

# ROS 入门与实践教程-睿抗比赛实践

ROS Tutorials for RAICOM



v2025.0.1

## 序言

本教程来源于 ASIR 研究组的本科生及研究生同学，在学习深圳元创兴公司提供的相关视频及文字资料后的总结笔记，由河南工业大学 ASIR 研究组整理。教程旨在为 ASIR 研究组内有意参加 RAICOM 竞赛的同学提供 ROS 学习及参赛项目实验参考。

教程内容开源，并由 ASIR 研究组同学（在 RAICOM 竞赛中获得国家级一等奖、二等奖多项）负责维护更新。

**指导教师：**吴翔，张世杰，张慧，王继龙

**v2025.0.1 版本贡献者名单：**

- 全篇内容 **LATEX** 整理排版 - 研 24 级张家建 | 自动化 2205 张钰浠
- 环境配置 - 机器人 2302 刘振翔
- Linux 基础 - 机器人 2201 闫佳宇
- 开发工具 - 机器人 2201 李宗洋
- ROS 开发流程简介 - 机器人 2301 郑高星
- 文件结构及构建 - 机器人 2301 郑高星
- ROS-Node - 机器人 2302 李博
- ROS-Topic - 机器人 2201 郑文轩
- ROS 常用组件 - 机器人 2201 李宗洋
- Service、Bash、Launch - 机器人 2201 白银多

## 目录

<b>1 ROS 简介</b>	<b>1</b>
1.1 概述 . . . . .	1
1.2 设计理念与核心目标 . . . . .	1
1.3 架构层次解析 . . . . .	1
1.3.1 系统层级架构 . . . . .	2
1.3.2 ROS 内部结构 . . . . .	2
1.4 核心优势与典型应用 . . . . .	2
1.4.1 技术优势 . . . . .	2
1.4.2 应用场景 . . . . .	3
1.5 挑战与演进方向 . . . . .	3
1.6 延伸阅读 . . . . .	3
<b>2 ROS 基础教程</b>	<b>4</b>
2.1 Linux 基础 . . . . .	4
2.1.1 Linux 常用命令 . . . . .	4
2.1.2 linux 系统组成 . . . . .	7
2.1.3 ROS 的常见技巧 . . . . .	10
2.2 ROS 的开发流程 . . . . .	14
2.3 ROS 的环境配置 . . . . .	15
2.4 ROS 的开发工具 . . . . .	18
2.5 ROS 的常用组件 . . . . .	22
2.6 ROS 的文件构建 . . . . .	32
2.6.1 Catkin 编译系统与工作空间 . . . . .	32
2.6.2 ROS 工程文件系统架构 . . . . .	33
2.6.3 工作空间操作全流程 . . . . .	33

2.6.4 功能包 (Package) 详解 . . . . .	34
2.6.5 高级配置技巧 . . . . .	35
2.7 ROS-Node . . . . .	35
2.8 ROS-Topic . . . . .	50
2.8.1 Ros Topic 概述 . . . . .	50
2.8.2 工作流程 . . . . .	51
2.8.3 小乌龟实例 . . . . .	51
2.8.4 自定义实例 . . . . .	52
2.8.5 常用 ROS Topic 指令 . . . . .	53
2.9 ROS-service、launch、bash . . . . .	53
2.9.1 ROS-service . . . . .	53
2.9.2 ROS_launch . . . . .	55
2.9.3 ROS_bash . . . . .	58
<b>3 工业组 . . . . .</b>	<b>58</b>
3.1 SSH 远程连接 . . . . .	58
3.2 机械臂的相机标定 . . . . .	62
3.2.1 camera_calibration 功能包 . . . . .	62
3.2.2 相机标定方法 . . . . .	63
3.3 自主充电 . . . . .	65
3.4 综合调度 . . . . .	65
<b>4 服务组 . . . . .</b>	<b>66</b>
4.1 导航 . . . . .	66
4.1.1 SLAM 建图 . . . . .	66
4.1.2 机器人定位 . . . . .	70
4.1.3 代价地图 . . . . .	70

4.1.4 路径规划 . . . . .	75
4.2 自主充电 . . . . .	85
4.3 语音和综合调度 . . . . .	88
4.3.1 环境配置 . . . . .	88
4.3.2 语音导航 . . . . .	89
4.3.3 客户端 launch 编写 . . . . .	91
4.3.4 综合调度 . . . . .	91
4.4 人脸识别 . . . . .	92
4.4.1 人脸识别概述 . . . . .	92
4.4.2 人脸识别实现 . . . . .	94

# 1 ROS 简介

## 1.1 概述

ROS（机器人操作系统，Robot Operating System）是一套专为机器人软件开发设计的开源操作系统架构。它不仅提供操作系统级别的底层功能（如进程管理、通信机制和硬件抽象层），还整合了丰富的工具、库和开发规范，极大地方便了开发者快速构建、测试和部署复杂的机器人系统。ROS 的出现推动了机器人领域的技术进步，加速了代码复用、模块化开发与跨平台协作，使项目的设计与实现更加高效灵活。



图 1: ROS 机器人系统架构示意图

## 1.2 设计理念与核心目标

ROS 的设计目标可概括为代码复用、松耦合架构与分布式计算，其核心理念体现为以下公式：

$$\text{ROS} = \text{Plumbing} + \text{Tools} + \text{Capabilities} + \text{Ecosystem}$$

- **Plumbing (通信机制)**: 实现节点间基于消息传递的交互，支持跨主机的分布式通信。
- **Tools (工具链)**: 包含开发调试工具（如 RViz、rqt）、仿真工具（如 Gazebo）和构建工具（如 catkin）。
- **Capabilities (高层功能)**: 封装导航、感知、运动规划等机器人核心能力。
- **Ecosystem (生态系统)**: 由全球开发者社区共同维护的开源生态，提供海量功能包与技术支持。

## 1.3 架构层次解析

ROS 架构可从不同视角进行描述，其核心层次如下：

### 1.3.1 系统层级架构

1. 操作系统层: ROS 作为元操作系统, 依赖底层 OS(如 Ubuntu Linux) 提供资源管理。支持 macOS 和 Windows 的有限功能。
2. 中间件层:
  - 通信协议: TCPROS/UDPROS 实现高效数据传输。
  - 实时通信: Nodelet 机制支持进程间零拷贝通信。
  - 基础库: 提供坐标变换 (TF)、运动控制 (MoveIt!) 等核心功能。
3. 应用层: 由功能包 (Package) 构成, 每个包包含节点 (Node)、配置文件与依赖关系。

### 1.3.2 ROS 内部结构

- 文件系统:
  - 工作空间 (Workspace): 包含 `src` (源码)、`build` (构建目录)、`devel` (开发目录)。
  - 功能包 (Package): 最小功能单元, 包含以下核心文件:
    - \* `CMakeLists.txt`: 定义编译规则与依赖项。
    - \* `package.xml`: 描述包元数据与版本信息。
    - \* `launch/`: 配置多节点启动脚本。
    - \* `msg/` 与 `srv/`: 定义消息与服务接口。
- 计算图:
  - 节点 (Node): 独立执行单元, 通过话题 (Topic) 与服务 (Service) 通信。
  - 消息 (Message): 标准化的数据结构, 支持跨语言交互。
  - 参数服务器 (Parameter Server): 全局配置管理。

## 1.4 核心优势与典型应用

### 1.4.1 技术优势

- 模块化设计: 功能包独立开发与复用, 降低系统耦合度。
- 分布式计算: 节点可跨主机部署, 支持负载均衡与容错。
- 多语言支持: C++、Python、Java 等多语言接口, 方便开发者选择适合自己的开发语言。

- **工具链完备**: 从调试 (rqt\_console)、数据可视化 (rqt\_plot) 到仿真 (Gazebo、RViz) 的全流程工具，提升开发效率。
- **跨平台支持**: 支持多种操作系统 (如 Ubuntu、Windows、macOS)，并且 ROS 2 提供了对实时系统 (如 RTOS) 的支持。

#### 1.4.2 应用场景

- **无人驾驶**: 如 Apollo 平台基于 ROS 实现感知、路径规划和控制决策；AutoWare 等开源项目也采用 ROS 构建自动驾驶系统。
- **工业机器人**: ABB、KUKA 等厂商集成 ROS 进行柔性控制和智能化生产线管理；通过 MoveIt! 实现复杂的运动规划。
- **服务机器人**: Pepper、NAO 等机器人利用 ROS 实现语音识别、人脸识别等人机交互功能。
- **医疗机器人**: 通过 ROS 控制高精度设备，如手术辅助机器人 (如 Da Vinci 系统的研究版本)。
- **无人机与移动平台**: 基于 ROS 开发的无人机和移动机器人，用于灾后救援、智能物流、环境监测等应用场景。

### 1.5 挑战与演进方向

- **实时性限制**: ROS 1.x 基于 TCP/IP 的通信架构存在一定延迟，无法满足高实时性。ROS 2 引入了基于 DDS (Data Distribution Service) 协议的通信机制，显著提升了实时性和可靠性。
- **安全性增强**: 在工业应用和敏感场景中，ROS 需要更加完善的安全机制，包括身份认证、数据加密和通信安全协议。ROS 2 通过 DDS 的安全扩展初步解决了部分问题，但仍需进一步改进。
- **硬件兼容性**: 未来需要支持更多异构硬件 (如 FPGA、TPU、GPU 和边缘计算设备)，以适应复杂的机器人任务需求。
- **性能优化**: 针对大规模机器人集群和复杂系统，优化 ROS 的性能以支持高并发任务。
- **国际化与标准化**: 推动 ROS 在全球范围内的标准化和规范化，使其在更多国家和行业成为通用的机器人开发平台。

### 1.6 延伸阅读

- ROS 官方文档: <https://wiki.ros.org>
- ROS Industrial 联盟: <https://rosindustrial.org>
- ROS 2 设计白皮书: <https://design.ros2.org>

## 2 ROS 基础教程

### 2.1 Linux 基础

#### 2.1.1 Linux 常用命令

##### 1. **pwd 命令:**

pwd 命令用于显示用户当前所处的目录。如果用户不知道自己当前所处的目录，就必须使用它。

Listing 1: pwd 命令执行示例

```
[root@NIEL7-1 etc]# pwd  
/etc
```

##### 2. **cd 命令:**

cd 命令用来在不同的目录中进行切换。用户在登录系统后，会处于用户的家目录 (\$HOME) 中，该目录一般以/home 开始，后跟用户名，这个目录就是用户的初始登录目录 (root 用户的家目录为/root)。如果用户想切换到其他的目录中，就可以使用 cd 命令，后跟想要切换的目录名。例如：

Listing 2: cd 命令操作示例

```
# 基础目录操作  
[root@RHEL7-1 etc]# cd          # 返回用户家目录 (/root)  
[root@RHEL7-1 ~]# cd din1       # 进入当前目录下的din1子目录  
[root@RHEL7-1 din1]# cd ~        # 使用波浪符返回家目录  
[root@RHEL7-1 ~]# cd ..         # 返回当前目录的父目录 (正确用法)  
[root@RHEL7-1 /]# cd            # 再次返回家目录  
[root@RHEL7-1 ~]# cd ../etc     # 进入父目录下的etc子目录  
[root@RHEL7-1 etc]# cd /din1/subdin1 # 使用绝对路径进入指定目录
```

##### 3. **ls 命令:**

ls 命令用来列出文件或目录信息。该命令的语法为：ls [参数] [目录或文件]

ls 命令的常用参数选项如下：

- **-a:** 显示所有文件，包括以“.”开头的隐藏文件
- **-A:** 显示指定目录下所有子目录及文件（含隐藏文件），不显示“.”和“..”
- **-c:** 按文件的修改时间排序
- **-C:** 分成多列显示
- **-d:** 仅显示目录名称（常与 -l 联用）
- **-l:** 以长格式显示文件详细信息
- **-i:** 显示文件 i 节点号

Listing 3: ls 命令操作示例

---

```
# 文件列表操作
[root@RHEL7-1 ~]# ls          # 列出当前目录下的文件及目录
[root@RHEL7-1 ~]# ls -a        # 显示包括隐藏文件在内的所有文件
[root@RHEL7-1 ~]# ls -t        # 按最后修改时间排序
[root@RHEL7-1 ~]# ls -F        # 显示文件类型标识符
[root@RHEL7-1 ~]# ls -l        # 长格式显示文件详细信息
[root@RHEL7-1 ~]# ls -lg       # 长格式显示并包含工作组信息
[root@RHEL7-1 ~]# ls -R       # 递归显示所有子目录内容
```

---

#### 4. more 命令：

在使用 cat 命令时，如果文件太长，用户只能看到文件的最后一部分。这时可以使用 more 命令，一页一页地分屏显示文件的内容。more 命令通常用于分屏显示文件内容。大部分情况下，可以不加任何参数选项执行 more 命令查看文件内容。执行 more 命令后，进入 more 状态，按“Enter”键可以向下移动一行，按“Space”键可以向下移动一页；按“Q”键可以退出 more 命令。该命令的语法为：`more [参数] 文件名`

more 命令的常用参数选项如下：`-num`：这里的 num 是一个数字，用来指定分页显示时每页的行数。`+num`：指定从文件的第 num 行开始显示。

例如：{[root@RHEL7-1]}#more file1 # 以分页方式查看 file1 文件的内容 {[root@RHEL7-1]}#cat file1 | more # 以分页方式查看 file1 文件的内容

#### 5. mkdir 命令：

`mkdir` 命令用于创建一个目录。该命令的语法为：`mkdir [参数] 目录名`

上述目录名可以为相对路径，也可以为绝对路径。`mkdir` 命令的常用参数选项如下：`-p`：在创建目录时，如果父目录不存在，则同时创建该目录及该目录的父目录。

例如：{[root@RHEL7-1]}#mkdir dir1 # 在当前目录下创建 dir1 子目录。 {[root@RHEL7-1]}#mkdir -p dir2/subdir2 # 在当前目录下的 dir2 目录中创建 subdir2 子目录

#### 6. rmdir 命令：

`rmdir` 命令用于删除空目录。该命令的语法为：`rmdir [参数] 目录名`

上述目录名可以为相对路径，也可以为绝对路径。但所删除的目录必须为空目录。`rmdir` 命令的常用参数选项如下：`-p`：在删除目录时，一同删除父目录，但父目录中必须没有其他目录及文件。

例如：{[root@RHEL7-1]}#rmdir dir1 # 在当前目录下删除 dir1 空子目录 {[root@RHEL7-1]}#rmdir -p dir2/subdir2 # 删除当前目录中 dir2/subdir2 子目录

#### 7. rm 命令：

`rm` 命令主要用于文件或目录的删除。该命令的语法为：`rm [参数] 文件名或目录名`

`rm` 命令的常用参数选项如下：

- i: 删除文件或目录时提示用户。
- f: 删除文件或目录时不提示用户。
- R: 递归删除目录，即包含目录下的文件和各级子目录。

### 8. mv 命令:

mv 命令主要用于文件或目录的移动或改名。该命令的语法为: `mv [参数] 源文件或目录 目标文件或目录`

mv 命令的常用参数选项如下:

- -i: 如果目标文件或目录存在，则提示是否覆盖目标文件或目录。
- -f: 无论目标文件或目录是否存在，直接覆盖目标文件或目录，不提示。

Listing 4: 文件移动操作示例

---

文件移动与重命名

```
[root@NIEL7-1 ~]# mv file1.txt /tmp/ # 将file1.txt移动到/tmp目录
[root@NIEL7-1 ~]# mv file2.txt file3.txt # 将file2.txt重命名为file3.txt
[root@NIEL7-1 ~]# mv -i file4.txt /tmp/ # 交互式移动（目标存在时提示）
[root@NIEL7-1 ~]# mv -f file5.txt /tmp/ # 强制覆盖移动
```

---

Listing 5: 目录移动操作示例

---

目录操作

```
[root@NIEL7-1 ~]# mv dir1/ /opt/ # 移动目录到/opt
[root@NIEL7-1 ~]# mv dir2/ dir3/ # 重命名目录
[root@NIEL7-1 ~]# mv -i dir4/ /tmp/ # 交互式目录移动
[root@NIEL7-1 ~]# mv -f dir5/ /tmp/ # 强制覆盖移动目录
```

---

关键说明:

- 移动文件时需确保目标目录存在
- 重命名操作本质是同一目录下的移动
- 使用 -i 参数可避免意外覆盖
- 目录移动会同时移动其所有子内容
- 超级用户权限下需谨慎使用 -f 参数

### 9. touch 命令:

touch 命令用于建立文件或更新文件的修改日期。该命令的语法为: `touch [参数] 文件名或目录名`

touch 命令的常用参数选项如下:

- -d `yyyymmdd`: 把文件的存取或修改时间改为 `yyyy 年 mm 月 dd 日`。

- **-a**: 只把文件的存取时间改为当前时间。
- **-m**: 只把文件的修改时间改为当前时间。

例如：

Listing 6: 创建与更新时间戳示例

---

文件创建与时间更新  
[root@RHEL7-1 ~]# touch aa # 创建aa文件（若存在则更新时间戳）  
[root@RHEL7-1 ~]# stat aa # 查看文件详细信息

---

Listing 7: 指定时间戳修改示例

---

自定义时间戳  
[root@RHEL7-1 ~]# touch -d 20180808 aa # 将时间戳设为2018年8月8日  
[root@RHEL7-1 ~]# stat aa

---

关键说明：

- 创建空文件时文件大小为 0 字节
- 时间戳格式为 YYYY-MM-DD HH:MM:SS
- stat 命令可验证时间戳修改结果
- 时间修改不会影响文件内容
- 支持批量操作多个文件

10. **clear 命令：** clear 命令用于清除字符终端屏幕内容。等价于 **CTRL+L**。
11. **Linux 命令的特点：** 在 Linux 系统中，命令区分大小写。在命令行中，可以使用“Tab”键自动补齐命令。利用向上或向下的光标键，可以翻查曾经执行过的历史命令，并再次执行。如果要在命令行上输入和执行多条命令，可以使用分号来分隔命令，如“**cd /; ls**”。断开一个长命令行，可以使用反斜杠“\”，将一个较长的命令分成多行表达，增强命令的可读性。执行后，shell 自动显示提示符“>”，表示正在输入一个长命令，此时可继续在新行上输入命令的后续部分。

### 2.1.2 linux 系统组成

1. linux 系统组成
  - (a) **bootloader**: 用于启动内核。嵌入式平台的 vivi, RedBoot, u-boot 等，其中 u-boot 在使用上最广泛。
  - (b) **linux 内核**。
  - (c) **根文件系统**: rootfs。

## 2. U-Boot (Universal Boot Loader)

- (a) U-Boot 简介: linux 系统启动必须使用一个 bootloader 程序, 即芯片上电之后必须首先启动 bootloader 程序。这段程序会初始化 DDR 等外设, 然后将 linux 内核从 flash (MMC、NOR FLASH、SD、MMC) 拷贝到 DDR 中, 最后启动 linux 内核
- (b) U-Boot 来源: 1) uboot 官网代码, 更新快, 基本包含所有常用芯片; (外设驱动不完全支持)
  - 2) 半导体芯片厂商代码: 厂家维护 uboot, 专门针对自家芯片, 对自家外设驱动支持, 优于 uboot 官网
  - 3) 开发板厂商代码: 在半导体厂商 uboot 基础上加入自家开发板的支持。

## 3. linux 内核: 使用正点原子官方文档

## 4. 根文件系统

- (a) 简介: linux 中的根文件系统更像是一个文件夹或者目录, 在这里有很多子文件夹或者子目录, 里面存在 linux 运行所必须的库、命令、配置文件等。根文件系统是 linux 内核启动以后挂载的第一个文件系统, 然后从根文件系统中读取初始化脚本, 比如 rcS、inittab 等等。如果不提供根文件系统, linux 内核会报内核崩溃错误。

- (b) 系统内容:

```
laser@ubuntu16:/usr/arm-linux-gnueabihf/lib$ cd /
laser@ubuntu16:/$ ls
bin boot cdrom dev etc home initrd.img initrd.img.old lib
lost+found media mnt opt proc root run sbtn snap srv sys
tar vmlinuz vmlinuz.old
```

图 2: 系统内容

```
Laser@ubuntu16:~$ ls -lht
总用量 104K
drwxr-xrwt 26 root root 4.0K 1月 15 14:17 sys
drwxr-xr-x 32 root root 1.1K 1月 15 14:10 run
drwxr-xr-x 18 root root 3.9K 12月 14 20:34 dev
drwxr-xr-x 142 root root 12K 12月 5 17:28 etc
dr-xr-xr-x 319 root root 0 12月 1 11:36 proc
drwxr-xr-x 14 root root 4.0K 11月 3 09:41 usr
drwxr-xr-x 5 root root 4.0K 11月 1 14:19 home
drwxr-xr-x 3 root root 4.0K 10月 31 11:39 boot
drwxr-xr-x 2 root root 4.0K 10月 31 11:39 bin
drwxr-xr-x 2 root root 12K 10月 31 11:39 sbin
drwx----- 8 root root 4.0K 9月 23 01:16 root
drwxr-xr-x 3 root root 4.0K 9月 23 00:35 mnt
drwx----- 2 root root 16K 9月 23 00:26 lost+found
drwxr-xr-x 2 root root 4.0K 1月 19 2022 media
drwxr-xr-x 2 root root 4.0K 1月 19 2022 snap
drwxr-xr-x 14 root root 4.0K 1月 19 2022 var
drwxr-xr-x 5 root root 4.0K 6月 3 2021 opt
lrwxrwxrwx 1 root root 34 6月 3 2021 initrd.img -> boot/initrd
-142-generic
lrwxrwxrwx 1 root root 34 6月 3 2021 initrd.img.old -> boot/i
15.0-128-generic
lrwxrwxrwx 1 root root 31 6月 3 2021 vmlinuz -> boot/vmlinuz-
eneric
lrwxrwxrwx 1 root root 31 6月 3 2021 vmlinuz.old -> boot/vml
28-generic
drwxr-xr-x 23 root root 4.0K 6月 3 2021 lib
drwxr-xr-x 2 root root 4.0K 6月 3 2021 lib64
drwxr-xr-x 2 root root 4.0K 6月 3 2021 lib32
drwxr-xr-x 2 root root 4.0K 12月 17 2020 srv
drwxrwxr-x 2 root root 4.0K 12月 17 2020 cdrom
Laser@ubuntu16:~$
```

图 3: 系统内容

- 1) /bin 目录 bin 文件就是可执行文件。一般是一些命令，比如 ls、mv 等命令。
  - 2) /dev 目录 dev 是 device 的缩写，此目录下的文件都是设备文件。在 linux 下一切皆文件。例如 /dev/ttymxc0 就表示 imx6ull 的串口 0。
  - 3) /etc 目录存放各种配置文件。
  - 4) /lib 目录 lib 是 library 的简称，这些库文件是共享库，命令和用户编写的应用程序要使用这些库文件。
  - 5) /mnt 目录临时挂载目录，可以在此目录中创建空子目录，如 /mnt/sd、/mnt/usb 等。
  - 6) /proc 目录一般是空的，系统启动后挂载 proc 文件系统，这里存储系统运行信息文件。
  - 7) /usr 目录 usr 是 unix software resource 的缩写，存放着很多软件，一般系统安装完成以后，这个目录占用空间最多。
  - 8) /var 目录存放可改变的数据。
  - 9) /sbin 目录存放一些可执行文件或命令，但仅限管理员使用。
  - 10) /sys 目录系统启动后此目录作为 sysfs 文件系统的挂载点，是一个基于 ram 的文件系统，用于提供内核数据结构信息。
  - 11) /opt 可选的文件、软件存放区，由用户自行决定存放内容。
- (c) busybox 初步生成根文件系统通过 busybox 构建根文件系统，产生 bin、sbin、usr 文件夹，以及 linuxrc 这个文件。
- 1) bin: 可执行文件，一般就是命令 2) sbin: 管理员用的可执行文件 3) usr: 软件

Linuxrc 文件: linux 内核 init 进程会查找用户空间的 init 程序, 若 bootargs 设置为 init=/linuxrc, 那么 linuxrc 就作为用户空间的 init 程序 (由 busybox 生成)。

```
zuozhongkai@ubuntu:~/linux/nfs/rootfs$ ls  
bin  linuxrc  sbin  usr  
zuozhongkai@ubuntu:~/linux/nfs/rootfs$
```

