动手学 ROS2

项目地址：https://github.com/fishros/d2lros2

公众号：鱼香 ROS

作者：小鱼（fishros）

店铺：淘宝搜店`鱼香 ROS`

注意，本文档为开源在线文档，实时更新，本 PDF 为每日生成版本

最新版在线地址：https://fishros.com/d2lros2

Table of Contents

- （一）ROS2入门篇
  - 第 1 章 ROS2介绍与安装
    - 章节导读
    - 基础篇-Linux基础
      - 1.Linux与Ubuntu系统介绍
      - 2.在虚拟机中安装Ubuntu
      - 3.玩转Ubuntu之常用指令
      - 4.玩转Ubuntu之编程工具
      - 5.玩转Ubuntu之常用软件
    - 入门篇-ROS2介绍安装
      - 1.ROS2前世今生
      - 2.ROS与ROS2对比
      - 3.动手安装ROS2
      - 4.ROS2初体验
    - 进阶篇-架构与中间件
      - 1.ROS2系统架构
      - 2.中间件DDS架构
  - 第 2 章 ROS2第一个节点
    - 章节导读
    - 基础篇-编程基础
      - 1.使用gcc编译ROS2节点
      - 2.使用make编译ROS2节点
      - 3.使用CMakeLists.txt编译ROS2节点
      - 4.CMake依赖查找流程
      - 5.Python依赖查找流程
      - 6.Python工具之Setup
    - 入门篇-动手使用ROS2
      - 1.ROS2节点介绍
      - 2.ROS2功能包与工作空间
      - 3.ROS2构建工具之Colcon
      - 4.使用RCLCPP编写节点
      - 5.使用RCLPY编写节点
    - 进阶篇-ROS2系统
      - 1.面向对象编程思想
      - 2.使用面向对象方式编写ROS2节点
      - 3.Colcon使用进阶
      - 4.ROS2节点发现与多机通信
  - 第 3 章 ROS2节点通信之话题与服务

- 第 3 章 ROS2节点通信之话题与服务
  - 章节导读
  - 基础篇-中间件与面向对象基础
    - 1.从底层理解通信
    - 2.通信中间件之ZMQ
    - 3.多线程回调函数和锁
  - 入门篇-话题与服务
    - 1.ROS2话题入门
    - 2.话题之RCLCPP实现
    - 3.话题之RCLPY实现
    - 4.ROS2服务入门
    - 5.服务之RCLCPP实现
    - 6.服务之RCLPY实现
    - 7.ROS2接口介绍
    - 8.自定义接口RCLCPP实战
    - 9.自定义接口RCLPY实战
  - 进阶篇-中间件进阶
    - 1.原始数据类型与包装类型
    - 2.通信质量Qos配置指南
    - 3.DDS进阶之Fast-DDS环境搭建
    - 4.使用DDS进行订阅发布
- 第 4 章 ROS2通信之参数与动作
  - 章节导读
  - 基础篇-控制概述
    - 1.开环控制与闭环控制
  - 入门篇-参数与动作
    - 1.参数（Param）通信
    - 2.参数之RCLCPP实现
    - 3.参数之RCLPY实现
    - 4.动作（Action）通信与自定义接口
    - 5.动作之CPP实现
    - 6.动作之RCLPY实现
    - 7.通信机制对比总结
  - 进阶篇-原理进阶
    - 1.ROS参数通信原理
    - 3.生命周期节点
- 第 5 章 ROS2常用工具
  - 章节导读
  - 基础篇-相关概念
  - 入门篇-常用工具

        - 1.启动管理工具-Launch
        - 2.命令行工具-ROS2CLI
        - 3.数据可视化工具-RVIZ
        - 4.常用调试小工具-RQT
        - 5.数据录播工具-rosbag
        - 6.兼容仿真工具-Gazebo
    - 进阶篇-工具进阶
        - 1.RVIZ2插件开发实践
        - 2.RQT插件开发实践
        - 3.兼容仿真工具WeBots
        - 4.兼容仿真工具UnityForROS2
        - 5.ROS2内存管理工具
- （二）机器人学篇
    - 第 6 章 运动学基础
        - 章节导读
        - 基础篇-数学基础
            - 1.矩阵与矩阵运算
            - 2.MiniConda与Jupyter介绍安装
            - 3.矩阵运算实战
        - 入门篇-机器人运动学
            - 1.空间坐标描述
            - 2.空间坐标描述实战
            - 3.姿态的多种表示
            - 4.姿态转换实战
            - 5.齐次坐标变换
            - 6.齐次坐标变换实战
            - 7.机器人运行学介绍
        - 进阶篇-运动学进阶
            - 待定
    - 第 7 章 ROS2运动学
        - 章节导读
        - 基础篇-常用工具

        - 待定
    - 入门篇-机器人运动学
        - 1.TF2介绍
        - 2.学会使用RVIZ2-TF组件
        - 3.坐标变换发布监听Python实现
        - 4.坐标变换发布监听C++实现
    - 进阶篇-TF时间机制
        - 1.ROS2-TF时间机制

- （三）建模仿真篇
    - 第 8 章 机器人建模
        - 章节导读
        - 基础篇-概念
            - 1.常见机器人构型
            - 2.常见建模软件及工具
        - 入门篇-机器人建模
            - 1.URDF统一机器人建模语言
            - 2.RVIZ2可视化URDF模型
            - 3.创建一个两轮差速模型
            - 4.通过JointStates控制RVIZ2关节
            -
        - 进阶篇-其他建模方式
            - Xacro介绍
            - 使用Xacro简化机器人模型
            - RVIZ2可视化机器人模型原理
            - SolidWorks导出URDF
            - 机械臂建模之DH参数
    - 第 9 章 机器人仿真
        - 章节导读
        - 基础篇-概念
            - 待定
        - 入门篇-机器人仿真
            - 1.机器人仿真介绍
            - 2.给两轮差速机器人添加物理参数
            - 3.在Gazebo加载机器人模型
            - 4.Gazebo仿真插件之两轮差速
            - 5.Gazebo仿真插件之IMU

        - 6.Gazebo仿真插件之激光雷达
        - 7.Gazebo仿真环境搭建
    - 进阶篇
        - 1.Gazebo仿真插件之超声波

- （四）Nav2导航篇
    - 第 10 章 SLAM建图
        - 章节导读
        - 基础篇-图像基础
            - 1.图像常见格式及存储
            - 2.栅格地图介绍
        - 入门篇-SLAM建图
            - 1.SLAM前世今生
            - 2.Carto介绍及安装
            - 3.配置FishBot进行建图
        - 进阶篇-Carto与地图
            - 1.ROS2地图加载与编辑
    - 第 11 章 Nav2导航仿真实战
        - 章节导读
        - 基础篇-基础知识
            - 1.行为树是什么
        - 入门篇-Nav2导航入门
            - 1.Nav2导航框架介绍
            - 2.为FishBot配置Nav2
            - 3.使用FishBot进行自主导航
            - 4.使用Nav2导航API进行导航
        - 进阶篇-Nav2框架
    - 第 12 章 Nav2进阶实践
        - 章节导读
        - 基础篇-路径搜索
            - 1.路径搜索搜索之A星算法-待更新
        - 入门篇-自定义Nav2插件
        - 进阶篇-优化配置
            - 暂定

- （五）ROS2控制硬件篇

- 第 13 章 嵌入式开发之从点灯开始
  - 章节导读
  - 基础篇-嵌入式开发介绍与环境搭建
    - 1.什么是单片机MCU
    - 2.单片机开发平台
    - 3.搭建PlateFormIO开发环境
    - 4.PIO工程结构&构建方式
    - 5.第一个HelloWord工程
    - 6.串口通信-接收实验
  - 入门篇-从点灯开始学起
    - 1.点灯基础-看懂LED驱动电路
    - 2.完成点灯学会GPIO输出
    - 3.使用按键学会GPIO输入
    - 4.电池电压测量-学会使用ADC
  - 进阶篇-学会使用第三方库
    - 1.学会安装第三方开源库
    - 2.使用开源库驱动IMU
    - 3.学会面向对象编程-封装IMU驱动
    - 4.使用开源库驱动OLED
    - 5.通讯协议小课堂-I2C通信
    - 6.I2C通信实验-点亮OLED
- 第 14 章 接入ROS2-MicroROS
  - 章节导读
  - 基础篇-第一个MicroROS节点
    - 1.MicroROS介绍与服务安装
    - 2.你的第一个MicroROS节点
  - 入门篇-在嵌入式平台实现话题与服务通信
    - 1.话题订阅-控制LED
    - 2.话题发布-上传电量信息
    - 3.服务实现-两数相加
  - 进阶篇-MicoROS原理与使用进阶
    - 1.控制OLED-自定义消息接口
    - 2.做个时钟-系统时间同步
    - 3.无线通讯-了解传输原理
    - 4.榨干性能-使用双核运行MicroROS
- 第 15 章 ROS2硬件实战（自制简易雷达）
  - 章节导读
  - 1.简易雷达原理介绍
  - 2.使用超声波测量距离

        - 3.使用pwm控制舵机角度
        - 4.实现循环扫描测量
        - 5.ROS雷达消息合成与发布

- （六）FishBot移动机器人开发(更新中)

    - 第 16 章 移动机器人控制系统搭建
        - 章节导读
        - 1.移动机器人底盘结构介绍
        - 2.从H桥说起-电机驱动原理介绍
        - 3.电机控制之正反转实验
        - 4.电机控制之速度控制实验
        - 5.电机控制之使用开源库驱动多路电机
        - 6.做个遥控车-订阅ROS2 Twist
        - 7.速度测量-学会使用编码器
        - 5.控制电机速度-PID控制器实现
        - 6.控制机器人速度-运动学逆解
        - 7.里程计计算-运动学正解与积分
        - 8.可视化里程计-发布Odometry
    - 第 17 章 FishBot建图与导航实现
        - 章节导读
        - FishBot建图实现
            - 1.可视化雷达点云-学会驱动雷达
            - 2.Cartographer安装与配置
            - 3.使用雷达+里程计完成建图
            - 4.使用纯雷达模式完成建图
            - 5.导航地图编译与保存
        - FishBot导航实现
            - 1.Nav2介绍与安装
            - 2.导航准备1-ROS标准REP105
            - 3.导航准备2-发布里程计TF
            - 4.Nav2地图导入与配置
            - 5.单点多点导航测试
    - 第 18 章 移动机器人导航进阶
        - 章节导读
        - 1.使用API进行导航
        - 2.自定义控制器介绍
        - 3.纯追踪控制器介绍
        - 4.纯追踪控制器代码实现
        - 5.自定义代价地图导航

        - 6.超声波数据发布

- （七）Moveit2机械臂篇
    - 第 19 章 Moveit2仿真
        - 章节导读
        - 基础篇-运动学
            - 机械臂正逆运动学
        - 入门篇-Moveit2仿真
            - Moveit2框架介绍
            - 通过URDF配置Moveit2
            - 通过键盘控制机械臂运动
            - Moveit2-API控制机械臂
            - 给机械臂添加夹爪控制器
        - 进阶篇-架构
            - Moveit2架构剖析
    - 第 20 章 Moveit2进阶
        - 章节导读
        - 基础篇-算法
            - 路径搜索之RRT算法
            - 路径平滑算法
        - 入门篇-Moveit2进阶
            - 自定义规划器
            - 自定义运动学求解器
        - 进阶篇-实战演练
            - 暂定
    - 第 21 章 Moveit2真机控制
        - 章节导读
        - 基础篇-动力学
            - 三次/五次样条插值
        - 入门篇-Moveit2真机控制
            - 机械臂硬件架构
            - 自制机械臂控制及ROS2Control配置
            - 使用Moveit2控制真实机械臂
        - 进阶篇-实战演练
            - 使用真实机械臂完成抓取

- ROS 2常用代码模板
  - rclcpp
    - 节点
    - 参数
    - tf2
    - 时间
    - 点云PCL
    - 解决方案
  - rclpy
    - 节点
    - 参数
    - tf2
    - 时间
  - urdf
    - xacro模板
    - gazebo插件
  - 其他相关
    - CMake
    - Colcon
    - Launch
    - 网络通讯
    - 功能包

第1章 ROS2介绍与安装

1.介绍

本章节分为三个部分，基础篇、入门篇和进阶篇。

- 入门篇，主要介绍Liunx系统的基本知识，带你安装好Ubuntu系统并安装一些常用的工具和软件。
- 基础篇，主要介绍ROS2是什么？和ROS之间的区别是什么？接着带你一起安装体验下ROS2.
- 进阶篇，主要对ROS2的系统架构以及中间件进行介绍，带你了解ROS2的系统原理。

2.食用方法

对于没有用过Ubuntu系统，不会敲命令行的小伙伴，建议你从入门篇开始学起，基础篇将为你提供后续学习的知识和系统环境。

对于已经对ROS2有所了解的同学，可以直接从进阶篇开始学习。

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

1.Linux与Ubuntu系统介绍

1.Linux是什么

1991年，一个名不见经传的芬兰研究生购买了自己的第一台PC，并且决定开始开发自己的操作系统。这个想法非常偶然，最初只是为了满足自己读写新闻和邮件的需求。这个芬兰人选择了Minix作为自己研究的对象。Minix是由荷兰教授Andrew S. Tanenbaum开发的一种模型操作系统，这个开放源代码的操作系统最初只是用于研究目的。

这个研究生名叫Linus Torvalds，他很快编写了自己的磁盘驱动程序和文件系统，并且慷慨地把源代码上传到互联网。Linus把这个操作系统命名为Linux，意指“Linus的Minix”（Linus’ Minix）。

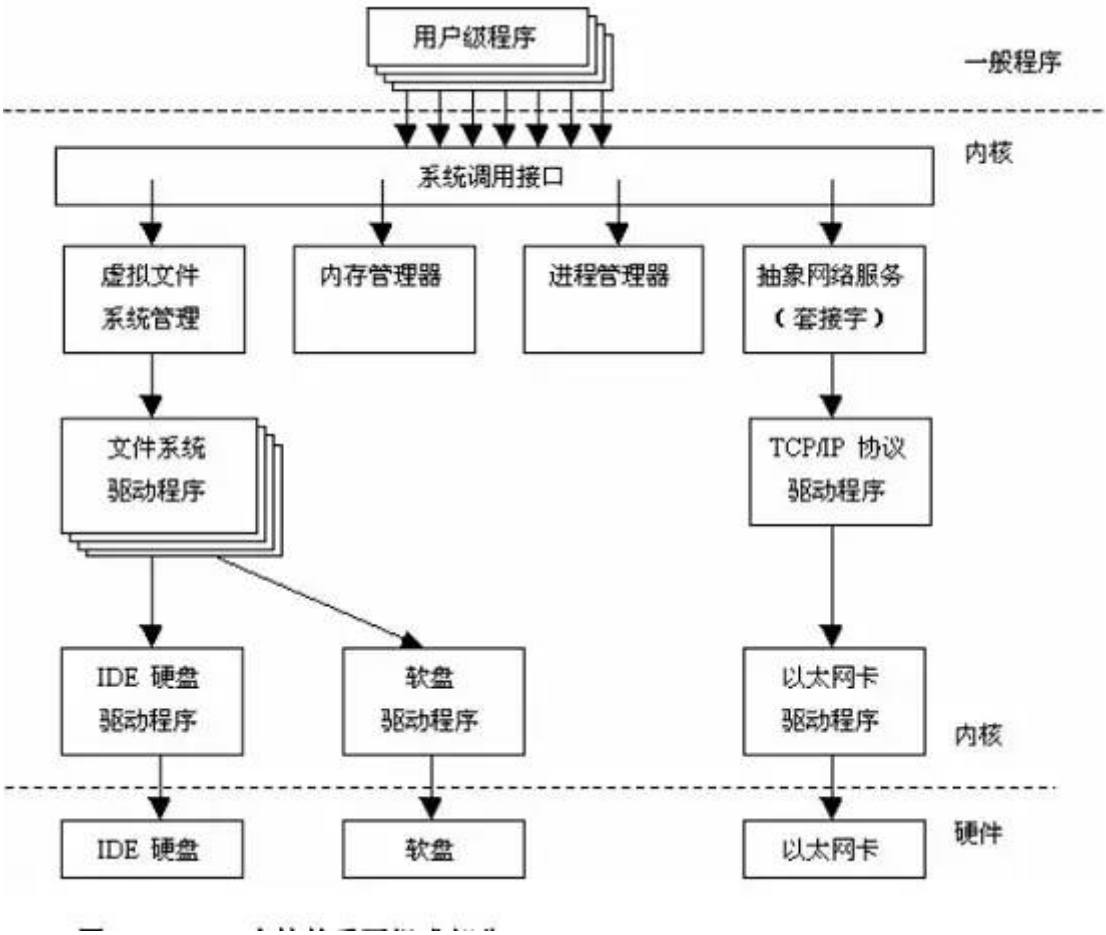
2.Linux系统内核

这里需要了解下什么是操作系统，以及Linux内核是什么？

- 操作系统：管理计算机**硬件**与**软件**资源的计算机程序。
- 内核：**内核是驱动硬件的程序**。

  基于硬件的第一层软件扩充，提供操作系统的最基本的功能，是操作系统工作的基础，它负责管理系统的进程、内存、设备驱动程序、文件和网络系统，决定着系统的性能和稳定性。

基于Linux内核衍生出了很多Linux系操作系统，Ubuntu就是其中之一。



2.Ubuntu是什么

> Ubuntu官网值得看看

2.1 Ubuntu介绍

Ubuntu基于linux内核的桌面PC操作系统，术语上喜欢称Ubuntu是一个 linux 发行版。

Ubuntu是一个以桌面应用为主的Linux操作系统，其名称来自非洲南部祖鲁语或豪萨语的“ubuntu"一词，意思是“人性”“我的存在是因为大家的存在"，是非洲传统的一种价值观。

2.2 Ubuntu版本

按照用途可以分为桌面版、服务器版、Iot版本、风味版、移动版和云上版。

1. 桌面版

   带有漂亮图形界面的桌面版，最易上手和使用。

2. 服务器版

   适用于服务器上的Ubuntu系统，包含常用的服务软件，但是没有图形界面。

3. Iot版

   Ubuntu Core是适用于云和物联网设备，重新设计的，安全的，事务化更新的，轻量级的Ubuntu。

4. 风味版

   Ubuntu风味版提供了一种特别的方式来体验不同默认应用程序、设置的Ubuntu，其由Ubuntu归档（Ubuntu Archive）提供软件包和更新。

3.Ubuntu版本发布规则及代号

注意被小鱼加粗的版本都是带LTS的长期支持版本，都是在偶数年的4月下旬发布的。

使用长期支持（LTS）版本的你同，你**会在较长的时间内获得安全、维护和功能的更新**。LTS 版本被认为是最稳定的版本，它经历了广泛的测试，并且大多包含了多年积累的改进。

版本号	代号	发布时间
22.04 LTS	**Jammy Jellyfish**	**2022-04-19**
21.10	Impish Indri	2021-10-14
21.04	Hirsute Hippo	2021-04-22 [13]
20.10	Groovy Gorilla	2020-10-22
20.04 LTS	**Focal Fossa**	**2020-04-23**
19.10	Eoan Ermine	2019-10-17
19.04	Disco Dingo	2019-4-19
18.10	Cosmic Cuttlefish	2018-10-18
18.04 LTS	**Bionic Beaver**	**2018-04-26**
17.10（GNOME成为默认桌面环境）	Artful Aardvark	2017-10-21
17.04	Zesty Zapus	2017-04-13
16.10	Yakkety Yak	2016-10-20
16.04 LTS	**Xenial Xerus**	**2016-04-21**

4.CPU架构是什么

推荐阅读：

- [Linux]CPU架构/指令集：RISC / CISC | arm | amd | X86/i386 | aarch64 - 千千寰宇 - 博客园 (cnblogs.com)

你需要知道的常见架构有：

- amd64
- arm
- aarch64
- x86/i386(不常用)

根据电脑使用的CPU架构不同，你安装Ubuntu系统时应该选择对应的Ubuntu安装镜像包。

同时注意：不同架构的不同操作系统的软件安装包也是不兼容的。

5.Ubuntu权限管理

推荐阅读：

- ubuntu 权限管理设置 - 掘金 (juejin.cn)

你需要知道的有：

- Linux 系统中的 超级用户 root 账号通常 用于系统的维护和管理，**对操作系统的所有资源 具有所有访问权限**
- sudo 命令用来以其他身份来执行命令，预设的身份为 root，所以我们**可以使用sudo + 命令来提升操作权限**
- chmod 命令可以用于**修改文件权限**

6.Ubuntu上如何安装软件

推荐阅读：

- https://www.cnblogs.com/xwdreamer/p/3623454.html
- Ubuntu安装软件有这5种方法 - 知乎 (zhihu.com)

你需要知道的是：

- 使用apt从服务器下载安装，你需要提前添加服务器地址和服务器的秘钥，这一步就叫添加源或者换源
- 使用源码进行编译安装，你需要下载源码和源码的各种依赖，之后编译出程序拷贝到系统中

- 为什么可以使用apt安装？其实就是软件开发者在自己电脑上编译好程序，把程序打包上传到服务器，你就可以从服务器下载安装了

参考链接：

- https://blog.51cto.com/chinajava/342515

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

2.在虚拟机中安装Ubuntu

为方便学习，可以先使用虚拟机在Windows上使用Ubuntu，当然除了虚拟机还有Windows子系统等方法，你可以自行尝试。

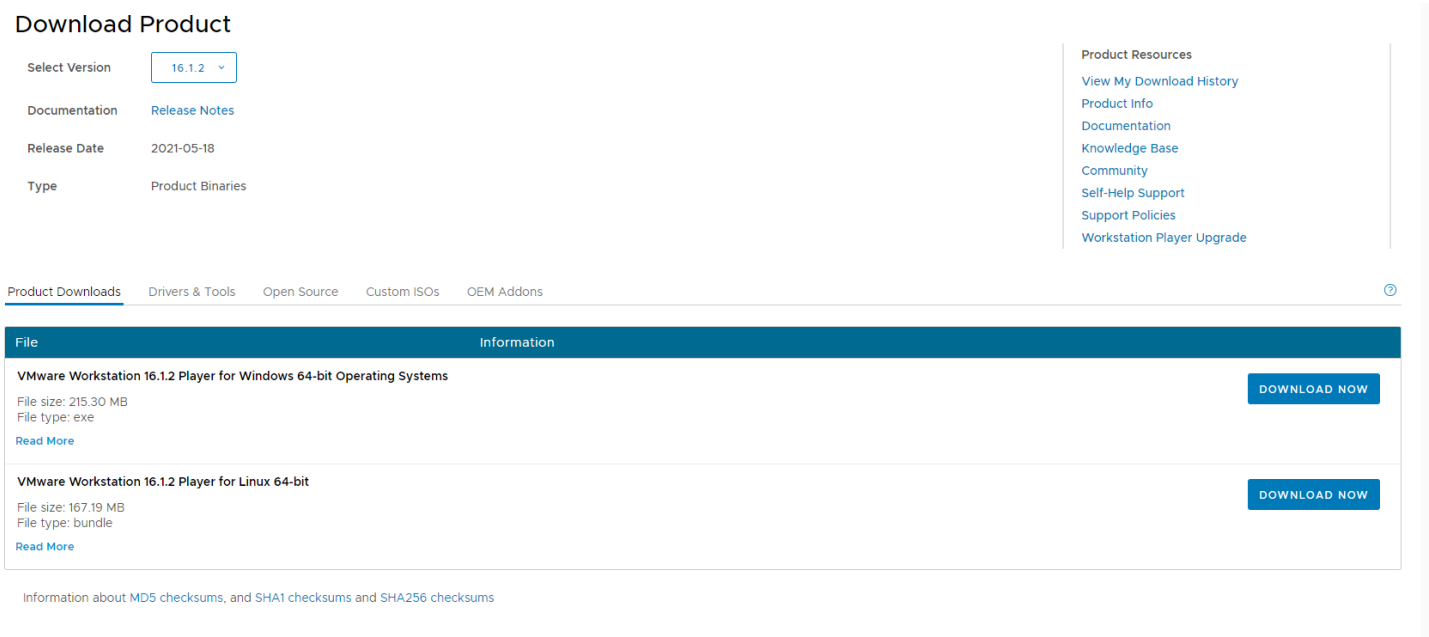
1.下载

所谓虚拟机，就是在你的电脑已有的系统上再使用软件模拟出另外一个系统。比较著名的软件就是Vmware了，因为Vmware是收费的，我们使用他们的非商业版本Vmware-Player。

1.1下载Vmvare

Vmvare官方下载链接：[VMware Workstation Player - My VMware]

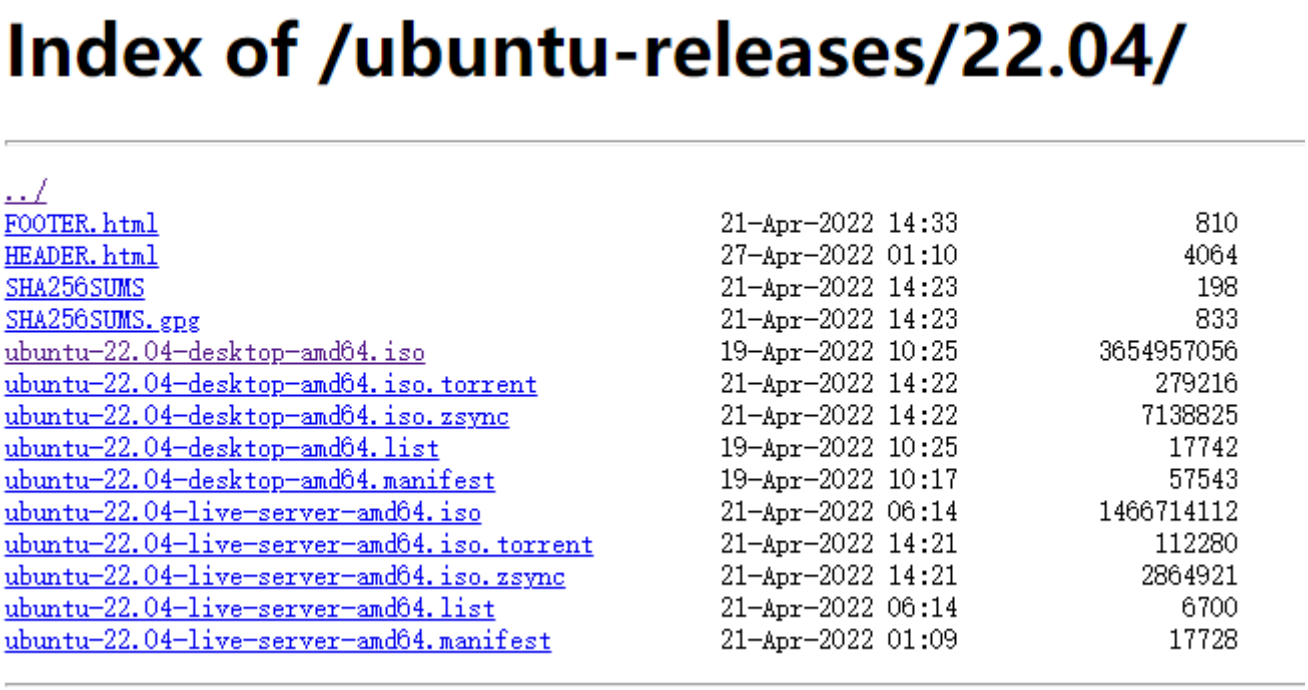
大家在浏览器里打开网页，然后点击下图中的DownLoad Now即可，注意上面一个是windows版本，下面一个是linux版本的。



1.1下载ubuntu

下载好后虚拟机安装包后，接下来下载ubuntu镜像文件。

下载地址：[Index of /ubuntu-releases/22.04/ (ustc.edu.cn)]，点开上面的网址，你会看到下面的页面：



Index of /ubuntu-releases/22.04/

../		
FOOTER.html	21-Apr-2022 14:33	810
HEADER.html	27-Apr-2022 01:10	4064
SHA256SUMS	21-Apr-2022 14:23	198
SHA256SUMS.gpg	21-Apr-2022 14:23	833
ubuntu-22.04-desktop-amd64.iso	19-Apr-2022 10:25	3654957056
ubuntu-22.04-desktop-amd64.iso.torrent	21-Apr-2022 14:22	279216
ubuntu-22.04-desktop-amd64.iso.zsync	21-Apr-2022 14:22	7138825
ubuntu-22.04-desktop-amd64.list	19-Apr-2022 10:25	17742
ubuntu-22.04-desktop-amd64.manifest	19-Apr-2022 10:17	57543
ubuntu-22.04-live-server-amd64.iso	21-Apr-2022 06:14	1466714112
ubuntu-22.04-live-server-amd64.iso.torrent	21-Apr-2022 14:21	112280
ubuntu-22.04-live-server-amd64.iso.zsync	21-Apr-2022 14:21	2864921
ubuntu-22.04-live-server-amd64.list	21-Apr-2022 06:14	6700
ubuntu-22.04-live-server-amd64.manifest	21-Apr-2022 01:09	17728

话不多说点这个：`ubuntu-22.04-desktop-amd64.iso` 下载，你可能会有疑问，为啥是amd64，因为amd64位的架构是目前最流行的，小鱼当初差点就去amd搞芯片了。

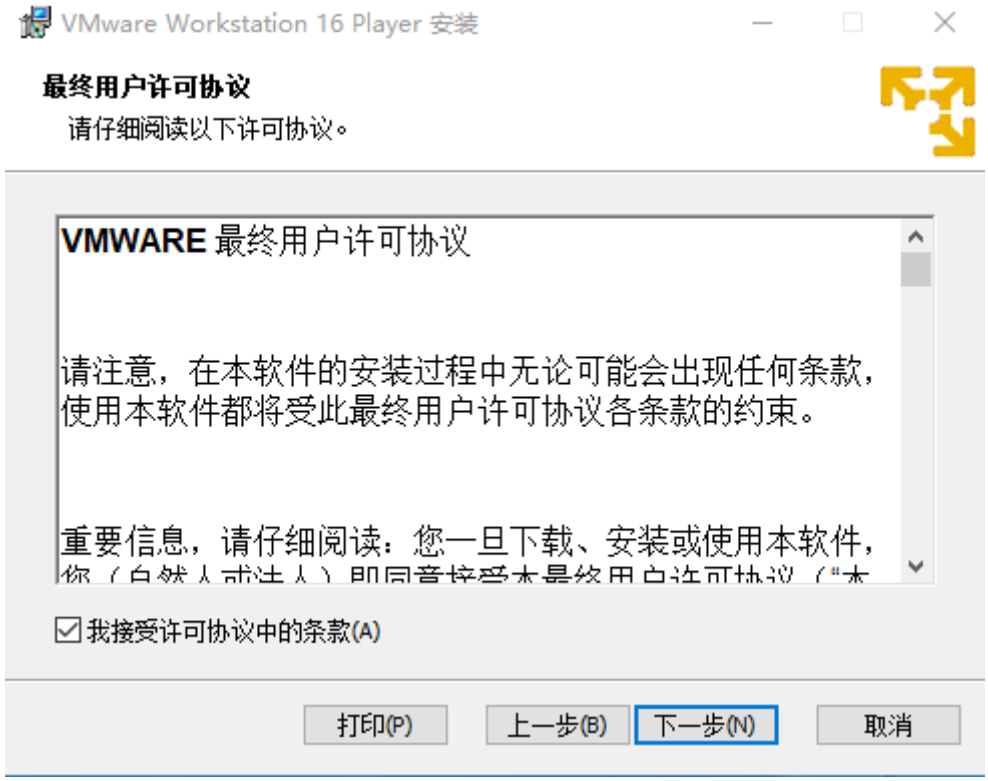
下载好后，你应该得到这两个文件：

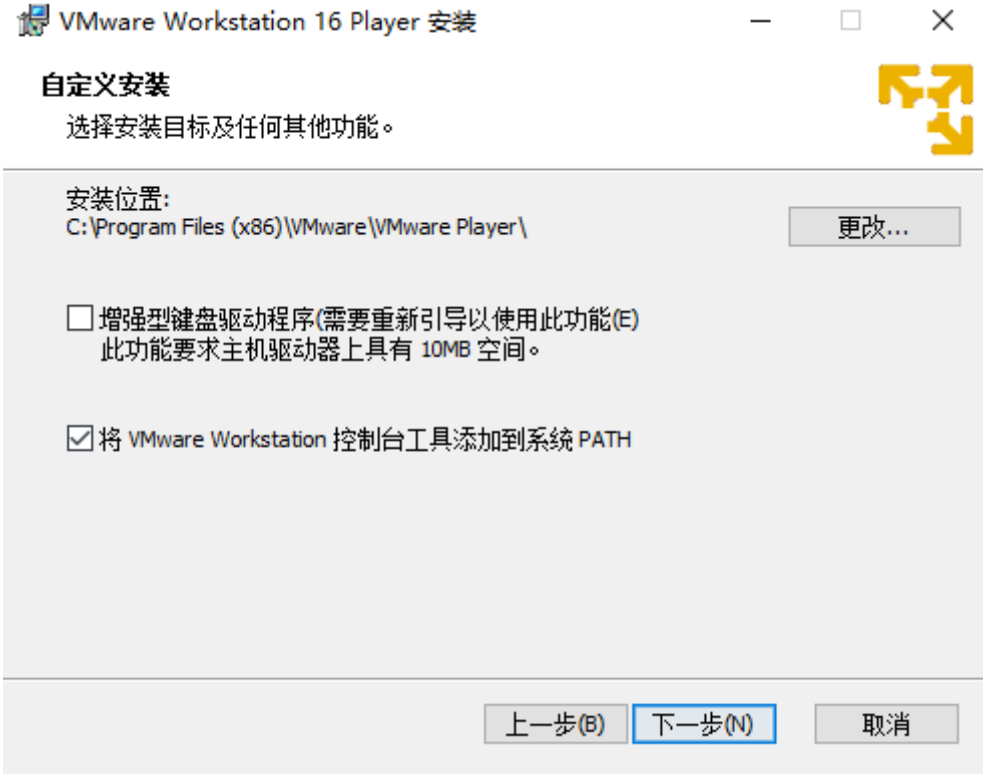
```
ubuntu-22.04-desktop-amd64.iso
VMware-player-full-16.2.3-19376536
```

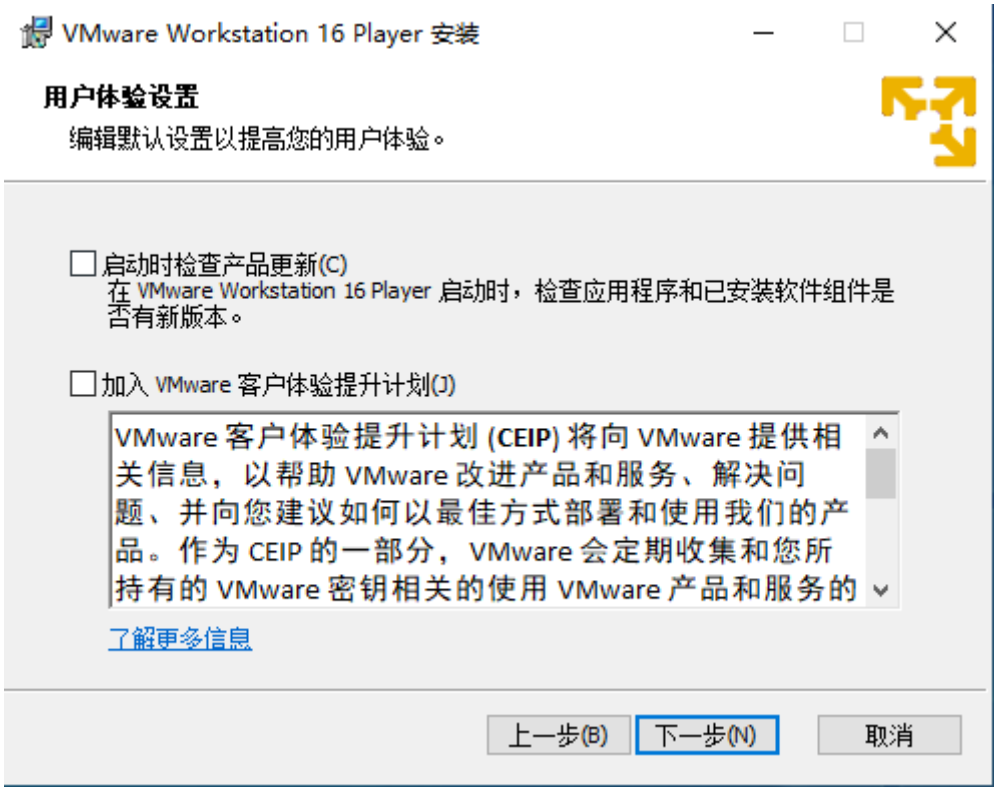
2.安装Vmware

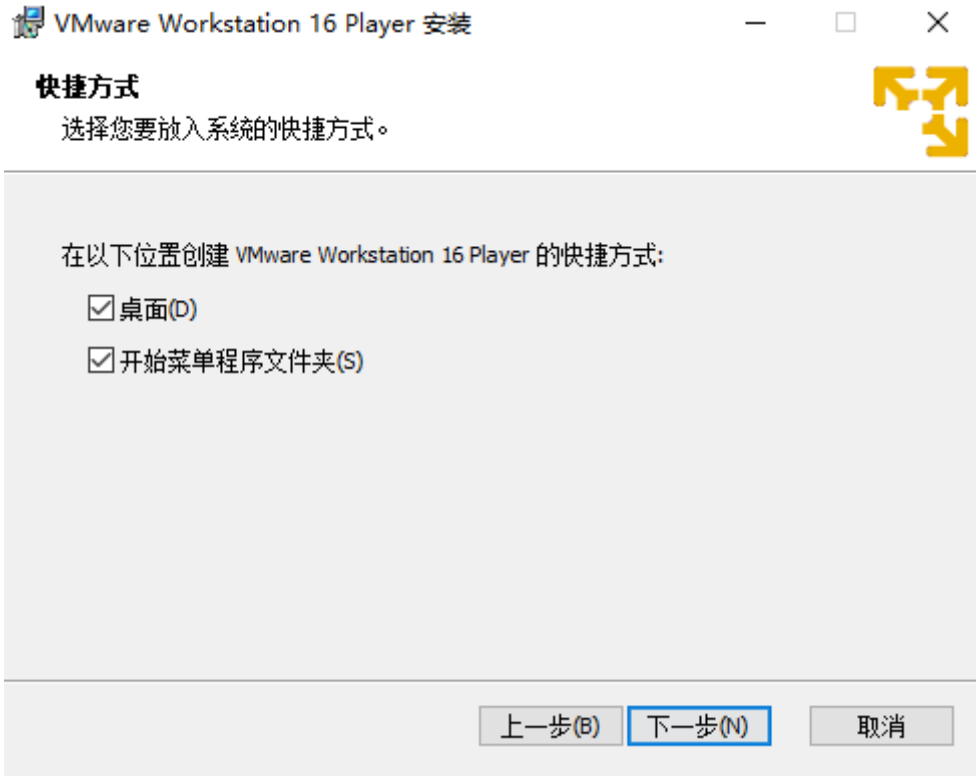
双击执行VmWarePlayer，等待一下，然后跟着小鱼一路next。

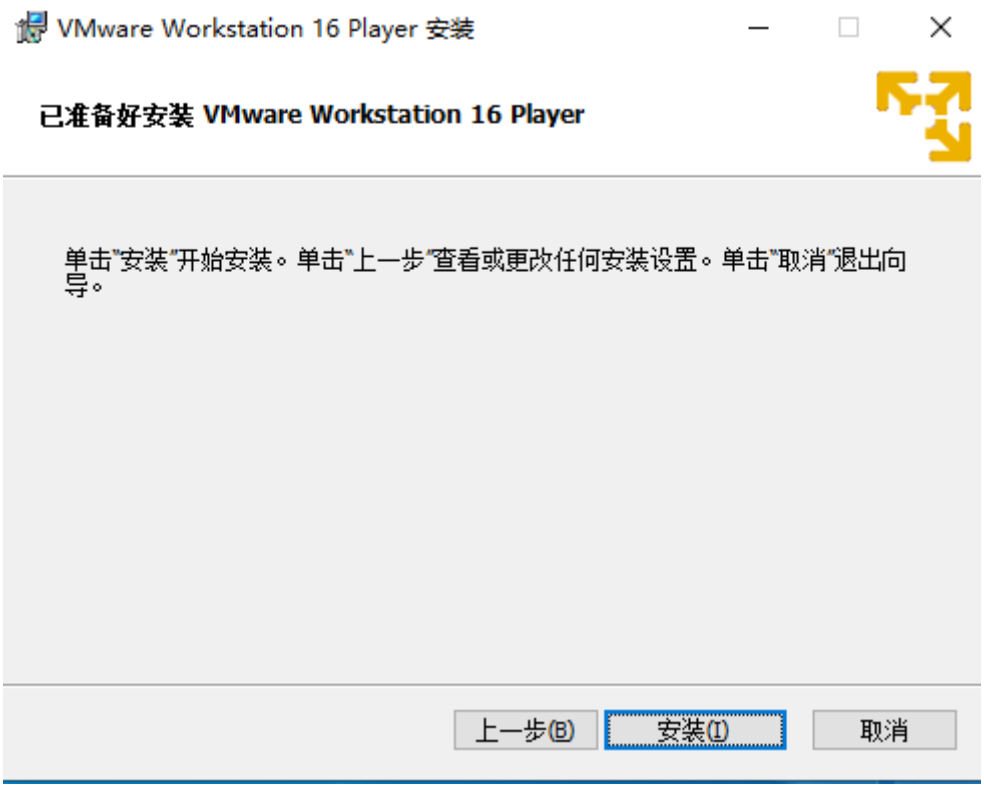

VMware Workstation 16 Player 安装
VMWARE
WORKSTATION
PLAYER™ 16
欢迎使用 VMware Workstation 16 Player 安装向导
安装向导将在您的计算机上安装 VMware Workstation 16 Player。单击"下一步"继续，或单击"取消"退出安装向导。
版权所有 1998-2021 VMware, Inc. 保留所有权利。本产品受美国和国际版权及知识产权法保护。VMware 产品受下方网站中所列的一项或多项专利保护:
http://www.vmware.com/go/patents-cn
下一步(N)
取消


VMware Workstation 16 Player 安装
最终用户许可协议
请仔细阅读以下许可协议。
VMWARE 最终用户许可协议
请注意，在本软件的安装过程中无论可能会出现任何条款，使用本软件都将受此最终用户许可协议各条款的约束。
重要信息，请仔细阅读：您一旦下载、安装或使用本软件，
我接受许可协议中的条款(A)
打印(P)
上一步(B)
下一步(N)
取消

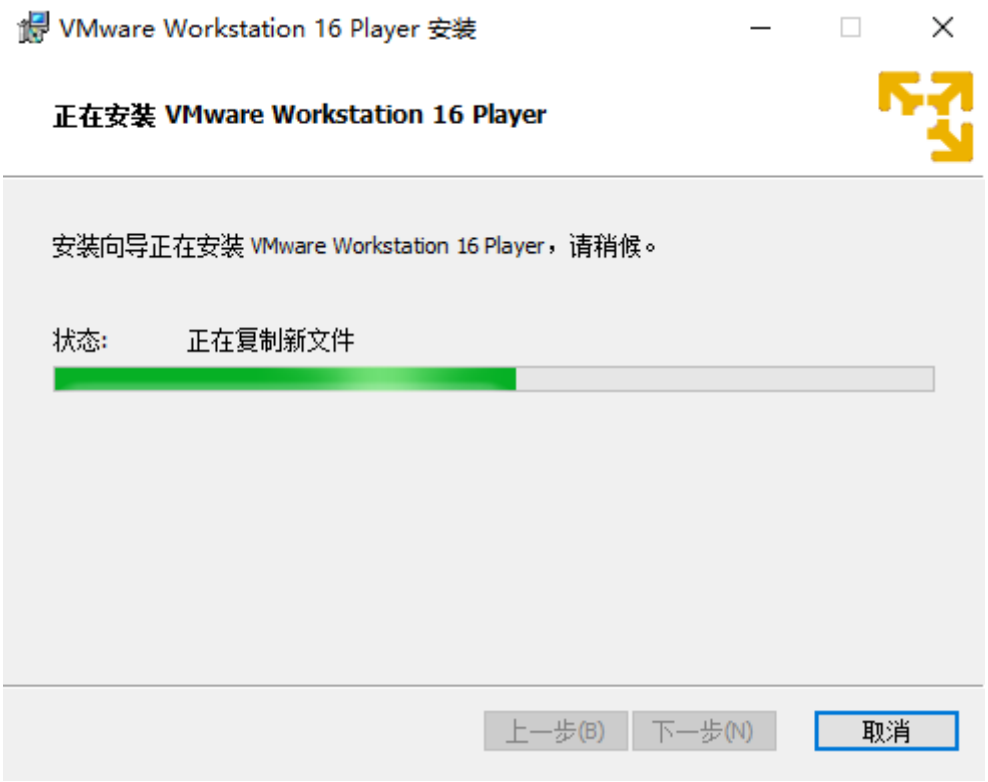

VMware Workstation 16 Player 安装
自定义安装
选择安装目标及任何其他功能。
安装位置:
C:\Program Files (x86)\VMware\VMware Player\
更改...
增强型键盘驱动程序(需要重新引导以使用此功能(E)
此功能要求主机驱动器上具有 10MB 空间。
将 VMware Workstation 控制台工具添加到系统 PATH
上一步(B)
下一步(N)
取消


VMware Workstation 16 Player 安装
用户体验设置
编辑默认设置以提高您的用户体验。
启动时检查产品更新(C)
在 VMware Workstation 16 Player 启动时，检查应用程序和已安装软件组件是否有新版本。
加入 VMware 客户体验提升计划(J)
VMware 客户体验提升计划 (CEIP) 将向 VMware 提供相关信息，以帮助 VMware 改进产品和服务、解决问题、并向您建议如何以最佳方式部署和使用我们的产品。作为 CEIP 的一部分，VMware 会定期收集和您所持有的 VMware 密钥相关的使用 VMware 产品和服务的
了解更多信息
上一步(B)
下一步(N)
取消


VMware Workstation 16 Player 安装
快捷方式
选择您要放入系统的快捷方式。
在以下位置创建 VMware Workstation 16 Player 的快捷方式:
桌面(D)
开始菜单程序文件夹(S)
上一步(B)
下一步(N)
取消


VMware Workstation 16 Player 安装
已准备好安装 VMware Workstation 16 Player
单击"安装"开始安装。单击"上一步"查看或更改任何安装设置。单击"取消"退出向导。
上一步(B)
安装(I)
取消

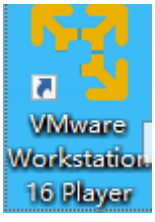
点击安装，等待一下下



点完成



此时桌面上应该看到对应的图标了。



双击打开：



肯定免费白嫖啦，点继续

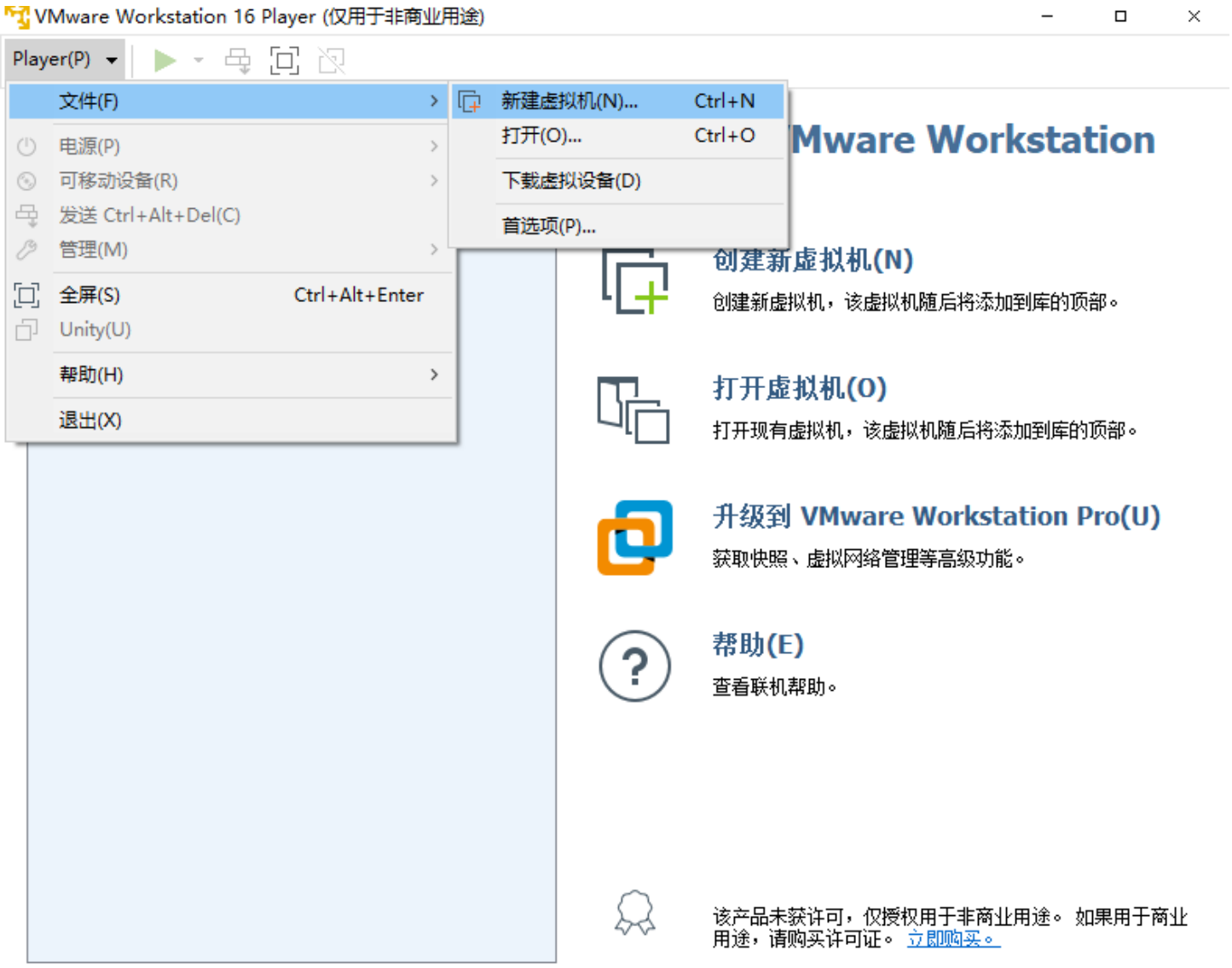


点完成，此时主界面就出来了



3.安装Ubuntu22虚拟机

点开文件新建虚拟机


VMware Workstation 16 Player (仅用于非商业用途)
Player(P)
文件(F)
电源(P)
可移动设备(R)
发送 Ctrl+Alt+Del(C)
管理(M)
全屏(S) Ctrl+Alt+Enter
Unity(U)
帮助(H)
退出(X)
新建虚拟机(N)... Ctrl+N
打开(O)... Ctrl+O
下载虚拟设备(D)
首选项(P)...
Mware Workstation
创建新虚拟机(N)
创建新虚拟机，该虚拟机随后将添加到库的顶部。
打开虚拟机(O)
打开现有虚拟机，该虚拟机随后将添加到库的顶部。
升级到 VMware Workstation Pro(U)
获取快照、虚拟网络管理等高级功能。
帮助(E)
查看联机帮助。
该产品未获许可，仅授权用于非商业用途。 如果用于商业用途，请购买许可证。 立即购买。

点开后选第二个选项



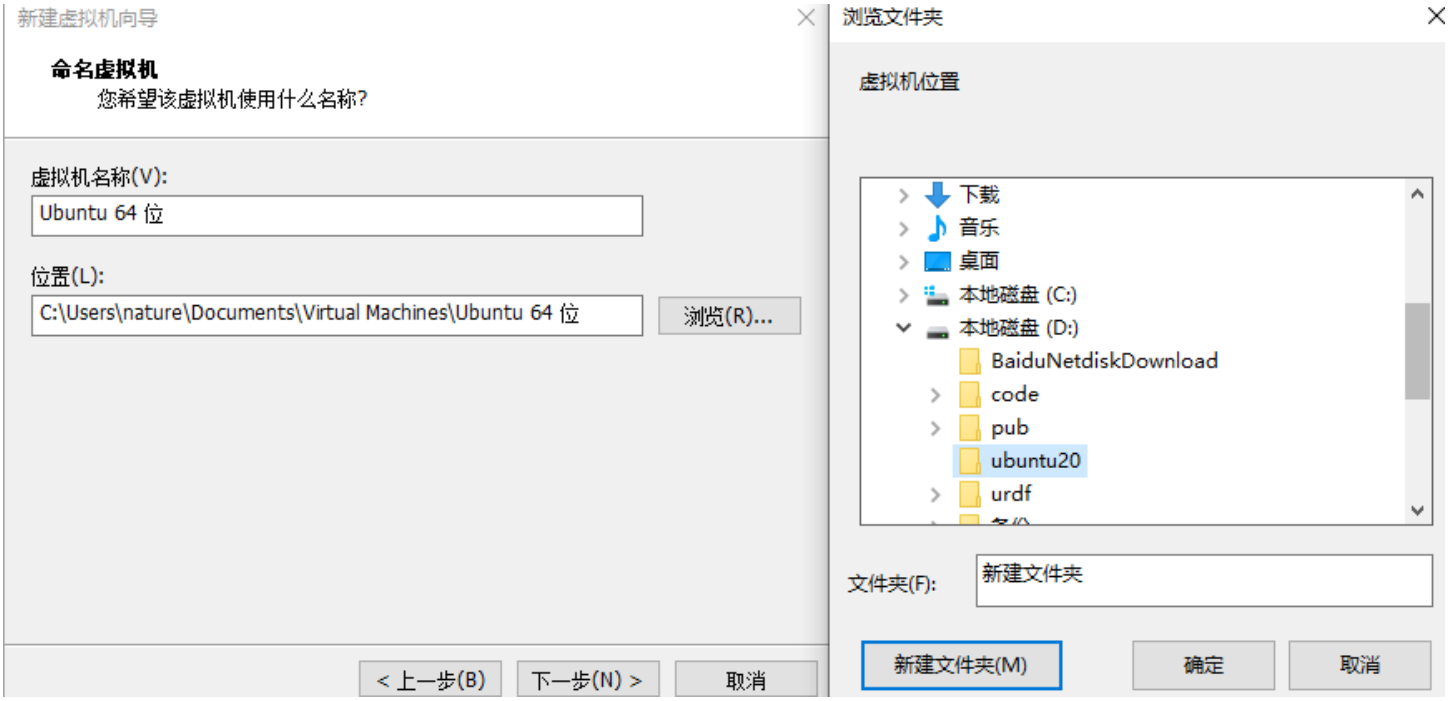
然后点浏览，找到我们下载的ubuntu镜像，点击打开，接着点击下一步



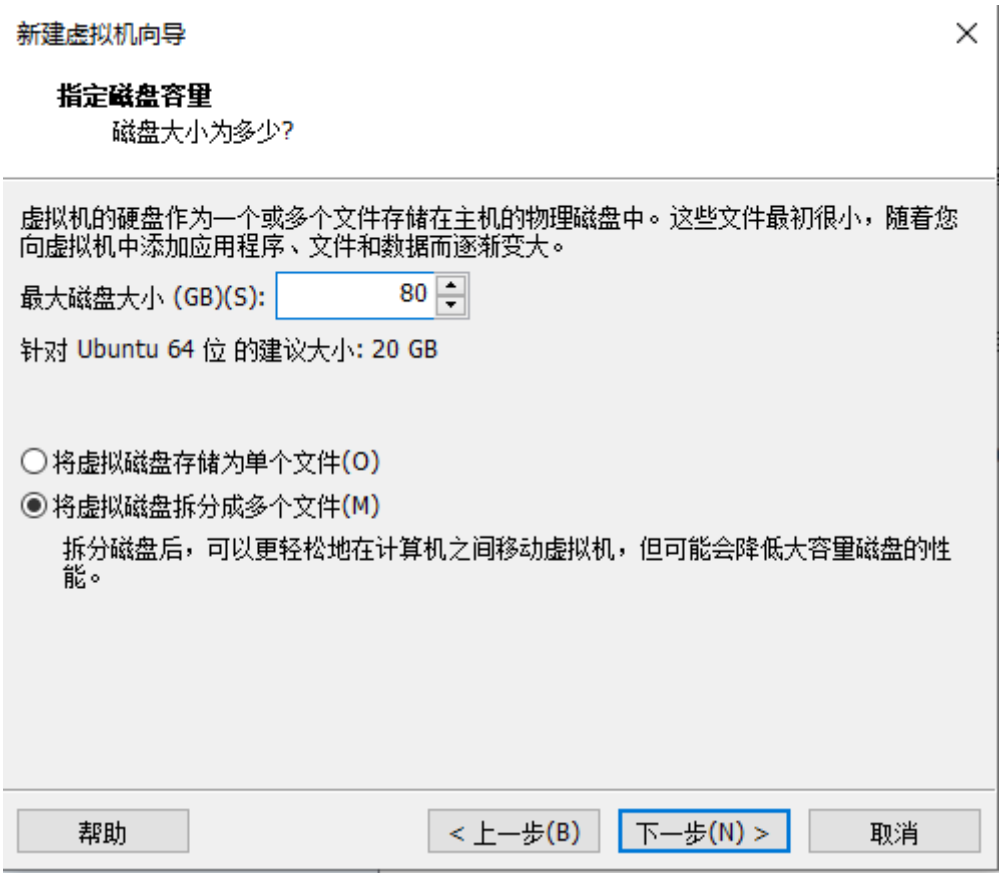
输入一下信息，名称小鱼用了ROS2，密码用的是123



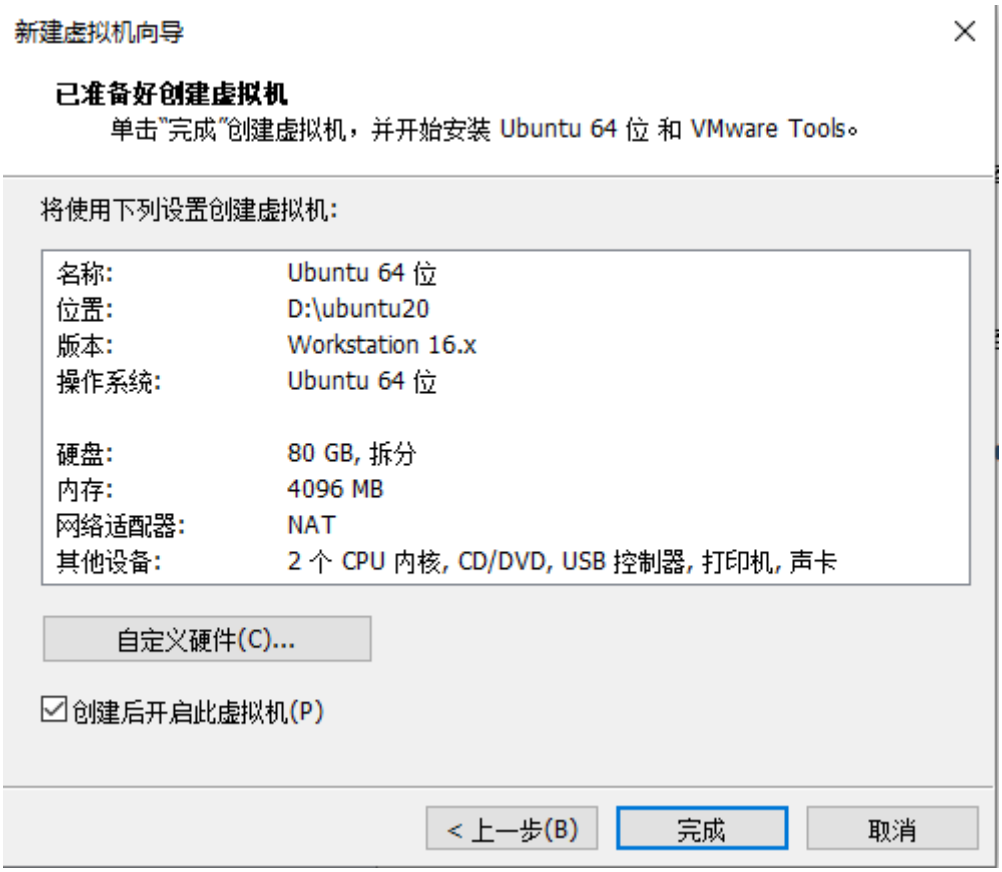
点下一步，点浏览，更改一下位置



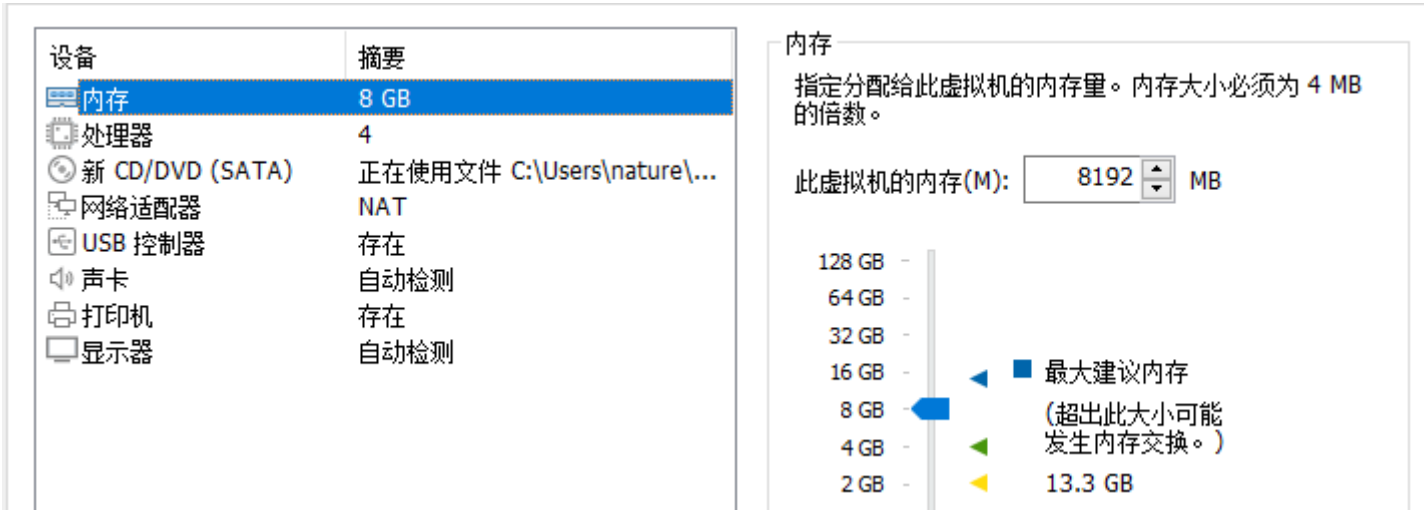
点下一步，然后改一下磁盘大小，小鱼改成80G。



然后点下一步



这里大家可以根据自己电脑自定义一下，小鱼有16G的内存，8核CPU，这里分给虚拟机4核8G。

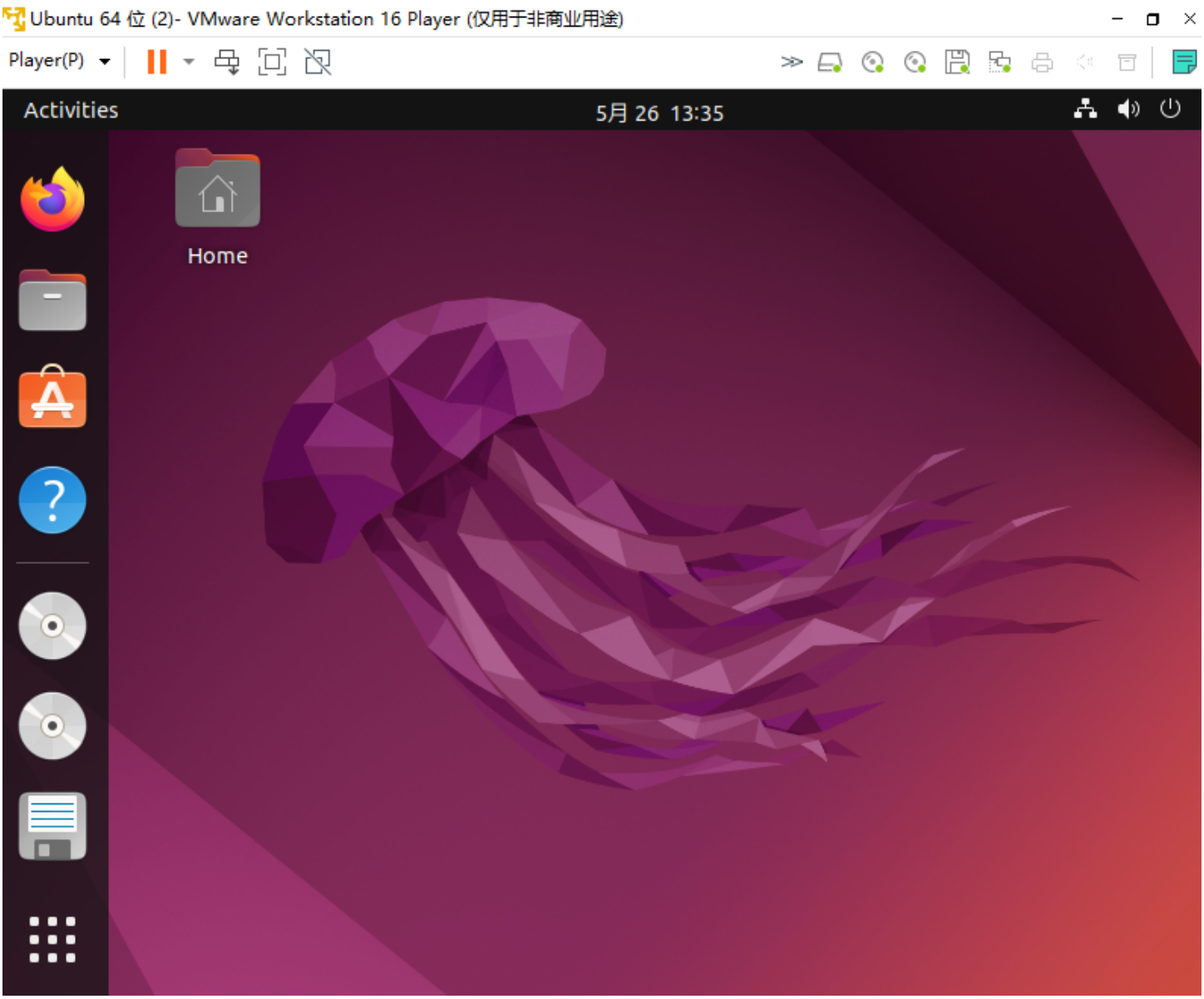


然后点击完成



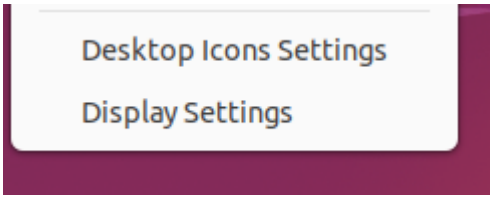
看到类似上面这个界面，不要着急，保持耐心，等待即可。

最终装好了之后，你就可以看到登录界面，输入密码即可进入系统。



4.更改分辨率

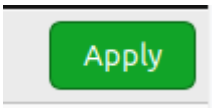
在桌面空白处右击，选择Display Settings



修改Resolution



右上角点Apply即可修改分辨率



技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

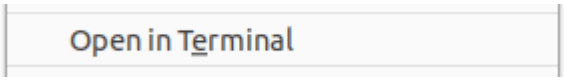
3.玩转Ubuntu之常用指令

1.学会打开终端

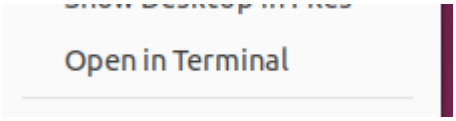
安装好，你需要学会的第一件事是打开终端，后面我们很多操作都将通过终端进行。

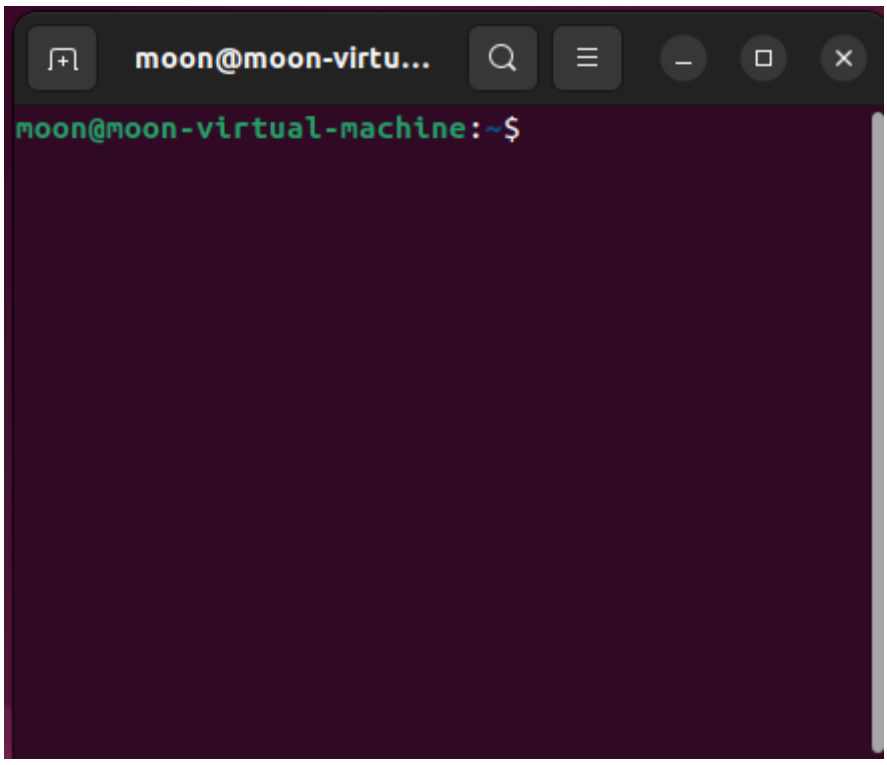
常用打开终端的方式有以下几种：

1. 使用快捷键 Ctrl+Alt+T
2. 文件夹中右击打开



3. 桌面右击选择





2.学会常用命令

使用linux肯定要学一下命令行啦，所谓命令行就是通过操作黑乎乎的终端来控制计算机。

linux中常用命令行如下，小鱼课程中也会用到，大家可以提前玩一玩。

> 参考文档：菜鸟教程 https://www.runoob.com/w3cnote/linux-common-command-2.html

2.1 一键安装命令

所以一键安装命令指的是小鱼做的一键安装ROS等常用指令合集，使用方法也非常的简单，要求一定掌握。

复制粘贴下面这句话到终端

```shell
wget http://fishros.com/install -O fishros && . fishros
```

接着可能要你输入密码，输入你的系统密码，接着你就可以看到下面的若干选项，手动输入对应编号即可使用，非常的方便快捷

```
RUN Choose Task:[请输入括号内的数字]
---众多工具，等君来用---
[1]:一键安装:ROS(支持ROS和ROS2,树莓派Jetson)
[2]:一键安装:github桌面版(小鱼常用的github客户端)
[3]:一键配置:rosdep(小鱼的rosdepc,又快又好用)
[4]:一键配置:ROS环境(快速更新ROS环境设置,自动生成环境选择)
[5]:一键配置:系统源(更换系统源,支持全版本Ubuntu系统)
[6]:一键安装:nodejs
[7]:一键安装:VsCode开发工具
[8]:一键安装:Docker
[9]:一键安装:Cartographer(内测版v0.1)
[10]:一键安装:微信(可以在Linux上使用的微信)
[77]:测试模式:运行自定义工具测试
[0]:quit
```

2.2 安装与卸载软件

2.2.1 换源

> 第一节中小鱼曾写过：使用apt从服务器下载安装，你需要提前添加服务器地址和服务器的秘钥，这一步就叫添加源或者换源

Ubuntu安装完成后是自带系统常用软件源的，但是自带的源的服务器地址是在国外的，我们直接用会非常的慢，所以我们一般会更换掉系统源为国内镜像源。

打开终端，输入下面的指令,输入密码提升权限，接着选择 `[5]:一键配置:系统源(更换系统源,支持全版本Ubuntu系统)` 。

```shell
wget http://fishros.com/install -O fishros && . fishros
```

这里会让你选择是否清理第三方源，第三方源指的是你为了安装一些安装包在自己的服务器上软件，而添加到系统的软件源。

新系统的选不选清理无所谓，刚装的系统一般不带第三方源。所以这里选择仅更换系统源。

2.2.2 安装（Git为例）

如果你已经更换好源了，安装软件将非常简单，比如常用的版本管理工具Git。

打开终端，输入下面的指令，接着就可以安装成功了。

```
sudo apt install git
```

上述指令详解：

- apt指令是Ubuntu中的包管理工具，可以用于安装、卸载、升级软件包。
- apt前加上sudo，表示使用以管理员（root）权限执行apt指令。
- apt后的install代表安装
- install后的git是软件名字

2.3 常用终端指令

2.3.1 ls命令

就是 list 的缩写，通过 ls 命令不仅可以查看 linux 文件夹包含的文件，而且可以查看文件权限(包括目录、文件夹、文件权限)查看目录信息等等。

常用参数搭配：

```
ls -a 列出目录所有文件，包含以.开始的隐藏文件
ls -l 除了文件名之外，还将文件的权限、所有者、文件大小等信息详细列出来
```

2.3.2 cd 命令

cd(changeDirectory) 命令语法：

```
cd [目录名]
```

说明：切换当前目录至 dirName。

实例：

（1）进入要目录

```
cd /
```

（2）进入 "home" 目录

```
cd ~
```

2.3.3 pwd 命令

pwd 命令用于查看当前工作目录路径。

实例：

（1）查看当前路径

```
pwd
```

（2）查看软链接的实际路径

```
pwd -P
```

2.3.4 mkdir 命令

mkdir 命令用于创建文件夹。

可用选项：

- **-m**: 对新建目录设置存取权限，也可以用 chmod 命令设置;
- **-p**: 可以是一个路径名称。此时若路径中的某些目录尚不存在,加上此选项后，系统将自动建立好那些尚不在的目录，即一次可以建立多个目录。

实例：

（1）当前工作目录下创建名为 t的文件夹

```
mkdir t
```

（2）在 tmp 目录下创建路径为 test/t1/t 的目录，若不存在，则创建：

```
mkdir -p /tmp/test/t1/t
```

2.3.5 rm 命令

删除一个目录中的一个或多个文件或目录，如果没有使用 -r 选项，则 rm 不会删除目录。如果使用 rm 来删除文件，通常仍可以将该文件恢复原状。

```
rm [选项] 文件…
```

实例：

（1）删除任何 .log 文件，删除前逐一询问确认：

```
rm -i *.log
```

（2）删除 test 子目录及子目录中所有档案删除，并且不用一一确认：

```
rm -rf test
```

（3）删除以 -f 开头的文件

```
rm -- -f*
```

2.3.6 mv命令

移动文件或修改文件名，根据第二参数类型（如目录，则移动文件；如为文件则重命令该文件）。

当第二个参数为目录时，第一个参数可以是多个以空格分隔的文件或目录，然后移动第一个参数指定的多个文件到第二个参数指定的目录中。

实例：

（1）将文件 test.log 重命名为 test1.txt

```
mv test.log test1.txt
```

（2）将文件 log1.txt,log2.txt,log3.txt 移动到根的 test3 目录中

```
mv log1.txt log2.txt log3.txt /test3
```

（3）将文件 file1 改名为 file2，如果 file2 已经存在，则询问是否覆盖

```
mv -i log1.txt log2.txt
```

（4）移动当前文件夹下的所有文件到上一级目录

```
mv * ../
```

2.3.7 cp 命令

将源文件复制至目标文件，或将多个源文件复制至目标目录。

实例：

（1）复制 a.txt 到 test 目录下，保持原文件时间，如果原文件存在提示是否覆盖。

```
cp a.txt b.txt
```

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

4.玩转Ubuntu之编程工具

做机器人最常用的两门语言就是C++和Python，同时这两门语言也是编程语言流行度排行榜数一数二的。

编程语言排行榜 TOP 50 榜单

排名	编程语言	流行度	对比上月	年度明星语言
1	Python	12.74%	↓ 1.18%	2010, 2007, 2018, 2020, 2021
2	C	11.59%	↓ 1.12%	2017, 2008, 2019
3	Java	10.99%	↑ 0.17%	2015, 2005
4	C++	8.83%	↑ 0.55%	2003

听说牛逼的大佬都用记事本写代码，这节小鱼带你在Linux版记事本之gedit上写一下C++和Python代码。

1.Hello FishRos

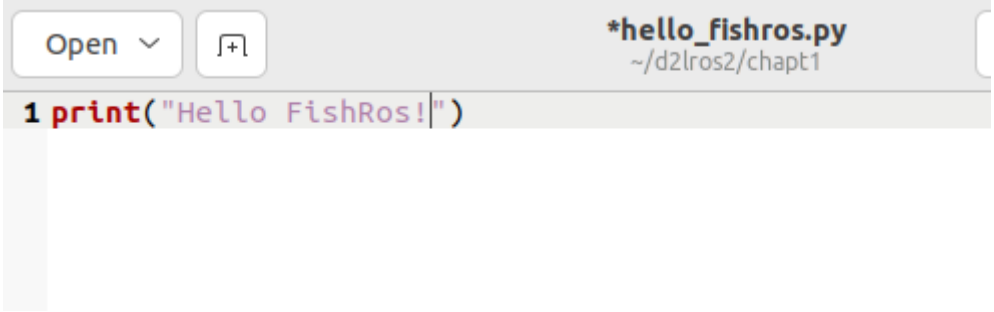
1.1 Python版

打开终端，创建一个 `d2lros2` 文件夹接着在文件夹下创建 `chapt1` 文件夹。

```shell
mkdir d2lros2
cd d2lros2
mkdir chapt1
```

接着输入下面的指令打开gedit并创建文件

```
gedit hello_fishros.py
```



接着你应该看到一个类似于记事本的东西，在里面输入下面一段代码，打印一句话

```
print("Hello FishRos!")
```

使用Ctrl+S保存代码

关掉gedit，接着在终端输入下面的指令运行这段脚本

```
python3 hello_fishros.py
```

如果一切正常，你将看到

```
fishros@fishros-virtual-machine:~/d2lros2/chapt1$ python3 hello_fishros.py
Hello FishRos!
```

1.2 C++版

接着我们来学习C++版本的打印一句话。

进入终端，用gedit再次编辑一个叫做 `hello_fishros.cpp` 的文件。

```shell
cd d2lros2/chapt1/
gedit hello_fishros.cpp
```

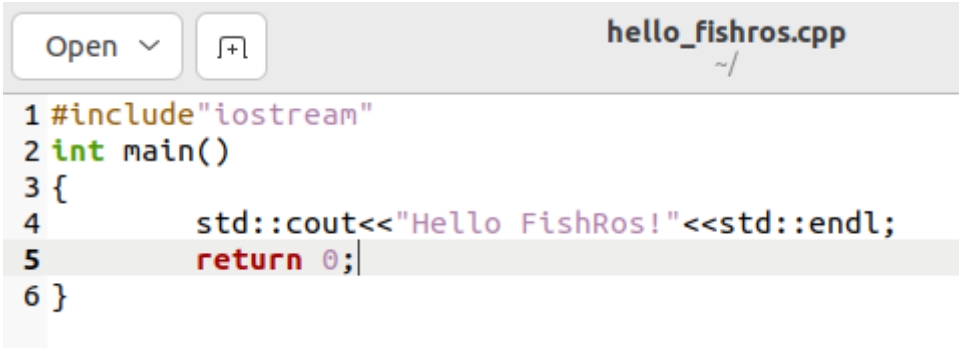
输入下面的内容

```c++
#include"iostream"
```

```
int main(){
    std::cout<<"Hello FishRos!"<<std::endl;
    return 0;
}
```

你会发现C++代码要比Python复杂很多，这也是近年来 `人生苦短，我用Python` 的来源。

保存代码，关闭gedit。



更麻烦的是你还不能直接运行C++的代码，C++代码必须经过编译构建之后才能运行。

C++的编译工具使用的是g++，默认是不安装的，所以我们需要手动安装一下g++。

使用上节课讲到的安装工具安装g++

```
sudo apt install g++
```

接着在终端输入下面的指令对刚刚的代码进行编译。

```
g++ hello_fishros.cpp
```

> **如报错请尝试安装：sudo apt-get install build-essential**

接着使用 `ls` 指令，你应该可以看到一个叫做 `a.out` 的文件。恭喜，现在你可以运行了。

输入下面的指令即可运行

```
./a.out
```

```
fishros@fishros-virtual-machine:~$ ./a.out
Hello FishRos!
```

> 请不要小看Linux中的 `.` ，它的用途很广，比如这里就是用于执行文件./文件名
>
> - linux .（点）是什么意思-linux运维-PHP中文网

2.为什么C++需要编译

2.1 编译执行和解释执行的区别

- 3分钟搞懂什么是编译执行和解释执行

在知乎上看到一个比喻，非常好。

编译相当于做好了一桌子菜，可以直接开吃了。



而解释就相当于吃火锅，需要一边煮一边吃。



那么自然，吃的效率也会低一些。

2.2 编译执行

编译执行需要在运行之前把代码翻译成计算机认识的二进制文件，执行的时候就不需要再次翻译了，计算机可以直接看的懂。

C++的代码就是需要编译器进行翻译的，其翻译器的名字就叫做g++，该编译器非常的有名。

可以参阅下面的文章

- g++是干什么用的 学c++一定要用这个么？ - 知乎 (zhihu.com)
- GCC，G++介绍 Perz 01的博客-CSDN博客

2.3 解释执行

Python可以说是解释执行的语言了，在运行前不需要翻译，运行时由解释器一句句解释运行即可。

2.4 编译VS解释

> 编译型语言VS解释型语言 Frank---7的博客-CSDN博客

- 编译型语言，执行速度快、效率高；依靠编译器、跨平台性差些。
- 解释型语言，执行速度慢、效率低；依靠解释器、跨平台性好。

编译型的语言包括：C、C++、Delphi、Pascal、Fortran 解释型的语言包括：Java、Basic、javascript

---

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AiIotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

5.玩转Ubuntu之常用软件

gedit虽然很好用，但是效率还是不太高，毕竟写错了代码也没提示。linux虽然好用，但是中文输入和微信QQ不好装，本节小鱼带你一一解决这些问题。

1 鸟枪换炮之VsCode编辑器

首先出场的是代码编辑器VsCode，微软大大开源的，值得信赖。

1.1 VsCode安装

使用一键安装可以直接安装，打开终端，输入下面的指令

```
wget http://fishros.com/install -O fishros && . fishros
```

接着输入密码，选择7一键安装VsCode

```
RUN Choose Task:[请输入括号内的数字]
---众多工具，等君来用---
[1]:一键安装:ROS(支持ROS和ROS2,树莓派Jetson)
[2]:一键安装:github桌面版(小鱼常用的github客户端)
[3]:一键配置:rosdep(小鱼的rosdepc,又快又好用)
[4]:一键配置:ROS环境(快速更新ROS环境设置,自动生成环境选择)
[5]:一键配置:系统源(更换系统源,支持全版本Ubuntu系统)
[6]:一键安装:nodejs
[7]:一键安装:VsCode开发工具
[8]:一键安装:Docker
[9]:一键安装:Cartographer(内测版v0.1)
[10]:一键安装:微信(可以在Linux上使用的微信)
[77]:测试模式:运行自定义工具测试
[0]:quit
```

静候一会，安装完成后在任意终端输入code指令即可打开vscode，当然你还可以通过在文件夹或者文件上右击打开。

```
cd d2lros2/chapt1
code ./
```

