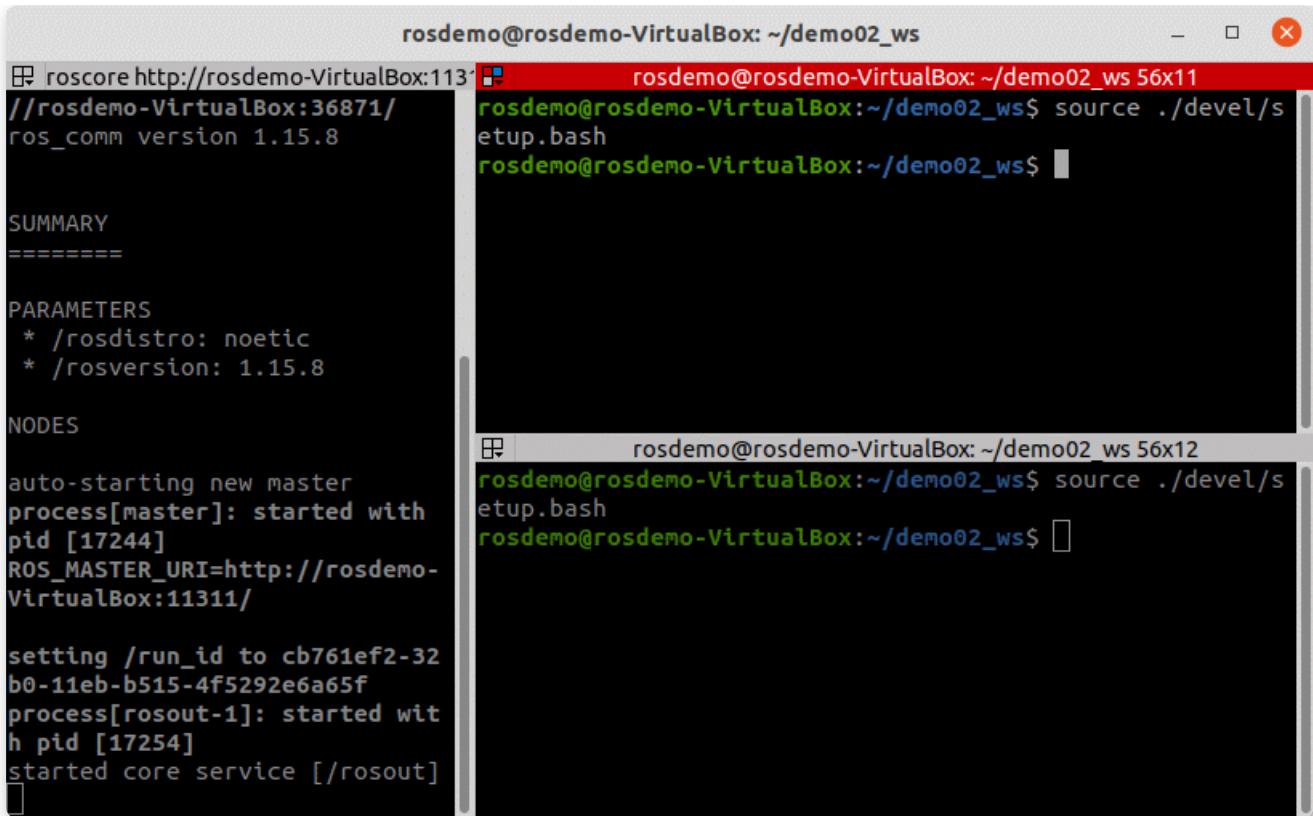
以请求响应的方式实现不同节点之间数据交互的通信模式。

作用

用于偶然的、对时效性有要求、有一定逻辑处理需求的数据传输场景。

案例

实现两个数字的求和，客户端节点，运行会向服务器发送两个数字，服务器端节点接收两个数字求和并将结果响应回客户端。



```

roscore http://rosdemo-VirtualBox:11311
//rosdemo-VirtualBox:36871/ ros_comm version 1.15.8

SUMMARY
=====

PARAMETERS
* /rosdistro: noetic
* /rosversion: 1.15.8

NODES
auto-starting new master
process[master]: started with
pid [17244]
ROS_MASTER_URI=http://rosdemo-
VirtualBox:11311/
setting /run_id to cb761ef2-32
b0-11eb-b515-4f5292e6a65f
process[rosout-1]: started wit
h pid [17254]
started core service [/rosout]

```

另请参考：

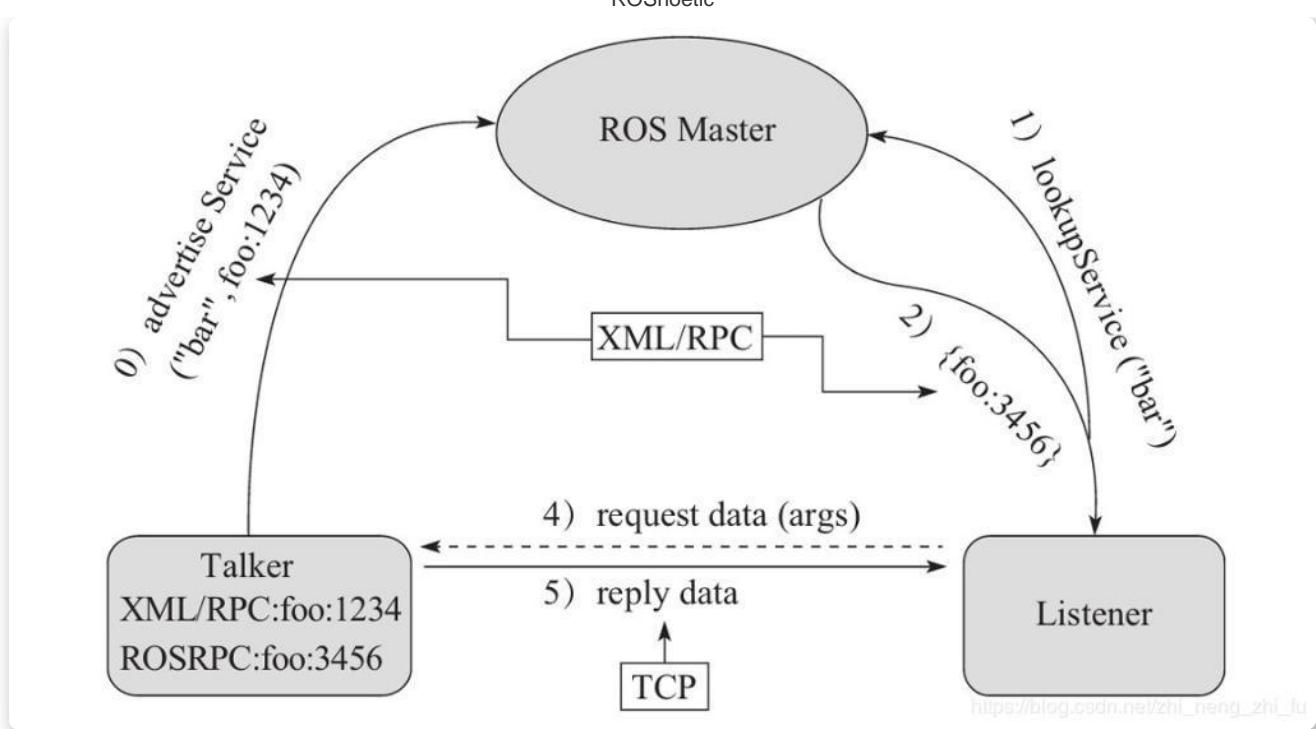
- <http://wiki.ros.org/ROS/Tutorials/CreatingMsgAndSrv>
- <http://wiki.ros.org/ROS/Tutorials/WritingServiceClient%28c%2B%2B%29>
- <http://wiki.ros.org/ROS/Tutorials/WritingServiceClient%28python%29>

2.2.1 服务通信理论模型

服务通信较之于话题通信更简单些，理论模型如下图所示，该模型中涉及到三个角色：

- ROS master(管理者)
- Server(服务端)
- Client(客户端)

ROS Master 负责保管 Server 和 Client 注册的信息，并匹配话题相同的 Server 与 Client，帮助 Server 与 Client 建立连接，连接建立后，Client 发送请求信息，Server 返回响应信息。



整个流程由以下步骤实现:

0. Server注册

Server 启动后，会通过RPC在 ROS Master 中注册自身信息，其中包含提供的服务的名称。ROS Master 会将节点的注册信息加入到注册表中。

1. Client注册

Client 启动后，也会通过RPC在 ROS Master 中注册自身信息，包含需要请求的服务的名称。ROS Master 会将节点的注册信息加入到注册表中。

2. ROS Master实现信息匹配

ROS Master 会根据注册表中的信息匹配Server和 Client，并通过 RPC 向 Client 发送 Server 的 TCP 地址信息。

3. Client发送请求

Client 根据步骤2响应的信息，使用 TCP 与 Server 建立网络连接，并发送请求数据。

4. Server发送响应

Server 接收、解析请求的数据，并产生响应结果返回给 Client。



注意：

1. 客户端请求被处理时，需要保证服务器已经启动；
2. 服务端和客户端都可以存在多个。

2.2.2 服务通信自定义srv

需求：



服务通信中，客户端提交两个整数至服务端，服务端求和并响应结果到客户端，请创建服务与客户端通信的数据载体

`int8, int16, int32, int64 (或者无符号类型: uint*)`
`float32, float64`
`string`
`time, duration`
`other msg files`
`variable-length array[] and fixed-length array[C]`

srv 文件内的可用数据类型与 msg 文件一致，且定义 srv 实现流程与自定义 msg 实现流程类似：

1. 按照固定格式创建srv文件
2. 编辑配置文件
3. 编译生成中间文件

1. 定义srv文件

服务通信中，数据分成两部分，请求与响应，在 srv 文件中请求和响应使用 `---` 分割，具体实现如下：

功能包下新建 srv 目录，添加 `xxx.srv` 文件，内容：

```

1 # 客户端请求时发送的两个数字
2 int32 num1
3 int32 num2
4 ---
5 # 服务器响应发送的数据
6 int32 sum

```

2. 编辑配置文件

package.xml中添加编译依赖与执行依赖

```

1 <build_depend>message_generation</build_depend>
2 <exec_depend>message_runtime</exec_depend>
3 <!--
4 exce_depend 以前对应的是 run_depend 现在非法
5 -->

```

CMakeLists.txt编辑 srv 相关配置

```

1 find_package(catkin REQUIRED COMPONENTS
2   roscpp
3   rospy
4   std_msgs
5   message_generation
6 )
7 # 需要加入 message_generation, 必须有 std_msgs
8 add_service_files(
9   FILES
10  AddInts.srv
11 )
12 generate_messages(
13   DEPENDENCIES
14   std_msgs
15 )

```

注意: 官网没有在 catkin_package 中配置 message_runtime, 经测试配置也可以

3. 编译

编译后的中间文件查看：

C++ 需要调用的中间文件(.../工作空间/devel/include/包名/xxx.h)

```

C AddIntsResponse.h
...
C AddIntsRequest.h
...
C AddIntsResponse.h

```

Python 需要调用的中间文件(.../工作空间/devel/lib/python3/dist-packages/包名/srv)

```

C Person.h
...
_Person.h

```

后续调用相关 srv 时，是从这些中间文件调用的

2.2.3 服务通信自定义srv调用A(C++)

需求：



编写服务通信，客户端提交两个整数至服务端，服务端求和并响应结果到客户端。

分析：

在模型实现中，ROS master 不需要实现，而连接的建立也已经被封装了，需要关注的关键点有三个：

1. 服务端
2. 客户端
3. 数据

流程：

1. 编写服务端实现；
2. 编写客户端实现；
3. 编辑配置文件；
4. 编译并执行。

0.vscode配置

需要像之前自定义 msg 实现一样配置c_cpp_properties.json 文件，如果以前已经配置且没有变更工作空间，可以忽略，如果需要配置，配置方式与之前相同：

```

1  {
2      "configurations": [
3          {
4              "browse": {
5                  "databaseFilename": "",
6                  "limitSymbolsToIncludedHeaders":
7                      true
8              },
9              "includePath": [
10                  "/opt/ros/noetic/include/**",
11                  "/usr/include/**",
12                  "/xxx/yyy工作空间/devel/include/**"
13          ],
14          "name": "ROS",
15          "intelliSenseMode": "gcc-x64",
16          "compilerPath": "/usr/bin/gcc",
17          "cStandard": "c11",
18          "cppStandard": "c++17"
19      }
20  }

```

```

18         }
19     ],
20     "version": 4
21 }

```

1.服务端

```

1  /*
2   * 需求:
3   * 编写两个节点实现服务通信, 客户端节点需要提交两个整
4   * 数到服务器
5   * 服务器需要解析客户端提交的数据, 相加后, 将结果响应
6   * 回客户端,
7   * 客户端再解析
8
9   * 服务器实现:
10  * 1.包含头文件
11  * 2.初始化 ROS 节点
12  * 3.创建 ROS 句柄
13  * 4.创建 服务 对象
14  * 5.回调函数处理请求并产生响应
15  * 6.由于请求有多个, 需要调用 ros::spin()
16
17 */
18 #include "ros/ros.h"
19 #include "demo03_server_client/AddInts.h"
20
21 // bool 返回值由于标志是否处理成功
22 bool doReq(demo03_server_client::AddInts::Request&
23             req,
24             demo03_server_client::AddInts::Response&
25             resp){
26     int num1 = req.num1;
27     int num2 = req.num2;
28
29     ROS_INFO("服务器接收到的请求数据为: num1 = %d, num2
30             = %d", num1, num2);
31
32     resp.sum = num1 + num2;
33
34     return true;
35 }

```

```

27     //逻辑处理
28     if (num1 < 0 || num2 < 0)
29     {
30         ROS_ERROR("提交的数据异常:数据不可以为负数");
31         return false;
32     }
33
34     //如果没有异常, 那么相加并将结果赋值给 resp
35     resp.sum = num1 + num2;
36     return true;
37
38
39 }
40
41 int main(int argc, char *argv[])
42 {
43     setlocale(LC_ALL, "");
44     // 2. 初始化 ROS 节点
45     ros::init(argc, argv, "AddInts_Server");
46     // 3. 创建 ROS 句柄
47     ros::NodeHandle nh;
48     // 4. 创建 服务 对象
49     ros::ServiceServer server =
50     nh.advertiseService("AddInts", doReq);
51     ROS_INFO("服务已经启动....");
52     // 5. 回调函数处理请求并产生响应
53     // 6. 由于请求有多个, 需要调用 ros::spin()
54     ros::spin();
55     return 0;

```

2.客户端

```

1 /*
2  * 需求:
3  * 编写两个节点实现服务通信, 客户端节点需要提交两个整
4  * 数到服务器

```

```

4           服务器需要解析客户端提交的数据, 相加后, 将结果响应
5           回客户端,
6           客户端再解析
7
8           服务器实现:
9           1. 包含头文件
10          2. 初始化 ROS 节点
11          3. 创建 ROS 句柄
12          4. 创建 客户端 对象
13          5. 请求服务, 接收响应
14 */
15 // 1. 包含头文件
16 #include "ros/ros.h"
17 #include "demo03_server_client/AddInts.h"
18
19 int main(int argc, char *argv[])
20 {
21     setlocale(LC_ALL, "");
22
23     // 调用时动态传值, 如果通过 launch 的 args 传参, 需要
24     // 传递的参数个数 +3
25     // if (argc != 5) // launch 传参(0-文件路径 1传入的
26     // 参数 2传入的参数 3节点名称 4日志路径)
27     {
28         ROS_ERROR("请提交两个整数");
29         return 1;
30     }
31
32     // 2. 初始化 ROS 节点
33     ros::init(argc, argv, "AddInts_Client");
34     // 3. 创建 ROS 句柄
35     ros::NodeHandle nh;
36     // 4. 创建 客户端 对象
37     ros::ServiceClient client =
nh.serviceClient<demo03_server_client::AddInts>
("AddInts");

```

```

38     //等待服务启动成功
39     //方式1
40     ros::service::waitForService("AddInts");
41     //方式2
42     // client.waitForExistence();
43     // 5.组织请求数据
44     demo03_server_client::AddInts ai;
45     ai.request.num1 = atoi(argv[1]);
46     ai.request.num2 = atoi(argv[2]);
47     // 6.发送请求,返回 bool 值, 标记是否成功
48     bool flag = client.call(ai);
49     // 7.处理响应
50     if (flag)
51     {
52         ROS_INFO("请求正常处理,响应结果:%d",ai.response.sum);
53     }
54     else
55     {
56         ROS_ERROR("请求处理失败....");
57         return 1;
58     }
59
60     return 0;
61 }
```

3.配置CMakeLists.txt

```

1 add_executable(AddInts_Server
2                 src/AddInts_Server.cpp)
3
4
5 add_dependencies(AddInts_Server
6                   ${PROJECT_NAME}_gencpp)
7 add_dependencies(AddInts_Client
8                   ${PROJECT_NAME}_gencpp)
```

```

7
8
9 target_link_libraries(AddInts_Server
10   ${catkin_LIBRARIES}
11 )
12 target_link_libraries(AddInts_Client
13   ${catkin_LIBRARIES}
14 )

```

4. 执行

流程:

- 需要先启动服务: `rosrun` 包名 服务
- 然后再调用客户端: `rosrun` 包名 客户端 参数1 参数2

结果:

会根据提交的数据响应相加后的结果。

注意:

如果先启动客户端, 那么会导致运行失败

优化:

在客户端发送请求前添加: `client.waitForExistence();`

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

这是一个阻塞式函数, 只有服务启动成功后才会继续执行

此处可以使用 `launch` 文件优化, 但是需要注意 `args` 传参特点

2.2.4 服务通信自定义srv调用B(Python)

需求:



编写服务通信，客户端提交两个整数至服务端，服务端求和并响应结果到客户端。

分析：

在模型实现中，ROS master 不需要实现，而连接的建立也已经被封装了，需要关注的关键点有三个：

1. 服务端
2. 客户端
3. 数据

流程：

1. 编写服务端实现；
2. 编写客户端实现；
3. 为python文件添加可执行权限；
4. 编辑配置文件；
5. 编译并执行。

0.vscode配置

需要像之前自定义 msg 实现一样配置settings.json 文件，如果以前已经配置且没有变更工作空间，可以忽略，如果需要配置，配置方式与之前相同：

```

1  {
2      "python.autoComplete.extraPaths": [
3          "/opt/ros/noetic/lib/python3/dist-packages",
4      ]
5  }

```

1.服务端

```

1  #! /usr/bin/env python
2  """
3      需求：
4          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

```

5 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，
 6 客户端再解析
 7
 8 服务器端实现：
 9 1. 导包
 10 2. 初始化 ROS 节点
 11 3. 创建服务对象
 12 4. 回调函数处理请求并产生响应
 13 5. `spin` 函数
 14
 15 """
 16 # 1. 导包
 17 import rospy
 18 from demo03_server_client.srv import
 AddInts, AddIntsRequest, AddIntsResponse
 19 # 回调函数的参数是请求对象，返回值是响应对象
 20 def doReq(req):
 21 # 解析提交的数据
 22 sum = req.num1 + req.num2
 23 rospy.loginfo("提交的数据: num1 = %d, num2 = %d,
 sum = %d", req.num1, req.num2, sum)
 24
 25 # 创建响应对象，赋值并返回
 26 # resp = AddIntsResponse()
 27 # resp.sum = sum
 28 resp = AddIntsResponse(sum)
 29 return resp
 30
 31
 32 if __name__ == "__main__":
 33 # 2. 初始化 ROS 节点
 34 rospy.init_node("addints_server_p")
 35 # 3. 创建服务对象
 36 server =
 rospy.Service("AddInts", AddInts, doReq)
 37 # 4. 回调函数处理请求并产生响应
 38 # 5. `spin` 函数
 39 rospy.spin()

2.客户端

```
1 #! /usr/bin/env python
2
3 """
4     需求：
5         编写两个节点实现服务通信，客户端节点需要提交两个整
6         数到服务器
7         服务器需要解析客户端提交的数据，相加后，将结果响应
8         回客户端，
9         客户端再解析
10
11
12     客户端实现：
13         1. 导包
14         2. 初始化 ROS 节点
15         3. 创建请求对象
16         4. 发送请求
17         5. 接收并处理响应
18
19
20 """
21 #1. 导包
22 import rospy
23 from demo03_server_client.srv import *
24 import sys
25
26 if __name__ == "__main__":
27
28     #优化实现
29     if len(sys.argv) != 3:
30         rospy.logerr("请正确提交参数")
31         sys.exit(1)
32
33
```

```

34      # 2. 初始化 ROS 节点
35      rospy.init_node("AddInts_Client_p")
36      # 3. 创建请求对象
37      client = rospy.ServiceProxy("AddInts", AddInts)
38      # 请求前, 等待服务已经就绪
39      # 方式1:
40      # rospy.wait_for_service("AddInts")
41      # 方式2
42      client.wait_for_service()
43      # 4. 发送请求, 接收并处理响应
44      # 方式1
45      # resp = client(3,4)
46      # 方式2
47      # resp = client/AddIntsRequest(1,5))
48      # 方式3
49      req = AddIntsRequest()
50      # req.num1 = 100
51      # req.num2 = 200
52
53      #优化
54      req.num1 = int(sys.argv[1])
55      req.num2 = int(sys.argv[2])
56
57      resp = client.call(req)
58      rospy.loginfo("响应结果:%d", resp.sum)

```

3. 设置权限

终端下进入 scripts 执行: chmod +x *.py

4. 配置 CMakeLists.txt

CMakeLists.txt

```

1 catkin_install_python(PROGRAMS
2   scripts/AddInts_Server_p.py
3   scripts/AddInts_Client_p.py
4   DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
5 )

```

5. 执行

流程:

- 需要先启动服务: `rosrun` 包名 服务
- 然后再调用客户端: `rosrun` 包名 客户端 参数1 参数2

结果:

会根据提交的数据响应相加后的结果。

2.3 参数服务器

参数服务器在ROS中主要用于实现不同节点之间的数据共享。参数服务器相当于是独立于所有节点的一个公共容器，可以将数据存储在该容器中，被不同的节点调用，当然不同的节点也可以往其中存储数据，关于参数服务器的典型应用场景如下：



导航实现时，会进行路径规划，比如：全局路径规划，设计一个从出发点到目标点的大致路径。本地路径规划，会根据当前路况生成时时的行进路径

上述场景中，全局路径规划和本地路径规划时，就会使用到参数服务器：

- 路径规划时，需要参考小车的尺寸，我们可以将这些尺寸信息存储到参数服务器，全局路径规划节点与本地路径规划节点都可以从参数服务器中调用这些参数

参数服务器，一般适用于存在数据共享的一些应用场景。

概念

以共享的方式实现不同节点之间数据交互的通信模式。

作用

存储一些多节点共享的数据，类似于全局变量。

案例

实现参数增删改查操作。

另请参考:

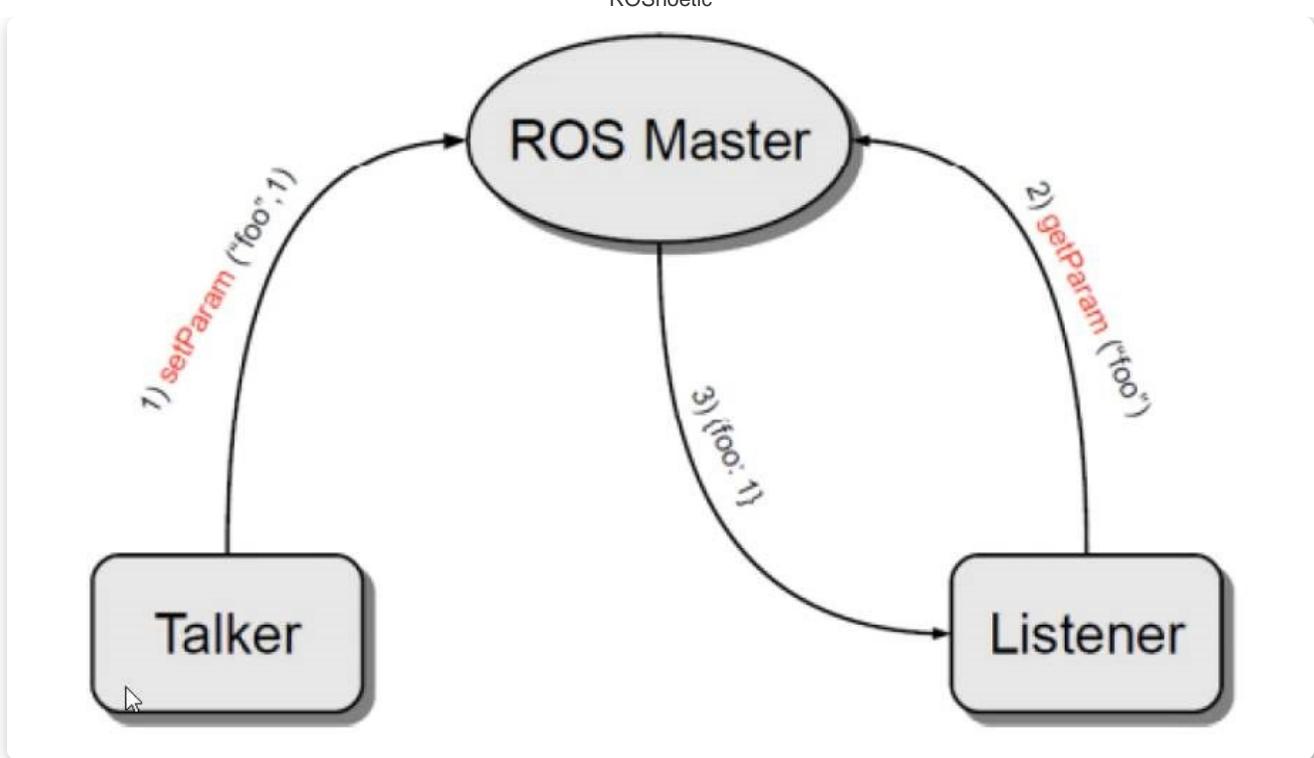
- <http://wiki.ros.org/Parameter%20Server>
- <http://wiki.ros.org/roscpp/Overview/Parameter%20Server>
- <http://wiki.ros.org/rospy/Overview/Parameter%20Server>

2.3.1 参数服务器理论模型

参数服务器实现是最为简单的，该模型如下图所示,该模型中涉及到三个角色:

- ROS Master (管理者)
- Talker (参数设置者)
- Listener (参数调用者)

ROS Master 作为一个公共容器保存参数，Talker 可以向容器中设置参数，Listener 可以获取参数。



整个流程由以下步骤实现:

1. Talker 设置参数

Talker 通过 RPC 向参数服务器发送参数(包括参数名与参数值), ROS Master 将参数保存到参数列表中。

2. Listener 获取参数

Listener 通过 RPC 向参数服务器发送参数查找请求, 请求中包含要查找的参数名。

3. ROS Master 向 Listener 发送参数值

ROS Master 根据步骤2请求提供的参数名查找参数值, 并将查询结果通过 RPC 发送给 Listener。

参数可使用数据类型:

- 32-bit integers
- booleans
- strings

- doubles
- iso8601 dates
- lists
- base64-encoded binary data
- 字典



注意:参数服务器不是为高性能而设计的,因此最好用于存储静态的非二进制的简单数据

2.3.2 参数操作A(C++)

需求:实现参数服务器参数的增删改查操作。

在 C++ 中实现参数服务器数据的增删改查,可以通过两套 API 实现:

- ros::NodeHandle
- ros::param

下面为具体操作演示

1.参数服务器新增(修改)参数

```

1  /*
2   * 参数服务器操作之新增与修改(二者API一样)_C++实现:
3   * 在 rosccpp 中提供了两套 API 实现参数操作
4   * ros::NodeHandle
5   *     setParam("键",值)
6   * ros::param
7   *     set("键","值")
8
9   * 示例:分别设置整形、浮点、字符串、bool、列表、字典等类型
10  *       参数
11  *       修改(相同的键,不同的值)
12 */
13 #include "ros/ros.h"
14

```

```

15 int main(int argc, char *argv[])
16 {
17     ros::init(argc,argv,"set_update_param");
18
19     std::vector<std::string> stus;
20     stus.push_back("zhangsan");
21     stus.push_back("李四");
22     stus.push_back("王五");
23     stus.push_back("孙大脑袋");
24
25     std::map<std::string, std::string> friends;
26     friends["guo"] = "huang";
27     friends["yuang"] = "xiao";
28
29     //NodeHandle-----
30     -----
30     ros::NodeHandle nh;
31     nh.setParam("nh_int",10); //整型
32     nh.setParam("nh_double",3.14); //浮点型
33     nh.setParam("nh_bool",true); //bool
34     nh.setParam("nh_string","hello NodeHandle");
35     //字符串
35     nh.setParam("nh_vector",stus); // vector
36     nh.setParam("nh_map",friends); // map
37
38     //修改演示(相同的键, 不同的值)
39     nh.setParam("nh_int",10000);
40
41     //param-----
41     -----
42     ros::param::set("param_int",20);
43     ros::param::set("param_double",3.14);
44     ros::param::set("param_string","Hello Param");
45     ros::param::set("param_bool",false);
46     ros::param::set("param_vector",stus);
47     ros::param::set("param_map",friends);
48
49     //修改演示(相同的键, 不同的值)
50     ros::param::set("param_int",20000);

```

```

51
52     return 0;
53 }
```

2.参数服务器获取参数

```

1  /*
2   * 参数服务器操作之查询_C++实现:
3   * 在 roscpp 中提供了两套 API 实现参数操作
4   * ros::NodeHandle
5
6   * param(键,默认值)
7   *      存在, 返回对应结果, 否则返回默认值
8
9   * getParam(键,存储结果的变量)
10  *      存在,返回 true,且将值赋值给参数2
11  *      若果键不存在, 那么返回值为 false, 且不为参数
12  *      2赋值
13
14  *      getParamCached键,存储结果的变量)--提高变量获取
15  *      效率
16
17  *      存在,返回 true,且将值赋值给参数2
18  *      若果键不存在, 那么返回值为 false, 且不为参数
19  *      2赋值
20
21  *      hasParam(键)
22  *      是否包含某个键, 存在返回 true, 否则返回
23  *      false
24
25  *      searchParam(参数1, 参数2)
26  *      搜索键, 参数1是被搜索的键, 参数2存储搜索结果
27  *      的变量
28
29  *      ros::param ----- 与 NodeHandle 类似
30
```

```
28
29
30
31
32 */
33
34 #include "ros/ros.h"
35
36 int main(int argc, char *argv[])
37 {
38     setlocale(LC_ALL,"");
39     ros::init(argc,argv,"get_param");
40
41     //NodeHandle-----
42     -----
43     /*
44         ros::NodeHandle nh;
45         // param 函数
46         int res1 = nh.param("nh_int",100); // 键存在
47         int res2 = nh.param("nh_int2",100); // 键不存在
48         ROS_INFO("param获取结果:%d,%d",res1,res2);
49
50         // getParam 函数
51         int nh_int_value;
52         double nh_double_value;
53         bool nh_bool_value;
54         std::string nh_string_value;
55         std::vector<std::string> stus;
56         std::map<std::string, std::string> friends;
57
58         nh.getParam("nh_int",nh_int_value);
59         nh.getParam("nh_double",nh_double_value);
60         nh.getParam("nh_bool",nh_bool_value);
61         nh.getParam("nh_string",nh_string_value);
62         nh.getParam("nh_vector",stus);
63         nh.getParam("nh_map",friends);
64
65         ROS_INFO("getParam获取的结果:%d,%.2f,%s,%d",
66                 nh_int_value,
```

```

66             nh_double_value,
67             nh_string_value.c_str(),
68             nh_bool_value
69         );
70         for (auto &&stu : stus)
71     {
72             ROS_INFO("stus 元素:%s",stu.c_str());
73         }
74
75         for (auto &&f : friends)
76     {
77             ROS_INFO("map 元素:%s =
78 %s",f.first.c_str(), f.second.c_str());
79         }
80
81         //getParamCached()
82         nh.getParamCached("nh_int",nh_int_value);
83         ROS_INFO("通过缓存获取数据:%d",nh_int_value);
84
85         //getParamNames()
86         std::vector<std::string> param_names1;
87         nh.getParamNames(param_names1);
88         for (auto &&name : param_names1)
89     {
90             ROS_INFO("名称解析name = %s",name.c_str());
91         }
92         ROS_INFO("-----");
93
94         ROS_INFO("存在 nh_int 吗?
95 %d",nh.hasParam("nh_int"));
96         ROS_INFO("存在 nh_inttt 吗?
97 %d",nh.hasParam("nh_inttt"));
98
99         std::string key;
100        nh.searchParam("nh_int",key);
101        ROS_INFO("搜索键:%s",key.c_str());
102    */

```

```

100     //param-----
101
102     ROS_INFO("+++++++++++++++++++++");
103     int res3 = ros::param::param("param_int",20);
104     //存在
105     int res4 = ros::param::param("param_int2",20);
106     // 不存在返回默认
107     ROS_INFO("param获取结果:%d,%d",res3,res4);
108
109     // getParam 函数
110     int param_int_value;
111     double param_double_value;
112     bool param_bool_value;
113     std::string param_string_value;
114     std::vector<std::string> param_stus;
115     std::map<std::string, std::string>
116     param_friends;
117
118     ros::param::get("param_int",param_int_value);
119
120     ros::param::get("param_double",param_double_value);
121
122     ros::param::get("param_bool",param_bool_value);
123
124     ros::param::get("param_string",param_string_value);
125
126     ros::param::get("param_vector",param_stus);
127     ros::param::get("param_map",param_friends);
128
129     ROS_INFO("getParam获取的结果:%d,%f,%s,%d",
130             param_int_value,
131             param_double_value,
132             param_string_value.c_str(),
133             param_bool_value
134         );
135
136     for (auto &&stu : param_stus)

```

```

128     {
129         ROS_INFO("stus 元素:%s",stu.c_str());
130     }
131
132     for (auto &&f : param_friends)
133     {
134         ROS_INFO("map 元素:%s =
135             %s",f.first.c_str(), f.second.c_str());
136     }
137
138     // getParamCached()
139
140     ros::param::getCached("param_int",param_int_value);
141
142     ROS_INFO("通过缓存获取数据:%d",param_int_value);
143
144     //getParamNames()
145     std::vector<std::string> param_names2;
146     ros::param::getParamNames(param_names2);
147     for (auto &&name : param_names2)
148     {
149         ROS_INFO("名称解析name = %s",name.c_str());
150
151         ROS_INFO("-----");
152
153         ROS_INFO("存在 param_int 吗?
154             %d",ros::param::has("param_int"));
155         ROS_INFO("存在 param_inttt 吗?
156             %d",ros::param::has("param_inttt"));
157
158         std::string key;
159         ros::param::search("param_int",key);
160         ROS_INFO("搜索键:%s",key.c_str());
161
162         return 0;
163     }

```

3.参数服务器删除参数

```
1  /*
2   * 参数服务器操作之删除_C++实现:
3   *
4   * ros::NodeHandle
5   *     deleteParam("键")
6   *     根据键删除参数, 删除成功, 返回 true, 否则(参数不存在), 返回 false
7   *
8   * ros::param
9   *     del("键")
10  *     根据键删除参数, 删除成功, 返回 true, 否则(参数不存在), 返回 false
11
12
13 */
14 #include "ros/ros.h"
15
16
17 int main(int argc, char *argv[])
18 {
19     setlocale(LC_ALL, "");
20     ros::init(argc, argv, "delete_param");
21
22     ros::NodeHandle nh;
23     bool r1 = nh.deleteParam("nh_int");
24     ROS_INFO("nh 删除结果:%d", r1);
25
26     bool r2 = ros::param::del("param_int");
27     ROS_INFO("param 删除结果:%d", r2);
28
29     return 0;
30 }
```

2.3.3 参数操作B(Python)

需求:实现参数服务器参数的增删改查操作。

1.参数服务器新增(修改)参数

```

1  #! /usr/bin/env python
2  """
3      参数服务器操作之新增与修改(二者API一样)_Python实现:
4  """
5
6  import rospy
7
8  if __name__ == "__main__":
9      rospy.init_node("set_update_paramter_p")
10
11     # 设置各种类型参数
12     rospy.set_param("p_int",10)
13     rospy.set_param("p_double",3.14)
14     rospy.set_param("p_bool",True)
15     rospy.set_param("p_string","hello python")
16     rospy.set_param("p_list",
17                     ["hello","haha","xixi"])
18     rospy.set_param("p_dict",
19                     {"name":"hulu","age":8})
20
21     # 修改
22     rospy.set_param("p_int",100)

```

2.参数服务器获取参数

```

1  #! /usr/bin/env python
2
3  """
4      参数服务器操作之查询_Python实现:
5      get_param(键,默认值)
6          当键存在时, 返回对应的值, 如果不存在返回默认值

```

```
7      get_param_cached
8      get_param_names
9      has_param
10     search_param
11 """
12
13 import rospy
14
15 if __name__ == "__main__":
16     rospy.init_node("get_param_p")
17
18     #获取参数
19     int_value = rospy.get_param("p_int",10000)
20     double_value = rospy.get_param("p_double")
21     bool_value = rospy.get_param("p_bool")
22     string_value = rospy.get_param("p_string")
23     p_list = rospy.get_param("p_list")
24     p_dict = rospy.get_param("p_dict")
25
26     rospy.loginfo("获取的数据:%d,%f,%d,%s",
27                   int_value,
28                   double_value,
29                   bool_value,
30                   string_value)
31     for ele in p_list:
32         rospy.loginfo("ele = %s", ele)
33
34     rospy.loginfo("name = %s, age =
35     %d",p_dict["name"],p_dict["age"])
36
37     # get_param_cached
38     int_cached = rospy.get_param_cached("p_int")
39     rospy.loginfo("缓存数据:%d",int_cached)
40
41     # get_param_names
42     names = rospy.get_param_names()
43     for name in names:
44         rospy.loginfo("name = %s",name)
```

```

45     rospy.loginfo("-"★80)
46
47     # has_param
48     flag = rospy.has_param("p_int")
49     rospy.loginfo("包含p_int吗? %d", flag)
50
51     # search_param
52     key = rospy.search_param("p_int")
53     rospy.loginfo("搜索的键 = %s", key)

```

3.参数服务器删除参数

```

1  #! /usr/bin/env python
2  """
3      参数服务器操作之删除_Python实现:
4      rospy.delete_param("键")
5      键存在时, 可以删除成功, 键不存在时, 会抛出异常
6  """
7  import rospy
8
9  if __name__ == "__main__":
10     rospy.init_node("delete_param_p")
11
12     try:
13         rospy.delete_param("p_int")
14     except Exception as e:
15         rospy.loginfo("删除失败")

```

2.4 常用命令

机器人系统中启动的节点少则几个, 多则十几个、几十个, 不同的节点名称各异, 通信时使用话题、服务、消息、参数等等都各不相同, 一个显而易见的问题是: 当需要自定义节点和其他某个已经存在的节点通信时, 如何获取对方的话题、以及消息载体的格式呢?

在 ROS 同提供了一些实用的命令行工具，可以用于获取不同节点的各类信息，常用的命令如下：

- `rosnode` : 操作节点
 - `rostopic` : 操作话题
 - `rosservice` : 操作服务
 - `rosmsg` : 操作msg消息
 - `rossrv` : 操作srv消息
 - `rosparam` : 操作参数
-

作用

和之前介绍的文件系统操作命令比较，文件操作命令是静态的，操作的是磁盘上的文件，而上述命令是动态的，在ROS程序启动后，可以动态的获取运行中的节点或参数的相关信息。

案例

本节将借助于2.1、2.2和2.3的通信实现介绍相关命令的基本使用，并通过练习ROS内置的小海龟例程来强化命令的应用。

另请参考：

- <http://wiki.ros.org/ROS/CommandLineTools>

ROS Command-line tools

目录

1. ROS Command-line tools
 1. Common user tools
 1. [rosbag](#)
 2. [rosbash](#)
 3. [roscd](#)
 4. [rosclean](#)
 5. [roscore](#)
 6. [rosdep](#)
 7. [rosed](#)
 8. [roscreate-pkg](#)
 9. [roscreate-stack](#)
 10. [rosrun](#)
 11. [roslaunch](#)
 12. [roslocate](#)
 13. [rosmake](#)
 14. [rosmsg](#)
 15. [rosnode](#)
 16. [rospack](#)
 17. [rosparam](#)
 18. [rossrv](#)
 19. [rosservice](#)
 20. [rosstack](#)
 21. [rostopic](#)
 22. [rosversion](#)
2. Graphical tools
 1. [rqt_bag](#)
 2. [rqt_deps](#)
 3. [rqt_graph](#)
 4. [rqt_plot](#)
3. Less-used tools
 1. [gendeps](#)

维基

[Distributions](#)
[ROS/Installation](#)
[ROS/Tutorials](#)
[RecentChanges](#)
[ROS/CommandLineT](#)

网页

只读网页

信息

附件

更多操作:

[源码](#) ▾

用户

登录

2.4.1 rosnode

rosnode 是用于获取节点信息的命令

- 1 **rosnode ping** 测试到节点的连接状态
- 2 **rosnode list** 列出活动节点
- 3 **rosnode info** 打印节点信息
- 4 **rosnode machine** 列出指定设备上节点
- 5 **rosnode kill** 杀死某个节点
- 6 **rosnode cleanup** 清除不可连接的节点

- **rosnode ping**

测试到节点的连接状态

- **rosnode list**

列出活动节点

- **rosnode info**

打印节点信息

- **rosnode machine**
列出指定设备上的节点
- **rosnode kill**
杀死某个节点
- **rosnode cleanup**
清除无用节点，启动乌龟节点，然后 `ctrl + c` 关闭，该节点并没被彻底清除，可以使用 `cleanup` 清除节点

2.4.2 **rostopic**

rostopic 包含 `rostopic` 命令行工具，用于显示有关 ROS 主题的调试信息，包括发布者，订阅者，发布频率和 ROS 消息。它还包含一个实验性 Python 库，用于动态获取有关主题的信息并与之交互。

1 <code>rostopic bw</code>	显示主题使用的带宽
2 <code>rostopic delay</code>	显示带有 <code>header</code> 的主题延迟
3 <code>rostopic echo</code>	打印消息到屏幕
4 <code>rostopic find</code>	根据类型查找主题
5 <code>rostopic hz</code>	显示主题的发布频率
6 <code>rostopic info</code>	显示主题相关信息
7 <code>rostopic list</code>	显示所有活动状态下的主题
8 <code>rostopic pub</code>	将数据发布到主题
9 <code>rostopic type</code>	打印主题类型

rostopic list(-v)

直接调用即可，控制台将打印当前运行状态下的主题名称

`rostopic list -v` : 获取话题详情(比如列出：发布者和订阅者个数...)

rostopic pub

可以直接调用命令向订阅者发布消息

为 `roboware` 自动生成的 `发布/订阅` 模型案例中的 `订阅者` 发布一条字符串

```

1 rostopic pub /主题名称 消息类型 消息内容
2 rostopic pub /chatter std_msgs/gagaxixi

```

为 小乌龟案例的 订阅者 发布一条运动信息

```

1 rostopic pub /turtle1/cmd_vel geometry_msgs/Twist
2   "linear:
3     x: 1.0
4     y: 0.0
5     z: 0.0
6   angular:
7     x: 0.0
8     y: 0.0
9     z: 2.0"
10 //只发布一次运动信息
11
12 rostopic pub -r 10 /turtle1/cmd_vel
   geometry_msgs/Twist
13   "linear:
14     x: 1.0
15     y: 0.0
16     z: 0.0
17   angular:
18     x: 0.0
19     y: 0.0
20     z: 2.0"
21 // 以 10HZ 的频率循环发送运动信息

```

rostopic echo

获取指定话题当前发布的消息

rostopic info

获取当前话题的小关信息

消息类型

发布者信息

订阅者信息

rostopic type

列出话题的消息类型

rostopic find 消息类型

根据消息类型查找话题

rostopic delay

列出消息头信息

rostopic hz

列出消息发布频率

rostopic bw

列出消息发布带宽

2.4.3 rosmsg

rosmsg是用于显示有关 ROS消息类型的信息的命令行工具。

rosmsg 演示

1	rosmsg show	显示消息描述
2	rosmsg info	显示消息信息
3	rosmsg list	列出所有消息
4	rosmsg md5	显示 md5 加密后的消息
5	rosmsg package	显示某个功能包下的所有消息
6	rosmsg packages	列出包含消息的功能包

rosmsg list

会列出当前 ROS 中的所有 msg

rosmsg packages

列出包含消息的所有包

rosmsg package

列出某个包下的所有msg

```
1 //rosmsg package 包名
2 rosmsg package turtlesim
```

rosmsg show

显示消息描述

```
1 //rosmsg show 消息名称
2 rosmsg show turtlesim/Pose
3 结果:
4 float32 x
5 float32 y
6 float32 theta
7 float32 linear_velocity
8 float32 angular_velocity
```

rosmsg info

作用与 rosmsg show 一样

rosmsg md5 (资料:http://wiki.ros.org/ROS/Technical%20Overview#Message_serialization_and_msg_MD5_sums)

一种校验算法，保证数据传输的一致性

2.4.4 rosservice

rosservice包含用于列出和查询ROS[Services](#)的rosservice命令行工具。

调用部分服务时，如果对相关工作空间没有配置 path，需要进入工作空间调用 `source ./devel/setup.bash`

1	rosservice args	打印服务参数
2	rosservice call	使用提供的参数调用服务
3	rosservice find	按照服务类型查找服务
4	rosservice info	打印有关服务的信息
5	rosservice list	列出所有活动的服务
6	rosservice type	打印服务类型
7	rosservice uri	打印服务的 ROSRPC uri

rosservice list

列出所有活动的 service

```

1 ~ rosservice list
2 /clear
3 /kill
4 /listener/get_loggers
5 /listener/set_logger_level
6 /reset
7 /rosout/get_loggers
8 /rosout/set_logger_level
9 /rostopic_4985_1578723066421/get_loggers
10 /rostopic_4985_1578723066421/set_logger_level
11 /rostopic_5582_1578724343069/get_loggers
12 /rostopic_5582_1578724343069/set_logger_level
13 /spawn
14 /turtle1/set_pen
15 /turtle1/teleport_absolute
16 /turtle1/teleport_relative
17 /turtlesim/get_loggers
18 /turtlesim/set_logger_level

```

rosservice args

打印服务参数

```

1 rosservice args /spawn
2 x y theta name

```

rosservice call

调用服务

为小乌龟的案例生成一只新的乌龟

```
1 rosservice call /spawn "x: 1.0
2 y: 2.0
3 theta: 0.0
4 name: 'xxx'"
5 name: "xxx"
6
7 //生成一只叫 xxx 的乌龟
```

rosservice find

根据消息类型获取话题

rosservice info

获取服务话题详情

rosservice type

获取消息类型

rosservice uri

获取服务器 uri

2.4.5 rossrv

rossrv是用于显示有关ROS服务类型的信息的命令行工具，与 rosmsg 使用语法高度雷同。

```

1 rossrv show      显示服务消息详情
2 rossrv info      显示服务消息相关信息
3 rossrv list      列出所有服务信息
4 rossrv md5       显示 md5 加密后的服务消息
5 rossrv package   显示某个包下所有服务消息
6 rossrv packages  显示包含服务消息的所有包

```

rossrv list

会列出当前 ROS 中的所有 srv 消息

rossrv packages

列出包含服务消息的所有包

rossrv package

列出某个包下的所有msg

```

1 //rossrv package 包名
2 rossrv package turtlesim

```

rossrv show

显示消息描述

```

1 //rossrv show 消息名称
2 rossrv show turtlesim/Spawn
3 结果:
4 float32 x
5 float32 y
6 float32 theta
7 string name
8 ---
9 string name

```

rossrv info

作用与 rossrv show 一致

```
rossrv md5
```

对 service 数据使用 md5 校验(加密)

2.4.6 rosparam

rosparam包含rosparam命令行工具，用于使用YAML编码文件在参数服务器上获取和设置ROS参数。

1	rosparam set	设置参数
2	rosparam get	获取参数
3	rosparam load	从外部文件加载参数
4	rosparam dump	将参数写出到外部文件
5	rosparam delete	删除参数
6	rosparam list	列出所有参数

rosparam list

列出所有参数

```
1 rosparam list
2
3 //默认结果
4 /rosdistro
5 /roslaunch/uris/host_helloros_virtual_machine__42911
6 /rosversion
7 /run_id
```

rosparam set

设置参数

```
1 rosparam set name huluwa
2
3 //再次调用 rosparam list 结果
4 /name
5 /rosdistro
6 /roslaunch/uris/host_helloros_virtual_machine__42911
7 /rosversion
8 /run_id
```

rosparam get

获取参数

```
1 rosparam get name
2
3 //结果
4 huluwa
```

rosparam delete

删除参数

```
1 rosparam delete name
2
3 //结果
4 //去除了name
```

rosparam load(先准备 yaml 文件)

从外部文件加载参数

```
1 rosparam load xxx.yaml
```

rosparam dump

将参数写出到外部文件

```
1 rosparam dump yyy.yaml
```

2.5 通信机制实操

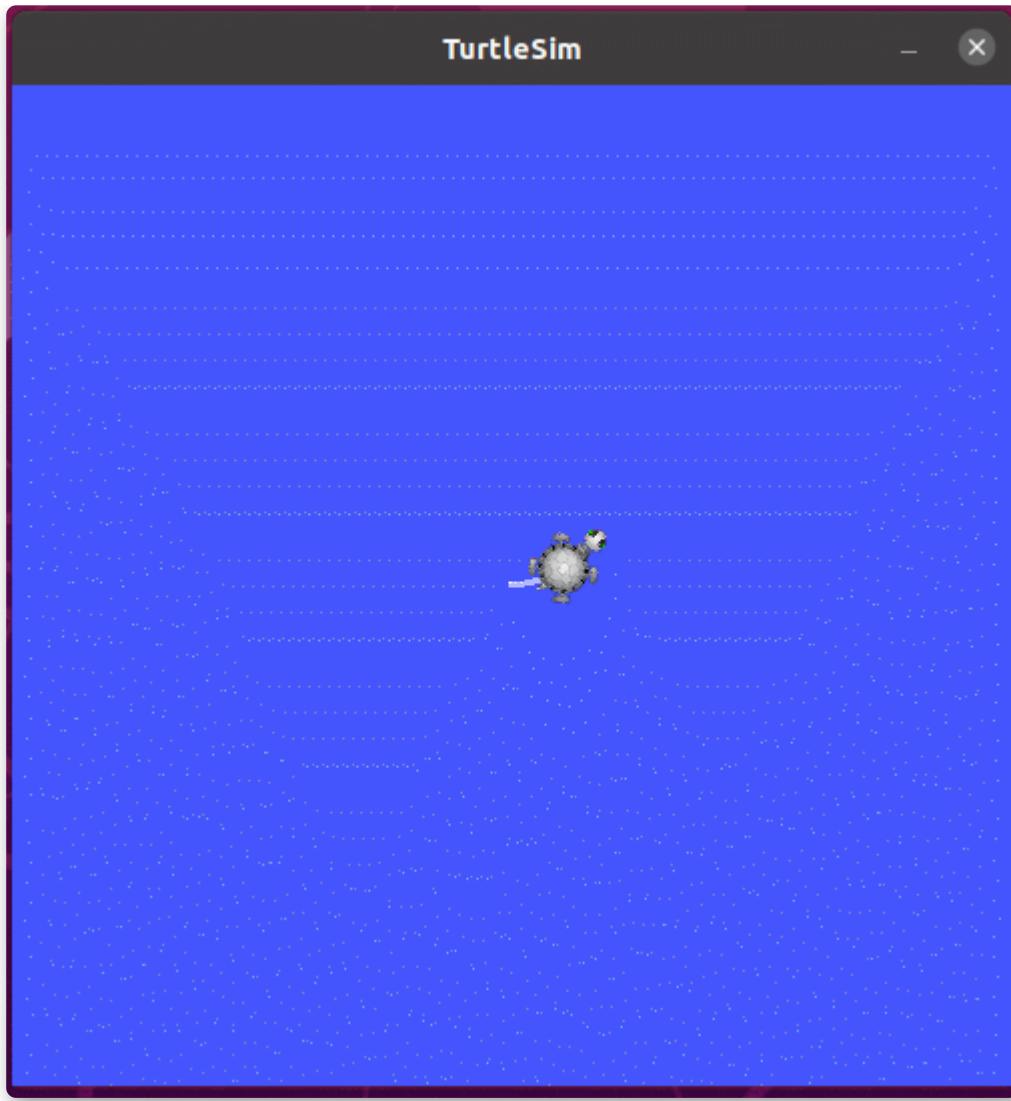
本节主要是通过ROS内置的turtlesim案例，结合已经介绍ROS命令获取节点、话题、话题消息、服务、服务消息与参数的信息，最终再以编码的方式实现乌龟运动的控制、乌龟位姿的订阅、乌龟生成与乌龟窗体背景颜色的修改。

目的:熟悉、强化通信模式应用

2.5.1 实操01_话题发布

需求描述:编码实现乌龟运动控制，让小乌龟做圆周运动。

结果演示:



实现分析:

1. 乌龟运动控制实现，关键节点有两个，一个是乌龟运动显示节点 `turtlesim_node`，另一个是控制节点，二者是订阅发布模式实现通信的，乌龟运动显示节点直接调用即可，运动控制节点之前是使用的 `turtle_teleop_key` 通过键盘控制，现在需要自定义控制节点。
2. 控制节点自实现时，首先需要了解控制节点与显示节点通信使用的话题与消息，可以使用 `ros` 命令结合计算图来获取。
3. 了解了话题与消息之后，通过 C++ 或 Python 编写运动控制节点，通过指定的话题，按照一定的逻辑发布消息即可。

实现流程:

1. 通过计算图结合 `ros` 命令获取话题与消息信息。
2. 编码实现运动控制节点。
3. 启动 `roscore`、`turtlesim_node` 以及自定义的控制节点，查看运行结果。

1.话题与消息获取

准备: 先启动键盘控制乌龟运动案例。

1.1话题获取

获取话题:/turtle1/cmd_vel

通过计算图查看话题，启动计算图：

```
1 rqt_graph
```

或者通过 rostopic 列出话题：

```
1 rostopic list
```

1.2消息获取

获取消息类型:geometry_msgs/Twist

```
1 rostopic type /turtle1/cmd_vel
```

获取消息格式:

```
1 rosmsg info geometry_msgs/Twist
```

响应结果:

```
1 geometry_msgs/Vector3 linear
2   float64 x
3   float64 y
4   float64 z
5 geometry_msgs/Vector3 angular
6   float64 x
7   float64 y
8   float64 z
```

linear(线速度)下的xyz分别对应在x、y和z方向上的速度(单位是m/s);

angular(角速度)下的xyz分别对应x轴上的翻滚、y轴上俯仰和z轴上偏航的速度(单位是rad/s)。



详情请查看补充资料。

2. 实现发布节点

创建功能包需要依赖的功能包: `roscpp` `rospy` `std_msgs` `geometry_msgs`

实现方案A: C++

```

1  /*
2   编写 ROS 节点, 控制小乌龟画圆
3
4   准备工作:
5   1. 获取topic(已知: /turtle1/cmd_vel)
6   2. 获取消息类型(已知: geometry_msgs/Twist)
7   3. 运行前, 注意先启动 turtlesim_node 节点
8
9   实现流程:
10  1. 包含头文件
11  2. 初始化 ROS 节点
12  3. 创建发布者对象
13  4. 循环发布运动控制消息
14 */
15
16 #include "ros/ros.h"
17 #include "geometry_msgs/Twist.h"
18
19 int main(int argc, char *argv[])
20 {
21     setlocale(LC_ALL, "");
22     // 2. 初始化 ROS 节点
23     ros::init(argc, argv, "control");
24     ros::NodeHandle nh;

```

```

25     // 3. 创建发布者对象
26     ros::Publisher pub =
27         nh.advertise<geometry_msgs::Twist>
28         ("/turtle1/cmd_vel", 1000);
29     // 4. 循环发布运动控制消息
30     //4-1. 组织消息
31     geometry_msgs::Twist msg;
32     msg.linear.x = 1.0;
33     msg.linear.y = 0.0;
34     msg.linear.z = 0.0;
35
36     msg.angular.x = 0.0;
37     msg.angular.y = 0.0;
38     msg.angular.z = 2.0;
39
40     //4-2. 设置发送频率
41     ros::Rate r(10);
42     //4-3. 循环发送
43     while (ros::ok())
44     {
45         pub.publish(msg);
46
47         ros::spinOnce();
48     }
49
50 }

```

配置文件此处略

实现方案B: Python

```

1  #! /usr/bin/env python
2  """
3      编写 ROS 节点，控制小乌龟画圆
4
5      准备工作：

```

```

6      1. 获取topic(已知: /turtle1/cmd_vel)
7      2. 获取消息类型(已知: geometry_msgs/Twist)
8      3. 运行前, 注意先启动 turtlesim_node 节点
9
10     实现流程:
11         1. 导包
12         2. 初始化 ROS 节点
13         3. 创建发布者对象
14         4. 循环发布运动控制消息
15
16     """
17
18     import rospy
19     from geometry_msgs.msg import Twist
20
21     if __name__ == "__main__":
22         # 2. 初始化 ROS 节点
23         rospy.init_node("control_circle_p")
24         # 3. 创建发布者对象
25         pub =
26             rospy.Publisher("/turtle1/cmd_vel", Twist, queue_size=1000)
27         # 4. 循环发布运动控制消息
28         rate = rospy.Rate(10)
29         msg = Twist()
30         msg.linear.x = 1.0
31         msg.linear.y = 0.0
32         msg.linear.z = 0.0
33         msg.angular.x = 0.0
34         msg.angular.y = 0.0
35         msg.angular.z = 0.5
36
37         while not rospy.is_shutdown():
38             pub.publish(msg)
39             rate.sleep()

```

权限设置以及配置文件此处略

3.运行

首先，启动 roscore；

然后启动乌龟显示节点；

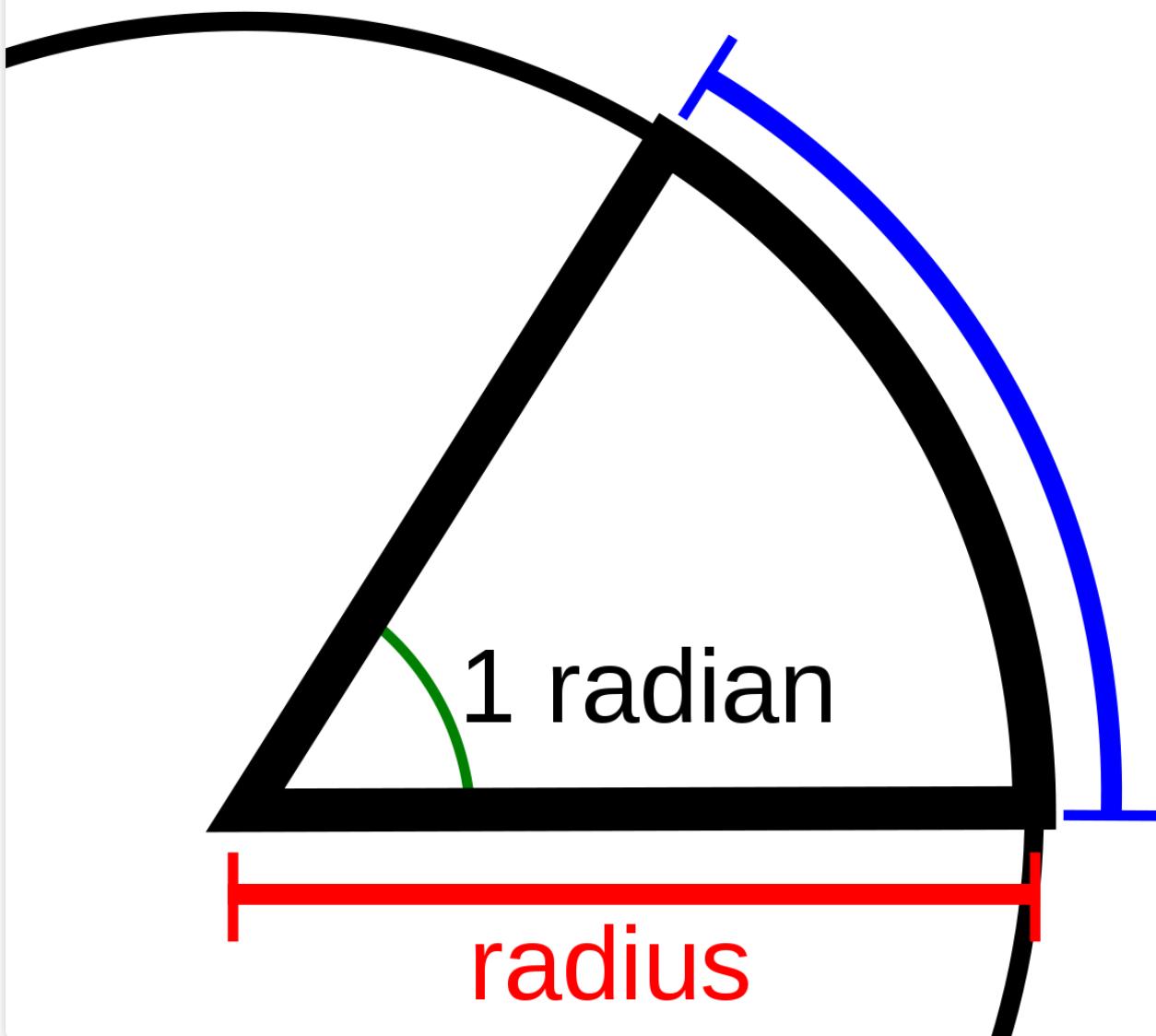
最后执行运动控制节点；

最终执行结果与演示结果类似。

补充资料1：

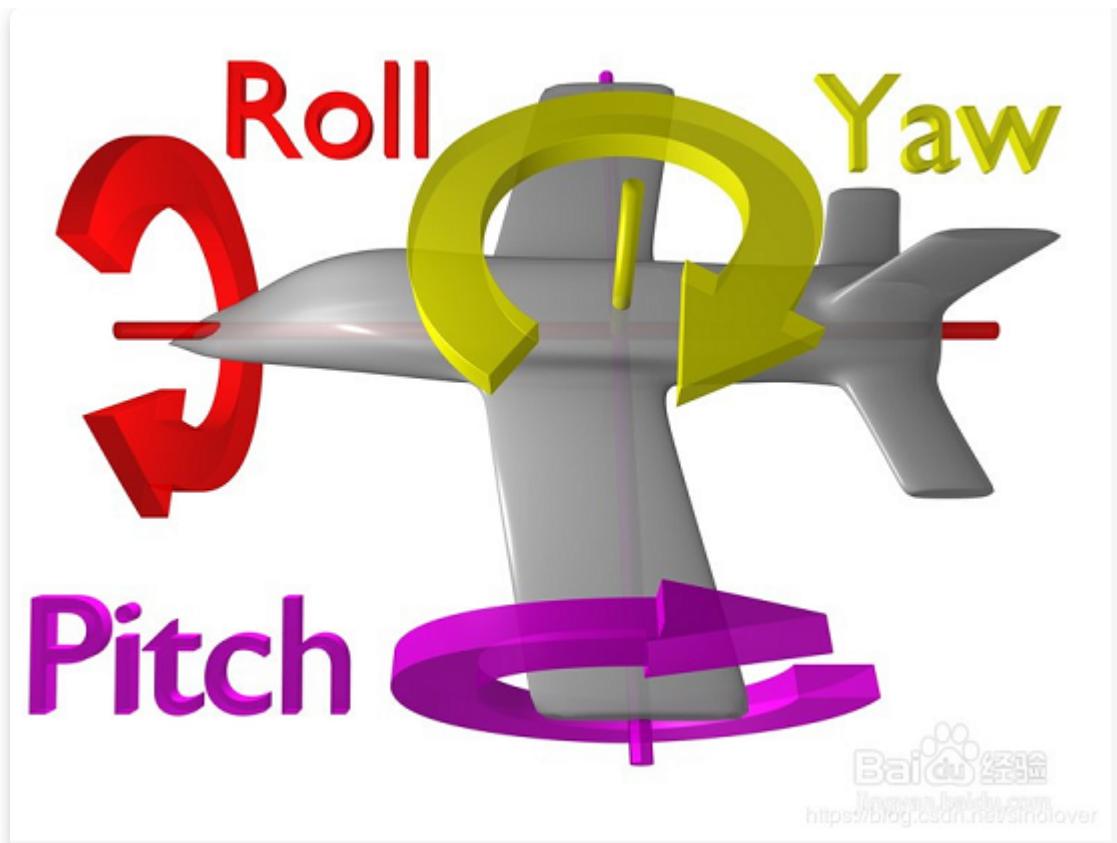
弧度：单位弧度定义为圆弧长度等于半径时的圆心角。

arc length = radius

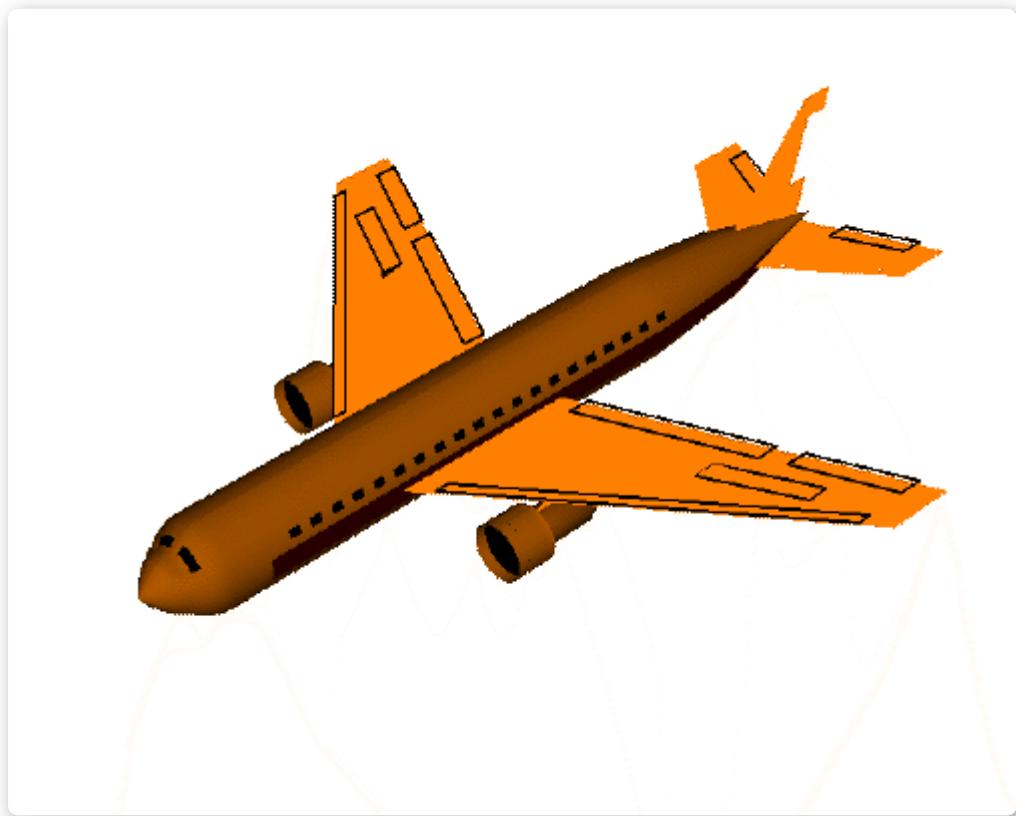


补充资料2:偏航、翻滚与俯仰

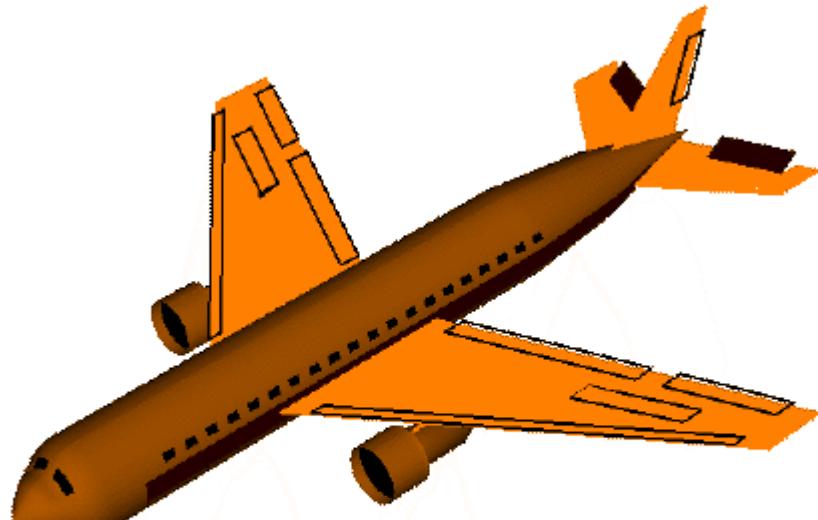
坐标系图解:



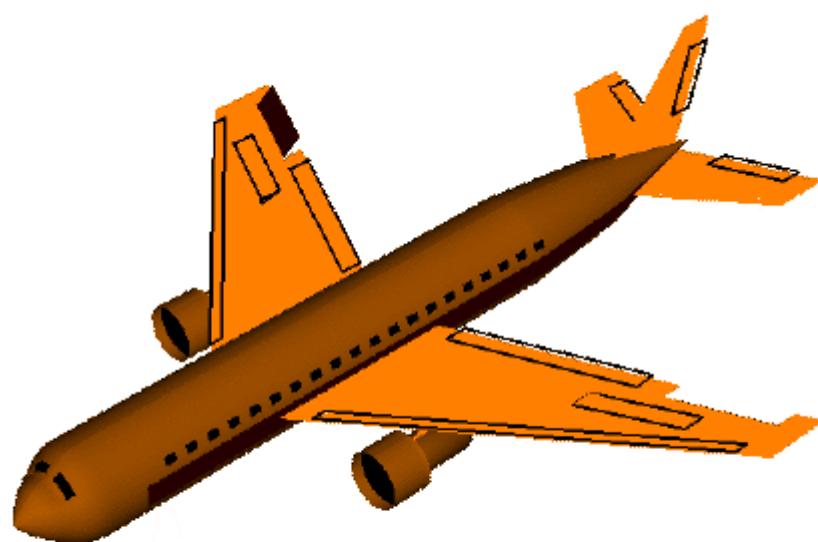
偏航:



俯仰:



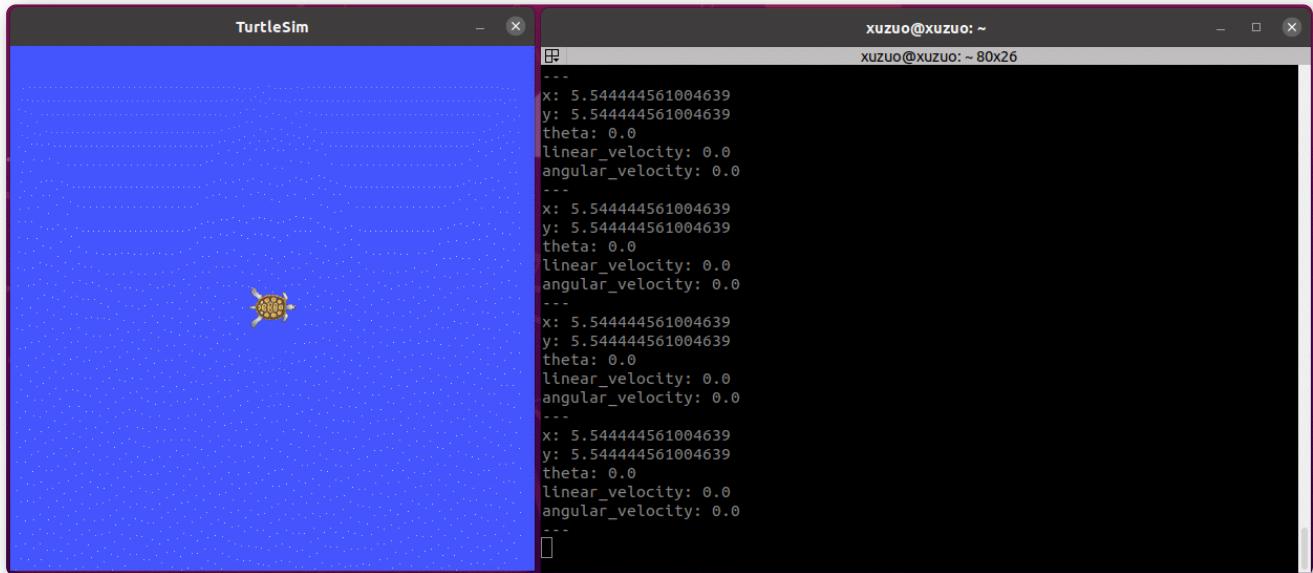
翻滚:



2.5.2 实操02_话题订阅

需求描述: 已知turtlesim中的乌龟显示节点，会发布当前乌龟的位姿(窗体中乌龟的坐标以及朝向)，要求控制乌龟运动，并时时打印当前乌龟的位姿。

结果演示:



实现分析:

1. 首先，需要启动乌龟显示以及运动控制节点并控制乌龟运动。
2. 要通过ROS命令，来获取乌龟位姿发布的话题以及消息。
3. 编写订阅节点，订阅并打印乌龟的位姿。

实现流程:

1. 通过ros命令获取话题与消息信息。
2. 编码实现位姿获取节点。
3. 启动 roscore、 turtlesim_node 、控制节点以及位姿订阅节点，控制乌龟运动并输出乌龟的位姿。

1.话题与消息获取

获取话题:/turtle1/pose

```
1 rostopic list
```

获取消息类型:turtlesim/Pose

```
1 rostopic type /turtle1/pose
```

获取消息格式:

```
1 rosmsg info turtlesim/Pose
```

响应结果:

```
1 float32 x
2 float32 y
3 float32 theta
4 float32 linear_velocity
5 float32 angular_velocity
```

2. 实现订阅节点

创建功能包需要依赖的功能包: roscpp rospy std_msgs turtlesim

实现方案A: C++

```
1 /*
2  订阅小乌龟的位姿：时时获取小乌龟在窗体中的坐标并打印
3  准备工作：
4      1. 获取话题名称 /turtle1/pose
5      2. 获取消息类型 turtlesim/Pose
6      3. 运行前启动 turtlesim_node 与
7          turtle_teleop_key 节点
8
9  实现流程：
10     1. 包含头文件
11     2. 初始化 ROS 节点
12     3. 创建 ROS 句柄
13     4. 创建订阅者对象
14     5. 回调函数处理订阅的数据
15     6. spin
16
17 #include "ros/ros.h"
18 #include "turtlesim/Pose.h"
19
20 void doPose(const turtlesim::Pose::ConstPtr& p){
```

```

21     ROS_INFO("乌龟位姿信息:x=%f,y=%f,theta=%f,lv=%f,av=%f",
22             p->x,p->y,p->theta,p->linear_velocity,p-
23             >angular_velocity
24     );
25
26 int main(int argc, char *argv[])
27 {
28     setlocale(LC_ALL,"");
29     // 2. 初始化 ROS 节点
30     ros::init(argc,argv,"sub_pose");
31     // 3. 创建 ROS 句柄
32     ros::NodeHandle nh;
33     // 4. 创建订阅者对象
34     ros::Subscriber sub =
35         nh.subscribe<turtlesim::Pose>
36         ("~/turtle1/pose",1000,doPose);
37     // 5. 回调函数处理订阅的数据
38     // 6. spin
39     ros::spin();
40     return 0;
41 }
```

配置文件此处略

实现方案B: Python

```

1  #! /usr/bin/env python
2 """
3     订阅小乌龟的位姿：时时获取小乌龟在窗体中的坐标并打印
4     准备工作：
5         1. 获取话题名称 /turtle1/pose
6         2. 获取消息类型 turtlesim/Pose
7         3. 运行前启动 turtlesim_node 与
8             turtle_teleop_key 节点
9
10    实现流程：
```

```

10      1. 导包
11      2. 初始化 ROS 节点
12      3. 创建订阅者对象
13      4. 回调函数处理订阅的数据
14      5. spin
15
16  """
17
18 import rospy
19 from turtlesim.msg import Pose
20
21 def doPose(data):
22     rospy.loginfo("乌龟坐标:x=%s, y=%s, theta=%s", data.x, data.y, data.theta)
23
24 if __name__ == "__main__":
25
26     # 2. 初始化 ROS 节点
27     rospy.init_node("sub_pose_p")
28
29     # 3. 创建订阅者对象
30     sub =
31         rospy.Subscriber("/turtle1/pose", Pose, doPose, queue_
32         _size=1000)
33         # 4. 回调函数处理订阅的数据
34         # 5. spin
35         rospy.spin()

```

权限设置以及配置文件此处略

3.运行

首先，启动 roscore；

然后启动乌龟显示节点，执行运动控制节点；

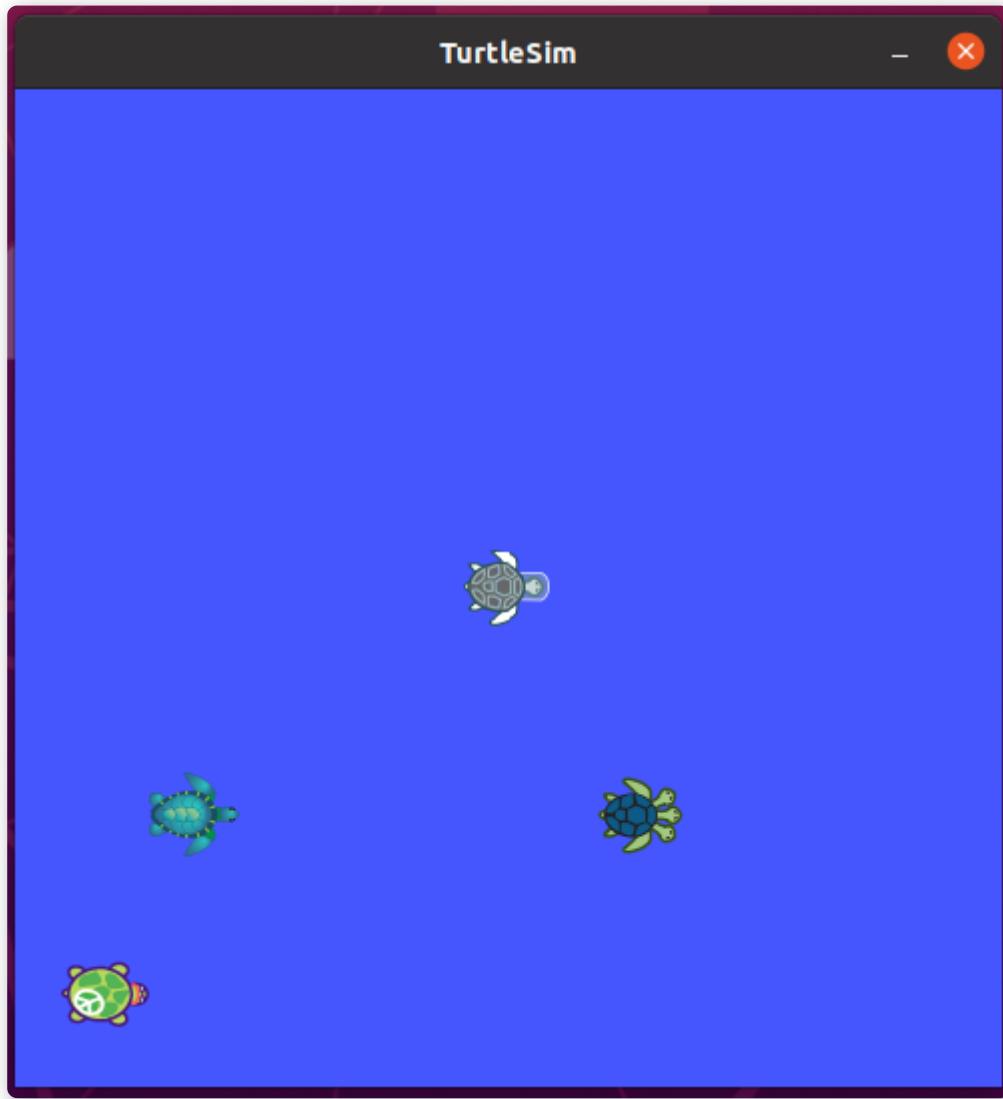
最后启动乌龟位姿订阅节点；

最终执行结果与演示结果类似。

2.5.3 实操03_服务调用

需求描述:编码实现向 turtlesim 发送请求，在乌龟显示节点的窗体指定位置生成一乌龟，这是一个服务请求操作。

结果演示:



实现分析:

1. 首先，需要启动乌龟显示节点。
2. 要通过ROS命令，来获取乌龟生成服务的服务名称以及服务消息类型。
3. 编写服务请求节点，生成新的乌龟。

实现流程:

1. 通过ros命令获取服务与服务消息信息。
2. 编码实现服务请求节点。

3. 启动 roscore、turtlesim_node、乌龟生成节点，生成新的乌龟。

1. 服务名称与服务消息获取

获取话题:/spawn

```
1 rosservice list
```

获取消息类型:turtlesim/Spawn

```
1 rosservice type /spawn
```

获取消息格式:

```
1 rossrv info turtlesim/Spawn
```

响应结果:

```
1 float32 x
2 float32 y
3 float32 theta
4 string name
5 ---
6 string name
```

2. 服务客户端实现

创建功能包需要依赖的功能包: roscpp rospy std_msgs turtlesim

实现方案A:C++

```
1 /*
2      生成一只小乌龟
3      准备工作:
4          1. 服务话题 /spawn
```

```

5      2.服务消息类型 turtlesim/Spawn
6      3.运行前先启动 turtlesim_node 节点
7

```

8 实现流程：

```

9      1.包含头文件
10     需要包含 turtlesim 包下资源，注意在
11     package.xml 配置
12     2.初始化 ros 节点
13     3.创建 ros 句柄
14     4.创建 service 客户端
15     5.等待服务启动
16     6.发送请求
17     7.处理响应
18 */
19
20 #include "ros/ros.h"
21 #include "turtlesim/Spawn.h"
22
23 int main(int argc, char *argv[])
24 {
25     setlocale(LC_ALL, "");
26     // 2.初始化 ros 节点
27     ros::init(argc, argv, "set_turtle");
28     // 3.创建 ros 句柄
29     ros::NodeHandle nh;
30     // 4.创建 service 客户端
31     ros::ServiceClient client =
32         nh.serviceClient<turtlesim::Spawn>("/spawn");
33     // 5.等待服务启动
34     // client.waitForExistence();
35     ros::service::waitForService("/spawn");
36     // 6.发送请求
37     turtlesim::Spawn spawn;
38     spawn.request.x = 1.0;
39     spawn.request.y = 1.0;
40     spawn.request.theta = 1.57;
41     spawn.request.name = "my_turtle";
42     bool flag = client.call(spawn);

```

```

42     // 7.处理响应结果
43     if (flag)
44     {
45         ROS_INFO("新的乌龟生成,名
46         字:%s", spawn.response.name.c_str());
47     } else {
48         ROS_INFO("乌龟生成失败！！！");
49     }
50
51     return 0;
52 }
```

配置文件此处略

实现方案B:Python

```

1  #! /usr/bin/env python
2 """
3     生成一只小乌龟
4     准备工作:
5         1.服务话题 /spawn
6         2.服务消息类型 turtlesim/Spawn
7         3.运行前先启动 turtlesim_node 节点
8
9     实现流程:
10         1.导包
11             需要包含 turtlesim 包下资源, 注意在
12             package.xml 配置
13         2.初始化 ros 节点
14         3.创建 service 客户端
15         4.等待服务启动
16         5.发送请求
17         6.处理响应
18 """
19
20 import rospy
```

```

21 from turtlesim.srv import
22   Spawn, SpawnRequest, SpawnResponse
23
24 if __name__ == "__main__":
25     # 2. 初始化 ros 节点
26     rospy.init_node("set_turtle_p")
27     # 3. 创建 service 客户端
28     client = rospy.ServiceProxy("/spawn", Spawn)
29     # 4. 等待服务启动
30     client.wait_for_service()
31     # 5. 发送请求
32     req = SpawnRequest()
33     req.x = 2.0
34     req.y = 2.0
35     req.theta = -1.57
36     req.name = "my_turtle_p"
37     try:
38         response = client.call(req)
39         # 6. 处理响应
40         rospy.loginfo("乌龟创建成功!",
41             "叫:%s", response.name)
42     except expression as identifier:
43         rospy.loginfo("服务调用失败")

```

权限设置以及配置文件此处略

3.运行

首先，启动 roscore；

然后启动乌龟显示节点；

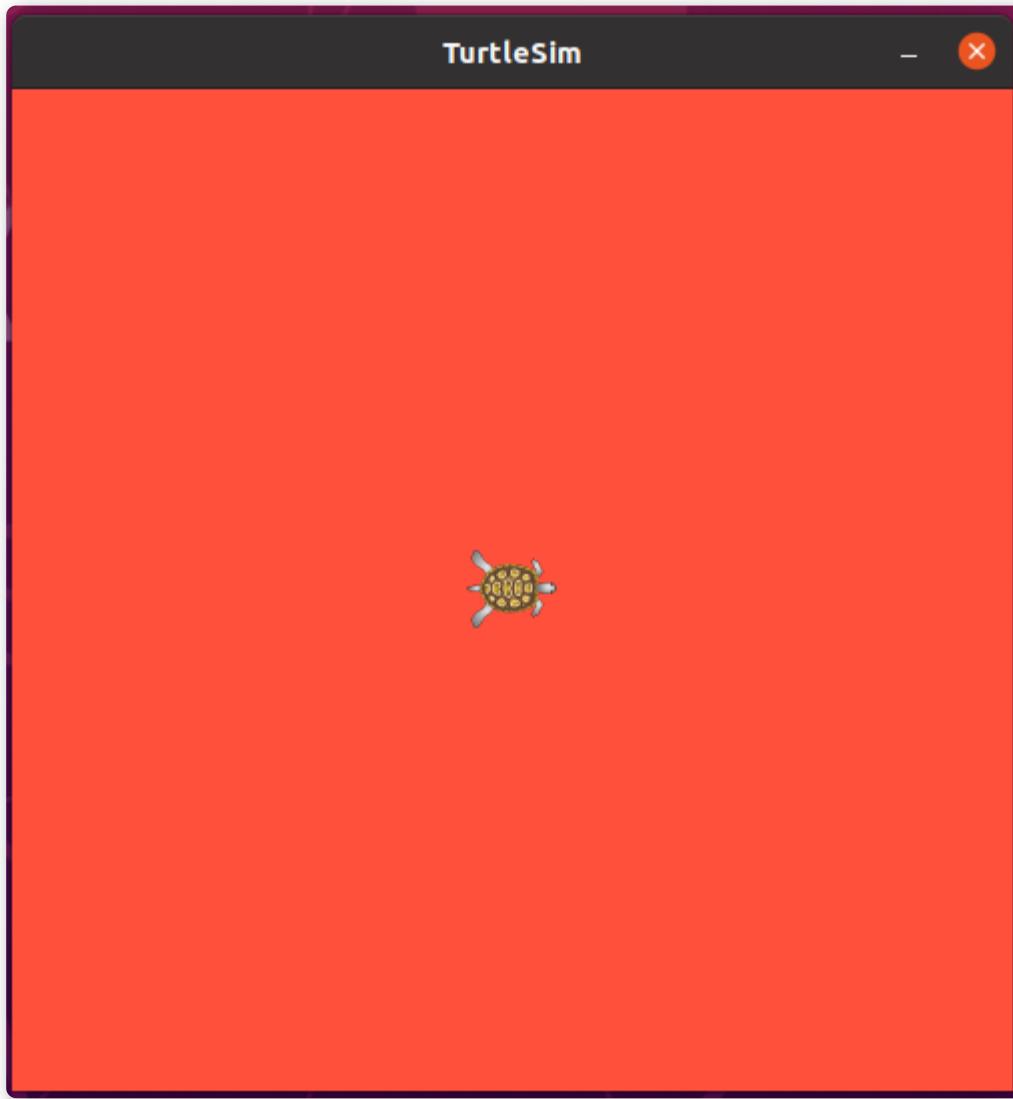
最后启动乌龟生成请求节点；

最终执行结果与演示结果类似。

2.5.4 实操04_参数设置

需求描述: 修改turtlesim乌龟显示节点窗体的背景色, 已知背景色是通过参数服务器的方式以rgb方式设置的。

结果演示:



实现分析:

1. 首先, 需要启动乌龟显示节点。
2. 要通过ROS命令, 来获取参数服务器中设置背景色的参数。
3. 编写参数设置节点, 修改参数服务器中的参数值。

实现流程:

1. 通过ros命令获取参数。
2. 编码实现参数设置节点。
3. 启动 roscore、turtlesim_node 与参数设置节点, 查看运行结果。

1.参数名获取

获取参数列表:

```
1 rosparam list
```

响应结果:

```
1 /turtlesim/background_b
2 /turtlesim/background_g
3 /turtlesim/background_r
```

2.参数修改

实现方案A:C++

```
1 /*
2      注意命名空间的使用。
3
4 */
5 #include "ros/ros.h"
6
7
8 int main(int argc, char *argv[])
9 {
10     ros::init(argc,argv,"haha");
11
12     ros::NodeHandle nh("turtlesim");
13     //ros::NodeHandle nh;
14
15     //
16     ros::param::set("/turtlesim/background_r",0);
17     //
18     ros::param::set("/turtlesim/background_g",0);
19     //
20     ros::param::set("/turtlesim/background_b",0);
```

```

19     nh.setParam("background_r",0);
20     nh.setParam("background_g",0);
21     nh.setParam("background_b",0);
22
23
24     return 0;
25 }
```

配置文件此处略

实现方案B:Python

```

1  #! /usr/bin/env python
2
3  import rospy
4
5  if __name__ == "__main__":
6      rospy.init_node("hehe")
7      #
8      rospy.set_param("/turtlesim/background_r",255)
9      #
10     rospy.set_param("/turtlesim/background_g",255)
11     #
12     rospy.set_param("/turtlesim/background_b",255)
13     rospy.set_param("background_r",255)
14     rospy.set_param("background_g",255)
15     rospy.set_param("background_b",255)  # 调用时,
16     需要传入 __ns:=xxx
```

权限设置以及配置文件此处略

3.运行

首先，启动 roscore；

然后启动背景色设置节点；

最后启动乌龟显示节点；

最终执行结果与演示结果类似。

PS: 注意节点启动顺序, 如果先启动乌龟显示节点, 后启动背景色设置节点, 那么颜色设置不会生效。

4.其他设置方式

方式1:修改小乌龟节点的背景色(命令行实现)

```
1 rosparam set /turtlesim/background_b 自定义数值
2 rosparam set /turtlesim/background_g 自定义数值
3 rosparam set /turtlesim/background_r 自定义数值
```

修改相关参数后, 重启 turtlesim_node 节点, 背景色就会发生改变了

方式2:启动节点时, 直接设置参数

```
1 rosrun turtlesim turtlesim_node _background_r:=100
 _background_g=0 _background_b=0
```

方式3:通过launch文件传参

```

1 <launch>
2   <node pkg="turtlesim" type="turtlesim_node"
3     name="set_bg" output="screen">
4       <!-- launch 传参策略1 -->
5       <param name="background_b" value="0"
6         type="int" />
7       <param name="background_g" value="0"
8         type="int" />
9       <param name="background_r" value="0"
10      type="int" /> -->
11
12 </

```

2.6 通信机制比较

三种通信机制中，参数服务器是一种数据共享机制，可以在不同的节点之间共享数据，话题通信与服务通信是在不同的节点之间传递数据的，三者是ROS中最基础也是应用最为广泛的通信机制。

这其中，话题通信和服务通信有一定的相似性也有本质上的差异，在此将二者做一下简单比较：

二者的实现流程是比较相似的，都是涉及到四个要素：

- 要素1：消息的发布方/客户端(Publisher/Client)
- 要素2：消息的订阅方/服务端(Subscriber/Server)
- 要素3：话题名称(Topic/Service)
- 要素4：数据载体(msg/srv)

可以概括为：两个节点通过话题关联到一起，并使用某种类型的数据载体实现数据传输。

二者的实现也是有本质差异的，具体比较如下：

	Topic(话题)	Service(服务)
通信模式	发布/订阅	请求/响应
同步性	异步	同步
底层协议	ROSTCP/ROSUDP	ROSTCP/ROSUDP
缓冲区	有	无
时效性	弱	强
节点关系	多对多	一对多(一个 Server)
通信数据	msg	srv
使用场景	连续高频的数据发布与接收： 雷达、里程计	偶尔调用或执行某一项特定功能： 拍照、语音识别

不同通信机制有一定的互补性，都有各自适应的应用场景。尤其是话题与服务通信，需要结合具体的应用场景与二者的差异，选择合适的通信机制。

2.7 本章小结

本章主要介绍了ROS中最基本的也是最核心的通信机制实现：话题通信、服务通信、参数服务器。每种通信机制，都介绍了如下内容：

- 伊始介绍了当前通信机制的应用场景；
- 介绍了当前通信机制的理论模型；
- 分别介绍了当前通信机制的C++与Python实现。

除此之外，还介绍了：

- ROS中的常用命令方便操作、调试节点以及通信信息；
- 通过实操又将上述知识点加以整合；
- 最后又着重比较了话题通信与服务通信的相同点以及差异。

掌握本章内容后，基本上就可以从容应对ROS中大部分应用场景了。

第3章 ROS通信机制进阶

上一章内容，主要介绍了ROS通信的实现，内容偏向于粗粒度的通信框架的讲解，没有详细介绍涉及的API，也没有封装代码，鉴于此，本章主要内容如下：

- ROS常用API介绍；
- ROS中自定义头文件与源文件的使用。

预期达成的学习目标：

- 熟练掌握ROS常用API；
- 掌握ROS中自定义头文件与源文件的配置。

3.1 常用API

首先，建议参考官方API文档或参考源码：

- ROS节点的初始化相关API；
- NodeHandle 的基本使用相关API；
- 话题的发布方，订阅方对象相关API；
- 服务的服务端，客户端对象相关API；
- 时间相关API；
- 日志输出相关API。

参数服务器相关API在第二章已经有详细介绍和应用，在此不再赘述。

另请参考：

- <http://wiki.ros.org/APIs>
- <https://docs.ros.org/en/api/roscpp/html/>

3.1.1 初始化

C++

初始化

```
1  /** @brief ROS初始化函数。  
2  *  
3  * 该函数可以解析并使用节点启动时传入的参数(通过参数设置节点  
4  * 名称、命名空间...)  
5  * 该函数有多个重载版本, 如果使用NodeHandle建议调用该版  
6  * 本。  
7  * \param argc 参数个数  
8  * \param argv 参数列表  
9  * \param name 节点名称, 需要保证其唯一性, 不允许包含命名  
空间  
10 * \param options 节点启动选项, 被封装进了  
11 * ros::init_options  
12 */  
13 void init(int &argc, char **argv, const  
std::string& name, uint32_t options = 0);
```

Python

初始化

```

1 def init_node(name, argv=None, anonymous=False,
2               log_level=None, disable_rostime=False,
3               disable_rosout=False, disable_signals=False,
4               xmlrpc_port=0, tcpros_port=0):
5     """
6     在ROS master中注册节点
7
8     @param name: 节点名称, 必须保证节点名称唯一, 节点名称
9     中不能使用命名空间(不能包含 '/')
10    @type name: str
11
12    @param anonymous: 取值为 true 时, 为节点名称后缀随
13    机编号
14    @type anonymous: bool
15    """

```

3.1.2 话题与服务相关对象

C++

在 roscpp 中, 话题和服务的相关对象一般由 `NodeHandle` 创建。

`NodeHandle` 有一个重要作用是可以用于设置命名空间, 这是后期的重点, 但是本章暂不介绍。

1.发布对象

对象获取:

```

1 /**
2 * \brief 根据话题生成发布对象
3 *
4 * 在 ROS master 注册并返回一个发布者对象, 该对象可以发布消
5 * 息
6 * 使用示例如下:

```

```

7  *
8  *   ros::Publisher pub =
9  *   handle.advertise<std_msgs::Empty>("my_topic", 1);
10 * \param topic 发布消息使用的话题
11 *
12 * \param queue_size 等待发送给订阅者的最大消息数量
13 *
14 * \param latch (optional) 如果为 true, 该话题发布的最后
15 *  一条消息将被保存, 并且后期当有订阅者连接时会将该消息发送给订
16 *  阅者
17 *
18 *
19 */
20 template <class M>
21 Publisher advertise(const std::string& topic,
22   uint32_t queue_size, bool latch = false)

```

消息发布函数:

```

1 /**
2 * 发布消息
3 */
4 template <typename M>
5 void publish(const M& message) const

```

2.订阅对象

对象获取:

```

1 /**
2  * \brief 生成某个话题的订阅对象
3  *
4  * 该函数将根据给定的话题在ROS master 注册, 并自动连接
5  * 相同主题的发布方, 每接收到一条消息, 都会调用回调

```

```
5     * 函数，并且传入该消息的共享指针，该消息不能被修改，因为
6     *
7     * 使用示例如下：
8
9 void callback(const std_msgs::Empty::ConstPtr&
10    message)
11 {
12
13 ros::Subscriber sub = handle.subscribe("my_topic",
14    1, callback);
15
16 * \param M [template] M 是指消息类型
17 * \param topic 订阅的话题
18 * \param queue_size 消息队列长度，超出长度时，头部的消息
19 将被弃用
20 * \param fp 当订阅到一条消息时，需要执行的回调函数
21 * \return 调用成功时，返回一个订阅者对象，失败时，返回空对
22
23 void callback(const std_msgs::Empty::ConstPtr&
24    message){...}
25 ros::NodeHandle nodeHandle;
26 ros::Subscriber sub =
27     nodeHandle.subscribe("my_topic", 1, callback);
28 if (sub) // Enter if subscriber is valid
29 {
30 ...
31 }
32 template<class M>
```

```

33 Subscriber subscribe(const std::string& topic,
  uint32_t queue_size, void(*fp)(const
  boost::shared_ptr<M const>&), const
  TransportHints& transport_hints =
  TransportHints())

```

3.服务对象

对象获取:

```

1 /**
2 * \brief 生成服务端对象
3 *
4 * 该函数可以连接到 ROS master，并提供一个具有给定名称的服
务对象。
5 *
6 * 使用示例如下：
7 \verbatim
8 bool callback(std_srvs::Empty& request,
  std_srvs::Empty& response)
9 {
10 return true;
11 }
12
13 ros::ServiceServer service =
  handle.advertiseService("my_service", callback);
14 \endverbatim
15 *
16 * \param service 服务的主题名称
17 * \param srv_func 接收到请求时，需要处理请求的回调函数
18 * \return 请求成功时返回服务对象，否则返回空对象：
19 \verbatim
20 bool Foo::callback(std_srvs::Empty& request,
  std_srvs::Empty& response)
21 {
22 return true;
23 }
24 ros::NodeHandle nodeHandle;

```

```

25 Foo foo_object;
26 ros::ServiceServer service =
  nodeHandle.advertiseService("my_service",
  callback);
27 if (service) // Enter if advertised service is
  valid
28 {
29 ...
30 }
31 \endverbatim
32
33 */
34 template<class MReq, class MRes>
35 ServiceServer advertiseService(const std::string&
  service, bool(*srv_func)(MReq&, MRes&))

```

4.客户端对象

对象获取:

```

1 /**
2  * @brief 创建一个服务客户端对象
3  *
4  * @param service_name 服务主题名称
5  *
6  * @param persistent = false,
7  *          const M_string&
8  *          header_values = M_string()
9
10

```

请求发送函数:

```

1  /**
2   * @brief 发送请求
3   * 返回值为 bool 类型, true, 请求处理成功, false, 处理失败。
4   */
5  template<class Service>
6  bool call(Service& service)

```

等待服务函数1:

```

1  /**
2   * ros::service::waitForService("addInts");
3   * \brief 等待服务可用, 否则一致处于阻塞状态
4   * \param service_name 被"等待"的服务的话题名称
5   * \param timeout 等待最大时常, 默认为 -1, 可以永久等待直至节点关闭
6   * \return 成功返回 true, 否则返回 false。
7   */
8  ROSCPP_DECL bool waitForService(const std::string& service_name, ros::Duration timeout =
  ros::Duration(-1));

```

等待服务函数2:

```

1  /**
2   * client.waitForExistence();
3   * \brief 等待服务可用, 否则一致处于阻塞状态
4   * \param timeout 等待最大时常, 默认为 -1, 可以永久等待直至节点关闭
5   * \return 成功返回 true, 否则返回 false。
6   */
7  bool waitForExistence(ros::Duration timeout =
  ros::Duration(-1));

```

Python

1.发布对象

对象获取:

```
1 class Publisher(Topic):
2     """
3     在ROS master注册为相关话题的发布方
4     """
5
6     def __init__(self, name, data_class,
7                  subscriber_listener=None, tcp_nodelay=False,
8                  latch=False, headers=None, queue_size=None):
9         """
10        Constructor
11        @param name: 话题名称
12        @type  name: str
13        @param data_class: 消息类型
14
15        @param latch: 如果为 true, 该话题发布的最后一条
16        消息将被保存, 并且后期当有订阅者连接时会将该消息发送给订阅者
17        @type  latch: bool
18
19        @param queue_size: 等待发送给订阅者的最大消息数
20        @type  queue_size: int
21
22        """
23
```

消息发布函数:

```
1 def publish(self, *args, **kwds):  
2     """  
3     发布消息  
4     """
```

2. 订阅对象

对象获取:

```
1 class Subscriber(Topic):  
2     """  
3     类注册为指定主题的订阅者，其中消息是给定类型的。  
4     """  
5     def __init__(self, name, data_class,  
6                  callback=None, callback_args=None,  
7                  queue_size=None,  
8                  buff_size=DEFAULT_BUFF_SIZE, tcp_nodelay=False):  
9         """  
10        Constructor.  
11  
12        @param name: 话题名称  
13        @type name: str  
14        @param data_class: 消息类型  
15        @type data_class: L{Message} class  
16        @param callback: 处理订阅到的消息的回调函数  
17        @type callback: fn(msg, cb_args)  
18  
19        @param queue_size: 消息队列长度，超出长度时，头  
20        部的消息将被弃用  
21        """
```

3. 服务对象

对象获取:

```
1 class Service(ServiceImpl):  
2     """  
3     声明一个ROS服务  
4  
5     使用示例::  
6         s = Service('getmapservice', GetMap,  
7         get_map_handler)
```

```
7      """
8
9      def __init__(self, name, service_class,
10                 handler,
11                 buff_size=DEFAULT_BUFF_SIZE,
12                 error_handler=None):
13
14      """
15
16      @param name: 服务主题名称 ``str``
17      @param service_class: 服务消息类型
18
19      @param handler: 回调函数，处理请求数据，并返回响应数据
20
21      @type handler: fn(req)->resp
22
23      """
24
```

4. 客户端对象

对象获取:

```
1 class ServiceProxy(_Service):
2     """
3     创建一个ROS服务的句柄
4
5     示例用法::
6         add_two_ints = ServiceProxy('add_two_ints',
7             AddTwoInts)
8         resp = add_two_ints(1, 2)
9
10    def __init__(self, name, service_class,
11                 persistent=False, headers=None):
12        """
13        ctor.
14        @param name: 服务主题名称
15        @type name: str
```

```

15     @param service_class: 服务消息类型
16     @type  service_class: Service class
17     """

```

请求发送函数:

```

1 def call(self, *args, **kwds):
2     """
3     发送请求, 返回值为响应数据
4
5     """
6

```

等待服务函数:

```

1 def wait_for_service(service, timeout=None):
2     """
3     调用该函数时, 程序会处于阻塞状态直到服务可用
4     @param service: 被等待的服务话题名称
5     @type  service: str
6     @param timeout: 超时时间
7     @type  timeout: double|rospy.Duration
8     """

```

3.1.3 回旋函数

C++

在ROS程序中，频繁的使用了 `ros::spin()` 和 `ros::spinOnce()` 两个回旋函数，可以用于处理回调函数。

1. `spinOnce()`

```

1  /**
2  * \brief 处理一轮回调
3  *
4  * 一般应用场景：
5  *      在循环体内，处理所有可用的回调函数
6  *
7  */
8 ROSCPP_DECL void spinOnce();

```

2.spin()

```

1 /**
2 * \brief 进入循环处理回调
3 */
4 ROSCPP_DECL void spin();

```

3.二者比较

相同点:二者都用于处理回调函数；

不同点:ros::spin() 是进入了循环执行回调函数，而 ros::spinOnce() 只会执行一次回调函数(没有循环)，在 ros::spin() 后的语句不会执行到，而 ros::spinOnce() 后的语句可以执行。

Python

```

1 def spin():
2     """
3     进入循环处理回调
4     """

```

3.1.4 时间

ROS中时间相关的API是极其常用，比如:获取当前时刻、持续时间的设置、执行频率、休眠、定时器...都与时间相关。

C++

1. 时刻

获取时刻，或是设置指定时刻：

```

1 ros::init(argc,argv,"hello_time");
2 ros::NodeHandle nh;//必须创建句柄，否则时间没有初始化，导致后续API调用失败
3 ros::Time right_now = ros::Time::now(); //将当前时刻封装成对象
4 ROS_INFO("当前时刻:%.2f",right_now.toSec()); //获取距离 1970年01月01日 00:00:00 的秒数
5 ROS_INFO("当前时刻:%d",right_now.sec); //获取距离 1970年01月01日 00:00:00 的秒数
6
7 ros::Time someTime(100,100000000); // 参数1:秒数 参数2:纳秒
8 ROS_INFO("时刻:%.2f",someTime.toSec()); //100.10
9 ros::Time someTime2(100.3); //直接传入 double 类型的秒数
10 ROS_INFO("时刻:%.2f",someTime2.toSec()); //100.30

```

2. 持续时间

设置一个时间区间(间隔)：

```

1 ROS_INFO("当前时刻:%.2f", ros::Time::now().toSec());
2 ros::Duration du(10); //持续10秒钟,参数是double类型的,以
3 秒为单位
4 du.sleep(); //按照指定的持续时间休眠
5 ROS_INFO("持续时间:%.2f", du.toSec()); //将持续时间换算成
6 秒
7 ROS_INFO("当前时刻:%.2f", ros::Time::now().toSec());

```

3.持续时间与时刻运算

为了方便使用, ROS中提供了时间与时刻的运算:

```

1 ROS_INFO("时间运算");
2 ros::Time now = ros::Time::now();
3 ros::Duration du1(10);
4 ros::Duration du2(20);
5 ROS_INFO("当前时刻:%.2f", now.toSec());
6 //1.time 与 duration 运算
7 ros::Time after_now = now + du1;
8 ros::Time before_now = now - du1;
9 ROS_INFO("当前时刻之后:%.2f", after_now.toSec());
10 ROS_INFO("当前时刻之前:%.2f", before_now.toSec());
11
12 //2.duration 之间相互运算
13 ros::Duration du3 = du1 + du2;
14 ros::Duration du4 = du1 - du2;
15 ROS_INFO("du3 = %.2f", du3.toSec());
16 ROS_INFO("du4 = %.2f", du4.toSec());
17 //PS: time 与 time 不可以运算
18 // ros::Time nn = now + before_now; //异常

```

4.设置运行频率

```

1 ros::Rate rate(1); //指定频率
2 while (true)
3 {
4     ROS_INFO("-----code-----");
5     rate.sleep(); //休眠, 休眠时间 = 1 / 频率。
6 }

```

5.定时器

ROS 中内置了专门的定时器，可以实现与 `ros::Rate` 类似的效果：

```

1 ros::NodeHandle nh; //必须创建句柄, 否则时间没有初始化,
2 导致后续API调用失败
3
4 // ROS 定时器
5 /**
6 * \brief 创建一个定时器, 按照指定频率调用回调函数。
7 * \param period 时间间隔
8 * \param callback 回调函数
9 * \param oneshot 如果设置为 true, 只执行一次回调函数, 设置
10 * 为 false, 就循环执行。
11 */
12 //Timer createTimer(Duration period, const
13 //                      TimerCallback& callback, bool oneshot = false,
14 //                      bool autostart = true) const;
15
16 // ros::Timer timer =
17 nh.createTimer(ros::Duration(0.5), doSomething);
18 ros::Timer timer =
19 nh.createTimer(ros::Duration(0.5), doSomething, true
20 ); //只执行一次
21
22 // ros::Timer timer =
23 nh.createTimer(ros::Duration(0.5), doSomething, false
24 , false); //需要手动启动

```

```
19 // timer.start();
20 ros::spin(); //必须 spin
```

定时器的回调函数:

```
1 void doSomeThing(const ros::TimerEvent &event){
2     ROS_INFO("-----");
3
4     ROS_INFO("event:%s", std::to_string(event.current_re
5 al.toSec()).c_str());
6 }
```

Python

1.时刻

获取时刻，或是设置指定时刻:

```
1 # 获取当前时刻
2 right_now = rospy.Time.now()
3 rospy.loginfo("当前时刻:%.2f", right_now.to_sec())
4 rospy.loginfo("当前时刻:%.2f", right_now.to_nsec())
5 # 自定义时刻
6 some_time1 = rospy.Time(1234.567891011)
7 some_time2 = rospy.Time(1234, 567891011)
8 rospy.loginfo("设置时刻1:%.2f", some_time1.to_sec())
9 rospy.loginfo("设置时刻2:%.2f", some_time2.to_sec())
10
11 # 从时间创建对象
12 # some_time3 = rospy.Time.from_seconds(543.21)
13 some_time3 = rospy.Time.from_sec(543.21) # 
14     from_sec 替换了 from_seconds
15 rospy.loginfo("设置时刻3:%.2f", some_time3.to_sec())
```

2.持续时间

设置一个时间区间(间隔):

```

1 # 持续时间相关API
2 rospy.loginfo("持续时间测试开始.....")
3 du = rospy.Duration(3.3)
4 rospy.loginfo("du1 持续时间:%.2f",du.to_sec())
5 rospy.sleep(du) #休眠函数
6 rospy.loginfo("持续时间测试结束.....")