图 4: Linuxrc 文件

添加/lib 库: 自行新建 lib 文件夹, 从交叉编译器获取 lib 库文件。

添加/usr/lib 库: 创建/usr/lib 目录, 将交叉编译器的 lib 库文件拷贝过去。

其他文件夹: 在根文件系统中创建 dev、proc、mnt、sys、tmp、root 等目录。在/etc 下创建启动配置文件等。

### 2.1.3 ROS 的常见技巧

#### 1. 如何查看隐藏文件:

Linux 系统下以“.”开头的文件即为隐藏文件。想要查看隐藏文件有下面两个方法:

方法一: 若在桌面图形化界面下, 进入需要显示的文件路径, 按 **Ctrl + h** 显示隐藏文件;

方案二: 若在命令行界面下, 则在需要显示的文件路径下使用命令行: **ls -a** 显示该文件夹下的所有文件, 包括隐藏文件。

#### 2. ubuntu 系统进入 github 慢的解决办法

ubuntu 系统中进入 github 特别慢, 解决方法如下:

1. 进入终端命令行模式, 输入: **sudo gedit /etc/hosts**
2. 用浏览器访问 [IPAddress.com](http://IPAddress.com), 使用 IP Lookup 工具获得 [github.com](https://github.com) 和 [github.global.ssl.fastly.net](https://github.global.ssl.fastly.net) 域名的 ip 地址
3. 在 gedit 打开的 /etc/hosts 文件中最后面添加如下格式:

```
140.81.112.1 github.com  
199.231.69.191 github.global.ssl.fastly.net
```

(上面两行中的 ip 要根据第 2 步中查询到的 ip 填写)

4. 保存 hosts 文件
5. 更新 DNS 缓存, 输入: **sudo /etc/init.d/networking restart**

### 3. 如何加入中文输入法

新装了 ubuntu20.04 的系统，由于装系统时，选择了安装英文版本，因此系统没有中文输入法，此时我想加入中文输入法



图 5: 添加输入源

我在“设置-区域与语言”中没有看到中文（智能拼音）这一项，因此没办法加入汉语拼音输入法。最终用下面方法解决了

- (a) 安装中文语言包由于我装系统时，选择了安装英文版本，所以先安装了一下中文语言包。

Listing 8: 安装中文语言包

---

```
更新软件包列表  
sudo apt-get update
```

---

```
安装简体中文语言包  
sudo apt-get install language-pack-zh-hans
```

- (b) 安装输入法

Listing 9: 安装输入法

---

```
安装ibus拼音输入法  
sudo apt install ibus-libpinyin
```

---

```
安装ibus输入法支持库  
sudo apt install ibus-clutter
```

安装完重启电脑

关键说明：

- `apt-get update` 用于更新软件包索引
- `language-pack-zh-hans` 提供简体中文支持
- `ibus-libpinyin` 是常用的拼音输入法引擎

- `ibus-clutter` 提供输入法框架支持
- 安装完成后需重启系统使配置生效

#### 4. 配置

重启电脑后，进入“设置-区域与语言”然后点击输入源下面的加号，

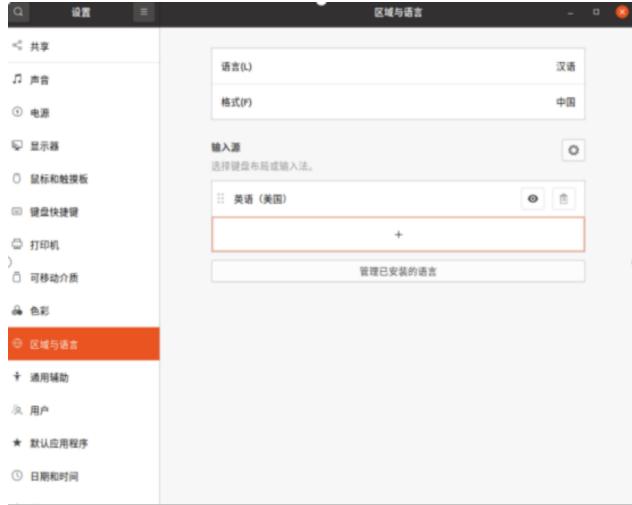


图 6: 配置

选择汉语出现了中文（智能拼音）这一选项，将这一项添加。

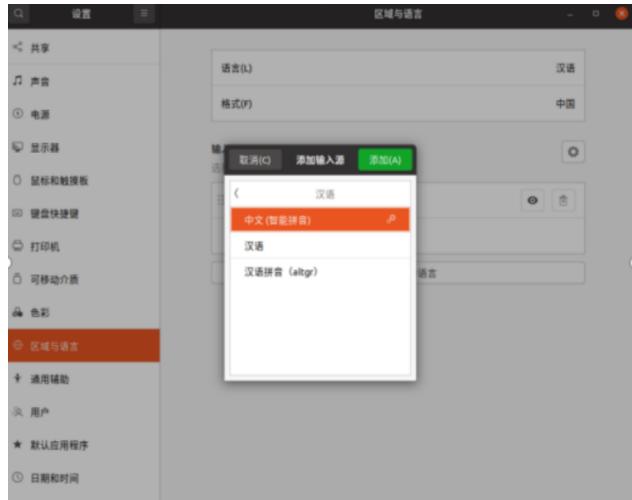


图 7: 添加

至此，中文输入法添加完毕。想要切换中英文输入法，点击桌面右上角的 `en`，选择“拼”，就可

以进行切换了。注意，输入法与系统的 Language 无关，在 English 的环境中，也可以使用中文输入法。

## 5. 如何换源

第一步打开图形界面配置



图 8: 图形界面配置

第二步选择站点在“下载自”中选择“其他站点”：

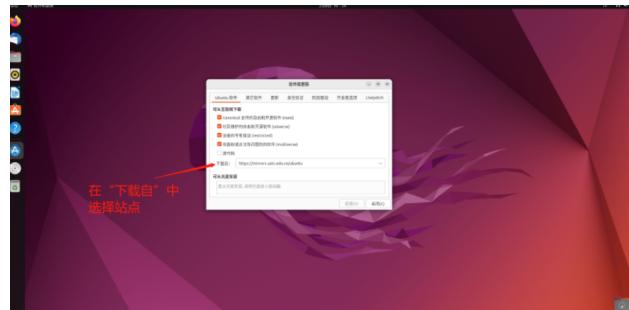


图 9: 选择站点

然后再“中国”条目下选择“mirrors.ustc.edu.cn”

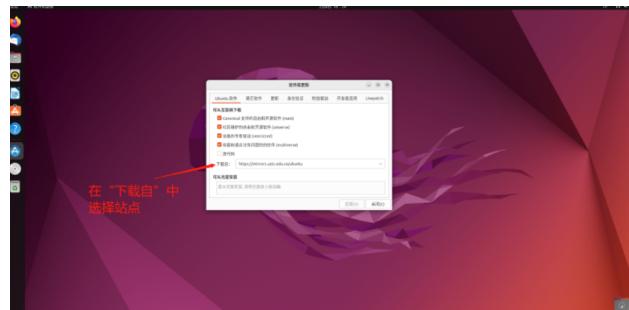


图 10: 完成换源

## 2.2 ROS 的开发流程

### ROS 开发流程简介

1. 环境搭建：首先需要在开发机器上安装适合的 ROS 版本，并配置环境变量。同时安装项目依赖，以确保开发环境的完整性。
2. 创建工作空间：开发前需要创建一个工作空间，用于管理项目中的各个包和代码文件。
3. 创建 ROS 包：根据项目需求创建一个包，包是 ROS 中功能的基本单位。包中包含源代码、配置文件以及依赖声明。
4. 开发节点（Node）：在包中编写节点代码，用 Python 或 C++ 实现具体功能。节点是 ROS 系统中的基本执行单元，它们通过话题（Topic）、服务（Service）等方式进行通信。
5. 启动和测试节点：通过 `roscore` 或直接运行节点，启动 ROS 主控进程，测试节点是否工作正常。
6. 创建 Launch 文件：如果需要同时运行多个节点，可以编写 Launch 文件，用于简化启动过程，自动管理节点启动顺序和参数。
7. 调试与优化：在程序运行中发现不足，耐心调试并进行优化升级。

### ROS 常用的概念

1. **node**: 节点。节点就是一些执行运算任务的进程。节点之间是通过传送消息进行通讯的；ROS 中，通常来讲我们写的 C++ 程序主函数所在的程序称为一个节点。
2. **message**: 消息。机器人需要传感器，传感器采集到的信息即 message(如位置消息，温度、湿度等)；消息以发布/订阅的方式传递。
3. **topic**: 话题。node 交换 Messages 的命名总线，异步通讯机制，传输消息。
4. **package**: 包。是组织 ROS 代码的最基本单位，每一个 Package 都可以包括库文件、可执行文件、脚本及其它文件。
5. **workspace**: 工作空间，用来存放很多不同 package。
6. **launch**: 启动文件，其目的是一次性启动多个节点。
7. **Master**: 节点管理器，ROS 名称服务，帮助节点找到彼此。
8. **publish**: 发布器，把相关的信息发送到 topic。
9. **subscribe**: 订阅器，订阅相应的 topic，接收话题的信息。

### 2.3 ROS 的环境配置

镜像文件和软件下载：首先声明，这个比赛官方配的 ROS 镜像文件是已经配好的，后续不需要再重新下载配置环境之类的，如果有兴趣想了解一下可以观看 b 站教学视频，不过学习睿抗的时候不建议用自己配的 ROS。

1. 这个压缩包体量很大，没有会员下载的话可以先挂在电脑上下载挂一晚上就差不多了，或者有一个同学有会员的话，下载完传输到 u 盘里面，其他同学直接用 u 盘复制一下就行，差不多十几分钟就结束了。

通过网盘分享的文件：RT4RAICOM [百度网盘](#) 提取码: vb4t

以上文件获取截至日期 2026/3/1。包含两个文件：分别为虚拟软件以及 ubuntu 18.04(包含 ros)的虚拟机系统。网盘资料如下图所示：

全部文件	文件名	大小	修改日期
	虚拟机软件以及机器人文件	-	2024-03-20 16:53

图 11: 网盘资料

文件夹内容文件如下图所示：

返回上一级   全部文件 - 虚拟机软件以及机器人文件
文件名
VMware17.0.exe
ubuntu元生.zip

图 12: 文件夹内容

2. 双击下载的虚拟机软件 VMware 17.0.exe

VMware17.0.exe	2024/3/20 14:18	应用程序	622.471 KB
----------------	-----------------	------	------------

图 13: 虚拟机软件

一路下一步即可，安装完成的时候需要输入许可码的时候，可使用一下许可码。激活密钥如下（任意一个都可以）：

JU090-6039P-08409-8J0QH-2YR7F
ZF3R0-FHED2-M80TY-8QYGC-NPKYF
FC7D0-D1YDL-M8DXZ-CYPZE-P2AY6
ZC3TK-63GE6-481JY-WWW5T-Z7ATA
1Z0G9-67285-FZG78-ZL3Q2-234JG

加载虚拟机 ubuntu18.04

1. 将下载的虚拟机系统如下图所示：



图 14: 虚拟机系统

解压到你想解压的地方 (磁盘至少需要 40G 以上的空间)，解压后如下图所示：



图 15: 解压后文件

2. 打开安装好的虚拟机软件，如下图所示：

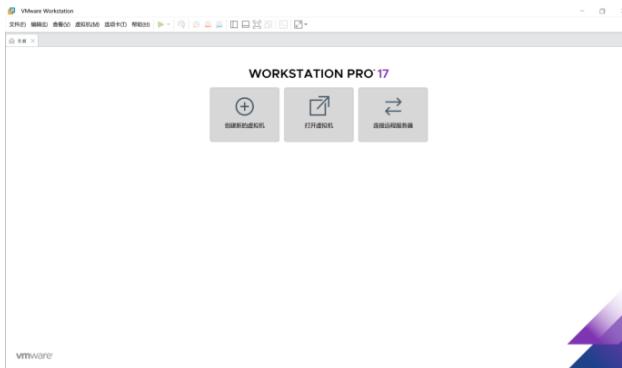


图 16: 虚拟机软件

3. 点击“打开虚拟机”，选择如下图所示，找到下载好的虚拟机系统所在的文件，也就是“ubuntu 元宝”文件夹下。

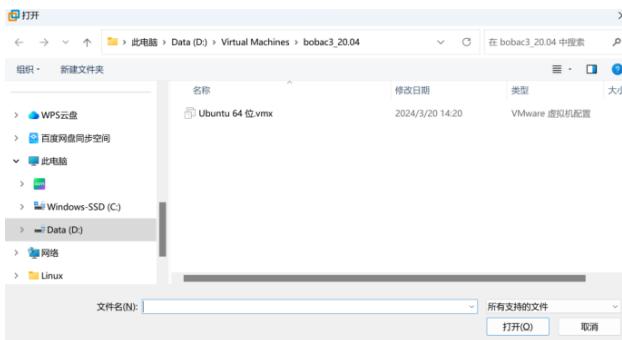


图 17: 文件夹内容

4. 选择上图中的”Ubuntu 64 位.vmx”文件，并点击打开，正常打开如下图所示：

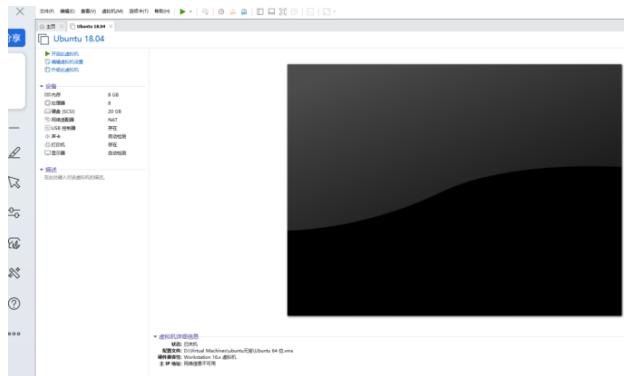


图 18: .vmx 文件

5. 点击上图中“开启此虚拟机”：



图 19: 开启问题

如遇上图内容选择“我已复制该虚拟机” 正常如下所示：



图 20: 正常启动

6. 进入系统：点击上图中的”bobac3” 输入密码：root 进入系统如下图所示：（可能你们的显示的是”ubuntu 18.04”，密码也是 root）

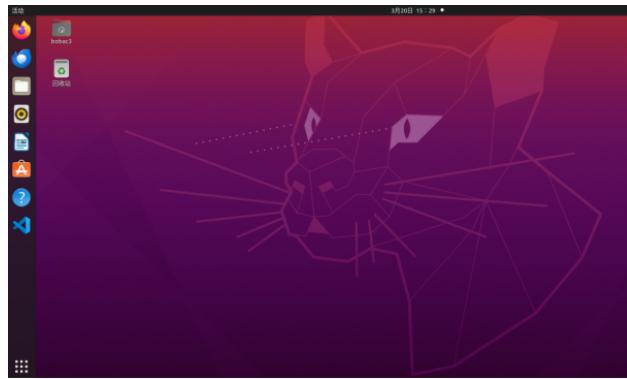


图 21: 完成安装

好啦，恭喜你已经完成了系统安装，加油吧，一年拿下国一。

## 2.4 ROS 的开发工具

### 开发工具介绍

#### 1. Terminator

(a) 介绍：在 ROS 中，需要频繁地使用到终端，且可能需要同时开启多个窗口。效果如下：

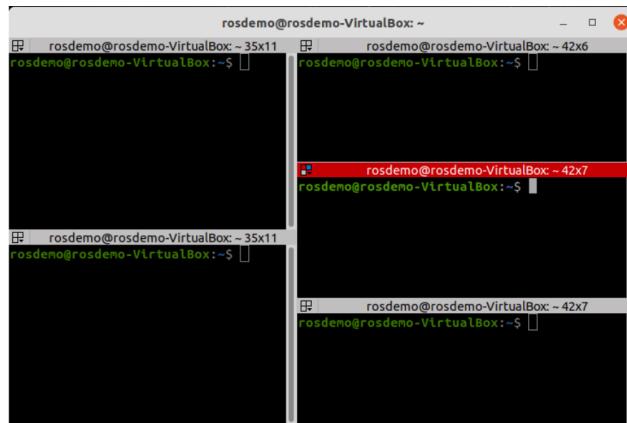


图 22: 效果展示

(b) 安装：(1)  $\text{Ctrl} + \text{Alt} + \text{T}$  打开终端 (2) 输入 `sudo apt install terminator`

(c) Terminator 常用快捷键 (1) 关于在同一标签内的操作

- $\text{Alt}+\text{Up}$  // 移动到上面的终端

- Alt+Down //移动到下面的终端
- Alt+Left //移动到左边的终端
- Alt+Right //移动到右边的终端
- Ctrl+Shift+0 //水平分割终端
- Ctrl+Shift+E //垂直分割终端
- Ctrl+Shift+Right //在垂直分割的终端中将分割条向右移动
- Ctrl+Shift+Left //在垂直分割的终端中将分割条向左移动
- Ctrl+Shift+Up //在水平分割的终端中将分割条向上移动
- Ctrl+Shift+Down //在水平分割的终端中将分割条向下移动
- Ctrl+Shift+S //隐藏/显示滚动条
- Ctrl+Shift+F //搜索
- Ctrl+Shift+C //复制选中的内容到剪贴板
- Ctrl+Shift+V //粘贴剪贴板的内容到此处
- Ctrl+Shift+W //关闭当前终端
- Ctrl+Shift+Q //退出当前窗口，所有终端都将被关闭
- Ctrl+Shift+X //最大化显示当前终端
- Ctrl+Shift+Z //最大化显示当前终端并使字体放大
- Ctrl+Shift+N or Ctrl+Tab //移动到下一个终端
- Ctrl+Shift+P or Ctrl+Shift+Tab //移动到之前的一个终端

## (2) 有关各个标签之间的操作

- F11 //全屏开关
- Ctrl+Shift+T //打开一个新的标签
- Ctrl+PageDown //移动到下一个标签
- Ctrl+PageUp //移动到上一个标签
- Ctrl+Shift+PageDown //将当前标签与其后一个标签交换位置
- Ctrl+Shift+PageUp //将当前标签与其前一个标签交换位置
- Ctrl+Plus (+) //增大字体
- Ctrl+Minus (-) //减小字体
- Ctrl+Zero (0) //恢复字体到原始大小
- Ctrl+Shift+R //重置终端状态
- Ctrl+Shift+G //重置终端状态并 clear 屏幕
- Super+g //绑定所有的终端，向一个终端输入会同步到所有终端
- Super+Shift+G //解除绑定

- Super+t //绑定当前标签的所有终端
- Super+Shift+T //解除绑定
- 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
- Super+i //打开一个新窗口，与原来的窗口使用不同的进程

## 2. VS CODE

(a) **简介:** 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。

(b) **安装 vscode:**

- (1) 下载 vscode 下载: Visual Studio Code - Code Editing. Redefined 历史版本下载: Visual Studio Code September 2024
- (2) 安装 sudo dpkg -i xxxx.deb
- (3) 卸载 sudo dpkg --purge code

## 3. 安装插件

(a) **中文（简体）语言包**

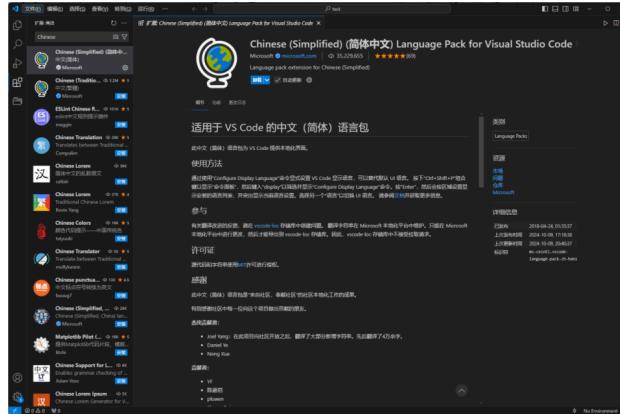


图 23: 中文（简体）语言包

(b) **ROS（最重要的插件），一定要安装预发布版本**

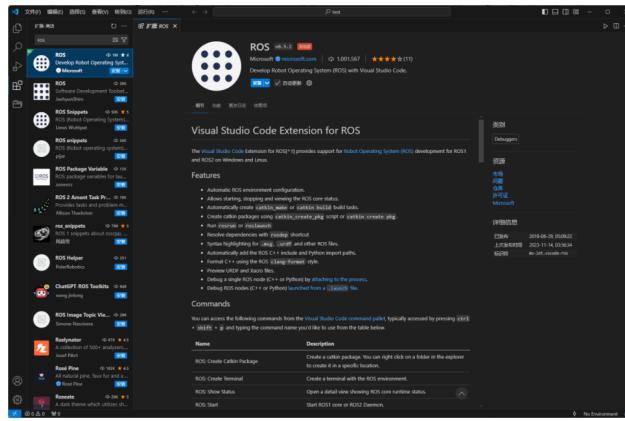


图 24: ROS

## (c) XML

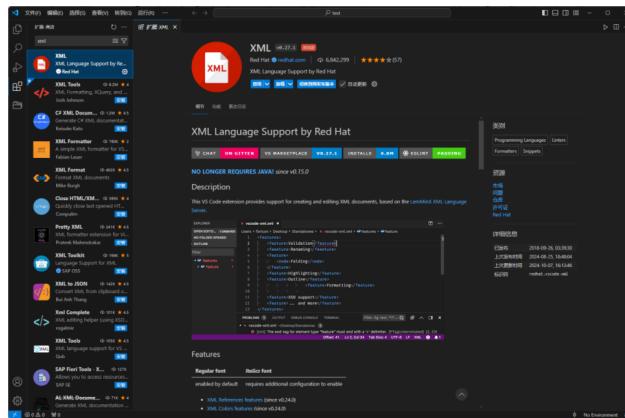


图 25: XML

## (d) YAML

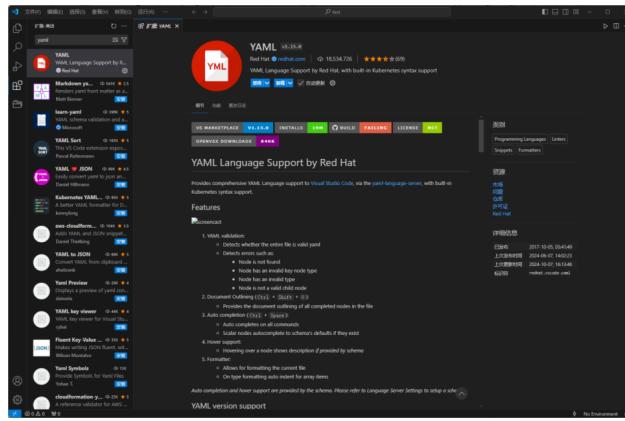


图 26: YAML

这些都是必备的，剩余的可自行安装自己喜欢用的插件

#### 4. 使用

##### (a) 创建 ROS 工作空间

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

注意：是 `xxxws` 不是 `src` 目录下。vscode 打开这个目录，会自动生成一个 `.vscode` 目录（不要删除）。

##### (b) 创建 ROS 功能包

- (1) `Ctrl+Shift+`` 打开终端进入 `src` 目录命令行创建功能包
- (2) 在 `src` 目录下右键点击选择 `create catkin package`

##### (c) 编写代码

##### (d) 调试代码

参考：vscode 下调试 ROS 项目，节点调试，多节点调试，roslaunch 调试 \_roslaunch debug-CSDN 博客。

## 2.5 ROS 的常用组件

### 1. rqt 工具箱

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。可以方便地实现 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。

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）;

安装: 一般只要安装的是 desktop-full 版本就会自带工具箱。如果需要安装，按照下面的方式（默认 ROS 版本为 noetic）安装:

---

```
sudo apt-get install ros-noetic-rqt
sudo apt-get install ros-noetic-rqt-common-plugins
```

---

基本使用:

启动 `rqt rosrun rqt_gui rqt_gui`

使用: 启动 rqt 之后，可以通过 plugins 添加所需的插件

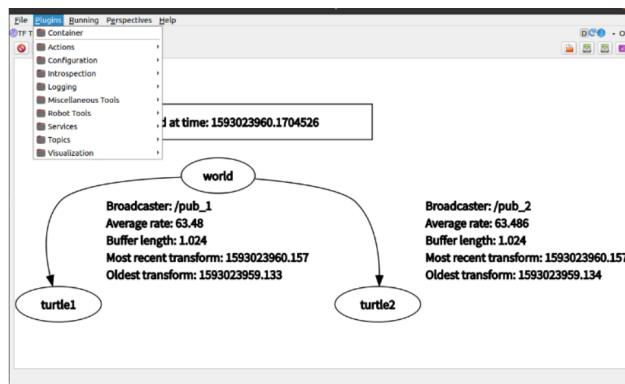


图 27: rqt 使用

## 2. rqt 常用插件

### (a) rqt\_graph

简介: 可可视化显示计算图启动: 可在 rqt 的 plugins 中添加，或者使用 `rqt_graph` 启动

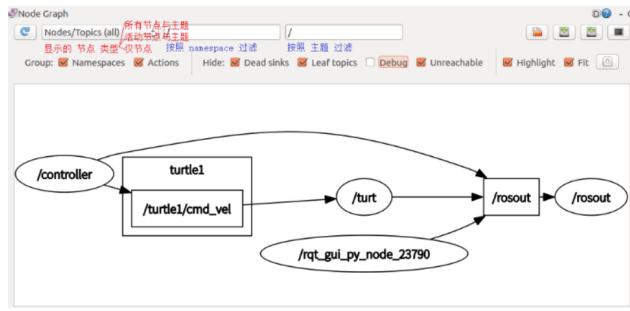


图 28: rqt\_graph 展示图

## (b) rqt\_console

简介: rqt\_console 是 ROS 中用于显示和过滤日志的图形化插件启动: 可以在 rqt 的 plugins 中添加, 或者使用 rqt\_console 启动

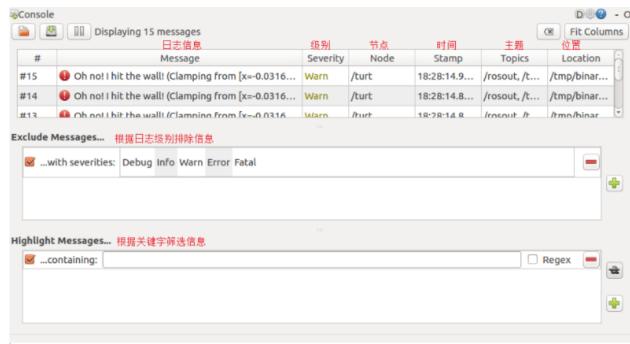


图 29: rqt\_console 展示图

## (c) 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

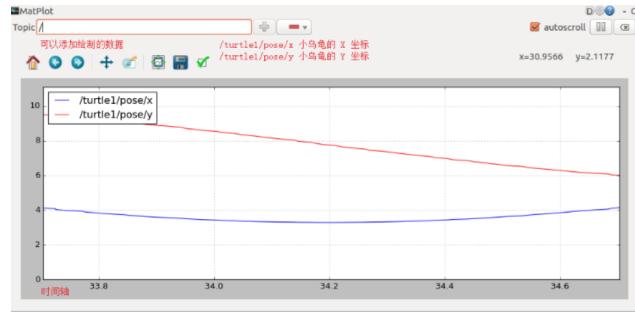


图 30: rqt\_plot 展示图

## (d) 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：

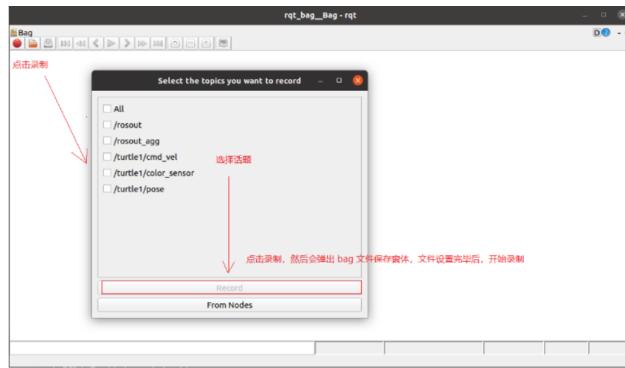


图 31: rqt\_bag 录制

重放：

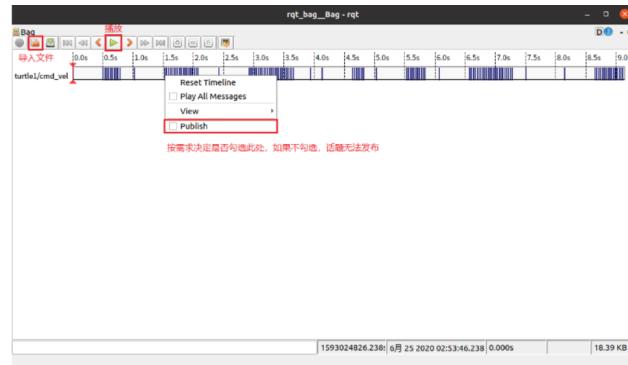


图 32: rqt\_bag 重放

### 3. rviz

#### (a) 概述

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，从相机获取的图像值等。

rviz 帮助开发者实现所有可监测信息的图形化显示，开发者也可以在 rviz 的控制界面下，通过按钮、滑动条、数值等方式，控制机器人的行为。

#### (b) 安装

一般只要安装的是 desktop-full 版本就会自带 rviz。如果需要安装，按照下面的方式（默认 ROS 版本为 noetic）安装：`sudo apt-get install ros-noetic-rviz`

#### (c) 基本使用

##### i. 启动

```
rqt
```

```
rosrun rviz rviz
```

##### ii. 界面介绍

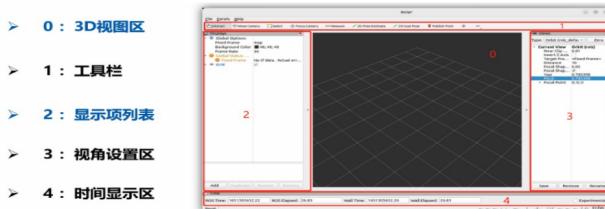


图 33: rviz 界面介绍

##### iii. 添加可视化数据

点击左下角的 Add 按钮，添加想要可视化的数据类型

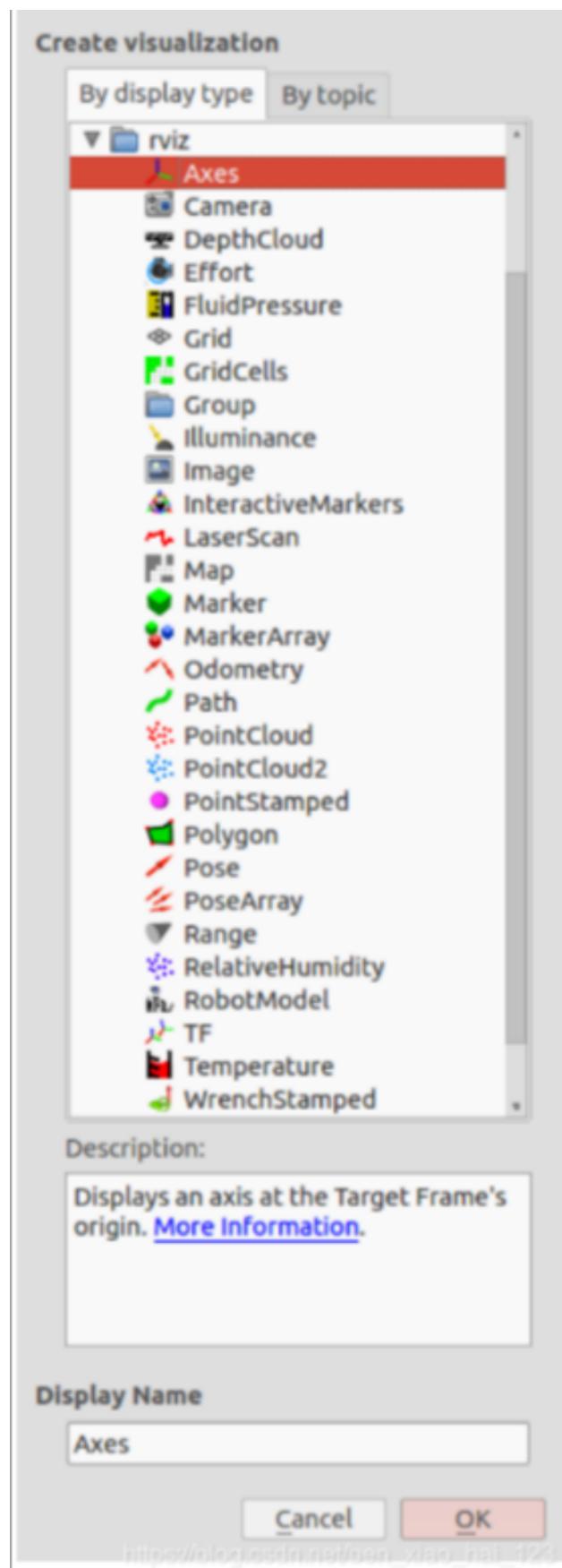


图 34: rviz 组件

#### iv. 组件简介

在 rviz2 中已经预定义了一些插件，这些插件名称、功能以及订阅的消息类型如下：

名称	功能	消息类型
Axes	显示 rviz2 默认的坐标系。	
Camera	显示相机图像，需要 CameraInfo。	sensor_msgs/msg/Image, sensor_msgs/msg/CameraInfo
Grid	以参考坐标系原点为中心的网格。	
Grid Cells	从网格中绘制单元格，常见于成本地图障碍物。	nav_msgs/msg/GridCells
Image	显示相机图像，但不需 CameraInfo。	sensor_msgs/msg/Image
InteractiveMarker	显示交互式标记服务器的 3D 对象，可鼠标交互。	visualization_msgs/msg/InteractiveMarker
Laser Scan	显示激光雷达数据。	sensor_msgs/msg/LaserScan
Map	显示地图数据。	nav_msgs/msg/OccupancyGrid
Markers	显示任意原始形状的几何体。	visualization_msgs/msg/Marker, MarkerArray
Path	显示导航路径数据。	nav_msgs/msg/Path
PointStamped	以小球形式绘制一个点。	geometry_msgs/msg/PointStamped
Pose	以箭头或坐标轴的方式绘制位姿。	geometry_msgs/msg/PoseStamped
Pose Array	绘制一组 Pose。	geometry_msgs/msg/PoseArray
Point Cloud2	绘制点云数据。	sensor_msgs/msg/PointCloud, sensor_msgs/msg/PointCloud2
Polygon	以线方式绘制多边形轮廓。	geometry_msgs/msg/Polygon
Odometry	显示里程计消息。	nav_msgs/msg/Odometry
Range	显示来自声纳或红外的距离测量圆锥。	sensor_msgs/msg/Range
RobotModel	显示机器人模型。	
TF	显示 tf 变换层次结构。	
Wrench	显示力（箭头）与扭矩（箭头 + 圆圈）。	geometry_msgs/msg/WrenchStamped
Oculus	将 RViz 场景渲染到 Oculus 头戴设备。	

#### v. 保存 rviz 配置

在可视化界面依次选择 File -> Save Config As 重命名为 xxx.rviz，然后保存到你选择的文件夹

#### vi. 使用已保存的 rviz 配置

在 launch 文件中添加以下内容：

```
<!-- Visualization - RViz-->
<node name="rviz" pkg="rviz" type="rviz"
      args="-d $(find xxx)/yyy/zzz.rviz"
      output="screen" />
```

-d 表示 load 加载 \$(find xxx) 表示定位到 xxx 功能包文件夹下 yyy 是 xxx 功能包文件夹下的 yyy 文件夹 zzz.rviz 是我们保存的 rviz 配置文件

#### 4. Gazebo

##### (a) 概述

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。

##### (b) 安装

一般只要安装的是 desktop-full 版本就会自带 rviz。如果需要安装，按照下面的方式（默认 ROS 版本为 noetic）安装：`sudo apt install gazebo`

##### (c) 基本使用

###### i. 启动

```
gazebo  
rosrun gazebo_ros gazebo
```

###### ii. 界面

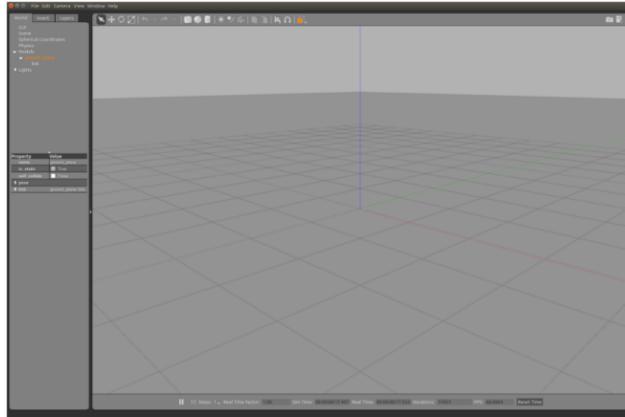


图 35: gazebo 界面

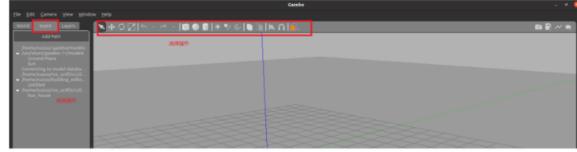
###### iii. 仿真环境搭建

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）

也可以直接下载使用官方或第三方提供的仿真环境插件。

**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

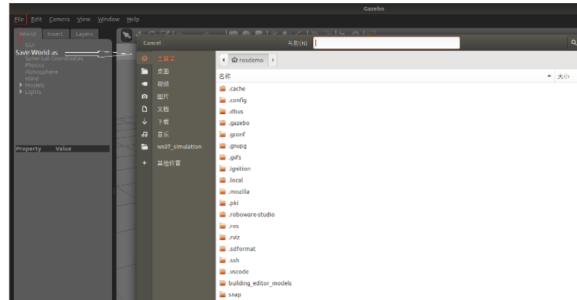
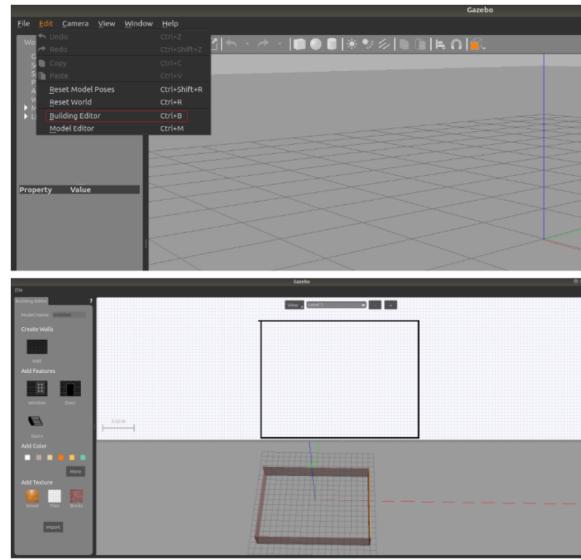


图 36: 仿真环境搭建

## 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



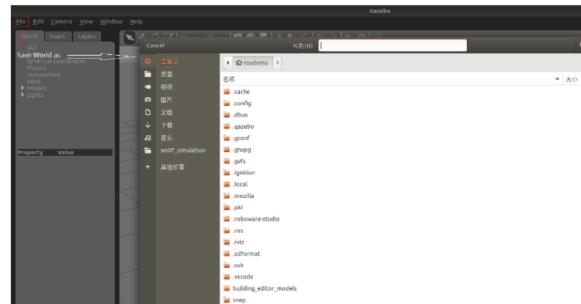
### 2.2 保存构建的环境

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

```
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

图 37: 仿真环境搭建

```

<launch>
    <!-- 将 Urdf 文件的内容加载到参数服务器 -->
    <param name="robot_description"
        command="$(find xacro)/xacro \
$(find demo02_urdf_gazebo)/urdf/xacro/my_base_camera_laser.urdf.xacro" />

    <!-- 启动 gazebo -->
    <include file="$(find gazebo_ros)/launch/empty_world.launch">
        <arg name="world_name"
            value="$(find demo02_urdf_gazebo)/worlds/hello.world" />
    </include>

    <!-- 在 gazebo 中显示机器人模型 -->
    <node pkg="gazebo_ros" type="spawn_model" name="model"
        args="-urdf -model mycar -param robot_description" />
</launch>

```

参数说明：核心代码：启动 `empty_world` 后，再根据 `arg` 加载自定义的仿真环境。

```

<include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name"
        value="$(find demo02_urdf_gazebo)/worlds/hello.world" />
</include>

<node pkg="gazebo_ros" type="spawn_model" name="model"
        args="-urdf -model mycar -param robot_description" />

```

在 Gazebo 中加载一个机器人模型，该功能由 `gazebo_ros` 下的 `spawn_model` 提供：

- `-urdf` 加载的是 urdf 文件
- `-model mycar` 模型名称是 mycar
- `-param robot_description` 从参数 `robot_description` 中载入模型
- `-x/-y/-z` 指定模型的坐标位置

(d) URDF 和 XACRO（自行了解）

## 2.6 ROS 的文件构建

### 2.6.1 Catkin 编译系统与工作空间

- 核心组件：

- `catkin`: ROS 定制的编译系统，基于 CMake 扩展
- 工作空间：包含 `build`, `devel`, `src` 三部分

- 核心文件：

- package.xml: 功能包元数据描述 (<name>, <depend> 等标签)
- CMakeLists.txt: 编译规则定义 (find\_package(), catkin\_package() 等指令)

## 2.6.2 ROS 工程文件系统架构

### 1. 典型结构:

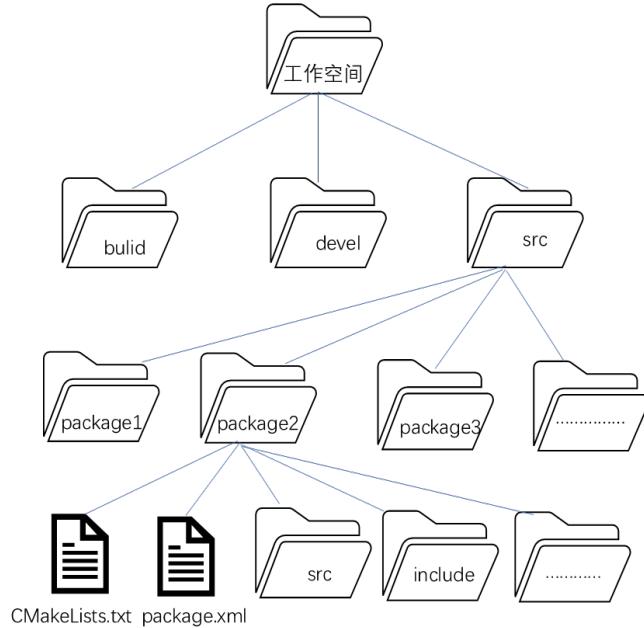


图 38: ROS 工程文件组织结构

### 2. 工作空间三要素:

- build/: 存放编译中间文件 (CMake 缓存、Makefile 等)
- devel/: 生成的可执行文件 (二进制程序、库文件、头文件)
- src/: 功能包源代码 (核心开发目录)

## 2.6.3 工作空间操作全流程

### 1. 创建工作空间:

```
mkdir -p ~/ros_workspace/src # 创建标准目录结构
```

### 2. 编译工作空间:

---

```
cd ~/ros_workspace
catkin_make # 执行编译
source devel/setup.bash # 激活环境
```

---

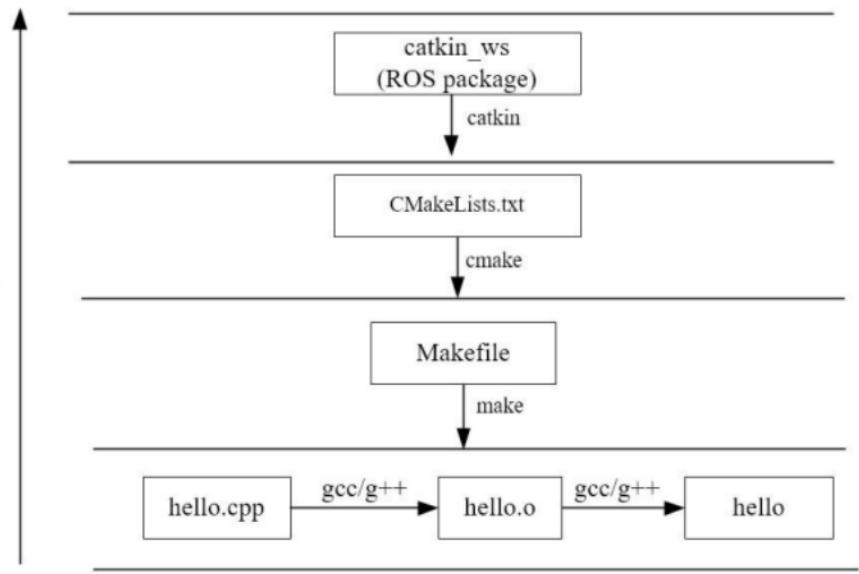


图 39: Catkin 编译流程示意图

### 3. 环境持久化配置:

---

```
echo "source ~/ros_workspace/devel/setup.bash" >> ~/.bashrc
source ~/.bashrc # 永久生效
```

---

#### 2.6.4 功能包 (Package) 详解

- 标准结构:

---

```
test_pkg/
    CMakeLists.txt      # 编译规则 (必须)
    package.xml        # 包描述文件 (必须)
    src/               # C++/Python源代码
    include/           # C++头文件
    launch/            # 启动配置
    msg/               # 自定义消息
```

---

- 创建功能包:

---

```
cd ~/ros_workspace/src
catkin_create_pkg test_pkg roscpp rospy std_msgs # 依赖核心包
```

---

- 包管理工具：

命令	功能
rospack list	列出所有 package
rosed [pkg]	跳转至 package 目录
rosls [pkg]	浏览 package 内容
rosdep install	自动安装依赖

## 2.6.5 高级配置技巧

- 多工作空间叠加：

---

```
source /opt/ros/noetic/setup.bash # 系统级
source ~/custom_ws/devel/setup.bash # 用户级
```

---

- 交叉编译配置：

- 设置 CMAKE\_TOOLCHAIN\_FILE
- 指定--cmake-args -DCMAKE\_BUILD\_TYPE=Release

## 2.7 ROS-Node

### 1. Node 以及 Master

在 ROS 的世界里，最小的进程单元就是节点（node）。一个功能包里可以有多个可执行文件，可执行文件在运行之后就成了一个进程（process），这个进程在 ROS 中就叫做节点。从程序角度来说，node 就是一个可执行文件（通常为 C++ 编译生成的可执行文件、Python 脚本）被执行，加载到了内存之中；从功能角度来说，通常一个 node 负责者机器人的某一个单独的功能。由于机器人的功能模块非常复杂，我们往往不会把所有功能都集中到一个 node 上，而会采用分布式的方式，把鸡蛋放到不同的篮子里。例如有一个 node 来控制底盘轮子的运动，有一个 node 驱动摄像头获取图像，有一个 node 驱动激光雷达，有一个 node 根据传感器信息进行路径规划……这样做可以降低程序发生崩溃的可能性，试想一下如果把所有功能都写到一个程序中，模块间的通信、异常处理将会很麻烦。

由于机器人的元器件很多，功能庞大，因此实际运行时往往会运行众多的 node，负责感知世界、控制运动、决策和计算等功能。那么如何合理的进行调配、管理这些 node？这就要利用 ROS 提供给我们的节点管理器 master，master 在整个网络通信架构里相当于管理中心，管理着各个 node。

node 首先在 master 处进行注册，之后 master 会将该 node 纳入整个 ROS 程序中。node 之间的通信也是先由 master 进行“牵线”，才能两两的进行点对点通信。当 ROS 程序启动时，第一步首先启动 master，由节点管理器处理依次启动 node。

## 2. roscpp

ROS 为机器人开发者们提供了不同语言的编程接口，比如 C++ 接口叫做 roscpp，Python 接口叫做 rospy，Java 接口叫做 rosjava。尽管语言不通，但这些接口都可以用来创建 topic、service、param，实现 ROS 的通信功能。Clinet Library 有点类似开发中的 Helper Class，把一些常用的基本功能做了封装。

目前 ROS 支持的 Client Library 包括：

Client Library	介绍
roscpp	ROS 的 C++ 库是目前最广泛应用的 ROS 客户端库，执行效率高
rospy	ROSD Python 库开发效率高，通常用在对运行时间没有太大要求的场合，例如配置，初始化等操作
roslisp	ROS 的 LISP 库
rosacs	Mono/.NET. 库可用任何 Mono/.NET 语言，包括 C#Iron, Python Iron Ruby 等
rosgo	ROSgo 语言库
rosjava	ROSjava 库
rosnodejs	javascript 客户端
...	...

目前最常用的只有 roscpp 和 rospy，而其余的语言版本基本都还是测试版。

## 3. Node 编写

大致介绍了 roscpp 的一些接口之后，我们来编写第一个 ROS 节点。

当执行一个 ROS 程序，就被加载到了内存中，就成为了一个进程，在 ROS 里叫做节点。每一个 ROS 的节点尽管功能不同，但都有必不可少的一些步骤，比如初始化、销毁，需要通行的场景通常都还需要节点的句柄。

- (a) 节点编写在我们建立发 ros\_workspace 的工作空间中创建了一个命名为 test\_pkg 的功能包。我们继续在 test\_pkg 中创建一个节点。在路径 ros\_workspace/src/test\_pkg/src 路径下创建一个 cpp 文件命名为:test.cpp。在终端输入一下命令进入指定路径;cd /ros\_workspace/src/test\_pkg/在终端输入一下命令创建 test.cpp: (注意运行上述名的时候一定要确保终端上的地址为 /ros\_workspace/src/test\_pkg/src) 正确的终端如下图所示:

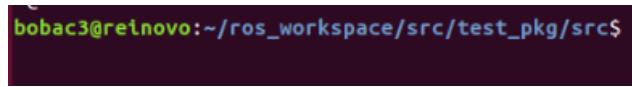


图 40: 终端