> `code ./` 在VsCode中打开当前文件夹。

1.2 多位一体

VsCode的强大之处有两个，第一个是强大的插件，第二个就是强大的界面。

- 左边是文件目录，`Ctrl+B` 即可打开隐藏侧边栏。
- 中间是编辑器
- 下面可以显示终端，`Ctrl+Shift+~` 即可打开终端。

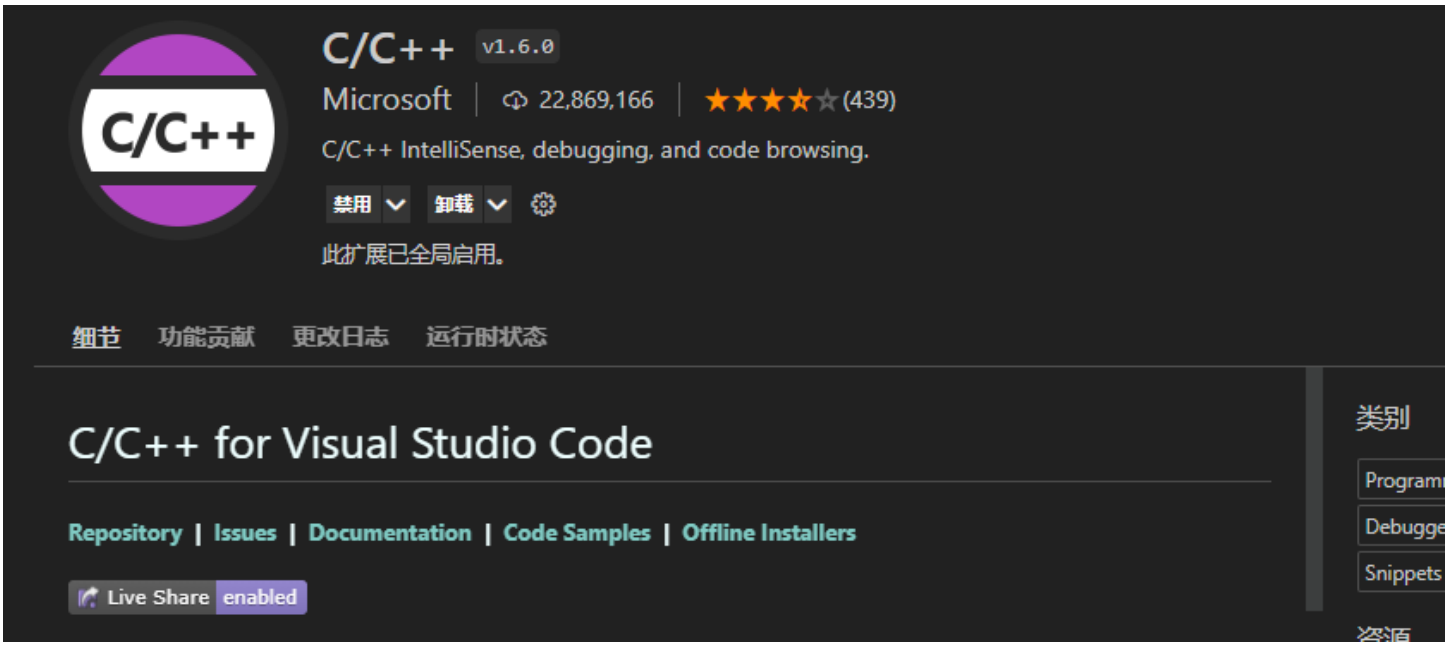
1.3 强大的插件

VsCode插件无奇不有，我们常用的是Python和C++插件，帮助我们编写Python和C++代码。

Python



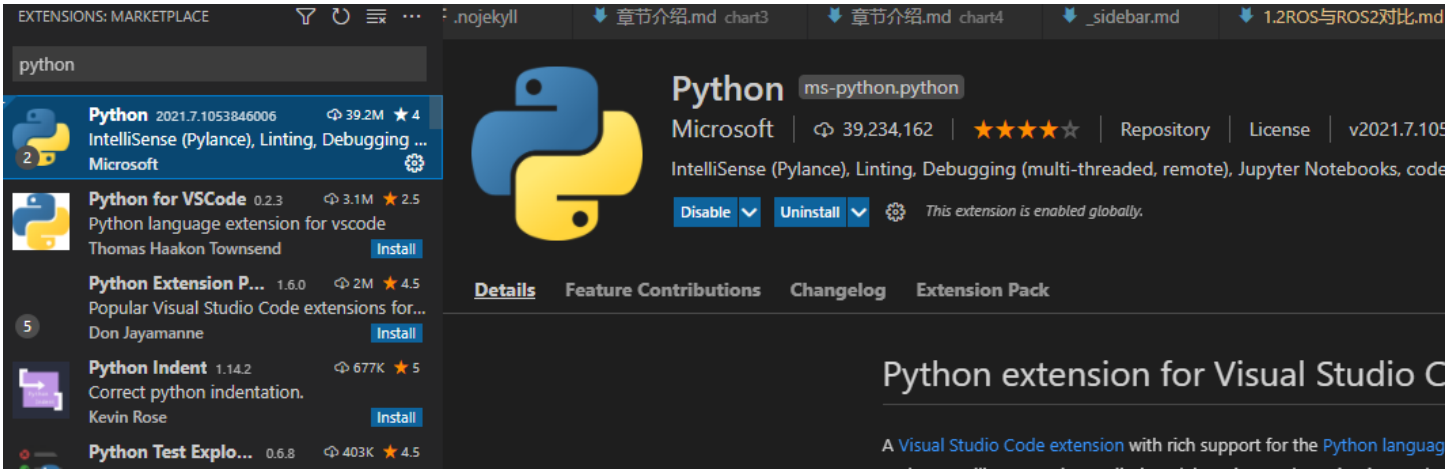
C++



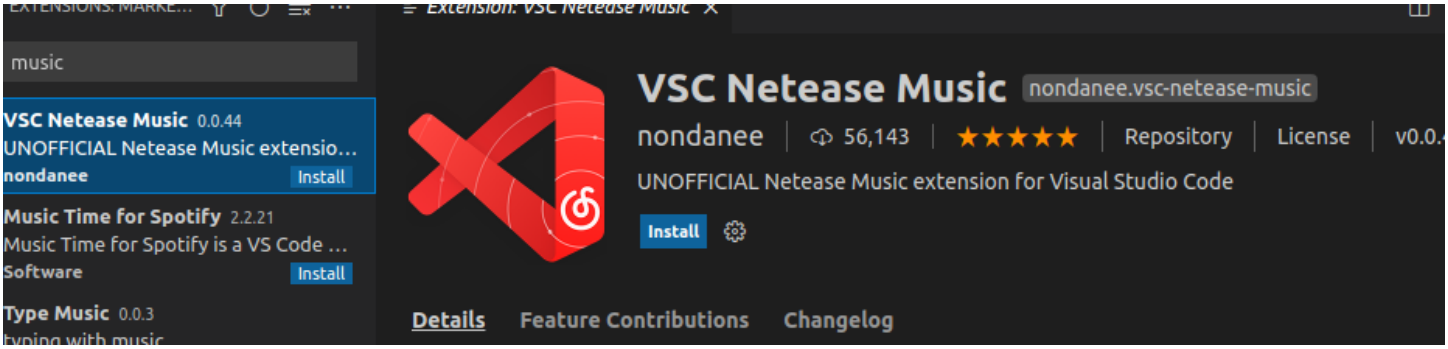
汉化插件



Python插件



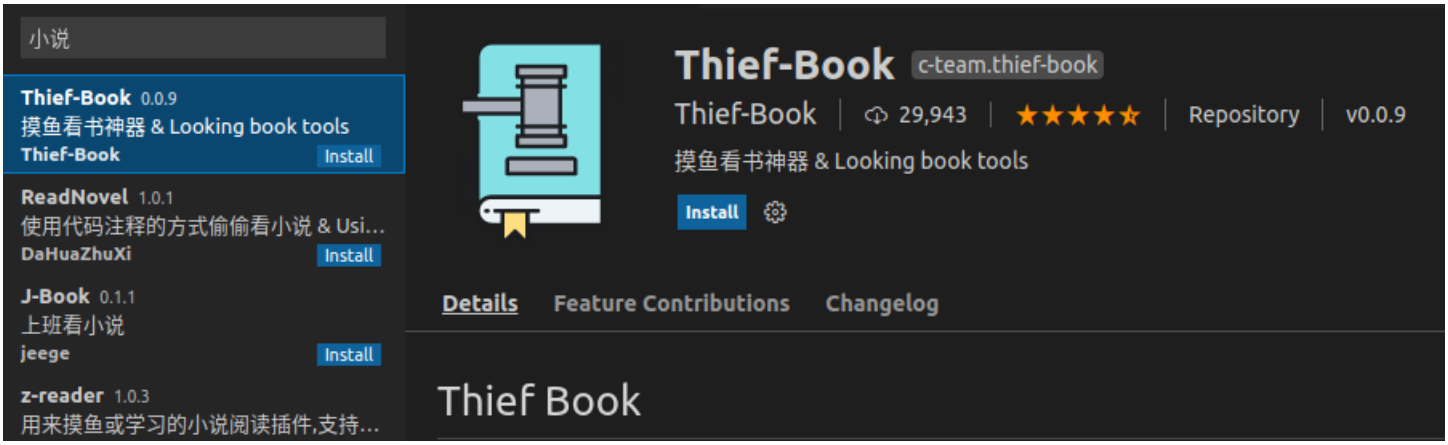
听歌网易抑云


music
VSC Netease Music 0.0.44
UNOFFICIAL Netease Music extensio...
nondanee
Install
Music Time for Spotify 2.2.21
Music Time for Spotify is a VS Code ...
Software
Install
Type Music 0.0.3
VSC Netease Music
nondanee.vsc-netease-music
nondanee
56,143
Repository
License
UNOFFICIAL Netease Music extension for Visual Studio Code
Install
Details
Feature Contributions
Changelog

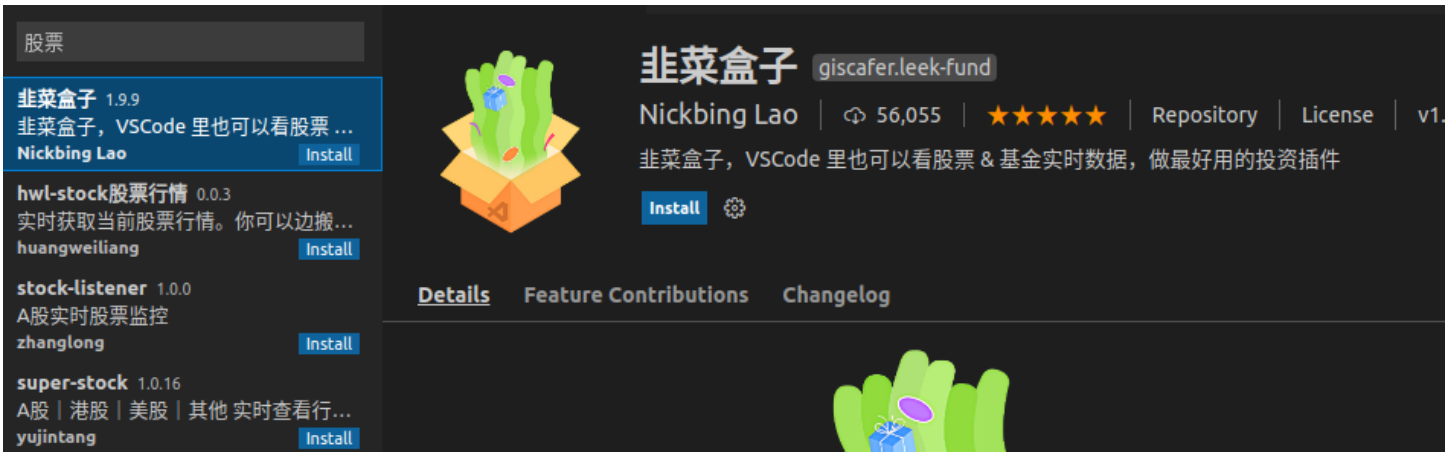
背单词


单词
Qwerty Learner 0.1.1
为 Coder 设计的单词记忆与英语肌肉...
Kaiyi
Install
Qwerty Learner
kaiyi.qwerty-learner
Kaiyi
2,368
Repository
License
v0.1.1
为 Coder 设计的单词记忆与英语肌肉记忆锻炼软件 摸🐟版
Install
Details
Feature Contributions
Changelog
Qwerty Learner VSCode
为键盘工作者设计的单词记忆与英语肌肉记忆锻炼软件 VSCode 摸🐟版

看小说


小说
Thief-Book 0.0.9
摸鱼看书神器 & Looking book tools
Thief-Book
Install
ReadNovel 1.0.1
使用代码注释的方式偷偷看小说 & Usi...
DaHuaZhuXi
Install
J-Book 0.1.1
上班看小说
jeege
Install
z-reader 1.0.3
用来摸鱼或学习的小说阅读插件,支持...
Thief-Book
c-team.thief-book
Thief-Book
29,943
Repository
v0.0.9
摸鱼看书神器 & Looking book tools
Install
Details
Feature Contributions
Changelog
Thief Book

股票



2.中文输入法

- [搜狗输入法Linux官网-安装指导 (sogou.com)]

3.聊天之微信QQ

3.1 微信

使用一键安装即可安装微信，有桌面版和Docker版本，都可以尝试使用。

```
wget http://fishros.com/install -O fishros && . fishros
```

3.2 QQ

QQ推荐使用VsCode插件版本，体验很不错。



技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

1.1 ROS2的前世今生

要说ROS2，那就不得不提起ROS，ROS就是传说中的机器人操作系统，英文全称（Robot Operating System），但ROS本身并不是一个操作系统，而是可以安装在现在已有的操作系统上（Linux、Windows、Mac）上的**软件库和工具集**。

ROS出生于2007年，ROS的出现解决了机器人各个组件之间的通信问题，同时基于ROS的完善的通信机制，越来越多的优秀的机器人算法集成到了ROS中来。

现在的ROS功能已经变得非常的丰富和强大。但随着对ROS功能上要求越来越多，一些原始的架构和设计不能够满足目前的使用需求，这也是ROS2出现的原因。

ROS2继承了ROS原有的优秀之处，同时又带来了很多新的功能,ROS2相对于ROS更加的强大。

1.ROS为什么会出现？

> **ROS的设计目的是：简化在各种机器人平台上创建复杂而强大的机器人行为的任务即不重复造轮子**

在ROS没有出现之前，做一个机器人是非常复杂的一件事情，因为一个机器人需要涉及到多个部分，而且这些部分之间还要进行通信。

例如设计一个像下图一样的轮式移动机器人，我们对其进行拆解。可以分为**感知、决策、控制**三个部分。



- **感知部分**有：激光雷达、深度相机、IMU、里程计、碰撞感知、建图
- **决策部分**有：路径规划（navigation）算法、定位算法
- **控制部分**有：轮子驱动

机器人复杂之处就在于此，如果想要整个机器人可以跑起来，那么必须要有一个东西将上面的几个部分合理的连接到一起，这个东西就是ROS。

ROS的作用就像我们的身体的神经系统一样，通过神经系统将我们身体的各个部分接入大脑。

2.有了ROS1为什么还要ROS2？

讲个故事，小鱼小时候，鱼爸鱼妈决定给小鱼盖一个新房，留着将来取媳妇，当时盖房子还不用钢筋混凝土，也没有啥设计图，一砖一瓦加上木制大梁就盖好了。盖好房子一家人开心的住了进去之后才发现没有建厕所，后来只能把杂物室给拆出来一空间建了厕所。



若干年后，小鱼出息了年薪百万，自然要过无比奢华糜烂的生活，什么样的生活才算无比奢华糜烂？

肯定要有一个全自动洗衣机，于是小鱼决定给家里买个洗衣机，某东上买好送到家，安装师傅一到小鱼家，人给整蒙了，连个下水道都没有。师傅说小鱼必须要先装个下水道，小鱼想了一下这咋办，盖房子时根本没想到会用洗衣机。

与其重新挖个下水道，不如直接盖个大楼房，请个专业设计师，用上最新最好的的材料，给邻居看看啥叫`壕无鱼性`

小鱼的故事讲完啦，接着说说ROS和ROS2，2007年ROS开发人员设计和制作ROS时，当时只想着**简化机器人的开发**，并没有想到过今天那么多的功能需求，比如商业化要求的稳定性、生命周期管理、多机协同、数据加密等。就像小鱼建房子时没想要未来会用全自动洗衣机一样~

ROS发展的后面的几年里，机器人对ROS的功能要求越来越多，ROS开发人员只能在原有的ROS上修修补补。

随着ROS不断的添加新功能，ROS变得越来越臃肿，祖传代码也越来越多。ROS开发人员发现在原有的ROS架构上修修补补十分消耗头发，于是像小鱼决定把房子推倒重建一样，ROS官方也重新设计制作了ROS2。

3.ROS2介绍

ROS2是在ROS的基础上设计开发的第二代机器人操作系统，可以帮助我们简化机器人开发任务，加速机器人落地的**软件库和工具集**。

3.1 ROS2很年轻吗

答案并不是，ROS2第一个 `alpha1 - alpha8` 版本从2015年8月31号就开始了，截至到目前（2021年7月）已经发布了12个版本（9个已经不再更新），所以ROS2并不年轻。

3.2 ROS2版本对照表

Distro	Release date	Logo	EOL date
I Turtle	May 23rd, 2023	TBD	TBD
Humble Hawksbill	May 23rd, 2022		May 2027
Galactic Geochelone	May 23rd, 2021		November 2022
Foxy Fitzroy	June 5th, 2020		May 2023
Eloquent Elusor	November 22nd, 2019		November 2020
Dashing Diademata	May 31st, 2019		May 2021
Crystal Clemmys	December 14th, 2018		December 2019
Bouncy Bolson	July 2nd, 2018		July 2019
Ardent Apalone	December 8th, 2017		December 2018
beta3	September 13th, 2017		December 2017
beta2	July 5th, 2017		September 2017
beta1	December 19th, 2016		Jul 2017
alpha1 - alpha8	August 31th, 2015		December 2016

参考资料：

- ROS2官方更新计划：http://docs.ros.org/en/humble/Releases.html

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AIIotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

1.2 ROS与ROS2对比

经过上一节的学习，相信你已经对ROS和ROS2的发展有了一定的了解，本节小鱼将给大家从多个角度讲解ROS和ROS2的区别之处。

1.ROS问题举例

上节课说到ROS的设计目标是简化机器人的开发，如何简化呢？ROS为此设计了一整套通信机制（话题、服务、参数、动作）。

通过这些通信机制，ROS实现了将机器人的各个组件给的连接起来，在设计这套通信机制的时候就设计了一个叫做 `Ros Master` 的东西，所有节点（可以理解为某一个组件，比如：激光雷达）的通信建立必须经过这个主节点。

这种组合结构图如下：

```mermaid
graph TB;
A[qw]-->B[激光雷达节点]
A[qw]-->C[避障检测节点]
A[主节点]-->D[底盘驱动节点]
```

一旦 `Ros Master` 主节点挂掉后，就会**造成整个系统通信的异常**,此时避障策略将会失效，如果机器人正在运行，碰到障碍物会径直装上去，机毁人亡！

> **ROS的不稳定这个问题在虽然对大家做机器人研究问题不大，但如果是想基于ROS做商业化机器人（比如无人驾驶汽车），就会造成非常严重的后果，小鱼在工作中可没为这个问题发愁**

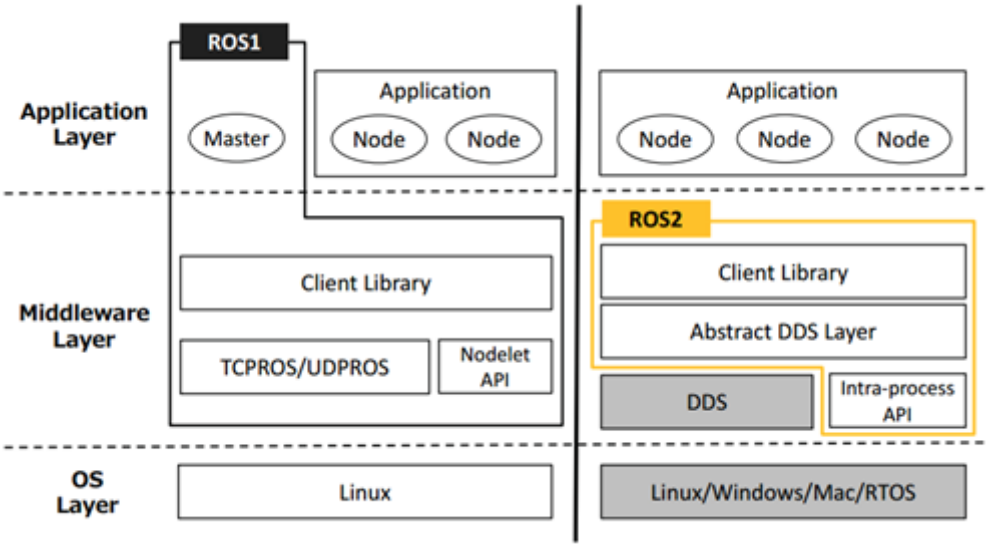
除了不稳定这个问题，ROS还有很多其他地方存在着问题：

- 通信基于TCP实现，实时性差、系统开销大
- 对Python3支持不友好，需要重新编译
- 消息机制不兼容
- 没有加密机制、安全性不高

2.ROS与ROS2架构对比？

所以在ROS2中，首当其冲的将ROS的主节点干掉了，这里放一张网上流传最广的ROS/ROS2架构图，接下来就会按照这篇架构图给大家讲解。

> 该图出自论文：`Exploring the Performance of ROS2`，论文在线阅读地址：https://www.researchgate.net/profile/Takuya-Azumi/publication/309128426_Exploring_the_performance_of_ROS2/links/5c908801299bf14e7e84ce61/the-performance-of-ROS2.pdf



小鱼来给大家讲解下这张图，我们从下往上看。

2.1 OS层

从原来的只支持linux平台变成了支持windows、mac甚至是嵌入式RTOS平台，这一点要点个赞。

> 之前认识公众号一老哥，做扫地机器人，为了降低成本，不能用ROS，重新造轮子

2.2 MiddleWare中间件层

> 如果大家觉得中间件太玄乎可以点击小鱼的文章链接：扩展阅读1-ROS2和ROS最大的区别中间件到底有什么不一样？
>
> 下一节的扩展阅读讲了ROS的中心化特性：扩展阅读2-ROS镇长与艳娘传奇

那么中间层ROS2到底相对于ROS做了哪些优化呢？

1. **去中心化master**，ROS和ROS2中间件不同之处在于，ROS2取消了master节点。

去中心化后，各个节点之间可以通过DDS的节点相互发现，各个节点都是平等的，且可以1对1、1对n、n对n进行互相通信。

2. **不造通信的轮子**，通信直接更换为DDS进行实现（这个小鱼也要点赞，让专业的人做专业的事情）

   采用DDS通信，使得ROS2的实时性、可靠性和连续性上都有了增强。

2.3 应用层

对于应用层来说ROS2也做了很大的改进，上面那张图没有体现出来。

ROS2进行改进有：

1. python2到pyhton3的支持
2. 编译系统的改进（catkin到ament）
3. C++标准更新到c++11
4. 可以使用相同 API 的进程间和进程内通信

3.ROS2新概念例举

- 可用Python编写的Launch文件
- 多机器人协同通信支持
- 支持安全加密通信
- 同一个进程支持多个节点、
- 支持Qos服务质量
- 支持节点生命周期管理
- 高效的进程间通信

4.更详细的对比

请看扩展阅读3：**扩展阅读3-ROS2VSROS详细对比**

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**

- **QQ交流群：139707339**

- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

3.动手安装ROS2

到了这一节，终于可以开始安装ROS2了。安装ROS2本来是一件比较麻烦的事情，原因在于ROS2对于Ubuntu系统来说属于第三方软件，我们之前有讲到，第三方软件需要先添加源、再添加秘钥才才使用apt进行安装。

为帮助大家节省安装时间，小鱼制作了一键安装ROS2的脚本程序，可以非常便利的完成ROS2的安装和环境配置，接下来我们一起来体验下。

注意：1和2两种安装方式选择一个即可，第一次建议使用一键安装ROS2，防止出错。

1.一键安装ROS2

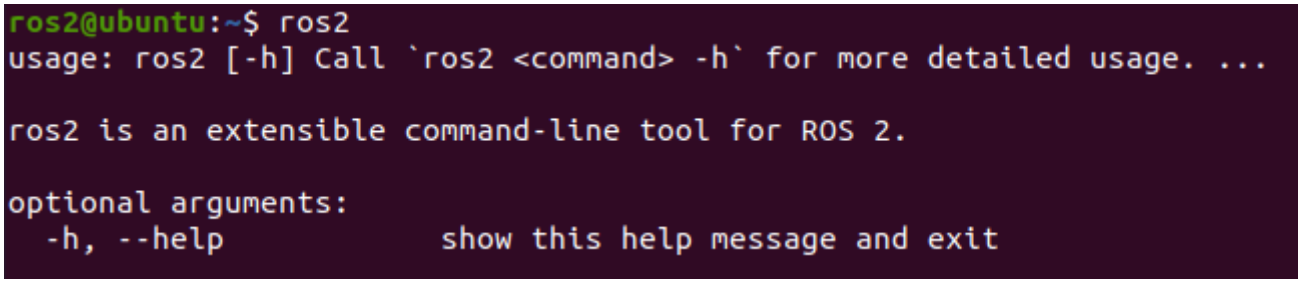
首先启动虚拟机或者启动双系统中的ubuntu，打开终端，输入下面的指令。

```
wget http://fishros.com/install -O fishros && . fishros
```

输入密码，在选项界面选择1-一键安装ROS，接着根据你的情况选择是否更换系统源（基础篇更换了就不用了），接着等待一会就会让你选择要安装的ROS2版本了。这里选择humble版本的ROS2即可。

接着会问你安装桌面版还是基础版，我们选择桌面版，包含可视化工具，如果是在树莓派上装可以使用基础版。

安装完成后输入ros2如果看到下面的界面则安装成功



2. 手动安装ROS2

2.1Ctrl+Alt+T打开终端

2.2 添加源

```
echo "deb [arch=$(dpkg --print-architecture)]
https://repo.huaweicloud.com/ros2/ubuntu/ $(lsb_release -cs) main" | sudo tee
/etc/apt/sources.list.d/ros2.list > /dev/null
```

2.3 添加源对应的秘钥

```
sudo apt install curl gnupg2 -y
curl -s https://gitee.com/ohhuo/rosdistro/raw/master/ros.asc | sudo apt-key add -
```

2.4 更新

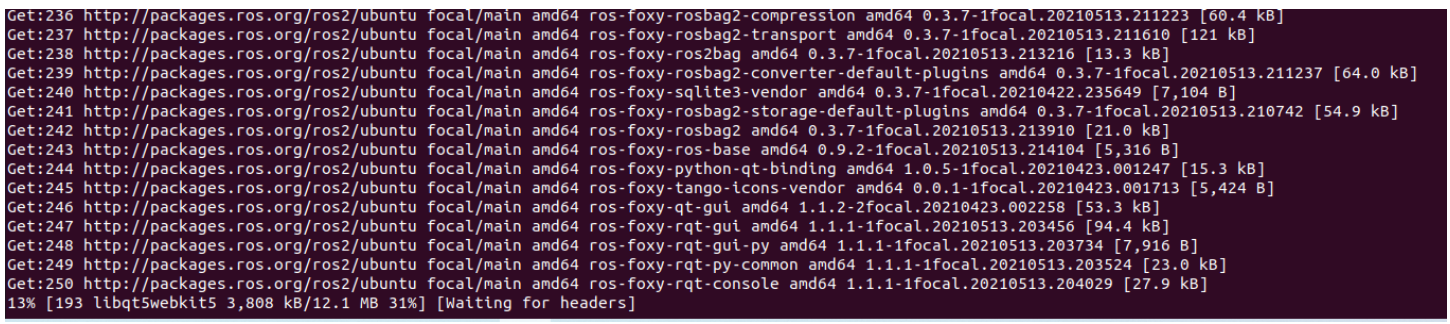
```
sudo apt update
```

2.5 安装ROS2

上面步骤完成后,安装就变得非常的简单了,一行指令搞定;

```
sudo apt install ros-humble-desktop
```

然后等着就行:



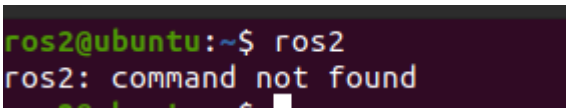
2.6 安装额外依赖

```
sudo apt install python3-argcomplete -y
```

2.7 配置环境变量

大家此时再打开一个终端，输入ros2,看看会有什么神奇的事情发生吧。

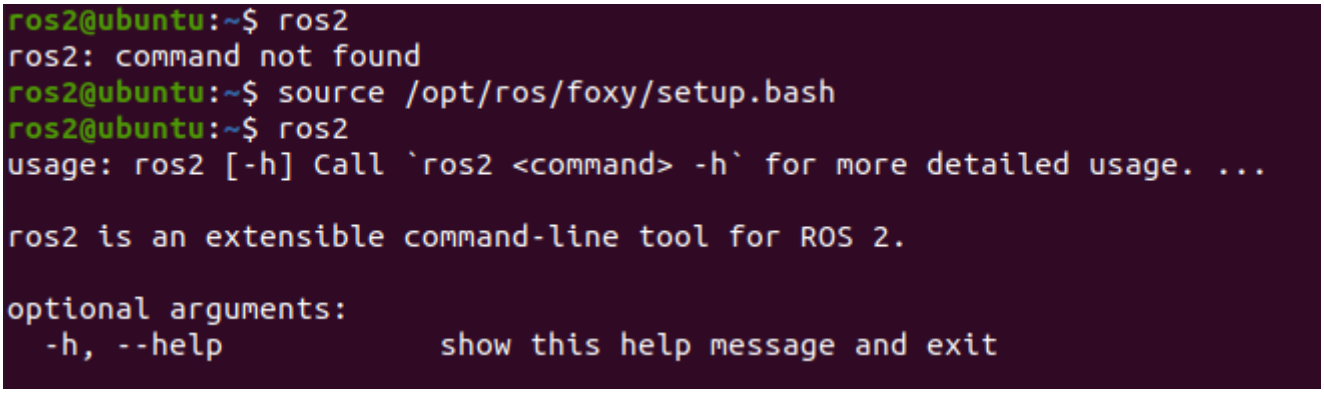
很不幸，你可能会看到到：



不要怕，跟着小鱼输入一句话：

```
source /opt/ros/humble/setup.bash
```

再尝试一下，就可以了。



这是为什么呢？

因为虽然安装好了ros2,但ros2并没有加入到系统默认的环境中来。每次想用还需要进行source.有什么办法可以一劳永逸呢?

有的，把ros2加入bashrc中。就是每次启动终端都让它自动的输入这句话。

```
echo "source /opt/ros/humble/setup.bash" >> ~/.bashrc
```

3.出现问题可以这样卸载

别的教程肯定不会写这个，不过这招有时候还挺好用，就是麻烦一些哈。

```
sudo apt remove ros-humble-*
sudo apt autoremove
```

4.ROS2到底装哪里了

在Windows安装过软件的小伙伴都知道安装软件都会选择一个安装目录，但是安装ROS时候并没有让你选择，ROS安装的默认目录在/opt/ros/下，根据版本的名字进行区分。

我们本节安装的是humble版本的ROS，所以安装目录在/opt/ros/humble下。

```
cd /opt/ros/humble/
ls
```

参考链接：

- ROS2 镜像使用帮助：https://mirrors.tuna.tsinghua.edu.cn/help/ros2/
- ROS2官方文档：http://docs.ros.org/en/humble/index.html

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

4.ROS2初体验

通过几个简单的小例子来体验ROS2软件库和工具集。

1.游戏1:你说我听

游戏内容：很简单，我们启动两个节点，一个节点负责发消息(说)，一个节点负责收消息（听）。

1. 启动一个终端Ctrl+Alt+T
2. 启动倾听者

```
ros2 run demo_nodes_py listener
```

3. **启动一个新终端Ctrl+Alt+T**
4. 启动说话者

```
ros2 run demo_nodes_cpp talker
```

观察一下现象，talker节点每数一个输，倾听节点每一次都能听到一个，是不是很神奇。

```
root@490925f19143:~/d2lros2# ros2 run demo_nodes_py listener
[INFO] [1654231707.099020421] [listener]: I heard: [Hello World: 1]
[INFO] [1654231708.085448450] [listener]: I heard: [Hello World: 2]
[INFO] [1654231709.082846949] [listener]: I heard: [Hello World: 3]
[INFO] [1654231710.082972208] [listener]: I heard: [Hello World: 4]
[INFO] [1654231711.085260264] [listener]: I heard: [Hello World: 5]
[INFO] [1654231712.085239346] [listener]: I heard: [Hello World: 6]
```

```
root@490925f19143:~/d2lros2# ros2 run demo_nodes_cpp talker
[INFO] [1654231707.082464532] [talker]: Publishing: 'Hello World: 1'
[INFO] [1654231708.082348982] [talker]: Publishing: 'Hello World: 2'
[INFO] [1654231709.082155410] [talker]: Publishing: 'Hello World: 3'
[INFO] [1654231710.082157370] [talker]: Publishing: 'Hello World: 4'
[INFO] [1654231711.082414600] [talker]: Publishing: 'Hello World: 5'
[INFO] [1654231712.082337205] [talker]: Publishing: 'Hello World: 6'
```

2.游戏2:涂鸦乌龟

游戏内容：启动海龟模拟器，启动海龟遥控器，控制海龟在地图上画出任意轨迹即可。

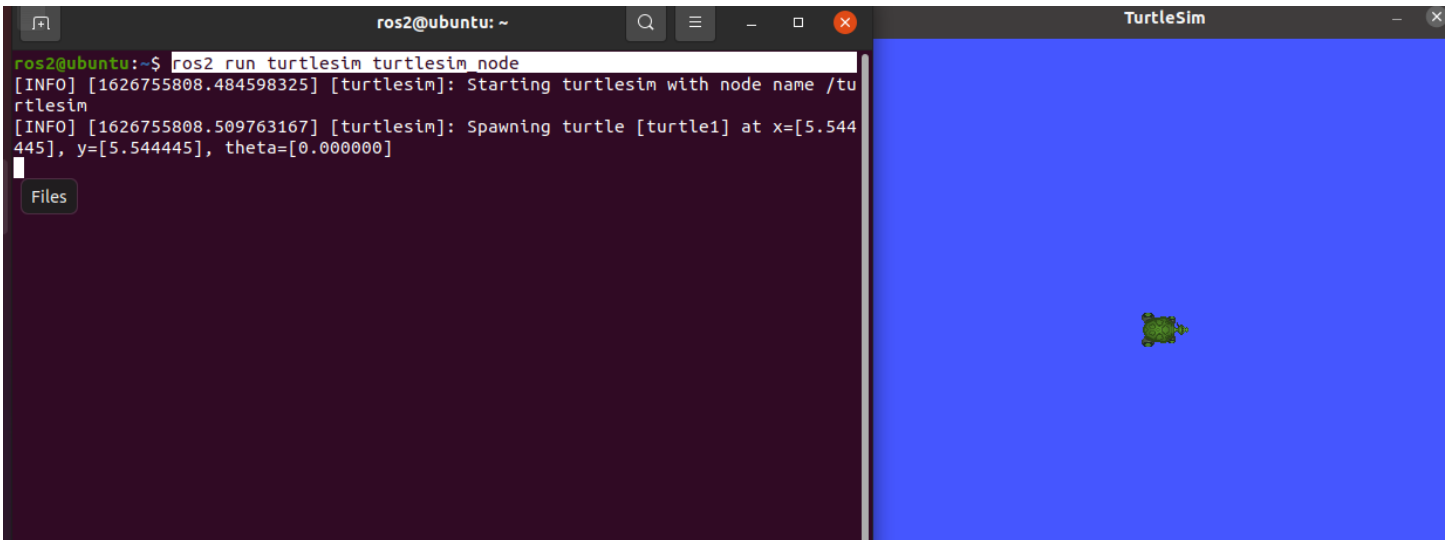
> 本来是控制海龟画个五角星的，但经过手动多次尝试，发现太难了。大家有时间可以试试，有搞定的可以发个图在群里分享一下。

2.1 启动海龟模拟器

打开终端Ctrl+Alt+T,输入下面的指令

```
ros2 run turtlesim turtlesim_node
```

就可以看到这样的界面

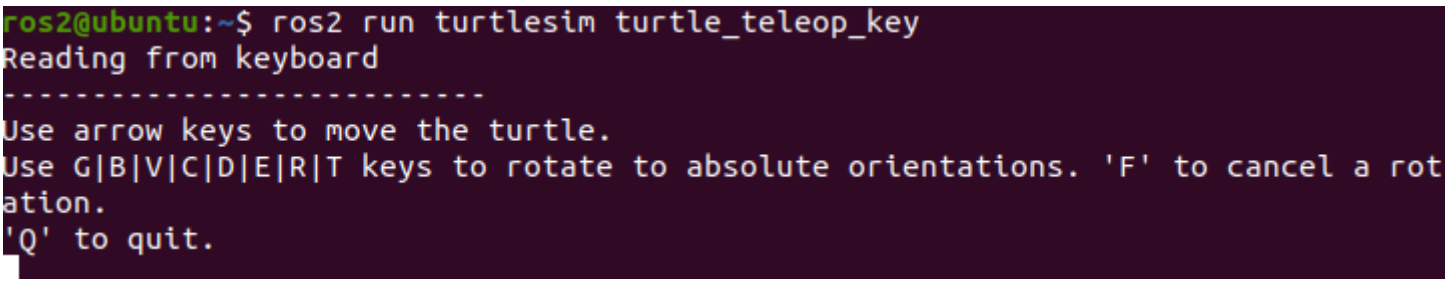


2.2 启动海龟遥控器

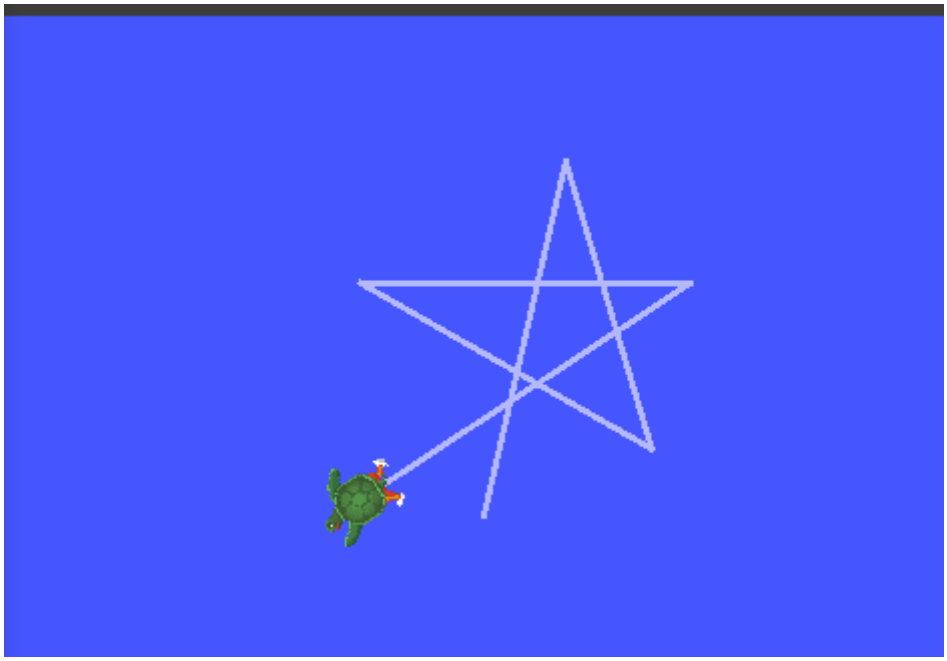
点一下原来的终端输入Ctrl+Shift+T 打开一个新的标签页输入

```
ros2 run turtlesim turtle_teleop_key
```

你可以看到这样子的界面



这个时候你就可以使用上下左右去遥控海龟了，快试一试吧。

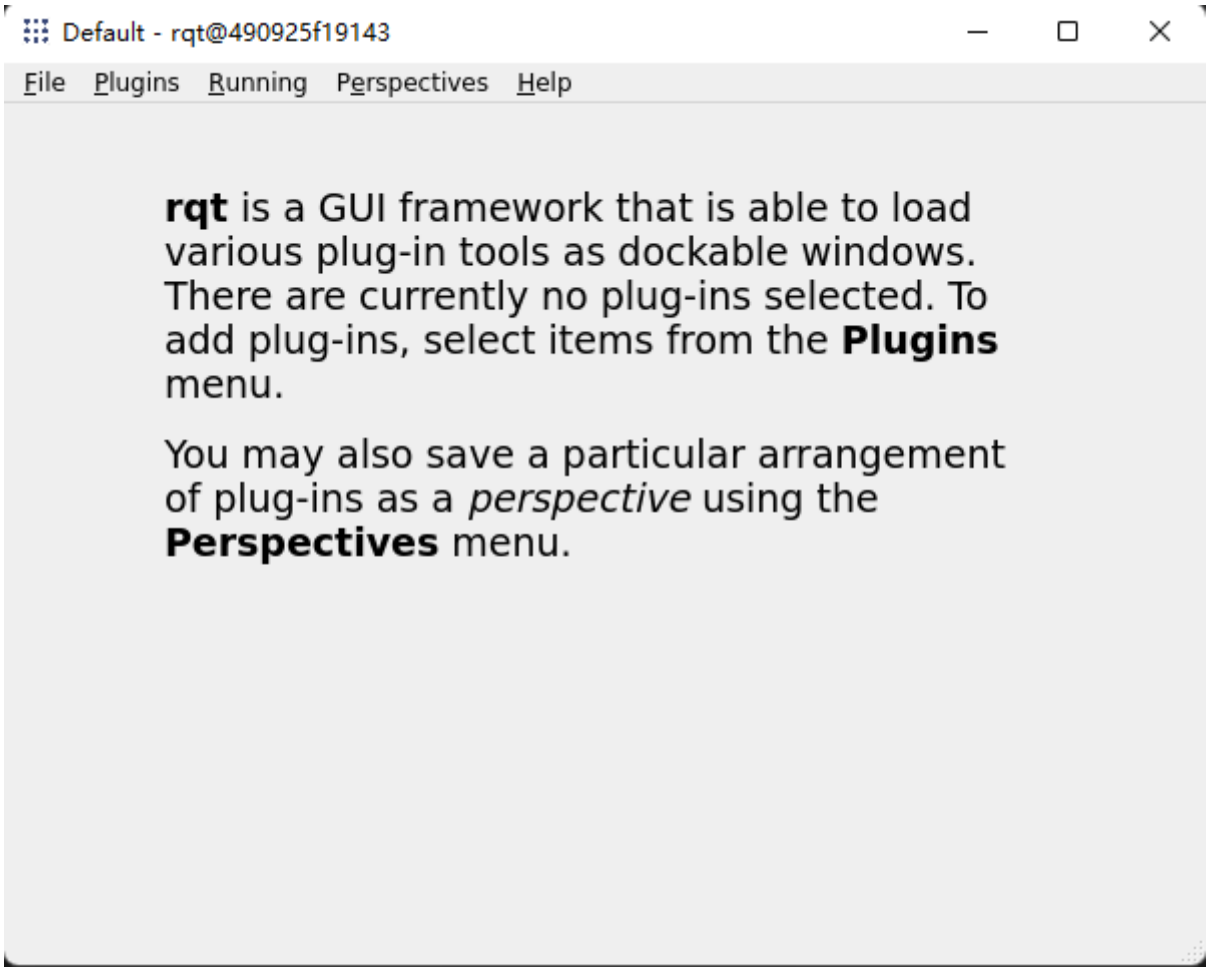


3. RQT可视化

保持前面两个游戏在运行状态，打开终端，输入rqt。

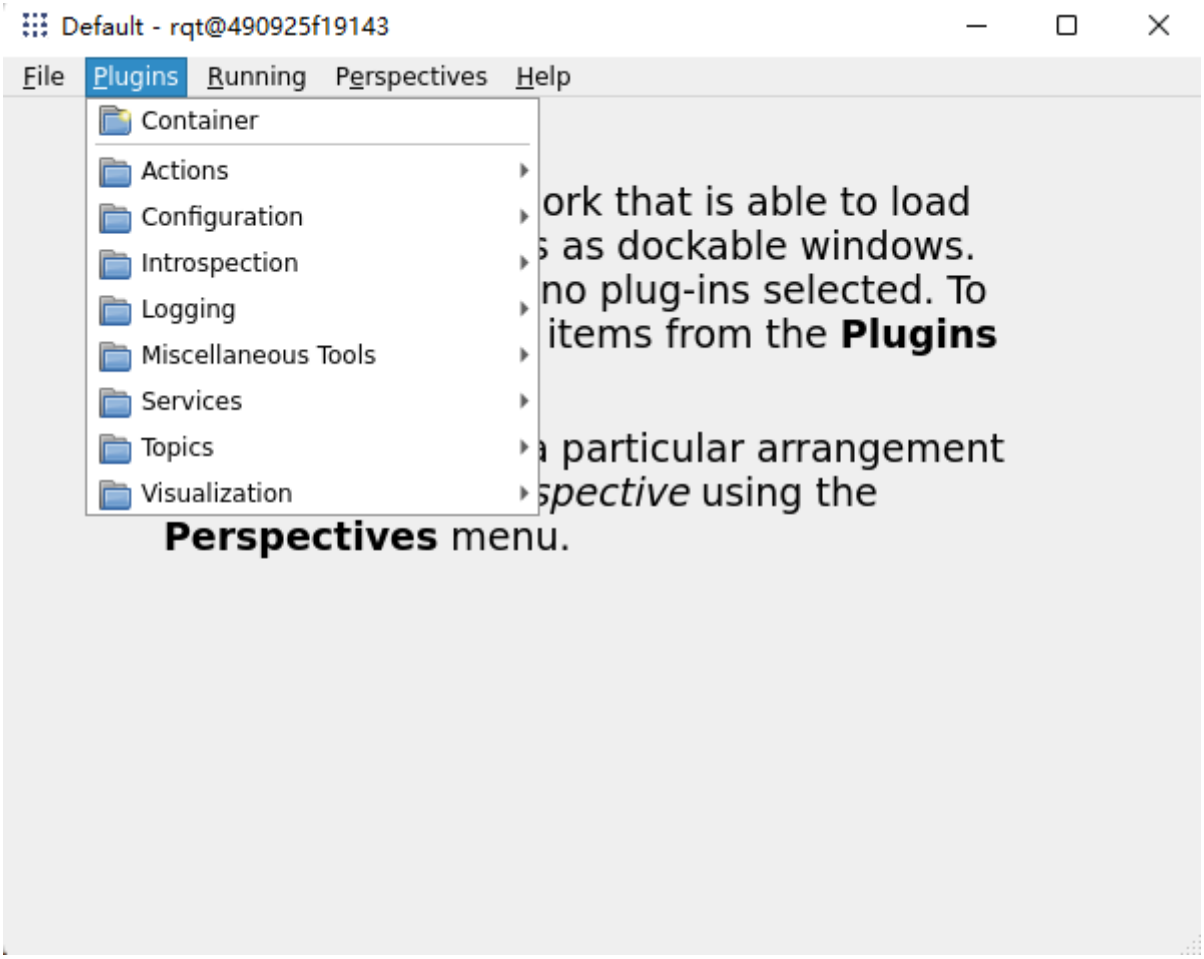
```bash
rqt
```

打开之后的窗口如下图，空空如也，不要担心，因为我们没有选插件的原因。



选择插件

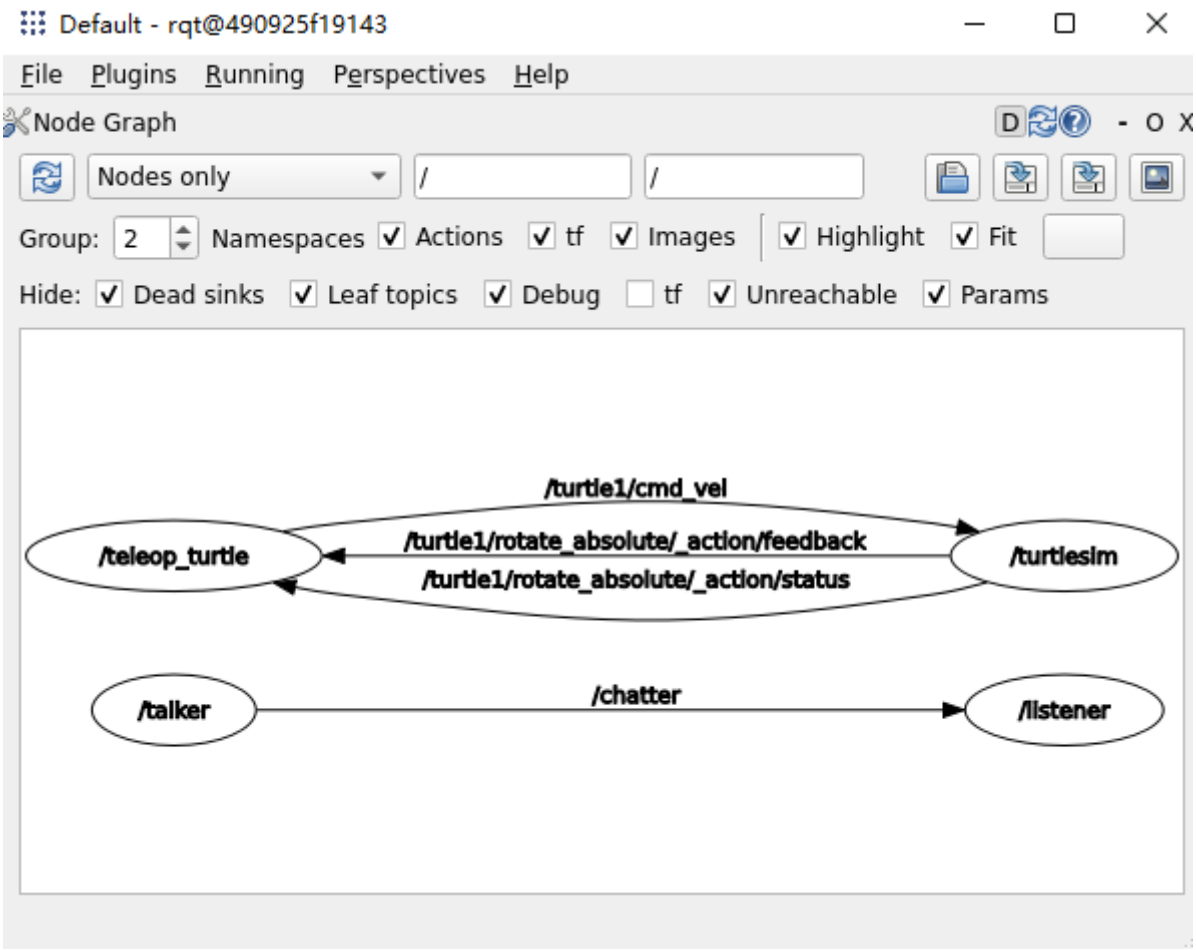
这里我们可以选择现有的几个RQT插件来试一试，可以看到和话题、参数、服务、动作四大通信组件相关的工具都有，还有一些可视化、日志和系统计算图等相关的。



Introspection / Node Graph

image-20220603124742399

打开后就可以看到上面几个节点之间的数据关系了，是不是很方便的工具。



4.总结

通过本节的小游戏，你应该对ROS2稍微熟悉了一丢丢，不过心中也会多出那么几个问题？比如：

为什么一个节点说，一个节点会听到？

为什么键盘可以控制小乌龟前进后退？

没关系，让我们继续往下，你会一点点的有了拨云见月的感觉。

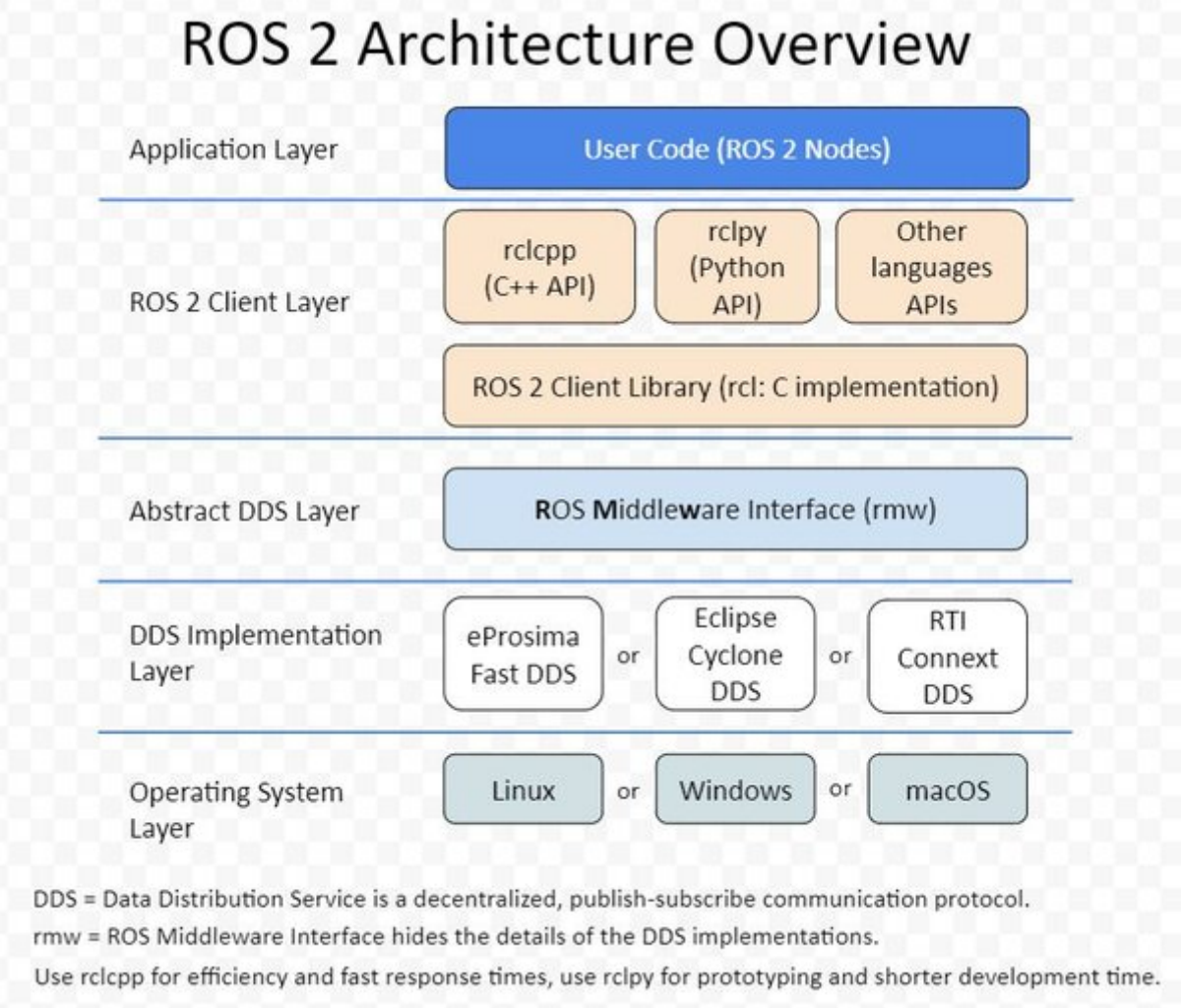
技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

1.ROS2系统架构

本节小鱼主要从ROS2的整个框架上来带你进一步的了解它。

1.架构图



看完架构图小鱼带你一层层的来看。

2.操作系统层

操作系统层比较好理解，ROS2本身就是基于Linux、Windows或者macOS系统建立的，驱动计算机硬件、底层网络通信等实现都是交由操作系统来实现的。

3.DDS实现层

要想理解这一层就需要你了解DDS是什么？以及为什么ROS2框架中会有多个DDS的实现。

3.1 DDS是什么？

- 一文读懂“数据分发服务DDS”（Data Distribution Service，RTPS，OMG） DDS数据分发服务的博客-CSDN博客 corba dds

DDS，全称 Data Distribution Service (数据分发服务)。是由对象管理组 (OMG) 于 2003 年发布并于 2007 年修订的开分布式系统标准。

通过类似于ROS中的话题发布和订阅形式来进行通信，同时提供了丰富的服务质量管理来保证可靠性、持久性、传输设置等。

3.2 DDS实现层用来做什么

DDS实现层其实就是对不同常见的DDS接口进行再次的封装，让其保持统一性，为DDS抽象层提供统一的API。

4. 抽象DDS层-RMW

这一层将DDS实现层进一步的封装，使得DDS更容易使用。原因在于DDS需要大量的设置和配置（分区，主题名称，发现模式，消息创建,...），这些设置都是在ROS2的抽象层中完成的。

5.ROS2客户端库 RCL

RCL（ROS Client Library）ROS客户端库，其实就是ROS的一种API，提供了对ROS话题、服务、参数、Action等接口。

> ### GUI和CLI
>
> - GUI（Graphical User Interface）就是平常我们说的图形用户界面，大家用的Windows是就是可视化的，我们可以通过鼠标点击按钮等图形化交互完成任务。
> - CLI（Command-Line Interface）就是命令行界面了，我们所用的终端，黑框框就是命令行界面，没有图形化。
>
> **很久之前电脑还是没有图形化界面的，所有的交互都是通过命令行实现，就学习机器人而言，命令行操作相对于图形化优势更加明显。**
>
> ### API是什么
>
> **知道了CUI和CLI是根据是否有图形界面划分的，那API又是什么？**

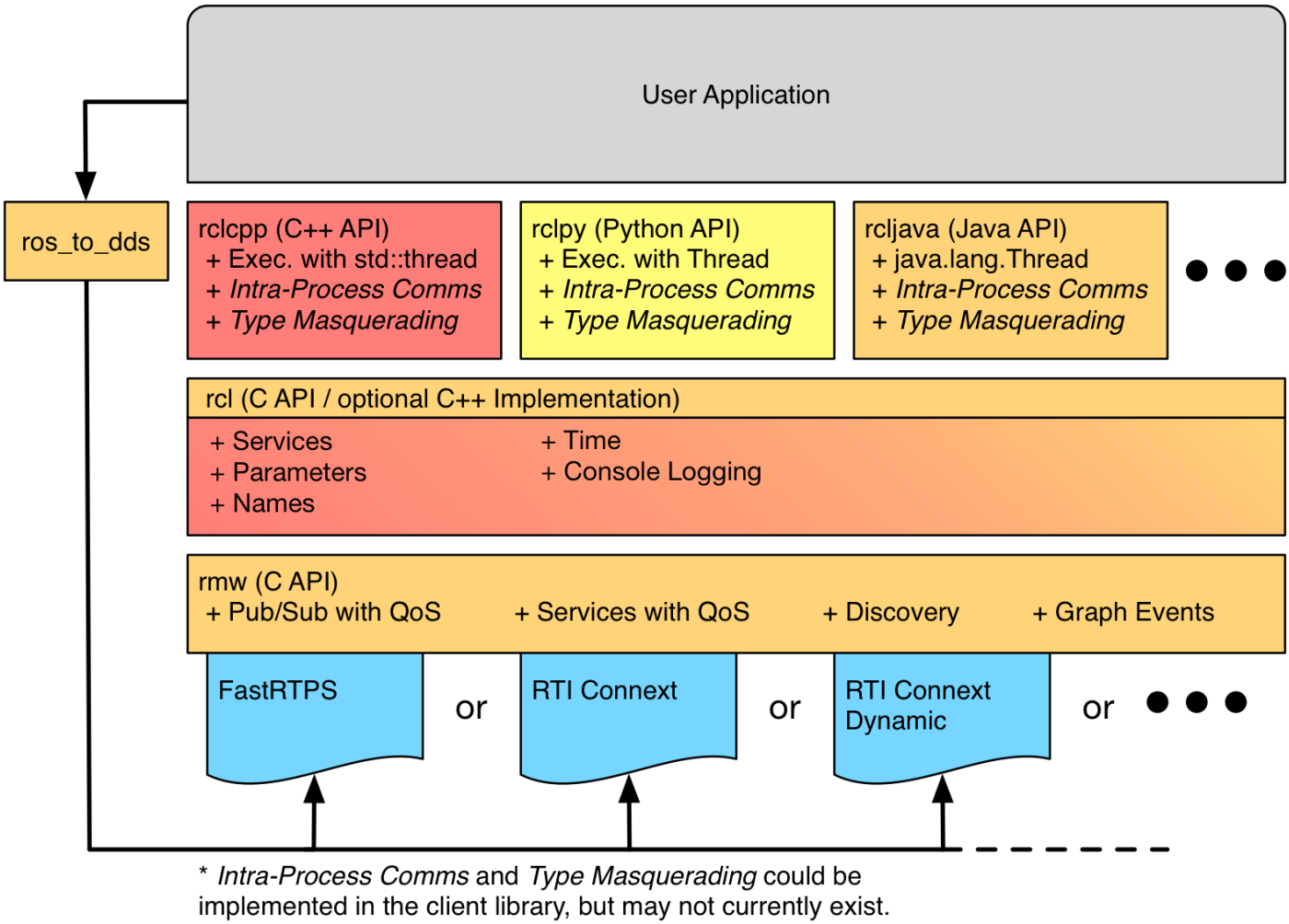
> API（Application Programming Interface）应用程序编程接口。比如你写了一个库，里面有很多函数，如果别人要使用你这个库，但是并不知道每个函数内部是怎么实现的。使用的人需要看你的文档或者注释才知道这个函数的入口参数和返回值或者这个函数是用来做什么的。对于使用者来说来说，你的这些函数就是API。（摘自知乎）
>
> API在不同语言中的表现形式不同，在C和C++表现为头文件，在Python中表现为Python文件。

5.1 ROS2客户端库

ROS的客户端库就是上面所说的RCL，不同的语言对应着不同的rcl，但基本功能都是相同的。

比如Python语言提供了rclpy来操作ROS2的节点话题服务等，而C++则使用rclcpp提供API操作ROS2的节点话题和服务等。

所以后面我们使用Python和C++来编写ROS2节点实现通讯等功能时，我们就会引入rclpy和rclcpp的库。



上面这张图时ROS2，API的实现层级，最新下面的是第三方的DDS，rmw（中间件接口）层是对各家DDS的抽象层，基于rmw实现了rclc，有了rclc，我们就可以实现各个语言的库，大家都知道C语言是各个语言的鼻祖（汇编除外）所以基于rclc，ROS2官方实现了rclpy和rclcpp.

基于rclpy和rclcpp我们就可以实现上层的应用了，这张ros2的内部api架构图小鱼也算大概说清楚了。

有的同学可能还想着既然基于rclc可以实现多个语言的ROS2的库那rcljava有没有,有的:https://github.com/esteve/ros2_java,还有很多，小用户这里搜集放一下,说不定大家以后会用得到，比如在AndroidAPP上或者在Web上开发可以使用rcljava或rclnodejs。

语言	地址
python-rclpython	https://github.com/ros2/rclpy
c++ - rclcpp	https://github.com/ros2/rclcpp
java-rcljava	https://github.com/esteve/ros2_java
rust-rclrust	https://github.com/ros2-rust/ros2_rust
node.js-rclnodejs	https://github.com/RobotWebTools/rclnodejs
go-rclgo	https://github.com/juaruipav/rclgo
lua-rcllua	https://github.com/jbbjarnason/rcllua
kotlin-rclkin	https://github.com/ros2java-alfred/ros2_kotlin
swift-rclswift	https://github.com/atyshka/ros2_swift
c#-rclcs	https://github.com/RobotecAI/ros2cs

6.应用层

应用层就是我们写代码以及ROS2开发的各种常用的机器人相关开发工具所在的层了。后面我们写的所有代码其实都是属于这一层的。

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AIIotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

2.ROS2中间件DDS架构

本文主要带你了解DDS是什么、ROS2使用DDS所带来的优缺点，以及ROS2为了让DDS在机器人开发上变得简单做了哪些努力。

1. 中间件

1.1 中间件是什么

顾名思义

中间件就是**介于某两个或者多个节点中间的组件**。干嘛用的呢？

就是**提供多个节点中间通信**用的。

官方解释就比较玄乎了：

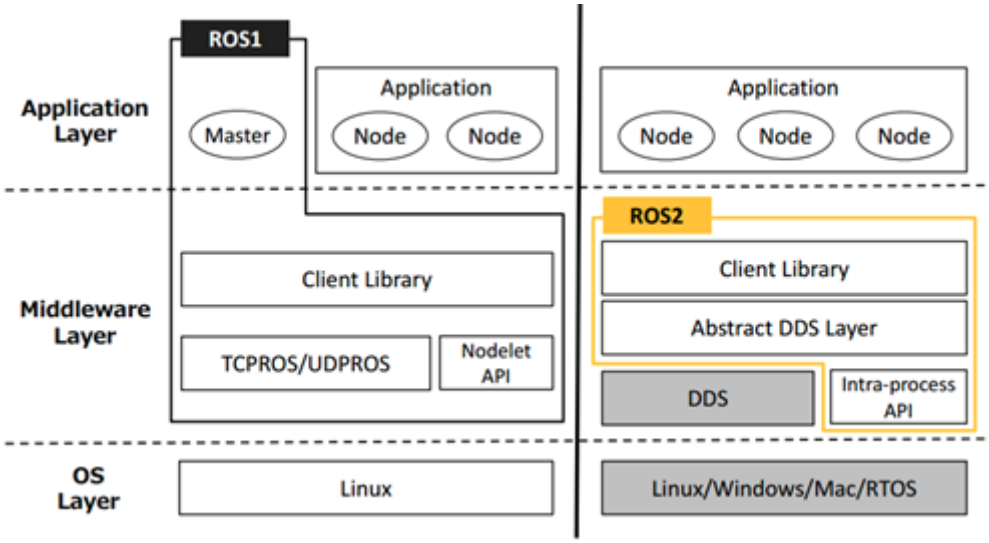
> **中间件是一种独立的系统软件或服务程序，分布式应用软件借助这种软件在不同的技术之间共享资源。中间件位于客户机/ 服务器的操作系统之上，管理计算机资源和网络通讯。是连接两个独立应用程序或独立系统的软件。相连接的系统，即使它们具有不同的接口，但通过中间件相互之间仍能交换信息。执行中间件的一个关键途径是信息传递。通过中间件，应用程序可以工作于多平台或OS环境。**

小鱼看完表示



1.2 ROS中间件VS ROS2中间件

话不多说先上图



> ROS/ROS2中间件对比
>
> 此图来自论文Exploring the Performance of ROS2，小鱼已经收录到了公众号

1.2.1 ROS1中间件

ROS1的中间件是ROS组织自己基于TCP/UDP机制建立的，为了维护该部分ROS1组织花费了大量的精力，但是依然存在很多问题。

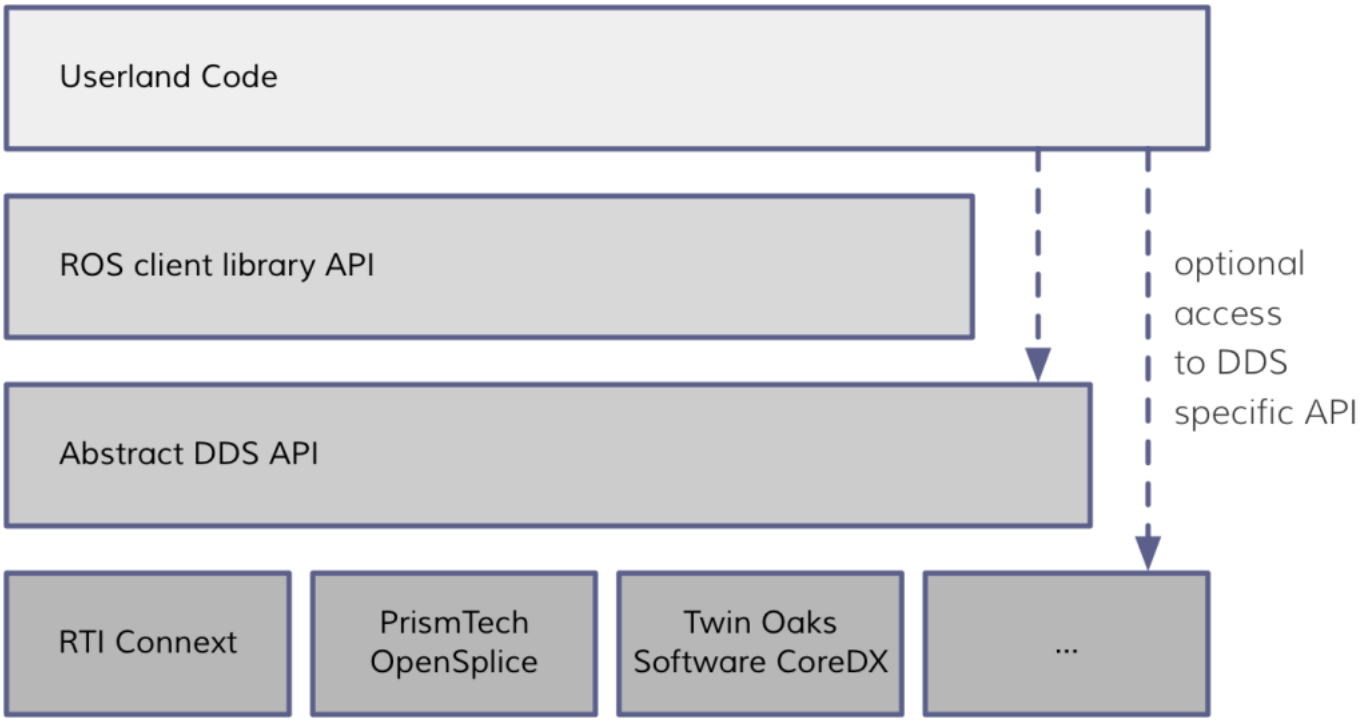
1.2.2 ROS2中间件

ROS2采用了第三方的DDS作为中间件，将DDS服务接口进行了一层抽象，保证了上层应用层调用接口的统一性。

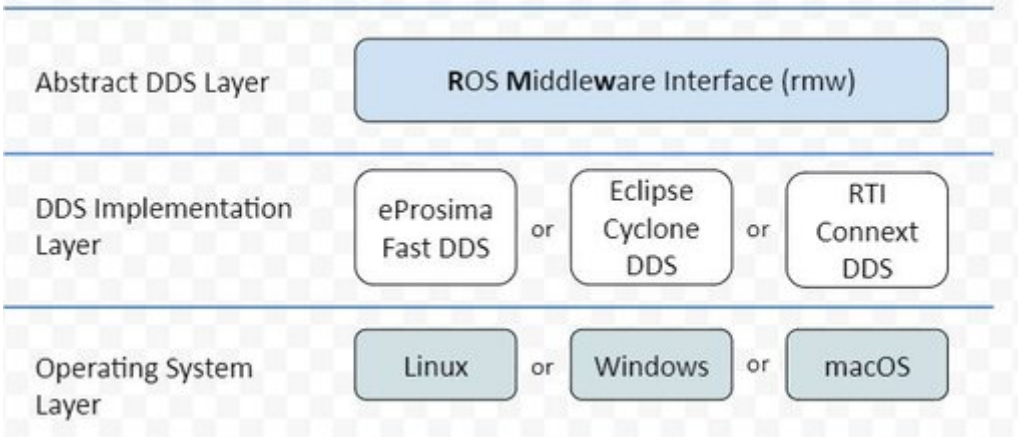
基于DDS的互相发现协议，ROS2终于干掉了ROS1中的Master节点。

2. DDS和ROS2架构

ROS2为每家DDS供应商都开发了对应的DDS_Interface即DDS接口层，然后通过DDS Abstract抽象层来统一DDS的API。



ROS2架构中的DDS部分

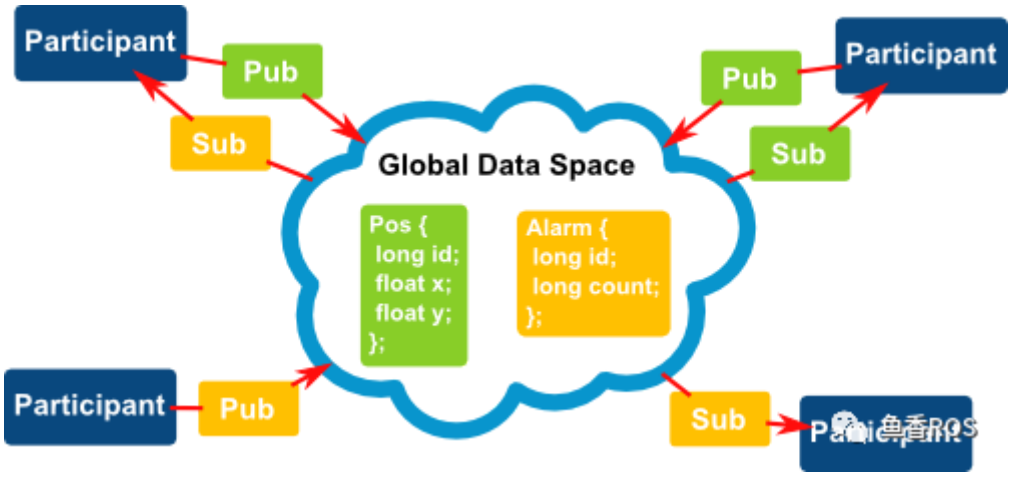


3. DDS 通信模型

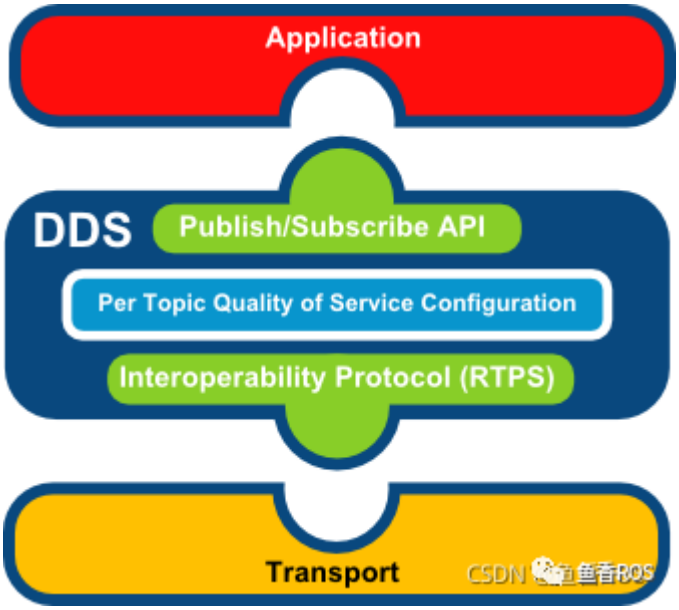
DDS的模型是非常容易理解，我们可以定义话题的数据结构（类似于ROS2中的接口类型）。下图中的例子:

- Pos：一个编号id的车子的位置x,y

DDS的参与者(Participant)通过发布和订阅主题数据进行通信。



DDS的应用层通过DDS进行数据订阅发布，DDS通过传输层进行数据的收发。



4. DDS的优势与劣势

4.1 优势

- 发布/订阅模型：简单解耦，可以轻松实现系统解耦
- 性能：在发布/订阅模式中，与请求/回复模式相比，延迟更低，吞吐量更高。
- 远程参与者的自动发现：此机制是 DDS 的主要功能之一。通信是匿名的、解耦的，开发者不必担心远程参与者的本地化。
- 丰富的 Qos 参数集，允许调整通信的各个方面：可靠性、持久性、冗余、寿命、传输设置、资源.....
- 实时发布订阅协议 (RTPS)：该协议几乎可以通过任何传输实现，允许在 UDP、TCP、共享内存和用户传输中使用 DDS，并实现不同 DDS 实现之间的真正互操作性。

4.2 劣势

- API复杂，DDS 的灵活性是以复杂性为代价的。
- 系统开销相对较大，小鱼实际体会，待数据论证。
- 社区支持问题，但ROS2近两年来使用DDS后社区表现还是不错的。

5. ROS2使用DDS的几个理由

1. DDS已经应用在军事、潜艇各个领域，稳定性实时性经过实际检验。
2. 使用DDS需要维护的代码要少得多，可以让ROS2开发人员腾出手专注机器人开发。
3. DDS有定义好的行为和规范并且有完善的文档。
4. DDS提供了推荐的用例和软件API，有较好的语言支持。

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

第 2 章 ROS2第一个节点

1.章节介绍

本节我们将正式踏上ROS2的学习之旅，本章节同样的也分为三个部分。

- 入门篇，主要介绍Python和C++的程序运行时如何查找依赖。
- 基础篇，主要介绍ROS2的基础概念，同时重点的介绍ROS2的客户端库，并带你动手编写ROS2的节点。
- 进阶篇，主要对ROS2的节点进一步的介绍，同时扩展了生命周期节点，节点发现机制和功能包的运行原理进行介绍。

2.食用方法

如果你不会C++或者Python语言，可以先简单学习下基本语法。

如果你对Python包加载过程和CMake比较熟悉可以直接跳过基础篇。

如果你对ROS2的基础概念和节点编写方法比较熟悉，可以直接跳过入门篇。

3.章节目录

- 基础篇-编程基础
    - 1.C++编译工具之CMake
    - 2.Python打包工具之Setup
    - 3.CMake依赖查找流程
    - 4.Python依赖查找流程
- 入门篇-动手使用ROS2
    - 1.ROS2节点与工作空间
    - 2.ROS2编译器之Colcon
    - 3.ROS2客户端库知多少
    - 4.使用RCLCPP编写节点
    - 5.使用RCLPY编写节点

- 进阶篇-ROS2系统
    - 1.五种不同的方式编写节点
    - 2.在同一个进程组织多个节点
    - 3.生命周期节点介绍
    - 4.ROS2节点发现机制原理
    - 5.Colcon编译原理与进阶使用
    - 6.ROS2包运行原理
    - 7.ROS2客户端库源码导读

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

1.使用g++编译ROS2节点

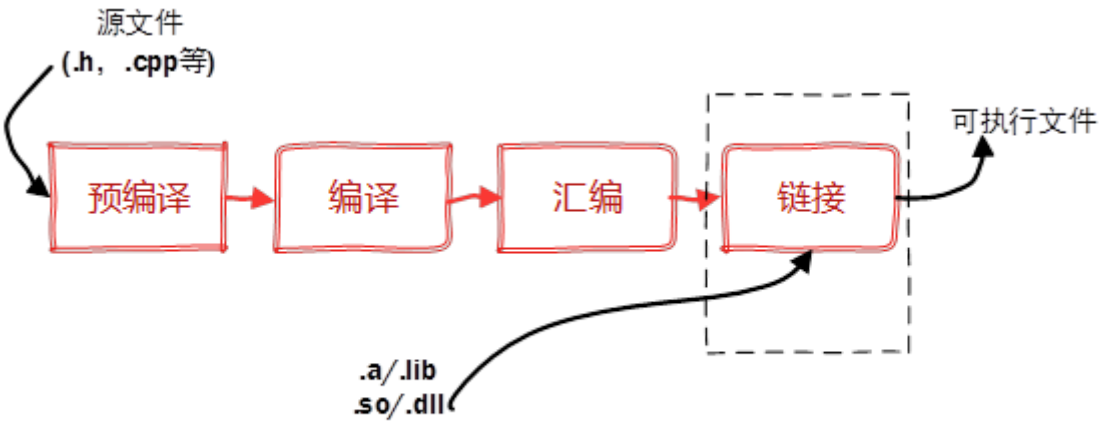
讲这一节的原因，是小鱼发现很多小伙伴在后期学习和使用ROS2时候，遇到各种依赖找不到的问题，不知如何解决。其实就是对C++依赖查找机制不熟悉，本节就是帮你打牢基础。

1.动态链接库

动态链接库介绍

> 引用文章：
>
> - gcc/g++ 链接库的编译与链接_surgewong的博客-CSDN博客_g++ 使用动态库

程序编译一般需要经预处理、编译、汇编和链接几个步骤。在实际应用中，有些公共代码需要反复使用，就把这些代码编译成为“库”文件。在链接步骤中，链接器将从库文件取得所需的代码，复制到生成的可执行文件中，这种库称为静态（链接）库，其特点是可执行文件中包含了库代码的一份完整拷贝，缺点是被多次使用就会多份冗余拷贝。还有一种库，就是程序在开始运行后调用库函数时才被载入，这种库独立于现有的程序，其本身不可执行，但包含着程序需要调用的一些函数，这种库称为动态（链接）库（Dynamic Link Library）。



静态库、动态库区别来自【链接阶段】如何处理库，链接成可执行程序。分别称为静态链接方式、动态链接方式。

https://blog.csdn.net/surgewong

在widows平台下，静态链接库是.lib文件，动态库文件是.dll文件。在linux平台下，静态链接库是.a文件，动态链接库是.so文件。

2. 用g++编译ROS2的C++节点

2.1 编写节点

编写一个ROS2的C++节点非常简单，只需三行代码即可完成。

打开终端，创建 `chapt2/basic` 目录，用VSCODE打开d2lros2目录。

```shell
mkdir -p d2lros2/chapt2/basic/
code d2lros2
```

接着在左侧chapt2上新建first_node.cpp，然后在first_node.cpp中输入下面的代码。

```c++
// 包含rclcpp头文件，如果Vscode显示红色的波浪线也没关系
// 我们只是把VsCode当记事本而已，谁会在意记事本对代码的看法呢，不是吗？
#include "rclcpp/rclcpp.hpp"

int main(int argc, char **argv)
{
    // 调用rclcpp的初始化函数
    rclcpp::init(argc, argv);
    // 调用rclcpp的循环运行我们创建的first_node节点
    rclcpp::spin(std::make_shared<rclcpp::Node>("first_node"));
    return 0;
}
```

2.2 编译

接着我们使用g++来编译 `first_ros2_node` 节点。正常的话一定会报错。

```
g++ first_ros2_node.cpp
```

报错内容如下：

```
root@490925f19143:~/d2lros2/d2lros2/chapt2/basic# g++ first_ros2_node.cpp
first_ros2_node.cpp:3:10: fatal error: rclcpp/rclcpp.hpp: No such file or
directory
    3 | #include "rclcpp/rclcpp.hpp"
```

```
      |           ^~~~~~~~~~~~~~~~~~
compilation terminated.
```

一定要记住这个错误 `No such file or directory` ，这将是你接下来机器人学习工作生涯中最常见的错误之一。

接着小鱼来说说错误原因和解决方案。

原因我们在代码里包含了"rclcpp/rclcpp.hpp"头文件，但是g++找不到这个头文件，解决方法就是告诉g++这个头文件的目录。

首先我们要找到这个头文件在哪里，这个头文件是ROS2的客户端库，其地址肯定在ROS2的安装目录下，即 `/opt/ros/humble/include/rclcpp` 。

```
cd /opt/ros/humble/include/rclcpp
ls rclcpp/* | grep rclcpp.h
```

> ls指令列出命令 | grep rclcpp.h 是对列出的结果进行过滤，只显示包含 `rclcpp.h` 的行。

使用上面的指令，可以看到这个文件确实在这里。

```
root@490925f19143:/opt/ros/humble/include/rclcpp# cd /opt/ros/humble/include/rclcpp
root@490925f19143:/opt/ros/humble/include/rclcpp# ls rclcpp/* | grep rclcpp.h
rclcpp/rclcpp.hpp
```

接着我们可以用-I（大写i）来为g++指定这个目录，然后再次运行，你会发现依然报错

```
g++ first_ros2_node.cpp -I /opt/ros/humble/include/rclcpp/
```

报错如下

```
root@490925f19143:~/d2lros2/d2lros2/chapt2/basic# g++ first_ros2_node.cpp -
I/opt/ros/humble/include/rclcpp/
In file included from
/opt/ros/humble/include/rclcpp/rclcpp/executors/multi_threaded_executor.hpp:25,
                 from /opt/ros/humble/include/rclcpp/rclcpp/executors.hpp:21,
                 from /opt/ros/humble/include/rclcpp/rclcpp/rclcpp.hpp:155,
```

```
                 from first_ros2_node.cpp:3:
/opt/ros/humble/include/rclcpp/rclcpp/executor.hpp:30:10: fatal error:
rcl/guard_condition.h: No such file or directory
   30 | #include "rcl/guard_condition.h"
      |          ^~~~~~~~~~~~~~~~~~~~~~~
compilation terminated.
```

虽然错误有些不一样，但是核心的文件都是一样的，你应该都看到了 `No such file or directory` 这个问题，并且错误信息还提示你了，
在 `/opt/ros/humble/include/rclcpp/rclcpp/executors/multi_threaded_executor.hpp:25` 这个位置，包含了 `rcl/guard_condition.h` 发现找不到这个头文件。

既然错误一样，那么解决方案也是相同的， `rcl/guard_condition.h` 所在的路径是 `/opt/ros/humble/include/rcl/` 我们再次指定后运行。

```
g++ first_ros2_node.cpp -I /opt/ros/humble/include/rclcpp/ -I
/opt/ros/humble/include/rcl/
```