```

3.持续时间与时刻运算

为了方便使用，ROS中提供了时间与时刻的运算:

```

1 rospy.loginfo("时间运算")
2 now = rospy.Time.now()
3 du1 = rospy.Duration(10)
4 du2 = rospy.Duration(20)
5 rospy.loginfo("当前时刻:%.2f",now.to_sec())
6 before_now = now - du1
7 after_now = now + du1
8 dd = du1 + du2
9 # now = now + now #非法
10 rospy.loginfo("之前时刻:%.2f",before_now.to_sec())
11 rospy.loginfo("之后时刻:%.2f",after_now.to_sec())
12 rospy.loginfo("持续时间相加:%.2f",dd.to_sec())

```

4.设置运行频率

```

1 # 设置执行频率
2 rate = rospy.Rate(0.5)
3 while not rospy.is_shutdown():
4     rate.sleep() #休眠
5     rospy.loginfo("++++++")

```

5.定时器

ROS 中内置了专门的定时器，可以实现与 `ros::Rate` 类似的效果：

```

1 #定时器设置
2 """
3 def __init__(self, period, callback,
4  oneshot=False, reset=False):
5     Constructor.
6     @param period: 回调函数的时间间隔
7     @type period: rospy.Duration
8     @param callback: 回调函数
9     @type callback: function taking
10    rospy.TimerEvent
11    @param oneshot: 设置为True, 就只执行一次, 否则循环执
12    行
13    @type oneshot: bool
14    @param reset: if True, timer is reset when
15    rostime moved backward. [default: False]
16    @type reset: bool
17 """
18
19 rospy.Timer(rospy.Duration(1),doMsg)
20 # rospy.Timer(rospy.Duration(1),doMsg,True) # 只执
21 行一次
22
23 rospy.spin()

```

回调函数：

```

1 def doMsg(event):
2     rospy.loginfo("++++++")
3     rospy.loginfo("当前时
4 刻:%s",str(event.current_real))

```

3.1.5 其他函数

在发布实现时，一般会循环发布消息，循环的判断条件一般由节点状态来控制，C++中可以通过 `ros::ok()` 来判断节点状态是否正常，而 python 中则通过 `rospy.is_shutdown()` 来实现判断，导致节点退出的原因主要有如下几种：

- 节点接收到了关闭信息，比如常用的 `ctrl + c` 快捷键就是关闭节点的信号；
- 同名节点启动，导致现有节点退出；
- 程序中的其他部分调用了节点关闭相关的API(C++中是 `ros::shutdown()`， python 中是 `rospy.signal_shutdown()`)

另外，日志相关的函数也是极其常用的，在ROS中日志被划分成如下级别：

- DEBUG(调试):只在调试时使用，此类消息不会输出到控制台；
- INFO(信息):标准消息，一般用于说明系统内正在执行的操作；
- WARN(警告):提醒一些异常情况，但程序仍然可以执行；
- ERROR(错误):提示错误信息，此类错误会影响程序运行；
- FATAL(严重错误):此类错误将阻止节点继续运行。

C++

1. 节点状态判断

```

1  /** \brief 检查节点是否已经退出
2  *
3  *  ros::shutdown() 被调用且执行完毕后，该函数将会返回
4  *  false
5  *  \return true 如果节点还健在， false 如果节点已经火化
6  *  了。
7  */
7 bool ok();

```

2. 节点关闭函数

```

1  /*
2  *   关闭节点
3  */
4  void shutdown();

```

3.日志函数

使用示例

```

1 ROS_DEBUG("hello,DEBUG"); //不会输出
2 ROS_INFO("hello,INFO"); //默认白色字体
3 ROS_WARN("Hello,WARNING"); //默认黄色字体
4 ROS_ERROR("hello,ERROR"); //默认红色字体
5 ROS_FATAL("hello,FATAL"); //默认红色字体

```

Python

1.节点状态判断

```

1 def is_shutdown():
2     """
3     @return: True 如果节点已经被关闭
4     @rtype: bool
5     """

```

2.节点关闭函数

```

1 def signal_shutdown(reason):
2     """
3     关闭节点
4     @param reason: 节点关闭的原因, 是一个字符串
5     @type reason: str
6     """
7 def on_shutdown(h):
8     """
9     节点被关闭时调用的函数
10    @param h: 关闭时调用的回调函数, 此函数无参
11    @type h: fn()
12    """

```

3.日志函数

使用示例

```

1 rospy.logdebug("hello,debug") #不会输出
2 rospy.loginfo("hello,info") #默认白色字体
3 rospy.logwarn("hello,warn") #默认黄色字体
4 rospy.logerr("hello,error") #默认红色字体
5 rospy.logfatal("hello,fatal") #默认红色字体

```

3.2 ROS中的头文件与源文件

本节主要介绍ROS的C++实现中，如何使用头文件与源文件的方式封装代码，具体内容如下：

1. 设置头文件，可执行文件作为源文件；
2. 分别设置头文件，源文件与可执行文件。

在ROS中关于头文件的使用，核心内容在于CMakeLists.txt文件的配置，不同的封装方式，配置上也有差异。

3.2.1 自定义头文件调用

需求:设计头文件，可执行文件本身作为源文件。

流程:

1. 编写头文件；
2. 编写可执行文件(同时也是源文件)；
3. 编辑配置文件并执行。

1.头文件

在功能包下的 include/功能包名 目录下新建头文件: hello.h，示例内容如下:

```

1 #ifndef _HELLO_H
2 #define _HELLO_H
3
4 namespace hello_ns{
5
6 class HelloPub {
7
8 public:
9     void run();
10 };
11
12 }
13
14 #endif

```

注意:

在 VScode 中，为了后续包含头文件时不抛出异常，请配置 .vscode 下 c_cpp_properties.json 的 includepath 属性

```
1 "/home/用户/工作空间/src/功能包/include/**"
```

2. 可执行文件

在 src 目录下新建文件:hello.cpp，示例内容如下:

```

1 #include "ros/ros.h"
2 #include "test_head/hello.h"
3
4 namespace hello_ns {
5
6 void HelloPub::run(){
7     ROS_INFO("自定义头文件的使用....");
8 }
9
10 }
11
12 int main(int argc, char *argv[])
13 {
14     setlocale(LC_ALL, "");
15     ros::init(argc, argv, "test_head_node");
16     hello_ns::HelloPub helloPub;
17     helloPub.run();
18     return 0;
19 }
```

3. 配置文件

配置CMakeLists.txt文件，头文件相关配置如下:

```

1 include_directories(
2     include
3     ${catkin_INCLUDE_DIRS}
4 )
```

可执行配置文件配置方式与之前一致:

```

1 add_executable(hello src/hello.cpp)
2
3 add_dependencies(hello
4   ${${PROJECT_NAME}_EXPORTED_TARGETS}
5   ${catkin_EXPORTED_TARGETS})
6
7 target_link_libraries(hello
8   ${catkin_LIBRARIES}
9 )

```

最后，编译并执行，控制台可以输出自定义的文本信息。

3.2.2 自定义源文件调用

需求:设计头文件与源文件，在可执行文件中包含头文件。

流程:

1. 编写头文件；
2. 编写源文件；
3. 编写可执行文件；
4. 编辑配置文件并执行。

1. 头文件

头文件设置于 3.2.1 类似，在功能包下的 `include/功能包名` 目录下新建头文件: `haha.h`，示例内容如下:

```

1 #ifndef _HAHA_H
2 #define _HAHA_H
3
4 namespace hello_ns {
5
6 class My {
7
8 public:

```

```

9     void run();
10
11 };
12
13 }
14
15 #endif

```

注意:

在 VScode 中, 为了后续包含头文件时不抛出异常, 请配置 .vscode 下 c_cpp_properties.json 的 includepath 属性

```
1 "/home/用户/工作空间/src/功能包/include/**"
```

2. 源文件

在 src 目录下新建文件:haha.cpp, 示例内容如下:

```

1 #include "test_head_src/haha.h"
2 #include "ros/ros.h"
3
4 namespace hello_ns{
5
6 void My::run(){
7     ROS_INFO("hello,head and src ...");
8 }
9
10 }

```

3. 可执行文件

在 src 目录下新建文件: use_head.cpp, 示例内容如下:

```

1 #include "ros/ros.h"
2 #include "test_head_src/haha.h"
3
4 int main(int argc, char *argv[])
5 {
6     ros::init(argc,argv,"hahah");
7     hello_ns::My my;
8     my.run();
9     return 0;
10 }
```

4.配置文件

头文件与源文件相关配置:

```

1 include_directories(
2 include
3     ${catkin_INCLUDE_DIRS}
4 )
5
6 ## 声明C++库
7 add_library(head
8     include/test_head_src/haha.h
9     src/haha.cpp
10 )
11
12 add_dependencies(head
13     ${${PROJECT_NAME}_EXPORTED_TARGETS}
14     ${catkin_EXPORTED_TARGETS})
15
16 target_link_libraries(head
17     ${catkin_LIBRARIES}
18 )
```

可执行文件配置:

```

1 add_executable(use_head src/use_head.cpp)
2
3 add_dependencies(use_head
4   ${${PROJECT_NAME}_EXPORTED_TARGETS}
5   ${catkin_EXPORTED_TARGETS})
6
7 #此处需要添加之前设置的 head 库
8 target_link_libraries(use_head
9   head
10  ${catkin_LIBRARIES})
11

```

3.3 Python模块导入

与C++类似的，在Python中导入其他模块时，也需要相关处理。

需求:首先新建一个Python文件A，再创建Python文件UseA，在UseA中导入A并调用A的实现。

实现:

1. 新建两个Python文件，使用 `import` 实现导入关系；
2. 添加可执行权限、编辑配置文件并执行UseA。

1.新建两个Python文件并使用import导入

文件A实现(包含一个变量):

```

1 #! /usr/bin/env python
2 num = 1000

```

文件B核心实现:

```
1 import os
2 import sys
3
4 path = os.path.abspath(".")
5 # 核心
6 sys.path.insert(0, path +
7                 "/src/plumbing_pub_sub/scripts")
8
9
10 .....
11 .....
12     rospy.loginfo("num = %d", tools.num)
```

2.添加可执行权限，编辑配置文件并执行

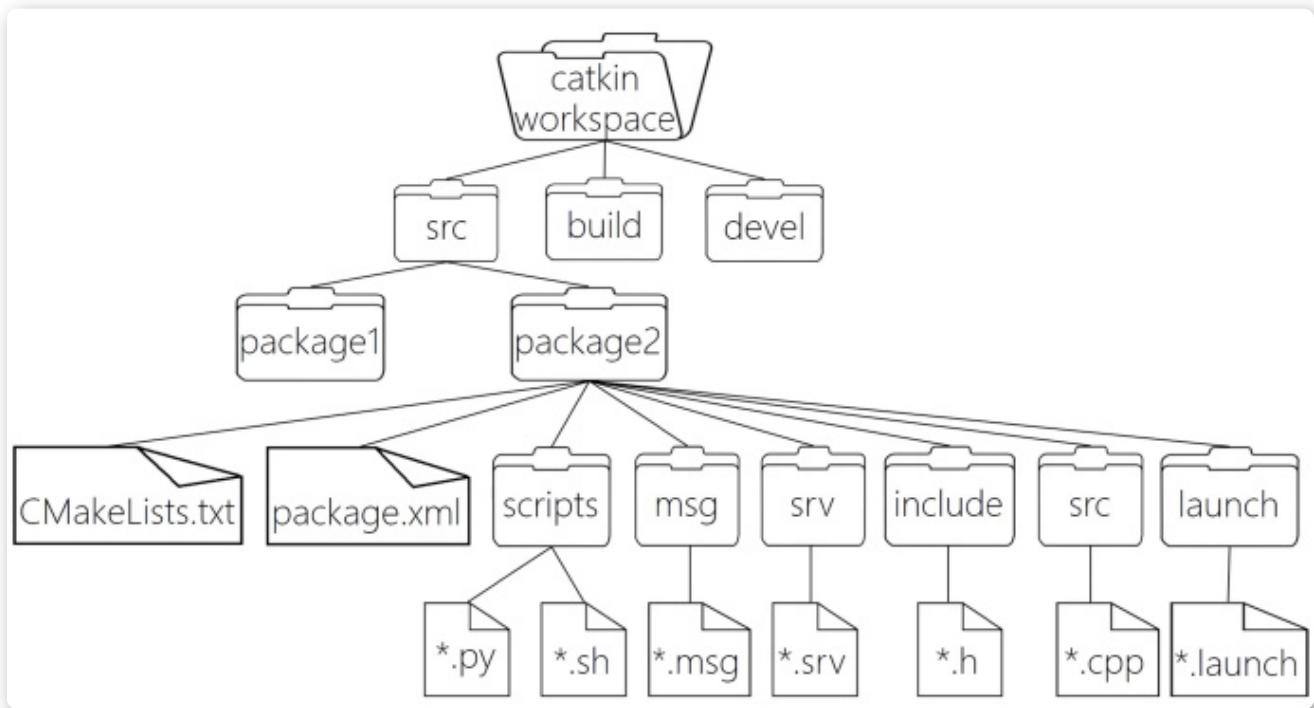
此过程略。

第4章 ROS运行管理

ROS是多进程(节点)的分布式框架，一个完整的ROS系统实现：



可能包含多台主机；
每台主机上又有多个工作空间(workspace)；
每个的工作空间中又包含多个功能包(package)；
每个功能包又包含多个节点(Node)，不同的节点都有自己的节点名称；
每个节点可能还会设置一个或多个话题(topic)...



在多级层深的ROS系统中，其实现与维护可能会出现一些问题，比如，如何关联不同的功能包，繁多的ROS节点应该如何启动？功能包、节点、话题、参数重名时应该如何处理？不同主机上的节点如何通信？

本章主要内容介绍在ROS中上述问题的解决策略(见本章目录)，预期达成学习目标也与上述问题对应：

- 掌握元功能包使用语法；
- 掌握launch文件的使用语法；
- 理解什么是ROS工作空间覆盖，以及存在什么安全隐患；
- 掌握节点名称重名时的处理方式；

- 掌握话题名称重名时的处理方式；
- 掌握参数名称重名时的处理方式；
- 能够实现ROS分布式通信。

4.1 ROS元功能包



场景:完成ROS中一个系统性的功能，可能涉及到多个功能包，比如实现了机器人导航模块，该模块下有地图、定位、路径规划...等不同的子级功能包。那么调用者安装该模块时，需要逐一的安装每一个功能包吗？

显而易见的，逐一安装功能包的效率低下，在ROS中，提供了一种方式可以将不同的功能包打包成一个功能包，当安装某个功能模块时，直接调用打包后的功能包即可，该包又称之为元功能包(metapackage)。

概念

MetaPackage是Linux的一个文件管理系统的概念。是ROS中的一个虚包，里面没有实质性的内容，但是它依赖了其他的软件包，通过这种方法可以把其他包组合起来，我们可以认为它是一本书的目录索引，告诉我们这个包集合中有哪些子包，并且该去哪里下载。

例如：

- `sudo apt install ros-noetic-desktop-full` 命令安装ros时就使用了元功能包，该元功能包依赖于ROS中的其他一些功能包，安装该包时会一并安装依赖。

还有一些常见的MetaPackage： `navigation` `moveit!` `turtlebot3`

作用

方便用户的安装，我们只需要这一个包就可以把其他相关的软件包组织到一起安装了。

实现

首先:新建一个功能包

然后:修改package.xml,内容如下:

```

1 <exec_depend>被集成的功能包</exec_depend>
2 .....
3 <export>
4   <metapackage />
5 </export>

```

最后:修改 CMakeLists.txt,内容如下:

```

1 cmake_minimum_required(VERSION 3.0.2)
2 project(demo)
3 find_package(catkin REQUIRED)
4 catkin_metapackage()

```

PS:CMakeLists.txt 中不可以有换行。

另请参考:

- <http://wiki.ros.org/catkin/package.xml#Metapackages>

4.2 ROS节点运行管理launch文件

关于 launch 文件的使用我们已经不陌生了，在第一章内容中，就曾经介绍到:



一个程序中可能需要启动多个节点，比如:ROS 内置的小乌龟案例，如果要控制乌龟运动，要启动多个窗口，分别启动 roscore、乌龟界面节点、键盘控制节点。如果每次都调用 rosrun 逐一启动，显然效率低下，如何优化？

采用的优化策略便是使用roslaunch 命令集合 launch 文件启动管理节点，并且在后续教程中，也多次使用到了 launch 文件。

概念

launch 文件是一个 XML 格式的文件，可以启动本地和远程的多个节点，还可以在参数服务器中设置参数。

作用

简化节点的配置与启动，提高ROS程序的启动效率。

使用

以 turtlesim 为例演示

1.新建launch文件

在功能包下添加 launch 目录，目录下新建 xxxx.launch 文件，编辑 launch 文件

```
1 <launch>
2   <node pkg="turtlesim" type="turtlesim_node"
3     name="myTurtle" output="screen" />
4   <node pkg="turtlesim" type="turtle_teleop_key"
5     name="myTurtleContro" output="screen" />
6 </launch>
```

2.调用 launch 文件

```
1 roslaunch 包名 xxx.launch
```

注意:roslaunch 命令执行launch文件时，首先会判断是否启动了 roscore,如果启动了，则不再启动，否则，会自动调用 roscore

PS:本节主要介绍launch文件的使用语法，launch文件中的标签，以及不同标签的一些常用属性。

另请参考：

- <http://wiki.ros.org/roslaunch/XML>

4.2.1 launch文件标签之launch

`<launch>` 标签是所有 launch 文件的根标签，充当其他标签的容器

1. 属性

- `deprecated = "弃用声明"`

告知用户当前 launch 文件已经弃用

2. 子级标签

所有其它标签都是launch的子级

4.2.2 launch文件标签之node

`<node>` 标签用于指定 ROS 节点，是最常见的标签，需要注意的是：roslaunch 命令不能保证按照 node 的声明顺序来启动节点(节点的启动是多进程的)

1. 属性

- `pkg="包名"`
节点所属的包
- `type="nodeType"`
节点类型(与之相同名称的可执行文件)

- `name="nodeName"`
节点名称(在 ROS 网络拓扑中节点的名称)
- `args="xxx xxx xxx"` (可选)
将参数传递给节点
- `machine="机器名"`
在指定机器上启动节点
- `respawn="true | false"` (可选)
如果节点退出, 是否自动重启
- `respawn_delay=" N"` (可选)
如果 `respawn` 为 `true`, 那么延迟 `N` 秒后启动节点
- `required="true | false"` (可选)
该节点是否必须, 如果为 `true`, 那么如果该节点退出, 将杀死整个 `roslaunch`
- `ns="xxx"` (可选)
在指定命名空间 `xxx` 中启动节点
- `clear_params="true | false"` (可选)
在启动前, 删除节点的私有空间的所有参数
- `output="log | screen"` (可选)
日志发送目标, 可以设置为 `log` 日志文件, 或 `screen` 屏幕, 默认是 `log`

2. 子级标签

- `env` 环境变量设置
- `remap` 重映射节点名称
- `rosparam` 参数设置
- `param` 参数设置

4.2.3 launch文件标签之include

include 标签用于将另一个 xml 格式的 launch 文件导入到当前文件

1. 属性

- file="\$(find 包名)/xxx/xxx.launch"

要包含的文件路径

- ns="xxx" (可选)

在指定命名空间导入文件

2. 子级标签

- env 环境变量设置

- arg 将参数传递给被包含的文件

4.2.4 launch文件标签之remap

用于话题重命名

1. 属性

- from="xxx"

原始话题名称

- to="yyy"

目标名称

2. 子级标签

- 无

4.2.5 launch文件标签之param

`<param>` 标签主要用于在参数服务器上设置参数，参数源可以在标签中通过 `value` 指定，也可以通过外部文件加载，在 `<node>` 标签中时，相当于私有命名空间。

1. 属性

- `name="命名空间/参数名"`
参数名称，可以包含命名空间
- `value="xxx"` (可选)
定义参数值，如果此处省略，必须指定外部文件作为参数源
- `type="str | int | double | bool | yaml"` (可选)
指定参数类型，如果未指定，`roslaunch` 会尝试确定参数类型，规则如下：
 - 如果包含 '!' 的数字解析为浮点型，否则为整型
 - "true" 和 "false" 是 `bool` 值(不区分大小写)
 - 其他是字符串

2. 子级标签

- 无

4.2.6 launch文件标签之rosparam

`<rosparam>` 标签可以从 YAML 文件导入参数，或将参数导出到 YAML 文件，也可以用来删除参数，`<rosparam>` 标签在 `<node>` 标签中时被视为私有。

1. 属性

- command="load | dump | delete" (可选, 默认 load)
加载、导出或删除参数
- file="\$(find xxxx)/xxx/yyy...."
加载或导出到的 yaml 文件
- param="参数名称"
- ns="命名空间" (可选)

2. 子级标签

- 无

4.2.7 launch文件标签之group

`<group>` 标签可以对节点分组, 具有 ns 属性, 可以让节点归属某个命名空间

1. 属性

- ns="名称空间" (可选)
- clear_params="true | false" (可选)

启动前, 是否删除组名称空间的所有参数(慎用....此功能危险)

2. 子级标签

- 除了launch 标签外的其他标签

4.2.8 launch文件标签之arg

`<arg>` 标签是用于动态传参, 类似于函数的参数, 可以增强launch文件的灵活性

1. 属性

- name="参数名称"
 - default="默认值" (可选)
 - value="数值" (可选)
不可以与 default 并存
 - doc="描述"
- 参数说明

2. 子级标签

- 无

3. 示例

- launch文件传参语法实现,hello.launch

```

1 <launch>
2   <arg name="xxx" />
3   <param name="param" value="$(arg xxx)" />
4 </launch>

```

命令行调用launch传参

```
1 roslaunch hello.launch xxx:=值
```

4.3 ROS工作空间覆盖

所谓工作空间覆盖，是指不同工作空间中，存在重名的功能包的情形。



ROS 开发中，会自定义工作空间且自定义工作空间可以同时存在多个，可能会出现一种情况：虽然特定工作空间内的功能包不能重名，但是自定义工作空间的功能包与内置的功能包可以重名或者不同的自定义的工作空间中也可以出现重名的功能包，那么调用该名称功能包时，会调用哪一个呢？比如：自定义工作空间A存在功能包 turtlesim，自定义工作空间B也存在功能包 turtlesim，当然系统内置空间也存在turtlesim，如果调用turtlesim包，会调用哪个工作空间中的呢？

实现

0.新建工作空间A与工作空间B，两个工作空间中都创建功能包:turtlesim。

1.在 `~/.bashrc` 文件下追加当前工作空间的 bash 格式如下：

```
1 source /home/用户/路径/工作空间A/devel/setup.bash
2 source /home/用户/路径/工作空间B/devel/setup.bash
```

2.新开命令行: `source .bashrc` 加载环境变量

3.查看ROS环境环境变量 `echo $ROS_PACKAGE_PATH`

结果:自定义工作空间B:自定义空间A:系统内置空间

4.调用命令: `roscd turtlesim` 会进入自定义工作空间B

原因

ROS 会解析 `.bashrc` 文件，并生成 `ROS_PACKAGE_PATH` ROS包路径，该变量中按照 `.bashrc` 中配置设置工作空间优先级，在设置时需要遵循一定的原则: `ROS_PACKAGE_PATH` 中的值，和 `.bashrc` 的配置顺序相反-->后配置的优先级更高，如果更改自定义空间A与自定义空间B的 `source` 顺序，那么调用时，将进入工作空间A。

结论

功能包重名时，会按照 `ROS_PACKAGE_PATH` 查找，配置在前的会优先执行。

隐患

存在安全隐患，比如当前工作空间B优先级更高，意味着当程序调用 `turtlesim` 时，不会调用工作空间A也不会调用系统内置的 `turtlesim`，如果工作空间A在实现时有其他功能包依赖于自身的 `turtlesim`，而按照ROS工作空间覆盖的涉及原则，那么实际执行时将会调用工作空间B的 `turtlesim`，从而导致执行异常，出现安全隐患。

BUG 说明:



当在 `.bashrc` 文件中 `source` 多个工作空间后，可能出现的情况，在 `ROS PACKAGE PATH` 中只包含两个工作空间，可以删除自定义工作空间的 `build` 与 `devel` 目录，重新 `catkin_make`，然后重新载入 `.bashrc` 文件，问题解决。

4.4 ROS节点名称重名



场景:ROS 中创建的节点是有名称的，C++初始化节点时通过 API: `ros::init(argc, argv, "xxxx")` 来定义节点名称，在Python 中初始化节点则通过 `rospy.init_node("yyyy")` 来定义节点名称。在ROS的网络拓扑中，是不可以出现重名的节点的，因为假设可以重名存在，那么调用时会产生混淆，这也就意味着，不可以启动重名节点或者同一个节点启动多次，的确，在ROS中如果启动重名节点的话，之前已经存在的节点会被直接关闭，但是如果有这种需求的话，怎么优化呢？

在ROS中给出的解决策略是使用命名空间或名称重映射。

命名空间就是为名称添加前缀，名称重映射是为名称起别名。这两种策略都可以解决节点重名问题，两种策略的实现途径有多种：

- `rosrun` 命令
- `launch` 文件
- 编码实现

以上三种途径都可以通过命名空间或名称重映射的方式，来避免节点重名，本节将对三者的使用逐一演示，三者要实现的需求类似。

案例

启动两个 `turtlesim_node` 节点，当然如果直接打开两个终端，直接启动，那么第一次启动的节点会关闭，并给出提示：

```
1 [ WARN] [1578812836.351049332]: Shutdown request
  received.
2 [ WARN] [1578812836.351207362]: Reason given for
  shutdown: [new node registered with same name]
```

因为两个节点不能重名，接下来将会介绍解决重名问题的多种方案。

4.4.1 rosrun设置命名空间与重映射

1.rosrun设置命名空间

1.1设置命名空间演示

语法: `rosrun` 包名 节点名 `__ns:=`新名称

```
1 rosrun turtlesim turtlesim_node __ns:=/xxx
2 rosrun turtlesim turtlesim_node __ns:=/yyy
```

两个节点都可以正常运行

1.2运行结果

`rosnode list` 查看节点信息,显示结果:

```
1 /xxx/turtlesim
2 /yyy/turtlesim
```

2.rosrun名称重映射

2.1为节点起别名

语法: `rosrun 包名 节点名 __name:=新名称`

```
1 rosrun turtlesim turtlesim_node __name:=t1 |
  rosrun turtlesim turtlesim_node /turtlesim:=t1(不
  适用于python)
2 rosrun turtlesim turtlesim_node __name:=t2 |
  rosrun turtlesim turtlesim_node /turtlesim:=t2(不
  适用于python)
```

两个节点都可以运行

2.2运行结果

`rosnode list` 查看节点信息,显示结果:

```
1 /t1
2 /t2
```

3.rosrun命名空间与名称重映射叠加

3.1设置命名空间同时名称重映射

语法: `rosrun 包名 节点名 ns:=新名称 name:=新名称`

```
1 rosrun turtlesim turtlesim_node __ns:=/xxx
  __name:=tn
```

3.2运行结果

`rosnode list` 查看节点信息,显示结果:

```
1 /xxx/tn
```



使用环境变量也可以设置命名空间,启动节点前在终端键入如下命令:

```
export ROS_NAMESPACE=xxxx
```

4.4.2 launch文件设置命名空间与重映射

介绍 launch 文件的使用语法时, 在 node 标签中有两个属性: name 和 ns, 二者分别是用于实现名称重映射与命名空间设置的。使用 launch 文件设置命名空间与名称重映射也比较简单。

1.launch文件

```
1 <launch>
2
3     <node pkg="turtlesim" type="turtlesim_node"
4       name="t1" />
5     <node pkg="turtlesim" type="turtlesim_node"
6       name="t2" />
7     <node pkg="turtlesim" type="turtlesim_node"
8       name="t1" ns="hello"/>
9
10 </launch>
```

在 node 标签中, name 属性是必须的, ns 可选。

2.运行

`rosnode list` 查看节点信息,显示结果:

```

1 /t1
2 /t2
3 /t1/hello

```

4.4.3 编码设置命名空间与重映射

如果自定义节点实现，那么可以更灵活的设置命名空间与重映射实现。

1.C++ 实现:重映射

1.1名称别名设置

核心代码

码: `ros::init(argc, argv, "zhangsan", ros::init_options::AnonymousName);`

1.2执行

会在名称后面添加时间戳。

2.C++ 实现:命名空间

2.1命名空间设置

核心代码

```

1 std::map<std::string, std::string> map;
2 map["__ns"] = "xxxx";
3 ros::init(map, "wangqiang");

```

2.2执行

节点名称设置了命名空间。

3. Python 实现: 重映射

3.1 名称别名设置

核心代码: `rospy.init_node("lisi", anonymous=True)`

3.2 执行

会在节点名称后缀时间戳。

4.5 ROS话题名称设置

在ROS中节点名称可能出现重名的情况，同理话题名称也可能重名。



在ROS中节点终端，不同的节点之间通信都依赖于话题，话题名称也可能出现重复的情况，这种情况下，系统虽然不会抛出异常，但是可能导致订阅的消息非预期的，从而导致节点运行异常。这种情况下需要将两个节点的话题名称由相同修改为不同。

又或者，两个节点是可以通信的，两个节点之间使用了相同的消息类型，但是由于，话题名称不同，导致通信失败。这种情况下需要将两个节点的话题名称由不同修改为相同。

在实际应用中，按照逻辑，有些时候可能需要将相同的话题名称设置为不同，也有可能将不同的话题名设置为相同。在ROS中给出的解决策略与节点名称重命名类似，也是使用名称重映射或为名称添加前缀。根据前缀不同，有全局、相对、和私有三种类型之分。

- 全局(参数名称直接参考ROS系统，与节点命名空间平级)
- 相对(参数名称参考的是节点的命名空间，与节点名称平级)
- 私有(参数名称参考节点名称，是节点名称的子级)

名称重映射是为名称起别名，为名称添加前缀，该实现比节点重名更复杂些，不单是使用命名空间作为前缀、还可以使用节点名称最为前缀。两种策略的实现途径有多种：

- `rosrun` 命令
- `launch` 文件
- 编码实现

本节将对三者的使用逐一演示，三者要实现的需求类似。

案例

在ROS中提供了一个比较好用的键盘控制功能包: `ros-noetic-teleop-twist-keyboard`，该功能包，可以控制机器人的运动，作用类似于乌龟的键盘控制节点，可以使用 `sudo apt install ros-noetic-teleop-twist-keyboard` 来安装该功能包，然后执行: `rosrun teleop_twist_keyboard teleop_twist_keyboard.py`，在启动乌龟显示节点，不过此时前者不能控制乌龟运动，因为，二者使用的话题名称不同，前者使用的是 `cmd_vel` 话题，后者使用的是 `/turtle1/cmd_vel` 话题。需要将话题名称修改为一致，才能使用，如何实现？

4.5.1 rosrun设置话题重映射

rosrun名称重映射语法: `rorun` 包名 节点名 话题名:=新话题名称

实现`teleop_twist_keyboard`与乌龟显示节点通信方案由两种：

1. 方案1

将 `teleop_twist_keyboard` 节点的话题设置为 `/turtle1/cmd_vel`

启动键盘控制节点: `rosrun teleop_twist_keyboard teleop_twist_keyboard.py /cmd_vel:=/turtle1/cmd_vel`

启动乌龟显示节点: `rosrun turtlesim turtlesim_node`

二者可以实现正常通信

2.方案2

将乌龟显示节点的话题设置为 `/cmd_vel`

启动键盘控制节点: `rosrun teleop_twist_keyboard teleop_twist_keyboard.py`

启动乌龟显示节点: `rosrun turtlesim turtlesim_node /turtle1/cmd_vel:=/cmd_vel`

二者可以实现正常通信

4.5.2 launch文件设置话题重映射

launch 文件设置话题重映射语法:

```
1 <node pkg="xxx" type="xxx" name="xxx">
2   <remap from="原话题" to="新话题" />
3 </node>
```

实现`teleop_twist_keyboard`与乌龟显示节点通信方案由两种:

1.方案1

将 `teleop_twist_keyboard` 节点的话题设置为 `/turtle1/cmd_vel`

```

1 <launch>
2
3     <node pkg="turtlesim" type="turtlesim_node"
4         name="t1" />
5     <node pkg="teleop_twist_keyboard"
6         type="teleop_twist_keyboard.py" name="key">
7         <remap from="/cmd_vel" to="/turtle1/cmd_vel" />
8     </node>
9
10 </launch>

```

二者可以实现正常通信

2. 方案2

将乌龟显示节点的话题设置为 `/cmd_vel`

```

1 <launch>
2     <node pkg="turtlesim" type="turtlesim_node"
3         name="t1">
4         <remap from="/turtle1/cmd_vel" to="/cmd_vel" />
5     </node>
6     <node pkg="teleop_twist_keyboard"
7         type="teleop_twist_keyboard.py" name="key" />
8
9 </launch>

```

二者可以实现正常通信

4.5.3 编码设置话题名称

话题的名称与节点的命名空间、节点的名称是有一定关系的，话题名称大致可以分为三种类型：

- 全局(话题参考ROS系统，与节点命名空间平级)
- 相对(话题参考的是节点的命名空间，与节点名称平级)

- 私有(话题参考节点名称, 是节点名称的子级)

结合编码演示具体关系。

1.C++ 实现

演示准备:

1. 初始化节点设置一个节点名称

```
1 ros::init(argc, argv, "hello")
```

2. 设置不同类型的话题

3. 启动节点时, 传递一个 __ns:=xxx

4. 节点启动后, 使用 rostopic 查看话题信息

1.1 全局名称

格式: 以 / 开头的名称, 和节点名称无关

比如: /xxx/yyy/zzz

示例1: ros::Publisher pub =
nh.advertise<std_msgs::String>("/chatter", 1000);

结果1: /chatter

示例2: ros::Publisher pub =
nh.advertise<std_msgs::String>("/chatter/money", 1000);

结果2: /chatter/money

1.2 相对名称

格式:非 / 开头的名称,参考命名空间(与节点名称平级)来确定话题名称

示例1: ros::Publisher pub =
nh.advertise<std_msgs::String>("chatter", 1000);

结果1: xxx/chatter

示例2: ros::Publisher pub =
nh.advertise<std_msgs::String>("chatter/money", 1000);

结果2: xxx/chatter/money

1.3 私有名称

格式:以 ~ 开头的名称

示例1:

```
1 ros::NodeHandle nh("~");
2 ros::Publisher pub = nh.advertise<std_msgs::String>
  ("chatter", 1000);
```

结果1: /xxx/hello/chatter

示例2:

```
1 ros::NodeHandle nh("~");
2 ros::Publisher pub = nh.advertise<std_msgs::String>
  ("chatter/money", 1000);
```

结果2: /xxx/hello/chatter/money

PS:当使用 ~ ,而话题名称有时 / 开头时,那么话题名称是绝对的

示例3:

```
1 ros::NodeHandle nh("~");
2 ros::Publisher pub = nh.advertise<std_msgs::String>
  ("/chatter/money", 1000);
```

结果3: /chatter/money

2. Python 实现

演示准备:

1. 初始化节点设置一个节点名称

```
1 rospy.init_node("hello")
```

2. 设置不同类型的话题

3. 启动节点时, 传递一个 __ns:=xxx

4. 节点启动后, 使用 rostopic 查看话题信息

2.1 全局名称

格式: 以 / 开头的名称, 和节点名称无关

示例1: pub =

```
rospy.Publisher("/chatter", String, queue_size=1000)
```

结果1: /chatter

示例2: pub =

```
rospy.Publisher("/chatter/money", String, queue_size=1000)
```

结果2: /chatter/money

2.2 相对名称

格式:非 / 开头的名称,参考命名空间(与节点名称平级)来确定话题名称

示例1: pub =

```
rospy.Publisher("chatter",String,queue_size=1000)
```

结果1: xxx/chatter

示例2: pub =

```
rospy.Publisher("chatter/money",String,queue_size=1000)
```

结果2: xxx/chatter/money

2.3 私有名称

格式:以 ~ 开头的名称

示例1: pub =

```
rospy.Publisher("~chatter",String,queue_size=1000)
```

结果1: /xxx/hello/chatter

示例2: pub =

```
rospy.Publisher("~chatter/money",String,queue_size=1000)
```

结果2: /xxx/hello/chatter/money

4.6 ROS参数名称设置

在ROS中节点名称话题名称可能出现重名的情况，同理参数名称也可能重名。



当参数名称重名时，那么就会产生覆盖，如何避免这种情况？

关于参数重名的处理，没有重映射实现，为了尽量的避免参数重名，都是使用为参数名添加前缀的方式，实现类似于话题名称，有全局、相对、和私有三种类型之分。

- 全局(参数名称直接参考ROS系统，与节点命名空间平级)
 - 相对(参数名称参考的是节点的命名空间，与节点名称平级)
 - 私有(参数名称参考节点名称，是节点名称的子级)
-

设置参数的方式也有三种：

- `rosrun` 命令
- `launch` 文件
- 编码实现

三种设置方式前面都已经有所涉及，但是之前没有涉及命名问题，本节将对三者命名的设置逐一演示。

案例

启动节点时，为参数服务器添加参数(需要注意参数名称设置)。

4.6.1 rosrun设置参数

`rosrun` 在启动节点时，也可以设置参数：

语法: `rosrun 包名 节点名称 _参数名:=参数值`

1. 设置参数

启动乌龟显示节点，并设置参数 `A = 100`

```
1 rosrun turtlesim turtlesim_node _A:=100
```

2. 运行

`rosparam list` 查看节点信息, 显示结果:

```
1 /turtlesim/A
2 /turtlesim/background_b
3 /turtlesim/background_g
4 /turtlesim/background_r
```

结果显示, 参数A前缀节点名称, 也就是说rosrun执行设置参数参数名使用的是私有模式

4.6.2 launch文件设置参数

通过 launch 文件设置参数的方式前面已经介绍过了, 可以在 node 标签外, 或 node 标签中通过 param 或 rosparam 来设置参数。在 node 标签外设置的参数是全局性质的, 参考的是 /, 在 node 标签中设置的参数是私有性质的, 参考的是 /命名空间/节点名称。

1. 设置参数

以 param 标签为例, 设置参数

```
1 <launch>
2
3     <param name="p1" value="100" />
4     <node pkg="turtlesim" type="turtlesim_node"
5       name="t1">
6         <param name="p2" value="100" />
7     </node>
8 </launch>
```

2.运行

`rosparam list` 查看节点信息,显示结果:

```
1 /p1
2 /t1/p1
```

运行结果与预期一致。

4.6.3 编码设置参数

编码的方式可以更方便的设置:全局、相对与私有参数。

1.C++实现

在 C++ 中, 可以使用 `ros::param` 或者 `ros::NodeHandle` 来设置参数。

1.1ros::param设置参数

设置参数调用API是`ros::param::set`, 该函数中, 参数1传入参数名称, 参数2是传入参数值, 参数1中参数名称设置时, 如果以 / 开头, 那么就是全局参数, 如果以 ~ 开头, 那么就是私有参数, 既不以 / 也不以 ~ 开头, 那么就是相对参数。代码示例:

```
1 ros::param::set("/set_A", 100); //全局,和命名空间以及节点名称无关
2 ros::param::set("set_B", 100); //相对,参考命名空间
3 ros::param::set("~set_C", 100); //私有,参考命名空间与节点名称
```

运行时, 假设设置的 namespace 为 xxx, 节点名称为 yyy, 使用 `rosparam list` 查看:

```

1 /set_A
2 /xxx/set_B
3 /xxx/yyy/set_C

```

1.2ros::NodeHandle设置参数

设置参数时，首先需要创建 NodeHandle 对象，然后调用该对象的 setParam 函数，该函数参数1为参数名，参数2为要设置的参数值，如果参数名以 / 开头，那么就是全局参数，如果参数名不以 / 开头，那么，该参数是相对参数还是私有参数与NodeHandle 对象有关，如果NodeHandle 对象创建时如果是调用的默认的无参构造，那么该参数是相对参数，如果NodeHandle 对象创建时是使用：

ros::NodeHandle nh("~")，那么该参数就是私有参数。代码示例：

```

1 ros::NodeHandle nh;
2 nh.setParam("/nh_A", 100); //全局,和命名空间以及节点名称
 无关
3
4 nh.setParam("nh_B", 100); //相对,参考命名空间
5
6 ros::NodeHandle nh_private("~");
7 nh_private.setParam("nh_C", 100); //私有,参考命名空间与节
  点名称

```

运行时，假设设置的 namespace 为 xxx，节点名称为 yyy，使用 rosparam list 查看：

```

1 /nh_A
2 /xxx/nh_B
3 /xxx/yyy/nh_C

```

2.python实现

python 中关于参数设置的语法实现比 C++ 简洁一些，调用的API时 `rospy.set_param`，该函数中，参数1传入参数名称，参数2是传入参数值，参数1中参数名称设置时，如果以 / 开头，那么就是全局参数，如果以 ~ 开头，那么就是私有参数，既不以 / 也不以 ~ 开头，那么就是相对参数。代码示例：

```
1 rospy.set_param("/py_A",100)    #全局,和命名空间以及节点名
称无关
2 rospy.set_param("py_B",100)    #相对,参考命名空间
3 rospy.set_param("~py_C",100)    #私有,参考命名空间与节点名
称
```

运行时，假设设置的 `namespace` 为 `xxx`，节点名称为 `yyy`，使用 `rosparam list` 查看：

```
1 /py_A
2 /xxx/py_B
3 /xxx/yyy/py_C
```

4.7 ROS分布式通信

ROS是一个分布式计算环境。一个运行中的ROS系统可以包含分布在多台计算机上多个节点。根据系统的配置方式，任何节点可能随时需要与任何其他节点进行通信。

因此，ROS对网络配置有某些要求：

- 所有端口上的所有机器之间必须有完整的双向连接。
- 每台计算机必须通过所有其他计算机都可以解析的名称来公告自己。

实现

1.准备

先要保证不同计算机处于同一网络中，最好分别设置固定IP，如果为虚拟机，需要将网络适配器改为桥接模式；

2.配置文件修改

分别修改不同计算机的 /etc/hosts 文件，在该文件中加入对方的IP地址和计算机名：

主机端：

```
1 从机的IP    从机计算机名
```

从机端：

```
1 主机的IP    主机计算机名
```

设置完毕，可以通过 ping 命令测试网络通信是否正常。



IP地址查看名: ifconfig

计算机名称查看: hostname

3.配置主机IP

配置主机的 IP 地址

~/.bashrc 追加

```
1 export ROS_MASTER_URI=http://主机IP:11311
2 export ROS_HOSTNAME=主机IP
```

4.配置从机IP

配置从机的 IP 地址，从机可以有多台，每台都做如下设置：

~/.bashrc 追加

```
1 export ROS_MASTER_URI=http://主机IP:11311
2 export ROS_HOSTNAME=从机IP
```

测试

1. 主机启动 roscore(必须)
2. 主机启动订阅节点，从机启动发布节点，测试通信是否正常
3. 反向测试，主机启动发布节点，从机启动订阅节点，测试通信是否正常

4.8 本章小结

本章主要介绍了ROS的运行管理机制，内容如下：

- 如何通过元功能包关联工作空间下的不同功能包
- 使用 launch 文件来管理维护 ROS 中的节点
- 在 ROS 中重名是经常出现的，重名时会导致什么情况？以及怎么避免重名？
- 如何实现 ROS 分布式通信？

本章的重点是"重名"相关的内容：

- 包名重复，会导致覆盖。
- 节点名称重复，会导致先启动的节点关闭
- 话题名称重复，无语法异常，但是可能导致通信实现出现逻辑问题
- 参数名称重复，会导致参数设置的覆盖

解决重名问题的实现方案有两种：

- 重映射(重新起名字)
- 为命名添加前缀

本章介绍的内容还是偏向语法层面的实现，下一章将开始介绍ROS中内置的一些较为实用的组件。

第5章 ROS常用组件

在ROS中内置一些比较实用的工具，通过这些工具可以方便快捷的实现某个功能或调试程序，从而提高开发效率，本章主要介绍ROS中内置的如下组件：

- TF坐标变换，实现不同类型的坐标系之间的转换；
- rosbag 用于录制ROS节点的执行过程并可以重放该过程；
- rqt 工具箱，集成了多款图形化的调试工具。

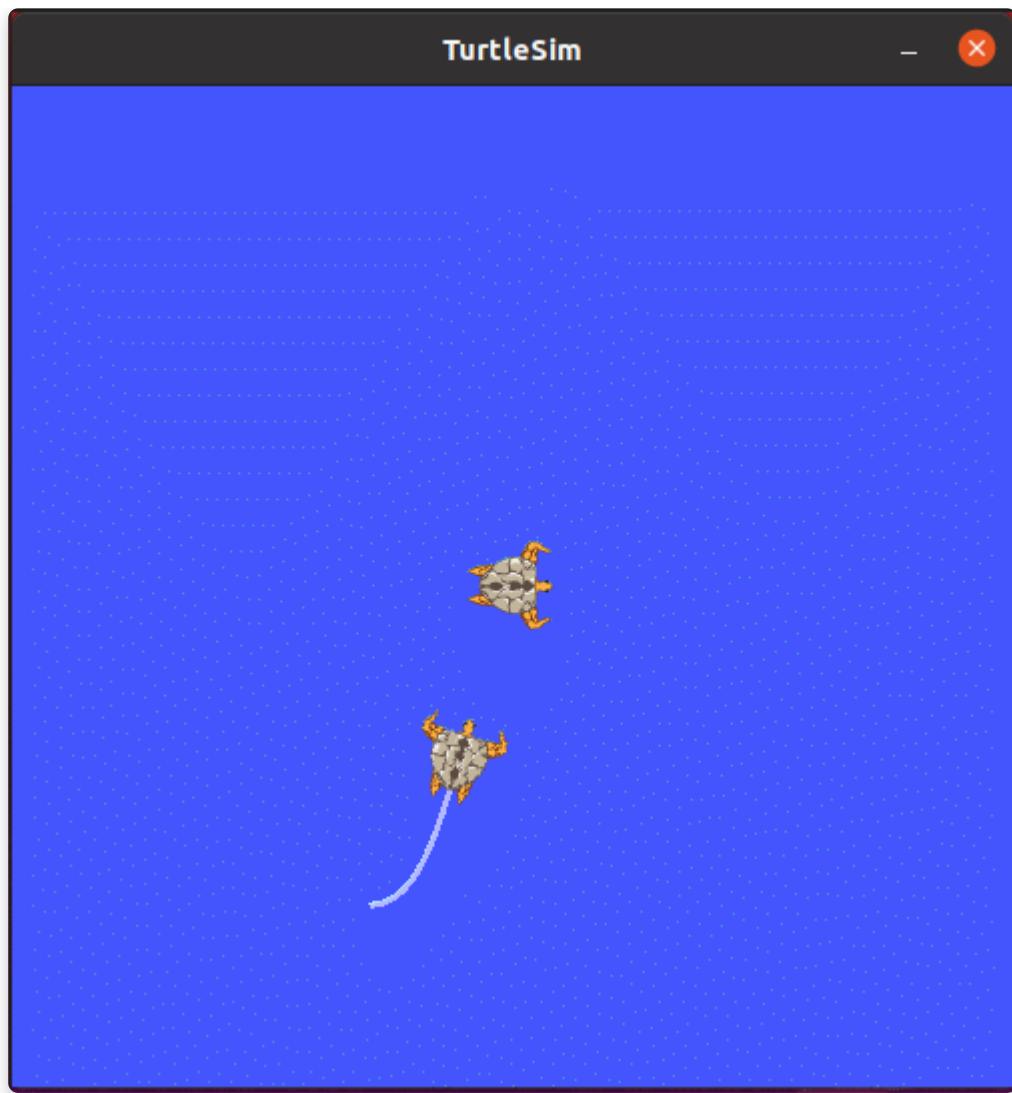
本章预期达成的学习目标：

- 了解 TF 坐标变换的概念以及应用场景；
- 能够独立完成TF案例:小乌龟跟随；
- 可以使用 rosbag 命令或编码的形式实现录制与回放；
- 能够熟练使用rqt中的图形化工具。

案例演示: 小乌龟跟随实现，该案例是ros中内置案例，终端下键入启动命令

```
1 roslaunch turtle_tf2 turtle_tf2_demo_cpp.launch`或
`roslaunch turtle_tf2 turtle_tf2_demo.launch
```

键盘可以控制一只乌龟运动，另一只跟随运动。



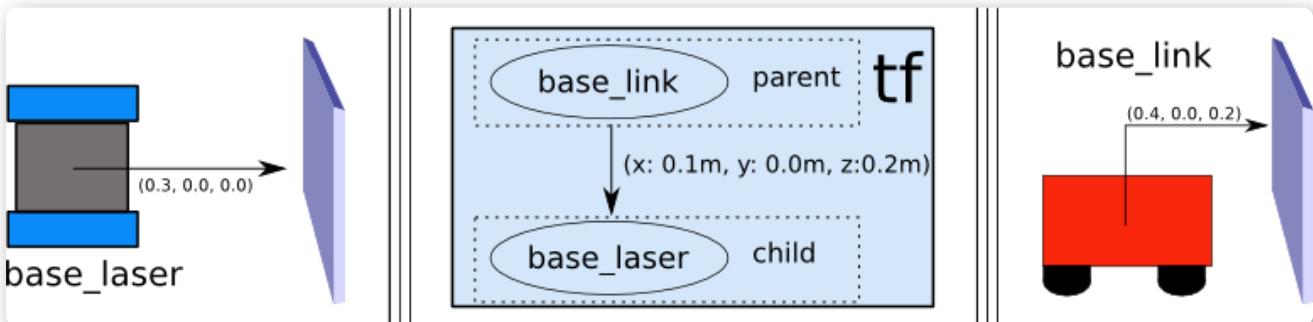
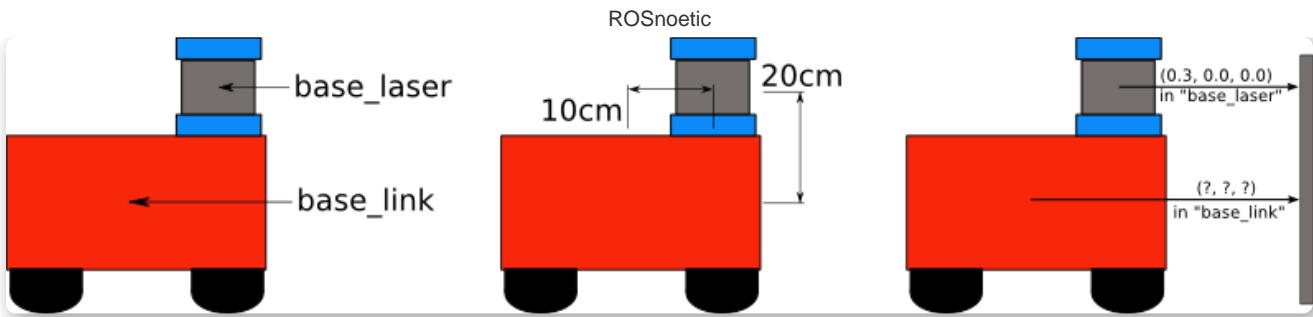
5.1 TF坐标变换

机器人系统上，有多个传感器，如激光雷达、摄像头等，有的传感器是可以感知机器人周边的物体方位(或者称之为:坐标，横向、纵向、高度的距离信息)的，以协助机器人定位障碍物，可以直接将物体相对该传感器的方位信息，等价于物体相对于机器人系统或机器人其它组件的方位信息吗？显示是不行的，这中间需要一个转换过程。更具体描述如下：

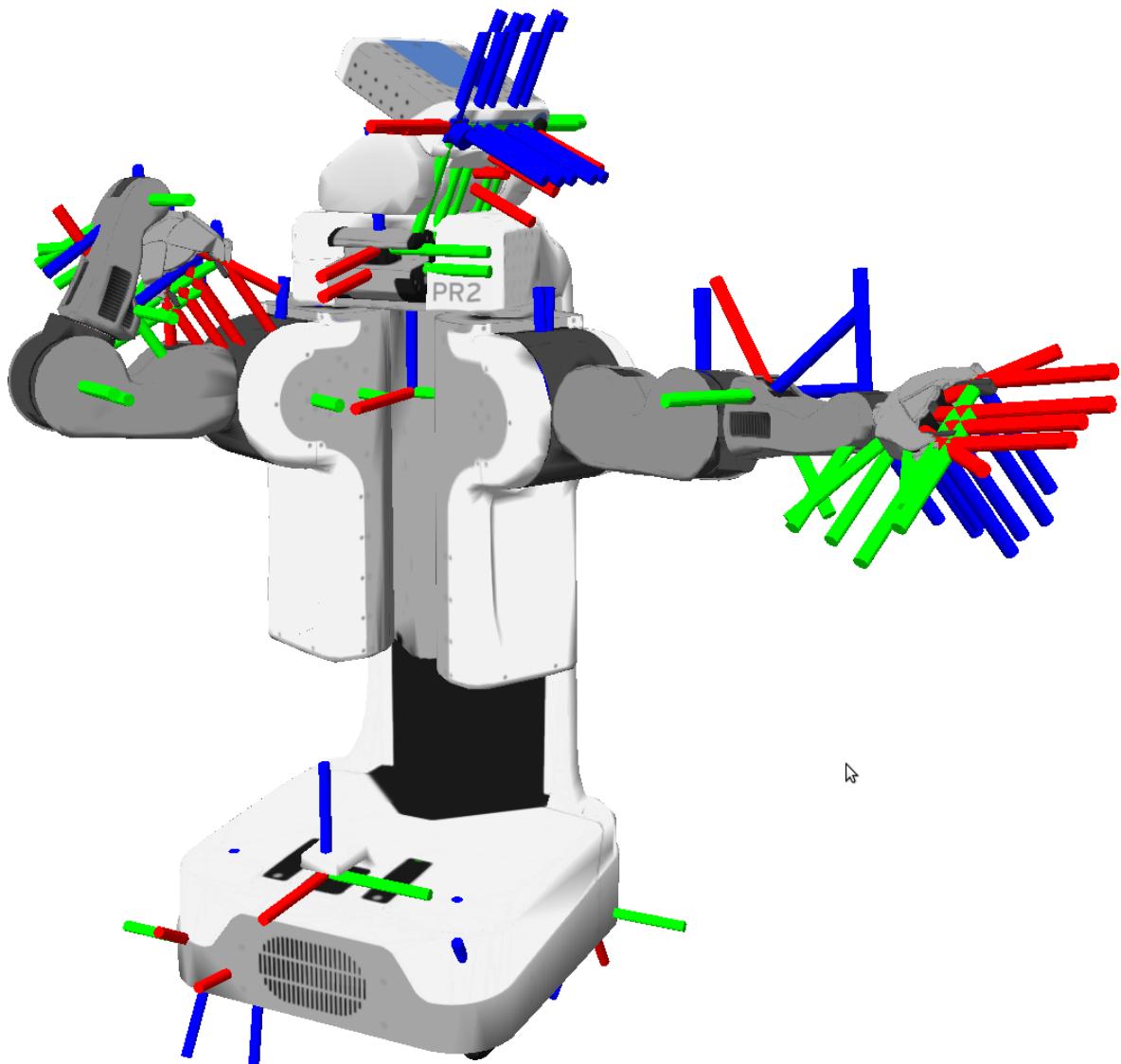


场景1:雷达与小车

现有一移动式机器人底盘，在底盘上安装了一雷达，雷达相对于底盘的偏移量已知，现雷达检测到一障碍物信息，获取到坐标分别为 (x, y, z) ，该坐标是以雷达为参考系的，如何将这个坐标转换成以小车为参考系的坐标呢？



场景2:现有一带机械臂的机器人(比如:PR2)需要夹取目标物,当前机器人头部摄像头可以探测到目标物的坐标(x,y,z),不过该坐标是以摄像头为参考系的,而实际操作目标物的是机械臂的夹具,当前我们需要将该坐标转换成相对于机械臂夹具的坐标,这个过程如何实现?

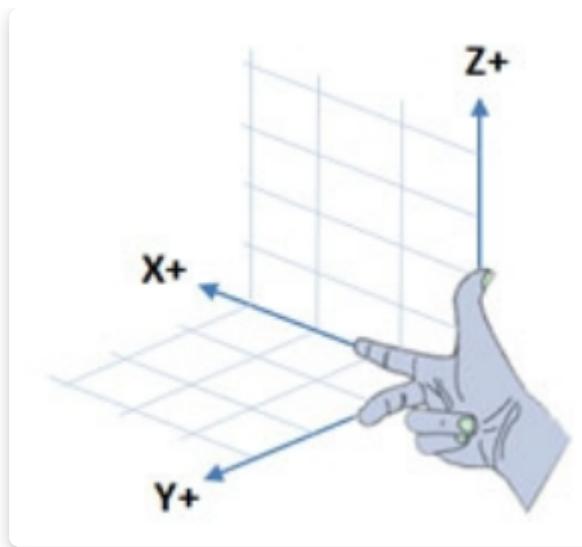


当然，根据我们高中学习的知识，在明确了不同坐标系之间的的相对关系，就可以实现任何坐标点在不同坐标系之间的转换，但是该计算实现是较为常用的，且算法也有点复杂，因此在 ROS 中直接封装了相关的模块: 坐标变换 (TF)。

概念

tf:TransForm Frame,坐标变换

坐标系:ROS 中是通过坐标系统开标定物体的，确切的将是通过右手坐标系来标定的。



作用

在 ROS 中用于实现不同坐标系之间的点或向量的转换。

案例

小乌龟跟随案例：如本章引言部分演示。

说明

在ROS中坐标变换最初对应的是tf，不过在 hydro 版本开始, tf 被弃用，迁移到 tf2,后者更为简洁高效， tf2对应的常用功能包有：

tf2_geometry_msgs:可以将ROS消息转换成tf2消息。

tf2:封装了坐标变换的常用消息。

tf2_ros:为tf2提供了roscpp和rospy绑定，封装了坐标变换常用的API。

另请参考:

- <http://wiki.ros.org/tf2>

5.1.1 坐标msg消息

订阅发布模型中数据载体 msg 是一个重要实现，首先需要了解一下，在坐标转换实现中常用的 msg: `geometry_msgs/TransformStamped` 和 `geometry_msgs/PointStamped`

前者用于传输坐标系相关位置信息，后者用于传输某个坐标系内坐标点的信息。在坐标变换中，频繁的需要使用到坐标系的相对关系以及坐标点信息。

1. `geometry_msgs/TransformStamped`

命令行键入: `rosmmsg info geometry_msgs/TransformStamped`

```

1 std_msgs/Header header          #头信息
2   uint32 seq                  # |-- 序
3   列号
4   time stamp                 # |-- 时
5   间戳
6   string frame_id            # |-- 坐标 ID
7   string child_frame_id       #子坐标系的
8   id
9   geometry_msgs/Transform transform #坐标信息
10  geometry_msgs/Vector3 translation #偏移量
11  float64 x                  # |-- X 方向的偏移量
12  float64 y                  # |-- Y 方向的偏移量
13  float64 z                  # |-- Z 方向上的偏移量
14  geometry_msgs/Quaternion rotation #四元数
15  float64 x
16  float64 y
17  float64 z
18  float64 w

```

四元数用于表示坐标的相对姿态

2.geometry_msgs/PointStamped

命令行键入: `rosmsg info geometry_msgs/PointStamped`

```

1 std_msgs/Header header          #头
2   uint32 seq                  # |-- 序
3   time stamp                # |-- 时
4   string frame_id            # |-- 所
   属坐标系的 id
5 geometry_msgs/Point point      #点坐标
6   float64 x                  # |-- 
   x y z 坐标
7   float64 y
8   float64 z

```

另请参考:

- http://docs.ros.org/en/api/geometry_msgs/html/msg/TransformStamped.html
- http://docs.ros.org/en/api/geometry_msgs/html/msg/PointStamped.html

5.1.2 静态坐标变换

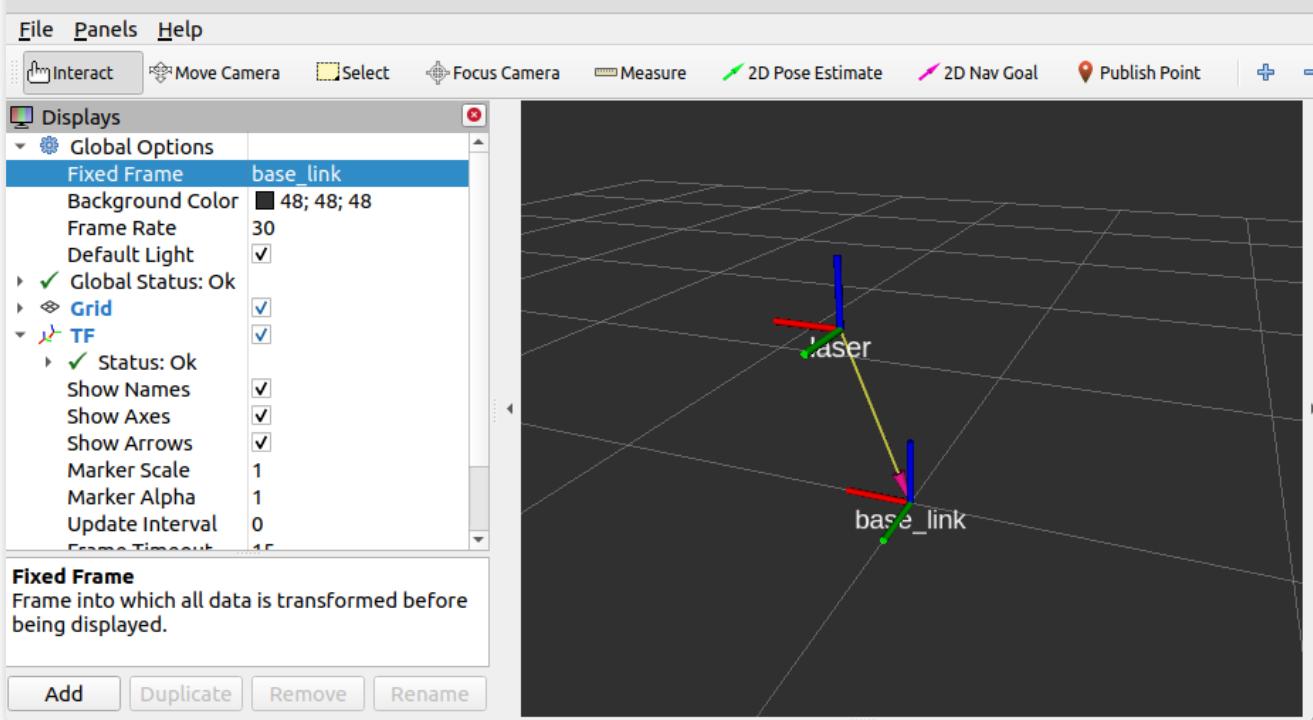
所谓静态坐标变换，是指两个坐标系之间的相对位置是固定的。

需求描述:

现有一机器人模型，核心构成包含主体与雷达，各对应一坐标系，坐标系的原点分别位于主体与雷达的物理中心，已知雷达原点相对于主体原点位移关系如下: x 0.2 y 0.0 z 0.5。当前雷达检测到一障碍物，在雷达坐标系中障碍物的坐标为 (2.0 3.0 5.0)，请问，该障碍物相对于主体的坐标是多少？

结果演示:

[INFO] [1610356308.935662701]: 转换后的数据:(2.20,3.00,5.50), 参考的坐标系是:base_link



实现分析:

1. 坐标系相对关系，可以通过发布方发布
2. 订阅方，订阅到发布的坐标系相对关系，再传入坐标点信息(可以写死)，然后借助于 tf 实现坐标变换，并将结果输出

实现流程:C++ 与 Python 实现流程一致

1. 新建功能包，添加依赖
2. 编写发布方实现
3. 编写订阅方实现
4. 执行并查看结果

方案A:C++实现

1. 创建功能包

创建项目功能包依赖于 `tf2`、`tf2_ros`、`tf2_geometry_msgs`、`roscpp`
`rospy` `std_msgs` `geometry_msgs`

2.发布方

```
1  /*
2   * 静态坐标变换发布方:
3   *   发布关于 laser 坐标系的位置信息
4
5   * 实现流程:
6   *   1. 包含头文件
7   *   2. 初始化 ROS 节点
8   *   3. 创建静态坐标转换广播器
9   *   4. 创建坐标系信息
10  *   5. 广播器发布坐标系信息
11  *   6. spin()
12 */
13
14
15 // 1. 包含头文件
16 #include "ros/ros.h"
17 #include "tf2_ros/static_transform_broadcaster.h"
18 #include "geometry_msgs/TransformStamped.h"
19 #include "tf2/LinearMath/Quaternion.h"
20
21 int main(int argc, char *argv[])
22 {
23     setlocale(LC_ALL, "");
24     // 2. 初始化 ROS 节点
25     ros::init(argc, argv, "static_brocast");
26     // 3. 创建静态坐标转换广播器
27     tf2_ros::StaticTransformBroadcaster
broadcaster;
28     // 4. 创建坐标系信息
29     geometry_msgs::TransformStamped ts;
30     //----设置头信息
31     ts.header.seq = 100;
32     ts.header.stamp = ros::Time::now();
33     ts.header.frame_id = "base_link";
34     //----设置子级坐标系
35     ts.child_frame_id = "laser";
```

```

36     //----设置子级相对于父级的偏移量
37     ts.transform.translation.x = 0.2;
38     ts.transform.translation.y = 0.0;
39     ts.transform.translation.z = 0.5;
40     //----设置四元数:将 欧拉角数据转换成四元数
41     tf2::Quaternion qtn;
42     qtn.setRPY(0,0,0);
43     ts.transform.rotation.x = qtn.getX();
44     ts.transform.rotation.y = qtn.getY();
45     ts.transform.rotation.z = qtn.getZ();
46     ts.transform.rotation.w = qtn.getW();
47     // 5.广播器发布坐标系信息
48     broadcaster.sendTransform(ts);
49     ros::spin();
50     return 0;
51 }
```

配置文件此处略。

3.订阅方

```

1  /*
2      订阅坐标系信息，生成一个相对于 子级坐标系的坐标点数据，
3      转换成父级坐标系中的坐标点
4
5      实现流程：
6          1.包含头文件
7          2.初始化 ROS 节点
8          3.创建 TF 订阅节点
9          4.生成一个坐标点(相对于子级坐标系)
10         5.转换坐标点(相对于父级坐标系)
11         6.spin()
12
13 //1.包含头文件
14 #include "ros/ros.h"
15 #include "tf2_ros/transform_listener.h"
16 #include "tf2_ros/buffer.h"
17 #include "geometry_msgs/PointStamped.h"
```

```

17 #include "tf2_geometry_msgs/tf2_geometry_msgs.h"
    //注意：调用 transform 必须包含该头文件
18
19 int main(int argc, char *argv[])
20 {
21     setlocale(LC_ALL, "");
22     // 2. 初始化 ROS 节点
23     ros::init(argc, argv, "tf_sub");
24     ros::NodeHandle nh;
25     // 3. 创建 TF 订阅节点
26     tf2_ros::Buffer buffer;
27     tf2_ros::TransformListener listener(buffer);
28
29     ros::Rate r(1);
30     while (ros::ok())
31     {
32         // 4. 生成一个坐标点(相对于子级坐标系)
33         geometry_msgs::PointStamped point_laser;
34         point_laser.header.frame_id = "laser";
35         point_laser.header.stamp =
36             ros::Time::now();
37         point_laser.point.x = 1;
38         point_laser.point.y = 2;
39         point_laser.point.z = 7.3;
40         // 5. 转换坐标点(相对于父级坐标系)
41         // 新建一个坐标点，用于接收转换结果
42         //-----使用 try 语句或休眠，否则可能由
43         //于缓存接收延迟而导致坐标转换失败-----
44         try
45         {
46             geometry_msgs::PointStamped
47             point_base;
48             point_base =
49                 buffer.transform(point_laser, "base_link");
50             ROS_INFO("转换后的数据: (%.2f, %.2f, %.2f),",
51             point_base.point.x, point_base.point.y, point_base.
52             point.z, point_base.header.frame_id.c_str());
53         }
54     }
55 }
```

```

48     }
49     catch(const std::exception& e)
50     {
51         // std::cerr << e.what() << '\n';
52         ROS_INFO("程序异常.....");
53     }
54
55
56     r.sleep();
57     ros::spinOnce();
58 }
59
60
61     return 0;
62 }
```

配置文件此处略。

4.执行

可以使用命令行或launch文件的方式分别启动发布节点与订阅节点，如果程序无异常，控制台将输出，坐标转换后的结果。

方案B:Python实现

1.创建功能包

创建项目功能包依赖于 tf2、tf2_ros、tf2_geometry_msgs、roscpp
rospy std_msgs geometry_msgs

2.发布方

```

1  #! /usr/bin/env python
2  """
3      静态坐标变换发布方:
4          发布关于 laser 坐标系的位置信息
```

```

5      实现流程:
6          1. 导包
7          2. 初始化 ROS 节点
8          3. 创建 静态坐标广播器
9          4. 创建并组织被广播的消息
10         5. 广播器发送消息
11         6. spin
12 """
13 # 1. 导包
14 import rospy
15 import tf2_ros
16 import tf
17 from geometry_msgs.msg import TransformStamped
18
19 if __name__ == "__main__":
20     # 2. 初始化 ROS 节点
21     rospy.init_node("static_tf_pub_p")
22     # 3. 创建 静态坐标广播器
23     broadcaster =
24         tf2_ros.StaticTransformBroadcaster()
25         # 4. 创建并组织被广播的消息
26         tfs = TransformStamped()
27         # --- 头信息
28         tfs.header.frame_id = "world"
29         tfs.header.stamp = rospy.Time.now()
30         tfs.header.seq = 101
31         # --- 子坐标系
32         tfs.child_frame_id = "radar"
33         # --- 坐标系相对信息
34         # ----- 偏移量
35         tfs.transform.translation.x = 0.2
36         tfs.transform.translation.y = 0.0
37         tfs.transform.translation.z = 0.5
38         # ----- 四元数
39         qtn =
40             tf.transformations.quaternion_from_euler(0,0,0)
41             tfs.transform.rotation.x = qtn[0]
42             tfs.transform.rotation.y = qtn[1]
43             tfs.transform.rotation.z = qtn[2]

```

```

42     tfs.transform.rotation.w = qtn[3]
43
44
45     # 5.广播器发送消息
46     broadcaster.sendTransform(tfs)
47     # 6.spin
48     rospy.spin()

```

权限设置以及配置文件此处略。

3.订阅方

```

1  #! /usr/bin/env python
2  """
3      订阅坐标系信息，生成一个相对于 子级坐标系的坐标点数据，  

4      转换成父级坐标系中的坐标点
5
6      实现流程：  

7          1. 导包  

8          2. 初始化 ROS 节点  

9          3. 创建 TF 订阅对象  

10         4. 创建一个 radar 坐标系中的坐标点  

11         5. 调研订阅对象的 API 将 4 中的点坐标转换成相对于  

12         world 的坐标  

13         6.spin
14 """
15 # 1. 导包
16 import rospy
17 import tf2_ros
18 # 不要使用 geometry_msgs，需要使用 tf2 内置的消息类型
19 from tf2_geometry_msgs import PointStamped
20 # from geometry_msgs.msg import PointStamped
21
22 if __name__ == "__main__":
23     # 2. 初始化 ROS 节点
24     rospy.init_node("static_sub_tf_p")
25     # 3. 创建 TF 订阅对象

```

```

26     buffer = tf2_ros.Buffer()
27     listener = tf2_ros.TransformListener(buffer)
28
29     rate = rospy.Rate(1)
30     while not rospy.is_shutdown():
31         # 4. 创建一个 radar 坐标系中的坐标点
32         point_source = PointStamped()
33         point_source.header.frame_id = "radar"
34         point_source.header.stamp =
35             rospy.Time.now()
36         point_source.point.x = 10
37         point_source.point.y = 2
38         point_source.point.z = 3
39
40         try:
41             # 5. 调研订阅对象的 API 将 4 中的点坐标转换成相对于 world 的坐标
42             point_target =
43                 buffer.transform(point_source, "world")
44             rospy.loginfo("转换结果:x = %.2f, y =
45             %.2f, z = %.2f",
46             point_target.point.x,
47             point_target.point.y,
48             point_target.point.z)
49         except Exception as e:
50             rospy.logerr("异常:%s", e)
51
52         # 6.spin
53         rate.sleep()

```

权限设置以及配置文件此处略。

PS: 在 tf2 的 python 实现中, tf2 已经封装了一些消息类型, 不可以使用 geometry_msgs.msg 中的类型

4. 执行

可以使用命令行或launch文件的方式分别启动发布节点与订阅节点，如果程序无异常，控制台将输出，坐标转换后的结果。

补充1:

当坐标系之间的相对位置固定时，那么所需参数也是固定的：父系坐标名称、子级坐标系名称、x偏移量、y偏移量、z偏移量、x翻滚角度、y俯仰角度、z偏航角度，实现逻辑相同，参数不同，那么ROS系统就已经封装好了专门的节点，使用方式如下：

```
1 rosrun tf2_ros static_transform_publisher x偏移量 y偏
    移量 z偏移量 z偏航角度 y俯仰角度 x翻滚角度 父级坐标系 子级坐
    标系
```

示例: `rosrun tf2_ros static_transform_publisher 0.2 0`
`0.5 0 0 0 /baselink /laser`

也建议使用该种方式直接实现静态坐标系相对信息发布。

补充2:

可以借助于rviz显示坐标系关系，具体操作：

- 新建窗口输入命令:rviz;
- 在启动的rviz中设置Fixed Frame为base_link;
- 点击左下的add按钮，在弹出的窗口中选择TF组件，即可显示坐标关系。

另请参考:

- <http://wiki.ros.org/tf2/Tutorials/Writing%20a%20tf2%20static%20broadcaster%20%28C%2B%2B%29>

- <http://wiki.ros.org/tf2/Tutorials/Writing%20a%20tf2%20static%20broadcaster%20%28Python%29>
- <http://wiki.ros.org/tf2/Tutorials/Writing%20a%20tf2%20listener%20%28C%2B%2B%29>
- <http://wiki.ros.org/tf2/Tutorials/Writing%20a%20tf2%20listener%20%28Python%29>

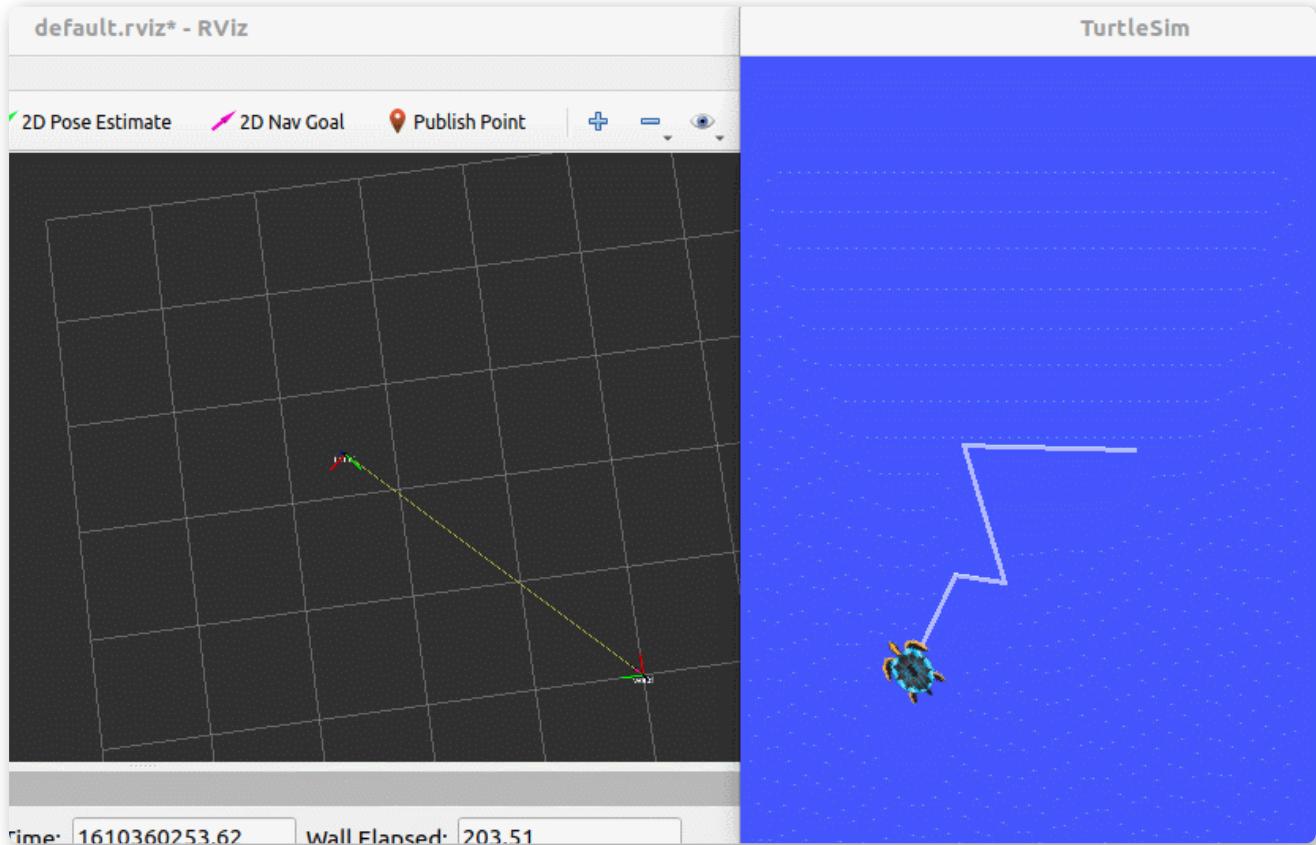
5.1.3 动态坐标变换

所谓动态坐标变换，是指两个坐标系之间的相对位置是变化的。

需求描述：

启动 turtlesim_node, 该节点中窗体有一个世界坐标系(左下角为坐标系原点)，乌龟是另一个坐标系，键盘控制乌龟运动，将两个坐标系的相对位置动态发布。

结果演示：



实现分析：

1. 乌龟本身不但可以看作坐标系，也是世界坐标系中的一个坐标点

2. 订阅 `turtle1/pose`, 可以获取乌龟在世界坐标系的 x 坐标、y 坐标、偏移量以及线速度和角速度
3. 将 `pose` 信息转换成 坐标系相对信息并发布

实现流程: C++ 与 Python 实现流程一致

1. 新建功能包, 添加依赖
2. 创建坐标相对关系发布方(同时需要订阅乌龟位姿信息)
3. 创建坐标相对关系订阅方
4. 执行

方案A:C++实现

1. 创建功能包

创建项目功能包依赖于 `tf2`、`tf2_ros`、`tf2_geometry_msgs`、`roscpp`
`rospy` `std_msgs` `geometry_msgs`、`turtlesim`

2. 发布方

```

1  /*
2   * 动态的坐标系相对姿态发布(一个坐标系相对于另一个坐标系的
3   * 相对姿态是不断变动的)
4
5   * 需求: 启动 turtlesim_node, 该节点中窗体有一个世界坐标
6   * 系(左下角为坐标系原点), 乌龟是另一个坐标系, 键盘
7   * 控制乌龟运动, 将两个坐标系的相对位置动态发布
8
9   * 实现分析:
10  * 1. 乌龟本身不但可以看作坐标系, 也是世界坐标系中的一个坐标点
11  * 2. 订阅 turtle1/pose, 可以获取乌龟在世界坐标系的
12  * x 坐标、y 坐标、偏移量以及线速度和角速度
13  * 3. 将 pose 信息转换成 坐标系相对信息并发布
14
15  * 实现流程:
16  * 1. 包含头文件

```

```

14         2. 初始化 ROS 节点
15         3. 创建 ROS 句柄
16         4. 创建订阅对象
17         5. 回调函数处理订阅到的数据(实现TF广播)
18             5-1. 创建 TF 广播器
19             5-2. 创建 广播的数据(通过 pose 设置)
20             5-3. 广播器发布数据
21         6. spin
22 */
23 // 1. 包含头文件
24 #include "ros/ros.h"
25 #include "turtlesim/Pose.h"
26 #include "tf2_ros/transform_broadcaster.h"
27 #include "geometry_msgs/TransformStamped.h"
28 #include "tf2/LinearMath/Quaternion.h"
29
30 void doPose(const turtlesim::Pose::ConstPtr& pose)
{
31     // 5-1. 创建 TF 广播器
32     static tf2_ros::TransformBroadcaster
broadcaster;
33     // 5-2. 创建 广播的数据(通过 pose 设置)
34     geometry_msgs::TransformStamped tfs;
35     // |----头设置
36     tfs.header.frame_id = "world";
37     tfs.header.stamp = ros::Time::now();
38
39     // |----坐标系 ID
40     tfs.child_frame_id = "turtle1";
41
42     // |----坐标系相对信息设置
43     tfs.transform.translation.x = pose->x;
44     tfs.transform.translation.y = pose->y;
45     tfs.transform.translation.z = 0.0; // 二维实现,
pose 中没有z, z 是 0
46     // |----- 四元数设置
47     tf2::Quaternion qtn;
48     qtn.setRPY(0,0,pose->theta);
49     tfs.transform.rotation.x = qtn.getX();

```

```

50     tfs.transform.rotation.y = qtn.getY();
51     tfs.transform.rotation.z = qtn.getZ();
52     tfs.transform.rotation.w = qtn.getW();
53
54
55     // 5-3.广播器发布数据
56     broadcaster.sendTransform(tfs);
57 }
58
59 int main(int argc, char *argv[])
60 {
61     setlocale(LC_ALL, "");
62     // 2.初始化 ROS 节点
63     ros::init(argc, argv, "dynamic_tf_pub");
64     // 3.创建 ROS 句柄
65     ros::NodeHandle nh;
66     // 4.创建订阅对象
67     ros::Subscriber sub =
68         nh.subscribe<turtlesim::Pose>
69         ("~/turtle1/pose", 1000, doPose);
70         // 5.回调函数处理订阅到的数据(实现TF广播)
71         //
72         // 6.spin
73         ros::spin();
74         return 0;
75 }
```

配置文件此处略。

3.订阅方

```

1 //1.包含头文件
2 #include "ros/ros.h"
3 #include "tf2_ros/transform_listener.h"
4 #include "tf2_ros/buffer.h"
5 #include "geometry_msgs/PointStamped.h"
6 #include "tf2_geometry_msgs/tf2_geometry_msgs.h"
//注意：调用 transform 必须包含该头文件
```

```

7
8 int main(int argc, char *argv[])
9 {
10     setlocale(LC_ALL, "");
11     // 2. 初始化 ROS 节点
12     ros::init(argc, argv, "dynamic_tf_sub");
13     ros::NodeHandle nh;
14     // 3. 创建 TF 订阅节点
15     tf2_ros::Buffer buffer;
16     tf2_ros::TransformListener listener(buffer);
17
18     ros::Rate r(1);
19     while (ros::ok())
20     {
21         // 4. 生成一个坐标点(相对于子级坐标系)
22         geometry_msgs::PointStamped point_laser;
23         point_laser.header.frame_id = "turtle1";
24         point_laser.header.stamp = ros::Time();
25         point_laser.point.x = 1;
26         point_laser.point.y = 1;
27         point_laser.point.z = 0;
28         // 5. 转换坐标点(相对于父级坐标系)
29         // 新建一个坐标点, 用于接收转换结果
30         //-----使用 try 语句或休眠, 否则可能由于缓存接收延迟而导致坐标转换失败-----
31         try
32         {
33             geometry_msgs::PointStamped
34             point_base;
35             point_base =
36             buffer.transform(point_laser, "world");
37             ROS_INFO("坐标点相对于 world 的坐标为:
38             (%.2f,%.2f,%.2f)", point_base.point.x, point_base.po
39             int.y, point_base.point.z);
40
41         }
42         catch(const std::exception& e)
43         {
44             // std::cerr << e.what() << '\n';

```

```

41         ROS_INFO("程序异常:%s",e.what());
42     }
43
44
45     r.sleep();
46     ros::spinOnce();
47 }
48
49
50     return 0;
51 }
```

配置文件此处略。

4.执行

可以使用命令行或launch文件的方式分别启动发布节点与订阅节点，如果程序无异常，与演示结果类似。

可以使用 rviz 查看坐标系相对关系。

方案B:Python实现

1.创建功能包

创建项目功能包依赖于 tf2、tf2_ros、tf2_geometry_msgs、roscpp
rospy std_msgs geometry_msgs、turtlesim

2.发布方

```

1  #! /usr/bin/env python
2  """

```

```

3      动态的坐标系相对姿态发布(一个坐标系相对于另一个坐标系的
4      相对姿态是不断变动的)
5
```

需求：启动 `turtlesim_node`, 该节点中窗体有一个世界坐标系(左下角为坐标系原点)，乌龟是另一个坐标系，键盘控制乌龟运动，将两个坐标系的相对位置动态发布

实现分析：

1. 乌龟本身不但可以看作坐标系，也是世界坐标系中的一个坐标点
2. 订阅 `turtle1/pose`, 可以获取乌龟在世界坐标系的 x 坐标、y 坐标、偏移量以及线速度和角速度
3. 将 `pose` 信息转换成 坐标系相对信息并发布

实现流程：

1. 导包
2. 初始化 ROS 节点
3. 订阅 `/turtle1/pose` 话题消息
4. 回调函数处理
 - 4-1. 创建 TF 广播器
 - 4-2. 创建 广播的数据(通过 `pose` 设置)
 - 4-3. 广播器发布数据
5. `spin`

```
"""
# 1. 导包
import rospy
import tf2_ros
import tf
from turtlesim.msg import Pose
from geometry_msgs.msg import TransformStamped

# 4. 回调函数处理
def doPose(pose):
    # 4-1. 创建 TF 广播器
    broadcaster = tf2_ros.TransformBroadcaster()
    # 4-2. 创建 广播的数据(通过 pose 设置)
    tfs = TransformStamped()
    tfs.header.frame_id = "world"
    tfs.header.stamp = rospy.Time.now()
    tfs.child_frame_id = "turtle1"
    tfs.transform.translation.x = pose.x
    tfs.transform.translation.y = pose.y
    tfs.transform.translation.z = 0.0
    # 4-3. 广播器发布数据
    broadcaster.sendTransform(tfs)

# 5. spin
rospy.spin()
"""
```

```

41     qtn =
42         tf.transformations.quaternion_from_euler(0,0,pose.
43             theta)
43         tfs.transform.rotation.x = qtn[0]
44         tfs.transform.rotation.y = qtn[1]
45         tfs.transform.rotation.z = qtn[2]
45         tfs.transform.rotation.w = qtn[3]
46         #           4-3.广播器发布数据
47         broadcaster.sendTransform(tfs)
48
49 if __name__ == "__main__":
50     # 2.初始化 ROS 节点
51     rospy.init_node("dynamic_tf_pub_p")
52     # 3.订阅 /turtle1/pose 话题消息
53     sub =
54         rospy.Subscriber("/turtle1/pose",Pose,doPose)
54         #       4.回调函数处理
55         #           4-1.创建 TF 广播器
56         #           4-2.创建 广播的数据(通过 pose 设置)
57         #           4-3.广播器发布数据
58         #       5.spin
59         rospy.spin()

```

权限设置以及配置文件此处略。

3.订阅方

```

1  #! /usr/bin/env python
2  """
3      动态的坐标系相对姿态发布(一个坐标系相对于另一个坐标系的
4          相对姿态是不断变动的)
5
6      需求: 启动 turtlesim_node,该节点中窗体有一个世界坐标
7          系(左下角为坐标系原点), 乌龟是另一个坐标系, 键盘
8          控制乌龟运动, 将两个坐标系的相对位置动态发布
9
10     实现分析:

```

```

9          1. 乌龟本身不但可以看作坐标系，也是世界坐标系中的一个坐标点
10         2. 订阅 turtle1/pose，可以获取乌龟在世界坐标系的 x 坐标、y 坐标、偏移量以及线速度和角速度
11         3. 将 pose 信息转换成 坐标系相对信息并发布
12         实现流程：
13         1. 导包
14         2. 初始化 ROS 节点
15         3. 创建 TF 订阅对象
16         4. 处理订阅的数据
17
18
19 """
20 # 1. 导包
21 import rospy
22 import tf2_ros
23 # 不要使用 geometry_msgs，需要使用 tf2 内置的消息类型
24 from tf2_geometry_msgs import PointStamped
25 # from geometry_msgs.msg import PointStamped
26
27 if __name__ == "__main__":
28     # 2. 初始化 ROS 节点
29     rospy.init_node("static_sub_tf_p")
30     # 3. 创建 TF 订阅对象
31     buffer = tf2_ros.Buffer()
32     listener = tf2_ros.TransformListener(buffer)
33
34     rate = rospy.Rate(1)
35     while not rospy.is_shutdown():
36         # 4. 创建一个 radar 坐标系中的坐标点
37         point_source = PointStamped()
38         point_source.header.frame_id = "turtle1"
39         point_source.header.stamp =
40             rospy.Time.now()
41         point_source.point.x = 10
42         point_source.point.y = 2
43         point_source.point.z = 3
44         try:

```

```

45      #      5.调研订阅对象的 API 将 4 中的点坐标转换成相对
46          point_target =
47          buffer.transform(point_source, "world", rospy.Duration(1))
48          rospy.loginfo("转换结果:x = %.2f, y =
49          %.2f, z = %.2f",
50          point_target.point.x,
51          point_target.point.y,
52          point_target.point.z)
53      except Exception as e:
54          rospy.logerr("异常:%s", e)
55      #      6.spin
56      rate.sleep()

```

权限设置以及配置文件此处略。

4.执行

可以使用命令行或launch文件的方式分别启动发布节点与订阅节点，如果程序无异常，与演示结果类似。

可以使用 rviz 查看坐标系相对关系。

另请参考：

- <http://wiki.ros.org/tf2/Tutorials/Writing%20a%20tf2%20broadcaster%20%28C%2B%2B%29>
- <http://wiki.ros.org/tf2/Tutorials/Writing%20a%20tf2%20broadcaster%20%28Python%29>

5.1.4 多坐标变换

需求描述：

现有坐标系统，父级坐标系统 world,下有两子级系统 son1, son2, son1 相对于 world, 以及 son2 相对于 world 的关系是已知的，求 son1原点在 son2中的坐标，又已知在 son1中一点的坐标，要求求出该点在 son2 中的坐标

实现分析:

1. 首先，需要发布 son1 相对于 world, 以及 son2 相对于 world 的坐标消息
2. 然后，需要订阅坐标发布消息，并取出订阅的消息，借助于 tf2 实现 son1 和 son2 的转换
3. 最后，还要实现坐标点的转换

实现流程:C++ 与 Python 实现流程一致

1. 新建功能包，添加依赖
 2. 创建坐标相对关系发布方(需要发布两个坐标相对关系)
 3. 创建坐标相对关系订阅方
 4. 执行
-

方案A:C++实现

1.创建功能包

创建项目功能包依赖于 tf2、tf2_ros、tf2_geometry_msgs、roscpp
rospy std_msgs geometry_msgs、turtlesim

2.发布方

为了方便，使用静态坐标变换发布

```

1 <launch>
2   <node pkg="tf2_ros"
3     type="static_transform_publisher" name="son1"
4     args="0.2 0.8 0.3 0 0 0 /world /son1"
5     output="screen" />
6   <node pkg="tf2_ros"
7     type="static_transform_publisher" name="son2"
8     args="0.5 0 0 0 0 0 /world /son2" output="screen" />
9 </launch>

```

3.订阅方

```

1 /*
2
3 需求：
4   现有坐标系统，父级坐标系统 world，下有两子级系统
5   son1, son2,
6   son1 相对于 world，以及 son2 相对于 world 的关系是
7   已知的，
8   求 son1 与 son2 中的坐标关系，又已知在 son1 中一点的坐
9   标，要求求出该点在 son2 中的坐标
10
11 实现流程：
12   1. 包含头文件
13   2. 初始化 ros 节点
14   3. 创建 ros 句柄
15   4. 创建 TF 订阅对象
16   5. 解析订阅信息中获取 son1 坐标系原点在 son2 中的坐标
17   解析 son1 中的点相对于 son2 的坐标
18   6. spin
19
20 */
21 //1. 包含头文件
22 #include <ros/ros.h>
23 #include <tf2_ros/transform_listener.h>
24 #include <tf2/LinearMath/Quaternion.h>
25 #include <tf2_geometry_msgs/tf2_geometry_msgs.h>
26 #include <geometry_msgs/TransformStamped.h>
27 #include <geometry_msgs/PointStamped.h>

```

```

24
25 int main(int argc, char *argv[])
26 {   setlocale(LC_ALL,"");
27     // 2. 初始化 ros 节点
28     ros::init(argc,argv,"sub_frames");
29     // 3. 创建 ros 句柄
30     ros::NodeHandle nh;
31     // 4. 创建 TF 订阅对象
32     tf2_ros::Buffer buffer;
33     tf2_ros::TransformListener listener(buffer);
34     // 5. 解析订阅信息中获取 son1 坐标系原点在 son2 中的
35     // 坐标
36     ros::Rate r(1);
37     while (ros::ok())
38     {
39         try
40         {
41             // 解析 son1 中的点相对于 son2 的坐标
42             geometry_msgs::TransformStamped tfs =
43             buffer.lookupTransform("son2","son1",ros::Time(0));
44             ROS_INFO("Son1 相对于 Son2 的坐标关系:父坐
45             标系ID=%s",tfs.header.frame_id.c_str());
46             ROS_INFO("Son1 相对于 Son2 的坐标关系:子坐
47             标系ID=%s",tfs.child_frame_id.c_str());
48             ROS_INFO("Son1 相对于 Son2 的坐标关
49             系:x=%f,y=%f,z=%f",
50             tfs.transform.translation.x,
51             tfs.transform.translation.y,
52             tfs.transform.translation.z
53             );
54
55             // 坐标点解析
56             geometry_msgs::PointStamped ps;
57             ps.header.frame_id = "son1";
58             ps.header.stamp = ros::Time::now();
59             ps.point.x = 1.0;
60             ps.point.y = 2.0;
61             ps.point.z = 3.0;

```

```

57
58         geometry_msgs::PointStamped psAtSon2;
59         psAtSon2 =
60             buffer.transform(ps, "son2");
61             ROS_INFO("在 Son2 中的坐
62             标:x=%f,y=%f,z=%f",
63             psAtSon2.point.x,
64             psAtSon2.point.y,
65             psAtSon2.point.z
66         );
67     }
68     catch(const std::exception& e)
69     {
70         // std::cerr << e.what() << '\n';
71         ROS_INFO("异常信息:%s",e.what());
72     }
73
74     r.sleep();
75     // 6.spin
76     ros::spinOnce();
77 }
78 }
```

配置文件此处略。

4.执行

可以使用命令行或launch文件的方式分别启动发布节点与订阅节点，如果程序无异常，将输出换算后的结果。

方案B:Python实现

1. 创建功能包

创建项目功能包依赖于 tf2、tf2_ros、tf2_geometry_msgs、roscpp
rospy std_msgs geometry_msgs、turtlesim

2. 发布方

为了方便，使用静态坐标变换发布

```

1 <launch>
2   <node pkg="tf2_ros"
3     type="static_transform_publisher" name="son1"
4     args="0.2 0.8 0.3 0 0 0 /world /son1"
5     output="screen" />
6   <node pkg="tf2_ros"
7     type="static_transform_publisher" name="son2"
8     args="0.5 0 0 0 0 0 /world /son2" output="screen" />
9 </launch>

```

3. 订阅方

```

1 #!/usr/bin/env python
2 """
3   需求：
4     现有坐标系统，父级坐标系统 world，下有两子级系统
5     son1, son2,
6     son1 相对于 world，以及 son2 相对于 world 的关
7     系是已知的，
8     求 son1 与 son2 中的坐标关系，又已知在 son1 中一
9     点的坐标，要求求出该点在 son2 中的坐标
10
11   实现流程：
12     1. 导包
13     2. 初始化 ROS 节点
14     3. 创建 TF 订阅对象
15     4. 调用 API 求出 son1 相对于 son2 的坐标关系
16     5. 创建一依赖于 son1 的坐标点，调用 API 求出该点
17       在 son2 中的坐标
18
19     6. spin

```

```

14
15 """
16 # 1. 导包
17 import rospy
18 import tf2_ros
19 from geometry_msgs.msg import TransformStamped
20 from tf2_geometry_msgs import PointStamped
21
22 if __name__ == "__main__":
23
24     # 2. 初始化 ROS 节点
25     rospy.init_node("frames_sub_p")
26     # 3. 创建 TF 订阅对象
27     buffer = tf2_ros.Buffer()
28     listener = tf2_ros.TransformListener(buffer)
29
30     rate = rospy.Rate(1)
31     while not rospy.is_shutdown():
32
33         try:
34             # 4. 调用 API 求出 son1 相对于 son2 的坐标关系
35             #lookup_transform(self, target_frame,
36             source_frame, time, timeout=rospy.Duration(0.0)):
37             tfs =
38                 buffer.lookup_transform("son2", "son1", rospy.Time(0))
39                 rospy.loginfo("son1 与 son2 相对关系:")
40                 rospy.loginfo("父级坐标
41                 系:%s", tfs.header.frame_id)
42                 rospy.loginfo("子级坐标
43                 系:%s", tfs.child_frame_id)
44                 rospy.loginfo("相对坐标:x=%.2f, y=%.2f,
45                 z=%.2f",
46                 tfs.transform.translation.x,
47                 tfs.transform.translation.y,
48                 tfs.transform.translation.z,

```

```

44
45      # 5. 创建一个依赖于 son1 的坐标点, 调用 API 求出该
46      # 点在 son2 中的坐标
47      point_source = PointStamped()
48      point_source.header.frame_id = "son1"
49      point_source.header.stamp =
50      rospy.Time.now()
51      point_source.point.x = 1
52      point_source.point.y = 1
53      point_source.point.z = 1
54
55      point_target =
56      buffer.transform(point_source, "son2", rospy.Duration(0.5))
57
58      rospy.loginfo("point_target 所属的坐标
59      系: %s", point_target.header.frame_id)
60      rospy.loginfo("坐标点相对于 son2 的坐标:
61      (%.2f, %.2f, %.2f)",
62      point_target.point.x,
63      point_target.point.y,
64      point_target.point.z
65
66      )
67
68      except Exception as e:
69      rospy.logerr("错误提示: %s", e)
70
71
72      rate.sleep()
73      # 6.spin
74      # rospy.spin()

```

权限设置以及配置文件此处略。

4. 执行

可以使用命令行或launch文件的方式分别启动发布节点与订阅节点, 如果程序无异常, 将输出换算后的结果。

5.1.5 坐标系关系查看

在机器人系统中，涉及的坐标系有多个，为了方便查看，ros 提供了专门的工具，可以用于生成显示坐标系关系的 pdf 文件，该文件包含树形结构的坐标系图谱。

6.1准备

首先调用 `rospack find tf2_tools` 查看是否包含该功能包，如果没有，请使用如下命令安装：

```
1 sudo apt install ros-noetic-tf2-tools
```

6.2使用

6.2.1生成 pdf 文件

启动坐标系广播程序之后，运行如下命令：

```
1 rosrun tf2_tools view_frames.py
```

会产生类似于下面的日志信息：

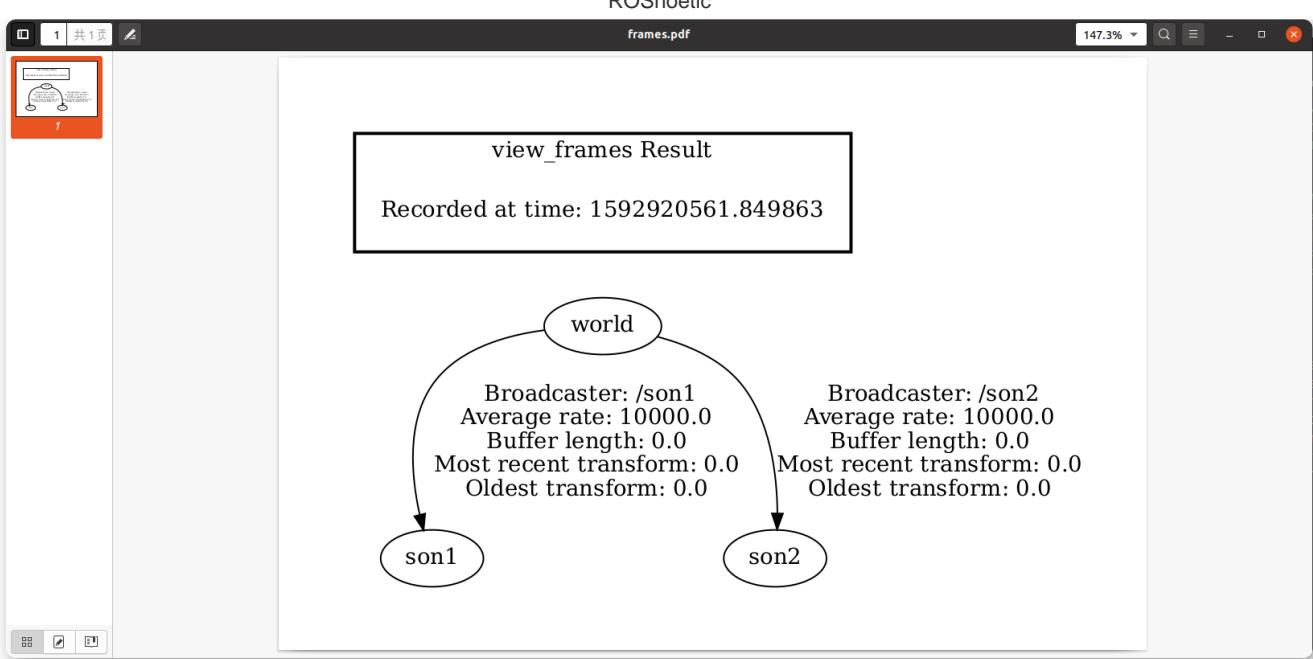
```
1 [INFO] [1592920556.827549]: Listening to tf data
  during 5 seconds...
2 [INFO] [1592920561.841536]: Generating graph in
  frames.pdf file...
```

查看当前目录会生成一个 `frames.pdf` 文件

6.2.2查看文件

可以直接进入目录打开文件，或者调用命令查看文件：`evince frames.pdf`

内如如图所示：



另请参考:

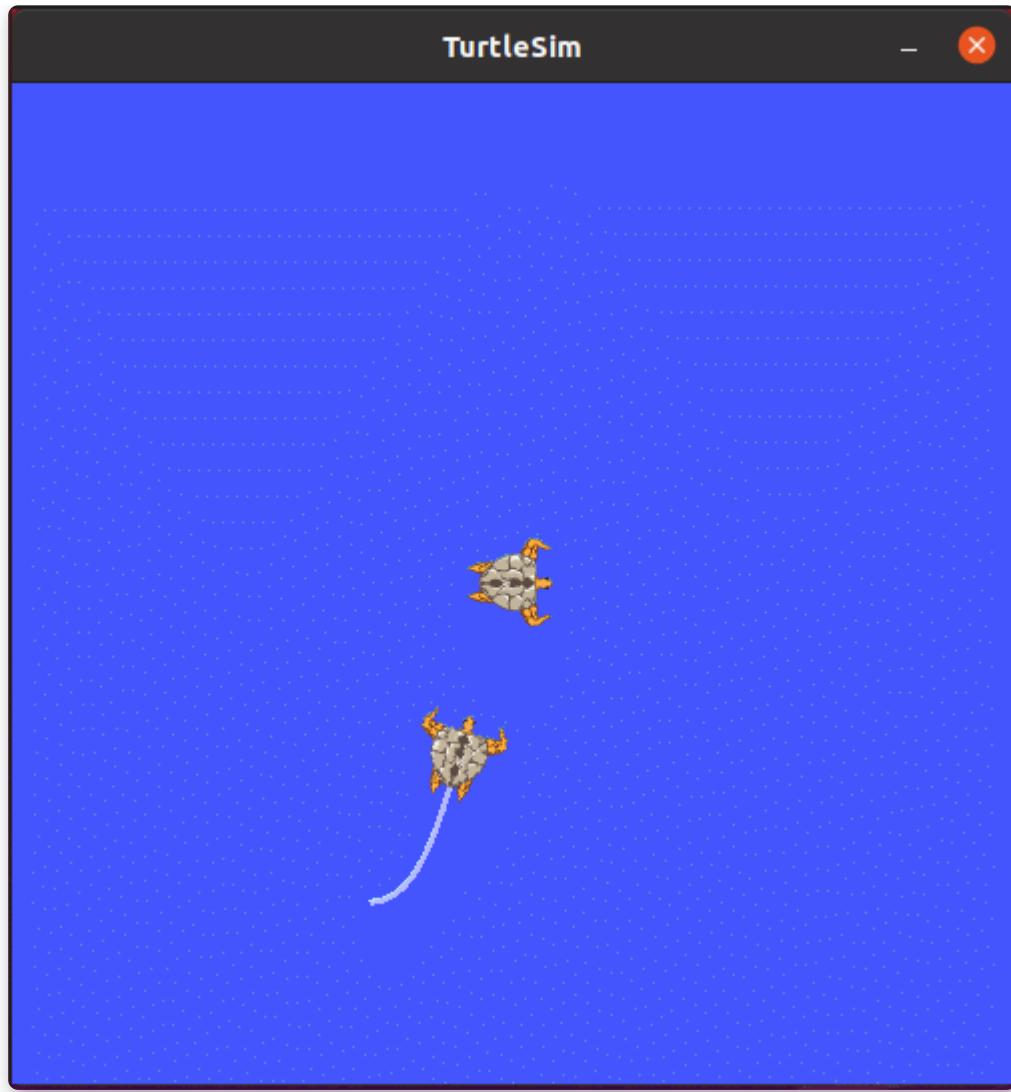
- http://wiki.ros.org/tf2_tools

5.1.6 TF坐标变换实操

需求描述:

程序启动之初: 产生两只乌龟, 中间的乌龟(A) 和 左下乌龟(B), B 会自动运行至A的位置, 并且键盘控制时, 只是控制 A 的运动, 但是 B 可以跟随 A 运行

结果演示:



实现分析:

乌龟跟随实现的核心，是乌龟A和B都要发布相对世界坐标系的坐标信息，然后，订阅到该信息需要转换获取A相对于B坐标系的信息，最后，再生成速度信息，并控制B运动。

1. 启动乌龟显示节点
2. 在乌龟显示窗体中生成一只新的乌龟(需要使用服务)
3. 编写两只乌龟发布坐标信息的节点
4. 编写订阅节点订阅坐标信息并生成新的相对关系生成速度信息

实现流程:C++ 与 Python 实现流程一致

1. 新建功能包，添加依赖
2. 编写服务客户端，用于生成一只新的乌龟
3. 编写发布方，发布两只乌龟的坐标信息
4. 编写订阅方，订阅两只乌龟信息，生成速度信息并发布
5. 运行

准备工作:

1.了解如何创建第二只乌龟，且不受键盘控制

创建第二只乌龟需要使用rosservice,话题使用的是 spawn

```
1 rosservice call /spawn "x: 1.0
2 y: 1.0
3 theta: 1.0
4 name: 'turtle_flow'"
5 name: "turtle_flow"
```

键盘是无法控制第二只乌龟运动的，因为使用的话题: /第二只乌龟名称/cmd_vel,对应的要控制乌龟运动必须发布对应的话题消息

2.了解如何获取两只乌龟的坐标

是通过话题 /乌龟名称/pose 来获取的

```
1 x: 1.0 //x坐标
2 y: 1.0 //y坐标
3 theta: -1.21437060833 //角度
4 linear_velocity: 0.0 //线速度
5 angular_velocity: 1.0 //角速度
```

方案A:C++实现

1.创建功能包

创建项目功能包依赖于 tf2、tf2_ros、tf2_geometry_msgs、roscpp
rospy std_msgs geometry_msgs、turtlesim

2.服务客户端(生成乌龟)

```
1  /*
2   * 创建第二只小乌龟
3  */
4 #include "ros/ros.h"
5 #include "turtlesim/Spawn.h"
6
7 int main(int argc, char *argv[])
8 {
9
10    setlocale(LC_ALL, "");
11
12    //执行初始化
13    ros::init(argc, argv, "create_turtle");
14    //创建节点
15    ros::NodeHandle nh;
16    //创建服务客户端
17    ros::ServiceClient client =
18        nh.serviceClient<turtlesim::Spawn>("/spawn");
19
20    ros::service::waitForService("/spawn");
21    turtlesim::Spawn spawn;
22    spawn.request.name = "turtle2";
23    spawn.request.x = 1.0;
24    spawn.request.y = 2.0;
25    spawn.request.theta = 3.12415926;
26    bool flag = client.call(spawn);
27    if (flag)
28    {
29        ROS_INFO("乌龟%s创建成
30        功!", spawn.response.name.c_str());
31    }
32    else
33    {
34        ROS_INFO("乌龟2创建失败!");
35    }
36}
```

```

35     ros::spin();
36
37     return 0;
38 }
```

配置文件此处略。

3.发布方(发布两只乌龟的坐标信息)

可以订阅乌龟的位姿信息，然后再转换成坐标信息，两只乌龟的实现逻辑相同，只是订阅的话题名称，生成的坐标信息等稍有差异，可以将差异部分通过参数传入：

- 该节点需要启动两次
- 每次启动时都需要传入乌龟节点名称(第一次是 `turtle1` 第二次是 `turtle2`)

```

1  /*
2   * 该文件实现：需要订阅 turtle1 和 turtle2 的 pose，然
3   * 后广播相对 world 的坐标系信息
4
5   * 注意：订阅的两只 turtle，除了命名空间(turtle1 和
6   * turtle2)不同外，
7   * 其他的话题名称和实现逻辑都是一样的，
8   * 所以我们可以将所需的命名空间通过 args 动态传入
9
10
11
12
13
14
15
16
17
18
19
```

实现流程：

1. 包含头文件
2. 初始化 `ros` 节点
3. 解析传入的命名空间
4. 创建 `ros` 句柄
5. 创建订阅对象
6. 回调函数处理订阅的 `pose` 信息
 - 6-1. 创建 `TF` 广播器
 - 6-2. 将 `pose` 信息转换成 `TransformStamped`
 - 6-3. 发布
7. `spin`

```

20 */
21 //1.包含头文件
22 #include "ros/ros.h"
23 #include "turtlesim/Pose.h"
24 #include "tf2_ros/transform_broadcaster.h"
25 #include "tf2/LinearMath/Quaternion.h"
26 #include "geometry_msgs/TransformStamped.h"
27 //保存乌龟名称
28 std::string turtle_name;
29
30
31 void doPose(const turtlesim::Pose::ConstPtr& pose)
{
32     // 6-1.创建 TF 广播器 -----
33     //----- 注意 static
34     static tf2_ros::TransformBroadcaster
35     broadcaster;
36
37     // 6-2.将 pose 信息转换成 TransformStamped
38     geometry_msgs::TransformStamped tfs;
39     tfs.header.frame_id = "world";
40     tfs.header.stamp = ros::Time::now();
41     tfs.child_frame_id = turtle_name;
42     tfs.transform.translation.x = pose->x;
43     tfs.transform.translation.y = pose->y;
44     tfs.transform.translation.z = 0.0;
45     tf2::Quaternion qtn;
46     qtn.setRPY(0,0,pose->theta);
47     tfs.transform.rotation.x = qtn.getX();
48     tfs.transform.rotation.y = qtn.getY();
49     tfs.transform.rotation.z = qtn.getZ();
50     tfs.transform.rotation.w = qtn.getW();
51
52     // 6-3.发布
53     broadcaster.sendTransform(tfs);
54
55 }
56
57 int main(int argc, char *argv[])
58 {
59     setlocale(LC_ALL, "");

```

```
56 // 2. 初始化 ros 节点
57 ros::init(argc, argv, "pub_tf");
58 // 3. 解析传入的命名空间
59 if (argc != 2)
60 {
61     ROS_ERROR("请传入正确的参数");
62 } else {
63     turtle_name = argv[1];
64     ROS_INFO("乌龟 %s 坐标发送启
65 动", turtle_name.c_str());
66
67 // 4. 创建 ros 句柄
68 ros::NodeHandle nh;
69 // 5. 创建订阅对象
70 ros::Subscriber sub =
71 nh.subscribe<turtlesim::Pose>(turtle_name +
72 "/pose", 1000, doPose);
73 // 6. 回调函数处理订阅的 pose 信息
74 // 6-1. 创建 TF 广播器
75 // 6-2. 将 pose 信息转换成
76 TransformStamped
77 // 6-3. 发布
78 // 7. spin
79 ros::spin();
80 return 0;
81 }
```

配置文件此处略。

4. 订阅方(解析坐标信息并生成速度信息)

```
1  /*
2   *      订阅 turtle1 和 turtle2 的 TF 广播信息，查找并转换
3   *      时间最近的 TF 信息
4   *      将 turtle1 转换成相对 turtle2 的坐标，在计算线速度和
5   *      角速度并发布
6   */
```

```

5      实现流程:
6          1. 包含头文件
7          2. 初始化 ros 节点
8          3. 创建 ros 句柄
9          4. 创建 TF 订阅对象
10         5. 处理订阅到的 TF
11         6. spin
12
13 */
14 //1. 包含头文件
15 #include "ros/ros.h"
16 #include "tf2_ros/transform_listener.h"
17 #include "geometry_msgs/TransformStamped.h"
18 #include "geometry_msgs/Twist.h"
19
20 int main(int argc, char *argv[])
21 {
22     setlocale(LC_ALL, "");
23     // 2. 初始化 ros 节点
24     ros::init(argc, argv, "sub_TF");
25     // 3. 创建 ros 句柄
26     ros::NodeHandle nh;
27     // 4. 创建 TF 订阅对象
28     tf2_ros::Buffer buffer;
29     tf2_ros::TransformListener listener(buffer);
30     // 5. 处理订阅到的 TF
31
32     // 需要创建发布 /turtle2/cmd_vel 的 publisher 对象
33
34     ros::Publisher pub =
35     nh.advertise<geometry_msgs::Twist>
36     ("/turtle2/cmd_vel", 1000);
37
38     ros::Rate rate(10);
39     while (ros::ok())
40     {
41         try
42         {

```

```

41          //5-1.先获取 turtle1 相对 turtle2 的坐标
42          // 信息
43          //5-2.根据坐标信息生成速度信息 --
44          //geometry_msgs/Twist.h
45          geometry_msgs::Twist twist;
46          twist.linear.x = 0.5 *
47          sqrt(pow(tfs.transform.translation.x,2) +
48          pow(tfs.transform.translation.y,2));
49          twist.angular.z = 4 *
50          atan2(tfs.transform.translation.y,tfs.transform.tr
51          anslation.x);
52          pub.publish(twist);
53      }
54      catch(const std::exception& e)
55      {
56          // std::cerr << e.what() << '\n';
57          ROS_INFO("错误提示:%s",e.what());
58      }
59
60      rate.sleep();
61      // 6.spin
62      ros::spinOnce();
63  }
64
65  return 0;
66 }
```

配置文件此处略。

5.运行

使用 launch 文件组织需要运行的节点，内容示例如下：

```
1  <!--
2      tf2 实现小乌龟跟随案例
3  -->
4 <launch>
5      <!-- 启动乌龟节点与键盘控制节点 -->
6      <node pkg="turtlesim" type="turtlesim_node"
7          name="turtle1" output="screen" />
8      <node pkg="turtlesim" type="turtle_teleop_key"
9          name="key_control" output="screen"/>
10     <!-- 启动创建第二只乌龟的节点 -->
11     <node pkg="demo_tf2_test"
12         type="Test01_Create_Turtle2" name="turtle2"
13         output="screen" />
14     <!-- 启动两个坐标发布节点 -->
15     <node pkg="demo_tf2_test"
16         type="Test02_TF2_Caster" name="caster1"
17         output="screen" args="turtle1" />
18     <node pkg="demo_tf2_test"
19         type="Test02_TF2_Caster" name="caster2"
20         output="screen" args="turtle2" />
21     <!-- 启动坐标转换节点 -->
22     <node pkg="demo_tf2_test"
23         type="Test03_TF2_Listener" name="listener"
24         output="screen" />
25 </launch>
```

方案B:Python实现

1. 创建功能包

创建项目功能包依赖于 tf2、tf2_ros、tf2_geometry_msgs、roscpp
rospy std_msgs geometry_msgs、turtlesim

2. 服务客户端(生成乌龟)

```

1  #! /usr/bin/env python
2  """
3      调用 service 服务在窗体指定位置生成一只乌龟
4      流程:
5          1. 导包
6          2. 初始化 ros 节点
7          3. 创建服务客户端
8          4. 等待服务启动
9          5. 创建请求数据
10         6. 发送请求并处理响应
11 """
12 #1. 导包
13 import rospy
14 from turtlesim.srv import Spawn, SpawnRequest,
15     SpawnResponse
16 if __name__ == "__main__":
17     # 2. 初始化 ros 节点
18     rospy.init_node("turtle_spawn_p")
19     # 3. 创建服务客户端
20     client = rospy.ServiceProxy("/spawn", Spawn)
21     # 4. 等待服务启动
22     client.wait_for_service()
23     # 5. 创建请求数据
24     req = SpawnRequest()
25     req.x = 1.0
26     req.y = 1.0
27     req.theta = 3.14
28     req.name = "turtle2"
29     # 6. 发送请求并处理响应
30     try:

```

```

31         response = client.call(req)
32         rospy.loginfo("乌龟创建成功, 名字
33     是:%s", response.name)
34     except Exception as e:
35         rospy.loginfo("服务调用失败....")