```
touch test.cpp
```

在终端输入以下命令打开 test.cpp

(注意运行上述名的时候一定要确保终端上的地址为 /ros\_workspace/src/test\_pkg/src)

正确的终端如下图所示：

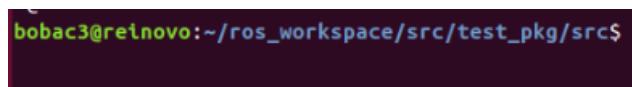


图 41: 终端

```
gedit test.cpp
```

输入以下代码：

```
#include<ros/ros.h>
#include<std_msgs/String.h>
#include<iostream>
int main(int argc,char **argv)
{ ros::init(argc,argv,"talker");
ros::NodeHandle n;
ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter",1000);
ros::Rate loop_rate(10);
int count = 0;
while(ros::ok())
{
    std_msgs::String msg;
    std::stringstream ss;
    ss<<"hello world" <<count;
    msg.data = ss.str();
    ROS_INFO("%s",msg.data.c_str());
    chatter_pub.publish(msg);
    ros::spinOnce();
}
```

```

loop_rate.sleep();
++count;
}
return 0;
}

```

上述测试代码创建了一个话题，并向其中发布“hello world”。

#### (b) CMakeLists.txt 文件修改

在完成了节点源码编写之后，需要修改 test\_pkg 包下的 CMakeLists.txt 文件（具体路径为：ros\_workspace/src/test\_pkg/CMakeLists.txt），将编辑的源码生成可执行文件。需要修改以下两条指令：

`add_executable(${PROJECT_NAME}_node src/test_pkg_node.cpp)` 该条指令用于指定将该功能包 src 目录下的哪个源文件编译为可执行程序，其中：

`${PROJECT_NAME}_node` 为生成可执行程序的名字，该名字可以任意指定。`.src/node.cpp` 为编译要使用的源码的文件名。

`target_link_libraries(${PROJECT_NAME}_node ${catkin_LIBRARIES})` 该条指令用于指定所使用的一些链接库。其中：

`${PROJECT_NAME}_node` 是在 `add_executable` 中生成的可执行文件。如下图所示为修改完成的 CMakeLists.txt 文件

```

33 ## Declare a C++ executable
34 ## With catkin make all packages are built within a single CMake context
35 ## The recommended prefix ensures that target names across packages don't collide
36 add_executable(${PROJECT_NAME}_node src/test.cpp)
37
38 ## Rename 最后的执行文件节点名 prefix 根据需要修改成自己的源文件
39 ## The above recommended prefix causes long target names, and renaming renames the
40 ## target back to the shorter version for ease of user use
41 ## e.g. "roslaunch someones pkg node" instead of "roslaunch someones pkg someones.pkg.node"
42 # set_target_properties(${PROJECT_NAME}_node PROPERTIES OUTPUT_NAME node PREFIX "")
43
44 ## Add cmake target dependencies of the executable
45 ## same as for the library above
46 # add_dependencies(${PROJECT_NAME}_node ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})
47
48 ## Specify libraries to link a library or executable target against
49 target_link_libraries(${PROJECT_NAME}_node
50   ${catkin_LIBRARIES})
51 )                                         编译执行文件所需要的依赖库

```

图 42: CMakeLists.txt 修改

当修改完 CMakeLists.txt 文件后就可以在工作空间的根目录下使用 `catkin_make` 进行编译了。`$ cd /ros_workspace $ catkin_make` 编译成功如下图所示：

```

reinovo@reinovo-ThinkPad-E470c: ~/ros_workspace
-- Using CATKIN_TEST_RESULTS_DIR: /home/reinovo/ros_workspace/build/test_results
-- Found gmock sources under '/usr/src/gmock': gmock will be built
-- Found gtest sources under '/usr/src/gmock': gtests will be built
-- Found Python nosetests: /usr/bin/nosetests-2.7
catkin 0.7.14
-- BUILD_SHARED_LIBS is on
-- Traversing 1 packages in topological order:
--   - test_pkg
-- +++ processing catkin package: 'test_pkg'
-- ==> add_subdirectory(test_pkg)
-- Configuring done
-- Generating done
-- Build files have been written to: /home/reinovo/ros_workspace/build
#####
## Running command: "make -j4 -l4" in "/home/reinovo/ros_workspace/build"
#####
Scanning dependencies of target test_pkg_node
[ 50%] Building CXX object test_pkg/CMakeFiles/test_pkg_node.dir/src/test.cpp.o
[100%] Linking CXX executable /home/reinovo/ros_workspace/devel/lib/test_pkg/test_pkg_node
[100%] Built target test_pkg_node
reinovo@reinovo-ThinkPad-E470c:~/ros_workspace$ 

```

图 43: 编译完成

编译生成的可执行文件位于: /home/reinovo/ros\_workspace/devel/lib/test\_pkg/路径下, 命名为: test\_pkg\_node。大家可以自行查看是否有该文件生成。

- (c) 打开一个终端输入以下命令: \$ roscore //打开 Master 如下图所示:

```

roscore http://reinovo-ThinkPad-E470c:11311/
reinovo@reinovo-ThinkPad-E470c:~/ros_workspace$ roscore
... logging to /home/reinovo/.ros/log/5a5cf08e-be7f-11e9-9adf-54e1add34717/roslaunch-reinovo-ThinkPad-E470c-17503.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
WARNING: disk usage in log directory [/home/reinovo/.ros/log] is over 1GB.
It's recommended that you use the 'rosclean' command.

started roslaunch server http://reinovo-ThinkPad-E470c:46677/
ros_comm version 1.12.14

SUMMARY
=====

PARAMETERS
  * /rosdistro: kinetic
  * /rosversion: 1.12.14

NODES
auto-starting new master
process[master]: started with pid [17514]
ROS_MASTER_URI=http://reinovo-ThinkPad-E470c:11311/

setting /run_id to 5a5cf08e-be7f-11e9-9adf-54e1add34717
process[rosout-1]: started with pid [17527]
started core service [/rosout]

```

图 44: 打开 master

重新打开一个终端输入以下命令运行 test\_pkg\_node 节点。\$ rosrun test\_pkg test\_pkg\_node 正确运行如下图所示:

```

reinovo@reinovo-ThinkPad-E470c:~$ rosrun test_pkg test_pkg_node
[ INFO] [1565779310.145610714]: hello world0
[ INFO] [1565779310.245769234]: hello world1
[ INFO] [1565779310.345800932]: hello world2
[ INFO] [1565779310.446099537]: hello world3
[ INFO] [1565779310.545771902]: hello world4
[ INFO] [1565779310.646336994]: hello world5
[ INFO] [1565779310.745764603]: hello world6
[ INFO] [1565779310.845799448]: hello world7
[ INFO] [1565779310.945820833]: hello world8
[ INFO] [1565779311.045802291]: hello world9
[ INFO] [1565779311.145793090]: hello world10
[ INFO] [1565779311.246316023]: hello world11
[ INFO] [1565779311.345805622]: hello world12
[ INFO] [1565779311.445749873]: hello world13
[ INFO] [1565779311.545801673]: hello world14
[ INFO] [1565779311.645779403]: hello world15
[ INFO] [1565779311.745797859]: hello world16
[ INFO] [1565779311.846101295]: hello world17
[ INFO] [1565779311.945804511]: hello world18
[ INFO] [1565779312.045799738]: hello world19
[ INFO] [1565779312.145796854]: hello world20
[ INFO] [1565779312.245798525]: hello world21
[ INFO] [1565779312.345797426]: hello world22

```

图 45: 打开 master

注意：用 rosrun 运行节点之前一定要先运行 roscore 不然会报错。

- (d) rosnode 命令: ros 提供了用于处理 node 的命令 rosnode。rosnode 命令的详细作用列表如下：

rosnode 命令	作用
rosnode list	列出当前运行的 node 信息
rosnode info node_name	显示出 node 的详细信息
rosnode kill node_name	结束某个 node
rosnode ping	测试连接节点
rosnode machine	列出在特定机器或列表机器上运行的节点
rosnode cleanup	清除不可到达节点的注册信息

#### (e) 头文件引用

- 引用当前包头文件在编写代码时我们经常会需要引用头文件，引用公用的头文件很容易，因为它们已经在标准库头文件路径中。但是如果要引用自定义的头文件就稍微麻烦点，我们首先查看软件包的目录结构，需要在正确的目录下创建头文件并修改 CMakeLists.txt 文件这样才能正确编译，下面来举例说明：

为了方便说明我们来创建一个头文件。

在路径： /ros\_workspace/src/test\_pkg/include/test\_pkg 下创建 test\_pkg.h 打开终端输入以下命令进入要创建头文件的目录：

```
$ cd /ros_workspace/src/test_pkg/include/test_pkg
```

输入以下命令创建头文件：

```
$ touch test_pkg.h
```

输入以下命令打开创建的头文件：

```
$ gedit test_pkg.h
```

输入以下代码：

```
#ifndef _TEST_PKG_
#define _TEST_PKG_
#define TEST_PKG_VER "1.0.0"
#define INIT_COUNT 100
int addTwoNum(int a,int b);
#endif
```

完成后保存关闭头文件，修改 test.cpp，如下代码：

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include "test_pkg/test_pkg.h" //自定义头文件
#include <iostream>
int addTwoNum(int a,int b)
{
    return a+b;
}//头文件函数实现部分
int main(int argc, char *argv[])
{
    ros::init(argc, argv, "talker");//这个名字是给系统看的
    ros::NodeHandle n;
    ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
    ros::Rate loop_rate(10);
    int count = INIT_COUNT;//调用了头文件中的宏定义
    ROS_INFO("test_pkg version:%s",TEST_PKG_VER);
    while (ros::ok())
    {
        std_msgs::String msg;
        std::stringstream ss;
        ss << "hello world " << count;
        msg.data = ss.str();
        ROS_INFO("%s", msg.data.c_str());chatter_pub.publish(msg);
        ros::spinOnce();
        loop_rate.sleep();
        ++count;
    }
    return 0;
}
```

源码修改完成之后需要修改修改 test\_pkg 中的 CMakeLists.txt 文件。如下图所示（注意修改后要保存）：

```
include_directories(
    include
    ${catkin_INCLUDE_DIRS}
)
```

图 46: CMakeLists.txt 修改

此处修改是添加了一个头文件的包含路径：include。当然，此处的 include 是一个相对路径，指的是当前功能包 test\_pkg 下的 include。这样编译器在编译源码的时候会到 include 文件下查找我们自定义的头文件 test\_pkg/test\_pkg.h。

到此我们就完成如何在当前包下引用自定义头文件的代码修改以及配置修改。现在我们需要对源码重新编译。

```
$ cd /ros_workspace
$ catkin_make
```

编译完成之后运行：

```
$ roscore
$ rosrun test_pkg test_pkg_node
```

正确运行如下图所示，成功的输出了头文件中定义的版本号：version: 1.0.0。以及从 hello world 100 开始输出。

```
reinovo@reinovo-ThinkPad-E470c:~/ros_workspace$ rosrun test_pkg test_pkg_node
[INFO] [1565793496.502304864]: test_pkg version:1.0.0
[INFO] [1565793496.502361663]: hello world 100
[INFO] [1565793496.602509440]: hello world 101
[INFO] [1565793496.702446285]: hello world 102
[INFO] [1565793496.802602711]: hello world 103
[INFO] [1565793496.902493206]: hello world 104
[INFO] [1565793497.002471977]: hello world 105
[INFO] [1565793497.102505953]: hello world 106
[INFO] [1565793497.202513237]: hello world 107
[INFO] [1565793497.302507519]: hello world 108
[INFO] [1565793497.402786850]: hello world 109
[INFO] [1565793497.502510956]: hello world 110
[INFO] [1565793497.603021142]: hello world 111
[INFO] [1565793497.702876996]: hello world 112
[INFO] [1565793497.803015857]: hello world 113
[INFO] [1565793497.903013992]: hello world 114
[INFO] [1565793498.002503128]: hello world 115
[INFO] [1565793498.102877084]: hello world 116
[INFO] [1565793498.202837514]: hello world 117
[INFO] [1565793498.302476473]: hello world 118
[INFO] [1565793498.402983882]: hello world 119
[INFO] [1565793498.503030304]: hello world 120
```

图 47: 成功引用了自定义头文件

## ii. 引用同一工作空间内其他软件包的头文件

在一些情况我们需要引用其他软件包中提供的函数或宏定义，这样可以一定程度上减少我们在两个节点之间需要进行通信的话题个数，下面我们通过举例来进行说明。为了方便说明我们需要在之前创建的工作空间中创建一个新的 package 并命名为 my\_pkg。如下图所示：

```
corvin@workspace:~/ros_workspace/src$ catkin_create_pkg my_pkg std_msgs rospy test_pkg
Created file my_pkg/package.xml
Created file my_pkg/CMakeLists.txt
Created folder my_pkg/include/my_pkg
Created folder my_pkg/src
Successfully created files in /home/corvin/ros_workspace/src/my_pkg. Please adjust the values in package.xml.
corvin@workspace:~/ros_workspace/src$ ls
CMakeLists.txt [my_pkg] test_pkg
```

图 48: 创建 my\_pkg 功能包

如何通过 catkin\_create\_pkg 命令来创建功能包前面已经讲过，这里要注意的是我们创建 my\_pkg 的目的是要引用 test\_pkg 中的自定义头文件，因此这里创建 my\_pkg 的时候要对 test\_pkg 进行依赖。

在路径 my\_pkg/src 路径下创建源码文件 my\_pkg.cpp。写入如下代码：

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <iostream>
#include "test_pkg/test_pkg.h" //自定义头文件
int main(int argc, char *argv[])
{
    ros::init(argc, argv, "my_pkg");
    ros::NodeHandle n;
    ros::Publisher chatter_pub = n.advertise<std_msgs::String>("my_chatter", 1000);
    ros::Rate loop_rate(10);
    int count = INIT_COUNT;//调用了头文件中的宏定义
    ROS_INFO("test_pkg version:%s,init count:%d",TEST_PKG_VER,INIT_COUNT);//将其他软件包头文件中声明的宏定义打印出来
    while (ros::ok())
    {
        std_msgs::String msg;
        std::stringstream ss;
        ss << "my_pkg " << count;
        msg.data = ss.str();
        ROS_INFO("%s", msg.data.c_str());
        chatter_pub.publish(msg);
        ros::spinOnce();
        loop_rate.sleep();
        ++count;
    }
    return 0;
}
```

完成源码编写之后首先我们来修改 my\_pkg 的 CMakeLists.txt 文件。如下图所示，在 my\_pkg 的 CMakeLists.txt 指定了要编译的源码。

```
## Declare a C++ executable
## With catkin_make all packages are built within a single CMake context
## The recommended prefix ensures that target names across packages don't collide
add_executable(${PROJECT_NAME}_node src/my_pkg.cpp)

## Rename C++ executable without prefix
## The above recommended prefix creates long target names, the following renames the
## target back to the shorter version for ease of user use
## e.g. "rosrun someones_pkg node" instead of "rosrun someones_pkg someones_pkg_node"
# set_target_properties(${PROJECT_NAME}_node PROPERTIES OUTPUT_NAME node PREFIX "")

## Add cmake target dependencies of the executable
my_pkg软件包的CMakeLists.txt
## same as for the library above
文件需要修改的部分，跟其他一般
# add_dependencies(${PROJECT_NAME}_node ${${PROJECT_NAME}_lib})

## Specify libraries to link a library or executable target against
target_link_libraries(${PROJECT_NAME}_node
${catkin_LIBRARIES}
)
```

图 49: my\_pkg CMakeLists.txt 修改

其次来修改 test\_pkg 的 CMakeLists.txt 文件，如下图所示。做如下修改的主要目的是通知其他软件包当前软件包含有自定义头文件，在 include 目录下。这样其他包在引用头文件的时候就可以到这个地方查找。

```
#####
## catkin specific configuration ##
#####
## The catkin_package macro generates cmake config files for your package
## Declare things to be passed to dependent projects
## INCLUDE_DIRS: uncomment this if you package contains header files
## LIBRARIES: libraries you create in this project that dependent projects also need
## CATKIN_DEPENDS: catkin_packages dependent projects also need
## DEPENDS: system dependencies of this project that dependent projects also need
catkin_package(
INCLUDE_DIRS include
# LIBRARIES test_pkg
# CATKIN_DEPENDS roscpp rospy std_msgs
# DEPENDS system_lib
)
```

依赖包test\_pkg的CMakeLists.txt需要  
修改的部分，通知其他软件包当前软件包  
包含有自定义头文件，在include目录下

图 50: test\_pkg CMakeLitst.txt 修改

对工作空间进行编译：

```
$ cd /ros_workspace/
```

```
$ catkin_make
```

编译完成之后可以运行节点查看结果。

```
$ roscore
```

```
$ rosrun my_pkg my_pkg_node
```

运行结果如下图所示，打印了头文件中定义的宏定义说明调用头文件成功。

```

reinovo@reinovo-ThinkPad-E470c: ~$ rosrun my_pkg my_pkg_node
[ INFO] [1565869578.8709934235]: test_pkg version:1.0.0,init count:100
[ INFO] [1565869578.870994507]: my_pkg 100
[ INFO] [1565869578.971266641]: my_pkg 101
[ INFO] [1565869579.071065282]: my_pkg 102
[ INFO] [1565869579.171215012]: my_pkg 103
[ INFO] [1565869579.271130844]: my_pkg 104
[ INFO] [1565869579.371078205]: my_pkg 105
[ INFO] [1565869579.471048468]: my_pkg 106
[ INFO] [1565869579.571125868]: my_pkg 107
[ INFO] [1565869579.671130594]: my_pkg 108
[ INFO] [1565869579.771131503]: my_pkg 109
[ INFO] [1565869579.871113581]: my_pkg 110
[ INFO] [1565869579.971529349]: my_pkg 111
[ INFO] [1565869580.071527747]: my_pkg 112
[ INFO] [1565869580.171084942]: my_pkg 113
[ INFO] [1565869580.271142478]: my_pkg 114
[ INFO] [1565869580.371145861]: my_pkg 115
[ INFO] [1565869580.471118428]: my_pkg 116
[ INFO] [1565869580.571065736]: my_pkg 117
[ INFO] [1565869580.671096007]: my_pkg 118
[ INFO] [1565869580.771056980]: my_pkg 119
[ INFO] [1565869580.871058572]: my_pkg 120
[ INFO] [1565869580.971068676]: my_pkg 121

```

图 51: my\_pkg\_node 测试结果

大家也可以自行修改以下头文件中的宏定义，重新运行节点查看输出结果。

#### (f) launch 文件编写

在 ROS 中一个节点程序一般只能完成功能单一的任务，但是一个完整的 ROS 机器人一般由很多个节点程序同时运行、相互协作才能完成复杂的任务，因此这就要求在启动机器人时就必须要启动很多个节点程序，一般的 ROS 机器人由十几个节点程序组成，复杂的几十个都有可能。

这就要求我们必须高效率的启动很多节点，而不是通过 rosrun 命令来依次启动十几个节点程序，launch 文件就是为解决这个需求而生，launch 文件就是一个 xml 格式的脚本文件，我们把需要启动的节点都写进 launch 文件中，这样我们就可以通过 roslaunch 工具来调用 launch 文件，执行这个脚本文件就一次性启动所有的节点程序。

launch 文件同样也遵循着 xml 格式规范，是一种标签文本，它的格式包括以下标签：

```

<launch> <!-根标签->
  <node> <!-需要启动的 node 及其参数->
  <include> <!-包含其他 launch->
  <machine> <!-指定运行的机器->
  <env-loader> <!-设置环境变量->
  <param> <!-定义参数到参数服务器->
  <rosparam> <!-启动 yaml 文件参数到参数服务器->
  <arg> <!-定义变量->
  <remap> <!-设定参数映射->
  <group> <!-设定命名空间->
</launch> <!-根标签->

```

官方参考链接:<http://wiki.ros.org/roslaunch/XML>

为了方便说明我们创建一个功能包来进行讲解。

```
$ cd /ros_workspace/src/
$ catkin_create_pkg robot Bringup
一般每一个机器人工程都包含一个命名为 ×××_bringup 的功能包，该功能包用来存放该机器人的各种启动文件。
robot Bringup 功能包创建成功后在功能的目录下创建一个 launch 文件夹并在 launch 文件夹下创建一个命名为 startup.launch。
$ cd /ros_workspace/src/robot_Bringup/
$ mkdir launch
$ cd launch
$ touch startup.launch
```

(g) 标签

i. <launch> 标签

每个 XML 文件都必须要包含一个根元素，根元素由一对 launch 标签定义：<launch> ...</launch>。

打开 startup.launch 输入以下内容如图所示：

```
<launch>
</launch>
```

图 52: 空的 launch 文件

ii. <node> 标签

<node> 标签的上一级根标签为 <launch> 标签，用于启动一个 ROS 节点。启动一个节点的写法如下：

```
<node pkg="package-name" type="executable-name" name="node-name" />
pkg,type,name,output 属性：
```

<node> 标签下有若干属性，至少要包含三个属性： pkg, type, name。Pkg 属性指定了要运行的节点属于哪个功能包，type 是指要运行节点的可执行文件的名称，name 属性给节点指派了名称，它将覆盖任何通过调用 ros::int 来赋予节点的名称。

现在打开 startup.launch 文件在里面运行 test\_pkg\_node 以及 my\_pkg\_node 两个节点。代码如下：

```
<launch>
<node pkg="test_pkg" type="test_pkg_node" name="test_pkg_node" output="screen"/>
<node pkg="my_pkg" type="my_pkg_node" name="my_pkg_node" output="screen"/>
</launch>
```

其中，output=“screen”属性可以将单个节点的标准输出到终端而不是存储在日志文件中。

编译工作空间并运行 startup.launch 文件。

```
$ cd /ros_workspace/
```

```
$ catkin_make
```

运行 launch 的命令格式：

roslaunch 包名称 launch 文件名称

运行 startup.launch 文件。

```
$ roslaunch robot Bringup startup.launch
```

```
reinovo@reinovo-ThinkPad-E470c: ~/ros_workspace
reinovo@reinovo-ThinkPad-E470c:~/ros_workspace$ roslaunch robot Bringup startup.launch
... logging to /home/reinovo/.ros/log/beddca70-c036-11e9-9371-54e1add34717/roslaunch-reinovo-ThinkPad-E470c-28346.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
WARNING: disk usage in log directory [/home/reinovo/.ros/log] is over 1GB.
It's recommended that you use the 'rosclean' command.