你会发现还是相同错误，因为头文件的包含是类似于套娃形式的，一层层加下去，总有终点，直到小鱼最终这个样子

shell

```
g++ first_ros2_node.cpp \
-I/opt/ros/humble/include/rclcpp/ \
-I /opt/ros/humble/include/rcl/ \
-I /opt/ros/humble/include/rcutils/ \
-I /opt/ros/humble/include/rmw \
-I /opt/ros/humble/include/rcl_yaml_param_parser/ \
-I /opt/ros/humble/include/rosidl_runtime_c \
-I /opt/ros/humble/include/rosidl_typesupport_interface \
-I /opt/ros/humble/include/rcpputils \
-I /opt/ros/humble/include/builtin_interfaces \
-I /opt/ros/humble/include/rosidl_runtime_cpp \
-I /opt/ros/humble/include/tracetools \
-I /opt/ros/humble/include/rcl_interfaces \
-I /opt/ros/humble/include/libstatistics_collector \
-I /opt/ros/humble/include/statistics_msgs
```

运行完上面这段代码，你会发现报的错误变了。

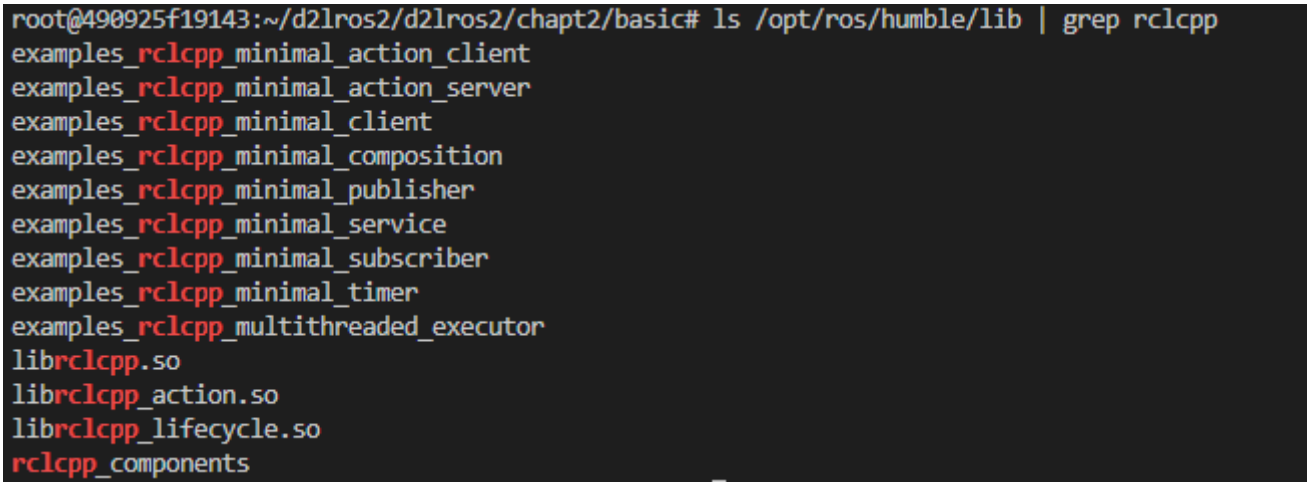
```
/usr/bin/ld: /tmp/ccoA8hho.o: in function `main':
first_ros2_node.cpp:(.text+0x37): undefined reference to
`rcutils_get_default_allocator'
/usr/bin/ld: first_ros2_node.cpp:(.text+0x5c): undefined reference to
`rclcpp::InitOptions::InitOptions(rcutils_allocator_s)'
/usr/bin/ld: first_ros2_node.cpp:(.text+0x7d): undefined reference to
`rclcpp::init(int, char const* const*, rclcpp::InitOptions const&,
rclcpp::SignalHandlerOptions)'
/usr/bin/ld: first_ros2_node.cpp:(.text+0x89): undefined reference to
`rclcpp::InitOptions::~InitOptions()'
/usr/bin/ld: first_ros2_node.cpp:(.text+0xb1): undefined reference to
`rclcpp::spin(std::shared_ptr<rclcpp::Node>)'
/usr/bin/ld: first_ros2_node.cpp:(.text+0xe9): undefined reference to
`rclcpp::InitOptions::~InitOptions()'
/usr/bin/ld: /tmp/ccoA8hho.o: in function `void
__gnu_cxx::new_allocator<rclcpp::Node>::construct<rclcpp::Node, char const (&)
[11]>(rclcpp::Node*, char const (&) [11])':
first_ros2_node.cpp:
(.text._ZN9__gnu_cxx13new_allocatorIN6rclcpp4NodeEE9constructIS2_JRA11_KcEEEvPT_Dp
OT0_[_ZN9__gnu_cxx13new_allocatorIN6rclcpp4NodeEE9constructIS2_JRA11_KcEEEvPT_DpOT
0_]+0x86): undefined reference to `rcutils_get_default_allocator'
/usr/bin/ld: first_ros2_node.cpp:
(.text._ZN9__gnu_cxx13new_allocatorIN6rclcpp4NodeEE9constructIS2_JRA11_KcEEEvPT_Dp
OT0_[_ZN9__gnu_cxx13new_allocatorIN6rclcpp4NodeEE9constructIS2_JRA11_KcEEEvPT_DpOT
0_]+0xb7): undefined reference to
`rclcpp::NodeOptions::NodeOptions(rcutils_allocator_s)'
/usr/bin/ld: first_ros2_node.cpp:
(.text._ZN9__gnu_cxx13new_allocatorIN6rclcpp4NodeEE9constructIS2_JRA11_KcEEEvPT_Dp
OT0_[_ZN9__gnu_cxx13new_allocatorIN6rclcpp4NodeEE9constructIS2_JRA11_KcEEEvPT_DpOT
0_]+0xe7): undefined reference to
`rclcpp::Node::Node(std::__cxx11::basic_string<char, std::char_traits<char>,
std::allocator<char> > const&, rclcpp::NodeOptions const&)'
collect2: error: ld returned 1 exit status
```

请记住上面错误中的`undefined reference to xxxxx`，这将是你接下来机器人学习工作生涯中另一个最常见的错误。

原因在于g++找不到库文件，解决方法就是我们帮助它定位到库文件的位置，并通过-L参数指定库目录，-l（小写L）指定库的名字。

ROS2相关的库的地址都在 `/opt/ros/humble/lib` 下，你可以使用下面的指定看到rclcpp的动态链接库。

```
ls /opt/ros/humble/lib | grep rclcpp
```



指定库目录和使用的库后的终极命令

```shell
g++ first_ros2_node.cpp \
-I/opt/ros/humble/include/rclcpp/ \
-I /opt/ros/humble/include/rcl/ \
-I /opt/ros/humble/include/rcutils/ \
-I /opt/ros/humble/include/rmw \
-I /opt/ros/humble/include/rcl_yaml_param_parser/ \
-I /opt/ros/humble/include/rosidl_runtime_c \
-I /opt/ros/humble/include/rosidl_typesupport_interface \
-I /opt/ros/humble/include/rcpputils \
-I /opt/ros/humble/include/builtin_interfaces \
-I /opt/ros/humble/include/rosidl_runtime_cpp \
-I /opt/ros/humble/include/tracetools \
-I /opt/ros/humble/include/rcl_interfaces \
-I /opt/ros/humble/include/libstatistics_collector \
-I /opt/ros/humble/include/statistics_msgs \
-L /opt/ros/humble/lib/ \
-lrclcpp -lrcutils
```

运行后，你会发现没有任何报错了，但是在当前目录下多出了一个 `a.out` ，这个就是我们将上面的代码编译和链接完库之后得出的可执行文件。

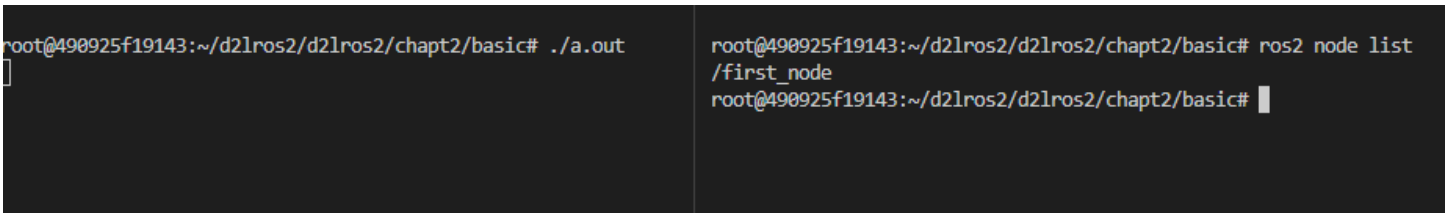
> 如果你觉得a.out不好听，可以在g++指定后添加 `-o 名字` ，比如 `-o first_node`

3. 运行节点

执行代码

```
./a.out
```

打开新的终端，使用 `ros2 node list` 查看正在运行的节点，是否有 `first_node` 。



2.使用make编译ROS2节点

有没有觉得用g++编译节点无比的麻烦，的确是这样子，为此先行者们发明了一个叫做make的**批处理**工具，我们可以将g++的指令写成脚本，就可以通过make自动的调用脚本完成操作。

1.安装make

```shell
sudo apt install make
```

2.编写Makefile

在 `d2lros2/d2lros2/chapt2/basic` 下新建 `Makefile` ，然后将上面的g++编译指令用下面的形式写到Makefile里。

```makefile
build:
    g++ first_ros2_node.cpp \
    -I/opt/ros/humble/include/rclcpp/ \
    -I /opt/ros/humble/include/rcl/ \
    -I /opt/ros/humble/include/rcutils/ \
    -I /opt/ros/humble/include/rmw \
    -I /opt/ros/humble/include/rcl_yaml_param_parser/ \
    -I /opt/ros/humble/include/rosidl_runtime_c \
    -I /opt/ros/humble/include/rosidl_typesupport_interface \
    -I /opt/ros/humble/include/rcpputils \
    -I /opt/ros/humble/include/builtin_interfaces \
    -I /opt/ros/humble/include/rosidl_runtime_cpp \
    -I /opt/ros/humble/include/tracetools \
    -I /opt/ros/humble/include/rcl_interfaces \
    -I /opt/ros/humble/include/libstatistics_collector \
    -I /opt/ros/humble/include/statistics_msgs \
    -L /opt/ros/humble/lib/ \
    -lrclcpp -lrcutils \
    -o first_node

# 顺便小鱼加个clean指令，用来删掉first_node
```

```
clean:
    rm first_node
```

3. 编译

在Makefile同级目录输入

```
make build
```

```
root@490925f19143:~/d2lros2/d2lros2/chapt2/basic# make build
g++ first_ros2_node.cpp \
-I/opt/ros/humble/include/rclcpp/ \
-I /opt/ros/humble/include/rcl/ \
-I /opt/ros/humble/include/rcutils/ \
-I /opt/ros/humble/include/rmw \
-I /opt/ros/humble/include/rcl_yaml_param_parser/ \
-I /opt/ros/humble/include/rosidl_runtime_c \
-I /opt/ros/humble/include/rosidl_typesupport_interface \
-I /opt/ros/humble/include/rcpputils \
-I /opt/ros/humble/include/builtin_interfaces \
-I /opt/ros/humble/include/rosidl_runtime_cpp \
-I /opt/ros/humble/include/tracetools \
-I /opt/ros/humble/include/rcl_interfaces \
-I /opt/ros/humble/include/libstatistics_collector \
-I /opt/ros/humble/include/statistics_msgs \
-L /opt/ros/humble/lib/ \
-lrclcpp -lrcutils \
-o first_node
root@490925f19143:~/d2lros2/d2lros2/chapt2/basic# ls
Makefile  first_node  first_ros2_node.cpp
```

可以看到make指令调用了脚本里的build下的指令，对代码进行了编译。同级目录下也产生了first_node可执行文件（绿色代表可执行）。

使用 `make clean` 指令即可删掉 `first_node` 节点。

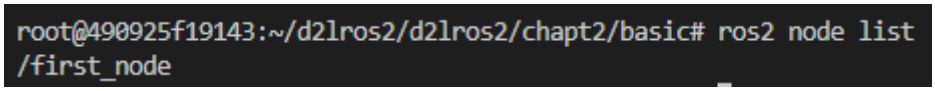
```
root@490925f19143:~/d2lros2/d2lros2/chapt2/basic# ls
Makefile  first_node  first_ros2_node.cpp
root@490925f19143:~/d2lros2/d2lros2/chapt2/basic# make clean
rm first_node
root@490925f19143:~/d2lros2/d2lros2/chapt2/basic# ls
Makefile  first_ros2_node.cpp
```

4.运行测试

```
./first_node
```

新开终端

```
ros2 node list
```



3.使用CMakeLists.txt编译ROS2节点

虽然通过make调用Makefile编译代码非常的方便，但是还是需要我们手写gcc指令来编译，那有没有什么办法可以自动生成Makefile呢？

答案是有的，那就是cmake工具。

cmake通过调用CMakeLists.txt直接生成Makefile。

1.安装Cmake

```shell
sudo apt install cmake
```

2.新建CMakeLists.txt

在 `d2lros2/d2lros2/chapt2/basic` 新建 `CMakeLists.txt` ，输入下面内容。

```cmake
cmake_minimum_required(VERSION 3.22)

project(first_node)

#include_directories 添加特定的头文件搜索路径 ，相当于指定g++编译器的-I参数
include_directories(/opt/ros/humble/include/rclcpp/)
include_directories(/opt/ros/humble/include/rcl/)
include_directories(/opt/ros/humble/include/rcutils/)
include_directories(/opt/ros/humble/include/rcl_yaml_param_parser/)
include_directories(/opt/ros/humble/include/rosidl_runtime_c/)
include_directories(/opt/ros/humble/include/rosidl_typesupport_interface/)
include_directories(/opt/ros/humble/include/rcpputils/)
include_directories(/opt/ros/humble/include/builtin_interfaces/)
include_directories(/opt/ros/humble/include/rmw/)
include_directories(/opt/ros/humble/include/rosidl_runtime_cpp/)
include_directories(/opt/ros/humble/include/tracetools/)
include_directories(/opt/ros/humble/include/rcl_interfaces/)
include_directories(/opt/ros/humble/include/libstatistics_collector/)
include_directories(/opt/ros/humble/include/statistics_msgs/)
```

```
# link_directories - 向工程添加多个特定的库文件搜索路径，相当于指定g++编译器的-L参数
link_directories(/opt/ros/humble/lib/)

# add_executable - 生成first_node可执行文件
add_executable(first_node first_ros2_node.cpp)

# target_link_libraries - 为first_node(目标) 添加需要动态链接库，相同于指定g++编译器-l参数
# 下面的语句代替 -lrclcpp -lrcutils
target_link_libraries(first_node rclcpp rcutils)
```

3.编译代码

我们一般会创建一个新的目录，运行cmake并进行编译，这样的好处是不会显得那么乱。

```shell
mkdir build
cd build
```

创建好文件夹，接着运行cmake指令，`..` 代表到上级目录找 `CMakeLists.txt` 。

```
cmake ..
```

运行完cmake你应该可以在build目录下看到cmake自动生成的Makefile了，接着就可以运行make指令进行编译

```
make
```

运行完上面的指令，就可以在build目录下发现 `first_node` 节点了。

```
root@490925f19143:~/d2lros2/d2lros2/chapt2/basic# mkdir build
root@490925f19143:~/d2lros2/d2lros2/chapt2/basic# cd build/
root@490925f19143:~/d2lros2/d2lros2/chapt2/basic/build# cmake ..
-- The C compiler identification is GNU 11.2.0
-- The CXX compiler identification is GNU 11.2.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /root/d2lros2/d2lros2/chapt2/basic/build
root@490925f19143:~/d2lros2/d2lros2/chapt2/basic/build# make
[ 50%] Building CXX object CMakeFiles/first_node.dir/first_ros2_node.cpp.o
[100%] Linking CXX executable first_node
[100%] Built target first_node
root@490925f19143:~/d2lros2/d2lros2/chapt2/basic/build# ls
CMakeCache.txt  CMakeFiles  Makefile  cmake_install.cmake  first_node
```

4.CMake依赖查找流程

上面我们用g++、make、cmake三种方式来编译ros2的C++节点。用cmake虽然成功了，但是CMakeLists.txt的内容依然非常的臃肿，我们需要将其进一步的简化。

1.优化CMakeList.txt

将上面的CmakLists.txt改成下面的样子

```cmake
cmake_minimum_required(VERSION 3.22)
project(first_node)

find_package(rclcpp REQUIRED)
add_executable(first_node first_ros2_node.cpp)
target_link_libraries(first_node rclcpp::rclcpp)
```

接着继续生成和编译

```
cmake ..
make
```

```
root@490925f19143:~/d2lros2/d2lros2/chapt2/basic/build# cmake ..
-- Found rclcpp: 16.0.1 (/opt/ros/humble/share/rclcpp/cmake)
-- Found rosidl_generator_c: 3.1.3 (/opt/ros/humble/share/rosidl_generator_c/cmake)
-- Found rosidl_adapter: 3.1.3 (/opt/ros/humble/share/rosidl_adapter/cmake)
-- Found rosidl_generator_cpp: 3.1.3 (/opt/ros/humble/share/rosidl_generator_cpp/cmake)
-- Using all available rosidl_typesupport_c: rosidl_typesupport_fastrtps_c;rosidl_typesupport_introspection_c
-- Using all available rosidl_typesupport_cpp: rosidl_typesupport_fastrtps_cpp;rosidl_typesupport_introspection_cpp
-- Found rmw_implementation_cmake: 6.1.0 (/opt/ros/humble/share/rmw_implementation_cmake/cmake)
-- Found rmw_fastrtps_cpp: 6.2.1 (/opt/ros/humble/share/rmw_fastrtps_cpp/cmake)
-- Using RMW implementation 'rmw_fastrtps_cpp' as default
-- Configuring done
-- Generating done
-- Build files have been written to: /root/d2lros2/d2lros2/chapt2/basic/build
root@490925f19143:~/d2lros2/d2lros2/chapt2/basic/build# make
Consolidate compiler generated dependencies of target first_node
[100%] Built target first_node
```

是不是非常的神奇，为什么可以浓缩成那么短的几句指令呢？

2.find_package查找路径

find_package查找路径对应的环境变量如下。

```
<package>_DIR
CMAKE_PREFIX_PATH
CMAKE_FRAMEWORK_PATH
CMAKE_APPBUNDLE_PATH
PATH
```

打开终端，输入指令：

```shell
echo $PATH
```

结果

```
PATH=/opt/ros/humble/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/
bin
```

观察PATH变量，你会发现 `/opt/ros/humble/bin` 赫然在其中， `PATH` 中的路径如果以 `bin` 或 `sbin` 结尾，则自动回退到上一级目录，接着检查这些目录下的

```text
<prefix>/(lib/<arch>|lib|share)/cmake/<name>*/          (U)
<prefix>/(lib/<arch>|lib|share)/<name>*/                (U)
<prefix>/(lib/<arch>|lib|share)/<name>*/(cmake|CMake)/  (U)
```

cmake找到这些目录后，会开始依次找 `<package>Config.cmake` 或 `Find<package>.cmake` 文件。找到后即可执行该文件并生成相关链接信息。

打开 `/opt/ros/humble/share/rclcpp/cmake` 你会发现 `rclcppConfig.cmake` 就在其中。

3.总结

本节小鱼带你通过多种方式进行节点的编译，主要是让你了解C++编译工具cmake以及其路径查找规则，以后在学习生涯中再遇到 `undefined reference to xxxxx` 和 `No such file or directory` 就再也不用慌

张了。

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

5.Python依赖查找流程

python的打包和引入依赖的方式相比C++要容易太多。本节小鱼带你来通过几个实例学习下Python的路径查找机制。

1.编写ROS2的Python节点

在 `d2lros2/d2lros2/chapt2/basic` 新建 `second_ros2_node.py` ，输入下面的内容

```python
# 导入rclpy库，如果Vscode显示红色的波浪线也没关系
# 我们只是把VsCode当记事本而已，谁会在意记事本对代码的看法呢，不是吗？
import rclpy
from rclpy.node import Node
# 调用rclcpp的初始化函数
rclpy.init()
# 调用rclcpp的循环运行我们创建的second_node节点
rclpy.spin(Node("second_node"))
```

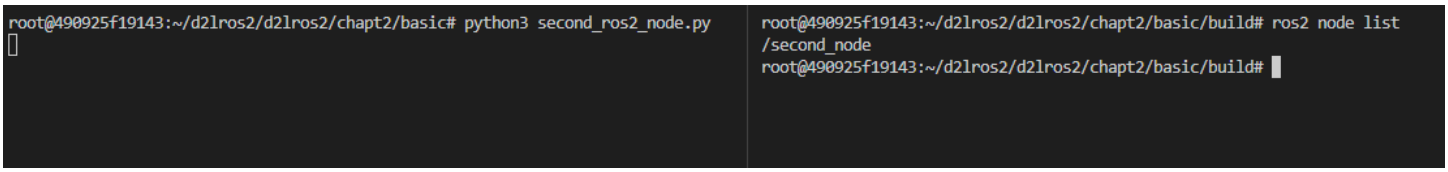
2.运行Python节点

打开终端，输入指令

```
ls
python3 second_ros2_node.py
```

打开新的终端，输入

```shell
ros2 node list
```



完美，四行代码写了个ROS2的Python节点。

那么问题来了，我们import rclpy，rclpy到底在哪里？python是如何找到的？

3.Python包查找流程

Python3运行 `import rclpy` 时候如何找到它的呢？答案是通过环境变量 `PYTHONPATH`

Ctrl+C打断节点运行，接着输入下面指令

```shell
echo $PYTHONPATH
```

结果

```python
/opt/ros/humble/lib/python3.10/site-
packages:/opt/ros/humble/local/lib/python3.10/dist-packages
```

你会发现里面有关于humble的python路径，在上面两个目录下找一下rclpy，看看能不能找到rclpy

查找第一个路径

```
ls -l /opt/ros/humble/lib/python3.10/site-packages | grep rclpy
```

没找到，第二个

```
ls -l /opt/ros/humble/local/lib/python3.10/dist-packages/ | grep rclpy
```

找到了

```
drwxr-xr-x 1 root root 4096 Jun  3 04:45 rclpy
drwxr-xr-x 2 root root 4096 May 23 22:23 rclpy-3.3.4-py3.10.egg-info
```

4.删除路径实验

使用 `export` 指令可以重新修改环境变量的值，尝试修改掉 `PYTHONPATH` 中ROS 2相关内容后之后再运行代码，看看是否还可以导入 `rclpy` 。

```shell
export PYTHONPATH=/opt/ros/humble/lib/python3.10/site-packages
echo $PYTHONPATH #重新echo查看
python3 second_ros2_node.py
```

提示如下

```
root@490925f19143:~/d2lros2/d2lros2/chapt2/basic# python3 second_ros2_node.py
Traceback (most recent call last):
  File "/root/d2lros2/d2lros2/chapt2/basic/second_ros2_node.py", line 3, in
<module>
    import rclpy
ModuleNotFoundError: No module named 'rclpy'
```

请你记住这个报错信息 `ModuleNotFoundError: No module named 'xxx'` ，这也是你未来学习过程中可能会经常会遇到的。

下次遇到时，你会想起小鱼这篇文章，然后对它轻蔑的一笑，接着找到这个库所在的目录，把它加到环境里。

5.总结

本节小鱼带你学习了Python编写ROS2节点的方式，并带你认识了一个常见的错误 `ModuleNotFoundError: No module named 'xxx'` 。

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**

- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

2.Python打包工具之Setup

> 本文摘自：Python 之打包工具 setup.py 奔跑的大西吉的博客-CSDN博客 python setup 打包

本部分只做了解即可，我们平时用的并不多，因为python的依赖并不是靠setup来查找的，但是C++却靠着CmakeLists.txt进行查找。

1. 为什么需要对项目分发打包？

平常我们习惯了使用 pip 来安装一些第三方模块，这个安装过程之所以简单，是因为模块开发者为我们默默地为我们做了所有繁杂的工作，而这个过程就是 `打包` 。

打包，就是将你的源代码进一步封装，并且将所有的项目部署工作都事先安排好，这样使用者拿到后即装即用，不用再操心如何部署的问题（如果你不想对照着一堆部署文档手工操作的话）。

不管你是在工作中，还是业余准备自己写一个可以上传到 PyPI 的项目，你都要学会如何打包你的项目。

Python 发展了这么些年了，项目打包工具也已经很成熟了。他们都有哪些呢？

你可能听过 `distutils` 、 `distutils2` 、 `setuptools` 等等，好像很熟悉，却又很陌生，他们都是什么关系呢？

2. 包分发的始祖：distutils

`distutils` 是 Python 的一个标准库，从命名上很容易看出它是一个分发（distribute）工具（utlis），它是 Python 官方开发的一个分发打包工具，所有后续的打包工具，全部都是基于它进行开发的。

`distutils` 的精髓在于编写 setup.py，它是模块分发与安装的指导文件。

那么如何编写 setup.py 呢？我会在后面进行详细的解析。

你有可能没写过 setup.py ，但你绝对使用过 setup.py 来做一些事情，比如下面这条命令，我们经常用它来进行模块的安装。

```text
python setup.py install
```

这样的安装方法是通过源码安装，与之对应的是通过二进制软件包的安装，同样我也会在后面进行介绍。

3. 分发工具升级：setuptools

`setuptools` 是 distutils 增强版，不包括在标准库中。其扩展了很多功能，能够帮助开发者更好的创建和分发 Python 包。大部分 Python 用户都会使用更先进的 setuptools 模块。

distribute，或许你在其他地方也见过它，这里也提一下。

distribute 是 setuptools 有一个分支版本，分支的原因可能是有一部分开发者认为 setuptools 开发太慢了。但现在，distribute 又合并回了 setuptools 中。因此，我们可以认为它们是同一个东西。

还有一个大包分发工具是 **distutils2**，其试图尝试充分利用distutils，detuptools 和 distribute 并成为 Python 标准库中的标准工具。但该计划并没有达到预期的目的，且已经是一个废弃的项目。

因此，setuptools 是一个优秀的，可靠的 Python 包安装与分发工具。

4. 超详细讲解 setup.py 的编写？

打包分发最关键的一步是编写 `setup.py` 文件。

以下是一个 setup.py 简单的使用示例

```
from setuptools import setup, find_packages

setup(
    # 指定项目名称，我们在后期打包时，这就是打包的包名称，当然打包时的名称可能还会包含下面的版本号哟
~
    name="mytest",
    # 指定版本号
    version="1.0",
    author="flp",
    author_email="flepeng@163.com",
    # 这是对当前项目的一个描述
    description="这只是一次测试",

    # 项目主页
    url="http://iswbm.com/",

    # 你要安装的包，通过 setuptools.find_packages 找到当前目录下有哪些包
    packages=find_packages()
```

```
    # 指定包名，即你需要打包的包名称，要实际在你本地存在哟，它会将指定包名下的所有"*.py"文件进行打包哟，但不会递归去拷贝所有的子包内容。
    # 综上所述，我们如果想要把一个包的所有"*.py"文件进行打包，应该在packages列表写下所有包的层级关系哟~这样就开源将指定包路径的所有".py"文件进行打包!
    packages=['devops', "devops.dev", "devops.ops"],
)
```

setup 函数常用的参数如下：

参数	说明
name	包名称
version	包版本
author	程序的作者
author_email	程序的作者的邮箱地址
maintainer	维护者
maintainer_email	维护者的邮箱地址
url	程序的官网地址
license	程序的授权信息
description	程序的简单描述
long_description	程序的详细描述
platforms	程序适用的软件平台列表
classifiers	程序的所属分类列表
keywords	程序的关键字列表
packages	需要处理的包目录(通常为包含 **init**.py 的文件夹)
py_modules	需要打包的 Python 单文件列表
download_url	程序的下载地址
cmdclass	添加自定义命令
package_data	指定包内需要包含的数据文件
include_package_data	自动包含包内所有受版本控制(cvs/svn/git)的数据文件
exclude_package_data	当 include_package_data 为 True 时该选项用于排除部分文件
data_files	打包时需要打包的数据文件，如图片，配置文件等
ext_modules	指定扩展模块
scripts	指定可执行脚本,安装时脚本会被安装到系统 PATH 路径下

package_dir	指定哪些目录下的文件被映射到哪个源码包
entry_points	动态发现服务和插件，下面详细讲
python_requires	指定运行时需要的Python版本
requires	指定依赖的其他包
provides	指定可以为哪些模块提供依赖
install_requires	应用于指定项目正确运行**所需的最低**要求
extras_require	当前包的高级/额外特性需要依赖的分发包
tests_require	在测试时需要使用的依赖包
setup_requires	指定运行 setup.py 文件本身所依赖的包
dependency_links	指定依赖包的下载地址
zip_safe	不压缩包，而是以目录的形式安装

推荐资料：

- Python 打包用户指南 — Python 打包用户指南

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

1.ROS2节点介绍

本节小鱼正式的来介绍节点的概念。

1. ROS2节点是什么

ROS2中每一个节点也是只负责一个单独的模块化的功能（比如一个节点负责控制车轮转动，一个节点负责从激光雷达获取数据、一个节点负责处理激光雷达的数据、一个节点负责定位等等）



2.节点之间如何交互？

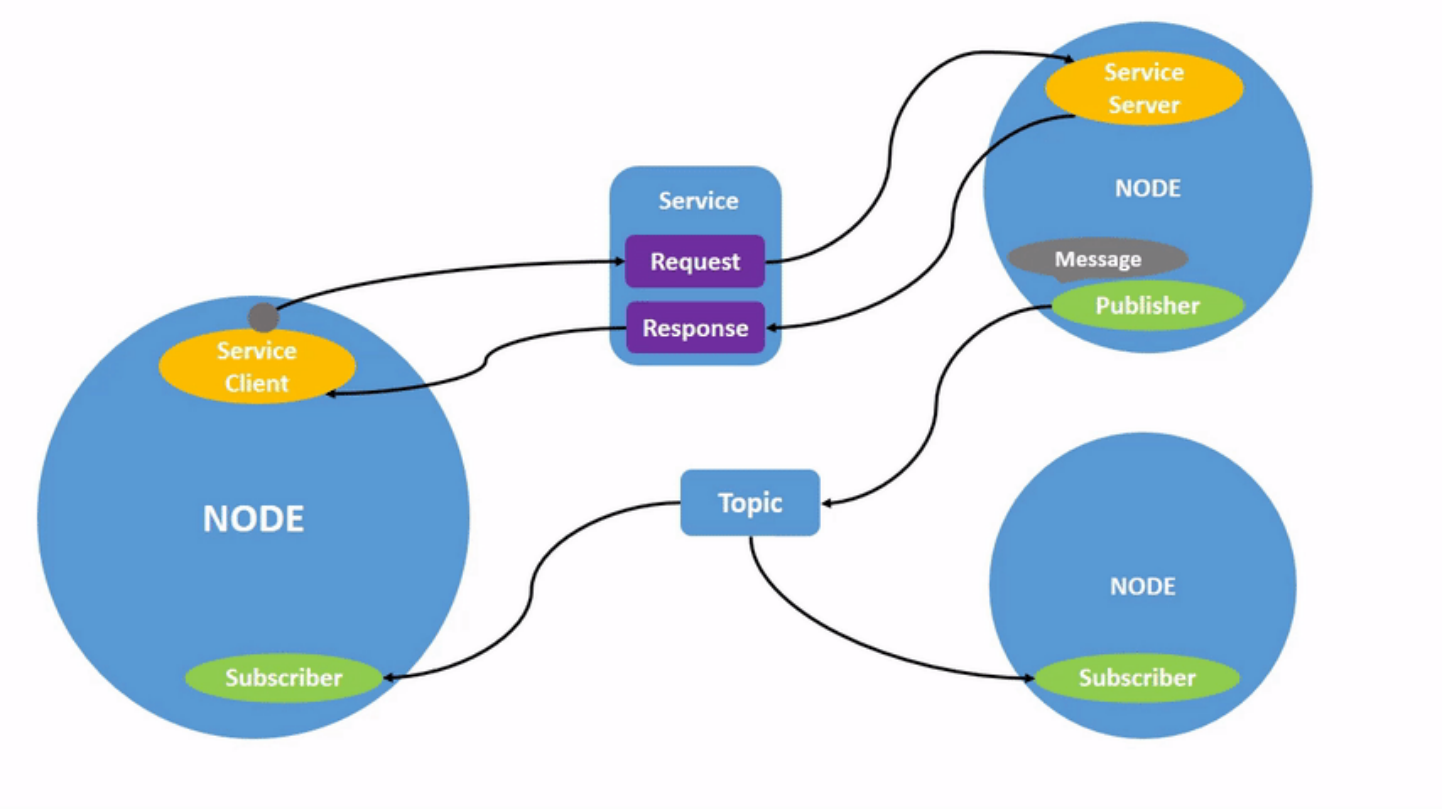
上面举了一个激光雷达的例子，一个节点负责获取激光雷达的扫描数据，一个节点负责处理激光雷达数据，比如去除噪点。

那节点与节点之间就必须要通信了，那他们之间该如何通信呢？ROS2早已为你准备好了一共四种通信方式:

- 话题-topics
- 服务-services
- 动作-Action
- 参数-parameters

这四种种通信方式的用途和使用方法，小鱼放到了第四和第五章来介绍，到时候同时会带大家手撸代码。

官方给了一张图，大家先大概看一下，帮助理解



3. 如何启动一个节点？

知道了节点的概念之后，我们该如何启动一个节点呢？

因为工作空间和包的概念，小鱼放到了下一讲，这里大家跟着小鱼一起运行一个节点，感受一下。

使用指令：

```
ros2 run <package_name> <executable_name>
```

指令意义：启动 包下的 中的节点。

使用样例：

```
ros2 run turtlesim turtlesim_node
```

大家可以尝试一下上面的指令，就是我们在第一章中启动小乌龟模拟器的那条指令。

运行之后可以看到一只小乌龟，接下来就可以试试下一节中提到的几个指令来查看节点信息和列表。

4. 通过命令行界面查看节点信息

4.1 ROS2命令行

ROS2的CLI，就是和ROS2相关的命令行操作。什么是命令行界面呢？这里小鱼再讲解一个概念，CLI（Command-Line Interface）和GUI（Graphical User Interface）

- GUI（Graphical User Interface）就是平常我们说的图形用户界面，大家用的Windows是就是可视化的，我们可以通过鼠标点击按钮等图形化交互完成任务。
- CLI（Command-Line Interface）就是命令行界面了，我们所用的终端，黑框框就是命令行界面，没有图形化。

很久之前电脑还是没有图形化界面的，所有的交互都是通过命令行实现，就学习机器人而言，命令行操作相对于图形化优势更加明显。

ROS2为我们提供了一系列指令，通过这些指令，可以实现对ROS2相关模块信息的获取设置等操作。

4.2 节点相关的CLI

运行节点(常用)

```
ros2 run <package_name> <executable_name>
```

查看节点列表(常用)：

```
ros2 node list
```

查看节点信息(常用)：

```
ros2 node info <node_name>
```

重映射节点名称

```
ros2 run turtlesim turtlesim_node --ros-args --remap __node:=my_turtle
```

运行节点时设置参数

```
ros2 run example_parameters_rclcpp parameters_basic --ros-args -p
rcl_log_level:=10
```

5.总结

通过本节课的学习，希望大家对节点的概念有所了解，下一节小鱼带你一起学习工作空间的相关概念

- ROS2命令行工具源码;ros2/ros2cli: ROS 2 command line interface tools (github.com)

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

2.ROS2工作空间

大家好，我是小鱼~上一节小鱼给大家介绍了一下节点，运行一个节点的时候使用的是

```
ros2 run 包名字 可执行文件名字
```

那你有没有想过，我们想找到一个节点（可执行文件），就必须要先知道它在哪个包，那问题就来了，想要找到某个包，该去哪里找？

答案就是：工作空间

> **注意：一个工作空间下可以有多个功能包，一个功能包可以有多个节点存在**

1. 工作空间

小鱼定义：工作空间是包含若干个功能包的目录，一开始大家把工作空间理解成一个文件夹就行了。这个文件夹包含下有 `src` 。所以一般新建一个工作空间的操作就像下面一样

```shell
cd d2lros2/chapt2/
mkdir -p chapt2_ws/src
```

是不是觉得就像创建一个目录（其实就是创建一个目录）

2.功能包是什么

功能包可以理解为存放节点的地方，ROS2中功能包根据编译方式的不同分为三种类型。

- ament_python，适用于python程序
- cmake，适用于C++
- ament_cmake，适用于C++程序,是cmake的增强版

后面小鱼会带你一起手把手创建功能包~

3.功能包获取的两种方式

3.1 安装获取

安装一般使用

```
sudo apt install ros-<version>-package_name
```

安装获取会自动放置到系统目录，不用再次手动source。

3.2 手动编译获取

手动编译相对麻烦一些，需要下载源码然后进行编译生成相关文件。

> 什么时候需要手动编译呢?一般我们能安装的功能包都是作者编译好程序将可执行文件上传到仓库中，然后我们才能够通过apt进行安装，如果作者还没来得及测试上传，或者忘记了测试上传，就会找不到对应的包，这时候就需要手动编译安装了。
>
> 另外一种就是我们需要对包的源码进行修改，这个时候也需要自己编译修改。

手动编译之后，需要手动source工作空间的install目录。

下一节学习完编译器colcon小鱼会通过实例带大家一起下载编译安装功能包~

4.与功能包相关的指令 ros2 pkg

```
create       Create a new ROS2 package
executables  Output a list of package specific executables
list         Output a list of available packages
prefix       Output the prefix path of a package
xml          Output the XML of the package manifest or a specific tag
```

1.创建功能包

```
ros2 pkg create <package-name>  --build-type  {cmake,ament_cmake,ament_python}  --
```

```
dependencies <依赖名字>
```

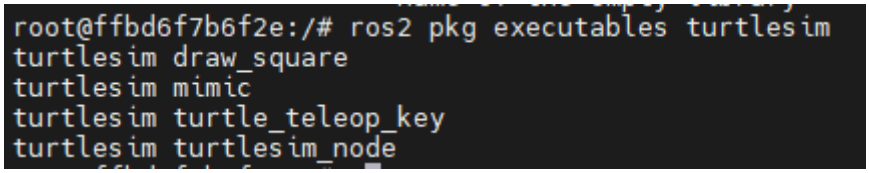
2.列出可执行文件

列出所有

```
ros2 pkg executables
```

列出`turtlesim`功能包的所有可执行文件

```
ros2 pkg executables turtlesim
```



3.列出所有的包

```
ros2 pkg list
```

4.输出某个包所在路径的前缀

```
ros2 pkg prefix  <package-name>
```

比如小乌龟

```
ros2 pkg prefix turtlesim
```

5.列出包的清单描述文件

> 每一个功能包都有一个标配的manifest.xml文件，用于记录这个包的名字，构建工具，编译信息，拥有者，干啥用的等信息。
>
> 通过这个信息，就可以自动为该功能包安装依赖，构建时确定编译顺序等

查看小乌龟模拟器功能包的信息。

```
ros2 pkg xml turtlesim
```

5.总结

介绍完工作空间和功能包，接下来就可以讲讲ROS2的编译工具colcon，下一讲我们就开始对代码动手了~

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

3. ROS2构建工具—Colcon

大家好，我是编译代码特别慢的小鱼。本节课我们来讲一下ROS2编译工具 `colcon`

本节小鱼会从下面几个方面来介绍：

1. Colcon是个啥
2. 安装colcon
3. 编个东西测试一下
4. 运行一个自己编的节点
5. colcon学习总结指令

1.Colcon是个啥

colcon其实是一个功能包构建工具，这个工具用来做什么的呢？

简单点说就是用来编译代码的，上几节跟大家讲了如何进行ROS2工作空间的创建，但没有说如何进行编译，其实就是用colcon。

ROS2默认是没有安装colcon的，所以小鱼就从如何安装colcon开始跟大家讲解colcon的使用方法。

> colcon想当于ros1中的catkin工具，学过ros1的同学可以辅助理解。没学过也没关系，用多了自然也就懂了。

2.安装colcon

如果使用一键安装ROS2，会帮你安装好这个工具，以防万一我们再装一次，打开终端复制粘贴进去即可。

```
sudo apt-get install python3-colcon-common-extensions
```

安装完成后，打开终端输入 `colcon` 即可看到其使用方法。

3. 编个东西测试一下

1. 创建一个工作区文件夹 `colcon_test_ws`

```
cd d2lros2/chapt2/
mkdir colcon_test_ws && cd colcon_test_ws
```

2. 下载个ROS2示例源码测试一下

```
git clone https://github.com/ros2/examples src/examples -b humble
```

> 如果这步克隆错误，你可能需要一个梯子翻个墙：点击这里试试小鱼用了很多年的梯子

3. 编译工程

```
colcon build
```

```
ros2@ubuntu:~/code/colcon_test$ colcon build
Starting >>> examples_rclcpp_minimal_action_client
Starting >>> examples_rclcpp_minimal_action_server
Starting >>> examples_rclcpp_minimal_client
Starting >>> examples_rclcpp_minimal_composition
Finished <<< examples_rclcpp_minimal_client [15.6s]
Starting >>> examples_rclcpp_minimal_publisher
Finished <<< examples_rclcpp_minimal_action_server [16.2s]
Starting >>> examples_rclcpp_minimal_service
Finished <<< examples_rclcpp_minimal_action_client [16.7s]
Starting >>> examples_rclcpp_minimal_subscriber
Finished <<< examples_rclcpp_minimal_composition [23.5s]
Starting >>> examples_rclcpp_minimal_timer
Finished <<< examples_rclcpp_minimal_service [7.47s]
Starting >>> examples_rclcpp_multithreaded_executor
Finished <<< examples_rclcpp_minimal_publisher [8.51s]
Starting >>> examples_rclpy_executors
Finished <<< examples_rclpy_executors [1.30s]
Starting >>> examples_rclpy_minimal_action_client
Finished <<< examples_rclpy_minimal_action_client [0.77s]
Starting >>> examples_rclpy_minimal_action_server
Finished <<< examples_rclpy_minimal_action_server [1.03s]
Starting >>> examples_rclpy_minimal_client
Finished <<< examples_rclpy_minimal_client [0.59s]
Starting >>> examples_rclpy_minimal_publisher
Finished <<< examples_rclcpp_minimal_subscriber [11.3s]
Starting >>> examples_rclpy_minimal_service
Finished <<< examples_rclpy_minimal_publisher [0.56s]
Starting >>> examples_rclpy_minimal_subscriber
Finished <<< examples_rclpy_minimal_service [0.61s]
Finished <<< examples_rclcpp_minimal_timer [5.22s]
Finished <<< examples_rclpy_minimal_subscriber [0.49s]
Finished <<< examples_rclcpp_multithreaded_executor [7.82s]

Summary: 16 packages finished [31.9s]
```

> 如果在编译中遇到 `Setuptools DeprecationWarning: setup.py install is deprecated.` 这个警告，可以通过更新setuptools解决。
>
> 详细操作见社区帖子：https://fishros.org.cn/forum/topic/254/

4. 编完之后的目录结构

构建完成后，在 `src` 同级目录我们应该会看到 `build` 、 `install` 和 `log` 目录:

```
.
├── build
├── install
├── log
└── src

4 directories, 0 files
```

- `build` 目录存储的是中间文件。对于每个包，将创建一个子文件夹，在其中调用例如CMake
- `install` 目录是每个软件包将安装到的位置。默认情况下，每个包都将安装到单独的子目录中。
- `log` 目录包含有关每个colcon调用的各种日志信息。

4.运行一个自己编的节点

1. 打开一个终端使用 cd colcon_test_ws进入我们刚刚创建的工作空间，先source 一下资源

```
source install/setup.bash
```

2. 运行一个订者节点，你将看不到任何打印，因为没有发布者

```
ros2 run examples_rclcpp_minimal_subscriber subscriber_member_function
```

3. 打开一个新的终端，先source，再运行一个发行者节点

```
source install/setup.bash
ros2 run examples_rclcpp_minimal_publisher publisher_member_function
```

```
ros2@ubuntu:~/code/colcon_test$ ros2 run examples_rclcpp_minimal_subscriber subscriber_member_function
[INFO] [1626787140.141413631] [minimal_subscriber]: I heard: 'Hello, world! 0'
[INFO] [1626787140.641108755] [minimal_subscriber]: I heard: 'Hello, world! 1'
[INFO] [1626787141.141173727] [minimal_subscriber]: I heard: 'Hello, world! 2'
[INFO] [1626787141.640650987] [minimal_subscriber]: I heard: 'Hello, world! 3'
[INFO] [1626787142.141448463] [minimal_subscriber]: I heard: 'Hello, world! 4'
[INFO] [1626787142.640612468] [minimal_subscriber]: I heard: 'Hello, world! 5'
[INFO] [1626787143.141255315] [minimal_subscriber]: I heard: 'Hello, world! 6'
```

5.本节学习指令

> 这个小鱼要特别说一下，因为ros2的build没有ros中的devel概念了，如果想达到devel目录那样的效果，就需要加这个参数。没有学过ros的请主动忽略这句话。

5.1 只编译一个包

```
colcon build --packages-select YOUR_PKG_NAME
```

5.2 不编译测试单元

```
colcon build --packages-select YOUR_PKG_NAME  --cmake-args -DBUILD_TESTING=0
```

5.3 运行编译的包的测试

```
colcon test
```

5.4 允许通过更改src下的部分文件来改变install（重要）

每次调整 python 脚本时都不必重新build了

```
colcon build --symlink-install
```

6.总结

通过本节课的学习，相信你已经掌握如何编译ROS2的工程了，先别着急结束，还有一个小作业~

参考资料:

- colcon官方文档 https://colcon.readthedocs.io/en/released/user/installation.html
- ROS2官网文档 https://docs.ros.org/en/humble/Tutorials/Colcon-Tutorial.html

---

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

4.使用RCLCPP编写节点

节点需要存在于功能包当中、功能包需要存在于工作空间当中。所以我们要想创建节点，就要先创建一个工作空间，再创建功能包。

1.创建工作空间和功能包

1.1 工作空间

工作空间就是文件夹，所以很简单。

```
cd d2lros2/chapt2/
mkdir -p chapt2_ws/src/
```

1.2 创建example_cpp功能包

创建example_cpp功能包，使用ament-cmake作为编译类型，并为其添加rclcpp依赖。

```
cd chapt2_ws/src
ros2 pkg create example_cpp --build-type ament_cmake --dependencies rclcpp
```

大家可以手写一下这个代码，感受一下。现在小鱼来讲一讲这条命令的含义和参数。

- pkg create 是创建包的意思
- --build-type 用来指定该包的编译类型，一共有三个可选项 `ament_python` 、 `ament_cmake` 、 `cmake`
- --dependencies 指的是这个功能包的依赖，这里小鱼给了一个ros2的python客户端接口 `rclpy`

打开终端，进入 `chapt2_ws/src` 运行上面的指令，创建完成后的目录结构如下：

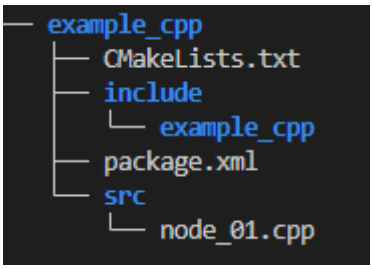
```
.
└── src
    └── example_cpp
        ├── CMakeLists.txt
        ├── include
```

```
│   └── example_cpp
├── package.xml
└── src

5 directories, 2 files
```

2.创建节点

接着我们在 `example_cpp/src` 下创建一个 `node_01.cpp` 文件，创建完成后的目录结构如下：



3.编写代码

3.1 编写代码

继续跟着小鱼一起输入代码，输入的时候可以边输边理解。

```cpp
#include "rclcpp/rclcpp.hpp"


int main(int argc, char **argv)
{
    /* 初始化rclcpp  */
    rclcpp::init(argc, argv);
    /*产生一个node_01的节点*/
    auto node = std::make_shared<rclcpp::Node>("node_01");
    // 打印一句自我介绍
    RCLCPP_INFO(node->get_logger(), "node_01节点已经启动.");
    /* 运行节点，并检测退出信号 Ctrl+C*/
    rclcpp::spin(node);
    /* 停止运行 */
```

```
    rclcpp::shutdown();
    return 0;
}
```

3.2 修改CmakeLists

在 `node_01.cpp` 中输入上面的内容后，还需要修改一下CMakeLists.txt。将其添加为可执行文件，并使用 `install` 指令将其安装到 `install` 目录。

在CmakeLists.txt最后一行加入下面两行代码。

```
add_executable(node_01 src/node_01.cpp)
ament_target_dependencies(node_01 rclcpp)
```

添加这两行代码的目的是让编译器编译node_01这个文件，接着在上面两行代码下面添加下面的代码。

shell

```
install(TARGETS
  node_01
  DESTINATION lib/${PROJECT_NAME}
)
```

2.编译运行节点

在 `chapt2_ws` 下依次输入下面的命令

2.1 编译节点

```
colcon build
```

2.2 source环境

```
source install/setup.bash
```

2.3 运行节点

```
ros2 run example_cpp node_01
```

不出意外，你可以看到

```
root@490925f19143:~/d2lros2/d2lros2/chapt2/chapt2_ws# colcon build
Starting >>> example_cpp
Finished <<< example_cpp [4.70s]

Summary: 1 package finished [4.84s]
root@490925f19143:~/d2lros2/d2lros2/chapt2/chapt2_ws# source install/setup.bash
root@490925f19143:~/d2lros2/d2lros2/chapt2/chapt2_ws# ros2 run example_cpp node_01
[INFO] [1654248314.147366428] [node_01]: node_01节点已经启动.
```

3.测试

当节点运行起来后，可以再尝试使用 `ros2 node list` 指令来查看现有的节点。这个时候你应该能看到：

```
root@490925f19143:~/d2lros2/d2lros2/chapt2/chapt2_ws# colcon build
Starting >>> example_cpp
Finished <<< example_cpp [4.70s]

Summary: 1 package finished [4.84s]
root@490925f19143:~/d2lros2/d2lros2/chapt2/chapt2_ws# source install/setup.bash
root@490925f19143:~/d2lros2/d2lros2/chapt2/chapt2_ws# ros2 run example_cpp node_01
[INFO] [1654248314.147366428] [node_01]: node_01节点已经启动.
```

```
root@490925f19143:~/d2lros2/d2lros2/chapt2/chapt2_ws# ros2 node list
/node_01
root@490925f19143:~/d2lros2/d2lros2/chapt2/chapt2_ws#
```

4.总结

至此，相信你已经掌握了如何编写一个C++版本的ros2节点了，但是这仅仅是编写ROS2节点方式之一，相比之下，小鱼更推荐你使用面向对象的方式编写节点，在进阶篇小鱼将会向你展示其写法。

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

5.使用RCLPY编写节点

1.创建C++功能包

创建一个名字叫做 `example_py` python版本的功能包。

```
cd chapt2/chapt2_ws/src/
ros2 pkg create example_py  --build-type ament_python --dependencies rclpy
```

创建完成后的目录结构

```
.
├── example_py
│   └── __init__.py
├── package.xml
├── resource
│   └── example_py
├── setup.cfg
├── setup.py
└── test
    ├── test_copyright.py
    ├── test_flake8.py
    └── test_pep257.py

3 directories, 8 files
```

2.编写程序

编写ROS2节点的一般步骤

```
1. 导入库文件
2. 初始化客户端库
3. 新建节点
```

```
    4. spin循环节点
    5. 关闭客户端库
```

在 `example_py/example_py` 下创建 `node_02.py` 接着我们开始编写代码。跟着小鱼一起边理解输入下面的代码，注释不用输。

```
import rclpy
from rclpy.node import Node

def main(args=None):
    """
    ros2运行该节点的入口函数
    编写ROS2节点的一般步骤
    1. 导入库文件
    2. 初始化客户端库
    3. 新建节点对象
    4. spin循环节点
    5. 关闭客户端库
    """
    rclpy.init(args=args) # 初始化rclpy
    node = Node("node_02")  # 新建一个节点
    node.get_logger().info("大家好，我是node_02.")
    rclpy.spin(node) # 保持节点运行，检测是否收到退出指令（Ctrl+C）
    rclpy.shutdown() # 关闭rclpy
```

代码编写完成用Crtl+S进行保存。接着修改 `setup.py` 。

```
    entry_points={
        'console_scripts': [
            "node_02 = example_py.node_02:main"
        ],
    },
)
```

`setup.py` 这段配置是声明一个ROS2的节点，声明后使用 `colcon build` 才能检测到，从而将其添加到 `install` 目录下。

完成上面的工作后，就可以编译运行了。

3.编译运行节点

打开vscode终端，进入 `chapt2/chapt2_ws/`

3.1 编译节点

```
cd chapt2/chapt2_ws/
colcon build
```

```
--- stderr: example_py
/usr/lib/python3/dist-packages/setuptools/command/install.py:34:
SetuptoolsDeprecationWarning: setup.py install is deprecated. Use build and pip
and other standards-based tools.
  warnings.warn(
---
```

如果在编译中看到上述错误没关系，不影响使用，ros2官方正在修复。 错误原因是setuptools版本太高造成，使用下面的指令可以进行版本的回退。

```
sudo pip install setuptools==58.2.0 --upgrade
```

3.2 source环境

```
source install/setup.bash
```

3.3 运行节点

```
ros2 run example_py node_02
```

运行结果

```
root@490925f19143:~/d2lros2/d2lros2/chapt2/chapt2_ws# colcon build
Starting >>> example_cpp
Starting >>> example_py
Finished <<< example_cpp [0.93s]
--- stderr: example_py
/usr/lib/python3/dist-packages/setuptools/command/install.py:34: SetuptoolsDeprecationWarning: set
ed tools.
  warnings.warn(
---
Finished <<< example_py [1.53s]

Summary: 2 packages finished [1.70s]
  1 package had stderr output: example_py
root@490925f19143:~/d2lros2/d2lros2/chapt2/chapt2_ws# source install/setup.bash
root@490925f19143:~/d2lros2/d2lros2/chapt2/chapt2_ws# ros2 run example_py node_02
[INFO] [1654249559.162166346] [node_02]: 大家好，我是node_02.
```

4.测试

当节点运行起来后，可以再尝试使用 `ros2 node list` 指令来查看现有的节点。这个时候你应该能看到：

```
root@490925f19143:~/d2lros2/d2lros2/chapt2/chapt2_ws# ros2 node list
/node_02
root@490925f19143:~/d2lros2/d2lros2/chapt2/chapt2_ws#
```

这说明你的节点已经运行起来了。

5.总结

本节我们学习了使用Python在工作空间的功能包里编写一个节点，代码是相同的，但是多了一些配置。

当然除了使用这种方法编写一个节点，还有其他方式，小鱼将其放到了进阶篇来讲。

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

3.4.3 OOP介绍

除了使用上节中的只定义一个main函数就完成编写一个Python节点外，还有另外两种方式。

本节小鱼就来讲一讲出现么多种编写节点的原因，并对其中较为重要的OOP方法进行介绍。

要做机器人离不开计算机编程，而计算机编程经过多年的发展，演变出了三种不同且常用的编程思想，分别是：

1. 面向过程编程思想。缩写：POP
2. 面向对象编程思想。缩写：OOP
3. 函数式思想。缩写：FP

1.Why

为什么了解这些编程思想呢？尤其是OOP。

小鱼遇到过很多同学，在阅读机器人相关开源程序代码时，比如导航框架Nav2、机械臂运动控制框架Moveit时发现，别人的代码，每一行好像都看得懂，但放一起就看不懂了，看别人函数调来调去，很快人就给整蒙了。不知道如何下手。

这其实就是对别人的编程思想不了解造成的，所以本节课小鱼就给提一提常见的三种编程思想，让大家脑子里有个概念，以后遇到了看不明白的程序，知道该往哪个方向去学习。

编程思想博大精深，小鱼这里只是对三种思想的基本介绍。

2.思想辨析

首先明确一件事情，编程是为了什么？

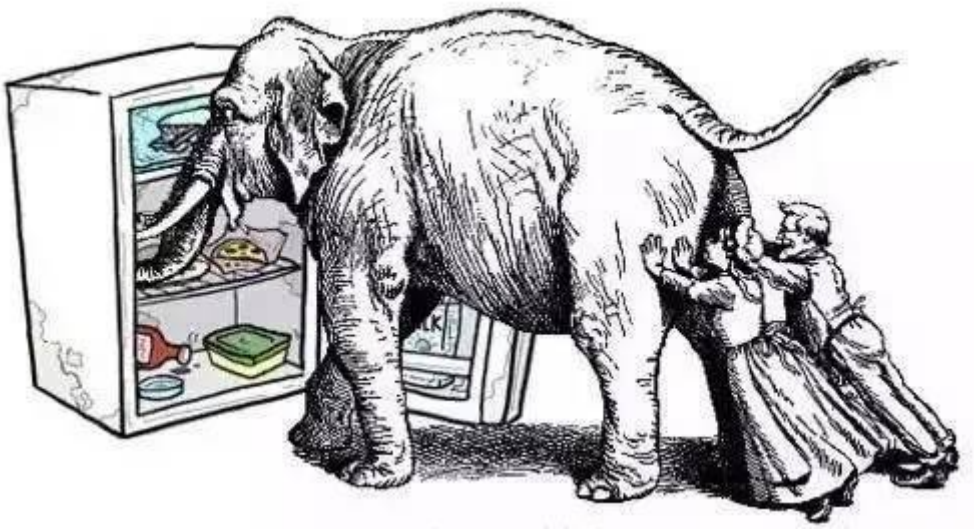
是为了赚钱吗？

不，我们写程序肯定是为了解决实际的问题的，那编程思想编程思想就是解决问题的思路（赚钱工具）

那这三种思想有什么区别呢？

2.1 用三种思想把大象装进冰箱

比如我们想把一只大象装进冰箱，分别用三种思想，我们看看有什么不一样。



2.1.1 面向过程思想

如果我们采用面向过程的思想，可以分为三步：

1. 打开冰箱门
2. 把大象塞进去
3. 关上冰箱门

面向过程编程就是分析出解决问题所需要步骤，然后分别实现每一步，再一步步执行即可。

2.1.2 面向对象思想

面向对象编程思想（OOP）怎么做呢？

那就要先知道面向对象是什么？搞清楚啥是对象？



对象是女朋友吗？答案肯定不是。

任何我们想要探究的事物都可以当作一个对象，比如我们可以把你家的冰箱理解为一个对象，我们就可以研究你家冰箱由哪些部分（指令装置等）组成，你家冰箱能干什么（制冷、调温等）？

接着我们开始下定义，就是取个高大上的名字

冰箱	定义	举例
冰箱的组成部分	冰箱的属性	制冷器，调温旋钮、灯带等
冰箱能干什么	冰箱的行为	制冷，调温、照明等

对象的行为其实是对其属性的操作，比如对制冷器操作就可以制冷，给灯带通电就可以照明。

对象 = 属性+行为

接着我们开始采用OOP的方法把大象装进冰箱

1. 调用：冰箱->打开门(行为)
2. 调用：冰箱->装东西(行为)
3. 调用：冰箱->关闭门(行为)

看起来和面向过程没啥区别，但我们的思想发生了重大的转变，我们把冰箱当作了一个独立的对象，我们是通过和冰箱这个对象交互完成了整个过程。

接着来看函数式编程

1. 定义关进（冰箱，大象）函数
2. 实现函数：关门(放入(开门(冰箱)，大象))

可以看到多层的函数嵌套调用，这就是函数编程的魅力，因为FP不是我们的机器人学习中的重点，这里就不过多讲解啦！

3.面向对象编程

简单介绍完，我们来说说今天的主角，面向对象编程OOP。

面向对象中有五个重要的概念，理解这五个概念相当于对OOP编程有了了解，下面小鱼一个个来介绍。

3.1 类与对象（抽象与具体）

我们通过调用你家美的冰箱的开门、装东西和关门三个行为来把大象装进冰箱。这时我们可以把`你家的美的冰箱`（具体的）称之为一个对象，而冰箱（抽象的）就称为一个类。

比如说鱼类和小鱼，鱼类就是一个类，而小鱼就是鱼类（抽象的）中的一个对象（具体的）。

在ROS2设计时这种抽象和具体的思想发挥着非常重要的作用，比如说DDS是有很多厂家的，ROS2为了匹配不同厂家的DDS，就设计除了DDS抽象层，而每一个具体的DDS厂家，我们可以称之为一个DDS的对象，是具体的。

3.2 封装、继承与多态

所谓封装就是将属性和行为封装在一起。上面已经介绍了**对象 = 属性+行为**，比如冰箱将冰箱的温度值（属性）和对温度值的操作（行为）等封装在一起。

继承，继承可以帮我们减少很多的工作量（比如王撕聪从他爹那里继承了很多钱，这样他就少奋斗了很多年），比如ROS2中的执行器类，通过继承执行器类实现了单线程执行器和多线程执行器，更多具体的例子我们在后续的学习中遇到再说。

多态，其实很简单，我们可以说鲤鱼是鱼类，草鱼是鱼类，鲤鱼是鱼类。同一个鱼类可以有多种不同的类型，即多态。更多的用法，等到写代码的时候再和小鱼一起解锁

4.如何选择code思想

三种编程思想，我们写程序的时候该如何选择呢?

小鱼个人的拙见是根据你的功能需求来，如果只需要实现一个很简单的功能，比如只是做一个键盘控制器，实现控制小车前进后退，直接采用面向过程的设计思想即可。

但如果是做一个稍大的工程，且后续要考虑功能的拓展性，这个时候就需要采用面向对象的思路来了。

参考链接：

-浅谈面向对象的编程思想：如何优雅地把大象装进冰箱？ SYSU 101的博客-CSDN博客

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

2.使用面向对象方式编写ROS2节点

1.C++版本

在 `d2lros2/chapt2/chapt2_ws/src/example_cpp/src` 下新建 `node_03.cpp` ，接着输入下面的代码。

```c++
#include "rclcpp/rclcpp.hpp"

/*
    创建一个类节点，名字叫做Node03,继承自Node.
*/
class Node03 : public rclcpp::Node
{

public:
    // 构造函数,有一个参数为节点名称
    Node03(std::string name) : Node(name)
    {
        // 打印一句
        RCLCPP_INFO(this->get_logger(), "大家好，我是%s.",name.c_str());
    }

private:

};

int main(int argc, char **argv)
{
    rclcpp::init(argc, argv);
    /*产生一个node_03的节点*/
    auto node = std::make_shared<Node03>("node_03");
    /* 运行节点，并检测退出信号*/
    rclcpp::spin(node);
    rclcpp::shutdown();
    return 0;
}
```

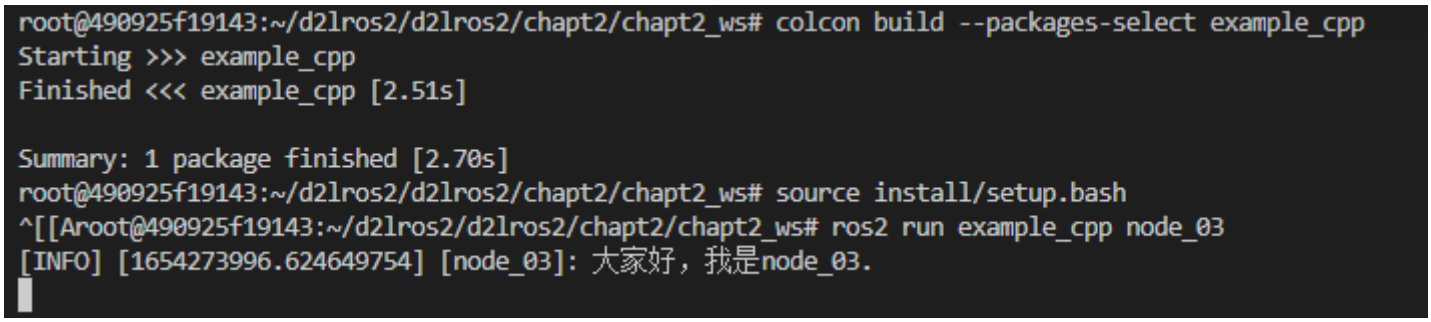
接着修改`CMakeLists.txt`，添加下方代码。

```
add_executable(node_03 src/node_03.cpp)
ament_target_dependencies(node_03 rclcpp)

install(TARGETS
  node_03
  DESTINATION lib/${PROJECT_NAME}
)
```

接着即可自行编译测试

```shell
colcon build --packages-select example_cpp
source install/setup.bash
ros2 run example_cpp node_03
```



2.Python版本

在`d2lros2/d2lros2/chapt2/chapt2_ws/src/example_py/example_py`下新建`node_04.py`，输入下面的代码

```python
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node


class Node04(Node):
    """
```

```
        创建一个Node04节点，并在初始化时输出一个话
        """
        def __init__(self,name):
            super().__init__(name)
            self.get_logger().info("大家好，我是%s!" % name)


    def main(args=None):
        rclpy.init(args=args) # 初始化rclpy
        node = Node04("node_04")  # 新建一个节点
        rclpy.spin(node) # 保持节点运行，检测是否收到退出指令（Ctrl+C）
        rclpy.shutdown() # 关闭rclpy
```

接着修改 `setup.py`

```python
        entry_points={
            'console_scripts': [
                "node_02 = example_py.node_02:main",
                "node_04 = example_py.node_04:main"
            ],
        },
```

> 注意格式和结尾的 `,` 符号， `console_scripts` 是个数组。

编译测试

```shell
    colcon build --packages-select example_py
    source install/setup.bash
    ros2 run example_py node_04
```

```
root@490925f19143:~/d2lros2/d2lros2/chapt2/chapt2_ws# colcon build --packages-select example_py
Starting >>> example_py
--- stderr: example_py
/usr/lib/python3/dist-packages/setuptools/command/install.py:34: SetuptoolsDeprecationWarning: setup.py inst
all is deprecated. Use build and pip and other standards-based tools.
  warnings.warn(
---
Finished <<< example_py [1.68s]

Summary: 1 package finished [1.85s]
  1 package had stderr output: example_py
root@490925f19143:~/d2lros2/d2lros2/chapt2/chapt2_ws# source install/setup.bash
root@490925f19143:~/d2lros2/d2lros2/chapt2/chapt2_ws# ros2 run example_py node_04
[INFO] [1654274207.562313029] [node_04]: 大家好，我是node_04!
```

3.总结

把节点写成一个类的形式对我们组织代码和使用ROS2的新特性有很多的好处，后面我们将以此种方式（用类建立节点）来学习后续内容。

3.Colcon使用进阶

基础篇中小鱼带你用gcc编译了ROS2节点。对你来说，使用CMake（GCC或Makefile）和 Python Setup打包工具依然可以完成ROS2代码的编译，那为什么还需要Colcon呢？

带着这个问题，我们来进一步的学习Colcon。

1.ROS生态中的构建系统和构建工具

1.1 构建系统与构建工具

两者的区分点在于针对的对象不同，构建系统之针对一个单独的包进行构建，而构建工具重点在于按照依赖关系依次调用构建系统完成一系列功能包的构建。

ROS中用到的构建系统：`CMake` 、`ament_cmake` 、`catkin` 、`Python setuptools` 。

ROS中用到的构建工具：`colcon` 、`catkin_make` 、`catkin_make_isolated` 、`catkin_tools` 。

很明显colcon作为构建工具，通过调用 `CMake` 、`Python setuptools` 完成构建。

1.2 常见构建系统

1.2.1 CMake

CMake 是一个跨平台构建系统生成器。项目使用独立于平台的文件指定其生成过程。用户通过使用CMake为其平台上的本机工具生成构建系统来构建项目。

通常用法有：`cmake` 、`make` 、`make intsall`

1.2.2 Python setuptools

`setuptools` 是Python包的打包常用工具。Python 包使用文件来描述依赖项，以及如何构建和安装内容。在ROS2中，功能包可以是“普通”Python包，而在ROS1中，任何Python功能都是从CMake文件触发setup.py进行打包。

通常的用法有：`python setup.py`

1.2.3 catkin

catkin基于CMake，并提供了一组方便的函数，使编写CMake包更容易。它自动生成 CMake 配置文件以及 pkg 配置文件。它还提供了注册不同类型测试的函数。

1.3 常见构建工具

1.3.1 catkin_make

该工具仅调用 CMake 一次，并使用 CMake 的函数在单个上下文中处理所有包。虽然这是一种有效的方法，因为所有包中的所有目标都可以并行化，但它具有明显的缺点。由于所有函数名称、目标和测试都共享一个命名空间，并且规模更大，这很容易导致冲突。

1.3.2 colcon



colcon是一个命令行工具，用于改进构建，测试和使用多个软件包的工作流程。它自动化了流程，处理了订购并设置了使用软件包的环境。

colcon 文档

1.3.3 ament_tools

`ament_tools` 由用于构建 ROS 2 包的独立 Python 3 包提供。它是为引导ROS 2项目而开发的，因此仅针对Python 3，并且可以在Linux，MacOS和Windows上运行。

`ament_tools` 支持构建以下软件包：

- 带有 `package.xml` 文件的 ROS 2 包。
- 带有 `package.xml` 普通的 CMake 包。
- 没有清单文件的普通 CMake 包（从 CMake 文件中提取包名称和依赖项）。
- 带有 `package.xml` 文件的 Python 包。
- 没有清单文件的 Python 包（从 `setup.py` 文件中提取包名称和依赖项）。

2.Colcon构建进阶

我们平时用的最多的场景是编译功能包，所以这里小鱼重点介绍build时候的一些参数。

2.1 build参数

2.1.0 构建指令

- `--packages-select` ，仅生成单个包（或选定的包）。
- `--packages-up-to` ，构建选定的包，包括其依赖项。
- `--packages-above` ，整个工作区，然后对其中一个包进行了更改。此指令将重构此包以及（递归地）依赖于此包的所有包。

2.1.1.指定构建后安装的目录

可以通过 `--build-base` 参数和 `--install-base` ，指定构建目录和安装目录。

2.1.2.合并构建目录

`--merge-install` ，使用 作为所有软件包的安装前缀，而不是安装基中的软件包特定子目录。--install-base

如果没有此选项，每个包都将提供自己的环境变量路径，从而导致非常长的环境变量值。

使用此选项时，添加到环境变量的大多数路径将相同，从而导致环境变量值更短。

2.1.3.符号链接安装

启用 `--symlink-install` 后将不会把文拷贝到install目录，而是通过创建符号链接的方式。

2.1.4.错误时继续安装

启用 `--continue-on-error` ，当发生错误的时候继续进行编译。

2.1.5 CMake参数

`--cmake-args` ，将任意参数传递给CMake。与其他选项匹配的参数必须以空格为前缀。

2.1.6 控制构建线程

- `--executor EXECUTOR` ，用于处理所有作业的执行程序。默认值是根据所有可用执行程序扩展的优先级选择的。要查看完整列表，请调用 `colcon extensions colcon_core.executor --verbose` 。
    - `sequential` [`colcon-core`]

一次处理一个包。

- `parallel` [`colcon-parallel-executor`]

  处理多个作业**平行**.

- --parallel-workers NUMBER
  - 要并行处理的最大作业数。默认值为 os.cpu_count() 给出的逻辑 CPU 内核数。

2.1.7 开启构建日志

使用 `--log-level` 可以设置日志级别，比如 `--log-level info` 。

3.总结

有关测试相关的小鱼暂时不讲了，毕竟国内的程序员写测试的还是很少的（😉）。

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AIIotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

4.ROS2节点发现与多机通信

本文主要讲解ROS2的DOMAIN_ID概念，并介绍在没有ROSMASTER的情况下，ROS2如何实现互相发现。



1.概述

如其他地方所解释的，ROS 2用于通讯的默认中间件是DDS。在DDS中，不同逻辑网络共享物理网络的主要机制称为域(Domain) ID。同一域上的ROS 2节点可以自由地相互发现并发送消息，而不同域上的ROS 2节点则不能。所有ROS 2节点默认使用域ID为0。为了避免在同一网络上运行ROS 2的不同计算机组之间互相干扰，应为每组设置不同的域ID。

2.选择域ID (短版本)

下面的文本解释了应该在ROS 2中使用的域ID范围的推导。要跳过该背景知识并且只是选择一个安全的数字，只需选择一个介于0和101之间的安全的域ID (包括0和101)。

3.选择域ID (长版本)

DDS使用域ID计算将用于发现和通讯的UDP端口。有关如何计算端口的详细信息，请参见 这篇文章 。我们知道在网络中，UDP端口是 无符号16位整型 。因此可以分配的最大端口号是65535。用链接（http://dev.ros2.fishros.com/doc/Concepts/About-Domain-ID.html#domain-id-to-udp-port-calculator）中的公式计算一下，这意味着可以分配的最高域账号是232，而可以分配的最低域账号是0。

特定平台的约束

为了实现最大的兼容性，在选择域账号时应遵循一些特定于平台的附加约束。特别是，最好避免在操作系统的临时端口范围 中分配域ID。这避免了ROS 2节点使用的端口与计算机上的其他网络服务之间可能的冲突。

以下是一些关于特定平台临时端口的提示。

Linux

默认情况下，linux内核使用端口32768-60999作为临时端口。这意味着域ID 0-101 和 215-232 可以安全使用，而不会与临时端口发生冲突。临时端口范围可在Linux中通过在 `/proc/sys/net/ipv4/ip_local_port_range` 中设置自定义值进行配置。如果使用自定义临时端口范围，则可能需要相应地调整上述数字。

macOS

默认情况下，macOS上的临时端口范围为49152-65535。这意味着域ID 0-166可以安全使用，而不会与临时端口发生冲突。通过为 `net.inet.ip.portrange.first` 和 `net.inet.ip.portrange.last` 设置自定义sysctl值，临时端口范围可在macOS中配置。如果使用自定义临时端口范围，则可能需要相应地调整上述数字。

Windows

默认情况下，Windows上的临时端口范围为49152-65535。这意味着域ID 0-166可以安全使用，不会与临时端口发生冲突。临时的端口范围可通过 使用netsh 在窗口中配置。如果使用自定义临时端口范围，则可能需要相应地调整上述数字。

参与者约束

对于计算机上运行的每个ROS 2进程，将创建一个DDS "participant" 。由于每个DDS参与者占用计算机上的两个端口，因此在一台计算机上运行120个以上的ROS 2进程可能会溢出到其他域ID或临时端口。

为了解释原因，我们考虑域ID编号1和2。

- 域ID 1使用端口7650和7651进行多播。
- 域ID 2使用端口7900和7901进行多播。
- 在域ID 1中创建第一个进程 (第0个参与者) 时，端口7660和7661用于单播。
- 在域ID 1中创建第120个进程 (第119个参与者) 时，端口7898和7899用于单播。
- 在域ID 1中创建第121个进程 (第120个参与者) 时，端口7900和7901用于单播，并与域ID 2重叠。

如果已知计算机一次只能在一个域ID上，并且域ID足够低，那么创建比这更多的ROS 2进程是安全的。

在选择特定平台域 ID 范围顶部的域 ID 时，还有一个限制因素需要考虑。

例如，假设一台ID为101的Linux计算机:

- 计算机上的第0个ROS 2进程将连接到端口32650、32651、32660和32661。
- 计算机上的第1个ROS 2进程将连接到端口32650、32651、32662和32663。
- 计算机上的第53个ROS 2进程将连接到端口32650、32651、32766和32767。
- 计算机上的第54个ROS 2进程将连接到端口32650、32651、32768和32769，运行在临时端口范围内。

因此，在Linux上使用域ID为101时应创建的最大进程数为54。同样，在Linux上使用域ID为232时应创建的最大进程数为63，因为最大端口号为65535。

macOS和Windows的情况相似，尽管数字不同。在macOS和Windows上，当选择166 (范围顶部) 的域账号时，运行到临时端口范围之前，可以在计算机上创建的ROS 2进程的最大数量为120。

4.域ID到UDP端口号计算器

> 到这里试用：
>
> http://dev.ros2.fishros.com/doc/Concepts/About-Domain-ID.html#domain-id-to-udp-port-calculator

---

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

第3章ROS2节点通信之话题与服务

1.章节介绍

上一节我们对ROS2的基本概念、构建工具以及采用Python和C++调用ROS2的客户端库完成了节点的编写。

本节我们将重心放回节点和节点之间的数据传递上来，因为ROS2给我们带来的最大便利除了调试工具就是通信机制上了。

同样的，本章的内容也分为三部分。

- 基础篇，主要介绍通信的原理以及编程语言的一些特性，这些其实属于计算机的基础部分知识，但在学校里可能又很难学到的部分。
- 入门篇，该篇主要分为话题通信、服务通信和自定义通信接口上来。
- 进阶篇，通过对DDS原生API的学习，带你深入了解ROS2的通信原理，为你深入使用ROS2打牢基础。

2.食用方法

如果你对通信的底层原理以及编程语言中多线程Lambda等不熟悉，建议你从基础章节开始学习。

如果你对ROS2的话题服务API使用以及自定义接口比较熟悉可以直接跳过入门篇。

3.章节目录

- 章节导读
- 基础篇-中间件与面向对象基础
  - 1.从底层理解通信
  - 2.通信中间件之ZMQ
  - 3.多线程回调函数和锁
- 入门篇-话题与服务
  - 1.ROS2话题入门
  - 2.话题之RCLCPP实现
  - 3.话题之RCLPY实现
  - 4.ROS2服务入门
  - 5.服务之RCLCPP实现
  - 6.服务之RCLPY实现
  - 7.ROS2接口介绍

- 8.自定义话题接口通信
- 9.自定义服务接口通信

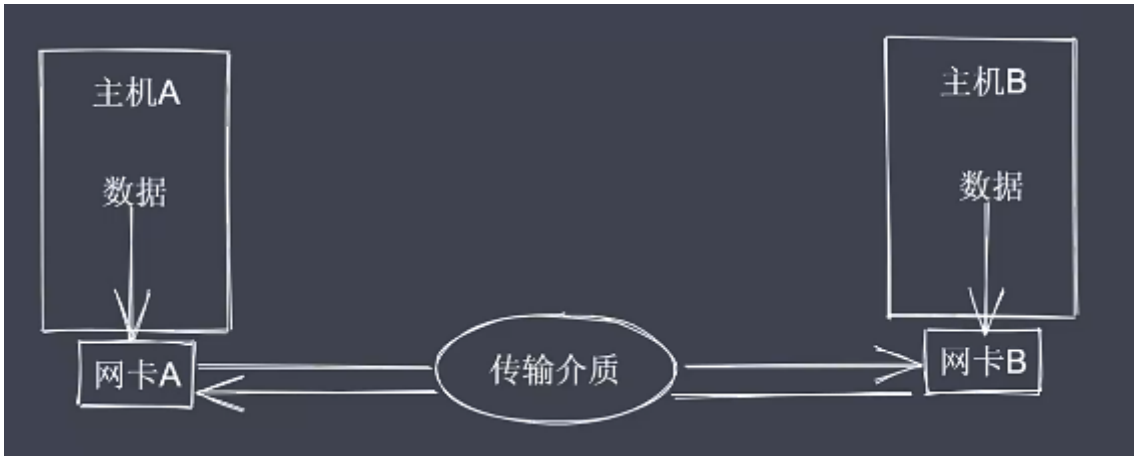
- 进阶篇-中间件进阶
  - 1.原始数据类型与包装类型
  - 2.通信质量Qos配置指南
  - 3.DDS进阶之Fast-DDS环境搭建
  - 4.使用DDS进行订阅发布
  - 5.ROS2消息序列化思考与实践

1.从底层理解通信

本文主要讲解ROS2中所涉及到的几个通信方式。

- 基于TCP/UDP的网络通信。
- 基于共享内存的进程间通信（IPC）。

通信的目的：传递数据



内容还在更新中，请关注公众号鱼香ROS，第一时间获取更新

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

2.通信中间件之ZMQ

- 零MQ (zeromq.org)

1.什么是ZeroMQ

在说pyzmq之前，肯定要了解一下ZeroMQ，小鱼了解ZeroMQ还是从FastDDS那里，因为FastDDS官网老是在哪里强调自己比ZeroMQ性能要好。

大家知道FastDDS是ROS2的通信中间件，那既然FastDDS比ZeroMQ性能好，那为啥小鱼还要介绍ZeroMQ呢？

原因是ZeroMQ非常的轻量，也就是小巧，占用资源少，看名字，Zero Message Queue，零消息队列。

ZeroMQ提供了各种（如进程内、进程间、TCP 和多播）消息传输的套接字，是不是听起来很强大的样子。



2.PyZmq

了解完ZMQ是啥后，我们再来看看PyZMQ。

因为文档过于完善，小鱼不打算跟大家费解释了，给一个官方网址，大家自行阅读

- https://pyzmq.readthedocs.io/en/latest/

pyzmq也提供了类似于订阅发布的方式来传递消息，还有更多的使用方法，比如客户端服务端这种，网上有大佬已经探索了，小鱼贴个链接在这里，大家有需要使用的可以去瞅一瞅。

- https://www.php.cn/python-tutorials-459626.html

3.总结

如果你在做机器人，需要去ROS，需要找一个高效的消息中间件，我想ZMQ应该比较适合你，当然也可以根据机器人的处理器和性能试试ROS2的FastDDS。

3.多线程回调函数和锁

内容还在更新中，请关注公众号鱼香ROS，第一时间获取更新

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

1.ROS2话题入门

话题是ROS2中最常用的通信方式之一，话题通信采取的是订阅发布模型。

1.订阅发布模型

一个节点发布数据到某个话题上，另外一个节点就可以通过订阅话题拿到数据。

```mermaid
graph LR
A[节点1] --发布-->B[topic_name]
B -- 订阅 --> C[节点2]
```

除了上述这种一个节点发布，一个节点接受的形式外，ROS2话题通信其实还可以是 `1对n` , `n对1` , `n对n` 的。

1对n

```mermaid
graph LR
A[节点1] --发布-->B[topic_name]
B -- 订阅 --> C[节点2]
B -- 订阅 --> D[节点3]
B -- 订阅 --> E[节点4]
```

n对1（同一个话题可以有多个发布者）

```mermaid
graph LR
A[节点1] --发布-->B[topic_name]
D[节点2] --发布-->B[topic_name]
B -- 订阅 --> C[节点2]
```

n对n

```mermaid
graph LR
A[节点1] --发布-->B[topic_name]
F[节点2] --发布-->B[topic_name]
```

```
G[节点3] --发布-->B[topic_name]
B -- 订阅 --> C[节点4]
B -- 订阅 --> D[节点5]
B -- 订阅 --> E[节点6]
```

还有一种就是ROS2节点可以订阅本身发布的话题

mermaid

```
graph LR
A[节点] --发布-->B[topic_name]
B -- 订阅 --> A[节点]
```

2. 消息接口

为了方便发送者和接收者进行数据的交换，ROS2帮我们在数据传递时做好了消息的序列化和反序列化（有关消息序列化相关内容请参考本章基础篇），而且ROS2的消息序列化与反序列化通信是可以做到跨编程语言、跨平台和跨设备之间的。

ROS2如何做到跨编程语言、跨平台和跨设备之间的数据收发呢？这就得益于通过定义消息接口文件了。

因为跨平台和设备实现较为复杂，小鱼这里简单说一下如何实现跨语言的。当我们定义好消息接口后，ROS2会根据消息接口内容生成不同语言的接口类，在不同编程语言中调用相同的类即可实现无感的消息序列化和反序列化。

通过对消息接口介绍，相信你肯定能猜到这样一条规则：**同一个话题，所有的发布者和接收者必须使用相同消息接口**。

3. ROS2话题工具

3.1 GUI工具

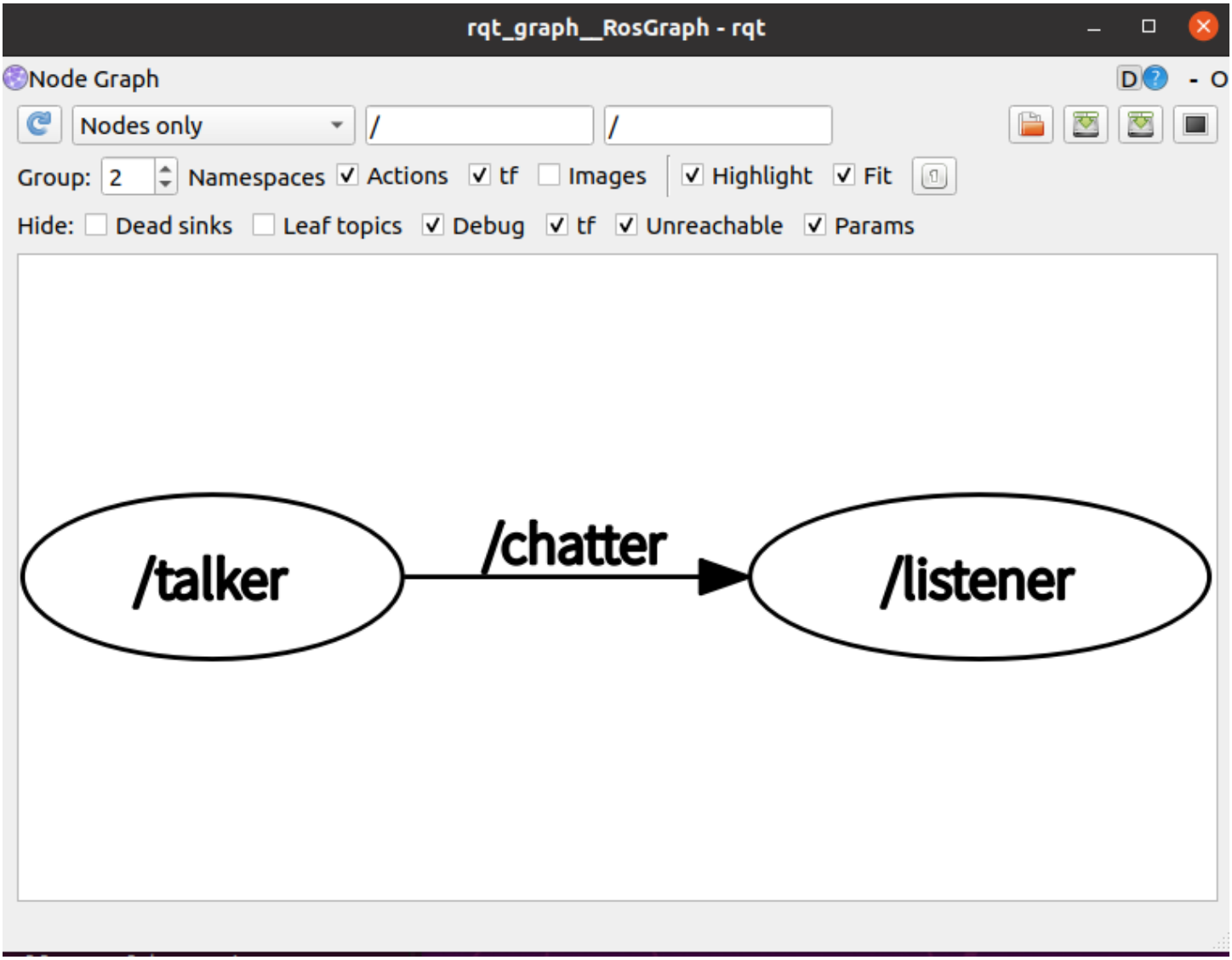
3.1.1 RQT工具之rqt_graph

ROS2作为一个强大的工具，在运行过程中，我们是可以通过命令来看到节点和节点之间的数据关系的。

运行我们第二章中的你说我听小demo。依次打开三个终端，分别输入下面三个命令。

```
ros2 run demo_nodes_py listener
ros2 run demo_nodes_cpp talker
rqt_graph
```

你将看到下面这张图



> 你可以尝试改变菜单栏的Hide或者Group选项，看一看下面图的变化，感受一下rqt_graph工具的强大。

这是一个很重要的工具，小鱼在学习和使用ROS2的过程中经常会用到它，来看一看数据到底是怎么走的，它可以帮我们搞清楚一个节点的输入和输出是什么。

3.2 CLI工具

还记得上一章的ros2 node指令吗？ros2也支持很多强大的topic指令。可以使用下面的指令查看。

```
ros2 topic -h
```

```
rros2@ubuntu:~$ ros2 topic -h
usage: ros2 topic [-h] [--include-hidden-topics]
                  Call `ros2 topic <command> -h` for more detailed usage. ...

Various topic related sub-commands

optional arguments:
  -h, --help            show this help message and exit
  --include-hidden-topics
                        Consider hidden topics as well

Commands:
  bw     Display bandwidth used by topic
  delay  Display delay of topic from timestamp in header
  echo   Output messages from a topic
  find   Output a list of available topics of a given type
  hz     Print the average publishing rate to screen
  info   Print information about a topic
  list   Output a list of available topics
  pub    Publish a message to a topic
  type   Print a topic's type

  Call `ros2 topic <command> -h` for more detailed usage.
```

本着学以致用的目的，小鱼先对比较常用的几个命令进行介绍，其他的我们用到的时候小鱼再介绍（现在介绍反而不好理解）。

3.2.1 ros2 topic list 返回系统中当前活动的所有主题的列表

命令

```
ros2 topic list
```

结果

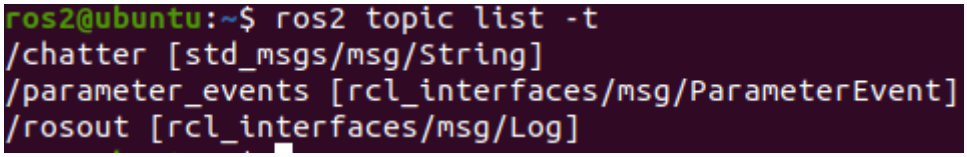
```
ros2@ubuntu:~$ ros2 topic list
/chatter
/parameter_events
/rosout
```

3.2.2 ros2 topic list -t 增加消息类型

命令

```
ros2 topic list -t
```

结果

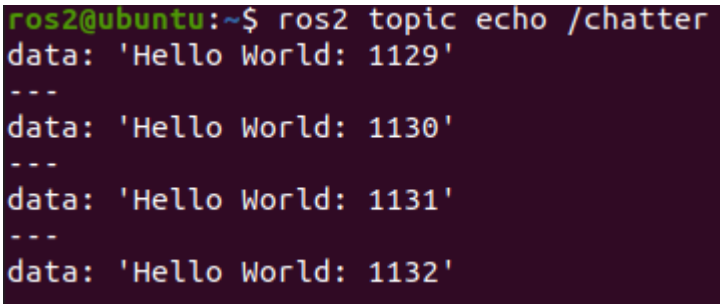


3.2.3 ros2 topic echo 打印实时话题内容

命令

```
ros2 topic echo /chatter
```

结果

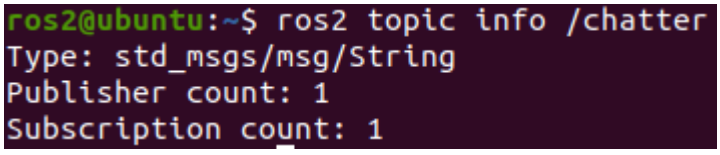


3.2,4 ros2 topic info 查看主题信息

命令

```
ros2 topic info  /chatter
```

结果



3.2.5 ros2 interface show 查看消息类型

上面一个指令告诉大家这个消息是std_msgs/msg/String，那String里面有什么呢？不妨来试一试。

命令

```
ros2 interface show std_msgs/msg/String
```

结果

```
ros2@ubuntu:~$ ros2 interface show std_msgs/msg/String
# This was originally provided as an example message.
# It is deprecated as of Foxy
# It is recommended to create your own semantically meaningful message.
# However if you would like to continue using this please use the equivalent in
example_msgs.

string data
```

3.2.6 ros2 topic pub arg 手动发布命令

关闭发布者，我们受到来发布

命令

```
ros2 topic pub /chatter std_msgs/msg/String 'data: "123"'
```

结果

```
ros2@ubuntu:~$ ros2 topic pub /chatter std_msgs/msg/String 'data: "123"'
publisher: beginning loop
publishing #1: std_msgs.msg.String(data='123')

publishing #2: std_msgs.msg.String(data='123')

publishing #3: std_msgs.msg.String(data='123')

publishing #4: std_msgs.msg.String(data='123')


[INFO] [1627991936.807227517] [listener]: I heard: [123]
[INFO] [1627991937.341446028] [listener]: I heard: [Hello World: 1592]
[INFO] [1627991937.806448432] [listener]: I heard: [123]
[INFO] [1627991938.342772336] [listener]: I heard: [Hello World: 1593]
```

4.最后

了解完话题，下一节小鱼就会带大家来动手写一写话题通信代码。

参考链接：Understanding ROS 2 topics — ROS 2 Documentation: Humble documentation

---

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

2.话题之RCLCPP实现

本节我们学习使用ROS2的RCLCPP客户端库来实现话题通信。

RCLCPP为Node类提供了丰富的API接口，其中就包括创建话题发布者和创建话题订阅者。

1.创建节点

本节我们将创建一个 `控制节点` 和一个 `被控节点` 。

控制节点创建一个话题发布者，发布控制命令（command）话题，接口类型为字符串（string），控制接点通过发布者发布控制命令（前进、后退、左转、右转、停止）。

被控节点创建一个订阅者，订阅控制命令，收到控制命令后根据命令内容打印对应速度出来。

```mermaid
graph LR;
A[控制节点]--command-->B[被控节点]
```

依次输入下面的命令，创建 `chapt3_ws` 工作空间、 `example_topic_rclcpp` 功能包和 `topic_publisher_01.cpp` 。

```shell
cd d2lros2/
mkdir -p chapt3/chapt3_ws/src
cd chapt3/chapt3_ws/src
ros2 pkg create example_topic_rclcpp --build-type ament_cmake --dependencies
rclcpp
touch example_topic_rclcpp/src/topic_publisher_01.cpp
```

完成后目录结构

```
.
└── src
    └── example_topic_rclcpp
        ├── CMakeLists.txt
        ├── include
        │   └── example_topic_rclcpp
```

```
        ├── package.xml
        └── src
            └── topic_publisher_01.cpp

5 directories, 3 files
```

接着采用面向对象方式写一个最简单的节点。

```
#include "rclcpp/rclcpp.hpp"

class TopicPublisher01 : public rclcpp::Node
{
public:
    // 构造函数,有一个参数为节点名称
    TopicPublisher01(std::string name) : Node(name)
    {
        RCLCPP_INFO(this->get_logger(), "%s节点已经启动.", name.c_str());
    }

private:
    // 声明节点
};

int main(int argc, char **argv)
{
    rclcpp::init(argc, argv);
    /*创建对应节点的共享指针对象*/
    auto node = std::make_shared<TopicPublisher01>("topic_publisher_01");
    /* 运行节点，并检测退出信号*/
    rclcpp::spin(node);
    rclcpp::shutdown();
    return 0;
}
```

修改CMakeLists.txt

```
add_executable(topic_publisher_01 src/topic_publisher_01.cpp)
ament_target_dependencies(topic_publisher_01 rclcpp)
```

```
install(TARGETS
  topic_publisher_01
  DESTINATION lib/${PROJECT_NAME}
)
```

接着可以编译测试下，注意运行colcon的目录。

```shell
cd chapt3/chapt3_ws/
colcon build --packages-select example_topic_rclcpp
source install/setup.bash
ros2 run example_topic_rclcpp topic_publisher_01
```

```
root@490925f19143:~/d2lros2/d2lros2/chapt3/chapt3_ws# colcon build --packages-select example_topic_rclcpp
Starting >>> example_topic_rclcpp
Finished <<< example_topic_rclcpp [0.83s]

Summary: 1 package finished [0.97s]
root@490925f19143:~/d2lros2/d2lros2/chapt3/chapt3_ws# source install/setup.bash
root@490925f19143:~/d2lros2/d2lros2/chapt3/chapt3_ws# ros2 run example_topic_rclcpp topic_publisher_01
[INFO] [1654345332.280011232] [topic_publisher_01]: topic_publisher_01节点已经启动.
```

2.编写发布者

2.1 学习使用API文档

想要创建发布者，只需要调用 `node` 的成员函数 `create_publisher` 并传入对应的参数即可。

有关API文档详细内容可以访问：[rclcpp: rclcpp: ROS Client Library for C++ (ros2.org)]

- Publisher
  - rclcpp::Node::create_publisher()
  - rclcpp::Publisher
  - rclcpp::Publisher::publish()
  - rclcpp/publisher.hpp

打开主页，可以看到创建发布者的函数，进去即可看到参数和详细解释。

◆ create_publisher()

```
template<typename MessageT , typename AllocatorT , typename PublisherT >
std::shared_ptr< PublisherT >
rclcpp::Node::create_publisher ( const std::string &                                  topic_name,
                                 const rclcpp::QoS &                                 qos,
                                 const PublisherOptionsWithAllocator< AllocatorT > & options = PublisherOptionsWithAllocator<AllocatorT>()
                               )
```

Create and return a **Publisher**.