```

权限设置以及配置文件此处略。

3.发布方(发布两只乌龟的坐标信息)

```

1  #! /usr/bin/env python
2 """
3     该文件实现:需要订阅 turtle1 和 turtle2 的 pose, 然
4     后广播相对 world 的坐标系信息
5
6     注意: 订阅的两只 turtle,除了命名空间(turtle1 和
7     turtle2)不同外,
8         其他的话题名称和实现逻辑都是一样的,
9         所以我们可以将所需的命名空间通过 args 动态传入
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26

```

实现流程:

1. 导包
2. 初始化 ros 节点
3. 解析传入的命名空间
4. 创建订阅对象
5. 回调函数处理订阅的 pose 信息
 - 5-1. 创建 TF 广播器
 - 5-2. 将 pose 信息转换成 TransformStamped
 - 5-3. 发布
6. spin

```

27 """
28 # 1. 导包
29 import rospy
30 import sys
31 from turtlesim.msg import Pose
32 from geometry_msgs.msg import TransformStamped
33 import tf2_ros
34 import tf_conversions

```

```

27
28 turtle_name = ""
29
30 def doPose(pose):
31     # rospy.loginfo("x = %.2f", pose.x)
32     #1.创建坐标系广播器
33     broadcaster = tf2_ros.TransformBroadcaster()
34     #2.将 pose 信息转换成 TransformStamped
35     tfs = TransformStamped()
36     tfs.header.frame_id = "world"
37     tfs.header.stamp = rospy.Time.now()
38
39     tfs.child_frame_id = turtle_name
40     tfs.transform.translation.x = pose.x
41     tfs.transform.translation.y = pose.y
42     tfs.transform.translation.z = 0.0
43
44     qtn =
45         tf_conversions.transformations.quaternion_from_euler(0, 0, pose.theta)
46         tfs.transform.rotation.x = qtn[0]
47         tfs.transform.rotation.y = qtn[1]
48         tfs.transform.rotation.z = qtn[2]
49         tfs.transform.rotation.w = qtn[3]
50
51     #3.广播器发布 tfs
52     broadcaster.sendTransform(tfs)
53
54 if __name__ == "__main__":
55     # 2.初始化 ros 节点
56     rospy.init_node("sub_tfs_p")
57     # 3.解析传入的命名空间
58     rospy.loginfo("-----")
59     %d", len(sys.argv))
60     if len(sys.argv) < 2:
61         rospy.loginfo("请传入参数:乌龟的命名空间")
62     else:
63         turtle_name = sys.argv[1]

```

```

63     rospy.loginfo("//////////乌
64     龟:%s",turtle_name)
65     rospy.Subscriber(turtle_name +
66     "/pose",Pose,doPose)
67     #      4.创建订阅对象
68     #      5.回调函数处理订阅的 pose 信息
69     #          5-1.创建 TF 广播器
70     #          5-2.将 pose 信息转换成
71     #              TransformStamped
72     #          5-3.发布
73     #      6.spin
74     rospy.spin()

```

权限设置以及配置文件此处略。

4.订阅方(解析坐标信息并生成速度信息)

```

1  #! /usr/bin/env python
2  """
3      订阅 turtle1 和 turtle2 的 TF 广播信息，查找并转换
4      时间最近的 TF 信息
5      将 turtle1 转换成相对 turtle2 的坐标，在计算线速度和
6      角速度并发布
7
8      实现流程：
9          1. 导包
10         2. 初始化 ros 节点
11         3. 创建 TF 订阅对象
12         4. 处理订阅到的 TF
13             4-1. 查找坐标系的相对关系
14             4-2. 生成速度信息，然后发布
15 """
16
17 # 1. 导包
18 import rospy
19 import tf2_ros
20
21 from geometry_msgs.msg import TransformStamped,
22 Twist

```

```

18 import math
19
20 if __name__ == "__main__":
21     # 2. 初始化 ros 节点
22     rospy.init_node("sub_tfs_p")
23     # 3. 创建 TF 订阅对象
24     buffer = tf2_ros.Buffer()
25     listener = tf2_ros.TransformListener(buffer)
26     # 4. 处理订阅到的 TF
27     rate = rospy.Rate(10)
28     # 创建速度发布对象
29     pub =
30         rospy.Publisher("/turtle2/cmd_vel", Twist, queue_size=1000)
31
32         while not rospy.is_shutdown():
33
34             rate.sleep()
35             try:
36                 #def lookup_transform(self,
37                 target_frame, source_frame, time,
38                 timeout=rospy.Duration(0.0)):
39                 trans =
40                     buffer.lookup_transform("turtle2", "turtle1", rospy.
41                     Time(0))
42
43                     # rospy.loginfo("相对坐标:
44                     # (%.2f,%.2f,%.2f)",
45                     #
46                     trans.transform.translation.x,
47                     #
48                     trans.transform.translation.y,
49                     #
50                     trans.transform.translation.z
51                     #
52                     )
53                     # 根据转变后的坐标计算出速度和角速度信息
54                     twist = Twist()
55                     # 距离 = x^2 + y^2 然后开方
56                     twist.linear.x = 0.5 *
57                     math.sqrt(math.pow(trans.transform.translation.x, 2
58                     ) + math.pow(trans.transform.translation.y, 2))

```

```

45          twist.angular.z = 4 *
math.atan2(trans.transform.translation.y,
trans.transform.translation.x)
46
47          pub.publish(twist)
48
49      except Exception as e:
50          rospy.logwarn("警告:%s",e)

```

权限设置以及配置文件此处略。

5.运行

使用 launch 文件组织需要运行的节点，内容示例如下：

```

1 <launch>
2     <node pkg="turtlesim" type="turtlesim_node"
name="turtle1" output="screen" />
3     <node pkg="turtlesim" type="turtle_teleop_key"
name="key_control" output="screen"/>
4
5     <node pkg="demo06_test_flow_p"
type="test01_turtle_spawn_p.py"
name="turtle_spawn" output="screen"/>
6
7     <node pkg="demo06_test_flow_p"
type="test02_turtle_tf_pub_p.py" name="tf_pub1"
args="turtle1" output="screen"/>
8     <node pkg="demo06_test_flow_p"
type="test02_turtle_tf_pub_p.py" name="tf_pub2"
args="turtle2" output="screen"/>
9     <node pkg="demo06_test_flow_p"
type="test03_turtle_tf_sub_p.py" name="tf_sub"
output="screen"/>
10
11 </launch>

```

5.1.7 TF2与TF

1.TF2与TF比较_简介

- TF2已经替换了TF，TF2是TF的超集，建议学习TF2而非TF
- TF2功能包的增强了内聚性，TF与TF2所依赖的功能包是不同的，TF对应的是tf包，TF2对应的是tf2和tf2_ros包，在TF2中不同类型的API实现做了分包处理。
- TF2实现效率更高，比如在TF2的静态坐标实现、TF2坐标变换监听器中的Buffer实现等

2.TF2与TF比较_静态坐标变换演示

接下来，我们通过静态坐标变换来演示TF2的实现效率。

2.1启动TF2与TF两个版本的静态坐标变换

TF2版静态坐标变换: rosrun tf2_ros

```
static_transform_publisher 0 0 0 0 0 0 /base_link /laser
```

TF版静态坐标变换: rosrun tf static_transform_publisher 0

```
0 0 0 0 0 /base_link /laser 100
```

会发现，TF版本的启动中最后多一个参数，该参数是指定发布频率

2.2运行结果比对

使用rostopic查看话题，包含/tf与/tf_static，前者是TF发布的话题，后者是TF2发布的话题，分别调用命令打印二者的话题消息

rostopic echo /tf:当前会循环输出坐标系信息

rostopic echo /tf_static:坐标系信息只有一次

2.3结论

如果是静态坐标转换，那么不同坐标系之间的相对状态是固定的，既然是固定的，那么没有必要重复发布坐标系的转换消息，很显然的，tf2实现较之于tf更为高效

5.1.8 小结

坐标变换在机器人系统中是一个极其重要的组成模块，在 ROS 中 TF2 组件是专门用于实现坐标变换的，TF2 实现具体内容又主要介绍了如下几部分：

1. 静态坐标变换广播器，可以编码方式或调用内置功能包来实现(建议后者)，适用于相对固定的坐标系关系

2. 动态坐标变换广播器，以编码的方式广播坐标系之间的相对关系，适用于易变的坐标系关系

3. 坐标变换监听器，用于监听广播器广播的坐标系消息，可以实现不同坐标系之间或同一点在不同坐标系之间的变换

4. 机器人系统中的坐标系关系是较为复杂的，还可以通过 `tf2_tools` 工具包来生成 ros 中的坐标系关系图

5. 当前 TF2 已经替换了 TF，官网建议直接学习 TF2，并且 TF 与 TF2 的使用流程与实现 API 比较类似，只要有任意一方的使用经验，另一方也可以做到触类旁通

5.2 rosbag

机器人传感器获取到的信息，有时我们可能需要时时处理，有时可能只是采集数据，事后分析，比如：



机器人导航实现中，可能需要绘制导航所需的全局地图，地图绘制实现，有两种方式，方式1：可以控制机器人运动，将机器人传感器感知到的数据时时处理，生成地图信息。方式2：同样是控制机器人运动，将机器人传感器感知到的数据留存，事后，再重新读取数据，生成地图信息。两种方式比较，显然方式2使用上更为灵活方便。

在ROS中关于数据的留存以及读取实现，提供了专门的工具: `rosbag`。

概念

是用于录制和回放 ROS 主题的一个工具集。

作用

实现了数据的复用，方便调试、测试。

本质

rosbag本质也是ros的节点，当录制时，rosbag是一个订阅节点，可以订阅话题消息并将订阅到的数据写入磁盘文件；当重放时，rosbag是一个发布节点，可以读取磁盘文件，发布文件中的话题消息。

另请参考：

- <http://wiki.ros.org/rosbag>

5.2.1 rosbag使用_命令行

需求：

ROS 内置的乌龟案例并操作，操作过程中使用 rosbag 录制，录制结束后，实现重放

实现：

1.准备

创建目录保存录制的文件

```
1 mkdir ./xxx
2 cd xxx
```

2.开始录制

```
1 rosbag record -a -O 目标文件
```

操作小乌龟一段时间，结束录制使用 `ctrl + c`，在创建的目录中会生成bag文件。

3.查看文件

```
1 rosbag info 文件名
```

4.回放文件

```
1 rosbag play 文件名
```

重启乌龟节点，会发现，乌龟按照录制时的轨迹运动。

另请参考：

- <http://wiki.ros.org/rosbag/Commandline>

5.2.2 rosbag使用_编码

命令实现不够灵活，可以使用编码的方式，增强录制与回放的灵活性,本节将通过简单的读写实现演示rosbag的编码实现。

方案A:C++实现

1.写bag

```
1 #include "ros/ros.h"
2 #include "rosbag/bag.h"
3 #include "std_msgs/String.h"
4
5
6 int main(int argc, char *argv[])
7 {
8     ros::init(argc, argv, "bag_write");
```

```

9      ros::NodeHandle nh;
10     //创建bag对象
11     rosbag::Bag bag;
12     //打开
13
14     bag.open("/home/rostdemo/demo/test.bag",rosbag::BagMode::Write);
15     //写
16     std_msgs::String msg;
17     msg.data = "hello world";
18     bag.write("/chatter",ros::Time::now(),msg);
19     bag.write("/chatter",ros::Time::now(),msg);
20     bag.write("/chatter",ros::Time::now(),msg);
21     bag.write("/chatter",ros::Time::now(),msg);
22     //关闭
23     bag.close();
24
25     return 0;
26 }
```

2.读bag

```

1  /*
2   * 读取 bag 文件:
3
4  */
5 #include "ros/ros.h"
6 #include "rosbag/bag.h"
7 #include "rosbag/view.h"
8 #include "std_msgs/String.h"
9 #include "std_msgs/Int32.h"
10
11 int main(int argc, char *argv[])
12 {
13
14     setlocale(LC_ALL,"");
15
16     ros::init(argc,argv,"bag_read");
```

```

17     ros::NodeHandle nh;
18
19     //创建 bag 对象
20     rosbag::Bag bag;
21     //打开 bag 文件
22
23         bag.open("/home/rosdemo/demo/test.bag",rosbag::BagMode::Read);
24         //读数据
25         for (rosbag::MessageInstance const m :
26             rosbag::View(bag))
27         {
28             std_msgs::String::ConstPtr p =
29             m.instantiate<std_msgs::String>();
30             if(p != nullptr){
31                 ROS_INFO("读取的数据:%s",p-
32             >data.c_str());
33         }
34     }
35 }
```

方案B:Python实现

1.写 bag

```

1 #! /usr/bin/env python
2 import rospy
3 import rosbag
4 from std_msgs.msg import String
5
6 if __name__ == "__main__":
7     #初始化节点
```

```

8     rospy.init_node("w_bag_p")
9
10    # 创建 rosbag 对象
11    bag =
12      rosbag.Bag("/home/rosdemo/demo/test.bag",'w')
13
14    # 写数据
15    s = String()
16    s.data= "hahahaha"
17
18    bag.write("chatter",s)
19    bag.write("chatter",s)
20    bag.write("chatter",s)
21    # 关闭流
22    bag.close()

```

2. 读bag

```

1  #! /usr/bin/env python
2  import rospy
3  import rosbag
4  from std_msgs.msg import String
5
6  if __name__ == "__main__":
7      # 初始化节点
8      rospy.init_node("w_bag_p")
9
10     # 创建 rosbag 对象
11     bag =
12       rosbag.Bag("/home/rosdemo/demo/test.bag",'r')
13       # 读数据
14       bagMessage = bag.read_messages("chatter")
15       for topic,msg,t in bagMessage:
16           rospy.loginfo("%s,%s,%s",topic,msg,t)
17       # 关闭流
18       bag.close()

```

另请参考:

- <http://wiki.ros.org/rosbag/Code%20API>

5.3 rqt工具箱

之前，在ROS中使用了一些实用的工具,比如: `ros_bag` 用于录制与回放、`tf2_tools` 可以生成 TF 树 这些工具大大提高了开发的便利性，但是也存在一些问题: 这些工具的启动和使用过程中涉及到一些命令操作，应用起来不够方便，在ROS中，提供了rqt工具箱，在调用工具时以图形化操作代替了命令操作，应用更便利，提高了操作效率，优化了用户体验。

概念

ROS基于 QT 框架，针对机器人开发提供了一系列可视化的工具，这些工具的集合就是rqt

作用

可以方便的实现 ROS 可视化调试，并且在同一窗口中打开多个部件，提高开发效率，优化用户体验。

组成

rqt 工具箱组成有三大部分

- `rqt`——核心实现，开发人员无需关注
 - `rqt_common_plugins`——rqt 中常用的工具套件
 - `rqt_robot_plugins`——运行中和机器人交互的插件(比如: `rviz`)
-

另请参考:

- <http://wiki.ros.org/rqt>

5.3.1 rqt 安装启动与基本使用

1. 安装

- 一般只要你安装的是desktop-full版本就会自带工具箱
- 如果需要安装可以以如下方式安装

```

1 $ sudo apt-get install ros-noetic-rqt
2 $ sudo apt-get install ros-noetic-rqt-common-
  plugins

```

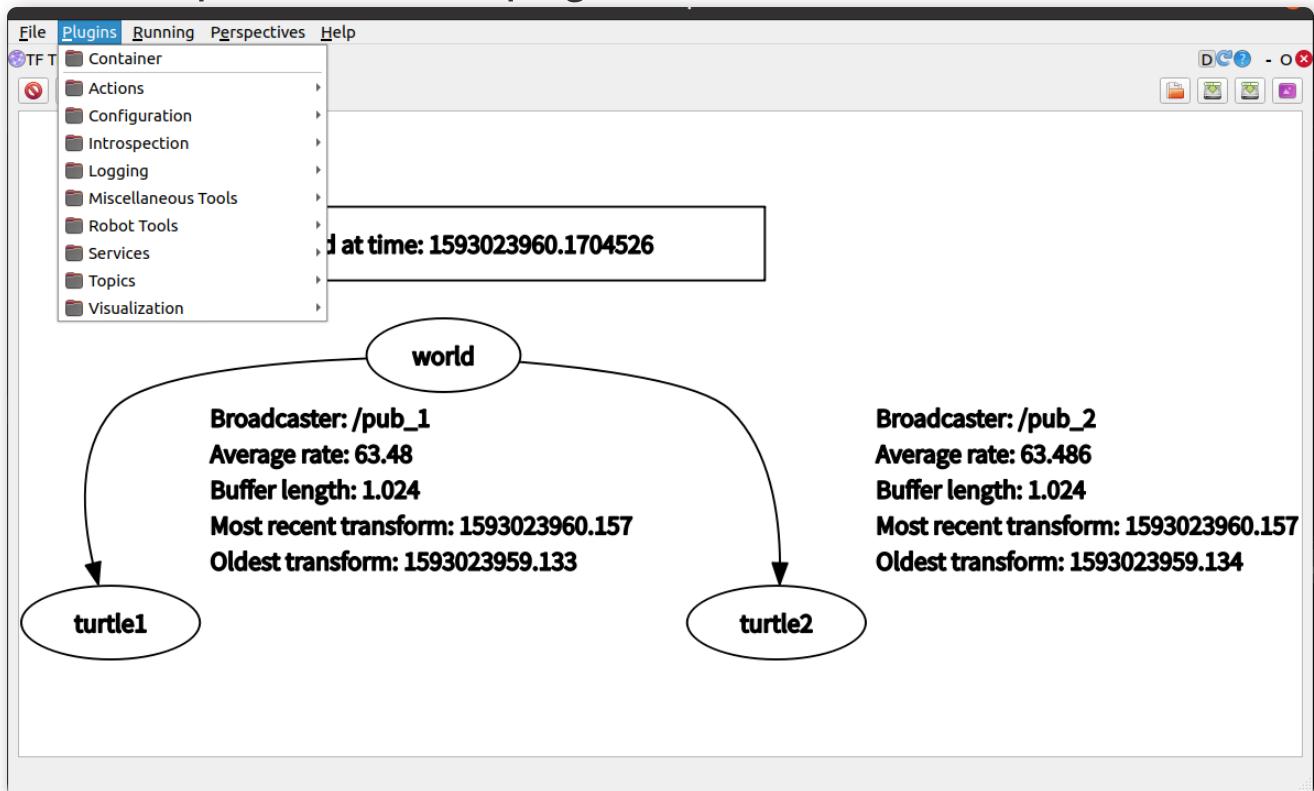
2. 启动

rqt 的启动方式有两种：

- 方式1: rqt
- 方式2: rosrun rqt_gui rqt_gui

3. 基本使用

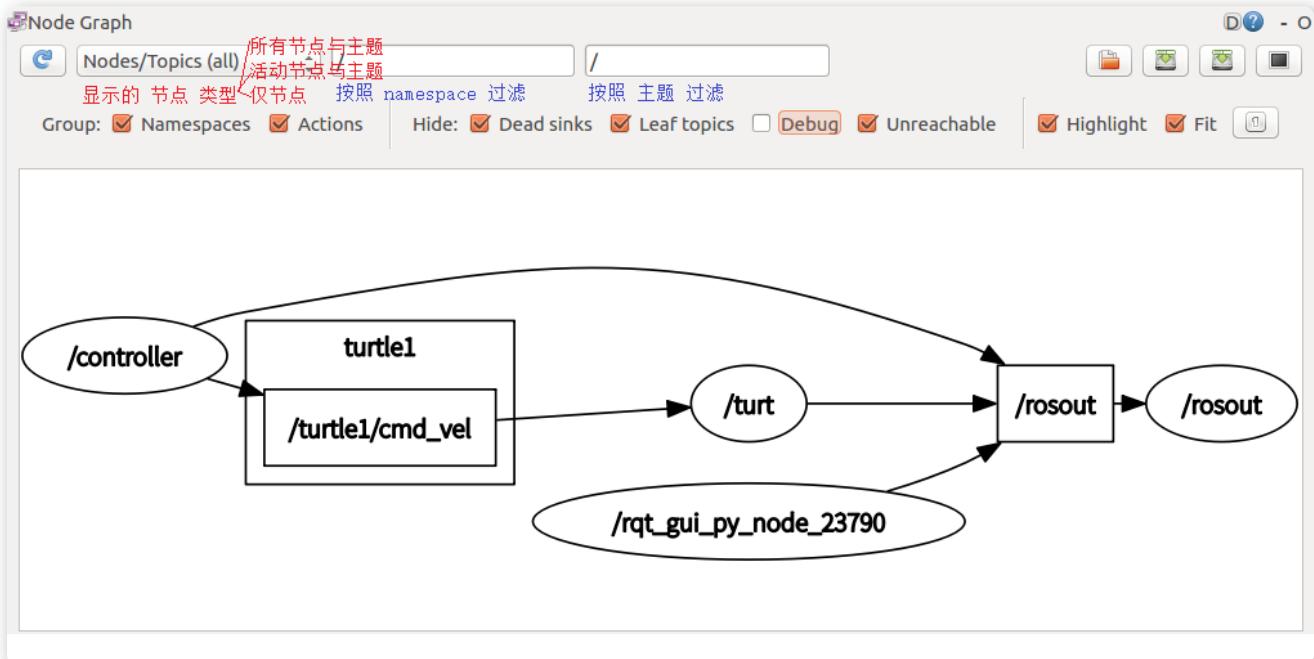
启动 rqt 之后，可以通过 plugins 添加所需的插件



5.3.2 rqt 常用插件:rqt_graph

简介:可视化显示计算图

启动:可以在 rqt 的 plugins 中添加, 或者使用 rqt_graph 启动



5.3.3 rqt 常用插件:rqt_console

简介:rqt_console 是 ROS 中用于显示和过滤日志的图形化插件

准备:编写 Node 节点输出各个级别的日志信息

```

1  /*
2   ROS 节点:输出各种级别的日志信息
3
4  */
5 #include "ros/ros.h"
6
7 int main(int argc, char *argv[])
8 {
9     ros::init(argc, argv, "log_demo");
10    ros::NodeHandle nh;
11
12    ros::Rate r(0.3);
13    while (ros::ok())

```

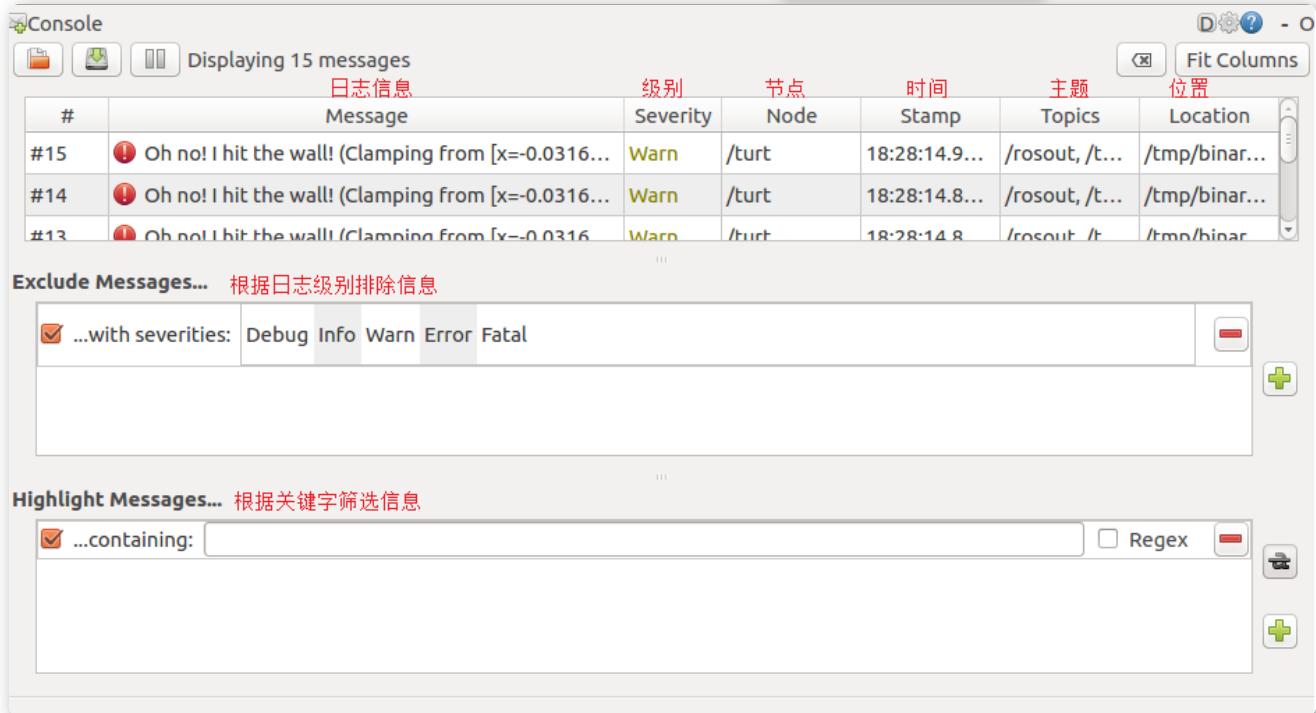
```

14     {
15         ROS_DEBUG("Debug message d");
16         ROS_INFO("Info message oooooooooooooo");
17         ROS_WARN("Warn message wwww");
18         ROS_ERROR("Erroe message
19             EEEEEEEEEEEEEEEE");
20         ROS_FATAL("Fatal message
21             FFFFFFFFFFFFFFFFFFFFFFFFF");
22         r.sleep();
23     }
24
25 }

```

启动:

可以在 rqt 的 plugins 中添加，或者使用 `rqt_console` 启动

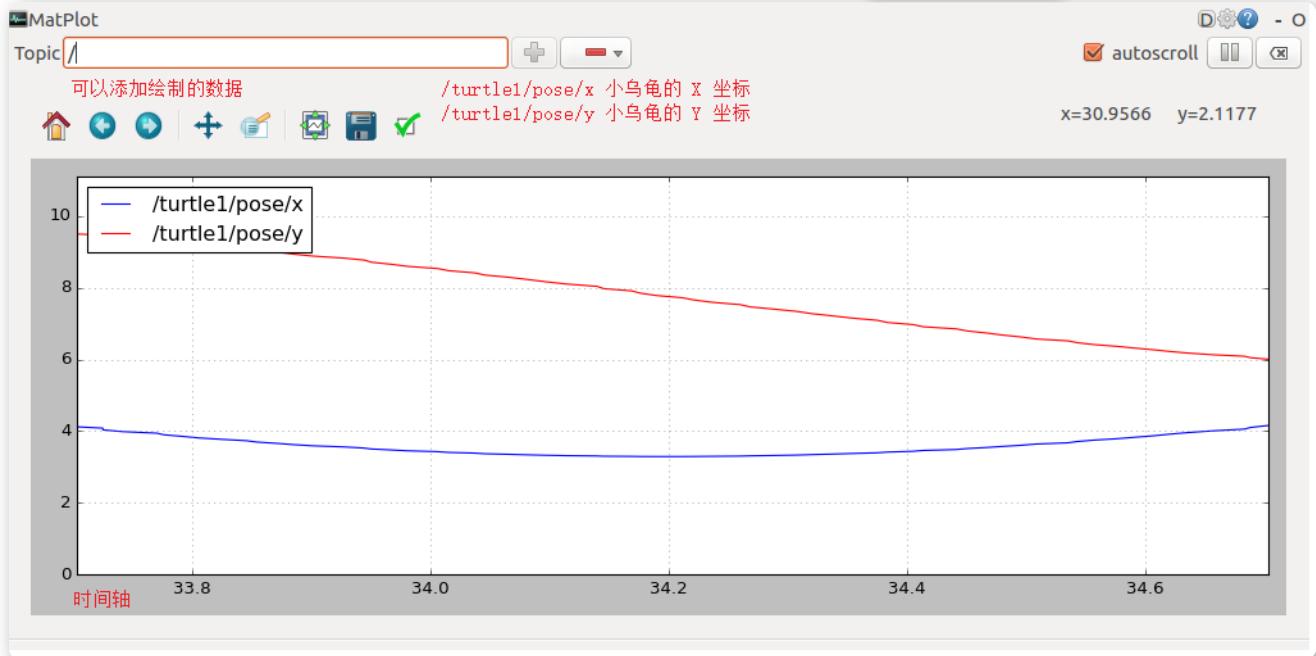


5.3.4 rqt常用插件:rqt_plot

简介:图形绘制插件，可以以 2D 绘图的方式绘制发布在 topic 上的数据

准备:启动 turtlesim 乌龟节点与键盘控制节点，通过 rqt_plot 获取乌龟位姿

启动:可以在 rqt 的 plugins 中添加, 或者使用 `rqt_plot` 启动



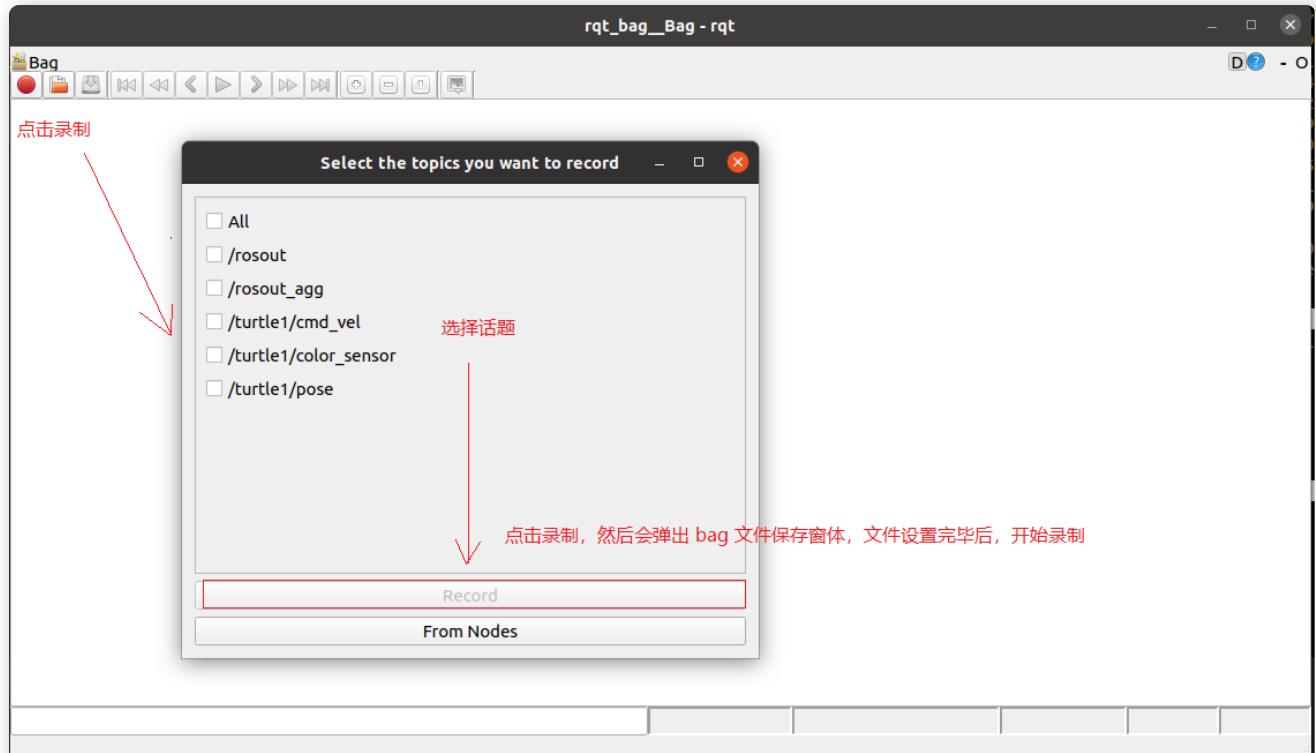
5.3.5 rqt常用插件:rqt_bag

简介:录制和重放 bag 文件的图形化插件

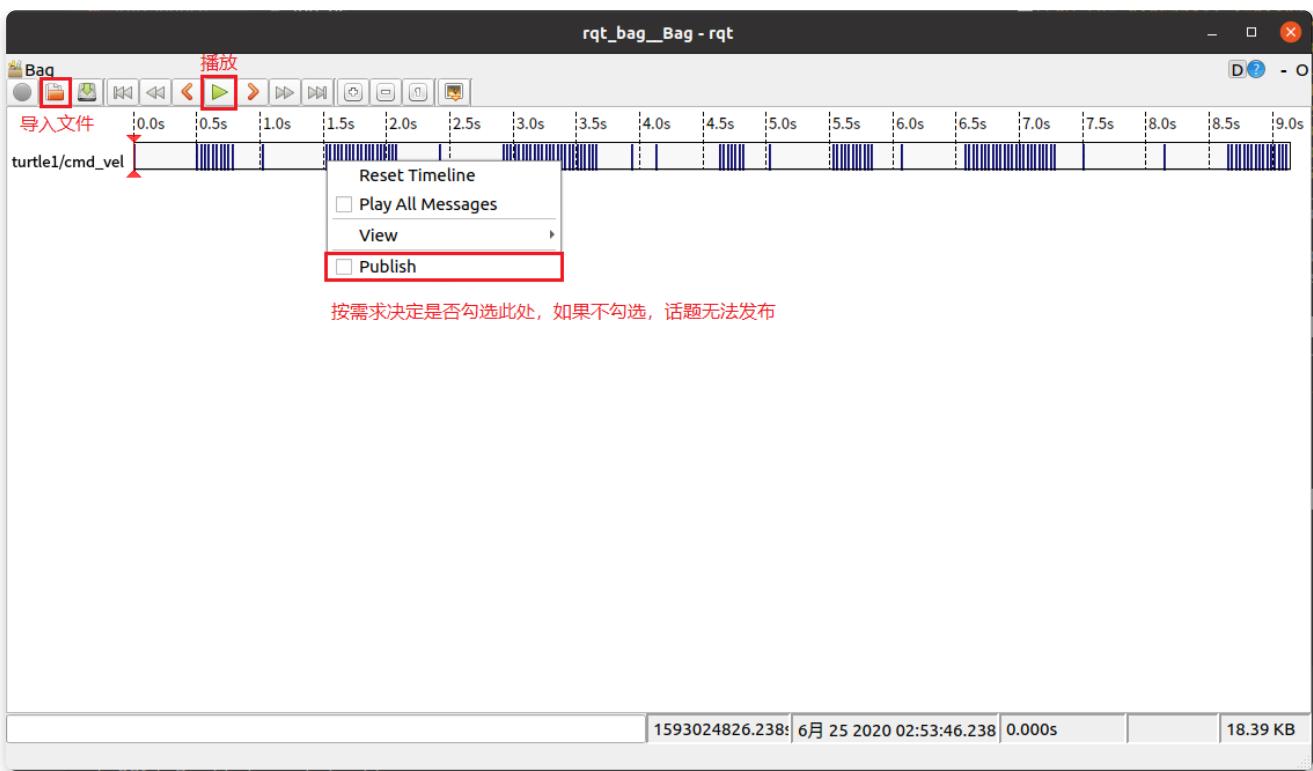
准备:启动 turtlesim 乌龟节点与键盘控制节点

启动:可以在 rqt 的 plugins 中添加, 或者使用 `rqt_bag` 启动

录制:



重放:



5.4 本章小结

本章主要介绍了ROS中的常用组件，内容如下：

- TF坐标变换(重点)
- rosbag 用于ros话题的录制与回放
- rqt工具箱，图形化方式调用组件，提高操作效率以及易用性

其中 TF坐标变换是重点，也是难点，需要大家熟练掌握坐标变换的应用场景以及代码实现。下一章开始将介绍机器人系统仿真，我们将在仿真环境下，创建机器人、控制机器人运动、搭建仿真环境，并以机器人的视角去感知世界。

第6章 机器人系统仿真

对于ROS新手而言，可能会有疑问：学习机器人操作系统，实体机器人是必须的吗？答案是否定的，机器人一般价格不菲，为了降低机器人学习、调试成本，在ROS中提供了系统的机器人仿真实现，通过仿真，可以实现大部分需求，本章主要就是围绕“仿真”展开的，比如，本章会介绍：

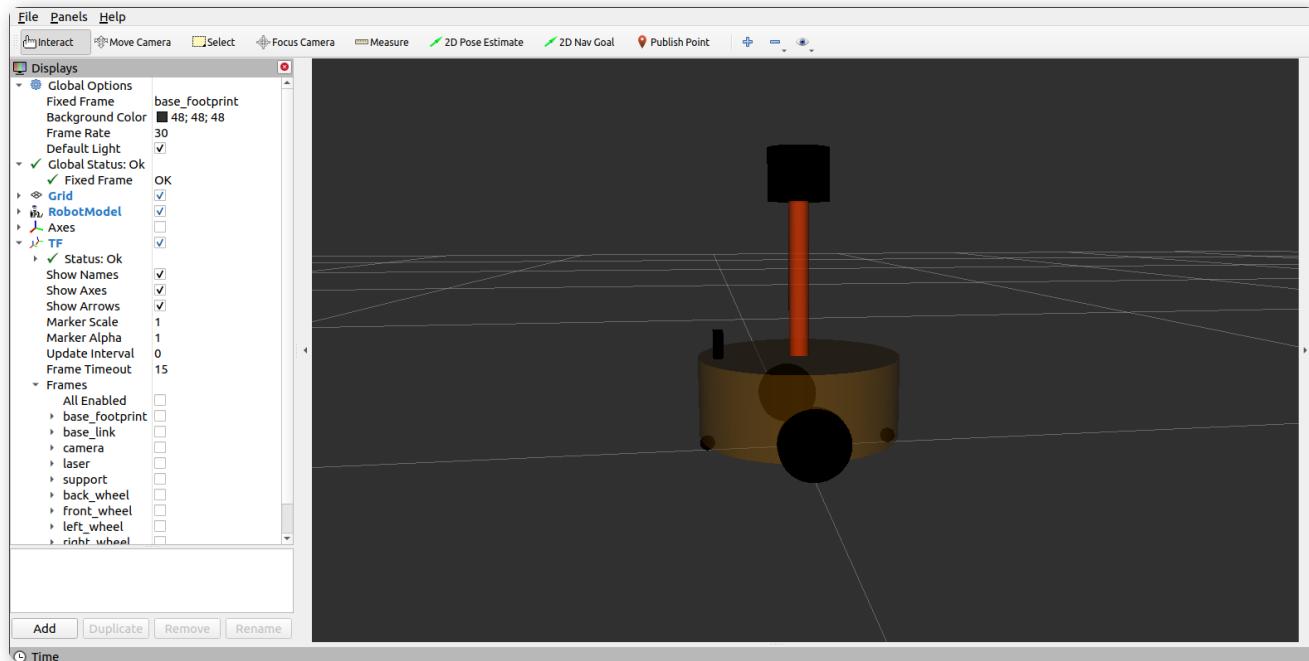
- 如何创建并显示机器人模型；
- 如何搭建仿真环境；
- 如何实现机器人模型与仿真环境的交互。

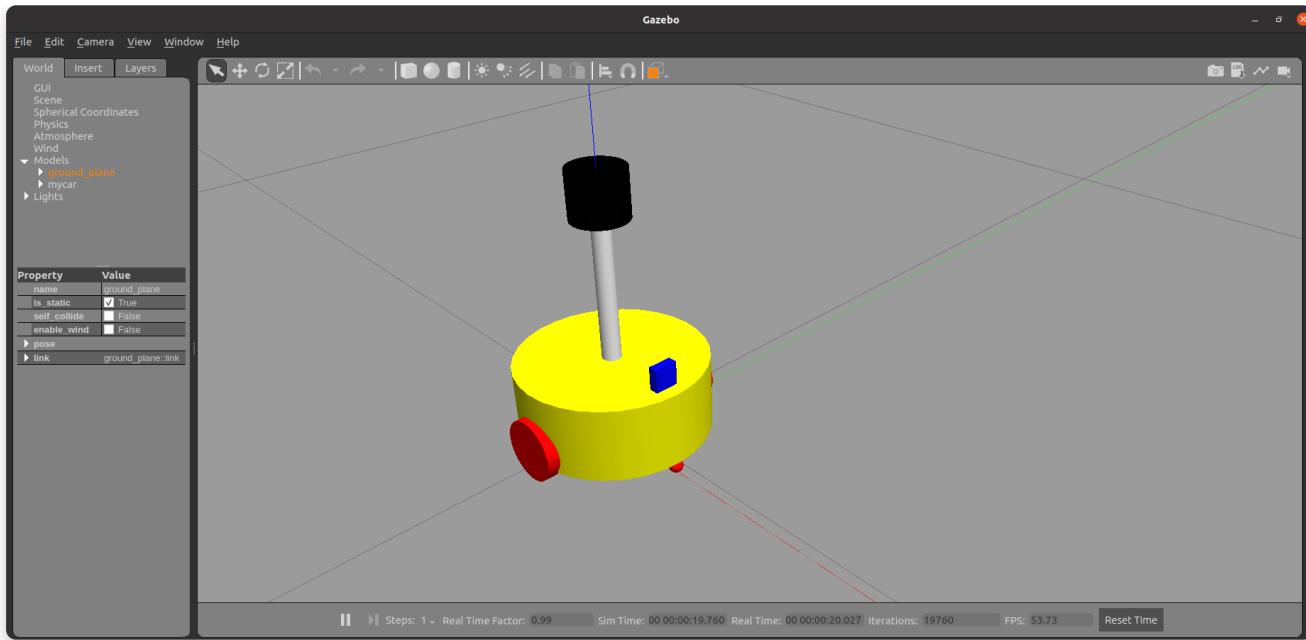
本章预期的学习目标如下：

- 能够独立使用URDF创建机器人模型，并在Rviz和Gazebo中分别显示；
- 能够使用Gazebo搭建仿真环境；
- 能够使用机器人模型中的传感器(雷达、摄像头、编码器...)获取仿真环境数据。

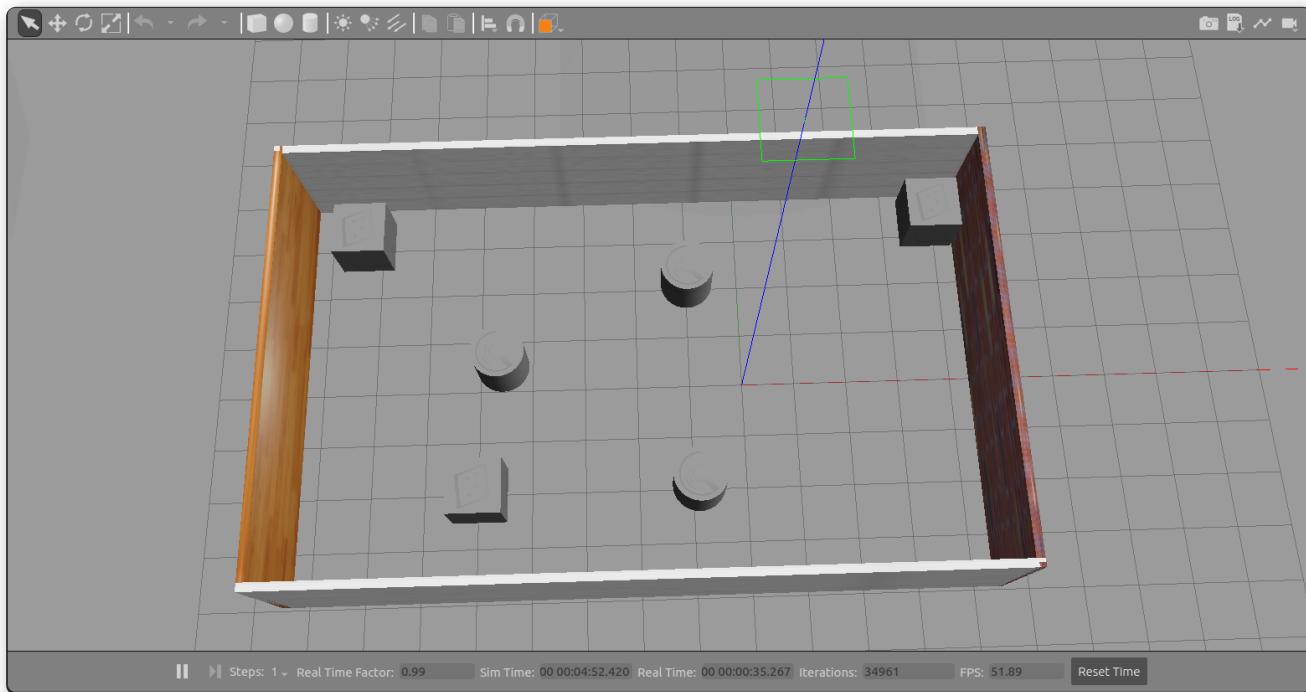
案例演示：

1. 创建并显示机器人模型

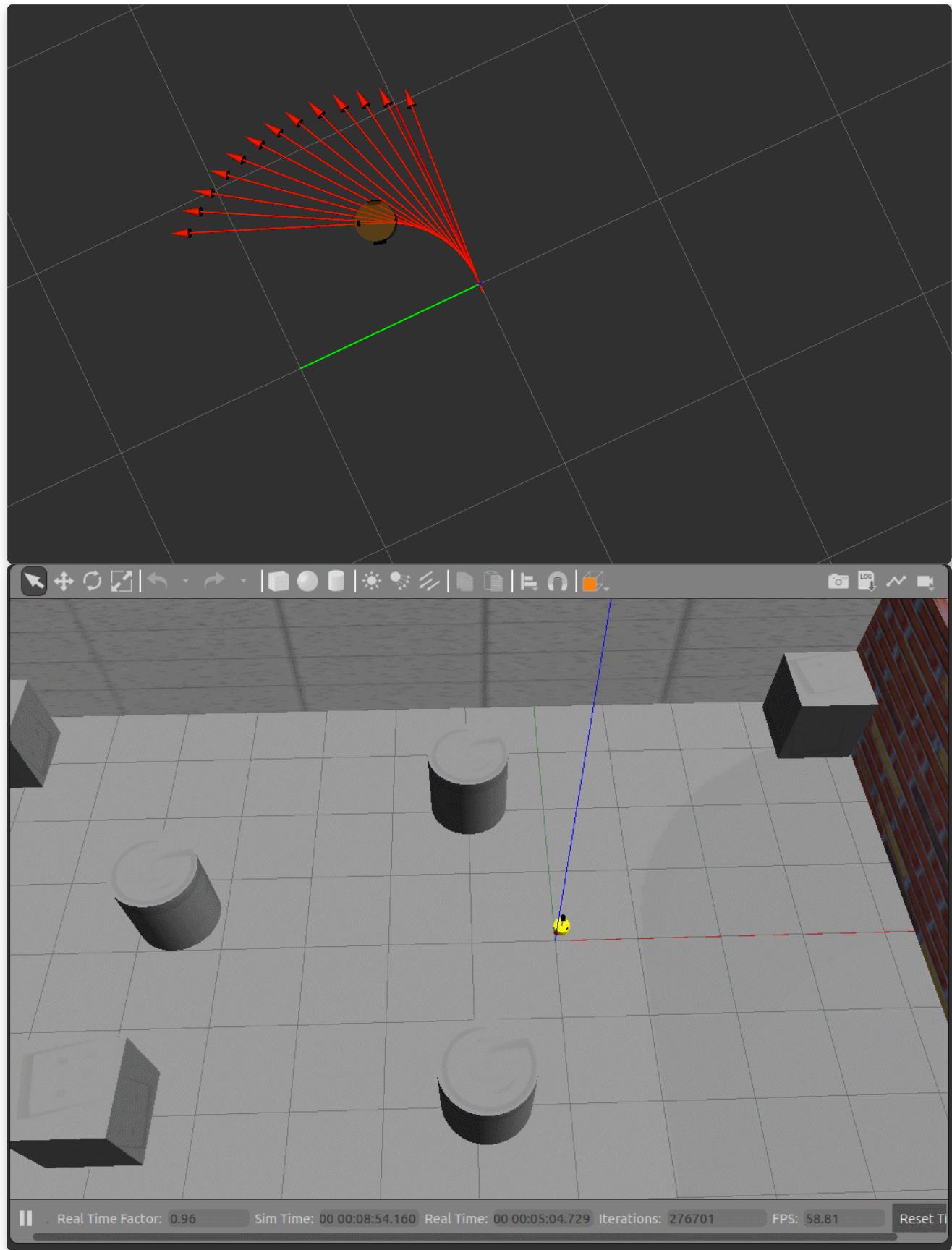




2.仿真环境搭建



3.控制机器人运动



4.雷达仿真

File Panels Help

Interact Move Camera Select Focus Camera Measure 2D Pose Estimate 2D Nav Goal Publish Point

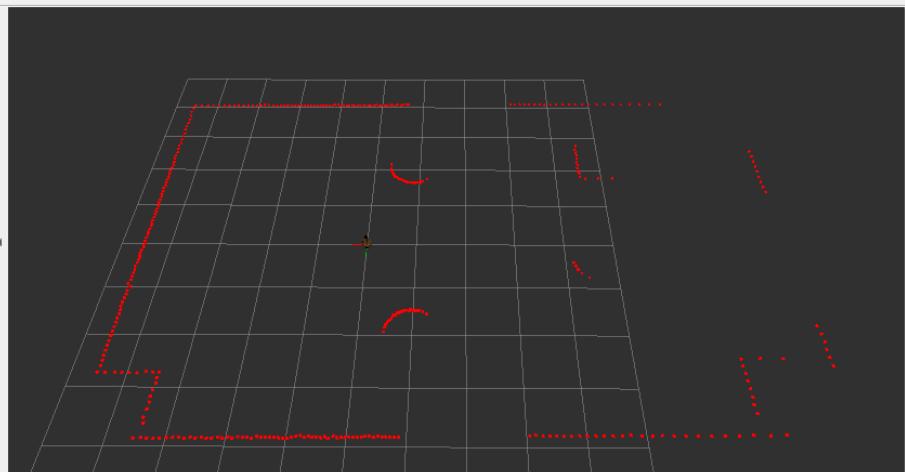
Displays

- Global Options
 - Fixed Frame base_footprint
 - Background Color 48; 48; 48
 - Frame Rate 30
 - Default Light ✓
- ✓ Global Status: Ok
 - ✓ Fixed Frame
 - OK
- Grid
- RobotModel
- LaserScan
- Axes
 - ✓ Status: Ok
 - Reference Frame <Fixed Frame>
 - Length 0.3
 - Radius 0.02
 - Alpha 1

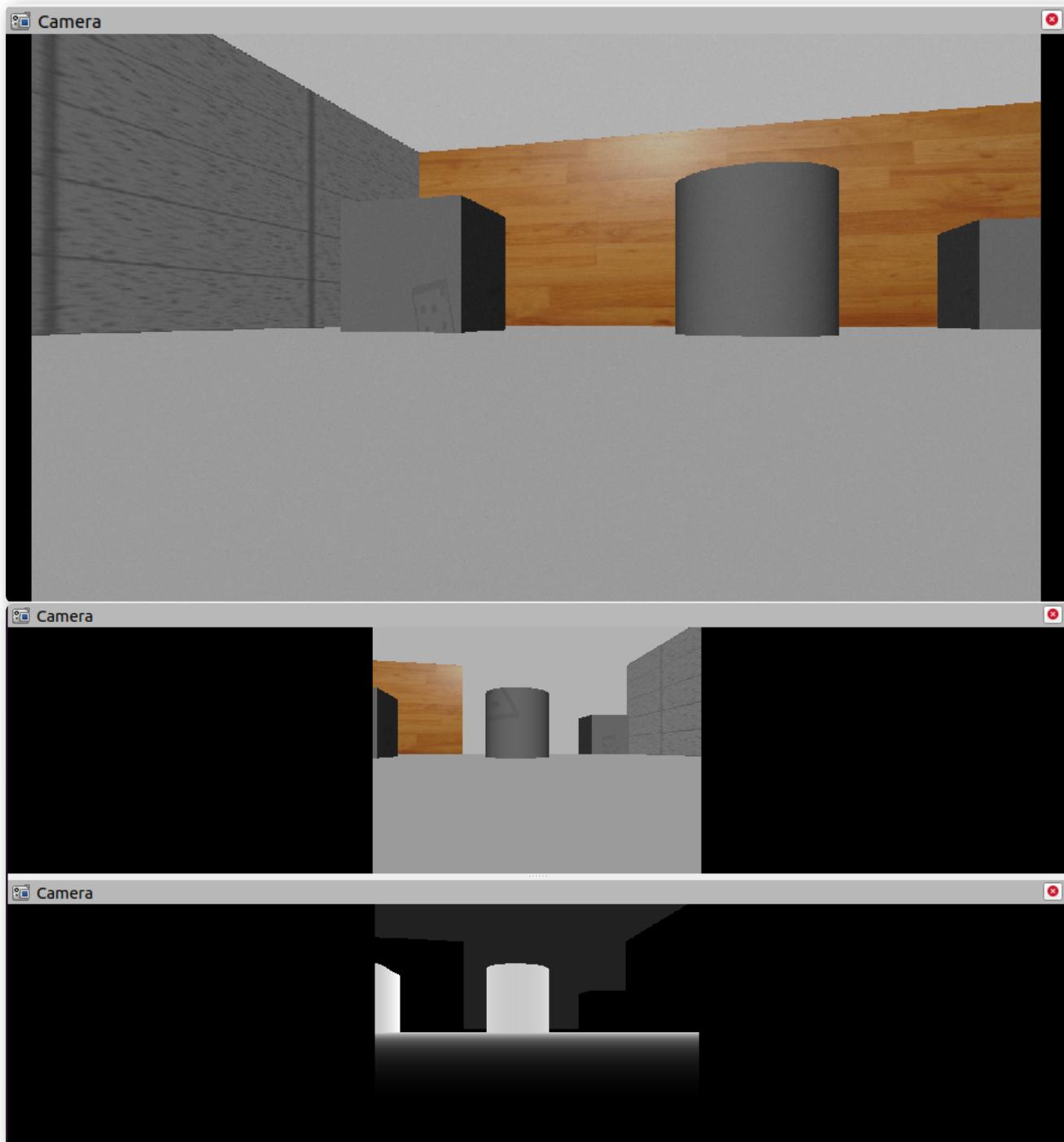
Length

Length of each axis, in meters.

Add Duplicate Remove Rename



5.摄像头仿真



6.1 概述

机器人操作系统学习、开发与测试过程中，会遇到诸多问题，比如：



场景1:机器人一般价格不菲，学习ROS要购买一台机器人吗？

场景2:机器人与之交互的外界环境具有多样性，如何实现复杂的环境设计？

场景3:测试时，直接将未经验证的程序部署到实体机器人运行，安全吗？

...

在诸如此类的场景中，ROS中的仿真就显得尤为重要了。

1.概念

机器人系统仿真：是通过计算机对实体机器人系统进行模拟的技术，在ROS中，仿真实现涉及的内容主要有三:对机器人建模(URDF)、创建仿真环境(Gazebo)以及感知环境(Rviz)等系统性实现。

2.作用

2.1 仿真优势:

仿真在机器人系统研发过程中占有举足轻重的地位，在研发与测试中较之于实体机器人实现，仿真有如下几点的显著优势：

1.低成本:当前机器人成本居高不下，动辄几十万，仿真可以大大降低成本，减小风险

2.高效:搭建的环境更为多样且灵活，可以提高测试效率以及测试覆盖率

3.高安全性:仿真环境下，无需考虑耗损问题

2.2 仿真缺陷:

机器人在仿真环境与实际环境下的表现差异较大，换言之，仿真并不能完全做到模拟真实的物理世界，存在一些"失真"的情况，原因：

1. 仿真器所使用的物理引擎目前还不能够完全精确模拟真实世界的物理情况

2. 仿真器构建的是关节驱动器（电机&齿轮箱）、传感器与信号通信的绝对理想情况，目前不支持模拟实际硬件缺陷或者一些临界状态等情形

3. 相关组件

3.1 URDF

URDF是 Unified Robot Description Format 的首字母缩写，直译为统一(标准化)机器人描述格式，可以以一种 XML 的方式描述机器人的部分结构，比如底盘、摄像头、激光雷达、机械臂以及不同关节的自由度.....,该文件可以被 C++ 内置的解释器转换成可视化的机器人模型，是 ROS 中实现机器人仿真的重要组件

3.2 rviz

RViz 是 ROS Visualization Tool 的首字母缩写，直译为ROS的三维可视化工具。它的主要目的是以三维方式显示ROS消息，可以将数据进行可视化表达。例如:可以显示机器人模型，可以无需编程就能表达激光测距仪 (LRF) 传感器中的传感器到障碍物的距离，RealSense、Kinect或Xtion等三维距离传感器的点云数据 (PCD， Point Cloud Data) ，从相机获取的图像值等

以“ros- [ROS_DISTRO] -desktop-full”命令安装ROS时，RViz会默认被安装。

运行使用命令 `rviz` 或 `rosrun rviz rviz`

如果rviz没有安装，请调用如下命令自行安装：

```
1 sudo apt install ros-[ROS_DISTRO]-rviz
```

3.3gazebo

Gazebo是一款3D动态模拟器，用于显示机器人模型并创建仿真环境，能够在复杂的室内和室外环境中准确有效地模拟机器人。与游戏引擎提供高保真度的视觉模拟类似，Gazebo提供高保真度的物理模拟，其提供一整套传感器模型，以及对用户和程序非常友好的交互方式。

以“ros- [ROS_DISTRO] -desktop-full”命令安装ROS时，gazebo会默认被安装。

运行使用命令 `gazebo` 或 `rosrun gazebo_ros gazebo`

注意1:*在 Ubuntu20.04 与 ROS Noetic 环境下，gazebo 启动异常以及解决*

- **问题1:**VMware: vmw_ioctl_command error Invalid argument(无效的参数)

解决:

```
echo "export SVGA_VGPU10=0" >> ~/.bashrc
source .bashrc
```

- **问题2:**[Err] [REST.cc:205] Error in REST request

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

然后将 `url : https://api.ignitionfuel.org` 使用#注释

再添加 `url: https://api.ignitionrobotics.org`

- **问题3:**启动时抛出异常: [gazebo-2] process has died [pid
xxx, exit code 255, cmd.....]

解决: `killall gzserver` 和 `killall gzclient`

注意2:*如果 gazebo 没有安装，请自行安装:*

1.添加源:

```

1 sudo sh -c 'echo "deb
http://packages.osrfoundation.org/gazebo/ubuntu-
stable `lsb_release -cs` main"
2 >
3 /etc/apt/sources.list.d/gazebo-stable.list'
4 wget http://packages.osrfoundation.org/gazebo.key -O
- | sudo apt-key add -

```

2. 安装：

```

1 sudo apt update
2 sudo apt install gazebo11
3 sudo apt install libgazebo11-dev

```

另请参考：

- <https://wiki.ros.org/urdf>
- <http://wiki.ros.org/rviz>
- http://gazebosim.org/tutorials?tut=ros_overview

课程说明：

机器人的系统仿真是一种集成实现，主要包含三部分：

- URDF 用于创建机器人模型
- Gazebo 用于搭建仿真环境
- Rviz 图形化的显示机器人各种传感器感知到的环境信息

三者应用中，只是创建 URDF 意义不大，一般需要结合 Gazebo 或 Rviz 使用，在 Gazebo 或 Rviz 中可以将 URDF 文件解析为图形化的机器人模型，一般的使用组合为：

- 如果非仿真环境，那么使用 URDF 结合 Rviz 直接显示感知的真实环境信息
- 如果是仿真环境，那么需要使用 URDF 结合 Gazebo 搭建仿真环境，并结合 Rviz 显示感知的虚拟环境信息

后续课程安排:

- 先介绍 URDF 与 Rviz 集成使用，在 Rviz 中只是显示机器人模型，主要用于学习 URDF 语法
- 再介绍 URDF 与 Gazebo 集成，主要学习 URDF 仿真相关语法以及仿真环境搭建
- 最后集成 URDF 与 Gazebo 与 Rviz，实现综合应用

素材链接:

- https://github.com/zx595306686/sim_demo.git

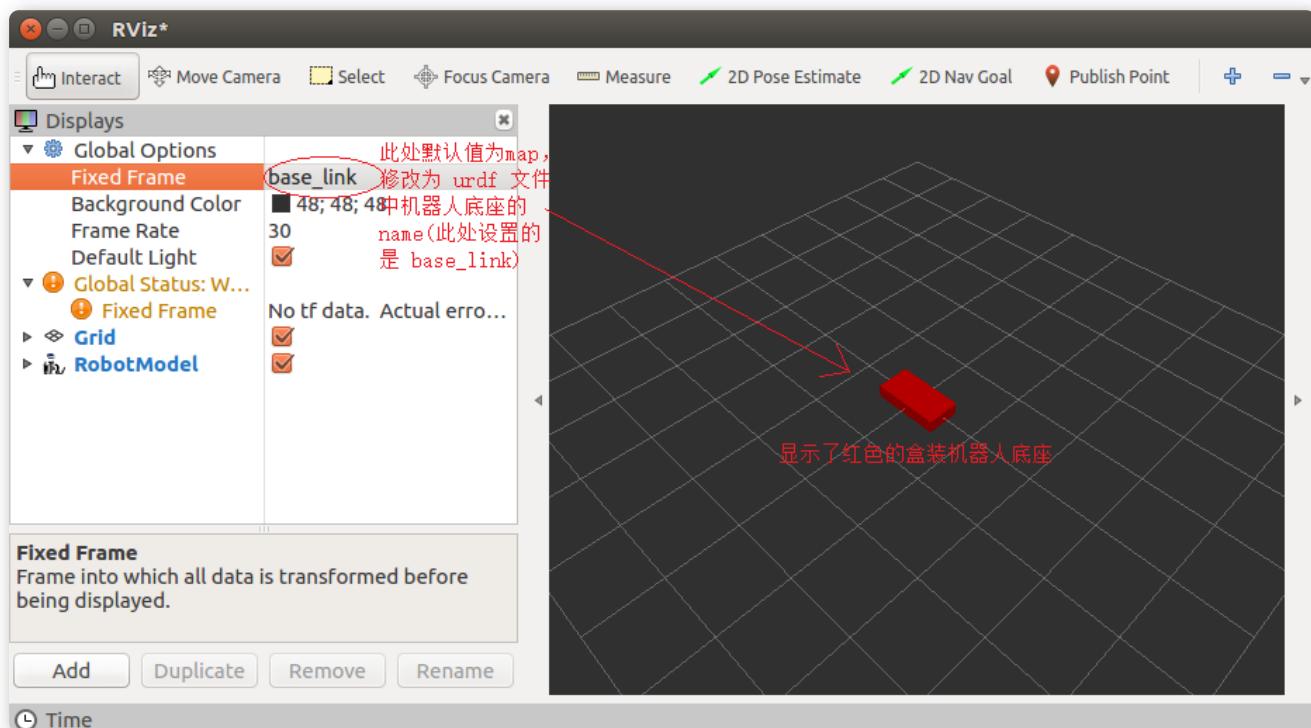
6.2 URDF集成Rviz基本流程

前面介绍过，URDF 不能单独使用，需要结合 Rviz 或 Gazebo，URDF 只是一个文件，需要在 Rviz 或 Gazebo 中渲染成图形化的机器人模型，当前，首先演示URDF与Rviz的集成使用，因为URDF与Rviz的集成较之于URDF与Gazebo的集成更为简单，后期，基于Rviz的集成实现，我们再进一步介绍URDF语法。

需求描述:

在 Rviz 中显示一个盒状机器人

结果演示:



实现流程：

1. 准备:新建功能包，导入依赖
2. 核心:编写 urdf 文件
3. 核心:在 launch 文件集成 URDF 与 Rviz
4. 在 Rviz 中显示机器人模型

1. 创建功能包，导入依赖

创建一个新的功能包，名称自定义，导入依赖包: `urdf` 与 `xacro`

在当前功能包下，再新建几个目录：

`urdf` : 存储 urdf 文件的目录

`meshes` : 机器人模型渲染文件(暂不使用)

`config` : 配置文件

`launch` : 存储 launch 启动文件

2. 编写 URDF 文件

新建一个子级文件夹: `urdf` (可选)，文件夹中添加一个 `.urdf` 文件, 复制如下内容：

```

1 <robot name="mycar">
2   <link name="base_link">
3     <visual>
4       <geometry>
5         <box size="0.5 0.2 0.1" />
6       </geometry>
7     </visual>
8   </link>
9 </robot>

```

3. 在 launch 文件中集成 URDF 与 Rviz

在 launch 目录下，新建一个 launch 文件，该 launch 文件需要启动 Rviz，并导入 urdf 文件，Rviz 启动后可以自动载入解析 urdf 文件，并显示机器人模型，核心问题：如何导入 urdf 文件？在 ROS 中，可以将 urdf 文件的路径设置到参数服务器，使用的参数名是：robot_description，示例代码如下：

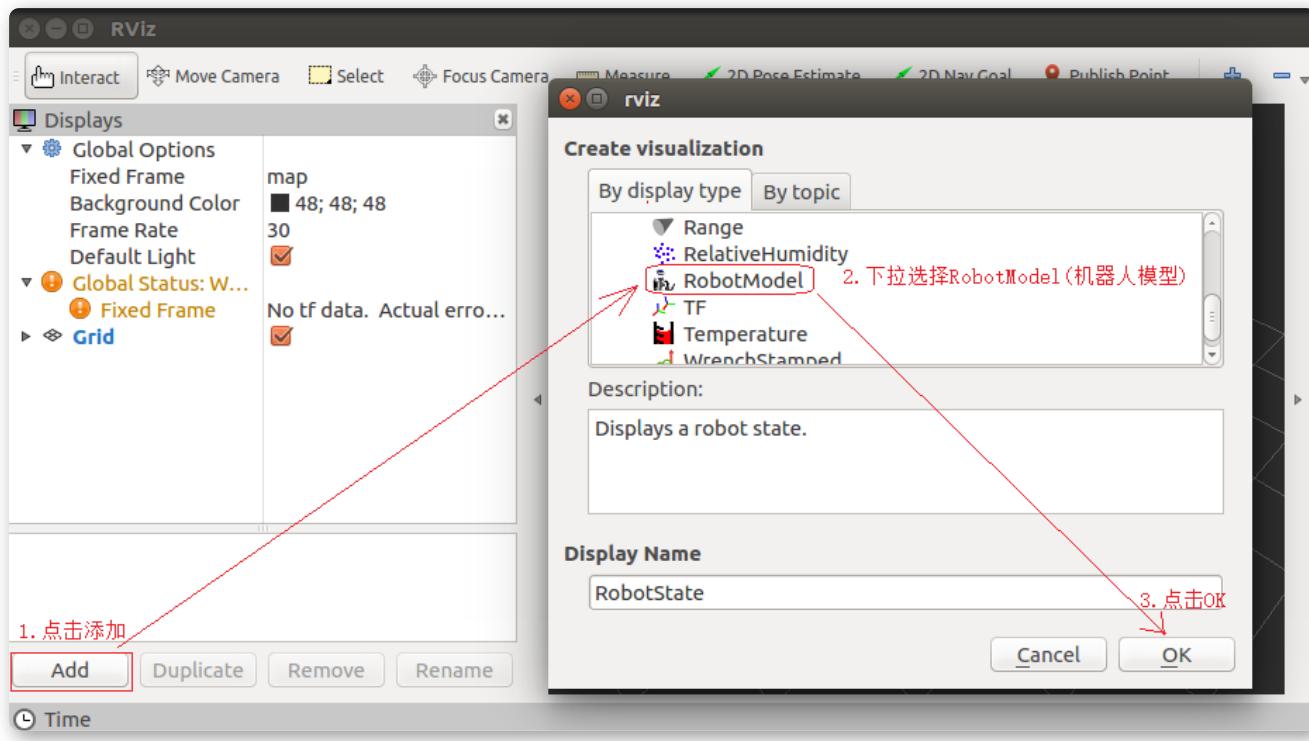
```

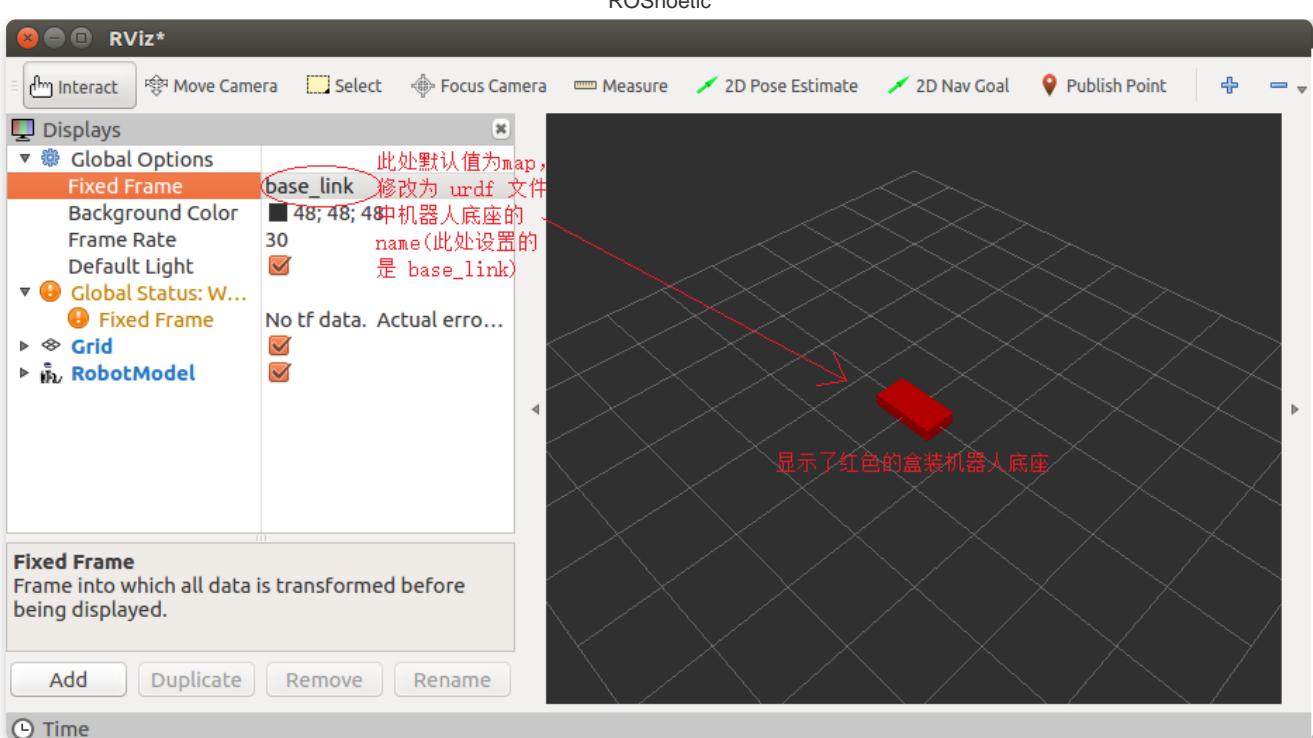
1 <launch>
2
3     <!-- 设置参数 -->
4     <param name="robot_description" textfile="$(find
包名)/urdf/urdf/urdf01_HelloWorld.urdf" />
5
6     <!-- 启动 rviz -->
7     <node pkg="rviz" type="rviz" name="rviz" />
8
9 </launch>

```

4. 在 Rviz 中显示机器人模型

rviz 启动后，会发现并没有盒装的机器人模型，这是因为默认情况下没有添加机器人显示组件，需要手动添加，添加方式如下：



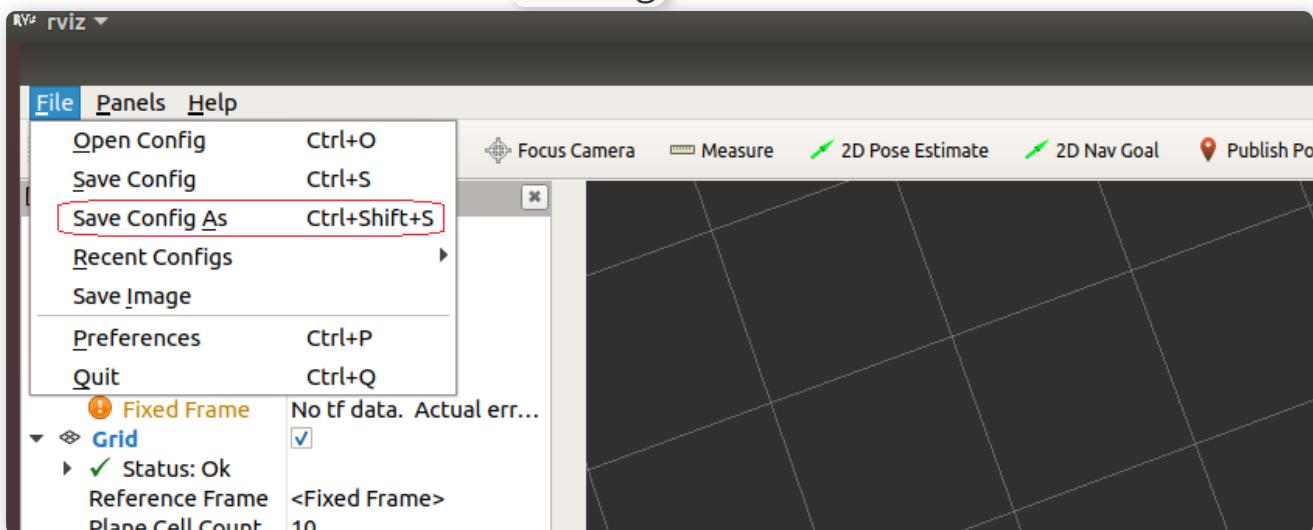


设置完毕后，可以正常显示了

5. 优化 rviz 启动

重复启动 `launch` 文件时，Rviz 之前的组件配置信息不会自动保存，需要重复执行步骤4的操作，为了方便使用，可以使用如下方式优化：

首先，将当前配置保存进 `config` 目录



然后，`launch` 文件中 Rviz 的启动配置添加参数: `args` ,值设置为 `-d` 配置文件路径

```

1 <launch>
2   <param name="robot_description" textfile="$(find
包名)/urdf/urdf/urdf01_HelloWorld.urdf" />
3   <node pkg="rviz" type="rviz" name="rviz" args="-
d $(find 报名)/config/rviz/show_mycar.rviz" />
4 </launch>

```

再启动时，就可以包含之前的组件配置了，使用更方便快捷。

6.3 URDF语法详解

URDF文件是一个标准的XML文件，在ROS中预定义了一系列的标签用于描述机器人模型，机器人模型可能较为复杂，但是ROS的URDF中机器人的组成却是较为简单，可以主要简化为两部分：连杆(link标签)与关节(joint标签)，接下来我们就通过案例了解一下URDF中的不同标签：

- robot 根标签，类似于 launch文件中的launch标签
- link 连杆标签
- joint 关节标签
- gazebo 集成gazebo需要使用的标签

关于gazebo标签，后期在使用gazebo仿真时，才需要使用到，用于配置仿真环境所需参数，比如：机器人材料属性、gazebo插件等，但是该标签不是机器人模型必须的，只有在仿真时才需设置

另请参考：

- <https://wiki.ros.org/urdf/XML>

6.3.1 URDF语法详解01_robot

robot

urdf 中为了保证 xml 语法的完整性，使用了 `robot` 标签作为根标签，所有的 `link` 和 `joint` 以及其他标签都必须包含在 `robot` 标签内，在该标签内可以通过 `name` 属性设置机器人模型的名称

1. 属性

`name`: 指定机器人模型的名称

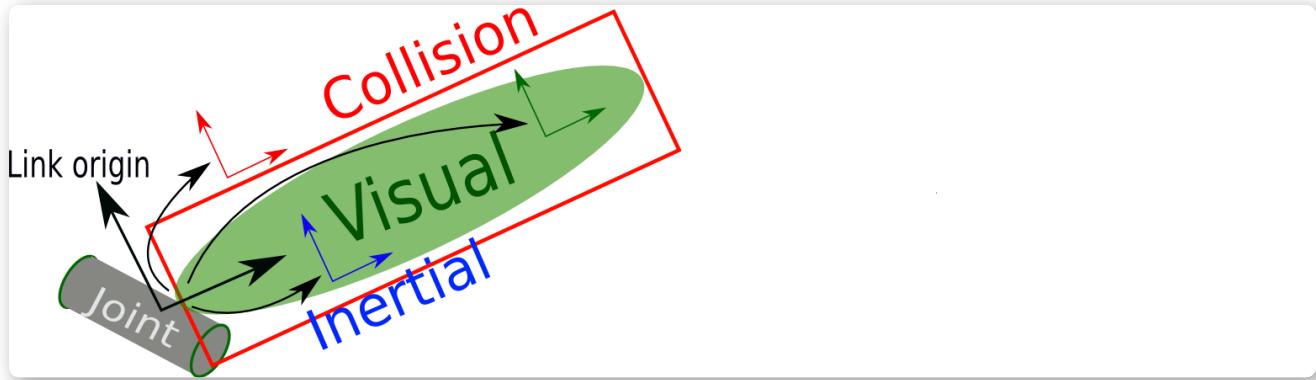
2. 子标签

其他标签都是子级标签

6.3.2 URDF语法详解02_link

link

urdf 中的 `link` 标签用于描述机器人某个部件(也即刚体部分)的外观和物理属性，比如：机器人底座、轮子、激光雷达、摄像头...每一个部件都对应一个 `link`，在 `link` 标签内，可以设计该部件的形状、尺寸、颜色、惯性矩阵、碰撞参数等一系列属性



1. 属性

- `name` --> 为连杆命名

2. 子标签

- `visual` ---> 描述外观(对应的数据是可视的)
 - `geometry` 设置连杆的形状
 - 标签1: `box`(盒状)
 - 属性: `size`=长(x) 宽(y) 高(z)
 - 标签2: `cylinder`(圆柱)
 - 属性: `radius`=半径 `length`=高度
 - 标签3: `sphere`(球体)
 - 属性: `radius`=半径
 - 标签4: `mesh`(为连杆添加皮肤)
 - 属性: `filename`=资源路径(格式:`package:///文件`)
 - `origin` 设置偏移量与倾斜弧度
 - 属性1: `xyz`=x偏移 y便宜 z偏移
 - 属性2: `rpy`=x翻滚 y俯仰 z偏航 (单位是弧度)
 - `material` 设置材料属性(颜色)
 - 属性: `name`
 - 标签: `color`
 - 属性: `rgba`=红绿蓝权重值与透明度 (每个权重值以及透明度取值[0,1])
- `collision` ---> 连杆的碰撞属性
- `Inertial` ---> 连杆的惯性矩阵

在此，只演示 `visual` 使用。

3. 案例

需求:分别生成长方体、圆柱与球体的机器人部件

```

1  <link name="base_link">
2    <visual>
3      <!-- 形状 -->

```

```

4      <geometry>
5          <!-- 长方体的长宽高 -->
6          <!-- <box size="0.5 0.3 0.1" /> -->
7
8          <!-- 圆柱, 半径和长度 -->
9          <!-- <cylinder radius="0.5" -->
10         <!-- 球体, 半径-->
11         <!-- <sphere radius="0.3" /> -->
12
13         </geometry>
14         <!-- xyz坐标 rpy翻滚俯仰与偏航角度
15         (3.14=180度 1.57=90度) -->
16         <origin xyz="0 0 0" rpy="0 0 0" />
17         <!-- 颜色: r=red g=green b=blue a=alpha
18         -->
19         <material name="black">
20             <color rgba="0.7 0.5 0 0.5" />
21         </material>
22     </visual>
23 </link>

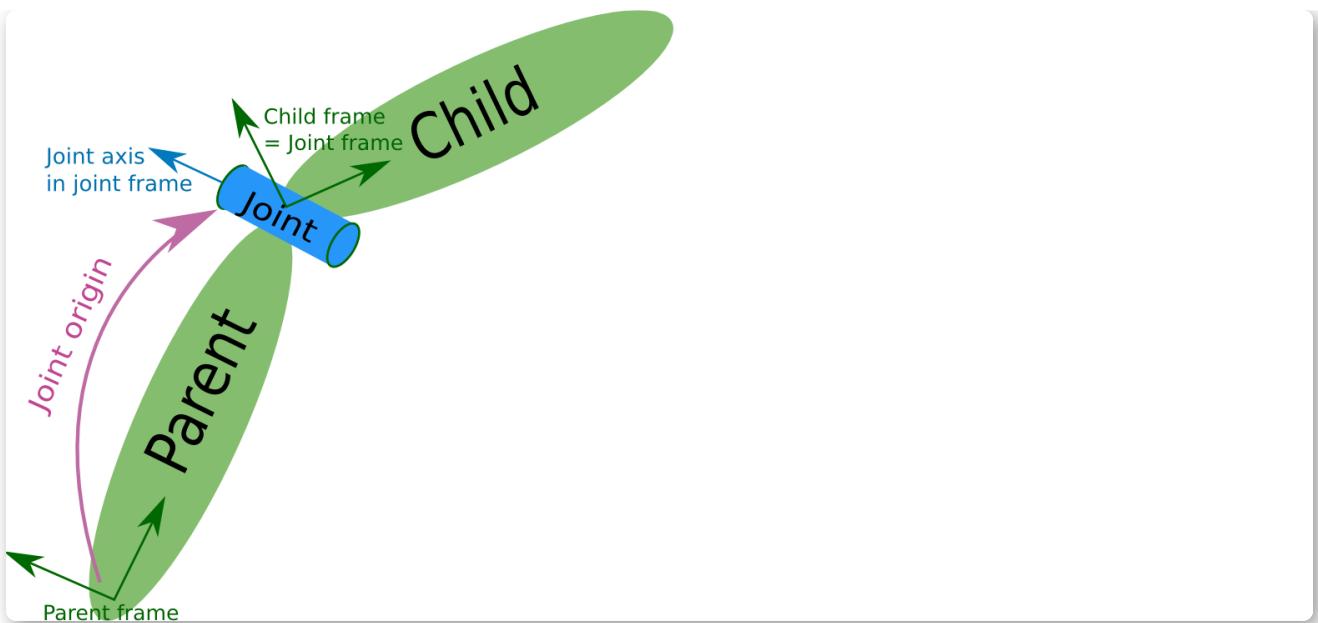
```

6.3.3 URDF语法详解03_joint

joint

urdf 中的 joint 标签用于描述机器人关节的运动学和动力学属性，还可以指定关节运动的安全极限，机器人的两个部件(分别称之为 parent link 与 child link)以"关节"的形式相连接，不同的关节有不同的运动形式: 旋转、滑动、固定、旋转速度、旋转角度限制....,比如:安装在底座上的轮子可以360度旋转，而摄像头则可能是完全固定在底座上。

joint标签对应的数据在模型中是不可见的



1. 属性

- name ---> 为关节命名
- type ---> 关节运动形式
 - continuous: 旋转关节, 可以绕单轴无限旋转
 - revolute: 旋转关节, 类似于 continuous, 但是有旋转角度限制
 - prismatic: 滑动关节, 沿某一轴线移动的关节, 有位置极限
 - planar: 平面关节, 允许在平面正交方向上平移或旋转
 - floating: 浮动关节, 允许进行平移、旋转运动
 - fixed: 固定关节, 不允许运动的特殊关节

2. 子标签

- parent(必需的)

parent link的名字是一个强制的属性:

- link: 父级连杆的名字, 是这个link在机器人结构树中的名字。

- child(必需的)

child link的名字是一个强制的属性:

- link: 子级连杆的名字, 是这个link在机器人结构树中的名字。

- origin

- 属性: xyz=各轴线上的偏移量 rpy=各轴线上的偏移弧度。
- axis
 - 属性: xyz用于设置围绕哪个关节轴运动。

3.案例

需求:创建机器人模型，底盘为长方体，在长方体的前面添加一摄像头，摄像头可以沿着 Z 轴 360 度旋转。

URDF文件示例如下:

```

1 <!--
2     需求： 创建机器人模型， 底盘为长方体，
3             在长方体的前面添加一摄像头，
4             摄像头可以沿着 Z 轴 360 度旋转
5
6     -->
7 <robot name="mycar">
8     <!-- 底盘 -->
9     <link name="base_link">
10    <visual>
11        <geometry>
12            <box size="0.5 0.2 0.1" />
13        </geometry>
14        <origin xyz="0 0 0" rpy="0 0 0" />
15        <material name="blue">
16            <color rgba="0 0 1.0 0.5" />
17        </material>
18    </visual>
19 </link>
20
21     <!-- 摄像头 -->
22 <link name="camera">
23     <visual>
24         <geometry>
25             <box size="0.02 0.05 0.05" />
26         </geometry>
27         <origin xyz="0 0 0" rpy="0 0 0" />

```

```
28         <material name="red">
29             <color rgba="1 0 0 0.5" />
30         </material>
31     </visual>
32 </link>
33
34     <!-- 关节 -->
35     <joint name="camera2baselink"
36     type="continuous">
37         <parent link="base_link"/>
38         <child link="camera" />
39         <!-- 需要计算两个 link 的物理中心之间的偏移量 --
40         >
41             <origin xyz="0.2 0 0.075" rpy="0 0 0" />
42             <axis xyz="0 0 1" />
43         </joint>
44
45 </robot>
```

launch文件示例如下：

```

1 <launch>
2
3     <param name="robot_description"
4       textfile="$(find
5         urdf_rviz_demo)/urdf/urdf/urdf03_joint.urdf" />
6
7     <node pkg="rviz" type="rviz" name="rviz"
8       args="-d $(find
9         urdf_rviz_demo)/config/helloworld.rviz" />
10
11    <!-- 添加关节状态发布节点 -->
12    <node pkg="joint_state_publisher"
13      type="joint_state_publisher"
14      name="joint_state_publisher" />
15
16    <!-- 添加机器人状态发布节点 -->
17    <node pkg="robot_state_publisher"
18      type="robot_state_publisher"
19      name="robot_state_publisher" />
20
21    <!-- 可选:用于控制关节运动的节点 -->
22    <node pkg="joint_state_publisher_gui"
23      type="joint_state_publisher_gui"
24      name="joint_state_publisher_gui" />
25
26
27 </launch>

```

PS:

1.状态发布节点在此是必须的:

```

1     <!-- 添加关节状态发布节点 -->
2     <node pkg="joint_state_publisher"
3       type="joint_state_publisher"
4       name="joint_state_publisher" />
5
6     <!-- 添加机器人状态发布节点 -->
7     <node pkg="robot_state_publisher"
8       type="robot_state_publisher"
9       name="robot_state_publisher" />

```

2.关节运动控制节点(可选)，会生成关节控制的UI，用于测试关节运动是否正常。

```

1      <!-- 可选:用于控制关节运动的节点 -->
2      <node pkg="joint_state_publisher_gui"
3          type="joint_state_publisher_gui"
4          name="joint_state_publisher_gui" />

```

4.base_footprint优化urdf

前面实现的机器人模型是半沉到地下的，因为默认情况下: 底盘的中心点位于地图原点上，所以会导致这种情况产生，可以使用的优化策略，将初始 link 设置为一个尺寸极小的 link(比如半径为 0.001m 的球体，或边长为 0.001m 的立方体)，然后再在初始 link 上添加底盘等刚体，这样实现，虽然仍然存在初始link半沉的现象，但是基本可以忽略了。这个初始 link 一般称之为 `base_footprint`

```

1  <!--
2
3      使用 base_footprint 优化
4
5  -->
6 <robot name="mycar">
7      <!-- 设置一个原点(机器人中心点的投影) -->
8      <link name="base_footprint">
9          <visual>
10         <geometry>
11             <sphere radius="0.001" />
12         </geometry>
13     </visual>
14 </link>
15
16     <!-- 添加底盘 -->
17     <link name="base_link">
18         <visual>
19             <geometry>
20                 <box size="0.5 0.2 0.1" />

```

```

21          </geometry>
22          <origin xyz="0 0 0" rpy="0 0 0" />
23          <material name="blue">
24              <color rgba="0 0 1.0 0.5" />
25          </material>
26      </visual>
27  </link>
28
29      <!-- 底盘与原点连接的关节 -->
30      <joint name="base_link2base_footprint"
31      type="fixed">
32          <parent link="base_footprint" />
33          <child link="base_link" />
34          <origin xyz="0 0 0.05" />
35      </joint>
36
37      <!-- 添加摄像头 -->
38      <link name="camera">
39          <visual>
40              <geometry>
41                  <box size="0.02 0.05 0.05" />
42              </geometry>
43              <origin xyz="0 0 0" rpy="0 0 0" />
44              <material name="red">
45                  <color rgba="1 0 0 0.5" />
46              </material>
47          </visual>
48      </link>
49      <!-- 关节 -->
50      <joint name="camera2baselink"
51      type="continuous">
52          <parent link="base_link"/>
53          <child link="camera" />
54          <origin xyz="0.2 0 0.075" rpy="0 0 0" />
55          <axis xyz="0 0 1" />
56      </joint>
57
58  </robot>

```

launch 文件内容不变。

5.遇到问题以及解决

问题1:

命令行输出如下错误提示

- 1 UnicodeEncodeError: 'ascii' codec can't encode characters in position 463-464: ordinal not in range(128)
- 2 [joint_state_publisher-3] process has died [pid 4443, exit code 1, cmd /opt/ros/melodic/lib/joint_state_publisher/joint_state_publisher __name:=joint_state_publisher __log:=/home/rosmelodic/.ros/log/b38967c0-0acb-11eb-aee3-0800278ee10c/joint_state_publisher-3.log].
- 3 log file: /home/rosmelodic/.ros/log/b38967c0-0acb-11eb-aee3-0800278ee10c/joint_state_publisher-3*.log

rviz中提示坐标变换异常，导致机器人部件显示结构异常

原因:编码问题导致的

解决:去除URDF中的中文注释

问题2:[ERROR] [1584370263.037038]: Could not find the GUI, install the 'joint_state_publisher_gui' package

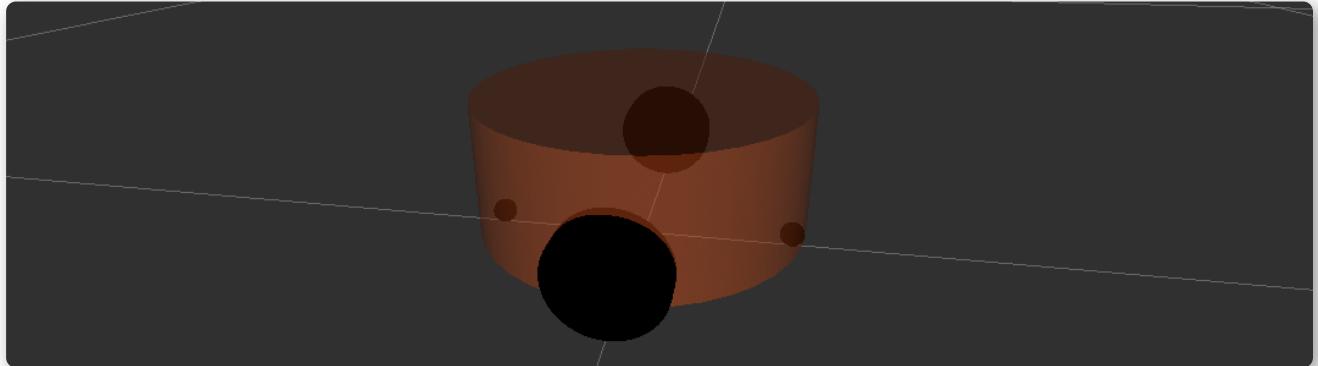
解决: `sudo apt install ros-noetic-joint-state-publisher-gui`

6.3.4 URDF练习

需求描述:

创建一个四轮圆柱状机器人模型，机器人参数如下,底盘为圆柱状，半径10cm，高 8cm，四轮由两个驱动轮和两个万向支撑轮组成，两个驱动轮半径为 3.25cm,轮胎宽度1.5cm，两个万向轮为球状，半径 0.75cm，底盘离地间距为 1.5cm(与万向轮直径一致)

结果演示:



实现流程:

创建机器人模型可以分步骤实现

1. 新建 urdf 文件，并与 launch 文件集成
2. 搭建底盘
3. 在底盘上添加两个驱动轮
4. 在底盘上添加两个万向轮

1.新建urdf以及launch文件

urdf 文件:基本实现

```

1 <robot name="mycar">
2     <!-- 设置 base_footprint -->
3     <link name="base_footprint">
4         <visual>
5             <geometry>
6                 <sphere radius="0.001" />
7             </geometry>
8         </visual>
9     </link>
10
11     <!-- 添加底盘 -->
12
13

```

```

14      <!-- 添加驱动轮 -->
15
16
17      <!-- 添加万向轮(支撑轮) -->
18
19  </robot>

```

launch 文件:

```

1 <launch>
2      <!-- 将 urdf 文件内容设置进参数服务器 -->
3      <param name="robot_description"
4          textfile="$(find
5              demo01_urdf_helloworld)/urdf/urdf/test.urdf" />
6
7      <!-- 启动 rviz -->
8      <node pkg="rviz" type="rviz" name="rviz_test"
9          args="-d $(find
10             demo01_urdf_helloworld)/config/helloworld.rviz" />
11
12      <!-- 启动机器人状态和关节状态发布节点 -->
13      <node pkg="robot_state_publisher"
14          type="robot_state_publisher"
15          name="robot_state_publisher" />
16
17      <node pkg="joint_state_publisher"
18          type="joint_state_publisher"
19          name="joint_state_publisher" />
20
21      <!-- 启动图形化的控制关节运动节点 -->
22      <node pkg="joint_state_publisher_gui"
23          type="joint_state_publisher_gui"
24          name="joint_state_publisher_gui" />
25
26  </launch>

```

2. 底盘搭建

```

1  <!--
2      参数
3          形状:圆柱
4          半径:10      cm
5          高度:8       cm
6          离地:1.5      cm
7
8      -->
9      <link name="base_link">
10     <visual>
11         <geometry>
12             <cylinder radius="0.1"
13             length="0.08" />
14         </geometry>
15         <origin xyz="0 0 0" rpy="0 0 0" />
16         <material name="yellow">
17             <color rgba="0.8 0.3 0.1 0.5" />
18         </material>
19     </visual>
20
21     <joint name="base_link2base_footprint"
22         type="fixed">
23         <parent link="base_footprint" />
24         <child link="base_link"/>
25         <origin xyz="0 0 0.055" />

```

3. 添加驱动轮

```

1  <!-- 添加驱动轮 -->
2      <!--
3          驱动轮是侧翻的圆柱
4          参数
5          半径: 3.25 cm

```

```

6           宽度: 1.5  cm
7           颜色: 黑色
8           关节设置:
9           x = 0
10          y = 底盘的半径 + 轮胎宽度 / 2
11          z = 离地间距 + 底盘长度 / 2 - 轮胎半径 =
12          1.5 + 4 - 3.25 = 2.25(cm)
13          axis = 0 1 0
14          -->
15          <link name="left_wheel">
16              <visual>
17                  <geometry>
18                      <cylinder radius="0.0325"
19                      length="0.015" />
20                  </geometry>
21                  <origin xyz="0 0 0" rpy="1.5705 0 0"
22                  />
23                  <material name="black">
24                      <color rgba="0.0 0.0 0.0 1.0" />
25                  </material>
26              </visual>
27          </link>
28
29          <joint name="left_wheel2base_link"
30          type="continuous">
31              <parent link="base_link" />
32              <child link="left_wheel" />
33              <origin xyz="0 0.1 -0.0225" />
34              <axis xyz="0 1 0" />
35          </joint>
36
37
38          <link name="right_wheel">
39              <visual>

```

```

40          <origin xyz="0 0 0" rpy="1.5705 0 0"
41      />
42          <material name="black">
43              <color rgba="0.0 0.0 0.0 1.0" />
44          </material>
45      </visual>
46  </link>
47
48      <joint name="right_wheel2base_link"
49      type="continuous">
50          <parent link="base_link" />
51          <child link="right_wheel" />
52          <origin xyz="0 -0.1 -0.0225" />
53          <axis xyz="0 1 0" />
54      </joint>

```

4.添加万向轮

```

1  <!-- 添加万向轮(支撑轮) -->
2
3      <!--
4          参数
5              形状: 球体
6              半径: 0.75 cm
7              颜色: 黑色
8
9          关节设置:
10         x = 自定义(底盘半径 - 万向轮半径) = 0.1 -
11         0.0075 = 0.0925(cm)
12         y = 0
13         z = 底盘长度 / 2 + 离地间距 / 2 = 0.08 /
14         2 + 0.015 / 2 = 0.0475
15         axis= 1 1 1
16
17     -->
18     <link name="front_wheel">
19         <visual>
20             <geometry>

```

```
18             <sphere radius="0.0075" />
19         </geometry>
20         <origin xyz="0 0 0" rpy="0 0 0" />
21         <material name="black">
22             <color rgba="0.0 0.0 0.0 1.0" />
23         </material>
24     </visual>
25 </link>
26
27     <joint name="front_wheel2base_link"
28         type="continuous">
29         <parent link="base_link" />
30         <child link="front_wheel" />
31         <origin xyz="0.0925 0 -0.0475" />
32         <axis xyz="1 1 1" />
33     </joint>
34
35     <link name="back_wheel">
36         <visual>
37             <geometry>
38                 <sphere radius="0.0075" />
39             </geometry>
40             <origin xyz="0 0 0" rpy="0 0 0" />
41             <material name="black">
42                 <color rgba="0.0 0.0 0.0 1.0" />
43             </material>
44         </visual>
45     </link>
46
47     <joint name="back_wheel2base_link"
48         type="continuous">
49         <parent link="base_link" />
50         <child link="back_wheel" />
51         <origin xyz="-0.0925 0 -0.0475" />
52         <axis xyz="1 1 1" />
53     </joint>
```

思考:

- 上述代码实现存在什么问题吗？比如复用性！

6.3.5 URDF工具

在 ROS 中，提供了一些工具来方便 URDF 文件的编写，比如：

- `check_urdf` 命令可以检查复杂的 urdf 文件是否存在语法问题
- `urdf_to_graphviz` 命令可以查看 urdf 模型结构，显示不同 link 的层级关系

当然，要使用工具之前，首先需要安装，安装命令：
`sudo apt install liburdfdom-tools`

1. `check_urdf` 语法检查

进入urdf文件所属目录，调用：`check_urdf urdf文件`，如果不抛出异常，说明文件合法，否则非法

```
ubuntu@Turtlebot3-virtual-machine:~/myros_demo/demo01_URDF/src/my_urdf02/urdf$ check_urdf mybot_test.urdf
robot name is: mycar
----- Successfully Parsed XML -----
root Link: base_link has 4 child(ren)
  child(1): left_back_wheel
  child(2): left_front_wheel
  child(3): right_back_wheel
  child(4): right_front_wheel
```

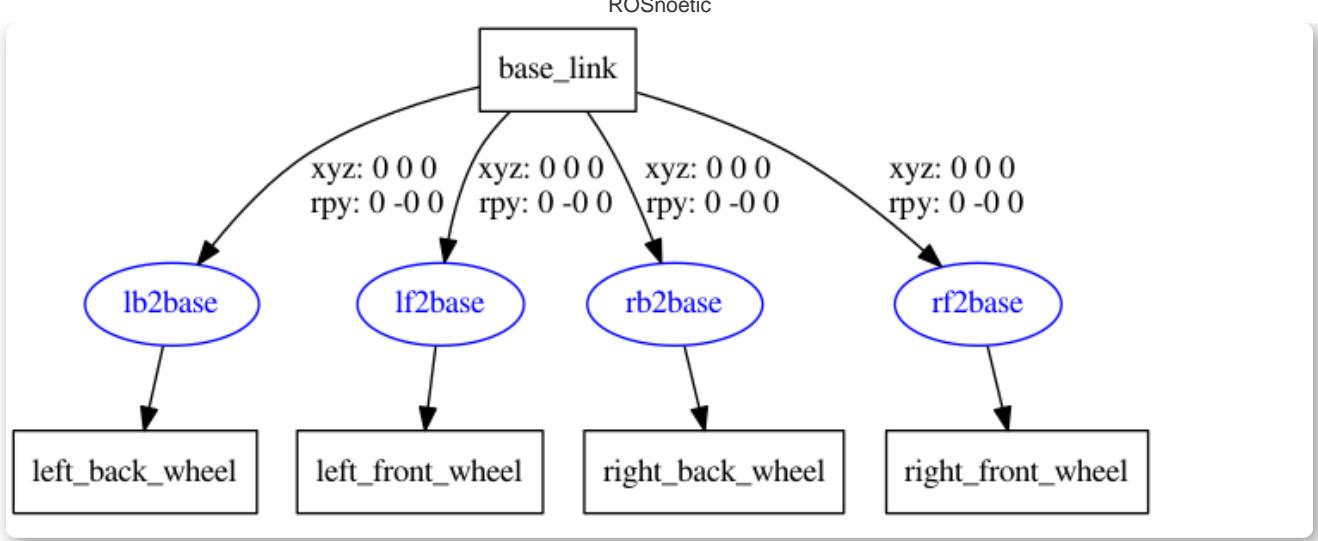
合法的 urdf 文件

```
ubuntu@Turtlebot3-virtual-machine:~/myros_demo/demo01_URDF/src/my_urdf02/urdf$ check_urdf mybot_test.urdf
Error:  joint [lf2base] has no type, check to see if it's a reference.
        at line 441 in /build/urdfdom-UJ3kd6/urdfdom-0.4.1/urdf_parser/src/join
.cpp
Error:  joint xml is not initialized correctly
        at line 207 in /build/urdfdom-UJ3kd6/urdfdom-0.4.1/urdf_parser/src/mod
e.cpp
ERROR: Model Parsing the xml failed
```

非法的 URDF 文件

2. `urdf_to_graphviz` 结构查看

进入urdf文件所属目录，调用：`urdf_to_graphviz urdf文件`，当前目录下会生成 pdf 文件



6.4 URDF优化_xacro

前面 URDF 文件构建机器人模型的过程中，存在若干问题。



问题1:在设计关节的位置时，需要按照一定的公式计算，公式是固定的，但是在 URDF 中依赖于人工计算，存在不便，容易计算失误，且当某些参数发生改变时，还需要重新计算。

问题2:URDF 中的部分内容是高度重复的，驱动轮与支撑轮的设计实现，不同轮子只是部分参数不同，形状、颜色、翻转量都是一致的，在实际应用中，构建复杂的机器人模型时，更是易于出现高度重复的设计，按照一般的编程涉及到重复代码应该考虑封装。

.....