started roslaunch server http://reinovo-ThinkPad-E470c:34305/
SUMMARY
=====
PARAMETERS
  * /rosdistro: kinetic
  * /rosversion: 1.12.14
NODES
  /
    my_pkg_node (my_pkg/my_pkg_node)
    test_pkg_node (test_pkg/test_pkg_node)

auto-starting new master
process[master]: started with pid [28356]
ROS_MASTER_URI=http://localhost:11311

setting /run_id to beddca70-c036-11e9-9371-54e1add34717
process[rosout-1]: started with pid [28369]
started core service [/rosout]
process[test_pkg_node-2]: started with pid [28373]
process[my_pkg_node-3]: started with pid [28382]
[ INFO] [1565967698.342004338]: test_pkg version:1.0.0
[ INFO] [1565967698.342096421]: hello world 100
[ INFO] [1565967698.347816539]: test_pkg version:1.0.0,init count:100
[ INFO] [1565967698.347884244]: my_pkg:150
[ INFO] [1565967698.442184147]: hello world 101
[ INFO] [1565967698.448115631]: my_pkg:151
[ INFO] [1565967698.542357207]: hello world 102
```

图 53: 运行结果

除了上述的 pkg, name, type, output 属性以外还有一些常用属性需要掌握。

### iii. <include> 标签

在当前 launch 文件中调用另一个 launch 文件，方便代码的复用，可以使用包含（include）标签：

```
<include file="$(find package-name)/launch-file-name">
```

由于直接输入路径信息很繁琐且容易出错，大多数包含元素都使用查找（find）命令搜索功能包的位置来替代直接输入绝对路径。

为了方便说明我们创建一个功能包 third\_pkg 并用 robot\_bringup 功能包中的 launch 物件来调用该功能包中的 launch 文件。

```
$ cd /ros_workspace/src/
$ catkin_create_pkg third_pkg std_msgs roscpp rospy
```

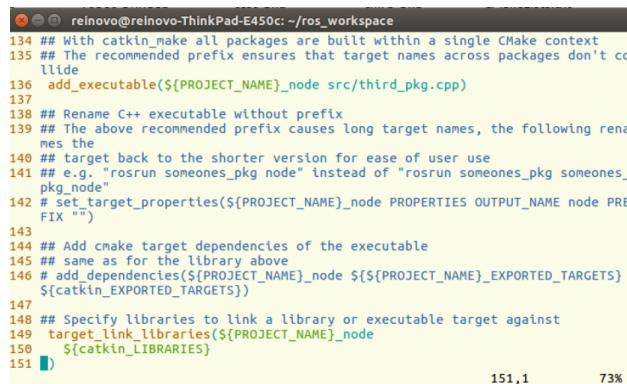
在 third\_pkg/src 路径下创建 third\_pkg.cpp 源文件，打开 third\_pkg.cpp 输入以下测试代码：

---

```
#include "ros/ros.h"
#include<std_msgs/String.h>
#include<iostream>
int main(int argc, char **argv)
{
    ros::init(argc, argv, "third_pkg");
    ros::NodeHandle n;/> 更新话题的消息格式为自定义的消息格式
    ros::Publisher chatter_pub = n.advertise<std_msgs::String>("third_pkg", 1000);
    ros::Rate loop_rate(10);
    int count = 0;
    while(ros::ok()){
        std_msgs::String msg;
        std::stringstream ss;
        ss<<"third_pkg:"<<count;
        msg.data = ss.str();
        ROS_INFO("%s", msg.data.c_str());
        chatter_pub.publish(msg); // 将消息发布到话题中
        ros::spinOnce();
        loop_rate.sleep();
        ++count;
    }
    return 0;
}
```

---

编辑源码之后同样要修改 CMakeLists.txt，将 third\_pkg.cpp 编译为可执行代码。如下图所示：



```
reinovo@reinovo-ThinkPad-E450: ~/ros_workspace
134 ## With catkin_make all packages are built within a single CMake context
135 ## The recommended prefix ensures that target names across packages don't collide
136 add_executable(${PROJECT_NAME}_node src/third_pkg.cpp)
137
138 ## Rename C++ executable without prefix
139 ## The above recommended prefix causes long target names, the following renames the
140 ## target back to the shorter version for ease of user use
141 ## e.g. "rosrun someones_pkg node" instead of "rosrun someones_pkg someones_
142 ## set_target_properties(${PROJECT_NAME}_node PROPERTIES OUTPUT_NAME node PREFIX "")"
143
144 ## Add cmake target dependencies of the executable
145 ## same as for the library above
146 # add_dependencies(${PROJECT_NAME}_node ${${PROJECT_NAME}_EXPORTED_TARGETS}
# ${catkin_EXPORTED_TARGETS})
147
148 ## Specify libraries to link a library or executable target against
149 target_link_libraries(${PROJECT_NAME}_node
150 ${catkin_LIBRARIES})
151
```

图 54: CMakeLists.txt 修改

编译工作空间。

在/third\_pkg 路径下创建一个 launch 文件夹：

```
$ cd /ros_workspace/src/third_pkg
$ mkdir launch
```

在 third\_pkg/launch 路径下新建一个 third\_pkg.launch，并输入以下代码：

```
<launch>
<node pkg="third_pkg" type="third_pkg_node" name="third_pkg_node" output="screen"/>
</launch>
```

打开 robot Bringup 功能包下的 startup.launch 文件，修改该如下：

```
<launch>
<node pkg="test_pkg" type="test_pkg_node" name="test_pkg_node" respawn="true"
      output="screen"/>
<node pkg="my_pkg" type="my_pkg_node" name="my_pkg_node" required="true"
      output="screen"/>
<include file="$(find third_pkg)/launch/third_pkg.launch"/>
</launch>
```

运行 startup.launch

```
$ rosrun robot Bringup startup.launch
```

```
reinovo@reinovo-ThinkPad-E450c:~$ rosrun robot Bringup startup.launch
... logging to /home/reinovo/.ros/log/1da1a56c-c0b0-11e9-816e-48e24458bd29/roslaunch-reinovo-ThinkPad-E450c-16566.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://reinovo-ThinkPad-E450c:35365/
SUMMARY
========
PARAMETERS
  * /rosdistro: kinetic
  * /rosversion: 1.12.14

NODES
/
  my_pkg_node (my_pkg/my_pkg_node)
  test_pkg_node (test_pkg/test_pkg_node)
  third_pkg_node (third_pkg/third_pkg_node)

auto-starting new master
process[master]: started with pid [16576]
ROS_MASTER_URI=http://localhost:11311

setting /run_id to 1da1a56c-c0b0-11e9-816e-48e24458bd29
process[rosout-1]: started with pid [16589]
started core service [/rosout]
process[test_pkg_node-2]: started with pid [16593]
process[my_pkg_node-3]: started with pid [16602]
process[third_pkg_node-4]: started with pid [16609]
[ INFO] [1566019826.516333738]: test_pkg version:1.0.0
[ INFO] [1566019826.516417989]: hello world 100
[ INFO] [1566019826.517698712]: test_pkg version:1.0.0,init count:100
[ INFO] [1566019826.517786516]: my_pkg:150
```

图 55：包含其他 launch 文件运行结果

## 2.8 ROS-Topic

### 2.8.1 Ros Topic 概述

在 ROS 中，**话题通信（Topic Communication）** 是一种用于异步传输数据的机制，它允许不同节点之间通过**发布（Publish）** 和 **订阅（Subscribe）** 进行通信。

#### ROS 话题通信的关键概念

##### 1. 话题（Topic）：

话题是数据传输的**通道或管道**。不同节点可以通过一个特定的**topic**进行通信，数据在该 topic 下传递。

每个话题都有一个**唯一的名字**，且话题基于**消息类型（Message Type）**，发布和订阅该话题的节点必须使用**相同的消息类型**。

##### 2. 发布（Publish）：

一个节点通过**发布器（Publisher）** 向话题发送消息。

发布节点（Publisher node）负责将数据**发送**到某个话题上。

##### 3. 订阅（Subscribe）：

另一个节点通过**订阅器（Subscriber）** 从该话题接收消息。

订阅节点（Subscriber node）会订阅某个话题并**接收**由发布节点发送的消息。

##### 4. 异步通信：

话题通信是**异步的**，发布节点和订阅节点不需要**同时存在**。订阅节点在连接后会**自动接收**最近发布的数据。

##### 5. 多对多关系：

一个话题可以有多个发布节点和多个订阅节点。多个节点可以向同一个话题发布消息，同时，多个节点也可以订阅该话题并接收消息。

##### 6. 消息类型（Message Type）：

每个话题的消息都有固定的类型。例如：

- std\_msgs/String (字符串消息)
- sensor\_msgs/Image (图像消息)

当然，我们也可以**自定义消息类型包**。

### 2.8.2 工作流程

**发布节点** 创建一个话题并向该话题**发布消息**。

**订阅节点** 订阅该话题并**接收来自发布节点的消息**。

ROS 的**主节点 (Master)** 负责管理节点之间的通信，如下图所示：

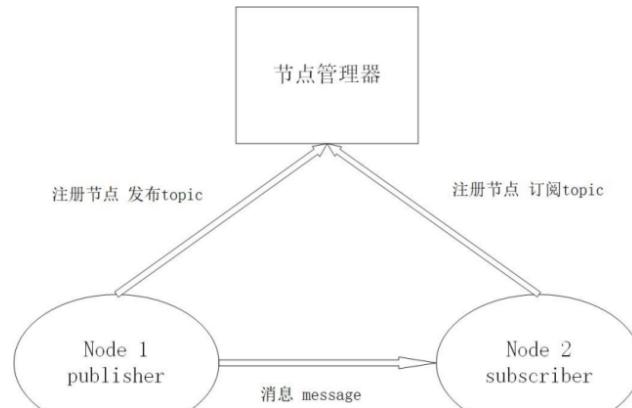


图 56: 工作流程

### 2.8.3 小乌龟实例

在 ROS 中，我们可以通过[小乌龟实验](#)来帮助理解话题通信。运行以下命令：

```
rosrun turtlesim turtlesim_node  
rosrun turtlesim turtle_teleop_key
```

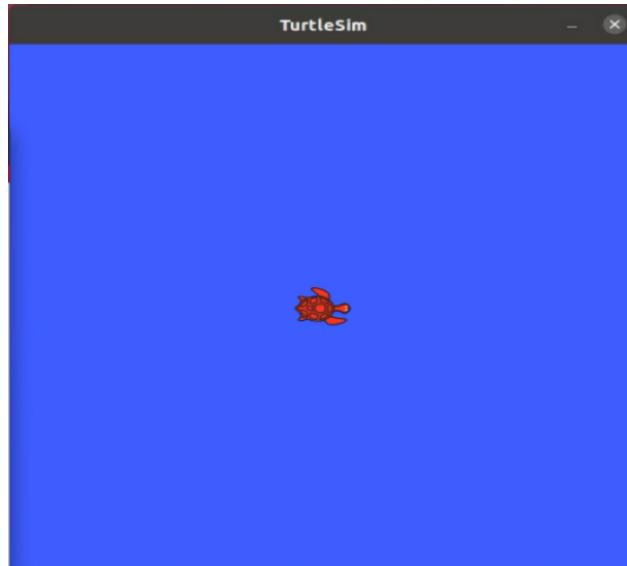


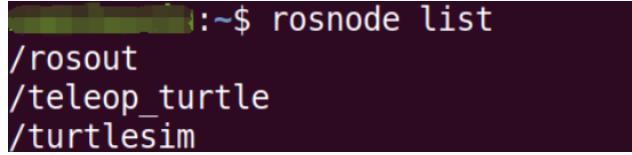
图 57: 小乌龟实例

然后，通过以下命令查看所有启动的节点：

---

```
rosnode list
```

---



```
/rosout
/teleop_turtle
/turtlesim
```

图 58: 查看启动节点

可以看到`/teleop_turtle` 通过发布`/turtle1/cmd_vel` 话题来控制`/turtlesim`。

## 2.8.4 自定义实例

### 定义发布者

---

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String

if __name__ == '__main__':
    rospy.init_node("talker")
    pub = rospy.Publisher("topic1", String, queue_size=10)

    msg = String()
    rate = rospy.Rate(1)

    count = 0
    while not rospy.is_shutdown():
        count += 1
        msg.data = "hello" + str(count)
        pub.publish(msg)
        rospy.loginfo("发布的数据是: %s", msg.data)
        rate.sleep()
```

---

### 定义订阅者

---

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String

def doMsg(msg):
    rospy.loginfo("我订阅的数据: %s", msg.data)

if __name__ == '__main__':
    rospy.init_node("listener")
    sub = rospy.Subscriber("topic1", String, doMsg, queue_size=10)
    rospy.spin()
```

---

```
[INFO] [1606383808.659671]: 发布的数据:hello2
[INFO] [1606383809.659525]: 发布的数据:hello3
[INFO] [1606383810.659677]: 发布的数据:hello4
[INFO] [1606383811.659872]: 发布的数据:hello5
[INFO] [1606383812.658887]: 发布的数据:hello6
[INFO] [1606383813.659863]: 发布的数据:hello7
[INFO] [1606383814.659873]: 发布的数据:hello8
[INFO] [1606383815.659979]: 发布的数据:hello9
[INFO] [1606383816.659627]: 发布的数据:hello10
[INFO] [1606383817.659813]: 发布的数据:hello11
[INFO] [1606383818.659676]: 发布的数据:hello12
[INFO] [1606383819.659087]: 发布的数据:hello13
```

图 59: 发布方结果

```
[INFO] [1606383816.660219]: 我订阅的数据:hello10
[INFO] [1606383817.660287]: 我订阅的数据:hello11
[INFO] [1606383818.660282]: 我订阅的数据:hello12
[INFO] [1606383819.659666]: 我订阅的数据:hello13
[INFO] [1606383820.660391]: 我订阅的数据:hello14
[INFO] [1606383821.659935]: 我订阅的数据:hello15
[INFO] [1606383822.660942]: 我订阅的数据:hello16
[INFO] [1606383823.660519]: 我订阅的数据:hello17
```

图 60: 订阅方结果

这两个节点通过话题名 `topic1` 进行通信，可通过 `rqt_graph` 查看。

### 2.8.5 常用 ROS Topic 指令

#### ROS Topic 命令参考

- `rostopic list` : 查看所有话题。
- `rostopic info <topic_name>` : 查看特定话题的信息。
- `rostopic echo <topic_name>` : 实时显示消息内容。
- `rostopic pub <topic_name> <message_type> <options>` : 发布消息。
- `rostopic type <topic_name>` : 查看消息类型。

## 2.9 ROS-service、launch、bash

### 2.9.1 ROS-service

#### 1. 概念：

服务也是 ROS 通信方式的一种，是通过两个节点来发送，接受数据，消息等。去学习服务，只需要与话题比较下即可，他们的区别是：话题只能一味的发送或接受消息是“一次性的”，而服务包含客户端和服务端，是能够进行消息的接受和发送的。

下面是官方的两张图：

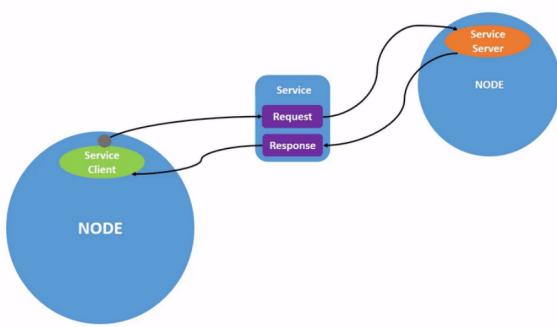


图 61: 效果图

客户端：客户端节点向服务端发送请求，并等待服务端的响应。客户端可以是任何发起请求的节点。

服务端：服务端节点接收请求并处理，然后返回响应。服务端需要在启动时提供一个特定的服务类型。

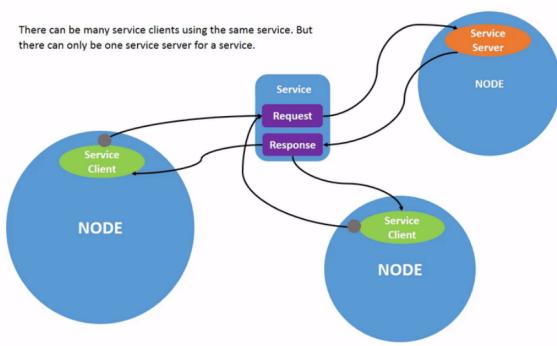


图 62: 效果图

## 2. 实现：

服务是通过.srv 文件来定义的，每个.srv 文件由两个部分组成：

请求部分 (Request)

响应部分 (Response)

一个服务文件的格式如下：

```
# ServiceRequest.srv
```

```
int32 a
```

```
int32 b
```

---

```
int32 sum
```

上面的例子定义了一个计算两个整数和的服务。请求部分包含两个整数 a 和 b，响应部分返回一个整数 sum，即  $a + b$ 。

—上是客户端发出消息的数据格式

—下是服务端接受消息的数据格式

## 2.9.2 ROS\_launch

### 1. 概念：

`roslaunch` 是 ROS (Robot Operating System) 中用于启动多个节点、加载配置文件和设定参数的工具。它可以自动化启动一组节点，使得复杂系统的启动变得简便。`roslaunch` 通过 `.launch` 文件来定义节点、参数、环境变量等。（一般为 XML 格式；在 ROS2 中也可用 Python 编写，功能更丰富。）在功能包中，一般会在 `launch/` 文件夹下创建如下图所示的结构：



图 63: 文件结构

### 编写规律：

在 ROS 中，编写 `.launch` 文件时，可以按照如下基本结构：

Listing 10: 编写规律

---

```
<launch>
  <node pkg="package_name" type="node_name" name="node_name" output="screen">
    <param name="param_name" value="param_value"/>
  </node>
  <!-- 可以在这里定义多个节点 -->
</launch>
```

---

### 启动一个节点，并管理参数：

下面示例展示了如何在同一个 `.launch` 文件中启动节点并设置若干参数：

Listing 11: 启动节点并管理参数

---

```
<launch>
  <node pkg="turtlebot3" type="turtlebot3_node" name="turtlebot" output="screen">
    <param name="use_sim_time" value="true"/>
    <param name="robot_name" value="my_robot"/>
  </node>
</launch>
```

---

### 启动多个节点:

Listing 12: 启动多个节点

---

```
<launch>
  <node pkg="package1" type="node1" name="node1" output="screen"/>
  <node pkg="package2" type="node2" name="node2" output="screen">
    <param name="some_param" value="42"/>
  </node>
</launch>
```

---

### 启动一个 launch:

Listing 13: 启动一个 launch

---

```
<launch>
  <!-- 启动 robot.launch 文件 -->
  <include file="$(find my_robot_package)/launch/robot.launch" />

  <!-- 启动 sensors.launch 文件 -->
  <include file="$(find my_robot_package)/launch/sensors.launch" />
</launch>
```

---

### 启动多个 launch:

Listing 14: 启动多个 launch

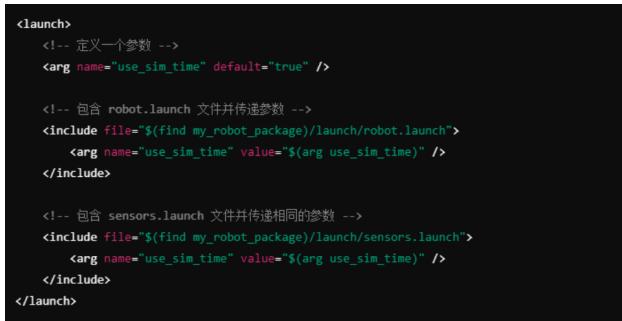
---

```
<launch>
  <include file="$(find my_robot_package_1)/launch/first_launch.launch" />
  <include file="$(find package_name2)/launch/second_launch.launch" />
  <include file="$(find package_name2)/launch/third_launch.launch" />
</launch>
```

---

### 向包含的 launch 文件传递参数

(见下图 64: 向包含的 launch 文件传递参数)



```
<launch>
  <!-- 定义一个参数 -->
  <arg name="use_sim_time" default="true" />

  <!-- 包含 robot.launch 文件并传递参数 -->
  <include file="$(find my_robot_package)/launch/robot.launch">
    <arg name="use_sim_time" value="$(arg use_sim_time)" />
  </include>

  <!-- 包含 sensors.launch 文件并传递相同的参数 -->
  <include file="$(find my_robot_package)/launch/sensors.launch">
    <arg name="use_sim_time" value="$(arg use_sim_time)" />
  </include>
</launch>
```

图 64: 向包含的 launch 文件传递参数

在实际代码中，可通过在父 .launch 文件中声明 `<arg name="..." />` 并在 `<include>` 标签中传递给子 .launch 文件。

## 使用多个参数

(见下图 65: 使用多个参数)

```
<launch>
    <!-- 定义多个参数 -->
    <arg name="robot_name" default="robot1" />
    <arg name="sensor_type" default="lidar" />

    <!-- 包含 robot.launch 并传递 robot_name 参数 -->
    <include file="$(find my_robot_package)/launch/robot.launch">
        <arg name="robot_name" value="$(arg robot_name)" />
    </include>

    <!-- 包含 sensors.launch 并传递 sensor_type 参数 -->
    <include file="$(find my_robot_package)/launch/sensors.launch">
        <arg name="sensor_type" value="$(arg sensor_type)" />
    </include>
</launch>
```

图 65: 使用多个参数

例如，在父文件中定义多个 `<arg>`，然后在 `<include>` 中逐一传递：

Listing 15: 使用多个参数 (与图 65 对应)

---

```
<launch>
    <arg name="param1" default="value1"/>
    <arg name="param2" default="value2"/>

    <include file="$(find some_pkg)/launch/child.launch">
        <arg name="param1" value="$(arg param1)"/>
        <arg name="param2" value="$(arg param2)"/>
    </include>
</launch>
```

---

最终，启动 launch

(见下图 66: 启动 launch)

```
roslaunch package_name file.launch
```

图 66: 启动 launch

在终端中使用 `roslaunch` 命令即可启动主 `.launch` 文件并通过命令行传参：

Listing 16: 命令行传参示例

---

```
roslaunch my_robot_package main.launch param1:=hello param2:=world
```

---

### 2.9.3 ROS\_bash

(见下图 67: ROS\_bash)

```

1  roscl:用于快速切换到某个 ROS 包的路径。          例:roscl turtlebot3_bringup
2
3  rosls:列出 ROS 包目录中的文件,与 roscl 配合使用。    例:rosls turtlebot3_bringup
4
5  rospack find:用于找到某个 ROS 包的路径。          例:rospack find turtlebot3_bringup
6
7  rosrun:用于直接运行某个包中的节点。          例:rosrun turtlesim turtlesim_node
8
9  rosmsg:用于查看消息类型。          例:rosmsg show geometry_msgs/Twist
10 rosrv:查看 ROS 服务类型。          例:rosrv show std_srvs/Empty
11
12 rosnode:管理和监控 ROS 节点。查看运行的节点:      例:rosnode list
13 rosnode list:查看节点信息:                         例:rosnode info
14
15 rostopic:管理和监控 ROS 话题。          例:rostopic echo
16 rostopic list:查看所有发布的话题。          例:rostopic list
17 rostopic info <topic_name>:查看某个话题的消息结构
18 rostopic echo <topic_name>:实时监听某个话题的消息:
19
20 rosparam:管理和查看 ROS 参数服务器上的参数。
21 rosparam list:查看所有参数
22 rosparam get <param_name>:获取某个参数的值
23 rosparam set <param_name> <value>:设置某个参数

```

图 67: ROS\_bash

在 ROS\_bash 相关操作中, 可通过 Shell 脚本简化 ROS 命令调用, 例如:

- source /opt/ros/noetic/setup.bash 加载 ROS 环境;
- 在脚本中执行多条 roslaunch、rosrun 等命令, 实现一键启动多个节点或参数配置。

## 3 工业组

### 3.1 SSH 远程连接

首先用自己的电脑(必须为 ubuntu 系统)搜索并连接机器人的 wifi(wifi 名字为 oryxbot 开头字样), 比如为: oryxbot-raicom-01, 机器人 IP 地址: 电池外置的机器人 IP 地址为: 192.168.8.1, 电池未外置的机器人 IP 地址为: 192.168.8.100。双系统正常连接 WiFi 网络即可。虚拟机需要做些更改:

- 把网络适配器修改为桥接模式;
- 每次更换 windows 网络时, 需开关一下 Ubuntu 的有线连接, 刷新 IP。

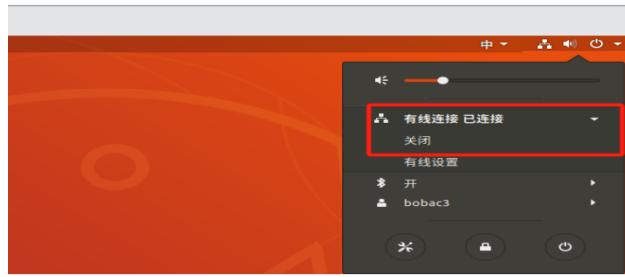


图 68: 有线连接

完成后，我们通过 ifconfig 查看 IP 地址，保证是 192.168.8.xxx 即可。打开终端，使用 ssh 连接机器人，输入以下指令

Listing 17: 查看 ip 地址