The **rclcpp::QoS** has several convenient constructors, including a conversion constructor for size_t, which mimics older API's that allows just a string and size_t to create a publisher.

For example, all of these cases will work:

```
pub = node->create_publisher<MsgT>("chatter", 10);  // implicitly KeepLast
pub = node->create_publisher<MsgT>("chatter", QoS(10));  // implicitly KeepLast
pub = node->create_publisher<MsgT>("chatter", QoS(KeepLast(10)));
pub = node->create_publisher<MsgT>("chatter", QoS(KeepAll()));
pub = node->create_publisher<MsgT>("chatter", QoS(1).best_effort().volatile());
{
  rclcpp::QoS custom_qos(KeepLast(10), rmw_qos_profile_sensor_data);
  pub = node->create_publisher<MsgT>("chatter", custom_qos);
}
```

The publisher options may optionally be passed as the third argument for any of the above cases.

Parameters

[in] **topic_name** The topic for this publisher to publish on.

[in] **qos** The Quality of **Service** settings for the publisher.

[in] **options** Additional options for the created **Publisher**.

Returns

Shared pointer to the created publisher.

通过文档可以看出，我们至少需要传入消息类型（msgT）、话题名称（topic_name）和服务指令（qos）。

2.2 导入消息接口

消息接口是ROS2通信时必须的一部分，通过消息接口ROS2才能完成消息的序列化和反序列化。ROS2为我们定义好了常用的消息接口，并生成了C++和Python的依赖文件，我们可以直接在程序中进行导入。

`ament_cmake` 类型功能包导入消息接口分为三步：

1. 在 `CMakeLists.txt` 中导入，具体是先 `find_packages` 再 `ament_target_dependencies` 。
2. 在 `packages.xml` 中导入，具体是添加 `depend` 标签并将消息接口写入。
3. 在代码中导入，C++中是 `#include"消息功能包/xxx/xxx.hpp"` 。

我们依次做完这三步后文件内容如下：

CMakeLists.txt

```
find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)
```

```
add_executable(topic_publisher_01 src/topic_publisher_01.cpp)
ament_target_dependencies(topic_publisher_01 rclcpp std_msgs)
```

packages.xml

```
  <buildtool_depend>ament_cmake</buildtool_depend>

  <depend>rclcpp</depend>
  <depend>std_msgs</depend>

  <test_depend>ament_lint_auto</test_depend>
  <test_depend>ament_lint_common</test_depend>
```

代码文件 `topic_publisher_01.cpp`

```
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"

class TopicPublisher01 : public rclcpp::Node
```

2.3 创建发布者

根据ROS2的RCLCPPAPI文档可以看出，我们需要提供消息接口、话题名称和服务质量Qos。

- 消息接口上面我们已经导入了，是 `std_msgs/msg/string.h` 。
- 话题名称（topic_name），我们就用 `control_command` 。
- Qos，Qos支持直接指定一个数字，这个数字对应的是 `KeepLast` 队列长度。一般设置成10，即如果一次性有100条消息，默认保留最新的10个，其余的都扔掉。

接着我们可以编写发布者的代码了。

```
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"

class TopicPublisher01 : public rclcpp::Node
```

```
{
public:
    // 构造函数,有一个参数为节点名称
    TopicPublisher01(std::string name) : Node(name)
    {
        RCLCPP_INFO(this->get_logger(), "大家好，我是%s.", name.c_str());
        // 创建发布者
        command_publisher_ = this->create_publisher<std_msgs::msg::String>
("command", 10);
    }

private:
    // 声明话题发布者
    rclcpp::Publisher<std_msgs::msg::String>::SharedPtr command_publisher_;
};
```

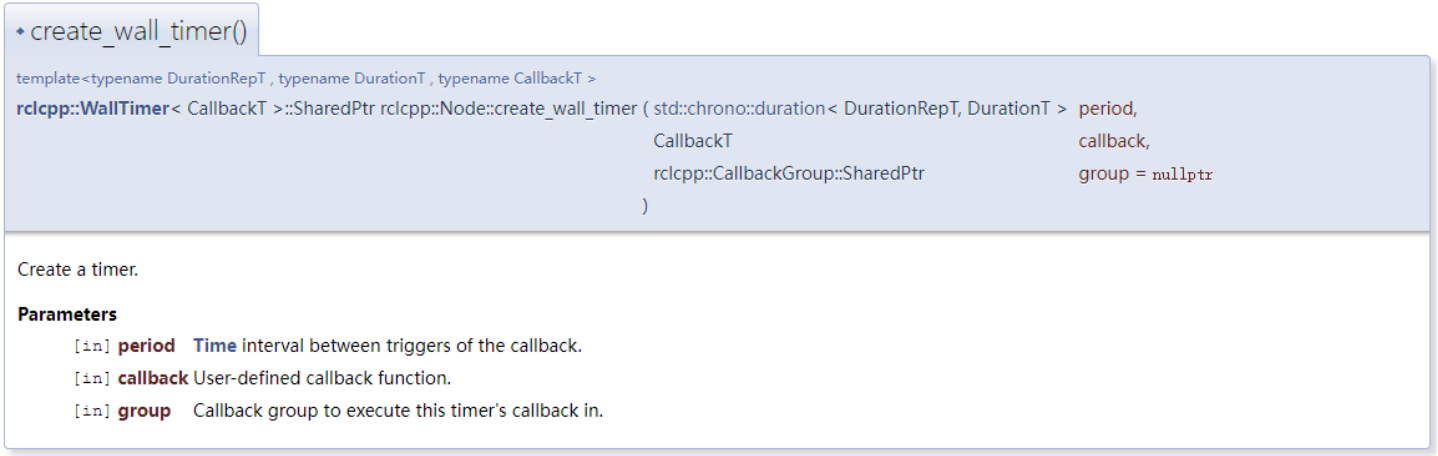
2.4 使用定时器定时发布数据

2.4.1 查看定时器API

虽然编写好了发布者，但是我们还没有发布数据，我们需要通过ROS2中的定时器来设置指定的周期调用回调函数，在回调函数里实现发布数据功能。

> 有关回调函数和定时器相关内容请参考基础篇多线程与回调函数相关内容。

再次找到RCLCPP文档，找到创建定时器函数，观察参数



- period，回调函数调用周期。

- callback，回调函数。
- group，调用回调函数所在的回调组，默认为nullptr。

2.4.2 编写代码

```c++
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"

class TopicPublisher01 : public rclcpp::Node
{
public:
    // 构造函数,有一个参数为节点名称
    TopicPublisher01(std::string name) : Node(name)
    {
        RCLCPP_INFO(this->get_logger(), "大家好，我是%s.", name.c_str());
        // 创建发布者
        command_publisher_ = this->create_publisher<std_msgs::msg::String>
("command", 10);
        // 创建定时器，500ms为周期，定时发布
        timer_ = this->create_wall_timer(std::chrono::milliseconds(500),
std::bind(&TopicPublisher01::timer_callback, this));
    }

private:
    void timer_callback()
    {
        // 创建消息
        std_msgs::msg::String message;
        message.data = "forward";
        // 日志打印
        RCLCPP_INFO(this->get_logger(), "Publishing: '%s'", message.data.c_str());
        // 发布消息
        command_publisher_->publish(message);
    }
    // 声名定时器指针
    rclcpp::TimerBase::SharedPtr timer_;
    // 声明话题发布者指针
    rclcpp::Publisher<std_msgs::msg::String>::SharedPtr command_publisher_;
};
```

2.4.3 代码讲解

定时器

定时器是ROS2中的另外一个常用功能，通过定时器可以实现按照一定周期调用某个函数以实现定时发布等逻辑。

定时器对应的类是 `rclcpp::TimerBase` ，调用 `create_wall_timer` 将返回其共享指针。

创建定时器时传入了两个参数，这两个参数都利用了C++11的新特性。

- `std::chrono::milliseconds(500)` ，代表500ms，chrono是c++ 11中的时间库，提供计时，时钟等功能。
- `std::bind(&TopicPublisher01::timer_callback, this)` ，bind() 函数的意义就像它的函数名一样，是用来绑定函数调用的某些参数的。

创建发布消息

`std_msgs::msg::String` 是通过ROS2的消息文件自动生成的类，其原始消息文件内容可以通过命令行查询

```
ros2 interface show std_msgs/msg/String
```

结果

```
# This was originally provided as an example message.
# It is deprecated as of Foxy
# It is recommended to create your own semantically meaningful message.
# However if you would like to continue using this please use the equivalent in
example_msgs.

string data
```

可以看到其内部包含了一个 `string data` ，**ROS2会将消息文件转换成一个类，并把其中的定义转换成类的成员函数。**

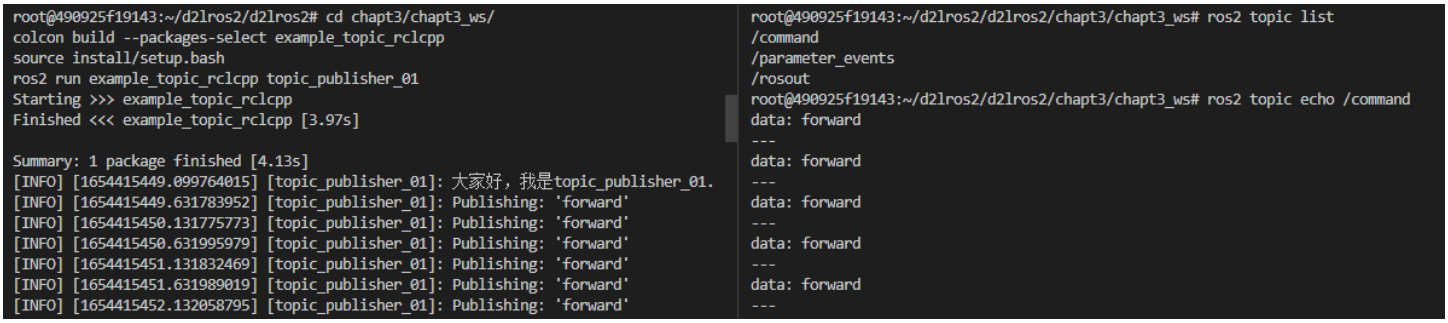
2.5 运行测试

编译，source，运行

```
cd chapt3/chapt3_ws/
colcon build --packages-select example_topic_rclcpp
source install/setup.bash
ros2 run example_topic_rclcpp topic_publisher_01
```

测试

```
# 查看列表
ros2 topic list
# 输出内容
ros2 topic echo /command
```



3.编写订阅者

之所以我们可以用命令行看到数据，原因在于CLI创建了一个订阅者来订阅 `/command` 指令。接下来我们将要手动创建一个节点订阅并处理数据。

3.1 创建订阅节点

shell

```
cd chapt3_ws/src/example_topic_rclcpp
touch src/topic_subscribe_01.cpp
```

编写代码

```c++
#include "rclcpp/rclcpp.hpp"

class TopicSubscribe01 : public rclcpp::Node
{
public:
    // 构造函数,有一个参数为节点名称
    TopicSubscribe01(std::string name) : Node(name)
    {
        RCLCPP_INFO(this->get_logger(), "大家好，我是%s.", name.c_str());
    }

private:
    // 声明节点
};

int main(int argc, char **argv)
{
    rclcpp::init(argc, argv);
    /*创建对应节点的共享指针对象*/
    auto node = std::make_shared<TopicSubscribe01>("topic_subscribe_01");
    /* 运行节点，并检测退出信号*/
    rclcpp::spin(node);
    rclcpp::shutdown();
    return 0;
}
```

CMakeLists.txt

```cmake
add_executable(topic_subscribe_01 src/topic_subscribe_01.cpp)
ament_target_dependencies(topic_subscribe_01 rclcpp)

install(TARGETS
topic_subscribe_01
  DESTINATION lib/${PROJECT_NAME}
)
```

编译测试

```shell
cd chapt3/chapt3_ws/
colcon build --packages-select example_topic_rclcpp
source install/setup.bash
ros2 run example_topic_rclcpp topic_subscribe_01
```

```
root@490925f19143:~/d2lros2/d2lros2/chapt3/chapt3_ws# ros2 run example_topic_rclcpp
topic_subscribe_01
[INFO] [1654419231.611756000] [topic_subscribe_01]: 大家好，我是topic_subscribe_01.
```

3.2 查看订阅者API文档

看API文档或者看头文件中关于函数的定义这是一个好习惯。



create_subscription()

```
template<typename MessageT , typename CallbackT , typename AllocatorT , typename CallbackMessageT , typename SubscriptionT , typename MessageMemoryStrategyT >
std::shared_ptr< SubscriptionT >
rclcpp::Node::create_subscription ( const std::string &                                  topic_name,
                                    const rclcpp::QoS &                                  qos,
                                    CallbackT &&                                         callback,
                                    const SubscriptionOptionsWithAllocator< AllocatorT > & options = SubscriptionOptionsWithAllocator<AllocatorT>(),
                                    typename MessageMemoryStrategyT::SharedPtr           msg_mem_strat = ( MessageMemoryStrategyT::create_default() )
                                  )
```

Create and return a **Subscription**.

Parameters

[in]	**topic_name**	The topic to subscribe on.
[in]	**qos**	QoS profile for Subcription.
[in]	**callback**	The user-defined callback function to receive a message
[in]	**options**	Additional options for the creation of the **Subscription**.
[in]	**msg_mem_strat**	The message memory strategy to use for allocating messages.

Returns

Shared pointer to the created subscription.

五个参数，但后面两个都是默认的参数，我们只需要有话题名称、Qos和回调函数即可。

3.3 编写代码

```
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"

class TopicSubscribe01 : public rclcpp::Node
{
public:
```

```
    TopicSubscribe01(std::string name) : Node(name)
    {
        RCLCPP_INFO(this->get_logger(), "大家好，我是%s.", name.c_str());
          // 创建一个订阅者订阅话题
        command_subscribe_ = this->create_subscription<std_msgs::msg::String>
("command", 10, std::bind(&TopicSubscribe01::command_callback, this,
std::placeholders::_1));
    }

private:
     // 声明一个订阅者
    rclcpp::Subscription<std_msgs::msg::String>::SharedPtr command_subscribe_;
     // 收到话题数据的回调函数
    void command_callback(const std_msgs::msg::String::SharedPtr msg)
    {
        double speed = 0.0f;
        if(msg->data == "forward")
        {
            speed = 0.2f;
        }
        RCLCPP_INFO(this->get_logger(), "收到[%s]指令，发送速度 %f", msg-
>data.c_str(),speed);
    }
};
```

依然的需要在 `CMakeLists.txt` 添加下 `std_msgs` 依赖

```cmake
ament_target_dependencies(topic_subscribe_01 rclcpp std_msgs)
```

`packages.xml` 就不需要了，同一个功能包，已经添加了。

3.4 运行测试

编译运行订阅节点

```
cd chapt3/chapt3_ws/
colcon build --packages-select example_topic_rclcpp
```

```
source install/setup.bash
ros2 run example_topic_rclcpp topic_subscribe_01
```

手动发布数据测试订阅者

```
ros2 topic pub /command std_msgs/msg/String "{data: forward}"
```



4.订阅发布测试

关闭上面启动的终端，重新运行指令

```
cd chapt3/chapt3_ws/
source install/setup.bash
ros2 run example_topic_rclcpp topic_subscribe_01
```

运行发布节点

```
cd chapt3/chapt3_ws/
source install/setup.bash
ros2 run example_topic_rclcpp topic_publisher_01
```

运行结果



查看计算图 `rqt`

打开RQT、选择Node Graph、点击



刷新下



技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

3.话题之RCLPY实现

有了前面的经验，实现Python版本的发布订阅也非常的轻松了，因为ROS2的API一致性保持的很好，这点值得点赞。

- Node — rclpy 0.6.1 documentation (ros2.org)

1.创建功能包和节点

创建功能包

```
cd chapt3/chapt3_ws/src/
ros2 pkg create example_topic_rclpy  --build-type ament_python --dependencies
rclpy
```

创建节点文件

```
cd example_topic_rclpy/example_topic_rclpy
touch topic_subscribe_02.py
touch topic_publisher_02.py
```

简单编写下代码，依然采用类的形式

发布者

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node

class NodePublisher02(Node):
    def __init__(self,name):
        super().__init__(name)
        self.get_logger().info("大家好，我是%s!" % name)

def main(args=None):
    rclpy.init(args=args) # 初始化rclpy
```

```
    node = NodePublisher02("topic_publisher_02")  # 新建一个节点
    rclpy.spin(node) # 保持节点运行，检测是否收到退出指令（Ctrl+C）
    rclpy.shutdown() # 关闭rclpy
```

订阅节点

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node

class NodeSubscribe02(Node):
    def __init__(self,name):
        super().__init__(name)
        self.get_logger().info("大家好，我是%s!" % name)

def main(args=None):
    rclpy.init(args=args) # 初始化rclpy
    node = NodeSubscribe02("topic_subscribe_02")  # 新建一个节点
    rclpy.spin(node) # 保持节点运行，检测是否收到退出指令（Ctrl+C）
    rclpy.shutdown() # 关闭rclpy
```

setup.py

```
    entry_points={
        'console_scripts': [
            "topic_publisher_02 = example_topic_rclpy.topic_publisher_02:main",
            "topic_subscribe_02 = example_topic_rclpy.topic_subscribe_02:main"
        ],
    },
```

2.编写订阅者

```
from std_msgs.msg import String
```

```
class NodeSubscribe02(Node):
    def __init__(self,name):
        super().__init__(name)
        self.get_logger().info("大家好，我是%s!" % name)
        # 创建订阅者
        self.command_subscribe_ =
self.create_subscription(String,"command",self.command_callback,10)

    def command_callback(self,msg):
        speed = 0.0
        if msg.data=="backup":
            speed = -0.2
        self.get_logger().info(f'收到[{msg.data}]命令，发送速度{speed}')
```

3.编写发布者

```
from std_msgs.msg import String

class NodePublisher02(Node):
    def __init__(self,name):
        super().__init__(name)
        self.get_logger().info("大家好，我是%s!" % name)
        self.command_publisher_ = self.create_publisher(String,"command", 10)
        self.timer = self.create_timer(0.5, self.timer_callback)

    def timer_callback(self):
        """
        定时器回调函数
        """
        msg = String()
        msg.data = 'backup'
        self.command_publisher_.publish(msg)
        self.get_logger().info(f'发布了指令：{msg.data}')    #打印一下发布的数据
```

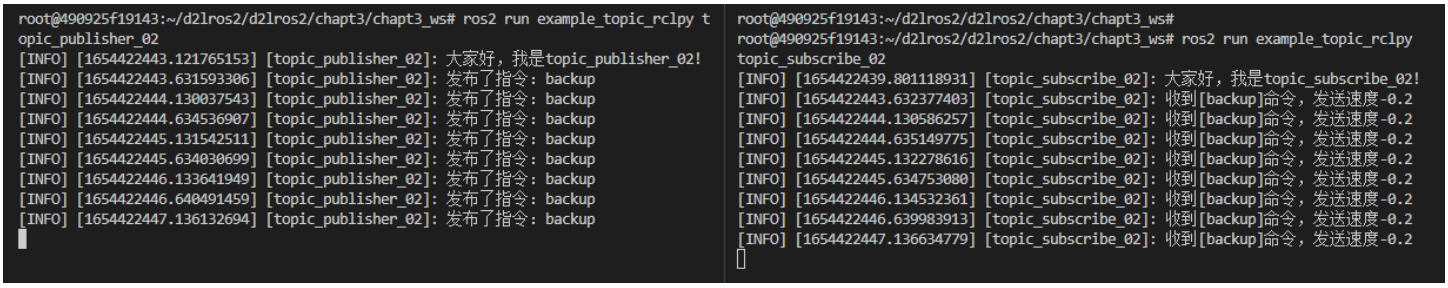
4.运行测试

4.1 发布节点

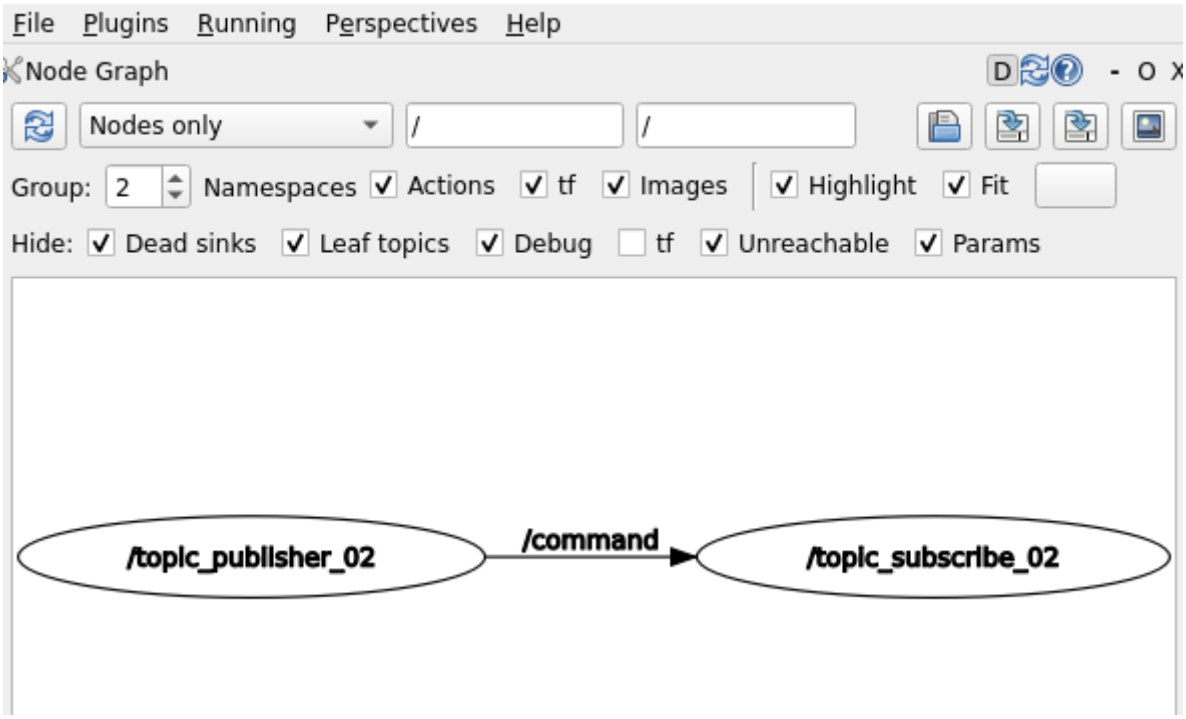
```
cd chapt3/chapt3_ws/
source install/setup.bash
ros2 run example_topic_rclpy topic_publisher_02
```

4.1 订阅节点

```
cd chapt3/chapt3_ws/
source install/setup.bash
ros2 run example_topic_rclpy topic_subscribe_02
```



4.2 RQT



技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

4.ROS2服务入门

大家好，帅鱼又蹬蹬蹬的游回来了。本节小鱼将要带大家一起了解一下什么是服务。

1.服务通信介绍

服务分为客户端和服务端，平时我们用的手机APP都可以成为客户端，而APP服务器对于软件来说就是服务端。

客户端发送请求给服务端，服务端可以根据客户端的请求做一些处理，然后返回结果给客户端。

```mermaid
graph
A[服务端-李四] --发送响应-->B[客户端-李三]
B[客户端]  --发送请求--> A[服务端]
```

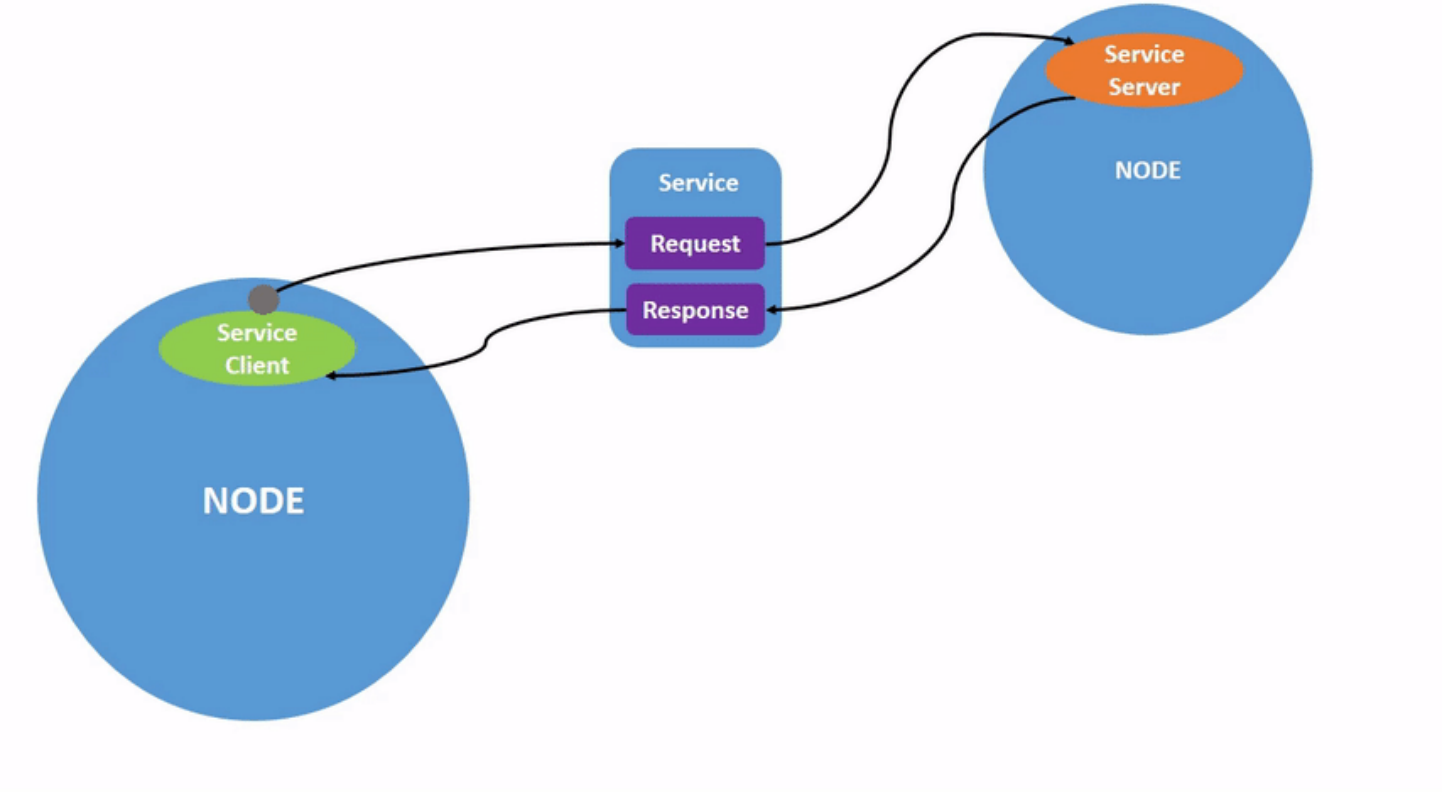
所以服务-客户端模型，也可以成为请求-响应模型。

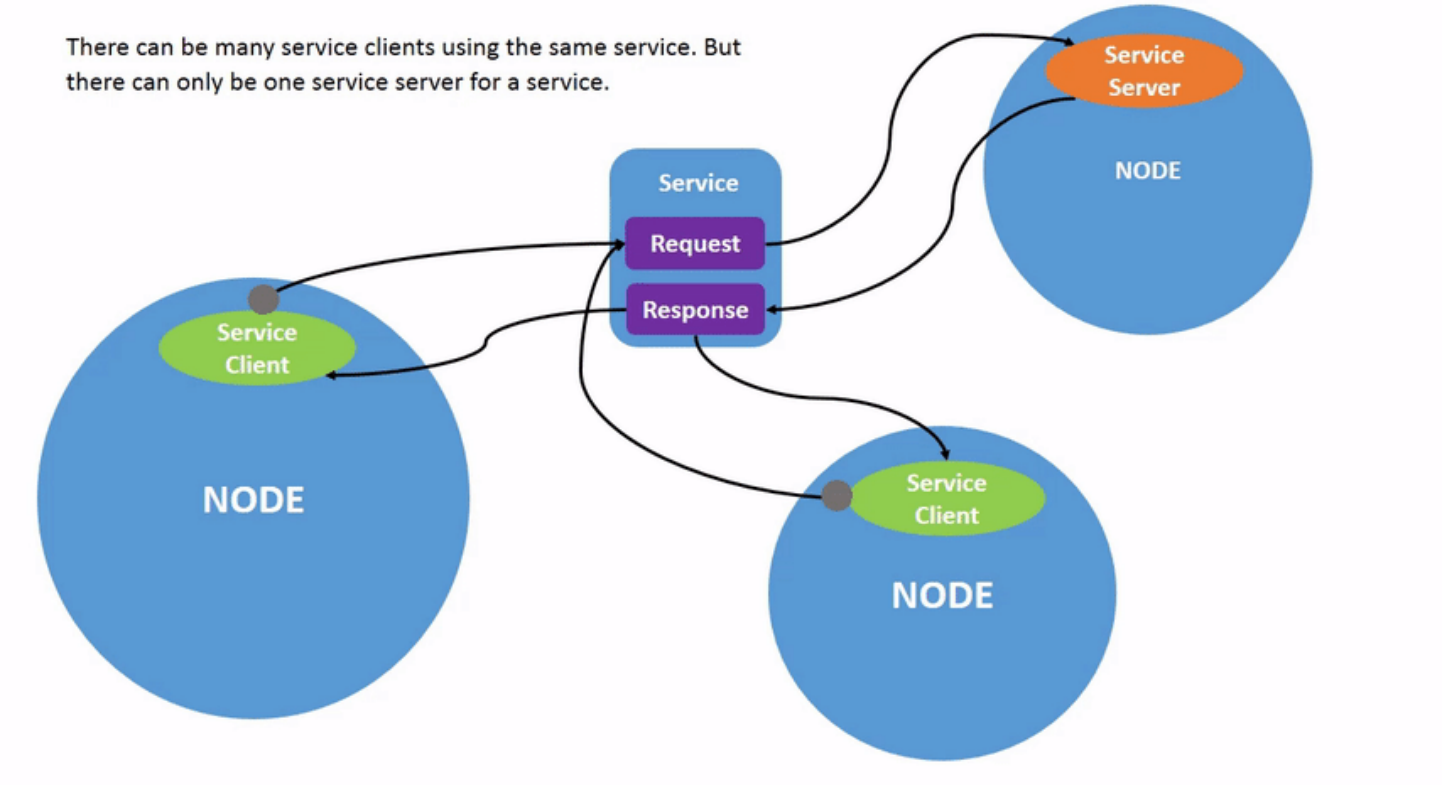
> **不知道你有没有感觉到服务和话题的不同之处，话题是没有返回的，适用于单向或大量的数据传递。而服务是双向的，客户端发送请求，服务端响应请求。**

同时服务还是有一些注意事项：

- 同一个服务（名称相同）有且只能有一个节点来提供
- 同一个服务可以被多个客户端调用

放两张官方形象的动图：





2.体验服务

在我们安装ROS2的时候其实系统为我们安装了一些样例程序，其中就有服务使用样例，我们可以先来体验一下。

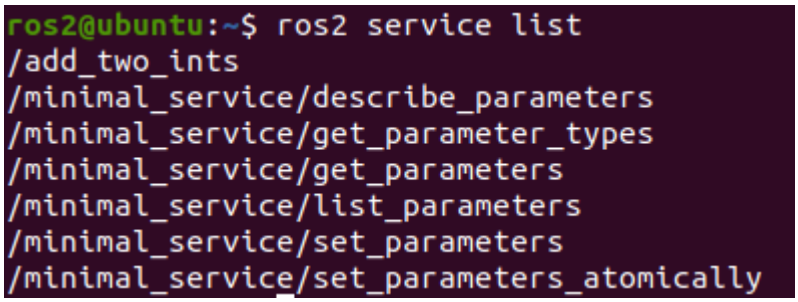
2.1 启动服务端

打开终端，运行下面的命令，这个命令用于运行一个服务节点，这个服务的功能是将两个数字相加，给定a，b两个数，返回sum也就是ab之和。

```
ros2 run examples_rclpy_minimal_service service
```

2.2 使用命令查看服务列表

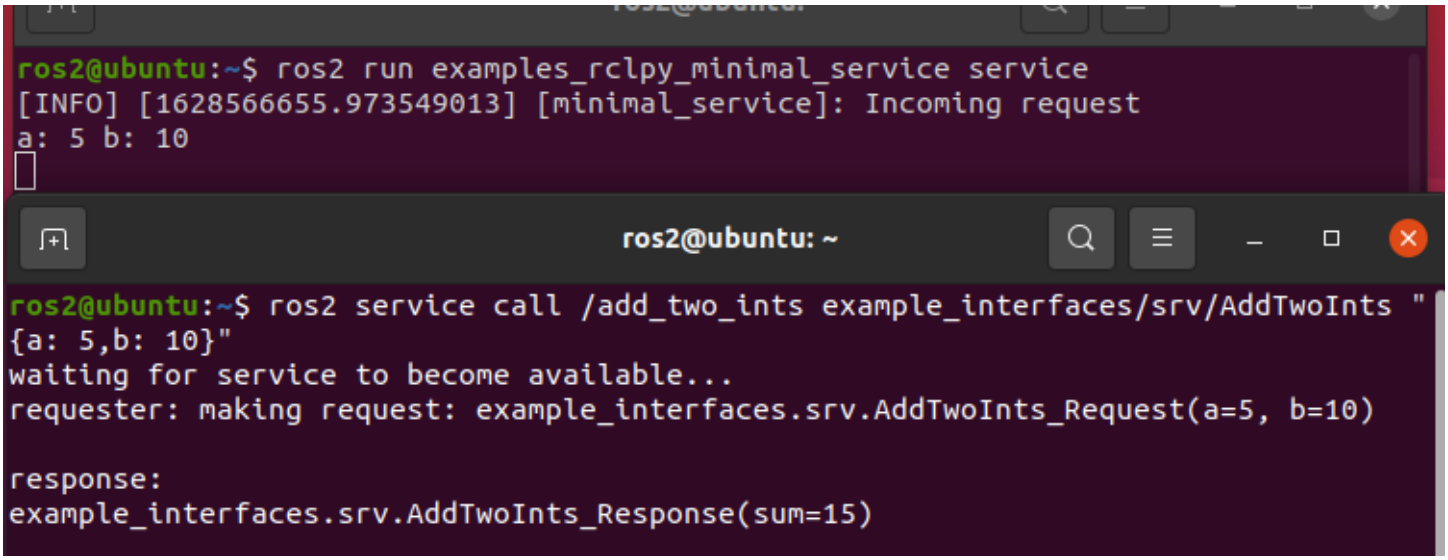
```
ros2 service list
```



2.3手动调用服务

再启动一个终端，输入下面的命令（注意a：、b：后的空格）。

```
ros2 service call /add_two_ints example_interfaces/srv/AddTwoInts "{a: 5,b: 10}"
```



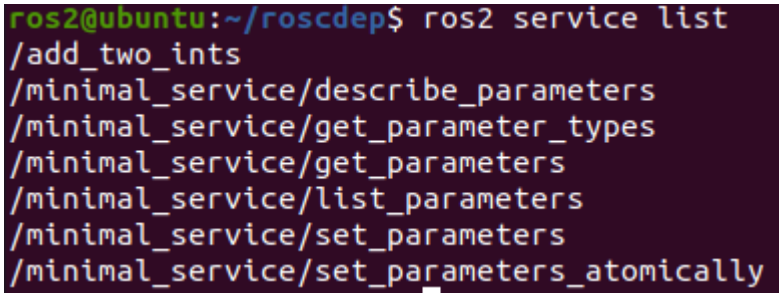
我们可以看到客户端请求两个数字5+10，服务端返回15。

3.ROS2服务常用命令

ROS2的命令行工具，小鱼觉得还是非常值得一学的，毕竟确实很实用（装X），之前已经给大家讲过了关于节点、话题、接口相关的命令了，现在小鱼说一下关于服务的那些命令行。

3.1查看服务列表

```
ros2 service list
```



3.2手动调用服务

```
ros2 service call /add_two_ints example_interfaces/srv/AddTwoInts "{a: 5,b: 10}"
```

```
ros2@ubuntu:~$ ros2 service call /add_two_ints example_interfaces/srv/AddTwoInts "
{a: 5,b: 10}"
waiting for service to become available...
requester: making request: example_interfaces.srv.AddTwoInts_Request(a=5, b=10)

response:
example_interfaces.srv.AddTwoInts_Response(sum=15)
```

如果不写参数值调用会怎么样？比如下面这种，大家可以尝试下。

```
ros2 service call /add_two_ints example_interfaces/srv/AddTwoInts
```

3.3 查看服务接口类型

```
ros2 service type /add_two_ints
```

```
ros2@ubuntu:~/roscdep$ ros2 service type /add_two_ints
example_interfaces/srv/AddTwoInts
```

4.4查找使用某一接口的服务

这个命令看起来和4.3刚好相反。

```
ros2 service find example_interfaces/srv/AddTwoInts
```

```
ros2@ubuntu:~/roscdep$ ros2 service find example_interfaces/srv/AddTwoInts
/add_two_ints
```

4.总结

本节大家和小鱼一起又多认识了一个小伙伴，ROS2的服务。

下一节我们将学习使用RCL在节点里创建服务端和客户端。

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

5.服务之RCLCPP实现

因为还没有学习如何自定义接口，所以我们先借着上一节的两数相加的示例接口，利用rclcpp提供的接口实现两数相加的服务端和客户端。

1.创建功能包和节点

```shell
cd chapt3/chapt3_ws/src
ros2 pkg create example_service_rclcpp --build-type ament_cmake --dependencies
rclcpp
touch example_service_rclcpp/src/service_server_01.cpp
touch example_service_rclcpp/src/service_client_01.cpp
```

面向对象方式写两个最简单的节点

service_server_01.cpp

```cpp
#include "rclcpp/rclcpp.hpp"

class ServiceServer01 : public rclcpp::Node {
public:
  ServiceServer01(std::string name) : Node(name) {
    RCLCPP_INFO(this->get_logger(), "节点已启动：%s.", name.c_str());
  }

private:
};

int main(int argc, char** argv) {
  rclcpp::init(argc, argv);
  auto node = std::make_shared<ServiceServer01>("service_server_01");
  rclcpp::spin(node);
  rclcpp::shutdown();
  return 0;
}
```

service_client_01.cpp

```cpp
#include "rclcpp/rclcpp.hpp"

class ServiceClient01 : public rclcpp::Node {
public:
  // 构造函数,有一个参数为节点名称
  ServiceClient01(std::string name) : Node(name) {
    RCLCPP_INFO(this->get_logger(), "节点已启动：%s.", name.c_str());
  }
private:
};

int main(int argc, char** argv) {
  rclcpp::init(argc, argv);
  /*创建对应节点的共享指针对象*/
  auto node = std::make_shared<ServiceClient01>("service_client_01");
  /* 运行节点，并检测退出信号*/
  rclcpp::spin(node);
  rclcpp::shutdown();
  return 0;
}
```

CMakeLists.txt

```shell
add_executable(service_client_01 src/service_client_01.cpp)
ament_target_dependencies(service_client_01 rclcpp)

add_executable(service_server_01 src/service_server_01.cpp)
ament_target_dependencies(service_server_01 rclcpp)

install(TARGETS
  service_server_01
  DESTINATION lib/${PROJECT_NAME}
)

install(TARGETS
  service_client_01
```

```
  DESTINATION lib/${PROJECT_NAME}
)
```

完成上面的步骤，即可编译测试了，相信你已经对这些步骤非常熟悉了。

```shell
cd chapt3/chapt3_ws/
colcon build --packages-select example_service_rclcpp

# 运行 service_server_01
source install/setup.bash
ros2 run example_service_rclcpp service_server_01

# 打开新终端运行  service_client_01
source install/setup.bash
ros2 run example_service_rclcpp service_client_01
```

2.服务端实现

2.1 导入接口

两数相加我们需要利用ROS2自带的 `example_interfaces` 接口，使用命令行可以查看这个接口的定义。

```
ros2 interface show example_interfaces/srv/AddTwoInts
```

结果

```
int64 a
int64 b
---
int64 sum
```

导入接口的三个步骤不知道你是否还记得。

> `ament_cmake` 类型功能包导入消息接口分为三步：
>
> 1. 在 `CMakeLists.txt` 中导入，具体是先 `find_packages` 再 `ament_target_dependencies` 。
> 2. 在 `packages.xml` 中导入，具体是添加 `depend` 标签并将消息接口写入。
> 3. 在代码中导入，C++中是 `#include"消息功能包/xxx/xxx.hpp"` 。

根据步骤改一下：

`CMakeLists.txt`

```
# 这里我们一次性把服务端和客户端对example_interfaces的依赖都加上
find_package(example_interfaces REQUIRED)

add_executable(service_client_01 src/service_client_01.cpp)
ament_target_dependencies(service_client_01 rclcpp example_interfaces)

add_executable(service_server_01 src/service_server_01.cpp)
ament_target_dependencies(service_server_01 rclcpp example_interfaces)
```

`packages.xml`

```
<depend>example_interfaces</depend>
```

代码

```
#include "example_interfaces/srv/add_two_ints.hpp"
```

2.2 编写代码

先看代码再解释

```c++
#include "example_interfaces/srv/add_two_ints.hpp"
#include "rclcpp/rclcpp.hpp"
```

```
class ServiceServer01 : public rclcpp::Node {
public:
  ServiceServer01(std::string name) : Node(name) {
    RCLCPP_INFO(this->get_logger(), "节点已启动：%s.", name.c_str());
    // 创建服务
    add_ints_server_ =
      this->create_service<example_interfaces::srv::AddTwoInts>(
        "add_two_ints_srv",
        std::bind(&ServiceServer01::handle_add_two_ints, this,
                  std::placeholders::_1, std::placeholders::_2));
  }

private:
  // 声明一个服务
  rclcpp::Service<example_interfaces::srv::AddTwoInts>::SharedPtr
    add_ints_server_;

  // 收到请求的处理函数
  void handle_add_two_ints(
    const std::shared_ptr<example_interfaces::srv::AddTwoInts::Request> request,
    std::shared_ptr<example_interfaces::srv::AddTwoInts::Response> response) {
    RCLCPP_INFO(this->get_logger(), "收到a: %ld b: %ld", request->a,
                request->b);
    response->sum = request->a + request->b;
  };
};
```

create_service，参考rclcpp API文档即可

create_service()

```
template<typename ServiceT , typename CallbackT >
rclcpp::Service< ServiceT >::SharedPtr rclcpp::Node::create_service ( const std::string &                service_name,
                                                                      CallbackT &&                      callback,
                                                                      const rmw_qos_profile_t &         qos_profile = rmw_qos_profile_services_default,
                                                                      rclcpp::CallbackGroup::SharedPtr  group = nullptr
                                                                    )
```

Create and return a **Service**.

Parameters

- `[in]` **service_name** The topic to service on.
- `[in]` **callback** User-defined callback function.
- `[in]` **qos_profile** rmw_qos_profile_t Quality of service profile for client.
- `[in]` **group** Callback group to call the service.

Returns

Shared pointer to the created service.

- ServiceT，消息接口 `example_interfaces::srv::AddTwoInts`
- service_name，服务名称
- callback，回调函数，使用成员函数作为回调函数，std::bind进行转换
- qos_profile，服务质量配置文件，默认 `rmw_qos_profile_services_default`
- group，调用服务的回调组，默认 `nullptr`

2.3 测试

```
cd chapt3_ws/
colcon build --packages-select example_service_rclcpp
source install/setup.bash
ros2 run example_service_rclcpp service_server_01
```

接着打开一个新的终端

```
# 你应该可以看到我们声明的服务
ros2 service list
# 使用命令行进行调用
ros2 service call /add_two_ints_srv example_interfaces/srv/AddTwoInts "{a: 5,b:
10}"
```

```
root@490925f19143:~/d2lros2/d2lros2/chapt3/chapt3_ws# source install/setup.bash
root@490925f19143:~/d2lros2/d2lros2/chapt3/chapt3_ws# ros2 run example_service_rclcpp
service_server_01
[INFO] [1654526465.961810490] [service_server_01]: 节点已启动：service_server_01.
[INFO] [1654526545.360777265] [service_server_01]: 收到a: 5 b: 10
```

```
root@490925f19143:~/d2lros2/d2lros2/chapt3/chapt3_ws# ros2 service call /add_two_ints_
srv example_interfaces/srv/AddTwoInts "{a: 5,b: 10}"
requester: making request: example_interfaces.srv.AddTwoInts_Request(a=5, b=10)

response:
example_interfaces.srv.AddTwoInts_Response(sum=15)

root@490925f19143:~/d2lros2/d2lros2/chapt3/chapt3_ws#
```

3.客户端实现

3.1 API接口

写代码时看API文档是个好习惯，先看看创建客户端的：地址

3.1.1 create_client

create_client() [1/2]

```
template<typename ServiceT >
rclcpp::Client<ServiceT>::SharedPtr rclcpp::Node::create_client ( const std::string &                service_name,
                                                                  const rmw_qos_profile_t &         qos_profile = rmw_qos_profile_services_default,
                                                                  rclcpp::CallbackGroup::SharedPtr  group = nullptr
                                                                )
```

Create and return a **Client**.

Parameters

- [in] **service_name** The topic to service on.
- [in] **qos_profile** rmw_qos_profile_t Quality of service profile for client.
- [in] **group** Callback group to call the service.

Returns

Shared pointer to the created client.

参数加上ServiceT（接口类型），一共有四个，都是老熟人了，就不介绍了。

3.1.2 async_send_request

接着我们来看看发送请求的API，地址

async_send_request() [2/3]

```
template<typename ServiceT >
template<typename CallbackT , typename std::enable_if< rclcpp::function_traits::same_arguments< CallbackT, CallbackType >::value >::type * = nullptr>
SharedFuture rclcpp::Client< ServiceT >::async_send_request ( SharedRequest  request,
                                                              CallbackT &&   cb
                                                            )
```

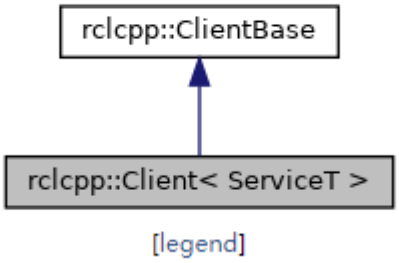
我们这里要用的是这个函数 `async_send_request()` 同时传入两个参数

- request，请求的消息，这里用于放a，b两个数。
- CallBack，回调函数，异步接收服务器的返回的函数。

> 至于为什么ROS2中那么多回调函数，以及用回调函数的好处，小鱼这里就不解释了，不清楚的小伙伴可以看看基础篇的内容。

3.1.3 wait_for_service

这个函数是用于等待服务上线的，这个函数并不在rclcpp::Client中定义，而是在其父类中定义的。



上面是继承图，在其父类中有这个函数的解释。

• wait_for_service()

template<typename RepT = int64_t, typename RatioT = std::milli>
bool rclcpp::ClientBase::wait_for_service (std::chrono::duration< RepT, RatioT > timeout = std::chrono::duration<RepT, RatioT>(-1))

Wait for a service to be ready.

Parameters

timeout maximum time to wait

Returns

`true` if the service is ready and the timeout is not over, `false` otherwise

参数就一个，等待的时间，返回值是bool类型的，上线了就是true，不上线就是false。

之所以会用的这个函数的原因是，再发送请求之前保证服务端启动了，避免发送一个请求出去而无人响应的尴尬局面。

最后还有一些小细节，先看代码小鱼再进一步的解释。

3.2 代码

c++

```
#include "example_interfaces/srv/add_two_ints.hpp"

class ServiceClient01 : public rclcpp::Node {
public:
  // 构造函数,有一个参数为节点名称
  ServiceClient01(std::string name) : Node(name) {
    RCLCPP_INFO(this->get_logger(), "节点已启动：%s.", name.c_str());
    // 创建客户端
    client_ = this->create_client<example_interfaces::srv::AddTwoInts>
("add_two_ints_srv");
  }

  void send_request(int a, int b) {
    RCLCPP_INFO(this->get_logger(), "计算%d+%d", a, b);

    // 1.等待服务端上线
    while (!client_->wait_for_service(std::chrono::seconds(1))) {
      //等待时检测rclcpp的状态
      if (!rclcpp::ok()) {
        RCLCPP_ERROR(this->get_logger(), "等待服务的过程中被打断...");
        return;
      }
      RCLCPP_INFO(this->get_logger(), "等待服务端上线中");
    }

    // 2.构造请求的
    auto request =
      std::make_shared<example_interfaces::srv::AddTwoInts_Request>();
    request->a = a;
    request->b = b;

    // 3.发送异步请求，然后等待返回，返回时调用回调函数
    client_->async_send_request(
      request, std::bind(&ServiceClient01::result_callback_, this,
                         std::placeholders::_1));
  };

private:
  // 声明客户端
  rclcpp::Client<example_interfaces::srv::AddTwoInts>::SharedPtr client_;
```

```
    void result_callback_(
      rclcpp::Client<example_interfaces::srv::AddTwoInts>::SharedFuture
        result_future) {
      auto response = result_future.get();
      RCLCPP_INFO(this->get_logger(), "计算结果：%ld", response->sum);
    }
  };
```

这里小鱼需要额外讲解的是回调函数 `void result_callback_(rclcpp::Client<example_interfaces::srv::AddTwoInts>::SharedFuture result_future)`

这个又臭又长的参数确实让人惊了下，函数的参数是客户端 `AddTwoInts` 类型的 `SharedFuture` 对象，这个对象的定义如下



可以看到其又是利用C++11的新特性 `std::shared_future` 创建的 `SharedResponse` 类模板。

类模板 `std::shared_future` 提供访问异步操作结果的机制，类似 std::future ，除了允许多个线程等候同一共享状态。

我们具体看看std::shared_future的API

获取结果	
get	返回结果 (公开成员函数)
状态	
valid	检查 future 是否拥有共享状态 (公开成员函数)
wait	等待结果变得可用 (公开成员函数)
wait_for	等待结果，如果在指定的超时间隔后仍然无法得到结果，则返回。 (公开成员函数)
wait_until	等待结果，如果在已经到达指定的时间点时仍然无法得到结果，则返回。 (公开成员函数)

可以看到使用 `get` 函数即可获取结果。所以下面这段代码的意思相信你已经大概理解了。

```
    auto response = result_future.get();
    RCLCPP_INFO(this->get_logger(), "计算结果：%ld", response->sum);
```

3.3 测试

最后还要修改下主函数，用于调用服务端发送请求。

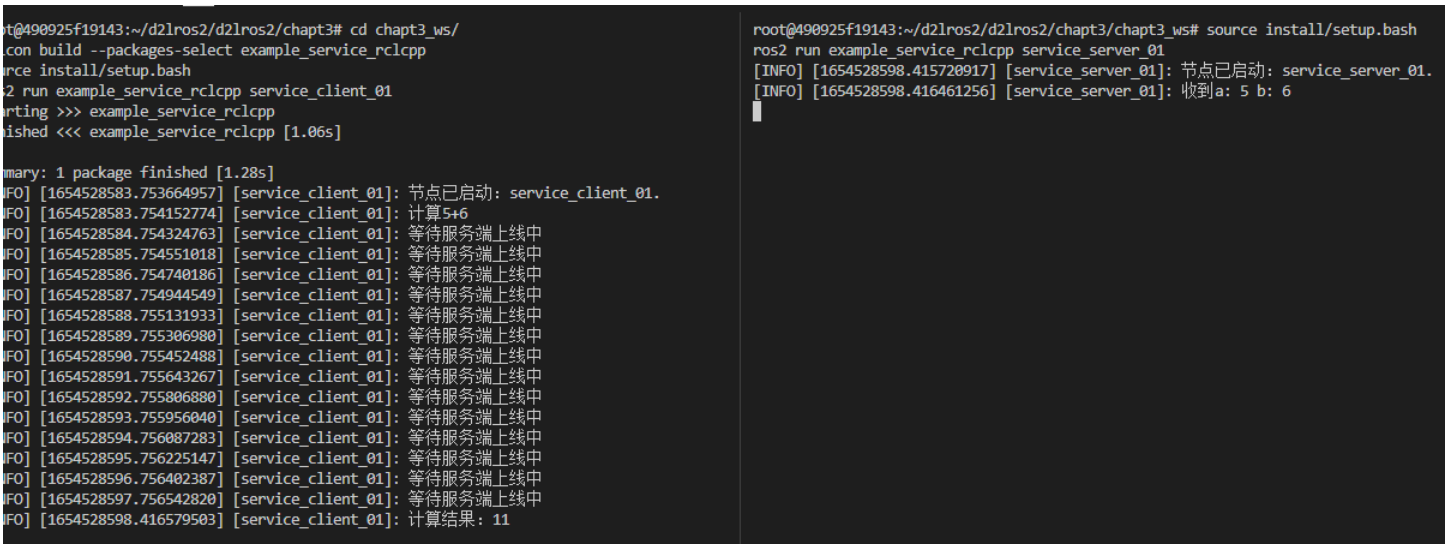
```c++
int main(int argc, char** argv) {
  rclcpp::init(argc, argv);
  /*创建对应节点的共享指针对象*/
  auto node = std::make_shared<ServiceClient01>("service_client_01");
  /* 运行节点，并检测退出信号*/
  //增加这一行，node->send_request(5, 6);，计算5+6结果
  node->send_request(5, 6);
  rclcpp::spin(node);
  rclcpp::shutdown();
  return 0;
}
```

接着编译运行客户端

```
cd chapt3_ws/
colcon build --packages-select example_service_rclcpp
source install/setup.bash
ros2 run example_service_rclcpp service_client_01
```

打开服务端，让服务上线

```
source install/setup.bash
ros2 run example_service_rclcpp service_server_01
```



4.总结

本节我们通过RCLCPP完成了服务服务端和客户端的编写，并学了一些C++语言的新特性。下一节我们学习使用rclpy实现相同的功能。

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

6.服务之RCLPY实现

1.创建功能包和节点

事到如今，小鱼也不藏着掖着了，创建功能包其实还可以加上一些参数，让这个过程变得更简单。

先上指令再说

```
cd chapt3/chapt3_ws/src
ros2 pkg create example_service_rclpy --build-type ament_python --dependencies
rclpy example_interfaces  --node-name service_server_02
```

接着你会惊奇的发现，依赖，setup.py中的安装配置，ROS2都帮你加好了。

这是因为 `--node-name service_server_02` 会帮你创建好节点文件和添加执行文件。

但是也有一些限制，比如只支持一个节点文件，所以我们还需要手动创建一个。

```
cd example_service_rclpy/example_service_rclpy/
touch service_client_02.py
```

修改下setup.py

```
    entry_points={
        'console_scripts': [
            "service_client_02 = example_service_rclpy.service_client_02:main",
            "service_server_02 = example_service_rclpy.service_server_02:main"
        ],
    },
```

接着面向对象来一筐，将两个节点的内容补充一下

service_server_02

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node

class ServiceServer02(Node):
    def __init__(self,name):
        super().__init__(name)
        self.get_logger().info("节点已启动：%s!" % name)

def main(args=None):
    rclpy.init(args=args) # 初始化rclpy
    node = ServiceServer02("service_server_02")  # 新建一个节点
    rclpy.spin(node) # 保持节点运行，检测是否收到退出指令（Ctrl+C）
    rclpy.shutdown() # 关闭rclpy
```

service_client_02

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from example_interfaces.srv import AddTwoInts

class ServiceClient02(Node):
    def __init__(self,name):
        super().__init__(name)
        self.get_logger().info("节点已启动：%s!" % name)

def main(args=None):
    rclpy.init(args=args) # 初始化rclpy
    node = ServiceClient02("service_client_02")  # 新建一个节点
    node.send_request(3,4)
    rclpy.spin(node) # 保持节点运行，检测是否收到退出指令（Ctrl+C）
    rclpy.shutdown() # 关闭rclpy
```

2.服务端实现

2.1 看 API

地址小鱼放这里，大家自行看下即可

- Node — rclpy 0.6.1 documentation (ros2.org)

create_service(*srv_type*, *srv_name*, *callback*, ***, *qos_profile=<rclpy.qos.QoSProfile object>*, *callback_group=None*)

Create a new service server.

Parameters:
- **srv_type** – The service type.
- **srv_name** (**str**) – The name of the service.
- **callback** (**Callable**[[~SrvTypeRequest, ~SrvTypeResponse], ~SrvTypeResponse]) – A user-defined callback function that is called when a service request received by the server.
- **qos_profile** (**QoSProfile**) – The quality of service profile to apply the service server.
- **callback_group** (**Optional**[**CallbackGroup**]) – The callback group for the service server. If `None`, then the nodes default callback group is used.

Return type: **Service**

2.2 写代码

```
# 导入接口
from example_interfaces.srv import AddTwoInts


class ServiceServer02(Node):
    def __init__(self,name):
        super().__init__(name)
        self.get_logger().info("节点已启动：%s!" % name)
        self.add_ints_server_ = self.create_service(AddTwoInts,"add_two_ints_srv",
self.handle_add_two_ints)

    def handle_add_two_ints(self,request, response):
        self.get_logger().info(f"收到请求，计算{request.a} + {request.b}")
```

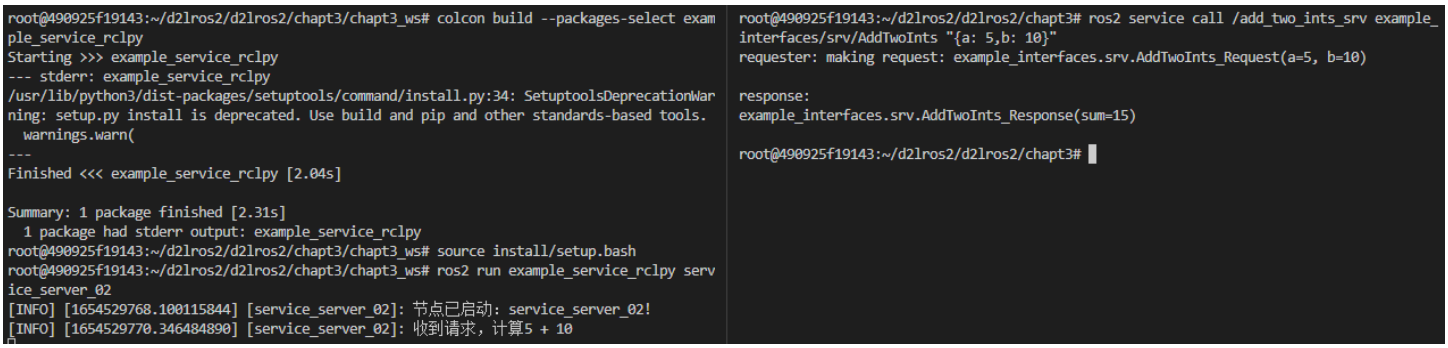
```
        response.sum = request.a + request.b
        return response
```

2.3 测试

```
colcon build --packages-select example_service_rclpy
source install/setup.bash
ros2 run example_service_rclpy service_server_02
```

打开新终端

```
ros2 service call /add_two_ints_srv example_interfaces/srv/AddTwoInts "{a: 5,b: 10}"
```



3.客户端实现

2.1 API

- [Node — rclpy 0.6.1 documentation (ros2.org)]

2.2 写代码

```
from example_interfaces.srv import AddTwoInts
```

```
class ServiceClient02(Node):
    def __init__(self,name):
        super().__init__(name)
        self.get_logger().info("节点已启动：%s!" % name)
        self.client_ = self.create_client(AddTwoInts,"add_two_ints_srv")

    def result_callback_(self, result_future):
        response = result_future.result()
        self.get_logger().info(f"收到返回结果：{response.sum}")

    def send_request(self, a, b):
        while rclpy.ok() and self.client_.wait_for_service(1)==False:
            self.get_logger().info(f"等待服务端上线....")

        request = AddTwoInts.Request()
        request.a = a
        request.b = b
        self.client_.call_async(request).add_done_callback(self.result_callback_)

def main(args=None):
    rclpy.init(args=args) # 初始化rclpy
    node = ServiceClient02("service_client_02")  # 新建一个节点
    # 调用函数发送请求
    node.send_request(3,4)
    rclpy.spin(node) # 保持节点运行，检测是否收到退出指令（Ctrl+C）
    rclpy.shutdown() # 关闭rclpy
```

同样是异步请求，rclpy客户端库定义的是 `call_async` 并且使用 `add_done_callback` 添加回调函数。

2.3 测试

编译启动客户端

```
colcon build --packages-select example_service_rclpy
source install/setup.bash
ros2 run example_service_rclpy service_client_02
```

启动服务端

```
source install/setup.bash
ros2 run example_service_rclpy service_server_02
```



4.总结

本节我们通过rclpy库实现了节点之间的服务通信。但是我们都是用别人的接口，下一节我们学习自定义接口并在代码中使用。

下面两节含实战内容，一定要跟着动手写哦。

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

7.ROS2接口介绍

本节小鱼将会带你学习认识一个新的概念，叫做interface，即接口。

1.ROS2接口介绍

1.1 什么是接口

接口其实是一种规范

你还记得前面几节的示例中，我们在代码中使用过这两种接口，这两种数据类型分别代表字符串和32位二进制的整型数据，是ROS 2提前定义的一种规范。

```
std_msgs/msg/String
std_msgs/msg/UInt32
```

1.2 为什么使用接口

小鱼举一个雷达的例子，不同的厂家生产出不同的类型的激光雷达，每种雷达驱动方式、扫描速率等等都不相同。

当机器人进行导航时，需要激光雷达的扫描数据，假如没有统一接口，每次更换一个种类的雷达，都需要重新做程序适配。

于是ROS2中定义了一个统一的接口叫做 `sensor_msgs/msg/LaserScan` ,现在几乎每个雷达的厂家都会编写程序将自己雷达的数据变成 `sensor_msgs/msg/LaserScan` 格式，提供给用户使用。

如果雷达的例子不好理解，大家可以把雷达换成手机充电器，USB接口是不是也是一种规范，所有的厂家都按照这种接口进行充电器和连接线的生产。

1.3. ROS2自带的接口

前面话题通信时 `std_msgs` 功能包是我们安装ROS2的时候ROS2为我们自动安装的，除了 `std_msgs` 之外，`ROS2` 还定义了很多做机器人常用的接口。

> 使用`ros2 interface package sensor_msgs`命令可以查看某一个接口包下所有的接口

比如：传感器类的消息包`sensor_msgs`

```
打开终端输入：ros2 interface package sensor_msgs
sensor_msgs/msg/JointState  #机器人关节数据
sensor_msgs/msg/Temperature #温度数据
sensor_msgs/msg/Imu #加速度传感器
sensor_msgs/msg/Image #图像
sensor_msgs/msg/LaserScan #雷达数据
......
```

虽然ROS2为我们定义了大量`有手就行，拿来就用`的接口，但有时候还是不能满足我们的变态想法，所以我们需要掌握自定义接口的方法。

2. 接口文件内容

2.1 可以定义的接口三种类型

小鱼提到过，ROS2提供了四种通信方式：

- 话题-Topics
- 服务-Services
- 动作-Action
- 参数-Parameters

除了参数之外，话题、服务和动作(Action)都支持自定义接口，每一种通信方式所适用的场景各不相同，所定义的接口也被分为话题接口、服务接口、动作接口三种。

2.2 接口形式

这三种接口定义起来有什么不一样的地方呢？小鱼先带大家直观感受一下：

话题接口格式：`xxx.msg`

```
int64 num
```

服务接口格式：`xxx.srv`

```
int64 a
int64 b
---
int64 sum
```

动作接口格式：`xxx.action`

```
int32 order
---
int32[] sequence
---
int32[] partial_sequence
```

2.3 接口数据类型

根据引用方式不同可以分为基础类型和包装类型两类。

基础类型有（同时后面加上[]可形成数组）

```
bool
byte
char
float32,float64
int8,uint8
int16,uint16
int32,uint32
int64,uint64
string
```

包装类型

即在已有的接口类型上进行包含，比如

```
uint32 id
string image_name
sensor_msgs/Image
```

2.4 接口如何生成代码

有的同学可能会问这样一个问题，我们只是简单的写了一下变量类型和名称，我们在程序里面怎么调用呢？

其实这里有一个转换的过程：将msg、srv、action文件转换为Python和C++的头文件。

```mermaid
graph LR
A[msg,srv,action] -->B[ROS2-IDL转换器]
B  --> C[Python的py,C++的.h头文件]
```

通过ROS2的IDL模块 产生了头文件，有了头文件，我们就可以在程序里导入并使用这个消息模块。

3.自定义接口实践

3.1 场景定义

给定一个机器人开发中的常见控制场景，我们设计满足要求的服务接口和话题接口。

设计两个节点

- 一个机器人节点，对外提供移动指定距离服务，移动完成后返回当前位置，同时对外发布机器人的位置和状态（是否在移动）。
- 机器人控制节点，通过服务控制机器人移动指定距离，并实时获取机器人的当前位置和状态。

假设机器人在坐标轴上，只能前后移动。

3.2 定义接口

服务接口 `MoveRobot.srv`

```
# 前进后退的距离
float32 distance
---
# 当前的位置
float32 pose
```

话题接口，采用基础类型 `RobotStatus.msg`

```
uint32 STATUS_MOVEING = 1
uint32 STATUS_STOP = 1
uint32  status
float32 pose
```

话题接口，混合包装类型 `RobotPose.msg`

```shell
uint32 STATUS_MOVEING = 1
uint32 STATUS_STOP = 2
uint32  status
geometry_msgs/Pose pose
```

3.3 创建接口功能包编接口

创建功能包

```shell
ros2 pkg create example_ros2_interfaces --build-type ament_cmake --dependencies
rosidl_default_generators geometry_msgs
```

注意功能包类型必须为：ament_cmake

依赖 `rosidl_default_generators` 必须添加， `geometry_msgs` 视内容情况添加（我们这里有 `geometry_msgs/Pose pose` 所以要添加）。

接着创建文件夹和文件将3.2中文件写入，注意话题接口放到 `msg` 文件夹下，以 `.msg` 结尾。服务接口放到 `srv` 文件夹下，以 `srv` 结尾。

```
.
├── CMakeLists.txt
├── msg
│   ├── RobotPose.msg
│   └── RobotStatus.msg
├── package.xml
└── srv
    └── MoveRobot.srv

2 directories, 5 files
```

接着修改 `CMakeLists.txt`

```cmake
find_package(rosidl_default_generators REQUIRED)
find_package(geometry_msgs REQUIRED)
# 添加下面的内容
rosidl_generate_interfaces(${PROJECT_NAME}
  "msg/RobotPose.msg"
  "msg/RobotStatus.msg"
  "srv/MoveRobot.srv"
  DEPENDENCIES geometry_msgs
)
```

接着修改 `package.xml`

```
  <buildtool_depend>ament_cmake</buildtool_depend>

  <depend>rosidl_default_generators</depend>
  <depend>geometry_msgs</depend>

  <member_of_group>rosidl_interface_packages</member_of_group> #添加这一行
```

```
    <test_depend>ament_lint_auto</test_depend>
    <test_depend>ament_lint_common</test_depend>
```

保存即可编译

```shell
colcon build --packages-select example_ros2_interfaces
```

编译完成后在 `chapt3_ws/install/example_ros2_interfaces/include` 下你应该可以看到C++的头文件。在 `chapt3_ws/install/example_ros2_interfaces/local/lib/python3.10/dist-packages` 下应该可以看到Python版本的头文件。

接下来的代码里我们就可以通过头文件导入和使用我们定义的接口了。

> @TODO 讲一下为什么要source

4.ROS2接口常用CLI命令

最后给大家讲一下ROS2接口相关的常用命令有哪些。

4.1查看接口列表

```
ros2 interface list
```

```
ros2@ros2:~/code/ros2/town_ws$ ros2 interface list
Messages:
    action_msgs/msg/GoalInfo
    action_msgs/msg/GoalStatus
    action_msgs/msg/GoalStatusArray
    actionlib_msgs/msg/GoalID
    actionlib_msgs/msg/GoalStatus
    actionlib_msgs/msg/GoalStatusArray
    builtin_interfaces/msg/Duration
    builtin_interfaces/msg/Time
    diagnostic_msgs/msg/DiagnosticArray
    diagnostic_msgs/msg/DiagnosticStatus
    diagnostic_msgs/msg/KeyValue
    example_interfaces/msg/Bool
    example_interfaces/msg/Byte
    example_interfaces/msg/ByteMultiArray
    example_interfaces/msg/Char
    example_interfaces/msg/Empty
    example_interfaces/msg/Float32
    example_interfaces/msg/Float32MultiArray
    example_interfaces/msg/Float64
    example_interfaces/msg/Float64MultiArray
    example_interfaces/msg/Int16
```

4.2 查看某一个接口详细的内容

```
ros2 interface show std_msgs/msg/String
```

```
ros2@ros2:~/code/ros2/town_ws$ ros2 interface show std_msgs/msg/String
# This was originally provided as an example message.
# It is deprecated as of Foxy
# It is recommended to create your own semantically meaningful message.
# However if you would like to continue using this please use the equivalent in example_msgs.

string data
```

8.自定义接口RCLCPP实战

本节我们就利用上一节创建好的消息接口进行代码编写代码，学习在实际的项目中使用自定义接口，同时也作为一个小练习，我们将在同一个节点里融合话题和服务两种通信方式。

1.创建功能包和节点

这里我们设计两个节点

- `example_interfaces_robot_01` ，机器人节点，对外提供控制机器人移动服务并发布机器人的状态。
- `example_interfaces_control_01` ，控制节点，发送机器人移动请求，订阅机器人状态话题。

在工作空间下的src文件夹中创建功能包 `example_ros2_interfaces` 添加 `example_ros2_interfaces` 和 `rclcpp` 依赖，并自动生成 `example_interfaces_robot_01` 节点。

因为 `--node-name` 只支持创建一个节点，我们再添加一下另外一个节点。

```shell
cd chapt3_ws/
ros2 pkg create example_interfaces_rclcpp --build-type ament_cmake --dependencies
rclcpp example_ros2_interfaces --destination-directory src --node-name
example_interfaces_robot_01
touch src/example_interfaces_rclcpp/src/example_interfaces_control_01.
```

`CMakeLists.txt`

```c++
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(example_ros2_interfaces REQUIRED)

add_executable(example_interfaces_robot_01 src/example_interfaces_robot_01.cpp)
target_include_directories(example_interfaces_robot_01 PUBLIC
  $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
  $<INSTALL_INTERFACE:include>)
target_compile_features(example_interfaces_robot_01 PUBLIC c_std_99 cxx_std_17)  #
Require C99 and C++17
ament_target_dependencies(
  example_interfaces_robot_01
```

```
  "rclcpp"
  "example_ros2_interfaces"
)

install(TARGETS example_interfaces_robot_01
  DESTINATION lib/${PROJECT_NAME})


add_executable(example_interfaces_control_01
src/example_interfaces_control_01.cpp)
target_include_directories(example_interfaces_control_01 PUBLIC
  $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
  $<INSTALL_INTERFACE:include>)
target_compile_features(example_interfaces_control_01 PUBLIC c_std_99 cxx_std_17)
# Require C99 and C++17
ament_target_dependencies(
  example_interfaces_control_01
  "rclcpp"
  "example_ros2_interfaces"
)

install(TARGETS example_interfaces_control_01
  DESTINATION lib/${PROJECT_NAME})
```

面向对象写一下两个节点的内容

example_interfaces_control_01.cpp

```c++
 #include "rclcpp/rclcpp.hpp"

class ExampleInterfacesControl : public rclcpp::Node {
public:
  // 构造函数,有一个参数为节点名称
  ExampleInterfacesControl(std::string name) : Node(name) {
    RCLCPP_INFO(this->get_logger(), "节点已启动：%s.", name.c_str());
  }

private:
};
```

```
int main(int argc, char** argv) {
  rclcpp::init(argc, argv);
  auto node = std::make_shared<ExampleInterfacesControl>
("example_interfaces_control_01");
  rclcpp::spin(node);
  rclcpp::shutdown();
  return 0;
}
```

example_interfaces_robot_01.cpp

```
c++
#include "rclcpp/rclcpp.hpp"

/*创建一个机器人类，模拟真实机器人*/
class Robot {
public:
  Robot() = default;
  ~Robot() = default;
private:
};


class ExampleInterfacesRobot : public rclcpp::Node {
public:
  ExampleInterfacesRobot(std::string name) : Node(name) {
    RCLCPP_INFO(this->get_logger(), "节点已启动：%s.", name.c_str());
  }

private:
  Robot robot;
};

int main(int argc, char** argv) {
  rclcpp::init(argc, argv);
  auto node = std::make_shared<ExampleInterfacesRobot>
("example_interfaces_robot_01");
  rclcpp::spin(node);
  rclcpp::shutdown();
```

```
  return 0;
}
```