如果在编程语言中，可以通过变量结合函数直接解决上述问题，在 ROS 中，已经给出了类似编程的优化方案，称之为:Xacro

概念

Xacro 是 XML Macros 的缩写，Xacro 是一种 XML 宏语言，是可编程的 XML。

原理

Xacro 可以声明变量，可以通过数学运算求解，使用流程控制控制执行顺序，还可以通过类似函数的实现，封装固定的逻辑，将逻辑中需要的可变的数据以参数的方式暴露出去，从而提高代码复用率以及程序的安全性。

作用

较之于纯粹的 URDF 实现，可以编写更安全、精简、易读性更强的机器人模型文件，且可以提高编写效率。

另请参考:

- <http://wiki.ros.org/xacro>

6.4.1 Xacro_快速体验

目的:简单了解 xacro 的基本语法。

需求描述:

使用xacro优化上一节案例中驱动轮实现，需要使用变量封装底盘的半径、高度，使用数学公式动态计算底盘的关节点坐标，使用 Xacro 宏封装轮子重复的代码并调用宏创建两个轮子(注意: 在此，演示 Xacro 的基本使用，不必生成合法的 URDF)。

准备:

创建功能包，导入 urdf 与 xacro。

1.Xacro文件编写

编写 Xacro 文件，以变量的方式封装属性(常量半径、高度、车轮半径...)，以函数的方式封装重复实现(车轮的添加)。

```
1 <robot name="mycar"
  xmlns:xacro="http://wiki.ros.org/xacro">
```

```

2      <!-- 属性封装 -->
3      <xacro:property name="wheel_radius"
4          value="0.0325" />
5      <xacro:property name="wheel_length"
6          value="0.0015" />
7      <xacro:property name="PI" value="3.1415927" />
8      <xacro:property name="base_link_length"
9          value="0.08" />
10     <xacro:property name="lidi_space"
11         value="0.015" />
12
13     <!-- 宏 -->
14     <xacro:macro name="wheel_func"
15         params="wheel_name flag" >
16         <link name="${wheel_name}_wheel">
17             <visual>
18                 <geometry>
19                     <cylinder
20                         radius="${wheel_radius}" length="${wheel_length}"
21                     />
22                     </geometry>
23
24             <origin xyz="0 0 0" rpy="${PI / 2}
25                 0 0" />
26
27             <material name="wheel_color">
28                 <color rgba="0 0 0 0.3" />
29             </material>
30             </visual>
31         </link>
32
33         <!-- 3-2.joint -->
34         <joint name="${wheel_name}2link"
35             type="continuous">
36             <parent link="base_link" />
37             <child link="${wheel_name}_wheel" />
38             <!--
39                 x 无偏移
40                 y 车体半径
41             -->

```

```

32           z z= 车体高度 / 2 + 离地间距 - 车轮半
33           径
34           -->
35           <origin xyz="0 ${0.1 * flag}
36           ${ (base_link_length / 2 + lidi_space -
37           wheel_radius) * -1}" rpy="0 0 0" />
38           <axis xyz="0 1 0" />
39           </joint>
40
41           </xacro:macro>
42           <xacro:wheel_func wheel_name="left" flag="1"
43           />
44           <xacro:wheel_func wheel_name="right" flag="-1"
45           />
46       </robot>

```

2.Xacro文件转换成 urdf 文件

命令行进入 xacro 文件 所属目录，执行: `rosrun xacro xacro` `xxx.xacro > xxx.urdf` , 会将 xacro 文件解析为 urdf 文件，内容如下:

```

1  <?xml version="1.0" ?>
2  <!--
=====
===== -->
3  <!-- | This document was autogenerated by xacro
4  from test.xacro | -->
5  <!-- | EDITING THIS FILE BY HAND IS NOT
6  RECOMMENDED | -->
7  <!--
=====
===== -->
8  <robot name="mycar">
9    <link name="left_wheel">
10   <visual>
11     <geometry>

```

```

10          <cylinder length="0.0015"
11          radius="0.0325"/>
12      </geometry>
13      <origin rpy="1.57079635 0 0" xyz="0 0 0"/>
14      <material name="wheel_color">
15          <color rgba="0 0 0 0.3"/>
16      </material>
17      </visual>
18  </link>
19  <!-- 3-2.joint -->
20  <joint name="left2link" type="continuous">
21      <parent link="base_link"/>
22      <child link="left_wheel"/>
23      <!--
24          x 无偏移
25          y 车体半径
26          z z= 车体高度 / 2 + 离地间距 - 车轮半
27          径
28          -->
29          <origin rpy="0 0 0" xyz="0 0.1 -0.0225"/>
30          <axis xyz="0 1 0"/>
31      </joint>
32      <link name="right_wheel">
33          <visual>
34              <geometry>
35                  <cylinder length="0.0015"
36                  radius="0.0325"/>
37              </geometry>
38              <origin rpy="1.57079635 0 0" xyz="0 0 0"/>
39              <material name="wheel_color">
40                  <color rgba="0 0 0 0.3"/>
41              </material>
42          </visual>
43      </link>
44  <!-- 3-2.joint -->
45  <joint name="right2link" type="continuous">
46      <parent link="base_link"/>
47      <child link="right_wheel"/>

```

```

46      <!--
47          x 无偏移
48          y 车体半径
49          z z= 车体高度 / 2 + 离地间距 - 车轮半
50          径
51          -->
52      <origin rpy="0 0 0" xyz="0 -0.1 -0.0225"/>
53      <axis xyz="0 1 0"/>
54  </joint>
55 </robot>

```

注意: 该案例编写生成的是非法的 URDF 文件, 目的在于演示 Xacro 的极简使用以及优点。

6.4.2 Xacro_语法详解

xacro 提供了可编程接口, 类似于计算机语言, 包括变量声明调用、函数声明与调用等语法实现。在使用 xacro 生成 urdf 时, 根标签 `robot` 中必须包含命名空间声明: `xmlns:xacro="http://wiki.ros.org/xacro"`

1. 属性与算数运算

用于封装 URDF 中的一些字段, 比如: PAI 值, 小车的尺寸, 轮子半径

属性定义

```
1 <xacro:property name="xxxx" value="yyyy" />
```

属性调用

```
1 ${属性名称}
```

算数运算

```
1 ${数学表达式}
```

2.宏

类似于函数实现，提高代码复用率，优化代码结构，提高安全性

宏定义

```
1 <xacro:macro name="宏名称" params="参数列表(多参数之间使用空格分隔)">
2
3     .....
4
5     参数调用格式: ${参数名}
6
7 </xacro:macro>
```

宏调用

```
1 <xacro:宏名称 参数1=xxx 参数2=xxx/>
```

3.文件包含

机器人由多部件组成，不同部件可能封装为单独的 xacro 文件，最后再将不同的文件集成，组合为完整机器人，可以使用文件包含实现

文件包含

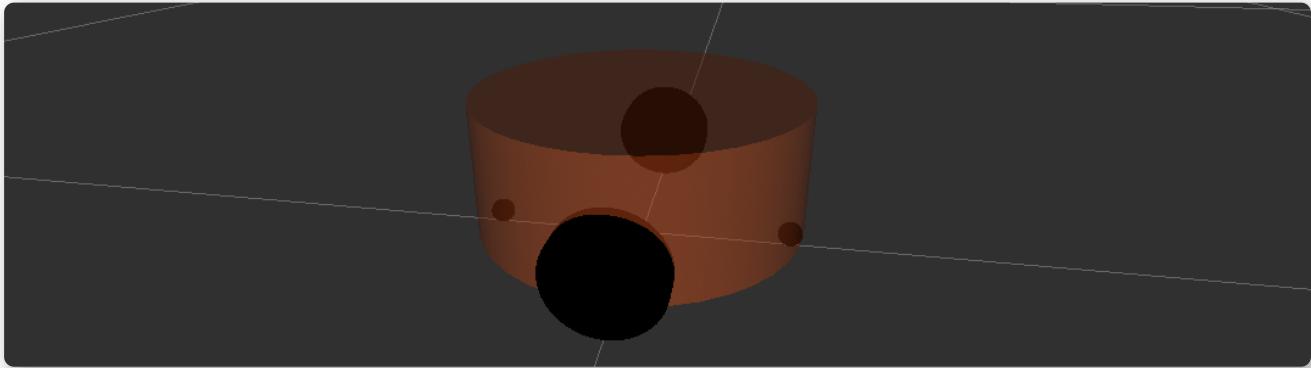
```
1 <robot name="xxx"
2     xmlns:xacro="http://wiki.ros.org/xacro">
3         <xacro:include filename="my_base.xacro" />
4         <xacro:include filename="my_camera.xacro" />
5         <xacro:include filename="my_laser.xacro" />
6     .....
7 </robot>
```

6.4.3 Xacro_完整使用流程示例

需求描述:

使用 Xacro 优化 URDF 版的小车底盘模型实现

结果演示:



1. 编写 Xacro 文件

```
1 <!--
2      使用 xacro 优化 URDF 版的小车底盘实现:
3
4      实现思路:
5          1. 将一些常量、变量封装为 xacro:property
6              比如:PI 值、小车底盘半径、离地间距、车轮半径、宽度
7              .....
8
9          2. 使用 宏 封装驱动轮以及支撑轮实现，调用相关宏生成驱动
10     轮与支撑轮
11
12     -->
13     <!-- 根标签，必须声明 xmlns:xacro -->
14     <robot name="my_base"
15         xmlns:xacro="http://www.ros.org/wiki/xacro">
16         <!-- 封装变量、常量 -->
17         <xacro:property name="PI" value="3.141"/>
18         <!-- 宏:黑色设置 -->
19         <material name="black">
20             <color rgba="0.0 0.0 0.0 1.0" />
21         </material>
22         <!-- 底盘属性 -->
```

```

19      <xacro:property name="base_footprint_radius"
20          value="0.001" /> <!-- base_footprint 半径 -->
21      <xacro:property name="base_link_radius"
22          value="0.1" /> <!-- base_link 半径 -->
23      <xacro:property name="base_link_length"
24          value="0.08" /> <!-- base_link 长 -->
25      <xacro:property name="earth_space"
26          value="0.015" /> <!-- 离地间距 -->
27
28      <!-- 底盘 -->
29      <link name="base_footprint">
30          <visual>
31              <geometry>
32                  <sphere
33                      radius="${base_footprint_radius}" />
34                  </geometry>
35          </visual>
36      </link>
37
38      <link name="base_link">
39          <visual>
40              <geometry>
41                  <cylinder radius="${base_link_radius}"
42                      length="${base_link_length}" />
43                  </geometry>
44                  <origin xyz="0 0 0" rpy="0 0 0" />
45                  <material name="yellow">
46                      <color rgba="0.5 0.3 0.0 0.5" />
47                  </material>
48          </visual>
49      </link>
50
51      <joint name="base_link2base_footprint"
52          type="fixed">
53          <parent link="base_footprint" />
54          <child link="base_link" />
55          <origin xyz="0 0 ${earth_space +
56              base_link_length / 2 }" />
57      </joint>

```

```

50
51      <!-- 驱动轮 -->
52      <!-- 驱动轮属性 -->
53      <xacro:property name="wheel_radius"
54          value="0.0325" /><!-- 半径 -->
55      <xacro:property name="wheel_length"
56          value="0.015" /><!-- 宽度 -->
57      <!-- 驱动轮宏实现 -->
58      <xacro:macro name="add_wheels" params="name
59          flag">
60          <link name="${name}_wheel">
61              <visual>
62                  <geometry>
63                      <cylinder radius="${wheel_radius}"
64                          length="${wheel_length}" />
65                  </geometry>
66                  <origin xyz="0.0 0.0 0.0" rpy="${PI / 2}
67 0.0 0.0" />
68                  <material name="black" />
69              </visual>
70          </link>
71
72          <joint name="${name}_wheel2base_link"
73              type="continuous">
74              <parent link="base_link" />
75              <child link="${name}_wheel" />
76              <origin xyz="0 ${flag * base_link_radius}
77 ${-(earth_space + base_link_length / 2 -
78 wheel_radius) }" />
79              <axis xyz="0 1 0" />
80          </joint>
81      </xacro:macro>
82      <xacro:add_wheels name="left" flag="1" />
83      <xacro:add_wheels name="right" flag="-1" />
84      <!-- 支撑轮 -->
85      <!-- 支撑轮属性 -->
86      <xacro:property name="support_wheel_radius"
87          value="0.0075" /> <!-- 支撑轮半径 -->
88
89

```

```

80      <!-- 支撑轮宏 -->
81      <xacro:macro name="add_support_wheel"
82          params="name flag" >
83          <link name="${name}_wheel">
84              <visual>
85                  <geometry>
86                      <sphere
87                          radius="${support_wheel_radius}" />
88                      </geometry>
89                      <origin xyz="0 0 0" rpy="0 0 0" />
90                      <material name="black" />
91                  </visual>
92          </link>
93
94          <joint name="${name}_wheel2base_link"
95              type="continuous">
96              <parent link="base_link" />
97              <child link="${name}_wheel" />
98              <origin xyz="${flag * (base_link_radius
99                  - support_wheel_radius)} 0 ${-(base_link_length /
100                     2 + earth_space / 2)}" />
101                  <axis xyz="1 1 1" />
102          </joint>
103      </xacro:macro>
104
105      <xacro:add_support_wheel name="front" flag="1"
106          />
107      <xacro:add_support_wheel name="back" flag="-1"
108          />
109
110  </robot>

```

2.集成launch文件

方式1:先将 xacro 文件转换出 urdf 文件, 然后集成

先将 xacro 文件解析成 urdf 文件: `rosrun xacro xacro xxx.xacro > xxx.urdf` 然后再按照之前的集成方式直接整合 launch 文件, 内容示例:

```
1 <launch>
2   <param name="robot_description" textfile="$(find
3     demo01_urdf_helloworld)/urdf/xacro/my_base.urdf" />
4   <node pkg="rviz" type="rviz" name="rviz" args="-
5     d $(find
6     demo01_urdf_helloworld)/config/helloworld.rviz" />
7   <node pkg="joint_state_publisher"
8     type="joint_state_publisher"
9     name="joint_state_publisher" output="screen" />
10  <node pkg="robot_state_publisher"
11    type="robot_state_publisher"
12    name="robot_state_publisher" output="screen" />
13  <node pkg="joint_state_publisher_gui"
14    type="joint_state_publisher_gui"
15    name="joint_state_publisher_gui" output="screen" />
16
17 </launch>
```

方式2:在 launch 文件中直接加载 xacro(建议使用)

launch 内容示例:

```

1 <launch>
2   <param name="robot_description" command="$(find
3     xacro)/xacro $(find
4     demo01_urdf_helloworld)/urdf/xacro/my_base.urdf.xacr
5     o" />
6
7   <node pkg="rviz" type="rviz" name="rviz" args="-
8     d $(find
9     demo01_urdf_helloworld)/config/helloworld.rviz" />
10  <node pkg="joint_state_publisher"
11    type="joint_state_publisher"
12    name="joint_state_publisher" output="screen" />
13  <node pkg="robot_state_publisher"
14    type="robot_state_publisher"
15    name="robot_state_publisher" output="screen" />
16  <node pkg="joint_state_publisher_gui"
17    type="joint_state_publisher_gui"
18    name="joint_state_publisher_gui" output="screen" />
19
20 </launch>

```

核心代码:

```

1 <param name="robot_description" command="$(find
2   xacro)/xacro $(find
3     demo01_urdf_helloworld)/urdf/xacro/my_base.urdf.xacr
4     o" />

```

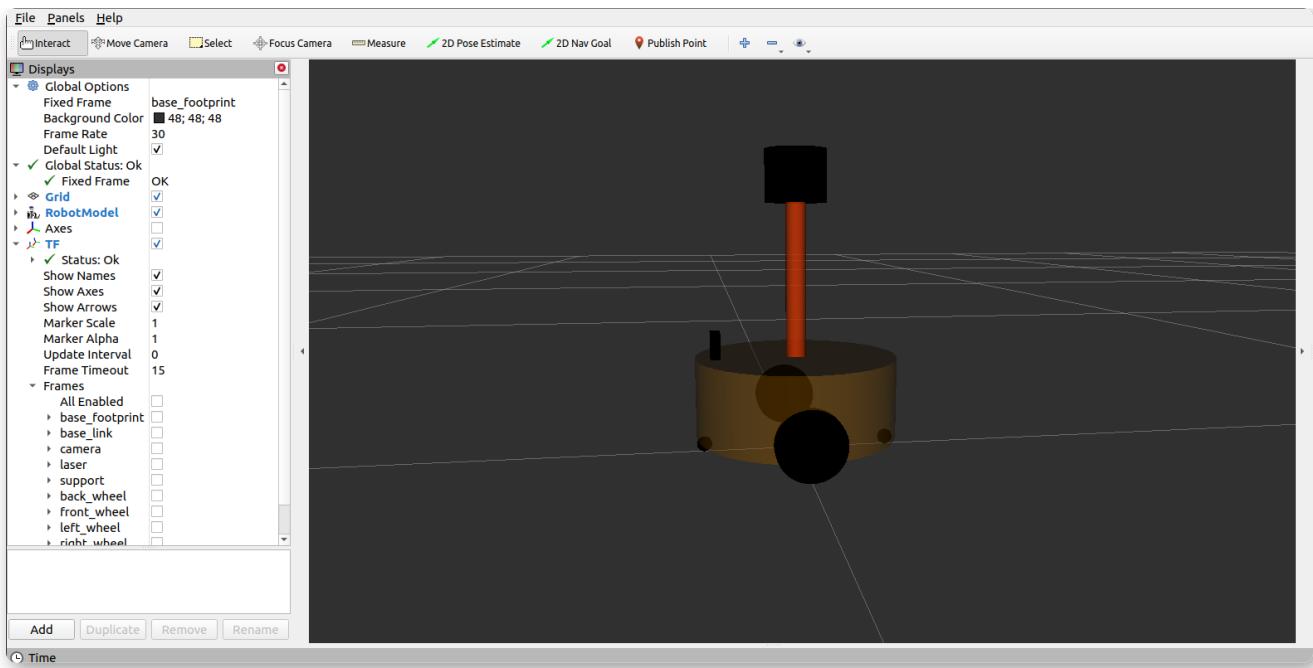
加载 `robot_description` 时使用 `command` 属性，属性值就是调用 `xacro` 功能包的 `xacro` 程序直接解析 `xacro` 文件。

6.4.4 Xacro_实操

需求描述:

在前面小车底盘基础之上，添加摄像头和雷达传感器。

结果演示:



实现分析:

机器人模型由多部件组成，可以将不同组件设置进单独文件，最终通过文件包含实现组件的拼装。

实现流程:

1. 首先编写摄像头和雷达的 xacro 文件
2. 然后再编写一个组合文件，组合底盘、摄像头与雷达
3. 最后，通过 launch 文件启动 Rviz 并显示模型

1. 摄像头和雷达 Xacro 文件实现

摄像头 xacro 文件:

```

1 <!-- 摄像头相关的 xacro 文件 -->
2 <robot name="my_camera"
  xmlns:xacro="http://wiki.ros.org/xacro">
3   <!-- 摄像头属性 -->
4     <xacro:property name="camera_length"
  value="0.01" /> <!-- 摄像头长度(x) -->
5     <xacro:property name="camera_width"
  value="0.025" /> <!-- 摄像头宽度(y) -->
6     <xacro:property name="camera_height"
  value="0.025" /> <!-- 摄像头高度(z) -->

```

```

7      <xacro:property name="camera_x" value="0.08"
8      /> <!-- 摄像头安装的x坐标 -->
9      <xacro:property name="camera_y" value="0.0" />
10     <!-- 摄像头安装的y坐标 -->
11     <xacro:property name="camera_z"
12     value="${base_link_length / 2 + camera_height /
13     2}" /> <!-- 摄像头安装的z坐标:底盘高度 / 2 + 摄像头高度
14     / 2 -->
15
16     <!-- 摄像头关节以及link -->
17     <link name="camera">
18         <visual>
19             <geometry>
20                 <box size="${camera_length}
21                 ${camera_width} ${camera_height}" />
22             </geometry>
23             <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0
24             0.0" />
25                 <material name="black" />
26             </visual>
27         </link>
28
29         <joint name="camera2base_link" type="fixed">
30             <parent link="base_link" />
31             <child link="camera" />
32             <origin xyz="${camera_x} ${camera_y}
33             ${camera_z}" />
34         </joint>
35     </robot>

```

雷达 xacro 文件:

```

1  <!--
2      小车底盘添加雷达
3  -->
4  <robot name="my_laser"
5      xmlns:xacro="http://wiki.ros.org/xacro">

```

```

6      <!-- 雷达支架 -->
7      <xacro:property name="support_length"
value="0.15" /> <!-- 支架长度 -->
8      <xacro:property name="support_radius"
value="0.01" /> <!-- 支架半径 -->
9      <xacro:property name="support_x" value="0.0"
/> <!-- 支架安装的x坐标 -->
10     <xacro:property name="support_y" value="0.0"
/> <!-- 支架安装的y坐标 -->
11     <xacro:property name="support_z"
value="${base_link_length / 2 + support_length /"
2}" /> <!-- 支架安装的z坐标:底盘高度 / 2 + 支架高度 / 2
-->
12
13     <link name="support">
14         <visual>
15             <geometry>
16                 <cylinder
radius="${support_radius}"
length="${support_length}" />
17                 </geometry>
18                 <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0
0.0" />
19                 <material name="red">
20                     <color rgba="0.8 0.2 0.0 0.8" />
21                 </material>
22             </visual>
23         </link>
24
25     <joint name="support2base_link" type="fixed">
26         <parent link="base_link" />
27         <child link="support" />
28         <origin xyz="${support_x} ${support_y}"
${support_z}" />
29     </joint>
30
31
32     <!-- 雷达属性 -->

```

```

33      <xacro:property name="laser_length"
34          value="0.05" /> <!-- 雷达长度 -->
35      <xacro:property name="laser_radius"
36          value="0.03" /> <!-- 雷达半径 -->
37      <xacro:property name="laser_x" value="0.0" />
38          <!-- 雷达安装的x坐标 -->
39      <xacro:property name="laser_y" value="0.0" />
40          <!-- 雷达安装的y坐标 -->
41      <xacro:property name="laser_z"
42          value="${support_length / 2 + laser_length / 2}" /> <!-- 雷达安装的z坐标:支架高度 / 2 + 雷达高度 / 2 -->
43
44
45
46
47
48
49
50      <joint name="laser2support" type="fixed">
51          <parent link="support" />
52          <child link="laser" />
53          <origin xyz="${laser_x} ${laser_y}"
54              ${laser_z}" />
55      </joint>
56  </robot>

```

2.组合底盘摄像头与雷达的 xacro 文件

```

1 <!-- 组合小车底盘与摄像头与雷达 -->
2 <robot name="my_car_camera"
3   xmlns:xacro="http://wiki.ros.org/xacro">
4     <xacro:include filename="my_base.urdf.xacro" />
5     <xacro:include filename="my_camera.urdf.xacro" />
6     <xacro:include filename="my_laser.urdf.xacro" />
7 </robot>

```

3.launch 文件

```

1 <launch>
2   <param name="robot_description" command="$(find
3     xacro)/xacro $(find
4       demo01_urdf_helloworld)/urdf/xacro/my_base_camera_la
5       ser.urdf.xacro" />
6
7   <node pkg="rviz" type="rviz" name="rviz" args="-
8     d $(find
9       demo01_urdf_helloworld)/config/helloworld.rviz" />
10  <node pkg="joint_state_publisher"
11    type="joint_state_publisher"
12    name="joint_state_publisher" output="screen" />
13  <node pkg="robot_state_publisher"
14    type="robot_state_publisher"
15    name="robot_state_publisher" output="screen" />
16  <node pkg="joint_state_publisher_gui"
17    type="joint_state_publisher_gui"
18    name="joint_state_publisher_gui" output="screen" />
19 </launch>

```

6.5 Rviz中控制机器人模型运动

通过 URDF 结合 rviz 可以创建并显示机器人模型，不过，当前实现的只是静态模型，如何控制模型的运动呢？在此，可以调用 Arbotix 实现此功能。

简介

Arbotix:Arbotix 是一款控制电机、舵机的控制板，并提供相应的 ros 功能包，这个功能包的功能不仅可以驱动真实的 Arbotix 控制板，它还提供一个差速控制器，通过接受速度控制指令更新机器人的 joint 状态，从而帮助我们实现机器人在 rviz 中的运动。

这个差速控制器在 arbotix_python 程序包中，完整的 arbotix 程序包还包括多种控制器，分别对应 dynamixel 电机、多关节机械臂以及不同形状的夹持器。

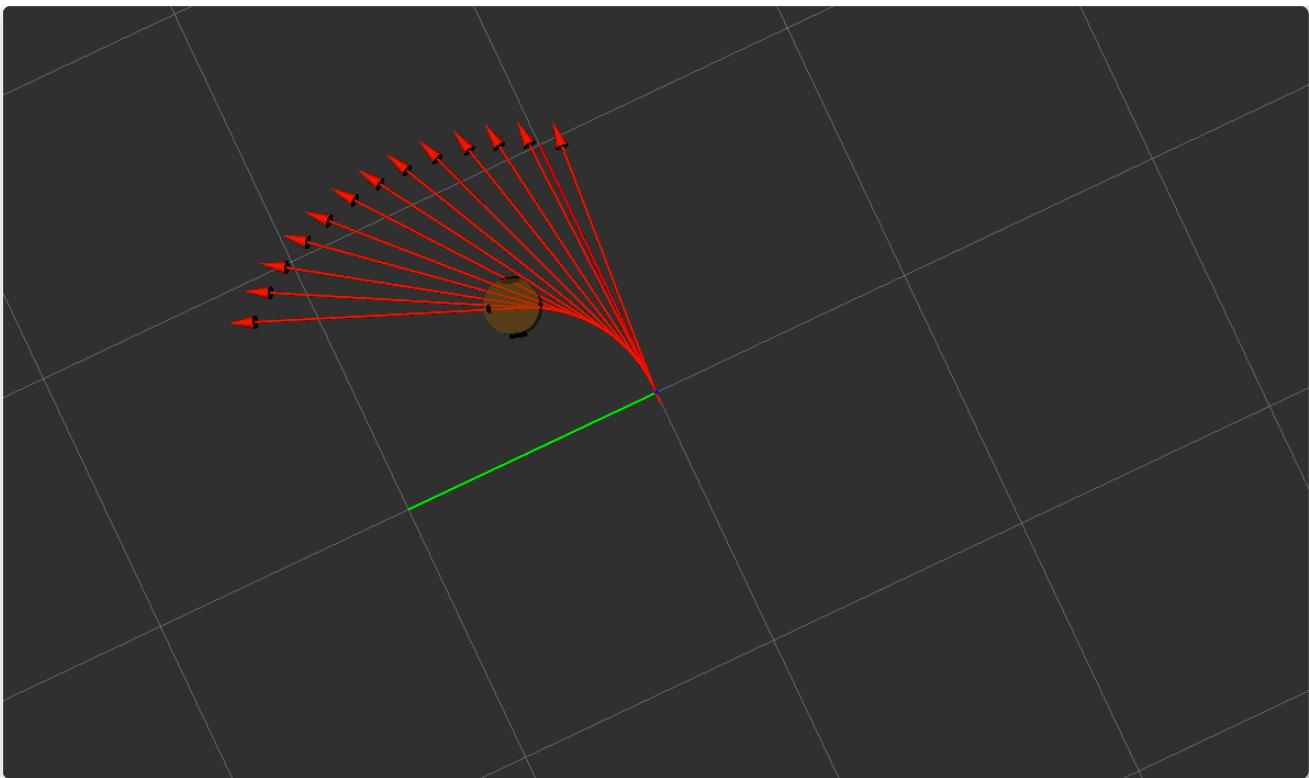
6.5.1 Arbotix使用流程

接下来，通过一个案例演示 arbotix 的使用。

需求描述:

控制机器人模型在 rviz 中做圆周运动

结果演示:



实现流程:

1. 安装 Arbotix
2. 创建新功能包，准备机器人 urdf、xacro 文件
3. 添加 Arbotix 配置文件
4. 编写 launch 文件配置 Arbotix
5. 启动 launch 文件并控制机器人模型运动

1. 安装 Arbotix

方式1: 命令行调用

```
1 sudo apt-get install ros-<<VersionName()>>-arbotix
```

将 <<VersionName()>> 替换成当前 ROS 版本名称，如果提示功能包无法定位，请采用方式2。

方式2: 源码安装

先从 github 下载源码，然后调用 catkin_make 编译

```
1 git clone
https://github.com/vanadiumlabs/arbotix_ros.git
```

2. 创建新功能包，准备机器人 urdf、xacro

urdf 和 xacro 调用上一讲实现即可

3. 添加 arbotix 所需的配置文件

添加 arbotix 所需配置文件

```

1 # 该文件是控制器配置,一个机器人模型可能有多个控制器, 比如:
2   底盘、机械臂、夹持器(机械手)....
3 # 因此, 根 name 是 controller
4 controllers: {
5   # 单控制器设置
6   base_controller: {
7     #类型: 差速控制器
8     type: diff_controller,
9     #参考坐标
10    base_frame_id: base_footprint,
11    #两个轮子之间的间距
12    base_width: 0.2,
13    #控制频率
14    ticks_meter: 2000,
15    #PID控制参数, 使机器人车轮快速达到预期速度
16    Kp: 12,
17    Kd: 12,
18    Ki: 0,
19    Ko: 50,
20    #加速限制
21    accel_limit: 1.0
22  }

```

另请参考: http://wiki.ros.org/arbotix_python/diff_controller

4.launch 文件中配置 arbotix 节点

launch 示例代码

```
1 <node name="arbotix" pkg="arbotix_python"
2   type="arbotix_driver" output="screen">
3   <rosparam file="$(find
4     my_urdf05_rviz)/config/hello.yaml" command="load" />
5     <param name="sim" value="true" />
6   </node>
```

代码解释:

调用了 arbotix_python 功能包下的 arbotix_driver 节点

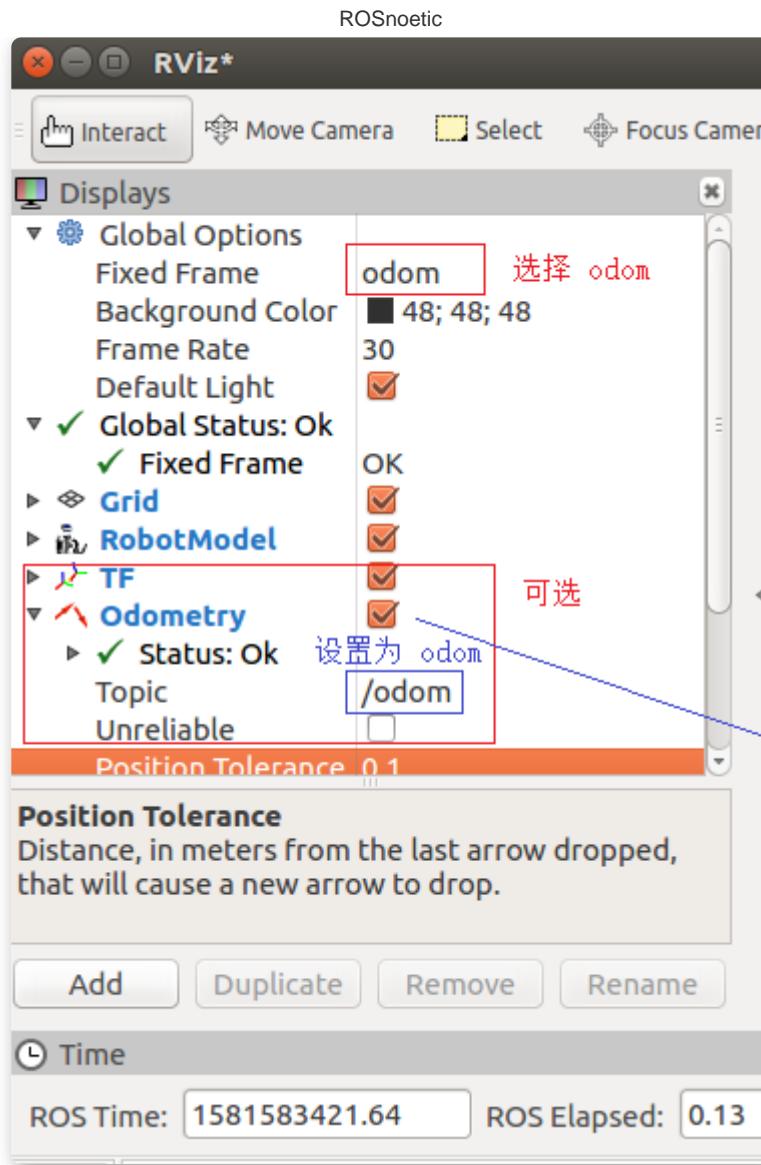
arbotix 驱动机器人运行时，需要获取机器人信息，可以通过 file 加载配置文件

在仿真环境下，需要配置 sim 为 true

5.启动 launch 文件并控制机器人模型运动

启动launch:roslaunch xxxxlaunch

配置 rviz:



控制小车运动:

此时调用 `rostopic list` 会发现一个熟悉的话题: `/cmd_vel`

```
ubuntu@Turtlebot3-virtual-machine:~$ rostopic list
/clicked_point
/cmd_vel
/diagnostics
/initialpose
/joint_states
/move_base_simple/goal
/odom
/rosout
/rosout_agg
/tf
/tf_static
```

也就说我们可以发布 `cmd_vel` 话题消息控制小车运动了，该实现策略有多种，可以另行编写节点，或者更简单些可以直接通过如下命令发布消息:

```
1 rostopic pub -r 10 /cmd_vel geometry_msgs/Twist
  '{linear: {x: 0.2, y: 0, z: 0}, angular: {x: 0, y: 0, z: 0.5}}'
```

现在，小车就可以运动起来了。

另请参考:

- <http://wiki.ros.org/arbotix>

6.6 URDF集成Gazebo

URDF 需要集成进 Rviz 或 Gazebo 才能显示可视化的机器人模型，前面已经介绍了URDF 与 Rviz 的集成，本节主要介绍:

- URDF 与 Gazebo 的基本集成流程；
- 如果要在 Gazebo 中显示机器人模型，URDF 需要做的一些额外配置；
- 关于Gazebo仿真环境的搭建。

6.6.1 URDF与Gazebo基本集成流程

URDF 与 Gazebo 集成流程与 Rviz 实现类似，主要步骤如下:

1. 创建功能包，导入依赖项
2. 编写 URDF 或 Xacro 文件
3. 启动 Gazebo 并显示机器人模型

1.创建功能包

创建新功能包，导入依赖包: urdf、xacro、gazebo_ros、gazebo_ros_control、gazebo_plugins

2.编写URDF文件

```

1  <!--
2      创建一个机器人模型(盒状即可)，显示在 Gazebo 中
3  -->
4
5  <robot name="mycar">
6      <link name="base_link">
7          <visual>
8              <geometry>

```

```

9          <box size="0.5 0.2 0.1" />
10         </geometry>
11         <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0
12           0.0" />
13         <material name="yellow">
14           <color rgba="0.5 0.3 0.0 1" />
15         </material>
16         </visual>
17         <collision>
18           <geometry>
19             <box size="0.5 0.2 0.1" />
20           </geometry>
21           <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0
22             0.0" />
23           </collision>
24         <inertial>
25           <origin xyz="0 0 0" />
26           <mass value="6" />
27           <inertia ixx="1" ixy="0" ixz="0"
28             iyy="1" iyz="0" izz="1" />
29           </inertial>
30         </link>
31         <gazebo reference="base_link">
32           <material>Gazebo/Black</material>
33         </gazebo>
34       </robot>

```

注意，当 URDF 需要与 Gazebo 集成时，和 Rviz 有明显区别：

1. 必须使用 `collision` 标签，因为既然是仿真环境，那么必然涉及到碰撞检测，`collision` 提供碰撞检测的依据。
2. 必须使用 `inertial` 标签，此标签标注了当前机器人某个刚体部分的惯性矩阵，用于一些力学相关的仿真计算。
3. 颜色设置，也需要重新使用 `gazebo` 标签标注，因为之前的颜色设置为了方便调试包含透明度，仿真环境下没有此选项。

3. 启动Gazebo并显示模型

launch 文件实现:

```
1 <launch>
2
3     <!-- 将 Urdf 文件的内容加载到参数服务器 -->
4     <param name="robot_description"
5         textfile="$(find
6             demo02_urdf_gazebo)/urdf/urdf01_helloworld.urdf"
7         />
8
9     <!-- 启动 gazebo -->
10    <include file="$(find
11        gazebo_ros)/launch/empty_world.launch" />
12
13     <!-- 在 gazebo 中显示机器人模型 -->
14     <node pkg="gazebo_ros" type="spawn_model"
15         name="model" args="-urdf -model mycar -param
16         robot_description" />
17 </launch>
```

代码解释:

```

1 <include file="$(find
  gazebo_ros)/launch/empty_world.launch" />
2 <!-- 启动 Gazebo 的仿真环境, 当前环境为空环境 -->
3 <node pkg="gazebo_ros" type="spawn_model"
  name="model" args="-urdf -model mycar -param
  robot_description" />
4
5 <!--
6   在 Gazebo 中加载一个机器人模型, 该功能由 gazebo_ros
7   下的 spawn_model 提供:
8     -urdf 加载的是 urdf 文件
9     -model mycar 模型名称是 mycar
10    -param robot_description 从参数
11      robot_description 中载入模型
12      -x 模型载入的 x 坐标
13      -y 模型载入的 y 坐标
14      -z 模型载入的 z 坐标
15  -->

```

6.6.2 URDF集成Gazebo相关设置

较之于 rviz, gazebo 在集成 URDF 时, 需要做些许修改, 比如: 必须添加 collision 碰撞属性相关参数、必须添加 inertial 惯性矩阵相关参数, 另外, 如果直接移植 Rviz 中机器人的颜色设置是没有显示的, 颜色设置也必须做相应的变更。

1. collision

如果机器人 link 是标准的几何体形状, 和 link 的 visual 属性设置一致即可。

2. inertial

惯性矩阵的设置需要结合 link 的质量与外形参数动态生成, 标准的球体、圆柱与立方体的惯性矩阵公式如下(已经封装为 xacro 实现):

球体惯性矩阵

```

1 <!-- Macro for inertia matrix -->
2   <xacro:macro name="sphere_inertial_matrix"
3     params="m r">
4     <inertial>
5       <mass value="${m}" />
6       <inertia ixx="${2*m*r*r/5}" ixy="0"
7         ixz="0"
8           iyy="${2*m*r*r/5}" iyz="0"
9             izz="${2*m*r*r/5}" />
10           </inertial>
11         </xacro:macro>

```

圆柱惯性矩阵

```

1 <xacro:macro name="cylinder_inertial_matrix"
2   params="m r h">
3   <inertial>
4     <mass value="${m}" />
5     <inertia ixx="${m*(3*r*r+h*h)/12}" ixy =
6       "0" ixz = "0"
7         iyy="${m*(3*r*r+h*h)/12}" iyz = "0"
8           izz="${m*r*r/2}" />
9         </inertial>
10       </xacro:macro>

```

立方体惯性矩阵

```

1 <xacro:macro name="Box_inertial_matrix" params="m l
2   w h">
3     <inertial>
4       <mass value="${m}" />
5       <inertia ixx="${m*(h*h + l*l)/12}"
6         ixy = "0" ixz = "0"
7         iyy="${m*(w*w + l*l)/12}" iyz=
8           "0"
9           izz="${m*(w*w + h*h)/12}" />
10      </inertial>
11    </xacro:macro>

```

需要注意的是，原则上，除了 base_footprint 外，机器人的每个刚体部分都需要设置惯性矩阵，且惯性矩阵必须经计算得出，如果随意定义刚体部分的惯性矩阵，那么可能会导致机器人在 Gazebo 中出现抖动，移动等现象。

3.颜色设置

在 gazebo 中显示 link 的颜色，必须要使用指定的标签：

```

1 <gazebo reference="link节点名称">
2   <material>Gazebo/Blue</material>
3 </gazebo>

```

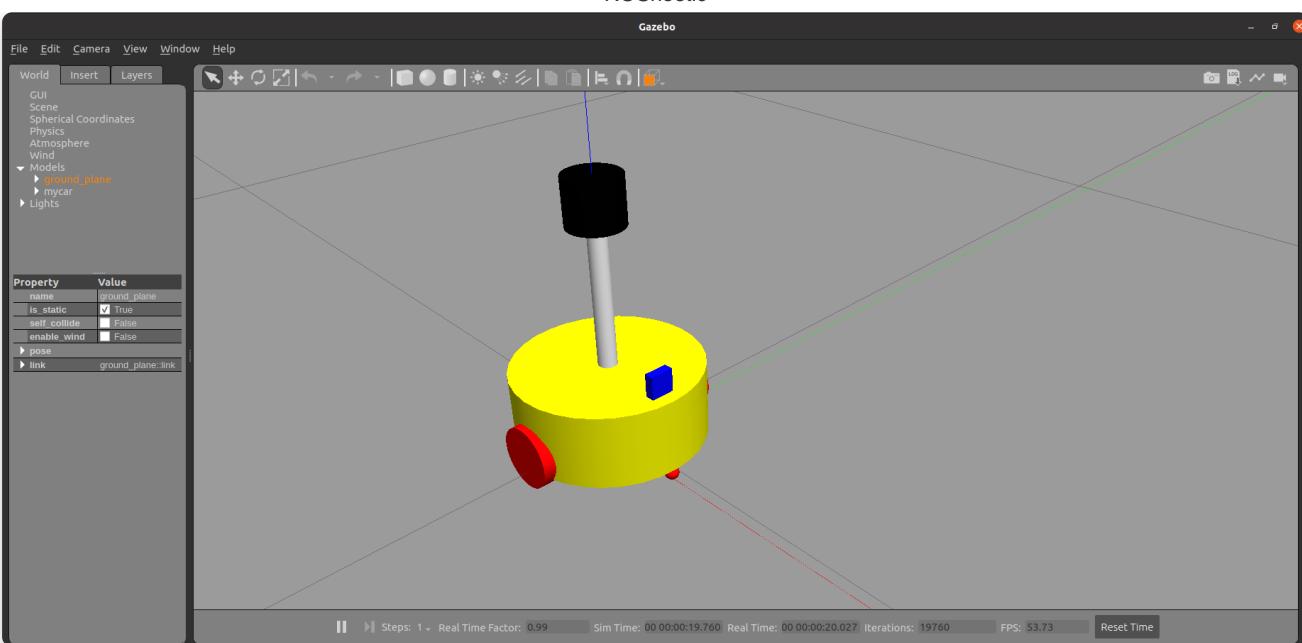
PS： material 标签中，设置的值区分大小写，颜色可以设置为 Red Blue Green Black

6.6.3 URDF集成Gazebo实操

需求描述：

将之前的机器人模型(xacro版)显示在 gazebo 中

结果演示：



实现流程：

1. 需要编写封装惯性矩阵算法的 xacro 文件
2. 为机器人模型中的每一个 link 添加 collision 和 inertial 标签，并且重置颜色属性
3. 在 launch 文件中启动 gazebo 并添加机器人模型

1. 编写封装惯性矩阵算法的 xacro 文件

```

1 <robot name="base"
2   xmlns:xacro="http://wiki.ros.org/xacro">
3     <!-- Macro for inertia matrix -->
4     <xacro:macro name="sphere_inertial_matrix"
5       params="m r">
6       <inertial>
7         <mass value="${m}" />
8         <inertia ixx="${2*m*r*r/5}" ixy="0"
9           ixz="0"
10          iyy="${2*m*r*r/5}" iyz="0"
11          izz="${2*m*r*r/5}" />
12       </inertial>
13     </xacro:macro>
14
15     <xacro:macro name="cylinder_inertial_matrix"
16       params="m r h">
17       <inertial>
18         <mass value="${m}" />

```

```

15          <inertia ixx="${m*(3*r*r+h*h)/12}" ixy
16          = "0" ixz = "0"
17          iyy="${m*(3*r*r+h*h)/12}" iyz =
18          "0"
19          izz="${m*r*r/2}" />
20      </inertial>
21  </xacro:macro>
22
23  <xacro:macro name="Box_inertial_matrix"
24  params="m l w h">
25      <inertial>
26          <mass value="${m}" />
27          <inertia ixx="${m*(h*h + l*l)/12}"
28          ixy = "0" ixz = "0"
29          iyy="${m*(w*w + l*l)/12}" iyz=
30          "0"
31          izz="${m*(w*w + h*h)/12}" />
32      </inertial>
33  </xacro:macro>
34 </robot>

```

2. 复制相关 xacro 文件，并设置 collision inertial 以及 color 等参数

A. 底盘 Xacro 文件

```

1  <!--
2      使用 xacro 优化 URDF 版的小车底盘实现:
3
4      实现思路:
5      1. 将一些常量、变量封装为 xacro:property
6          比如:PI 值、小车底盘半径、离地间距、车轮半径、宽度
7          ....
8      2. 使用 宏 封装驱动轮以及支撑轮实现，调用相关宏生成驱动
9          轮与支撑轮
10     <!-- 根标签，必须声明 xmlns:xacro -->

```

```

11 <robot name="my_base"
  xmlns:xacro="http://www.ros.org/wiki/xacro">
12     <!-- 封装变量、常量 -->
13     <!-- PI 值设置精度需要高一些, 否则后续车轮翻转量计算
        时, 可能会出现肉眼不能察觉的车轮倾斜, 从而导致模型抖动 -->
14     <xacro:property name="PI" value="3.1415926"/>
15     <!-- 宏:黑色设置 -->
16     <material name="black">
17         <color rgba="0.0 0.0 0.0 1.0" />
18     </material>
19     <!-- 底盘属性 -->
20     <xacro:property name="base_footprint_radius"
  value="0.001" /> <!-- base_footprint 半径 -->
21     <xacro:property name="base_link_radius"
  value="0.1" /> <!-- base_link 半径 -->
22     <xacro:property name="base_link_length"
  value="0.08" /> <!-- base_link 长 -->
23     <xacro:property name="earth_space"
  value="0.015" /> <!-- 离地间距 -->
24     <xacro:property name="base_link_m" value="0.5"
  /> <!-- 质量 -->
25
26     <!-- 底盘 -->
27     <link name="base_footprint">
28         <visual>
29             <geometry>
30                 <sphere
  radius="${base_footprint_radius}" />
31             </geometry>
32         </visual>
33     </link>
34
35     <link name="base_link">
36         <visual>
37             <geometry>
38                 <cylinder radius="${base_link_radius}"
  length="${base_link_length}" />
39             </geometry>
40             <origin xyz="0 0 0" rpy="0 0 0" />

```

```

41      <material name="yellow">
42          <color rgba="0.5 0.3 0.0 0.5" />
43      </material>
44      </visual>
45      <collision>
46          <geometry>
47              <cylinder radius="${base_link_radius}"
48                  length="${base_link_length}" />
49          </geometry>
50          <origin xyz="0 0 0" rpy="0 0 0" />
51      </collision>
52      <xacro:cylinder_inertial_matrix
53          m="${base_link_m}" r="${base_link_radius}"
54          h="${base_link_length}" />
55
56      </link>
57
58
59
60      <joint name="base_link2base_footprint"
61          type="fixed">
62          <parent link="base_footprint" />
63          <child link="base_link" />
64          <origin xyz="0 0 ${earth_space +
65              base_link_length / 2 }" />
66      </joint>
67      <gazebo reference="base_link">
68          <material>Gazebo/Yellow</material>
69      </gazebo>
70
71      <!-- 驱动轮 -->
72      <!-- 驱动轮属性 -->
73      <xacro:property name="wheel_radius"
74          value="0.0325" /><!-- 半径 -->
75      <xacro:property name="wheel_length"
76          value="0.015" /><!-- 宽度 -->
77      <xacro:property name="wheel_m" value="0.05" />
78      <!-- 质量 -->
79
80      <!-- 驱动轮宏实现 -->

```

```

72      <xacro:macro name="add_wheels" params="name
73        flag">
74          <link name="${name}_wheel">
75            <visual>
76              <geometry>
77                <cylinder radius="${wheel_radius}"
78                  length="${wheel_length}" />
79                </geometry>
80                <origin xyz="0.0 0.0 0.0" rpy="${PI / 2}
81                  0.0 0.0" />
82                <material name="black" />
83              </visual>
84              <collision>
85                <geometry>
86                  <cylinder radius="${wheel_radius}"
87                    length="${wheel_length}" />
88                  </geometry>
89                  <origin xyz="0.0 0.0 0.0" rpy="${PI / 2}
90                    0.0 0.0" />
91                </collision>
92                <xacro:cylinder_inertial_matrix
93                  m="${wheel_m}" r="${wheel_radius}"
94                  h="${wheel_length}" />
95
96            </link>
97
98            <joint name="${name}_wheel2base_link"
99              type="continuous">
100              <parent link="base_link" />
101              <child link="${name}_wheel" />
102              <origin xyz="0 ${flag * base_link_radius}
103                ${-(earth_space + base_link_length / 2 -
104                  wheel_radius)}" />
105              <axis xyz="0 1 0" />
106            </joint>
107
108            <gazebo reference="${name}_wheel">
109              <material>Gazebo/Red</material>
110            </gazebo>

```

```

101
102      </xacro:macro>
103      <xacro:add_wheels name="left" flag="1" />
104      <xacro:add_wheels name="right" flag="-1" />
105      <!-- 支撑轮 -->
106      <!-- 支撑轮属性 -->
107      <xacro:property name="support_wheel_radius"
108          value="0.0075" /> <!-- 支撑轮半径 -->
109
110      <!-- 支撑轮宏 -->
111      <xacro:macro name="add_support_wheel"
112          params="name flag" >
113          <link name="${name}_wheel">
114              <visual>
115                  <geometry>
116                      <sphere
117                          radius="${support_wheel_radius}" />
118                      </geometry>
119                      <origin xyz="0 0 0" rpy="0 0 0" />
120                      <material name="black" />
121                  </visual>
122                  <collision>
123                      <geometry>
124                          <sphere
125                              radius="${support_wheel_radius}" />
126                          </geometry>
127                          <origin xyz="0 0 0" rpy="0 0 0" />
128                      </collision>
129                      <xacro:sphere_inertial_matrix
130                          m="${support_wheel_m}" r="${support_wheel_radius}" />
131                  </link>
132
133                  <joint name="${name}_wheel2base_link"
134                      type="continuous">
135                      <parent link="base_link" />
136                      <child link="${name}_wheel" />

```

```

132          <origin xyz="${flag * (base_link_radius
133              - support_wheel_radius)} 0 ${-(base_link_length /
134                  2 + earth_space / 2)}" />
135          <axis xyz="1 1 1" />
136      </joint>
137      <gazebo reference="${name}_wheel">
138          <material>Gazebo/Red</material>
139      </gazebo>
140  </xacro:macro>
141
140  <xacro:add_support_wheel name="front" flag="1"
141      />
141  <xacro:add_support_wheel name="back" flag="-1"
142      />
143
144 </robot>

```

注意: 如果机器人模型在 Gazebo 中产生了抖动, 滑动, 缓慢位移 ... 诸如此类情况, 请查看

1. 惯性矩阵是否设置了, 且设置是否正确合理
2. 车轮翻转需要依赖于 PI 值, 如果 PI 值精度偏低, 也可能导致上述情况产生

B.摄像头 Xacro 文件

```

1 <!-- 摄像头相关的 xacro 文件 -->
2 <robot name="my_camera"
3   xmlns:xacro="http://wiki.ros.org/xacro">
4     <!-- 摄像头属性 -->
5     <xacro:property name="camera_length"
6       value="0.01" /> <!-- 摄像头长度(x) -->
7     <xacro:property name="camera_width"
8       value="0.025" /> <!-- 摄像头宽度(y) -->
9     <xacro:property name="camera_height"
10       value="0.025" /> <!-- 摄像头高度(z) -->
11     <xacro:property name="camera_x" value="0.08"
12       /> <!-- 摄像头安装的x坐标 -->

```

```

8      <xacro:property name="camera_y" value="0.0" />
9      <!-- 摄像头安装的y坐标 -->
10     <xacro:property name="camera_z"
11       value="${base_link_length / 2 + camera_height /
12       2}" /> <!-- 摄像头安装的z坐标:底盘高度 / 2 + 摄像头高度
13       / 2 -->
14
15     <xacro:property name="camera_m" value="0.01"
16   /> <!-- 摄像头质量 -->
17
18     <!-- 摄像头关节以及link -->
19     <link name="camera">
20       <visual>
21         <geometry>
22           <box size="${camera_length}
23             ${camera_width} ${camera_height}" />
24           </geometry>
25           <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0
26             0.0" />
27           <material name="black" />
28         </visual>
29         <collision>
30           <geometry>
31             <box size="${camera_length}
32               ${camera_width} ${camera_height}" />
33             </geometry>
34             <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0
35               0.0" />
36           </collision>
37           <xacro:Box_inertial_matrix m="${camera_m}"
38             l="${camera_length}" w="${camera_width}"
39             h="${camera_height}" />
40         </link>
41
42         <joint name="camera2base_link" type="fixed">
43           <parent link="base_link" />
44           <child link="camera" />
45           <origin xyz="${camera_x} ${camera_y}
46             ${camera_z}" />

```

```

35      </joint>
36      <gazebo reference="camera">
37          <material>Gazebo/Blue</material>
38      </gazebo>
39  </robot>

```

C.雷达 Xacro 文件

```

1  <!--
2      小车底盘添加雷达
3  -->
4  <robot name="my_laser"
5      xmlns:xacro="http://wiki.ros.org/xacro">
6
7      <!-- 雷达支架 -->
8      <xacro:property name="support_length"
9          value="0.15" /> <!-- 支架长度 -->
10     <xacro:property name="support_radius"
11         value="0.01" /> <!-- 支架半径 -->
12     <xacro:property name="support_x" value="0.0"
13         /> <!-- 支架安装的x坐标 -->
14     <xacro:property name="support_y" value="0.0"
15         /> <!-- 支架安装的y坐标 -->
16     <xacro:property name="support_z"
17         value="${base_link_length / 2 + support_length /
18             2}" /> <!-- 支架安装的z坐标:底盘高度 / 2 + 支架高度 / 2
19         -->
20
21     <xacro:property name="support_m" value="0.02"
22         /> <!-- 支架质量 -->
23
24     <link name="support">
25         <visual>
26             <geometry>
27                 <cylinder
28                     radius="${support_radius}"
29                     length="${support_length}" />
30             </geometry>
31     </link>
32
33     <gazebo reference="camera">
34         <material>Gazebo/Blue</material>
35     </gazebo>
36  </robot>

```

```

20          <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0
21          0.0" />
22          <material name="red">
23              <color rgba="0.8 0.2 0.0 0.8" />
24          </material>
25          </visual>
26
27          <collision>
28              <geometry>
29                  <cylinder
30                      radius="${support_radius}"
31                      length="${support_length}" />
32                  </geometry>
33                  <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0
34                      0.0" />
35              </collision>
36
37          <xacro:cylinder_inertial_matrix
38              m="${support_m}" r="${support_radius}"
39              h="${support_length}" />
40
41      </link>
42
43      <joint name="support2base_link" type="fixed">
44          <parent link="base_link" />
45          <child link="support" />
46          <origin xyz="${support_x} ${support_y}
47              ${support_z}" />
48      </joint>
49
50      <gazebo reference="support">
51          <material>Gazebo/White</material>
52      </gazebo>
53
54      <!-- 雷达属性 -->
55      <xacro:property name="laser_length"
56          value="0.05" /> <!-- 雷达长度 -->
57      <xacro:property name="laser_radius"
58          value="0.03" /> <!-- 雷达半径 -->

```

```

50      <xacro:property name="laser_x" value="0.0" />
51      <!-- 雷达安装的x坐标 -->
52      <xacro:property name="laser_y" value="0.0" />
53      <!-- 雷达安装的y坐标 -->
54      <xacro:property name="laser_z"
55      value="${support_length / 2 + laser_length / 2}" /> <!-- 雷达安装的z坐标:支架高度 / 2 + 雷达高度 / 2 -->
56      <!-- 雷达关节以及link -->
57      <link name="laser">
58          <visual>
59              <geometry>
60                  <cylinder radius="${laser_radius}" length="${laser_length}" />
61              </geometry>
62              <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0
63 0.0" />
64              <material name="black" />
65          </visual>
66          <collision>
67              <geometry>
68                  <cylinder radius="${laser_radius}" length="${laser_length}" />
69              </geometry>
70              <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0
71 0.0" />
72          </collision>
73          <xacro:cylinder_inertial_matrix
74 m="${laser_m}" r="${laser_radius}"
75 h="${laser_length}" />
76      </link>
77
78      <joint name="laser2support" type="fixed">
79          <parent link="support" />
80          <child link="laser" />

```

```

77         <origin xyz="${laser_x} ${laser_y}
78             ${laser_z}" />
79         </joint>
80         <gazebo reference="laser">
81             <material>Gazebo/Black</material>
82         </gazebo>
83     </robot>

```

D.组合底盘、摄像头与雷达的 Xacro 文件

```

1 <!-- 组合小车底盘与摄像头 -->
2 <robot name="my_car_camera"
3     xmlns:xacro="http://wiki.ros.org/xacro">
4     <xacro:include filename="my_head.urdf.xacro" />
5     <xacro:include filename="my_base.urdf.xacro" />
6     <xacro:include filename="my_camera.urdf.xacro"
7     />
8     <xacro:include filename="my_laser.urdf.xacro" />
9 </robot>

```

3.在 gazebo 中执行

launch 文件:

```

1 <launch>
2   <!-- 将 Urdf 文件的内容加载到参数服务器 -->
3   <param name="robot_description" command="$(find
xacro)/xacro $(find
demo02_urdf_gazebo)/urdf/xacro/my_base_camera_laser.
urdf.xacro" />
4   <!-- 启动 gazebo -->
5   <include file="$(find
gazebo_ros)/launch/empty_world.launch" />
6
7   <!-- 在 gazebo 中显示机器人模型 -->
8   <node pkg="gazebo_ros" type="spawn_model"
name="model" args="-urdf -model mycar -param
robot_description" />
9 </launch>

```

6.6.4 Gazebo仿真环境搭建

到目前为止，我们已经可以将机器人模型显示在 Gazebo 之中了，但是当前默认情况下，在 Gazebo 中机器人模型是在 empty world 中，并没有类似于房间、家具、道路、树木... 之类的仿真物，如何在 Gazebo 中创建仿真环境呢？

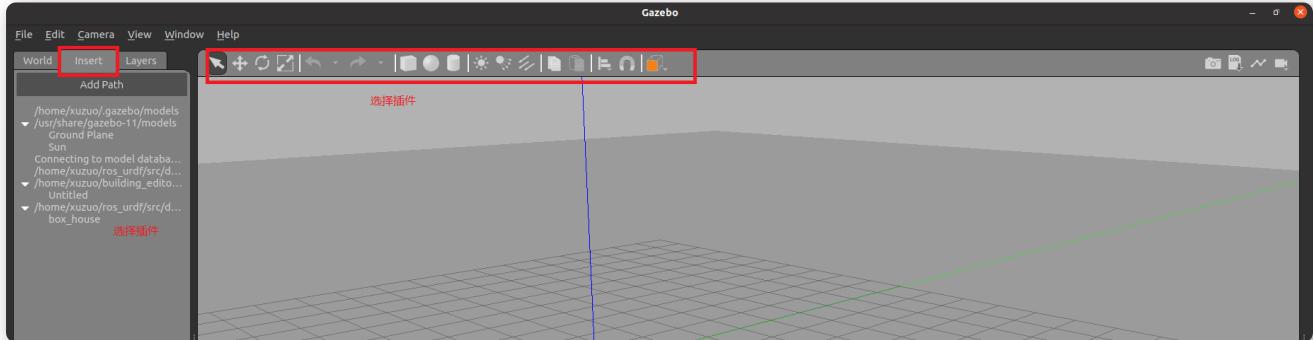
Gazebo 中创建仿真实现方式有两种：

- 方式1：直接添加内置组件创建仿真环境
- 方式2：手动绘制仿真环境(更为灵活)

也还可以直接下载使用官方或第三方提高的仿真环境插件。

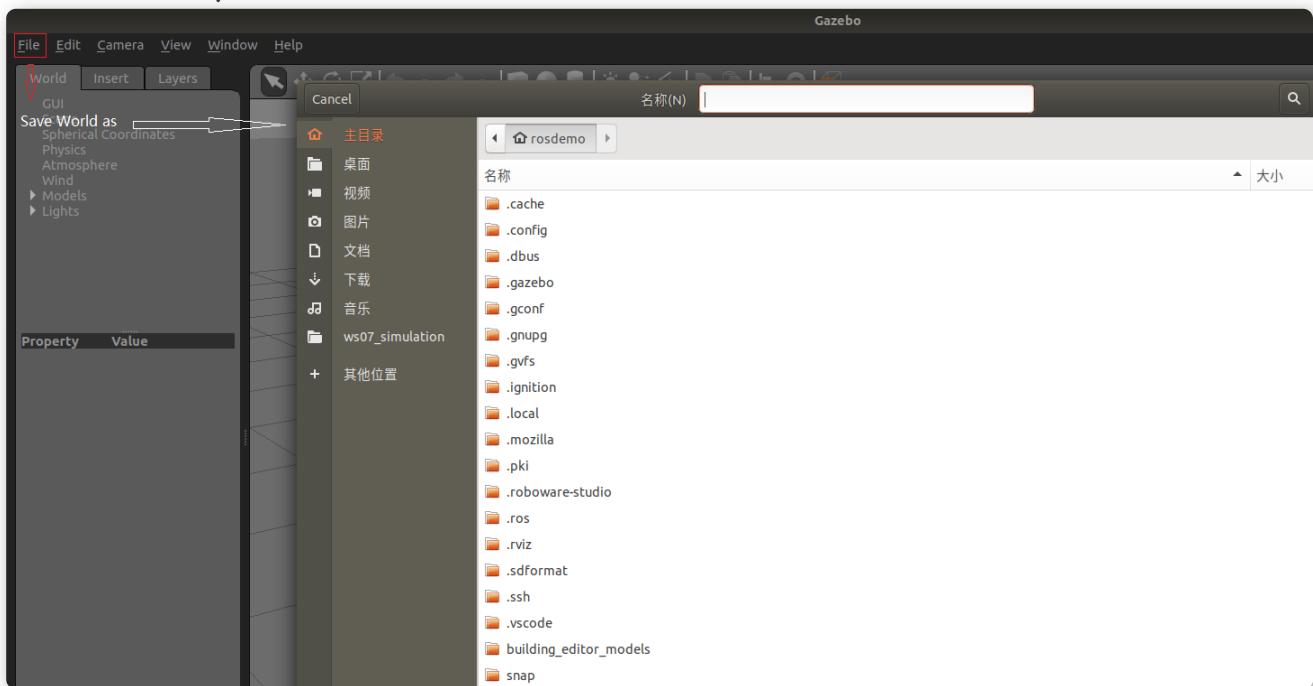
1. 添加内置组件创建仿真环境

1.1启动 Gazebo 并添加组件



1.2保存仿真环境

添加完毕后, 选择 file ---> Save World as 选择保存路径(功能包下: worlds 目录), 文件名自定义, 后缀名设置为 .world



1.3 启动

```

1 <launch>
2
3     <!-- 将 Urdf 文件的内容加载到参数服务器 -->
4     <param name="robot_description"
5         command="$(find xacro)/xacro $(find
6             demo02_urdf_gazebo)/urdf/xacro/my_base_camera_lase
7             r.urdf.xacro" />
8     <!-- 启动 gazebo -->
9     <include file="$(find
10        gazebo_ros)/launch/empty_world.launch">
11         <arg name="world_name" value="$(find
12             demo02_urdf_gazebo)/worlds/hello.world" />
13     </include>
14
15     <!-- 在 gazebo 中显示机器人模型 -->
16     <node pkg="gazebo_ros" type="spawn_model"
17         name="model" args="-urdf -model mycar -param
18             robot_description" />
19 </launch>

```

核心代码: 启动 empty_world 后, 再根据 `arg` 加载自定义的仿真环境

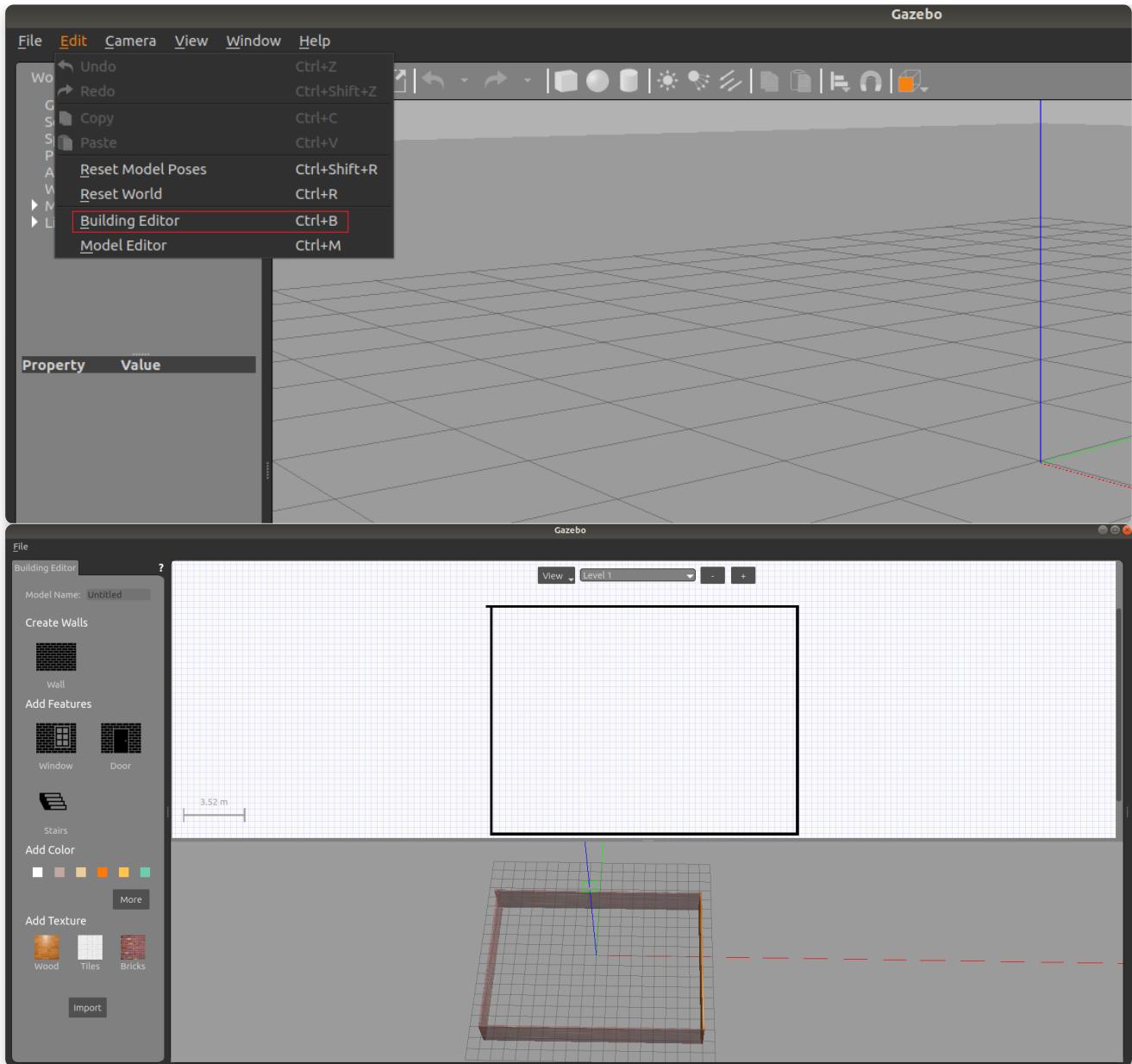
```

1 <include file="$(find
2     gazebo_ros)/launch/empty_world.launch">
3     <arg name="world_name" value="$(find
4         demo02_urdf_gazebo)/worlds/hello.world" />
5     </include>

```

2.自定义仿真环境

2.1 启动 gazebo 打开构建面板, 绘制仿真环境



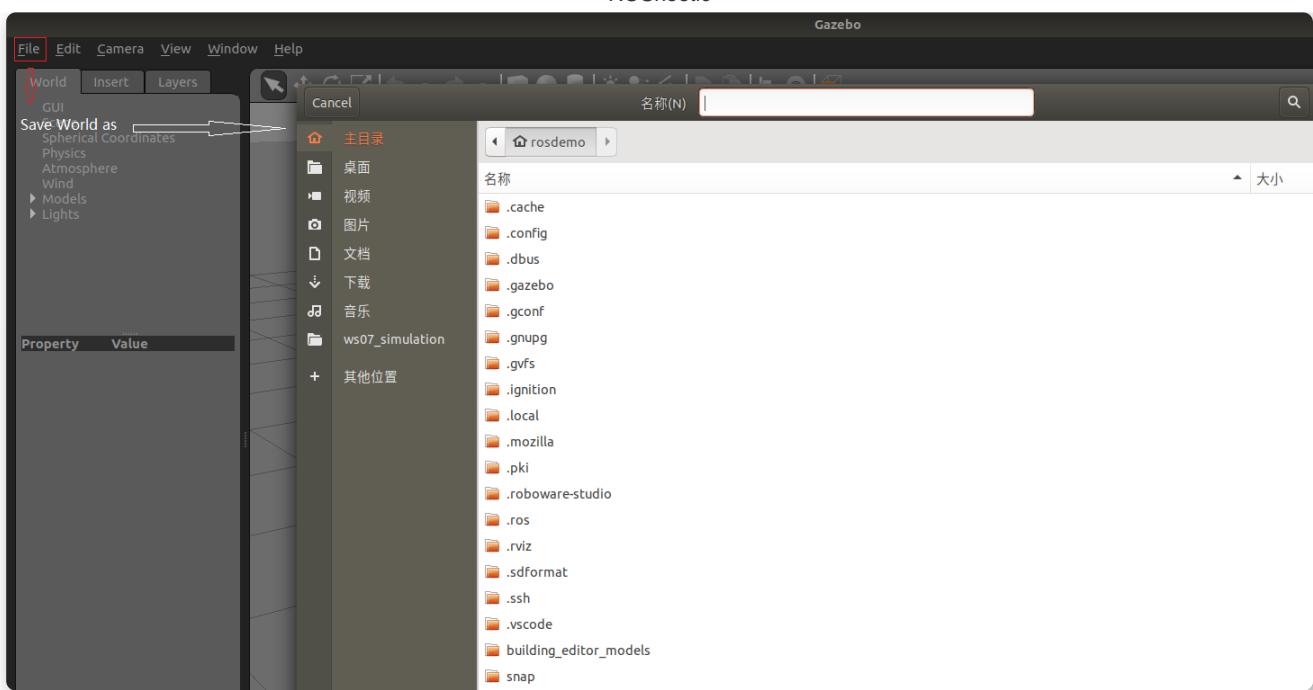
2.2 保存构建的环境

点击: 左上角 file ---> Save (保存路径功能包下的: models)

然后 file ---> Exit Building Editor

2.3 保存为 world 文件

可以像方式1一样再添加一些插件, 然后保存为 world 文件(保存路径功能包下的: worlds)



2.4 启动

同方式1

3. 使用官方提供的插件

当前 Gazebo 提供的仿真道具有限，还可以下载官方支持，可以提供更为丰富的仿真实现，具体实现如下：

3.1 下载官方模型库

```
1 git clone https://github.com/osrf/gazebo_models
```

之前是: hg clone

https://bitbucket.org/osrf/gazebo_models 但是已经不可用

注意: 此过程可能比较耗时

3.2 将模型库复制进 gazebo

将得到的gazebo_models文件夹内容复制到 /usr/share/gazebo-*/models

3.3 应用

重启 Gazebo，选择左侧菜单栏的 insert 可以选择并插入相关道具了

6.7 URDF、Gazebo与Rviz综合应用

关于URDF(Xacro)、Rviz 和 Gazebo 三者的关系，前面已有阐述: URDF 用于创建机器人模型、Rviz 可以显示机器人感知到的环境信息，Gazebo 用于仿真，可以模拟外界环境，以及机器人的一些传感器，如何在 Gazebo 中运行这些传感器，并显示这些传感器的数据(机器人的视角)呢？本节主要介绍的重点就是将三者结合:通过 Gazebo 模拟机器人的传感器，然后在 Rviz 中显示这些传感器感知到的数据。主要内容包括:

- 运动控制以及里程计信息显示
- 雷达信息仿真以及显示
- 摄像头信息仿真以及显示
- kinect 信息仿真以及显示

另请参考:

- http://gazebosim.org/tutorials?tut=ros_gzplugins

6.7.1 机器人运动控制以及里程计信息显示

gazebo 中已经可以正常显示机器人模型了，那么如何像在 rviz 中一样控制机器人运动呢？在此，需要涉及到ros中的组件: ros_control。

1.ros_control 简介

场景:同一套 ROS 程序，如何部署在不同的机器人系统上，比如：开发阶段为了提高效率是在仿真平台上测试的，部署时又有不同的实体机器人平台，不同平台的实现是有差异的，如何保证 ROS 程序的可移植性？ROS 内置的解决方式是 ros_control。

ros_control:是一组软件包，它包含了控制器接口，控制器管理器，传输和硬件接口。**ros_control** 是一套机器人控制的中间件，是一套规范，不同的机器人平台只要按照这套规范实现，那么就可以保证与ROS 程序兼容，通过这套规范，实现了一种可插拔的架构设计，大大提高了程序设计的效率与灵活性。

gazebo 已经实现了 **ros_control** 的相关接口，如果需要在 **gazebo** 中控制机器人运动，直接调用相关接口即可

2.运动控制实现流程(Gazebo)

承上，运动控制基本流程：

1. 已经创建完毕的机器人模型，编写一个单独的 xacro 文件，为机器人模型添加传动装置以及控制器
2. 将此文件集成进xacro文件
3. 启动 Gazebo 并发布 /cmd_vel 消息控制机器人运动

2.1 为 joint 添加传动装置以及控制器

两轮差速配置

```

1 <robot name="my_car_move"
  xmlns:xacro="http://wiki.ros.org/xacro">
2
3     <!-- 传动实现:用于连接控制器与关节 -->
4     <xacro:macro name="joint_trans"
5         params="joint_name">
6         <!-- Transmission is important to link the
7             joints and the controller -->
8         <transmission name="${joint_name}_trans">
9
10        <type>transmission_interface/SimpleTransmission</
11        type>
12        <joint name="${joint_name}">
13            <hardwareInterface>hardware_interface/VelocityJoi
14            ntInterface</hardwareInterface>
15        </joint>

```

```

11          <actuator name="${joint_name}_motor">
12
13      <hardwareInterface>hardware_interface/VelocityJointInterface</hardwareInterface>
14
15      <mechanicalReduction>1</mechanicalReduction>
16          </actuator>
17          </transmission>
18      </xacro:macro>
19
20      <!-- 每一个驱动轮都需要配置传动装置 -->
21      <xacro:joint_trans
22          joint_name="left_wheel2base_link" />
23      <xacro:joint_trans
24          joint_name="right_wheel2base_link" />
25
26      <!-- 控制器 -->
27      <gazebo>
28          <plugin
29              name="differential_drive_controller"
30              filename="libgazebo_ros_diff_drive.so">
31              <rosDebugLevel>Debug</rosDebugLevel>
32              <publishWheelTF>true</publishWheelTF>
33              <robotNamespace>/</robotNamespace>
34              <publishTf>1</publishTf>
35
36          <publishWheelJointState>true</publishWheelJointState>
37              <alwaysOn>true</alwaysOn>
38              <updateRate>100.0</updateRate>
39              <legacyMode>true</legacyMode>
40
41          <leftJoint>left_wheel2base_link</leftJoint> <!--
42          左轮 -->
43
44          <rightJoint>right_wheel2base_link</rightJoint>
45          <!-- 右轮 -->
46              <wheelSeparation>${base_link_radius *
47              2}</wheelSeparation> <!-- 车轮间距 -->

```

```

36           <wheelDiameter>${wheel_radius * 2}
37           </wheelDiameter> <!-- 车轮直径 -->
38           <broadcastTF>1</broadcastTF>
39           <wheelTorque>30</wheelTorque>
40
41           <wheelAcceleration>1.8</wheelAcceleration>
42           <commandTopic>cmd_vel</commandTopic>
43           <!-- 运动控制话题 -->
44           <odometryFrame>odom</odometryFrame>
45           <odometryTopic>odom</odometryTopic>
46           <!-- 里程计话题 -->
47
48           <robotBaseFrame>base_footprint</robotBaseFrame>
49           <!-- 根坐标系 -->
50           </plugin>
51
52       </gazebo>
53
54   </robot>

```

2.2 xacro文件集成

最后还需要将上述 xacro 文件集成进总的机器人模型文件，代码示例如下：

```

1 <!-- 组合小车底盘与摄像头 -->
2 <robot name="my_car_camera"
3   xmlns:xacro="http://wiki.ros.org/xacro">
4   <xacro:include filename="my_head.urdf.xacro" />
5   <xacro:include filename="my_base.urdf.xacro" />
6   <xacro:include filename="my_camera.urdf.xacro" />
7   <xacro:include filename="my_laser.urdf.xacro" />
8   <xacro:include filename="move.urdf.xacro" />
9
10 </robot>

```

当前核心：包含 控制器以及传动配置的 xacro 文件

```

1 <xacro:include filename="move.urdf.xacro" />

```

2.3 启动 gazebo 并控制机器人运动

launch文件:

```

1 <launch>
2
3     <!-- 将 Urdf 文件的内容加载到参数服务器 -->
4     <param name="robot_description"
5         command="$(find xacro)/xacro $(find
6             demo02_urdf_gazebo)/urdf/xacro/my_base_camera_lase
7             r.urdf.xacro" />
8     <!-- 启动 gazebo -->
9     <include file="$(find
10        gazebo_ros)/launch/empty_world.launch">
11         <arg name="world_name" value="$(find
12             demo02_urdf_gazebo)/worlds/hello.world" />
13     </include>
14
15     <!-- 在 gazebo 中显示机器人模型 -->
16     <node pkg="gazebo_ros" type="spawn_model"
17         name="model" args="-urdf -model mycar -param
18             robot_description" />
19 </launch>

```

启动 launch 文件，使用 topic list 查看话题列表，会发现多了 /cmd_vel
然后发布 vmd_vel 消息控制即可

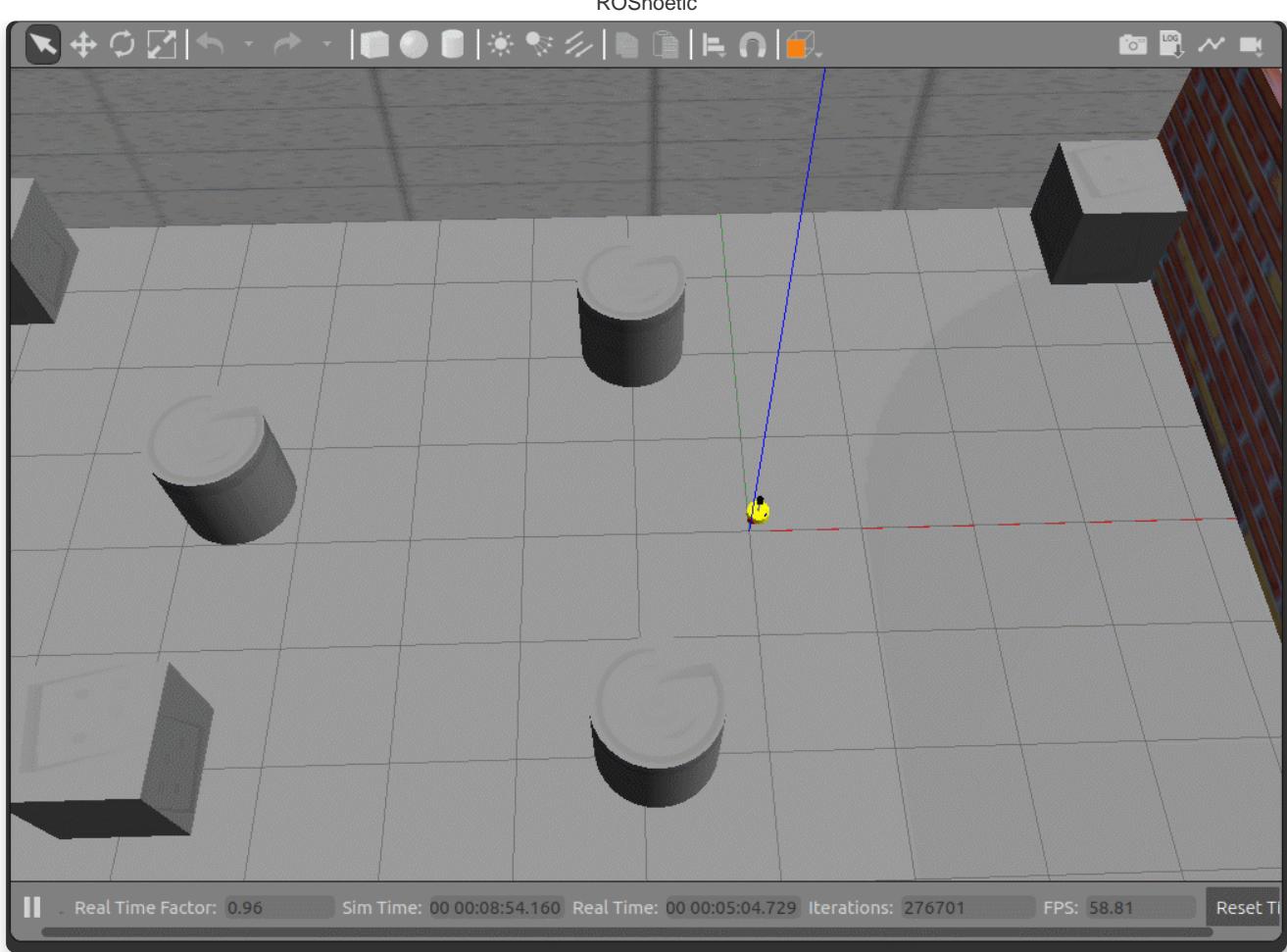
使用命令控制(或者可以编写单独的节点控制)

```

1 rostopic pub -r 10 /cmd_vel geometry_msgs/Twist
2     '{linear: {x: 0.2, y: 0, z: 0}, angular: {x: 0, y:
3         0, z: 0.5}}'

```

接下来我们会发现: 小车在 Gazebo 中已经正常运行起来了



3.Rviz查看里程计信息

在 Gazebo 的仿真环境中，机器人的里程计信息以及运动朝向等信息是无法获取的，可以通过 Rviz 显示机器人的里程计信息以及运动朝向

里程计: 机器人相对出发点坐标系的位姿状态(X 坐标 Y 坐标 Z坐标以及朝向)。

3.1启动 Rviz

launch 文件

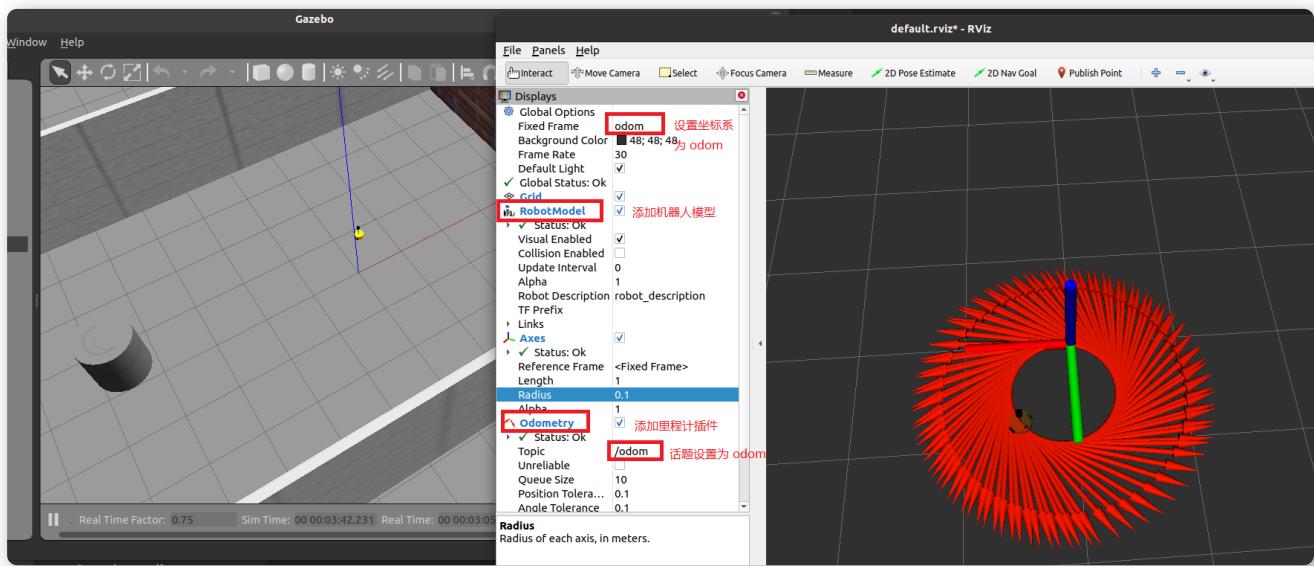
```

1 <launch>
2     <!-- 启动 rviz -->
3     <node pkg="rviz" type="rviz" name="rviz" />
4
5     <!-- 关节以及机器人状态发布节点 -->
6     <node name="joint_state_publisher"
7         pkg="joint_state_publisher"
8         type="joint_state_publisher" />
9     <node name="robot_state_publisher"
10        pkg="robot_state_publisher"
11        type="robot_state_publisher" />
12
13 </launch>

```

3.2 添加组件

执行 launch 文件后，在 Rviz 中添加图示组件：



6.7.2 雷达信息仿真以及显示

通过 Gazebo 模拟激光雷达传感器，并在 Rviz 中显示激光数据。

实现流程：

雷达仿真基本流程：

1. 已经创建完毕的机器人模型，编写一个单独的 xacro 文件，为机器人模型添加雷达配置；

2. 将此文件集成进xacro文件；
3. 启动 Gazebo，使用 Rviz 显示雷达信息。

1.Gazebo 仿真雷达

1.1 新建 Xacro 文件，配置雷达传感器信息

```
1 <robot name="my_sensors"
2   xmlns:xacro="http://wiki.ros.org/xacro">
3
4   <!-- 雷达 -->
5   <gazebo reference="laser">
6     <sensor type="ray" name="rplidar">
7       <pose>0 0 0 0 0 0</pose>
8       <visualize>true</visualize>
9       <update_rate>5.5</update_rate>
10      <ray>
11        <scan>
12          <horizontal>
13            <samples>360</samples>
14            <resolution>1</resolution>
15            <min_angle>-3</min_angle>
16            <max_angle>3</max_angle>
17          </horizontal>
18        </scan>
19        <range>
20          <min>0.10</min>
21          <max>30.0</max>
22          <resolution>0.01</resolution>
23        </range>
24        <noise>
25          <type>gaussian</type>
26          <mean>0.0</mean>
27          <stddev>0.01</stddev>
28        </noise>
29      </ray>
30      <plugin name="gazebo_rplidar"
  filename="libgazebo_ros_laser.so">
    <topicName>/scan</topicName>
```

```

31      <frameName>laser</frameName>
32      </plugin>
33  </sensor>
34  </gazebo>
35
36 </robot>

```

1.2 xacro 文件集成

将步骤1的 Xacro 文件集成进总的机器人模型文件，代码示例如下：

```

1 <!-- 组合小车底盘与传感器 -->
2 <robot name="my_car_camera"
  xmlns:xacro="http://wiki.ros.org/xacro">
3   <xacro:include filename="my_head.urdf.xacro">
4   </>
5   <xacro:include filename="my_base.urdf.xacro">
6   </>
7   <xacro:include filename="my_camera.urdf.xacro">
8   </>
9   <xacro:include filename="my_laser.urdf.xacro">
10  </>
11  <xacro:include filename="move.urdf.xacro" />
12  <!-- 雷达仿真的 xacro 文件 -->
13  <xacro:include
14    filename="my_sensors_laser.urdf.xacro" />
15 </robot>

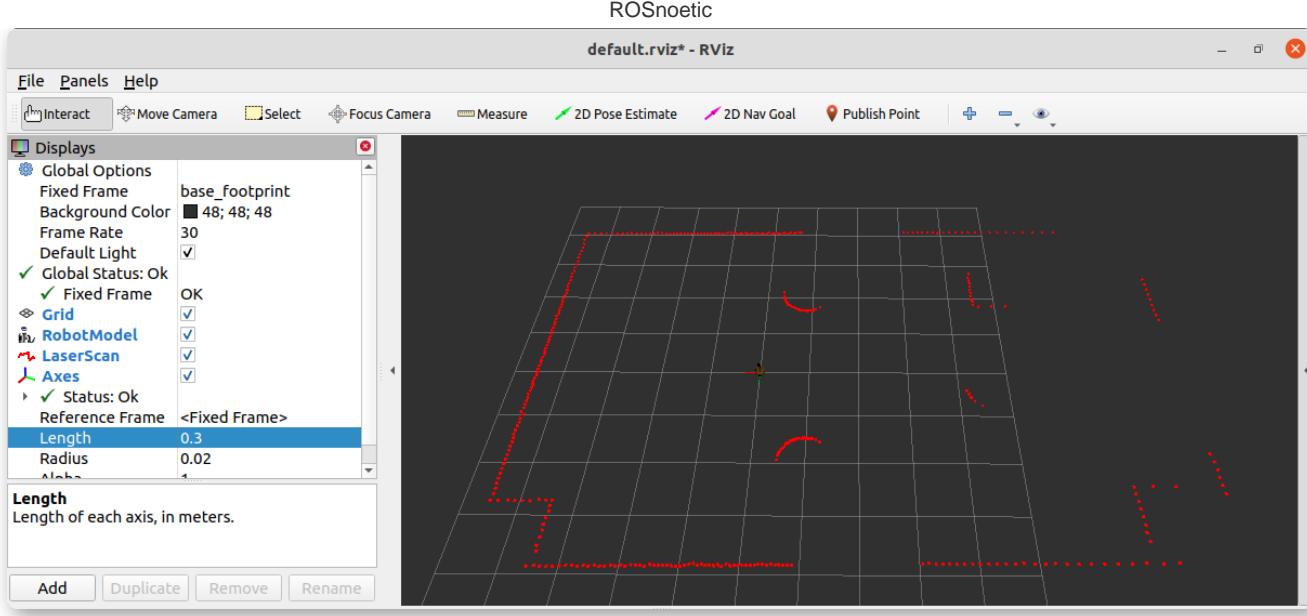
```

1.3启动仿真环境

编写launch文件，启动gazebo，此处略...

2.Rviz 显示雷达数据

先启动 rviz,添加雷达信息显示插件



6.7.3 摄像头信息仿真以及显示

通过 Gazebo 模拟摄像头传感器，并在 Rviz 中显示摄像头数据。

实现流程：

摄像头仿真基本流程：

1. 已经创建完毕的机器人模型，编写一个单独的 xacro 文件，为机器人模型添加摄像头配置；
2. 将此文件集成进 xacro 文件；
3. 启动 Gazebo，使用 Rviz 显示摄像头信息。

1.Gazebo 仿真摄像头

1.1 新建 Xacro 文件，配置摄像头传感器信息

```

1 <robot name="my_sensors"
  xmlns:xacro="http://wiki.ros.org/xacro">
2   <!-- 被引用的link -->
3   <gazebo reference="camera">
4     <!-- 类型设置为 camera -->
5     <sensor type="camera" name="camera_node">
6       <update_rate>30.0</update_rate> <!-- 更新频率
-->

```

```

7      <!-- 摄像头基本信息设置 -->
8      <camera name="head">
9          <horizontal_fov>1.3962634</horizontal_fov>
10         <image>
11             <width>1280</width>
12             <height>720</height>
13             <format>R8G8B8</format>
14         </image>
15         <clip>
16             <near>0.02</near>
17             <far>300</far>
18         </clip>
19         <noise>
20             <type>gaussian</type>
21             <mean>0.0</mean>
22             <stddev>0.007</stddev>
23         </noise>
24     </camera>
25     <!-- 核心插件 -->
26     <plugin name="gazebo_camera"
27         filename="libgazebo_ros_camera.so">
28         <alwaysOn>true</alwaysOn>
29         <updateRate>0.0</updateRate>
30         <cameraName>/camera</cameraName>
31         <imageTopicName>image_raw</imageTopicName>
32
33         <cameraInfoTopicName>camera_info</cameraInfoTopic
34             Name>
35             <frameName>camera</frameName>
36             <hackBaseline>0.07</hackBaseline>
37             <distortionK1>0.0</distortionK1>
38             <distortionK2>0.0</distortionK2>
39             <distortionK3>0.0</distortionK3>
40             <distortionT1>0.0</distortionT1>
41             <distortionT2>0.0</distortionT2>
42         </plugin>
43     </sensor>
44 </gazebo>
45 </robot>

```

1.2 xacro 文件集成

将步骤1的 Xacro 文件集成进总的机器人模型文件，代码示例如下：

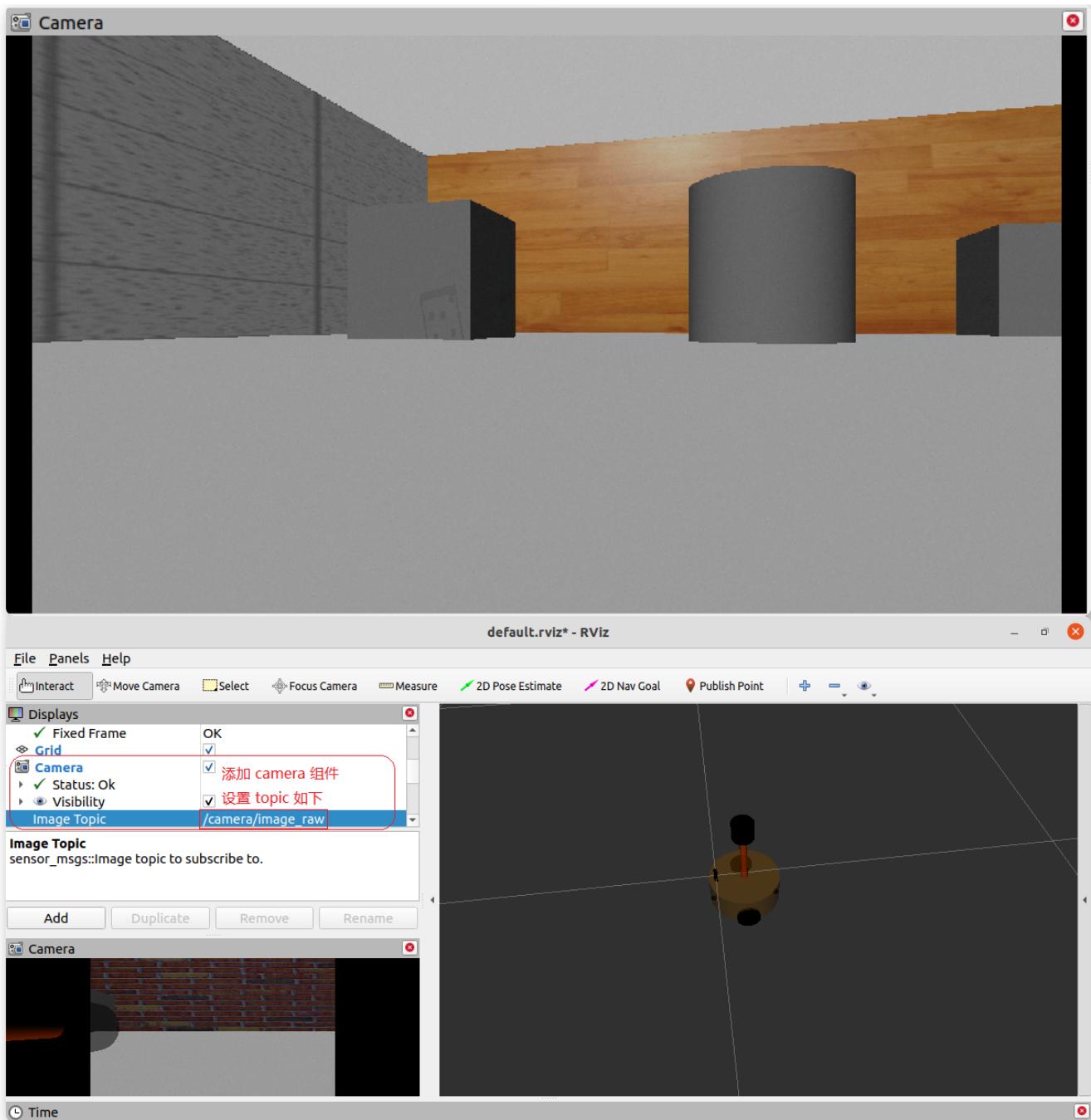
```
1 <!-- 组合小车底盘与传感器 -->
2 <robot name="my_car_camera"
  xmlns:xacro="http://wiki.ros.org/xacro">
3   <xacro:include filename="my_head.urdf.xacro"
  />
4   <xacro:include filename="my_base.urdf.xacro"
  />
5   <xacro:include filename="my_camera.urdf.xacro"
  />
6   <xacro:include filename="my_laser.urdf.xacro"
  />
7   <xacro:include filename="move.urdf.xacro" />
8   <!-- 摄像头仿真的 xacro 文件 -->
9   <xacro:include
  filename="my_sensors_camara.urdf.xacro" />
10 </robot>
```

1.3启动仿真环境

编写launch文件，启动gazebo，此处略...