```
#电脑端
$ ssh oryxbot@192.168.8.1
```

```
bobac3@reinovo:~$ ifconfig
ens33: flags=1103<UP,BROADCAST,STP,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.8.168  netmask 255.255.255.0  broadcast 192.168.8.255
                inet6 fe80::192:168.8.168%ens33  prefixlen 64  scoped_id 0x20<link>
                    ether 00:0c:29:eb:02:91  txqueuelen 1000  (以太网)
                    RX packets 626222  bytes 292802239 (292.8 MB)
                    RX errors 0  dropped 4  overruns 0  frame 0
                    TX packets 327629  bytes 80202979 (80.2 MB)
                    TX errors 0  dropped 0  overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
                inet6 ::1  prefixlen 128  scoped_id 0x10<host>
                    loop  txqueuelen 1000  (本地环回)
                    RX packets 281858780  bytes 618322349205 (618.3 GB)
                    RX errors 0  dropped 0  overruns 0  frame 0
                    TX packets 281858780  bytes 618322349205 (618.3 GB)
                    TX errors 0  dropped 0  overruns 0  carrier 0  collisions 0

bobac3@reinovo:~$
```

图 69: 查看 IP 地址

连接成功后，终端的用户名就变为 oryxbot，这个终端就成为机器人终端了。

```
-bash: /home/oryxbot/new_workspace-devel/setup.bash: No such file or directory
oryxbot@reinovo:~$
```

图 70: 机器人终端

机器人搜索网络，输入以下指令：

Listing 18: 搜索网络

```
#机器人端
$ nmcli dev wifi list
```

IN-USE	BSSID	SSID	MODE	CHAN	RATE
	4C:C6:4C:C9:30:94	reinovo-5G_plus	Infra	4	270 Mb/s
	5C:B0:0A:4A:F7:F8	JHS	Infra	11	130 Mb/s
	84:DF:9E:1F:6F:50	yunzhikeyuan	Infra	11	130 Mb/s
	3B:2A:57:AE:35:D4	reinovo	Infra	6	130 Mb/s
	18:2A:57:AE:35:D4	reinovo	Infra	1	130 Mb/s
	32:24:A9:85:6F:07	DIRECT-07-HP OfficeJet Pro 7730	Infra	36	65 Mb/s
	18:2A:57:AE:35:D8	reinovo-5G	Infra	36	270 Mb/s
	38:87:D5:63:2E:07	oryxbot-04	Infra	11	54 Mb/s
	18:2A:57:A1:2F:50	reinovo-5G	Infra	36	270 Mb/s
	76:C6:D0:2D:F0:30	reinovo_test	Infra	1	130 Mb/s
	82:C6:D0:2D:F0:30	reinovo_test_GUEST	Infra	1	130 Mb/s
	04:83:C4:1D:F9:9F	Msnakes-wrt	Infra	1	130 Mb/s
	22:C9:F2:E9:86:A2	Redmi 9A	Infra	13	117 Mb/s
	8C:DE:F9:85:BF:A1	Reinovo_大培训室	Infra	11	270 Mb/s
	92:DE:F9:85:BF:A1	--	Infra	11	270 Mb/s
	36:ED:45:C2:E8:B7	光	Infra	11	130 Mb/s

图 71: 网络搜索

然后我们选择所需要来连接的 WiFi 名 (即 SSID)，输入以下指令连接：

Listing 19: 连接 wifi

```
#机器人端
$ sudo nmcli dev wifi connect "SSID" password "PASSWORD" ifname wlp4s0
```

SSID：为要连接的无线网络名称；PASSWORD：无线网络的密码；wlp4s0：机器人的无线网卡名，在机器人终端用 ifconfig 命令查看。

TX packets 309 bytes 46183 (45.1 KB)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
txp2s0: [ 1-0s-100-300-0p ] BROADCAST RUNNING MULTICAST <eno1> mtu 1500
inet6 192.168.1.194 brd 192.168.1.255 scopeid 0x0<global>
inet6 240e:3b2:3cf0:dd:5e27:bff:3:347:417b brd 240e:3b2:3cf0:dd:5e27:bff:3:347:417b/64 scopeid 0x0<link>
inet6 240e:3b2:3cf0:db:87c:cef1:954:93a1 brd 240e:3b2:3cf0:db:87c:cef1:954:93a1/64 scopeid 0x0<link>
ether fc:b3:bc:22:96:42 brd 00:0c:29:ff:ff:ff
TX packets 3224 bytes 46435 (46.4 KB)
TX errors 0 dropped 0 overruns 0 frame 0
TX packets 3224 bytes 46435 (46.4 KB)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

图 72: 无线网卡名称

连接成功会出现如图所示内容：

```
oryxbot@reinovo:~$ sudo nmcli dev wifi connect "reinovo-5G" password "reinovo888" ifname wlp2s0
[sudo] password for oryxbot:
Device 'wlp2s0' successfully activated with '1cc6ae6f-4bec-4159-91d6-fdc10743d6c'.
```

图 73: 连接成功

ros 是一种分布式软件框架，节点之间通过松耦合的方式进行组合。通过以下四步可以实现分布式多机通信。首先两台/多台机器连接到同一个无线网络下，假设你当前使用的 oryxbot 的 IP 地址为 192.168.8.1。

- 使用你的电脑连接机器人的 wifi;
- 在你的电脑端通过 `ifconfig` 命令查看你电脑的 IP 地址，假设为 192.168.8.1;
- 测试网络是否联通，如下图所示在机器人端使用 `ping` 命令；
- 在 `/.bashrc` 文件添加 IP 地址和设备名称。

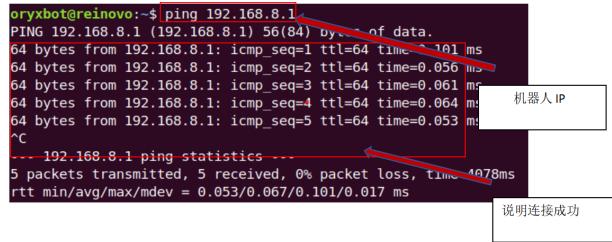


图 74: 网络测试

电脑终端：输入以下命令打开 `bashrc` 文件

Listing 20: 打开 `bashrc` 文件

```
#电脑终端
bobac3@reinovo:~$ gedit .bashrc
bashrc文件:
电脑端 .bashrc文件:
source /opt/ros/kinetic/setup.bash
source ~/oryxbot\_ws/devel/setup.bash
#上面不重要！！！直接拉到最下面
export ROS_MASTER_URI=http://192.168.8.xxx:11311      #添加这一句，机器人ip
export ROS_IP=192.168.8.xxx    #电脑ip
电脑ip地址查看
#电脑终端
bobac3@reinovo:~$ ifconfig
```

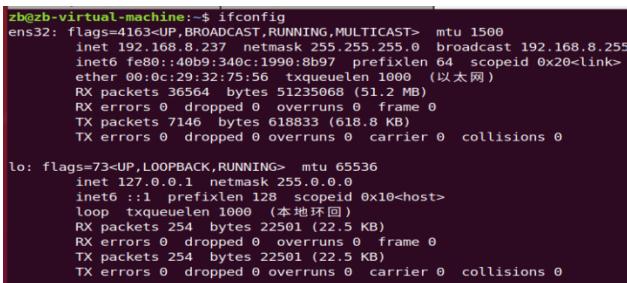


图 75: 电脑 IP 查看

Oryxbot 机器人终端：

---

Listing 21: Oryxbot 机器人终端

---

```
#机器人终端
oryxbot@reinovo:$ sudo vim ~/.bashrc
Oryxbot 机器人端 bashrc 文件：
机器人端 /.bashrc 文件：
source /opt/ros/noetic/setup.bash
source /oryxbot_ws/devel/setup.bash
# 添加机器人 ip 地址
export ROS_IP=192.168.8.xxx #机器人 ip， 这里默认已经添加好了.
```

---

## 3.2 机械臂的相机标定

摄像头这种精密仪器对于光学器件的要求较高，由于摄像头内部或者外部的一些原因，生成的图像往往会发生畸变，为了避免数据源造成的误差，需要对摄像头的参数进行标定。ROS 官方提供了用于单目双目摄像头标定的功能包-camera\_calibration，我们机器人出厂时都对摄像头进行了内外参标定，只要不调整摄像头焦距等就不用再次进行标定。如果需要进行内参标定，为了避免网络延迟带来的影响一般摄像头插在哪里就在哪里进行标定，可供选择的方式有给机器人装上屏幕键鼠进行标定，或者将摄像头插在自己电脑上进行标定然后将标定好的内参文件通过 U 盘或者 scp 等方式拷贝到机器人端。

### 3.2.1 camera\_calibration 功能包

首先在插有待标定摄像头的电脑（本章命令均在插有待标定摄像头电脑里运行）使用以下命令安装摄像头标定功能包

---

Listing 22: 安装功能包

---

```
$ sudo apt-get install ros-melodic-camera-calibration
```

---

标定需要使用到棋盘格图案的标定靶，可以在光盘或者网上找到。请你将该标定靶打印出来贴到平面硬纸板上备用。

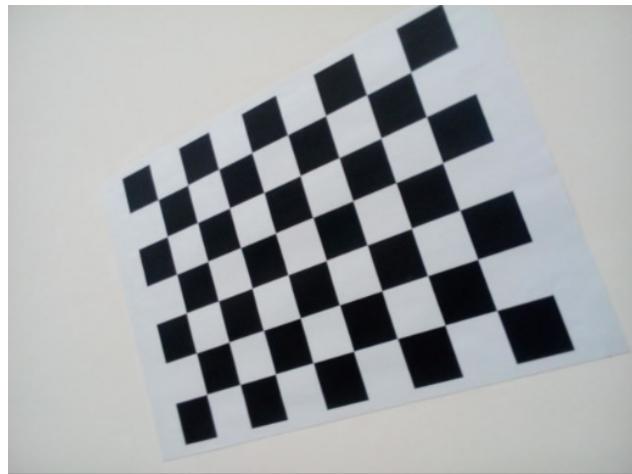


图 76: 标定靶

一切就绪后准备开始标定摄像头。首先使用以下命令启动 usb 摄像头

Listing 23: 启动摄像头

```
$ roslaunch usb_cam usb_cam-test.launch
```

然后使用以下命令启动标定程序：

Listing 24: 启动标定程序

```
$ rosrun camera_calibration cameracalibrator.py --size 8x6 --square 0.025 image:=/usb_cam/image_raw  
→ camera:=/usb_cam
```

### 3.2.2 相机标定方法

Cameracalibrator.py 标定程序需要以下几个输入参数。

- **Size**: 标定靶的内部角点个数，这里使用的棋盘格一共六行，每行有八个内部角点。
- **Square**: 这个参数对应每个棋盘格的边长，单位是米。
- **Image** 和 **camera**: 设置摄像头发布的图像话题。
- 在 **/.bashrc** 文件添加 IP 地址和设备名称。

根据使用的摄像头和标定靶尺寸相应修改以上参数，即可启动标定程序。标定程序启动成功以后，将标定靶放在摄像头视野内，应该可以看到以下图形界面。在没有标定成功前，右边的按钮都为灰色，

不能点击。为了提高标定的准确性，应该尽量让标定靶出现在摄像头视野范围内的各个区域，界面右上角的进度条会提示标定进度，

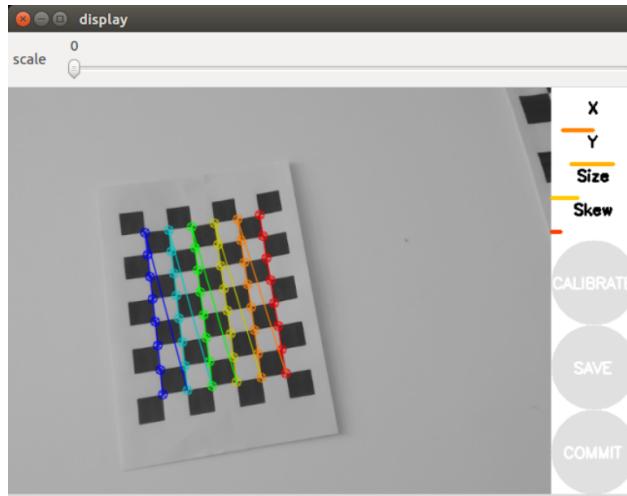


图 77: 标定过程

- **X:** 标定靶在摄像头视野中的左右移动。
- **Y:** 标定靶在摄像头视野中的上下移动。
- **Size:** 标定靶在摄像头视野中的前后移动。
- **Skew:** 标定靶在摄像头视野中的倾斜转动。

不断在视野中移动标定配，直到“CALIBRATE”按钮变色，表示标定程序的参数采集完成。点击“CALIBRATE”按钮，标定程序开始自动计算摄像头的标定参数，这个过程满要等待段时间，界面可能会变成灰色无响应状态，注意无响应最长可达十到二十分钟，期间不要关闭。参数计算完成后界面回复，而且终端也会有标定结果的显示。

```
[image]
width
640
height
480
[narrow_stereo]
camera matrix
780.287751 0.000000 328.343394
0.000000 799.870107 210.421971
0.000000 0.000000 1.000000

distortion
-0.382160 0.210415 0.001052 -0.001239 0.000000

rectification
1.000000 0.000000 0.000000
0.000000 1.000000 0.000000
0.000000 0.000000 1.000000

projection
727.420105 0.000000 328.370755 0.000000
0.000000 769.150330 208.109058 0.000000
0.000000 0.000000 1.000000 0.000000
```

图 78: 标定结果

点击 save 按钮，终端会打印生成文件及路径

Listing 25: 文件及路径

---

```
'Wrote calibration data to', '/tmp/calibrationdata.tar.gz'
```

---

我们在终端中可以看到路径是在计算机目录/tmp 下。点击 commit 按钮，提交数据并退出程序，在计算机目录打开/tmp 文件夹，可以看到 calibrationdata.tar.gz，解压文件后从中找到 ost.yaml 文件，将文件放到 “ /fox\_ws/src/ar\_pose\_adjust/cam\_info” 文件夹下，如果是车体摄像头则改名为 base\_camera.yaml，如果是手臂摄像头则更名为 hand\_camera.yaml，并且将文件里的 camera\_name 参数修改为 head\_camera。

### 3.3 自主充电

自主充电：记录充电桩的地图位置坐标，在任务导航栏中加入该坐标，在该坐标点首先对准二维码，并且前进 20cm，之后设置停留两秒，即可完成充电。

### 3.4 综合调度

综合调度：与省赛综合调度调试方法一样，启动小车后直接“start”即可按照设定的程序运行。

## 4 服务组

### 4.1 导航

#### 4.1.1 SLAM 建图

SLAM: Simultaneous Localization And Mapping(同步定位与地图构建); Localization(定位) 指机器人对自身的定位, 即机器人在地图上的位置; Mapping(地图构建) 指机器人对探测到的地图构建栅格地图。定位与构图是同时进行的。

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

ROS 中的 SLAM 工具包:

Hector\_Mapping

- 安装依赖: rosdep, nav\_msgs, visualization\_msgs, tf, message\_filters, laser\_geometry, message\_generation, catkin, message\_runtime

命令行安装:

```
sudo apt install ros-noetic-hector-mapping
```

- 节点启动:

```
rosrun hector_mapping hector_mapping
```

- Hector\_Mapping 建图: 仿真地图; 启动 Hector\_Mapping; rviz 可视化; 控制机器人移动, 示例代码

Listing 26: 示例代码

---

```
<launch>
  <include file = "$(find wpr_simulation)/launch/wpb_stage_slam.launch"/> # 加载仿真地图
  <node pkg = "hector_mapping" type = "hector_mapping" name ="hector_mapping"/> # 启动
    ↳ Hector_Mapping
  <node pkg = "rviz" type = "rviz" name = "rviz" args = "-d
/home/robotsheep/navigation_ws/src/slam_pkg/rviz/slam.rviz"/> # rviz可视化
  <node pkg = "wpr_simulation" type = "keyboard_vel_ctrl" name ="keyboard_vel_ctrl" output =
    ↳ screen"/> # 控制机器人移动
</launch>
```

---

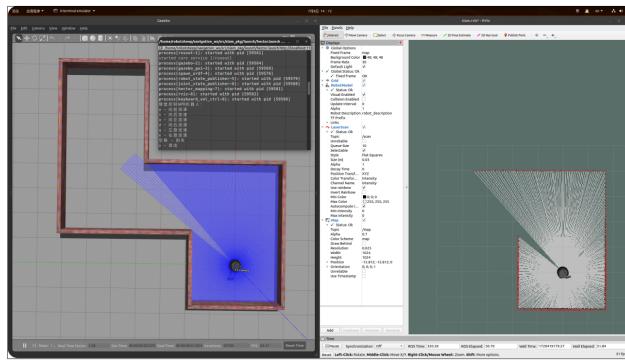


图 79: 效果图

参考文档:

- **Hector\_Mapping pkg:** [https://index.ros.org/p/hector\\_mapping/github-tu-darmstadt -ros-pkg-hecto](https://index.ros.org/p/hector_mapping/github-tu-darmstadt -ros-pkg-hecto)
- **Hector\_Mapping wiki:** [https://wiki.ros.org/hector\\_mapping](https://wiki.ros.org/hector_mapping)

### Gmapping

- 安装依赖: `nav_msgs, openslam_gmapping, roscpp, rostest, tf, nodelet, catkin`  
命令行安装:  
`sudo apt install ros-noetic-hector-mapping`
- 节点启动:  
`rosrun gmapping slam_gmapping`
- **Gmapping 建图:** 仿真地图; 启动 Gmapping; rviz 可视化; 控制机器人移动.  
示例代码

Listing 27: 示例代码

---

```

<launch>
  <include file = "$(find wpr_simulation)/launch/wpb_stage_corridor.launch"/> # 加载仿真地图
  <node pkg = "gmapping" type = "slam_gmapping" name ="slam_gmapping"/>
  <node pkg = "rviz" type = "rviz" name = "rviz" args = "-d
  /home/robotsheep/navigation_ws/src/slam_pkg/rviz/slam.rviz"/> # rviz可视化
  <node pkg = "wpr_simulation" type = "keyboard\_\_vel\_\_ctrl" name= "keyboard_vel_ctrl" output =
    ↳ "screen"/> # 控制机器人移动
</launch>

```

---

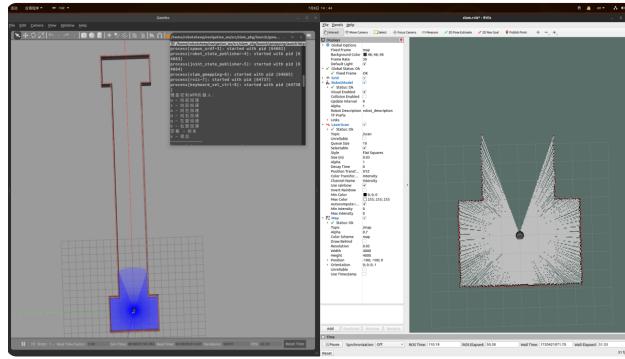


图 80: 效果图

参考文档:

- Gmapping pkg: [https://index.ros.org/p/gmapping/github-ros-perception-slam\\_gmapping/#noetic](https://index.ros.org/p/gmapping/github-ros-perception-slam_gmapping/#noetic)
- Gmapping wiki: <https://wiki.ros.org/gmapping>

Hector\_Mapping 和 Gmapping 的使用: Hector\_Mapping 建图原理是模型匹配, 当机器人检测一段时间内模型一致, 模型无法匹配, 机器人无法完成 SLAM, 适合不具备大量相似场景的地图.

Gmapping 建图是模型匹配 + 里程计 odom 修正, 弥补了 Hector\_Mapping 无法在模型一致的场景建图, 适用于大部分情况, 默认使用 Gmapping 即可.

保存地图

- 使用 Hector\_Mapping 或者 Gmapping 建图。
- 新开一个终端。
- 使用 map\_server 的 map\_saver 节点:

```
rosrun map_server map_saver [--occ <threshold_occupied>] [--free<threshold_free>]
[-f <mapname>] map:=/your/costmap/topic
```

示例

Listing 28: 示例

---

```
rosrun map_server map_saver -f map # 保存在当前目录下, 名字为map.
rosrun map_server map_saver -f map map:=/home/robotsheep/maps # 保存在/home/robotsheep/maps下, 名
    ↪ 字为map.
```

之后会在目录下生成俩个文件 .pgm 和 .yaml

---

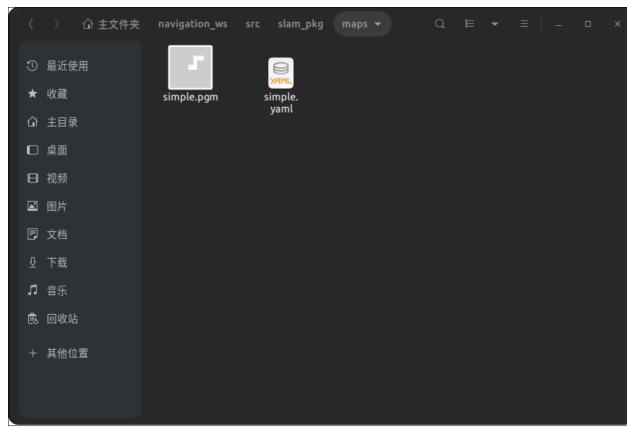


图 81: 文件示例

yaml 文件内容如下:

A screenshot of a code editor window titled 'simple.yaml' located at '~/navigation\_ws/src/slam\_pkg/maps'. The code editor displays the following YAML configuration:

```
1 image: simple.pgm
2 resolution: 0.025000
3 origin: [-12.812499, -12.812499, 0.000000]
4 negate: 0
5 occupied_thresh: 0.65
6 free_thresh: 0.196
7
```

The first line 'image: simple.pgm' is highlighted with a yellow background.

图 82: yaml 文件

参数:

- **image**: 包含占用数据的图像文件的路径; 可以是绝对路径, 也可以是相对于 YAML 文件位置的路径。
- **resolution**: 地图分辨率, 米/像素。
- **origin**: 地图左下角像素的二维姿态, 为 (x, y, yaw), yaw 为逆时针旋转 (yaw=0 表示不旋转)。系统的许多部分目前都忽略了 yaw。**occupied\_thresh**: 占用概率大于此阈值的像素被视为完全占用。**free\_thresh**: 占用概率小于此阈值的像素被视为完全空闲。
- **negate**: 白色/黑色的空闲/占用语义是否应该反转 (阈值的解释不受影响)。

参考文档:

- map\_server pkg: [https://index.ros.org/p/map\\_server/github-ros-planning-navigation/#noetic](https://index.ros.org/p/map_server/github-ros-planning-navigation/#noetic)
- map\_server wiki: [http://wiki.ros.org/map\\_server](http://wiki.ros.org/map_server)

#### 4.1.2 机器人定位

AMCL: Adaptive Monte Carlo Localization(自适应蒙塔卡洛定位算法)。

是一种使用粒子滤波在已知地图中进行定位的算法，同时使用了里程计和激光雷达数据，具有较强的自我纠错功能。AMCL 提供在已知地图（即 SLAM 建的地图）上机器人的位置。

AMCL 原理：粒子滤波很粗浅的说就是一开始在地图空间很均匀的撒一把粒子，然后通过获取机器人的 motion 来移动粒子，比如机器人向前移动了一米，所有的粒子也就向前移动一米，不管现在这个粒子的位置对不对。使用每个粒子所处位置模拟一个传感器信息跟观察到的传感器信息（一般是激光）作对比，从而赋给每个粒子一个概率。之后根据生成的概率来重新生成粒子，概率越高的生成的概率越大。这样的迭代之后，所有的粒子会慢慢地收敛到一起，机器人的确切位置也就被推算出来了。

ROS 的 AMCL 工具包

- 依赖: message\_filters, tf2\_geometry\_msgs, catkin, map\_server, rostest, tf2\_py, diagnostic\_updater, dynamic\_reconfigure, geometry\_msgs, nav\_msgs, rosbag, roscpp, sensor\_msgs, std\_srvs, tf2, tf2\_msgs, tf2\_ros.
- 命令行安装：  
`sudo apt install ros-noetic-amcl`
- 节点启动：  
`rosrun amcl amcl`

参考文档：

- 1. AMCL pkg: <https://index.ros.org/p/amcl/github-ros-planning-navigation/#noetic-deps>
- 2. AMCL wiki: <http://wiki.ros.org/amcl>

#### 4.1.3 代价地图

将地图栅格化，对于每一个栅格的状态要么占用，要么空闲，要么未知（即初始化状态）。

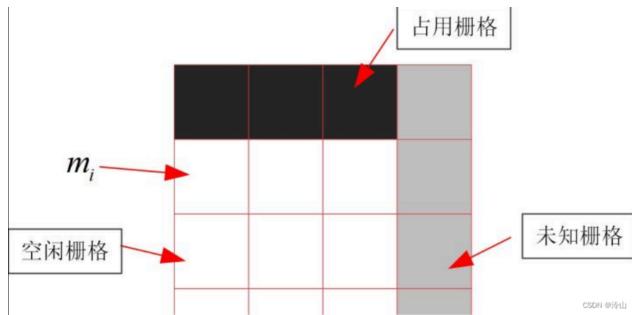


图 83: 地图格栅化

在 ros 中，地图的数据按照行优先的顺序，从栅格矩阵的 (0,0) 位置开始排列成一个数组示例图：

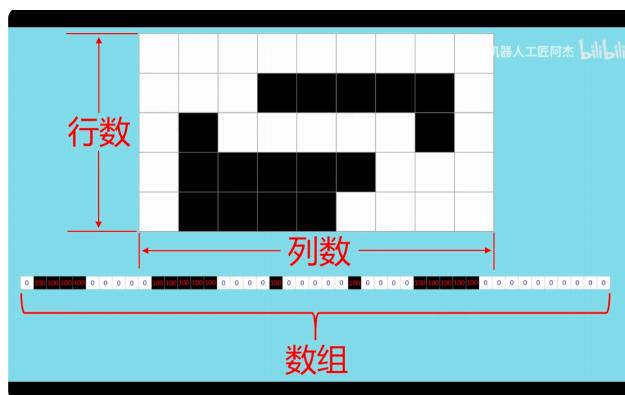


图 84: 示意图

栅格里障碍物占据值的取值范围是从 0 到 100，如果栅格里的障碍物状况未知，则栅格数值为-1。  
示例：

Listing 29: 示例

```
#!/usr/bin/env python3
# coding=utf-8
import rospy
from nav_msgs.msg import OccupancyGrid
if __name__ == "__main__":
    rospy.init_node("map_pub_node")
    pub = rospy.Publisher("/map", OccupancyGrid, queue_size=10)
    rate = rospy.Rate(30)
    while not rospy.is_shutdown():
        msg = OccupancyGrid()
        msg.header.frame_id = "map"
        msg.header.stamp = rospy.Time.now()
```

```

msg.info.origin.position.x = 0
msg.info.origin.position.y = 0
msg.info.resolution = 1.0
msg.info.width = 4
msg.info.height = 2
msg.data = [0] * 4 * 2 # 创建 2 x 4 的二维数组
msg.data[0] = 100
msg.data[1] = 100
msg.data[2] = 0
msg.data[3] = -1
msg.data[4] = 0
msg.data[5] = 0
msg.data[6] = 0
msg.data[7] = 0
pub.publish(msg)
rate.sleep()

```

效果图：

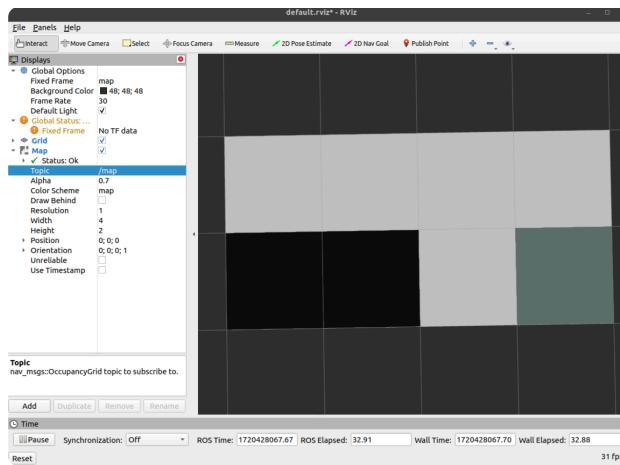


图 85: 效果图

2D 格栅矩阵

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ -100 & -100 & 0 & -1 \end{bmatrix}$$

图 86: 格栅矩阵

栅格地图-> 代价地图，代价地图预览：

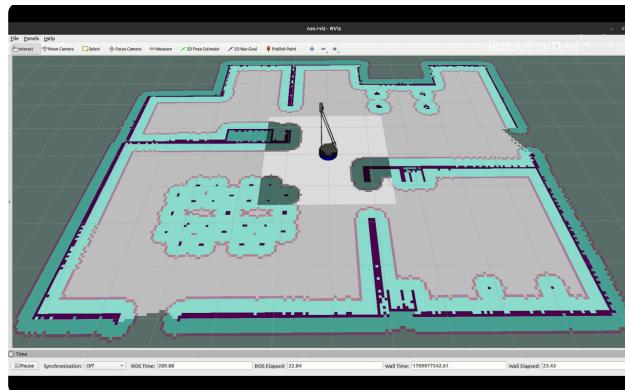


图 87: 代价地图预览

机器人在移动时需要与障碍物之间保持一定的安全缓冲距离。因此在栅格地图外加两层膨胀，就像两层排斥力场，越靠近中心，排斥力越大，越不允许机器人到达。

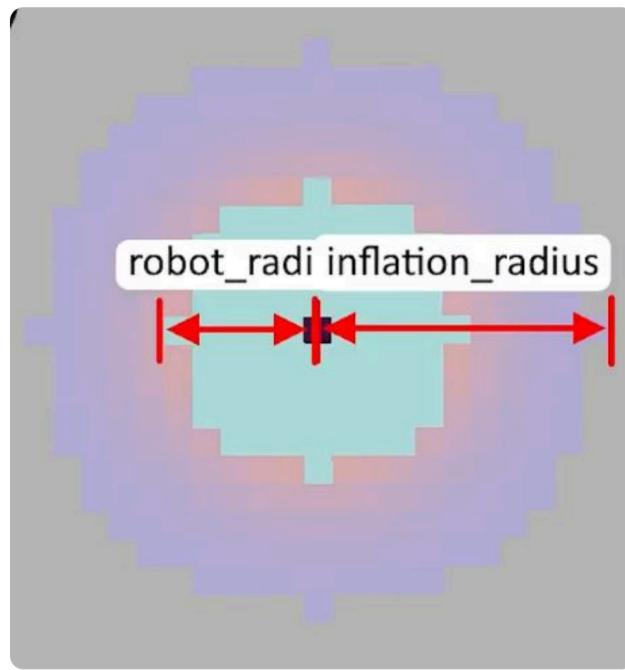


图 88: 膨胀示意

栅格地图加上两层膨胀便组成了代价地图，即越靠近中心代价越大，机器人越容易撞上。

### 代价地图的图层分布

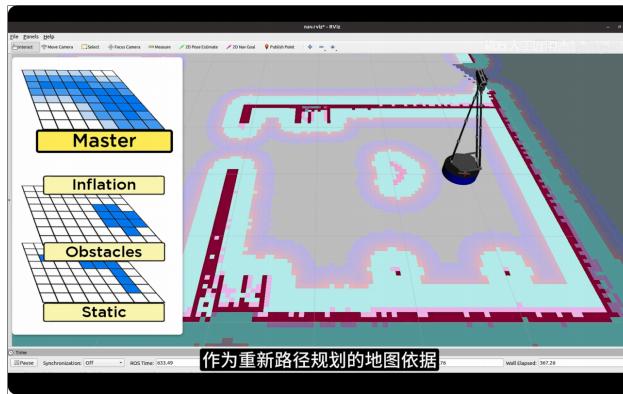


图 89: 图层分布

- **Static Map Layer:** 静态地图层，基本上不变的地图层，通常都是 SLAM 建立完成的静态地图。
- **Obstacle Map Layer:** 障碍地图层，用于动态的记录传感器感知到的障碍物信息。
- **Inflation Layer:** 膨胀层，在以上两层地图上进行膨胀（向外扩张），以避免机器人的撞上障碍物。
- **Other Layers :** 你还可以通过插件的形式自己实现 costmap，目前已有 SocialCostmap Layer、Range Sensor Layer 等开源插件。

### 代价地图分类

- 全局代价地图 (`global_costmap`)：全局代价地图是 SLAM 建图后加上膨胀后的静态地图，仅反映在 SLAM 建图时的信息，作为 `GlobalPlanner` 的参考，详见代价地图预览全局。
- 局部代价地图 (`local_costmap`)：局部代价地图是机器人实施观测到的一个区域内的代价地图，反映实际导航中地图的信息，作为 `LocalPlanner` 的参考，详见代价地图预览中以机器人为中心的区域。

参考文档：

- 棚格地图: [https://blog.csdn.net/qq\\_28087491/article/details/125441314](https://blog.csdn.net/qq_28087491/article/details/125441314)
- 代价地图: [https://blog.csdn.net/qq\\_32115101/article/details/81121220](https://blog.csdn.net/qq_32115101/article/details/81121220)

#### 4.1.4 路径规划

##### 4.1.4.1 ROS 的 move\_base 工具包

简介：move\_base 包允许您使用 navigation stack 将机器人移动到所需位置。move\_base 节点提供了一个 ROS 接口，用于配置、运行和与机器人上的 navigation stack 交互。

安装

- 安装依赖: `cmake_modules,message_generation,tf2_geometry_msgs,catkin,message_runtime,actionlib, costmap_2d,dynamic_reconfigure,geometry_msgs,move_base_msgs,nav_core,nav_msgs,pluginlib,roscpp, rospy,std_srvs,tf2_ros,visualization_msgs,base_local_planner,clear_costmap_recovery,navfn,rotate_recovery`
- 命令行安装: `sudo apt install ros-noetic-move-base`

流程图

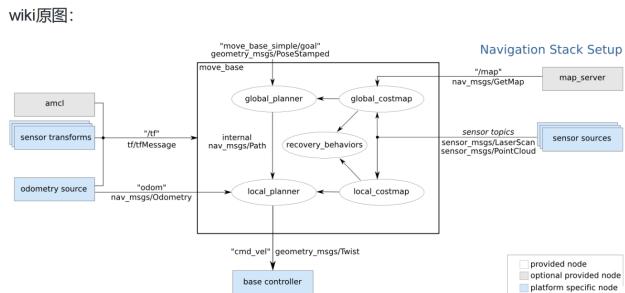


图 90: 流程图

蓝色节点因机器人平台而异，灰色节点是可选的，但为所有系统提供，白色节点是必需的，但也为所有系统提供。

双语流程图

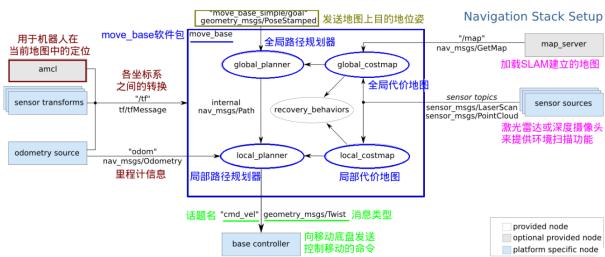


图 91: 双流程图

正常情况：

- 向机器人发送地图上的目的地位姿。
  - 全局规划器从 `global_costmap` 规划出一条全局导航路线作为参考。
  - 在机器人实际导航中, `local_planner` 再根据 `local_costmap` 对 `global_costmap` 给的路径进行修正。
  - 实际运动中, 机器人沿着 `local_planner` 修正过后的路线进行移动。

当 local\_planner 规划不出路线时：机器人会进入 recovery\_behaviors，直至规划出路线或者规划不出路线，放弃导航。

机器人的恢复行为 (recovery\_behaviors)

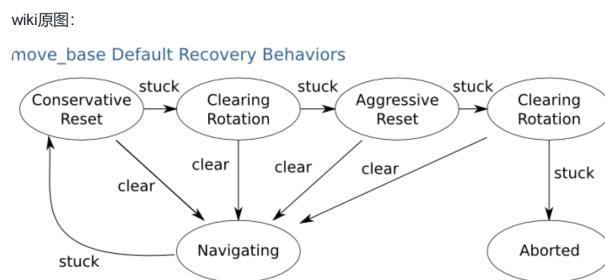


图 92: 机器人恢复行为 wiki 原图

中文图

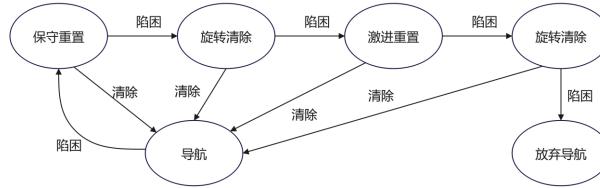


图 93: 机器人恢复行为中文图

参考文档:

- move\_base pkg: [https://index.ros.org/p/move\\_base/github-ros-planning-navigation/#noetic](https://index.ros.org/p/move_base/github-ros-planning-navigation/#noetic)
- move\_base wiki: [http://wiki.ros.org/move\\_base](http://wiki.ros.org/move_base)

#### 4.1.4.2 全局规划器 (base\_global\_planner)

简介: 全局规划器基于全局代价地图生成从起始位置到目标位置的最优路径, 确保机器人避开在slam 建立出的图中已知障碍物并高效导航。

通俗的来讲, 例如我们要从宿舍 (A03) 到教学楼 (4 教), 打开地图导航 app, 会自动为我们规划出路线, 但是是在已有地图即静态地图上规划的。如果此时规划的路径中有障碍物就没有办法预测, 因为只能对当前地图当时的路况进行路径规划, 而不能进行实时的规划。

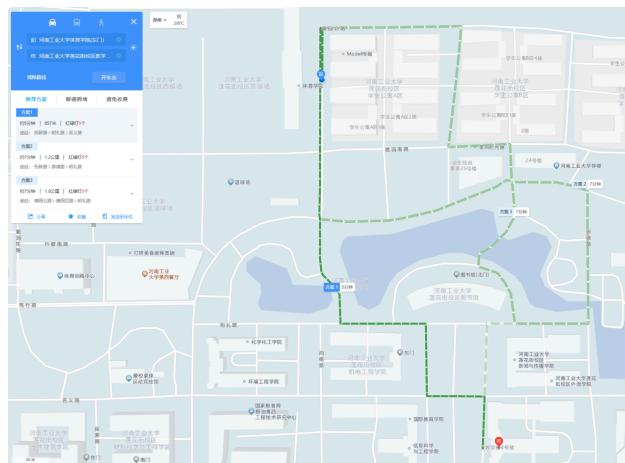


图 94: 路径示例图

在图中，我们可以看到有多条路径可以到达目的地，也就是所谓的条条大路通罗马。但是各个路径的侧重点不同，有的路径侧重于出发点和终点的行驶距离最短，有的侧重于不拥挤... 等。

#### 4.1.4.3 全局规划算法

对于机器人的导航也是如此，不同的全局规划算法的侧重点也不一样，下面我罗列几个常用的全局规划算法：

Dijkstra 算法:Dijkstra 算法是一种经典的图算法，基于贪心策略解决单源最短路径问题，适用于加权有向或无向图，要求边权非负；其核心思想是从起点开始，逐步找到当前未访问顶点中距离起点最近的顶点，更新其邻接顶点的最短路径，直至所有顶点都被访问。是一种典型的广度优先算法。

Dijkstra 算法的原理可以通过以下类比来理解：假设你在一个城市的某个位置，想要知道从你出发到城市中其他所有地方的最短路线，而道路之间的距离是已知的。你可以按照以下步骤规划：

- 标记起点为已知的最近位置：把你当前所在位置的距离记为 0。其他地方的距离都先标记为“无限大”（因为你还不知道怎么走过去）。
- 探索最近的地方：从你目前已知的地点中，选择一个距离最短的地方去尝试访问。
- 一旦到达这个地方，检查以它为中转站是否能更短地到达其他地方。如果可以，更新那些地方的距离。
- 不断选择剩余中距离最短的地点，继续更新其他地点的距离，直到所有地方都被探索完毕。

整个过程就像不断向外扩展一张网，从最近的地方开始逐步覆盖整个城市，每次都选择最短的路径进行优化，最终确保每个地方的距离是最短的。最后，Dijkstra 算法的结果就是：从起点出发，到每个目的地的最短距离，以及对应的路径。

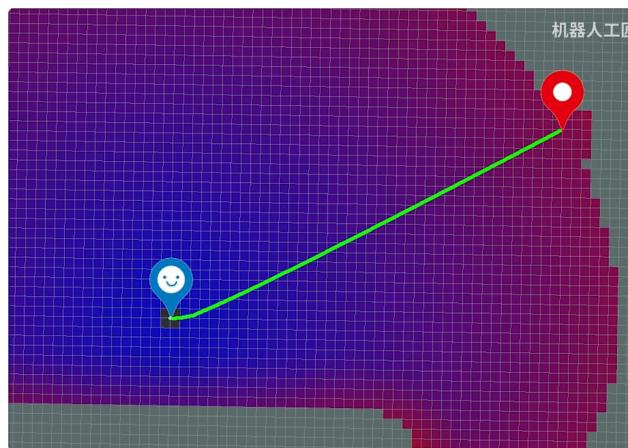


图 95: 路径示例图

优点：

- 路径最优性：对于静态环境中规划最短路径，Dijkstra 算法能够确保从起点到目标点的路径是全局最优的
- 适应性强：适合在明确地图的机器人导航中应用，如栅格地图或拓扑地图，尤其是边权非负的情况下
- 实时性优化：结合堆优化等技术，算法在实际应用中计算速度较快，适合中小规模地图路径规划。
- 确定性强：每次运行结果相同（无随机性），对机器人需要高可靠性的任务非常友好。

缺点：

- 计算成本较高：对于大规模环境（例如城市级别地图），算法计算复杂度较高，可能导致实时性不足
- 动态环境适应性差：在动态环境中（如机器人需要规避移动障碍物），每次变化都可能需要重新计算路径，效率较低。
- 存储需求较大：在高分辨率地图上，算法需要存储大量节点信息，可能对存储资源要求较高。
- 局限性（无法处理负权边）：虽然机器人应用中通常不涉及负权，但如果存在动态代价调整（例如能量消耗、风险等），需要额外调整算法。

#### 4.1.4.4 A\* 算法

A\* 算法是一种基于 Dijkstra 算法的改进路径规划算法，结合了实际代价  $g(n)g(n)$  和预估代价  $h(n)h(n)$ （启发式函数）的综合评估来选择扩展节点，从而高效地搜索从起点到目标点的最优路径；它兼具准确性和高效性，适用于静态地图中复杂环境的路径规划。是一种典型的深度优先的算法。

A\* 算法的原理可以通过以下类比来理解：假设你在城市中寻找最近的咖啡店

- 起点是你当前的位置，目标是找到最近的咖啡店。
- 地图上的道路有不同的长度（实际代价  $g(n)g(n)$ ），你可以测量到目前为止走的距离。
- 你知道大概的方向（预估代价  $h(n)h(n)$ ），例如，你能用地图判断咖啡店的直线距离

怎么做？每次选择下一步时，你会考虑：

- 已经走过的距离（实际代价  $g(n)g(n)$ ）。
- 还需要走的估计距离（预估代价  $h(n)h(n)$ ）。
- 两者相加，选择“总路程”最小的方向走。

### 具体过程

- 从起点出发，尝试所有能走的路，计算它们的“总路程”（走过的 + 预估的）。
- 选择“总路程”最小的方向继续走。
- 走到一个地方后，再尝试更新从这里到附近其他地方的最短路径（比如找到了一条绕路但更快的方式）。
- 一直重复这个过程，直到到达咖啡店。

A\* 的聪明之处：A\* 不仅关注已经走过的路，还用预估的直线距离来指引方向，这比“盲目地探索每条路”（Dijkstra）更快，更接近人类的直觉。

想象一下，如果你知道咖啡店在“东南方向”，即使目前走的路稍微绕一点，只要整体朝东南方向前进，你也会更快到达，而不会去尝试完全无关的方向（比如往西北走）。这就是 A\* 算法的核心思想：既考虑实际代价  $g(n)g(n)$ ，也利用预估代价  $h(n)h(n)$  来指引方向，快速找到目标。

A\* 算法的优缺点：

优点：

- 高效性：A\* 算法比 Dijkstra 算法更高效，因为它利用了启发式函数（预估代价  $h(n)h(n)$ ）来引导搜索方向，减少了不必要的探索，通常能更快地找到最短路径。
- 最优性：如果启发式函数  $h(n)h(n)$  是一致的和乐观的（即不会高估到目标的距离），A\* 保证能找到从起点到目标的最短路径。
- 灵活性：A\* 可以根据不同的场景设计不同的启发式函数，使得它能够适应各种不同的环境，例如平面图、网格图等。
- 对于复杂的路径规划问题，如机器人导航，A\* 能够在具有障碍物和不规则地形的环境中高效地寻找最短路径。

缺点：

- 计算资源消耗大：A\* 算法的计算复杂度较高，特别是在大规模图（例如广阔的城市地图或大型游戏世界）中，可能需要大量的内存和计算资源。
- 依赖启发式函数：A\* 的效率和准确性强烈依赖于启发式函数的设计。如果启发式函数不合适（如高估距离），可能会导致搜索效率下降或无法保证最优性。
- 无法处理动态变化的环境：如果环境中的障碍物或路径发生变化，A\* 算法可能需要重新计算整个路径，这在动态环境中效率较低。
- 可能会计算冗余路径：在某些情况下，A\* 可能会扩展大量不必要的节点，特别是在启发式函数不够精确或问题规模过大的情况下，可能导致不必要的计算量。

#### 4.1.4.5 ROS 中用的全局规划器

全局规划器	变量名称
Navfn	navfn/NavfnROS
Global_planner	global_planner/GlobalPlanner

Listing 30: 节点参数设计

---

在move\_base节点参数中便可设置