保存编译即可测试

```shell
cd chapt3_ws/
colcon build
source install/setup.bash
```

2.编写机器人类

```c++
// 导入上一节定义的消息接口
#include "example_ros2_interfaces/msg/robot_status.hpp"
#include "example_ros2_interfaces/srv/move_robot.hpp"
#include "rclcpp/rclcpp.hpp"

/*
 * 测试指令：ros2 service call /move_robot example_ros2_interfaces/srv/MoveRobot "
{distance: 5}"
 */

class Robot {
public:
  Robot() = default;
  ~Robot() = default;
  /**
   * @brief 移动指定的距离
   *
   * @param distance
   * @return float
   */
  float move_distance(float distance) {
    status_ = example_ros2_interfaces::msg::RobotStatus::STATUS_MOVEING;
    target_pose_ += distance;
    // 当目标距离和当前距离大于0.01则持续向目标移动
    while (fabs(target_pose_ - current_pose_) > 0.01) {
```

```
        // 每一步移动当前到目标距离的1/10
        float step = distance / fabs(distance) * fabs(target_pose_ - current_pose_)
* 0.1;
        current_pose_ += step;
        std::cout << "移动了：" << step << "当前位置：" << current_pose_ << std::endl;
        // 当前线程休眠500ms
        std::this_thread::sleep_for(std::chrono::milliseconds(500));
      }
      status_ = example_ros2_interfaces::msg::RobotStatus::STATUS_STOP;
      return current_pose_;
    }
    /**
     * @brief Get the current pose
     *
     * @return float
     */
    float get_current_pose() { return current_pose_; }

    /**
     * @brief Get the status
     *
     * @return int
     *  1 example_ros2_interfaces::msg::RobotStatus::STATUS_MOVEING
     *  2 example_ros2_interfaces::msg::RobotStatus::STATUS_STOP
     */
    int get_status() { return status_; }

  private:
    // 声明当前位置
    float current_pose_ = 0.0;
    // 目标距离
    float target_pose_ = 0.0;
    int status_ = example_ros2_interfaces::msg::RobotStatus::STATUS_STOP;
  };
```

该类的实现比较简单，对外提供获取当前状态、获取当前位置和移动一定的距离三个接口，其中移动指定距离这个函数每移动一步会休眠500ms。

3.编写机器人节点逻辑

接着我们就可以利用接口编写机器人节点了

```c++
class ExampleInterfacesRobot : public rclcpp::Node {
public:
  ExampleInterfacesRobot(std::string name) : Node(name) {
    RCLCPP_INFO(this->get_logger(), "节点已启动：%s.", name.c_str());
    /*创建move_robot服务*/
    move_robot_server_ = this-
>create_service<example_ros2_interfaces::srv::MoveRobot>(
      "move_robot", std::bind(&ExampleInterfacesRobot::handle_move_robot, this,
std::placeholders::_1, std::placeholders::_2));
    /*创建发布者*/
    robot_status_publisher_ = this-
>create_publisher<example_ros2_interfaces::msg::RobotStatus>("robot_status", 10);
    /*创建一个周期为500ms的定时器*/
    timer_ = this->create_wall_timer(std::chrono::milliseconds(500),
std::bind(&ExampleInterfacesRobot::timer_callback, this));
  }

private:
  Robot robot; /*实例化机器人*/
  rclcpp::TimerBase::SharedPtr timer_; /*定时器，用于定时发布机器人位置*/
  rclcpp::Service<example_ros2_interfaces::srv::MoveRobot>::SharedPtr
move_robot_server_; /*移动机器人服务*/
  rclcpp::Publisher<example_ros2_interfaces::msg::RobotStatus>::SharedPtr
robot_status_publisher_; /*发布机器人位姿发布者*/

  /**
   * @brief 500ms 定时回调函数，
   *
   */
  void timer_callback() {
    // 创建消息
    example_ros2_interfaces::msg::RobotStatus message;
    message.status = robot.get_status();
    message.pose = robot.get_current_pose();
    RCLCPP_INFO(this->get_logger(), "Publishing: %f", robot.get_current_pose());
    // 发布消息
    robot_status_publisher_->publish(message);
  };
```

```
  /**
   * @brief 收到话题数据的回调函数
   *
   * @param request 请求共享指针，包含移动距离
   * @param response 响应的共享指针，包含当前位置信息
   */
  void handle_move_robot(const
std::shared_ptr<example_ros2_interfaces::srv::MoveRobot::Request> request,

std::shared_ptr<example_ros2_interfaces::srv::MoveRobot::Response> response) {
    RCLCPP_INFO(this->get_logger(), "收到请求移动距离：%f，当前位置：%f", request-
>distance, robot.get_current_pose());
    robot.move_distance(request->distance);
    response->pose = robot.get_current_pose();
  };
};
```

逻辑也比较简单，利用定时器不断发送数据，收到请求后调用机器人类的move_distance接口来移动机器人。

4.编写控制节点

控制节点类代码

头文件部分

```
#include "rclcpp/rclcpp.hpp"
#include "example_ros2_interfaces/srv/move_robot.hpp"
#include "example_ros2_interfaces/msg/robot_status.hpp"
```

`ExampleInterfacesControl` 类

```c++
class ExampleInterfacesControl : public rclcpp::Node {
public:
  ExampleInterfacesControl(std::string name) : Node(name) {
    RCLCPP_INFO(this->get_logger(), "节点已启动：%s.", name.c_str());
```

```
    /*创建move_robot客户端*/
    client_ = this->create_client<example_ros2_interfaces::srv::MoveRobot>(
      "move_robot");
    /*订阅机器人状态话题*/
    robot_status_subscribe_ = this-
>create_subscription<example_ros2_interfaces::msg::RobotStatus>("robot_status",
10, std::bind(&ExampleInterfacesControl::robot_status_callback_, this,
std::placeholders::_1));
  }


  /**
   * @brief 发送移动机器人请求函数
   * 步骤：1.等待服务上线
   *       2.构造发送请求
   *
   * @param distance
   */
  void move_robot(float distance) {
    RCLCPP_INFO(this->get_logger(), "请求让机器人移动%f", distance);

    /*等待服务端上线*/
    while (!client_->wait_for_service(std::chrono::seconds(1))) {
      //等待时检测rclcpp的状态
      if (!rclcpp::ok()) {
        RCLCPP_ERROR(this->get_logger(), "等待服务的过程中被打断...");
        return;
      }
      RCLCPP_INFO(this->get_logger(), "等待服务端上线中");
    }

    // 构造请求
    auto request =
      std::make_shared<example_ros2_interfaces::srv::MoveRobot::Request>();
    request->distance = distance;

    // 发送异步请求，然后等待返回，返回时调用回调函数
    client_->async_send_request(
      request, std::bind(&ExampleInterfacesControl::result_callback_, this,
                         std::placeholders::_1));
  };
```

```
private:
  // 声明客户端
  rclcpp::Client<example_ros2_interfaces::srv::MoveRobot>::SharedPtr client_;
  rclcpp::Subscription<example_ros2_interfaces::msg::RobotStatus>::SharedPtr
robot_status_subscribe_;
  /* 机器人移动结果回调函数 */
  void result_callback_(
    rclcpp::Client<example_ros2_interfaces::srv::MoveRobot>::SharedFuture
      result_future) {
    auto response = result_future.get();
    RCLCPP_INFO(this->get_logger(), "收到移动结果：%f", response->pose);
  }

  /**
   * @brief 机器人状态话题接收回调函数
   *
   * @param msg
   */
  void robot_status_callback_(const
example_ros2_interfaces::msg::RobotStatus::SharedPtr msg)
  {
    RCLCPP_INFO(this->get_logger(), "收到状态数据位置：%f 状态：%d", msg->pose ,msg-
>status);
  }
};
```

main函数

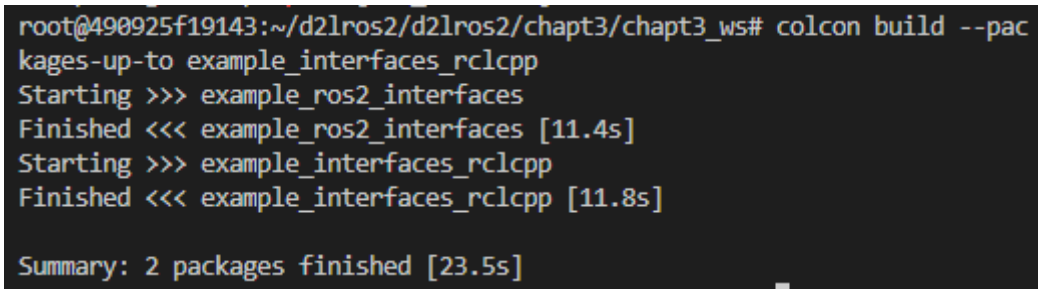
```
c++
int main(int argc, char** argv) {
  rclcpp::init(argc, argv);
  auto node = std::make_shared<ExampleInterfacesControl>
("example_interfaces_control_01");
  /*这里调用了服务，让机器人向前移动5m*/
  node->move_robot(5.0);
  rclcpp::spin(node);
  rclcpp::shutdown();
  return 0;
}
```

5.测试运行

5.1 编译

```shell
colcon build --packages-up-to example_interfaces_rclcpp
```

又遇到了个新的指令 `--packages-up-to` ，在colcon使用进阶篇小鱼有讲到，编译一个节点及其依赖，使用这个指令你会发现，以先后顺序编译了 `example_ros2_interfaces` 再编译 `example_interfaces_rclcpp` 。



5.2 测试

控制端

```shell
source install/setup.bash
ros2 run example_interfaces_rclcpp example_interfaces_control_01
```

服务端

```shell
source install/setup.bash
ros2 run example_interfaces_rclcpp  example_interfaces_robot_01
```

服务端启动后机器人开始移动，时间为： `1654693733.053691559`

```
root@490925f19143:~/d2lros2/d2lros2/chapt3/chapt3_ws# source install/setup.bash
root@490925f19143:~/d2lros2/d2lros2/chapt3/chapt3_ws# ros2 run example_interfaces_rclcpp
example_interfaces_control_01
[INFO] [1654693729.047813277] [example_interfaces_control_01]: 节点已启动：example_interf
aces_control_01.
[INFO] [1654693729.052752907] [example_interfaces_control_01]: 请求让机器人移动5.000000
[INFO] [1654693730.052943385] [example_interfaces_control_01]: 等待服务端上线中
[INFO] [1654693731.053193530] [example_interfaces_control_01]: 等待服务端上线中
[INFO] [1654693732.053361193] [example_interfaces_control_01]: 等待服务端上线中
[INFO] [1654693733.053691559] [example_interfaces_control_01]: 等待服务端上线中
```

```
root@490925f19143:~/d2lros2/d2lros2/chapt3/chapt3_ws# source install/setup.bash
root@490925f19143:~/d2lros2/d2lros2/chapt3/chapt3_ws# ros2 run example_interfaces_rclcpp
  example_interfaces_robot_01
[INFO] [1654693733.979567219] [example_interfaces_robot_01]: 节点已启动：example_interfa
ces_robot_01.
[INFO] [1654693733.982717461] [example_interfaces_robot_01]: 收到请求移动距离：5.000000
，当前位置：0.000000
移动了：0.5当前位置：0.5
移动了：0.45当前位置：0.95
移动了：0.405当前位置：1.355
移动了：0.3645当前位置：1.7195
移动了：0.32805当前位置：2.04755
移动了：0.295245当前位置：2.34279
移动了：0.265721当前位置：2.60852
移动了：0.239148当前位置：2.84766
移动了：0.215234当前位置：3.0629
移动了：0.19371当前位置：3.25661
移动了：0.174339当前位置：3.43095
移动了：0.156905当前位置：3.58785
移动了：0.141215当前位置：3.72907
```

移动结束，`收到移动结果：4.990017`，时间为：`1654693763.501926752`

```
root@490925f19143:~/d2lros2/d2lros2/chapt3/chapt3_ws# source install/setup.bash
root@490925f19143:~/d2lros2/d2lros2/chapt3/chapt3_ws# ros2 run example_interfaces_rclcpp
example_interfaces_control_01
[INFO] [1654693729.047813277] [example_interfaces_control_01]: 节点已启动：example_interf
aces_control_01.
[INFO] [1654693729.052752907] [example_interfaces_control_01]: 请求让机器人移动5.000000
[INFO] [1654693730.052943385] [example_interfaces_control_01]: 等待服务端上线中
[INFO] [1654693731.053193530] [example_interfaces_control_01]: 等待服务端上线中
[INFO] [1654693732.053361193] [example_interfaces_control_01]: 等待服务端上线中
[INFO] [1654693733.053691559] [example_interfaces_control_01]: 等待服务端上线中
[INFO] [1654693763.501926752] [example_interfaces_control_01]: 收到移动结果：4.990017
[INFO] [1654693763.502540846] [example_interfaces_control_01]: 收到状态数据位置：4.990017
 状态：1
[INFO] [1654693763.984007450] [example_interfaces_control_01]: 收到状态数据位置：4.990017
 状态：1
[INFO] [1654693764.483032762] [example_interfaces_control_01]: 收到状态数据位置：4.990017
 状态：1
```

```
移动了：0.0171685当前位置：4.84548
移动了：0.0154516当前位置：4.86094
移动了：0.0139065当前位置：4.87484
移动了：0.0125158当前位置：4.88736
移动了：0.0112642当前位置：4.89862
移动了：0.0101378当前位置：4.90876
移动了：0.00912399当前位置：4.91788
移动了：0.00821161当前位置：4.9261
移动了：0.00739045当前位置：4.93349
移动了：0.006514当前位置：4.94014
移动了：0.00598626当前位置：4.94612
移动了：0.00538764当前位置：4.95151
移动了：0.00484886当前位置：4.95636
移动了：0.00436397当前位置：4.96072
移动了：0.00392756当前位置：4.96465
移动了：0.00353479当前位置：4.96819
移动了：0.00318131当前位置：4.97137
移动了：0.00286317当前位置：4.97423
移动了：0.00257683当前位置：4.97681
移动了：0.00231915当前位置：4.97913
移动了：0.00208721当前位置：4.98122
移动了：0.0018785当前位置：4.98309
移动了：0.00169067当前位置：4.98478
移动了：0.00152159当前位置：4.98631
移动了：0.00136943当前位置：4.98768
移动了：0.00123248当前位置：4.98891
移动了：0.00110922当前位置：4.99002
[INFO] [1654693763.501026271] [example_interfaces_robot_01]: Publishing: 4.990017
[INFO] [1654693763.982997239] [example_interfaces_robot_01]: Publishing: 4.990017
[INFO] [1654693764.482856298] [example_interfaces_robot_01]: Publishing: 4.990017
```

5.3 思考

虽然机器人可以移动了，客户端也可以收到机器人的位置了，但是聪明的你应该发现了，在机器人移动期间，控制端就收不到了来自机器人端的实时位置信息的话题发布了。

原因是服务端调用机器人移动的时候造成了主线程的阻塞和休眠，只有机器人完成移动后才会退出，造成了发布数据的定时器回调无法正常进行。

解决这个问题的方法有很多，比如开个单独给服务开个线程，比如换一种通信方式，带着问题和好奇心，我们继续学习，小鱼将带你解决掉它。

9.自定义接口RCLPY实战

上一节我们使用RCLCPP的API通过自定义接口实现控制节点和机器人节点之间的话题与服务通信。本节我们以RCLPY客户端库为例，给大家讲解实现方法。

1.创建功能包

这里我们依然设计两个节点

- `example_interfaces_robot_02` ，机器人节点，对外提供控制机器人移动服务并发布机器人的状态。
- `example_interfaces_control_02` ，控制节点，发送机器人移动请求，订阅机器人状态话题。

```sherll
cd chapt3_ws/
ros2 pkg create example_interfaces_rclpy --build-type ament_python --dependencies
rclpy example_ros2_interfaces --destination-directory src --node-name
example_interfaces_robot_02 --maintainer-name "fishros" --maintainer-email
"fishros@foxmail.com"
touch src/example_interfaces_rclpy/src/example_interfaces_control_02.py
```

setup.py

```shell
maintainer='fishros',
maintainer_email='fishros@foxmail.com',
entry_points={
        'console_scripts': [
            'example_interfaces_control_02 =
example_interfaces_rclpy.example_interfaces_control_02:main',
            'example_interfaces_robot_02 =
example_interfaces_rclpy.example_interfaces_robot_02:main'
        ],
    },
```

> 这里小鱼又加了两个选项
>
> - `--maintainer-name "fishros"` ，指定拥有者的名字
> - `--maintainer-email "fishros@foxmail.com"` ，指定拥有者邮箱

example_interfaces_robot_02.py

```python
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node

class Robot():
    def __init__(self) -> None:
        pass

class ExampleInterfacesRobot02(Node):
    def __init__(self,name):
        super().__init__(name)
        self.get_logger().info("节点已启动：%s!" % name)

def main(args=None):
    rclpy.init(args=args) # 初始化rclpy
    node = ExampleInterfacesRobot02("example_interfaces_robot_02")  # 新建一个节点
    rclpy.spin(node) # 保持节点运行，检测是否收到退出指令（Ctrl+C）
    rclpy.shutdown() # 关闭rclpy
```

example_interfaces_control_02.py

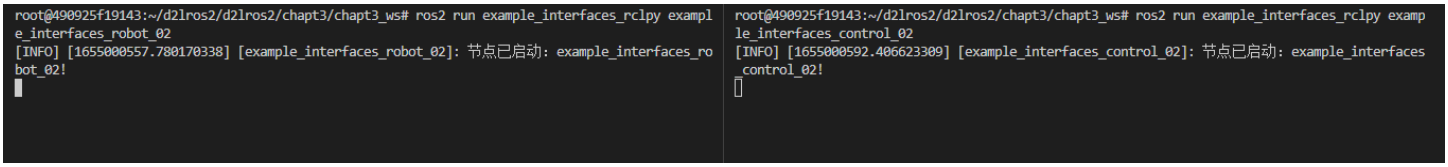
```python
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node

class ExampleInterfacesControl02(Node):
    def __init__(self,name):
        super().__init__(name)
        self.get_logger().info("节点已启动：%s!" % name)

def main(args=None):
    rclpy.init(args=args) # 初始化rclpy
    node = ExampleInterfacesControl02("example_interfaces_control_02")  # 新建一个节
点
    rclpy.spin(node) # 保持节点运行，检测是否收到退出指令（Ctrl+C）
    rclpy.shutdown() # 关闭rclpy
```

编译测试

```
# 新终端
colcon build --packages-up-to example_interfaces_rclpy
source install/setup.bash
ros2 run example_interfaces_rclpy example_interfaces_robot_02
# 新终端
source install/setup.bash
ros2 run example_interfaces_rclpy example_interfaces_control_02
```



2.编写机器人类

源码与解析

```python
from example_ros2_interfaces.msg import RobotStatus
import math
from time import sleep

class Robot():
    def __init__(self) -> None:
        self.current_pose_ = 0.0
        self.target_pose_ = 0.0
        self.status_ = RobotStatus.STATUS_STOP

    def get_status(self):
        return self.status_

    def get_current_pose(self):
        return self.current_pose_

    def move_distance(self,distance):
        self.status_ = RobotStatus.STATUS_MOVEING # 更新状态为移动、
        self.target_pose_ += distance # 更新目标位置
```

```
        while math.fabs(self.target_pose_ - self.current_pose_) > 0.01:
            step = distance / math.fabs(distance) * math.fabs(self.target_pose_ -
self.current_pose_) * 0.1 # 计算一步移动距离
            self.current_pose_  += step # 移动一步
            print(f"移动了：{step}当前位置：{self.current_pose_}")
            sleep(0.5) #休息0.5s
        self.status_ = RobotStatus.STATUS_STOP # 更新状态为停止
        return self.current_pose_
```

3.编写机器人节点

python

```
class ExampleInterfacesRobot02(Node):
    def __init__(self,name):
        super().__init__(name)
        self.get_logger().info("节点已启动：%s!" % name)
        self.robot = Robot()
        self.move_robot_server_ = self.create_service(MoveRobot,"move_robot",
self.handle_move_robot)
        self.robot_status_publisher_ =
self.create_publisher(RobotStatus,"robot_status", 10)
        self.publisher_timer_ = self.create_timer(0.5,
self.publisher_timer_callback)


    def publisher_timer_callback(self):
        """
        定时器回调发布数据函数
        """
        msg = RobotStatus() #构造消息
        msg.status = self.robot.get_status()
        msg.pose = self.robot.get_current_pose()
        self.robot_status_publisher_.publish(msg) # 发布消息
        self.get_logger().info(f'发布了当前的状态：{msg.status} 位置：{msg.pose}')

    def handle_move_robot(self,request, response):
        self.robot.move_distance(request.distance)
```

```
        response.pose = self.robot.get_current_pose()
        return response
```

逻辑与RCLCPP版本一致，创建服务和发布者，并创建定时器定时调用发布者完成发布。

4.编写控制节点

```python
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from example_ros2_interfaces.msg import RobotStatus
from example_ros2_interfaces.srv import MoveRobot


class ExampleInterfacesControl02(Node):
    def __init__(self,name):
        super().__init__(name)
        self.get_logger().info("节点已启动：%s!" % name)
        self.client_ = self.create_client(MoveRobot,"move_robot")
        self.robot_status_subscribe_ =
self.create_subscription(RobotStatus,"robot_status",self.robot_status_callback,10)

    def robot_status_callback(self,msg):
        self.get_logger().info(f"收到状态数据位置：{msg.pose} 状态：{msg.status}")

    def move_result_callback_(self, result_future):
        response = result_future.result()
        self.get_logger().info(f"收到返回结果：{response.pose}")

    def move_robot(self, distance):
        while rclpy.ok() and self.client_.wait_for_service(1)==False:
            self.get_logger().info(f"等待服务端上线....")
        request = MoveRobot.Request()
        request.distance = distance
        self.get_logger().info(f"请求服务让机器人移动{distance}")

self.client_.call_async(request).add_done_callback(self.move_result_callback_)
```

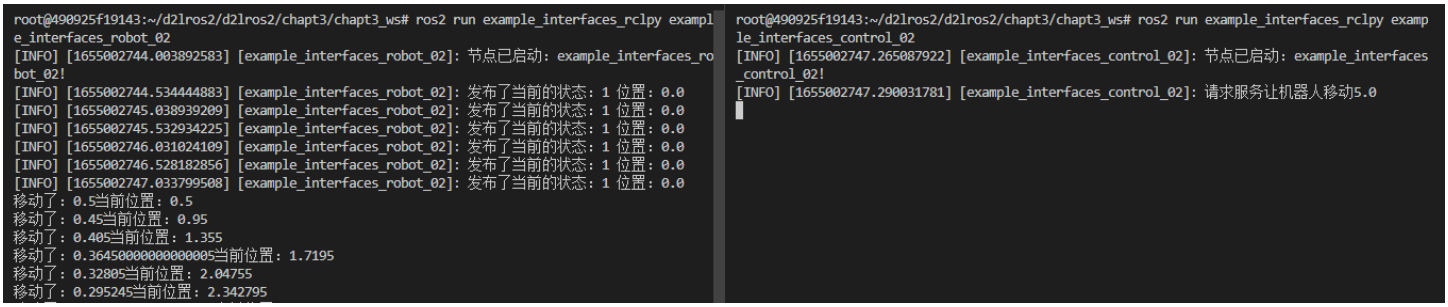
```
    def main(args=None):
        rclpy.init(args=args) # 初始化rclpy
        node = ExampleInterfacesControl02("example_interfaces_control_02")  # 新建一个节
    点
        node.move_robot(5.0) #移动5米
        rclpy.spin(node) # 保持节点运行，检测是否收到退出指令（Ctrl+C）
        rclpy.shutdown() # 关闭rclpy
```

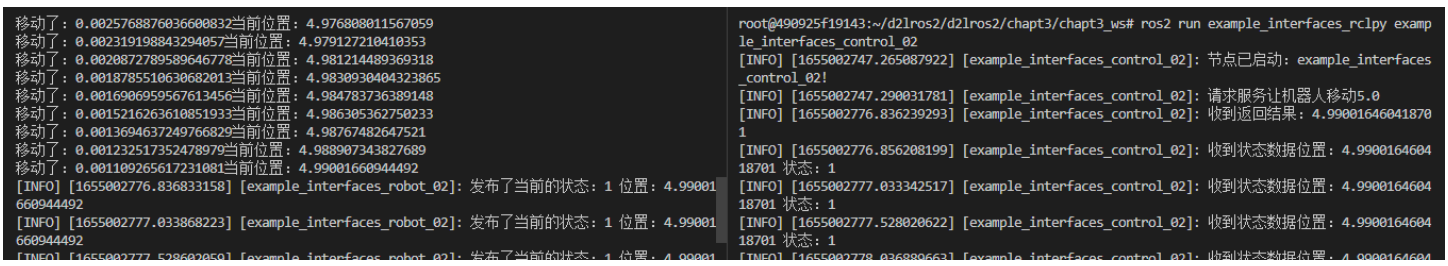
控制节点逻辑也与 `RCLCPP` 版本一致，创建一个订阅者和客户端，在主函数中请求服务端进行移动。

5.运行测试

```
# 新终端
colcon build --packages-up-to example_interfaces_rclpy
source install/setup.bash
ros2 run example_interfaces_rclpy example_interfaces_robot_02
# 新终端
source install/setup.bash
ros2 run example_interfaces_rclpy example_interfaces_control_02
```

同样的，你会发现在机器人移动期间是机器人节点并没有发布机器人位姿出来，在进阶篇中我们可以使用ROS2的多线程执行器和回调组来解决这个问题。





在ROS2中定义接口，需要编写一个接口文件，该文件后缀为 `msg` 、 `srv` 、 `action` 。

在接口文件中定义通信过程中所使用的数据类型和数据名称，那可用的数据类型和数据名称有哪些呢？今天小鱼就带你详细了解一下ROS2接口文件中的数据类型和数据名称。

1.数据名称

数据名称就是一个字符串，没啥好说的，符合编程语言变量的命名规则就行（比如不能是数字开头）

2.数据类型

2.1 数据类型有哪些呢？

这里小鱼可以告诉你，原始的数据类型只有九类。其中每一个都可以在后面加上 `[]` 将其变成数组形式（从一个变成多个）

```
bool
byte
char
float32, float64
int8, uint8
int16, uint16
int32, uint32
int64, uint64
string
```

> 上面这九类中，官方也在考虑新增一些和删除一些，目前还是支持的，后续小鱼会根据资料再更新一下本文。

2.2 类型扩展（套娃）

2.2.1 第一层套娃

ROS2基于上面的九类基础数据类型，为我们定义出了很多拿来就用的数据类型，比如我们在前面章节中用到的图像数据类型 `sensor_msgs/Image`,我们可以使用下面的命令来看一下其组成：

```
ros2 interface show sensor_msgs/msg/Image
```

```
ros2@ubuntu:~/code/town_ws$ ros2 interface show sensor_msgs/msg/Image
# This message contains an uncompressed image
# (0, 0) is at top-left corner of image

std_msgs/Header header # Header timestamp should be acquisition time of image
                      # Header frame_id should be optical frame of camera
                      # origin of frame should be optical center of cameara
                      # +x should point to the right in the image
                      # +y should point down in the image
                      # +z should point into to plane of the image
                      # If the frame_id here and the frame_id of the CameraInfo
                      # message associated with the image conflict
                      # the behavior is undefined

uint32 height         # image height, that is, number of rows
uint32 width          # image width, that is, number of columns

# The legal values for encoding are in file src/image_encodings.cpp
# If you want to standardize a new string format, join
# ros-users@lists.ros.org and send an email proposing a new encoding.

string encoding       # Encoding of pixels -- channel meaning, ordering, size
                      # taken from the list of strings in include/sensor_msgs/image_encodings.hpp

uint8 is_bigendian    # is this data bigendian?
uint32 step           # Full row length in bytes
uint8[] data          # actual matrix data, size is (step * rows)
```

去掉单行的注释后的样子如下：

```
std_msgs/Header header # Header timestamp should be acquisition time of image
uint32 height         # image height, that is, number of rows
uint32 width          # image width, that is, number of columns
string encoding       # Encoding of pixels -- channel meaning, ordering, size
uint8 is_bigendian    # is this data bigendian?
uint32 step           # Full row length in bytes
uint8[] data          # actual matrix data, size is (step * rows)
```

我们可以看到，除了第一行 `std_msgs/Header header` 之外的其他部分都是由基础类型组成。

2.2.2 第二层套娃

那 `std_msgs/Header` 由什么组成呢？我们再次使用下面的指令查看一下：

```
ros2 interface show std_msgs/msg/Header
```

结果如下：

```
builtin_interfaces/Time stamp # Two-integer timestamp that is expressed as seconds
and nanoseconds.
string frame_id # Transform frame with which this data is associated.
```

2.2.3 第三层套娃

看完上面的结果，除了基本类型 `string` 和我们发现还有一层 `builtin_interfaces/Time` ，我们再查看一下这个接口类型。

```
ros2 interface show builtin_interfaces/msg/Time
```

结果如下：

```
# Time indicates a specific point in time, relative to a clock's 0 point.

# The seconds component, valid over all int32 values.
int32 sec

# The nanoseconds component, valid in the range [0, 10e9).
uint32 nanosec
```

我们发现结果全都是基本类型了，终于我们把套娃给解开了。

3.接口类型总结

通过基本类型的组合，可以构成一个新的数据类型，而新的数据类型又可以和基本类型或者另外一个数据类型互相组成另一个数据类型。所以我们可以说ROS2中的数据类型有无数种。

4.总结

相信看完这篇文章，以后再也不会遇到看不懂的数据类型了。

2.通信质量Qos配置指南

请参考通信质量Qos配置指南

大家好，我是小鱼，昨天被群友在群里催了DDS相关的文章里，说好的来源码体验一下DDS来着，小鱼不能再咕咕咕了，所以今天就分享一下，FastDDS的安装和体验。

1.论FastDDS的三种打开方式

FastDDS和普通ROS包一样，有二进制安装、源码编译、Docker三种安装方式。

因为官方把二进制和Docker放到了官网。。 而且要填写个人信息才能下载。。 而且下载速度超级超级慢。。 而且不方便观摩源码。。 所以小鱼带你一起从源码进行安装。

本来想做成一键安装的，省的大家敲脚本了！！

爱一个人绝对不能惯着他，鱼粉也是，为了让大家多复制粘贴时多思考一下（其实是为了凑文章字数），小鱼就带大家一起一步步安装编译源码

不过源码安装也很简单，大家不要害怕。。

因为DDS和ROS2相关，我们也可以使用colcon来编译，就不用cmake了(有需要cmake的自行到官网找)

2.源码编译安装FastDDS

下载编译DDS分为三步，第一步如果你已经安装了ROS2可以跳过。。

1.安装工具和依赖库

安装工具

```
sudo apt install python3-colcon-common-extensions python3-vcstool zip openjdk-8-jdk  -y
```

安装依赖库

```
sudo apt-get install libasio-dev -y
```

2.创建目录，下载仓库

```
mkdir -p fastdds_ws/src
cd fastdds_ws && wget https://downloads.gradle-dn.com/distributions/gradle-6.4-bin.zip && unzip gradle-6.4-bin.zip
wget http://fishros.com/tools/files/fastrtps.repos && vcs import src < fastrtps.repos
```

> 安装Fast DDS依赖项的 repos 文件时出现404：Not Found

3.编译

```
colcon build
cd src/fastddsgen/ &&  gradle assemble
```

最后一步:配置环境变量

xxx是你的目录前缀

```
echo 'source xxx/fastdds_ws/install/setup.bash' >> ~/.bashrc
echo 'export PATH=$PATH:xxx/fastdds_ws/gradle-6.4/bin/' >> ~/.bashrc
echo 'export DDSGEN=xxx/fastdds_ws/src/fastddsgen/scripts' >> ~/.bashrc
```

3.HelloFish例程

DDS使用的RTPS，就是Real-Time Publish Subscribe协议，其实和ROS与ROS2中的发布订阅的感觉时一样的，所以我们就跑一个例程来收发消息，消息内容就叫 `HelloFish`

小鱼写的程序已经准备好了，放到了github上，大家可以直接下载下来编译测试哦~

下载代码

```
git clone https://github.com/fishros/dds_tutorial.git
```

编译例程

```
cd dds_tutorial/examples/01-hellofishros
mkdir build && cd build
cmake ..
make
```

执行例程

开一个终端

```
./DDSHelloFishRosPublisher
```

再开一个终端

```
./DDSHelloFishRosSubscribe
```

查看结果

正确结果像下面这样子，已经证明一切OK了~

```
(base) root@d4864f4ccde3:~/code/hello_dds/build# ./DDSHelloFishRosPublisher
Starting publisher.
Publisher matched.
Message: HelloFishRos!关注小鱼!学习机器人不迷路~ with index: 1 SENT
Message: HelloFishRos!关注小鱼!学习机器人不迷路~ with index: 2 SENT
Message: HelloFishRos!关注小鱼!学习机器人不迷路~ with index: 3 SENT
Message: HelloFishRos!关注小鱼!学习机器人不迷路~ with index: 4 SENT
Message: HelloFishRos!关注小鱼!学习机器人不迷路~ with index: 5 SENT
Message: HelloFishRos!关注小鱼!学习机器人不迷路~ with index: 6 SENT
Message: HelloFishRos!关注小鱼!学习机器人不迷路~ with index: 7 SENT
Message: HelloFishRos!关注小鱼!学习机器人不迷路~ with index: 8 SENT
Message: HelloFishRos!关注小鱼!学习机器人不迷路~ with index: 9 SENT
Message: HelloFishRos!关注小鱼!学习机器人不迷路~ with index: 10 SENT
```

```
(base) root@d4864f4ccde3:~/code/hello_dds/build# ./DDSHelloFishRosSubscribe
Starting subscriber.
Subscriber matched.
Message: HelloFishRos!关注小鱼!学习机器人不迷路~ with index: 1 RECEIVED.
Message: HelloFishRos!关注小鱼!学习机器人不迷路~ with index: 2 RECEIVED.
Message: HelloFishRos!关注小鱼!学习机器人不迷路~ with index: 3 RECEIVED.
Message: HelloFishRos!关注小鱼!学习机器人不迷路~ with index: 4 RECEIVED.
Message: HelloFishRos!关注小鱼!学习机器人不迷路~ with index: 5 RECEIVED.
Message: HelloFishRos!关注小鱼!学习机器人不迷路~ with index: 6 RECEIVED.
Message: HelloFishRos!关注小鱼!学习机器人不迷路~ with index: 7 RECEIVED.
Message: HelloFishRos!关注小鱼!学习机器人不迷路~ with index: 8 RECEIVED.
Message: HelloFishRos!关注小鱼!学习机器人不迷路~ with index: 9 RECEIVED.
Message: HelloFishRos!关注小鱼!学习机器人不迷路~ with index: 10 RECEIVED.
```

4.总结

看到熟悉的发布订阅是不是很神奇，FASTDDS底层采用了多种协议进行数据的传输，包括不靠谱但真的很快的UDP，靠谱但是不怎么快的TCP，还有感觉不传输的内存交换（SHM)。

为了给大家展示一下什么叫做低产（懒），小鱼决定明天再给大家讲解代码~

第 4 章 ROS2通信之参数与动作

- 章节导读
- 基础篇-控制概述
  - 机器人控制概述
- 入门篇-参数与动作 1.参数（Param）通信 2.参数之RCLCPP实现 3.参数之RCLPY实现 4.动作（Action）通信与自定义接口 5.动作之RCLCPP实现 6.动作之RCLPY实现 7.通信机制对比总结
- 进阶篇-原理进阶
  - 参数与动作通信机制原理
  - 3.生命周期节点介绍
  - 高效的ROS2进程内通信
  - 使用ZeroMQ进行订阅发布

1.开关控制与闭环控制

机器人的控制可以分为两种类型：开环控制和闭环控制。

一、闭环控制介绍

开环控制是指机器人按照预先设定的命令执行任务，但并不会对执行过程中的状态进行反馈和调整。简单来说，开环控制就是机器人盲目地按照指令执行任务，不考虑实际执行情况是否符合预期。开环控制的优点是简单易用，适用于一些简单的任务，如基本的运动控制或简单的搬运。但是，它的缺点也很明显，因为机器人无法感知执行任务的实际情况，因此无法自动调整行动，导致执行任务的成功率低，可靠性差。

二、闭环控制介绍

闭环控制是指机器人通过传感器或其他检测设备获取执行任务过程中的状态信息，将这些信息反馈给控制系统，从而实现对机器人执行任务过程中的实时控制和调整。在闭环控制中，机器人执行任务的过程中会根据反馈信息调整执行动作，确保机器人按照预期的方式完成任务。闭环控制的优点是能够根据实际情况进行实时调整，提高了机器人执行任务的成功率和可靠性。

三、哪个更常用？

在机器人应用中，闭环控制更加普遍，因为它能够根据反馈信息实时调整机器人的动作，确保机器人按照预期的方式执行任务。同时，闭环控制还可以帮助机器人适应不同的工作环境，增强机器人的鲁棒性和自适应能力，实现更高效、更精准的控制。在一些高精度和复杂的应用场景中，闭环控制已经成为机器人控制的标准，比如精密加工、医疗手术、自动驾驶等。

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

1.ROS2参数通信介绍

大家好，我是爱吃瓜皮的小鱼。在前面的机器人控制示例中我们，机器人每移动一步休息500ms，加入我们想让机器人休息时间少一些，就需要手动的修改源码减少其值，这非常的不方便。

在机器人开发中，会有很多参数和设置可以后期需要调整的，如果都放到源码里很难实现动态修改和管理，ROS2为了解决这一问题，提出了参数这一通信机制。

1.参数通信是什么？

1.1 参数定义

ROS2官方对参数的定义是：

A parameter is a configuration value of a node. You can think of parameters as node settings.

有请翻译官小鱼

参数是节点的一个配置值，你可以任务参数是一个节点的设置

感谢小鱼同学的精彩翻译，ROS2的参数就是节点的设置，和我们上面提出的需求不谋而合，有了参数我们就可以实现动态的改变李四写小说的速度了

1.2 参数组成成分

ROS2参数是由键值对组成的，此话怎讲？键值对指的是就是名字和数值，比方说

- 名字：李四写小说周期，值：5s
- 名字：显示器亮度,值：60%

名字的数据类型小鱼不多说肯定是字符串了，值的数据类型呢？我们这里用到的是5是整形数据，显然只有一个整形是不够用的，ROS2支持的参数值的类型如下：

- bool 和bool[]，布尔类型用来表示开关，比如我们可以控制雷达控制节点，开始扫描和停止扫描。
- int64 和int64[]，整形表示一个数字，含义可以自己来定义，这里我们可以用来表示李四节点写小说的周期值
- float64 和float64[]，浮点型，可以表示小数类型的参数值
- string 和string[]，字符串，可以用来表示雷达控制节点中真实雷达的ip地址
- byte[]，字节数组，这个可以用来表示图片，点云数据等信息

2.体验参数

我们使用乌龟模拟器来体验一下参数，同时讲解一下常用的参数的命令行工具。

2.1 运行小乌龟模拟器节点和小乌龟控制节点

打开终端

```
ros2 run turtlesim turtlesim_node
```

再打开一个终端

```
ros2 run turtlesim turtle_teleop_key
```

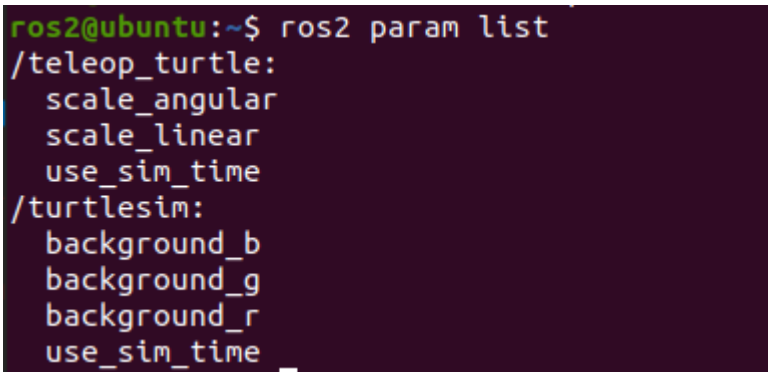
可以看到下面的蓝蓝的模拟器



2.2 查看节点有哪些参数（设置）

我们可以使用下面的指令来查看所有节点的参数列表，打开一个终端，运行下面的指令

```
ros2 param list
```



写代码为什么要做到见名知意?我们看到乌龟模拟器的四个参数，background背景bgr指的是blue、green、red。简而言之就是背景颜色。那这几个参数应该可以控制乌龟模拟器的背景颜色。

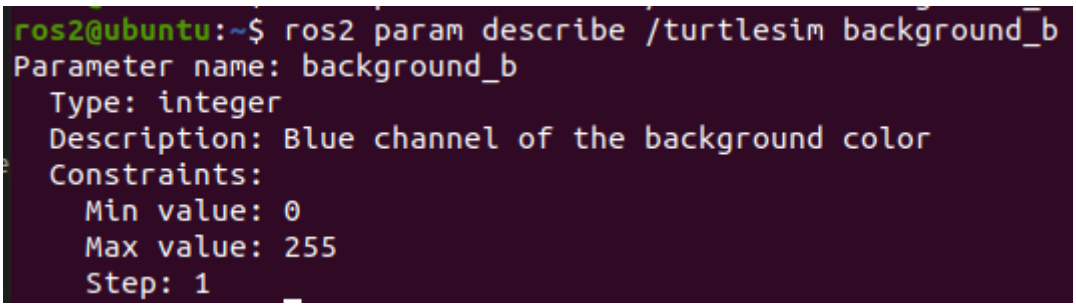
> 最后一个use_sim_time是每个节点都带的，后面小鱼写篇文章稍微讲讲。

如果看不懂，还可以有一个方法详细查看一个参数的信息。

```
ros2 param describe <node_name> <param_name>
```

比如：

```
ros2 param describe /turtlesim background_b
```



这里就可以详细的看到参数的名字，参数的描述，参数的类型，还有对参数的约束，最大值最小值等。

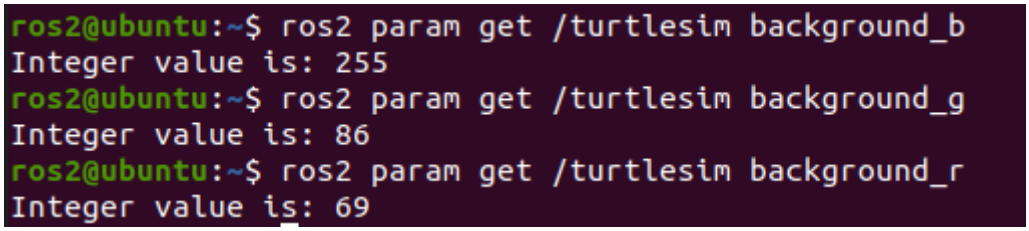
2.3 查看参数值

参数的组成由名字和值（键值组成），名字可以通过 `param list` 获取，值该使用指令获取呢？

下面这个命令行工具可以帮助我们获取参数的值

```
ros2 param get /turtlesim background_b
```

运行一下，你会发现结果是255，蓝色进度条打满，再看看r红色和g绿色。



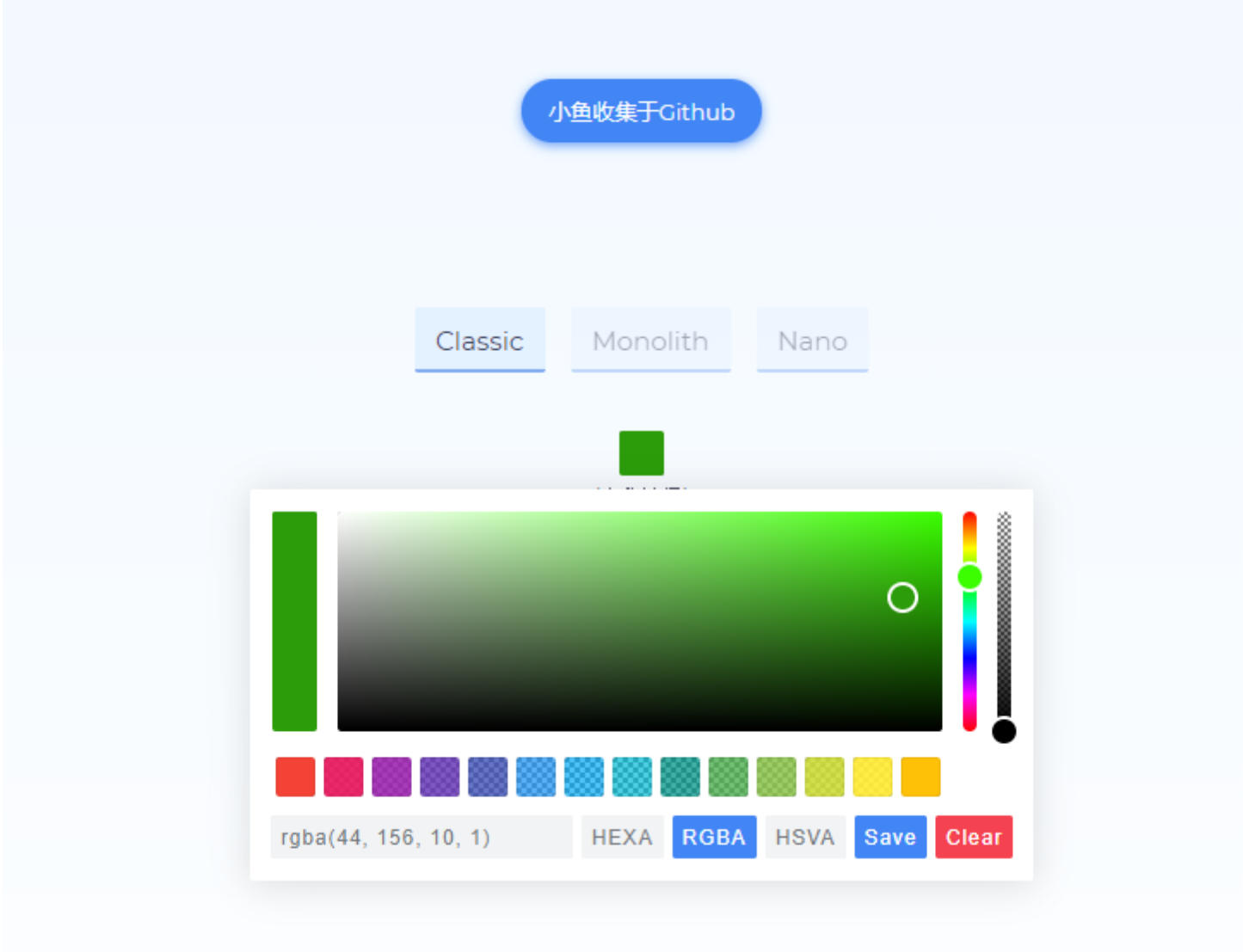
分别是255,86,69

2.4 设置参数

找到了参数和值，接着我们来改变一下乌龟模拟器的颜色。

打开小鱼精心准备的在线工具：https://fishros.com/tools/pickr

选取一个自己喜欢的颜色，这里小鱼就选绿色，因为乌龟模拟器换成绿色的应该很奇怪。



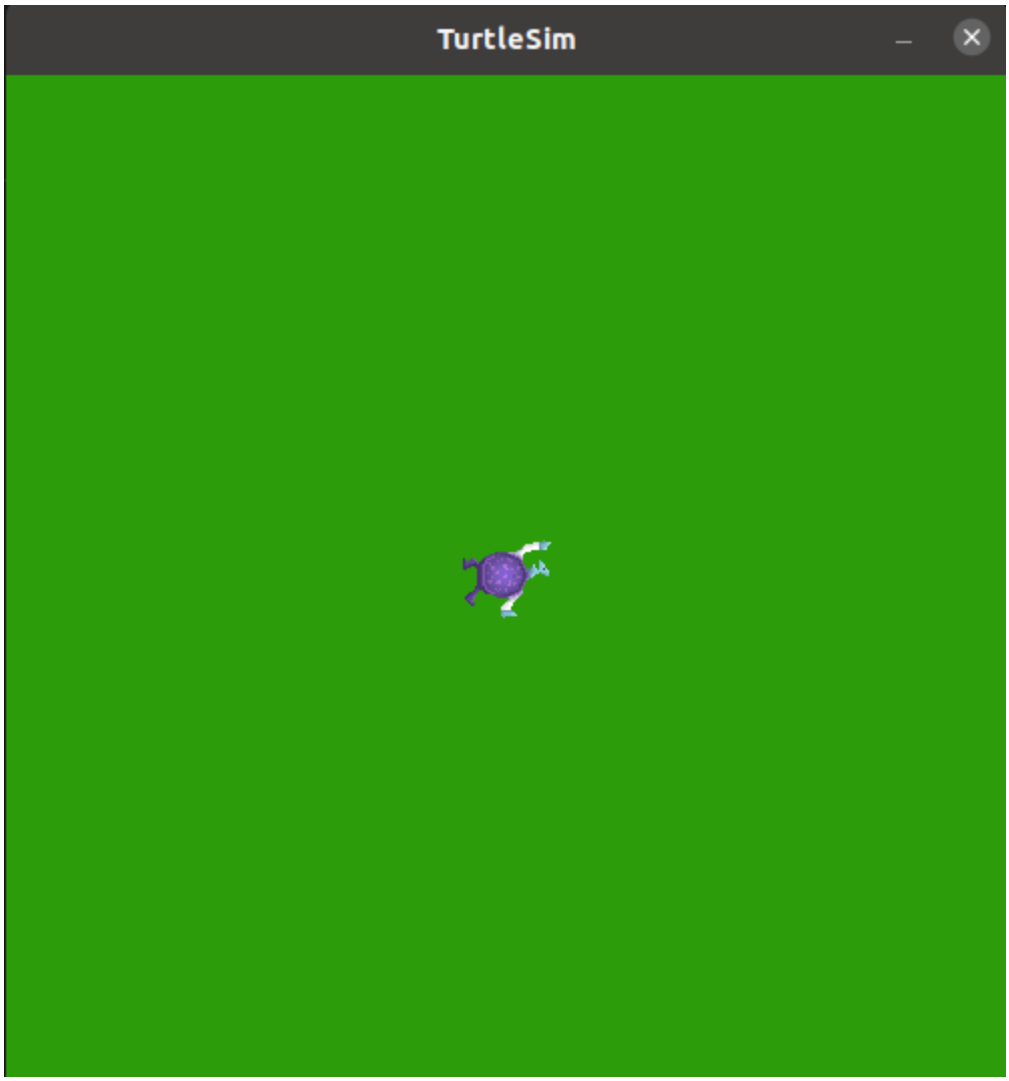
可以看到当前的这个颜色,r为44，g为156，b为10，接着我们可以使用下面的指令来设置参数的值。

```
ros2 param set <node_name> <parameter_name> <value>
```

我们依次修改参数值：

```
ros2 param set /turtlesim background_r 44
ros2 param set /turtlesim background_g 156
ros2 param set /turtlesim background_b 10
```

接着你可以看到这样的颜色的乌龟模拟器（绿的令人发慌）



> 需要留意的是，我们修改的背景数据并没有被存储，只是临时修改。重新启动节点乌龟模拟器依然还是原来的蓝色，不是我们想要的绿色的。

2.5 把参数存起来

把参数存起来其实就相当去把当前的参数值拍一张快照，然后保存下来，后面可以用于恢复参数到当前的数值。

可以使用下面的命令进行操作：

```
ros2 param dump <node_name>
```

2.5.1 给乌龟模拟器参数拍照

比如我们要保存乌龟模拟器的节点数据，可以采用下面的指令;

```
ros2 param dump /turtlesim
```

```
ros2@ubuntu:~$ ros2 param dump /turtlesim
Saving to:  ./turtlesim.yaml
```

文件被保存成了yaml格式，用cat指令看一看

```
cat ./turtlesim.yaml
```

```
ros2@ubuntu:~$ cat ./turtlesim.yaml
/turtlesim:
  ros__parameters:
    background_b: 10
    background_g: 156
    background_r: 44
    use_sim_time: false
```

2.5.2 恢复参数值

我们 `Ctrl+C` 关闭乌龟模拟器，然后再重新运行。

```
ros2 run turtlesim turtlesim_node
```



可以看到模拟器又变成了蓝色了，接着通过param的load的方法把参数值恢复成我们之前存储的。

```
ros2 param load  /turtlesim ./turtlesim.yaml
```

```
ros2@ubuntu:~$ ros2 param load  /turtlesim ./turtlesim.yaml
Set parameter background_b successful
Set parameter background_g successful
Set parameter background_r successful
Set parameter use_sim_time successful
```

几乎是瞬间，乌龟模拟器又被我们搞绿了

2.5.3 启动节点时加载参数快照

有什么办法一开始就让乌龟模拟器变成绿色？答案有的。

ros2 的run 指令支持下面这种 `骚操作` 。

```
ros2 run <package_name> <executable_name> --ros-args --params-file <file_name>
```

关闭我们的乌龟模拟器，使用下面的指令重新运行

```
ros2 run turtlesim turtlesim_node --ros-args --params-file ./turtlesim.yaml
```



可以看到一上来就时绿了的模拟器。

3.总结

本节小鱼带大家一起学习了参数的基本概念，同时动手把参数的命令行工具玩了一遍。

2.参数之RCLCPP实现

上节我们通过参数控制了小乌龟模拟器的背景色，但是我们并不知道小乌龟模拟器是如何接收到参数并将其应用的，本节我们就学习使用ROS2的RCLCPP中参数相关的API实现对ROS2打印的日志级别控制。

ROS2将日志分为五个级别，在RCLCPP中通过不同的宏可以实现不同日志级别日志的打印，例程如下：

```cpp
RCLCPP_DEBUG(this->get_logger(), "我是DEBUG级别的日志，我被打印出来了!");
RCLCPP_INFO(this->get_logger(), "我是INFO级别的日志，我被打印出来了!");
RCLCPP_WARN(this->get_logger(), "我是WARN级别的日志，我被打印出来了!");
RCLCPP_ERROR(this->get_logger(), "我是ERROR级别的日志，我被打印出来了!");
RCLCPP_FATAL(this->get_logger(), "我是FATAL级别的日志，我被打印出来了!");
```

有时候日志太多，会让人眼花缭乱找不到重要信息，所以我们需要对日志的级别进行过滤，比如只看INFO以上级别的，ROS2中可以通过已有的API设置日志的级别，RCLCPP中API如下：

```
this->get_logger().set_level(log_level);
```

1.创建功能包和节点

我们创建一个功能包和测试节点，声明参数并实现动态修改打印的日志级别功能。

```shell
mkdir -p chapt4/chapt4_ws/
ros2 pkg create example_parameters_rclcpp --build-type ament_cmake --dependencies
rclcpp --destination-directory src --node-name parameters_basic --maintainer-name
"fishros" --maintainer-email "fishros@foxmail.com"
```

parameters_basic.cpp

```c++
#include <chrono>
#include "rclcpp/rclcpp.hpp"

class ParametersBasicNode : public rclcpp::Node {
```

```
 public:
  explicit ParametersBasicNode(std::string name) : Node(name) {
    RCLCPP_INFO(this->get_logger(), "节点已启动：%s.", name.c_str());
  }
 private:
};

int main(int argc, char** argv) {
  rclcpp::init(argc, argv);
  /*创建对应节点的共享指针对象*/
  auto node = std::make_shared<ParametersBasicNode>("parameters_basic");
  /* 运行节点，并检测退出信号*/
  rclcpp::spin(node);
  rclcpp::shutdown();
  return 0;
}
```

构建测试

```shell
colcon build  --packages-select example_parameters_rclcpp
source install/setup.bash
ros2 run example_parameters_rclcpp parameters_basic
```

2.RCLCPP参数API

在RCLCPP的API中，关于参数相关的函数比较多些，但都是围绕**参数获取**、**参数设置**、**参数描述**、**列出参数**、**添加**和**移除**参数回调事件。

- rclcpp: rclcpp: ROS Client Library for C++ (ros2.org)

3.使用参数控制节点日志级别

```cpp
#include <chrono>
#include "rclcpp/rclcpp.hpp"
/*
```

```
    # declare_parameter             声明和初始化一个参数
    # describe_parameter(name)  通过参数名字获取参数的描述
    # get_parameter                 通过参数名字获取一个参数
    # set_parameter                 设置参数的值
*/
class ParametersBasicNode : public rclcpp::Node {
 public:
  // 构造函数,有一个参数为节点名称
  explicit ParametersBasicNode(std::string name) : Node(name) {
    RCLCPP_INFO(this->get_logger(), "节点已启动：%s.", name.c_str());
    this->declare_parameter("rcl_log_level", 0);     /*声明参数*/
    this->get_parameter("rcl_log_level", log_level); /*获取参数*/
    /*设置日志级别*/
    this->get_logger().set_level((rclcpp::Logger::Level)log_level);
    using namespace std::literals::chrono_literals;
    timer_ = this->create_wall_timer(
        500ms, std::bind(&ParametersBasicNode::timer_callback, this));
  }

 private:
  int log_level;
  rclcpp::TimerBase::SharedPtr timer_;

  void timer_callback() {
    this->get_parameter("rcl_log_level", log_level); /*获取参数*/
    /*设置日志级别*/
    this->get_logger().set_level((rclcpp::Logger::Level)log_level);
    std::cout<<"======================================================="
<<std::endl;
    RCLCPP_DEBUG(this->get_logger(), "我是DEBUG级别的日志，我被打印出来了!");
    RCLCPP_INFO(this->get_logger(), "我是INFO级别的日志，我被打印出来了!");
    RCLCPP_WARN(this->get_logger(), "我是WARN级别的日志，我被打印出来了!");
    RCLCPP_ERROR(this->get_logger(), "我是ERROR级别的日志，我被打印出来了!");
    RCLCPP_FATAL(this->get_logger(), "我是FATAL级别的日志，我被打印出来了!");
  }
};

int main(int argc, char** argv) {
  rclcpp::init(argc, argv);
  /*创建对应节点的共享指针对象*/
  auto node = std::make_shared<ParametersBasicNode>("parameters_basic");
  /* 运行节点，并检测退出信号*/
```

```
  rclcpp::spin(node);
  rclcpp::shutdown();
  return 0;
}
```

代码解析

这里我们使用了三个参数相关的函数和一个设置节点日志级别的函数

- declare_parameter，参数有两个参数名和参数值。
- get_parameter，参数有两个，参数名和放入结果的变量。

设置日志级别

- set_level，设置日志级别，ROS2的日志级别定义在文件 `/opt/ros/humble/include/rcutils/rcutils/logging.h` 的167-175行。

```cpp
/// The severity levels of log messages / loggers.
enum RCUTILS_LOG_SEVERITY
{
  RCUTILS_LOG_SEVERITY_UNSET = 0,  ///< The unset log level
  RCUTILS_LOG_SEVERITY_DEBUG = 10,  ///< The debug log level
  RCUTILS_LOG_SEVERITY_INFO = 20,  ///< The info log level
  RCUTILS_LOG_SEVERITY_WARN = 30,  ///< The warn log level
  RCUTILS_LOG_SEVERITY_ERROR = 40,  ///< The error log level
  RCUTILS_LOG_SEVERITY_FATAL = 50,  ///< The fatal log level
};
```

4.编译测试

```shell
colcon build --packages-select example_parameters_rclcpp
source install/setup.bash
ros2 run example_parameters_rclcpp parameters_basic
```

运行后你会发现DEBUG级别的日志并没有被打印出来，原因在于我们将节点的日志级别设置为了0，0对应的日志级别为 `RCUTILS_LOG_SEVERITY_UNSET` 即未设置使用默认级别，**节点默认的日志级别就是INFO级别**的，

所以只能打印INFO以上的日志信息。

```
root@490925f19143:~/d2lros2/d2lros2/chapt4/chapt4_ws# ros2 run example_parameters_rclcpp parameters_basic
[INFO] [1655141696.106985244] [parameters_basic]: 节点已启动: parameters_basic.
==================================================
[INFO] [1655141696.607344515] [parameters_basic]: 我是INFO级别的日志，我被打印出来了!
[WARN] [1655141696.607399755] [parameters_basic]: 我是WARN级别的日志，我被打印出来了!
[ERROR] [1655141696.607413604] [parameters_basic]: 我是ERROR级别的日志，我被打印出来了!
[FATAL] [1655141696.607420224] [parameters_basic]: 我是FATAL级别的日志，我被打印出来了!
==================================================
[INFO] [1655141697.107369735] [parameters_basic]: 我是INFO级别的日志，我被打印出来了!
[WARN] [1655141697.107481692] [parameters_basic]: 我是WARN级别的日志，我被打印出来了!
[ERROR] [1655141697.107546009] [parameters_basic]: 我是ERROR级别的日志，我被打印出来了!
[FATAL] [1655141697.107563083] [parameters_basic]: 我是FATAL级别的日志，我被打印出来了!
==================================================
[INFO] [1655141697.607320678] [parameters_basic]: 我是INFO级别的日志，我被打印出来了!
[WARN] [1655141697.607387321] [parameters_basic]: 我是WARN级别的日志，我被打印出来了!
[ERROR] [1655141697.607394267] [parameters_basic]: 我是ERROR级别的日志，我被打印出来了!
[FATAL] [1655141697.607398030] [parameters_basic]: 我是FATAL级别的日志，我被打印出来了!
```

运行节点的时候可以指定参数的值，我们尝试将 `log_level` 的值改成 `10` 即 `DEBUG` 级别。

```shell
ros2 run example_parameters_rclcpp parameters_basic --ros-args -p
rcl_log_level:=10
```

```
root@490925f19143:~/d2lros2/d2lros2/chapt4/chapt4_ws# ros2 run example_parameters_rclcpp parameters_basic --ros-args -p rcl_log_level:=10
[INFO] [1655143516.378360571] [parameters_basic]: 节点已启动: parameters_basic.
==========10==========
[DEBUG] [1655143516.878575239] [parameters_basic]: 我是DEBUG级别的日志，我被打印出来了!
[INFO] [1655143516.878620798] [parameters_basic]: 我是INFO级别的日志，我被打印出来了!
[WARN] [1655143516.878635344] [parameters_basic]: 我是WARN级别的日志，我被打印出来了!
[ERROR] [1655143516.878649736] [parameters_basic]: 我是ERROR级别的日志，我被打印出来了!
[FATAL] [1655143516.878670705] [parameters_basic]: 我是FATAL级别的日志，我被打印出来了!
==========10==========
[DEBUG] [1655143517.378570016] [parameters_basic]: 我是DEBUG级别的日志，我被打印出来了!
[INFO] [1655143517.378614943] [parameters_basic]: 我是INFO级别的日志，我被打印出来了!
[WARN] [1655143517.378629255] [parameters_basic]: 我是WARN级别的日志，我被打印出来了!
[ERROR] [1655143517.378643196] [parameters_basic]: 我是ERROR级别的日志，我被打印出来了!
[FATAL] [1655143517.378657436] [parameters_basic]: 我是FATAL级别的日志，我被打印出来了!
==========10==========
[DEBUG] [1655143517.878571992] [parameters_basic]: 我是DEBUG级别的日志，我被打印出来了!
[INFO] [1655143517.878622128] [parameters_basic]: 我是INFO级别的日志，我被打印出来了!
[WARN] [1655143517.878636777] [parameters_basic]: 我是WARN级别的日志，我被打印出来了!
[ERROR] [1655143517.878651059] [parameters_basic]: 我是ERROR级别的日志，我被打印出来了!
[FATAL] [1655143517.878665258] [parameters_basic]: 我是FATAL级别的日志，我被打印出来了!
```

再试试其他级别-FATAL

```
root@490925f19143:~/d2lros2/d2lros2/chapt4/chapt4_ws# ros2 run example_parameters_rclcpp parameters_basic --ros-args -p rcl_log_level:=50
[INFO] [1655143558.458583765] [parameters_basic]: 节点已启动: parameters_basic.
==========================50============================
[FATAL] [1655143558.958798505] [parameters_basic]: 我是FATAL级别的日志，我被打印出来了!
==========================50============================
[FATAL] [1655143559.458778649] [parameters_basic]: 我是FATAL级别的日志，我被打印出来了!
==========================50============================
[FATAL] [1655143559.958824430] [parameters_basic]: 我是FATAL级别的日志，我被打印出来了!
==========================50============================
[FATAL] [1655143560.458777964] [parameters_basic]: 我是FATAL级别的日志，我被打印出来了!
==========================50============================
[FATAL] [1655143560.958786848] [parameters_basic]: 我是FATAL级别的日志，我被打印出来了!
==========================50============================
[FATAL] [1655143561.458781981] [parameters_basic]: 我是FATAL级别的日志，我被打印出来了!
==========================50============================
[FATAL] [1655143561.958788323] [parameters_basic]: 我是FATAL级别的日志，我被打印出来了!
```

除了在节点运行前通过CLI传递参数，在运动的过程中也可以动态的修改参数

```shell
#查看参数列表
ros2 param list
#设置参数级别
ros2 param set /parameters_basic rcl_log_level 10
```

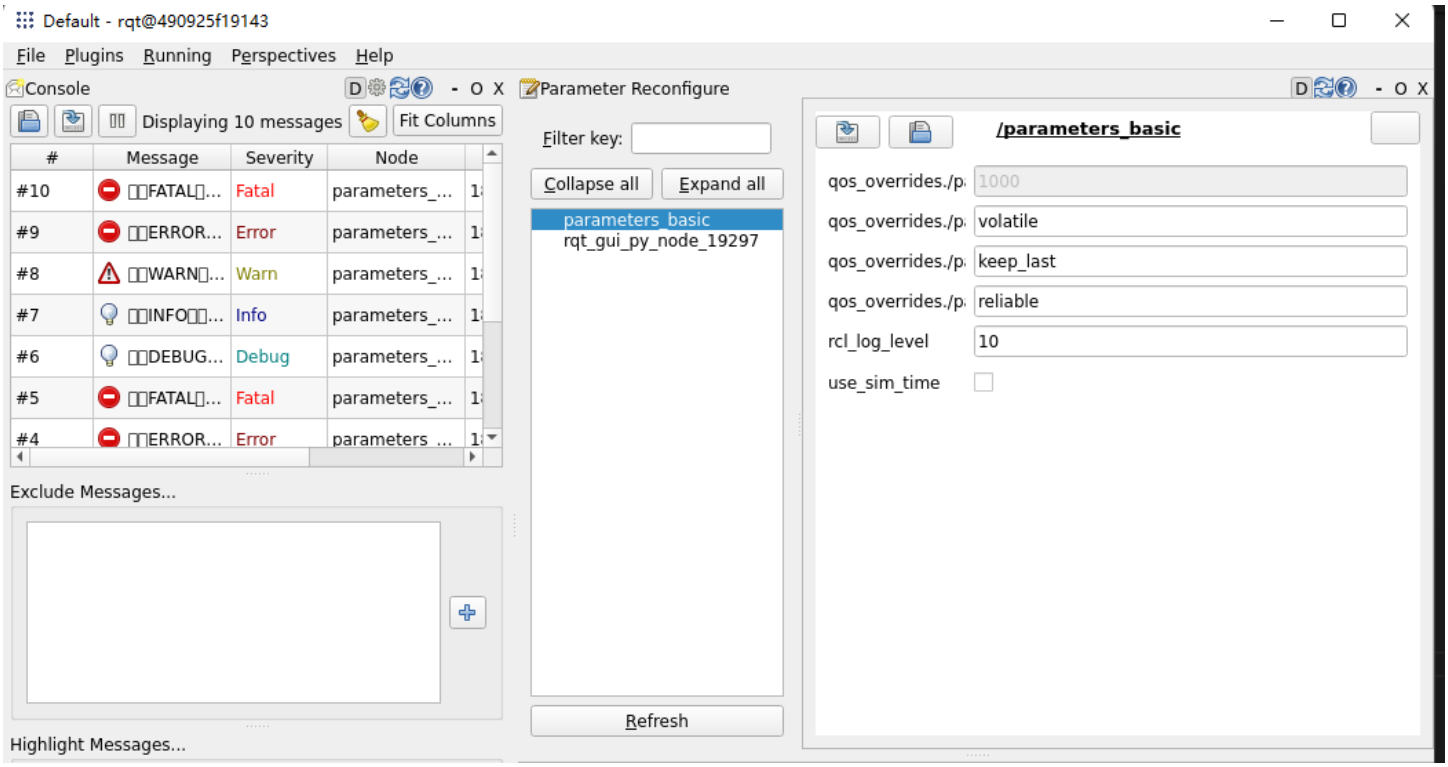
```
root@490925f19143:~/d2lros2/d2lros2/chapt4/chapt4_ws# ros2 param list
/parameters_basic:
  qos_overrides./parameter_events.publisher.depth
  qos_overrides./parameter_events.publisher.durability
  qos_overrides./parameter_events.publisher.history
  qos_overrides./parameter_events.publisher.reliability
  rcl_log_level
  use_sim_time
root@490925f19143:~/d2lros2/d2lros2/chapt4/chapt4_ws# ros2 param set /parameters_basic rcl_log_level 10
Set parameter successful
```

5.总结

上面我们通过参数实现了动态控制节点日志级别的功能，其实像这样的功能ROS2早已为我们准备好了，在运行任意节点时候可以通过CLI传递日志级别配置。

```shell
ros2 run package-name node-name --ros-args --log-level debug
```

除了命令行设置参数和查看日志，通过rqt也可以可视化设置和查看


Default - rqt@490925f19143
File Plugins Running Perspectives Help
Console
Displaying 10 messages
Fit Columns
Message Severity Node
#10 □□FATAL□... Fatal parameters_...
#9 □□ERROR... Error parameters_...
#8 □□WARN□... Warn parameters_...
#7 □□INFO□□... Info parameters_...
#6 □□DEBUG... Debug parameters_...
#5 □□FATAL□... Fatal parameters_...
#4 □□ERROR... Error parameters_...
Exclude Messages...
Highlight Messages...
Parameter Reconfigure
Filter key:
Collapse all
Expand all
parameters_basic
rqt_gui_py_node_19297
Refresh
/parameters_basic
qos_overrides./p. 1000
qos_overrides./p. volatile
qos_overrides./p. keep_last
qos_overrides./p. reliable
rcl_log_level 10
use_sim_time

3.参数之RCLPY实现

上节利用RCLCPP的API实现了参数相关的基础操作（声明、获取），本节我们采用RCLPY的API来尝试实现相同的功能。

1.创建功能包和节点

```shell
cd chapt4/chapt4_ws/
ros2 pkg create example_parameters_rclpy --build-type ament_python --dependencies
rclpy --destination-directory src --node-name parameters_basic --maintainer-name
"fishros" --maintainer-email "fishros@foxmail.com"
```

parameters_basic.py

```python
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node

class ParametersBasicNode(Node):
    """
    创建一个ParametersBasicNode节点，并在初始化时输出一个话
    """
    def __init__(self,name):
        super().__init__(name)
        self.get_logger().info(f"节点已启动：{name}!")

def main(args=None):
    rclpy.init(args=args) # 初始化rclpy
    node = ParametersBasicNode("parameters_basic")  # 新建一个节点
    rclpy.spin(node) # 保持节点运行，检测是否收到退出指令（Ctrl+C）
    rclpy.shutdown() # 关闭rclpy
```

2.RCLPY参数API

- https://docs.ros2.org/latest/api/rclpy/api/node.html

https://docs.ros2.org/latest/api/rclpy/api/node.html

Return type: Timer

declare_parameter(*name*, *value=None*, *descriptor=rcl_interfaces.msg.ParameterDescriptor(name='', type=0, description='', additional_constraints='', read_only=False, floating_point_range=[], integer_range=[])*, *ignore_override=False*)

Declare and initialize a parameter.

This method, if successful, will result in any callback registered with `add_on_set_parameters_callback()` to be called.

Parameters:
- **name** (`str`) – Fully-qualified name of the parameter, including its namespace.
- **value** (`Optional[Any]`) – Value of the parameter to declare.
- **descriptor** (`ParameterDescriptor`) – Descriptor for the parameter to declare.
- **ignore_override** (`bool`) – True if overrides shall not be taken into account; False otherwise.

Return type: `Parameter`

Returns: Parameter with the effectively assigned value.

Raises: ParameterAlreadyDeclaredException if the parameter had already been declared.

Raises: InvalidParameterException if the parameter name is invalid.

Raises: InvalidParameterValueException if the registered callback rejects the parameter.

3.使用参数控制节点日志级别

```python
class ParametersBasicNode(Node):
    """
    创建一个ParametersBasicNode节点，并在初始化时输出一个话
    """

    def __init__(self, name):
        super().__init__(name)
        self.get_logger().info(f"节点已启动：{name}!")
        # 声明参数
        self.declare_parameter('rcl_log_level', 0)
        # 获取参数
        log_level = self.get_parameter("rcl_log_level").value
        # 设置参数
        self.get_logger().set_level(log_level)
        # 定时修改
        self.timer = self.create_timer(0.5, self.timer_callback)
```

```
    def timer_callback(self):
        """定时器回调函数"""
        # 获取参数
        log_level = self.get_parameter("rcl_log_level").value
        # 设置参数
        self.get_logger().set_level(log_level)
        print(
            f"========================{log_level}============================")
        self.get_logger().debug("我是DEBUG级别的日志，我被打印出来了!")
        self.get_logger().info("我是INFO级别的日志，我被打印出来了!")
        self.get_logger().warn("我是WARN级别的日志，我被打印出来了!")
        self.get_logger().error("我是ERROR级别的日志，我被打印出来了!")
        self.get_logger().fatal("我是FATAL级别的日志，我被打印出来了!")
```

4.编译测试

```shell
colcon build  --packages-select example_parameters_rclpy
source install/setup.bash
ros2 run example_parameters_rclpy parameters_basic
```

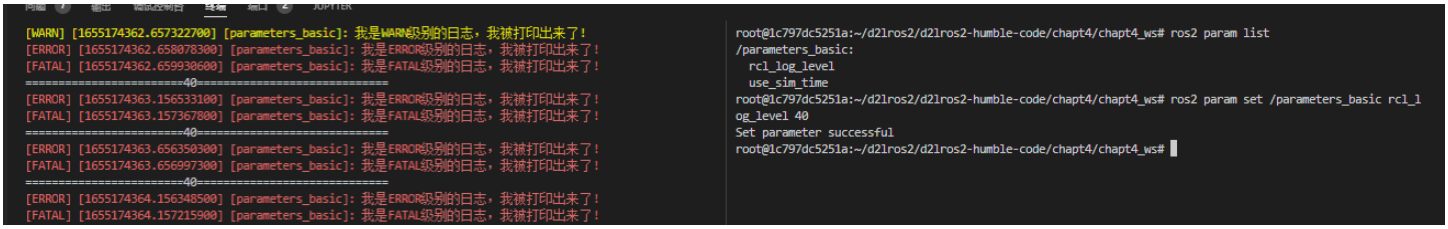
```
root@1c797dc5251a:~/d2lros2/d2lros2-humble-code/chapt4/chapt4_ws# ros2 run example_parameters_rclpy parameters_basic
[INFO] [1655174156.735414500] [parameters_basic]: 节点已启动: parameters_basic!
======================0============================
[INFO] [1655174157.238748200] [parameters_basic]: 我是INFO级别的日志，我被打印出来了!
[WARN] [1655174157.243187700] [parameters_basic]: 我是WARN级别的日志，我被打印出来了!
[ERROR] [1655174157.244416500] [parameters_basic]: 我是ERROR级别的日志，我被打印出来了!
[FATAL] [1655174157.245589100] [parameters_basic]: 我是FATAL级别的日志，我被打印出来了!
======================0============================
[INFO] [1655174157.738268300] [parameters_basic]: 我是INFO级别的日志，我被打印出来了!
[WARN] [1655174157.739100100] [parameters_basic]: 我是WARN级别的日志，我被打印出来了!
[ERROR] [1655174157.740106200] [parameters_basic]: 我是ERROR级别的日志，我被打印出来了!
[FATAL] [1655174157.740759200] [parameters_basic]: 我是FATAL级别的日志，我被打印出来了!
======================0============================
[INFO] [1655174158.239195300] [parameters_basic]: 我是INFO级别的日志，我被打印出来了!
[WARN] [1655174158.241196100] [parameters_basic]: 我是WARN级别的日志，我被打印出来了!
[ERROR] [1655174158.242354800] [parameters_basic]: 我是ERROR级别的日志，我被打印出来了!
[FATAL] [1655174158.244164800] [parameters_basic]: 我是FATAL级别的日志，我被打印出来了!
```

指定参数值测试

```shell
ros2 run example_parameters_rclpy parameters_basic --ros-args -p rcl_log_level:=10
```

动态设置参数测试

```shell
ros2 param list
ros2 param set /parameters_basic rcl_log_level 40
```



5.总结

虽然实现了动态的设置参数，但还有些不够完美，参数的值被外界改变，我们的节点并没有收到任何通知，本节实现的动态改变是通过一个定时器不断的轮询获取参数值实现的，那有没有什么办法可以接收到参数改变的通知呢？答案在进阶篇。

4.动作（Action）通信与自定义接口

你好，我是小鱼。通过前面章节的学习，你已经掌握了ROS2中四大通信利器中话题、服务、参数这三个，还差最后一个就能将ROS2的通信机制全部打包带回家了，这节课小鱼就带你一起认识一下Action，并带你动手体验一下Action通信。

1.Action背景

前面章节学习了话题、服务、参数。

话题适用于节点间单向的频繁的数据传输，服务则适用于节点间双向的数据传递，而参数则用于动态调整节点的设置，动作Action和他们三个有什么不同之处呢？

如果这些问题体现在机器人上，可能是这样子的。我们通过服务服务发送一个目标点给机器人，让机器人移动到该点：

- 你不知道机器人有没有处理移动到目标点的请求（不能确认服务端接收并处理目标）
- 假设机器人收到了请求，你不知道机器人此时的位置和距离目标点的距离（没有反馈）
- 假设机器人移动一半，你想让机器人停下来，也没有办法通知机器人

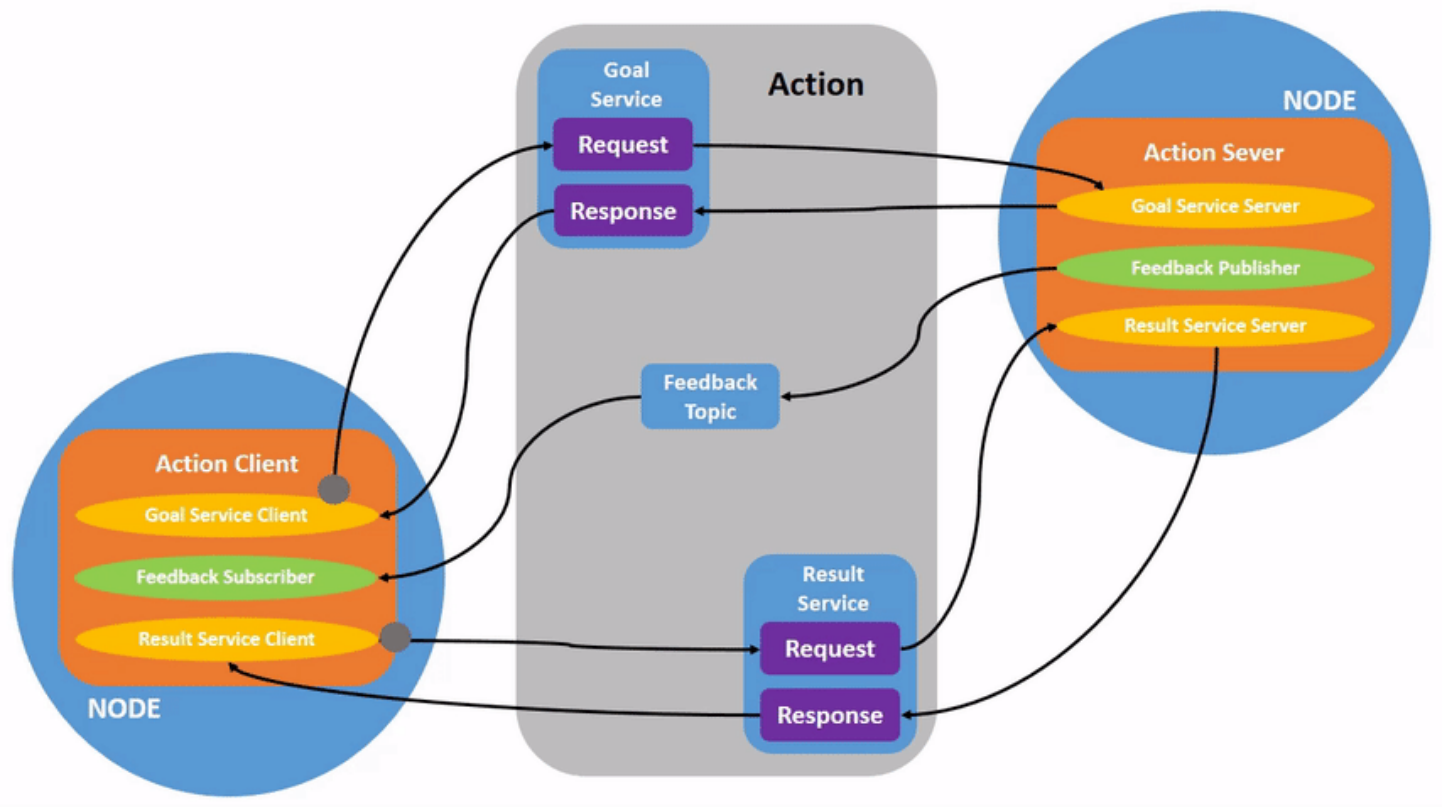
上面的场景在机器人控制当中经常出现，比如控制导航程序，控制机械臂运动，控制小乌龟旋转等，很显然单个话题和服务不能满足我们的使用，因此ROS2针对控制这一场景，基于原有的话题和服务，设计了动作（Action）这一通信方式来解决这一问题。

2.Action的组成部分

知道了Action的出现原因，接着说说Action的三大组成部分目标、反馈和结果。

- 目标：即Action客户端告诉服务端要做什么，服务端针对该目标要有响应。解决了不能确认服务端接收并处理目标问题
- 反馈：即Action服务端告诉客户端此时做的进度如何（类似与工作汇报）。解决执行过程中没有反馈问题
- 结果：即Action服务端最终告诉客户端其执行结果，结果最后返回，用于表示任务最终执行情况。

> **参数是由服务构建出来了，而Action是由话题和服务共同构建出来的（一个Action = 三个服务+两个话题）三个服务分别是：1.目标传递服务 2.结果传递服务 3.取消执行服务 两个话题：1.反馈话题（服务发布，客户端订阅）2.状态话题（服务端发布，客户端订阅）**



3.感受Action

带着前面对Action的了解，接着我们一起来了直观的通过小乌龟的案例来感受一下Action的魅力。

3.1 启动乌龟模拟器和键盘控制节点

乌龟模拟器

```
ros2 run turtlesim turtlesim_node
```

键盘控制节点

```
ros2 run turtlesim turtle_teleop_key
```

打开键盘控制节点后，你应该窗口中可以看到下面的提示

```
Use arrow keys to move the turtle.
Use G|B|V|C|D|E|R|T keys to rotate to absolute orientations. 'F' to cancel a
rotation.
```

有请翻译官小鱼（其实用Deppl翻译的）

```
使用方向键移动乌龟。
用G、B、V、C、D、E、R、T键旋转到绝对方向。'F'可以取消旋转。
```

这段提示什么意思呢？其实就是字面的意思，

小乌龟键盘控制节点，提供两种可选的控制方式。

- 方向键，通过话题(Topic)控制小乌龟的（直接发送移动话题）
- 绝对旋转，则是采用动作(Action）来控制的小乌龟

3.2 使用绝对旋转(Action)控制小乌龟

使用绝对旋转控制小乌龟即使用Action来控制小乌龟。

在小乌龟的遥控窗口我们使用键盘上的F按键周围的按键来尝试运行控制下小乌龟的方向，你会看到小乌龟根据我们所按下按键所在的方向来在原地进行旋转。

同时在旋转的过程中，我们可以使用F按键，来取消小乌龟的运动。

4. Action的CLI工具

Action的命令行工具一共有三个,下面我们一一介绍。

4.1 action list

该命令用于获取目前系统中的action列表。

```
ros2 action list
```

你将看到

```
/turtle1/rotate_absolute
```

如果在list后加入-t参数，即可看到action的类型

```
/turtle1/rotate_absolute [turtlesim/action/RotateAbsolute]
```

知道了接口类型之后，可以使用接口相关CLI指令获取接口的信息

```
ros2 interface show turtlesim/action/RotateAbsolute
```

结果

```
# The desired heading in radians
float32 theta
---
# The angular displacement in radians to the starting position
float32 delta
---
# The remaining rotation in radians
float32 remaining
```

4.2 action info

查看action信息，在终端中输入下面的指令。

```
ros2 action info /turtle1/rotate_absolute
```

你将看到,action客户端和服务段的数量以及名字。

```
Action: /turtle1/rotate_absolute
Action clients: 1
    /teleop_turtle
Action servers: 1
    /turtlesim
```