2.Rviz 显示摄像头数据

执行 gazebo 并启动 Rviz, 在 Rviz 中添加摄像头组件。



6.7.4 kinect信息仿真以及显示

通过 Gazebo 模拟kinect摄像头，并在 Rviz 中显示kinect摄像头数据。

实现流程：

kinect摄像头仿真基本流程：

1. 已经创建完毕的机器人模型，编写一个单独的 xacro 文件，为机器人模型添加kinect摄像头配置；
2. 将此文件集成进xacro文件；
3. 启动 Gazebo，使用 Rviz 显示kinect摄像头信息。

1. Gazebo仿真Kinect

1.1 新建 Xacro 文件，配置 kinetic 传感器信息

```
1 <robot name="my_sensors"
2   xmlns:xacro="http://wiki.ros.org/xacro">
3     <gazebo reference="kinect link名称">
4       <sensor type="depth" name="camera">
5         <always_on>true</always_on>
6         <update_rate>20.0</update_rate>
7         <camera>
8           <horizontal_fov>${60.0*PI/180.0}
9         </horizontal_fov>
10        <image>
11          <format>R8G8B8</format>
12          <width>640</width>
13          <height>480</height>
14        </image>
15        <clip>
16          <near>0.05</near>
17          <far>8.0</far>
18        </clip>
19      </camera>
20      <plugin name="kinect_camera_controller"
21        filename="libgazebo_ros_openni_kinect.so">
22        <cameraName>camera</cameraName>
23        <alwaysOn>true</alwaysOn>
24        <updateRate>10</updateRate>
25
26        <imageTopicName>rgb/image_raw</imageTopicName>
27
28        <depthImageTopicName>depth/image_raw</depthImageT
29          opicName>
30
31        <pointCloudTopicName>depth/points</pointCloudTopi
32          cName>
```

```
25 <cameraInfoTopicName>rgb/camera_info</cameraInfoTopicName>
26 <depthImageCameraInfoTopicName>depth/camera_info</depthImageCameraInfoTopicName>
27     <frameName>kinect link名称</frameName>
28     <baseline>0.1</baseline>
29     <distortion_k1>0.0</distortion_k1>
30     <distortion_k2>0.0</distortion_k2>
31     <distortion_k3>0.0</distortion_k3>
32     <distortion_t1>0.0</distortion_t1>
33     <distortion_t2>0.0</distortion_t2>
34     <pointCloudCutoff>0.4</pointCloudCutoff>
35     </plugin>
36   </sensor>
37 </gazebo>
38
39 </robot>
```

1.2 xacro 文件集成

将步骤1的 Xacro 文件集成进总的机器人模型文件，代码示例如下：

```

1 <!-- 组合小车底盘与传感器 -->
2 <robot name="my_car_camera"
  xmlns:xacro="http://wiki.ros.org/xacro">
3   <xacro:include filename="my_head.urdf.xacro"
  />
4   <xacro:include filename="my_base.urdf.xacro"
  />
5   <xacro:include filename="my_camera.urdf.xacro"
  />
6   <xacro:include filename="my_laser.urdf.xacro"
  />
7   <xacro:include filename="move.urdf.xacro" />
8   <!-- kinect仿真的 xacro 文件 -->
9   <xacro:include
  filename="my_sensors_kinect.urdf.xacro" />
10 </robot>

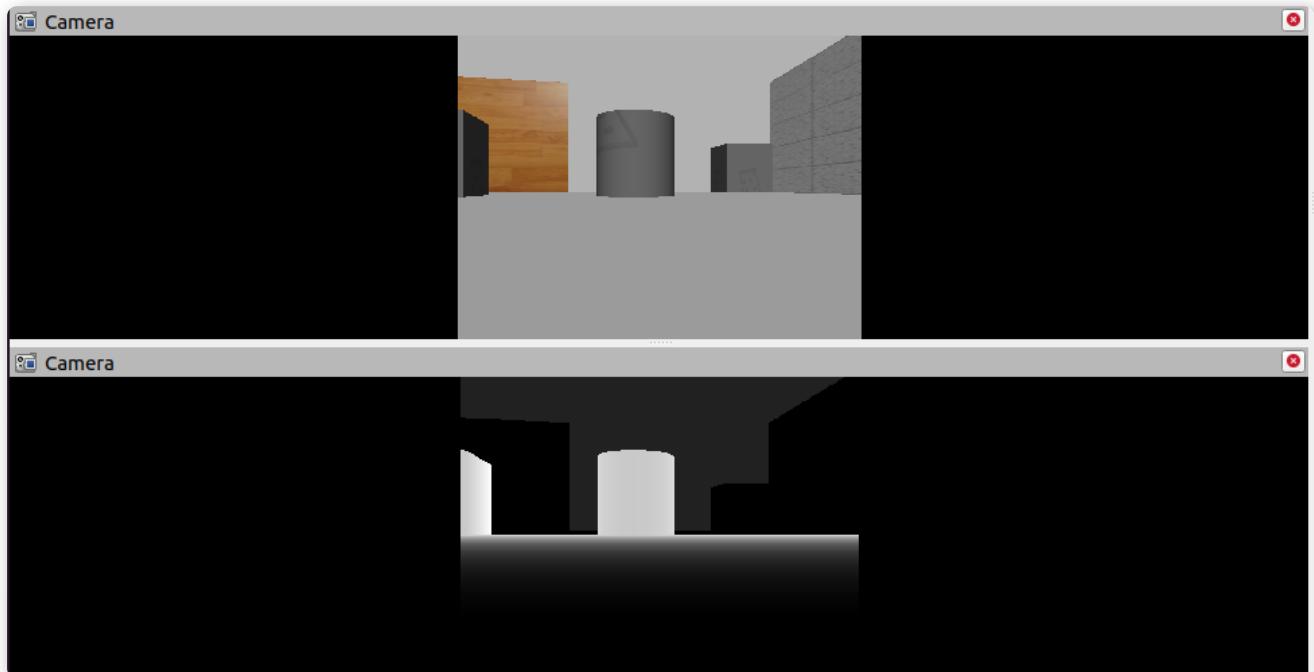
```

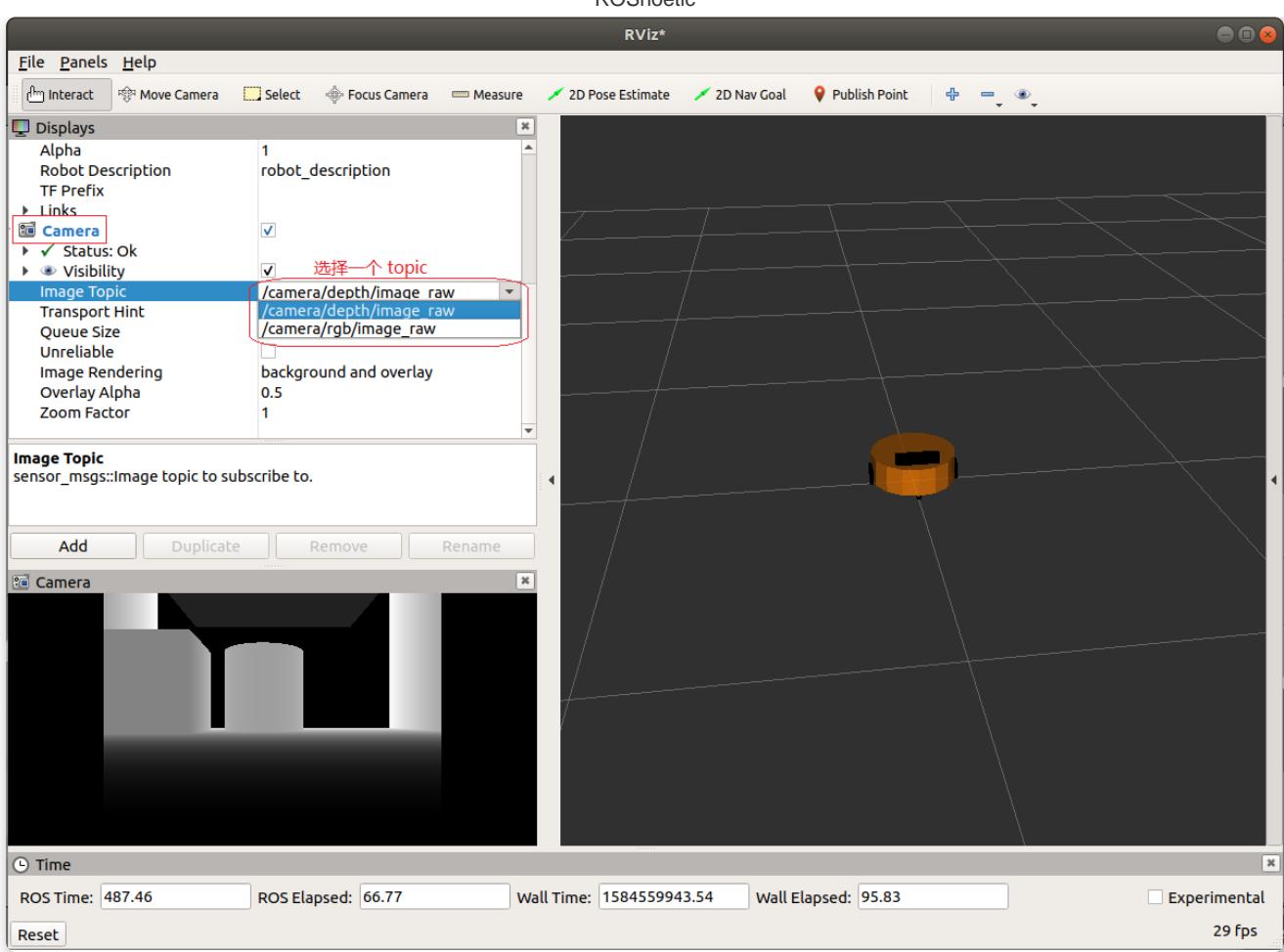
1.3启动仿真环境

编写launch文件，启动gazebo，此处略...

2 Rviz 显示 Kinect 数据

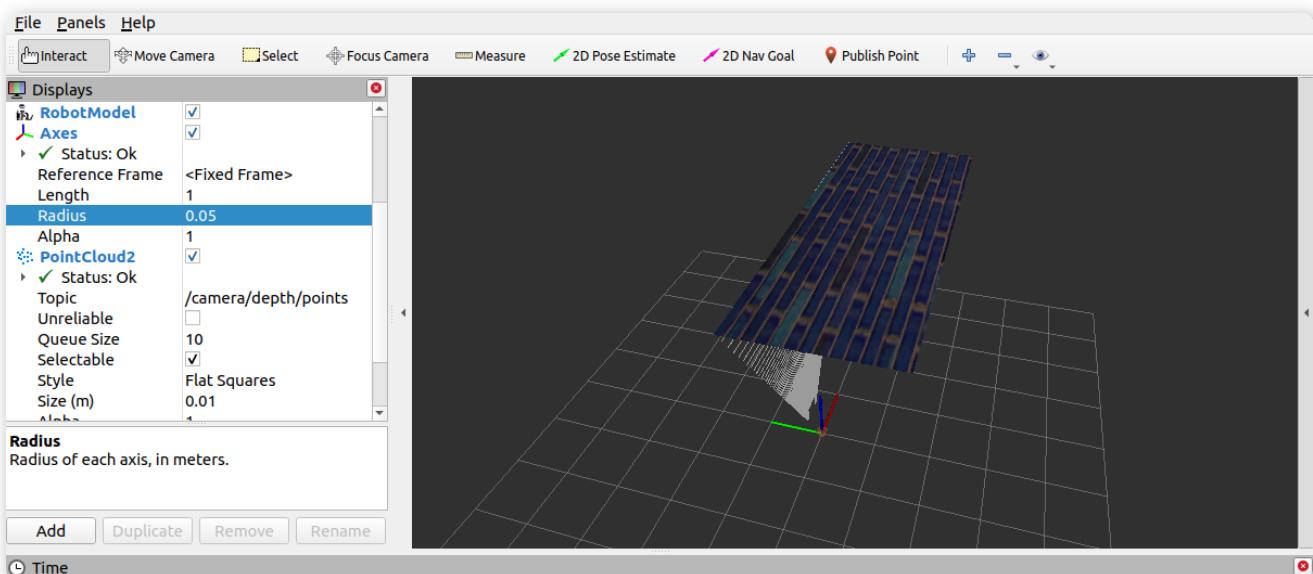
启动 rviz,添加摄像头组件查看数据





补充:kinect 点云数据显示

在kinect中也可以以点云的方式显示感知周围环境，在 rviz 中操作如下:



问题:在rviz中显示时错位。

原因:在kinect中图像数据与点云数据使用了两套坐标系统,且两套坐标系统位姿并不一致。

解决:

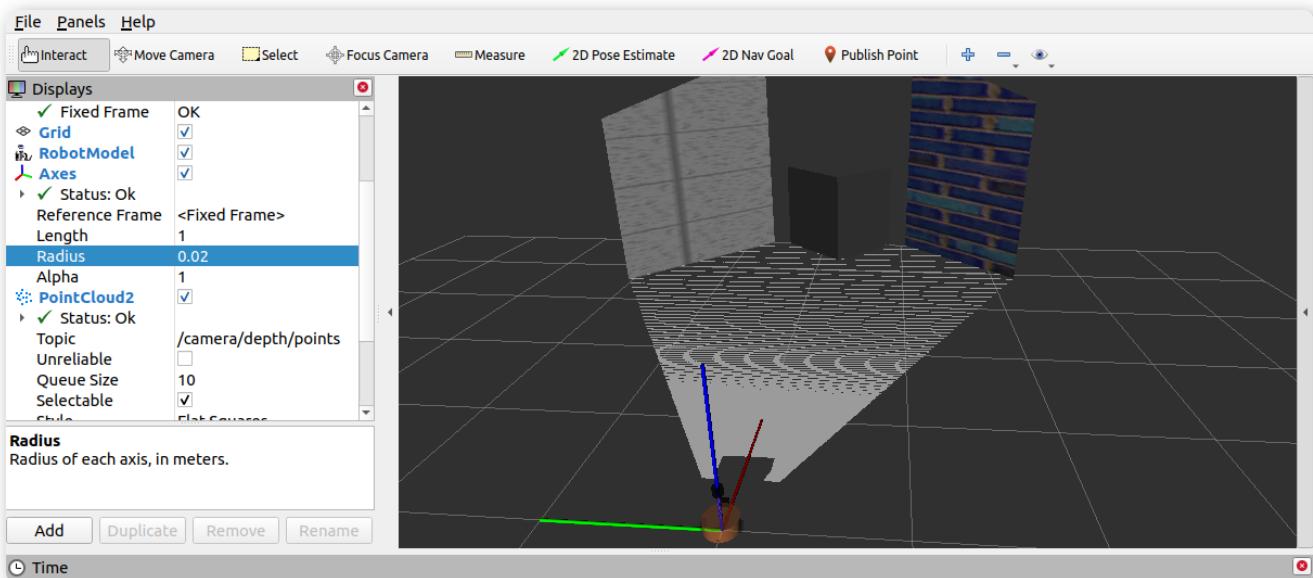
1.在插件中为kinect设置坐标系,修改配置文件的 `<frameName>` 标签内容:

```
1 <frameName>support_depth</frameName>
```

2.发布新设置的坐标系到kinect连杆的坐标变换关系,在启动rviz的launch中,添加:

```
1 <node pkg="tf2_ros"
  type="static_transform_publisher"
  name="static_transform_publisher" args="0 0 0 -1.57
  0 -1.57 /support /support_depth" />
```

3.启动rviz,重新显示。



6.8 本章小结

本章主要介绍了ROS中仿真实现涉及的三大知识点:

- URDF(Xacro)
- Rviz

- Gazebo

URDF 是用于描述机器人模型的 xml 文件，可以使用不同的标签具代表不同含义，URDF 编写机器人模型代码冗余，xacro 可以优化 URDF 实现，代码实现更为精简、高效、易读。容易混淆的是Rviz与Gazebo，在此我们着重比较以下二者的区别：



rviz是三维可视化工具，强调把已有的数据可视化显示；

gazebo是三维物理仿真平台，强调的是创建一个虚拟的仿真环境。

rviz需要已有数据。

rviz提供了很多插件，这些插件可以显示图像、模型、路径等信息，但是前提都是这些数据已经以话题、参数的形式发布，rviz做的事情就是订阅这些数据，并完成可视化的渲染，让开发者更容易理解数据的意义。

gazebo不是显示工具，强调的是仿真，它不需要数据，而是创造数据。

我们可以在gazebo中免费创建一个机器人世界，不仅可以仿真机器人的运动功能，还可以仿真机器人的传感器数据。而这些数据就可以放到rviz中显示，所以使用gazebo的时候，经常也会和rviz配合使用。当我们手上没有机器人硬件或实验环境难以搭建时，仿真往往是非常有用的利器。

综上，如果你手上已经有机器人硬件平台，并且在上边可以完成需要的功能，用rviz应该就可以满足开发需求。

如果你手上没有机器人硬件，或者想在仿真环境中做一些算法、应用的测试，gazebo+rviz应该是你需要的。

另外，rviz配合其他功能包也可以建立一个简单的仿真环境，比如rviz+ArbotiX。

第7章 机器人导航(仿真)

导航是机器人系统中最重要的模块之一，比如现在较为流行的服务型室内机器人，就是依赖于机器人导航来实现室内自主移动的，本章主要就是介绍仿真环境下的导航实现，主要内容有：

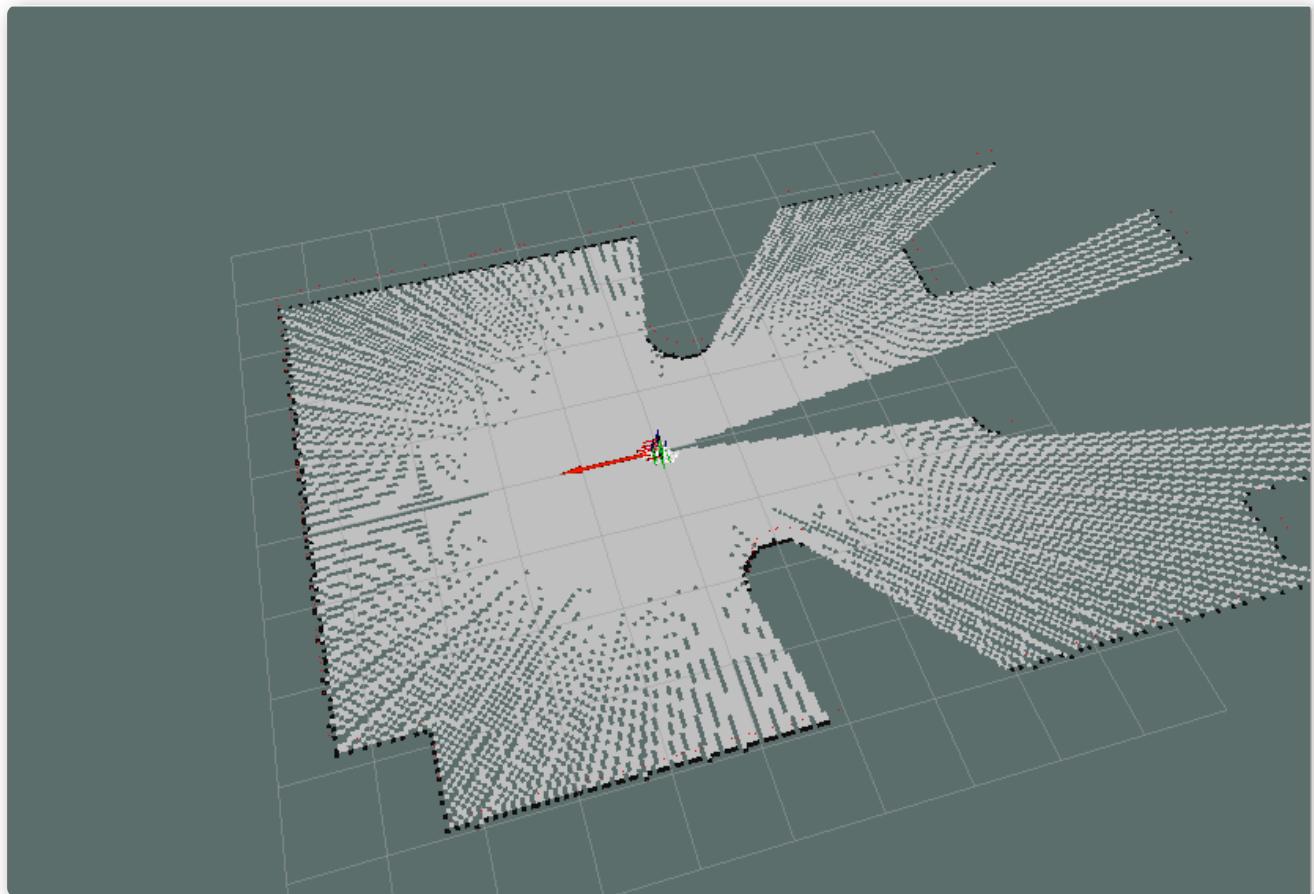
- 导航相关概念
- 导航实现:机器人建图(SLAM)、地图服务、定位、路径规划....以可视化操作为主。
- 导航消息:了解地图、里程计、雷达、摄像头等相关消息格式。

预期达成的学习目标:

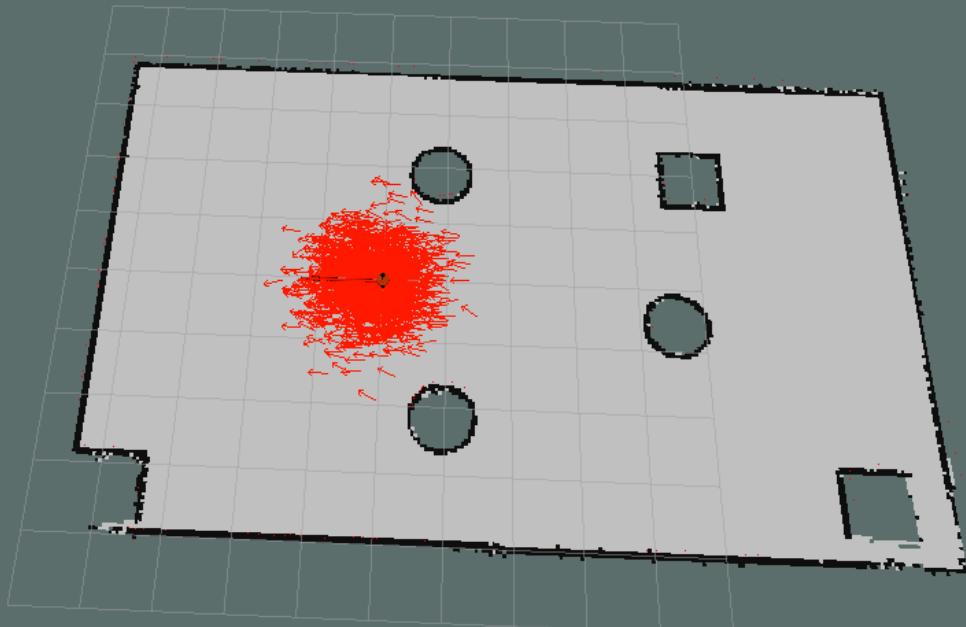
- 了解导航模块中的组成部分以及相关概念
- 能够在仿真环境下独立完成机器人导航

案例演示:

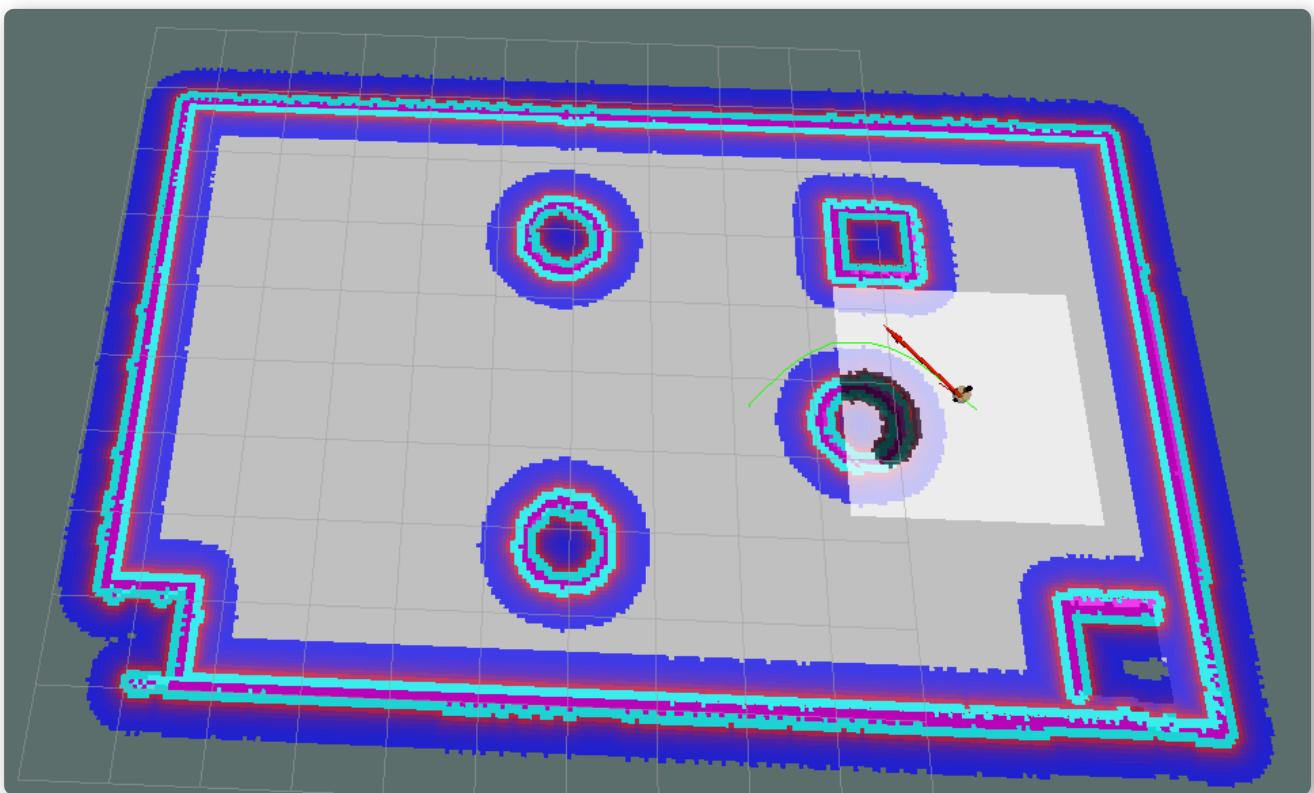
SLAM建图



定位



导航实现



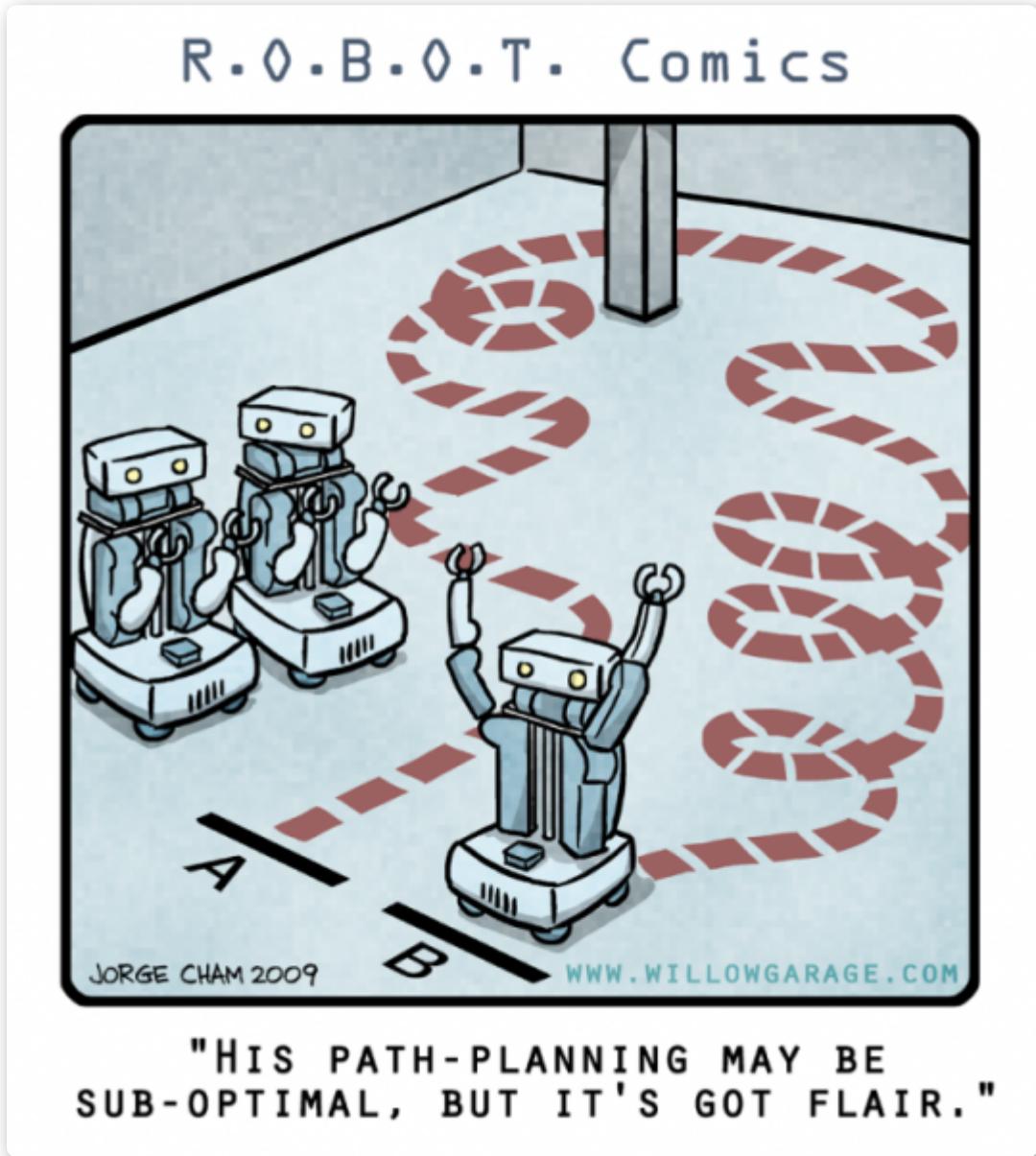
7.1 概述

1. 概念

在ROS中机器人导航(Navigation)由多个功能包组合实现，ROS中又称之为导航功能包集，关于导航模块，官方介绍如下：

- 1 一个二维导航堆栈，它接收来自里程计、传感器流和目标姿态的信息，并输出发送到移动底盘的安全速度命令。

更通俗的讲：导航其实就是机器人自主的从 A 点移动到 B 点的过程。



2. 作用

秉着"不重复发明轮子"的原则，ROS 中导航相关的功能包集为机器人导航提供了一套通用的实现，开发者不再需要关注于导航算法、硬件交互... 等偏复杂、偏底层的实现，这些实现都由更专业的研发人员管理、迭代和维护，开发者可以更专注于上层功能，而对于导航功能的调用，只需要根据自身机器人相关参数合理设置各模块的配置文件即可，当然，如果有必要，也可以基于现有的功能包二次开发实现一些定制化需求，这样可以大大提高研发效率，缩短产品落地时间。总而言之，对于一般开发者而言，ROS 的导航功能包集优势如下：

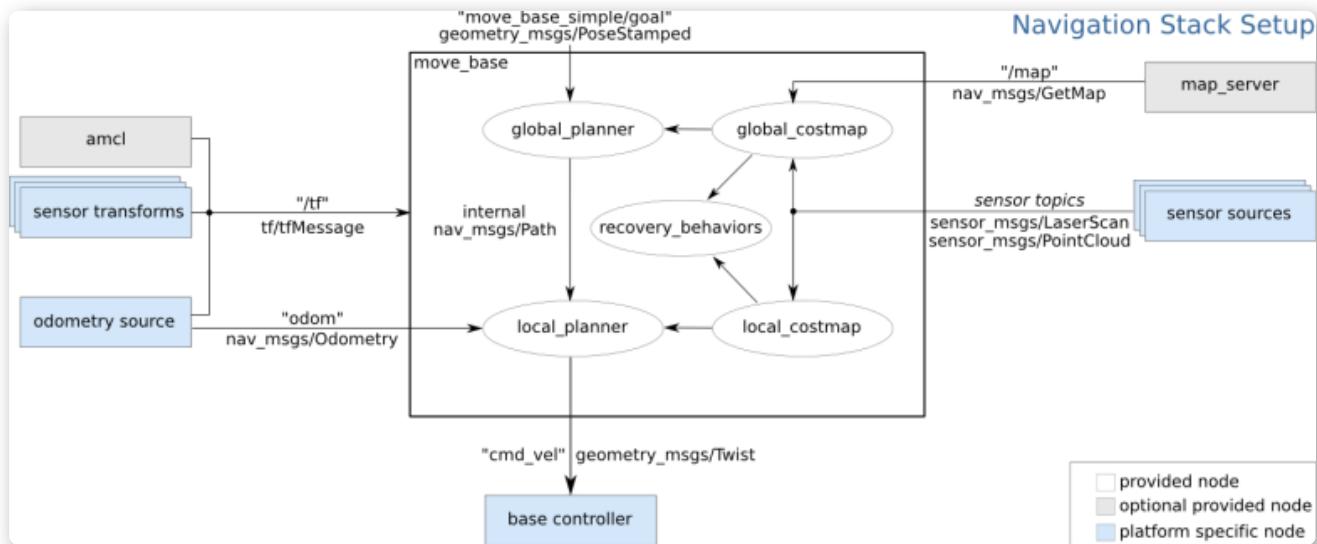
- 安全: 由专业团队开发和维护
- 功能: 功能更稳定且全面
- 高效: 解放开发者, 让开发者更专注于上层功能实现

另请参考:

- <http://wiki.ros.org/navigation>

7.1.1 导航模块简介

机器人是如何实现导航的呢? 或换言之, 机器人是如何从 A 点移动到 B 点呢? ROS 官方为了提供了一张导航功能包集的图示, 该图中囊括了 ROS 导航的一些关键技术:



假定我们已经以特定方式配置机器人, 导航功能包集将使其可以运动。上图概述了这种配置方式。白色的部分是必须且已实现的组件, 灰色的部分是可选且已实现的组件, 蓝色的部分是必须为每一个机器人平台创建的组件。

总结下来, 涉及的关键技术有如下五点:

1. 全局地图
2. 自身定位
3. 路径规划
4. 运动控制
5. 环境感知

机器人导航实现与无人驾驶类似，关键技术也是由上述五点组成，只是无人驾驶是基于室外的，而我们当前介绍的机器人导航更多是基于室内的。

1.全局地图

在现实生活中，当我们需要实现导航时，可能会首先参考一张全局性质的地图，然后根据地图来确定自身的位置、目的地位置，并且也会根据地图显示来规划一条大致的路线.... 对于机器人导航而言，也是如此，在机器人导航中地图是一个重要的组成元素，当然如果要使用地图，首先需要绘制地图。关于地图建模技术不断涌现，这其中有一门称之为 SLAM 的理论脱颖而出：

1. **SLAM**(simultaneous localization and mapping),也称为CML (Concurrent Mapping and Localization), 即时定位与地图构建，或并发建图与定位。SLAM问题可以描述为：机器人在未知环境中从一个未知位置开始移动,在移动过程中根据位置估计和地图进行自身定位，同时在自身定位的基础上建造增量式地图，以绘制出外部环境的完全地图。
2. 在 ROS 中，较为常用的 SLAM 实现也比较多，比如: gmapping、hector_slam、cartographer、rgbdslam、ORB_SLAM
3. 当然如果要完成 SLAM，机器人必须要具备感知外界环境的能力，尤其是要具备获取周围环境深度信息的能力。感知的实现需要依赖于传感器，比如: 激光雷达、摄像头、RGB-D摄像头...
4. SLAM 可以用于地图生成，而生成的地图还需要被保存以待后续使用，在 ROS 中保存地图的功能包是 map_server

另外注意: SLAM 虽然是机器人导航的重要技术之一，但是二者并不等价，确切的讲，SLAM 只是实现地图构建和即时定位。

2.自身定位

导航伊始和导航过程中，机器人都需要确定当前自身的位置，如果在室外，那么 GPS 是一个不错的选择，而如果室内、隧道、地下或一些特殊的屏蔽 GPS 信号的区域，由于 GPS 信号弱化甚至完全不可用，那么就必须另辟蹊径了，比如前面的 SLAM 就可以实现自身定位，除此之外，ROS 中还提供了一个用于定位的功能包: amcl

amcl(adaptiveMonteCarloLocalization)自适应的蒙特卡洛定位,是用于2D移动机器人的概率定位系统。它实现了自适应（或KLD采样）蒙特卡洛定位方法，该方法使用粒子过滤器根据已知地图跟踪机器人的姿态。

3.路径规划

导航就是机器人从A点运动至B点的过程，在这一过程中，机器人需要根据目标位置计算全局运动路线，并且在运动过程中，还需要时时根据出现的一些动态障碍物调整运动路线，直至到达目标点，该过程就称之为路径规划。在ROS中提供了 move_base 包来实现路径规则,该功能包主要由两大规划器组成:

1. 全局路径规划(gloable_planner)

根据给定的目标点和全局地图实现总体的路径规划，使用 Dijkstra 或 A* 算法进行全局路径规划，计算最优路线，作为全局路线

2. 本地时时规划(local_planner)

在实际导航过程中，机器人可能无法按照给定的全局最优路线运行，比如:机器人在运行中，可能会随时出现一定的障碍物... 本地规划的作用就是使用一定算法(Dynamic Window Approaches) 来实现障碍物的规避，并选取当前最优路径以尽量符合全局最优路径

全局路径规划与本地路径规划是相对的，全局路径规划侧重于全局、宏观实现，而本地路径规划侧重与当前、微观实现。

4.运动控制

导航功能包集假定它可以通过话题"cmd_vel"发布 `geometry_msgs/Twist` 类型的消息，这个消息基于机器人的基座坐标系，它传递的是运动命令。这意味着必须有一个节点订阅"cmd_vel"话题，将该话题上的速度命令转换为电机命令并发送。

5.环境感知

感知周围环境信息，比如: 摄像头、激光雷达、编码器...，摄像头、激光雷达可以用于感知外界环境的深度信息，编码器可以感知电机的转速信息，进而可以获取速度信息并生成里程计信息。

在导航功能包集中，环境感知也是一重要模块实现，它为其他模块提供了支持。其他模块诸如: SLAM、amcl、move_base 都需要依赖于环境感知。

7.1.2 导航之坐标系

1.简介

定位是导航中的重要实现之一，所谓定位，就是参考某个坐标系(比如:以机器人的出发点为原点创建坐标系)在该坐标系中标注机器人。定位原理看似简单，但是这个这个坐标系不是客观存在的，我们也无法以上帝视角确定机器人的位姿，定位实现需要依赖于机器人自身，机器人需要逆向推导参考系原点并计算坐标系相对关系，该过程实现常用方式有两种:

- 通过里程计定位:时时收集机器人的速度信息计算并发布机器人坐标系与父级参考系的相对关系。
- 通过传感器定位:通过传感器收集外界环境信息通过匹配计算并发布机器人坐标系与父级参考系的相对关系。

两种方式在导航中都会经常使用。

2.特点

两种定位方式都有各自的优缺点。

里程计定位:

- 优点:里程计定位信息是连续的，没有离散的跳跃。
- 缺点:里程计存在累计误差，不利于长距离或长期定位。

传感器定位:

- 优点:比里程计定位更精准；
- 缺点:传感器定位会出现跳变的情况，且传感器定位在标志物较少的环境下，其定位精度会大打折扣。

两种定位方式优缺点互补，应用时一般二者结合使用。

3.坐标系变换

上述两种定位实现中，机器人坐标系一般使用机器人模型中的根坐标系 (base_link 或 base_footprint)，里程计定位时，父级坐标系一般称之为 odom，如果通过传感器定位，父级参考系一般称之为 map。当二者结合使用时，map 和 odom 都是机器人模型根坐标系的父级，这是不符合坐标变换中"单继承"的原则的，所以，一般会将转换关系设置为: map -> doom -> base_link 或 base_footprint。

另请参考:

- <https://www.ros.org/reps/rep-0105.html>

7.1.3 导航条件说明

导航实现，在硬件和软件方面是由一定要求的，需要提前准备。

1.硬件

虽然导航功能包集被设计成尽可能的通用，在使用时仍然有三个主要的硬件限制：

1. 它是为差速驱动的轮式机器人设计的。它假设底盘受到理想的运动命令的控制并可实现预期的结果，命令的格式为：x速度分量，y速度分量，角速度(theta)分量。
2. 它需要在底盘上安装一个单线激光雷达。这个激光雷达用于构建地图和定位。
3. 导航功能包集是为正方形的机器人开发的，所以方形或圆形的机器人将是性能最好的。它也可以工作在任意形状和大小的机器人上，但是较大的机器人将很难通过狭窄的空间。

2.软件

导航功能实现之前，需要搭建一些软件环境：

1. 毋庸置疑的，必须先要安装 ROS

2. 当前导航基于仿真环境，先保证上一章的机器人系统仿真可以正常执行

在仿真环境下，机器人可以正常接收 /cmd_vel 消息，并发布里程计消息，传感器消息发布也正常，也即导航模块中的运动控制和环境感知实现完毕

后续导航实现中，我们主要关注于：使用 SLAM 绘制地图、地图服务、自身定位与路径规划。

7.2 导航实现

本节内容主要介绍导航的完整性实现，旨在掌握机器人导航的基本流程，该章涉及的主要内容如下：

- SLAM建图(选用较为常见的gmapping)
- 地图服务(可以保存和重现地图)
- 机器人定位
- 路径规划
- 上述流程介绍完毕，还会对功能进一步集成实现探索式的SLAM建图。

准备工作

请先安装相关的ROS功能包：

- 安装 gmapping 包(用于构建地图): `sudo apt install ros-
<ROS版本>-gmapping`
- 安装地图服务包(用于保存与读取地图): `sudo apt install ros-
<ROS版本>-map-server`
- 安装 navigation 包(用于定位以及路径规划): `sudo apt install
ros-<ROS版本>-navigation`

新建功能包，并导入依赖: `gmapping map_server amcl move_base`

7.2.1 导航实现01_SLAM建图

SLAM算法有多种，当前我们选用gmapping，后续会再介绍其他几种常用的SLAM实现。

1.gmapping简介

gmapping是ROS开源社区中较为常用且比较成熟的SLAM算法之一，gmapping可以根据移动机器人里程计数据和激光雷达数据来绘制二维的栅格地图，对应的，gmapping对硬件也有一定的要求：

- 该移动机器人可以发布里程计消息
- 机器人需要发布雷达消息(该消息可以通过水平固定安装的雷达发布，或者也可以将深度相机消息转换成雷达消息)

关于里程计与雷达数据，仿真环境中可以正常获取的，不再赘述，栅格地图如案例所示。

gmapping安装前面也有介绍，命令如下：



```
1 sudo apt install ros-<ROS版本>-gmapping
```

2.gmapping节点说明

gmapping功能包中的核心节点是:slam_gmapping。为了方便调用，需要先了解该节点订阅的话题、发布的话题、服务以及相关参数。

2.1 订阅的Topic

tf (tf/tfMessage)

- 用于雷达、底盘与里程计之间的坐标变换消息。

scan(sensor_msgs/LaserScan)

- SLAM所需的雷达信息。

2.2发布的Topic

map_metadata(nav_msgs/MapMetaData)

- 地图元数据，包括地图的宽度、高度、分辨率等，该消息会固定更新。

map(nav_msgs/OccupancyGrid)

- 地图栅格数据，一般会在rviz中以图形化的方式显示。

~entropy(std_msgs/Float64)

- 机器人姿态分布熵估计(值越大，不确定性越大)。

2.3服务

dynamic_map(nav_msgs/GetMap)

- 用于获取地图数据。

2.4参数

~base_frame(string, default:"base_link")

- 机器人基坐标系。

~map_frame(string, default:"map")

- 地图坐标系。

~odom_frame(string, default:"odom")

- 里程计坐标系。

~map_update_interval(float, default: 5.0)

- 地图更新频率，根据指定的值设计更新间隔。

~maxUrange(float, default: 80.0)

- 激光探测的最大可用范围(超出此阈值，被截断)。

`~maxRange(float)`

- 激光探测的最大范围。

.... 参数较多，上述是几个较为常用的参数，其他参数介绍可参考官网。

2.5所需的坐标变换

雷达坐标系→基坐标系

- 一般由 `robot_state_publisher` 或 `static_transform_publisher` 发布。

基坐标系→里程计坐标系

- 一般由里程计节点发布。

2.6发布的坐标变换

地图坐标系→里程计坐标系

- 地图到里程计坐标系之间的变换。

3.gmapping使用

3.1编写gmapping节点相关launch文件

launch文件编写可以参考 github 的演示 launch文件：https://github.com/ros-perception/slam_gmapping/blob/melodic-devel/gmapping/launch/slam_gmapping_pr2.launch

复制并修改如下：

```

1 <launch>
2 <param name="use_sim_time" value="true"/>
3   <node pkg="gmapping" type="slam_gmapping"
4     name="slam_gmapping" output="screen">
5       <remap from="scan" to="scan"/>

```

```

5      <param name="base_frame"
6          value="base_footprint"/><!--底盘坐标系-->
7      <param name="odom_frame" value="odom"/> <!--
8          里程计坐标系-->
9      <param name="map_update_interval"
10         value="5.0"/>
11      <param name="maxUrangle" value="16.0"/>
12      <param name="sigma" value="0.05"/>
13      <param name="kernelSize" value="1"/>
14      <param name="lstep" value="0.05"/>
15      <param name="astep" value="0.05"/>
16      <param name="iterations" value="5"/>
17      <param name="lsigma" value="0.075"/>
18      <param name="ogain" value="3.0"/>
19      <param name="lskip" value="0"/>
20      <param name="srr" value="0.1"/>
21      <param name="srt" value="0.2"/>
22      <param name="str" value="0.1"/>
23      <param name="stt" value="0.2"/>
24      <param name="linearUpdate" value="1.0"/>
25      <param name="angularUpdate" value="0.5"/>
26      <param name="temporalUpdate" value="3.0"/>
27      <param name="resampleThreshold"
28          value="0.5"/>
29      <param name="particles" value="30"/>
30      <param name="xmin" value="-50.0"/>
31      <param name="ymin" value="-50.0"/>
32      <param name="xmax" value="50.0"/>
33      <param name="ymax" value="50.0"/>
34      <param name="delta" value="0.05"/>
35      <param name="llsamplerange" value="0.01"/>
36      <param name="llsamplestep" value="0.01"/>
37      <param name="lasamplerange" value="0.005"/>
38      <param name="lasamplestep" value="0.005"/>
39
40      </node>
41
42      <node pkg="joint_state_publisher"
43          name="joint_state_publisher"
44          type="joint_state_publisher" />

```

```

38      <node pkg="robot_state_publisher"
39          name="robot_state_publisher"
40          type="robot_state_publisher" />
41
42      <!-- 可以保存 rviz 配置并后期直接使用-->
43      <!--
44          <node pkg="rviz" type="rviz" name="rviz"
45          args="-d $(find my_nav_sum)/rviz/gmapping.rviz"/>
46      -->
47  </launch>

```

关键代码解释：

```

1 <remap from="scan" to="scan"/><!-- 雷达话题 -->
2 <param name="base_frame" value="base_footprint"/><!--
-底盘坐标系-->
3 <param name="odom_frame" value="odom"/> <!--里程计坐
标系-->

```

3.2执行

1.先启动 Gazebo 仿真环境(此过程略)

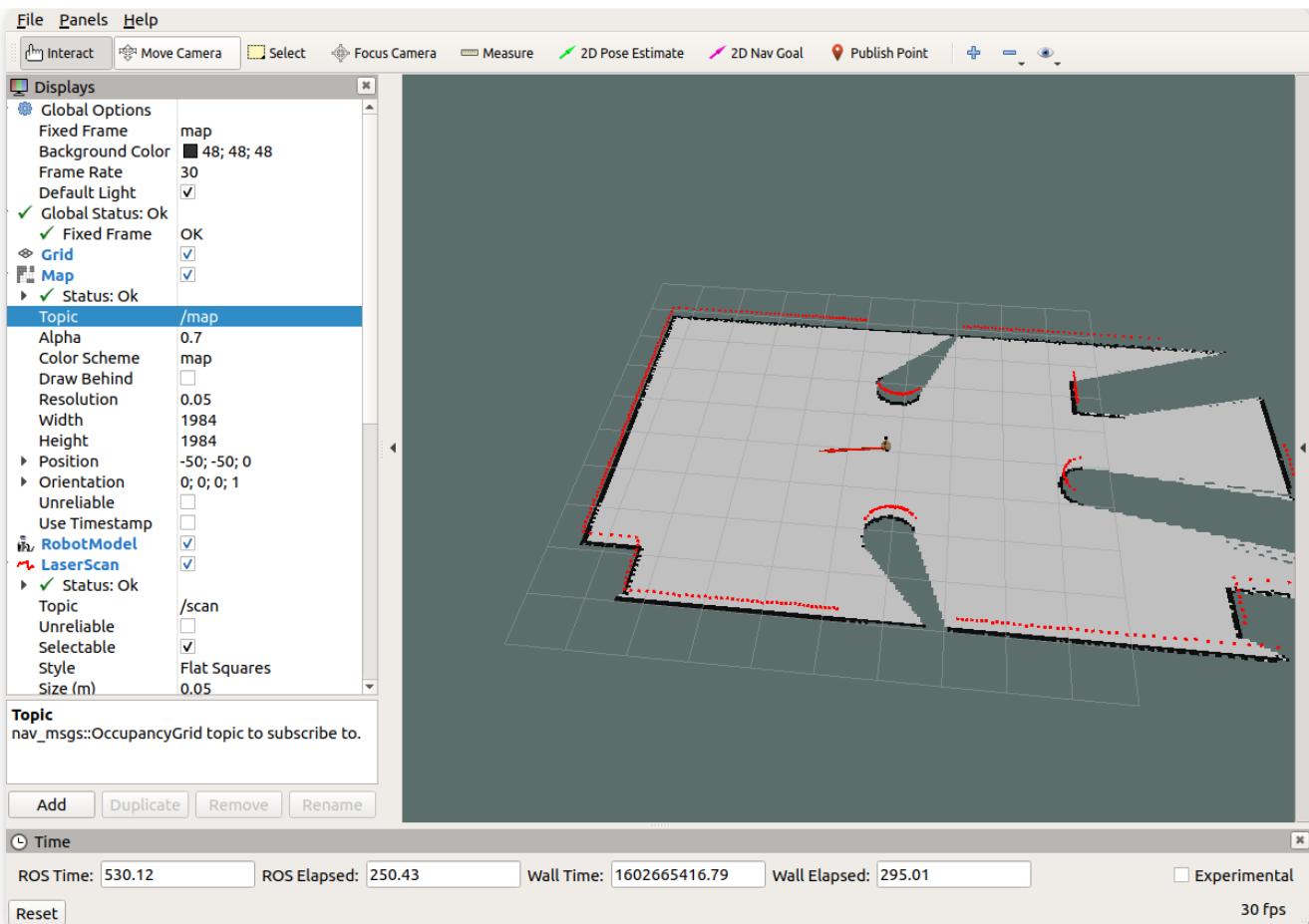
2.然后再启动地图绘制的 launch 文件:

```
1 roslaunch 包名 launch文件名
```

3.启动键盘控制节点，用于控制机器人运动建图

```
1 rosrun teleop_twist_keyboard
teleop_twist_keyboard.py
```

4.在 rviz 中添加组件，显示栅格地图



最后，就可以通过键盘控制gazebo中的机器人运动，同时，在rviz中可以显示gmapping发布的栅格地图数据了，下一步，还需要将地图单独保存。

另请参考：

- <http://wiki.ros.org/gmapping>

7.2.2 导航实现02_地图服务

上一节我们已经实现通过gmapping的构建地图并在rviz中显示了地图，不过，上一节中地图数据是保存在内存中的，当节点关闭时，数据也会被一并释放，我们需要将栅格地图序列化到的磁盘以持久化存储，后期还要通过反序列化读取磁盘的地图数据再执行后续操作。在ROS中，地图数据的序列化与反序列化可以通过 map_server 功能包实现。

1.map_server简介

map_server功能包中提供了两个节点: map_saver 和 map_server, 前者用于将栅格地图保存到磁盘, 后者读取磁盘的栅格地图并以服务的方式提供出去。

map_server安装前面也有介绍, 命令如下:



```
1 sudo apt install ros-<ROS版本>-map-server
```

2.map_server使用之地图保存节点(map_saver)

2.1map_saver节点说明

订阅的topic:

map(nav_msgs/OccupancyGrid)

- 订阅此话题用于生成地图文件。

2.2地图保存launch文件

地图保存的语法比较简单, 编写一个launch文件, 内容如下:

```
1 <launch>
2   <arg name="filename" value="$(find
3     mycar_nav)/map/nav" />
4   <node name="map_save" pkg="map_server"
5     type="map_saver" args="-f $(arg filename)" />
6 </launch>
```

其中 mymap 是指地图的保存路径以及保存的文件名称。

SLAM建图完毕后, 执行该launch文件即可。

测试:



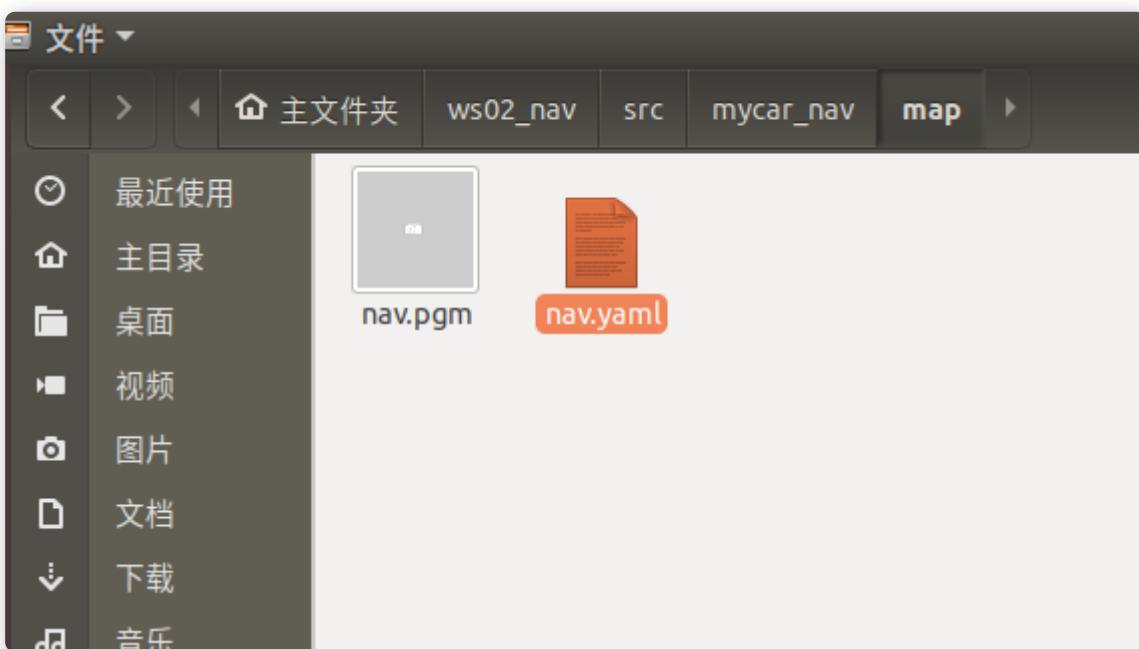
首先，参考上一节，依次启动仿真环境，键盘控制节点与SLAM节点；

然后，通过键盘控制机器人运动并绘图；

最后，通过上述地图保存方式保存地图。

结果：在指定路径下会生成两个文件，`xxx.pgm` 与 `xxx.yaml`

2.3 保存结果解释



`xxx.pgm` 本质是一张图片，直接使用图片查看程序即可打开。

`xxx.yaml` 保存的是地图的元数据信息，用于描述图片，内容格式如下：

```

1 image: /home/rosmelodic/ws02_nav/src/mycar_nav/map/nav.pgm
2 resolution: 0.050000
3 origin: [-50.000000, -50.000000, 0.000000]
4 negate: 0
5 occupied_thresh: 0.65
6 free_thresh: 0.196

```

解释：

- **image**: 被描述的图片资源路径, 可以是绝对路径也可以是相对路径。
- **resolution**: 图片分片率(单位: m/像素)。
- **origin**: 地图中左下像素的二维姿势, 为 (x, y, 偏航) , 偏航为逆时针旋转 (偏航= 0表示无旋转) 。
- **occupied_thresh**: 占用概率大于此阈值的像素被视为完全占用。
- **free_thresh**: 占用率小于此阈值的像素被视为完全空闲。
- **negate**: 是否应该颠倒白色/黑色自由/占用的语义。

`map_server` 中障碍物计算规则:

1. 地图中的每一个像素取值在 [0,255] 之间, 白色为 255, 黑色为 0, 该值设为 x ;
2. `map_server` 会将像素值作为判断是否是障碍物的依据, 首先计算比例: $p = (255 - x) / 255.0$, 白色为0, 黑色为1(`negate`为true, 则 $p = x / 255.0$);
3. 根据步骤2计算的比例判断是否是障碍物, 如果 $p > \text{occupied_thresh}$ 那么视为障碍物, 如果 $p < \text{free_thresh}$ 那么视为无物。

备注:

- 图片也可以根据需求编辑。

3.map_server使用之地图服务(`map_server`)

3.1map_server节点说明

发布的话题

`map_metadata` (`nav_msgs / MapMetaData`)

- 发布地图元数据。

`map` (`nav_msgs / OccupancyGrid`)

- 地图数据。

服务

`static_map` (`nav_msgs / GetMap`)

- 通过此服务获取地图。

参数

~frame_id (字符串, 默认值: “map”)

- 地图坐标系。

3.2地图读取

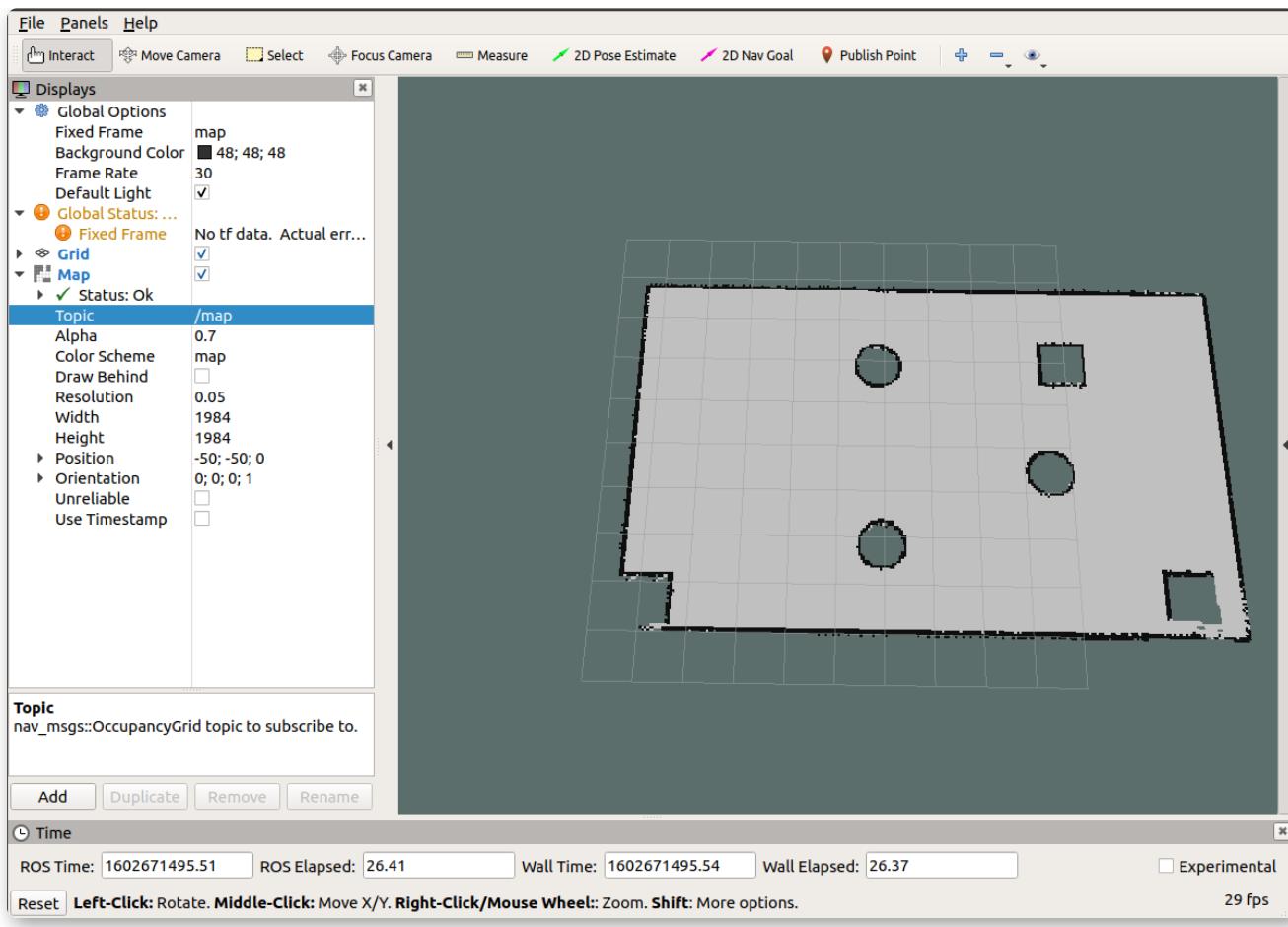
通过 map_server 的 map_server 节点可以读取栅格地图数据, 编写 launch 文件如下:

```
1 <launch>
2     <!-- 设置地图的配置文件 -->
3     <arg name="map" default="nav.yaml" />
4     <!-- 运行地图服务器, 并且加载设置的地图-->
5     <node name="map_server" pkg="map_server"
6         type="map_server" args="$(find mycar_nav)/map/${arg
map})"/>
7 </launch>
```

其中参数是地图描述文件的资源路径, 执行该launch文件, 该节点会发布话题:map(nav_msgs/OccupancyGrid)

3.3地图显示

在 rviz 中使用 map 组件可以显示栅格地图:



另请参考：

- http://wiki.ros.org/map_server

7.2.3 导航实现03_定位

所谓定位就是推算机器人自身在全局地图中的位置，当然，SLAM中也包含定位算法实现，不过SLAM的定位是用于构建全局地图的，是属于导航开始之前的阶段，而当前定位是用于导航中，导航中，机器人需要按照设定的路线运动，通过定位可以判断机器人的实际轨迹是否符合预期。在ROS的导航功能包集navigation中提供了 amcl 功能包，用于实现导航中的机器人定位。

1.amcl简介

AMCL(adaptive Monte Carlo Localization) 是用于2D移动机器人的概率定位系统，它实现了自适应（或KLD采样）蒙特卡洛定位方法，可以根据已有地图使用粒子滤波器推算机器人位置。

amcl已经被集成到了navigation包，navigation安装前面也有介绍，命令如下：



```
1 sudo apt install ros-<ROS版本>-navigation
```

2.amcl节点说明

amcl 功能包中的核心节点是:amcl。为了方便调用，需要先了解该节点订阅的话题、发布的话题、服务以及相关参数。

3.1订阅的Topic

scan(sensor_msgs/LaserScan)

- 激光雷达数据。

tf(tf/tfMessage)

- 坐标变换消息。

initialpose(geometry_msgs/PoseWithCovarianceStamped)

- 用来初始化粒子滤波器的均值和协方差。

map(nav_msgs/OccupancyGrid)

- 获取地图数据。

3.2发布的Topic

amcl_pose(geometry_msgs/PoseWithCovarianceStamped)

- 机器人在地图中的位姿估计。

particlecloud(geometry_msgs/PoseArray)

- 位姿估计集合，rviz中可以被 PoseArray 订阅然后图形化显示机器人的位姿估计集合。

`tf(tf/tfMessage)`

- 发布从 `odom` 到 `map` 的转换。

3.3服务

`global_localization(std_srvs/Empty)`

- 初始化全局定位的服务。

`request_nomotion_update(std_srvs/Empty)`

- 手动执行更新和发布更新的粒子的服务。

`set_map(nav_msgs/SetMap)`

- 手动设置新地图和姿态的服务。

3.4调用的服务

`static_map(nav_msgs/GetMap)`

- 调用此服务获取地图数据。

3.5参数

`~odom_model_type(string, default:"diff")`

- 里程计模型选择: "diff", "omni", "diff-corrected", "omni-corrected" (diff 差速、omni 全向轮)

`~odom_frame_id(string, default:"odom")`

- 里程计坐标系。

`~base_frame_id(string, default:"base_link")`

- 机器人极坐标系。

`~global_frame_id(string, default:"map")`

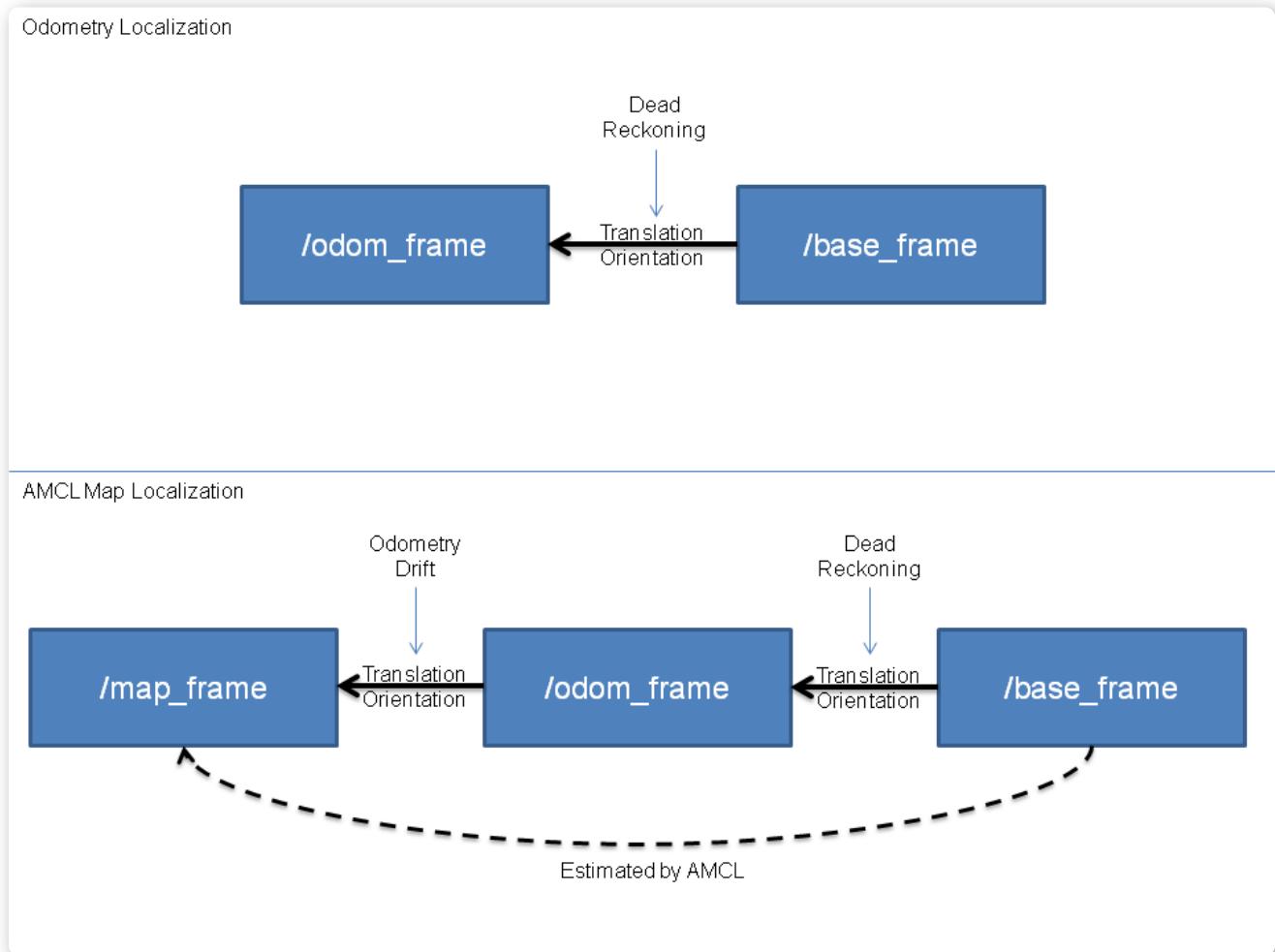
- 地图坐标系。

.... 参数较多，上述是几个较为常用的参数，其他参数介绍可参考官网。

3.6坐标变换

里程计本身也是可以协助机器人定位的，不过里程计存在累计误差且一些特殊情况时(车轮打滑)会出现定位错误的情况，amcl 则可以通过估算机器人在地图坐标系下的姿态，再结合里程计提高定位准确度。

- 里程计定位:只是通过里程计数据实现 /odom_frame 与 /base_frame 之间的坐标变换。
- amcl定位: 可以提供 /map_frame、/odom_frame 与 /base_frame 之间的坐标变换。



3.amcl使用

3.1编写amcl节点相关的launch文件

关于launch文件的实现，在amcl功能包下的example目录已经给出了示例，可以作为参考，具体实现：

```
1 roscd amcl
2 ls examples
```

该目录下会列出两个文件: amcl_diff.launch 和 amcl_omni.launch 文件, 前者适用于差分移动机器人, 后者适用于全向移动机器人, 可以按需选择, 此处参考前者, 新建 launch 文件, 复制 amcl_diff.launch 文件内容并修改如下:

```
1 <launch>
2 <node pkg="amcl" type="amcl" name="amcl"
  output="screen">
3   <!-- Publish scans from best pose at a max of 10
  Hz -->
4   <param name="odom_model_type" value="diff"/><!--
  里程计模式为差分 -->
5   <param name="odom_alpha5" value="0.1"/>
6   <param name="transform_tolerance" value="0.2" />
7   <param name="gui_publish_rate" value="10.0"/>
8   <param name="laser_max_beams" value="30"/>
9   <param name="min_particles" value="500"/>
10  <param name="max_particles" value="5000"/>
11  <param name="kld_err" value="0.05"/>
12  <param name="kld_z" value="0.99"/>
13  <param name="odom_alpha1" value="0.2"/>
14  <param name="odom_alpha2" value="0.2"/>
15  <!-- translation std dev, m -->
16  <param name="odom_alpha3" value="0.8"/>
17  <param name="odom_alpha4" value="0.2"/>
18  <param name="laser_z_hit" value="0.5"/>
19  <param name="laser_z_short" value="0.05"/>
20  <param name="laser_z_max" value="0.05"/>
21  <param name="laser_z_rand" value="0.5"/>
22  <param name="laser_sigma_hit" value="0.2"/>
23  <param name="laser_lambda_short" value="0.1"/>
24  <param name="laser_lambda_short" value="0.1"/>
25  <param name="laser_model_type"
  value="likelihood_field"/>
```

```

26    <!-- <param name="laser_model_type"
27      value="beam"/> -->
28    <param name="laser_likelihood_max_dist"
29      value="2.0"/>
30
31    <param name="odom_frame_id" value="odom"/><!-- 里
程计坐标系 -->
32    <param name="base_frame_id"
33      value="base_footprint"/><!-- 添加机器人基坐标系 -->
34    <param name="global_frame_id" value="map"/><!--
添加地图坐标系 -->
35    <param name="resample_interval" value="1"/>
36    <param name="transform_tolerance" value="0.1"/>
37    <param name="recovery_alpha_slow" value="0.0"/>
38    <param name="recovery_alpha_fast" value="0.0"/>
39  </node>
40 </launch>

```

3.2 编写测试launch文件

amcl节点是不可以单独运行的，运行 amcl 节点之前，需要先加载全局地图，然后启动 rviz 显示定位结果，上述节点可以集成进launch文件，内容示例如下：

```

1 <launch>
2     <!-- 设置地图的配置文件 -->
3     <arg name="map" default="nav.yaml" />
4     <!-- 运行地图服务器，并且加载设置的地图-->
5     <node name="map_server" pkg="map_server"
6         type="map_server" args="$(find
7             mycar_nav)/map/${arg map}" />
8     <!-- 启动AMCL节点 -->
9     <include file="$(find
10        mycar_nav)/launch/amcl.launch" />
11     <!-- 运行rviz -->
12     <node pkg="rviz" type="rviz" name="rviz"/>
13 </launch>

```

当然，launch文件中地图服务节点和amcl节点中的包名、文件名需要根据自己的设置修改。

3.3执行

1.先启动 Gazebo 仿真环境(此过程略)；

2.启动键盘控制节点：

```

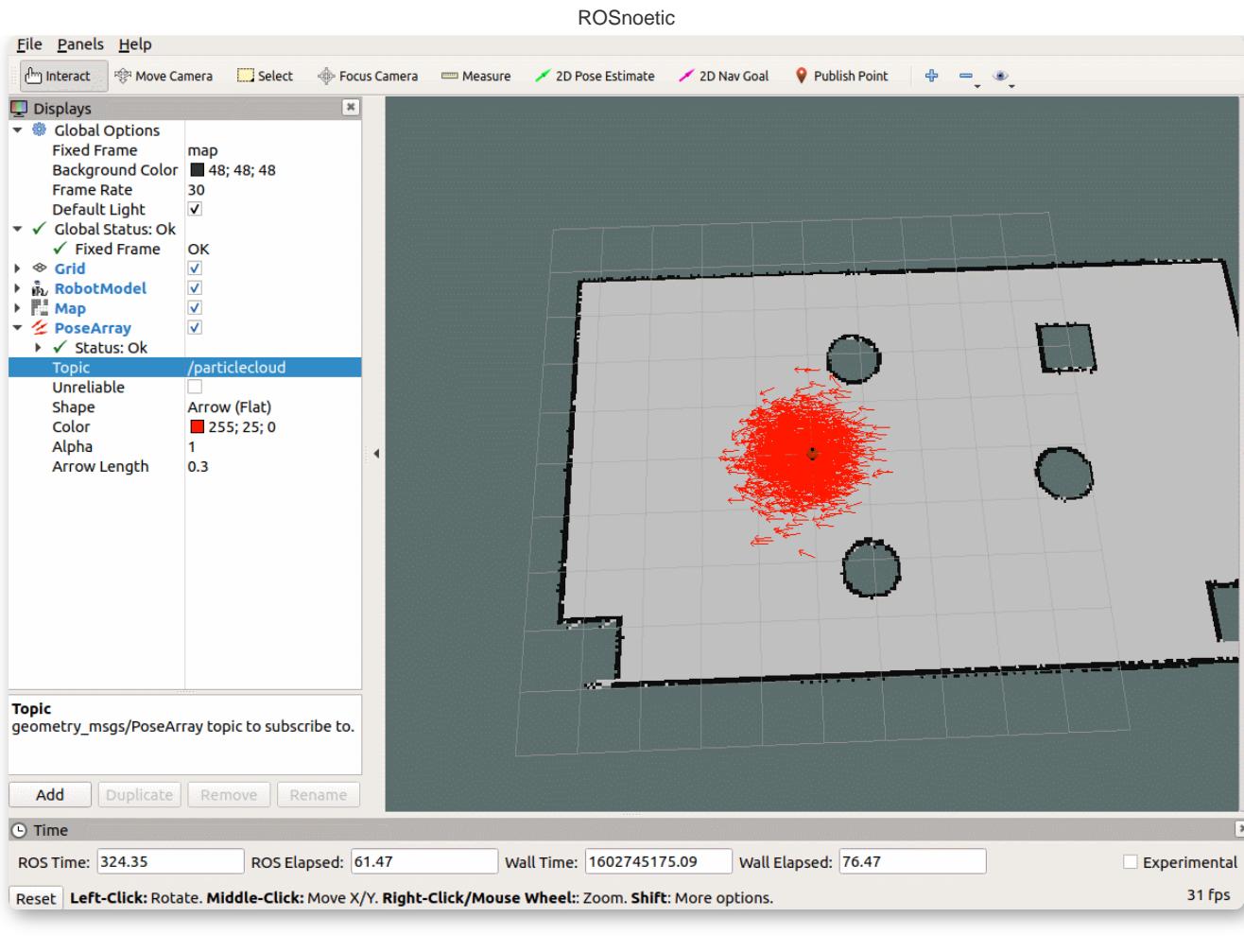
1 rosrun teleop_twist_keyboard
  teleop_twist_keyboard.py

```

3.启动上一步中集成地图服务、amcl 与 rviz 的 launch 文件；

4.在启动的 rviz 中，添加RobotModel、Map组件，分别显示机器人模型与地图，添加 posearray 插件，设置topic为particlecloud来显示 amcl 预估的当前机器人的位姿，箭头越是密集，说明当前机器人处于此位置的概率越高；

5.通过键盘控制机器人运动，会发现 posearray 也随之而改变。



另请参考：

- <http://wiki.ros.org/amcl>

7.2.4 导航实现04_路径规划

毋庸置疑的，路径规划是导航中的核心功能之一，在ROS的导航功能包集navigation中提供了move_base功能包，用于实现此功能。

1. move_base简介

move_base功能包提供了基于动作(action)的路径规划实现，move_base可以根据给定的目标点，控制机器人底盘运动至目标位置，并且在运动过程中会连续反馈机器人自身的姿态与目标点的状态信息。如前所述(7.1)move_base主要由全局路径规划与本地路径规划组成。

move_base已经被集成到了navigation包，navigation安装前面也有介绍，命令如下：



```
1 sudo apt install ros-<ROS版本>-navigation
```

2.move_base节点说明

move_base功能包中的核心节点是:move_base。为了方便调用，需要先了解该节点action、订阅的话题、发布的话题、服务以及相关参数。

2.1动作

动作订阅

move_base/goal(move_base_msgs/MoveBaseActionGoal)

- move_base 的运动规划目标。

move_base/cancel(actionlib_msgs/GoalID)

- 取消目标。

动作发布

move_base/feedback(move_base_msgs/MoveBaseActionFeedback)

- 连续反馈的信息，包含机器人底盘坐标。

move_base/status(actionlib_msgs/GoalStatusArray)

- 发送到move_base的目标状态信息。

move_base/result(move_base_msgs/MoveBaseActionResult)

- 操作结果(此处为空)。

2.2订阅的Topic

move_base_simple/goal(geometry_msgs/PoseStamped)

- 运动规划目标(与action相比，没有连续反馈，无法追踪机器人执行状态)。

2.3发布的Topic

cmd_vel(geometry_msgs/Twist)

- 输出到机器人底盘的运动控制消息。

2.4服务

~make_plan(nav_msgs/GetPlan)

- 请求该服务，可以获取给定目标的规划路径，但是并不执行该路径规划。

~clear_unknown_space(std_srvs/Empty)

- 允许用户直接清除机器人周围的未知空间。

~clear_costmaps(std_srvs/Empty)

- 允许清除代价地图中的障碍物，可能会导致机器人与障碍物碰撞，请慎用。

2.5参数

请参考官网。

3.move_base与代价地图

3.1概念

机器人导航(尤其是路径规划模块)是依赖于地图的，地图在SLAM时已经有所介绍了，ROS中的地图其实就是一张图片，这张图片有宽度、高度、分辨率等元数据，在图片中使用灰度值来表示障碍物存在的概率。不过SLAM构建的地图在导航中是不可以直接使用的，因为：

1. SLAM构建的地图是静态地图，而导航过程中，障碍物信息是可变的，可能障碍物被移走了，也可能添加了新的障碍物，导航中需要时时的获取障碍物信息；
2. 在靠近障碍物边缘时，虽然此处是空闲区域，但是机器人在进入该区域后可能由于其他一些因素，比如：惯性、或者不规则形体的机器人

转弯时可能会与障碍物产生碰撞，安全起见，最好在地图的障碍物边缘设置警戒区，尽量禁止机器人进入...

所以，静态地图无法直接应用于导航，其基础之上需要添加一些辅助信息的地图，比如时时获取的障碍物数据，基于静态地图添加的膨胀区等数据。

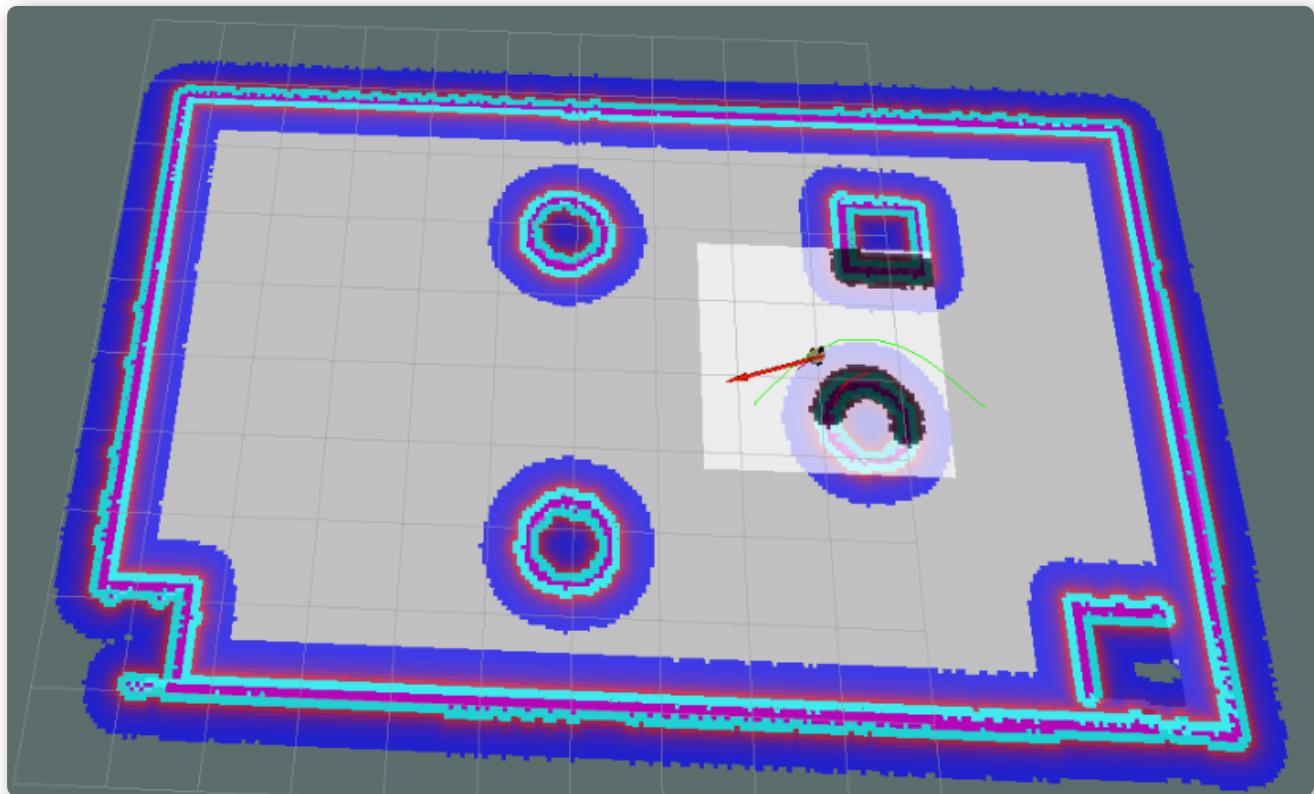
3.2组成

代价地图有两张:global_costmap(全局代价地图) 和 local_costmap(本地代价地图)，前者用于全局路径规划，后者用于本地路径规划。

两张代价地图都可以多层叠加,一般有以下层级:

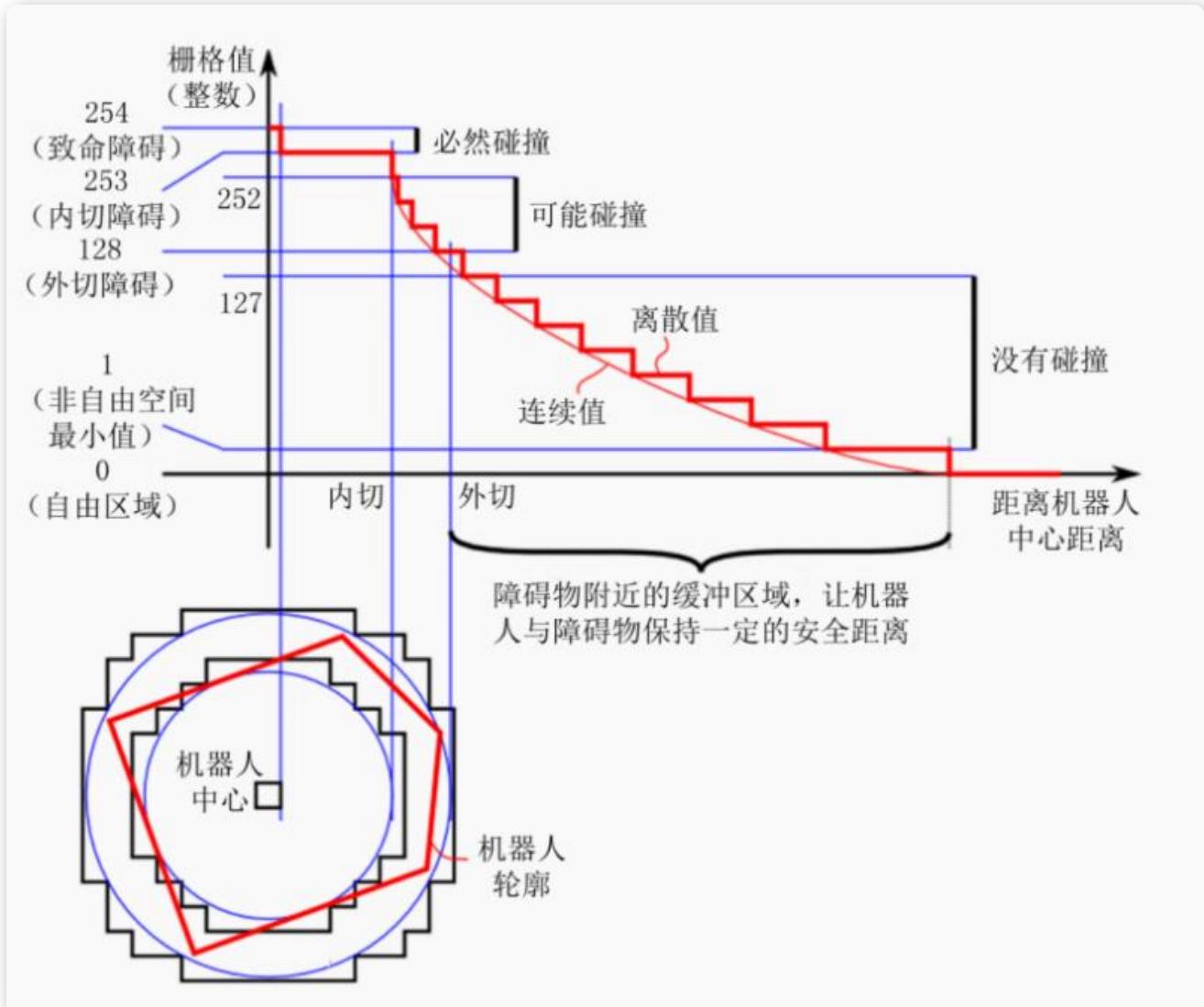
- **Static Map Layer:** 静态地图层，SLAM构建的静态地图。
- **Obstacle Map Layer:** 障碍地图层，传感器感知的障碍物信息。
- **Inflation Layer:** 膨胀层，在以上两层地图上进行膨胀（向外扩张），以避免机器人的外壳会撞上障碍物。
- **Other Layers:** 自定义costmap。

多个layer可以按需自由搭配。



3.3碰撞算法

在ROS中，如何计算代价值呢？请看下图：



上图中，横轴是距离机器人中心的距离，纵轴是代价地图中栅格的灰度值。

- 致命障碍:栅格值为254，此时障碍物与机器人中心重叠，必然发生碰撞；
- 内切障碍:栅格值为253，此时障碍物处于机器人的内切圆内，必然发生碰撞；
- 外切障碍:栅格值为[128,252]，此时障碍物处于其机器人的外切圆内，处于碰撞临界，不一定发生碰撞；
- 非自由空间:栅格值为(0,127]，此时机器人处于障碍物附近，属于危险警戒区，进入此区域，将来可能会发生碰撞；
- 自由区域:栅格值为0，此处机器人可以自由通过；
- 未知区域:栅格值为255，还没探明是否有障碍物。

膨胀空间的设置可以参考非自由空间。

4.move_base使用

路径规划算法在move_base功能包的move_base节点中已经封装完毕了，但是还不可以直接调用，因为算法虽然已经封装了，但是该功能包面向的是各种类型支持ROS的机器人，不同类型机器人可能大小尺寸不同，传感器不同，速度不同，应用场景不同....最后可能会导致不同的路径规划结果，那么在调用路径规划节点之前，我们还需要配置机器人参数。具体实现如下：

1. 先编写launch文件模板
2. 编写配置文件
3. 集成导航相关的launch文件
4. 测试

4.1 launch文件

关于move_base节点的调用，模板如下：

```

1 <launch>
2
3     <node pkg="move_base" type="move_base"
4       respawn="false" name="move_base" output="screen"
5       clear_params="true">
6         <rosparam file="$(find 功能
包)/param/costmap_common_params.yaml"
7           command="load" ns="global_costmap" />
8         <rosparam file="$(find 功能
包)/param/costmap_common_params.yaml"
9           command="load" ns="local_costmap" />
10        <rosparam file="$(find 功能
包)/param/local_costmap_params.yaml" command="load"
11          />
12        <rosparam file="$(find 功能
包)/param/global_costmap_params.yaml"
13          command="load" />
14        <rosparam file="$(find 功能
包)/param/base_local_planner_params.yaml"
15          command="load" />
16      </node>
17
18  </launch>

```

launch文件解释:

启动了 move_base 功能包下的 move_base 节点， respawn 为 false，意味着该节点关闭后，不会被重启； clear_params 为 true，意味着每次启动该节点都要清空私有参数然后重新载入；通过 rosparam 会载入若干 yaml 文件用于配置参数，这些yaml文件的配置以及作用详见下一小节内容。

4.2配置文件

关于配置文件的编写，可以参考一些成熟的机器人的路径规划实现，比如: turtlebot3，github链接：https://github.com/ROBOTIS-GIT/turtlebot3/tree/master/turtlebot3_navigation/param，先下载这些配置文件备用。

在功能包下新建 param 目录，复制下载的文件到此目录：

costmap_common_params_burger.yaml、
local_costmap_params.yaml、global_costmap_params.yaml、
base_local_planner_params.yaml，并将
costmap_common_params_burger.yaml 重命名
为:costmap_common_params.yaml。

配置文件修改以及解释：

4.2.1 costmap_common_params.yaml

该文件是move_base 在全局路径规划与本地路径规划时调用的通用参数，包括:机器人的尺寸、距离障碍物的安全距离、传感器信息等。配置参考如下：

```

1 #机器人几何参, 如果机器人是圆形, 设置 robot_radius, 如果是
2 #其他形状设置 footprint
3 robot_radius: 0.12 #圆形
4 # footprint: [[-0.12, -0.12], [-0.12, 0.12],
5 # [0.12, 0.12], [0.12, -0.12]] #其他形状
6
7
8
9 #膨胀半径, 扩展在碰撞区域以外的代价区域, 使得机器人规划路径
10 #避开障碍物
11 inflation_radius: 0.2
12 #代价比例系数, 越大则代价值越小
13 cost_scaling_factor: 3.0
14
15 #地图类型
16 map_type: costmap
17 #导航包所需要的传感器
18 observation_sources: scan

```

```

18 #对传感器的坐标系和数据进行配置。这个也会用于代价地图添加和
  清除障碍物。例如，你可以用激光雷达传感器用于在代价地图添加障
  碍物，再添加kinect用于导航和清除障碍物。
19 scan: {sensor_frame: laser, data_type: LaserScan,
  topic: scan, marking: true, clearing: true}

```

4.2.2global_costmap_params.yaml

该文件用于全局代价地图参数设置：

```

1 global_costmap:
2   global_frame: map #地图坐标系
3   robot_base_frame: base_footprint #机器人坐标系
4   # 以此实现坐标变换
5
6   update_frequency: 1.0 #代价地图更新频率
7   publish_frequency: 1.0 #代价地图的发布频率
8   transform_tolerance: 0.5 #等待坐标变换发布信息的超时
  时间
9
10  static_map: true # 是否使用一个地图或者地图服务器来初
    始化全局代价地图，如果不使用静态地图，这个参数为false.

```

4.2.3local_costmap_params.yaml

该文件用于局部代价地图参数设置：

```

1 local_costmap:
2   global_frame: odom #里程计坐标系
3   robot_base_frame: base_footprint #机器人坐标系
4
5   update_frequency: 10.0 #代价地图更新频率
6   publish_frequency: 10.0 #代价地图的发布频率
7   transform_tolerance: 0.5 #等待坐标变换发布信息的超时
8   time
9   static_map: false #不需要静态地图, 可以提升导航效果
10  rolling_window: true #是否使用动态窗口, 默认为false,
11  在静态的全局地图中, 地图不会变化
12  width: 3 # 局部地图宽度 单位是 m
13  height: 3 # 局部地图高度 单位是 m
14  resolution: 0.05 # 局部地图分辨率 单位是 m, 一般与静态
15  地图分辨率保持一致

```

4.2.4 base_local_planner_params

基本的局部规划器参数配置, 这个配置文件设定了机器人的最大和最小速度限制值, 也设定了加速度的阈值。

```

1 TrajectoryPlannerROS:
2
3 # Robot Configuration Parameters
4   max_vel_x: 0.5 # X 方向最大速度
5   min_vel_x: 0.1 # X 方向最小速度
6
7   max_vel_theta: 1.0 #
8   min_vel_theta: -1.0
9   min_in_place_vel_theta: 1.0
10
11  acc_lim_x: 1.0 # X 加速限制
12  acc_lim_y: 0.0 # Y 加速限制
13  acc_lim_theta: 0.6 # 角速度加速限制
14
15 # Goal Tolerance Parameters, 目标公差
16  xy_goal_tolerance: 0.10

```

```

17     yaw_goal_tolerance: 0.05
18
19 # Differential-drive robot configuration
20 # 是否是全向移动机器人
21     holonomic_robot: false
22
23 # Forward Simulation Parameters, 前进模拟参数
24     sim_time: 0.8
25     vx_samples: 18
26     vtheta_samples: 20
27     sim_granularity: 0.05

```

4.2.5参数配置技巧

以上配置在实操中，可能会出现机器人在本地路径规划时与全局路径规划不符而进入膨胀区域出现假死的情况，如何尽量避免这种情形呢？



全局路径规划与本地路径规划虽然设置的参数是一样的，但是二者路径规划和避障的职能不同，可以采用不同的参数设置策略：

- 全局代价地图可以将膨胀半径和障碍物系数设置的偏大一些；
- 本地代价地图可以将膨胀半径和障碍物系数设置的偏小一些。

这样，在全局路径规划时，规划的路径会尽量远离障碍物，而本地路径规划时，机器人即便偏离全局路径也会和障碍物之间保留更大的自由空间，从而避免了陷入“假死”的情形。

4.3launch文件集成

如果要实现导航，需要集成地图服务、amcl、move_base与Rviz等，集成示例如下：

```

1 <launch>
2     <!-- 设置地图的配置文件 -->
3     <arg name="map" default="nav.yaml" />
4     <!-- 运行地图服务器，并且加载设置的地图-->

```

```
5      <node name="map_server" pkg="map_server"
6          type="map_server" args="$(find
7              mycar_nav)/map/$(arg map)"/>
8          <!-- 启动AMCL节点 -->
9          <include file="$(find
10             mycar_nav)/launch/amcl.launch" />
11         <!-- 运行move_base节点 -->
12         <include file="$(find
13             mycar_nav)/launch/path.launch" />
14         <!-- 运行rviz -->
15         <node pkg="rviz" type="rviz" name="rviz"
16             args="-d $(find mycar_nav)/rviz/nav.rviz" />
17
18     </launch>
```

4.4 测试

1. 先启动 Gazebo 仿真环境(此过程略);
2. 启动导航相关的 launch 文件;
3. 添加Rviz组件(参考演示结果),可以将配置数据保存, 后期直接调用;

全局代价地图与本地代价地图组件配置如下:

Global Options

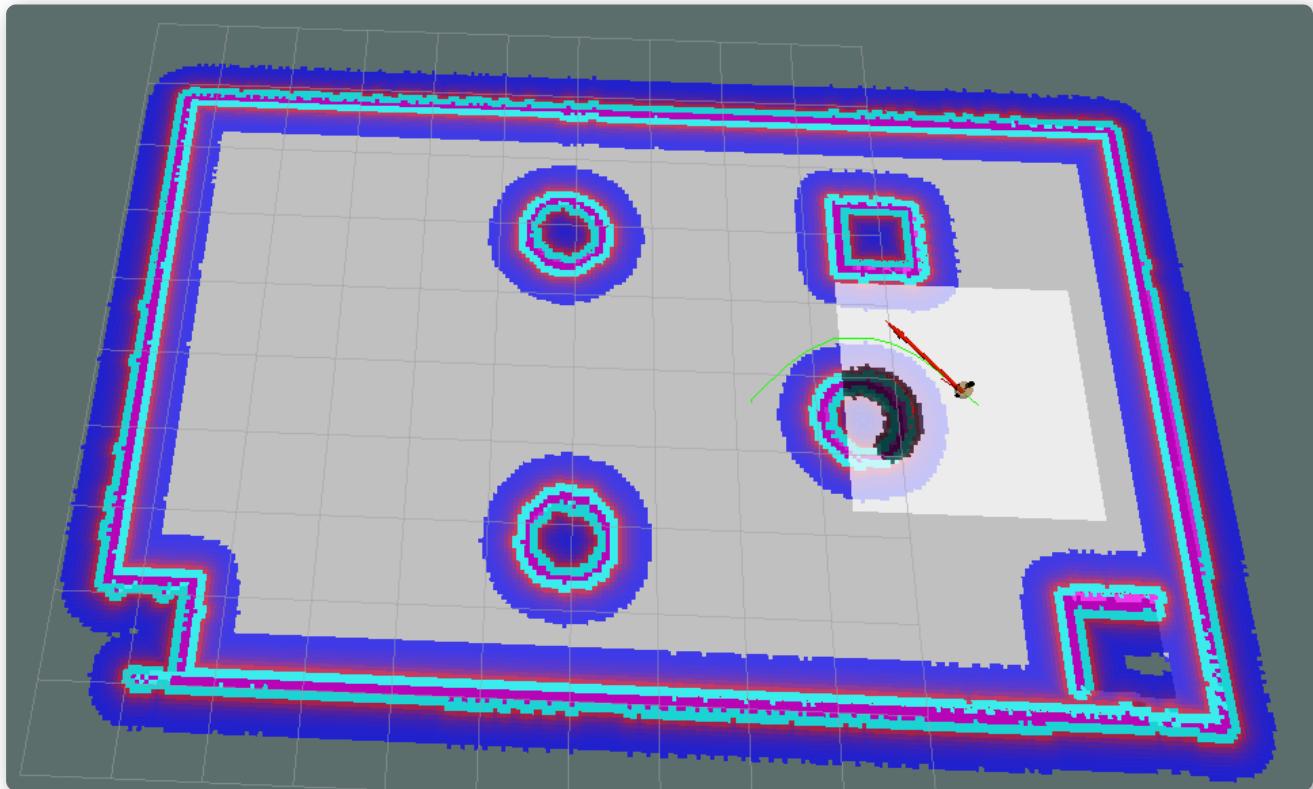
- ✓ Global Status: Ok
- ❖ Grid
- ❖ Map
- ❖ RobotModel
- ❖ Odometry
- ❖ LaserScan
- ❖ PoseArray
- ❖ Map
- ▶ ✓ Status: Ok
 - Topic /move_base/global_costmap/costmap
 - Alpha 0.7
 - Color Scheme costmap
 - Draw Behind
 - Resolution 0.05
 - Width 1984
 - Height 1984
- ▶ Position
- ▶ Orientation
- Unreliable
- Use Timestamp
- ❖ Map
- ▶ ✓ Status: Ok
 - Topic /move_base/local_costmap/costmap
 - Alpha 0.7
 - Color Scheme map
 - Draw Behind
 - Resolution 0.05
 - Width 60
 - Height 60
- ▶ Position
- ▶ Orientation
- Unreliable
- Use Timestamp

全局路径规划与本地路径规划组件配置如下：

▶ Global Options

- ✓ Global Status: Ok
- ❖ Grid
- ❖ Map
- ❖ RobotModel
- ❖ Odometry
- ❖ LaserScan
- ❖ PoseArray
- ❖ Map
- ❖ Map
- ▶ Path
 - ✓ Status: Ok
 - Topic /move_base/TrajectoryPlannerROS/global_plan
 - Unreliable
 - Line Style Lines
 - Color 25; 255; 0
 - Alpha 1
 - Buffer Length 1
 - Offset
 - Pose Style None
- ▶ Path
 - ✓ Status: Ok
 - Topic /move_base/TrajectoryPlannerROS/local_plan
 - Unreliable
 - Line Style Lines
 - Color 164; 0; 0
 - Alpha 1
 - Buffer Length 1
 - Offset
 - Pose Style None

4.通过Rviz工具栏的 2D Nav Goal设置目的地实现导航。



5.也可以在导航过程中，添加新的障碍物，机器人也可以自动躲避障碍物。

另请参考：

- http://wiki.ros.org/move_base

7.2.5 导航与SLAM建图



场景:在 7.2.1 导航实现01_SLAM建图中，我们是通过键盘控制机器人移动实现建图的，而后续又介绍了机器人的自主移动实现，那么可不可以将二者结合，实现机器人自主移动的SLAM建图呢？

上述需求是可行的。虽然可能会有疑问，导航时需要地图信息，之前导航实现时，是通过 map_server 包的 map_server 节点来发布地图信息的，如果不先通过SLAM建图，那么如何发布地图信息呢？SLAM建图过程中本身就会时时发布地图信息，所以无需再使用map_server，SLAM已经发布了话题为 /map 的地图消息了，且导航需要定位模块，SLAM本身也是可以实现定位的。

该过程实现比较简单，步骤如下：

1. 编写launch文件，集成SLAM与move_base相关节点；
2. 执行launch文件并测试。

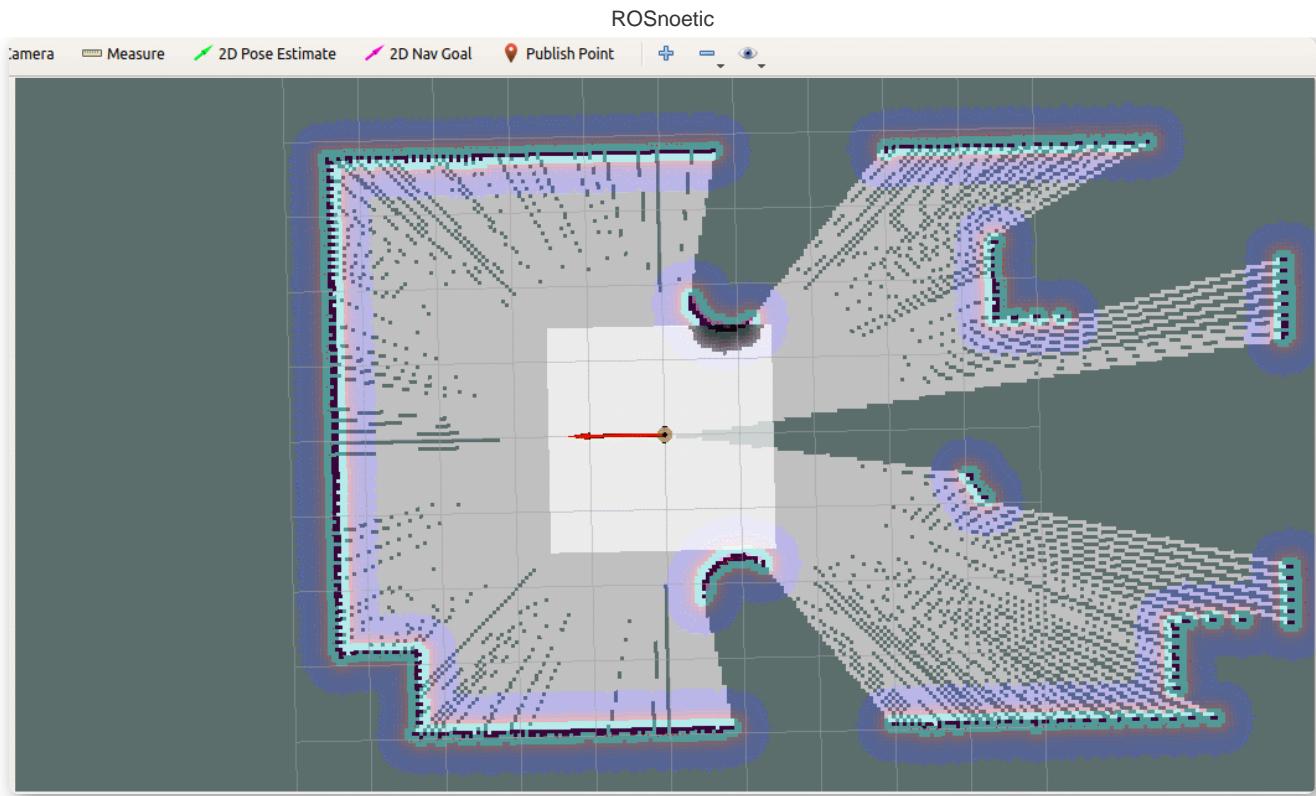
1. 编写launc文件

当前launch文件实现，无需调用map_server的相关节点，只需要启动SLAM节点与move_base节点，示例内容如下：

```
1 <launch>
2   <!-- 启动SLAM节点 -->
3   <include file="$(find
4     mycar_nav)/launch/slam.launch" />
5   <!-- 运行move_base节点 -->
6   <include file="$(find
7     mycar_nav)/launch/path.launch" />
8   <!-- 运行rviz -->
9   <node pkg="rviz" type="rviz" name="rviz" args="-
d $(find mycar_nav)/rviz/nav.rviz" />
10 </launch>
```

2. 测试

1. 首先运行gazebo仿真环境；
2. 然后执行launch文件；
3. 在rviz中通过2D Nav Goal设置目标点，机器人开始自主移动并建图了；
4. 最后可以使用 map_server 保存地图。



7.3 导航相关消息

在导航功能包集中包含了诸多节点，毋庸置疑的，不同节点之间的通信使用到了消息中间件(数据载体)，在上一节的实现中，这些消息已经在rviz中做了可视化处理，比如:地图、雷达、摄像头、里程计、路径规划...的相关消息在rviz中提供了相关组件，本节主要介绍这些消息的具体格式。

7.3.1 导航之地图

地图相关的消息主要有两个：

`nav_msgs/MapMetaData`

- 地图元数据，包括地图的宽度、高度、分辨率等。

`nav_msgs/OccupancyGrid`

- 地图栅格数据，一般会在rviz中以图形化的方式显示。

1.nav_msgs/MapMetaData

调用 `rosmsg info nav_msgs/MapMetaData` 显示消息内容如下:

```

1 time map_load_time
2 float32 resolution #地图分辨率
3 uint32 width #地图宽度
4 uint32 height #地图高度
5 geometry_msgs/Pose origin #地图位姿数据
6   geometry_msgs/Point position
7     float64 x
8     float64 y
9     float64 z
10    geometry_msgs/Quaternion orientation
11      float64 x
12      float64 y
13      float64 z
14      float64 w

```

2.nav_msgs/OccupancyGrid

调用 `rosmsg info nav_msgs/OccupancyGrid` 显示消息内容如下:

```

1 std_msgs/Header header
2 uint32 seq
3 time stamp
4 string frame_id
5 #--- 地图元数据
6 nav_msgs/MapMetaData info
7   time map_load_time
8   float32 resolution
9   uint32 width
10  uint32 height
11  geometry_msgs/Pose origin
12    geometry_msgs/Point position
13      float64 x
14      float64 y
15      float64 z

```

```

16     geometry_msgs/Quaternion orientation
17         float64 x
18         float64 y
19         float64 z
20         float64 w
21 #--- 地图内容数据, 数组长度 = width * height
22 int8[] data

```

7.3.2 导航之里程计

里程计相关消息是:nav_msgs/Odometry, 调用 `rosmsg info nav_msgs/Odometry` 显示消息内容如下:

```

1 std_msgs/Header header
2     uint32 seq
3     time stamp
4     string frame_id
5     string child_frame_id
6     geometry_msgs/PoseWithCovariance pose
7         geometry_msgs/Pose pose #里程计位姿
8             geometry_msgs/Point position
9                 float64 x
10                float64 y
11                float64 z
12                geometry_msgs/Quaternion orientation
13                    float64 x
14                    float64 y
15                    float64 z
16                    float64 w
17                    float64[36] covariance
18     geometry_msgs/TwistWithCovariance twist
19         geometry_msgs/Twist twist #速度
20             geometry_msgs/Vector3 linear
21                 float64 x
22                 float64 y
23                 float64 z
24             geometry_msgs/Vector3 angular
25                 float64 x

```

```

26     float64 y
27     float64 z
28 # 协方差矩阵
29 float64[36] covariance

```

7.3.3 导航之坐标变换

坐标变换相关消息是: `tf/tfMessage`, 调用 `rosmsg info tf/tfMessage` 显示消息内容如下:

```

1 geometry_msgs/TransformStamped[] transforms #包含了
    多个坐标系相对关系数据的数组
2     std_msgs/Header header
3     uint32 seq
4     time stamp
5     string frame_id
6     string child_frame_id
7     geometry_msgs/Transform transform
8         geometry_msgs/Vector3 translation
9             float64 x
10            float64 y
11            float64 z
12         geometry_msgs/Quaternion rotation
13             float64 x
14             float64 y
15             float64 z
16             float64 w

```

7.3.4 导航之定位

定位相关消息是: `geometry_msgs/PoseArray`, 调用 `rosmsg info geometry_msgs/PoseArray` 显示消息内容如下:

```

1 std_msgs/Header header
2     uint32 seq
3     time stamp
4     string frame_id

```

```

5 geometry_msgs/Pose[] poses #预估的点位姿组成的数组
6   geometry_msgs/Point position
7     float64 x
8     float64 y
9     float64 z
10    geometry_msgs/Quaternion orientation
11      float64 x
12      float64 y
13      float64 z
14      float64 w

```

7.3.5 导航之目标点与路径规划

目标点相关消息是:move_base_msgs/MoveBaseActionGoal, 调用
`rosmg info move_base_msgs/MoveBaseActionGoal` 显示消息内容
 如下:

```

1 std_msgs/Header header
2   uint32 seq
3   time stamp
4   string frame_id
5 actionlib_msgs/GoalID goal_id
6   time stamp
7   string id
8 move_base_msgs/MoveBaseGoal goal
9   geometry_msgs/PoseStamped target_pose
10  std_msgs/Header header
11    uint32 seq
12    time stamp
13    string frame_id
14    geometry_msgs/Pose pose #目标点位姿
15      geometry_msgs/Point position
16        float64 x
17        float64 y
18        float64 z
19        geometry_msgs/Quaternion orientation
20          float64 x
21          float64 y

```

```

22           float64 z
23           float64 w

```

路径规划相关消息是:nav_msgs/Path，调用 `rosmsg info nav_msgs/Path` 显示消息内容如下:

```

1 std_msgs/Header header
2   uint32 seq
3   time stamp
4   string frame_id
5 geometry_msgs/PoseStamped[] poses #由一系列点组成的数组
6   std_msgs/Header header
7     uint32 seq
8     time stamp
9     string frame_id
10    geometry_msgs/Pose pose
11      geometry_msgs/Point position
12        float64 x
13        float64 y
14        float64 z
15      geometry_msgs/Quaternion orientation
16        float64 x
17        float64 y
18        float64 z
19        float64 w

```

7.3.6 导航之激光雷达

激光雷达相关消息是:sensor_msgs/LaserScan，调用 `rosmsg info sensor_msgs/LaserScan` 显示消息内容如下:

```

1 std_msgs/Header header
2   uint32 seq
3   time stamp
4   string frame_id
5   float32 angle_min #起始扫描角度(rad)
6   float32 angle_max #终止扫描角度(rad)
7   float32 angle_increment #测量值之间的角距离(rad)
8   float32 time_increment #测量间隔时间(s)
9   float32 scan_time #扫描间隔时间(s)
10  float32 range_min #最小有效距离值(m)
11  float32 range_max #最大有效距离值(m)
12  float32[] ranges #一个周期的扫描数据
13  float32[] intensities #扫描强度数据, 如果设备不支持强度
    数据, 该数组为空

```

7.3.7 导航之相机

深度相机相关消息有:sensor_msgs/Image、
sensor_msgs/CompressedImage、sensor_msgs/PointCloud2

sensor_msgs/Image 对应的一般的图像数据，
sensor_msgs/CompressedImage 对应压缩后的图像数据，
sensor_msgs/PointCloud2 对应的是点云数据(带有深度信息的图像数据)。

调用 `rosmmsg info sensor_msgs/Image` 显示消息内容如下：

```

1 std_msgs/Header header
2   uint32 seq
3   time stamp
4   string frame_id
5   uint32 height #高度
6   uint32 width #宽度
7   string encoding #编码格式:RGB、YUV等
8   uint8 is_bigendian #图像大小端存储模式
9   uint32 step #一行图像数据的字节数, 作为步进参数
10  uint8[] data #图像数据, 长度等于 step * height

```

调用 `rosmsg info sensor_msgs/CompressedImage` 显示消息内容如下：

```

1 std_msgs/Header header
2   uint32 seq
3   time stamp
4   string frame_id
5   string format #压缩编码格式(jpeg、png、bmp)
6   uint8[] data #压缩后的数据

```

调用 `rosmsg info sensor_msgs/PointCloud2` 显示消息内容如下：

```

1 std_msgs/Header header
2   uint32 seq
3   time stamp
4   string frame_id
5   uint32 height #高度
6   uint32 width #宽度
7   sensor_msgs/PointField[] fields #每个点的数据类型
8     uint8 INT8=1
9     uint8 UINT8=2
10    uint8 INT16=3
11    uint8 UINT16=4
12    uint8 INT32=5
13    uint8 UINT32=6
14    uint8 FLOAT32=7
15    uint8 FLOAT64=8
16    string name
17    uint32 offset
18    uint8 datatype
19    uint32 count
20    bool is_bigendian #图像大小端存储模式
21    uint32 point_step #单点的数据字节步长
22    uint32 row_step #一行数据的字节步长
23    uint8[] data #存储点云的数组，总长度为 row_step *
24      height
24    bool is_dense #是否有无效点

```

7.3.8 深度图像转激光数据

本节介绍ROS中的一个功能包:depthimage_to_laserscan, 顾名思义, 该功能包可以将深度图像信息转换成激光雷达信息, 应用场景如下:



在诸多SLAM算法中, 一般都需要订阅激光雷达数据用于构建地图, 因为激光雷达可以感知周围环境的深度信息, 而深度相机也具备感知深度信息的功能, 且最初激光雷达价格比价比较昂贵, 那么在传感器选型上可以选用深度相机代替激光雷达吗?