```
<launch>
<node pkg = "move_base" type = "move_base" name = "move_base">
<!-- 其他参数 -->
<param name = "base_global_planner" value =
"global_planner/GlobalPlanner" />
<param name = "GlobalPlanner/use_dijkstra" value = "true" />
<param name = "GlobalPlanner/use_grid_path" value = "false" />
<!-- 由于现在的电脑算力足够， 默认使用dijkstra算法即可 -->
</node>\\
</launch>\\"
```

---

Navfn (navigation function) 和 Global\_planner 都包含了 Dijkstra 算法和 A\* 算法，默认使用的都是 Dijkstra 算法。

起初只有 Navfn 没有 Global\_planner，但是 Navfn 的 A\* 算法有 bug，一直没人修复。Navfn 默认使用的是 Dijkstra 算法，因此只要不用 A\* 算法就没有问题。

但是为了解决这个问题，于是 Global\_planner 应运而生，修复了 A\* 的 bug，且性能更强，功能扩展性好，且有维护，因此不断取代 Navfn 成为了主流。

但是注意 move\_base 默认依然使用的是 Navfn，因此要用 Global\_planner，就要自己设置参数了。

此外，ROS 是支持自己编写规划器，按照 pluginlib 的格式自行编写规划器插件即可。

参考文档：

- `global_planner` pkg: [https://index.ros.org/p/global\\_planner/github-ros-planning-navigation/#noetic](https://index.ros.org/p/global_planner/github-ros-planning-navigation/#noetic)
- `global_planner` wiki: [http://wiki.ros.org/global\\_planner](http://wiki.ros.org/global_planner)

#### 4.1.4.6 局部规划器 (local\_planner)

**简介:** 局部规划器负责在全局规划器生成的路径基础上，实时计算和调整机器人在动态环境中的运动路径，以避开临时出现的障碍物，确保机器人平滑、安全地到达目标。

局部规划器是真正决定机器人是如何移动的，先前我们已经了解全局规划器在静态地图上的路径规划，但是实际上在路径上可能存在当时地图没有的障碍物，为了能够正确的导航到目标点，所以我们引入了局部规划器，实时观测机器人的周围情况，适当调整修改机器人的运动路径，绕开障碍物，正确抵达目标点。

**ROS 中的局部规划器:** 局部规划器至关重要，不按规划路线走，或者对临时的障碍物处理不当，有可能导致导航失败，甚至可能因为碰撞对机器人造成损伤，得不偿失。

因此很多开发人员都自己开发局部规划器，但还是有很多开源规划器供大家使用，下面我罗列几个常用的局部规划器：

局部规划器	变量名称
Trajectory Planner	<code>base_local_planner/TrajectoryPlannerROS</code>
DWA Planner	<code>dwa_local_planner/DWAPlannerROS</code>
Eband Planner	<code>eband_local_planner/EBandPlannerROS</code>
TEB Planner	<code>teb_local_planner/TebLocalPlannerROS</code>

Trajectory Planner 和 DWA Planner 是 ROS 自带的局部规划器，第三方的 Eband Planner 和 TEB Planner。

在 move\_base 节点参数中便可设置

Listing 31: 节点参数设计

---

```
<launch>
<node pkg = "move_base" type = "move_base" name = "move_base">
<!-- 其他参数 -->
<param name = "base\_local\_planner" value ="dwa\_local\_planner/DWAPlannerROS" />
</node>
</launch>
```

---

Trajectory Planner 是 ROS 默认使用的规划器，内部实现使用的是 DWA 算法，但是代码质量不高。于是，另起炉灶开发了 DWA Planner，代码可读性更好，运行效率更高，因此选择一般使用 DWA Planner 代替 Trajectory Planner。

DWA Planner:DWA (Dynamic Window Approach) 算法是一种常用于机器人局部规划的算法，它通过考虑机器人当前的速度、加速度以及动态约束，计算一个可能的运动窗口，并从中选择最佳的运动轨迹。

可以把 DWA (Dynamic Window Approach) 算法类比为开车时选择行驶路线的过程：

- 定义动态窗口：想象你正在开车，车速有一个限制，比如最大时速和加速度。你并不是随意决定任何速度，而是根据当前的车速和加速度限制，决定在接下来的几秒钟内你可以选择的合理车速范围——这就像动态窗口。
- 生成候选轨迹：在这个车速范围内，你开始考虑不同的行驶路线：你可以选择快一点、慢一点，或者在不同的方向上微调车速，尝试不同的行驶方式。这就像在道路上选择不同的行驶路径，避免碰到障碍物（如其他车辆或路标）并尽量快速到达目的地。
- 评估轨迹：每条行驶路线你都会评估一下，看看是否有障碍物（比如前方有停驶的车辆）、是否平稳（比如急刹车或急转弯不安全）以及是否更靠近目的地。你希望选择一个既安全又高效的路线。
- 选择最佳轨迹：最后，你选择一条最佳的路线，可能是最不拥堵、最安全、且能够快速接近目的地的路线，然后按照这个路线行驶。

优点：

- 实时性强：DWA 算法能够快速计算出适合当前时刻的局部轨迹，适用于动态环境下的实时路径规划，能够及时响应障碍物或目标的变化。
- 计算效率高：由于 DWA 算法基于机器人的当前速度和加速度限制进行计算，它生成候选轨迹的过程简单、快速，计算量较小，因此适合在资源有限的实时系统中使用。
- 适应性强：DWA 能够根据周围的环境动态地调整运动轨迹，避开临时出现的障碍物，同时保持机器人平稳移动。
- 避障效果好：DWA 算法能够有效避免碰撞，它会评估每条候选轨迹的安全性，选择避免障碍物的最优路径。

缺点：

- 局部最优：DWA 算法通常生成的路径是局部最优的，它只能在当前的局部环境中做出最好的选择，可能会错过更长远的全局优化路径，因此可能无法保证全局最优解。

- 对初始条件敏感：该算法的效果很大程度上依赖于机器人当前的速度、加速度以及所设置的参数。如果起始条件不合适，可能导致规划的路径不理想，甚至无法避开障碍物。
- 对动态障碍物的适应性差：尽管 DWA 能够应对一些简单的动态障碍物，但如果环境中有大量快速变化的障碍物，DWA 的效果可能下降，机器人可能无法及时做出反应。
- 路径平滑性不足：DWA 算法关注的是快速避障和实时性，生成的路径可能不是非常平滑，特别是在复杂环境中，机器人可能会做出急剧的转弯或速度变化。

TEB Planner: TEB (Timed Elastic Band) 算法是一种局部路径规划方法，它通过在时间和空间约束下优化机器人的运动轨迹。该算法将路径表示为一个由多个离散点组成的弹性带，并通过优化这些点的位置和时间来生成平滑、可行且高效的轨迹。TEB 算法不仅考虑了避障，还能优化轨迹的平滑性和速度，确保机器人在动态环境中实时调整路径，避免碰撞，并尽可能高效地接近目标。

用一个简单的类比解释，想象你正在骑自行车穿越公园。

- 路径像一条“弹性带”：想象你骑车的路线不是固定的，而是像一条弹性带，你可以根据需要拉伸或收缩，绕开公园里的树木、池塘或者石头。
- 考虑时间和速度：除了避开障碍物，你还需要在一定时间内到达终点。你可以调整骑车的速度，快速骑行，或者在需要时减速，确保自己不撞到东西，同时保持在合理的时间内到达目的地。
- 不断调整路线：你可能会遇到一只突然跑出来的小狗，或者前方有一个大水坑。TEB 算法就像你在这种情况下，快速决定是否绕路、减速或者加速，确保顺利通过。
- 平滑的骑行：你不会突然加速或猛刹车，否则会摔倒。TEB 算法确保你骑行的路线平滑，避免任何剧烈的转弯或突然停止。

优点：

- 平滑的路径规划：TEB 算法能够生成平滑的路径，避免了机器人在运动过程中出现剧烈的转弯或突然的速度变化，从而提供更稳定的运动表现。
- 实时优化：它能够实时根据环境变化（如动态障碍物或目标位置）调整路径，使机器人在复杂环境中能够灵活应对。
- 考虑时间因素：TEB 不仅优化路径的空间布局，还加入了时间维度，能够有效控制机器人的速度，使路径规划更加符合实际运动需求，避免不必要的加速或减速。
- 适应动态环境：该算法适合应用于动态环境，能够在机器人运动过程中不断调整轨迹，避免碰撞并实时响应环境变化。

缺点：

- 计算复杂度较高：TEB 算法需要在每次迭代中对路径进行优化，计算量较大，尤其在复杂环境中，计算时间可能较长，影响实时性。
- 对参数敏感：算法的效果依赖于多个参数的设置，如权重、时间步长等。参数调整不当可能导致路径规划不理想，甚至出现不稳定的行为。
- 依赖精确的地图和定位：TEB 算法需要准确的环境地图和机器人的位置估计。如果地图不准确或定位误差较大，可能会影响路径规划的效果。
- 适应性有限：尽管 TEB 算法能够适应动态环境，但对于一些极端或高度复杂的环境（例如有大量动态障碍物或快速变化的情况），它的性能可能会有所下降。

注意 TEB Planner 到达目的地会有一个弧线倒车的动作，所以尽量选择目标点有足够的后部空间来倒车。

参考资料：

- DWA Planner wiki: [http://wiki.ros.org/dwa\\_local\\_planner?distro=noetic](http://wiki.ros.org/dwa_local_planner?distro=noetic)
- TEB Planner wiki: [http://wiki.ros.org/teb\\_local\\_planner](http://wiki.ros.org/teb_local_planner)

## 4.2 自主充电

- 目的：在机器人到达目的地附近时，通过摄像头，ar-track-alvar(关于 ar 码识别，追踪的功能包，可通过

```
sudo apt-get install ros-melodic-ar-track-alvar
```

图 96: 代码

来下载) 识别充电桩上的 AR 码，得到 AR 码的位置，根据 AR 的位置控制机器人正对 AR 码，并最终停留在 AR 码的正前方。最终控制机器人向前走固定的距离，走上充电桩进行充电

- 实现：

1. 调用 /track 服务控制机器人跟随 AR 码，调整机器人姿态；类型：ar\_pose/Track

```

| /track
| int8 ar_id;跟踪的ar id
| float32 goal_dist: 机器人中心点(机器人半径为0.19m)与目标ar码的距离
| ---
| string message
| bool success

```

图 97: 代码

2. 调用/relative\_move 服务, 控制机器人相对运动, 走到充电桩上。类型: relative\_move/SetRelativeMove

```

| /relative_move
| geometry_msgs/Pose2D goal :移动方向及距离。
| string global_frame :全局坐标系一般为/odom, 使用导航时可设置为/map。
| bool finishStopObstacle :停障碍(由于障碍物遮挡导致的停车, 一般在开启避障后)时是否直接结束
| ---
| bool success
| string message

```

图 98: 代码

3. 创建 sed\_localization 功能包, 终端输入:

```

$ cd ~/ros_workspace/src
$ catkin_create_pkg sed_localization roscpp ar_pose relative_move

```

图 99: 代码

4. 在 /ros\_workspace/src/sed\_localization/src 路径下创建 tracking.cpp 文件:

```

$ rosdep sed_localization/src
$ touch tracking.cpp

```

图 100: 代码

打开 tracking.cpp 文件, 写入以下内容:

Listing 32: 创建 tracking.cpp 文件

---

```

#include <ros/ros.h>
#include <relative_move/SetRelativeMove.h>
#include <ar_pose/Track.h>
int main(int argc, char** argv)
{
    ros::init(argc, argv, "auto_charge");// 初始化ros\
    ros::NodeHandle n;// 定义句柄
    ros::ServiceClient relative_move_client = n.serviceClient<relative_move::SetRelativeMove
    ↪ >("relative_move");// 定义相对运动客户端

```

```

ros::ServiceClient ar_track_client = n.serviceClient<ar_pose::Track>("track");// 定义 ar 码
    ↳ 跟踪客户端
relative_move::SetRelativeMove RelativeMove_data;
ar_pose::Track Track_data;
ros::service::waitForService("relative_move");// 等待服务启动
ros::service::waitForService("track");// 等待服务启动
Track_data.request.ar_id = 0; // 跟踪 0 号 ar 码
Track_data.request.goal_dist = 0.3;
ar_track_client.call(Track_data);// 调用服务
// 定义请求值
RelativeMove_data.request.goal.x = -0.1;
RelativeMove_data.request.global_frame = "odom";
relative_move_client.call(RelativeMove_data);// 调用服务
std::cout << "定位完成" << std::endl;
std::cout << "现在退出程序" << std::endl;
return 0;
}

```

---

修改 /ros\_workspace/src/sed\_localization/CMakeLists.txt 文件:

```

add_executable(track_node src/tracking.cpp)
add_dependencies(track_node ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})
target_link_libraries(track_node
    ${catkin_LIBRARIES}
)

```

图 101: 代码

编译功能包, 终端输入:

```

3 $ cd ~/ros_workspace/
9 $ catkin_make

```

图 102: 代码

- 创建启动文件

- 在路径 sed\_localization/launch 下创建 tracking.launch 文件, 终端输入:

```

$ cd ~/ros_workspace/src/sed_localization
$ mkdir launch
$ cd launch
$ touch tracking.launch

```

图 103: 代码

2. 打开 tracking.launch 文件, 写入下面内容:

Listing 33: 创建启动文件

---

```
<launch>
<!-- 打开底盘相机二维码检测 -->
<include file="$(find ar_pose)/launch/ar_base.launch"/>
<!-- 打开相对移动功能（服务端） -->
<include file="$(find relative_move)/launch/relative_move.launch"/>
<!-- 实验节点 -->
<node pkg="sed_localization" name="sed_localization" type="track_node" output="screen"/>
</launch>
```

---

- 运行程序, 终端输入: 运行以下命令启动导航

```
# 真实机器人
$ roslaunch bobac3_navigation bobac3_nav_2d.launch
# 仿真机器人
$ roslaunch bobac3_navigation demo_nav_2d.launch
```

图 104: 代码

将机器人导航至充电桩前（相机可以看到充电桩 ar 码）后, 运行以下代码, 对准充电桩。

```
$ roslaunch secondary_localization track.launch
```

图 105: 代码

## 4.3 语音和综合调度

### 4.3.1 环境配置

- 下载语音采集驱动在终端中输入以下指令 (要联网)  
`sudo apt install libasound2-dev`
- 下载下面百度网盘链接的功能包, 并将其放到虚拟机系统 `ros_workspace/src` 路径下。链接:  
<https://pan.baidu.com/s/1j2RNCZLwqX0y5KLndBtWcA> 提取码: v0mk
- 然后编译工作空间 `cd /ros_workspace catkin_make`
- 安装命令行播放软件 `sudo apt-get install sox`

### 4.3.2 语音导航

流程图：

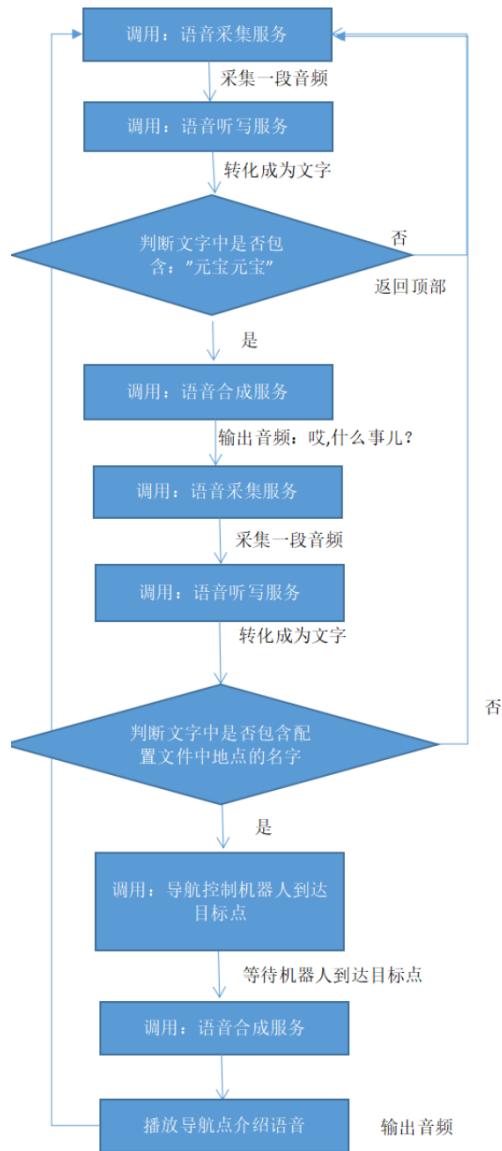


图 106: 流程图

代码展示

Listing 34: 代码展示

```
#include <ros/ros.h>
#include <robot_audio/robot_iat.h>
```

```
#include <robot_audio/Collect.h>
#include <robot_audio/robot\_tts.h>
using namespace std;
class Interaction {
public:
Interaction();
string voice_collect();
string voice_dictation(const string& filename);
string voice_tts(const string& text);
private:
ros::NodeHandle n;
ros::ServiceClient collect_client, dictation_client, tts_client;
};
Interaction::Interaction() {
collect_client = n.serviceClient<robot_audio::Collect>("voice_collect");
dictation_client = n.serviceClient<robot_audio::robot_iat>("voice_iat");
tts_client = n.serviceClient<robot_audio::robot_tts>("voice_tts");
ac = make_unique<AC>("move_base", true);
}
string Interaction::voice_collect() {
ros::service::waitForService("voice\_\_collect");
robot_audio::Collect srv;
srv.request.collect_flag = 1;
collect_client.call(srv);
return srv.response.voice_filename;
}
string Interaction::voice_dictation(const string& filename) {
ros::service::waitForService("voice_iat");
robot_audio::robot_iat srv;
srv.request.audiopath = filename;
dictation_client.call(srv);
return srv.response.text;
}
string Interaction::voice_tts(const string& text) {
ros::service::waitForService("voice_tts");
robot_audio::robot_tts srv;
srv.request.text = text;
tts_client.call(srv);
string cmd = "play " + srv.response.audiopath;
system(cmd.c_str());
sleep(1);
return srv.response.audiopath;
}
int main(int argc, char** argv) {
setlocale(LC_ALL,"");
ros::init(argc, argv, "interaction");
Interaction audio;
string dir, text, path;
while (ros::ok())
{
dir = audio.voice_collect();
text = audio.voice_dictation(dir);
```

```

if (text.find("提示词") != string::npos)
{
    audio.voice_tts("回答词");
    写一个函数实现功能，然后在此处调用，可实现语音调用功能
}
}

return 0;
}

```

---

- `voice_collect`、`voice_dictation`、`voice_tts` 客户端请求函数
- `text.find(" 提示词") != string::npos`  
“提示词”换成你要语音说的唤醒任务词，例如带我参观一圈等。然后就执行 if 语句里面的功能。
- `audio.voice_tts(" 回答词")`  
“回答词”是机器人回答你说的话，可自行设置。例如好的，开始执行等。

### 4.3.3 客户端 launch 编写

Listing 35: launch 编写

---

```

<launch>



<node name="voice\_\_collect" pkg="robot\_audio" type="voice\_collect\_node" output="screen">
    <!-- 音频文件目录 -->
    <param name="audio\_file" type="string" value="./AIUI/audio/audio.wav"/>
</node>

<node pkg="robot\_audio" type="voice\_aiui\_node" name="voice\_aiui\_node"/>
</launch>

```

---

功能代码是开启了客户端，`launch` 是开启服务端，只有客户端和服务端都开启，才能正常运行，且运行时一定要联网！不联网的情况下，机器人会识别不出你说的话，不能转换为文字，自然不能执行任务。

### 4.3.4 综合调度

完成比赛的两种方式

- 各个任务各写一个 `launch` 来完成优点：每个任务相互独立，相互影响较少缺点：要启动三次 `launch`，比较浪费时间
- 集成所有的任务全在一个功能代码里，根据不同的语音唤醒词，来区别各个任务。优点：省时间，省力缺点：三个任务集成在一个，中间有错误就一个也执行不起来。可自行选择调度模式。

集成的综合调度：

Listing 36: 调度代码

---

```

while (ros::ok())
{
    dir = audio.voice_collect();
    text = audio.voice_dictation(dir);
    if (text.find("提示词1") != string::npos)
    {
        audio.voice_tts("回答词1");
        写一个函数实现功能，然后在此处调用，可实现语音调用功能
    }
    if (text.find("提示词2") != string::npos)
    {
        audio.voice_tts("回答词2");
        写一个函数实现功能，然后在此处调用，可实现语音调用功能
    }
}

```

---

以此为例，根据不同的提示词调用不同的功能，从而实现各个任务的实现。

Listing 37: 任务实现

---

```

<launch>
<!-- 功能节点 -->
<!-- 加入自己的功能代码节点 -->
<!-- 打开语音采集节点 -->
<node name="voice_collect" pkg="robot_audio" type="voice_collect_node" output="screen">
<!-- 音频文件目录 -->
<param name="audio_file" type="string" value="./AIUI/audio/audio.wav"/>
</node>
<!-- 打开语音服务 -->
<node pkg="robot_audio" type="voice_aiui_node" name="voice_aiui_node"/>
</launch>

```

然后需要的节点和 launch 同样也要集成在这一个 launch 里，节点和节点之间，服务端和客户端之间的关系及运行环境都  
→ 要集成这里。

---

## 4.4 人脸识别

### 4.4.1 人脸识别概述

ROS 中的人脸识别是一种在机器人应用中使用计算机视觉技术来检测和识别人脸的能力。这种功能通常用于交互式机器人、安防机器人、服务机器人等场景，帮助机器人识别和与人类进行互动。

主要步骤

- 图像采集：机器人使用相机传感器（如 RGB 或 RGB-D 相机）捕获视频或图像。在本文中将使用

ubuntu 中软件商店中的“茄子”相机来获取照片。



图 107: 图像采集

- 人脸检测：通过使用现有的人脸检测算法（如 OpenCV 中的 Haar 特征分类器、Dlib、或者深度学习模型）来定位图像中的人脸。
- 人脸识别：检测到人脸后，可以使用特征提取方法（如 HOG、LBPH、或者深度学习模型如 FaceNet）从人脸中提取特征，然后与数据库中的已知人脸进行比较来识别身份。

#### 常用的 ROS 人脸识别包

- **face\_recognition**: 这是一个用于人脸识别的 ROS 包，提供了基础的人脸识别功能，基于 OpenCV 和 Dlib 等库。你可以使用 Python 和命令行工具提取、识别、操作人脸。（本文将采用这个包依照元创科技的人脸识别培训教程来完成）
- **opencv\_apps**: 这是一个 ROS 包，集成了 OpenCV 的功能，包括人脸检测。它可以直接在 ROS 中使用，用于简单的人脸检测任务。
- **vision\_opencv**: 这是 ROS 中集成 OpenCV 的工具包，可以使用 OpenCV 的所有功能，包括人脸检测和识别。
- **dlib\_face\_recognition**: Dlib 是一个常用于人脸检测和识别的库，结合 ROS 可以实现更高精度的人脸识别功能。

#### 4.4.2 人脸识别实现

首先需要下载元创科技提供的软件包，里面包含了所需代码文件，使用功能包之前需要安装以下的库：pip install dlib==19.19.0,pip install face\_recognition。

安装完成后可以打开里面有两个自定义消息类型，分别是 face date 与 face result 其中 face date 包含了人脸的信息比如名字还有人脸框的大小，face result 包含了返回的结果（检测到的人脸个数）。这个功能包还包括了 face\_rec\_service.py 节点，这个节点可以识别相机中人脸的数据，启动方式 rosrun face\_rec face\_rec\_service.launch。还有 face\_rec\_topic.py 节点启动这个节点会持续不断地识别人脸，启动方式：rosrun face\_rec face\_rec\_topic.launch

人脸信息的添加：如果想使用自己的人脸信息，只需要在 face\_date 中新建一个文件夹将这个文件夹命名为人脸的名字，然后将人脸照片放到这个文件夹中就可以了。在比赛中要求人机交互，需要将人脸识别与语音综合调度起来，可以先按照上述将人脸识别整理好，在 face\_date 中加入人脸数据后，运行 face\_rec\_service.py 就可以检测了，需要注意的是检测到的人脸数据如果返回的是中文则需要在运行节点的终端中查看。

##### 注意事项

获取照片时，需要使用摄像头，如果你使用的是虚拟机则还需要将摄像头与虚拟机连接。点击虚拟机右下角如下所示：

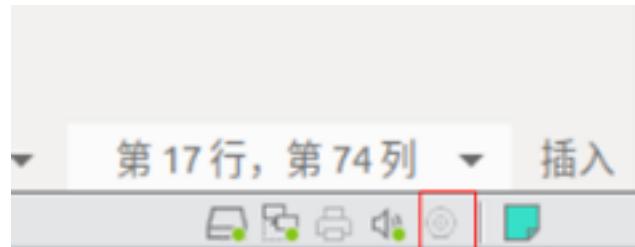


图 108: 相机连接

在弹出的框中选择：“连接（断开与主机的连接）”选项。然后会弹出如下对话框，点击确定。



图 109: 对话框

如果成功则如下图所示：



图 110: 连接成功

这就成功的连接上了摄像头。