4.3 action send_goal

该指令用于发送actin请求到服务端，我们可以模拟下，让小乌龟转到我们定义的角度。

我们只需要把goal发给服务端即可，根据goal的定义，我们可以看到goal是由一个浮点类型的theta组成的，我们把theta发给服务端。

发送弧度制1.6给小乌龟

```
ros2 action send_goal /turtle1/rotate_absolute turtlesim/action/RotateAbsolute "
{theta: 1.6}"
```

结果

```
Waiting for an action server to become available...
Sending goal:
     theta: 1.6

Goal accepted with ID: b215ad060899444793229171e76481c7

Result:
    delta: -1.5840003490447998

Goal finished with status: SUCCEEDED
```

我们可以看到goal和result，但是没有看到feedback，这里我们需要加一个参数 --feedback

加上这个参数我们再发送一次。

```
ros2 action send_goal /turtle1/rotate_absolute turtlesim/action/RotateAbsolute "
{theta: 1.5}" --feedback
```

可以看到了，这次的日志中多出了很多实时的反馈，反馈的数值是小乌龟当前的角度与我们给定的目标角度之间的差值，可以看到一直在变小。

```
Waiting for an action server to become available...
Sending goal:
     theta: 1.5

Feedback:
    remaining: -0.0840003490447998

Goal accepted with ID: b368de0ed1a54e00890f1b078f4671c8

Feedback:
    remaining: -0.06800031661987305

Feedback:
    remaining: -0.05200028419494629

Feedback:
    remaining: -0.03600025177001953

Feedback:
    remaining: -0.020000219345092773

Feedback:
    remaining: -0.004000186920166016

Result:
    delta: 0.08000016212463379

Goal finished with status: SUCCEEDED
```

5.自定义通信接口

我们接下来以控制机器人移动到点为例子，来学习Action通信。因为这个场景是我们自己定义的，ROS2并没有拿来就用的接口，所以我们需要自定义Action通信接口。

5.1 创建接口功能包

创建功能包

```shell
cd chapt4_ws/
ros2 pkg create robot_control_interfaces --build-type ament_cmake --destination-directory src --maintainer-name "fishros" --maintainer-email "fishros@foxmail.com"
```

创建接口文件

```shell
mkdir -p src/robot_control_interfaces/action
touch src/robot_control_interfaces/action/MoveRobot.action
```

packages.xml

```xml
  <depend>rosidl_default_generators</depend>
  <member_of_group>rosidl_interface_packages</member_of_group>
```

CMakeLists.txt

```cmake
find_package(ament_cmake REQUIRED)
find_package(rosidl_default_generators REQUIRED)


rosidl_generate_interfaces(${PROJECT_NAME}
  "action/MoveRobot.action"
)
```

5.2 编写接口

```
# Goal: 要移动的距离
float32 distance
---
# Result: 最终的位置
float32 pose
---
# Feedback: 中间反馈的位置和状态
float32 pose
uint32 status
uint32 STATUS_MOVEING = 3
uint32 STATUS_STOP = 4
```

5.3 编译生成接口

```shell
colcon build --packages-select robot_control_interfaces
```

编译成功后，即可看到C++头文件和Python相关文件

- C++ Action消息头文件目录 `install/robot_control_interfaces/include`
- Python Action消息文件目录 `install/robot_control_interfaces/local/lib/python3.10`

6.总结

本节我们学习了Action并手动创建了Action的接口，下一节小鱼将带你一起使用接口完成任务。

5.动作之CPP实现

上一节小鱼介绍了Action通信并一起自定义了一个接口，本节我们尝试使用接口来利用动作通信控制机器人。

1.创建功能包和节点

1.1 创建功能包

创建example_action_rclcpp功能包，添加 `robot_control_interfaces` 、 `rclcpp_action` 、 `rclcpp` 三个依赖，自动创建 `action_robot_01` 节点，并手动创建 `action_control_01.cpp` 节点。

```shell
cd chapt4_ws/
ros2 pkg create example_action_rclcpp --build-type ament_cmake --dependencies
rclcpp rclcpp_action robot_control_interfaces --destination-directory src --node-
name action_robot_01 --maintainer-name "fishros" --maintainer-email
"fishros@foxmail.com"
touch src/example_action_rclcpp/src/action_control_01.cpp
```

接着我们创建Robot类的头文件和CPP文件。

```
touch src/example_action_rclcpp/include/example_action_rclcpp/robot.h
touch src/example_action_rclcpp/src/robot.cpp
```

创建完成后目录结构

```
.
├── CMakeLists.txt
├── include
│   └── example_action_rclcpp
│       └── robot.h
├── package.xml
└── src
    ├── action_control_01.cpp
    ├── action_robot_01.cpp
```

```
    └── robot.cpp

3 directories, 6 files
```

1.2 robot.h

```cpp
/*
copyright
*/
#ifndef EXAMPLE_ACTIONI_RCLCPP_ROBOT_H_
#define EXAMPLE_ACTIONI_RCLCPP_ROBOT_H_
#include "rclcpp/rclcpp.hpp"
#include "robot_control_interfaces/action/move_robot.hpp"

class Robot {
 public:
  Robot() = default;
  ~Robot() = default;
 private:
};

#endif  // EXAMPLE_ACTIONI_RCLCPP_ROBOT_H_
```

1.3 robot.cpp

暂时为空，第二部分编写

```cpp
#include "example_action_rclcpp/robot.h"
```

1.4 action_robot_01.cpp

```cpp
#include "example_action_rclcpp/robot.h"
#include "rclcpp/rclcpp.hpp"
```

```cpp
#include "rclcpp_action/rclcpp_action.hpp"
#include "robot_control_interfaces/action/move_robot.hpp"

// 创建一个ActionServer类
class ActionRobot01 : public rclcpp::Node {
 public:
  explicit ActionRobot01(std::string name) : Node(name) {
    RCLCPP_INFO(this->get_logger(), "节点已启动：%s.", name.c_str());
  }


};

int main(int argc, char** argv) {
  rclcpp::init(argc, argv);
  auto action_server = std::make_shared<ActionRobot01>("action_robot_01");
  rclcpp::spin(action_server);
  rclcpp::shutdown();
  return 0;
}
```

1.5 action_control_01.cpp

```cpp
#include "rclcpp/rclcpp.hpp"
#include "rclcpp_action/rclcpp_action.hpp"
#include "robot_control_interfaces/action/move_robot.hpp"

class ActionControl01 : public rclcpp::Node {
 public:
  explicit ActionControl01(std::string name): Node(name) {
    RCLCPP_INFO(this->get_logger(), "节点已启动：%s.", name.c_str());
  }


};  // class ActionControl01

int main(int argc, char** argv) {
  rclcpp::init(argc, argv);
  auto action_client = std::make_shared<ActionControl01>("action_robot_cpp");
```

```
  rclcpp::spin(action_client);
  rclcpp::shutdown();
  return 0;
}
```

1.6 CMakeList.txt

```cmake
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(robot_control_interfaces REQUIRED)
find_package(example_interfaces REQUIRED)
find_package(rclcpp_action REQUIRED)

# action_robot节点

add_executable(action_robot_01
    src/robot.cpp
    src/action_robot_01.cpp
)
target_include_directories(action_robot_01 PUBLIC
  $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
  $<INSTALL_INTERFACE:include>)
target_compile_features(action_robot_01 PUBLIC c_std_99 cxx_std_17)  # Require C99
and C++17
ament_target_dependencies(
  action_robot_01
  "rclcpp"
  "rclcpp_action"
  "robot_control_interfaces"
  "example_interfaces"
)

install(TARGETS action_robot_01
  DESTINATION lib/${PROJECT_NAME})

# action_control节点

add_executable(action_control_01
  src/action_control_01.cpp
```

```
)
target_include_directories(action_control_01 PUBLIC
$<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
$<INSTALL_INTERFACE:include>)
target_compile_features(action_control_01 PUBLIC c_std_99 cxx_std_17)  # Require
C99 and C++17
ament_target_dependencies(
  action_control_01
  "rclcpp"
  "rclcpp_action"
  "robot_control_interfaces"
  "example_interfaces"
)

install(TARGETS action_control_01
DESTINATION lib/${PROJECT_NAME})
```

2.编写机器人类

机器人类主要负责移动机器人和提供当前机器人的状态，我们设计几个函数来实现该功能。

2.1 robot.h

cpp

```
#ifndef EXAMPLE_ACTIONI_RCLCPP_ROBOT_H_
#define EXAMPLE_ACTIONI_RCLCPP_ROBOT_H_
#include "rclcpp/rclcpp.hpp"
#include "robot_control_interfaces/action/move_robot.hpp"

class Robot {
 public:
  using MoveRobot = robot_control_interfaces::action::MoveRobot;
  Robot() = default;
  ~Robot() = default;
  float move_step(); /*移动一小步，请间隔500ms调用一次*/
  bool set_goal(float distance); /*移动一段距离*/
  float get_current_pose();
  int get_status();
  bool close_goal(); /*是否接近目标*/
```

```
  void stop_move();  /*停止移动*/

 private:
  float current_pose_ = 0.0;              /*声明当前位置*/
  float target_pose_ = 0.0;               /*目标距离*/
  float move_distance_ = 0.0;             /*目标距离*/
  std::atomic<bool> cancel_flag_{false}; /*取消标志*/
  int status_ = MoveRobot::Feedback::STATUS_STOP;
};

#endif  // EXAMPLE_ACTIONI_RCLCPP_ROBOT_H_
```

2.2 robot.cpp

cpp

```
#include "example_action_rclcpp/robot.h"

/*移动一小步，请间隔500ms调用一次*/
float Robot::move_step() {
  int direct = move_distance_ / fabs(move_distance_);
  float step = direct * fabs(target_pose_ - current_pose_) *
               0.1; /* 每一步移动当前到目标距离的1/10*/
  current_pose_ += step;
  std::cout << "移动了：" << step << "当前位置：" << current_pose_ << std::endl;
  return current_pose_;
}

/*移动一段距离*/
bool Robot::set_goal(float distance) {
  move_distance_ = distance;
  target_pose_ += move_distance_;

  /* 当目标距离和当前距离大于0.01同意向目标移动 */
  if (close_goal()) {
    status_ = MoveRobot::Feedback::STATUS_STOP;
    return false;
  }
  status_ = MoveRobot::Feedback::STATUS_MOVEING;
  return true;
}
```

```
float Robot::get_current_pose() { return current_pose_; }
int Robot::get_status() { return status_; }
/*是否接近目标*/
bool Robot::close_goal() { return fabs(target_pose_ - current_pose_) < 0.01; }
void Robot::stop_move() {
  status_ = MoveRobot::Feedback::STATUS_STOP;
} /*停止移动*/
```

3.编写机器人节点

cpp

```
class ActionRobot01 : public rclcpp::Node {
 public:
  using MoveRobot = robot_control_interfaces::action::MoveRobot;
  using GoalHandleMoveRobot = rclcpp_action::ServerGoalHandle<MoveRobot>;

  explicit ActionRobot01(std::string name) : Node(name) {
    RCLCPP_INFO(this->get_logger(), "节点已启动：%s.", name.c_str());

    using namespace std::placeholders;  // NOLINT

    this->action_server_ = rclcpp_action::create_server<MoveRobot>(
        this, "move_robot",
        std::bind(&ActionRobot01::handle_goal, this, _1, _2),
        std::bind(&ActionRobot01::handle_cancel, this, _1),
        std::bind(&ActionRobot01::handle_accepted, this, _1));
  }

 private:
  Robot robot;
  rclcpp_action::Server<MoveRobot>::SharedPtr action_server_;

  rclcpp_action::GoalResponse handle_goal(
      const rclcpp_action::GoalUUID& uuid,
      std::shared_ptr<const MoveRobot::Goal> goal) {
    RCLCPP_INFO(this->get_logger(), "Received goal request with distance %f",
                goal->distance);
    (void)uuid;
```

```
  if (fabs(goal->distance > 100)) {
    RCLCPP_WARN(this->get_logger(), "目标距离太远了，本机器人表示拒绝！");
    return rclcpp_action::GoalResponse::REJECT;
  }
  RCLCPP_INFO(this->get_logger(),
              "目标距离%f我可以走到，本机器人接受，准备出发！",
              goal->distance);
  return rclcpp_action::GoalResponse::ACCEPT_AND_EXECUTE;
}

rclcpp_action::CancelResponse handle_cancel(
    const std::shared_ptr<GoalHandleMoveRobot> goal_handle) {
  RCLCPP_INFO(this->get_logger(), "Received request to cancel goal");
  (void)goal_handle;
  robot.stop_move(); /*认可取消执行，让机器人停下来*/
  return rclcpp_action::CancelResponse::ACCEPT;
}

void execute_move(const std::shared_ptr<GoalHandleMoveRobot> goal_handle) {
  const auto goal = goal_handle->get_goal();
  RCLCPP_INFO(this->get_logger(), "开始执行移动 %f 。。。", goal->distance);

  auto result = std::make_shared<MoveRobot::Result>();
  rclcpp::Rate rate = rclcpp::Rate(2);
  robot.set_goal(goal->distance);
  while (rclcpp::ok() && !robot.close_goal()) {
    robot.move_step();
    auto feedback = std::make_shared<MoveRobot::Feedback>();
    feedback->pose = robot.get_current_pose();
    feedback->status = robot.get_status();
    goal_handle->publish_feedback(feedback);
    /*检测任务是否被取消*/
    if (goal_handle->is_canceling()) {
      result->pose = robot.get_current_pose();
      goal_handle->canceled(result);
      RCLCPP_INFO(this->get_logger(), "Goal Canceled");
      return;
    }
    RCLCPP_INFO(this->get_logger(), "Publish Feedback"); /*Publish feedback*/
    rate.sleep();
  }
```

```
      result->pose = robot.get_current_pose();
      goal_handle->succeed(result);
      RCLCPP_INFO(this->get_logger(), "Goal Succeeded");
    }

    void handle_accepted(const std::shared_ptr<GoalHandleMoveRobot> goal_handle) {
      using std::placeholders::_1;
      std::thread{std::bind(&ActionRobot01::execute_move, this, _1), goal_handle}
          .detach();
    }
  };
```

代码解析

上面的代码信息量有些大，但都是围绕着Action展开的，小鱼带你一步步分解。

首先找到创建Action的API：https://docs.ros2.org/latest/api/rclcpp_action/



• create_server() [2/2]

```
template<typename ActionT , typename NodeT >
Server<ActionT>::SharedPtr rclcpp_action::create_server ( NodeT                                  node,
                                                          const std::string &                    name,
                                                          typename Server< ActionT >::GoalCallback handle_goal,
                                                          typename Server< ActionT >::CancelCallback handle_cancel,
                                                          typename Server< ActionT >::AcceptedCallback handle_accepted,
                                                          const rcl_action_server_options_t &    options = rcl_action_server_get_default_options(),
                                                          rclcpp::CallbackGroup::SharedPtr       group = nullptr
                                                        )
```

Create an action server.

All provided callback functions must be non-blocking.

See also

Server::Server() for more information.

Parameters

[in]	**node**	The action server will be added to this node.
[in]	**name**	The action name.
[in]	**handle_goal**	A callback that decides if a goal should be accepted or rejected.
[in]	**handle_cancel**	A callback that decides if a goal should be attempted to be canceled. The return from this callback only indicates if the server will try to cancel a goal. It does not indicate if the goal was actually canceled.
[in]	**handle_accepted**	A callback that is called to give the user a handle to the goal.
[in]	**options**	options to pass to the underlying rcl_action_server_t.
[in]	**group**	The action server will be added to this callback group. If nullptr, then the action server is added to the default callback group.

Action使用了三个回调函数，分别用于处理收到目标、收到停止、确认接受执行。

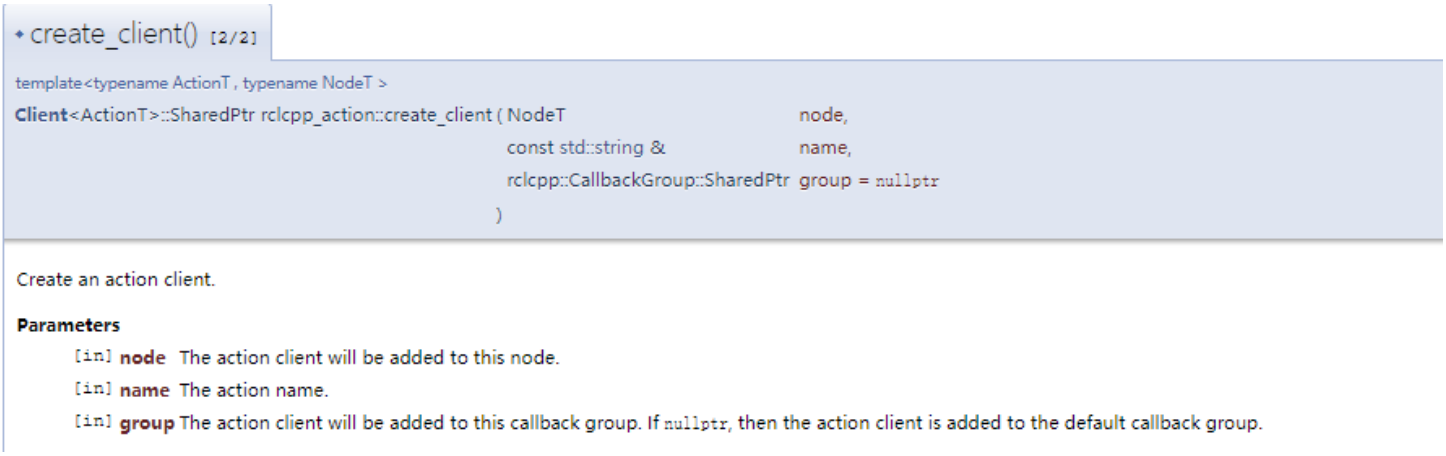
- handle_goal，收到目标，反馈是否可以执行该目标，可以则返回 `ACCEPT_AND_EXECUTE` ,不可以则返回 `REJECT`
- handle_cancel，收到取消运行请求，可以则返回 `ACCEPT` ，不可以返回 `REJECT` 。
- handle_accepted，处理接受请求，当handle_goal中对移动请求 `ACCEPT` 后则进入到这里进行执行，这里我们是单独开了个线程进行执行 `execute_move` 函数，目的是避免阻塞主线程。

执行函数 `execute_move` ，调用机器人，进行一步步的移动，这里我们采用了while循环的形式，不断调用机器人移动并获取机器人的位置，通过feedback进行反馈。同时检测任务是否被取消，如顺利执行完成则反馈最终结果。

代码中我们还用到了Rate函数来精准控制循环的周期，让其保持为2HZ，关于Rate等流程控制的工具，小鱼放到进阶篇来讲解。

4.编写控制节点

先看API：[rclcpp_action: rclcpp_action Namespace Reference (ros2.org)]



create_client() [2/2]

```
template<typename ActionT , typename NodeT >
Client<ActionT>::SharedPtr rclcpp_action::create_client ( NodeT                          node,
                                                          const std::string &            name,
                                                          rclcpp::CallbackGroup::SharedPtr group = nullptr
                                                        )
```

Create an action client.

Parameters

[in] **node** The action client will be added to this node.

[in] **name** The action name.

[in] **group** The action client will be added to this callback group. If `nullptr`, then the action client is added to the default callback group.

接着看代码

```cpp
class ActionControl01 : public rclcpp::Node {
 public:
  using MoveRobot = robot_control_interfaces::action::MoveRobot;
  using GoalHandleMoveRobot = rclcpp_action::ClientGoalHandle<MoveRobot>;

  explicit ActionControl01(
      std::string name,
      const rclcpp::NodeOptions& node_options = rclcpp::NodeOptions())
      : Node(name, node_options) {
    RCLCPP_INFO(this->get_logger(), "节点已启动：%s.", name.c_str());

    this->client_ptr_ =
        rclcpp_action::create_client<MoveRobot>(this, "move_robot");

    this->timer_ =
        this->create_wall_timer(std::chrono::milliseconds(500),
                                std::bind(&ActionControl01::send_goal, this));
```

```
  }

  void send_goal() {
    using namespace std::placeholders;

    this->timer_->cancel();

    if (!this->client_ptr_->wait_for_action_server(std::chrono::seconds(10))) {
      RCLCPP_ERROR(this->get_logger(),
                   "Action server not available after waiting");
      rclcpp::shutdown();
      return;
    }

    auto goal_msg = MoveRobot::Goal();
    goal_msg.distance = 10;

    RCLCPP_INFO(this->get_logger(), "Sending goal");

    auto send_goal_options =
        rclcpp_action::Client<MoveRobot>::SendGoalOptions();
    send_goal_options.goal_response_callback =
        std::bind(&ActionControl01::goal_response_callback, this, _1);
    send_goal_options.feedback_callback =
        std::bind(&ActionControl01::feedback_callback, this, _1, _2);
    send_goal_options.result_callback =
        std::bind(&ActionControl01::result_callback, this, _1);
    this->client_ptr_->async_send_goal(goal_msg, send_goal_options);
  }

private:
  rclcpp_action::Client<MoveRobot>::SharedPtr client_ptr_;
  rclcpp::TimerBase::SharedPtr timer_;

  void goal_response_callback(GoalHandleMoveRobot::SharedPtr goal_handle) {
    if (!goal_handle) {
      RCLCPP_ERROR(this->get_logger(), "Goal was rejected by server");
    } else {
      RCLCPP_INFO(this->get_logger(),
                  "Goal accepted by server, waiting for result");
    }
  }
```

```
    void feedback_callback(
        GoalHandleMoveRobot::SharedPtr,
        const std::shared_ptr<const MoveRobot::Feedback> feedback) {
      RCLCPP_INFO(this->get_logger(), "Feedback current pose:%f", feedback->pose);
    }

    void result_callback(const GoalHandleMoveRobot::WrappedResult& result) {
      switch (result.code) {
        case rclcpp_action::ResultCode::SUCCEEDED:
          break;
        case rclcpp_action::ResultCode::ABORTED:
          RCLCPP_ERROR(this->get_logger(), "Goal was aborted");
          return;
        case rclcpp_action::ResultCode::CANCELED:
          RCLCPP_ERROR(this->get_logger(), "Goal was canceled");
          return;
        default:
          RCLCPP_ERROR(this->get_logger(), "Unknown result code");
          return;
      }

      RCLCPP_INFO(this->get_logger(), "Result received: %f", result.result->pose);
      // rclcpp::shutdown();
    }
  };  // class ActionControl01
```

代码解析

创建客户端简单，发送请求的时候可以指定三个回调函数：

- goal_response_callback，目标的响应回调函数。
- feedback_callback，执行过程中进度反馈接收回调函数。
- result_callback，最终结果接收的回调函数。

这里利用了定时器完成了定时请求的功能，请求一次后就立马使用 `timer_->cancel();` 取消掉了这个定时器，如此就实现了节点启动后定时发一次请求的功能。

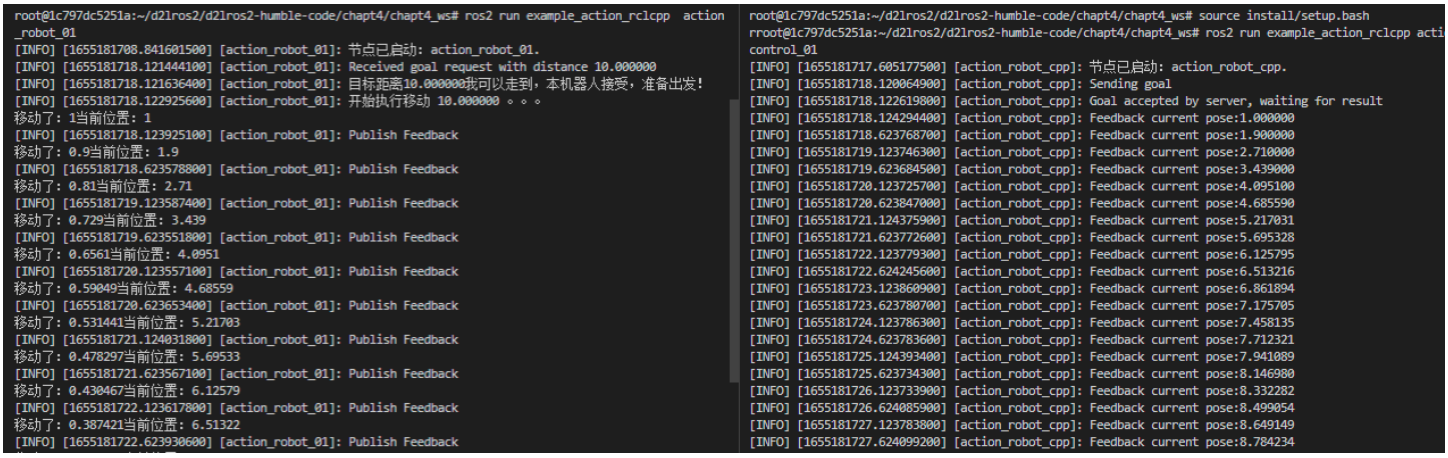
5.编译测试

一个终端，运行机器人节点

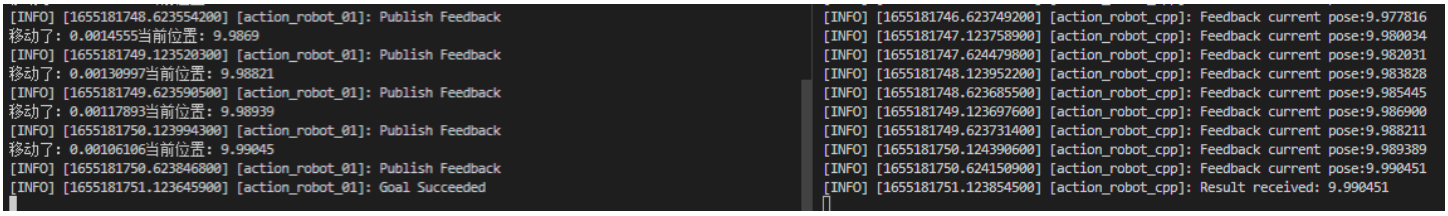
```
cd chapt4_ws/
colcon build --packages-up-to example_action_rclcpp
source install/setup.bash
ros2 run example_action_rclcpp  action_robot_01
```

新终端，运行控制节点

```
source install/setup.bash
ros2 run example_action_rclcpp action_control_01
```



执行完成



6.总结与测试

上面只做了简单的测试，你可以尝试再编写一个定时器，在节点启动的第5s时发送取消执行请求，看看是否可以让机器人停下来。

本节我们利用rclcpp_action的API实现了Action通信的测试，Action在后续的机器人开发中并没那么常用，但是其思想比较重要，应该掌握。

6.动作之RCLPY实现

上节我们再C++中结合RCLCPP和RCLCPPACTION库实现了Action通信，本节我们利用RCLPY在Python中实现相同的功能。

1.创建功能包和节点

1.1 创建功能包

```shell
cd chapt4_ws/
ros2 pkg create example_action_rclpy --build-type ament_python --dependencies
rclpy robot_control_interfaces --destination-directory src --node-name
action_robot_02 --maintainer-name "fishros" --maintainer-email
"fishros@foxmail.com"
# 手动再创建action_control_02节点文件
touch src/example_action_rclpy/example_action_rclpy/action_control_02.py
#手动创建机器人类robot.py
touch src/example_action_rclpy/example_action_rclpy/robot.py
```

1.2 robot.py

```
from robot_control_interfaces.action import MoveRobot
import math

class Robot():
    """机器人类，模拟一个机器人"""

    def __init__(self) -> None:
        pass

    def get_status(self):
        """获取状态"""
        pass

    def get_current_pose(self):
```

```
        """获取当前位置"""
        pass

    def close_goal(self):
        """接近目标"""
        pass

    def stop_move(self):
        """停止移动"""
        pass

    def move_step(self):
        """移动一小步"""
        pass

    def set_goal(self, distance):
        """设置目标"""
        pass
```

1.3 action_robot_02.py

```shell
#!/usr/bin/env python3

import time
# 导入rclpy相关库
import rclpy
from rclpy.node import Node
from rclpy.action import ActionServer
from rclpy.action.server import ServerGoalHandle
# 导入接口
from robot_control_interfaces.action import MoveRobot
# 导入机器人类
from example_action_rclpy.robot import Robot
#from rclpy.executors import MultiThreadedExecutor
#from rclpy.callback_groups import MutuallyExclusiveCallbackGroup

class ActionRobot02(Node):
    """机器人端Action服务"""
```

```
    def __init__(self,name):
        super().__init__(name)
        self.get_logger().info(f"节点已启动：{name}!")

def main(args=None):
    """主函数"""
    rclpy.init(args=args)
    action_robot_02 = ActionRobot02("action_robot_02")
    # 采用多线程执行器解决rate死锁问题
    # executor = MultiThreadedExecutor()
    # executor.add_node(action_robot_02)
    # executor.spin()
    rclpy.spin(action_robot_02)
    rclpy.shutdown()
```

1.4 action_control_02.py

python

```
import rclpy
from rclpy.action import ActionClient
from rclpy.node import Node
# 导入Action接口
from robot_control_interfaces.action import MoveRobot

class ActionControl02(Node):
    """Action客户端"""

    def __init__(self,name):
        super().__init__(name)
        self.get_logger().info(f"节点已启动：{name}!")

def main(args=None):
    """主函数"""
    rclpy.init(args=args)
    action_robot_02 = ActionControl02("action_control_02")
    rclpy.spin(action_robot_02)
    rclpy.shutdown()
```

1.5 setup.py

```python
    entry_points={
        'console_scripts': [
            'action_robot_02 = example_action_rclpy.action_robot_02:main',
            'action_control_02 = example_action_rclpy.action_control_02:main'
        ],
    },
```

接着就可以自行编译测试是否可以将节点运行起来了

2.编写机器人类

```python
class Robot():
    """机器人类，模拟一个机器人"""

    def __init__(self) -> None:
        self.current_pose_ = 0.0
        self.target_pose_ = 0.0
        self.move_distance_ = 0.0
        self.status_ = MoveRobot.Feedback

    def get_status(self):
        """获取状态"""
        return self.status_

    def get_current_pose(self):
        """获取当前位置"""
        return self.current_pose_

    def close_goal(self):
        """接近目标"""
        return math.fabs(self.target_pose_ - self.current_pose_) < 0.01

    def stop_move(self):
        """停止移动"""
        self.status_ = MoveRobot.Feedback.STATUS_STOP
```

```
    def move_step(self):
        """移动一小步"""
        direct = self.move_distance_ / math.fabs(self.move_distance_)
        step = direct * math.fabs(self.target_pose_ - self.current_pose_) * 0.1
        self.current_pose_ += step  # 移动一步
        print(f"移动了：{step}当前位置：{self.current_pose_}")
        return self.current_pose_

    def set_goal(self, distance):
        """设置目标"""
        self.move_distance_ = distance
        self.target_pose_ += distance  # 更新目标位置

        if self.close_goal():
            self.stop_move()
            return False

        self.status_ = MoveRobot.Feedback.STATUS_MOVEING  # 更新状态为移动
        return True
```

代码不复杂，就不进行解析了

3.编写机器人节点

```python
class ActionRobot02(Node):
    """机器人端Action服务"""

    def __init__(self,name):
        super().__init__(name)
        self.get_logger().info(f"节点已启动：{name}!")

        self.robot_ = Robot()

        self.action_server_ = ActionServer(
            self, MoveRobot, 'move_robot', self.execute_callback
            # ,callback_group=MutuallyExclusiveCallbackGroup()
        )
```

```python
    def execute_callback(self, goal_handle: ServerGoalHandle):
        """执行回调函数,若采用默认handle_goal函数则会自动调用"""
        self.get_logger().info('执行移动机器人')
        feedback_msg = MoveRobot.Feedback()
        self.robot_.set_goal(goal_handle.request.distance)

        # rate = self.create_rate(2)
        while rclpy.ok() and not self.robot_.close_goal():
            # move
            self.robot_.move_step()
            # feedback
            feedback_msg.pose = self.robot_.get_current_pose()
            feedback_msg.status = self.robot_.get_status()
            goal_handle.publish_feedback(feedback_msg)
            # cancel check
            if goal_handle.is_cancel_requested:
                result = MoveRobot.Result()
                result.pose = self.robot_.get_current_pose()
                return result
            # rate.sleep() # Rate会造成死锁，单线程执行器时不能使用
            time.sleep(0.5)

        goal_handle.succeed()
        result = MoveRobot.Result()
        result.pose = self.robot_.get_current_pose()
        return result
```

眞是人生苦短，我用Python，这里代码就变得简单了

只指定了一个回调函数 `self.execute_callback` ，原因在于Python这为我们封装好了几个默认的函数。

打开文件 `/opt/ros/humble/local/lib/python3.10/dist-packages/rclpy/action/server.py` ，查看源码如下

```python
class ActionServer(Waitable):
    """ROS Action server."""

    def __init__(
        self,
        node,
```

```
        action_type,
        action_name,
        execute_callback,
        *,
        callback_group=None,
        goal_callback=default_goal_callback,
        handle_accepted_callback=default_handle_accepted_callback,
        cancel_callback=default_cancel_callback,
        goal_service_qos_profile=qos_profile_services_default,
        result_service_qos_profile=qos_profile_services_default,
        cancel_service_qos_profile=qos_profile_services_default,
        feedback_pub_qos_profile=QoSProfile(depth=10),
        status_pub_qos_profile=qos_profile_action_status_default,
        result_timeout=900
    ):
        """
        Create an ActionServer.

        :param node: The ROS node to add the action server to.
        :param action_type: Type of the action.
        :param action_name: Name of the action.
            Used as part of the underlying topic and service names.
        :param execute_callback: Callback function for processing accepted goals.
            This is called if when :class:`ServerGoalHandle.execute()` is called
for
            a goal handle that is being tracked by this action server.
        :param callback_group: Callback group to add the action server to.
            If None, then the node's default callback group is used.
        :param goal_callback: Callback function for handling new goal requests.
        :param handle_accepted_callback: Callback function for handling newly
accepted goals.
            Passes an instance of `ServerGoalHandle` as an argument.
        :param cancel_callback: Callback function for handling cancel requests.
        :param goal_service_qos_profile: QoS profile for the goal service.
        :param result_service_qos_profile: QoS profile for the result service.
        :param cancel_service_qos_profile: QoS profile for the cancel service.
        :param feedback_pub_qos_profile: QoS profile for the feedback publisher.
        :param status_pub_qos_profile: QoS profile for the status publisher.
        :param result_timeout: How long in seconds a result is kept by the server
after a goal
```

```
            reaches a terminal state.
        """
```

刚刚说的的几个默认函数

```python
def default_handle_accepted_callback(goal_handle):
    """Execute the goal."""
    goal_handle.execute()


def default_goal_callback(goal_request):
    """Accept all goals."""
    return GoalResponse.ACCEPT


def default_cancel_callback(cancel_request):
    """No cancellations."""
    return CancelResponse.REJECT
```

4.编写控制节点

```python
class ActionControl02(Node):
    """Action客户端"""

    def __init__(self, name):
        super().__init__(name)
        self.get_logger().info(f"节点已启动：{name}!")
        self.action_client_ = ActionClient(self, MoveRobot, 'move_robot')
        self.send_goal_timer_ = self.create_timer(1, self.send_goal)

    def send_goal(self):
        """发送目标"""
        self.send_goal_timer_.cancel()
        goal_msg = MoveRobot.Goal()
        goal_msg.distance = 5.0
        self.action_client_.wait_for_server()
```

```
        self._send_goal_future = self.action_client_.send_goal_async(goal_msg,

feedback_callback=self.feedback_callback)
        self._send_goal_future.add_done_callback(self.goal_response_callback)

    def goal_response_callback(self, future):
        """收到目标处理结果"""
        goal_handle = future.result()
        if not goal_handle.accepted:
            self.get_logger().info('Goal rejected :(')
            return
        self.get_logger().info('Goal accepted :)')
        self._get_result_future = goal_handle.get_result_async()
        self._get_result_future.add_done_callback(self.get_result_callback)

    def get_result_callback(self, future):
        """获取结果反馈"""
        result = future.result().result
        self.get_logger().info(f'Result: {result.pose}')

    def feedback_callback(self, feedback_msg):
        """获取回调反馈"""
        feedback = feedback_msg.feedback
        self.get_logger().info(f'Received feedback: {feedback.pose}')
```

控制节点依然采用三个回调函数实现数据的接收

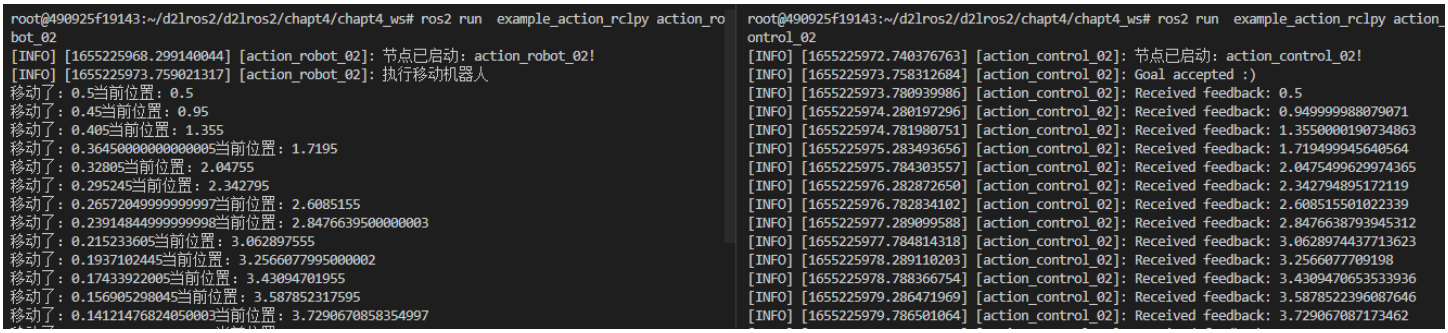
- goal_response_callback，收到目标处理结果。
- get_result_callback，获取结果反馈。
- feedback_callback，接收过程信息。

5.构建测试

接着我们可以编译进行测试。

```
cd chapt4_ws/
colcon build --packages-up-to example_action_rclpy
# 运行机器人节点
source install/setup.bash
```

```
ros2 run example_action_rclpy  action_robot_02
# 新终端
source install/setup.bash
ros2 run example_action_rclpy  action_control_02
```



6.总结

本节我们学习了使用Python编写Action，在设计上Python显得比C++更好用些，但背后的逻辑都是一样的，下一节我们将对ROS2节点通信进行总结。

7.通信机制对比总结

恭喜恭喜恭喜你！完成了ROS2所有通信机制的学习，基于ROS2的通信机制，编写程序将机器人各个组件互相连接，完成一个属于自己的机器人指日可待！

1.话题

话题是单向的，而且不需要等待服务端上线，直接发就行，数据的实时性比较高。

频率高，实时性强的传感器数据的传递一般使用话题实现。

2.服务

服务是双向的，客户端发送请求后，服务端有响应，可以得知服务端的处理结果。

频率较低，强调服务特性和反馈的场景一般使用服务实现。

3.参数

参数是节点的设置，用于配置节点，原理基于服务。

4.动作

动作适用于需要实时反馈的场景，原理基于话题和服务。

5.总结

本节课我们稍微总结了下ROS2的通信机制，下一节课我们将对ROS2的中常用的工具进行介绍，让我们保持好奇心，继续往下学习吧～

1.ROS参数通信原理介绍

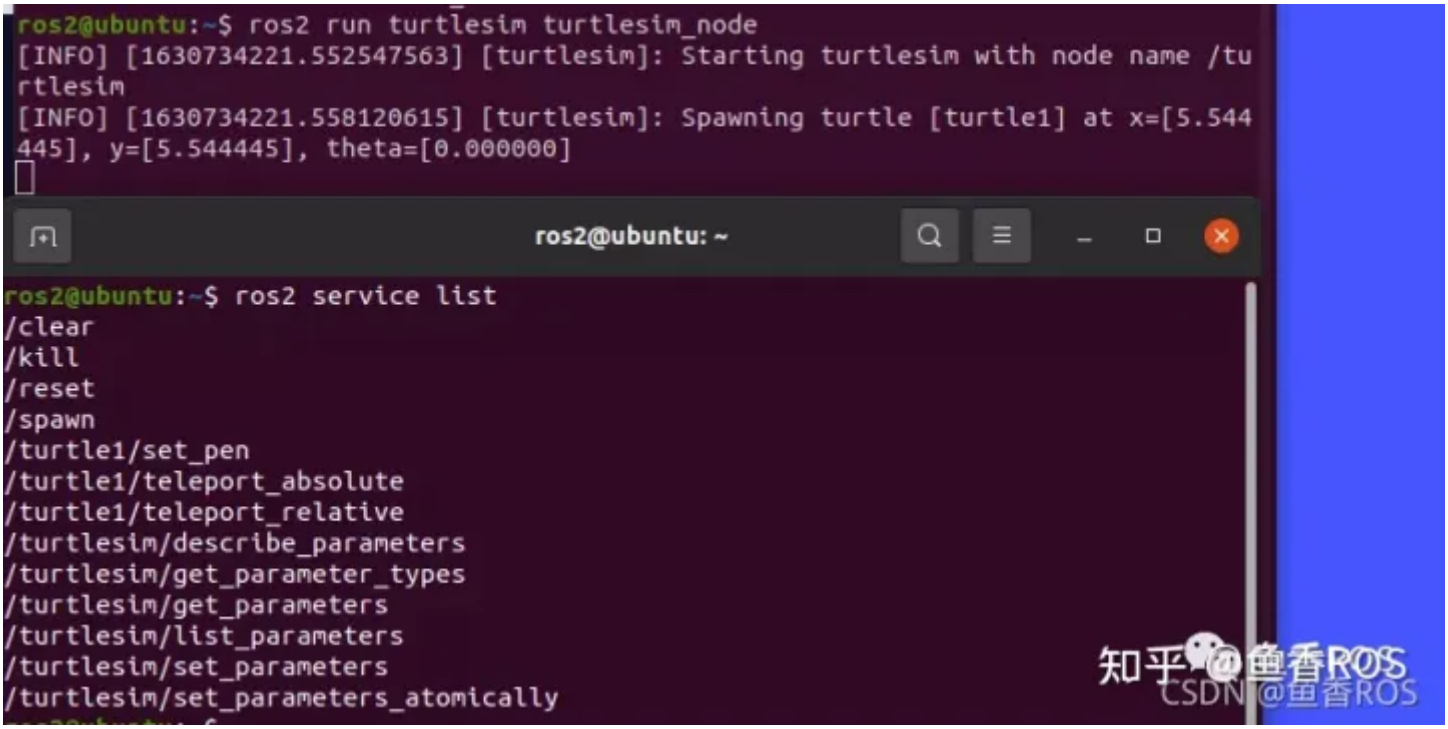
ROS2的参数其实是用服务实现的，小鱼是怎么知道的呢？

随意运行一个节点，你使用下面的指令，就可以看到多出来很多的参数相关的服务。

```
ros2 service list
```

比如启动乌龟模拟器

```
ros2 run turtlesim turtlesim_node
```



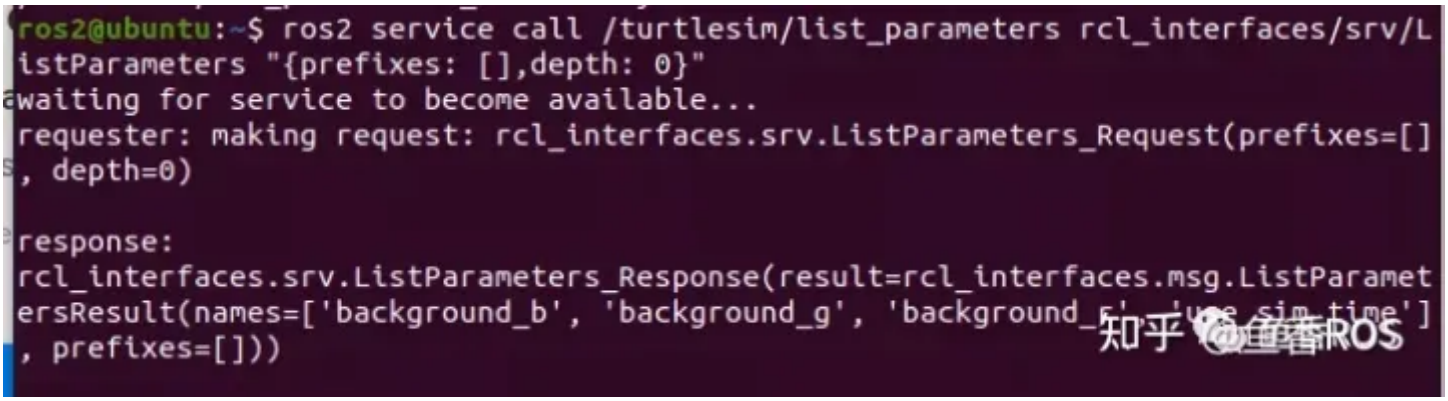
多出来的这些服务就是用于操作这个节点的参数的

```shell
/turtlesim/describe_parameters
/turtlesim/get_parameter_types
/turtlesim/get_parameters
/turtlesim/list_parameters
/turtlesim/set_parameters
/turtlesim/set_parameters_atomically
```

我们如何使用服务查看参数呢？

手动调一下服务就行了~

```shell
ros2 service call /turtlesim/list_parameters rcl_interfaces/srv/ListParameters "
{prefixes: [],depth: 0}"
```



这里可以看到结果里的四个参数

```
names=['background_b', 'background_g', 'background_r','use_sim_time']
```

采用 `ros2 param list` 再看看对不对

```
ros2 param list
```

是不是长的一样

总结

通过上面的实验告诉我们ROS2的参数操作其实就是通过服务通信方式实现的，获取参数列表，set和get操作就是操作相应的服务

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**

- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

3. 生命周期节点

以前在ROS1中，节点的启动顺序无法被控制，这对整个机器人系统来说是一件非常危险和不可控的事，比如说机器人传感器还未启动就开始进行数据的读取了。

在ROS2中提出了生命周期节点的概念，通过生命周期来控制和检测节点状态。

本节小鱼就带你一起学习使用下ROS2的生命周期节点。

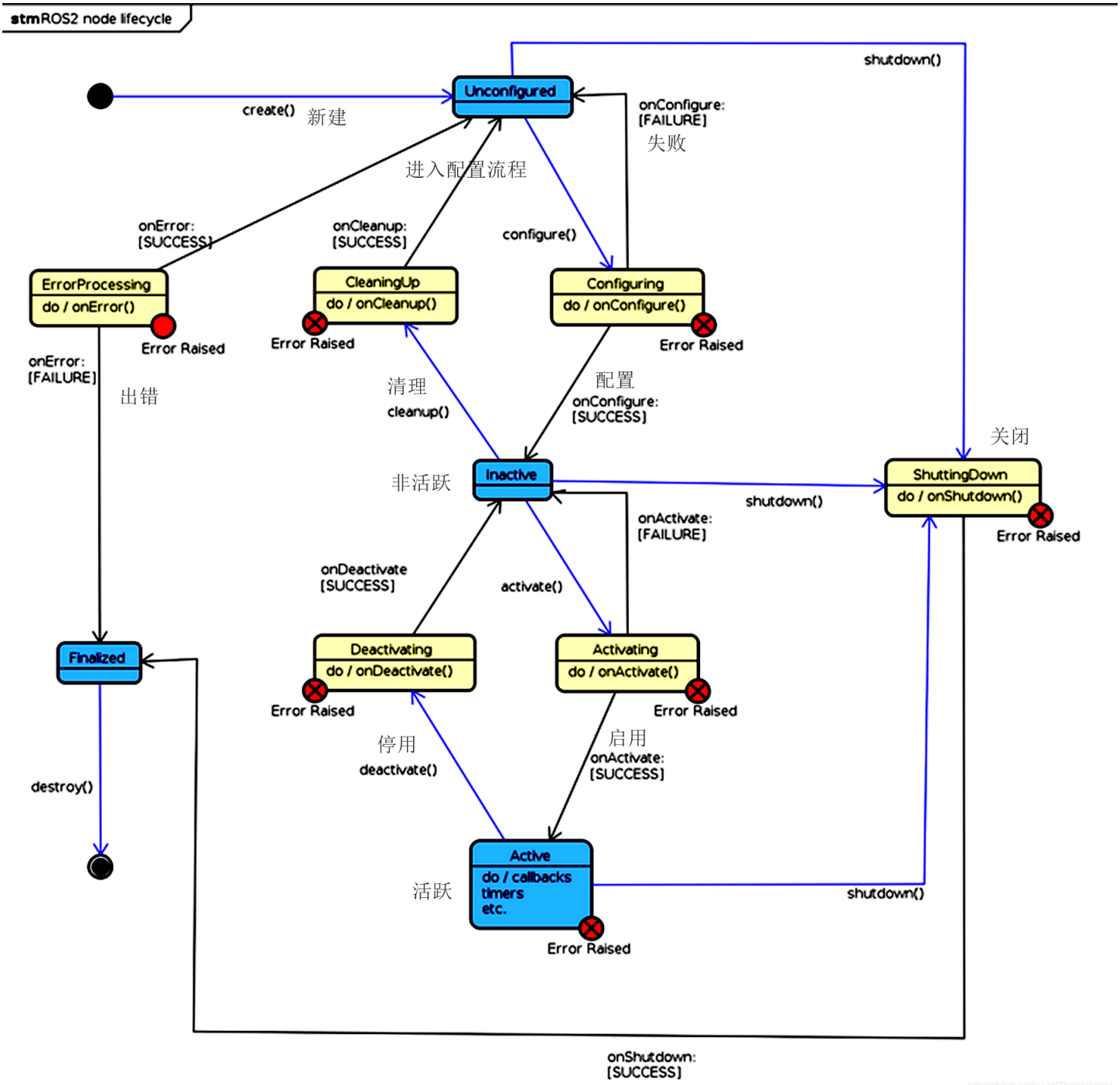
1.生命周期节点介绍

ROS2生命周期节点是利用状态机构成的，状态直接的转换依靠ROS2的通信机制完成。

生命周期节点主要有以下几个状态

- 未配置状态（Unconfigured），节点开始时的第一个状态，并在出现错误后结束。没有执行，其主要目的是错误恢复。
- 非活跃状态（Inactivate），节点持有资源（发布者、监听者等）和配置（参数、内部变量），但什么也不做。 没有执行，没有传输，传入的数据可以保存在缓冲区中，但不能读取， 主要目的是允许重新配置。
- 活跃状态（Activate）， 正常执行。
- 已完成状态（Finalized），节点已被销毁。

具体的状态之间转换关系请参考下图。



2.C++生命周期实例

3.Python生命周期实例

4.总结

参考文章

- ROS2极简总结-新增概念_zhangrelay的博客-CSDN博客_qos ros

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

章节导读

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

1.启动管理工具-Launch

大家好，我是小鱼。今天我们来讲一讲机器人启动系统之Launch。

1.Launch启动工具介绍

1.1 问题描述

对于一个机器人系统来说，往往由很多个不同功能的节点组成，启动一个机器人系统时往往需要启动多个节点，同时根据应用场景和机器人的不同，每个节点还会有不同的配置项。

如果每个节点我们都开一个新终端，敲`ros2 run`指令并写一堆参数，这是多么浪费生命且令人绝望的事情。

除了启动，你会发现，一个个关闭也是很难受的。

1.2 解决方案

可不可以编写一个类似于脚本的文件来管理节点的启动呢？

ROS2设计时就为我们想好了，为我们设计了一套完整的语法和规则的文件来帮助我们组织节点的启动，这个武器就叫launch文件。

launch文件允许我们同时启动和配置多个包含 ROS 2 节点的可执行文件

> **在ROS1中launch文件只有一种格式以.launch结尾的xml文档，不熟悉的同学写起来被xml语法折磨的死去活来。不过在ROS2中不要担心，因为在ROS2你可以使用Python代码来编写launch文件**

2.编写第一个ROS2的launch文件

2.1 三种编写launch文件的方法

ROS2的launch文件有三种格式，python、xml、yaml。其中ROS2官方推荐的时python方式编写launch文件。 原因在于，相较于XML和YAML，**Python是一个编程语言，更加的灵活，我们可以利用Python的很多库来做一些其他工作**（比如创建一些初始化的目录等）。

> 除了灵活还有另外一个原因是ros2/launch（一般launch共功能）和ros2/launch_ros（ROS 2 launch的特性）是用 Python 编写的，我们使用python编写launch文件可以使用 XML 和 YAML 中不能用的launch功能。 要说使用python版本的launch有什么坏处，那就是写起来比yaml要冗余

2.2 使用Python编写Launch

我们的目标是编写一个launch文件，最后使用launch指令，同时启动服务端和客户端节点。

2.2.1 创建功能包和launch文件

创建文件夹和功能包，接着touch一个launch文件，后缀为 `.py` 。

```
mkdir -p chapt4/chapt4_ws/src
cd chapt4/chapt4_ws/src
ros2 pkg create robot_startup --build-type ament_cmake --destination-directory src
mkdir -p src/robot_startup/launch
touch src/robot_startup/launch/example_action.launch.py
```

2.2.2 启动多个节点的示例

我们需要导入两个库，一个叫做LaunchDescription，用于对launch文件内容进行描述，一个是Node，用于声明节点所在的位置。

> 注意这里要定一个名字叫做 `generate_launch_description` 的函数，ROS2会对该函数名字做识别。

```python
# 导入库
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    """launch内容描述函数，由ros2 launch 扫描调用"""
    action_robot_01 = Node(
        package="example_action_rclcpp",
        executable="action_robot_01"
    )
    action_control_01 = Node(
```

```
        package="example_action_rclcpp",
        executable="action_control_01"
    )
    # 创建LaunchDescription对象launch_description,用于描述launch文件
    launch_description = LaunchDescription(
        [action_robot_01, action_control_01])
    # 返回让ROS2根据launch描述执行节点
    return launch_description
```

2.2.3 将launch文件拷贝到安装目录

如果你编写完成后直接编译你会发现install目录下根本没有你编写的launch文件，后续launch自然也找不到这个launch文件。

因为我们用的是 `ament_cmake` 类型功能包，所以这里要使用cmake命令进行文件的拷贝

```cmake
install(DIRECTORY launch
  DESTINATION share/${PROJECT_NAME})
```

如果是 `ament_python` 功能包版

```python
from setuptools import setup
from glob import glob
import os

setup(
    name=package_name,
    version='0.0.0',
    packages=[package_name],
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
        (os.path.join('share', package_name, 'launch'),
glob('launch/*.launch.py')),
    ],
```

```
    },
)
```

2.2.4 编译测试

使用colcon指令编译我们的程序

```python
colcon build
```

编译完成后，在 `chapt5/chapt5_ws/install/robot_startup/share/robot_startup/launch` 目录下你应该就可以看到被cmake拷贝过去的launch文件了。

接着运行`

```sheel
# source 第四章的工作目录，这样才能找到对应的节点，不信你可以不source试试
source ../../chapt4/chapt4_ws/install/setup.bash
source install/setup.bash
ros2 launch robot_startup example_action.launch.py
# 新终端
ros2 node list #即可看到两个节点
```

```
root@1c797dc5251a:~/d2lros2/d2lros2-humble-code/chapt5/chapt5_ws# colcon build
Starting >>> robot_startup
Finished <<< robot_startup [1.37s]

Summary: 1 package finished [1.65s]
root@1c797dc5251a:~/d2lros2/d2lros2-humble-code/chapt5/chapt5_ws# ros2 launch robot_startup example_action.launch.py
[INFO] [launch]: All log files can be found below /root/.ros/log/2022-06-16-05-53-47-990748-1c797dc5251a-35744
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [action_robot_01-1]: process started with pid [35745]
[INFO] [action_control_01-2]: process started with pid [35747]
[action_robot_01-1] [INFO] [1655358828.206670300] [action_robot_01]: 节点已启动: action_robot_01.
[action_control_01-2] [INFO] [1655358828.208105500] [action_robot_cpp]: 节点已启动: action_robot_cpp.
[action_control_01-2] [INFO] [1655358828.723823300] [action_robot_cpp]: Sending goal
[action_robot_01-1] [INFO] [1655358828.724125900] [action_robot_01]: Received goal request with distance 10.000000
[action_robot_01-1] [INFO] [1655358828.724155600] [action_robot_01]: 目标距离10.000000我可以走到，本机器人接受，准备出发!
[action_control_01-2] [INFO] [1655358828.724381700] [action_robot_cpp]: Goal accepted by server, waiting for result
[action_control_01-2] [INFO] [1655358828.724633400] [action_robot_cpp]: Feedback current pose:1.000000
[action_robot_01-1] [INFO] [1655358828.724415200] [action_robot_01]: 开始执行移动 10.000000 。。。
[action_robot_01-1] [INFO] [1655358828.724589000] [action_robot_01]: Publish Feedback
[action_robot_01-1] [INFO] [1655358829.224691200] [action_robot_01]: Publish Feedback
[action_control_01-2] [INFO] [1655358829.224778400] [action_robot_cpp]: Feedback current pose:1.900000
[action_robot_01-1] [INFO] [1655358829.724685500] [action_robot_01]: Publish Feedback
[action_control_01-2] [INFO] [1655358829.724761600] [action_robot_cpp]: Feedback current pose:2.710000
[action_robot_01-1] [INFO] [1655358830.224663200] [action_robot_01]: Publish Feedback
[action_control_01-2] [INFO] [1655358830.224746100] [action_robot_cpp]: Feedback current pose:3.439000
[action_robot_01-1] [INFO] [1655358830.724684000] [action_robot_01]: Publish Feedback
[action_control_01-2] [INFO] [1655358830.724763300] [action_robot_cpp]: Feedback current pose:4.095100
[action_robot_01-1] [INFO] [1655358831.224768200] [action_robot_01]: Publish Feedback
[action_control_01-2] [INFO] [1655358831.224902900] [action_robot_cpp]: Feedback current pose:4.685590
```

3 添加参数&修改命名空间

接着我们尝试使用launch运行参数节点，并通过launch传递参数，和给节点以不同的命名空间。

新建 `chapt5/chapt5_ws/src/robot_startup/launch/example_param_rclcpp.launch.py` 。

编写内容如下

```python
# 导入库
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    """launch内容描述函数，由ros2 launch 扫描调用"""
    parameters_basic1 = Node(
        package="example_parameters_rclcpp",
        namespace="rclcpp",
        executable="parameters_basic",
        parameters=[{'rcl_log_level': 40}]
    )
    parameters_basic2 = Node(
        package="example_parameters_rclpy",
```

```
            namespace="rclpy",
            executable="parameters_basic",
            parameters=[{'rcl_log_level': 50}]
        )
        # 创建LaunchDescription对象launch_description,用于描述launch文件
        launch_description = LaunchDescription(
            [parameters_basic1, parameters_basic2])
        # 返回让ROS2根据launch描述执行节点
        return launch_description
```

编译运行测试

```shell
# source 第四章的工作目录，这样才能找到对应的节点，不信你可以不source试试
source ../../chapt4/chapt4_ws/install/setup.bash
source install/setup.bash
ros2 launch robot_startup example_param_rclcpp.launch.py
# 新终端
ros2 node list #即可看到两个节点
```

```
root@1c797dc5251a:~/d2lros2/d2lros2-humble-code/chapt5/chapt5_ws# ros2 launch robot_startup example_param_rclcpp.launch.py
[INFO] [launch]: All log files can be found below /root/.ros/log/2022-06-16-06-01-00-181464-1c797dc5251a-39878
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [parameters_basic-1]: process started with pid [39886]
[INFO] [parameters_basic-2]: process started with pid [39888]
[parameters_basic-1] [INFO] [1655359260.392691000] [rclcpp.parameters_basic]: 节点已启动: parameters_basic.
[parameters_basic-2] [INFO] [1655359260.526857300] [rclpy.parameters_basic]: 节点已启动: parameters_basic!
[parameters_basic-1] [ERROR] [1655359260.893055500] [rclcpp.parameters_basic]: 我是ERROR级别的日志，我被打印出来了!
[parameters_basic-1] [FATAL] [1655359260.893099200] [rclcpp.parameters_basic]: 我是FATAL级别的日志，我被打印出来了!
[parameters_basic-2] [FATAL] [1655359261.028687400] [rclpy.parameters_basic]: 我是FATAL级别的日志，我被打印出来了!
[parameters_basic-1] [ERROR] [1655359261.393027300] [rclcpp.parameters_basic]: 我是ERROR级别的日志，我被打印出来了!
[parameters_basic-1] [FATAL] [1655359261.393086500] [rclcpp.parameters_basic]: 我是FATAL级别的日志，我被打印出来了!
[parameters_basic-2] [FATAL] [1655359261.528270300] [rclpy.parameters_basic]: 我是FATAL级别的日志，我被打印出来了!
[parameters_basic-1] [ERROR] [1655359261.893015500] [rclcpp.parameters_basic]: 我是ERROR级别的日志，我被打印出来了!
[parameters_basic-1] [FATAL] [1655359261.893057600] [rclcpp.parameters_basic]: 我是FATAL级别的日志，我被打印出来了!
[parameters_basic-2] [FATAL] [1655359262.028385000] [rclpy.parameters_basic]: 我是FATAL级别的日志，我被打印出来了!
[parameters_basic-1] [ERROR] [1655359262.393015700] [rclcpp.parameters_basic]: 我是ERROR级别的日志，我被打印出来了!
[parameters_basic-1] [FATAL] [1655359262.393058300] [rclcpp.parameters_basic]: 我是FATAL级别的日志，我被打印出来了!
[parameters_basic-2] [FATAL] [1655359262.528365500] [rclpy.parameters_basic]: 我是FATAL级别的日志，我被打印出来了!
[parameters_basic-1] [ERROR] [1655359262.893027100] [rclcpp.parameters_basic]: 我是ERROR级别的日志，我被打印出来了!
[parameters_basic-1] [FATAL] [1655359262.893071600] [rclcpp.parameters_basic]: 我是FATAL级别的日志，我被打印出来了!
[parameters_basic-2] [FATAL] [1655359263.028827300] [rclpy.parameters_basic]: 我是FATAL级别的日志，我被打印出来了!
[parameters_basic-1] [ERROR] [1655359263.393068400] [rclcpp.parameters_basic]: 我是ERROR级别的日志，我被打印出来了!
[parameters_basic-1] [FATAL] [1655359263.393134100] [rclcpp.parameters_basic]: 我是FATAL级别的日志，我被打印出来了!
[parameters_basic-2] [FATAL] [1655359263.530222600] [rclpy.parameters_basic]: 我是FATAL级别的日志，我被打印出来了!
```

4. 总结

今天只是简单讲了下，使用launch文件来同时启动多个节点，对节点的一些参数也可以在launch中配置。

launch还有很多更深入的用法，范子琦同学总结的很不错，这里放个链接：

- http://www.robotsfan.com/posts/7a5950c4.html

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

2.ROS2命令行工具

本节我们来对ROS2的命令行工具进行一个小结。

1.命令小结

打开终端，输入ros2，你将看到下面的内容：

```
usage: ros2 [-h] Call `ros2 <command> -h` for more detailed usage. ...

ros2 is an extensible command-line tool for ROS 2.

optional arguments:
  -h, --help            show this help message and exit

Commands:
  action     Various action related sub-commands
  bag        Various rosbag related sub-commands
  component  Various component related sub-commands
  daemon     Various daemon related sub-commands
  doctor     Check ROS setup and other potential issues
  interface  Show information about ROS interfaces
  launch     Run a launch file
  lifecycle  Various lifecycle related sub-commands
  multicast  Various multicast related sub-commands
  node       Various node related sub-commands
  param      Various param related sub-commands
  pkg        Various package related sub-commands
  run        Run a package specific executable
  security   Various security related sub-commands
  service    Various service related sub-commands
  topic      Various topic related sub-commands
  wtf        Use `wtf` as alias to `doctor`

  Call `ros2 <command> -h` for more detailed usage.
```

每一个Command都是对应着ROS2目前所拥有的工具，其实每一个我们在前面的章节中几乎都使用过，而那些没有使用的到的，几乎都是不常用的，所以大家只需要将前面章节中的CLI掌握即可

这里小鱼只是提示下，当我们忘记了某个命令行工具的时候该怎么办，可以使用对应的指令加上-h，即可获取其使用方法。

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

4. RVIZ2

本节我们来说一说，机器人开发中常用的数据可视化工具RVIZ2。

1.RVIZ2是什么

RVIZ2是ROS2中的一个非常重要且常用的数据可视化工具。

那数据指的是什么数据？可视化又是怎么可视化的？RVIZ2又是如何实现不同数据的可视化的呢？

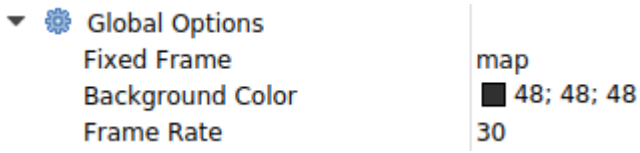
答案如下：

- 数据：各种调试机器人时常用的数据，比如：图像数据、三维点云数据、地图数据、TF数据，机器人模型数据等等
- 可视化：可视化就是让你直观的看到数据，比如说一个三维的点(100,100,100),通过RVIZ可以将其显示在空间中
- 如何做到不同数据的可视化：强大的插件，如果没有你的数据，你可以自己再写一个插件，即插即用，方便快捷

> **注意：RVIZ强调将数据可视化出来，是已有数据的情况下，把数据显示出来而以，而我们后面要讲的gazebo仿真软件是通过模拟真实环境产生数据，两者用途并不一样。**

2.RVIZ2 基础配置

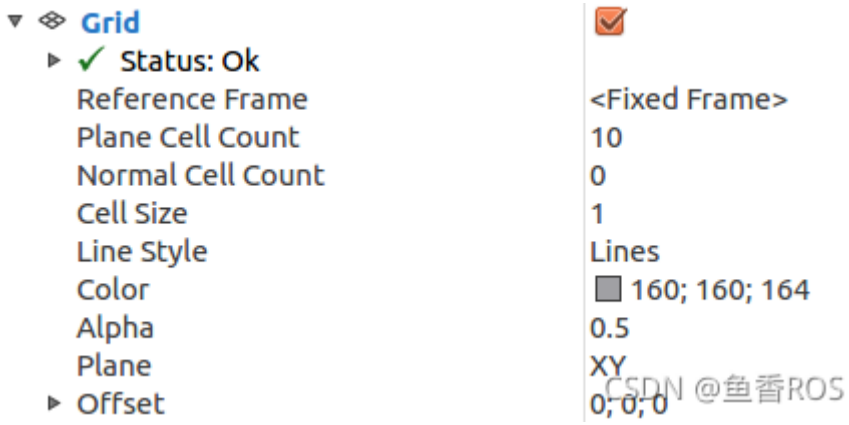
2.1 全局配置



- Fixed Frame：所有帧参考的帧的名称，坐标都是相对的，这个就是告诉RVIZ你是相对谁的，一般是设置成map或者odom
- Frame Rate：用于设置更新 3D 视图的最大频率。

2.2 网格

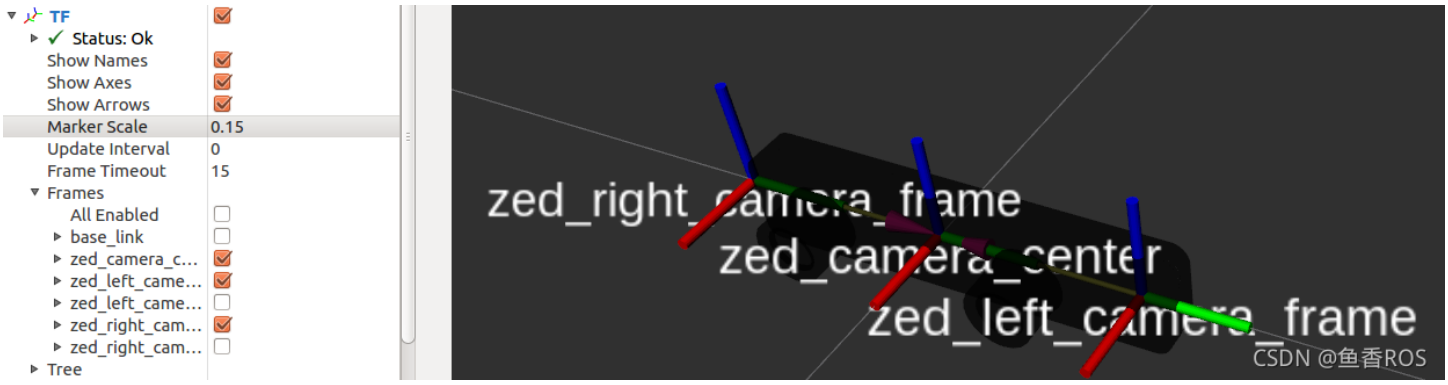
用于可视化通常与地板平面相关联的网格



- Reference frame：帧用作网格坐标参考（通常：）
- Plane cell count: 单元格中网格的大小
- Normal cell count：在沿垂直于叶栅平面的网格数（正常：0）
- Cell size：每个网格单元的尺寸（以米为单位）
- Plane：标识网格平面的两个轴

2.3 机器人模型

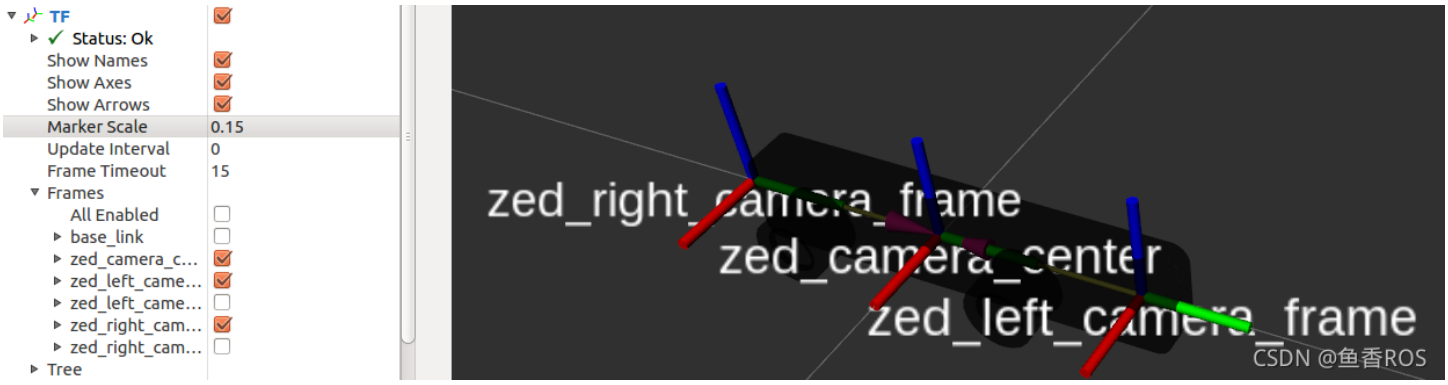
根据 URDF 模型的描述来可视化机器人的模型。



- Visual enabled: 启用/禁用模型的 3D 可视化
- Description Source：机器人模型文件的来源，可以在File和Topic之间进行选择
- Description Topic: 机器人模型文件所在的话题

2.4 TF

可视化构成 TF 广播的所有帧的位置和方向



- Marker Scale: 将字和坐标系标识调整的小一些，使其更加可见且不那么混乱
- Update interval：以秒为单位的TF广播更新时间

最佳实践，勾选你想看的Frames，直观的看到两个坐标之间的关系

3.总结

看完之后是不是还挺不明所以的，因为大多插件都是和坐标相关的，这个要大家学习了下一章节机器人学和URDF建模之后就非常的清晰了

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

3. RQT工具

大家好，我是小鱼，今天来说说ROS2的RQT工具。

前面小鱼介绍过rqt_graph这个工具，我们在平时编写ROS2程序中经常使用，除了rqt_graph,ROS2中还有很多非常易用的RQT工具，一起来体验下

1. RQT是什么

RQT是一个GUI框架，通过插件的方式实现了各种各样的界面工具。

强行解读下：RQT就像插座，任何电器只要符合插座的型号就可以插上去工作。

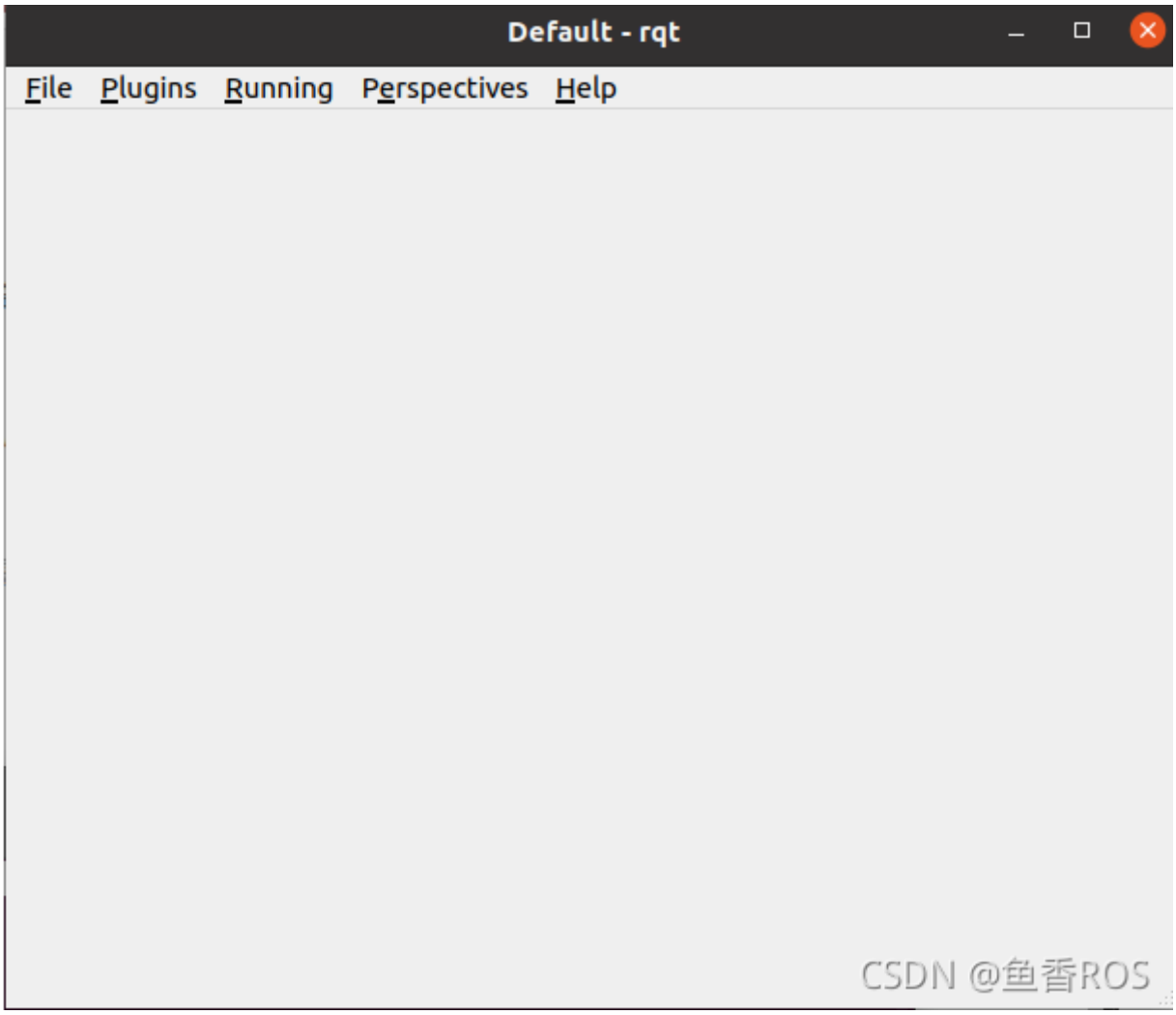
> 说到这里你应该对ROS2的插件化设计感到无比震撼，上节的bag话题录制的存储格式也是插件式的。

2. 体验RQT

没有复杂的指令，一句命令行就可以调出rqt界面。

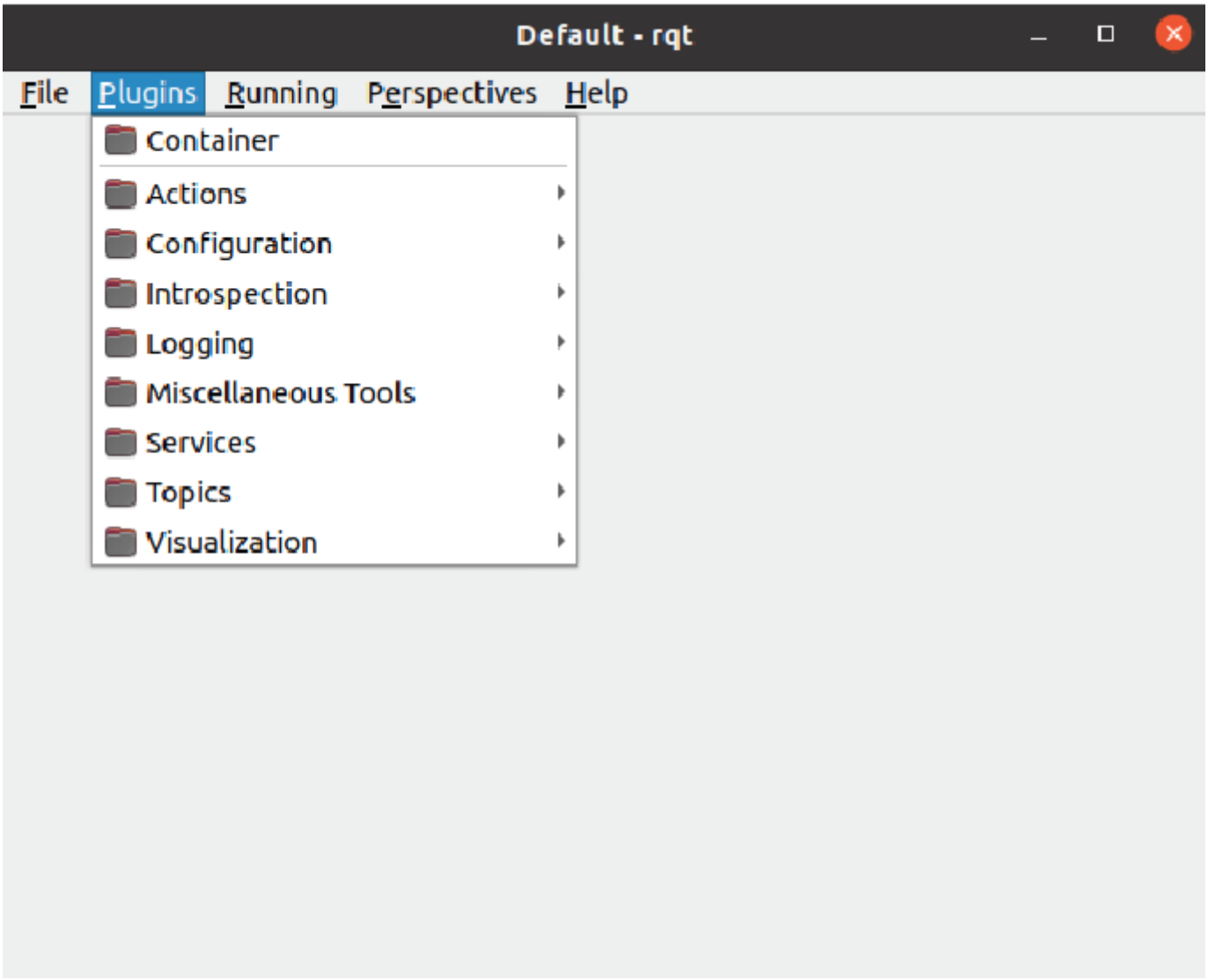
```bash
rqt
```

打开之后的窗口如下图，空空如也，不要担心，因为我们没有选插件的原因。



2.1 选择插件

这里我们可以选择现有的几个RQT插件来试一试，可以看到和话题、参数、服务、动作四大通信组件相关的工具都有，还有一些可视化、日志和系统计算图等相关的。

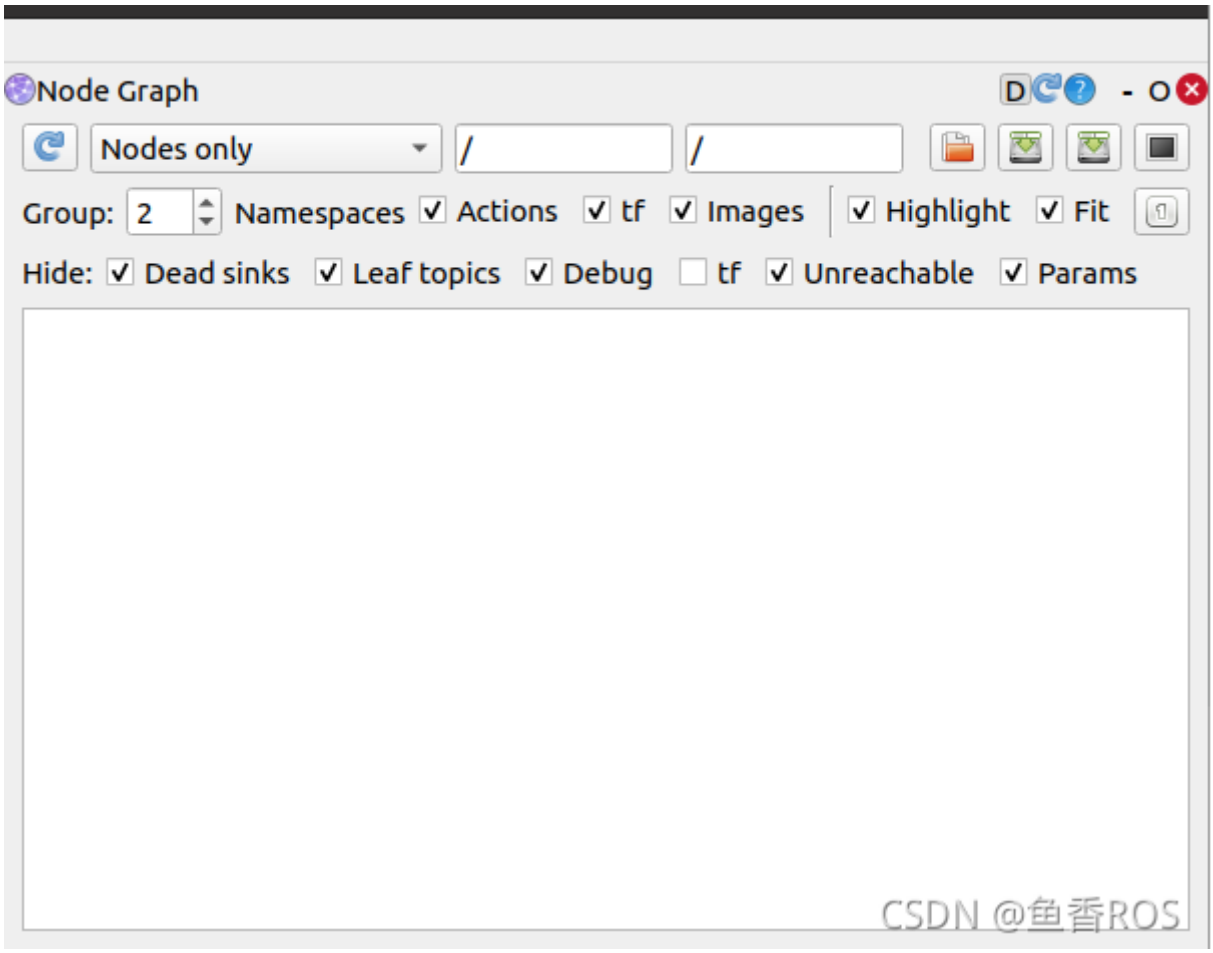


我们按照比较常用的几个来看一下，其他的大家有个印象，后续用到再用。

2.2 插件大观

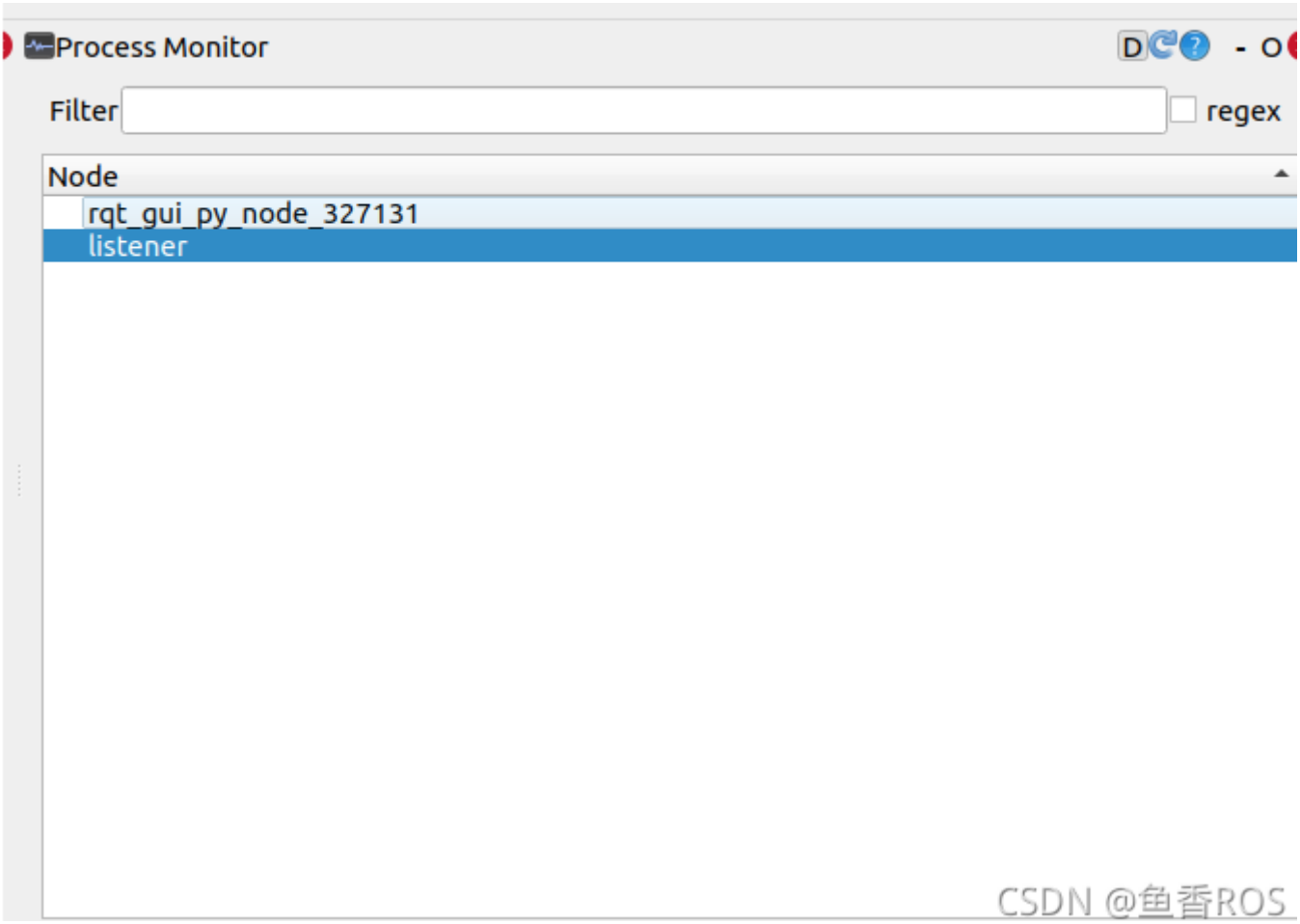
Introspection / Node Graph

第一个是肯定是rqt_graph,插件名字叫做Node Graph,这个名字小鱼觉得更加的贴切，用rqt_graph更多的是为了延续ROS1中的用法，这个**插件用于查看节点和节点之间的关系的**。