答案是可以的, 不过二者发布的消息类型是完全不同的, 如果想要实现传感器的置换, 那么就需要将深度相机发布的三维的图形信息转换成二维的激光雷达信息, 这一功能就是通过depthimage_to_laserscan来实现的。

1. depthimage_to_laserscan简介

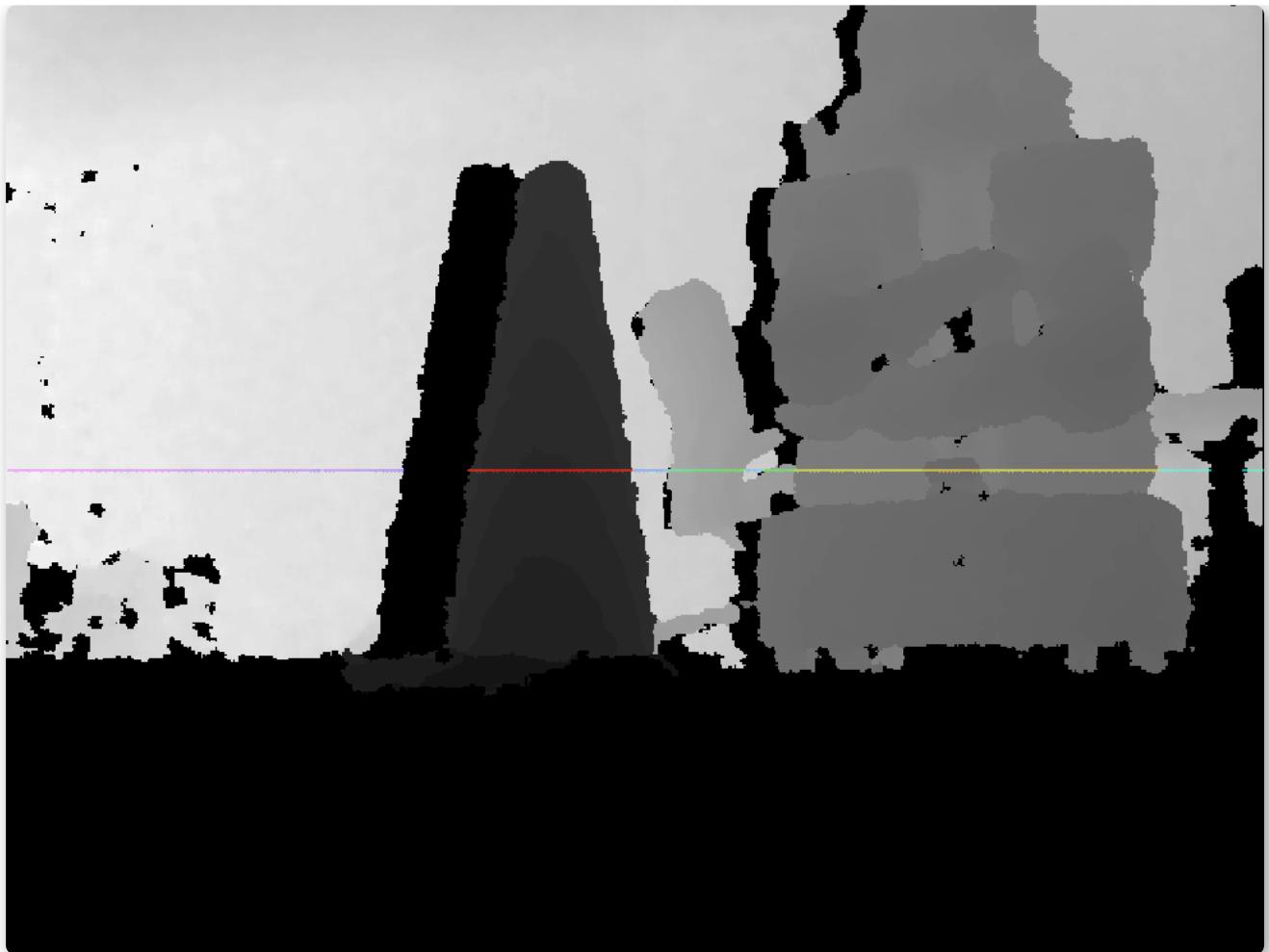
1.1 原理

depthimage_to_laserscan将实现深度图像与雷达数据转换的原理比较简单, 雷达数据是二维的、平面的, 深度图像是三维的, 是若干二维(水平)数据的纵向叠加, 如果将三维的数据转换成二维数据, 只需要取深度图的某一层即可, 为了方面理解, 请看官方示例:

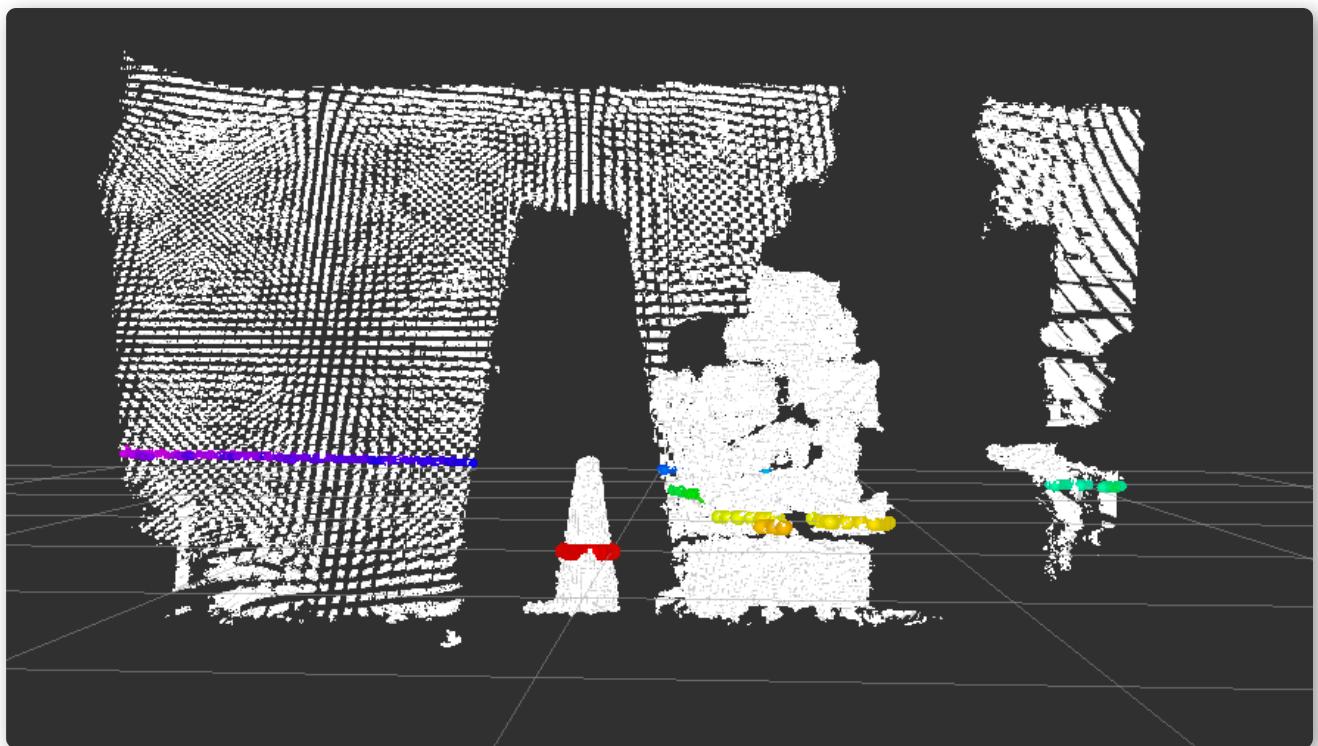
图一:深度相机与外部环境(实物图)



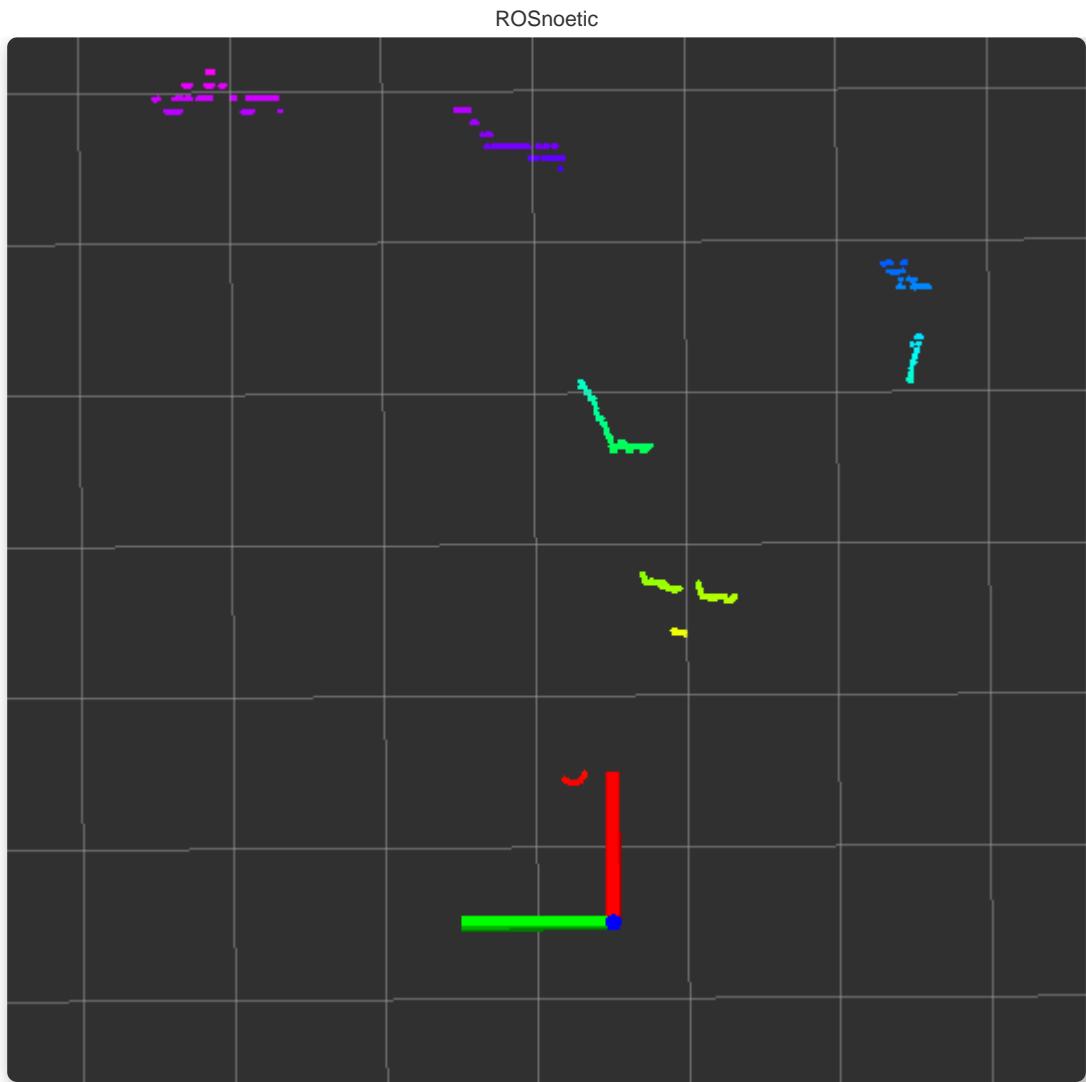
图二:深度相机发布的图片信息，图中彩线对应的是要转换成雷达信息的数据



图三:将图二以点云的方式显示更为直观, 图中彩线对应的仍然是要转换成雷达信息的数据



图四:转换之后的结果图(俯视)



1.2优缺点

优点:深度相机的成本一般低于激光雷达，可以降低硬件成本；

缺点:深度相机较之于激光雷达无论是检测范围还是精度都有不小的差距，SLAM效果可能不如激光雷达理想。

1.3安装

使用之前请先安装,命令如下:

```
1 sudo apt-get install ros-melodic-depthimage-to-laserscan
```

2.depthimage_to_laserscan节点说明

depthimage_to_laserscan 功能包的核心节点是:depthimage_to_laserscan，为了方便调用，需要先了解该节点订阅的话题、发布的话题以及相关参数。

2.1订阅的Topic

image(sensor_msgs/Image)

- 输入图像信息。

camera_info(sensor_msgs/ CameraInfo)

- 关联图像的相机信息。通常不需要重新映射，因为camera_info将从与image相同的命名空间中进行订阅。

2.2发布的Topic

scan(sensor_msgs/LaserScan)

- 发布转换成的激光雷达类型数据。

2.3参数

该节点参数较少，只有如下几个，一般需要设置的是: output_frame_id。

~scan_height(int, default: 1 pixel)

- 设置用于生成激光雷达信息的象素行数。

~scan_time(double, default: 1/30.0Hz (0.033s))

- 两次扫描的时间间隔。

~range_min(double, default: 0.45m)

- 返回的最小范围。结合range_max使用，只会获取 range_min 与 range_max 之间的数据。

~range_max(double, default: 10.0m)

- 返回的最大范围。结合range_min使用，只会获取 range_min 与 range_max 之间的数据。

~output_frame_id(str, default: camera_depth_frame)

- 激光信息的ID。

3.depthimage_to_laserscan使用

3.1编写launch文件

编写launch文件执行，将深度信息转换成雷达信息

```

1 <launch>
2   <node pkg="depthimage_to_laserscan"
3     type="depthimage_to_laserscan"
4     name="depthimage_to_laserscan">
5     <remap from="image"
6       to="/camera/depth/image_raw" />
7     <param name="output_frame_id" value="camera"
8       />
9   </node>
10 </launch>

```

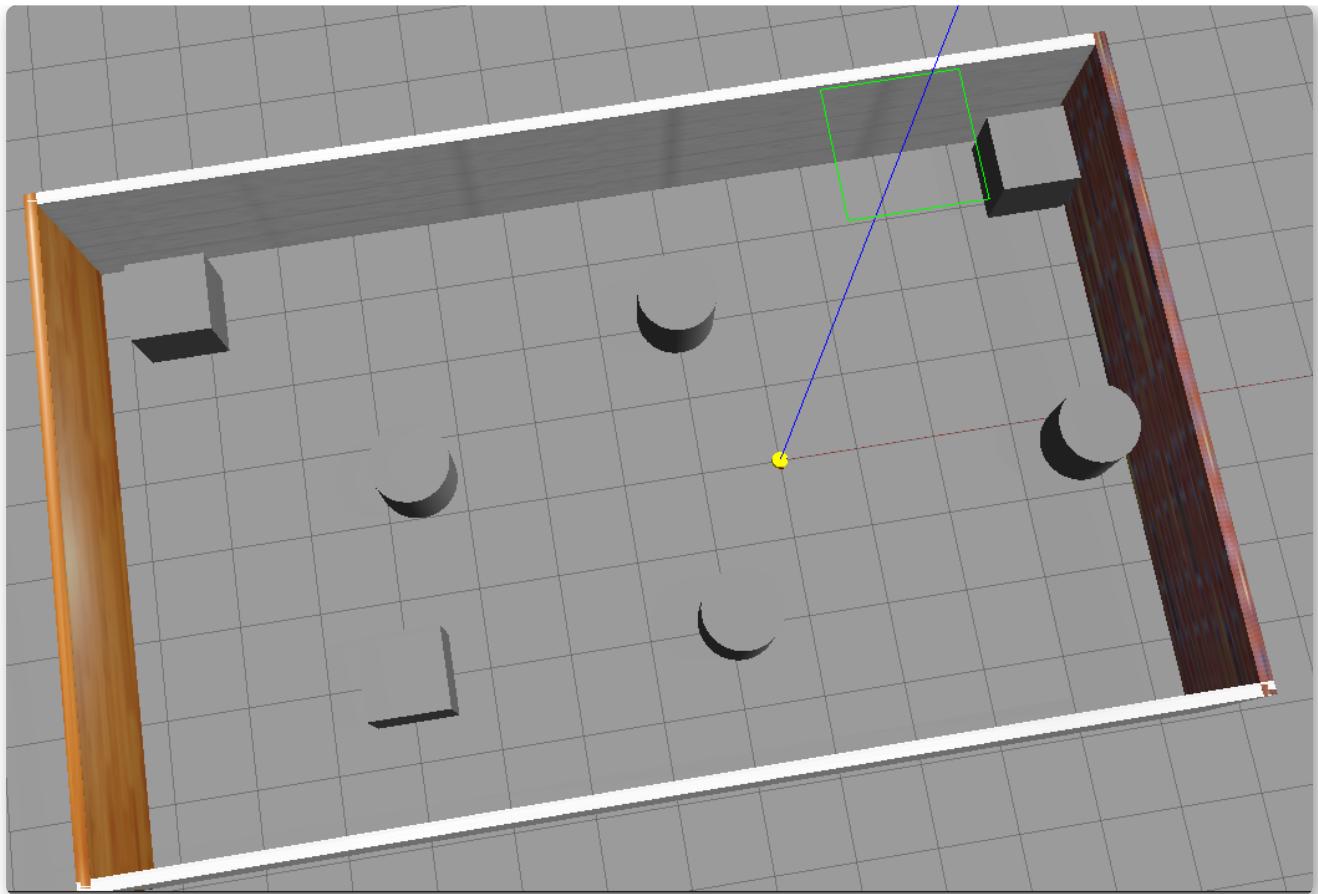
订阅的话题需要根据深度相机发布的话题设置，output_frame_id需要与深度相机的坐标系一致。

3.2修改URDF文件

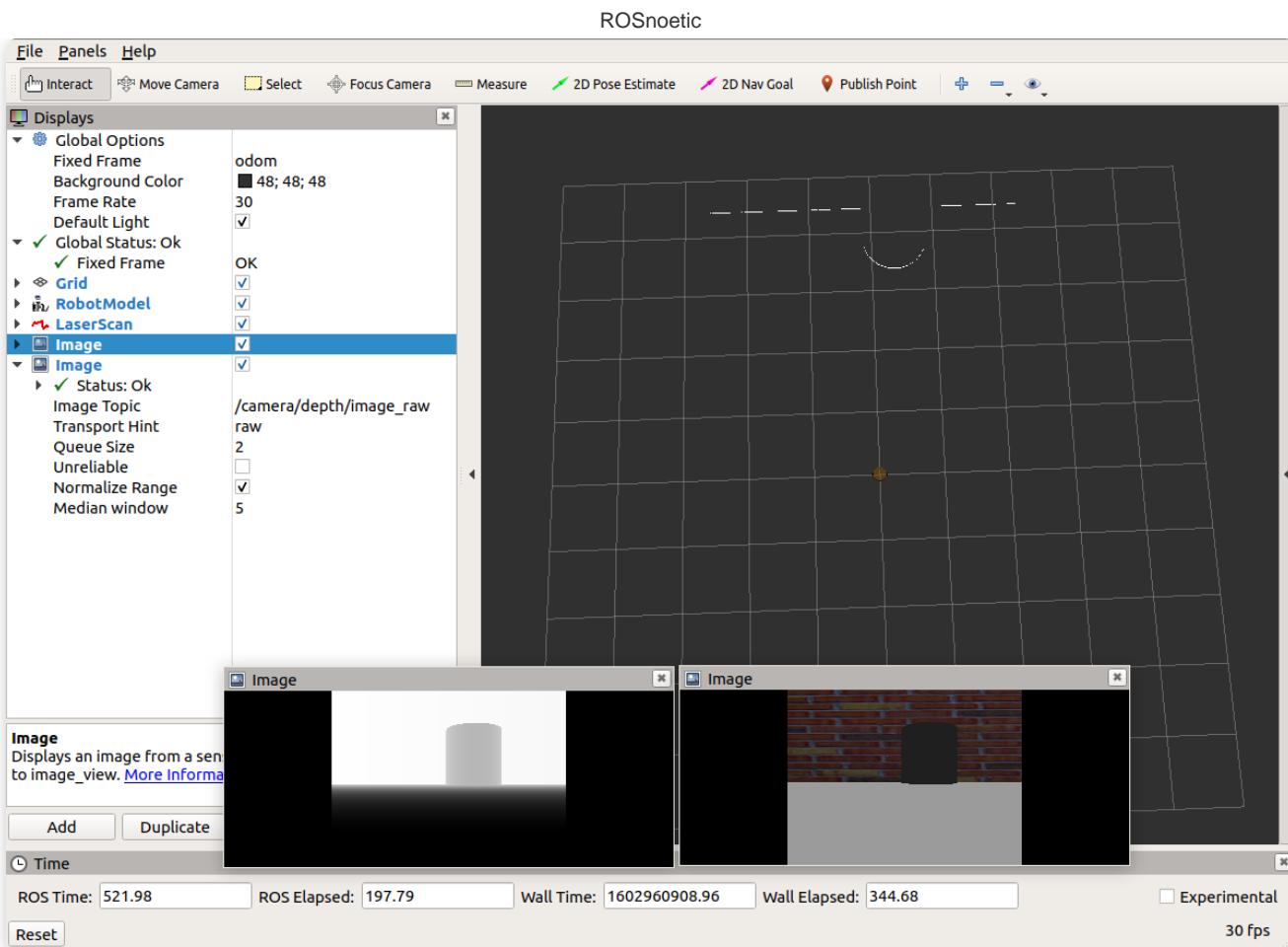
经过信息转换之后，深度相机也将发布雷达数据，为了不产生混淆，可以注释掉 xacro 文件中的关于激光雷达的部分内容。

3.3执行

1.启动gazebo仿真环境，如下：



2.启动rviz并添加相关组件(image、LaserScan)，结果如下：



4.SLAM应用

现在我们已经实现并测试通过深度图像信息转换成激光雷达信息了，接下来是实践阶段，通过深度相机实现SLAM，流程如下：

- 1.先启动 Gazebo 仿真环境；
- 2.启动转换节点；
- 3.再启动地图绘制的 launch 文件；
- 4.启动键盘控制节点，用于控制机器人运动建图；

```
1 rosrun teleop_twist_keyboard
  teleop_twist_keyboard.py
```

5.在 rviz 中添加组件，显示栅格地图最后，就可以通过键盘控制gazebo 中的机器人运动，同时，在rviz中可以显示gmapping发布的栅格地图数据了，但是，前面也介绍了，由于精度和检测范围的原因，尤其再加之环境的特征点偏少，建图效果可能并不理想，建图中甚至会出现地图偏移的情况。

7.4 本章小结

本章介绍了在仿真环境下的机器人导航实现，主要内容如下：

- 导航概念以及架构设计
- SLAM概念以及gmapping实现
- 地图的序列化与反序列化
- 定位实现
- 路径规划实现
- 导航中涉及的消息解释

导航整体设计架构中，包含地图、定位、路径规划、感知以及控制等实现，感知与控制模块在上一章机器人系统仿真中已经实现了，因此没有做过多介绍，其他部分，当前也是基于仿真环境实现的，后续，我们将搭建一台实体机器人并实现导航功能。

第8章 机器人平台设计

学习到当前阶段大家对ROS已经有一定的认知了，但是之前的内容更偏理论，尤其是介绍完第6章仿真与第7章导航之后，想必相当一部分同学有些疑惑：



实体机器人与仿真实现有什么区别？

ROS系统如何控制机器人底盘运动，并计算里程计数据呢？

实际的传感器如雷达、摄像头等应该怎么使用呢？

...

机器人系统是一套机电一体化的设备，机器人设计也是高度集成的系统性实现，为了给大家解答上述疑惑，方便机器人硬件的快速上手，本章去繁就简旨在从0到1的设计一款入门级、低成本、简单但又具备一定扩展性的两轮差速机器人，学习完本章内容之后，你甚至可以构建属于自己的机器人平台。

本章主要介绍内容如下：

- 机器人的组成部分；
- Arduino 基本使用；
- Arduino 与电机驱动；
- 底盘控制实现；
- 基于树莓派的ROS环境搭建；
- 激光雷达与相机的基本使用与集成。

本章学习目标如下：

- 能够独立搭建机器人平台。

注意：

- 该章内容会使用到ROS的分布式框架，树莓派端作为主机，PC端作为从机；

- PC端使用的ROS版本为noetic，树莓派端使用的版本为melodic，因为树莓派需要与底盘交互，而相关功能包还未更新。

案例演示：

1.机器人底盘实现

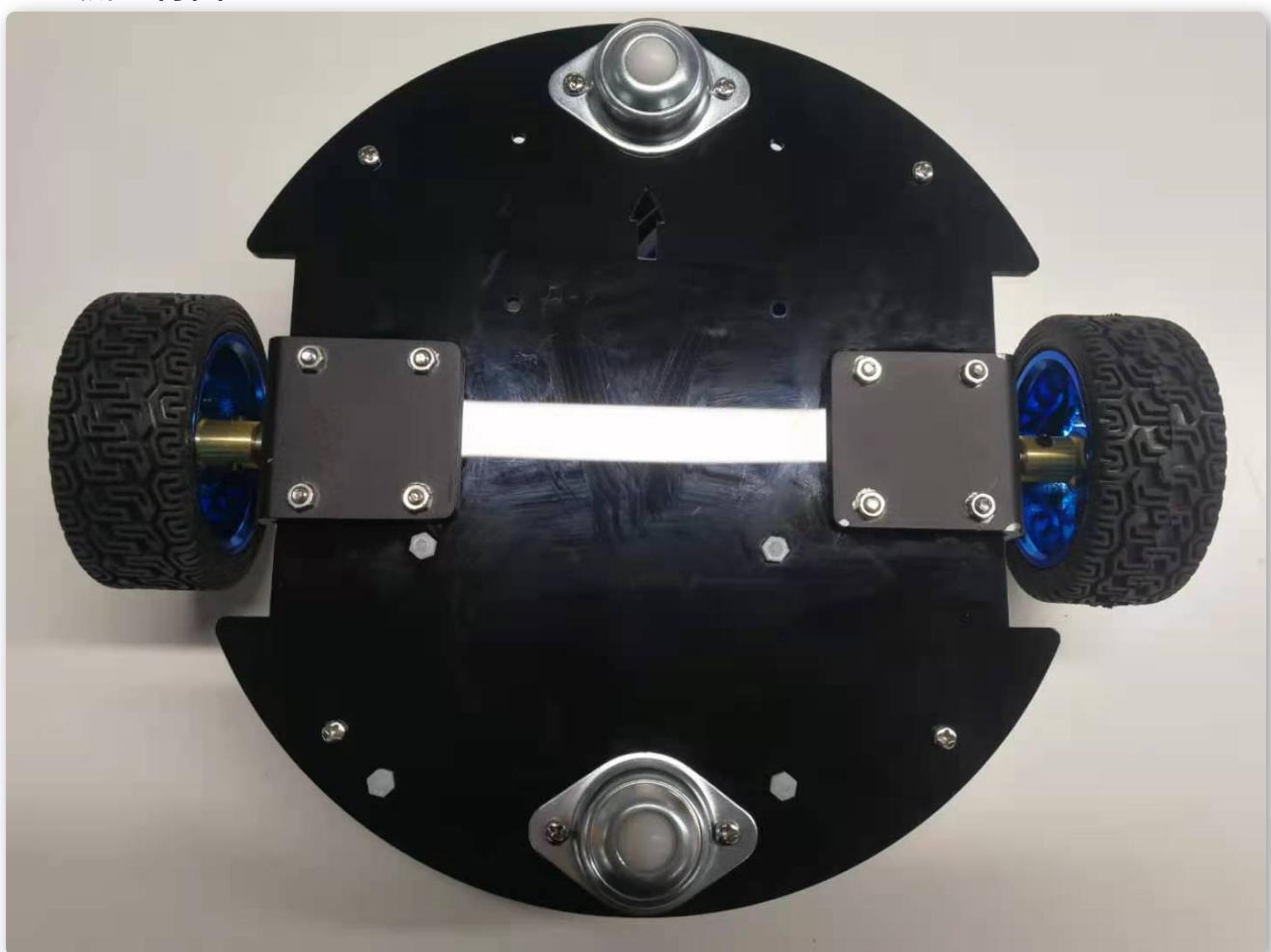
底盘正面



Arduino 与 电机驱动板



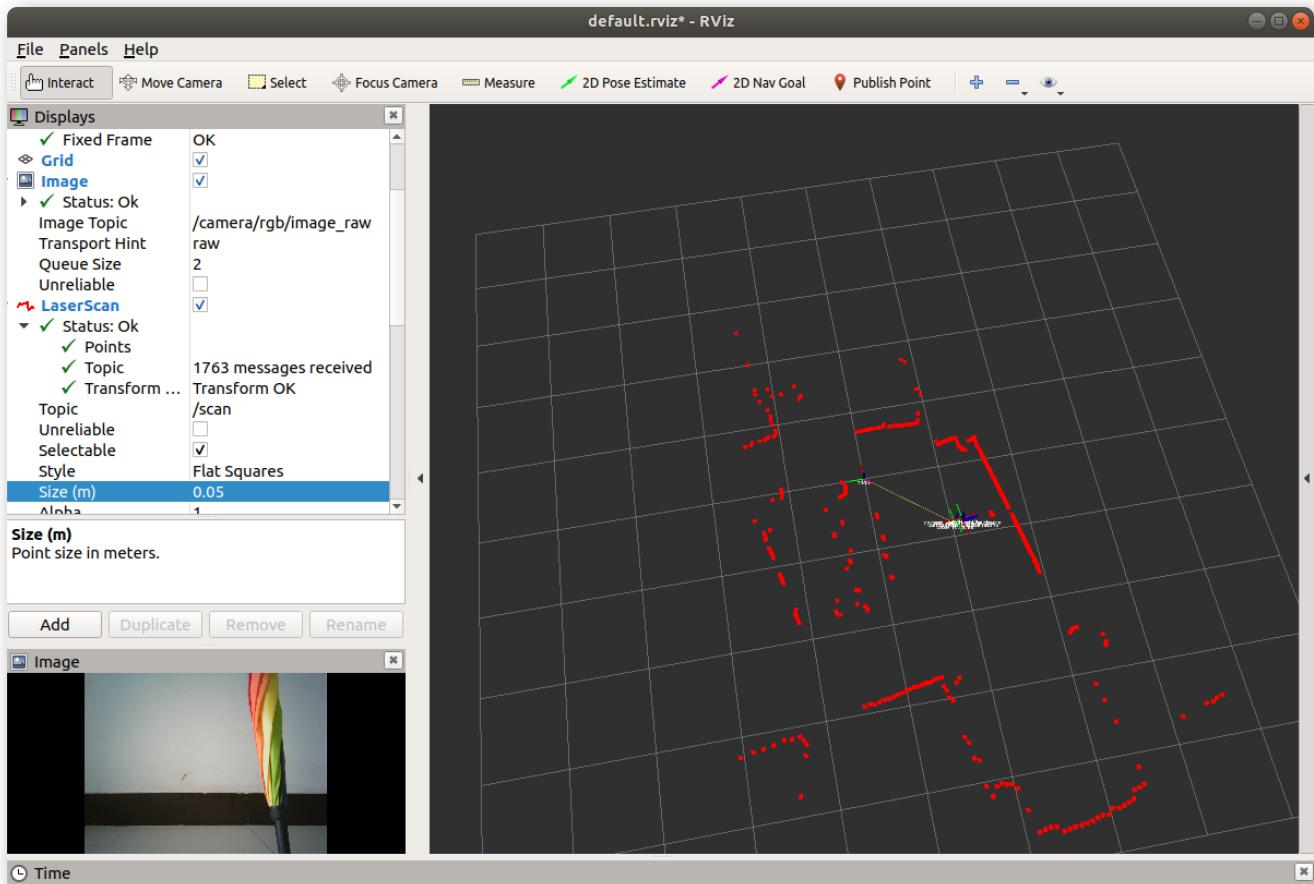
底盘背面



2.机器人控制系统以及传感器实现



3.机器人集成效果



8.1 概述

立足角度不同，对机器人组成的认识也会有明显差异，从控制的角度来看，机器人系统可以分为四部分：



传感系统、控制系统、驱动系统、执行机构。

1. 传感系统

它由内部传感器模块和外部传感器模块组成，获取内部和外部环境中有用的信息，相当于人体的感官与神经，内部传感系统包括电机的编码器、陀螺仪等，可以通过自身信号反馈检测位姿状态；外部传感系统包括摄像头、红外、声纳等，用于感知外部环境。

2. 控制系统

控制系统的任务是根据机器人的作业指令以及从传感器反馈回来的信号，输出控制命令信号，类似于人的大脑。控制系统需要基于处理器实现，在处理器之上，控制系统需要完成算法处理、关节控制、人机交互等复杂功能。

3. 驱动系统

驱动系统主要负责驱动执行机构，将控制系统下达的命令转换成执行机构所需要的信号，相当于人的小脑与神经。采用的动力源不同，驱动系统的传动方式也不同。驱动系统的传动方式主要有四种：液压式、气压式、电气式和机械式。电力驱动是目前使用最多的一种驱动方式，其特点是电源取用方便，响应快，驱动力大，信号检测、传递、处理方便，并可以采用多种灵活的控制方式，驱动电机一般采用步进电机或伺服电机。

4. 执行机构

执行机构是机器人组成中的机械部分，类似于人的手与脚，比如：机器人的行走部分与机械臂。

在当前机器人系统中，各组成部分对应硬件清单如下：

执行机构: 主体使用亚克力板拼装，由两个直流电机带动主动轮以及保持平衡的两个万向轮实现机器人行走，由于执行机构比较简单，不再做单独介绍。

驱动系统: 电池、arduino 以及电机驱动模块；

控制系统: 树莓派；

传感系统: 编码器、单线激光雷达、相机；

其中，执行机构与驱动系统构成了机器人底盘。

8.2 机器人平台设计之arduino基础

在构建差分轮式机器人平台时，驱动系统的常用实现有 STM32 或 Arduino，在此，我们选用后者，因为 Arduino 相较而言更简单、易于上手。本节将介绍如下内容：

- arduino 简介；
- arduino 开发环境搭建；
- arduino 基本语法。

概念

Arduino 是一款便捷灵活、方便上手的开源电子原型平台。在它上面可以进行简单的电路控制设计，Arduino 能够通过各种各样的传感器来感知环境，通过控制灯光、马达和其他的装置来反馈、影响环境。

作用

或多或少你可能听说过“集成电路”（又称“微电路”、“微芯片”或“芯片”）这种概念，集成电路（integrated circuit）是一种微型电子器件或部件，通过集成电路再结合一些外围的电子元器件、传感器等，可以感知环境（温度、湿度、声音），也可以影响环境（控制灯的开关、调节电机转速）。但

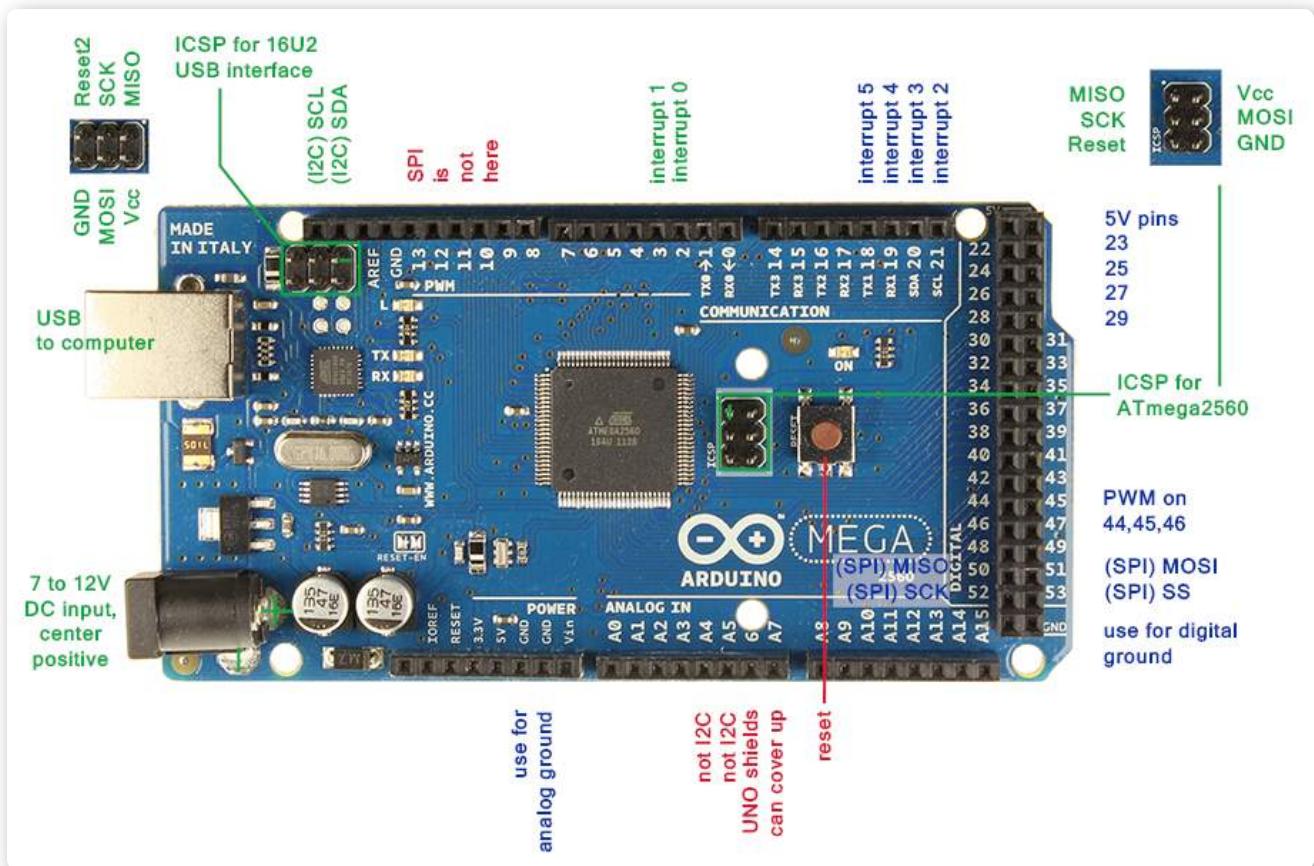
是传统的集成电路应用比较繁琐，一般需要具有一定电子知识基础，并懂得如何进行相关的程序设计的工程师才能熟练使用，而Arduino的出现才使得以往高度专业的集成电路变得平易近人，Arduino主要优点如下：

- **简单:**在硬件方面，Arduino本身是一款非常容易使用的印刷电路板。电路板上装有专用集成电路，并将集成电路的功能引脚引出方便我们外接使用。同时，电路板还设计有USB接口方便与电脑连接；
- **易学:**只需要掌握 C/C++ 基本语法即可；
- **易用:**Arduino提供了专门的程序开发环境Arduino IDE，可以提高程序实现效率。

当前，Arduino已经成为全世界电子爱好者电子制作过程中的重要选项之一。

组成

Arduino 体系主要包含硬件和软件两大部分。硬件部分是可以用来做电路连接的各种型号的Arduino电路板(下图为本章内容使用的 arduino mega 2560)；软件部分则是Arduino IDE。你只要在IDE中编写程序代码，将程序上传到Arduino电路板后，程序便会告诉Arduino电路板要做些什么了。

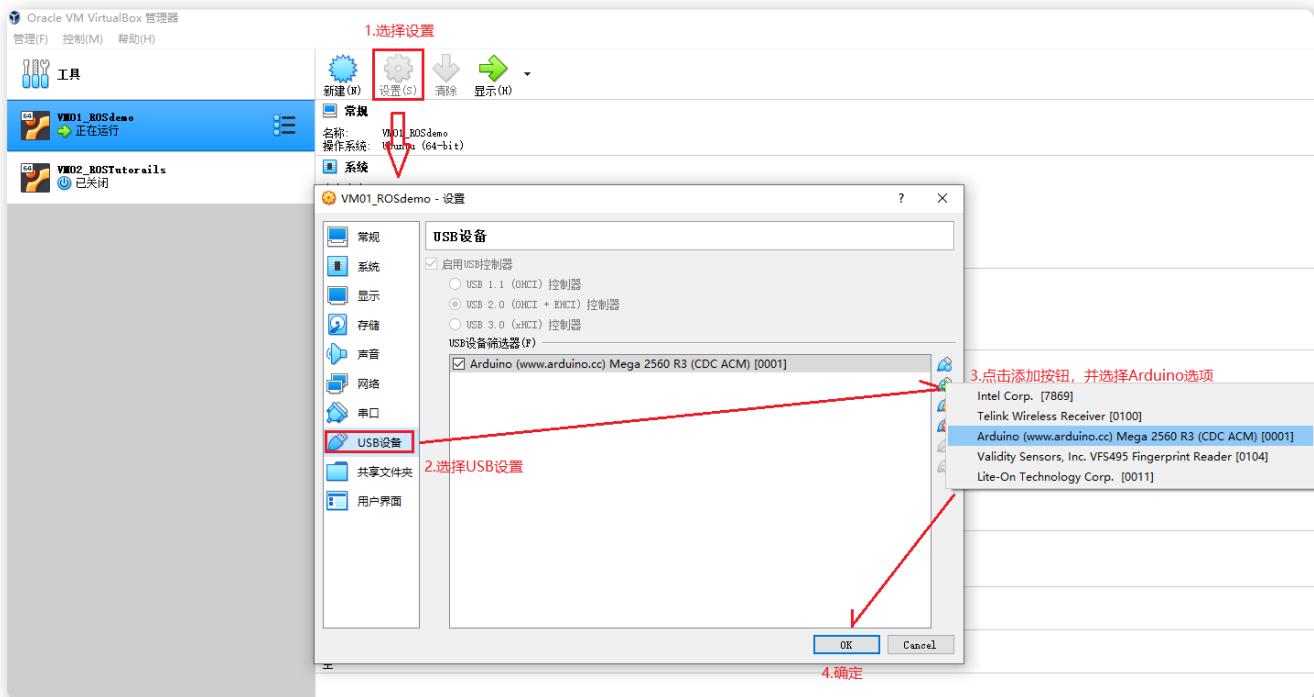


8.2.1 arduino 开发环境搭建

基于Arduino的开发实现，毋庸置疑的必须先要准备Arduino电路板(建议型号:Arduino Mega 2560)，除了硬件之外，还需要准备软件环境，安装Arduino IDE，在Ubuntu下，Arduino开发环境的搭建步骤如下：

1. 硬件准备: Arduino电路板连接ubuntu
2. 软件准备: 安装 Arduino IDE
3. 编写 Arduino 程序并上传至 Arduino 电路板。

1.Arduino 连接 Ubuntu



你需要确保你对这个接口有访问的权限。假设你的Arduino连接的是 `/dev/ttyACM0`，那么就运行下面这个命令：

```
1 $ ls -l /dev/ttyACM0
```

然后你就可以看到类似于下面的输出结果：



```
crw-rw-- 1 root dialout 166, 0 2013-02-24 08:31 /dev/ttyACM0
```

我们注意到在上面的结果中，只有root和” dialout” 组才有读写权限。因此，你需要成为 `dialout` 组的一个成员。

命令如下：

```
1 $ sudo usermod -a -G dialout your_user_name
```

在这个命令中 `your_user_name` 就是你在Linux下登录的用户名。然后需要重启使之生效。执行完上面的操作之后，你可以运行下面的命令查看一下：

```
1 $ groups
```

然后如果你可以在列出的组中找到dialout，这就说明你已经加入到dialout中了。

2. 安装 Arduino IDE

1. 下载arduino ide安装包

官方下载链接：<https://www.arduino.cc/en/Main/Software>

Download the Arduino IDE

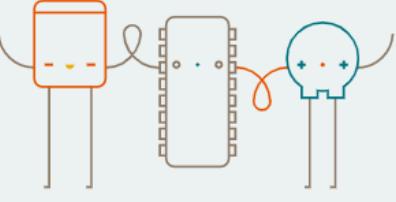


The screenshot shows the official Arduino website's download page. On the left, there's a large teal circular icon with a white infinity symbol containing a minus and a plus sign. To its right, the text "ARDUINO 1.8.13" is displayed, followed by a brief description of the software: "The open-source Arduino Software (IDE) makes it easy to write code and upload it to the board. It runs on Windows, Mac OS X, and Linux. The environment is written in Java and based on Processing and other open-source software. This software can be used with any Arduino board. Refer to the [Getting Started](#) page for installation instructions." On the right side, there are download links for different operating systems. For Windows, it offers a "Windows Installer, for Windows 7 and up" and a "Windows ZIP file for non admin install". For Mac OS X, it says "Mac OS X 10.10 or newer". For Linux, it lists "Linux 32 bits", "Linux 64 bits", "Linux ARM 32 bits", and "Linux ARM 64 bits". At the bottom, there are links for "Release Notes", "Source Code", and "Checksums (sha512)".

选择对应版本即可

Contribute to the Arduino Software

Consider supporting the Arduino Software by contributing to its development. (US tax payers, please note this contribution is not tax deductible). [Learn more on how your contribution will be used.](#)



SINCE MARCH 2015, THE ARDUINO IDE HAS BEEN DOWNLOADED **SO MANY** TIMES. (IMPRESSIVE!) NO LONGER JUST FOR ARDUINO AND GENUINO BOARDS, HUNDREDS OF COMPANIES AROUND THE WORLD ARE USING THE IDE TO PROGRAM THEIR DEVICES, INCLUDING COMPATIBLES, CLONES, AND EVEN COUNTERFEITS. HELP ACCELERATE ITS DEVELOPMENT WITH A SMALL CONTRIBUTION! REMEMBER: OPEN SOURCE IS LOVE!

\$3 **\$5** **\$10** **\$25** **\$50** **OTHER**

JUST DOWNLOAD **CONTRIBUTE & DOWNLOAD**

可以选择捐款，或者 JUST DOWNLOAD。

2. 使用tar命令对压缩包解压

```
1 tar -xvf arduino-1.x.y-linux64.tar.xz
```

3. 将解压后的文件移动到/opt下

```
1 sudo mv arduino-1.x.y /opt
```

4. 进入安装目录,对install.sh添加可执行权限,并执行安装

```
1 cd /opt/arduino-1.x.y
2 sudo chmod +x install.sh
3 sudo ./install.sh
```

5. 启动并配置 Arduino IDE

在命令行直接输入:arduino,或者点击左下的显示应用程序搜索 arduino IDE。启动如下:

文件 编辑 项目 工具 帮助

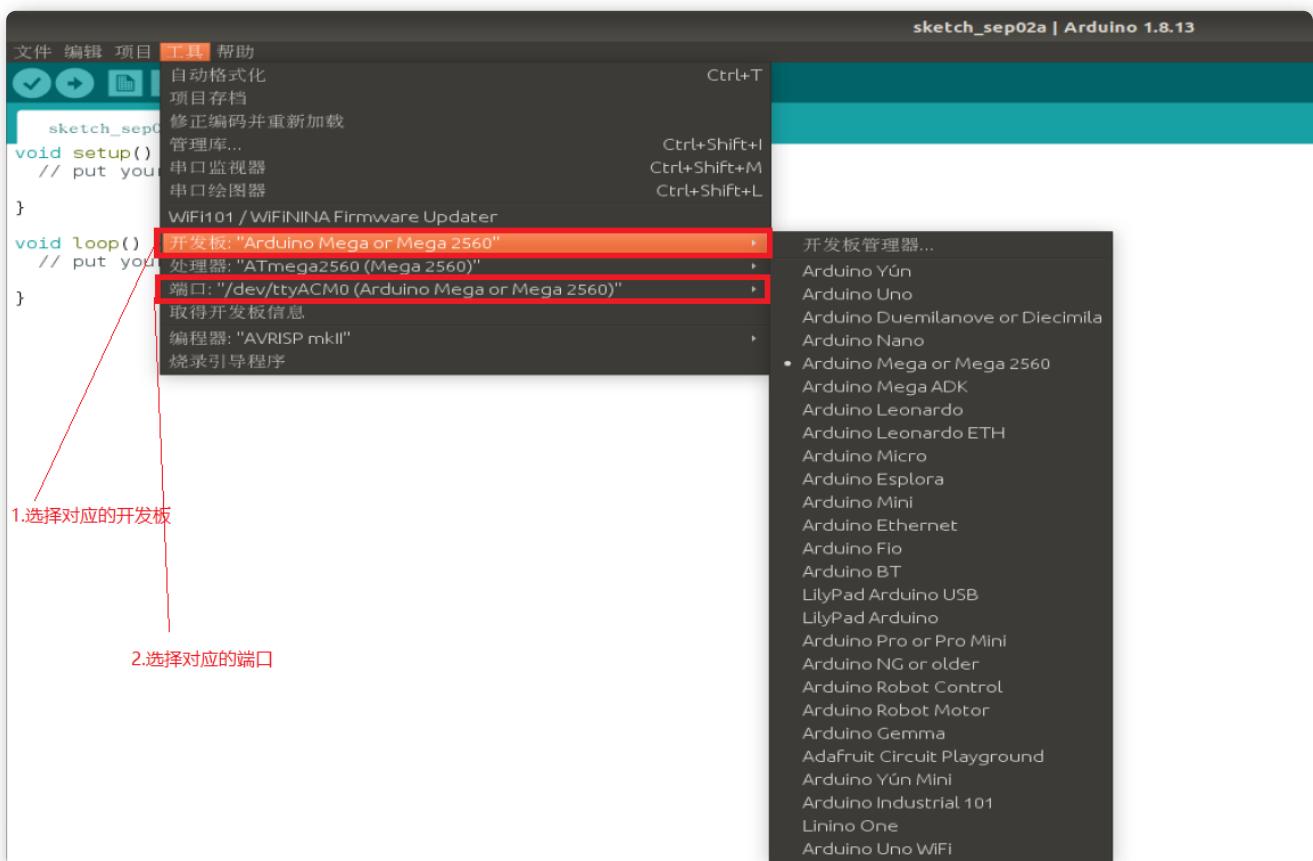


```
sketch_sep02a
void setup() {
  // put your setup code here, to run once:
}

void loop() {
  // put your main code here, to run repeatedly:
}
```

Arduino Mega or Mega 2560, ATmega2560 (Mega 2560) 在 /dev/ttyACM0

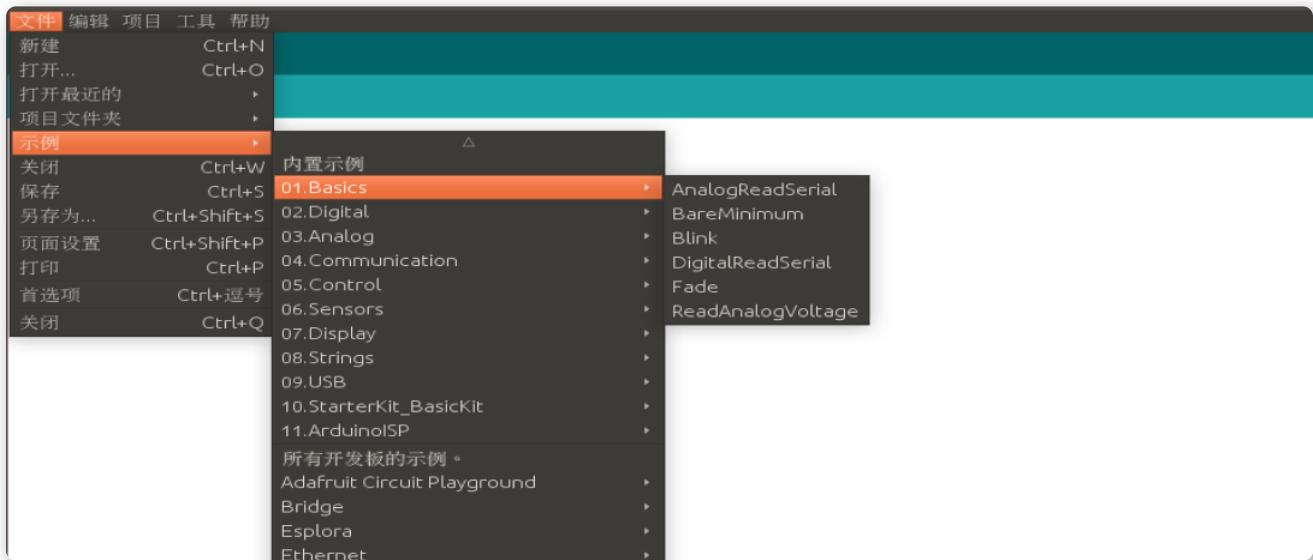
Arduino IDE 配置如下：



3. Hello World 实现

Arduino IDE 中已经内置了一些相关案例，在此，我们通过一个经典的控制 LED 等闪烁案例来演示 Arduino 的使用流程：

1.案例调用



2.编译及上传

先点击左上的编译按钮，可用于语法检测，编译无异常，再点击右侧的上传按钮，上传至 Arduino 电路板



3.运行结果

电路板上的 LED 灯闪烁

4.代码解释

```

1 // 初始化函数
2 void setup() {
3     //将LED灯引脚(引脚值为13, 被封装为了LED_BUTLIN)设置为
4     pinMode(LED_BUILTIN, OUTPUT);
5 }
6
7 // 循环执行函数
8 void loop() {
9     digitalWrite(LED_BUILTIN, HIGH);      // 打开LED灯
10    delay(1000);                      // 休眠1000毫秒
11    digitalWrite(LED_BUILTIN, LOW);      // 关闭LED灯
12    delay(1000);                      // 休眠1000毫秒
13 }

```

setup 与 loop 函数是固定格式。

8.2.2 arduino 基本语法概述

Arduino 的语言系统在设计时参考了C、C++、Java，是一种综合性的简洁语言，语法更类似于C++，但是不支持C++的异常处理，没有STL库，你可以把它当作是精简后的C++。

Arduino 基本语法中，注释、宏定义、库文件包含、变量、函数、流程控制、类、继承、多态..... 都与 C++ 高度类似，在此不再赘述，着重要介绍的是，Arduino中的一些API实现。

1.程序结构

一个 Arduino 程序分为两大部分: setup() 与 loop() 函数。

- **void setup():**在这个函数里初始化Arduino的程序，使主循环程序在开始之前设置好相关参数,初始化变量、设置针脚的输出\输入类型、设置波特率...。该函数只会在上电或重启时执行一次。
- **void loop():**这是Arduino的主函数。这套程序会一直重复执行，直到电源被断开。

2.常量

在 Arduino 中封装了一些常用常量,比如:

- **HIGH | LOW** (引脚电压定义)
- **INPUT|OUTPUT** (数字引脚 (Digital pins) 定义)
- **true | false** (逻辑层定义)

3.通信_Serial

Serial用于Arduino控制板和一台计算机或其他设备之间的通信。您可以使用Arduino IDE内置的串口监视器与Arduino板通信。点击工具栏上的串口监视器按钮, 调用begin()函数 (选择相同的波特率) 。

- **Serial.begin()** 初始化串口波特率

描述:将串行数据传输速率设置为位/秒 (波特) 。与计算机进行通信时, 可以使用这些波特率: 300, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600或115200。当然, 您也可以指定其他波特率 - 例如, 引脚0和1和一个元件进行通信, 它需要一个特定的波特率。

语法:Serial.begin(speed)

参数:speed: 位/秒 (波特) - long

返回:无

- **Serial.print()** 从串口打印输出数据

需求:

以人们可读的ASCII文本形式打印数据到串口输出。此命令可以采取多种形式。每个数字的打印输出使用的是ASCII字符。浮点型同样打印输出的是ASCII字符, 保留到小数点后两位。Bytes型则打印输出单个字符。字符和字符串原样打印输出。Serial.print()打印输出数据不换行, Serial.println()打印输出数据自动换行处理。

语法:Serial.print(val)

参数

val: 打印输出的值 - 任何数据类型

返回:字节 `print()` 将返回写入的字节数, 但是是否使用 (或读出) 这个数字是可设定的

- `Serial.println()` 打印输出数据自动换行处理。参考 `Serial.print()`;
- `Serial.available()`

描述:获取从串口读取有效的字节数 (字符)。这是已经传输到, 并存储在串行接收缓冲区 (能够存储64个字节) 的数据。 `available()` 继承了 `Stream`类。

语法:`Serial.available()`

参数:无

返回:可读取的字节数

- `Serial.read()`

描述:读取传入的串口的数据。 `read()` 继承自 `Stream`类。

语法:`serial.read()`

参数:无

返回:传入的串口数据的第一个字节 (或-1, 如果没有可用的数据)

4. 函数_数字IO

- `pinMode()`

描述:将指定的引脚配置成输出或输入。

语法:`pinMode(pin, mode)`

参数

pin:要设置模式的引脚

mode:INPUT或OUTPUT

返回:无

- `digitalWrite()`

描述:给一个数字引脚写入HIGH或者LOW。

语法:`digitalWrite(pin, value)`

参数

pin: 引脚编号 (如1,5,10, A0, A3)

value: HIGH or LOW

返回:无

- **digitalRead()**

描述:读取指定引脚的值, HIGH或LOW。

语法:digitalRead (PIN)

参数

pin: 你想读取的引脚号 (int)

返回:HIGH 或 LOW

注意:如果引脚悬空, digitalRead()会返回HIGH或LOW (随机变化)

5. 函数_模拟IO

- **analogWrite() PWM**

描述:从一个引脚输出模拟值 (PWM)。可用于让LED以不同的亮度点亮或驱动电机以不同的速度旋转。analogWrite()输出结束后, 该引脚将产生一个稳定的特殊占空比方波, 直到下次调用analogWrite() (或在同一引脚调用digitalRead()或digitalWrite())。PWM信号的频率大约是490赫兹。

在大多数arduino板 (ATmega168或ATmega328), 只有引脚3, 5, 6, 9, 10和11可以实现该功能。在aduino Mega上, 引脚2到13可以实现该功能。老的Arduino板 (ATmega8) 的只有引脚9、10、11可以使用analogWrite()。在使用analogWrite()前, 你不需要调用pinMode()来设置引脚为输出引脚。

语法:analogWrite (pin,value)

参数

pin: 用于输入数值的引脚。

value: 占空比: 0 (完全关闭) 到255 (完全打开) 之间。

返回:无

6. 函数_时间

- **delay()**

描述:使程序暂定设定的时间（单位毫秒）。（一秒等于1000毫秒）

语法:delay(ms)

参数

ms: 暂停的毫秒数 (unsigned long)

返回:无

- **millis()**

描述:返回Arduino开发板从运行当前程序开始的毫秒数。这个数字将在约50天后溢出（归零）。

参数:无

返回:返回从运行当前程序开始的毫秒数（无符号长整数）。

7. 函数_中断

- **attachInterrupt()**

描述:当发生外部中断时，调用一个指定函数。当中断发生时，该函数会取代正在执行的程序。大多数的Arduino板有两个外部中断：0（数字引脚2）和1（数字引脚3）。

arduino Mege还有其它有四个外部中断：数字2（引脚21），3（引脚20），4（引脚19），5（引脚18）。

语法:attachInterrupt(interrupt, function, mode)

interrupt: 中断引脚数

function: 中断发生时调用的函数，此函数必须不带参数和不返回任何值。该函数有时被称为中断服务程序。

mode: 定义何时发生中断以下四个constants预定有效值：

- LOW 当引脚为低电平时，触发中断
- CHANGE 当引脚电平发生改变时，触发中断
- RISING 当引脚由低电平变为高电平时，触发中断

- FALLING 当引脚由高电平变为低电平时，触发中断。

返回:无

注意事项:当中断函数发生时，delay()和millis()的数值将不会继续变化。当中断发生时，串口收到的数据可能会丢失。你应该声明一个变量来在未发生中断时储存变量。

- **noInterrupts()** (禁止中断)

描述:禁止中断（重新使能中断interrupts()）。中断允许在后台运行一些重要任务，默认使能中断。禁止中断时部分函数会无法工作，通信中接收到的信息也可能会丢失。

中断会稍影响计时代码，在某些特定的代码中也会失效。

参数:无

返回:无

- **interrupts()** (中断)

描述:重新启用中断（使用noInterrupts()命令后将被禁用）。中断允许一些重要任务在后台运行，默认状态是启用的。禁用中断后一些函数可能无法工作，并传入信息可能会被忽略。中断会稍微打乱代码的时间，但是在关键部分可以禁用中断。

参数:无

返回:无

Arduino 的API还有很多，但是受于篇幅限制，当前只是简单介绍了和本教程相关的一些API实现。

8.2.3 arduino 基本语法演示

通信操作

1. 通信实现01

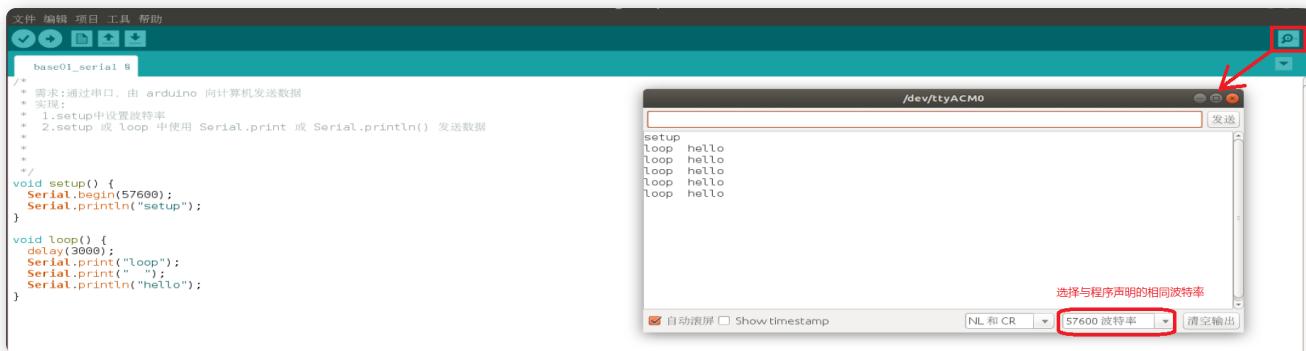
需求: 通过串口，由 arduino 向计算机发送数据

实现:

```

1  /*
2   * 需求:通过串口, 由 arduino 向计算机发送数据
3   * 实现:
4   *   1.setup中设置波特率
5   *   2.setup 或 loop 中使用 Serial.print 或
6   *     Serial.println() 发送数据
7   *
8   *
9   */
10 void setup() {
11   Serial.begin(57600);
12   Serial.println("setup");
13 }
14
15 void loop() {
16   delay(3000);
17   Serial.print("loop");
18   Serial.print(" ");
19   Serial.println("hello");
20 }

```



2. 通信实现02

需求: 通过串口, 由计算机向Arduino发送数据

实现:

```

1  /*
2   * 需求:通过串口, 由计算机向 arduino 发送数据
3   * 实现:

```

```

4  * 1. setup中设置波特率
5  * 2. loop 中接收发送的数据，并打印
6  *
7  *
8  *
9  */
10 char num;
11 void setup() {
12     Serial.begin(57600);
13 }
14
15 void loop() {
16     if(Serial.available() > 0){
17         num = Serial.read();
18         Serial.print("I accept:");
19         Serial.println(num);
20     }
21 }

```



8.2.4 arduino 基本语法演示02

1. 数字IO操作

需求:控制LED灯开关，在一个循环周期内前两秒使LED灯处于点亮状态，后两秒关闭LED灯

实现:

```

1  /*
2   * 控制LED灯开关，在一个循环周期内前两秒使LED灯处于点亮状态，后两秒关闭LED灯

```

```

3  * 1. setup 中设置引脚为输出模式
4  * 2. loop 中向引脚输出高电压, 休眠 2000 毫秒后, 再输出低
5  *
6  */
7 int led = 13;
8 void setup() {
9     Serial.begin(57600);
10    pinMode(led,OUTPUT);
11
12 }
13
14 void loop() {
15
16     digitalWrite(led,HIGH); //输出高电压
17     delay(2000);
18
19     digitalWrite(led,LOW); //输出低电压
20     delay(2000);
21
22 }

```

2. 模拟IO操作

需求:控制LED灯亮度

原理:在1中LED灯只有关闭或开启两种状态, 是无法控制 LED 灯亮度, 如果要实现此功能, 那么需要借助于 PWM(Pulse width modulation 脉冲宽度调制)技术, 通过设置占空比为LED间歇性供电, PWM 的取值范围 [0,255]。

实现:

```

1 /*
2  * 需求:控制LED灯亮度
3  * 实现:
4  * 1. setup 中设置 led 灯的引脚为输出模式
5  * 2. 设置不同的 PWM 并输出
6  *

```

```

7  /*
8  int led = 13;
9  int l1 = 255;
10 int l2 = 50;
11 int l3 = 0;
12 void setup() {
13     pinMode(led,OUTPUT);
14 }
15
16 void loop() {
17     analogWrite(led,l1);
18     delay(2000);
19     analogWrite(led,l2);
20     delay(2000);
21     analogWrite(led,l3);
22     delay(2000);
23
24 }

```

运行结果:在一个周期内LED灯亮度递减直至熄灭

8.2.5 arduino 基本语法演示03

需求:调用 `millis()` 函数获取程序当前已经执行的时间, 调用`delay()`函数实现休眠

实现:

```

1 /*
2  * 需求:调用 millis() 函数获取程序当前已经执行的时间, 调用
3  * delay()函数实现休眠
4  * 1.setup 中设置波特率
5  * 2.loop 中使用delay休眠, 使用millis获取程序执行时间并
6  * 输出
7  */
8

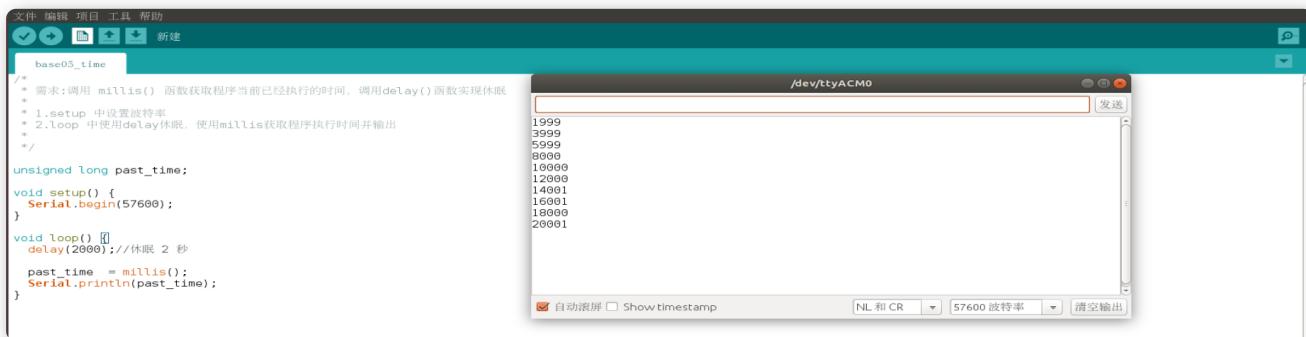
```

```

9  unsigned long past_time;
10
11 void setup() {
12     Serial.begin(57600);
13 }
14
15 void loop() {
16     delay(2000); //休眠 2 秒
17
18     past_time = millis();
19     Serial.println(past_time);
20 }

```

通过串口监视器查看输出结果。



8.3 机器人平台设计之电机驱动

对于构建轮式机器人而言，电机驱动是一重要实现环节。



场景:在机器人架构中，如果要实现机器人移动，其中一种实现策略是:控制系统会先发布预期的车辆速度信息，然后驱动系统订阅到该信息，不断调整电机转速直至达到预期速度，调速过程中还需要时时获取实际速度并反馈给控制系统，控制系统会计算实际位移并生成里程计信息。

在上述流程中，控制系统(ROS端)其实就是典型的发布和订阅实现，而具体到驱动系统(Arduino)层面，需要解决的问题有如下几点:

- 一个周期伊始，Arduino 如何订阅控制系统发布的速度相关信息？
- 一个周期结束，Arduino 如何发布实际速度相关信息到控制系统？

- 一个周期之中，Arduino 如何驱动电机(正传、反转)?
- 一个周期之中，Arduino 如何实现电机测速?
- 一个周期之中，Arduino 如何实现电机调速?

在整个闭环实现中，前两个问题涉及到驱动系统与控制系统的通信，其中控制系统会将串口通信的相关实现封装，暂时不需要关注，而Arduino端数据的接收与发送都可以通过之前介绍的 Serial 相关API实现，本节主要介绍后面三个问题的解决方式也即电机基本控制、电机测速以及电机调速实现，主要内容如下：

- 硬件:主要介绍电机类型与结构以及电机驱动板；
- 电机转向控制与电机转速的控制；
- 电机测速实现；
- 电机调速实现。

8.3.1 硬件_电机与电机驱动板

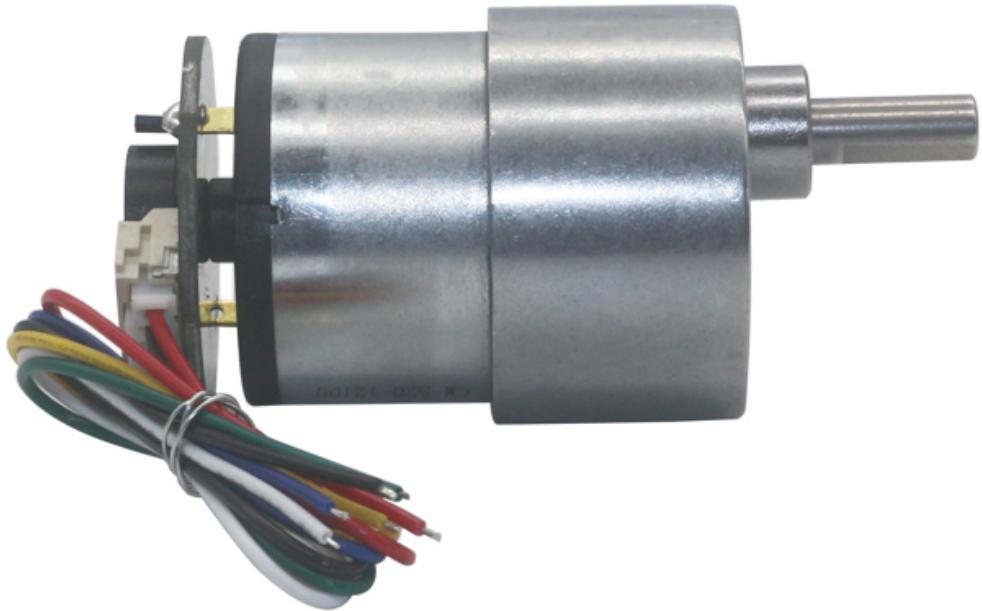
如果要通过Arduino实现电机相关操作(比如:转向控制、转速控制、测速等)，那么必须先要具备两点前提知识：

1. 需要简单了解电机类型、机械结构以及各项参数，这些是和机器人的负载、极限速度、测速结果等休戚相关的；
2. 还需要选配合适的电机驱动板，因为Arduino的输出电流不足以直接驱动电机，需要通过电机驱动板放大电机控制信号。

当前我们的机器人平台使用的电机为直流减速电机，电机驱动板为基于L298P实现的电路板。接下来就分别介绍这两个模块：

1. 直流减速电机

如图所示，相当一部分ROS智能车中使用的直流减速电机与之类似，主要由三部分构成：



- 减速箱
- 电机主体
- 编码器



电机主体通过输入轴与减速箱相连接，通过减速箱的减速效果，最终外端的输出轴会按照比例(取决于减速箱减速比)降低电机输入轴的转速，当然速度降低之后，将提升电机的力矩。

尾部是AB相霍尔编码器，通过AB编码器输出的波形图，可以判断电机的转向以及计算电机转速

另外，即便电机外观相同，具体参数也可能存在差异，参数需要商家提供，需要了解的参数如下：

- 额定电压
- 额定电流
- 额定功率
- 额定扭矩
- 减速比
- 减速前转速
- 减速后转速
- 编码器精度



主要参数：

额定扭矩:额定扭矩和机器人质量以及有效负荷相关，二者正比例相关，额定扭矩越大，可支持的机器人质量以及有效负荷越高；

减速比:电机输入轴与输出轴的减速比例，比如：减速比为90，意味着电机主体旋转90圈，输出轴旋转1圈。

减速后转速:与减速比相关，是电机减速箱输出轴的转速，单位是 rpm(转/分)，减速后转速与减速前转速存在转换关系：减速后转速 = 减速前转速 / 减速比。另外，可以根据官方给定的额定功率下的减速后转速结合车轮参数来确定小车最大速度。

编码器精度:是指编码器旋转一圈单相(当前编码器有AB两相)输出的脉冲数；

注意:电机输入轴旋转一圈的同时，编码器旋转一圈，如果输出轴旋转一圈，那么编码器的旋转圈数和减速比一致(比如减速比是90，那么输出轴旋转一圈，编码器旋转90圈)。

编码器输出的脉冲数计算公式则是：输出轴旋转一圈产生的脉冲数 = 减速比 * 编码器旋转一圈发送的脉冲数(比如：减速比为90，编码器旋转一圈输出11个脉冲，那么输出轴旋转一圈总共产生 $11 * 90$ 也即990个脉冲)。

电机编码器



M1: 电机电源+(和M2对调可以正反转)

GND: 编码器电源-

C2: 信号线

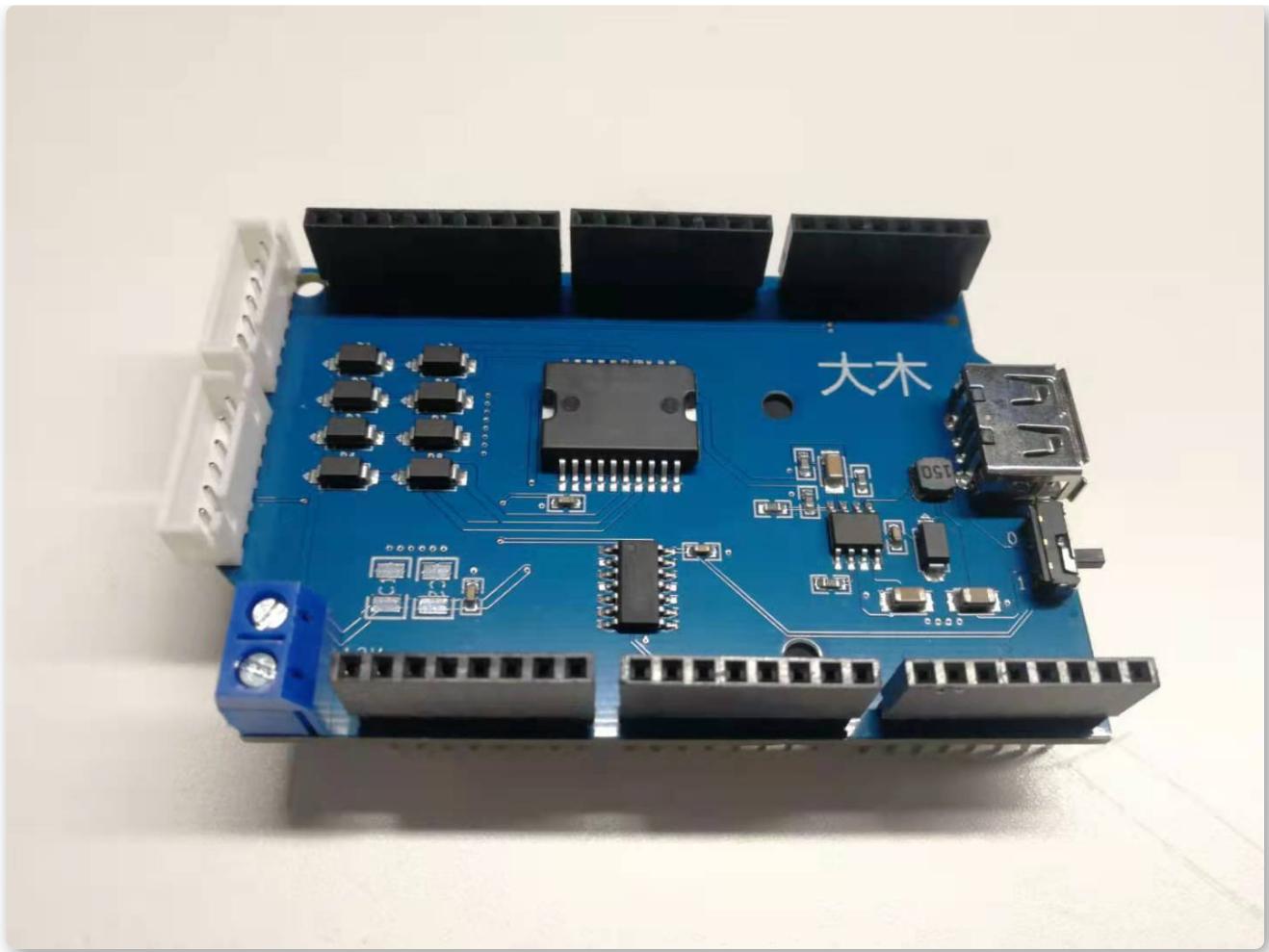
C1: 信号线

VCC:编码器电源+

M2: 电机电源-(和M1对调可以正反转)

2. 电机驱动板

电机驱动板可选型号较多，比如:TB6612、L298N、L298P...但是这些电机驱动板与电机相连时，需要使用杜邦线，接线会显得凌乱，本节会采用一款基于L298P优化的电机驱动板，该驱动板可以使用端子线直接连接电机，接线更规整、美观。



端子线母头对应的引脚(自上而下)

母头1: 4、地线、21、20、5V输入、5

母头2: 7、地线、18、19、5V输入、6

PS: 电机驱动板使用时, 需要打开USB接口处的电源开关。

3.准备工作

组装底盘: 集成电池、Arduino、电机驱动板与电机



先安装Arduino、安装电机(接端子线)与万向轮，将电机驱动板与Arduino集成；

然后将电池的正负极分别接入电机驱动模块的12V与GND(注意:正负极不可接反，12V接红线，GND接黑线)；

最后将电机通过端子线与驱动板相连。

8.3.2 电机基本控制实现

在ROS智能车中，控制车辆的前进、后退以及速度调节，那么就涉及到电机的转向与转速控制，本节主要就是介绍相关知识点。

需求:控制单个电机转动，先控制电机以某个速率正向转动N秒，再让电机停止N秒，再控制电机以某个速率逆向转动N秒，最后让电机停止N秒，如此循环。

实现流程:

1. 编写Arduino程序，setup中设置引脚模式，loop中控制电机运动；

2. 上传并查看运行结果。

1. 编码

前提知识点：

1. 左电机的M1与M2对应的是引脚4(DIRA)和引脚5(PWMA)，引脚4控制转向，引脚5输出PWM。右电机的M1与M2对应的是引脚6(PWMB)和引脚7(DIRB)，引脚7控制转向，引脚6输出PWM。
2. 可以通过PWM控制电机转速。

代码：

```
1  /*
2   * 电机转动控制
3   * 1. 定义接线中电机对应的引脚
4   * 2. setup 中设置引脚为输出模式
5   * 3. loop中控制电机转动
6   *
7   */
8
9 int DIRA = 4;
10 int PWMA = 5;
11
12 void setup() {
13     //两个引脚都设置为 OUTPUT
14     pinMode(DIRA,OUTPUT);
15     pinMode(PWMA,OUTPUT);
16 }
17
18 void loop() {
19     //先正向转动3秒
20     digitalWrite(DIRA,HIGH);
21     analogWrite(PWMA,100);
22     delay(3000);
23     //停止3秒
24     digitalWrite(DIRA,HIGH);
25     analogWrite(PWMA,0);
26     delay(3000);
```

```

27 //再反向转动3秒
28 digitalWrite(DIRA,LOW);
29 analogWrite(PWMA,100);
30 delay(3000);
31 //停止3秒
32 digitalWrite(DIRA,LOW);
33 analogWrite(PWMA,0);
34 delay(3000);

35
36 /*
37 * 注意：
38 * 1.可以通过将DIRA设置为HIGH或LOW来控制电机转向，但是
39 * 哪个标志位正转或反转需要根据需求判断，转向是相对的。
40 * 2.PWM的取值为 [0,255]，该值可自己设置。
41 *
42 */
43 }

```

2.运行

程序上传到Arduino上，如无异常，电机开始转动，转动结果与需求描述类似。

8.3.3 电机测速01_理论

测速实现是调速实现的前提，本节主要介绍AB相增量式编码器测速原理。

1.概念

百度百科关于编码器介绍如下：

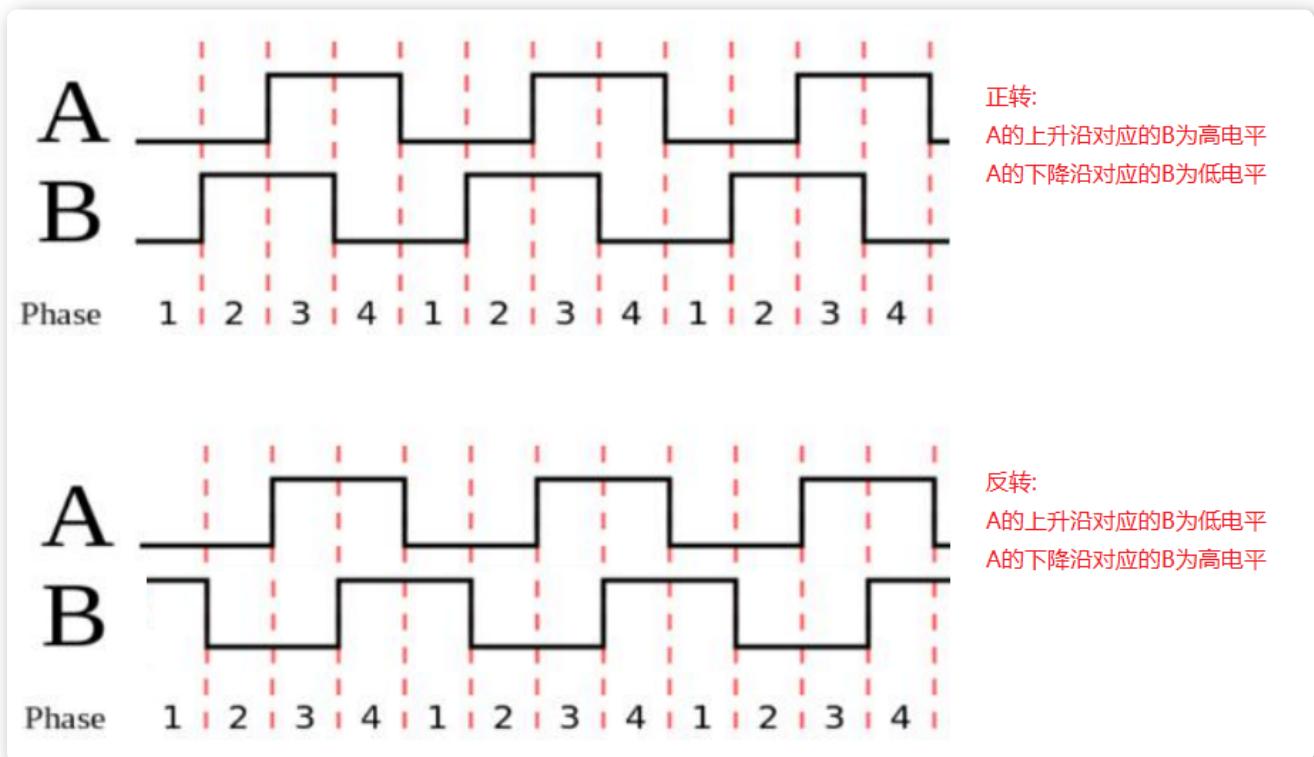
编码器（encoder）是将信号（如比特流）或数据进行编制、转换为可用以通讯、传输和存储的信号形式的设备。编码器把角位移或直线位移转换成电信号，前者称为码盘，后者称为码尺。按照读出方式编码器可以分为接触式和非接触式两种；按照工作原理编码器可分为增量式和绝对式两类。增量式编码器是将位移转换成周期性的电信号，再把这个电信号转变成计数脉冲，用脉冲

的个数表示位移的大小。绝对式编码器的每一个位置对应一个确定的数字码，因此它的示值只与测量的起始和终止位置有关，而与测量的中间过程无关。

2. 测速原理

关于编码器相关概念简单了解即可，在此需要着重介绍的是 AB 相增量式编码器测速原理：

AB相编码器主要构成为A相与B相，每一相每转过单位的角度就发出一个脉冲信号(一圈可以发出N个脉冲信号)，A相、B相为相互延迟1/4周期的脉冲输出，根据延迟关系可以区别正反转，而且通过取A相、B相的上升和下降沿可以进行单频或2倍频或4倍频测速。



3. 测速举例

假设编码器旋转1圈输出11个脉冲，减速比为 90。伪代码如下：

单频计数:

```

1 //设置一个计数器
2 int count = 0;
3 //当A为上升沿时
4 if(B为高电平){
5     count++;
6 }else {
7     count--;
8 }
9 //....
10 //速度=单位时间内统计的脉冲的个数 / (11*90) / 单位时间

```

2倍频计数:

```

1 //设置一个计数器
2 int count = 0;
3 //当A为上升沿时
4 if(B为高电平){
5     count++;
6 }else {
7     count--;
8 }
9 //当A为下降沿时
10 if(B为低电平){
11     count++;
12 }else {
13     count--;
14 }
15
16 //....
17 //速度=单位时间内统计的脉冲的个数 / (11*2*90) / 单位时间

```

4倍频计数:

```

1 //设置一个计数器
2 int count = 0;
3 //当A为上升沿时

```

```

4  if(B为高电平){
5      count++;
6  }else {
7      count--;
8 }
9 //当A为下降沿时
10 if(B为低电平){
11     count++;
12 }else {
13     count--;
14 }
15 //当B为上升沿时
16 if(A为低电平){
17     count++;
18 } else {
19     count--;
20 }
21 //当B为下降沿时
22 if(A为高电平){
23     count++;
24 } else {
25     count--;
26 }
27 //....
28 //速度=单位时间内统计的脉冲的个数 / (11*4*90) / 单位时间

```

8.3.4 电机测速02_实现

需求:统计并输出电机转速。

思路:先统计单位时间内以单频或2倍频或4倍频的方式统计脉冲数，再除以一圈对应的脉冲数，最后再除以时间所得即为电机转速。

核心:计数时，需要在A相或B相的上升沿或下降沿触发时，实现计数，在此需要使用中断引脚与中断函数。

Arduino Mega 2560 的中断引脚:2 (interrupt 0), 3 (interrupt 1), 18 (interrupt 5), 19 (interrupt 4), 20 (interrupt 3), 21 (interrupt 2)

实现流程:

1. 编写Arduino程序先实现脉冲数统计;
2. 编写Arduino程序再实现转速计算相关实现;
3. 上传到Arduino并测试。

1. 编码实现脉冲统计

核心知识点:attachInterrupt()函数(请参考 8.2.2 介绍)。

代码:

```

1  /*
2   * 测速实现:
3   *   阶段1:脉冲数统计
4   *   阶段2:速度计算
5   *
6   * 阶段1:
7   *   1. 定义所使用的中断引脚,以及计数器(使用 volatile 修饰)
8   *   2. setup 中设置波特率, 将引脚设置为输入模式
9   *   3. 使用 attachInterrupt() 函数为引脚添加中断出发时机
10  *      以及中断函数
11  *   4. 中断函数编写计算算法, 并打印
12  *      A. 单频统计只需要统计单相上升沿或下降沿
13  *      B. 2倍频统计需要统计单相的上升沿和下降沿
14  *      C. 4倍频统计需要统计两相的上升沿和下降沿
15  *
16  *
17  */
18 int motor_A = 21; //中端口是2
19 int motor_B = 20; //中断口是3
20 volatile int count = 0; //如果是正转, 那么每计数一次自增
21   1, 如果是反转, 那么每计数一次自减1

```

```
22
23 void count_A(){
24     //单频计数实现
25     //手动旋转电机一圈，输出结果为 一圈脉冲数 * 减速比
26     /*if(digitalRead(motor_A) == HIGH){
27
28         if(digitalRead(motor_B) == LOW){//A 高 B 低
29             count++;
30         } else{//A 高 B 高
31             count--;
32         }
33
34
35 }*/
```

```
36
37 //2倍频计数实现
38 //手动旋转电机一圈，输出结果为 一圈脉冲数 * 减速比 * 2
39 if(digitalRead(motor_A) == HIGH){
40
41     if(digitalRead(motor_B) == HIGH){//A 高 B 高
42         count++;
43     } else{//A 高 B 低
44         count--;
45     }
46
47
48 } else {
49
50     if(digitalRead(motor_B) == LOW){//A 低 B 低
51         count++;
52     } else{//A 低 B 高
53         count--;
54     }
55
56 }
57
58 }
59
60 //与A实现类似
```

```

61 //4倍频计数实现
62 //手动旋转电机一圈，输出结果为 一圈脉冲数 * 减速比 * 4
63 void count_B(){
64     if(digitalRead(motor_B) == HIGH){
65
66         if(digitalRead(motor_A) == LOW){//B 高 A 低
67             count++;
68         } else{//B 高 A 高
69             count--;
70         }
71
72     } else {
73
74         if(digitalRead(motor_A) == HIGH){//B 低 A 高
75             count++;
76         } else{//B 低 A 低
77             count--;
78         }
79     }
80
81 }
82
83 }
84
85 void setup() {
86     Serial.begin(57600); //设置波特率
87     pinMode(motor_A, INPUT);
88     pinMode(motor_B, INPUT);
89     attachInterrupt(2, count_A, CHANGE); //当电平发生改变
时触发中断函数
90     //四倍频统计需要为B相也添加中断
91     attachInterrupt(3, count_B, CHANGE);
92 }
93
94
95 void loop() {
96     //测试计数器输出
97     delay(2000);
98     Serial.println(count);

```

```
99
100 }
```

2. 转速计算

思路:需要定义一个开始时间(用于记录每个测速周期的开始时刻), 还需要定义一个时间区间(比如50毫秒), 时时获取当前时刻, 当当前时刻 - 上传结束时刻 \geq 时间区间时, 就获取当前计数并根据测速公式计算时时速度, 计算完毕, 计数器归零, 重置开始时间

核心知识点:当使用中断函数中的变量时, 需要先禁止中断 **noInterrupts()**, 调用完毕, 再重启中断**interrupts()**(关于**noInterrupts**与**interrupts**请参考 8.2.2 介绍)。

代码(核心):

2中代码除了 **loop** 实现, 无需修改。

```
1 int reduction = 90; //减速比, 根据电机参数设置, 比如 15
| 30 | 60
2 int pulse = 11; //编码器旋转一圈产生的脉冲数该值需要参考
商家电机参数
3 int per_round = pulse * reduction * 4; //车轮旋转一
圈产生的脉冲数
4 long start_time = millis(); //一个计算周期的开始时刻, 初
始值为 millis();
5 long interval_time = 50; //一个计算周期 50ms
6 double current_vel;
7
8 //获取当前转速的函数
9 void get_current_vel(){
10     long right_now = millis();
11     long past_time = right_now - start_time; //计算逝去
的时间
12     if(past_time >= interval_time){ //如果逝去时间大于等
于一个计算周期
13         //1. 禁止中断
14         noInterrupts();
```

```

15      //2.计算转速 转速单位可以是秒，也可以是分钟... 自定义
16      //即可
17      current_vel = (double)count / per_round /
18      past_time * 1000 * 60;
19      //3.重置计数器
20      count = 0;
21      //4.重置开始时间
22      start_time = right_now;
23      //5.重启中断
24      interrupts();
25
26  }
27 }
28
29 void loop() {
30
31     delay(10);
32     get_current_vel();
33
34 }
```

3. 测试

将代码上传至Arduino，打开串口监视器，手动旋转电机，可以查看到转速信息。

8.3.5 电机调速01_PID控制理论



场景：

速度信息可以以m/s为单位，或者也可以转换成转速 r/s，而电机的转速是由PWM脉冲宽度来控制的，如何根据速度信息量化成合适的PWM值呢？

比如:现有一辆行驶中的无人车,要求将车速调整至100KM/h,那么应该如何向电机输出PWM值?或换言之,如何控制油门?

调速实现策略由多种, PID其中较为常用。

PID简介

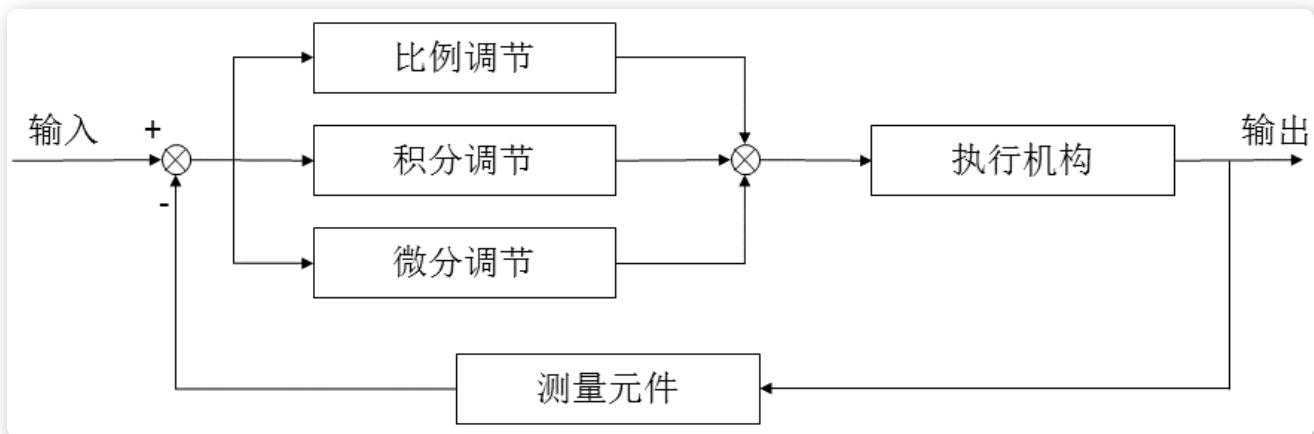
PID算法是一种经典、简单、高效的动态速度调节方式, P代表比例, I代表积分, D代表微分。

PID公式如下:

$$\mu(t) = K_P e(t) + K_I \int_0^t e(t) dt + K_D \frac{de(t)}{dt}$$

- $e(t)$ 作为 PID 控制的输入;
- $u(t)$ 作为 PID 控制器的输出和被控对象的输入;
- K_p 控制器的比例系数;
- K_i 控制器的积分时间, 也称积分系数;
- K_d 控制器的微分时间, 也称微分系数。

上述公式稍显晦涩, PID控制原理框架图更有助于理解:



1.P

如果实现上述场景中的车速控制，一种简单的实现方式是：确定目标速度，获取当前速度，使用 $(\text{目标速度}-\text{当前速度}) \times \text{某一系数}$ 计算结果为输出的PWM，再获取当前速度，使用 $(\text{目标速度}-\text{当前速度}) \times \text{某一系数}$ 计算结果为输出的PWM并输出...如此循环在上述模型中，调速实现是一个闭环，每一次循环都会根据当前时速与目标时速的差值，再乘以以固定系数，计算出需要输出的PWM值，这其中的系数，称之为比例。

2.I

上述模型算法中，最终速度与预期速度存在稳态误差，这意味着最终结果可能永远无法达成预期，解决的方法就是使用积分I。每次调速时，输出的PWM还要累加根据积分I计算的结果，以消除静态误差。

3.D

当I值设置的过大时，可能会出先"超速"的情况，超速之后可能需要多次调整，产生系统震荡，解决这种情况可以使用D微分，当速度越是接近目标速度时，D就会越施加反方向力，减弱P的控制，起到类似"阻尼"的作用。通过D的使用可以减小系统震荡。

综上，PID闭环控制实现是结合了比例、积分和微分的一种控制机制，通过P可以以比例的方式计算输出，通过I可以消除稳态误差，通过D可以减小系统震荡，三者相结合，最终是要快速、精准且稳定的达成预期结果，而要实现该结果，还需要对这三个数值反复测试、调整...下一节将介绍在 Arduino 中 PID控制的具体实现，其中就包括PID库的调用，以及PID调试的具体方式。

8.3.6 电机调速02_PID控制实现

了解了PID原理以及计算公式之后，我们可以在程序中自实现PID相关算法实现，不过，在Arduino中该算法已经被封装了，直接整合调用即可，从而提高程序的安全性与开发效率。该库是:Arduino-PID-Library，接下来通过一个案例演示该库的使用。

需求:通过PID控制电机转速，预期转速为 80r/m。

实现流程:

1. 添加Arduino-PID-Library；
2. 编写Arduino程序直接调用相关实现；
3. 使用串口绘图器调试PID值。

1.添加Arduino-PID-Library

首先在 GitHub 下载 PID 库: `git clone https://github.com/br3ttb/Arduino-PID-Library`

然后将该文件夹移动到 arduino 的 libraries 下: `sudo cp -r Arduino-PID-Library /home/用户名/Arduino/libraries`

还要重命名文件夹: `sudo mv Arduino-PID-Library ArduinoPIDLibrary`

最后重启 ArduinoIDE

2.编码

PID调速中，测速是实现闭环的关键实现，所以需要复制之前的电机控制代码以及测速代码。

完整代码实现:

```

1  /*
2   * PID 调速实现:
3   * 1. 代码准备, 复制并修改电机控制以及测速代码
4   * 2. 包含PID头文件
5   * 3. 创建PID对象
6   * 4. 在setup中启用自动调试
7   * 5. 调试并更新PWM
8   *
9   */
10
11 #include <PID_v1.h>
12

```

```

13 int DIRA = 4;
14 int PWMA = 5;
15
16 int motor_A = 21; //中端口是2
17 int motor_B = 20; //中断口是3
18 volatile int count = 0; //如果是正转，那么每计数一次自增
19   1，如果是反转，那么每计数一次自减1
20
21 void count_A(){
22   //单频计数实现
23   //手动旋转电机一圈，输出结果为 一圈脉冲数 * 减速比
24   /*if(digitalRead(motor_A) == HIGH){
25
26     if(digitalRead(motor_B) == LOW){ //A 高 B 低
27       count++;
28     } else { //A 高 B 高
29       count--;
30     }
31
32
33   }*/
34
35   //2倍频计数实现
36   //手动旋转电机一圈，输出结果为 一圈脉冲数 * 减速比 * 2
37   if(digitalRead(motor_A) == HIGH){
38
39     if(digitalRead(motor_B) == HIGH){ //A 高 B 高
40       count++;
41     } else { //A 高 B 低
42       count--;
43     }
44
45
46   } else {
47
48     if(digitalRead(motor_B) == LOW){ //A 低 B 低
49       count++;
50     } else { //A 低 B 高

```

```

51         count--;
52     }
53
54 }
55
56 }
57
58 //与A实现类似
59 //4倍频计数实现
60 //手动旋转电机一圈，输出结果为 一圈脉冲数 * 减速比 * 4
61 void count_B(){
62     if(digitalRead(motor_B) == HIGH){
63
64         if(digitalRead(motor_A) == LOW){//B 高 A 低
65             count++;
66         } else{//B 高 A 高
67             count--;
68         }
69
70     } else {
71
73         if(digitalRead(motor_A) == HIGH){//B 低 A 高
74             count++;
75         } else{//B 低 A 低
76             count--;
77         }
78
79     }
80
81 }
82
83
84 int reduction = 90;//减速比，根据电机参数设置，比如 15
| 30 | 60
85 int pulse = 11; //编码器旋转一圈产生的脉冲数该值需要参考
商家电机参数
86 int per_round = pulse * reduction * 4;//车轮旋转一
圈产生的脉冲数

```

```

87 long start_time = millis(); //一个计算周期的开始时刻,
    初始值为 millis();
88 long interval_time = 50; //一个计算周期 50ms
89 double current_vel;
90
91 //获取当前转速的函数
92 void get_current_vel(){
93     long right_now = millis();
94     long past_time = right_now - start_time; //计算逝
    去的时间
95     if(past_time >= interval_time){ //如果逝去时间大于等
    于一个计算周期
96         //1.禁止中断
97         noInterrupts();
98         //2.计算转速 转速单位可以是秒, 也可以是分钟... 自定义
    即可
99         current_vel = (double)count / per_round /
    past_time * 1000 * 60;
100        //3.重置计数器
101        count = 0;
102        //4.重置开始时间
103        start_time = right_now;
104        //5.重启中断
105        interrupts();
106
107        Serial.println(current_vel);
108
109    }
110 }
111
112 //-----PID-----
113 //创建 PID 对象
114 //1.当前转速 2.计算输出的pwm 3.目标转速 4.kp 5.ki 6.kd
    7.当输入与目标值出现偏差时, 向哪个方向控制
115 double pwm; //电机驱动的PWM值
116 double target = 80;
117 double kp=1.5, ki=3.0, kd=0.1;

```

```

118 PID
    pid(&current_vel,&pwm,&target,kp,ki,kd,DIRECT);
119
120 //速度更新函数
121 void update_vel(){
122     //获取当前速度
123     get_current_vel();
124     pid.Compute(); //计算需要输出的PWM
125     digitalWrite(DIRA,HIGH);
126     analogWrite(PWMA,pwm);
127
128 }
129
130 void setup() {
131     Serial.begin(57600); //设置波特率
132     pinMode(18,INPUT);
133     pinMode(19,INPUT);
134     //两个电机驱动引脚都设置为 OUTPUT
135     pinMode(DIRA,OUTPUT);
136     pinMode(PWMA,OUTPUT);
137
138     attachInterrupt(2,count_A,CHANGE); //当电平发生改变
时触发中断函数
139     //四倍频统计需要为B相也添加中断
140     attachInterrupt(3,count_B,CHANGE);
141
142     pid.SetMode(AUTOMATIC);
143 }
144
145
146
147 void loop() {
148     delay(10);
149     update_vel();
150
151 }

```

核心代码解释:

1.包含PID头文件

```
1 #include <PID_v1.h>
```

2.创建PID对象

```
1 //创建 PID 对象
2 //1.当前转速 2.计算输出的pwm 3.目标转速 4.kp 5.ki 6.kd
3   7.当输入与目标值出现偏差时, 向哪个方向控制
4 double pwm;//电机驱动的PWM值
5 double target = 120;
6 double kp=1.5, ki=3.0, kd=0.1;
7 PID pid(&current_vel,&pwm,&target,kp,ki,kd,DIRECT);
```

3.setup中启用PID自动控制

```
1 pid.SetMode(AUTOMATIC);
```

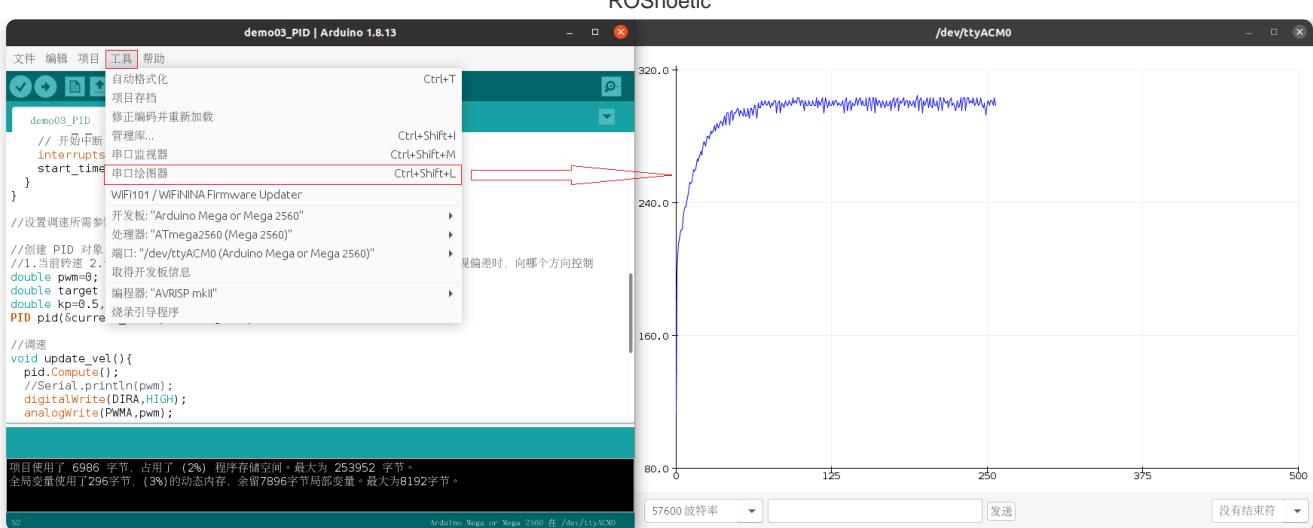
4.计算输出值

```
1 pid.Compute();
```

4.调试

PID控制的最终预期结果，是要快速、精准、稳定的达成预期结果，P主要用于控制响应速度，I主要用于控制精度，D主要用于减小震荡增强系统稳定性，三者的取值是需要反复调试的，调试过程中需要查看系统的响应曲线，根据响应曲线以确定合适的PID值。

在 Arduino 中响应曲线的查看可以借助于 `Serial.println()` 将结果输出，然后再选择菜单栏的工具下串口绘图器以图形化的方式显示响应结果：



PID调试技巧:

参数整定找最佳，从小到大顺序查

先是比例后积分，最后再专把微分加

曲线振属荡很频繁，比例度盘要放大

曲线漂浮绕大湾，比例度盘往小扳

曲线偏离回复慢，积分时间往下降

曲线波动周期长，积分时间再加长

曲线振荡频率快，先把微分降下来

动差大来波动慢。微分时间应加长

理想曲线两个波，前高后低4比1

一看二调多分析，调节质量不会低

8.4 机器人平台设计之底盘实现

在ros中还提供了一个已经封装了的模块: `ros_arduino_bridge`，该模块由下位机驱动和上位机控制两部分组成，通过该模块可以更为快捷、方便的实现自己的机器人平台。本节介绍的主要内容如下：

- `ros_arduino_bridge` 的架构；
- 下位机端 Arduino 程序修改。

注意:官方提供的案例所需硬件国内不易购买,需要修改源码适配当前硬件。

8.4.1 底盘实现_概述

1.ros_arduino_bridge 简介

该功能包包含Arduino库和用来控制Arduino的ROS驱动包,它旨在成为在ROS下运行Arduino控制的机器人的完整解决方案。

其中当前主要关注的是:功能包集中一个兼容不同驱动的机器人的基本控制器 (base controller) ,它可以接收ROS Twist类型的消息,可以发布里程计数据。

特点:

- 可以直接支持ping声呐和Sharp红外线传感器;
- 也可以从通用的模拟和数字信号的传感器读取数据;
- 可以控制数字信号的输出;
- 支持PWM伺服机;
- 如果使用所要求的硬件的话,可以配置基本功能;
- 如果你的Arduino编程基础好的话,并且具有python基础的话,你就很自由的改动代码来满足你的硬件要求。

注意:

- 官方提供的部分硬件不易采购,需要修改下位机程序,以适配当前硬件。

系统要求:

- 如果只是安装调试下位机,那么不必安装ROS系统,只要有arduino 开发环境即可;
- 而上位机调试,适用于 ROS Indigo 及更高版本,但是暂不支持最新版本 noetic,所以上位机需要使用其它版本的ROS(建议 melodic);

下载:

进入ROS工作空间的src目录,输入命令:

```
1 git clone
  https://github.com/hbrobotics/ros_arduino_bridge.git
```

2.ros_arduino_bridge 架构

文件结构说明

```
1 ┌── ros_arduino_bridge          #
  metapackage (元功能包)
2 |   ┌── CMakeLists.txt
3 |   └── package.xml
4 └── ros_arduino_firmware      #固件
  包, 更新到Arduino
5   ┌── CMakeLists.txt
6   ┌── package.xml
7   └── src
8     └── libraries             #库
  目录
9     ┌── MegaRobogaiaPololu      #针
  对Pololu电机控制器, MegaRobogaia编码器的头文件定义
10    |   ┌── commands.h          #定
  义命令头文件
11    |   ┌── diff_controller.h    #差
  分轮PID控制头文件
12    |   ┌── MegaRobogaiaPololu.ino
  #PID实现文件
13    |   ┌── sensors.h           #传
  感器相关实现, 超声波测距, Ping函数
14    |   ┌── servos.h            #伺
  服器头文件
15    └── ROSArduinoBridge
  #Arduino相关库定义
16      ┌── commands.h          #定
  义命令
17      ┌── diff_controller.h    #差
  分轮PID控制头文件
18      ┌── encoder_driver.h      #编
  码器驱动头文件
```

```

19 |           └── encoder_driver.ino      #编
  码器驱动实现, 读取编码器数据, 重置编码器等
20 |           └── motor_driver.h        #电
  机驱动头文件
21 |           └── motor_driver.ino      #电
  机驱动实现, 初始化控制器, 设置速度
22 |           └── ROSArduinoBridge.ino  #核
  心功能实现, 程序入口
23 |           └── sensors.h          #传
  感器头文件及实现
24 |           └── servos.h           #伺
  服器头文件, 定义插脚, 类
25 |           └── servos.ino         #伺
  服器实现
26 └── ros_arduino_msgs                 #消息
  定义包
27 |   └── CMakeLists.txt
28 |   └── msg                         #定
  义消息
29 |   |   └── AnalogFloat.msg        #定
  义模拟IO浮点消息
30 |   |   └── Analog.msg           #定
  义模拟IO数字消息
31 |   |   └── ArduinoConstants.msg  #定
  义常量消息
32 |   |   └── Digital.msg          #定
  义数字IO消息
33 |   |   └── SensorState.msg      #定
  义传感器状态消息
34 |   └── package.xml
35 |   └── srv                         #定
  义服务
36 |   └── AnalogRead.srv            #模
  拟IO输入
37 |   └── AnalogWrite.srv          #模
  拟IO输出
38 |   └── DigitalRead.srv          #数
  字IO输入

```

```

39 |     └── DigitalSetDirection.srv      #数字
  IO设置方向
40 |     └── DigitalWrite.srv            #数
  字IO输入
41 |     └── ServoRead.srv             #伺
  服电机输入
42 |         └── ServoWrite.srv        #伺
  服电机输出
43 └── ros_arduino_python            #ROS
  相关的Python包, 用于上位机, 树莓派等开发板或电脑等。
44     └── CMakeLists.txt
45     └── config                    #配置
  目录
46     |     └── arduino_params.yaml  #定
  义相关参数, 端口, rate, PID, sensors等默认参数。由
  arduino.launch调用
47     └── launch
48     |     └── arduino.launch      #启
  动文件
49     └── nodes
50     |     └── arduino_node.py
  #python文件, 实际处理节点, 由arduino.launch调用, 即可单独
  调用。
51     └── package.xml
52     └── setup.py
53     └── src
  #Python类包目录
54         └── ros_arduino_python
55             └── arduino_driver.py
  #Arduino驱动类
56             └── arduino_sensors.py
  #Arduino传感器类
57             └── base_controller.py      #基本
  控制类, 订阅cmd_vel话题, 发布odom话题
58             └── __init__.py          #类包
  默认空文件

```

上述目录结构虽然复杂, 但是关注的只有两大部分:

- ros_arduino_bridge/ros_arduino_firmware/src/libraries/ROSArduinoBridge
- ros_arduino_bridge/ros_arduino_python/config/arduino_params.yaml

前者是Arduino端的固件包实现，需要修改并上传至Arduino电路板；

后者是ROS端的一个配置文件，相关驱动已经封装完毕，我们只需要修改配置信息即可。

整体而言，借助于 ros_arduino_bridge 可以大大提高我们的开发效率。

3.案例实现

基于ros_arduino_bridge的底盘实现具体步骤如下：

- 了解并修改Arduino端程序主入口ROSArduinoBridge.ino 文件；
- Arduino端添加编码器驱动；
- Arduino端添加电机驱动模块；
- Arduino端实现PID调试；

8.4.2 底盘实现_01Arduino端入口

ros_arduino_bridge/ros_arduino_firmware/src/libraries/ROSArduinoBridge 下的 RosArduinoBridge.ino 是 Arduino 端程序的主入口，

源文件(添加中文注释)内容如下：

```

1  /*****
2   * ROSArduinoBridge
3   * 可以通过一组简单的串口命令来控制差分机器人并接收回传的
4   * 传感器与里程计
5   * 数据， 默认使用的是 Arduino Mega + Pololu 电机驱动模
6   * 块， 如果使用其他的
7   * 编码器或电机驱动需要重写 readEncoder() 与
8   * setMotorSpeed() 函数
9   * A set of simple serial commands to control a
10  * differential drive

```

7 robot and receive back sensor and odometry
8 data. Default
9 configuration assumes use of an Arduino Mega +
10 Pololu motor
11 controller shield + Robogaia Mega Encoder
12 shield. Edit the
13 readEncoder() and setMotorSpeed() wrapper
14 functions if using
15 different motor controller or encoder method.
16 Created for the Pi Robot Project:
17 <http://www.pirobot.org>
18 and the Home Brew Robotics Club (HBRC):
19 <http://hbrobotics.org>
20 Authors: Patrick Goebel, James Nugen
21
22 Inspired and modeled after the ArbotiX driver
23 by Michael Ferguson
24
25 Software License Agreement (BSD License)
26
27 Copyright (c) 2012, Patrick Goebel.
28 All rights reserved.
29
30 Redistribution and use in source and binary
31 forms, with or without
32 modification, are permitted provided that the
33 following conditions
34 are met:
35
36 * Redistributions of source code must retain
37 the above copyright
38 notice, this list of conditions and the
39 following disclaimer.
40 * Redistributions in binary form must
41 reproduce the above
42 copyright notice, this list of conditions
43 and the following

33 disclaimer in the documentation and/or
34 other materials provided
35 with the distribution.