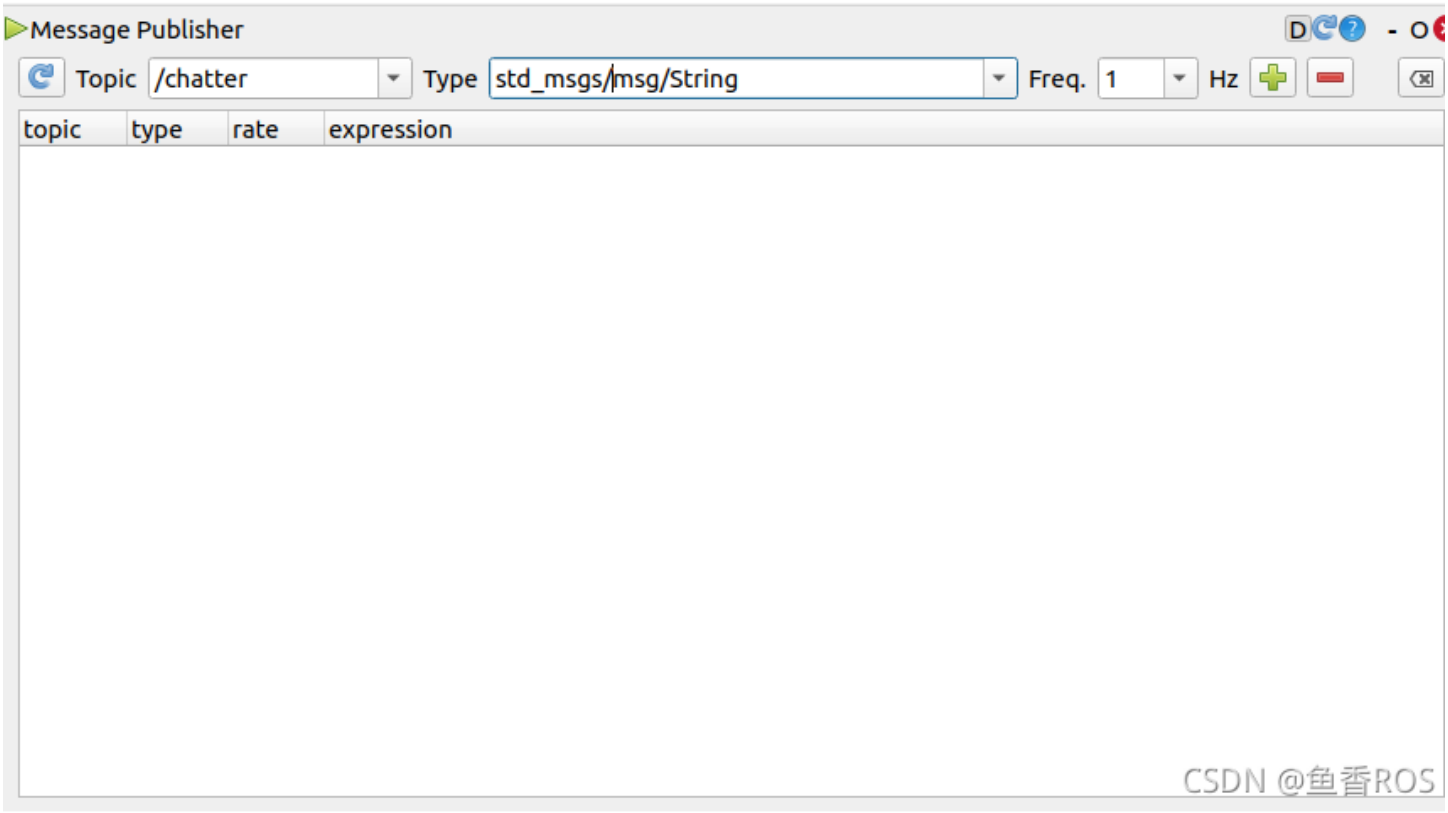
Introspection / Process Monitor

这个插件可以看到所有与ROS2相关的进程



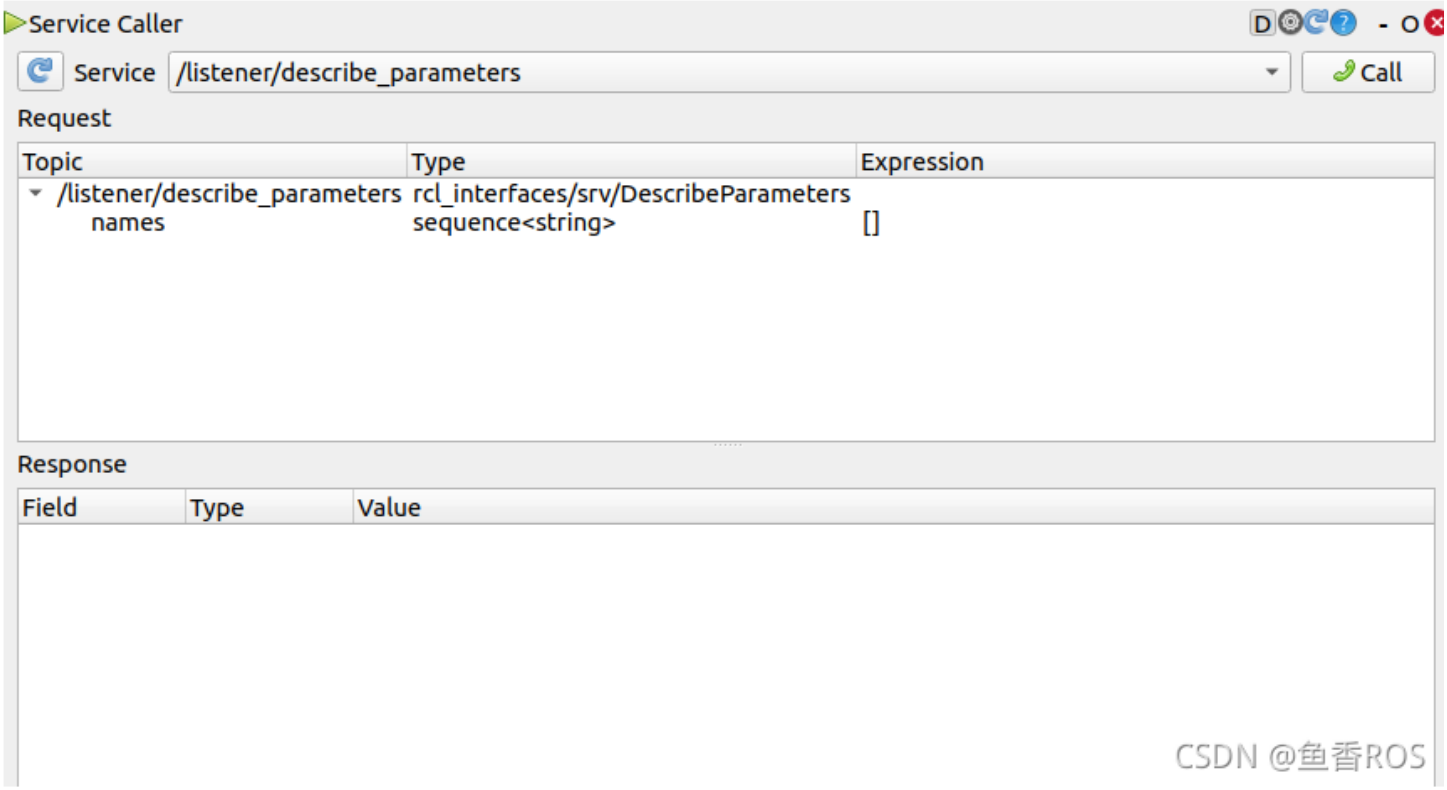
Topic/ Message Publisher

可以图形化发布话题数据



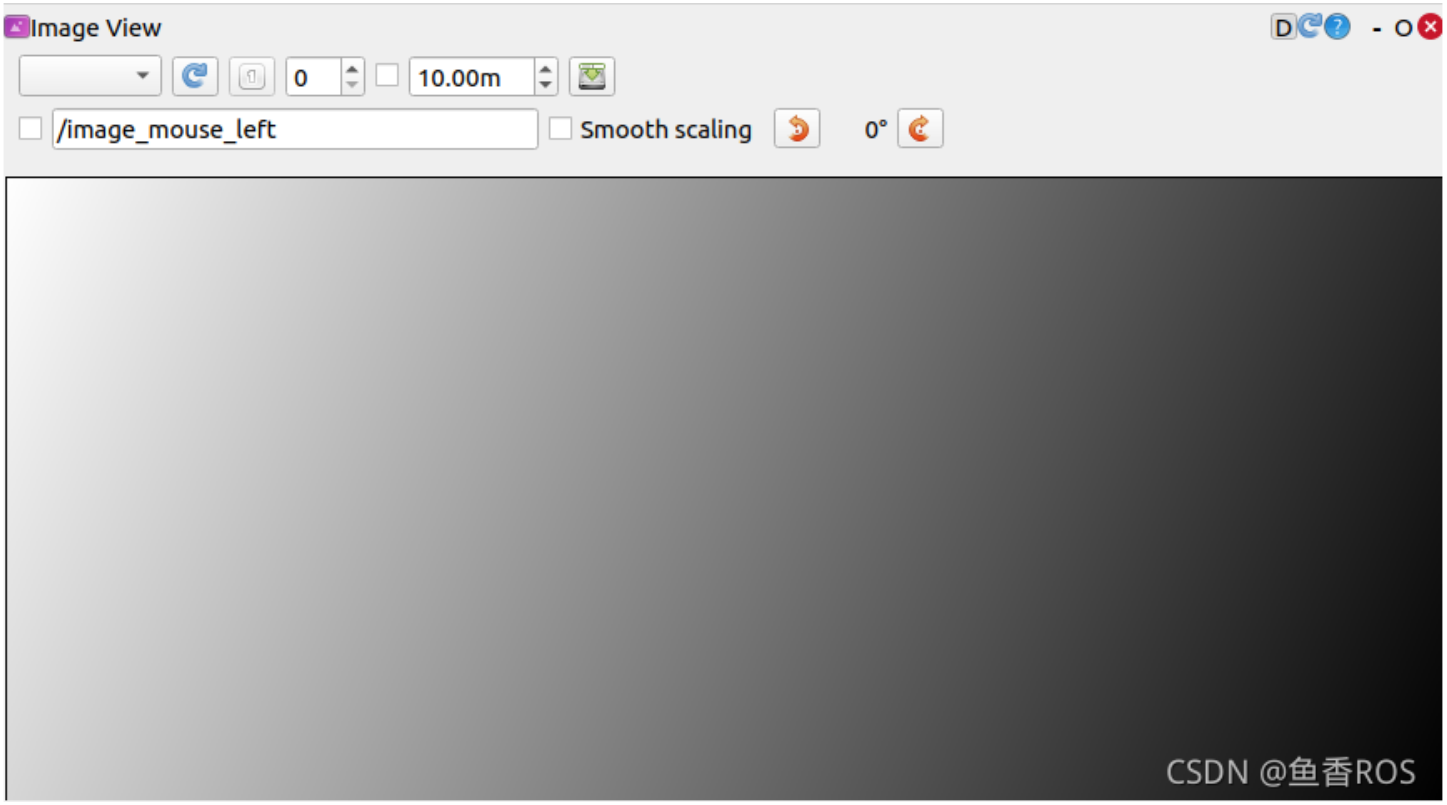
Service /Service Caller

图形化调用服务工具



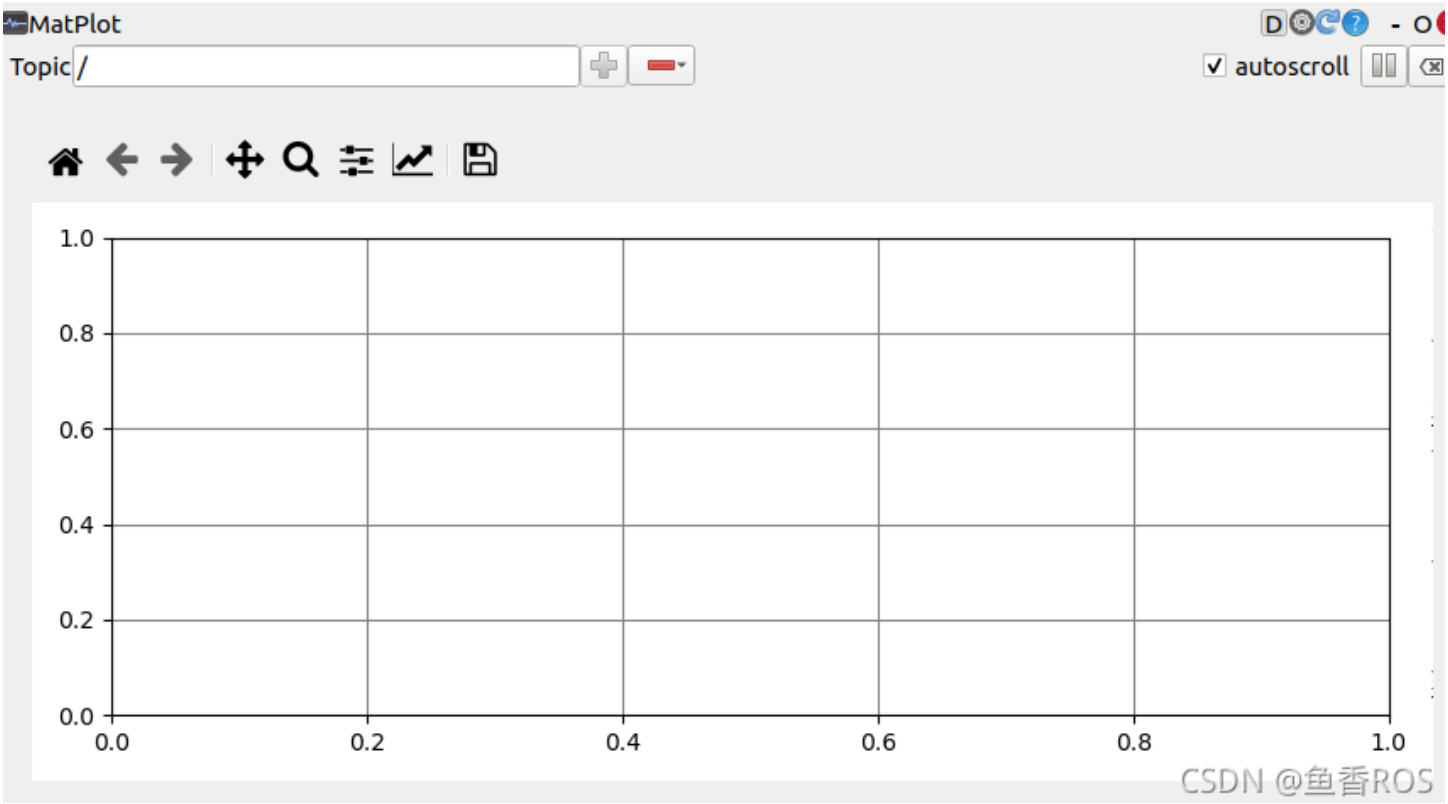
Visualization / Image View

看图像话题数据的Image View

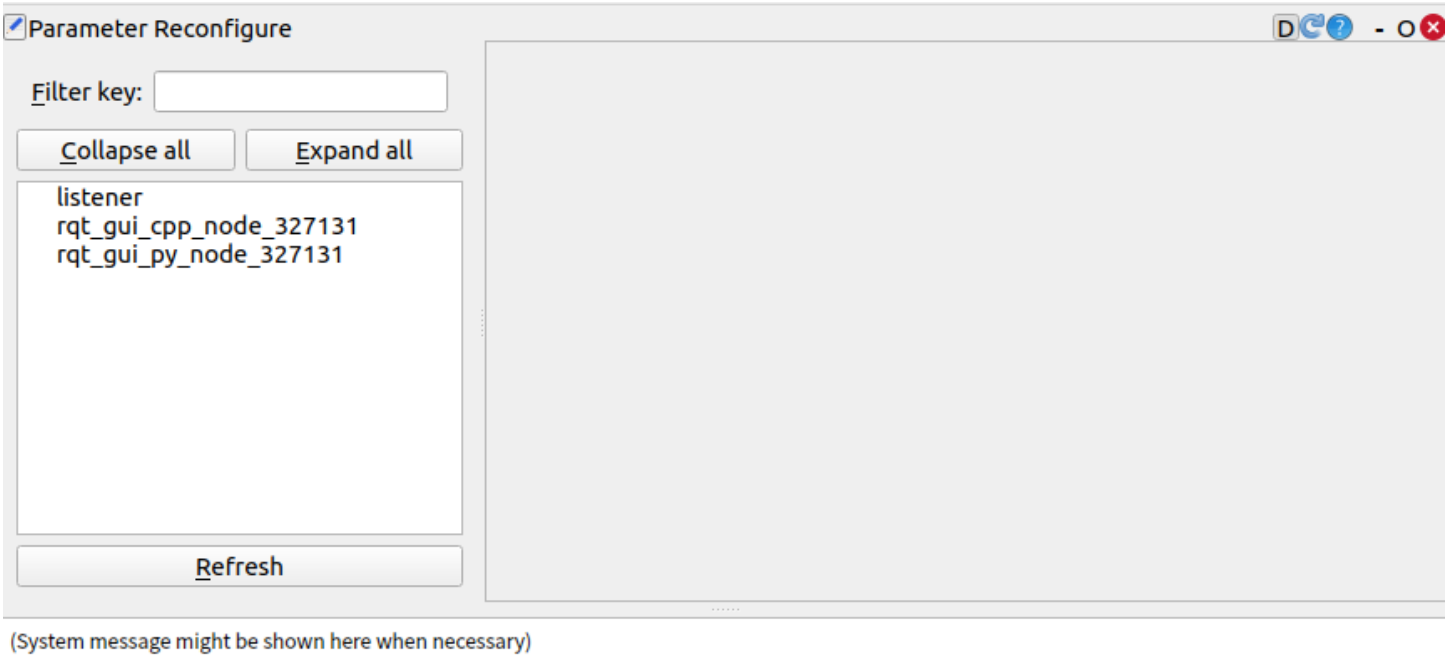


Visualization / MatPlot

话题数据图形化工具MqtPlot，小鱼就是用这个工具来调PID的



Configuration / Parameter Reconfigure



CSDN @鱼香ROS

3. 总结

小鱼准备有时间开发一些RQT工具来满足平时的一些特殊场景的使用，到时写一个教程出来，下一节我们一起学习ROS2中的RVIZ2工具。

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

2时光记录仪之rosbag2

本节我们来介绍ROS2中常用的一个CLI工具——rosbag2，这个工具用于记录话题的数据（就像小鱼录视频一样）。

我们就可以使用这个指令将话题数据存储为文件 ，后续我们无需启动节点，直接可以将bag文件里的话题数据发布出来。

> 这个工具在我们做一个真实机器人的时候非常有用，比如我们可以录制一段机器人发生问题的话题数据，录制完成后可以多次发布出来进行测试和实验，也可以将话题数据分享给别人用于验证算法等。

我们尝试使用bag工具来记录话题数据，并二次重放。

一、安装

当我们安装ROS2的时候，这个命令行工具已经为我们自动安装了，这里我们就不需要再次安装。

二、记录

2.1 常用指令

启动talker

```
ros2 run demo_nodes_cpp talker
```

2.1.1 记录

`/topic-name` 为话题名字

```bash
ros2 bag record /topic-name
```

2.1.2 记录多个话题的数据

```bash
ros2 bag record topic-name1  topic-name2
```

2.1.3 记录所有话题

```bash
ros2 bag record -a
```

2.1.4其他选项

-o name 自定义输出文件的名字

```bash
ros2 bag record -o file-name topic-name
```

-s 存储格式

目前仅支持sqllite3,其他还带拓展，后续更新小鱼再更新。

2.2 录制chatter

2.2.1 启动talker

运行talker节点

```bash
ros2 run demo_nodes_cpp talker
```

```
ros2@ros2-TM1613R:~/town_ws/rosbag2_2021_10_03-15_31_41$ ros2 bag info rosbag2_2021_10_03-15_31_
41_0.db3
[INFO] [1633247548.381559563] [rosbag2_storage]: Opened database 'rosbag2_2021_10_03-15_31_41_0.
db3' for READ_ONLY.

Files:             rosbag2_2021_10_03-15_31_41_0.db3
Bag size:          16.0 KiB
Storage id:        sqlite3
Duration:          130.0s
Start:             Oct  3 2021 15:31:44.42 (1633246304.42)
End:               Oct  3 2021 15:33:54.42 (1633246434.42)
Messages:          27
Topic information: Topic: /sexy_girl | Type: std_msgs/msg/String | Count: 27 | Serialization For
mat: cdr
```
CSDN @鱼香ROS

2.2.2 录制

接着使用像下面的指令就可以进行话题数据的录制了

```bash
ros2 bag record /chatter
```

如何停止录制呢？我们直接在终端中使用 `Ctrl+C` 指令打断录制即可

接着你会在终端中发现多处一个文件夹，名字叫做 `rosbag2_xxxxxx.db3`

打开文件夹，可以看到内容

```
ros2@ros2-TM1613R:~/town_ws/rosbag2_2021_10_03-15_31_41$ tree
.
├── metadata.yaml
└── rosbag2_2021_10_03-15_31_41_0.db3

0 directories, 2 files
```

这样我们就完成了录制。

三、查看录制出话题的信息

我们在播放一个视频前，可以通过文件信息查看视频的相关信息，比如话题记录的时间，大小，类型，数量

```bash
ros2 bag info bag-file
```

四、播放

4.1 播放话题数据

接着我们就可以重新的播放数据,使用下面的指令可以播放数据

```bash
ros2 bag play xxx.db3
```

使用ros2的topic的指令来查看数据

```bash
ros2 topic echo /chatter
```

4.2 播放选项

4.2.1 倍速播放 -r

-r选项可以修改播放速率，比如 -r 值，比如 -r 10,就是10倍速，十倍速播放话题

```bash
ros2 bag play rosbag2_2021_10_03-15_31_41_0.db3 -r 10
```

4.2.2 -l 循环播放

单曲循环就是它了

```bash
ros2 bag play rosbag2_2021_10_03-15_31_41_0.db3  -l
```

4.2.3 播放单个话题

```bash
ros2 bag play rosbag2_2021_10_03-15_31_41_0.db3 --topics /chatter
```

五、总结

相信你已经掌握了ROS2的bag工具~

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

6.兼容仿真工具-Gazebo

今天说说Gazebo，有些同学没有学习RVIZ和Gazebo之前，分不清Gazebo和Rviz之间的区别，只道是Gazebo和RVIZ都能显示机器人模型。

1.Gazebo VS Rviz2

昨天小鱼有说RVIZ2是什么：

文章中讲道**RVIZ2是用来可视化数据的软件，核心要义是将数据展示出来（我们不生产数据只做数据的搬运工）。 而Gazebo是用于模拟真实环境生产数据的（我们不搬运数据只做数据的生产者）**

所以Gazebo可以根据我们所提供的机器人模型文件，传感器配置参数，给机器人创造一个虚拟的环境，虚拟的电机和虚拟的传感器，并通过ROS/ROS2的相关功能包把传感器数据电机数据等发送出来（生产数据）。

这样我们就不用花一分钱，就拥有了各式各样的机器人和传感器（一万八的雷达，也只不过是用鼠标拖拽一下）。

2.Gazebo集成ROS2

Gazebo 是一个独立的应用程序，可以独立于 ROS 或 ROS 2 使用。

Gazebo与ROS 版本的集成是通过一组叫做 `gazebo_ros_pkgs` 的包 完成的， `gazebo_ros_pkgs` 将Gazebo和ROS2连接起来。

```mermaid
graph LR
A[Gazebo] -- API --> B(gazebo_ros-pkgs)
B --ROS2通信机制--> C[ROS2]
```

2.1 gazebo_ros_pkgs

gazebo_ros_pkgs不是一个包，是一堆包如下：

- gazebo_dev：开发Gazebo插件可以用的API
- gazebo_msgs：定义的ROS2和Gazebo之间的接口（Topic/Service/Action）

- gazebo_ros：提供方便的 C++ 类和函数，可供其他插件使用，例如转换和测试实用程序。它还提供了一些通常有用的插件。gazebo_ros::Node
- gazebo_plugins：一系列 Gazebo 插件，将传感器和其他功能暴露给 ROS2 例如:
  1. `gazebo_ros_camera` 发布ROS2图像
  2. `gazebo_ros_diff_drive` 通过ROS2控制和获取两轮驱动机器人的接口

> ROS1的插件迁移到ROS2进度：https://github.com/ros-simulation/gazebo_ros_pkgs/wiki ![已经从ROS1迁移的plugin](6.兼容仿真工具-Gazebo - /imgs/watermark,type_ZHJvaWRzYW5zZmFsbGJhY2s,shadow_50,text_Q1NETiBA6bG86aaZUk9T,size_20

3. 两轮差速小demo

3.1安装gazebo

因为安装ROS2不会默认安装gazebo，所以我们要手动安装,一行命令很简单，如果提示找不到先去更新下ROS2的源。

```shell
sudo apt install gazebo
```

3.2 安装ROS2的两轮差速功能包

一行代码全给装了，不差这点空间

```shell
sudo apt install ros-humble-gazebo-*
```

3.3 运行两轮差速demo

一行代码搞定

```shell
gazebo
/opt/ros/humble/share/gazebo_plugins/worlds/gazebo_ros_diff_drive_demo.world
```

然后你就可以看到一个死丑死丑的小车 ![car](6.兼容仿真工具-Gazebo -
/imgs/watermark,type_ZHJvaWRzYW5zZmFsbGJhY2s,shadow_50,text_Q1NETiBA6bG86aaZUk9T,size_20,color_FFF

3.4 查看话题

通过下面的指令可看到话题和话题的类型，把目光放到这个话题 `/demo/cmd_demo` ,下面我们就通过这个话题来控制小车动起来。

```shell
ros2@ros2-TM1613R:~$ ros2 topic list -t
/clock [rosgraph_msgs/msg/Clock]
/demo/cmd_demo [geometry_msgs/msg/Twist]
/demo/odom_demo [nav_msgs/msg/Odometry]
/parameter_events [rcl_interfaces/msg/ParameterEvent]
/rosout [rcl_interfaces/msg/Log]
/tf [tf2_msgs/msg/TFMessage]
```

3.5 让小车前进

```shell
ros2 topic pub /demo/cmd_demo geometry_msgs/msg/Twist "{linear: {x: 0.2,y: 0,z:
0},angular: {x: 0,y: 0,z: 0}}"
```

然后就可以看到小车动了起来。

![动起来](6.兼容仿真工具-Gazebo -
/imgs/watermark,type_ZHJvaWRzYW5zZmFsbGJhY2s,shadow_50,text_Q1NETiBA6bG86aaZUk9T,size_20,color_FFF

4.总结

- RVIZ2是用来可视化数据的软件，核心要义是将数据展示出来（我们不生产数据只做数据的搬运工）
- Gazebo是用于模拟真实环境生产数据的（我们不搬运数据只做数据的生产者)
- Gazebo是独立于ROS/ROS2的软件（还有很多仿真软件可以用ROS/ROS2）
- ROS2和Gazebo之间的桥梁是：gazebo_ros_pkgs

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

1.矩阵与矩阵运算

大家好，我是小鱼。本节我们来学习一下线性代数的基础中的矩阵部分，矩阵作为我们学习机器人学中最常用的基础知识，后面学习过程中我们会经常遇到，比如：表示旋转的旋转矩阵、坐标变换中的齐次矩阵、关节速度映射雅可比矩阵、仿真中的惯性矩阵等等。所以很有必要在正式学习之前，了解一下矩阵的概念及常用的矩阵定义。

1.矩阵介绍

1.1 矩阵定义

由mn个数$a_{ij}(i=1,2,..,m;j=1,2...,n)$排成的m行n列的矩阵表格 $$ \begin{bmatrix} a_{11}}&{a_{12}}&{\cdots}&{a_{1n}}\\ {a_{21}}&{a_{22}}&{\cdots}&{a_{2n}}\\ {\vdots}&{\vdots}&{\ddots}&{\vdots}\\ {a_{m1}}&{a_{m2}}&{\cdots}&{a_{mn}}\\ \end{bmatrix} $$ 称为一个mn的矩阵,记为为A或$(a_{ij})_{m*n}(i=1,2,..,m;j=1,2...,n)$，当$m=n$时称$A$为$n$阶方阵.

> **矩阵就是一堆可能存在着某种联系数的组合，编号规则也很简单，第一行第一列的数编号为a_{11},第二行第一列叫做a_{21}，以此类推**

如果两个矩阵都是m*n个数组成，则称两个矩阵为同型矩阵。

1.2 零矩阵

所谓零矩阵，就是矩阵中每个数都是0,比如一个$3$3$的$0$矩阵（零矩阵常用$O$来表示） $$ O_{33} =\begin{bmatrix}{0}&{0}&{0}\\{0}&{0}&{0}\\{0}&{0}&{0}\\\end{bmatrix} $$

零矩阵是不是和自然数零一样神奇呢？

根据矩阵的运算法则，零矩阵有以下性质，下一节我们会来动手验证。

- 任何矩阵（前提符合运算法则）与零矩阵相加、减结果都是其本身 $$ A-O=A \\ A+O=A $$
- 零矩阵与任何矩阵的相乘结果都是零矩阵（注意，矩阵型号可能会变）

1.3 单位矩阵

主对角线上的元素都为1，其余元素全为0的n阶矩阵称为n阶单位矩阵,常用符号I表示，如$I3$ $$ I_{3} =\begin{bmatrix}{1}&{0}&{0}\{0}&{1}&{0}\{0}&{0}&{1}\\end{bmatrix} $$

单位矩阵的性质与自然数1相似

根据矩阵的运算法则，单位矩阵有以下性质：

任何矩阵与单位矩阵的乘积结果为其本身 $$ AI_n = A\\ I_nB = B $$

2.矩阵的运算

2.1加减法运算

两个矩阵相加减，即其对应元素相加减。

设矩阵 $$ A=\begin{bmatrix} {a_{11}}&{a_{12}}&{\cdots}&{a_{1n}}\ {a_{21}}&{a_{22}}&{\cdots}&{a_{2n}}\ {\vdots}&{\vdots}&{\ddots}&{\vdots}\ {a_{m1}}&{a_{m2}}&{\cdots}&{a_{mn}}\ \end{bmatrix},B=\begin{bmatrix} {b_{11}}&{b_{12}}&{\cdots}&{b_{1n}}\ {b_{21}}&{b_{22}}&{\cdots}&{b_{2n}}\ {\vdots}&{\vdots}&{\ddots}&{\vdots}\ {b_{m1}}&{b_{m2}}&{\cdots}&{b_{mn}}\ \end{bmatrix} $$ 有

$$ A\pm;B=\begin{bmatrix} {a_{11}\pm;b_{11}}&{a_{12}\pm;b_{12}}&{\cdots}&{a_{1n}\pm;b_{1n}}\ {a_{21}\pm;b_{21}}&{a_{22}\pm;b_{22}}&{\cdots}&{a_{2n}\pm;b_{2n}}\ {\vdots}&{\vdots}&{\ddots}&{\vdots}\ {a_{m1}\pm;b_{m1}}&{a_{m2}\pm;b_{m2}}&{\cdots}&{a_{mn}\pm;b_{mn}}\ \end{bmatrix} $$

> **只有两个矩阵为同型矩阵时才能进行加减运算。**

运算性质

- 交换律：$A+B=B+A$
- 结合律：$(A+B)+C=A+(B+C)$
- 乘法结合律：$(AB)C=A(BC)$

栗子：

$$ A=\begin{bmatrix}{1}&{0}&{1}\{1}&{0}&{0}\{0}&{0}&{1}\\end{bmatrix},B=\begin{bmatrix}{0}&{0}&{1}\{1}&{0}&{0}\{0}&{0}&{1}\\end{bmatrix} $$ $$ A+B= B+A =\begin{bmatrix}{1}&{0}&{2}\{2}&{0}&{0}\{0}&{0}&{2}\\end{bmatrix} $$

2.2乘法运算

乘法运算分为两种，一种是标量乘法，一种是矩阵乘法。

2.2.1 标量乘法

标量乘法即一个矩阵和一个数相乘。运算法则：将矩阵的每一个元素都乘上这个数即可

栗子： $$ A = \begin{bmatrix}{1}&{2}\{3}&{4}\\end{bmatrix}\ $$ $$ 2\times A= 2\times \begin{bmatrix}{1}&{2}\{3}&{4}\\end{bmatrix} = \begin{bmatrix}{2 \times 1}&{2 \times 2}\{2 \times 3}&{2 \times 4}\\end{bmatrix} =\begin{bmatrix}{2}&{4}\{6}&{8}\\end{bmatrix} $$

2.2.2 矩阵运算

运算法则

设矩阵$A\times B = C = (c_{ij})*{m*n}$，则$C$的第$i$行第$j$列的元素$c*{ij}$的值等于矩阵A的第$i$行元素和矩阵B的第$j$列元素两两乘积之和。

栗子：

设： $$ A = \begin{bmatrix}{1}&{2}\{3}&{4}\\end{bmatrix}, B = \begin{bmatrix}{1}&{0}\{0}&{3}\\end{bmatrix} \ $$

则 $$ A \times B = C = \begin{bmatrix}{c_{11}}&{c_{12}}\{c_{21}}&{c_{22}}\\end{bmatrix} $$

$$ c_{11} =1*1+2*0 = 1 $$

$$ c_{12} =1*0+2*3 = 6 $$

$$ c_{21} = 3*1+4*0 = 3 $$

$$ c_{22} = 3+0 + 4*3 = 12 $$

$$ C = \begin{bmatrix}{1}&{6}\{3}&{12}\\end{bmatrix} $$

> 乘积之和其实就是点乘运算，比如栗子： $$ a = [1,2,3],b = [0,1,2]\ a\cdot b =1*0+2*1+3*2 = 8 $$
>
> 放一张摘抄的很形象的图片：
>
> 
> 矩阵的乘法的意义是非常有意思的，这里放一个链接，欢迎大家阅读：矩阵乘法的本质是什么？

运算性质

尝试计算下上面栗子中的$B\times A$的值，得到的结果依然是上面栗子中的C吗?

答案：并不是，一般情况下，矩阵的乘法并不满足交换律

矩阵的运算规律：

- 结合律 $$ (A\times B)\times C = A \times(B\times C) $$
- 分配律 $$ A\times(B+C) = A\times B + A\times C $$

2.3转置运算

转置运算定义非常简单，将矩阵的对应行列元素互换(右上角加${T}$表示） $$ C = \begin{bmatrix}{c_{11}}&{c_{12}}\\{c_{21}}&{c_{22}}\\\end{bmatrix} ,C^{T} = \begin{bmatrix}{c_{11}}&{c_{21}}\\{c_{12}}&{c_{22}}\\\end{bmatrix} \ $$ **栗子：** $$ A = \begin{bmatrix}{1}&{2}\\{3}&{4}\\\end{bmatrix},A^T = \begin{bmatrix}{1}&{3}\\{2}&{4}\\\end{bmatrix} $$ **运算规律** $$ (A^T)^T = A $$

$$ (A+B)^T = A^T + B^T $$

$$ (AB)^T = B^TA^T $$

3.重要定义

3.1 矩阵的逆

3.1.1 定义

A,B是n阶方阵，E是n阶单位矩阵，若$AB=BA=E$,则称A是可逆矩阵，并称B是A的逆，且逆矩阵是唯一的，记作A^{-1}

这个我们本节课就不举栗子了，下节课我们使用代码来直接求。

> 注意：不一定所有的矩阵都是可逆的

3.1.2 运算规律

$$ (A^{-1})^{-1} = A $$

$$ AB可逆，(AB)^{-1}=B^{-1}A^{-1} $$

4.总结

本节主要介绍了下矩阵的定义的一些性质，还有很多需要补充的，由于时间原因先发一下文章，小鱼要去录课了

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

2.MiniConda与Jupyter介绍安装

大家好，我是小鱼，上一节我们介绍了机器人学的基础——矩阵相关知识，秉持着学以至用的原则，小鱼准备带大家先用代码来实现下矩阵的创建以及矩阵的运算。

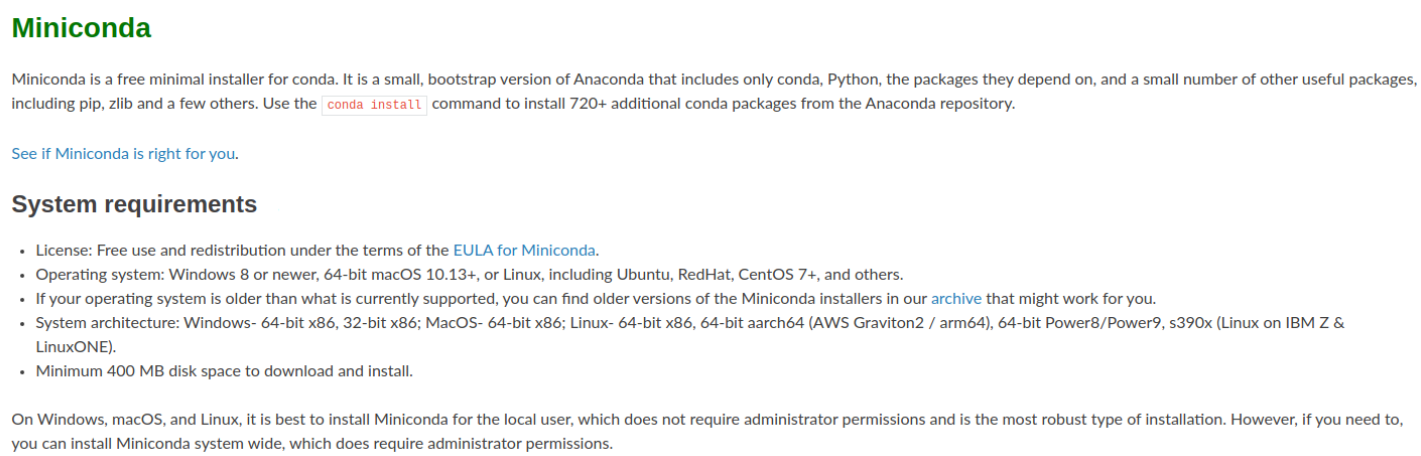
工欲善其事，必先利其器，如果是编写机器人相关算法，小鱼最为推荐的就是使用jupyter来和ros进行通信了。

所以本节课小鱼就带你一起安装好MiniConda，并在MiniConda中安装好jupyter并配置好ros2环境。

1.MiniConda与Jupyter介绍

1.1 MiniConda 介绍

Miniconda 是 Conda 的免费最小安装程序。



Miniconda

Miniconda is a free minimal installer for conda. It is a small, bootstrap version of Anaconda that includes only conda, Python, the packages they depend on, and a small number of other useful packages, including pip, zlib and a few others. Use the `conda install` command to install 720+ additional conda packages from the Anaconda repository.

See if Miniconda is right for you.

System requirements

- License: Free use and redistribution under the terms of the EULA for Miniconda.
- Operating system: Windows 8 or newer, 64-bit macOS 10.13+, or Linux, including Ubuntu, RedHat, CentOS 7+, and others.
- If your operating system is older than what is currently supported, you can find older versions of the Miniconda installers in our archive that might work for you.
- System architecture: Windows- 64-bit x86, 32-bit x86; MacOS- 64-bit x86; Linux- 64-bit x86, 64-bit aarch64 (AWS Graviton2 / arm64), 64-bit Power8/Power9, s390x (Linux on IBM Z & LinuxONE).
- Minimum 400 MB disk space to download and install.

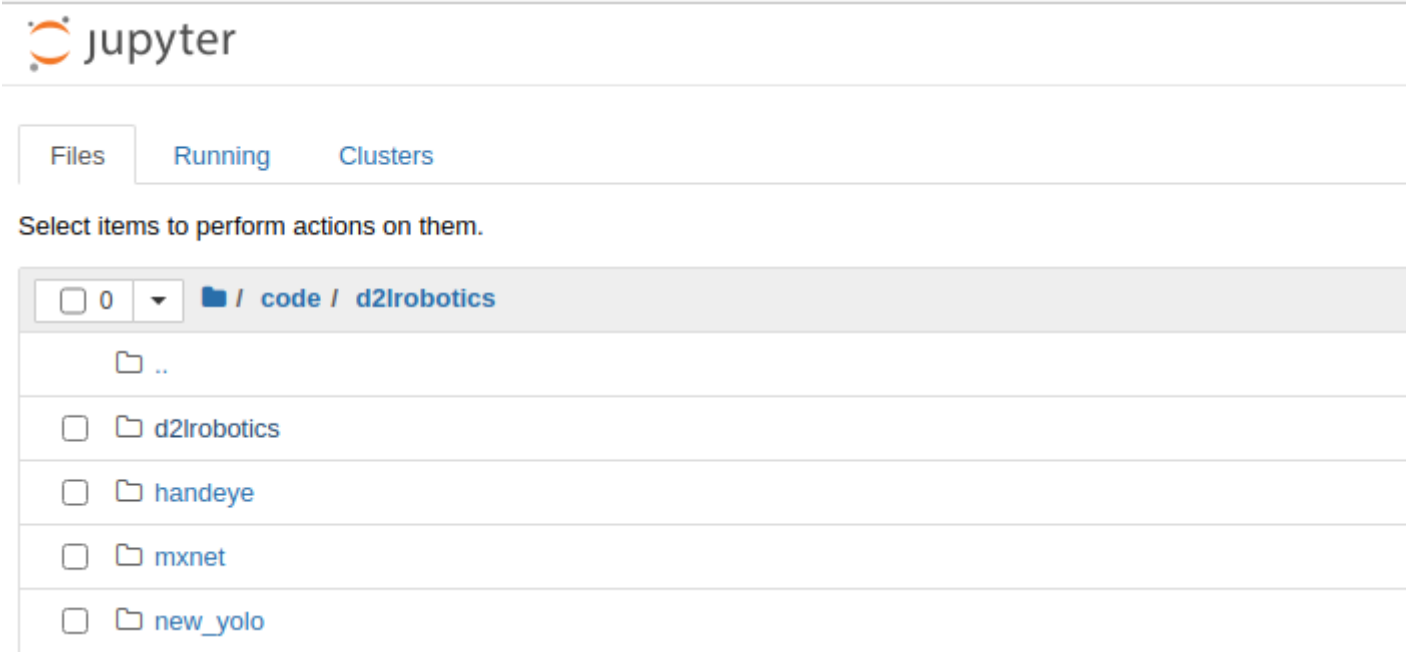
On Windows, macOS, and Linux, it is best to install Miniconda for the local user, which does not require administrator permissions and is the most robust type of installation. However, if you need to, you can install Miniconda system wide, which does require administrator permissions.

Conda是什么呢？ Conda是在Windows、macOS和Linux上运行的开源软件包管理系统和环境管理系统。Conda可以快速安装、运行和更新软件包及其依赖项。

所以相比于conda的大体积miniconda更加的小和易用。

1.2 Jupyter介绍

可以把jupyter理解为一个**可以在网页运行python**语言的工具。



jupyter提出了文学化编程的概念，让我们可以在单元格(cell)中，**像在笔记本上写文章一样，随时写，随时运行，随时根据运行的结果来修改我们的代码**，这样的交互式编程模式可以极大的提升我们编写代码的效率。

2.安装miniconda与jupyer

2.1 miniconda下载

- minconda下载地址：https://docs.conda.io/en/latest/miniconda.html

这里小鱼推荐选择 `Python3.8 Miniconda3 Linux 64-bit`

Linux installers

Linux

Python version	Name	Size	SHA256 hash
Python 3.9	Miniconda3 Linux 64-bit	63.6 MiB	1ea2f885b4dbc3098662845560bc64271
	Miniconda3 Linux-aarch64 64-bit	62.6 MiB	4879820a10718743f945d88ef142c3a4b
	Miniconda3 Linux-ppc64le 64-bit	60.6 MiB	fa92ee4773611f58ed9333f977d32bbb6
	Miniconda3 Linux-s390x 64-bit	57.1 MiB	1faed9abecf4a4ddd4e0d8891fc2cdaa3
Python 3.8	Miniconda3 Linux 64-bit	98.8 MiB	935d72deb16e42739d696449772903955
	Miniconda3 Linux-aarch64 64-bit	94.8 MiB	19584b4fb5c0656e0cf9de72aaa0b0a79
	Miniconda3 Linux-ppc64le 64-bit	93.3 MiB	c1ac79540cb77b2e0ca5b9f78b3bc3675
	Miniconda3 Linux-s390x 64-bit	89.0 MiB	55f514110a50e98549a68912cbb03e43a

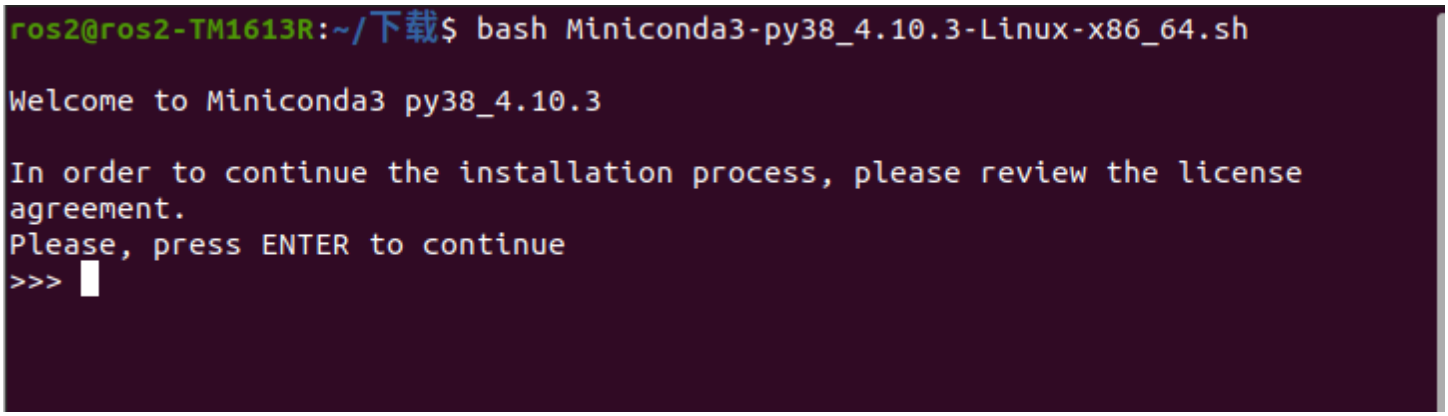
下载完成后你可以得到一个.sh的安装文件



2.2 miniconda安装

在安装目录文件夹空白处右击，在终端中打开,输入下面指令即可安装

```bash
bash Miniconda3-py38_4.10.3-Linux-x86_64.sh
```

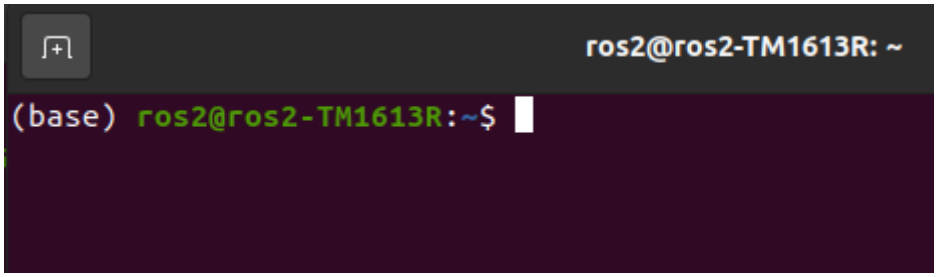


一路点enter，然后输入yes，回车即可完成安装，最后一步会提醒你是否要自动初始化环境，这里我们先选择yes。

没有选没事，跟着小鱼来一起初始化一次conda，打开终端输入下面的命令即可

```bash
cd ~/miniconda3/bin
./conda init
```

重启终端，你会发现你的终端前多了一个 `base` ,这代表你已经装好了conda。



如果想退出conda环境可以输入

```bash
conda deactivate
```

2.3 jupyter安装

方便起见，我们就在base环境下安装jupyter，使用的是清华大学源。

```bash
pip3 install jupyter -i https://pypi.tuna.tsinghua.edu.cn/simple
```

安装完成后，我们就可以使用下面的指令启动jupyter了，不出意外你的浏览器将要跳出来，然后自动打开conda页面。

```bash
jupyter-notebook
```

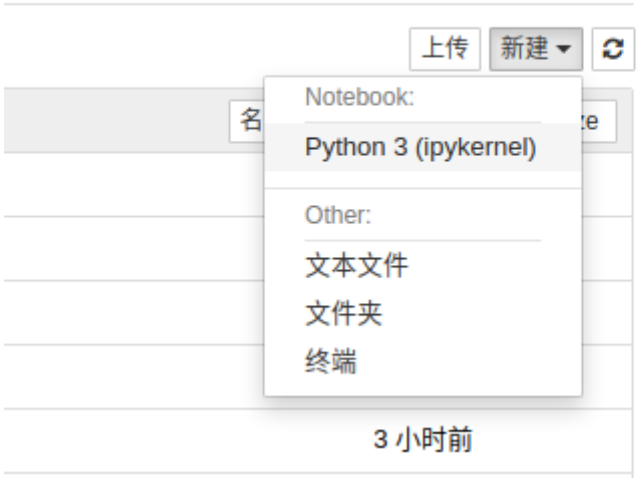
```
(base) ros2@ros2-TM1613R:~$ jupyter-notebook
[I 23:22:28.982 NotebookApp] 把notebook 服务cookie密码写入 /home/ros2/.local/share/jupyter/runtime/notebook_cookie_secret
[I 23:22:29.153 NotebookApp] 启动notebooks 在本地路径: /home/ros2
[I 23:22:29.153 NotebookApp] Jupyter Notebook 6.4.4 is running at:
[I 23:22:29.153 NotebookApp] http://localhost:8888/?token=0f9df50d31418c57d9d41abf14aa2d9a556c65a2cb5fd7e2
[I 23:22:29.153 NotebookApp]  or http://127.0.0.1:8888/?token=0f9df50d31418c57d9d41abf14aa2d9a556c65a2cb5fd7e2
[I 23:22:29.153 NotebookApp] 使用control-c停止此服务器并关闭所有内核(两次跳过确认).
[C 23:22:29.174 NotebookApp]
```

3.使用jupyter编写节点李四

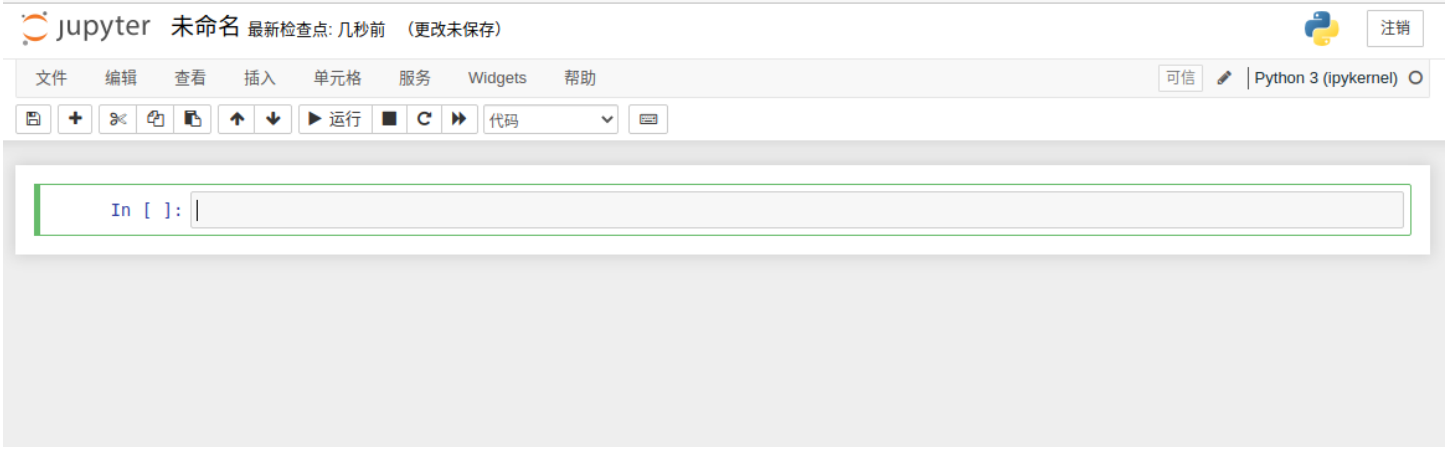
接着小鱼带你一起使用jupyter来写ros2的代码。

3.1 新建Python3并编写第一行代码

点击新建Python3



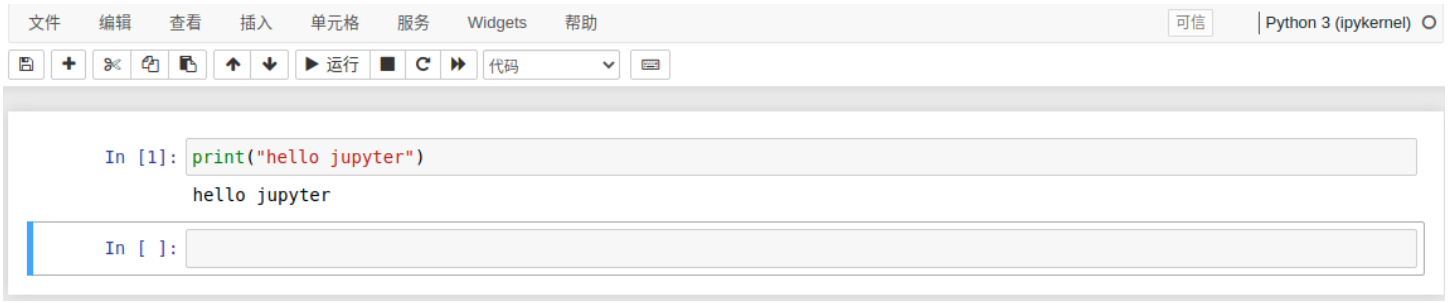
然后你可以看到一行行的单元格。



接着我们就可以在单元格里输入python代码，然后使用 `Shift+Enter` 来运行这行代码，比如我们可以试一试打印一句话

```bash
print("hello jupyter")
```

接着我们使用 `Shift+Enter` 来运行这行代码，可以看到这行代码的结果，随之展示在了该句打印的下方，我们可以修改打印的内容，然后重新使用 `Shift+Enter` 来运行，下面的数值也会随之改变。



3.2 编写ROS2代码

使用ros2无非是使用其rclpy客户端库，jupyter完美的支持了该库，一起和小鱼用import来导入吧。

第一行输入下面的指令，点击Shift+Enter可以看到没有任何报错，即导入成功了。

```python
import rclpy
from rclpy.node import Node
```

接着在和小鱼一起初始化客户端库，新建节点李四和spin节点吧，完整的代码如下：

```python
import rclpy
from rclpy.node import Node
rclpy.init() # 初始化rclpy
node = Node("li4")  # 新建一个节点
node.get_logger().info("大家好，我是作家li4.")
rclpy.spin(node) # 保持节点运行，检测是否收到退出指令（Ctrl+C）
```

分步输入运行

0.导入rclpy

```
In [2]: import rclpy
        from rclpy.node import Node
```

1.初始化rclpy

```
In [4]: rclpy.init()
```

2.新建一个节点

```
In [6]: node = Node("li4")
        node.get_logger().info("大家好，我是作家li4.")
        [INFO] [1634311650.628754260] [li4]: 大家好，我是作家li4.
Out[6]: True

In [*]: rclpy.spin(node)
```

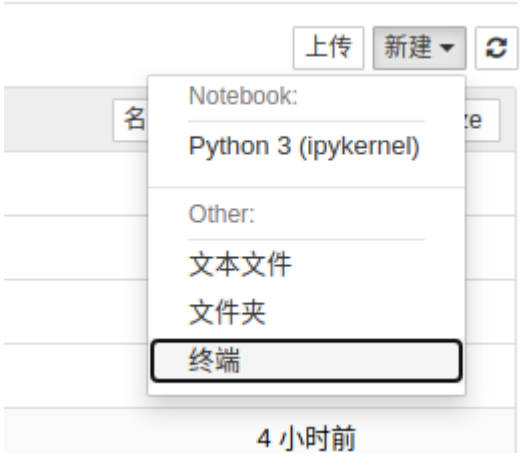
注意这里我们没有调用shutdown,因为我们还不希望rclpy关掉。

3.3 测试

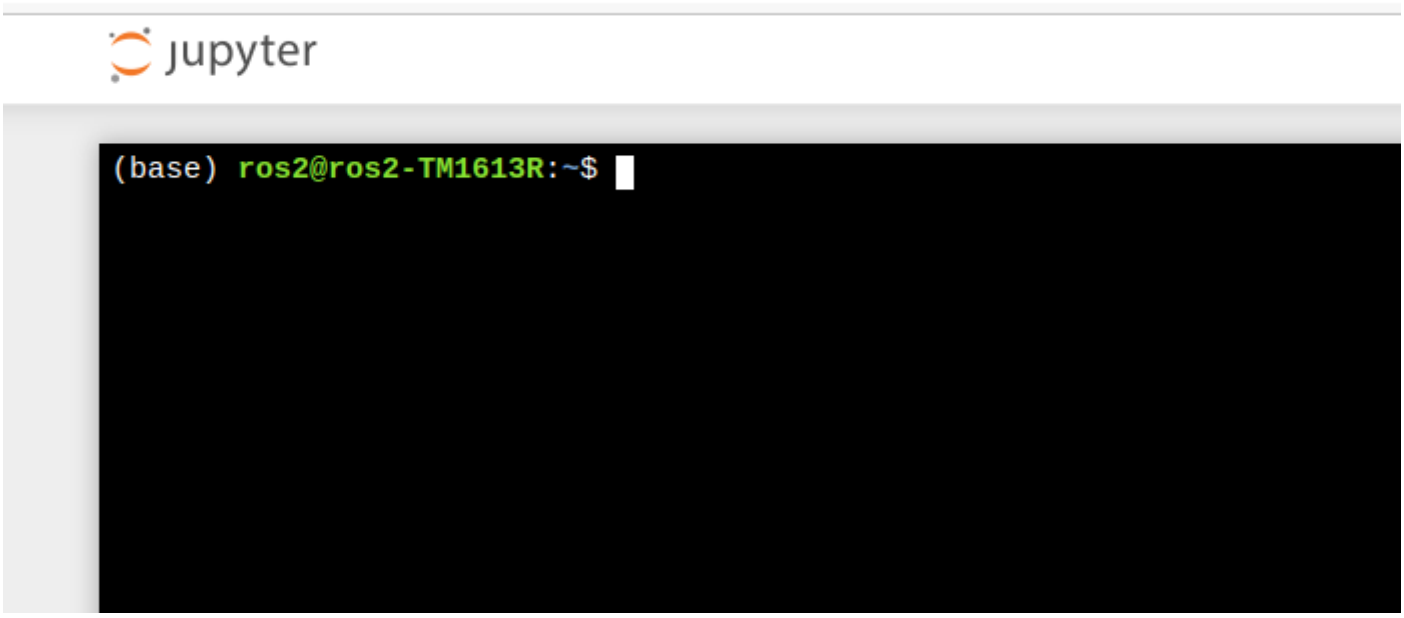
我们接下来用，ros2的命令行工具查看一下li4节点是否在线。

在jupyter中运行命令行和打开终端也是非常方便的，为了学习jupyter，我们使用jupyter自带的终端来运行ros2的指令。

我们新建一个终端，回到文件夹目录视图，点击新建终端



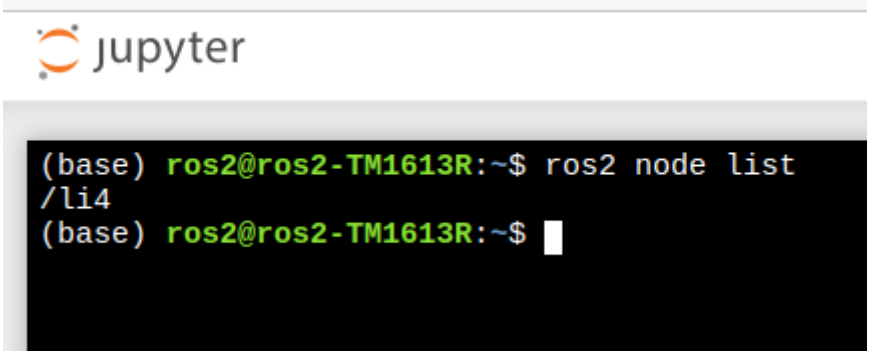
打开后的样子如下



在终端中输入

```python
ros2 node list
```

可以看到李四已经出现了



4.总结

小鱼不能说吹爆jupyter，但用python和ros写算法调试程序，jupyter真的是非常的好用，还有更强大的画图功能等等，小鱼后面在机器人学学习课程中都会一一讲到，并且带着大家一起来熟悉jupyter与ros2的更多知识。

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

3.矩阵运算实战

上一节我们安装好了MiniConda和Jupyter，本节课我们继续回到学习机器人学的路上来。首先我们来学习使用一个矩阵库Numpy，并且通过Numpy将我们第一节学习到的矩阵相关计算一一实现。

1.NumPy是什么

`NumPy` 是一个功能强大的 `Python` 库，主要用于对多维数组执行计算。NumPy这个词来源于两个单词-- `Numerical` 和 `Python` 。NumPy提供了大量的库函数和操作，可以帮助程序员轻松地进行数值计算。

通过这个库我们可以非常简单的完成矩阵的创建、矩阵的计算，不管是二维还是多维都非常的 `easy` .

NumPy可以使用在以下场景：

- **机器学习模型：** 在编写机器学习算法时，需要对矩阵进行各种数值计算。例如矩阵乘法、换位、加法等。NumPy提供了一个非常好的库，用于简单(在编写代码方面)和快速(在速度方面)计算。NumPy数组可用于存储训练数据和机器学习模型的参数。
- **图像处理和计算机图形学：** 计算机中的图像表示为多维数字数组。NumPy成为同样情况下最自然的选择。实际上，NumPy提供了一些优秀的库函数来快速处理图像。例如，镜像图像、按特定角度旋转图像等。
- **数学任务：** NumPy对于执行各种数学任务非常有用，如数值积分、微分、内插、外推等。因此，当涉及到数学任务时，它形成了一种基于Python的对MATLAB的快速替代。

2.Numpy安装

我们尝试在单元格内导入numpy，如果报错说明你没有安装

```bash
import numpy as np
```

```
In [1]: import numpy as np
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
/tmp/ipykernel_212911/3593848235.py in <module>
----> 1 import numpy as np

ModuleNotFoundError: No module named 'numpy'
```

在conda环境中安装非常的简单，也是一句话的事，在单元格内输入下面的代码，然后 `Shift+Enter` 执行即可。

> 注意指令前面有一个!，感叹号代表我们输入的是一句命令行

```bash
!pip install numpy  -i https://pypi.tuna.tsinghua.edu.cn/simple
```

```
In [2]: !pip install numpy  -i https://pypi.tuna.tsinghua.edu.cn/simple
Looking in indexes: https://pypi.tuna.tsinghua.edu.cn/simple
Collecting numpy
  Downloading https://pypi.tuna.tsinghua.edu.cn/packages/aa/69/260a4a1cc89cc00b51f432db048c396952f5c05dfa1345a1b3dbd9ea3544/numpy-1.21.2-cp38-cp38-manylinux_2_12_x86_64.manylinux2010_x86_64.whl (15.8 MB)
     |████████████████████████████████| 15.8 MB 334 kB/s eta 0:00:01
Installing collected packages: numpy
Successfully installed numpy-1.21.2
```

安装完成后，再次导入就不会报错了。

3.创建矩阵

导入完成之后我们就可以通过numpy来创建矩阵。

> numpy中有矩阵和数组两个概念，数组相比矩阵更加灵活

3.1 创建单位矩阵

创建一个3*3的零矩阵。

```bash
np.identity(3)
```

```
In [7]: np.identity(3)
Out[7]: array([[1., 0., 0.],
               [0., 1., 0.],
               [0., 0., 1.]])
```

3.2 创建零矩阵

创建一个3*3的零矩阵。

```python
np.zeros([3,3])
```

```
In [3]: import numpy as np

In [6]: np.zeros([3,3])
Out[6]: array([[0., 0., 0.],
               [0., 0., 0.],
               [0., 0., 0.]])
```

3.3 创建随机矩阵

创建一个3*3的随机矩阵。

```python
np.random.rand(3,3)
```

```
In [21]: np.random.rand(3,3)
Out[21]: array([[0.5204291 , 0.6720943 , 0.10043254],
                [0.92904076, 0.45134085, 0.5213488 ],
                [0.13795989, 0.56057858, 0.32739865]])
```

3.4 从已有数组创建矩阵

假设我们已经有了数据，我们想创建一个矩阵怎么办呢？

比如我们创建一个2*2的矩阵，矩阵的数据分别是[1,2,3,4]。

> 我们可以通过reshape改变矩阵的形状，这里我们把矩阵变成了2*2的样子。

```python
np.asarray([1,2,3,4]).reshape(2,2)
```

```
In [25]: np.asarray([1,2,3,4]).reshape(2,2)
Out[25]: array([[1, 2],
                [3, 4]])
```

3.5 判断两个矩阵是否相等

numpy的allclose方法，比较两个array是不是每一元素都相等，默认在1e-05的误差范围内。

我们做一个测试：

```python
print("零矩阵和单位矩阵",np.allclose(np.zeros([3,3]),np.identity(3)))
print("单位矩阵和单位矩阵",np.allclose(np.identity(3),np.identity(3)))
```

```
In [57]: print("零矩阵和单位矩阵",np.allclose(np.zeros([3,3]),np.identity(3)))
         print("单位矩阵和单位矩阵",np.allclose(np.identity(3),np.identity(3)))
         零矩阵和单位矩阵 False
         单位矩阵和单位矩阵 True
```

4.计算矩阵

计算矩阵我们主要对第一节中的矩阵算法进行验证。

4.1 矩阵加法/减法

加法使用 `np.add` ,减法 `np.subtract`

```python
A = np.asarray([1,2,3,4]).reshape(2,2)
B = np.zeros(2)
print(np.add(A,B))
```

```
:
A = np.asarray([1,2,3,4]).reshape(2,2)
B = np.zeros([2,2])
print(np.add(A,B))
```

```
[[1. 2.]
 [3. 4.]]
```

4.2 矩阵乘法

乘法使用 `np.dot`

（前提同形）任何矩阵乘上零矩阵等于零矩阵，任何矩阵乘上单位矩阵等于其本身

```python
A = np.asarray([1,2,3,4]).reshape(2,2)
B = np.zeros([2,2])
C = np.identity(2)
print(np.dot(A,B))
print(np.dot(A,C))
```

```
In [37]:
A = np.asarray([1,2,3,4]).reshape(2,2)
B = np.zeros([2,2])
C = np.identity(2)
print(np.dot(A,B))
print(np.dot(A,C))
```

```
[[0. 0.]
 [0. 0.]]
[[1. 2.]
 [3. 4.]]
```

4.3 矩阵求逆

矩阵求逆使用 `np.linalg.inv`

矩阵的逆与矩阵的乘积为单位矩阵

```python
A = np.asarray([1,2,3,4]).reshape(2,2)
A_INV = np.linalg.inv(A)
```

```
print(A_INV)
print(np.dot(A,A_INV))
```

```
In [41]: A = np.asarray([1,2,3,4]).reshape(2,2)
         A_INV = np.linalg.inv(A)
         print(A_INV)
         print(np.dot(A,A_INV))
```

```
[[-2.   1. ]
 [ 1.5 -0.5]]
[[1.0000000e+00 0.0000000e+00]
 [8.8817842e-16 1.0000000e+00]]
```

需要注意的是，并不是所有的矩阵都有逆，比如零矩阵就没有逆，如果尝试用 `numpy` 来求逆则会出现错误

python

```
O = np.zeros(2)
O_INV = np.linalg.inv(O)
```

```
In [70]: O = np.zeros(2)
         O_INV = np.linalg.inv(O)
```

```
---------------------------------------------------------------------------
LinAlgError                               Traceback (most recent call last)
/tmp/ipykernel_212911/174116926.py in <module>
      1 O = np.zeros(2)
----> 2 O_INV = np.linalg.inv(O)

<__array_function__ internals> in inv(*args, **kwargs)

~/miniconda3/lib/python3.8/site-packages/numpy/linalg/linalg.py in inv(a)
    537     """
    538     a, wrap = _makearray(a)
--> 539     _assert_stacked_2d(a)
    540     _assert_stacked_square(a)
    541     t, result_t = _commonType(a)
```

另外如果使用的是 `conda` 提供的 `numpy` 包，对于某些不存在逆的矩阵，并不会抛出错误。但是对于通过 `pip` 安装的相同版本甚至更高版本的 `numpy` ，则不会出现类似问题。

python

```
"""
numpy packages isntalled from conda:
numpy                     1.21.2           py38h20f2e39_0
numpy-base                1.21.2           py38h79a1101_0
"""
c=np.asarray([
    1,2,3,
```

```
    4,5,6,
    7,8,9,
]).reshape((3,3))

print("矩阵的行列式：",np.linalg.det(c))
print("矩阵的秩：",np.linalg.matrix_rank(c))
c_inv=np.linalg.inv(c)
print("矩阵c和c_inv点乘的结果：","\n",np.dot(c,c_inv))
```

```
[16]  ✓ 0.4s
...  矩阵的行列式:  -9.516197353929 94e-16
     矩阵的秩:  2
     矩阵c和c_inv点乘的结果:
      [[ 0.   1.  -0.5]
      [ 0.   2.  -1. ]
      [ 0.   3.   2.5]]
```

4.4 矩阵转置

矩阵转置在矩阵后使用 `.T` 即可

```python
A = np.asarray([1,2,3,4]).reshape(2,2)
A.T
```

```
In [43]: A = np.asarray([1,2,3,4]).reshape(2,2)
         A.T

Out[43]: array([[1, 3],
                [2, 4]])
```

5.总结

今天一起学习了使用numpy操作矩阵，还有更多的资料可以在公众号后台回复numpy获取，我们下节见～

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

1.空间坐标描述

1.何为位姿

1.1 引言

在书籍《天才在左，疯子在右》中有这样一个故事，有一位十七岁的少年，他对量子力学有着独特的见解和远超同龄人的知识储备，他说四维生物突破了时空的界限，在它的观察中，将人类一生的活动看成一条连续的线，就像一条长长的虫子，它可以看到人的过去，也可以看到人的未来。

我们对四维空间不了解，但对三维立体和二维的平面每天都有接触。一张铺平的中国地图是二维平面的，拔地而起的高楼大厦是三维立体的。

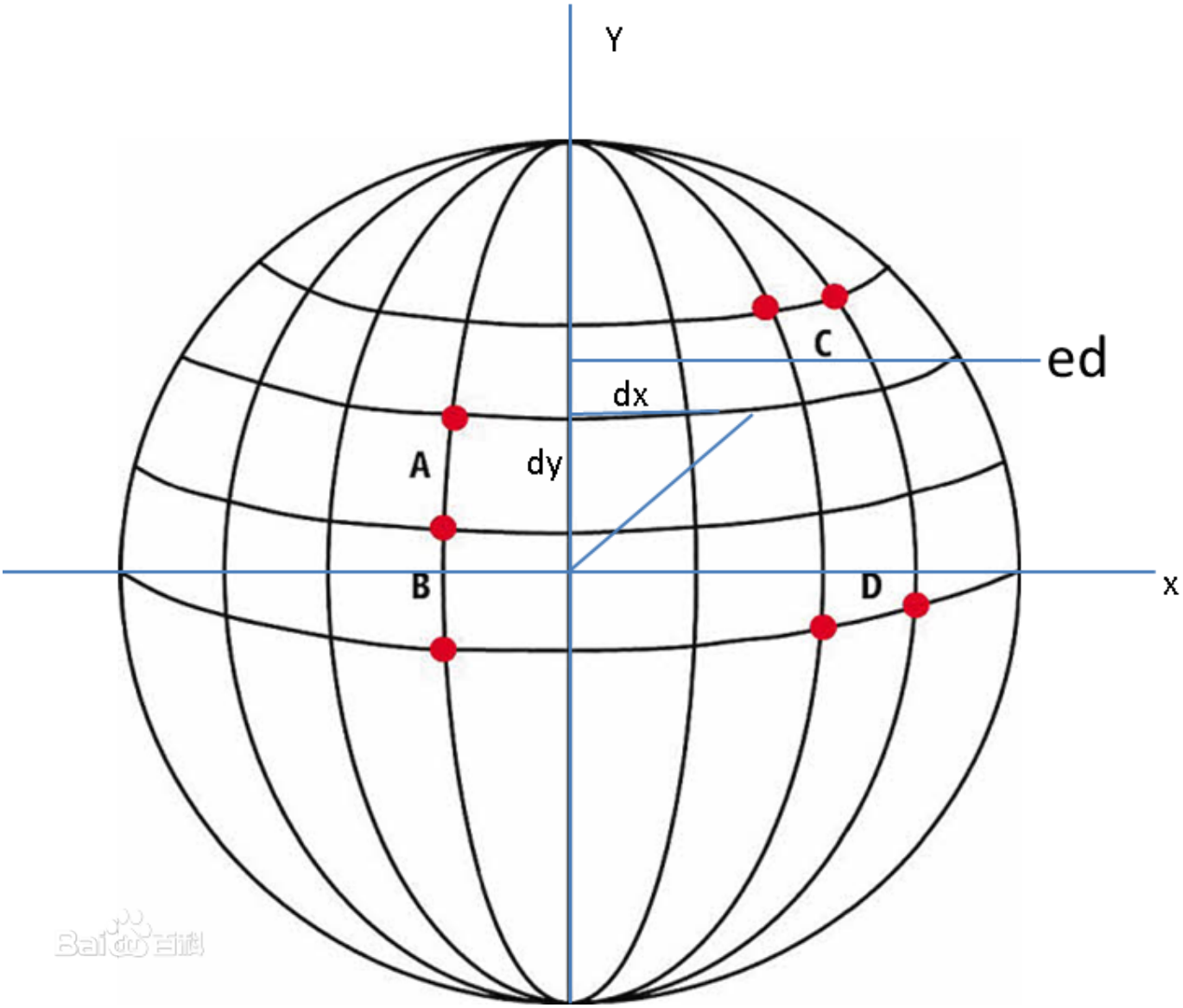
那我们如何用数字表示高楼在地图中的位姿呢？又如何用数字表示某个高楼中你的位姿呢？

1.2 我们先来看地图中高楼的位姿

在地图上，我们可以利用经纬度坐标系来描述位姿，在经纬度坐标系中，我们可以使用经度值和纬度值表示地图上的某一个大楼的坐标，比如天安门的经纬度 `经度：116.38 ，纬度：39.90`

经纬度坐标系是以经纬度原点（几内亚湾）建立的二维平面直角坐标系，因此我们可以使用x,y来描述该坐标系中的任意一个点的位置。

> 经度的起点就是指0度经线，位于英国格林尼治天文台旧址。 纬度的起点就是指0度纬线，就是指赤道。而经度0和纬度0的交点，大致位于非洲西部的几内亚湾，位于海上。



通过经纬度可以定位到地图中的建筑物的位置，但我们依然无法得知这个建筑物的朝向（东南西北），所以除了描述位置的经纬度外，我们还需要增加一个theta(θ),表示朝向。这样我们就得到了一个物体在二维平面中用数学描述的三个值，经度（x）、维度（y）和朝向（theta）。

1.3 高楼中你的位姿

现实世界是三维的，除了经纬度还有海拔高度。所以采用经度（x）、维度（y）并不能完整的描述出空间中物体的位置，例如只有经纬度无法表示你所处的楼层的海拔高度，这样就没办法确定你的位置。

我们同样可以在经纬度原点建立一个三维空间直角坐标系，采用经度（x）、维度（y）和海拔高度（z）三个值来描述三维空间中任意一点的位置。

和二维空间中类似，知道了你在某一个大楼中的位置还不够，并不能判断出你的姿态（躺平的、站着的还是侧卧的），那我们该如何描述三维空间中的姿态呢?