36 THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT
37 HOLDERS AND CONTRIBUTORS
38 "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,
39 INCLUDING, BUT NOT
40 LIMITED TO, THE IMPLIED WARRANTIES OF
41 MERCHANTABILITY AND FITNESS
42 FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO
43 EVENT SHALL THE
44 COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR
45 ANY DIRECT, INDIRECT,
46 INCIDENTAL, SPECIAL, EXEMPLARY, OR
47 CONSEQUENTIAL DAMAGES (INCLUDING,
48 BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE
49 GOODS OR SERVICES;
50 LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
51 INTERRUPTION) HOWEVER
52 CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER
53 IN CONTRACT, STRICT
54 LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
55 OTHERWISE) ARISING IN
56 ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN
57 IF ADVISED OF THE
58 * POSSIBILITY OF SUCH DAMAGE.

59 ******/

60 //是否启用基座控制器

61 // #define USE_BASE // Enable the base
62 controller code

63 #undef USE_BASE // Disable the base controller
64 code

65

66 /* Define the motor controller and encoder library
67 you are using */

68 //启用基座控制器需要设置的电机驱动以及编码器驱动

69 #ifdef USE_BASE

```
56  /* The Pololu VNH5019 dual motor driver shield
57  */
58
59  /* The Pololu MC33926 dual motor driver shield
60  */
61
62  /* The RoboGaia encoder shield */
63  #define ROBOGAIA
64
65  /* Encoders directly attached to Arduino board
66  */
67
68  /* L298 Motor driver*/
69  // #define L298_MOTOR_DRIVER
70 #endif
71
72 //是否启用舵机
73 #define USE_SERVOS // Enable use of PWM servos as
74 // defined in servos.h
75 // #undef USE_SERVOS // Disable use of PWM
76 // servos
77
78 /* Serial port baud rate */
79 //波特率
80 #define BAUDRATE 57600
81
82 /* Maximum PWM signal */
83 //最大PWM值
84 #define MAX_PWM 255
85
86 //根据Arduino型号来包含对应的头文件
87 #if defined(ARDUINO) && ARDUINO >= 100
88 #include "Arduino.h"
89 #else
90 #include "WProgram.h"
91 #endif
```

```

90
91 /* Include definition of serial commands */
92 //串口命令
93 #include "commands.h"
94
95 /* Sensor functions */
96 //传感器文件
97 #include "sensors.h"
98
99 /* Include servo support if required */
100 //如果启用舵机，需要包含的头文件
101 #ifdef USE_SERVOS
102     #include <Servo.h>
103     #include "servos.h"
104 #endif
105
106 //如果启用基座控制器需要包含的头文件
107 #ifdef USE_BASE
108     /* Motor driver function definitions */
109     #include "motor_driver.h" //电机驱动
110
111     /* Encoder driver function definitions */
112     #include "encoder_driver.h" //编码器驱动
113
114     /* PID parameters and functions */
115     #include "diff_controller.h" //PID调速
116
117     /* Run the PID loop at 30 times per second */
118     #define PID_RATE 30 // Hz 调速频率
119
120     /* Convert the rate into an interval */
121     const int PID_INTERVAL = 1000 / PID_RATE; //调速
122     周期
123
124     /* Track the next time we make a PID calculation */
125     unsigned long nextPID = PID_INTERVAL;

```

```

126  /* Stop the robot if it hasn't received a
127  movement command
128  in this number of milliseconds */
129  #define AUTO_STOP_INTERVAL 2000 //自动结束时间(可
130  按需修改)
131
132 /* Variable initialization */
133
134 // A pair of variables to help parse serial
135 // commands (thanks Fergs)
136 int arg = 0;
137 int index = 0;
138
139 // Variable to hold an input character
140 char chr;
141
142 // Variable to hold the current single-character
143 // command
144 char cmd;
145
146 // Character arrays to hold the first and second
147 // arguments
148 char argv1[16];
149 char argv2[16];
150
151
152 // The arguments converted to integers
153 long arg1;
154 long arg2;
155
156
157 /* Clear the current command parameters */
158 //重置命令
159 void resetCommand() {
160     cmd = NULL;
161     memset(argv1, 0, sizeof(argv1));
162     memset(argv2, 0, sizeof(argv2));
163     arg1 = 0;
164     arg2 = 0;

```

```
160     arg = 0;
161     index = 0;
162 }
163
164 /* Run a command. Commands are defined in
   commands.h */
165 //执行串口命令
166 int runCommand() {
167     int i = 0;
168     char *p = argv1;
169     char *str;
170     int pid_args[4];
171     arg1 = atoi(argv1);
172     arg2 = atoi(argv2);
173
174     switch(cmd) {
175     case GET_BAUDRATE:
176         Serial.println(BAUDRATE);
177         break;
178     case ANALOG_READ:
179         Serial.println(analogRead(arg1));
180         break;
181     case DIGITAL_READ:
182         Serial.println(digitalRead(arg1));
183         break;
184     case ANALOG_WRITE:
185         analogWrite(arg1, arg2);
186         Serial.println("OK");
187         break;
188     case DIGITAL_WRITE:
189         if (arg2 == 0) digitalWrite(arg1, LOW);
190         else if (arg2 == 1) digitalWrite(arg1, HIGH);
191         Serial.println("OK");
192         break;
193     case PIN_MODE:
194         if (arg2 == 0) pinMode(arg1, INPUT);
195         else if (arg2 == 1) pinMode(arg1, OUTPUT);
196         Serial.println("OK");
197         break;
```

```

198  case PING:
199      Serial.println(Ping(arg1));
200      break;
201 #ifdef USE_SERVOS
202     case SERVO_WRITE:
203         servos[arg1].setTargetPosition(arg2);
204         Serial.println("OK");
205         break;
206     case SERVO_READ:
207
208         Serial.println(servos[arg1].getServo().read());
209         break;
210 #endif
211 #ifdef USE_BASE
212     case READ_ENCODERS:
213         Serial.print(readEncoder(LEFT));
214         Serial.print(" ");
215         Serial.println(readEncoder(RIGHT));
216         break;
217     case RESET_ENCODERS:
218         resetEncoders();
219         resetPID();
220         Serial.println("OK");
221         break;
222     case MOTOR_SPEEDS: //传入电机控制命令
223         /* Reset the auto stop timer */
224         lastMotorCommand = millis();
225         if (arg1 == 0 && arg2 == 0) {
226             setMotorSpeeds(0, 0);
227             resetPID();
228             moving = 0;
229         }
230         else moving = 1;
231         leftPID.TargetTicksPerFrame = arg1;
232         rightPID.TargetTicksPerFrame = arg2;
233         Serial.println("OK");
234         break;
235     case UPDATE_PID:

```

```

236     while ((str = strtok_r(p, ":", &p)) != '\0') {
237         pid_args[i] = atoi(str);
238         i++;
239     }
240     Kp = pid_args[0];
241     Kd = pid_args[1];
242     Ki = pid_args[2];
243     Ko = pid_args[3];
244     Serial.println("OK");
245     break;
246 #endif
247 default:
248     Serial.println("Invalid Command");
249     break;
250 }
251 }
252
253 /* Setup function--runs once at startup. */
254 void setup() {
255     Serial.begin(BAUDRATE);
256
257     // Initialize the motor controller if used
258 #ifdef USE_BASE
259     #ifdef ARDUINO_ENC_COUNTER
260         //set as inputs
261         DDRD &= ~(1<<LEFT_ENC_PIN_A);
262         DDRD &= ~(1<<LEFT_ENC_PIN_B);
263         DDRC &= ~(1<<RIGHT_ENC_PIN_A);
264         DDRC &= ~(1<<RIGHT_ENC_PIN_B);
265
266         //enable pull up resistors
267         PORTD |= (1<<LEFT_ENC_PIN_A);
268         PORTD |= (1<<LEFT_ENC_PIN_B);
269         PORTC |= (1<<RIGHT_ENC_PIN_A);
270         PORTC |= (1<<RIGHT_ENC_PIN_B);
271
272         // tell pin change mask to listen to left
encoder pins

```

```

273     PCMSK2 |= (1 << LEFT_ENC_PIN_A) | (1 <<
274         LEFT_ENC_PIN_B);
275         // tell pin change mask to listen to right
276         // encoder pins
277     PCMSK1 |= (1 << RIGHT_ENC_PIN_A) | (1 <<
278         RIGHT_ENC_PIN_B);
279
280     // enable PCINT1 and PCINT2 interrupt in the
281     // general interrupt mask
282     PCICR |= (1 << PCIE1) | (1 << PCIE2);
283 #endif
284     initMotorController(); //初始化电机控制
285     resetPID(); //重置 PID
286 #endif
287
288 /* Attach servos if used */
289 #ifdef USE_SERVOS
290     int i;
291     for (i = 0; i < N_SERVOS; i++) {
292         servos[i].initServo(
293             servoPins[i],
294             stepDelay[i],
295             servoInitPosition[i]);
296     }
297 #endif
298 }
299
300 /* Enter the main loop.  Read and parse input from
301 the serial port
302 and run any valid commands. Run a PID
303 calculation at the target
304 interval and check for auto-stop conditions.
305 */
306 void loop() {
307     //读取串口命令
308     while (Serial.available() > 0) {
309
310         // Read the next character
311         chr = Serial.read();

```

```

306
307     // Terminate a command with a CR
308     if (chr == 13) {
309         if (arg == 1) argv1[index] = NULL;
310         else if (arg == 2) argv2[index] = NULL;
311         runCommand();
312         resetCommand();
313     }
314     // Use spaces to delimit parts of the command
315     else if (chr == ' ') {
316         // Step through the arguments
317         if (arg == 0) arg = 1;
318         else if (arg == 1) {
319             argv1[index] = NULL;
320             arg = 2;
321             index = 0;
322         }
323         continue;
324     }
325     else {
326         if (arg == 0) {
327             // The first arg is the single-letter
328             command
329             cmd = chr;
330         }
331         else if (arg == 1) {
332             // Subsequent arguments can be more than
333             one character
334             argv1[index] = chr;
335             index++;
336         }
337         else if (arg == 2) {
338             argv2[index] = chr;
339             index++;
340         }
341     }

```

```

342 // If we are using base control, run a PID
343 // calculation at the appropriate intervals
344 #ifdef USE_BASE
345     if (millis() > nextPID) {
346         updatePID(); //PID调速
347         nextPID += PID_INTERVAL;
348     }
349     // Check to see if we have exceeded the auto-
350     // stop interval
351     if ((millis() - lastMotorCommand) >
352         AUTO_STOP_INTERVAL) {
353         setMotorSpeeds(0, 0);
354         moving = 0;
355     }
356 #endif
357
358 // Sweep servos
359 #ifdef USE_SERVOS
360     int i;
361     for (i = 0; i < N_SERVOS; i++) {
362         servos[i].doSweep();
363     }
364 #endif
365 }
```

这其中，需要关注的是基座控制器以及串口命令的相关部分，而由于没有使用舵机，所以舵机控制器部分暂不介绍。

1.串口命令

在主程序中，包含了 commands.h，该文件中包含了当前程序预定义的串口命令，可以编译程序并上传至 Arduino 电路板，然后打开串口监视器测试（当前程序并未修改，所以并非所有串口可用）：

- w 可以用于控制引脚电平
- x 可以用于模拟输出

以LED灯控制为例，通过串口监视器录入命令：

- w 13 0 == LED灯关闭
- w 13 1 == LED灯打开
- x 13 50 == LED灯PWM值为50

2.启用基座控制器

源码默认没有启用基座控制器、启用了舵机，我们需要启用基座控制器，禁用舵机。修改后代码如下：

```

1 #define USE_BASE           // Enable the base
2 // #undef USE_BASE        // Disable the base
3
4 /* Define the motor controller and encoder library
   you are using */
5 #ifdef USE_BASE
6     /* The Pololu VNH5019 dual motor driver shield
    */
7     // #define POLOLU_VNH5019
8
9     /* The Pololu MC33926 dual motor driver shield
    */
10    // #define POLOLU_MC33926
11
12    /* The RoboGaia encoder shield */
13    // #define ROBOGAIA
14
15    /* Encoders directly attached to Arduino board
    */
16    // #define ARDUINO_ENC_COUNTER
17
18    /* L298 Motor driver */
19    // #define L298_MOTOR_DRIVER
20 #endif
21
22 // #define USE_SERVOS // Enable use of PWM servos
   as defined in servos.h

```

```
23 #undef USE_SERVOS      // Disable use of PWM servos
```

注意:我们没有使用官方的电机驱动模块以及编码器,后期需要自定义电机驱动与编码器实现。

8.4.3 底盘实现_02Arduino端编码器驱动

测速是整个PID闭环控制中的必须环节,我们必须修改代码适配当前AB相编码器,虽然需要重写功能,但是测速部分内容已经封装,只需要实现编码器计数即可,大致实现流程如下:

1. ROSArduinoBridge.ino 中需要注释之前的编码器驱动,添加自定义编码器驱动;
2. encoder_driver.h 中设置编码器引脚并声明初始化函数以及中断函数;
3. encoder_driver.ino 中实现编码器计数以及重置函数;
4. ROSArduinoBridge.ino 中 setup 函数调用编码器初始化函数。
5. 测试

1. 定义编码器驱动

ROSArduinoBridge.ino需要添加编码器宏定义,代码如下:

```
1 #define USE_BASE          // Enable the base
2 // #undef USE_BASE       // Disable the base
3
4 /* Define the motor controller and encoder library
5  you are using */
6 #ifdef USE_BASE
7     /* The Pololu VNH5019 dual motor driver shield
8     */
9     // #define POLOLU_VNH5019
10
11    /* The Pololu MC33926 dual motor driver shield
12    */
13    // #define POLOLU_MC33926
```

```

11
12     /* The RoboGaia encoder shield */
13     // #define ROBOGAIA
14
15     /* Encoders directly attached to Arduino board
16     */
17     // #define ARDUINO_ENC_COUNTER
18     #define ARDUINO_MY_COUNTER
19
20     /* L298 Motor driver*/
21     // #define L298_MOTOR_DRIVER
22
23     #define L298P_MOTOR_DRIVER
24 #endif

```

先去除 `#define L298P_MOTOR_DRIVER` 的注释，否则后续编译会抛出异常。

2.修改encoder_driver.h文件

修改后内容如下：

```

1  /*
2   ****
3   **** Encoder driver function definitions - by James
4   Nugen
5   ****
6   ****
7   #ifdef ARDUINO_ENC_COUNTER
8   //below can be changed, but should be PORTD
9   pins;
10  //otherwise additional changes in the code are
11  required
12  #define LEFT_ENC_PIN_A PD2 //pin 2

```

```

10 #define LEFT_ENC_PIN_B PD3 //pin 3
11
12 //below can be changed, but should be PORTC pins
13 #define RIGHT_ENC_PIN_A PC4 //pin A4
14 #define RIGHT_ENC_PIN_B PC5 //pin A5
15 #elif defined ARDUINO_MY_COUNTER
16 #define LEFT_A 21
17 #define LEFT_B 20
18 #define RIGHT_A 18
19 #define RIGHT_B 19
20 void initEncoders();
21 void leftEncoderEventA();
22 void leftEncoderEventB();
23 void rightEncoderEventA();
24 void rightEncoderEventB();
25 #endif
26
27 long readEncoder(int i);
28 void resetEncoder(int i);
29 void resetEncoders();

```

3.修改encoder_driver.ino 文件

主要添加内容如下:

```

1 #elif defined ARDUINO_MY_COUNTER
2   volatile long left_count = 0L;
3   volatile long right_count = 0L;
4   void initEncoders(){
5     pinMode(LEFT_A,INPUT); // 21 --- 2
6     pinMode(LEFT_B,INPUT); // 20 --- 3
7     pinMode(RIGHT_A,INPUT); // 18 --- 5
8     pinMode(RIGHT_B,INPUT); // 19 --- 4
9
10    attachInterrupt(2, leftEncoderEventA, CHANGE);
11    attachInterrupt(3, leftEncoderEventB, CHANGE);
12    attachInterrupt(5, rightEncoderEventA, CHANGE);
13    attachInterrupt(4, rightEncoderEventB, CHANGE);

```

```
14  }
15  void leftEncoderEventA(){
16      if(digitalRead(LEFT_A) == HIGH){
17          if(digitalRead(LEFT_B) == HIGH){
18              left_count++;
19          } else {
20              left_count--;
21          }
22      } else {
23          if(digitalRead(LEFT_B) == LOW){
24              left_count++;
25          } else {
26              left_count--;
27          }
28      }
29  }
30  void leftEncoderEventB(){
31      if(digitalRead(LEFT_B) == HIGH){
32          if(digitalRead(LEFT_A) == LOW){
33              left_count++;
34          } else {
35              left_count--;
36          }
37      } else {
38          if(digitalRead(LEFT_A) == HIGH){
39              left_count++;
40          } else {
41              left_count--;
42          }
43      }
44  }
45  void rightEncoderEventA(){
46      if(digitalRead(RIGHT_A) == HIGH){
47          if(digitalRead(RIGHT_B) == HIGH){
48              right_count++;
49          } else {
50              right_count--;
51          }
52      } else {
```

```
53     if(digitalRead(LEFT_A) == LOW) {
54         left_count++;
55     } else {
56         left_count--;
57     }
58 }
59
60 void rightEncoderEventB(){
61     if(digitalRead(LEFT_B) == HIGH){
62         if(digitalRead(LEFT_A) == LOW){
63             right_count++;
64         } else {
65             right_count--;
66         }
67     } else {
68         if(digitalRead(LEFT_A) == HIGH){
69             right_count++;
70         } else {
71             right_count--;
72         }
73     }
74 }
75
76 long readEncoder(int i) {
77     if (i == LEFT) return left_count;
78     else return right_count;
79 }
80
81 /* Wrap the encoder reset function */
82 void resetEncoder(int i) {
83     if (i == LEFT){
84         left_count=0L;
85         return;
86     } else {
87         right_count=0L;
88         return;
89     }
90 }
```

4. ROSArduinoBridge.ino 实现初始化

setup 添加语句: initEncoders();

```

1 void setup() {
2     Serial.begin(BAUDRATE);
3
4 // Initialize the motor controller if used */
5 #ifdef USE_BASE
6     #ifdef ARDUINO_ENC_COUNTER
7         //set as inputs
8         DDRD &= ~(1<<LEFT_ENC_PIN_A);
9         DDRD &= ~(1<<LEFT_ENC_PIN_B);
10        DDRC &= ~(1<<RIGHT_ENC_PIN_A);
11        DDRC &= ~(1<<RIGHT_ENC_PIN_B);
12
13        //enable pull up resistors
14        PORTD |= (1<<LEFT_ENC_PIN_A);
15        PORTD |= (1<<LEFT_ENC_PIN_B);
16        PORTC |= (1<<RIGHT_ENC_PIN_A);
17        PORTC |= (1<<RIGHT_ENC_PIN_B);
18
19        // tell pin change mask to listen to left
20        // encoder pins
21        PCMSK2 |= (1 << LEFT_ENC_PIN_A) | (1 <<
22        LEFT_ENC_PIN_B);
23        // tell pin change mask to listen to right
24        // encoder pins
25        PCMSK1 |= (1 << RIGHT_ENC_PIN_A) | (1 <<
26        RIGHT_ENC_PIN_B);
27
28        // enable PCINT1 and PCINT2 interrupt in the
29        // general interrupt mask
30        PCICR |= (1 << PCIE1) | (1 << PCIE2);
31    #elif defined ARDUINO_MY_COUNTER
32        initEncoders();
33    #endif
34    initMotorController();

```

```

30     resetPID();
31 #endif
32
33 /* Attach servos if used */
34 #ifdef USE_SERVOS
35     int i;
36     for (i = 0; i < N_SERVOS; i++) {
37         servos[i].initServo(
38             servoPins[i],
39             stepDelay[i],
40             servoInitPosition[i]);
41     }
42 #endif
43
44
45
46 }

```

5. 测试

编译并上传程序，打开串口监视器，然后旋转车轮，在串口监视器中录入 e 即可查看左右编码器计数，录入命令 r 可以重置计数。

8.4.4 底盘实现_03Arduino端电机驱动

自定义电机驱动的实现与上一节的编码器驱动流程类似：

1. ROSArduinoBridge.ino 中需要注释之前的电机驱动，添加自定义电机驱动；
2. motor_driver.h 中设置左右电机引脚；
3. motor_driver.ino 中实现初始化与速度设置函数；
4. 测试

1. 定义电机驱动

ROSArduinoBridge.ino需要添加电机宏定义,代码如下:

```

1 #define USE_BASE          // Enable the base
  controller code
2 //#undef USE_BASE        // Disable the base
  controller code
3
4 /* Define the motor controller and encoder library
   you are using */
5 #ifdef USE_BASE
6     /* The Pololu VNH5019 dual motor driver shield
   */
7     // #define POLOLU_VNH5019
8
9     /* The Pololu MC33926 dual motor driver shield
   */
10    // #define POLOLU_MC33926
11
12    /* The RoboGaia encoder shield */
13    // #define ROBOGAIA
14
15    /* Encoders directly attached to Arduino board
   */
16    // #define ARDUINO_ENC_COUNTER
17    /* 使用自定义的编码器驱动 */
18    #define ARDUINO_MY_COUNTER
19
20    /* L298 Motor driver */
21    // #define L298_MOTOR_DRIVER
22    // 使用自定义的L298P电机驱动
23    #define L298P_MOTOR_DRIVER
24 #endif

```

2.修改motor_driver.h文件

修改后内容如下：

```

1 /*****
  ****

```

2 Motor driver function definitions – by James
 3 Nugen

```

3 ****
4 ****
5 #ifdef L298_MOTOR_DRIVER
6   #define RIGHT_MOTOR_BACKWARD 5
7   #define LEFT_MOTOR_BACKWARD 6
8   #define RIGHT_MOTOR_FORWARD 9
9   #define LEFT_MOTOR_FORWARD 10
10   #define RIGHT_MOTOR_ENABLE 12
11   #define LEFT_MOTOR_ENABLE 13
12 #elif defined L298P_MOTOR_DRIVER
13   #define DIRA 4
14   #define PWMA 5
15   #define DIRB 7
16   #define PWMB 6
17 #endif
18
19 void initMotorController();
20 void setMotorSpeed(int i, int spd);
21 void setMotorSpeeds(int leftSpeed, int
  rightSpeed);
```

3.修改motor_driver.ino 文件

主要添加内容如下:

```

1 #elif defined L298P_MOTOR_DRIVER
2   void initMotorController(){
3     pinMode(DIRA,OUTPUT);
4     pinMode(PWMA,OUTPUT);
5     pinMode(DIRB,OUTPUT);
6     pinMode(PWMB,OUTPUT);
7 }
8 void setMotorSpeed(int i, int spd){
9     unsigned char reverse = 0;
```

```

10
11     if (spd < 0)
12     {
13         spd = -spd;
14         reverse = 1;
15     }
16     if (spd > 255)
17         spd = 255;
18
19     if (i == LEFT) {
20         if (reverse == 0) {
21             digitalWrite(DIRA,HIGH);
22         } else if (reverse == 1) {
23             digitalWrite(DIRA,LOW);
24         }
25         analogWrite(PWMA,spd);
26     } else /*if (i == RIGHT) //no need for
condition*/ {
27         if (reverse == 0) {
28             digitalWrite(DIRB,LOW);
29         } else if (reverse == 1) {
30             digitalWrite(DIRB,HIGH);
31         }
32         analogWrite(PWMB,spd);
33     }
34 }
35 void setMotorSpeeds(int leftSpeed, int
rightSpeed){
36     setMotorSpeed(LEFT, leftSpeed);
37     setMotorSpeed(RIGHT, rightSpeed);
38 }

```

4. 测试

编译并上传程序，打开串口监视器，然后输入命令，命令格式为: m num1 num2， num1和num2分别为单位时间内左右电机各自转动的编码器计数，而默认单位时间为 1/30 秒。



举例，假设车轮旋转一圈编码器计数为 3960(减速比90，编码器分辨率11且采用4倍频计数)，当输入命令为 m 200 100 时：

左电机转速为: $200 * 30 * 60 / 3960 = 90.9$ (r/m)

右电机转速为: $100 * 30 * 60 / 3960 = 45.45$ (r/m)

8.4.5 底盘实现_04Arduino端PID控制

上一节最后测试时，电机可能会出现抖动、顿挫的现象，显而易见的这是由于PID参数设置不合理导致的，本节将介绍ros_arduino_bridge中的PID调试，大致流程如下：

1. 了解ros_arduino_bridge中PID调试的流程；
2. 实现PID调试。

1.ros_arduino_bridge中PID调试源码分析

基本思想：

1. 先定义调试频率(周期)，并预先设置下一次的结束时刻；
2. 当当前时刻大于预设的结束时刻时，即进行PID调试，且重置下一次调试结束时刻
3. PID代码在diff_controller中实现，PID的目标值是命令输入的转速，当前转速则是通过读取当前编码器计数再减去上一次调试结束时记录的编码器计数获取；
4. 最后输出 PWM

ROSArduinoBridge.ino 中和PID控制相关的变量：

```

1 #ifdef USE_BASE
2  /* Motor driver function definitions */
3  #include "motor_driver.h"
4
5  /* Encoder driver function definitions */
6  #include "encoder_driver.h"
7

```

```

8  /* PID parameters and functions */
9  #include "diff_controller.h"
10
11 /* Run the PID loop at 30 times per second */
12 #define PID_RATE 30 // Hz PID调试频率
13
14 /* Convert the rate into an interval */
15 const int PID_INTERVAL = 1000 / PID_RATE; // PID
16 调试周期
17
18 /* Track the next time we make a PID calculation */
19
20 /* Stop the robot if it hasn't received a
21 movement command
22 in this number of milliseconds */
23 #define AUTO_STOP_INTERVAL 5000
24 long lastMotorCommand = AUTO_STOP_INTERVAL;
25 #endif

```

ROSArduinoBridge.ino 的 runCommand()函数中:

```

1 #ifdef USE_BASE
2   case READ_ENCODERS:
3     Serial.print(readEncoder(LEFT));
4     Serial.print(" ");
5     Serial.println(readEncoder(RIGHT));
6     break;
7   case RESET_ENCODERS:
8     resetEncoders();
9     resetPID();
10  Serial.println("OK");
11  break;
12  case MOTOR_SPEEDS: //-----
-----
```

```

13  /* Reset the auto stop timer */
14  lastMotorCommand = millis();
15  if (arg1 == 0 && arg2 == 0) {
16      setMotorSpeeds(0, 0);
17      resetPID();
18      moving = 0;
19  }
20  else moving = 1;
21  //设置左右电机目标转速分别为参数1和参数2
22  leftPID.TargetTicksPerFrame = arg1;
23  rightPID.TargetTicksPerFrame = arg2;
24  Serial.println("OK");
25  break;
26 case UPDATE_PID:
27  while ((str = strtok_r(p, ":", &p)) != '\0') {
28      pid_args[i] = atoi(str);
29      i++;
30  }
31  Kp = pid_args[0];
32  Kd = pid_args[1];
33  Ki = pid_args[2];
34  Ko = pid_args[3];
35  Serial.println("OK");
36  break;
37 #endif

```

ROSArduinoBridge.ino 的 loop() 函数中:

```

1 #ifdef USE_BASE
2     //如果当前时刻大于 nextPID,那么就执行PID调速，并在
3     //nextPID 上自增一个PID调试周期
4     if (millis() > nextPID) {
5         updatePID();
6         nextPID += PID_INTERVAL;
7     }
8     // Check to see if we have exceeded the auto-
9     // stop interval
10    if ((millis() - lastMotorCommand) >
11        AUTO_STOP_INTERVAL) {
12        setMotorSpeeds(0, 0);
13        moving = 0;
14    }
15 #endif

```

diff_controller.h 中的PID调试代码:

```

1 /* Functions and type-defs for PID control.
2
3     Taken mostly from Mike Ferguson's ArbotiX code
4     which lives at:
5
6     http://vanadium-ros-
7     pkg.googlecode.com/svn/trunk/arbotix/
8 */
9
10 /* PID setpoint info For a Motor */
11 typedef struct {
12     double TargetTicksPerFrame;      // target speed
13     in ticks per frame 目标转速
14     long Encoder;                  // encoder count
15     编码器计数
16     long PrevEnc;                 // last encoder
17     count 上次的编码器计数
18
19 */

```

```

15  * Using previous input (PrevInput) instead of
  PrevError to avoid derivative kick,
16  * see
  http://brettbeauregard.com/blog/2011/04/improving-
    the-beginner%E2%80%99s-pid-derivative-kick/
17  */
18  int PrevInput;           // last input
19  //int PrevErr;           // last error
20
21  /*
22  * Using integrated term (ITerm) instead of
  integrated error (Ierror),
23  * to allow tuning changes,
24  * see
  http://brettbeauregard.com/blog/2011/04/improving-
    the-beginner%E2%80%99s-pid-tuning-changes/
25  */
26  //int Ierror;
27  int ITerm;               //integrated term
28
29  long output;             // last motor
  setting
30 }
31 SetPointInfo;
32
33 SetPointInfo leftPID, rightPID;
34
35 /* PID Parameters */
36 int Kp = 20;
37 int Kd = 12;
38 int Ki = 0;
39 int Ko = 50;
40
41 unsigned char moving = 0; // is the base in
  motion?
42
43 /*
44 * Initialize PID variables to zero to prevent
  startup spikes

```

```

45 * when turning PID on to start moving
46 * In particular, assign both Encoder and PrevEnc
47 * the current encoder value
48 * See
49 * http://brettbeauregard.com/blog/2011/04/improving-
50 * Note that the assumption here is that PID is
51 * only turned on
52 * when going from stop to moving, that's why we
53 * can init everything on zero.
54 */
55 void resetPID(){
56     leftPID.TargetTicksPerFrame = 0.0;
57     leftPID.Encoder = readEncoder(LEFT);
58     leftPID.PrevEnc = leftPID.Encoder;
59     leftPID.output = 0;
60     leftPID.PrevInput = 0;
61     leftPID.ITerm = 0;
62
63     rightPID.TargetTicksPerFrame = 0.0;
64     rightPID.Encoder = readEncoder(RIGHT);
65     rightPID.PrevEnc = rightPID.Encoder;
66     rightPID.output = 0;
67     rightPID.PrevInput = 0;
68     rightPID.ITerm = 0;
69 }
70
71 /* PID routine to compute the next motor commands
72 */
73 //左右电机具体调试函数
74 void doPID(SetPointInfo * p) {
75     long Perror;
76     long output;
77     int input;
78
79     //Perror = p->TargetTicksPerFrame - (p->Encoder
80     - p->PrevEnc);
81     input = p->Encoder - p->PrevEnc;
82     Perror = p->TargetTicksPerFrame - input;

```

```

77
78     //根据 input 绘图
79     //Serial.println(input);
80     /*
81      * Avoid derivative kick and allow tuning
82      changes,
83      * see
84      http://brettbeauregard.com/blog/2011/04/improving-
85      the-beginner%E2%80%99s-pid-derivative-kick/
86      * see
87      http://brettbeauregard.com/blog/2011/04/improving-
88      the-beginner%E2%80%99s-pid-tuning-changes/
89      */
90     //output = (Kp * Perror + Kd * (Perror - p-
91     >PrevErr) + Ki * p->Ierror) / Ko;
92     // p->PrevErr = Perror;
93     output = (Kp * Perror - Kd * (input - p-
94     >PrevInput) + p->ITerm) / Ko;
95     p->PrevEnc = p->Encoder;
96
97     output += p->output;
98     // Accumulate Integral error *or* Limit output.
99     // Stop accumulating when output saturates
100    if (output >= MAX_PWM)
101        output = MAX_PWM;
102    else if (output <= -MAX_PWM)
103        output = -MAX_PWM;
104    else
105        /*
106         * allow turning changes, see
107         http://brettbeauregard.com/blog/2011/04/improving-
108         the-beginner%E2%80%99s-pid-tuning-changes/
109         */
110         p->ITerm += Ki * Perror;
111
112         p->output = output;
113         p->PrevInput = input;
114     }
115
116

```

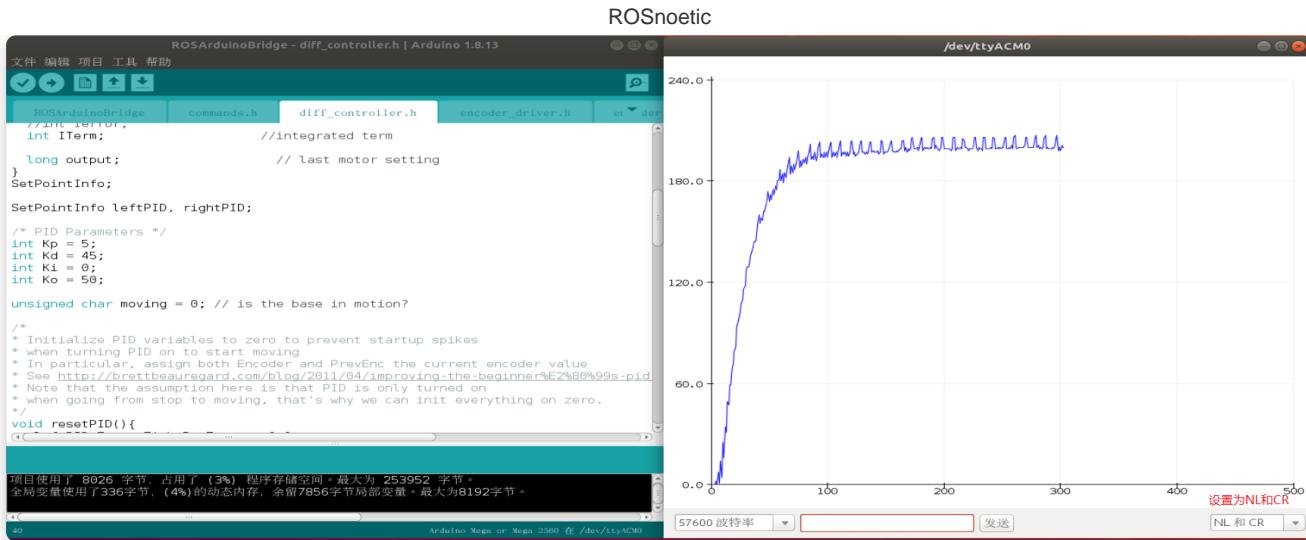
```

107  /* Read the encoder values and call the PID
108   routine */
109 void updatePID() {
110   /* Read the encoders */
111   leftPID.Encoder = readEncoder(LEFT);
112   rightPID.Encoder = readEncoder(RIGHT);
113
114   /* If we're not moving there is nothing more to
115    do */
116   if (!moving){
117     /*
118      * Reset PIDs once, to prevent startup spikes,
119      * see
120      http://brettbeauregard.com/blog/2011/04/improving-
121      the-beginner%E2%80%99s-pid-initialization/
122      * PrevInput is considered a good proxy to
123      detect
124      * whether reset has already happened
125      */
126      if (leftPID.PrevInput != 0 ||
127          rightPID.PrevInput != 0) resetPID();
128
129      return;
130
131      /* Compute PID update for each motor */
132      doPID(&rightPID);
133      doPID(&leftPID);
134
135      /* Set the motor speeds accordingly */
136      setMotorSpeeds(leftPID.output, rightPID.output);
137
138  }

```

2.PID调试

调试时，需要在 `diff_controller.h` 中打印 `input` 的值，然后通过串口绘图器输入命令: `m` 参数1 参数2，根据绘图结果调试: `Kp`、`Ki` 和 `Kd` 的值。



- 调试时，可以先调试单个电机的PID，比如，可以先注释 `doPID(&rightPID);`；
- PID算法不同，即便算法相同，如果参与运算的数据单位不同，都会导致不同的调试结果，不可以直接复用之前的调试结果。

PID调试技巧可以参考之前介绍。

8.5 机器人平台设计之控制系统

我们机器人平台的控制系统应该如何设计？ROS系统的控制系统选择是多样的，一般常用的有基于ARM、x86等架构的处理器，比如：PC、工控机、树莓派...，不同的处理器都存在一定的优缺点，PC和工控机，处理器性能强大，但是功耗高、体积大、灵活性差。嵌入式系统则反之。



那么应该如何选择控制系统呢？

比如：如果是中大型机器人，可以使用PC或工控机等作为控制系统；但是如果是小型或微型机器人，就应该使用嵌入式系统吗？

我们机器人平台属于小型甚至微型机器人，虽然也可以使用PC作为机器人的控制系统，不过无论是从尺寸、负载能力还是扩展性的角度来看显然都是不适宜的。但是如果只是将控制系统简单小型化，比如使用树莓派，处理复杂的算法或比较耗资源的仿真实现显然又不能满足算力的要求...当前情形好像陷入了两难的境地。

ROS是一种分布式设计框架，针对小型或微型机器人平台的控制系统，可以选择多处理器的实现策略。具体实现是“PC + 嵌入式”，可以使用嵌入式系统(比如树莓派)充当机器人本体的控制系统，而PC则实现远程监控，通过前者实现数据采集与直接的底盘控制，而后者则远程实现图形显示以及功能运算。本节就主要介绍的就是这种多处理器的组合式框架实现，具体内容如下：

- 树莓派概述；
- 实现树莓派与PC的分布式系统搭建；
- 使用 ssh 远程连接树莓派；
- 树莓派端安装并配置 `ros_arduino_bridge`。

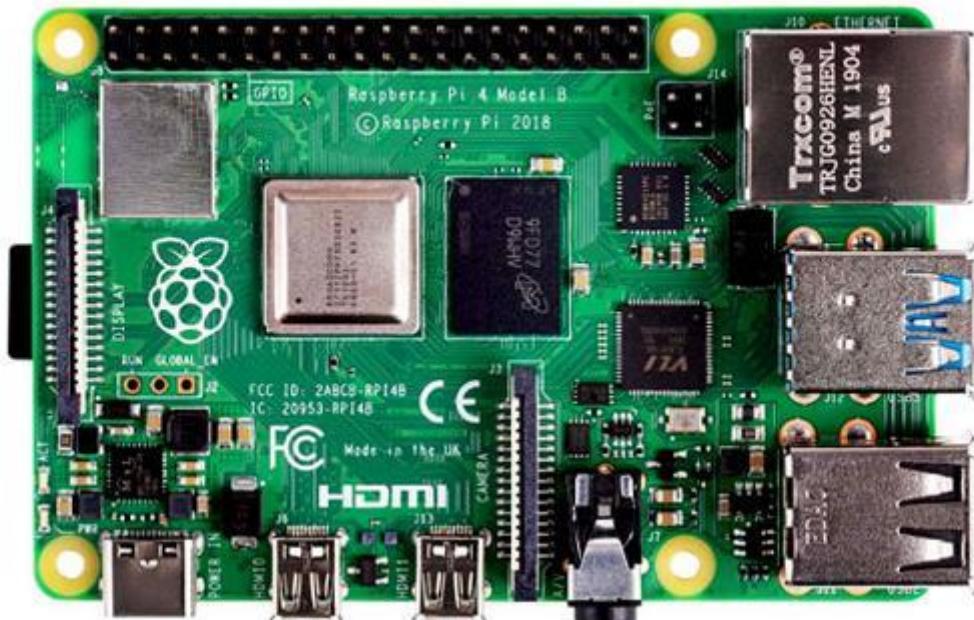
8.5.1 控制系统实现_树莓派概述

概念

Raspberry Pi(中文名为“树莓派”，简写为RPi，(或者RasPi / RPI)是为学习计算机编程教育而设计)，只有信用卡大小的微型电脑，其系统基于Linux。随着Windows 10 IoT的发布，我们也将可以用上运行Windows的树莓派。

结构

它是一款基于ARM的微型电脑主板，以SD/MicroSD卡为内存硬盘，卡片主板周围有1/2/4个USB接口和一个以太网接口（A型没有网口），可连接键盘、鼠标和网线，同时拥有视频模拟信号的电视输出接口和HDMI高清视频输出接口，以上部件全部整合在一张仅比信用卡稍大的主板上，具备所有PC的基本功能只需接通电视机和键盘，就能执行如电子表格、文字处理、玩游戏、播放高清视频等诸多功能，下图为树莓派4b。



配件

单独一块树莓派主板是无法运行的，必须集成一些配件才能实现一定的功能，树莓派周边配件是比较丰富的，比如：USB电源、SD卡、读卡器、HDMI连接线、显示屏、键盘、鼠标、保护壳、风扇等等，除此之外还有各式各样的传感器:声音传感器、温度传感器、土壤湿度传感器....对于我们教程而言，所需的配件比较简单，硬件清单如下：

- 树莓派主板
- 电源线
- SD卡(已安装 Ubuntu 以及 ROS)
- 显示屏或 HDMI采集卡以及配套的数据线
- 鼠标、键盘

接线以及使用

1. 树莓派连接显示屏



2.树莓派连接HDMI采集卡



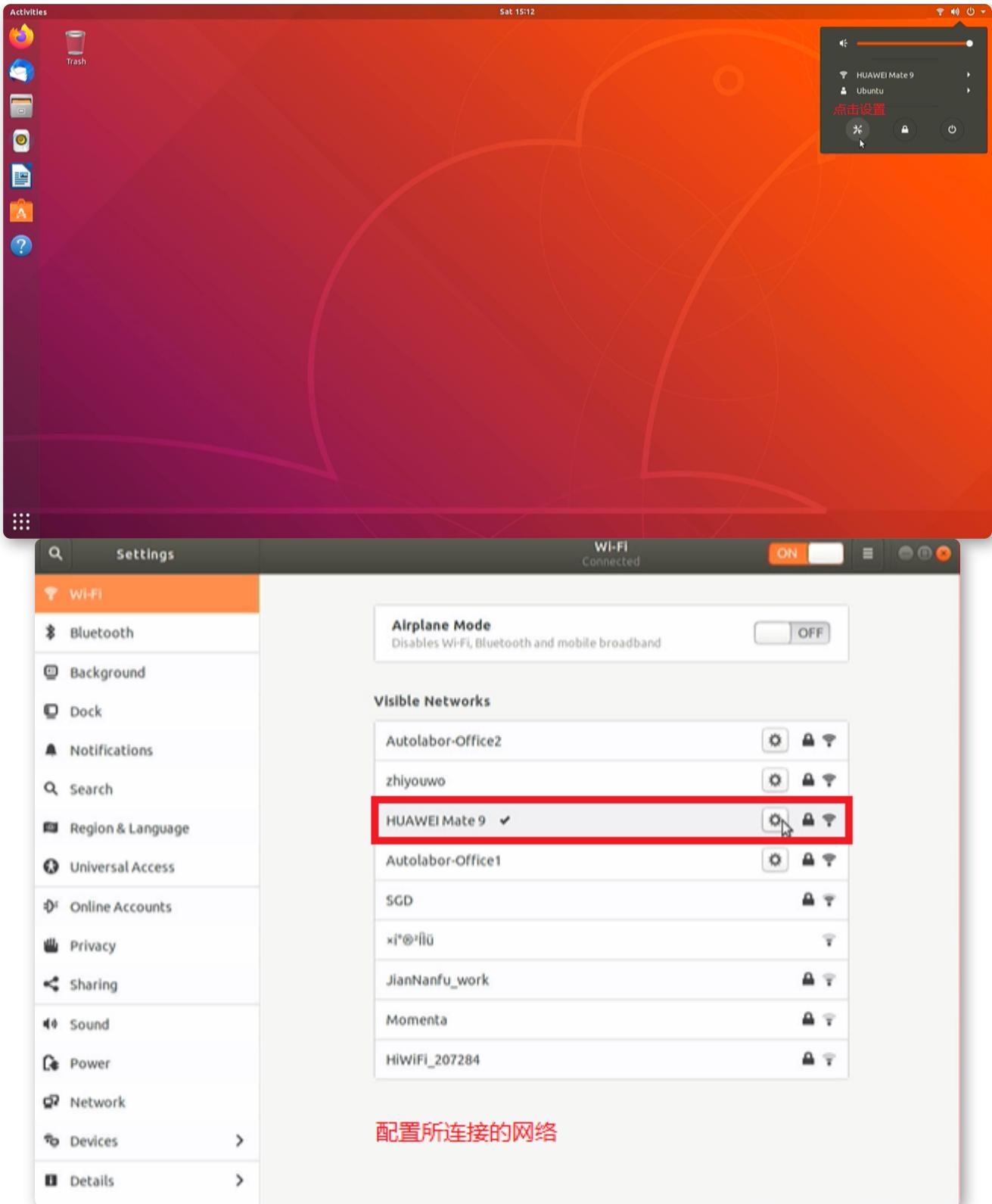
使用 windows 的相机查看运行结果，打开windows相机并点击更改相机即可：

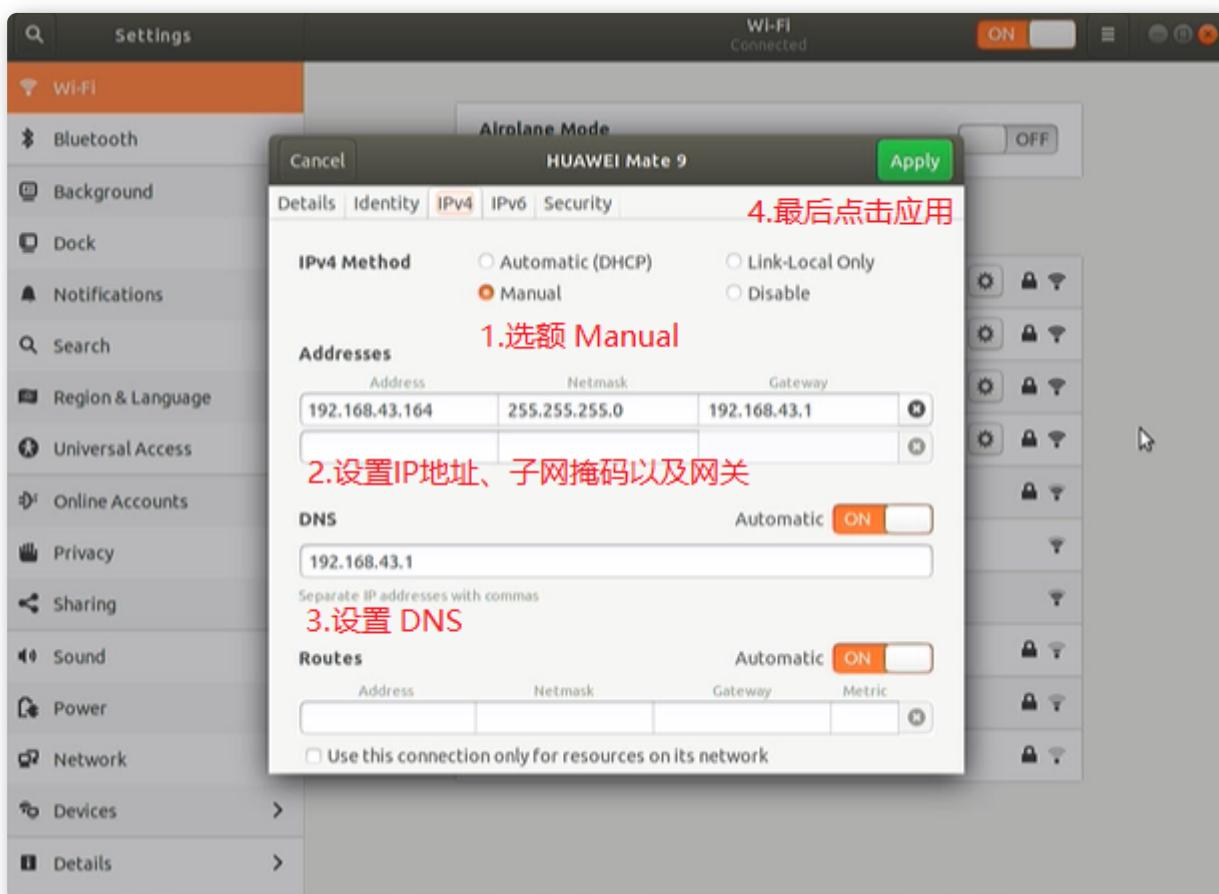
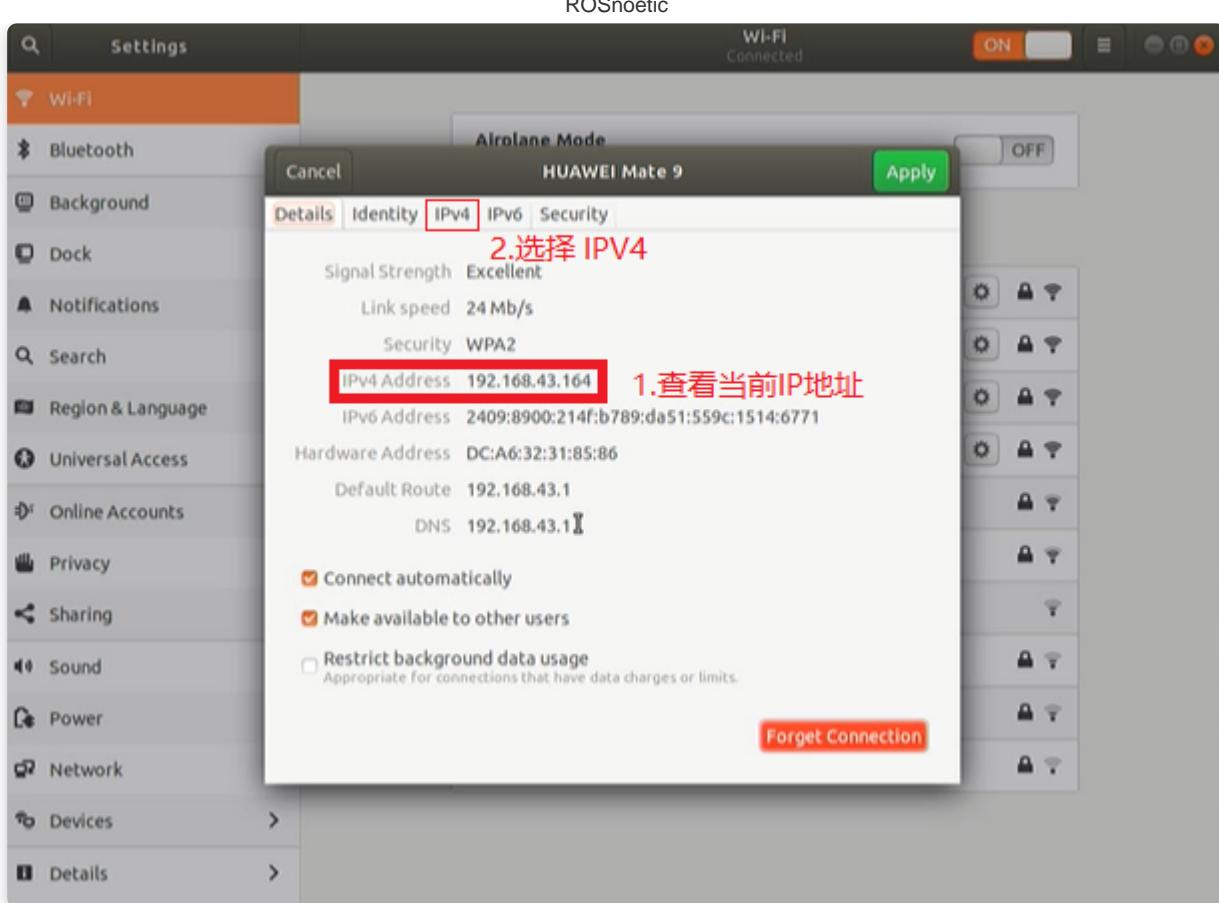


8.5.2 控制系统实现_分布式框架

当前分布式框架搭建时，树莓派是作为主机，而PC则作为从机，关于分布式框架的搭建流程，我们在第4章4.7节中已有详细介绍，按照流程实现即可，不过在实现此流程前，还需要做准备工作：为树莓派连接无线网络，并设置固定IP，实现如下：

- 1.硬件准备：使用显示屏或 HDMI采集卡连接树莓派并启动；
- 2.为树莓派连接无线网络；
- 3.为树莓派配置静态IP，具体操作如图：





固定IP配置完毕后，按照4.7节演示，配置分布式框架即可。

8.5.3 控制系统实现_ssh远程连接

在多处理器的分布式架构中，不同的ROS系统之间可能会频繁的涉及到文件的传输，比如：



我们在PC端编写ROS程序，而最终需要在树莓派上运行，如何将相关目录以及文件从PC上传到树莓派？

SSH 是常用手段之一。

概念

SSH (Secure Shell) 是一种通用的、功能强大的、基于软件的网络安全解决方案。计算机每次向网络发送数据时，SSH都会自动对其进行加密。数据到达目的地时，SSH自动对加密数据进行解密。整个过程都是透明的，使用OpenSSH工具将会增进你的系统安全性。SSH安装容易、使用简单。

实现

SSH实现架构上分为客户端和服务器端两大部分，客户端是数据的发送方，服务端是数据的接收方，当前场景下，我们需要从PC端发送数据到树莓派，那么PC端属于客户端，而树莓派属于服务端，整个实现具体流程是：

1. 分别安装SSH客户端与服务端
2. 服务端启动SSH服务
3. 客户端远程登陆服务端
4. 实现数据传输

1. 安装SSH客户端与服务端

默认情况下，Ubuntu系统已经安装了SSH客户端，因此只需要在树莓派安装服务端即可(如果树莓派安装的是服务版的Ubuntu，默认会安装SSH服务并已设置成了开机自启动)：

```
1 sudo apt-get install openssh-server
```

如果客户端需要自行安装，那么调用如下命令：

```
1 sudo apt-get install openssh-client
```

2.服务端启动SSH服务

树莓派启动 ssh 服务：

```
1 sudo /etc/init.d/ssh start
```

启动后查看服务是否正常运行：

```
1 ps -e | grep ssh
```

如果启动成功，会包含 sshd 与 ssh 两个程序。

以后需要频繁的使用ssh登录树莓派，为了简化实现，可以将树莓派的ssh服务设置为开机自启动，命令如下：

```
1 sudo systemctl enable ssh
```

3.客户端远程登陆服务端

登陆树莓派可以调用如下命令：

```
1 ssh 账号@ip地址
```

然后根据提示，录入登陆密码，即可成功登陆。

如果退出登陆，可以调用exit命令：

```
1 exit
```

4. 实现数据传输

上传文件:

```
1 scp 本地文件路径 账号@ip:树莓派路径
```

上传文件夹:

```
1 scp -r 本地文件夹路径 账号@ip:树莓派路径
```

下载文件:

```
1 scp 账号@ip:树莓派路径 本地文件夹路径
```

下载文件夹:

```
1 scp -r 账号@ip:树莓派路径 本地文件夹路径
```

使用优化

每次登陆树莓派时，都需要输入密码，使用不方便，可以借助密钥简化登陆过程，实现免密登陆，提高操作效率，实现思想是:生成一对公钥私钥，私钥存储在本地，公钥上传至服务器，每次登陆时，本地直接上传私钥到服务器，服务器有匹配的公钥就认为是合法用户，直接创建SSH连接即可。具体实现步骤只有两步:

1. 本地生成密钥对
2. 将公钥上传至树莓派

1. 生成密钥对

本地客户端生成公私钥：（一路回车默认即可）

```
1 ssh-keygen
```

上面这个命令会在用户目录.ssh文件夹下创建公私钥:

1. id_rsa (私钥)
2. id_rsa.pub (公钥)

2. 将公钥上传至树莓派

上传命令：

```
1 ssh-copy-id -i ~/.ssh/id_rsa.pub 账号@ip
```

上面这条命令是写到服务器上的ssh目录下，该目录下有文件authorized_keys保存了公钥内容。

以后再登陆树莓派就无需录入密码了。

8.5.4 控制系统实现_安装ros_arduino_bridge

如果你已经搭建并测试通过了分布式环境，下一步，就可以将ros_arduino_bridge功能包上传至树莓派，并在PC端通过键盘控制小车的运动了，实现流程如下：

1. 系统准备；
2. 程序修改；
3. 从PC端上传程序至树莓派；
4. 分别启动PC与树莓派端相关节点，并实现运动控制。

1. 系统准备

ros_arduino_bridge是依赖于python-serial功能包的，请先在树莓派端安装该功能包，安装命令：

```
1 $ sudo apt-get install python-serial
```

或

```
1 $ sudo pip install --upgrade pyserial
```

或

```
1 $ sudo easy_install -U pyserial
```

2.程序修改

ros_arduino_bridge的ROS端功能包主要是使用ros_arduino_python，程序入口是该包launch目录下的arduino.launch文件，内容如下：

```
1 <launch>
2   <node name="arduino" pkg="ros_arduino_python"
3     type="arduino_node.py" output="screen">
4     <rosparam file="$(find
5       ros_arduino_python)/config/my_arduino_params.yaml"
6       command="load" />
7   </node>
8 </launch>
```

需要载入yaml格式的配置文件，该文件在 config 目录下已经提供了模板，只需要复制文件并按需配置即可，复制文件并重命名，配置如下：

```
1 # For a direct USB cable connection, the port name
2 # is typically
3 # /dev/ttyACM# where # is a number such as 0,
4 # 1, 2, etc
5 # For a wireless connection like XBee, the port is
6 # typically
7 # /dev/ttyUSB# where # is a number such as 0, 1,
8 # 2, etc.
9
10 port: /dev/ttyACM0 #视情况设置，一般设置为
11 /dev/ttyACM0 或 /dev/ttyUSB0
12 baud: 57600 #波特率
13 timeout: 0.1 #超时时间
14
15 rate: 50
16 sensorstate_rate: 10
17
```

```
13 use_base_controller: True #启用基座控制器
14 base_controller_rate: 10
15
16 # For a robot that uses base_footprint, change
  base_frame to base_footprint
17 base_frame: base_footprint #base_frame 设置
18
19 # === Robot drivetrain parameters
20 wheel_diameter: 0.065 #车轮直径
21 wheel_track: 0.21 #轮间距
22 encoder_resolution: 3960#编码器精度(一圈的脉冲数 * 倍频
  * 减速比)
23 #gear_reduction: 1 #减速比
24 #motors_reversed: False #转向取反
25
26 # === PID parameters PID参数, 需要自己调节
27 Kp: 5
28 Kd: 45
29 Ki: 0
30 Ko: 50
31 accel_limit: 1.0
32
33 # === Sensor definitions. Examples only - edit
  for your robot.
34 #      Sensor type can be one of the follow (case
  sensitive!):
35 #      * Ping
36 #      * GP2D12
37 #      * Analog
38 #      * Digital
39 #      * PololuMotorCurrent
40 #      * PhidgetsVoltage
41 #      * PhidgetsCurrent (20 Amp, DC)
42
43
44
45 sensors: {
46   #motor_current_left: {pin: 0, type:
  PololuMotorCurrent, rate: 5},
```

```

47  #motor_current_right:  {pin: 1, type:
  PololuMotorCurrent, rate: 5},
48  #ir_front_center:       {pin: 2, type: GP2D12,
  rate: 10},
49  #sonar_front_center:   {pin: 5, type: Ping,
  rate: 10},
50  arduino_led:           {pin: 13, type: Digital,
  rate: 5, direction: output}
51 }

```

3.程序上传

请先在树莓派端创建工作空间，在PC端进入本地工作空间的src目录，调用程序上传命令：

```

1 scp -r ros_arduino_bridge/ 树莓派用户名@树莓派ip:~/工作
空间/src

```

在树莓派端进入工作空间并编译：

```

1 catkin_make

```

4.测试

现启动树莓派端程序，再启动PC端程序。

树莓派端

启动 ros_arduino_bridge 节点：

```

1 roslaunch ros_arduino_python arduino.launch

```

PC端

启动键盘控制节点：

```
1 rosrun teleop_twist_keyboard  
teleop_twist_keyboard.py
```

如无异常，现在就可以在PC端通过键盘控制小车运动了，并且PC端还可以使用rviz查看小车的里程计信息。

资料:控制系统实现_树莓派安装ROS

在树莓派上搭建ROS环境需要两步实现:

1. 在树莓派上安装Ubuntu
2. 基于Ubuntu安装ROS

版本选择:

- Ubuntu选用18.04
- ROS选用melodic
- 树莓派选用4b

具体实现流程如下。

1.Ubuntu安装

1.1硬件准备

- 树莓派
- 读卡器
- TF卡(建议16G以及以上)
- 显示屏或 HDMI采集卡 以及配套的数据线
- 鼠标键盘
- 网线

1.2软件准备

1.Ubuntu18.04下载并解压，下载地址:<https://ubuntu-mate.org/download/>

Download your
Ubuntu Pi
image



Raspberry Pi 2



Raspberry Pi 3



Raspberry Pi 4

Ubuntu 20.04.1 LTS

RECOMMENDED

The version of Ubuntu with long
term support, until April 2025.

[Download 64-bit](#)

选择64位

[Download 64-bit](#)

[Download 32-bit](#)

[Download 32-bit](#)

[Download 32-bit](#)

Ubuntu 18.04.5

The previous LTS version of
Ubuntu for projects without
20.04 support.

[Download 64-bit](#)

[Download 64-bit](#)

[Download 32-bit](#)

[Download 32-bit](#)

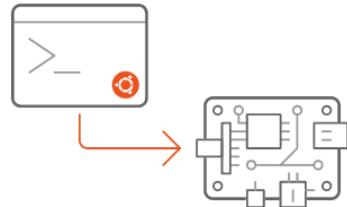
[Download 32-bit](#)

Thank you for downloading
Ubuntu Server 20.04.1
for Raspberry Pi

Your download should start automatically. If it doesn't, [download now](#).

You can [verify your download](#).

[进入下一页选择下载方式并下载](#)



2.win32 Disk Imager烧录软件下载并安装，下载地址:<https://sourceforge.net/projects/win32diskimager/>

根据提示下载并安装

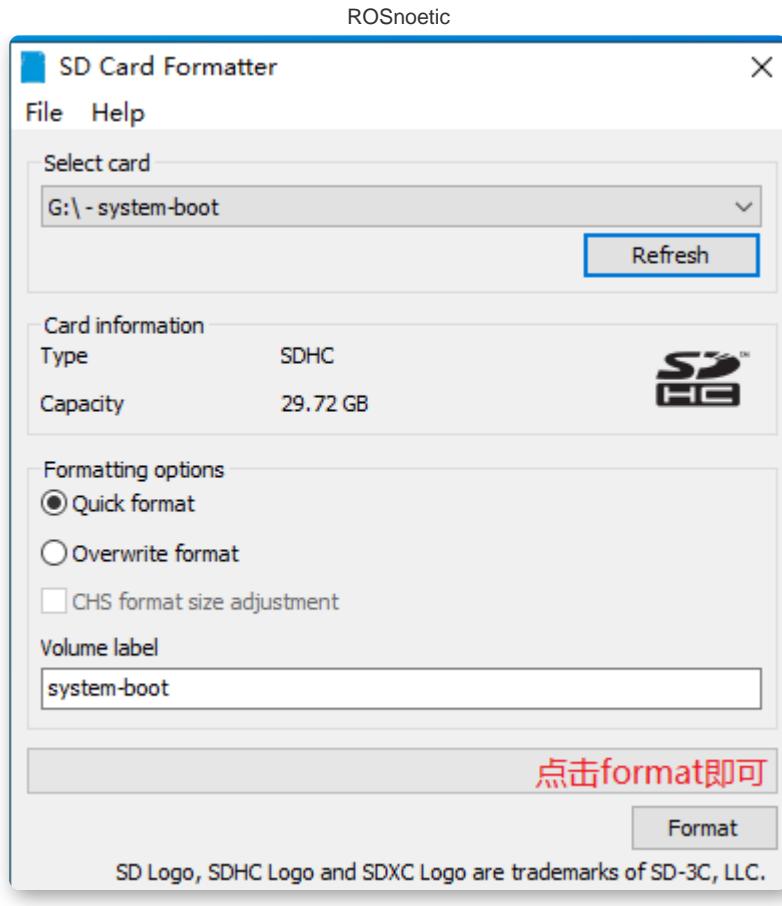
3.如果TF卡已有内容，在使用之前需要执行格式化，比如可以使用SD Card Formatter:

SD Card Formatter下载并安装，下载地址:<https://www.sdcard.org/downloads/formatter/>

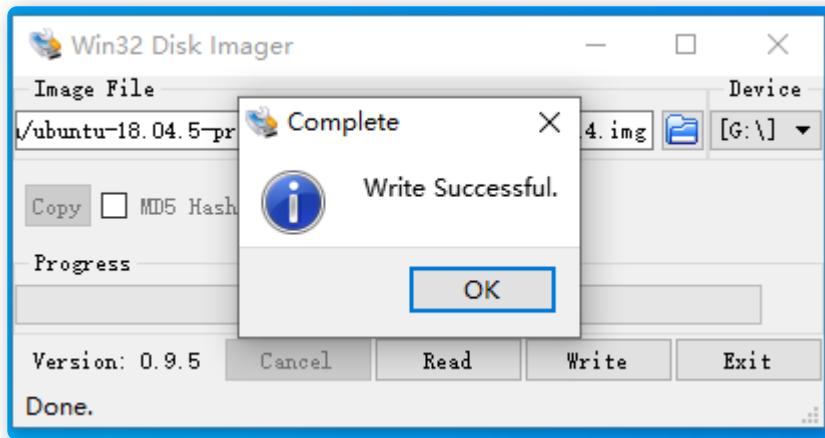
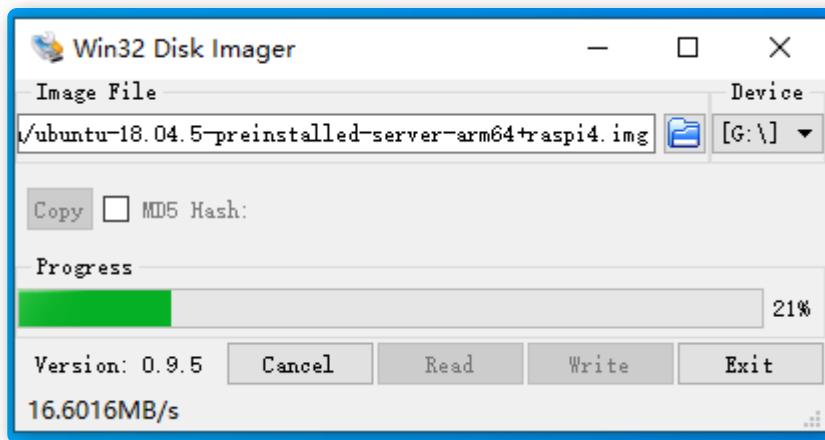
1.3系统烧录

1.将TF卡插入读卡器，读卡器插入计算机；

2.如果TF卡已有内容，请先格式化(如无数据，此步骤略过)；



3.启动win32 Disk Imager，选择先行下载的Ubuntu18.04镜像并写入TF卡；



1.4系统安装

1.系统启动以及登录

取下TF卡插入树莓派，连接网线，启动树莓派，启动时是命令行界面，登陆使用默认账号密码，

默认账号:ubuntu

默认密码:ubuntu

还需要根据提示修改密码。

更改密码后，系统安装完毕，不过此时是命令行式操作，下一步需要安装桌面。

2.桌面安装

为了安装方便，建议使用ssh远程登录(需要先安装ssh，可以参考:8.5.3)

首先，调用命令: ifconfig 获取树莓派的 ip 地址；

然后，远程调用 ssh ubuntu@ip地址登录；

接下来，可以直接安装桌面，但是为了提高安装效率，建议更换下载源，使用国内资源:

阿里云源

```
1 deb https://mirrors.aliyun.com/ubuntu-ports/ disco
  main restricted universe multiverse
2 deb-src https://mirrors.aliyun.com/ubuntu-ports/
  disco main restricted universe multiverse
3 deb https://mirrors.aliyun.com/ubuntu-ports/
  disco-security main restricted universe multiverse
4 deb-src https://mirrors.aliyun.com/ubuntu-ports/
  disco-security main restricted universe multiverse
5 deb https://mirrors.aliyun.com/ubuntu-ports/
  disco-updates main restricted universe multiverse
6 deb-src https://mirrors.aliyun.com/ubuntu-ports/
  disco-updates main restricted universe multiverse
7 deb https://mirrors.aliyun.com/ubuntu-ports/
  disco-backports main restricted universe
  multiverse
8 deb-src https://mirrors.aliyun.com/ubuntu-ports/
  disco-backports main restricted universe
  multiverse
9 deb https://mirrors.aliyun.com/ubuntu-ports/
  disco-proposed main restricted universe multiverse
10 deb-src https://mirrors.aliyun.com/ubuntu-ports/
  disco-proposed main restricted universe multiverse
```

中科大源

```
1 deb https://mirrors.ustc.edu.cn/ubuntu-ports/
  disco main restricted universe multiverse
2 deb-src https://mirrors.ustc.edu.cn/ubuntu-ports/
  disco main restricted universe multiverse
3 deb https://mirrors.ustc.edu.cn/ubuntu-ports/
  disco-updates main restricted universe multiverse
4 deb-src https://mirrors.ustc.edu.cn/ubuntu-ports/
  disco-updates main restricted universe multiverse
5 deb https://mirrors.ustc.edu.cn/ubuntu-ports/
  disco-backports main restricted universe
  multiverse
6 deb-src https://mirrors.ustc.edu.cn/ubuntu-ports/
  disco-backports main restricted universe
  multiverse
7 deb https://mirrors.ustc.edu.cn/ubuntu-ports/
  disco-security main restricted universe multiverse
8 deb-src https://mirrors.ustc.edu.cn/ubuntu-ports/
  disco-security main restricted universe multiverse
9 deb https://mirrors.ustc.edu.cn/ubuntu-ports/
  disco-proposed main restricted universe multiverse
10 deb-src https://mirrors.ustc.edu.cn/ubuntu-ports/
  disco-proposed main restricted universe multiverse
```

清华源

```

1 deb https://mirrors.tuna.tsinghua.edu.cn/ubuntu-
  ports/ disco main restricted universe multiverse
2 deb-src
  https://mirrors.tuna.tsinghua.edu.cn/ubuntu-ports/
  disco main restricted universe multiverse
3 deb https://mirrors.tuna.tsinghua.edu.cn/ubuntu-
  ports/ disco-updates main restricted universe
  multiverse
4 deb-src
  https://mirrors.tuna.tsinghua.edu.cn/ubuntu-ports/
  disco-updates main restricted universe multiverse
5 deb https://mirrors.tuna.tsinghua.edu.cn/ubuntu-
  ports/ disco-backports main restricted universe
  multiverse
6 deb-src
  https://mirrors.tuna.tsinghua.edu.cn/ubuntu-ports/
  disco-backports main restricted universe
  multiverse
7 deb https://mirrors.tuna.tsinghua.edu.cn/ubuntu-
  ports/ disco-security main restricted universe
  multiverse
8 deb-src
  https://mirrors.tuna.tsinghua.edu.cn/ubuntu-ports/
  disco-security main restricted universe multiverse
9 deb https://mirrors.tuna.tsinghua.edu.cn/ubuntu-
  ports/ disco-proposed main restricted universe
  multiverse
10 deb-src
  https://mirrors.tuna.tsinghua.edu.cn/ubuntu-ports/
  disco-proposed main restricted universe multiverse

```

修改/etc/apt/sources.list文件，将上述资源的任意一个复制进文件。

```
1 sudo nano /etc/apt/sources.list
```

最后，安装桌面环境（可选择：xubuntu-desktop、lubuntu-desktop、kubuntu-desktop）

```
1 sudo apt-get install ubuntu-desktop
```

3.重启桌面安装完毕

4.同步时间

默认情况下，树莓派系统时间是格林威治时间，而我们处于东八区，相差八个小时，需要将时间，设置为北京时间。

在/etc/profile文件中增加一行`export TZ='CST-8'`，并使文件立即生效，执行命令：

`source /etc/profile`或者`. /etc/profile`。

2.ROS安装

在树莓派上安装ROS与PC上安装流程类似：

1.配置软件与更新

首先打开“软件和更新”对话框，具体可以在Ubuntu搜索按钮中搜索。打开并配置（确保勾选了"restricted"，"universe"，"和"multiverse."），可参考PC实现。

2.设置安装源

官方默认安装源：

```
1 sudo sh -c 'echo "deb
http://packages.ros.org/ros/ubuntu \$ (lsb_release -
sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```

或来自国内中科大的安装源

```
1 sudo sh -c '. /etc/lsb-release && echo "deb
http://mirrors.ustc.edu.cn/ros/ubuntu/ `lsb_release
-sc` main" > /etc/apt/sources.list.d/ros-
latest.list'
```

或来自国内清华的安装源

```
1 sudo sh -c '.
 /etc/lsb-release && echo "deb
 http://mirrors.tuna.tsinghua.edu.cn/ros/ubuntu/
 `lsb_release -cs` main" >
 /etc/apt/sources.list.d/ros-latest.list'
```

PS:回车后,可能需要输入管理员密码

3.设置key

```
1 sudo apt-key adv --keyserver
 'hkp://keyserver.ubuntu.com:80' --recv-key
 C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
```

4.安装

首先需要更新 apt(以前是 apt-get, 官方建议使用 apt 而非 apt-get),apt 是用于从互联网仓库搜索、安装、升级、卸载软件或操作系统的工具。

```
1 sudo apt update
```

等待...

然后, 再安装所需类型的 ROS:ROS 多个类型:**Desktop-Full**、**Desktop**、**ROS-Base**。由于在分布式架构中, 树莓派担当角色较为简单, 在此选择 Desktop 或 ROS-Base 安装

```
1 sudo apt install ros-melodic-desktop
```

5.环境配置

配置环境变量, 方便在任意 终端中使用 ROS。

```
1 echo "source /opt/ros/melodic/setup.bash" >>
 ~/.bashrc
2 source ~/.bashrc
```

6.构建软件包的依赖关系

到目前为止，已经安装了运行核心ROS软件包所需的软件。要创建和管理您自己的ROS工作区，还需要安装其他常用依赖：

```
1 sudo apt install python-rosdep python-rosinstall
  python-rosinstall-generator python-wstool build-essential
```

安装并初始化rosdep，在使用许多ROS工具之前，需要初始化rosdep。rosdep使您可以轻松地为要编译的源安装系统依赖：

```
1 sudo apt install python-rosdep
```

使用以下命令，可以初始化rosdep。

```
1 sudo rosdep init
2 rosdep update
```

8.6 机器人平台设计之传感器

当前机器人平台使用的传感器主要有三种：编码器、激光雷达与相机。编码器主要用于测速实现，在之前已有详细介绍，不再赘述，本节主要介绍激光雷达与相机的使用。

8.6.1 传感器_激光雷达简介

激光雷达是现今机器人尤其是无人车领域及最重要、最关键也是最常见的传感器之一，是机器人感知外界的一种重要手段。

概念

激光雷达(LiDAR)，英文全称为：Light Detection And Ranging，即光探测与测量。

原理

激光雷达可以发射激光束，光束照射到物体上，再反射回激光雷达，可以通过三角法测距或TOF测距计算出激光雷达与物体的距离。甚至也可以通过测量反射回来的信号中的某些特性而确定物体特征，比如:物体的材质。

注意:如果物体表面光滑(比如镜子)，光束照射后产生镜面反射，可能无法捕获返回的激光而出现识别失误的情况。

优点

激光雷达在测距方面精准(激光雷达的测量精度可达厘米级)、高效，是机器人测距的不二之选。

- **具有极高的分辨率:**激光雷达工作于光学波段，频率比微波高2~3个数量级以上，因此，与微波雷达相比，激光雷达具有极高的距离分辨率、角分辨率和速度分辨率；
- **抗干扰能力强:**激光波长短，可发射发散角非常小 (μrad 量级) 的激光束，多路径效应小 (不会形成定向发射与微波或者毫米波产生多路径效应) ，可探测低空/超低空目标；
- **获取的信息量丰富:**可直接获取目标的距离、角度、反射强度、速度等信息，生成目标多维度图像；
- **可全天时工作:**激光主动探测，不依赖于外界光照条件或目标本身的辐射特性。它只需发射自己的激光束，通过探测发射激光束的回波信号来获取目标信息。

缺点

激光雷达虽然优点多多，但也存在一些局限性:

- **成本:**居高不下
- **环境:**易受天气影响(大雾、雨天、烟尘)
- **属性识别能力弱:**激光雷达的点云数据是物体的几何外形呈现，无法如同人类视觉一样，分辨物体的物理特征，比如:颜色、纹理...

分类

根据线束数量的多少，激光雷达可分为单线束激光雷达与多线束(4线、8线、16线、32线、64线)激光雷达。单线激光雷达扫描一次只产生一条扫描线，其所获得的数据为2D数据，因此无法区别有关目标物体的3D信息。多线束激光雷达就是将多个横向扫描结果纵向叠加，从而获得3D数据，当然，线束越多，纵向的垂直视野角度越大。

8.6.2 传感器_雷达使用

思岚A1激光雷达(下图)是一款性价比较高的单线激光雷达。



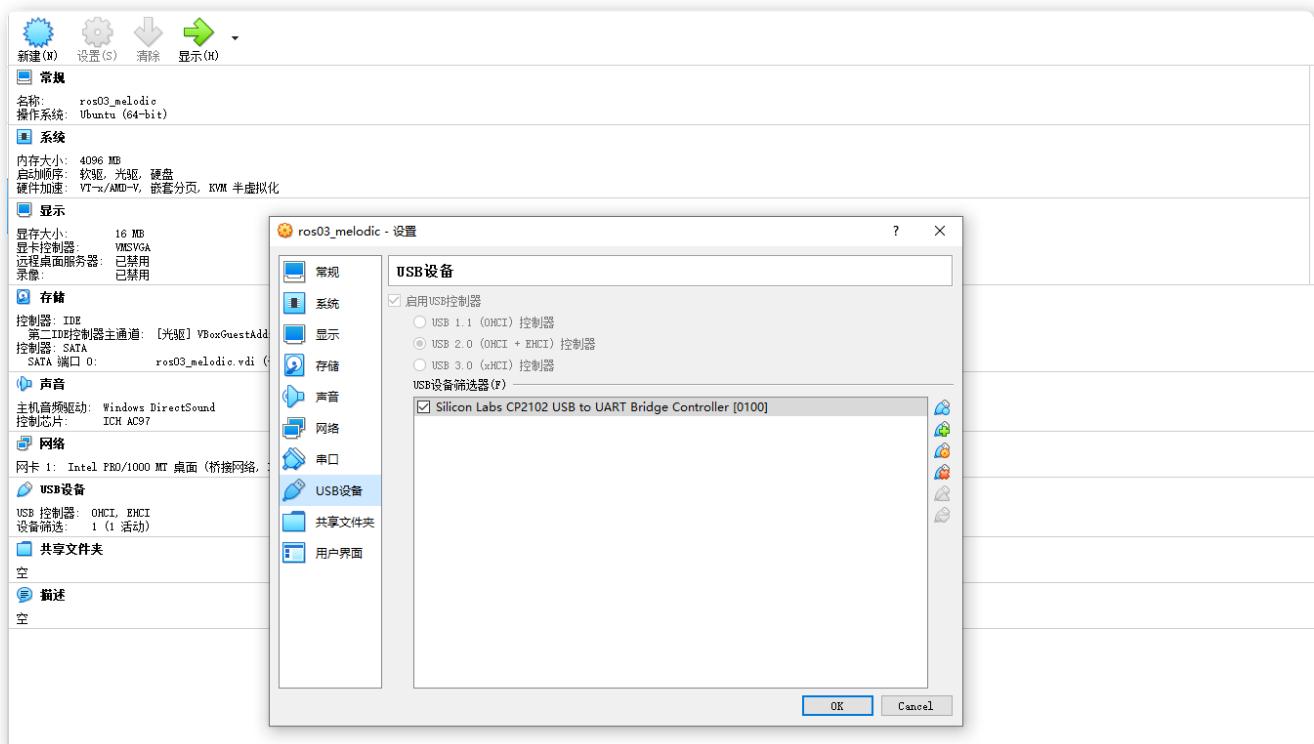
使用流程如下：

1. 硬件准备；
2. 软件安装；
3. 启动并测试。

1.硬件准备

1.雷达连接上位机

当前直接连接树莓派即可，如果连接的是虚拟机，注意VirtualBox或VMware的相关设置。



2.确认当前的 USB 转串口终端并修改权限

USB查看命令：

```
1 ll /dev/ttyUSB*
```

授权(将当前用户添加进dialout组，与arduino类似):

```
1 sudo usermod -a -G dialout your_user_name
```

不要忘记重启，重启之后才可以生效。

2.软件安装

进入工作空间的src目录，下载相关雷达驱动包，下载命令如下：

```
1 git clone https://github.com/slamtec/rplidar_ros
```

返回工作空间，调用 `catkin_make` 编译，并 `source ./devel/setup.bash`，为端口设置别名(将端口 `ttyUSBX` 映射到 `rplidar`)：

```
1 cd src/rplidar_ros/scripts/
2 ./create_udev_rules.sh
```

3.启动并测试

1.rplidar.launch文件准备

首先确认端口,编辑 `rplidar.launch` 文件

```
1 <launch>
2   <node name="rplidarNode"
3     pkg="rplidar_ros"  type="rplidarNode"
4     output="screen">
5     <param name="serial_port"           type="string"
6       value="/dev/rplidar"/>
7     <param name="serial_baudrate"      type="int"
8       value="115200"/><!--A1/A2 -->
9     <!--param name="serial_baudrate"    type="int"
10    value="256000"--><!--A3 -->
11     <param name="frame_id"           type="string"
12       value="laser"/>
13     <param name="inverted"          type="bool"
14       value="false"/>
15     <param name="angle_compensate"    type="bool"
16       value="true"/>
17   </node>
18 </launch>
```

frame_id 也可以修改，当使用URDF显示机器人模型时，需要与 URDF 中雷达 id 一致

2.终端中执行 launch 文件

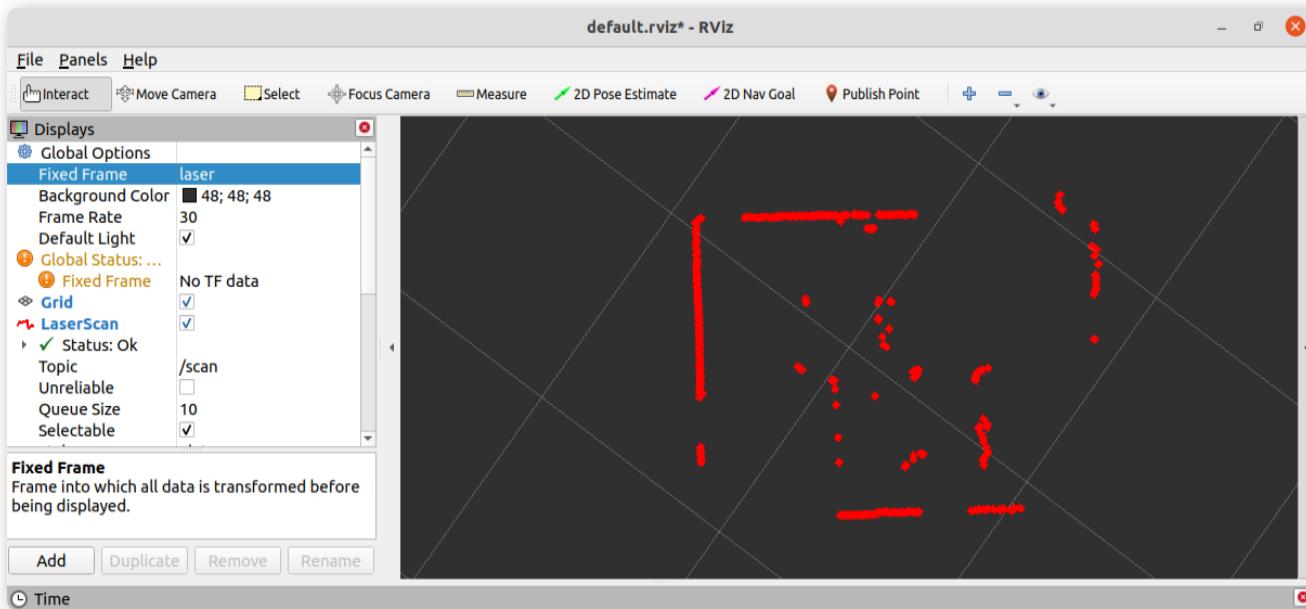
终端工作空间下输入命令：

```
1 rosrun rplidar_ros rplidar.launch
```

如果异常，雷达开始旋转

3.rviz中订阅雷达相关消息

启动 rviz，添加 LaserScan 插件：



注意: Fixed Frame 设置需要参考 rplidar.launch 中设置的 frame_id，Topic 一般设置为 /scan，Size 可以自由调整。

8.6.3 传感器_相机简介

相机是机器人系统中另一比较重要的传感器，与雷达类似的，相机也是机器人感知外界环境的重要手段之一，并且随着机器视觉、无人驾驶等技术的兴起，相机在物体识别、行为识别、SLAM中等都有着广泛的应用。

根据工作原理的差异可以将相机大致划分成三类:单目相机、双目相机与深度相机。

1. 单目相机

单目相机是将三维世界二维化，它是将拍摄场景在相机的成像平面上留下一个投影，静止状态下是无法通过单目相机确定深度信息。向如下动图展示的一样，在二维图形中，甚至不能根据图片中物体的大小来判断物体距离。



2. 双目相机

识破上面的“骗局”只需要移动单目相机，再换一个角度拍摄一张照片即可，当角度切换后，可以将两张照片组合还原为一个立体的三维世界。

双目相机的原理也是如此，双目相机是由两个单目相机组成的，即便在静止状态下，也可以生成两张图片，两个单目相机之间存在一定的距离也称之为基线，通过这个基线以及两个单目项目分别生成的图片，可以来估算每个象素的空间位置。

3. 深度相机

深度相机也称之为RGB-D相机，顾名思义，深度相机也可以用于获取物体深度信息。深度相机一般基于结构光或ToF(Time-of-Flight)原理实现测距。

前者是通过近红外激光器，将具有一定结构特征的光线投射到被拍摄物体上，再由专门的红外摄像头进行采集。光线照射到不同深度的物体上时，会采集到不同的图像相位信息，然后通过运算单元将这种结构的变化换算成深度信息，后者实现则类似于激光雷达，也是根据光线的往返时间来计算深度信息。

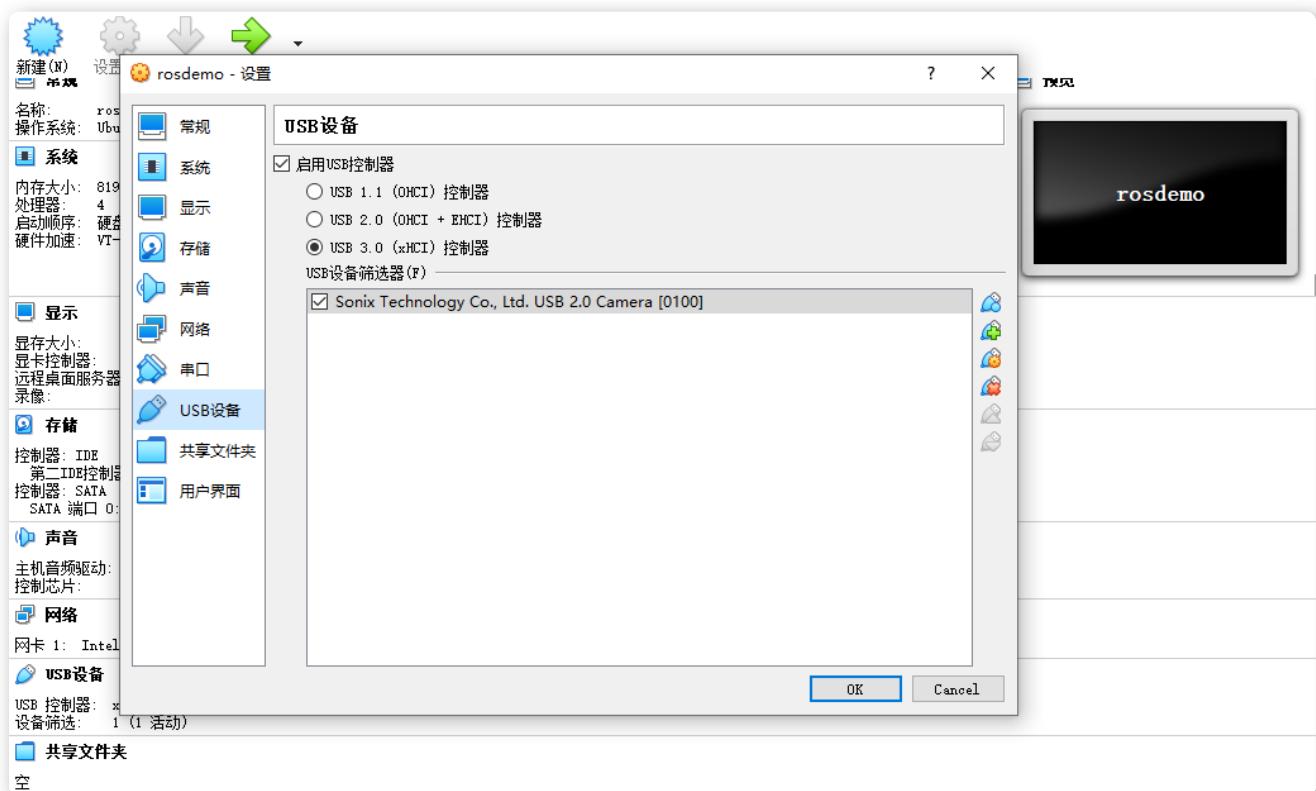
8.6.4 传感器_相机使用

使用流程如下：

1. 硬件准备；
2. 软件安装；
3. 启动并测试。

1. 硬件准备

当前直接连接树莓派即可，如果连接的是虚拟机，注意VirtualBox或VMware的相关设置。



2. 软件准备

安装USB摄像头软件包，命令如下：

```
1 sudo apt-get install ros-ROS版本-usb-cam
```

或者也可以从github直接下载源码：

```
1 git clone https://github.com/ros-drivers/usb_cam.git
```

3. 测试

1. launch文件准备

在软件包中内置了测试用的launch文件，内容如下：

```

1 <launch>
2   <node name="usb_cam" pkg="usb_cam"
3     type="usb_cam_node" output="screen" >
4     <param name="video_device" value="/dev/video0"
5   />
6     <param name="image_width" value="640" />
7     <param name="image_height" value="480" />
8     <param name="pixel_format" value="yuyv" />
9     <param name="camera_frame_id" value="usb_cam"
10    />
11    <param name="io_method" value="mmap"/>
12  </node>
13  <node name="image_view" pkg="image_view"
14    type="image_view" respawn="false" output="screen">
15    <remap from="image" to="/usb_cam/image_raw"/>
16    <param name="autosize" value="true" />
17  </node>
18 </launch>

```

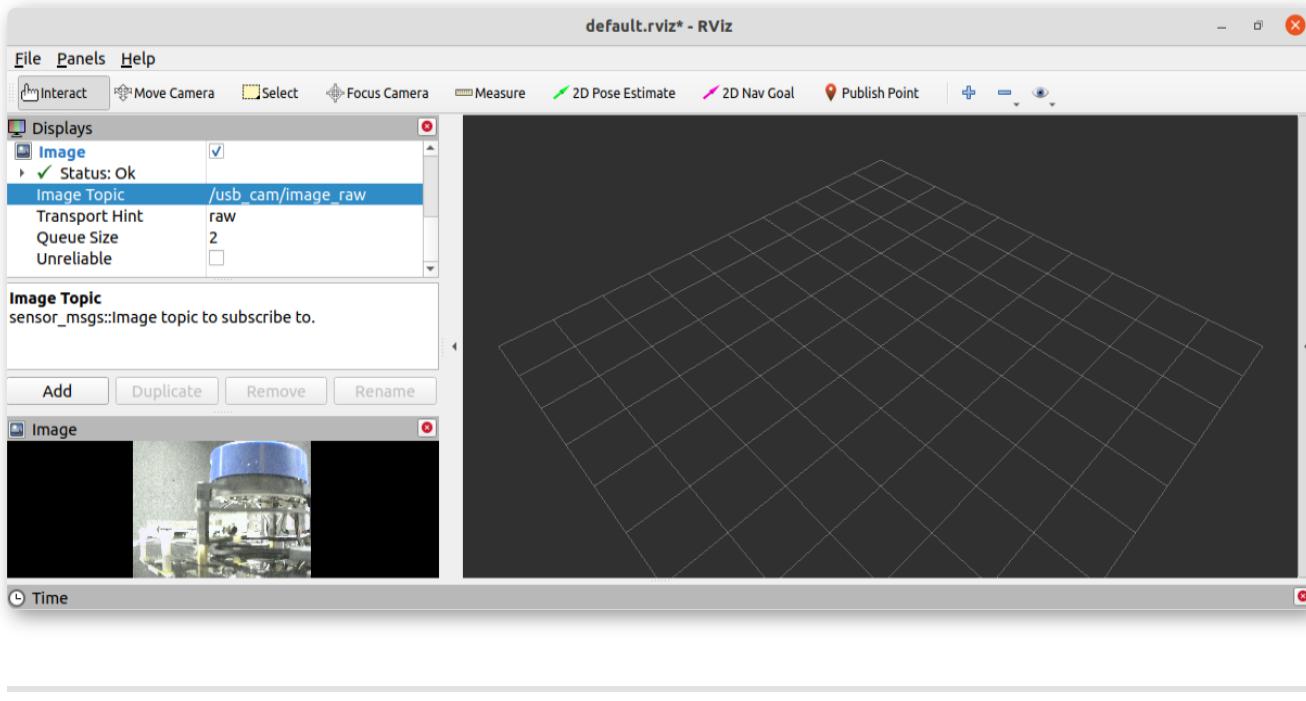
节点 `usb_cam` 用于启动相机，节点 `image_view` 以图形化窗口的方式显示图像数据，需要查看相机的端口并修改 `usb_cam` 中的 `video_device` 参数，并且如果将摄像头连接到了树莓派，且通过 `ssh` 远程访问树莓派的话，需要注释 `image_view` 节点，因为在终端中无法显示图形化界面。

2. 启动launch文件

```
1 roslaunch usb_cam usb_cam-test.launch
```

3.rviz显示

启动 rviz，添加 LaserScan 插件：



8.6.5 传感器_集成

之前已经分别介绍了底盘、雷达、相机等相关节点的安装、配置以及使用，不过之前的实现还存在一些问题：



1. 机器人启动时，需要逐一启动底盘控制、相机与激光雷达，操作冗余；
2. 如果只是简单的启动这些节点，那么在 rviz 中显示时，会发现出现了 TF 转换异常，比如参考坐标系设置为 odom 时，雷达信息显示失败。

本节将介绍如何把传感器(激光雷达与相机)集成以解决上述问题，所谓集成主要是优化底盘、雷达、相机相关节点的启动并通过坐标变换实现机器人底盘与里程计、雷达和相机的关联，实现步骤如下：

1. 编写用于集成的 launch 文件；
2. 发布TF坐标变换；
3. 启动并测试。

1.launch文件

新建功能包:

```
1 catkin_create_pkg mycar_start roscpp rospy std_msgs
  ros_arduino_python usb_cam rplidar_ros
```

功能包下创建launch文件夹，launch文件夹中新建launch文件，文件名自定义。

内容如下:

```
1 <!-- 机器人启动文件:
2           1.启动底盘
3           2.启动激光雷达
4           3.启动摄像头
5     -->
6 <launch>
7   <include file="$(find
8     ros_arduino_python)/launch/arduino.launch" />
9   <include file="$(find
10    usb_cam)/launch/usb_cam-test.launch" />
11   <include file="$(find
12     rplidar_ros)/launch/rplidar.launch" />
13 </launch>
```

2.坐标变换

如果启动时加载了机器人模型，且模型中设置的坐标系名称与机器人实体中设置的坐标系一致，那么可以不再添加坐标变换，因为机器人模型可以发布坐标变换信息，如果没有启动机器人模型，就需要自定义坐标变换实现了，继续新建launch文件。

内容如下:

```

1 <!-- 机器人启动文件:
2           当不包含机器人模型时, 需要发布坐标变换
3 -->
4
5 <launch>
6   <include file="$(find
7     mycar_start)/launch/start.launch" />
8   <node name="camera2basefootprint" pkg="tf2_ros"
9     type="static_transform_publisher" args="0.08 0 0.1 0
  0 0 /base_footprint /camera_link"/>
10  <node name="rplidar2basefootprint" pkg="tf2_ros"
11    type="static_transform_publisher" args="0 0 0.1 0 0
  0 /base_footprint /laser"/>
12 </launch>

```

3. 测试

最后, 就可以启动PC端与树莓派端相关节点并运行查看结果了:

1. 树莓派

直接执行上一步的机器人启动launch文件:

```
1 roslaunch 自定义包 自定义launch文件
```

2. PC端

启动键盘控制节点:

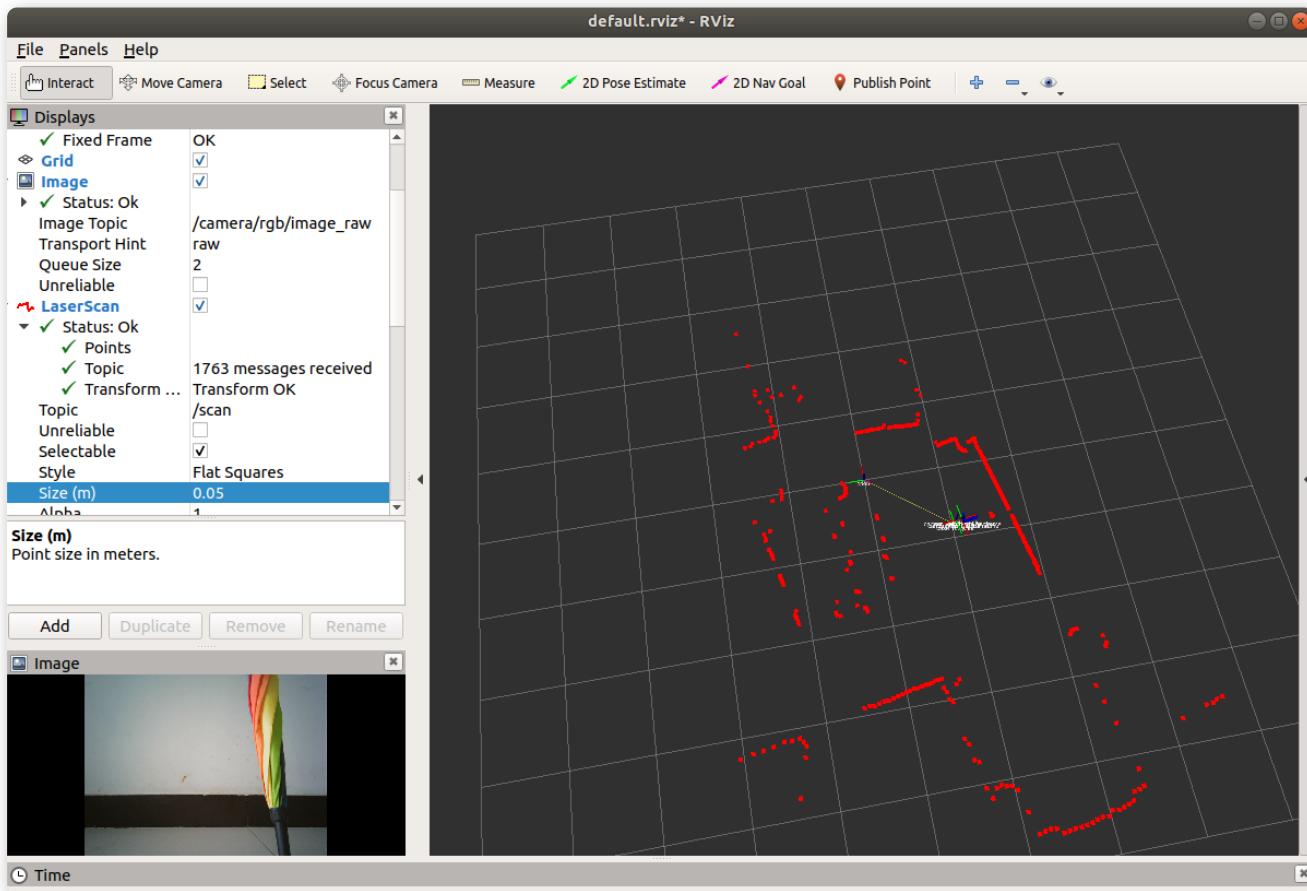
```
1 rosrun teleop_twist_keyboard
  teleop_twist_keyboard.py
```

还需要启动rviz:

```
1 rviz
```

3.结果显示

在rviz中添加laserscan、image等插件，并通过键盘控制机器人运动，查看rviz中的显示结果：



8.7 本章小结

本章从0到1的介绍了如何构建低成本、实验性的机器人平台，主要内容是围绕机器人的组成展开的，也即：执行机构、驱动系统、控制系统、传感系统。

我们也主要围绕这几个方面做一下总结：

执行机构

执行机构是纯硬件实现，在我们的机器人平台中，主要是机器人的行走部分，行走部分的核心是电机，电机的一些参数以及不同参数之间的换算是需要了解的。

驱动系统

驱动系统我们采用的是简单、易上手的Arduino再结合电机驱动模块，主要介绍了arduino的基本使用，并通过ros_arduino_bridge搭建了机器人底盘，该底盘可以解析速度消息并转换成控制电机运动的PWM信号，还可以发布里程计消息。

控制系统

控制系统是通过PC与树莓派多处理器结合的方式来实现的，PC扮演了监控的角色，而树莓派则担当数据下发与采集的角色，具体介绍了PC与树莓派的分布式框架实现、如何通过SSH实现远程登陆以及ros_arduino_bridge在树莓派上部署。

传感系统

传感系统则介绍了机器人中一些常用的传感器的相关内容，其中，在驱动系统实现时，就涉及到了内部传感器编码器的工作原理以及使用，最后机器人系统集成时又介绍了相机与激光雷达的概念以及应用。

本章内容最终结果就是搭建了一个机器人平台，并且安装、调试了各个组成模块，下一章开始我们将基于这个机器人平台整合各个模块并实现导航功能。

第9章 机器人导航(实体)

第7章内容介绍了在仿真环境下的机器人导航，第8章内容介绍了两轮差速机器人的软硬件实现，如果你已经按照第8章内容构建了自己的机器人平台，那么就可以尝试将仿真环境下的导航功能迁移到机器人实体了，本章就主要介绍该迁移过程的实现，学习内容如下：

- 比较仿真环境与真实环境的导航实现；
- 介绍VScode远程开发的实现；
- 实体机器人导航实现。

预期达成学习目标：

- 了解仿真环境与真实环境下导航实现的区别；
- 能够搭建VScode远程开发环境；
- 能够实现实体机器人的建图与导航。

9.1 概述

实体机器人导航与仿真环境下的导航核心实现基本一致，主要区别在于导航实现之前，基本环境的搭建有所不同，比如：导航场景、传感器、机器人模型等，大致区别如下：

	仿真环境	实体机器人
导航场景	依赖于gazebo搭建的仿真环境	依赖于现实环境
传感器	在gazebo中通过插件来模拟一些列传感器，比如：雷达、摄像头、编码器....	使用的是真实的传感器
机器人	依赖于机器人模型，以实现仿真环境下的机器人的显示。通过	机器人模型不是必须的，如果不使用机器人可以通过static_transform_publisher发布导航必须的坐标变换；如果使用机器

**人
模
型**

仿真环境`ate_publisher`、`joint_state_publisher`实现机器人各部件的坐标变换。

人模型也可以借助实体机器人`robot_state_publisher`、`joint_state_publisher`发布坐标变换，当然后者可以在rviz中显示机器人模型，更友好。

总体而言，在实体机器人导航中，可以完全脱离gazebo，而机器人模型是否使用，可以按需选择。

除此之外，实体机器人导航的开发环境也发生了改变，程序最终需要被部署在机器人端，我们可以在本地开发，然后通过ssh上传至机器人，或者也可以通过VScode的远程开发插件直接在机器人端编写程序。

9.2 VScode远程开发

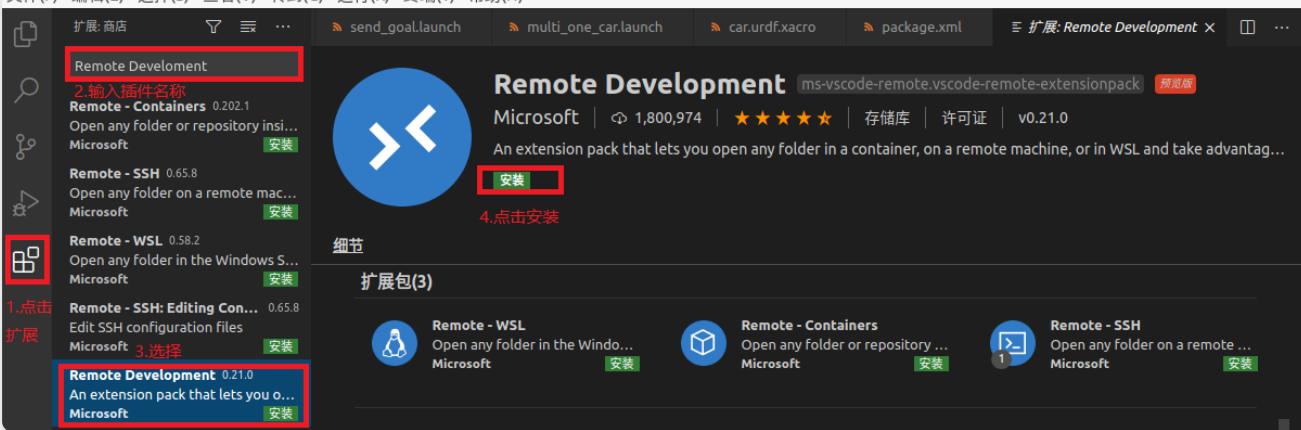
在第8章，我们介绍了ssh远程连接的使用，借助于ssh可以远程操作树莓派端，但是也存在诸多不便，比如：编辑文件内容时，需要使用vi编辑器，且在一个终端内，无法同时编辑多个文件，本节将介绍一较为实用的功能——VSCode远程开发，我们可以在VScode中以图形化的方式在树莓派上远程开发程序，比ssh的使用更方便快捷，可以大大提高程序开发效率。

1.准备工作

VScode远程开发依赖于ssh，请首先按照8.5.3内容配置ssh远程连接。

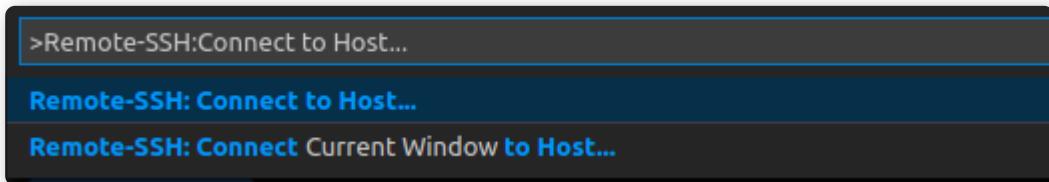
2.为VScode安装远程开发插件

启动VScode，首先点击侧边栏的扩展按钮，然后在“扩展：商店”的搜索栏输入“Remote Development”并点击同名插件，最后在右侧显示区中点击“安装”。

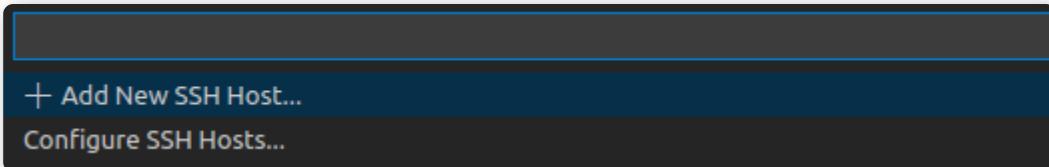


3.配置远程连接

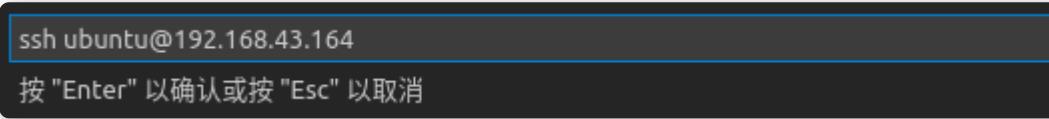
步骤1：使用快捷键 `ctrl + shift + p` 打开命令输入窗口，并输入 `Remote-SSH:Connect to Host...`，弹出列表中选择与之同名的选项。



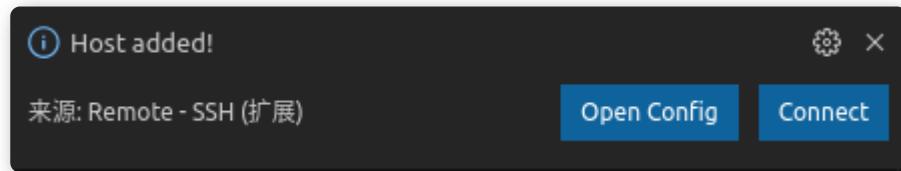
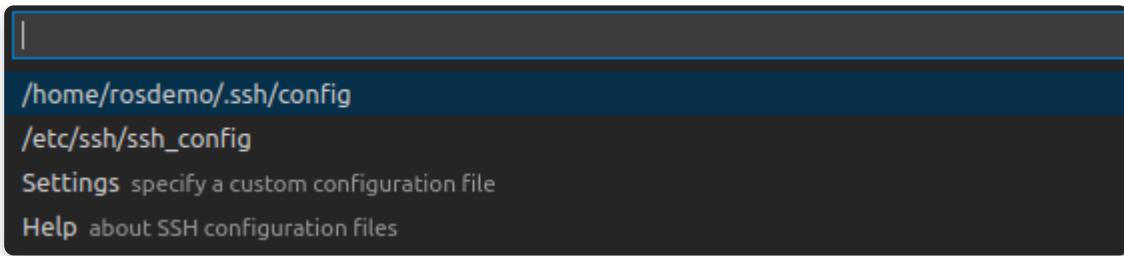
步骤2：步骤1完成将弹出一个新的命令窗口如下，选择下拉列表中的 `Add New SSH Host`。



步骤3：步骤2完成又将弹出一个新的命令窗口，在其中输入：`ssh ubuntu@192.168.43.164`，其中，`ubuntu` 需要替换为你的登录账号，`192.18.43.164` 则替换为你的树莓派的ip地址。

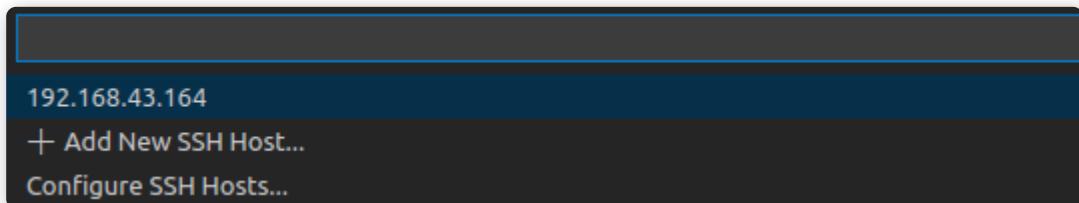


步骤4：选择步骤3完成后的弹窗列表中的第一个选项(或直接回车)，即可完成配置，配置成果后会有提示信息。

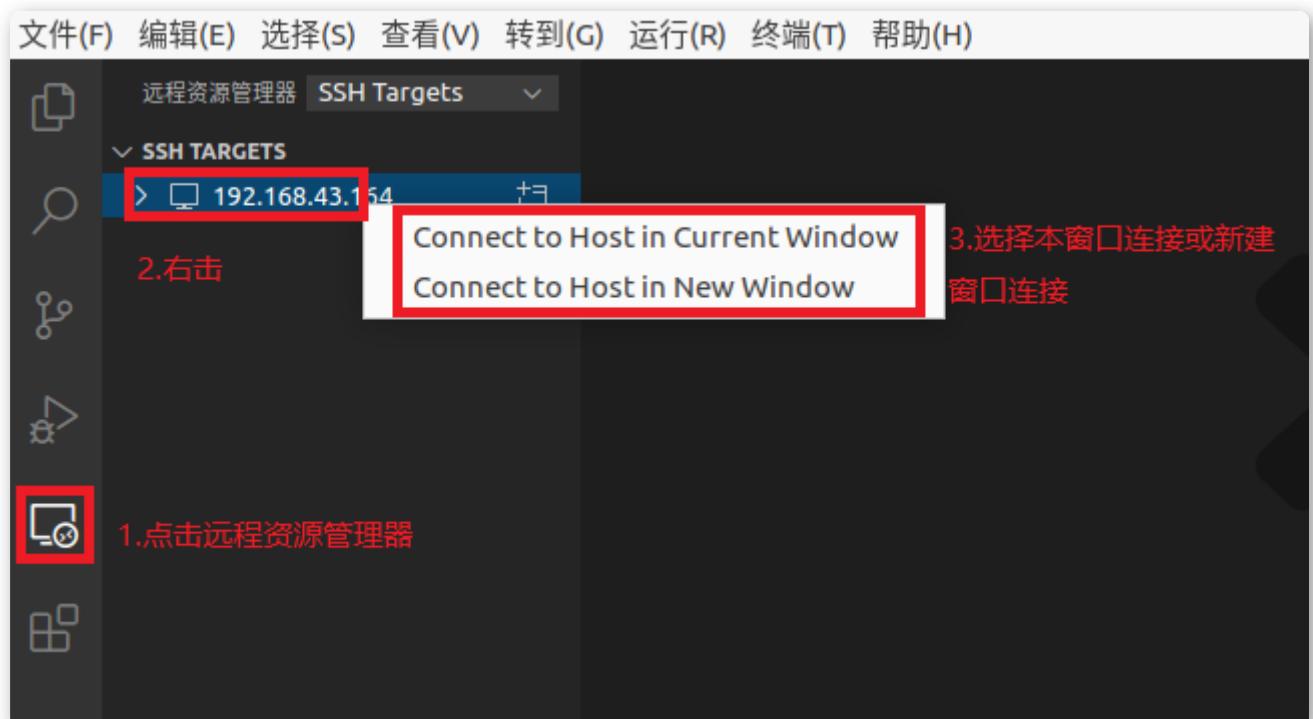


4. 使用

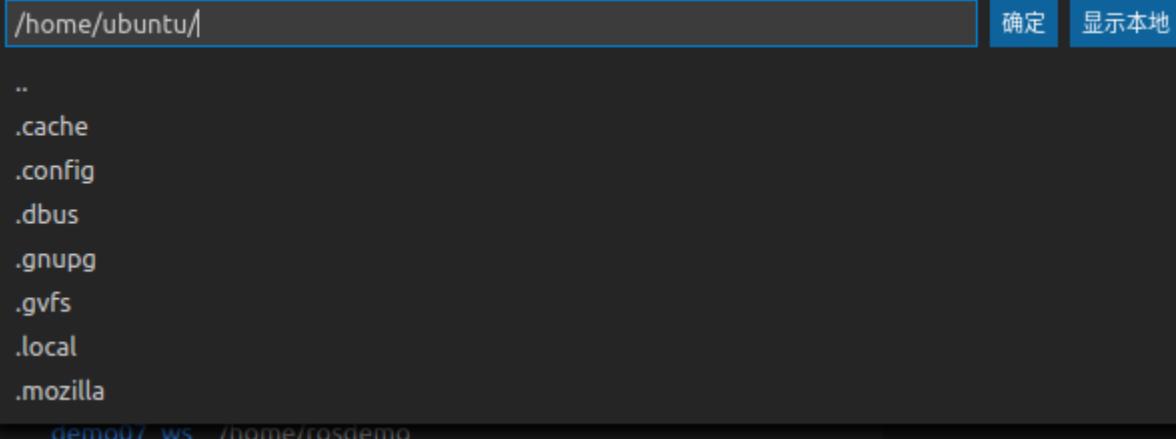
步骤1：继续使用快捷键 `ctrl + shift + p` 打开命令输入窗口，并输入 `Remote-SSH:Connect to Host...`，此时列表中将显示步骤3中配置的ip地址，直接选择，选择后，VScode将打卡一个新的窗口。



或者，也可以点击侧边栏的 `远程资源管理器`，在弹出的服务器列表中选择要连接的服务器，并右击，选择在本窗口或新窗口中实现远程连接。



步骤2：选择菜单栏的 `文件` 下的 `打开文件夹`，在弹窗列表中选择需要打开的文件夹并点击确定即可。



最终，我们就可以像操作本地文件一样实现远程开发了。

9.3 导航实现

本节介绍实体机器人导航的基本实现流程。该流程实现与7.2节内容类似，主要内容仍然集中在SLAM、地图服务、定位与路径规划，本节内容不再重复介绍7.2节中各个知识点的实现细节，而是注重知识点应用。

实体机器人导航实现流程如下：

1. 准备工作；
2. SLAM实现；
3. 地图服务；
4. 定位实现；
5. 路径规划；

本节最后还会将导航与SLAM结合，实现自主移动的SLAM建图。

9.3.1 导航实现01_准备工作

1.1 分布式架构

分布式架构搭建完毕且能正常运行，在PC端可以远程登陆机器人端。

1.2 功能包安装

在机器人端安装导航所需功能包：

- 安装 gmapping 包(用于构建地图): `sudo apt install ros-<ROS版本>-gmapping`
- 安装地图服务包(用于保存与读取地图): `sudo apt install ros-<ROS版本>-map-server`
- 安装 navigation 包(用于定位以及路径规划): `sudo apt install ros-<ROS版本>-navigation`

新建功能 (包名自定义, 比如: nav) , 并导入依赖: gmapping
map_server amcl move_base

1.3 机器人模型以及坐标变换

机器人的不同部件有不同的坐标系, 我们需要将这些坐标系集成进同一坐标树, 实现方案有两种:

1. 不同的部件相对于机器人底盘其位置都是固定的, 可以通过发布静态坐标变换以实现集成;
2. 可以通过加载机器人URDF文件结合 `robot_state_publisher`、`joint_state_publisher`实现不同坐标系的集成。

方案1在上一章中已做演示, 接下来介绍方案2的实现。

1.3.1 创建机器人模型相关的功能包

创建功能包: `catkin_create_pkg mycar_description urdf xacro`。

1.3.2 准备机器人模型文件

在功能包下新建 urdf 目录, 编写具体的 urdf 文件 (机器人模型相关 URDF文件的编写可以参考第6章内容) , 示例如下:

文件car.urdf.xacro用于集成不同的机器人部件, 内容如下:

```

1 <robot name="mycar"
  xmlns:xacro="http://wiki.ros.org/xacro">
2
3   <xacro:include filename="car_base.urdf.xacro" />
4   <xacro:include filename="car_camera.urdf.xacro" />
5   <xacro:include filename="car_laser.urdf.xacro" />
6
7 </robot>

```

文件car_base.urdf.xacro机器人底盘实现，内容如下：

```

1 <robot name="mycar"
  xmlns:xacro="http://wiki.ros.org/xacro">
2
3   <xacro:property name="footprint_radius"
  value="0.001" />
4   <link name="base_footprint">
5     <visual>
6       <geometry>
7         <sphere
  radius="${footprint_radius}" />
8         </geometry>
9       </visual>
10      </link>
11
12
13   <xacro:property name="base_radius" value="0.1"
  />
14   <xacro:property name="base_length"
  value="0.08" />
15   <xacro:property name="lidi" value="0.015" />
16   <xacro:property name="base_joint_z"
  value="${base_length / 2 + lidi}" />
17   <link name="base_link">
18     <visual>
19       <geometry>

```

```
20                     <cylinder radius="0.1"
21             length="0.08" />
22         </geometry>
23
24         <origin xyz="0 0 0" rpy="0 0 0" />
25
26         <material name="baselink_color">
27             <color rgba="1.0 0.5 0.2 0.5" />
28         </material>
29     </visual>
30
31 </link>
32
33 <joint name="link2footprint" type="fixed">
34     <parent link="base_footprint" />
35     <child link="base_link" />
36     <origin xyz="0 0 0.055" rpy="0 0 0" />
37 </joint>
38
39
40     <xacro:property name="wheel_radius"
41         value="0.0325" />
42     <xacro:property name="wheel_length"
43         value="0.015" />
44     <xacro:property name="PI" value="3.1415927" />
45     <xacro:property name="wheel_joint_z"
46         value="${(base_length / 2 + lidi - wheel_radius) *
47             -1}" />
48
49
50     <xacro:macro name="wheel_func"
51         params="wheel_name flag">
52         <link name="${wheel_name}_wheel">
53             <visual>
54                 <geometry>
```

```

51          <cylinder
52              radius="${wheel_radius}" length="${wheel_length}"
53          />
54          </geometry>
55
56          <origin xyz="0 0 0" rpy="${PI / 2}
57              0 0" />
58
59          <material name="wheel_color">
60              <color rgba="0 0 0 0.3" />
61          </material>
62          </visual>
63
64      </link>
65
66
67      <joint name="${wheel_name}2link"
68          type="continuous">
69          <parent link="base_link" />
70          <child link="${wheel_name}_wheel" />
71
72          <origin xyz="0 ${0.1 * flag}
73              ${wheel_joint_z}" rpy="0 0 0" />
74          <axis xyz="0 1 0" />
75          </joint>
76
77
78      <xacro:macro>
79          <xacro:wheel_func wheel_name="left" flag="1"
80          />
81          <xacro:wheel_func wheel_name="right" flag="-1"
82          />
83
84
85          <xacro:property name="small_wheel_radius"
86          value="0.0075" />
87          <xacro:property name="small_joint_z"
88          value="${(base_length / 2 + lidi -
89          small_wheel_radius) * -1}" />

```

```

80
81      <xacro:macro name="small_wheel_func"
82          params="small_wheel_name flag">
83          <link name="${small_wheel_name}_wheel">
84              <visual>
85                  <geometry>
86                      <sphere
87                          radius="${small_wheel_radius}" />
88                  </geometry>
89
89
90                  <origin xyz="0 0 0" rpy="0 0 0" />
91
92          <material name="wheel_color">
93              <color rgba="0 0 0 0.3" />
94          </material>
95      </visual>
96
97      </link>
98
99      <joint name="${small_wheel_name}2link"
100         type="continuous">
101          <parent link="base_link" />
102          <child
103              link="${small_wheel_name}_wheel" />
104
105          <origin xyz="${0.08 * flag} 0
106              ${small_joint_z}" rpy="0 0 0" />
107              <axis xyz="0 1 0" />
108          </joint>
109
110      </xacro:macro >
111      <xacro:small_wheel_func
112          small_wheel_name="front" flag="1"/>
113      <xacro:small_wheel_func
114          small_wheel_name="back" flag="-1"/>
115
116      </robot>

```

文件car_camera.urdf.xacro机器人摄像头实现，内容如下：

```
1 <robot name="mycar"
  xmlns:xacro="http://wiki.ros.org/xacro">
2
3   <xacro:property name="camera_length"
  value="0.02" />
4   <xacro:property name="camera_width"
  value="0.05" />
5   <xacro:property name="camera_height"
  value="0.05" />
6   <xacro:property name="joint_camera_x"
  value="0.08" />
7   <xacro:property name="joint_camera_y"
  value="0" />
8   <xacro:property name="joint_camera_z"
  value="${base_length / 2 + camera_height / 2}" />
9
10  <link name="camera">
11    <visual>
12      <geometry>
13        <box size="${camera_length}
14          ${camera_width} ${camera_height}" />
15        </geometry>
16        <origin xyz="0 0 0" rpy="0 0 0" />
17        <material name="black">
18          <color rgba="0 0 0 0.8" />
19        </material>
20      </visual>
21    </link>
22
23    <joint name="camera2base" type="fixed">
24      <parent link="base_link" />
25      <child link="camera" />
26      <origin xyz="${joint_camera_x}
27        ${joint_camera_y} ${joint_camera_z}" rpy="0 0 0"
28        />
29    </joint>
```

```
27  
28 </robot>
```

文件car_laser.urdf.xacro机器人雷达实现，内容如下：

```
1 <robot name="mycar"  
2   xmlns:xacro="http://wiki.ros.org/xacro">  
3     <xacro:property name="support_radius"  
4       value="0.01" />  
5     <xacro:property name="support_length"  
6       value="0.15" />  
7     <xacro:property name="laser_radius"  
8       value="0.03" />  
9     <xacro:property name="laser_length"  
10    value="0.05" />  
11    <xacro:property name="joint_support_x"  
12      value="0" />  
13    <xacro:property name="joint_support_y"  
14      value="0" />  
15    <xacro:property name="joint_support_z"  
16      value="${base_length / 2 + support_length / 2}" />  
17    <xacro:property name="joint_laser_x" value="0"  
18  />  
19    <xacro:property name="joint_laser_y" value="0"  
20  />  
21    <xacro:property name="joint_laser_z"  
22      value="${support_length / 2 + laser_length / 2}"  
23  />  
24  
25    <link name="support">  
26      <visual>  
27        <geometry>
```

```

20             <cylinder
21             radius="${support_radius}"
22             length="${support_length}" />
23             </geometry>
24             <material name="yellow">
25               <color rgba="0.8 0.5 0.0 0.5" />
26             </material>
27           </visual>
28
29     </link>
30
31   <joint name="support2base" type="fixed">
32     <parent link="base_link" />
33     <child link="support"/>
34     <origin xyz="${joint_support_x}"
35       ${joint_support_y} ${joint_support_z}" rpy="0 0 0"
36     />
37     </joint>
38   <link name="laser">
39     <visual>
40       <geometry>
41         <cylinder radius="${laser_radius}"
42           length="${laser_length}" />
43         </geometry>
44         <material name="black">
45           <color rgba="0 0 0 0.5" />
46         </material>
47       </visual>
48
49     </link>
50
51   <joint name="laser2support" type="fixed">
52     <parent link="support" />
53     <child link="laser"/>
54     <origin xyz="${joint_laser_x}"
55       ${joint_laser_y} ${joint_laser_z}" rpy="0 0 0" />
56   </joint>
57 </robot>

```

1.3.3 在launch文件加载机器人模型

launch文件（文件名称自定义，比如：car.launch）内容示例如下：

```

1 <launch>
2   <param name="robot_description" command="$(find
  xacro)/xacro $(find
  mycar_description)/urdf/car.urdf.xacro" />
3   <node pkg="joint_state_publisher"
  name="joint_state_publisher"
  type="joint_state_publisher" />
4   <node pkg="robot_state_publisher"
  name="robot_state_publisher"
  type="robot_state_publisher" />
5 </launch>

```

为了使用方便，还可以将该文件包含进启动机器人的launch文件中，示例如下：

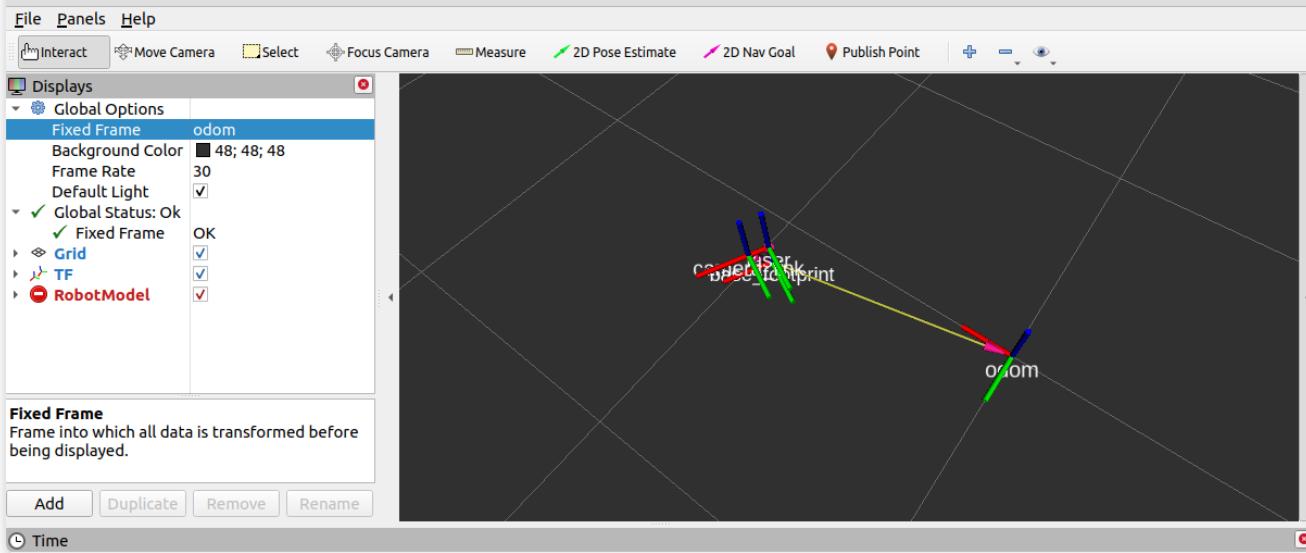
```

1 <launch>
2   <include file="$(find
  ros_arduino_python)/launch/arduino.launch" />
3   <include file="$(find
  usb_cam)/launch/usb_cam-test.launch" />
4   <include file="$(find
  rplidar_ros)/launch/rplidar.launch" />
5   <!-- 机器人模型加载文件 -->
6   <include file="$(find
  mycar_description)/launch/car.launch" />
7 </launch>

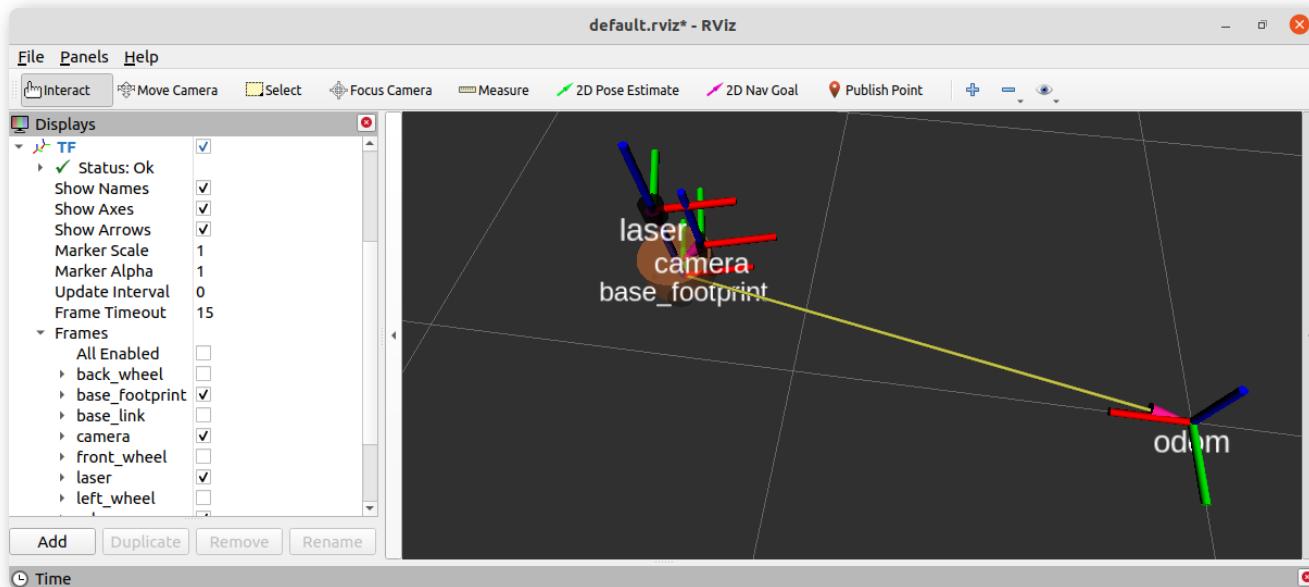
```

1.4结果演示

不使用机器人模型时，机器人端启动机器人(使用包含TF坐标换的launch文件)，从机端启动rviz，在rviz中添加RobotModel与TF组件，rviz中结果(此时显示机器人模型异常，且TF中只有代码中发布的坐标变换)：



使用机器人模型时，机器人端加载机器人模型（执行上一步的launch文件）且启动机器人，从机端启动rviz，，在rviz中添加RobotModel与TF组件rviz中结果(此时显示机器人模型，且TF坐标变换正常)：



后续，在导航时使用机器人模型。

9.3.2 导航实现02_SLAM建图

关于建图实现，仍然选用第7章中学习的：gmapping，实现如下：

2.1 编写gmapping节点相关launch文件

在上一节创建的导航功能包中新建 launch 目录，并新建launch文件（文件名自定义，比如：gmapping.launch），代码示例如下：

```

1 <launch>
2   <node pkg="gmapping" type="slam_gmapping"
3     name="slam_gmapping" output="screen">
4     <remap from="scan" to="scan"/>
5     <param name="base_frame"
6       value="base_footprint"/><!--底盘坐标系-->
7     <param name="odom_frame" value="odom"/> <!--
8       里程计坐标系-->
9     <param name="map_update_interval"
10    value="5.0"/>
11    <param name="maxUrange" value="16.0"/>
12    <param name="sigma" value="0.05"/>
13    <param name="kernelSize" value="1"/>
14    <param name="lstep" value="0.05"/>
15    <param name="astep" value="0.05"/>
16    <param name="iterations" value="5"/>
17    <param name="lsigma" value="0.075"/>
18    <param name="ogain" value="3.0"/>
19    <param name="lskip" value="0"/>
20    <param name="srr" value="0.1"/>
21    <param name="srt" value="0.2"/>
22    <param name="str" value="0.1"/>
23    <param name="stt" value="0.2"/>
24    <param name="linearUpdate" value="1.0"/>
25    <param name="angularUpdate" value="0.5"/>
26    <param name="temporalUpdate" value="3.0"/>
27    <param name="resampleThreshold"
28      value="0.5"/>
29    <param name="particles" value="30"/>
30    <param name="xmin" value="-50.0"/>
31    <param name="ymin" value="-50.0"/>
32    <param name="xmax" value="50.0"/>
33    <param name="ymax" value="50.0"/>

```

```

29      <param name="delta" value="0.05"/>
30      <param name="llsamplerange" value="0.01"/>
31      <param name="llsamplestep" value="0.01"/>
32      <param name="lasamplerange" value="0.005"/>
33      <param name="lasamplestep" value="0.005"/>
34  </node>
35 </launch>

```

关键代码解释：

```

1 <remap from="scan" to="scan"/><!-- 雷达话题 -->
2 <param name="base_frame" value="base_footprint"/><!--
-底盘坐标系-->
3 <param name="odom_frame" value="odom"/> <!--里程计坐
标系-->

```

2.2 执行

1. 执行相关launch文件，启动机器人并加载机器人模型： rosrun mycar_start start.launch；

2. 启动地图绘制的 launch 文件： rosrun nav gmapping.launch；

3. 启动键盘控制节点，用于控制机器人运动建图： rosrun teleop_twist_keyboard teleop_twist_keyboard.py

4. 在 rviz 中添加地图显示组件，通过键盘控制机器人运动，同时，在 rviz 中可以显示 gmapping 发布的栅格地图数据了，该显示结果与仿真环境下类似。下一步，还需要将地图单独保存。

9.3.3 导航实现03_地图服务

可以通过 map_server 实现地图的保存与读取。

3.1地图保存launch文件

首先在自定义的导航功能包下新建 map 目录，用于保存生成的地图数据。地图保存的语法比较简单，编写一个launch文件，内容如下：

```
1 <launch>
2   <arg name="filename" value="$(find nav)/map/nav"
3   />
4   <node name="map_save" pkg="map_server"
5   type="map_saver" args="-f $(arg filename)" />
6 </launch>
```

其中通过 filename 指定了地图的保存路径以及保存的文件名称。

SLAM建图完毕后，执行该launch文件即可。

测试：

首先，参考上一节，依次启动仿真环境，键盘控制节点与SLAM节点；

然后，通过键盘控制机器人运动并绘图；

最后，通过上述地图保存方式保存地图。

结果：在指定路径下会生成两个文件，xxx.pgm 与 xxx.yaml

3.2地图读取

通过 map_server 的 map_server 节点可以读取栅格地图数据，编写 launch 文件如下：

```

1 <launch>
2     <!-- 设置地图的配置文件 -->
3     <arg name="map" default="nav.yaml" />
4     <!-- 运行地图服务器，并且加载设置的地图-->
5     <node name="map_server" pkg="map_server"
6       type="map_server" args="$(find mycar_nav)/map/${arg
7       map})"/>
8 </launch>

```

其中参数是地图描述文件的资源路径，执行该launch文件，该节点会发布话题:map(nav_msgs/OccupancyGrid)，最后，在 rviz 中使用 map 组件可以显示栅格地图。

9.3.4 导航实现04_定位

在ROS的导航功能包集navigation中提供了 amcl 功能包，用于实现导航中的机器人定位。

4.1 编写amcl节点相关的launch文件

```

1 <launch>
2     <node pkg="amcl" type="amcl" name="amcl"
3       output="screen">
4         <!-- Publish scans from best pose at a max of
5         10 Hz -->
6         <param name="odom_model_type" value="diff"/>
7         <!-- 里程计模式为差分 -->
8         <param name="odom_alpha5" value="0.1"/>
9         <param name="transform_tolerance" value="0.2"
10        />
11        <param name="gui_publish_rate" value="10.0"/>
12        <param name="laser_max_beams" value="30"/>
13        <param name="min_particles" value="500"/>
14        <param name="max_particles" value="5000"/>
15        <param name="kld_err" value="0.05"/>
16        <param name="kld_z" value="0.99"/>
17        <param name="odom_alpha1" value="0.2"/>

```

```

14      <param name="odom_alpha2" value="0.2"/>
15      <!-- translation std dev, m -->
16      <param name="odom_alpha3" value="0.8"/>
17      <param name="odom_alpha4" value="0.2"/>
18      <param name="laser_z_hit" value="0.5"/>
19      <param name="laser_z_short" value="0.05"/>
20      <param name="laser_z_max" value="0.05"/>
21      <param name="laser_z_rand" value="0.5"/>
22      <param name="laser_sigma_hit" value="0.2"/>
23      <param name="laser_lambda_short" value="0.1"/>
24      <param name="laser_lambda_short" value="0.1"/>
25      <param name="laser_model_type"
26          value="likelihood_field"/>
27          <!-- <param name="laser_model_type"
28          value="beam"/> -->
29          <param name="laser_likelihood_max_dist"
30          value="2.0"/>
31          <param name="update_min_d" value="0.2"/>
32          <param name="update_min_a" value="0.5"/>
33
34          <param name="odom_frame_id" value="odom"/><!--
35          里程计坐标系 -->
36          <param name="base_frame_id"
37          value="base_footprint"/><!-- 添加机器人基坐标系 -->
38          <param name="global_frame_id" value="map"/><!--
39          - 添加地图坐标系 -->
40
41      </node>
42  </launch>

```

4.2 编写测试launch文件

amcl节点是不可以单独运行的，运行 amcl 节点之前，需要先加载全局地图，然后启动 rviz 显示定位结果，上述节点可以集成进launch文件，内容示例如下：

```

1 <launch>
2     <!-- 设置地图的配置文件 -->
3     <arg name="map" default="nav.yaml" />
4     <!-- 运行地图服务器，并且加载设置的地图-->
5     <node name="map_server" pkg="map_server"
6       type="map_server" args="$(find nav)/map/${arg
7       map})"/>
8     <!-- 启动AMCL节点 -->
9     <include file="$(find nav)/launch/amcl.launch"
  />
9 </launch>

```

当然，launch文件中地图服务节点和amcl节点中的包名、文件名需要根据自己的设置修改。

4.3执行

1.执行相关launch文件，启动机器人并加载机器人模型： rosrun mycar_start start.launch；

2.启动键盘控制节点： rosrun teleop_twist_keyboard teleop_twist_keyboard.py；

3.启动上一步中集成地图服务、amcl的launch文件： rosrun nav test_amcl.launch；

4.启动rviz并添加RobotModel、Map组件，分别显示机器人模型与地图，添加 posearray 插件，设置topic为particlecloud来显示 amcl 预估的当前机器人的位姿，箭头越是密集，说明当前机器人处于此位置的概率越高；

5.通过键盘控制机器人运动，会发现 posearray 也随之而改变。运行结果与仿真环境下类似。

9.3.5 导航实现05_路径规划

路径规划仍然使用 navigation 功能包集中的 move_base 功能包。

5.1 编写 launch 文件

关于 move_base 节点的调用，模板如下：

```
1 <launch>
2
3     <node pkg="move_base" type="move_base"
4       respawn="false" name="move_base" output="screen"
5       clear_params="true">
6         <rosparam file="$(find
7           nav)/param/costmap_common_params.yaml"
8           command="load" ns="global_costmap" />
9         <rosparam file="$(find
10          nav)/param/costmap_common_params.yaml"
11          command="load" ns="local_costmap" />
12         <rosparam file="$(find
13           nav)/param/local_costmap_params.yaml"
14           command="load" />
15         <rosparam file="$(find
16           nav)/param/global_costmap_params.yaml"
17           command="load" />
18         <rosparam file="$(find
19           nav)/param/base_local_planner_params.yaml"
20           command="load" />
21     </node>
22
23 </launch>
```

5.2 编写配置文件

可参考仿真实现。

1.costmap_common_params.yaml

该文件是move_base 在全局路径规划与本地路径规划时调用的通用参数，包括:机器人的尺寸、距离障碍物的安全距离、传感器信息等。配置参考如下:

```

1 #机器人几何参, 如果机器人是圆形, 设置 robot_radius, 如果是
2 #其他形状设置 footprint
3 robot_radius: 0.12 #圆形
4 # footprint: [[-0.12, -0.12], [-0.12, 0.12],
5 [0.12, 0.12], [0.12, -0.12]] #其他形状
6
7
8
9 #膨胀半径, 扩展在碰撞区域以外的代价区域, 使得机器人规划路径
10 #避开障碍物
11 inflation_radius: 0.2
12 #代价比例系数, 越大则代价值越小
13 cost_scaling_factor: 3.0
14
15 #地图类型
16 map_type: costmap
17 #导航包所需要的传感器
18 observation_sources: scan
19 #对传感器的坐标系和数据进行配置。这个也会用于代价地图添加和
20 #清除障碍物。例如, 你可以用激光雷达传感器用于在代价地图添加障
21 #碍物, 再添加kinect用于导航和清除障碍物。
22 scan: {sensor_frame: laser, data_type: LaserScan,
23 topic: scan, marking: true, clearing: true}

```

2.global_costmap_params.yaml

该文件用于全局代价地图参数设置:

```

1  global_costmap:
2    global_frame: map #地图坐标系
3    robot_base_frame: base_footprint #机器人坐标系
4    # 以此实现坐标变换
5
6    update_frequency: 1.0 #代价地图更新频率
7    publish_frequency: 1.0 #代价地图的发布频率
8    transform_tolerance: 0.5 #等待坐标变换发布信息的超时
9    time
10   static_map: true # 是否使用一个地图或者地图服务器来初
11   始化全局代价地图, 如果不使用静态地图, 这个参数为false.

```

3.local_costmap_params.yaml

该文件用于局部代价地图参数设置:

```

1  local_costmap:
2    global_frame: odom #里程计坐标系
3    robot_base_frame: base_footprint #机器人坐标系
4
5    update_frequency: 10.0 #代价地图更新频率
6    publish_frequency: 10.0 #代价地图的发布频率
7    transform_tolerance: 0.5 #等待坐标变换发布信息的超时
8    time
9    static_map: false #不需要静态地图, 可以提升导航效果
10   rolling_window: true #是否使用动态窗口, 默认为false,
11   在静态的全局地图中, 地图不会变化
12   width: 3 # 局部地图宽度 单位是 m
13   height: 3 # 局部地图高度 单位是 m
14   resolution: 0.05 # 局部地图分辨率 单位是 m, 一般与静态
15   地图分辨率保持一致

```

4.base_local_planner_params

基本的局部规划器参数配置，这个配置文件设定了机器人的最大和最小速度限制值，也设定了加速度的阈值。

```

1 TrajectoryPlannerROS:
2
3 # Robot Configuration Parameters
4   max_vel_x: 0.5 # X 方向最大速度
5   min_vel_x: 0.1 # X 方向最小速度
6
7   max_vel_theta: 1.0 #
8   min_vel_theta: -1.0
9   min_in_place_vel_theta: 1.0
10
11  acc_lim_x: 1.0 # X 加速限制
12  acc_lim_y: 0.0 # Y 加速限制
13  acc_lim_theta: 0.6 # 角速度加速限制
14
15 # Goal Tolerance Parameters, 目标公差
16  xy_goal_tolerance: 0.10
17  yaw_goal_tolerance: 0.05
18
19 # Differential-drive robot configuration
20 # 是否是全向移动机器人
21  holonomic_robot: false
22
23 # Forward Simulation Parameters, 前进模拟参数
24  sim_time: 0.8
25  vx_samples: 18
26  vtheta_samples: 20
27  sim_granularity: 0.05

```

5.3 launch文件集成

如果要实现导航，需要集成地图服务、amcl、move_base 等，集成示例如下：

```

1 <launch>
2     <!-- 设置地图的配置文件 -->
3     <arg name="map" default="nav.yaml" />
4     <!-- 运行地图服务器，并且加载设置的地图-->
5     <node name="map_server" pkg="map_server"
6       type="map_server" args="$(find nav)/map/${arg
7       map})"/>
8     <!-- 启动AMCL节点 -->
9     <include file="$(find nav)/launch/amcl.launch"
10    />
11
12 </launch>

```

5.4 测试

1. 执行相关launch文件，启动机器人并加载机器人模型： `roslaunch mycar_start start.launch`；

2. 启动导航相关的 launch 文件： `roslaunch nav nav.launch`；

3. 添加Rviz组件实现导航（参考仿真实现）。

9.3.6 导航与SLAM建图

与仿真环境类似的，也可以实现机器人自主移动的SLAM建图，步骤如下：

1. 编写launch文件，集成SLAM与move_base相关节点；
2. 执行launch文件并测试。

6.1 编写launch文件

当前launch文件（名称自定义，比如：auto_slam.launch）实现，无需调用map_server的相关节点，只需要启动SLAM节点与move_base节点，示例内容如下：

```

1 <launch>
2     <!-- 启动SLAM节点 -->
3     <include file="$(find
4         nav)/launch/gmapping.launch" />
5     <!-- 运行move_base节点 -->
6     <include file="$(find
7         nav)/launch/move_base.launch" />
8 </launch>

```

6.2 测试

1. 执行相关launch文件，启动机器人并加载机器人模型：roslaunch mycar_start start.launch；

2. 然后执行当前launch文件：roslaunch nav auto_slam.launch；

3. 在rviz中通过2D Nav Goal设置目标点，机器人开始自主移动并建图了；

4. 最后可以使用map_server保存地图。

9.4 本章小结

本章主要介绍关于实体机器人导航实现的，主要内容如下：

- 仿真环境与真实环境下的区别；
- VSCode远程开发环境的搭建；
- 实体机器人的导航实现。

整体而言，将仿真环境下的导航功能迁移到实体机器人并不复杂，不过在实际迁移时，需要调整导航相关的一些参数。

第 10 章 ROS进阶

在本教程的第二章内容介绍了ROS的核心实现:通信机制 ——话题通信、服务通信和参数服务器。三者结合可以满足ROS中的大多数数据传输相关的应用场景，但是在一些特定场景下可能就有些力不从心了，本章主要介绍之前的通信机制存在的问题以及对应的优化策略，本章主要内容如下：

- action通信；
- 动态参数；
- pluginlib；
- nodelet。

本章预期达成的学习目标：

- 了解服务通信应用的局限性(action的应用场景)，熟练掌握action的理论模型与实现流程；
- 了解参数服务器应用的局限性(动态配置参数的应用场景)，熟练掌握动态配置参数的实现流程；
- 了解插件的概念以及使用流程；
- 了解nodelet的应用场景以及使用流程。

10.1 action通信

关于action通信，我们先从之前导航中的应用场景开始介绍，描述如下：



机器人导航到某个目标点,此过程需要一个节点A发布目标信息，然后一个节点B接收到请求并控制移动，最终响应目标达成状态信息。

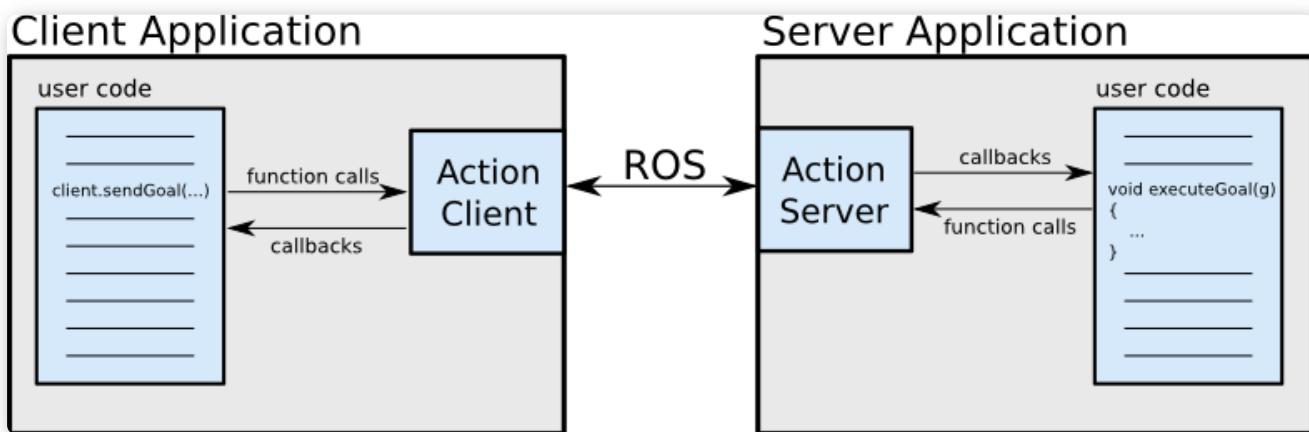
乍一看，这好像是服务通信实现，因为需求中要A发送目标，B执行并返回结果，这是一个典型的基于请求响应的应答模式，不过，如果只是使用基本的服务通信实现，存在一个问题：导航是一个过程，是耗时操作，如果使用服务通信，那么只有在导航结束时，才会产生响应结果，而在导航过程中，节点A是不会获取到任何反馈的，从而可能出现程序"假死"的现象，过程的不可控意味着不良的用户体验，以及逻辑处理的缺陷(比如:导航中止的需求无法实现)。

更合理的方案应该是:导航过程中, 可以连续反馈当前机器人状态信息, 当导航终止时, 再返回最终的执行结果。在ROS中, 该实现策略称之为:**action 通信**。

概念

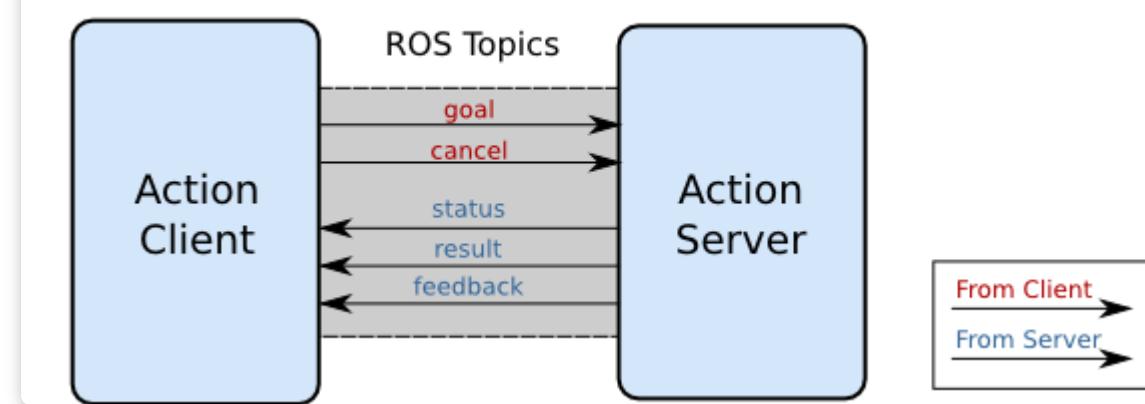
在ROS中提供了actionlib功能包集, 用于实现 action 通信。action 是一种类似于服务通信的实现, 其实现模型也包含请求和响应, 但是不同的是, 在请求和响应的过程中, 服务端还可以连续的反馈当前任务进度, 客户端可以接收连续反馈并且还可以取消任务。

action结构图解:



action通信接口图解:

Action Interface



- **goal**:目标任务;
- **cancel**:取消任务;
- **status**:服务端状态;
- **result**:最终执行结果(只会发布一次);

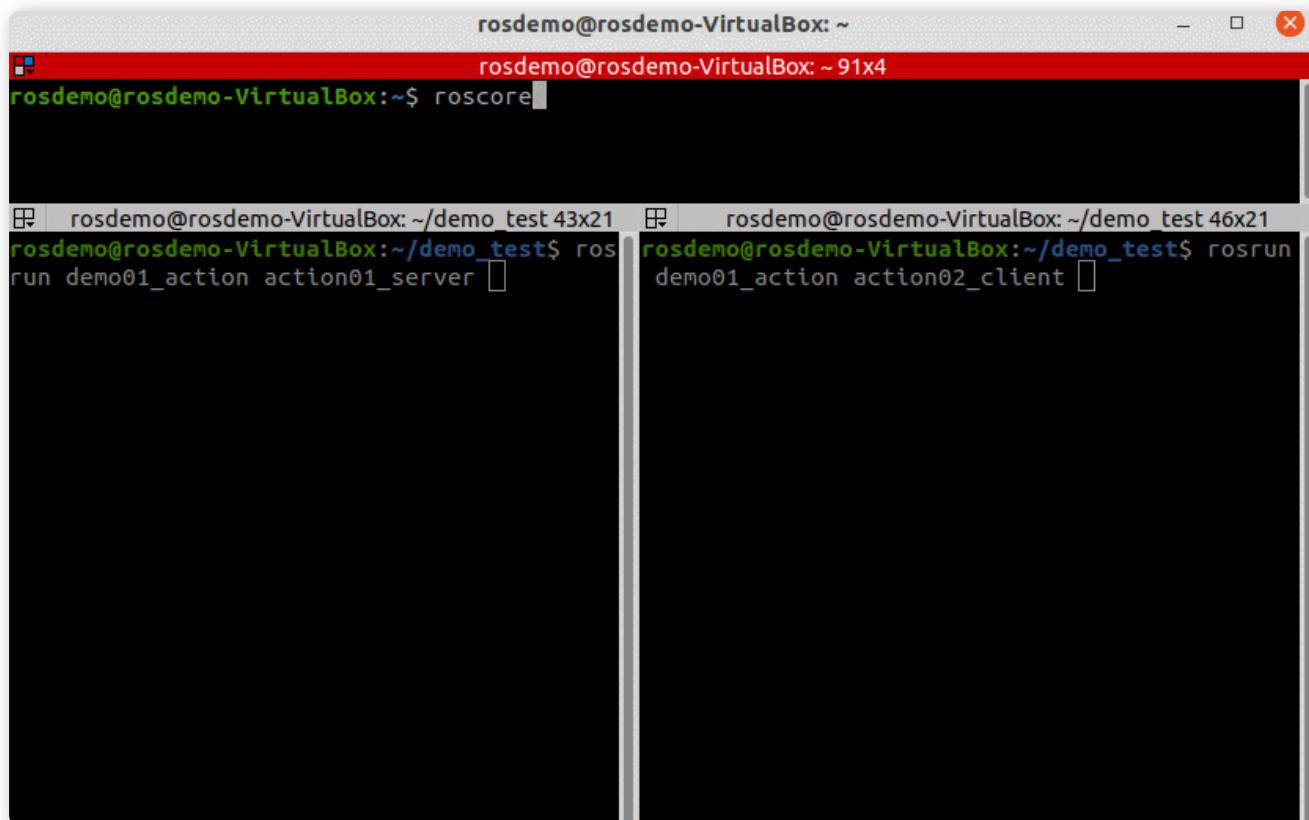
- **feedback**:连续反馈(可以发布多次)。

作用

一般适用于耗时的请求响应场景,用以获取连续的状态反馈。

案例

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。



```
rosdemo@rosdemo-VirtualBox: ~
rosdemo@rosdemo-VirtualBox: ~ 91x4
rosdemo@rosdemo-VirtualBox:~$ roscore

rosdemo@rosdemo-VirtualBox: ~/demo_test 43x21
rosdemo@rosdemo-VirtualBox:~/demo_test$ rosrun demo01_action action01_server

rosdemo@rosdemo-VirtualBox: ~/demo_test 46x21
rosdemo@rosdemo-VirtualBox:~/demo_test$ rosrun demo01_action action02_client
```

另请参考:

- <http://wiki.ros.org/actionlib>
- http://wiki.ros.org/actionlib_tutorials/Tutorials

10.1.1 action通信自定义action文件

action、srv、msg文件内的可用数据类型一致，且三者实现流程类似：

1. 按照固定格式创建action文件；
2. 编辑配置文件；
3. 编译生成中间文件。

1. 定义action文件

首先新建功能包，并导入依赖： `roscpp rospy std_msgs`
`actionlib actionlib_msgs`；

然后功能包下新建 action 目录，新增 Xxx.action(比如: `AddInts.action`)。

action 文件内容组成为三部分: 请求目标值、最终响应结果、连续反馈，三者之间使用 `---` 分割示例内容如下：

```

1 #目标值
2 int32 num
3 ---
4 #最终结果
5 int32 result
6 ---
7 #连续反馈
8 float64 progress_bar

```

2. 编辑配置文件

`CMakeLists.txt`

```

1 find_package
2 (catkin REQUIRED COMPONENTS
3   roscpp
4   rospy
5   std_msgs
6   actionlib

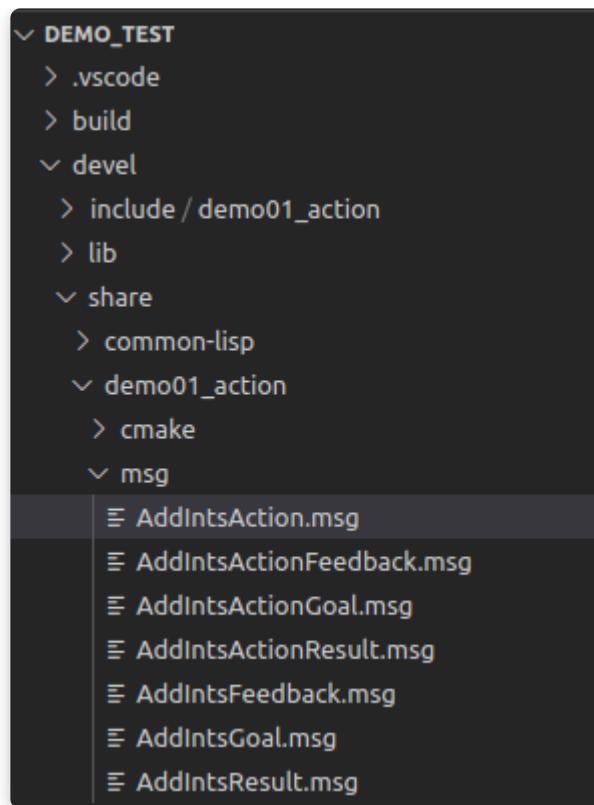
```

```
7     actionlib_msgs
8  )
9 add_action_files(
10   FILES
11   AddInts.action
12  )
13 generate_messages(
14   DEPENDENCIES
15   std_msgs
16   actionlib_msgs
17  )
18 catkin_package(
19
20 #   INCLUDE_DIRS include
21 #   LIBRARIES demo04_action
22
23 CATKIN_DEPENDS roscpp rospy std_msgs actionlib
24   actionlib_msgs
25 #   DEPENDS system_lib
26
27 )
```

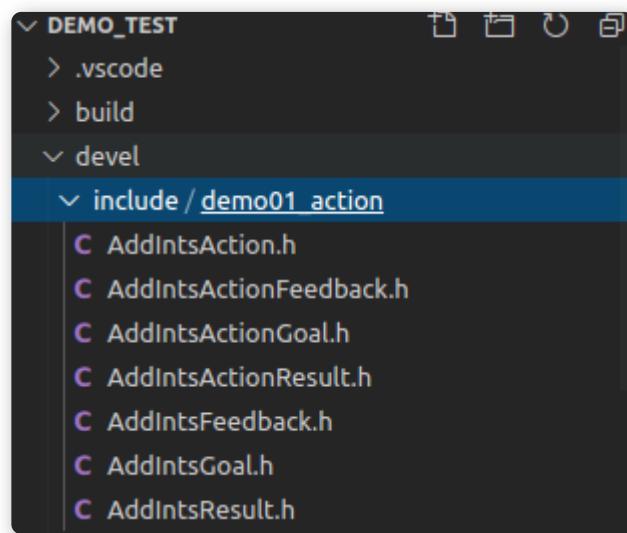
3. 编译

编译后会生成一些中间文件。

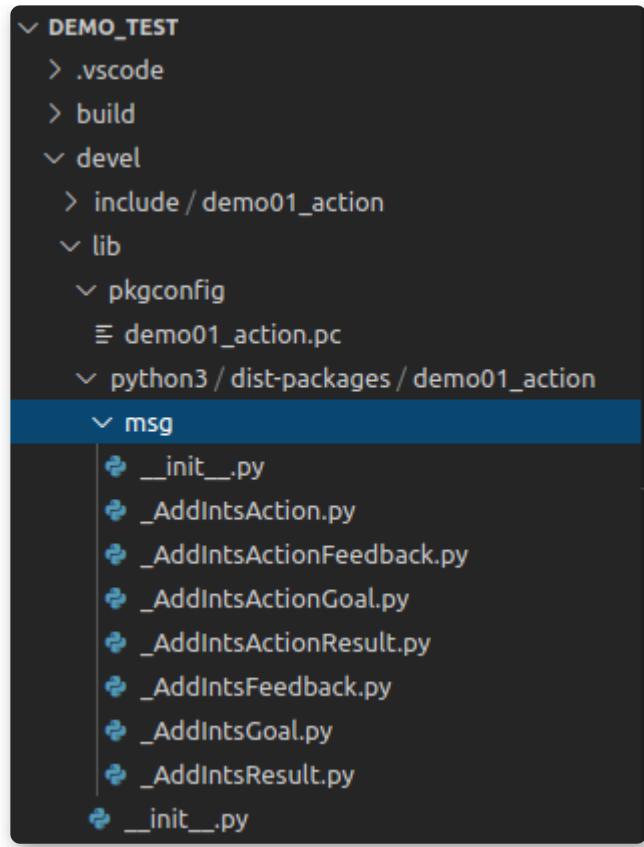
msg文件(.../工作空间/devel/share/包名/msg/xxx.msg):



C++ 调用的文件(.../工作空间/devel/include/包名/xxx.h):



Python 调用的文件(.../工作空间/devel/lib/python3/dist-packages/包名/msg/xxx.py):



10.1.2 action通信自定义action文件调用A(C++)

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. 编写action服务端实现；
2. 编写action客户端实现；
3. 编辑CMakeLists.txt；
4. 编译并执行。

0.vscode配置

需要像之前自定义 msg 实现一样配置c_cpp_properties.json 文件，如果以前已经配置且没有变更工作空间，可以忽略，如果需要配置，配置方式与之前相同：

```

1  {
2      "configurations": [
3          {
4              "browse": {
5                  "databaseFilename": "",
6                  "limitSymbolsToIncludedHeaders":
7                      true
8              },
9              "includePath": [
10                  "/opt/ros/noetic/include/**",
11                  "/usr/include/**",
12                  "/xxx/yyy工作空间/devel/include/**"
13                  //配置 head 文件的路径
14                  ],
15                  "name": "ROS",
16                  "intelliSenseMode": "gcc-x64",
17                  "compilerPath": "/usr/bin/gcc",
18                  "cStandard": "c11",
19                  "cppStandard": "c++17"
20                  }
21          ],
22          "version": 4
23      }

```

1.服务端

```

1 #include "ros/ros.h"
2 #include "actionlib/server/simple_action_server.h"
3 #include "demo01_action/AddIntsAction.h"
4 /*
5      需求：

```

6 创建两个ROS节点，服务器和客户端，
 7 客户端可以向服务器发送目标数据N（一个整型数据）
 8 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，
 9 这是基于请求响应模式的，
 10 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，
 11 为了良好的用户体验，需要服务器在计算过程中，
 12 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。

13
 14 流程：
 15 1. 包含头文件；
 16 2. 初始化ROS节点；
 17 3. 创建NodeHandle；
 18 4. 创建action服务对象；
 19 5. 处理请求，产生反馈与响应；
 20 6. spin().
 21
 22 */
 23
 24 **typedef**
 actionlib::SimpleActionServer<demo01_action::AddIn
 tsAction> Server;
 25
 26
 27 **void** cb(**const** demo01_action::AddIntsGoalConstPtr
 &goal, Server* server){
 28 //获取目标值
 29 **int** num = goal->num;
 30 ROS_INFO("目标值:%d", num);
 31 //累加并响应连续反馈
 32 **int** result = 0;
 33 demo01_action::AddIntsFeedback feedback; //连续
 反馈
 34 ros::Rate rate(10); //通过频率设置休眠时间
 35 **for** (**int** i = 1; i <= num; i++)
 36 {
 37 result += i;

```

38         //组织连续数据并发布
39         feedback.progress_bar = i / (double)num;
40         server->publishFeedback(feedback);
41         rate.sleep();
42     }
43     //设置最终结果
44     demo01_action::AddIntsResult r;
45     r.result = result;
46     server->setSucceeded(r);
47     ROS_INFO("最终结果:%d",r.result);
48 }
49
50 int main(int argc, char *argv[])
51 {
52     setlocale(LC_ALL,"");
53     ROS_INFO("action服务端实现");
54     // 2.初始化ROS节点;
55     ros::init(argc,argv,"AddInts_server");
56     // 3.创建NodeHandle;
57     ros::NodeHandle nh;
58     // 4.创建action服务对象;
59     /*SimpleActionServer<ros::NodeHandle n,
60                  std::string name,
61                  boost::function<void
62 (const demo01_action::AddIntsGoalConstPtr &)>
63 execute_callback,
64                         bool auto_start)
65 */
66     //actionlib::SimpleActionServer<demo01_action::AddIntsAction> server(...);
67     Server
68     server(nh,"addInts",boost::bind(&cb,_1,&server),false);
69     server.start();
70     // 5.处理请求,产生反馈与响应;
71
72     // 6.spin().
73     ros::spin();

```

```
71     return 0;
72 }
```

PS:

可以先配置CMakeLists.tx文件并启动上述action服务端，然后通过
rostopic 查看话题，向action相关话题发送消息，或订阅action相关话题的
消息。

2.客户端

```
1 #include "ros/ros.h"
2 #include "actionlib/client/simple_action_client.h"
3 #include "demo01_action/AddIntsAction.h"
4
5 /*
6     需求：
7         创建两个ROS节点，服务器和客户端，  

8             客户端可以向服务器发送目标数据N（一个整型数据）  

9                 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  

10                的过程，返回给客户端，  

11                    这是基于请求响应模式的，  

12                        又已知服务器从接收到请求到产生响应是一个耗时操作，  

13                            每累加一次耗时0.1s，  

14                                为了良好的用户体验，需要服务器在计算过程中，  

15                                    每累加一次，就给客户端响应一次百分比格式的执行进  

16                                    度，使用action实现。
17
18     流程：
19         1. 包含头文件；  

20             2. 初始化ROS节点；  

21                 3. 创建NodeHandle；  

22                     4. 创建action客户端对象；  

23                         5. 发送目标，处理反馈以及最终结果；  

24                             6. spin()。
25 */
```

```

24 typedef
25   actionlib::SimpleActionClient<demo01_action::AddIn
26   tsAction> Client;
27
28
29 //处理最终结果
30 void done_cb(const
31   actionlib::SimpleClientGoalState &state, const
32   demo01_action::AddIntsResultConstPtr &result){
33   if (state.state_ == state.SUCCEEDED)
34   {
35     ROS_INFO("最终结果:%d",result->result);
36   } else {
37     ROS_INFO("任务失败！");
38   }
39
40 }
41 //服务已经激活
42 void active_cb(){
43   ROS_INFO("服务已经被激活....");
44 }
45 //处理连续反馈
46 void feedback_cb(const
47   demo01_action::AddIntsFeedbackConstPtr &feedback){
48   ROS_INFO("当前进度:%.2f",feedback-
49   >progress_bar);
50 }
51
52
53 int main(int argc, char *argv[])
54 {
55   setlocale(LC_ALL,"");
56   // 2.初始化ROS节点;
57   ros::init(argc,argv,"AddInts_client");
58   // 3.创建NodeHandle;
59   ros::NodeHandle nh;
60   // 4.创建action客户端对象;
61   // SimpleActionClient(ros::NodeHandle & n,
62   const std::string & name, bool spin_thread = true)

```

```

56      //
57      actionlib::SimpleActionClient<demo01_action::AddIntsAction> client(nh, "addInts");
58      Client client(nh, "addInts", true);
59      //等待服务启动
60      client.waitForServer();
61      // 5.发送目标, 处理反馈以及最终结果;
62      /*
63          void sendGoal(const
64              demo01_action::AddIntsGoal &goal,
65              boost::function<void (const
66                  actionlib::SimpleClientGoalState &state, const
67                  demo01_action::AddIntsResultConstPtr &result)>
68              done_cb,
69              boost::function<void ()> active_cb,
70              boost::function<void (const
71                  demo01_action::AddIntsFeedbackConstPtr &feedback)>
72              feedback_cb)
73
74          */
75          demo01_action::AddIntsGoal goal;
76          goal.num = 10;
77
78          client.sendGoal(goal, &done_cb, &active_cb, &feedback_cb);
79          // 6.spin().
80          ros::spin();
81          return 0;
82      }

```

PS:等待服务启动, 只可以使用 `client.waitForServer();`, 之前服务中等待启动的另一种方式

`ros::service::waitForService("addInts");` 不适用

3. 编译配置文件

```

1 add_executable(action01_server
  src/action01_server.cpp)
2 add_executable(action02_client
  src/action02_client.cpp)
3 ...
4
5 add_dependencies(action01_server
  ${${PROJECT_NAME}_EXPORTED_TARGETS}
  ${catkin_EXPORTED_TARGETS})
6 add_dependencies(action02_client
  ${${PROJECT_NAME}_EXPORTED_TARGETS}
  ${catkin_EXPORTED_TARGETS})
7 ...
8
9 target_link_libraries(action01_server
10   ${catkin_LIBRARIES}
11 )
12 target_link_libraries(action02_client
13   ${catkin_LIBRARIES}
14 )

```

4. 执行

首先启动 roscore，然后分别启动action服务端与action客户端，最终运行结果与案例类似。

10.1.3 action通信自定义action文件调用 (Python)

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. 编写action服务端实现；
2. 编写action客户端实现；
3. 编辑CMakeLists.txt；
4. 编译并执行。

0.vscode配置

需要像之前自定义 msg 实现一样配置settings.json 文件，如果以前已经配置且没有变更工作空间，可以忽略，如果需要配置，配置方式与之前相同：

```

1  {
2      "python.autoComplete.extraPaths": [
3          "/opt/ros/noetic/lib/python3/dist-packages",
4          "/xxx/yyy工作空间/devel/lib/python3/dist-
5          packages"
6      ]
7  }

```

1.服务端

```

1  #! /usr/bin/env python
2  import rospy
3  import actionlib
4  from demo01_action.msg import *
5  """
6      需求：
7          创建两个ROS 节点，服务器和客户端，

```

8 客户端可以向服务器发送目标数据N(一个整型数据)服务
 9 服务器会计算 1 到 N 之间所有整数的和，
 10 这是一个循环累加的过程，返回给客户端，这是基于请求
 11 响应模式的，
 12 又已知服务器从接收到请求到产生响应是一个耗时操作，
 13 每累加一次耗时0.1s，
 14 为了良好的用户体验，需要服务器在计算过程中，
 15 每累加一次，就给客户端响应一次百分比格式的执行进
 16 度，使用 `action`实现。

13 流程：

14 1. 导包
 15 2. 初始化 ROS 节点
 16 3. 使用类封装，然后创建对象
 17 4. 创建服务器对象
 18 5. 处理请求数据产生响应结果，中间还要连续反馈
 19 6. `spin`
 20 """
 21
 22 class MyActionServer:
 23 def __init__(self):
 24 #SimpleActionServer(name, ActionSpec,
 25 execute_cb=None, auto_start=True)
 26 self.server =
 27 actionlib.SimpleActionServer("addInts", AddIntsAction, self.cb, False)
 28 self.server.start()
 29 rospy.loginfo("服务端启动")
 30
 31 def cb(self,goal):
 32 rospy.loginfo("服务端处理请求:")
 33 #1. 解析目标值
 34 num = goal.num
 35 #2. 循环累加，连续反馈
 36 rate = rospy.Rate(10)
 37 sum = 0
 38 for i in range(1,num + 1):
 39 # 累加
 40 sum = sum + i

```

40          # 计算进度并连续反馈
41          feedBack = i / num
42          rospy.loginfo("当前进度: %.2f", feedBack)
43
44          feedBack_obj = AddIntsFeedback()
45          feedBack_obj.progress_bar = feedBack
46
47          self.server.publish_feedback(feedBack_obj)
48          rate.sleep()
49          #3. 响应最终结果
50          result = AddIntsResult()
51          result.result = sum
52          self.server.set_succeeded(result)
53          rospy.loginfo("响应结果: %d", sum)
54 if __name__ == "__main__":
55     rospy.init_node("action_server_p")
56     server = MyActionServer()
57     rospy.spin()

```

PS:

可以先配置CMakeLists.tx文件并启动上述action服务端，然后通过
rostopic 查看话题，向action相关话题发送消息，或订阅action相关话题的消息。

2.客户端

```

1  #! /usr/bin/env python
2
3  import rospy
4  import actionlib
5  from demo01_action.msg import *
6
7  """
8  需求:
9  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
10  器会计算 1 到 N 之间所有整数的和，
```

```

11      这是一个循环累加的过程，返回给客户端，这是基于请求
12      响应模式的，
13      又已知服务器从接收到请求到产生响应是一个耗时操作，
14      每累加一次耗时0.1s，
15      为了良好的用户体验，需要服务器在计算过程中，
16      每累加一次，就给客户端响应一次百分比格式的执行进
17      度，使用 action实现。
18      流程：
19          1. 导包
20          2. 初始化 ROS 节点
21          3. 创建 action Client 对象
22          4. 等待服务
23          5. 组织目标对象并发送
24          6. 编写回调，激活、连续反馈、最终响应
25          7. spin
26
27      """
28
29      def done_cb(state,result):
30          if state == actionlib.GoalStatus.SUCCEEDED:
31              rospy.loginfo("响应结果:%d",result.result)
32
33      def active_cb():
34          rospy.loginfo("服务被激活....")
35
36      def fb_cb(fb):
37          rospy.loginfo("当前进度:%.2f",fb.progress_bar)
38
39      if __name__ == "__main__":
40          # 2. 初始化 ROS 节点
41          rospy.init_node("action_client_p")
42          # 3. 创建 action Client 对象
43          client =
44              actionlib.SimpleActionClient("addInts",AddIntsActi
45          on)
46          # 4. 等待服务
47          client.wait_for_server()
48          # 5. 组织目标对象并发送
49          goal_obj = AddIntsGoal()

```

```

45     goal_obj.num = 10
46
47     client.send_goal(goal_obj,done_cb,active_cb,fb_cb
48
49     # 6.编写回调，激活、连续反馈、最终响应
50     # 7.spin
51     rospy.spin()

```

3.编辑配置文件

先为 Python 文件添加可执行权限: `chmod +x *.py`

```

1 catkin_install_python(PROGRAMS
2   scripts/action01_server_p.py
3   scripts/action02_client_p.py
4   DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
5 )

```

4.执行

首先启动 `roscore`，然后分别启动action服务端与action客户端，最终运行结果与案例类似。

10.2 动态参数

参数服务器的数据被修改时，如果节点不重新访问，那么就不能获取修改后的数据，例如在乌龟背景色修改的案例中，先启动乌龟显示节点，然后再修改参数服务器中关于背景色设置的参数，那么窗体的背景色是不会修改的，必须要重启乌龟显示节点才能生效。而一些特殊场景下，是要求要能做到动态获取的，也即，参数一旦修改，能够通知节点参数已经修改并读取修改后的数据，比如：



机器人调试时，需要修改机器人轮廓信息(长宽高)、传感器位姿信息....，如果这些信息存储在参数服务器中，那么意味着需要重启节点，才能使更新设置生效，但是希望修改完毕之后，某些节点能够即时更新这些参数信息。

在ROS中针对这种场景已经给出的解决方案: **dynamic reconfigure** 动态配置参数。

动态配置参数，之所以能够实现即时更新，因为被设计成CS架构，客户端修改参数就是向服务器发送请求，服务器接收到请求之后，读取修改后的是参数。

概念

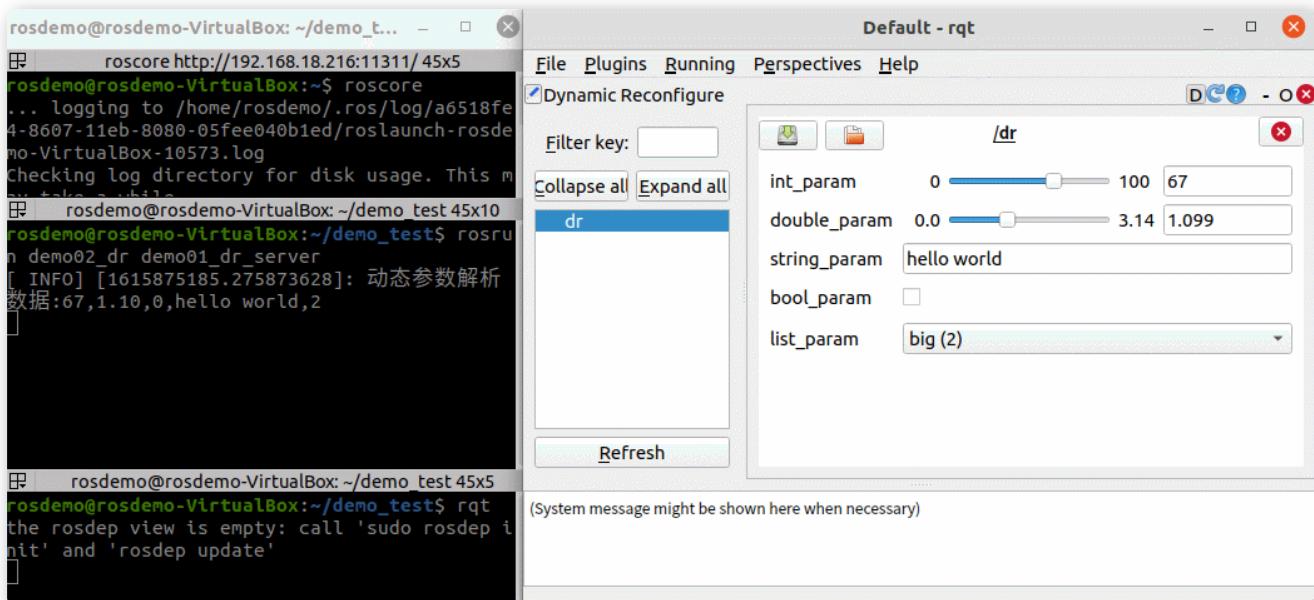
一种可以在运行时更新参数而无需重启节点的参数配置策略。

作用

主要应用于需要动态更新参数的场景，比如参数调试、功能切换等。典型应用:导航时参数的动态调试。

案例

编写两个节点，一个节点可以动态修改参数，另一个节点时时解析修改后的数据。



另请参考:

- http://wiki.ros.org/dynamic_reconfigure
- http://wiki.ros.org/dynamic_reconfigure/Tutorials

10.2.1 动态参数客户端

需求:



编写两个节点，一个节点可以动态修改参数，另一个节点时时解析修改后的数据。

客户端实现流程:

- 新建并编辑 .cfg 文件;
- 编辑CMakeLists.txt;
- 编译。

1.新建功能包

新建功能包，添加依赖: `roscpp rospy std_msgs dynamic_reconfigure`。

2.添加.cfg文件

新建 cfg 文件夹，添加 xxx.cfg 文件(并添加可执行权限)，cfg 文件其实就是一个 python 文件,用于生成参数修改的客户端(GUI)。

```

1 #!/usr/bin/env python
2 """
3 4生成动态参数 int,double,bool,string,列表
4 5实现流程:
5 6 1. 导包

```

```

6 7      2. 创建生成器
7 8      3. 向生成器添加若干参数
8 9      4. 生成中间文件并退出
9 10
10 """
11 # 1. 导包
12 from
13 dynamic_reconfigure.parameter_generator_catkin
14 import *
15 PACKAGE = "demo02_dr"
16 # 2. 创建生成器
17 gen = ParameterGenerator()
18
19 # 3. 向生成器添加若干参数
20 #add(name, paramtype, level, description,
21 #      default=None, min=None, max=None, edit_method="")
22 gen.add("int_param", int_t, 0, "整型参数", 50, 0, 100)
23 gen.add("double_param", double_t, 0, "浮点参
24 数", 1.57, 0, 3.14)
25 gen.add("string_param", str_t, 0, "字符串参数", "hello
26 world ")
27 gen.add("bool_param", bool_t, 0, "bool参数", True)
28
29 many_enum = gen.enum([gen.const("small", int_t, 0, "a
30 small size"),
31                         gen.const("medium", int_t, 1, "a
32 medium size"),
33                         gen.const("big", int_t, 2, "a big
34 size")]
35                         ], "a car size set")
36
37 gen.add("list_param", int_t, 0, "列表参数", 0, 0, 2,
38         edit_method=many_enum)
39
40 # 4. 生成中间文件并退出
41 exit(gen.generate(PACKAGE, "dr_node", "dr"))

```

chmod +x xxx.cfg 添加权限

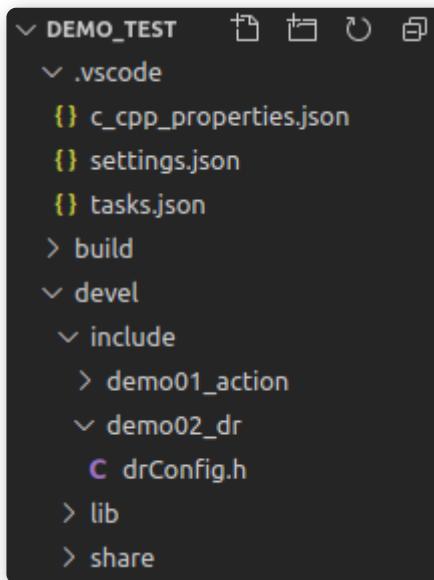
3.配置 CMakeLists.txt

```
1 generate_dynamic_reconfigure_options(  
2   cfg/mycar.cfg  
3 )
```

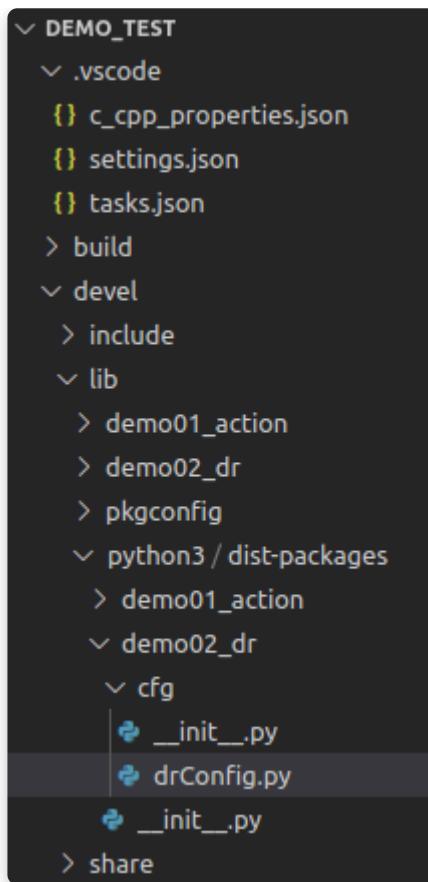
4.编译

编译后会生成中间文件

C++ 需要调用的头文件:



Python需要调用的文件:



10.2.2 动态参数服务端A(C++)

需求:



编写两个节点，一个节点可以动态修改参数，另一个节点时时解析修改后的数据。

服务端实现流程:

- 新建并编辑 c++ 文件;
- 编辑CMakeLists.txt;
- 编译并执行。

0.vscode配置

需要像之前自定义 msg 实现一样配置settings.json 文件，如果以前已经配置且没有变更工作空间，可以忽略，如果需要配置，配置方式与之前相同：

```

1  {
2      "configurations": [
3          {
4              "browse": {
5                  "databaseFilename": "",
6                  "limitSymbolsToIncludedHeaders":
7                      true
8              },
9              "includePath": [
10                  "/opt/ros/noetic/include/**",
11                  "/usr/include/**",
12                  "/xxx/yyy工作空间/devel/include/**"
13                  //配置 head 文件的路径
14                  ],
15                  "name": "ROS",
16                  "intelliSenseMode": "gcc-x64",
17                  "compilerPath": "/usr/bin/gcc",
18                  "cStandard": "c11",
19                  "cppStandard": "c++17"
20          }
21      ],
22      "version": 4
23  }

```

1.服务器代码实现

新建cpp文件，内容如下：

```

1 #include "ros/ros.h"
2 #include "dynamic_reconfigure/server.h"
3 #include "demo02_dr/drConfig.h"
4 /*
5      动态参数服务端：参数被修改时直接打印

```

```

6      实现流程:
7          1. 包含头文件
8          2. 初始化 ros 节点
9          3. 创建服务器对象
10         4. 创建回调对象(使用回调函数, 打印修改后的参数)
11         5. 服务器对象调用回调对象
12         6. spin()
13     */
14
15 void cb(demo02_dr::drConfig& config, uint32_t
16 level){
17     ROS_INFO("动态参数解析数据:%d,%.2f,%d,%s,%d",
18             config.int_param,
19             config.double_param,
20             config.bool_param,
21             config.string_param.c_str(),
22             config.list_param
23 );
24 }
25
26 int main(int argc, char *argv[])
27 {
28     setlocale(LC_ALL, "");
29     // 2. 初始化 ros 节点
30     ros::init(argc, argv, "dr");
31     // 3. 创建服务器对象
32
33     dynamic_reconfigure::Server<demo02_dr::drConfig>
34     server;
35     // 4. 创建回调对象(使用回调函数, 打印修改后的参数)
36
37     dynamic_reconfigure::Server<demo02_dr::drConfig>:
38     :CallbackType cbType;
39     cbType = boost::bind(&cb,_1,_2);
40     // 5. 服务器对象调用回调对象
41     server.setCallback(cbType);
42     // 6. spin()
43     ros::spin();
44     return 0;

```

2. 编译配置文件

```

1 add_executable(demo01_dr_server
  src/demo01_dr_server.cpp)
2 ...
3
4 add_dependencies(demo01_dr_server
  ${${PROJECT_NAME}_EXPORTED_TARGETS}
  ${catkin_EXPORTED_TARGETS})
5 ...
6
7 target_link_libraries(demo01_dr_server
8   ${catkin_LIBRARIES}
9 )

```

3. 执行

先启动 `roscore`

启动服务端: `rosrun` 功能包 `xxxx`

启动客户端: `rosrun rqt_gui rqt_gui -s rqt_reconfigure` 或
`rosrun rqt_reconfigure rqt_reconfigure`

最终可以通过客户端提供的界面修改数据，并且修改完毕后，服务端会即时输出修改后的结果，最终运行结果与示例类似。

PS:ROS版本较新时，可能没有提供客户端相关的功能包导致 `rosrun rqt_reconfigure rqt_reconfigure` 调用会抛出异常。

10.2.3 动态参数服务端B(Python)

需求:



编写两个节点，一个节点可以动态修改参数，另一个节点时时解析修改后的数据。

服务端实现流程：

- 新建并编辑 Python 文件；
- 编辑CMakeLists.txt；
- 编译并执行。

0.vscode配置

需要像之前自定义 msg 实现一样配置settings.json 文件，如果以前已经配置且没有变更工作空间，可以忽略，如果需要配置，配置方式与之前相同：

```

1  {
2      "python.autoComplete.extraPaths": [
3          "/opt/ros/noetic/lib/python3/dist-packages",
4          "/xxx/yyy工作空间-devel/lib/python3/dist-
5          packages"
6      ]
7  }

```

1.服务器代码实现

新建python文件，内容如下：

```

1  #! /usr/bin/env python
2  import rospy
3  from dynamic_reconfigure.server import Server
4  from demo02_dr.cfg import drConfig
5
6  """
7      动态参数服务端：参数被修改时直接打印
8      实现流程：
9          1. 导包
10             2. 初始化 ros 节点

```

```

11         3. 创建服务对象
12         4. 回调函数处理
13         5. spin
14 """
15 # 回调函数
16 def cb(config, level):
17     rospy.loginfo("python 动态参数服务解
析:%d,%.2f,%d,%s,%d",
18                 config.int_param,
19                 config.double_param,
20                 config.bool_param,
21                 config.string_param,
22                 config.list_param
23 )
24     return config
25
26 if __name__ == "__main__":
27     # 2. 初始化 ros 节点
28     rospy.init_node("dr_p")
29     # 3. 创建服务对象
30     server = Server(drConfig, cb)
31     # 4. 回调函数处理
32     # 5. spin
33     rospy.spin()

```

2. 编辑配置文件

先为 Python 文件添加可执行权限: `chmod +x *.py`

```

1 catkin_install_python(PROGRAMS
2     scripts/demo01_dr_server_p.py
3     DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
4 )

```

3. 执行

先启动 `roscore`

启动服务端: `rosrun` 功能包 `xxxx.py`

启动客户端: `rosrun rqt_gui rqt_gui -s rqt_reconfigure` 或 `rosrun rqt_reconfigure rqt_reconfigure`

最终可以通过客户端提供的界面修改数据，并且修改完毕后，服务端会即时输出修改后的结果，最终运行结果与示例类似。

PS:ROS版本较新时，可能没有提供客户端相关的功能包导致 `rosrun rqt_reconfigure rqt_reconfigure` 调用会抛出异常。

10.3 pluginlib

pluginlib直译是插件库，所谓插件字面意思就是可插拔的组件，比如:以计算机为例，可以通过USB接口自由插拔的键盘、鼠标、U盘...都可以看作是插件实现，其基本原理就是通过规范化的USB接口协议实现计算机与USB设备的自由组合。同理，在软件编程中，插件是一种遵循一定规范的应用程序接口编写出来的程序，插件程序依赖于某个应用程序，且应用程序可以与不同的插件程序自由组合。在ROS中，也会经常使用到插件，场景如下:



1. 导航插件:在导航中，涉及到路径规划模块，路径规划算法有多种，也可以自实现，导航应用时，可能需要测试不同算法的优劣以选择更合适的实现，这种场景下，ROS中就是通过插件的方式来实现不同算法的灵活切换的。

2. rviz插件:在rviz中已经提供了丰富的功能实现，但是即便如此，特定场景下，开发者可能需要实现某些定制化功能并集成到rviz中，这一集成过程也是基于插件的。

概念

pluginlib是一个c++库， 用来从一个ROS功能包中加载和卸载插件(plugin)。插件是指从运行时库中动态加载的类。通过使用Pluginlib，不必将某个应用程序显式地链接到包含某个类的库， Pluginlib可以随时打开包含类的库，而不需要应用程序事先知道包含类定义的库或者头文件。

作用

- 结构清晰；
 - 低耦合，易修改，可维护性强；
 - 可移植性强，更具复用性；
 - 结构容易调整，插件可以自由增减；
-

另请参考:

- <http://wiki.ros.org/pluginlib>
- <http://wiki.ros.org/pluginlib/Tutorials/Writing%20and%20Using%20a%20Simple%20Plugin>

10.3.1 pluginlib使用

需求:

以插件的方式实现正多边形的相关计算。

实现流程:

1. 准备；
2. 创建基类；
3. 创建插件类；
4. 注册插件；
5. 构建插件库；
6. 使插件可用于ROS工具链；
 - 配置xml

- 导出插件

7. 使用插件;

8. 执行。

1.准备

创建功能包xxx导入依赖: roscpp pluginlib。

在 VSCode 中需要配置 .vscode/c_cpp_properties.json 文件中关于 includepath 选项的设置。

```

1  {
2      "configurations": [
3          {
4              "browse": {
5                  "databaseFilename": "",
6                  "limitSymbolsToIncludedHeaders":
7                      true
8                  },
9                  "includePath": [
10                     "/opt/ros/noetic/include/**",
11                     "/usr/include/**",
12                     "/.../yyy工作空间/功能包/include/**"
13                     //配置 head 文件的路径
14                     ],
15                     "name": "ROS",
16                     "intelliSenseMode": "gcc-x64",
17                     "compilerPath": "/usr/bin/gcc",
18                     "cStandard": "c11",
19                     "cppStandard": "c++17"
20                 }
21             ],
22             "version": 4
23         }

```

2. 创建基类

在 xxx/include/xxx下新建C++头文件: polygon_base.h, 所有的插件类都需要继承此基类, 内容如下:

```

1 #ifndef XXX_POLYGON_BASE_H_
2 #define XXX_POLYGON_BASE_H_
3
4 namespace polygon_base
5 {
6     class RegularPolygon
7     {
8         public:
9             virtual void initialize(double side_length)
= 0;
10            virtual double area() = 0;
11            virtual ~RegularPolygon(){}
12
13     protected:
14         RegularPolygon(){}
15     };
16 };
17 #endif

```

PS: 基类必须提供无参构造函数, 所以关于多边形的边长没有通过构造函数而是通过单独编写的initialize函数传参。

3. 创建插件

在 xxx/include/xxx下新建C++头文件: polygon_plugins.h, 内容如下:

```

1 #ifndef XXX_POLYGON_PLUGINS_H_
2 #define XXX_POLYGON_PLUGINS_H_
3 #include <xxx/polygon_base.h>
4 #include <cmath>
5
6 namespace polygon_plugins
7 {

```

```
8  class Triangle : public
9    polygon_base::RegularPolygon
10   {
11     public:
12       Triangle(){}
13
14       void initialize(double side_length)
15       {
16         side_length_ = side_length;
17       }
18
19       double area()
20       {
21         return 0.5 * side_length_ * getHeight();
22       }
23
24       double getHeight()
25       {
26         return sqrt((side_length_ * side_length_ )
27 - ((side_length_ / 2) * (side_length_ / 2)));
28       }
29
30     private:
31       double side_length_;
32   };
33
34  class Square : public
35    polygon_base::RegularPolygon
36  {
37     public:
38       Square(){}
39
40       void initialize(double side_length)
41       {
42         side_length_ = side_length;
43       }
44
45       double area()
46       {
47
```

```

44         return side_length_ * side_length_;
45     }
46
47     private:
48     double side_length_;
49
50 };
51 };
52 #endif

```

该文件中创建了正方形与三角形两个衍生类继承基类。

4.注册插件

在 src 目录下新建 polygon_plugins.cpp 文件，内容如下：

```

1 //pluginlib 宏，可以注册插件类
2 #include <pluginlib/class_list_macros.h>
3 #include <xxx/polygon_base.h>
4 #include <xxx/polygon_plugins.h>
5
6 //参数1：衍生类 参数2：基类
7 PLUGINLIB_EXPORT_CLASS(polygon_plugins::Triangle,
  polygon_base::RegularPolygon)
8 PLUGINLIB_EXPORT_CLASS(polygon_plugins::Square,
  polygon_base::RegularPolygon)

```

该文件会将两个衍生类注册为插件。

5.构建插件库

在 CMakeLists.txt 文件中设置内容如下：

```

1 include_directories(include)
2 add_library(polygon_plugins src/polygon_plugins.cpp)

```

至此，可以调用 catkin_make 编译，编译完成后，在工作空间/devel/lib 目录下，会生成相关的 .so 文件。

6.使插件可用于ROS工具链

6.1配置xml

功能包下新建文件: polygon_plugins.xml, 内容如下:

```

1  <!-- 插件库的相对路径 -->
2  <library path="lib/libpolygon_plugins">
3      <!-- type="插件类" base_class_type="基类" -->
4      <class type="polygon_plugins::Triangle"
       base_class_type="polygon_base::RegularPolygon">
5          <!-- 描述信息 -->
6          <description>This is a triangle plugin.
</description>
7      </class>
8      <class type="polygon_plugins::Square"
       base_class_type="polygon_base::RegularPolygon">
9          <description>This is a square plugin.
</description>
10     </class>
11 </library>

```

6.2导出插件

package.xml文件中设置内容如下:

```

1 <export>
2   <xxx plugin="${prefix}/polygon_plugins.xml" />
3 </export>

```

标签的名称应与基类所属的功能包名称一致, plugin属性值为上一步中创建的xml文件。

编译后, 可以调用 `rospack plugins --attrib=plugin xxx` 命令查看配置是否正常, 如无异常, 会返回 .xml 文件的完整路径, 这意味着插件已经正确的集成到了ROS工具链。

7. 使用插件

src 下新建c++文件: polygon_loader.cpp, 内容如下:

```
1 //类加载器相关的头文件
2 #include <pluginlib/class_loader.h>
3 #include <xxx/polygon_base.h>
4
5 int main(int argc, char** argv)
6 {
7     //类加载器 -- 参数1:基类功能包名称 参数2:基类全限定名称
8     pluginlib::ClassLoader<polygon_base::RegularPoly
9     gon> poly_loader("xxx",
10     "polygon_base::RegularPolygon");
11
12     try
13     {
14         //创建插件类实例 -- 参数:插件类全限定名称
15
16         boost::shared_ptr<polygon_base::RegularPolygon>
17         triangle =
18         poly_loader.createInstance("polygon_plugins::Tri
19         gle");
20         triangle->initialize(10.0);
21
22         ROS_INFO("Triangle area: %.2f", triangle-
23         >area());
24         ROS_INFO("Square area: %.2f", square->area());
25     }
26     catch(pluginlib::PluginlibException& ex)
27     {
28     }
```

```

24     ROS_ERROR("The plugin failed to load for some
25     reason. Error: %s", ex.what());
26
27     return 0;
28 }
```

8.执行

修改CMakeLists.txt文件，内容如下：

```

1 add_executable(polygon_loader
2   src/polygon_loader.cpp)
2 target_link_libraries(polygon_loader
3   ${catkin_LIBRARIES})
```

编译然后执行: polygon_loader，结果如下：

```

1 [ INFO] [WallTime: 1279658450.869089666]: Triangle
area: 43.30
2 [ INFO] [WallTime: 1279658450.869138007]: Square
area: 100.00
```

10.4 nodelet

ROS通信是基于Node(节点)的，Node使用方便、易于扩展，可以满足ROS中大多数应用场景，但是也存在一些局限性，由于一个Node启动之后独占一根进程，不同Node之间数据交互其实是不同进程之间的数据交互，当传输类似于图片、点云的大容量数据时，会出现延时与阻塞的情况，比如：



现在需要编写一个相机驱动，在该驱动中有两个节点实现：其中节点A负责发布原始图像数据，节点B订阅原始图像数据并在图像上标注人脸。如果节点A与节点B仍按照之前实现，两个节点分别对应不同的进程，在两个进程之间传递容量可观图像数据，可能就会出现延时的情况，那么该

如何优化呢？

ROS中给出的解决方案是:Nodelet，通过Nodelet可以将多个节点集成进一个进程。

概念

nodelet软件包旨在提供在同一进程中运行多个算法(节点)的方式，不同算法之间通过传递指向数据的指针来代替了数据本身的传输(类似于编程传值与传址的区别)，从而实现零成本的数据拷贝。

nodelet功能包的核心实现也是插件，是对插件的进一步封装：

- 不同算法被封装进插件类，可以像单独的节点一样运行；
- 在该功能包中提供插件类实现的基类:Nodelet；
- 并且提供了加载插件类的类加载器:NodeletLoader。

作用

应用于大容量数据传输的场景，提高节点间的数据交互效率，避免延时与阻塞。

另请参考：

- <http://wiki.ros.org/nodelet/>
- <http://wiki.ros.org/nodelet/Tutorials/Running%20a%20nodelet>
- https://github.com/ros/common_tutorials/tree/noetic-devel/nodelet_tutorial_math

10.4.1 使用演示

在ROS中内置了nodelet案例，我们先以该案例演示nodelet的基本使用语法，基本流程如下：

1. 案例简介；
2. nodelet基本使用语法；
3. 内置案例调用。

1.案例简介

以“ros- [ROS_DISTRO] -desktop-full”命令安装ROS时，nodelet默认被安装，如未安装，请调用如下命令自行安装：

```
1 sudo apt install ros-<<ROS_DISTRO>>-nodelet-tutorial-math
```

在该案例中，定义了一个Nodelet插件类:Plus，这个节点可以订阅一个数字，并将订阅到的数字与参数服务器中的 value 参数相加后再发布。

需求:再同一线程中启动两个Plus节点A与B，向A发布一个数字，然后经A处理后，再发布并作为B的输入，最后打印B的输出。

2.nodelet 基本使用语法

使用语法如下：

```
1 nodelet load pkg/Type manager - Launch a nodelet of
    type pkg/Type on manager manager
2 nodelet standalone pkg/Type - Launch a nodelet of
    type pkg/Type in a standalone node
3 nodelet unload name manager - Unload a nodelet a
    nodelet by name from manager
4 nodelet manager - Launch a nodelet
    manager node
```

3.内置案例调用

1.启动roscore

```
1 roscore
```

2.启动manager

```
1 rosrun nodelet nodelet manager __name:=mymanager
```

`__name:=` 用于设置管理器名称。

3.添加nodelet节点

添加第一个节点:

```
1 rosrun nodelet nodelet load
nodelet_tutorial_math/Plus mymanager __name:=n1
_value:=100
```

添加第二个节点:

```
1 rosrun nodelet nodelet load
nodelet_tutorial_math/Plus mymanager __name:=n2
_value:=-50 /n2/in:=/n1/out
```

PS: 解释



`rosrun nodelet nodelet load nodelet_tutorial_math/Plus mymanager __name:=n1 _value:=100`

1. `rosnode list` 查看, nodelet 的节点名称是: `/n1`;
2. `rostopic list` 查看, 订阅的话题是: `/n1/in`, 发布的话题是: `/n1/out`;
3. `rosparam list` 查看, 参数名称是: `/n1/value`。

`rosrun nodelet nodelet standalone nodelet_tutorial_math/Plus mymanager __name:=n2 _value:=-50 /n2/in:=/n1/out`

1. 第二个nodelet 与第一个同理;
2. 第二个nodelet 订阅的话题由 `/n2/in` 重映射为 `/n1/out`。

优化:也可以将上述实现集成进launch文件:

```

1 <launch>
2   <!-- 设置nodelet管理器 -->
3   <node pkg="nodelet" type="nodelet"
4     name="mymanager" args="manager" output="screen" />
5   <!-- 启动节点1, 名称为 n1, 参数 /n1/value 为100 -->
6   <node pkg="nodelet" type="nodelet" name="n1"
7     args="load nodelet_tutorial_math/Plus mymanager"
8     output="screen" >
9     <param name="value" value="100" />
10    </node>
11    <!-- 启动节点2, 名称为 n2, 参数 /n2/value 为-50 -->
12    <node pkg="nodelet" type="nodelet" name="n2"
13      args="load nodelet_tutorial_math/Plus mymanager"
14      output="screen" >
15      <param name="value" value="-50" />
16      <remap from="/n2/in" to="/n1/out" />
17    </node>
18
19 </launch>

```

4.执行

向节点n1发布消息:

```

1 rostopic pub -r 10 /n1/in std_msgs/Float64 "data:
10.0"

```

打印节点n2发布的消息:

```

1 rostopic echo /n2/out

```

最终输出结果应该是:60。

10.4.2 nodelet实现

nodelet本质也是插件，实现流程与插件实现流程类似，并且更为简单，不需要自定义接口，也不需要使用类加载器加载插件类。

需求:参考 nodelet 案例，编写 nodelet 插件类，可以订阅输入数据，设置参数，发布订阅数据与参数相加的结果。

流程:

1. 准备；
2. 创建插件类并注册插件；
3. 构建插件库；
4. 使插件可用于ROS工具链；
5. 执行。

1.准备

新建功能包，导入依赖: roscpp、nodelet；

2.创建插件类并注册插件

```

1 #include "nodelet/nodelet.h"
2 #include "pluginlib/class_list_macros.h"
3 #include "ros/ros.h"
4 #include "std_msgs/Float64.h"
5
6 namespace nodelet_demo_ns {
7 class MyPlus: public nodelet::Nodelet {
8     public:
9     MyPlus(){
10         value = 0.0;
11     }
12     void onInit(){
13         //获取 NodeHandle
14         ros::NodeHandle& nh =
getPrivateNodeHandle();
15         //从参数服务器获取参数

```

```

16         nh.getParam("value",value);
17         //创建发布与订阅对象
18         pub = nh.advertise<std_msgs::Float64>
19             ("out",100);
20             sub = nh.subscribe<std_msgs::Float64>
21             ("in",100,&MyPlus::doCb,this);
22
23     void doCb(const std_msgs::Float64::ConstPtr&
24     p){
25         double num = p->data;
26         //数据处理
27         double result = num + value;
28         std_msgs::Float64 r;
29         r.data = result;
30         //发布
31         pub.publish(r);
32     }
33     private:
34     ros::Publisher pub;
35     ros::Subscriber sub;
36     double value;
37 };
38 }
39 PLUGINLIB_EXPORT_CLASS(nodelet_demo_ns::MyPlus,nod
elet::Nodelet)

```

3.构建插件库

CMakeLists.txt配置如下：

```

1 ...
2 add_library(mynodeletlib
3   src/myplus.cpp
4 )
5 ...
6 target_link_libraries(mynodeletlib
7   ${catkin_LIBRARIES}
8 )

```

编译后，会在 `工作空间/devel/lib/` 先生成文件：
`libmynodeletlib.so`。

4.使插件可用于ROS工具链

4.1配置xml

新建 xml 文件，名称自定义(比如:my_plus.xml)，内容如下：

```

1 <library path="lib/libmynodeletlib">
2   <class name="demo04_nodelet/MyPlus"
3     type="nodelet_demo_ns::MyPlus"
4     base_class_type="nodelet::Nodelet" >
5     <description>hello</description>
6   </class>
7 </library>

```

4.2导出插件

```

1 <export>
2   <!-- Other tools can request additional
3     information be placed here -->
4   <nodelet plugin="${prefix}/my_plus.xml" />
5 </export>

```

5. 执行

可以通过launch文件执行nodelet，示例内容如下：

```
1 <launch>
2   <node pkg="nodelet" type="nodelet" name="my"
3     args="manager" output="screen" />
4   <node pkg="nodelet" type="nodelet" name="p1"
5     args="load demo04_nodelet/MyPlus my"
6     output="screen">
7     <param name="value" value="100" />
8     <remap from="/p1/out" to="con" />
9   </node>
10  <node pkg="nodelet" type="nodelet" name="p2"
11    args="load demo04_nodelet/MyPlus my"
12    output="screen">
13    <param name="value" value="-50" />
14    <remap from="/p2/in" to="con" />
15  </node>
16
17 </launch>
```

运行launch文件，可以参考上一节方式向 p1发布数据，并订阅p2输出的数据，最终运行结果也与上一节类似。

10.5 本章小结

本章介绍了ROS中的一些进阶内容，主要内容如下：

- Action 通信；
- 动态参数；
- pluginlib；
- nodelet。

上述内容其实都是对之前通信机制缺陷的进一步优化：action较之于以往的服务通信是带有连续反馈的，更适用于耗时的请求响应场景；动态参数较之于参数服务器实现，则可以保证参数读取的实时性；最后，nodelet可以动态加载多个节点到同一进程，不再是一个节点独占一个进程，从而可以零成本的实现不同节点之间的数据交互，降低了数据传输的延时，提高了数据传输的效率；当然，nodelet是插件的应用之一，所以在介绍 nodelet 之前，我们又先学习了 pluginlib，借助 pluginlib 可以实现可插拔的设计，让程序更为灵活、易于扩展且方便维护。