答案是旋转矩阵，旋转矩阵是什么？我们接着往下看。

1.4 参考坐标系

无论是在二维空间还是在三维空间，我们想要描述一个物体的位置和姿态第一步就是确定一个参考坐标系，物体的位置和姿态描述我们都是以这个坐标系作为参考的。

> **参考坐标系，这一点很重要，因为没有绝对的坐标，只有相对的坐标。**

2.位置的表示

从第一节背景中可以得知，二维平面中的位置可以用x,y表示，三维空间中的位置可以用x,y,z来表示。

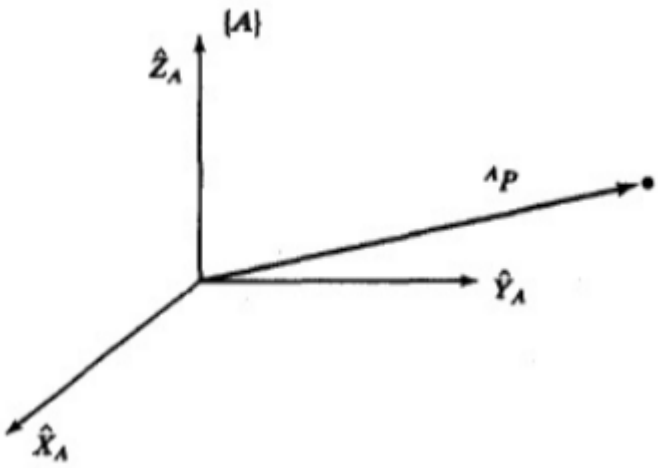
无论是在三维空间还是在二维平面，我们都可以使用x,y,z来表示位置，只不过对于二维空间来说，z的值是默认的一个固定不变的值，比如0

假如我们确定了一个空间直角坐标系A,我们就可以使用x,y,z来确定A坐标系中任意一点P的位置，可以记作${^A}P$

$$ {^A}P= \begin{bmatrix} x \ y \ z \ \end{bmatrix} $$

2.1 位置矢量

直角坐标系{A}其实可以看作是由三个互相正交（两两垂直）的单位矢量组成的。 那么在坐标系{A}中的一点P也可以写作矢量形式，其矢量形式由其在三个单位矢量上的分量组成。



所以参考坐标系{A}中一点P也可以写作 $$ {^A}P= \begin{bmatrix} {^x}p \ {^y}p \ {^z}p \ \end{bmatrix} = [{^x}p，{^y}p , {^z}p]^T \tag{位置矢量} $$

如果说位置矢量不太好理解，可以简单的认为就是坐标轴的x,y,z

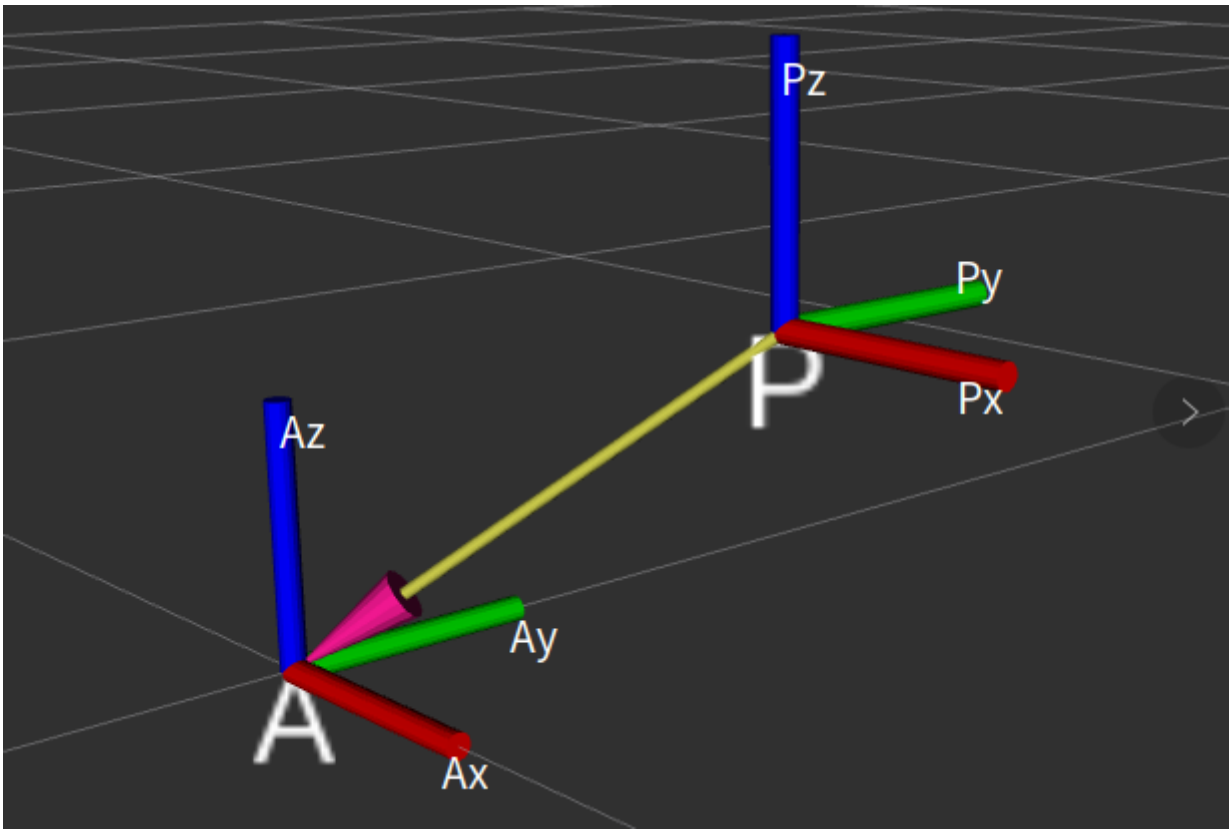
$$ \begin{bmatrix}{x}\{y}\{z}\\end{bmatrix} $$

3.姿态的表示

在背景中小鱼提到除了位置，坐标描述还有另外一个非常重要的组成部分——姿态。

接着上面的说，我们已经知道坐标系A中的一个点P的位置，我们如何描述P点在{A}坐标系下的姿态呢？

我们可以以P点为原点再建立一个坐标系(也可以认为该坐标系固定在物体P点上)，这样我们就可以通过描述**新坐标系{P}**和参考坐标系{A}**之间的姿态关系来表示${^A}P$点的姿态。



> 上图是使用RVIZ2绘制的，红色代表x轴，绿色代表y轴，蓝色代表z轴。

从图中可以看出，坐标系{P}的每一个轴和参考坐标系的每一个轴之间都有一个角度，比如Px轴和Ax,Ay,Az三个轴之间存在三个角度，通过这三个角度我们就可以确定Px轴和参考坐标系{A}之间的关系，以此类推，我们也可以确定Py轴和Pz轴分别与{A}之间的关系。

每个轴的相对姿态关系确定了，坐标系之间的姿态也就确定了，${^A}P$点的姿态也就确定了。

3.1 旋转矩阵

我们将上述坐标系{P}的三个轴相对于参考坐标系{A}三个轴的共九个角度的余弦值，组成一个3*3的矩阵，该矩阵就是旋转矩阵，因该矩阵是{P}相对于{A}的姿态关系的表示，故记作${^A_P}R$ $$ {^A_P}R=[{^A}x_{P} \ {^A}y_{P} \ {^A}z_{P}] = \begin{bmatrix}{r_{11}}&{r_{12}}&{r_{13}}\{r_{21}}&{r_{22}}&{r_{23}}\{r_{31}}&{r_{32}}&{r_{33}}\\end{bmatrix} \tag{旋转矩阵} $$

两个向量的点乘为两个向量的长度与它们夹角余弦的积,所以$r11$可以表示为单位向量P_{x}与A_{x}的点积，旋转矩阵就可以写为下面的形式

$$ {^A_P}R = \begin{bmatrix} {P_{x}\cdot A_x} & {P_{y}\cdot A_x} & {P_{z}\cdot A_x}\ {P_{x}\cdot A_y} & {P_{y}\cdot A_y} & {P_{z}\cdot A_y}\ {P_{x}\cdot A_z} & {P_{y}\cdot A_z} & {P_{z}\cdot A_z}\ \end{bmatrix} $$

将${^A_P}R$进行转置可得${^A_P}R^T$

$$ {^A_P}R^T = \begin{bmatrix} {P_{x}\cdot A_x} & {P_{x}\cdot A_y} & {P_{x}\cdot A_z}\ {P_{y}\cdot A_x} & {P_{y}\cdot A_y} & {P_{y}\cdot A_z}\ {P_{z}\cdot A_x} & {P_{z}\cdot A_y} & {P_{z}\cdot A_z}\ \end{bmatrix} $$

可以看出${^A_P}R^T$其实表示 `坐标系{P}` 作为参考坐标系下 `坐标系{A}` 的姿态，即 $$ {^A_P}R^T = {^P_A}R = {^A_P}R^{-1} $$

需要注意的是：

- 当两个坐标系之间姿态没有变化，即坐标系间x,y,z轴方向对应重合，则旋转矩阵为单位矩阵，这个很好求得，有兴趣的同学可以算一下
- 从矩阵的角度看，矩阵的逆等于矩阵的转置，则该矩阵为正交矩阵，显而易见，旋转矩阵是正交矩阵

4.位置+姿态

通过位置矢量我们可以描述一个点在特定参考坐标系下的位置，通过旋转矩阵可以描述一个点在特定参考坐标系下的姿态。

在机器人当中位置和姿态一般会一起出现，所以我们将其组合就叫做 `位姿`

4.1 位姿描述的多个含义

- 含义1：表示特定参考坐标系下某个物体（点）的位置和姿态，比如我们描述参考坐标系{A}中物体（点）P的位置和姿态

- 含义2：表示两个坐标系之间的位姿关系，比如位置可以表示坐标系{A}和坐标系{B}原点位置关系，姿态可以表示两个坐标系坐标轴的朝向关系

  坐标系之间关系，$^AP_{Bo}$表示两坐标系原点之间位置矢量 $$ {B} = { ^A_BR, {^A}P_{Bo} } $$

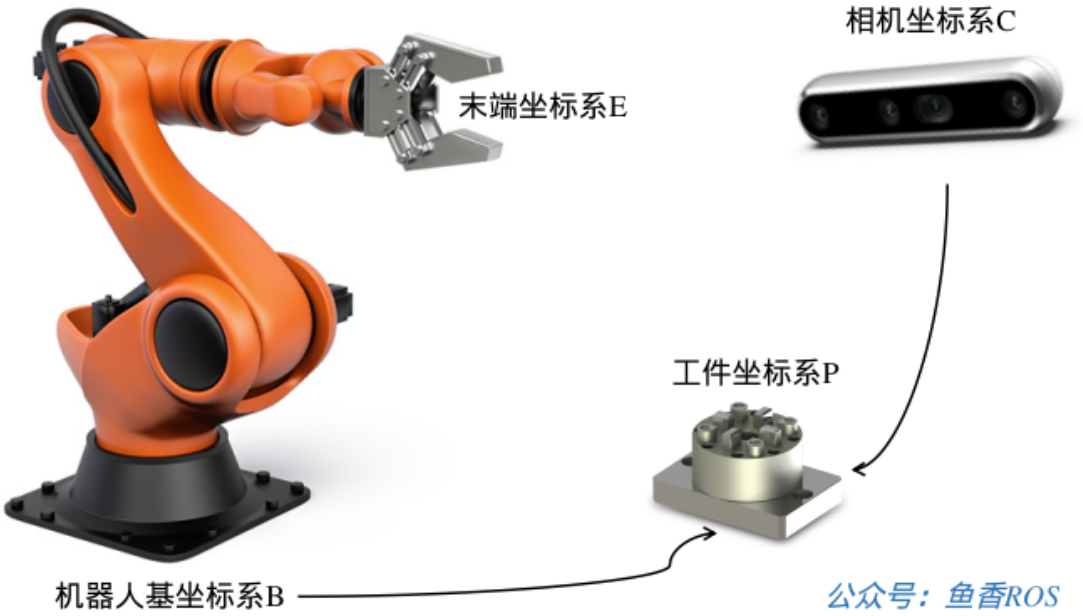
- 含义3：两个物体之间的关系，我们通常把坐标系固定在物体上，这样就可以表示两个物体之间的位姿关系，比如自行车前轮和后轮的关系

小鱼说：学会了位置和姿态描述，三维空间坐标关系描述相信已经难不倒你了

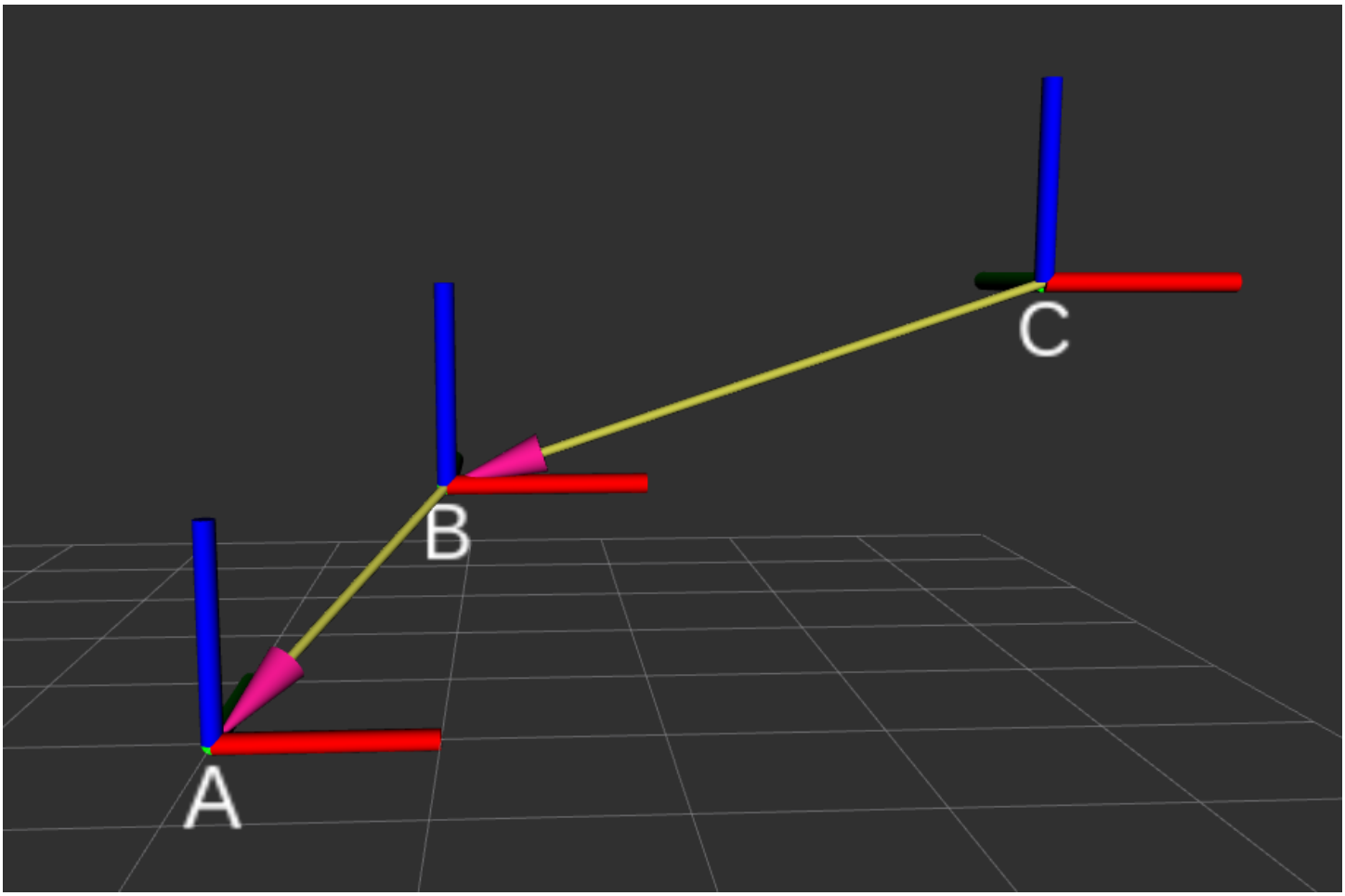
5.坐标变换

位姿是相对的，同一个物体在不同的参考坐标系下的位姿数据肯定是不同的。在后续的学习和使用当中，我们会经常需要求同一个点在不同坐标系的位姿表示，这就要求我们掌握坐标变换的方法了。

比如在手眼系统中，我们可以通过视觉算法获取到 `工件坐标系P` 鱼 `相机坐标系C` 之间的关系，我们想要控制 `机械臂的末端` 运动到 `工件坐标系P` 进行夹取，那么我们就要知道 `工件坐标系P` 在 `机器人基坐标系B` 下的位姿，如何获取呢？这就需要进行坐标变换了。



5.1 平移坐标变换



如图，坐标系{A}、{B}、{C}的姿态是相同的，其之间的姿态对应的旋转矩阵都是单位矩阵。在我们已知：

- {A}为参考坐标系，{B}的位置矢量${^A_B}P=[1,1,1]^T$
- {B}为参考坐标系，{C}的位置矢量${^B_C}P=[3,0,1]^T$

求：

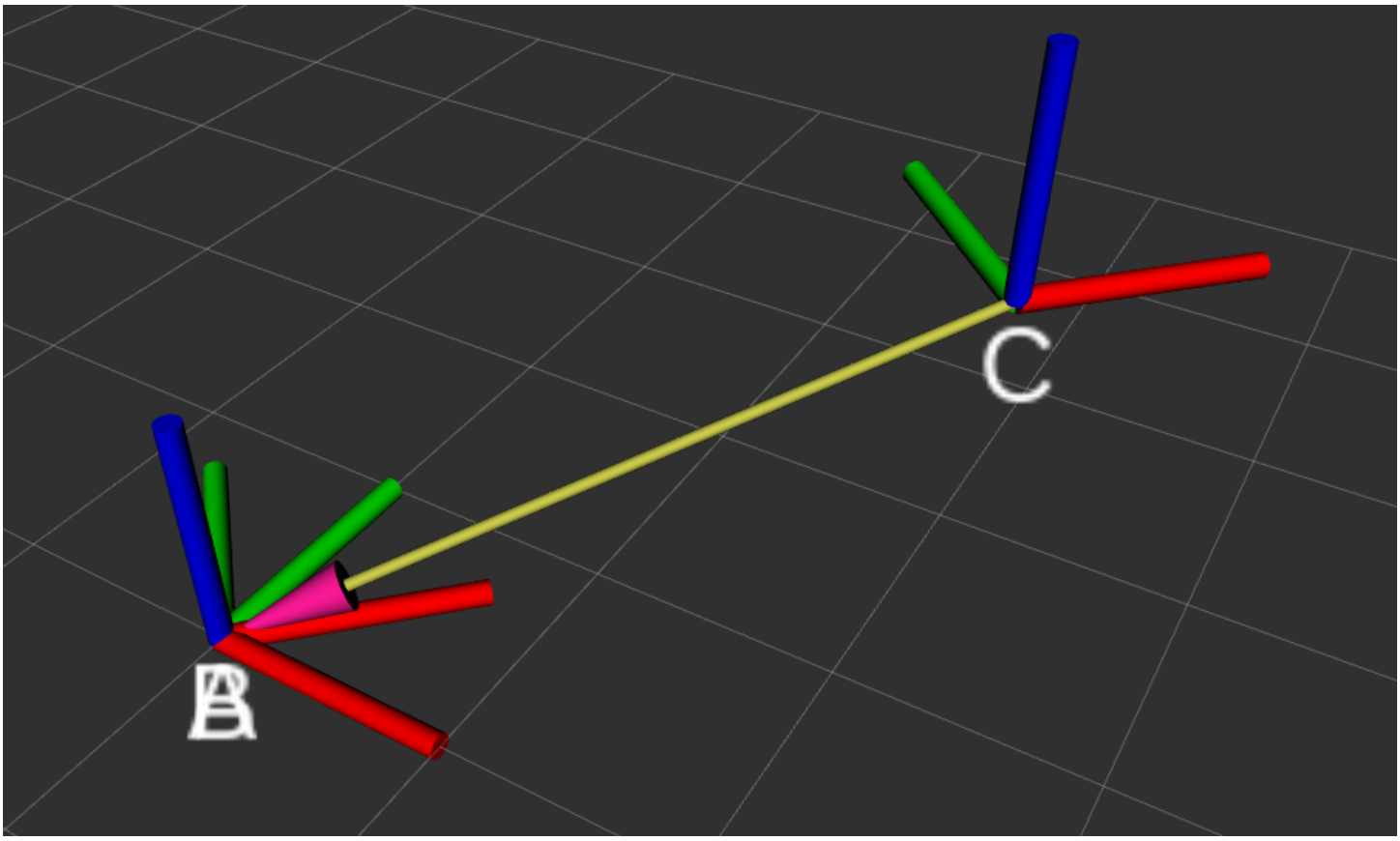
- {A}为参考坐标系，{C}的位置矢量${^A_C}P$

大家可以利用简单的几何知识手算一下，再继续往下看

正确的答案应该是： $$ {^A_C}P=[4,1,2]^T $$ 算法也很简单，直接将${^A_B}P$鱼${^B_C}P$相加即可 $$ {^A_C}P ={^A_B}P+{^B_C}P=[1,1,1]^T+[3,0,1]^T=[4,1,2]^T $$ 由此可知，我们通过平移坐标变换可以求出同一个点（{P}）在相同姿态不同位置坐标系({A}鱼{B})下的不同表示，也可以得到坐标的平移方程 $$ {^A_C}P = {^A_B}P+{^B_C}P \tag{坐标平移方程} $$ 小鱼要提醒大家注意：在上述例子中，{A},{B},{C}三个坐标系的姿态都是相同的，所以${^A_C}R={^A_B}R={^B_C}R$，即都是单位旋转矩阵

5.2 旋转坐标变换

如果坐标系之间姿态不同，同一个点的位姿在不同参考姿态下的位置和姿态也会不一样。



上图中，坐标系{A}鱼坐标系{B}原点重合（连名字都连一块了），我们已知：

- {B}坐标系绕{A}的z轴即A_z旋转了45°
- {B}鱼{C}姿态相同，{B}为参考坐标系，{C}的位置矢量为${^B_C}P=[3,0,1]^T$

求

- {A}为参考坐标系，{C}的位置矢量${^A_C}P$
- {A}为参考坐标系，{C}的旋转矩阵${^A_C}R$

这次相对来说困难一些，不过我们依然可以手推出来,跟着小鱼的思路一起往下走

5.2.1 求${^A_C}R$

因为{B}鱼{C}姿态相同，所以{A}参考坐标系下{C}的姿态和{A}参考坐标系下{B}的姿态是相同的，即： $$ {^A_B}R={^A_C}R $$ 接着只需要求${^A_B}R$即可，因为{B}坐标系绕{A}的z轴即A_z旋转了45°,我们可以一一计算旋转矩阵各个元素的值 $$ {^A_B}R ={^A_C}R = \begin{bmatrix} {B_{x}\cdot A_x} & {B_{y}\cdot A_x} & {B_{z}\cdot A_x}\ {B_{x}\cdot A_y} & {B_{y}\cdot A_y} & {B_{z}\cdot A_y}\ {B_{x}\cdot A_z} & {B_{y}\cdot A_z} & {B_{z}\cdot A_z}\ \end{bmatrix} = \begin{bmatrix} {cos(45)} & cos(90+45) & cos(90)\ {cos(90-45)} & cos(45) & cos(90)\ {cos(90)} & cos(90) & cos(0)\ \end{bmatrix} = \begin{bmatrix} {cos(45)} & cos(90+45) & 0\ {cos(90-45)} & cos(45) & 0\ {0} &0 &1\ \end{bmatrix} $$ 将结果算出来，再利用工具来验证一下，看看最终结果对不对

Rotation matrix

```
[  0.7071068, -0.7071068,  0.0000000;
   0.7071068,  0.7071068,  0.0000000;
   0.0000000,  0.0000000,  1.0000000 ]
```

到此我们求出了${^A_C}P$鱼${^A_C}R$，但聪明的鱼粉肯定不会就此罢休，我们接着来举一反三

{B}绕着{A}的z轴旋转了45度我们计算出来了，那如果{B}绕着{A}旋转θ角度该如何呢?相信聪明的鱼粉心中已经有了答案 $$ R(z,\theta)= \begin{bmatrix} {cos\theta} & -sin\theta & 0\ {sin\theta} & cos\theta & 0\ {0} &0 &1\ \end{bmatrix} \tag{小鱼提示：重要方程1} $$ 那如果不是绕着z轴旋转，而是绕着x轴呢？ $$ R(x,\theta)= \begin{bmatrix} 1 & 0 & 0\ 0 & {cos\theta} & -sin\theta \ 0&{sin\theta} & cos\theta \

\end{bmatrix} \tag{小鱼提示：重要方程2} $$ 绕着y轴呢？ $$ R(y,\theta)= \begin{bmatrix} {cos\theta} & 0 & {sin\theta}\ 0 &1 &0\ {-sin\theta} & 0 &cos\theta \

\end{bmatrix} \tag{小鱼提示：重要方程3} $$

5.2.2 求^A_CP

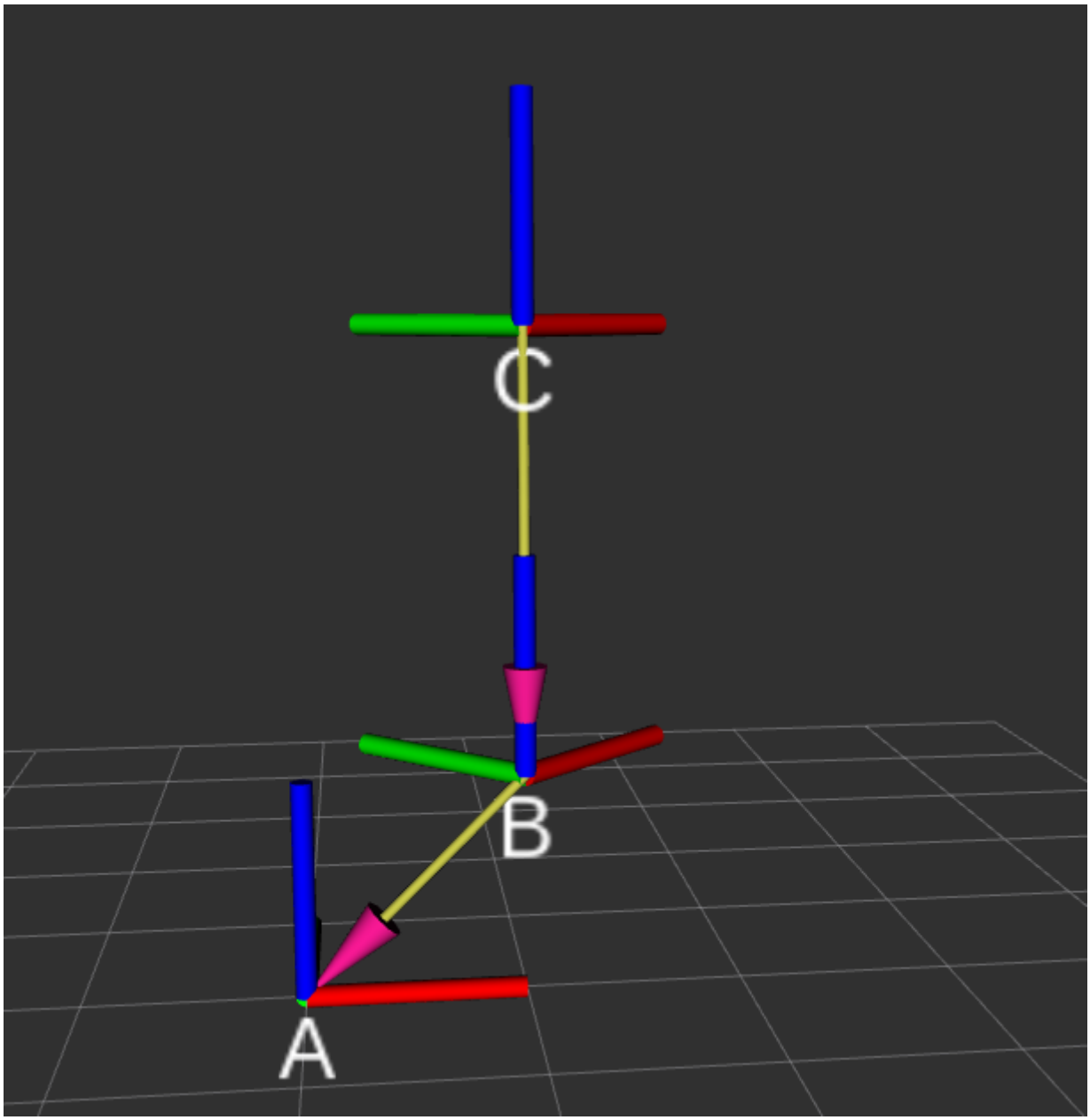
现在我们已知^B_CP和^A_BR,那如何求^A_CP呢，我们可以使用下面这个方程 $$ ^A_CP = {^A_BR}{^B_CP}\tag{坐标旋转方程} $$ 关于这个方程，其实是将^B_CP在{B}上各轴的分量变换到了{A}的各轴上

${^A_BR}$是一个$3*3$的矩阵，${^B_CP}$是$3*1$的矩阵，其相乘结果还是一个$3*1$的矩阵，利用前几节学习的矩阵乘法手算或者使用numpy进行计算，得到最终结果为： $$ ^A_CP = {^A_BR}{^B_CP}\ = [2.12,2.12,1]^T $$

> **小思考:{C}的参考坐标系从{B}变成了{A},{C}在空间中的位置发生了变化了吗？**

5.3平移旋转复合变换

在我们平时搞机（器人）时，一般情况下两个坐标系原点不重合姿态也不相同。我们将坐标变换拆分成先绕参考坐标系旋转，再绕参考坐标系平移两步，这样我们就得到了坐标的复合变换方程 $$ ^A_CP = {^A_BR}{^B_CP}+^A_BP $$ 这个公式其实也很好理解，${^A_BR}{^B_CP}$就是将{C}在{B}上的各轴分量转换到{A}上，再和原来在{A}上各轴的分量再相加。不理解的同学也没事，后面我们使用最多的还是齐次变换矩阵，根据方程直接撸代码就行了。

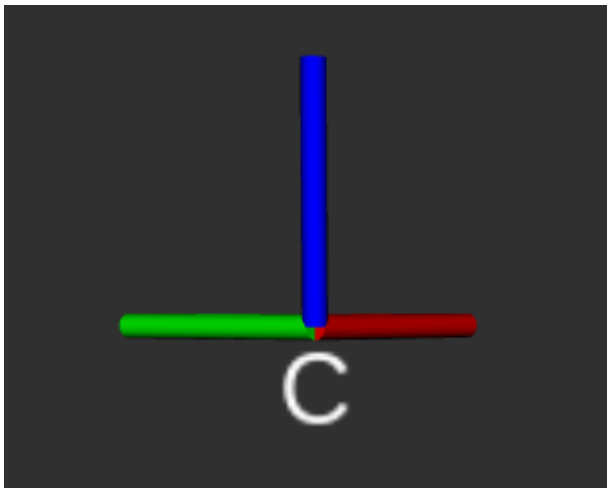


6.左手还是右手

最后小鱼要说的是左手坐标系和右手坐标系，一般情况下我们都是使用右手坐标系搞机。

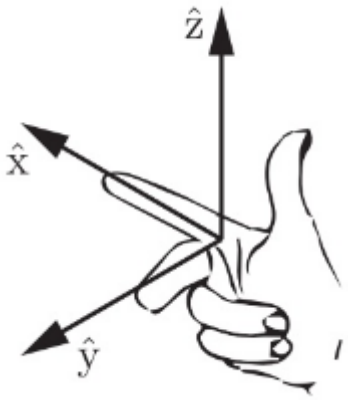
用右手确定的坐标系就是右手坐标系了，跟着小鱼一起做：

拿起你的右手，先给自己竖个大拇指，然后打开手掌，将大拇指的方向朝向下图中的蓝色z轴，让剩下的四根手指朝向红色的x轴，此时朝向手心外的就是绿绿的y轴的方向了。



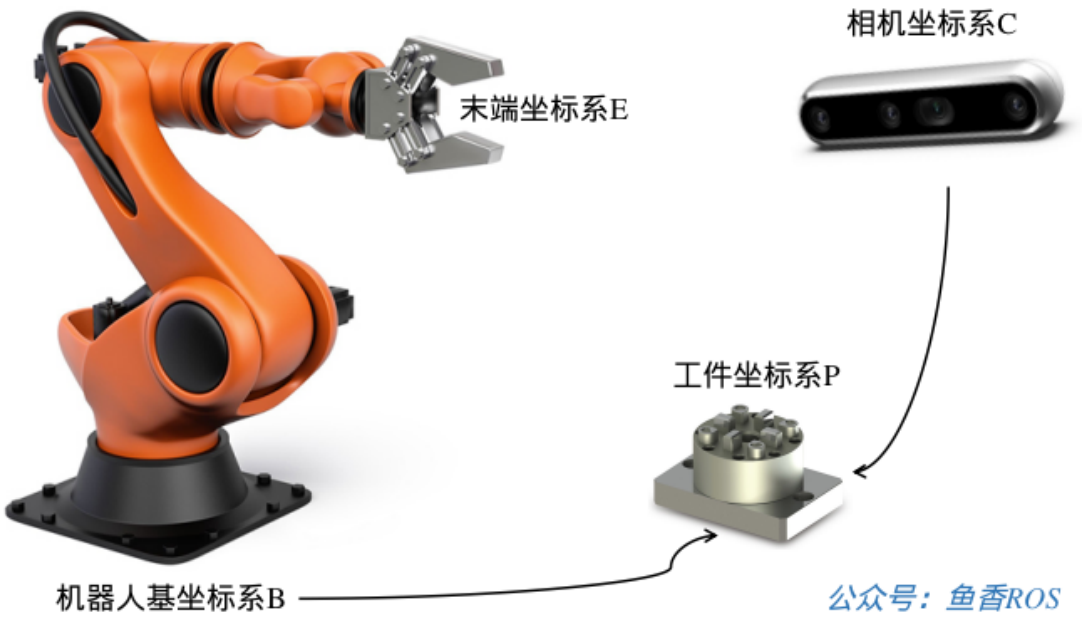
如果你换左手，做上述动作，你会发现y轴是朝手心里的。

再放张图给肢体不协调的同学：



7.练习

光说不练假把式，我们来做个手眼转换题

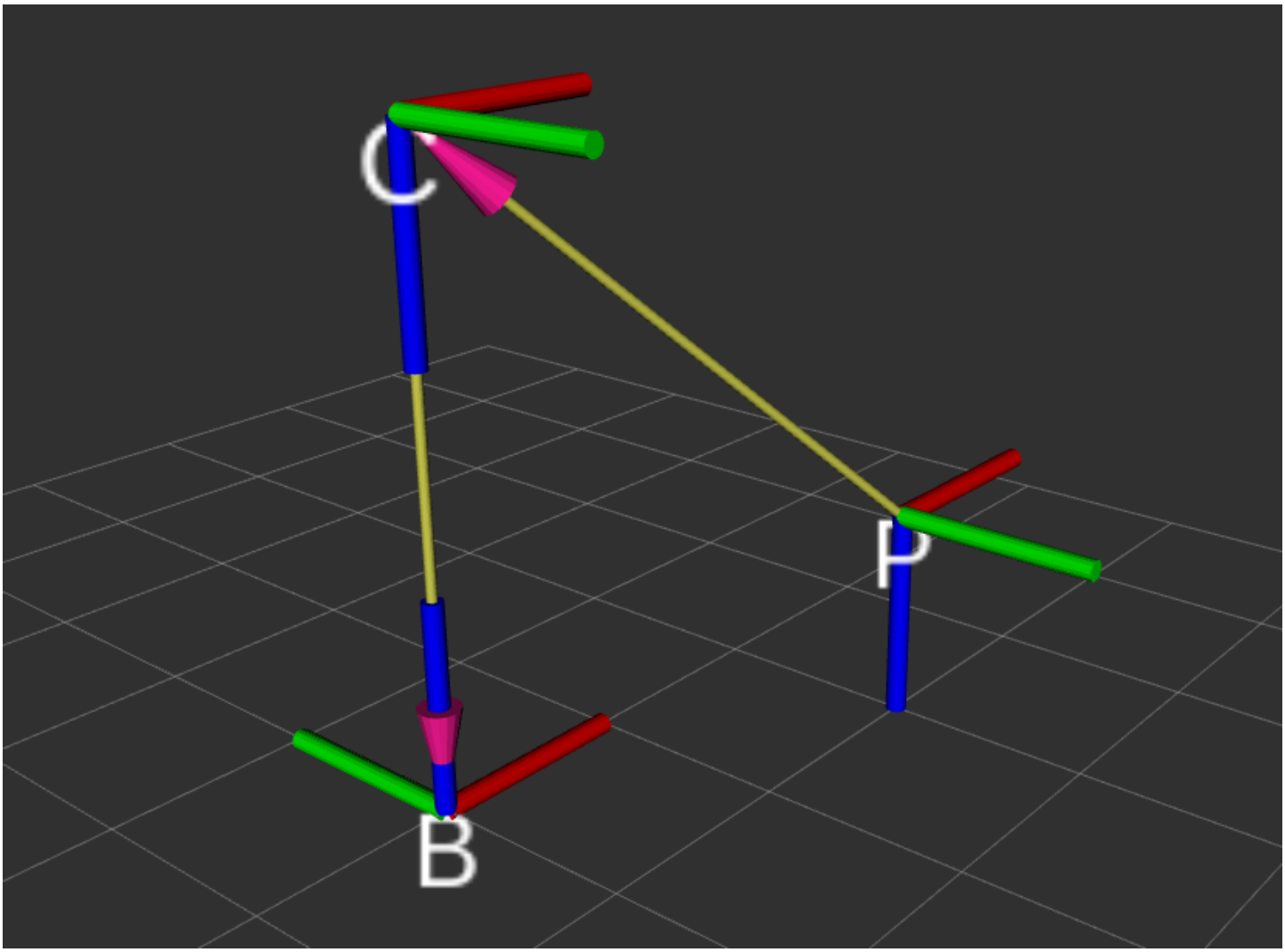


如图🔓示，已知：

1.相机坐标系{C}为参考坐标系，工具坐标系{P}的位置矢量在相机坐标系{C}的 `x,y,z` 各轴投影为$2,1,2$，并且工具坐标系和相机坐标系姿态相同。

2.机器人基坐标系{B}为参考坐标系，相机坐标系{C}的位置矢量在{B}各轴的投影为$0,0,3$，坐标系{C}绕着坐标系{B}的x轴转了180度

可以参考下图看题目



求：

{B}为参考坐标系，坐标系{P}的位置矢量和旋转矩阵

答案：

位置矢量： $$ [2,-1,1]^T $$

旋转矩阵： $$ \begin{bmatrix} 1 & 0 & 0\ 0 & -1 & 0\ 0 &0 & -1\ \end{bmatrix} $$

参考文档

- 机器人学导论
- 机器人学基础

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**

- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

2.空间坐标描述实战

大家好，我是小鱼，上一节我们学习了坐标描述和坐标变换的理论知识，本节课我们把重点放到动手实现上，通过numpy实现坐标的描述和变换，同时结合ROS2鱼RVIZ进行坐标关系可视化与求解。

通过本节你将掌握：

- 使用numpy表示位置矢量和旋转矩阵
- 使用numpy进行平移与旋转坐标变换
- 了解ROS2中TF2的概念
- 使用tf2相关CLI工具进行坐标变换
- 使用python操作TF进行坐标变换

1.numpy表示位姿

在前几节中，小鱼带你安装使用了ROS2和MiniConda，并学习使用Numpy进行矩阵相关运算的验证。

那我们如何使用numpy中的矩阵表示位置矢量和旋转矩阵呢？

1.1 位置表示

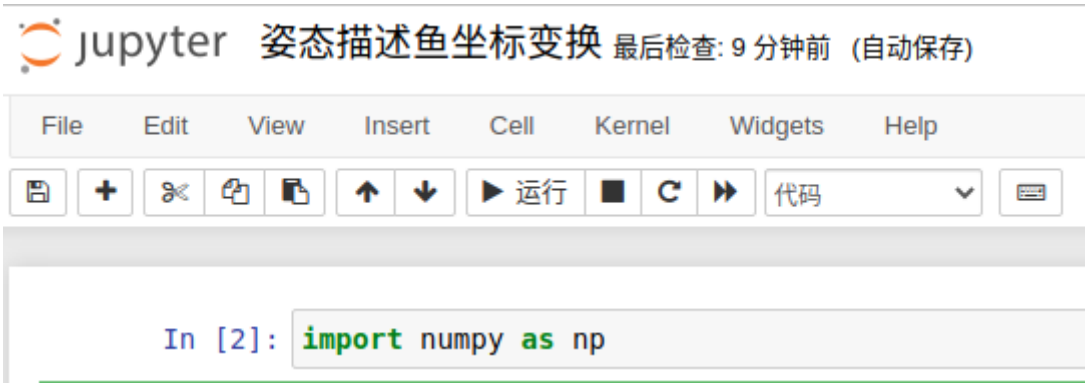
上一节中我们使用一个3行1列的矩阵表示了位置，那么我们在numpy中自然也可以使用这样一个矩阵表示。

打开终端，输入下面指令打开jupyter

```shell
jupyter-notebook
```

新建一个python3的代码并重命名，接着在单元格中导入numpy

```
import numpy as np
```



1.1.1 位置矩阵

使用3*1的矩阵表示位置，我们新建一个沿着x、y、z各平移1个单位的位置矢量。

```
np.asarray([1.0,1.0,1.0]).reshape(3,1)
```

```
In [4]: np.asarray([1.0,1.0,1.0]).reshape(3,1)
Out[4]: array([[1.],
               [1.],
               [1.]])
```

1.2 姿态表示

姿态可以使用 `3*3` 的旋转矩阵表示, `3*3` 的单位矩阵代表没有姿态变换，注意没有姿态变换不是零矩阵，而是单位矩阵。

我们新建一个旋转矩阵，用该旋转矩阵表示绕着z轴旋转45度，可以这样写：

```
import math
import numpy as np
theta = math.radians(45)
R_AB = np.asarray([math.cos(theta),-math.sin(theta),0,
                   math.sin(theta),math.cos(theta),0,
                   0,0,1
                  ]).reshape(3,3)
print(R_AB)
```

运行后得到的旋转矩阵是不是和上节的一样

```
In [5]: import math
        import numpy as np
```

```
In [10]: theta = math.radians(45)
         R_AB = np.asarray([math.cos(theta),-math.sin(theta),0,
                            math.sin(theta),math.cos(theta),0,
                            0,0,1
                           ]).reshape(3,3)
         print(R_AB)
```

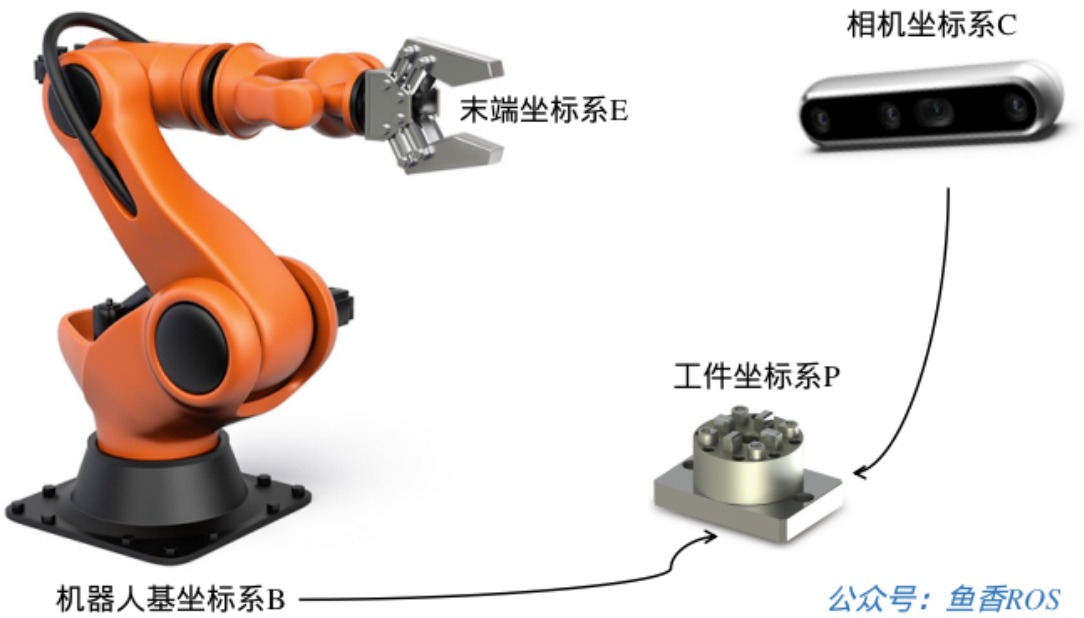
```
[[ 0.70710678 -0.70710678  0.        ]
 [ 0.70710678  0.70710678  0.        ]
 [ 0.          0.          1.        ]]
```

2.numpy坐标变换

掌握了使用numpy表示位置和姿态后，接着我们使用numpy来完成上一节的小练习

2.1 题目

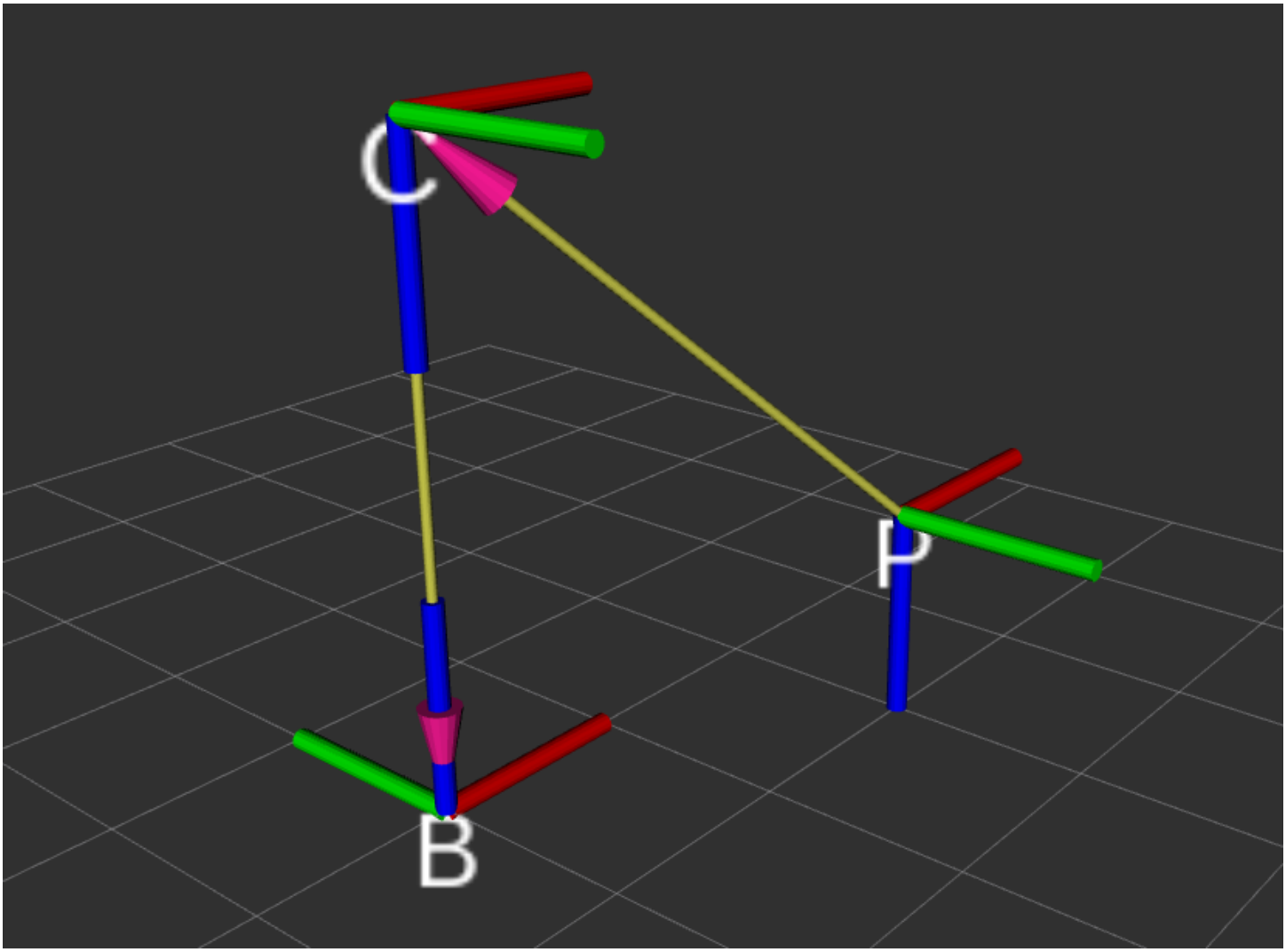
如图🔓示，已知：



1.相机坐标系{C}为参考坐标系，工具坐标系{P}的位置矢量在相机坐标系{C}的 `x,y,z` 各轴投影为$2,1,2$，并且工具坐标系和相机坐标系姿态相同。

2.机器人基坐标系{B}为参考坐标系，相机坐标系{C}的位置矢量在{B}各轴的投影为$0,0,3$，坐标系{C}绕着坐标系{B}的x轴转了180度

可以参考下图看题目



求：

{B}为参考坐标系，坐标系{P}的位置矢量和旋转矩阵

2.2 使用numpy求解

2.2.1 旋转矩阵求解

这里我们就需要使用复合坐标变换了，根据坐标变换规则有： $$ ^B_PR=^B_CR^C_PR $$ {C}和{P}姿态相同，所以^C_PR是一个单位矩阵。又因为{C}绕着{B}的x旋转了180度，根据上节的重要公式2可知 $$ R(x,\theta)= \begin{bmatrix} 1 & 0 & 0\ 0 & {cos\theta} & -sin\theta \ 0&{sin\theta} & cos\theta \

\end{bmatrix} \tag{小鱼提示：重要方程2} $$ 所以^B_CR对应的程序可以这样写

```python
import math
import numpy as np
theta = math.radians(180)
R_BC = np.asarray([1,0,0,
                   0,math.cos(theta),-math.sin(theta),
                   0,math.sin(theta),math.cos(theta)]).reshape(3,3)
```

所以^B_PR可以用程序求得：

```
R_BP = R_BC*np.identity(3)
print("旋转矩阵R_BP：\n",R_BP)
```

```
In [8]: import math
        import numpy as np
        theta = math.radians(180)
        R_BC = np.asarray([1,0,0,
                           0,math.cos(theta),-math.sin(theta),
                           0,math.sin(theta),math.cos(theta)]).reshape(3,3)
        R_BP = R_BC*np.identity(3)
        print("旋转矩阵R_BP: \n",R_BP)

        旋转矩阵R_BP:
         [[ 1.  0.  0.]
         [ 0. -1. -0.]
         [ 0.  0. -1.]]
```

2.2.2 平移矩阵求解

根据复合变换规则可知：$$ ^B_PP=^B_CR^C_PP+^B_CP $$ 根据描述有：$^B_CP=[0,0,3]^T$、$^C_PP=[2,1,2]^T$

所以可以写这样写程序：

```python
P_BC = np.asarray([0,0,3]).reshape(3,1)
P_CP = np.asarray([2,1,2]).reshape(3,1)
P_BP = np.add(np.dot(R_BC,P_CP),P_BC)
print("位置矢量P_BP:\n",P_BP.T)
```

运行下，可以得到结果

```
In [25]: P_BC = np.asarray([0,0,3]).reshape(3,1)
         P_CP = np.asarray([2,1,2]).reshape(3,1)
         P_BP = np.add(np.dot(R_BC,P_CP),P_BC)
         print("位置矢量P_BP:\n",P_BP.T)

         位置矢量P_BP:
          [[ 2. -1.  1.]]
```

2.3 结果对比

与上节课答案一致

位置矢量：$[2,-1,1]^T$

旋转矩阵： $$\begin{bmatrix} 1 & 0 & 0\ 0 & -1 & 0\ 0 &0 & -1\ \end{bmatrix} \tag{小鱼提示：重要方程1}$$

以上大家跟着小鱼一起学习了使用numpy表示位置和姿态，并使用了numpy来进行坐标变换，接下来我们直接使用ROS2中的TF工具不写一行代码进行坐标发布，同时我们还可以使用RVIZ2直观的查看坐标关系。

3.姿态的不同表示

大家好，我是小鱼，本节课，我们来学习姿态的多种表示方式。在前面的课程中，我们一共接触了三种姿态的表示方式：

1. 旋转矩阵-在位姿描述一节中
2. 坐标轴旋转-绕xyz轴旋转不同的角度(欧拉角)
3. 四元数-ROS2的TF2中的姿态描述

本节小鱼将对以上的三种姿态描述进行归类与介绍，并对他们之间的转换方法进行讲解，下一节小鱼带你一起通过代码直观的观察和操作姿态变换。

小鱼将常用的坐标描述分为三类，共五种。这五种也是小鱼在平时工作中所接触到的几乎所有姿态描述方法，三类共五种方法如下：

- 旋转矩阵-`旋转矩阵`
- 坐标轴旋转-`固定轴欧拉角`,`非固定轴欧拉角`
- 任意轴旋转-`等效轴角`,`四元数`

常用的坐标转换包括：

- `固定角`与`四元数`互转
- `固定角`与`旋转矩阵`互转
- `四元数`与`旋转矩阵`互转

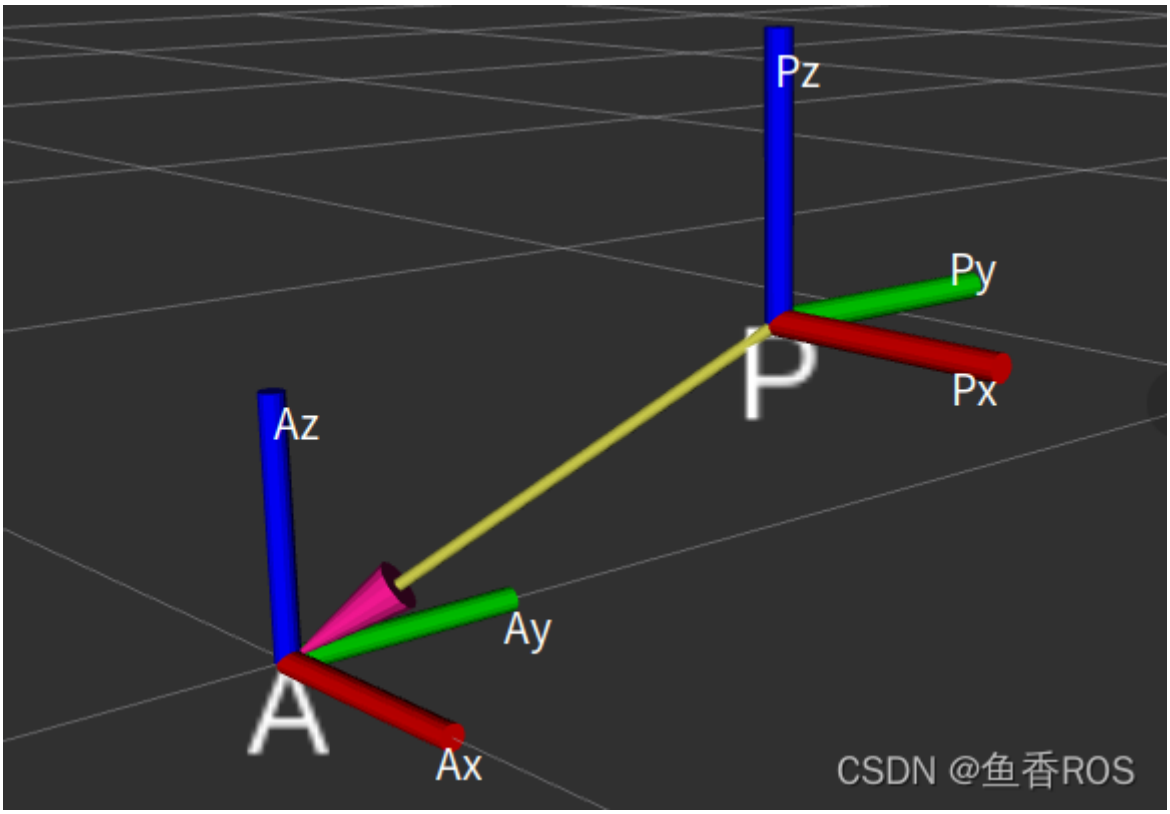
1.旋转矩阵

关于旋转矩阵我们在前几节教程中已经介绍了，旋转矩阵采用的是`旋转后的坐标系`三个轴分别与`原坐标系`三个轴的夹角余弦值共九个数字组成的3*3矩阵。

旋转矩阵一般记作R

> **若两个坐标系姿态相同，其旋转矩阵为单位矩阵。**

1.1 旋转矩阵的描述



如图，描述 坐标系{P} 和 参考坐标系{A} 之间的姿态关系的旋转矩阵用符号${^A_P}R$来表示。

$$ {^A_P}R=[{^A}x_{P} \ {^A}y_{P} \ {^A}z_{P}] = \begin{bmatrix}{r_{11}}&{r_{12}}&{r_{13}}\{r_{21}}&{r_{22}}&{r_{23}}\{r_{31}}&{r_{32}}&{r_{33}}\\end{bmatrix} \tag{旋转矩阵} $$

两个向量的点乘=两个向量的长度(1)与它们夹角余弦的积，所以$r11$可以表示为向量P_{x}与A_{x}的点积，旋转矩阵就可以写为下面的形式

$$ {^A_P}R = \begin{bmatrix} {P_{x}\cdot A_x} & {P_{y}\cdot A_x} & {P_{z}\cdot A_x}\ {P_{x}\cdot A_y} & {P_{y}\cdot A_y} & {P_{z}\cdot A_y}\ {P_{x}\cdot A_z} & {P_{y}\cdot A_z} & {P_{z}\cdot A_z}\ \end{bmatrix} $$

1.2 绕某一轴旋转θ角的旋转矩阵

新的坐标系 绕 原坐标系 某一坐标轴旋转任意角度得到的旋转矩阵有如下等式。

绕x轴旋转θ后姿态矩阵

$$ R(x,\theta)= \begin{bmatrix} 1 & 0 & 0\ 0 & {cos\theta} & -sin\theta \ 0&{sin\theta} & cos\theta \ \end{bmatrix} $$

绕y轴旋转θ后姿态矩阵

$$ R(y,\theta)= \begin{bmatrix} {cos\theta} & 0 & {sin\theta}\ 0 &1 &0\ {-sin\theta} & 0 &cos\theta \ \end{bmatrix} $$

绕z轴旋转θ后姿态矩阵

$$ R(z,\theta)= \begin{bmatrix} {cos\theta} & -sin\theta & 0\ {sin\theta} & cos\theta & 0\ {0} &0 &1\ \end{bmatrix} $$

2.欧拉角-绕坐标轴的旋转

2.1 12种旋转顺序

旋转矩阵是一个冗余的（九个值之间存在约束关系），可以只需要三个参数来表示的矩阵。聪明的鱼粉肯定会想到，假如知道坐标系绕分别绕X、Y、Z轴的旋转角度，不就同样可以表示旋转了吗？

这个猜想是对的，结合1.2中绕三个轴旋转的三个θ，按照特定的顺序将对应的旋转矩阵乘起来就可以确定一个旋转矩阵。

但需要注意的是，矩阵的乘法不具备交换性，所以旋转顺序不同会造成不同的结果。

比方说若是先绕自身x轴旋转α，再绕自身y轴旋转β：

$$ R(x,\alpha)R(y,\beta)= \begin{bmatrix} 1 & 0 & 0\ 0 & {cos\alpha} & -sin\alpha \ 0&{sin\alpha} & cos\alpha \ \end{bmatrix} \begin{bmatrix} {cos\beta} & 0 & {sin\beta}\ 0 &1 &0\ {-sin\beta} & 0 &cos\beta \end{bmatrix}= \begin{bmatrix} {cos\beta} & 0 & {sin\beta}\ sin\alpha sin\beta &cos\alpha & -sin\alpha cos\beta\ {-cos\alpha sin\beta} & sin\alpha &cos\alpha cos\beta \ \end{bmatrix} $$

比方说若是，先绕自身y轴旋转β，绕自身x轴旋转α：

$$ R(y,\beta)R(x,\alpha)= \begin{bmatrix} {cos\beta} & 0 & {sin\beta}\ 0 &1 &0\ {-sin\beta} & 0 &cos\beta \ \end{bmatrix} \begin{bmatrix} 1 & 0 & 0\ 0 & {cos\alpha} & -sin\alpha \ 0&{sin\alpha} & cos\alpha \ \end{bmatrix}= \begin{bmatrix} cos\beta & sin\alpha sin\beta & {sin\beta}\ 0 & {cos\alpha} & -sin\alpha \ {-sin\beta} & sin\alpha &cos\alpha sin\beta \ \end{bmatrix} $$

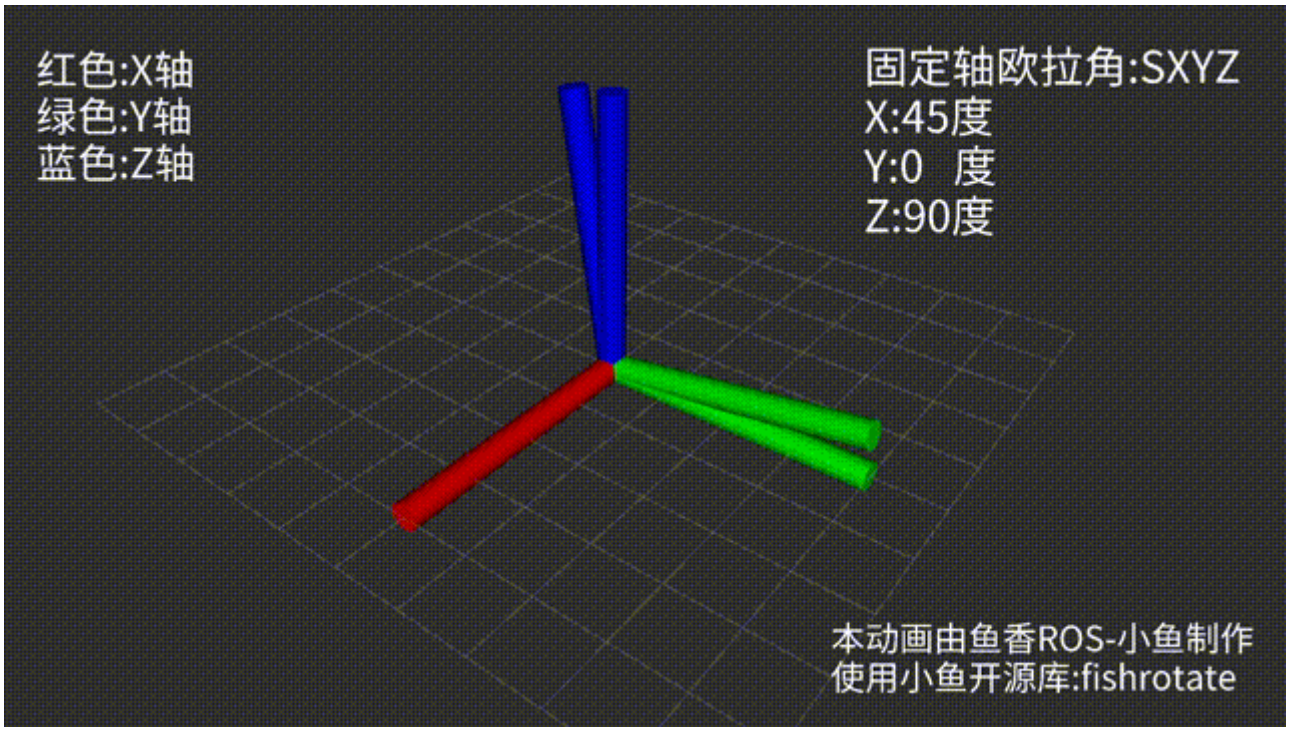
所以我们对旋转顺序做排列组合，可以得到12种旋转顺序:

- `xyz`, `xyx`, `xzy`
- `xzx`, `yzx`, `yzy`
- `yxz`, `yxy`, `zxy`
- `zxz`, `zyx`, `zyz`

2.2 两种参考坐标系

除了要考虑旋转时所绕轴的顺序，还要考虑参考坐标系（坐标轴）的不同。

2.2.1 参考固定的坐标系

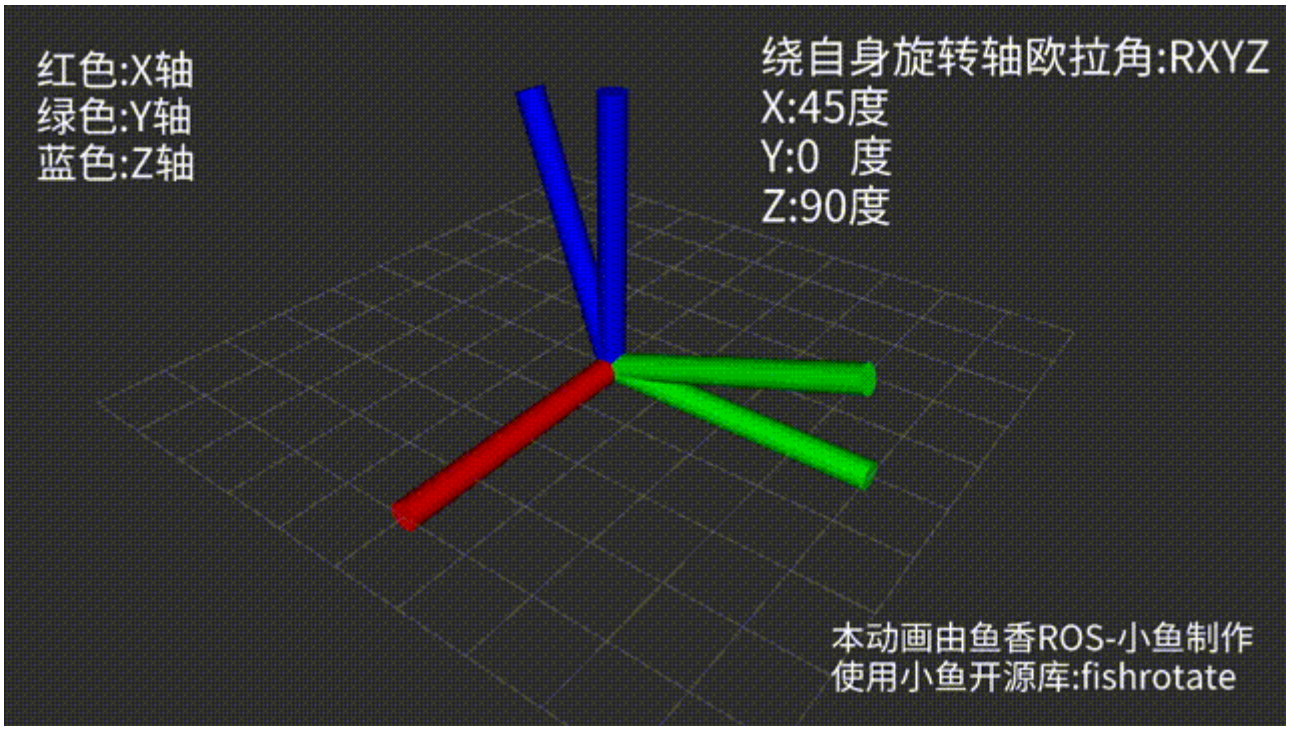


假设坐标系B与坐标系A初始姿态相同

- 坐标系{B}绕坐标系A的x轴 `Ax` 旋转α
- 接着坐标系{B}绕着A的y轴 `Ay` 旋转β
- 接着绕 `Az` 旋转γ

上述三次旋转，都是以A坐标系的xyz轴为参考坐标系进行旋转，该旋转方式为固定旋转轴的旋转，通常称之为固定角欧拉角或固定轴旋转。

2.2.2 参考自身坐标系



我们也可以不沿着坐标系A的各轴旋转，而是绕旋转之后B的某一轴再次旋转，我们称之为非固定旋转轴的欧拉角。

> 小鱼说：无论是参考自身坐标系还是参考固定的坐标系，都有12种旋转方式，所以欧拉角有12*2=24种旋转方式，后面的计算中我们也将直观的感受到24种旋转方式的不同。

2.3 固定转轴欧拉角 转 旋转矩阵

首先我们来考虑绕固定的坐标系旋转如何转换成旋转矩阵

我们以XYZ的旋转顺序来举例说明，其他旋转顺序类似

现在假设A、B两个坐标系重合，B坐标系绕A坐标系的X轴旋转45度，绕A的Z轴旋转90度.

求旋转之后A为参考坐标系，B坐标系的姿态$^A_BR_{XYZ(45,0,90)}$

小鱼先告诉你最终的结果： $$
^A_BR_{XYZ(45,0,90)}=R_{Z(90)}R_{Y(0)}R_{X(45)}=R_{Z(90)}R_{X(45)}=\begin{bmatrix} 0.&-0.70710678 &
0.70710678\ -1&0&0\ 0&0.70710678&0.70710678\ \end{bmatrix} $$

为什么结果是将绕Z轴的旋转矩阵乘绕X轴的旋转矩阵呢？

这里引用林沛群老师的解释：

我们可以假设一个向量v固定在B坐标系上，那我们让B坐标系绕着A坐标系的三个轴做旋转，就可以认为是让向量v绕着坐标系A的三个轴做旋转，那先转的肯定先乘，所以我们先让向量v乘上Rx(45)，再让其乘上Rz(90)，

即： $$ v' = R_{Z(90)}(R_{X(45)}v) $$ 根据矩阵乘法的结合律，括号可以去掉： $$ v' = R_{Z(90)}R_{X(45)}v $$

所以我们可以得到，绕固定轴XYZ旋转的欧拉角转旋转矩阵方法: $$ R_{XYZ(\gamma,\beta,\alpha)}=R_{Z(\alpha)}R_{Y(\beta)}R_{X(\gamma)} $$ 最终结果：

$$\boldsymbol{R}_{XYZ}(\gamma,\beta,\alpha)=\begin{bmatrix} c\alpha c\beta & c\alpha s\beta s\gamma-s\alpha c\gamma & c\alpha s\beta c\gamma+s\alpha s\gamma \\ s\alpha c\beta & s\alpha s\beta s\gamma+c\alpha c\gamma & s\alpha s\beta c\gamma-c\alpha s\gamma \\ -s\beta & c\beta s\gamma & c\beta c\gamma \end{bmatrix}$$

根据旋转顺序不同，固定角有12种旋转方式，这里我们给出了绕固定轴以XYZ顺序旋转欧拉角的转旋转矩阵的等式，其他旋转顺序对应的旋转矩阵可以尝试自行推导。

2.2 非固定旋转轴的欧拉角

非固定旋转轴，即每次旋转是绕着自身的坐标轴进行旋转，其旋转动图如2.2.2节所示。

非固定旋转的欧拉角转旋转矩阵推导也很简单，我们以旋转顺序ZYX为例子分析

2.2.1 ZYX

因为每次旋转都是绕着自身进行的，我们可以将每次的旋转进行拆解

$$ ^A_BR={^A_{B'}R}{^{B'}_{B''}R}{^{B''}_{B}R} $$

等式右边的三次旋转按照顺序Z-Y-X进行的，所以最终B坐标系在A坐标系下的姿态为：

$$ ^A_BR_{Z'Y'X'}=R_{Z(\alpha)}R_{Y(\beta)}R_{X(\gamma)} $$

最终结果太难敲，小鱼直接截图啦

$$\boldsymbol{R}_{Z'Y'X'}(\alpha,\beta,\gamma)=\begin{bmatrix} c\alpha c\beta & c\alpha s\beta s\gamma-s\alpha c\gamma & c\alpha s\beta c\gamma+s\alpha s\gamma \\ s\alpha c\beta & -s\alpha s\beta s\gamma+c\alpha c\gamma & -s\alpha s\beta c\gamma-c\alpha s\gamma \\ -s\beta & c\beta s\gamma & c\beta c\gamma \end{bmatrix}$$

> **旋转矩阵转欧拉角的方法需要使用双参变量的反正切函数，我们后面在程序当中直接调用对应函数即可实现，这里对原理就不再进行推导了**

3.轴角

在介绍四元数之前，我们先来说说等效角度轴线，这种表示姿态的方式。

上一节欧拉角中无论是绕着自身的某个轴旋转，还是绕着固定的坐标系的某个轴进行旋转，旋转时参考的轴都是坐标系的主轴

假如我们参考的轴不是主轴，那么任何姿态都可以通过选择适当的轴和角度得到，换句话说，两个坐标系之间的任何姿态都可以通过绕某一个特定的轴(矢量)旋转特定的角度得到。

说到这里相信你已经理解了轴角的意义，接着我们给出轴角和旋转矩阵之间的转换关系

轴角转旋转矩阵

假设坐标系B和参考坐标系A重合，将B绕着A坐标系下的矢量^AK按右手定则旋转θ角度，旋转之后B坐标系在A坐标系下的姿态可以用 $$ ^A_BR(K,\theta) $$ 表示，注意矢量K为单位矢量(模长为1)，K为一个3*1的矢量 $$ ^AK=[k_x,k_y,k_z]^T $$

在已知矢量K和θ的情况下，我们如何得到旋转矩阵呢?

有等式：

$$\boldsymbol{R}_K(\theta) = \begin{bmatrix} k_x k_x v\theta + c\theta & k_x k_y v\theta - k_z s\theta & k_x k_z v\theta + k_y s\theta \\ k_x k_y v\theta + k_z s\theta & k_y k_y v\theta + c\theta & k_y k_z v\theta - k_x s\theta \\ k_x k_z v\theta - k_y s\theta & k_y k_z v\theta + k_x s\theta & k_z k_z v\theta + c\theta \end{bmatrix}$$

其中

$$ c\theta=cos\theta \ s\theta=sin\theta \ v\theta=1-cos\theta $$

θ的符号由右手定则确定，右手大拇指指向矢量K的方向.

旋转矩阵转轴角需要根据情况讨论，该部分转换我们直接调用相应函数实现，这里对其原理不再叙述，感兴趣的同学可以参考：https://en.wikipedia.org/wiki/Rotation_matrix#Axis_of_a_rotation

4.四元数

除了轴角可以使用一个数字表示角度，三个数字表示旋转轴，一共四个数字表示旋转外。还有另外一种四个数字表示表示旋转的方式——四元数。

四元数的四个数字由一个实部和三个虚部组成，是一个超复数形式

$$ q = w + xi + yj + z*k $$

关于四元数的由来有个小故事，小鱼分享一下：

> 1843年10月16日的傍晚，英国数学家哈密顿和他的妻子一起步行去都柏林，途中经过布鲁哈姆桥时，他的脚步突然放慢了。妻子以为他要尽情欣赏周围的景色，于是也放慢了脚步。其实哈密顿此时正在思考他久久不能解决的问题。早在1828年，他就想发明一种新的代数，用来描述绕空间一定轴转动并同时进行伸缩的向量的运动。他设想这种新代数应包含四个分量：两个来固定转动轴，一个来规定转动角度，第四个来规定向量的伸缩。但是在构造新代数的过程中，由于他受传统观念的影响，不肯放弃乘法交换律，故屡受挫折。哈密顿盲目地相信，普通代数最重要的规律必定继续存在于他寻找的代数中。然而此刻，他的脑际突然产生了一个闪念：在所寻找的代数中，能否让交换律不成立呢？比方说，A×B不等于B×A而是等于负的B×A。这个想法太大胆了，他感到非常激动。哈密顿马上掏出笔记本，把他的思想火花记录下来。这一火花就是I，J，K之间的基本方程，即四元数乘法基本公式。哈密顿因此把1843年10月16日称为四元数的生日。此后，哈密顿一生的最后22年几乎完全致力于四元数的研究，成果发表在他去世后出版的《四元数基础》一书中。四元数的出现，推倒了传统代数的关卡，故有数学史上里程碑的美誉。后人为了纪念这一发明，特意在当年哈密顿刻划过的石头上镶嵌了一块水泥板，上面清楚地记载着1843年曾经发生的故事。

四元数在机器人中使用的非常多，甚至在量子力学中都有使用，关于四元数旋转的本质，小鱼也学习了很久才搞清楚，B站上3B1B的视频非常经典，大家自行食用。

> 在机器人学当中用到的四元数都是单位四元数（四维单位超球体在三维空间的投影）,下文中提到的四元数默认指单位四元数



接着我们来说说四元数常用的转换

四元数转旋转矩阵

$$R = \begin{bmatrix} 1-2y^2-2z^2 & 2(xy-zw) & 2(xz+yw) \\ 2(xy+zw) & 1-2x^2-2z^2 & 2(yz-xw) \\ 2(xz-yw) & 2(yz+xw) & 1-2x^2-2y^2 \end{bmatrix}$$

旋转矩阵转四元数

$$\begin{aligned} x &= \frac{r_{32}-r_{23}}{4w} \\ y &= \frac{r_{13}-r_{31}}{4w} \\ z &= \frac{r_{21}-r_{12}}{4w} \\ w &= \frac{1}{2}\sqrt{1+r_{11}+r_{22}+r_{33}} \end{aligned}$$

四元数转欧拉角

$$\begin{bmatrix} \varphi \\ \theta \\ \psi \end{bmatrix} = \begin{bmatrix} atan2(2(wx+yz),1-2(x^2+y^2)) \\ arcsin(2(wy-zx)) \\ atan2(2(wz+xy),1-2(y^2+z^2)) \end{bmatrix}$$

欧拉角转四元数

$$\begin{aligned} \mathbf{q}_{\mathrm{IB}} &= \begin{bmatrix} \cos(\psi/2) \\ 0 \\ 0 \\ \sin(\psi/2) \end{bmatrix} \begin{bmatrix} \cos(\theta/2) \\ 0 \\ \sin(\theta/2) \\ 0 \end{bmatrix} \begin{bmatrix} \cos(\phi/2) \\ \sin(\phi/2) \\ 0 \\ 0 \end{bmatrix} \\ &= \begin{bmatrix} \cos(\phi/2)\cos(\theta/2)\cos(\psi/2)+\sin(\phi/2)\sin(\theta/2)\sin(\psi/2) \\ \sin(\phi/2)\cos(\theta/2)\cos(\psi/2)-\cos(\phi/2)\sin(\theta/2)\sin(\psi/2) \\ \cos(\phi/2)\sin(\theta/2)\cos(\psi/2)+\sin(\phi/2)\cos(\theta/2)\sin(\psi/2) \\ \cos(\phi/2)\cos(\theta/2)\sin(\psi/2)-\sin(\phi/2)\sin(\theta/2)\cos(\psi/2) \end{bmatrix} \end{aligned}$$

轴角转四元数

轴:$^AK=[k_x,k_y,k_z]^T$ 角:θ $$ x= k_x\sin(\theta/2)\ y= k_y\sin(\theta/2) \ z= k_z\sin(\theta/2)\ w = \cos(\theta/2) $$

> 小计算: $w^2+x^2+y^2+z^2=1$

4.姿态转换实战

上一节小鱼带你一起学习了四种姿态表示方式，这节课我们就利用相关的开源库，来完成姿态的不同表示方式之间的转换。

包含12中转换形式，其中一些转换的计算方式，前面几节课中小鱼有给出相应的公式。

四元数模块:quaternions

- 四元数 转 旋转矩阵
- 旋转矩阵 转 四元数
- 四元数 转 轴角
- 轴角 转 四元数

欧拉模块:euler

- 欧拉角 转 四元数
- 四元数 转 欧拉角
- 欧拉角 转 旋转矩阵
- 旋转矩阵 转 欧拉角
- 欧拉角 转 轴角
- 轴角 转欧拉角

轴角模块:axangles

- 轴角 转 旋转矩阵
- 旋转矩阵 转 轴角

本节主要推荐的是Python的相关实现，用的是小鱼常用的transforms3d库,该库的api设计非常的巴适，小鱼是非常的爱~

1.安装

安装使用pip即可

```
pip install transforms3d -i https://pypi.tuna.tsinghua.edu.cn/simple
```

2.四元数相关转换

四元数模块在transforms3d.quaternions里，直接导入即可使用

2.1 四元数与旋转矩阵互转

```
import transforms3d as tfs
import numpy as np
# 四元数转旋转矩阵
tfs.quaternions.quat2mat([1,0,0,0])
# 旋转矩阵转四元数
tfs.quaternions.mat2quat(np.asarray([[1., 0., 0.],[0., 1., 0.],[0., 0., 1.]]))
```

2.2 四元数与轴角互转

```
import transforms3d as tfs
import numpy as np
# 四元数转旋轴角
tfs.quaternions.quat2axangle([1,0,0,0])
# 轴角转四元数
tfs.quaternions.axangle2quat([1,0,0],0.5)
```

3.欧拉角相关转换

四元数模块在transforms3d.euler里，直接导入即可使用

3.1 欧拉角与四元数互转

```
import transforms3d as tfs
import numpy as np
# 固定轴欧拉角转四元数
tfs.euler.euler2quat(0,0,0,"sxyz")
```

```
# 四元数转固定轴欧拉角
tfs.euler.quat2euler([1,0,0,0],"sxyz")
```

3.2 欧拉角与旋转矩阵互转

```
import transforms3d as tfs
import numpy as np
# 固定轴欧拉角转旋转矩阵
tfs.euler.euler2mat(0,0,0,"sxyz")
# 旋转矩阵转固定轴欧拉角
tfs.euler.mat2euler(np.asarray([[1., 0., 0.],[0., 1., 0.],[0., 0., 1.]]),"sxyz")
```

3.3 欧拉角与轴角互转

```
import transforms3d as tfs
import numpy as np
# 固定轴欧拉角轴角
tfs.euler.euler2axangle(0,0,0,"sxyz")
# 轴角转固定轴欧拉角
tfs.euler.axangle2euler([1,0,0],0.5,"sxyz")
```

4.轴角相关转换

四元数模块在transforms3d.axangle里，直接导入即可使用

4.1 轴角与旋转矩阵互转

```
import transforms3d as tfs
import numpy as np
# 轴角转旋转矩阵
tfs.axangles.axangle2mat([1,0,0],0.5)
```

```
# 旋转矩阵转轴角
tfs.axangles.mat2axangle(np.asarray([[1., 0., 0.],[0., 1., 0.],[0., 0., 1.]]))
```

5.C++实现Eigen

> 开源地址,欢迎一起加入完善: https://gitee.com/ohhuo/transforms3d_cpp



参考文档：

- http://matthew-brett.github.io/transforms3d/

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**

- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

5.齐次坐标变换

前面几节中，小鱼带你一起学习了使用TF进行坐标的变换，也带你通过旋转和平移求解了坐标的变换关系，但计算的过程中旋转和平移是分开计算的，那有没有一种方法，可以让旋转矩阵和平移向量合并到同一个矩阵里呢？

答案是有的，我们可以将$3*3$的旋转矩阵和$3*1$的平移矩阵进行组合，并添加一行(0,0,0,1)使其变成一个$4*4$的方阵，其组合方式如下：

有旋转矩阵 $$ R = \begin{bmatrix}{r_{11}}&{r_{12}}&{r_{13}}\{r_{21}}&{r_{22}}&{r_{23}}\{r_{31}}&{r_{32}}&{r_{33}}\\end{bmatrix} \tag{旋转矩阵} $$ 平移矩阵 $$ P= \begin{bmatrix}{x}\{y}\{z}\\end{bmatrix} \tag{平移矩阵} $$

合并成齐次变换矩阵 $$ T = \begin{bmatrix}{r_{11}}&{r_{12}}&{r_{13}}&{x} \{r_{21}}&{r_{22}}&{r_{23}}&{y} \{r_{31}}&{r_{32}}&{r_{33}}&{z} \0&0&0&1 \\end{bmatrix} \tag{齐次矩阵} $$

为什么要这样写，我们可以简单的推导一下，矩阵是支持分块运算的，我们将上面的矩阵进行分块 $$ T = \begin{bmatrix}{R}&{P} \0&1\\end{bmatrix} \tag{齐次矩阵} $$ 假设^A_BT表示B坐标系到A坐标系的齐次变换，B坐标系下的点C坐标为^B_CP，求C在A坐标系下的坐标^A_CP

我们将^A_BT乘^B_CP上，可得 $$ ^A_CP= \begin{bmatrix}{^A_BR}&{^A_BP}\0&1\\end{bmatrix} \begin{bmatrix}{^B_CP}\1\\end{bmatrix} = {^A_BR}{^B_CP}+^A_BP $$ 根据前面学习的平移+旋转复合坐标变换公式，正确的结果如下 $$ ^A_CP = {^A_BR}{^B_CP}+^A_BP $$ 你会发现，两者最终结果完全相同，也就是说，我们的平移加旋转复合变换，可以直接用齐次变换矩阵代替。

1.齐次变换矩阵特性

接着我们来探索一下齐次变换矩阵的一些特性

2.1.齐次变换矩阵的符号表示

一般使用H或者T来表示齐次变换矩阵，矩阵的左上角标明参考坐标系，矩阵左下角标明目标坐标系，比如^A_BT表示B坐标系到A坐标系的变换关系(平移+旋转)

2.2.齐次变换矩阵的逆的几何含义

就像矩阵的逆一样，齐次变换矩阵也有逆，其逆也有对应的几何含义，比如

比如^A_BT表示B坐标系到A坐标系的变换关系

那么

^A_BT的逆$^A_BT^{-1}=^B_AT$表示A坐标系到B坐标系的变换关系

2.3. 齐次变换矩阵的乘法的几何含义

3.3.1齐次矩阵与平移向量相乘

齐次矩阵与平移向量相乘，即可求出某个向量在另一坐标系下的表示，上面例子中即是如此。

3.3.2齐次矩阵与齐次矩阵相乘

齐次矩阵与齐次矩阵相乘，可以转换不同坐标系之间的关系，比如： $$ ^A_BT^B_CT=^A_CT $$ 比如当我们有一个六自由度的机械臂，知道两两相邻关节之间的关系，那么就可以通过其次矩阵相乘的方法求出，关节6在关节0下的位置和姿态： $$ ^0_1T^1_2T^2_3T^3_4T^4_5T^5_6T=^0_6T $$

3.练习

练习小鱼放到了下一节了，毕竟不希望大家用手来算~

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

6.齐次坐标变换实战

上一节我们对齐次矩阵的组成和齐次矩阵的求逆和乘法两个运算的几何意义进行了介绍。

本节课我们就通过对应的函数和库实现齐次矩阵的生成，齐次矩阵的乘法和求逆。

1.齐次矩阵的合成与分解

齐次矩阵的的生成可以一个姿态和一个平移向量组成，因为姿态可以用四元数、欧拉角、轴角、旋转矩阵四种方式来表示

所以我们考虑先将对应的姿态转成旋转矩阵，然后使用numpy讲旋转矩阵和平移向量填写到齐次矩阵对应的位置即可

1.1旋转矩阵+平移向量

```
#导入库
import numpy as np
import transforms3d as tfs
```

```
# 定义旋转矩阵R和平移向量T
R = np.asarray([[1., 0., 0.],[0., 1., 0.],[0., 0., 1.]])
T = np.asarray([1,0,1])
R,T
```

```
In [49]: #导入库
         import numpy as np
         import transforms3d as tfs

In [50]: # 定义旋转矩阵R和平移向量T
         R = np.asarray([[1., 0., 0.],[0., 1., 0.],[0., 0., 1.]])
         T = np.asarray([1,0,1])
         R,T

Out[50]: (array([[1., 0., 0.],
                 [0., 1., 0.],
                 [0., 0., 1.]]),
          array([1, 0, 1]))
```

1.1.1 使用numpy方法合成齐次变换矩阵

```
temp = np.hstack((R,T.reshape(3,1)))
np.vstack((temp,[0,0,0,1]))
```

```
In  [45]: temp = np.hstack((R,T.reshape(3,1)))
          np.vstack((temp,[0,0,0,1]))

Out[45]: array([[1., 0., 0., 1.],
                [0., 1., 0., 0.],
                [0., 0., 1., 1.],
                [0., 0., 0., 1.]])
```

1.1.2 使用tfs中的函数合成齐次变换矩阵

```
tfs.affines.compose(T,R,[1,1,1])
```

```
In  [37]: tfs.affines.compose(T,R,[1,1,1])

Out[37]: array([[1., 0., 0., 1.],
                [0., 1., 0., 0.],
                [0., 0., 1., 1.],
                [0., 0., 0., 1.]])
```

1.2四元数+平移向量

思路:先将四元数转换成旋转矩阵，然后再利用1.1合成齐次矩阵

```
R = tfs.quaternions.quat2mat([1,0,0,0])
tfs.affines.compose(T,R,[1,1,1])
```

```
In  [48]: R = tfs.quaternions.quat2mat([1,0,0,0])
          tfs.affines.compose(T,R,[1,1,1])

Out[48]: array([[1., 0., 0., 1.],
                [0., 1., 0., 0.],
                [0., 0., 1., 1.],
                [0., 0., 0., 1.]])
```

1.3 练习

1.3.1 练习1

已知相机坐标系{C}为参考坐标系，工具坐标系{P}的位置矢量在相机坐标系{C} `x,y,z` 各轴投影为$2,1,2$，并且工具坐标系和相机坐标系姿态相同，求^C_PT

1.3.2 练习2

已知机器人基坐标系{B}为参考坐标系，相机坐标系{C}在的位置矢量在{B}各轴的投影为$0,0,3$,坐标系{C}和绕着坐标系{B}的x轴转了180度，求^B_CT

2.齐次矩阵的分解

齐次矩阵的分解指的是已有齐次矩阵的情况下，将其分解为姿态和平移两部分

2.1 将qcjz分解为固定轴欧拉角和平移向量

```
tfs.euler.mat2euler(T[0:3,0:3]),T[:3,3:4]
```

```
In [29]: tfs.euler.mat2euler(T[0:3,0:3]),T[:3,3:4]
Out[29]: ((0.0, -0.0, 0.0),
          array([[2.],
                 [1.],
                 [2.]]))
```

2.3 将qcjz分解为四元数和平移向量

```
tfs.quaternions.mat2quat(T[0:3,0:3]),T[:3,3:4]
```

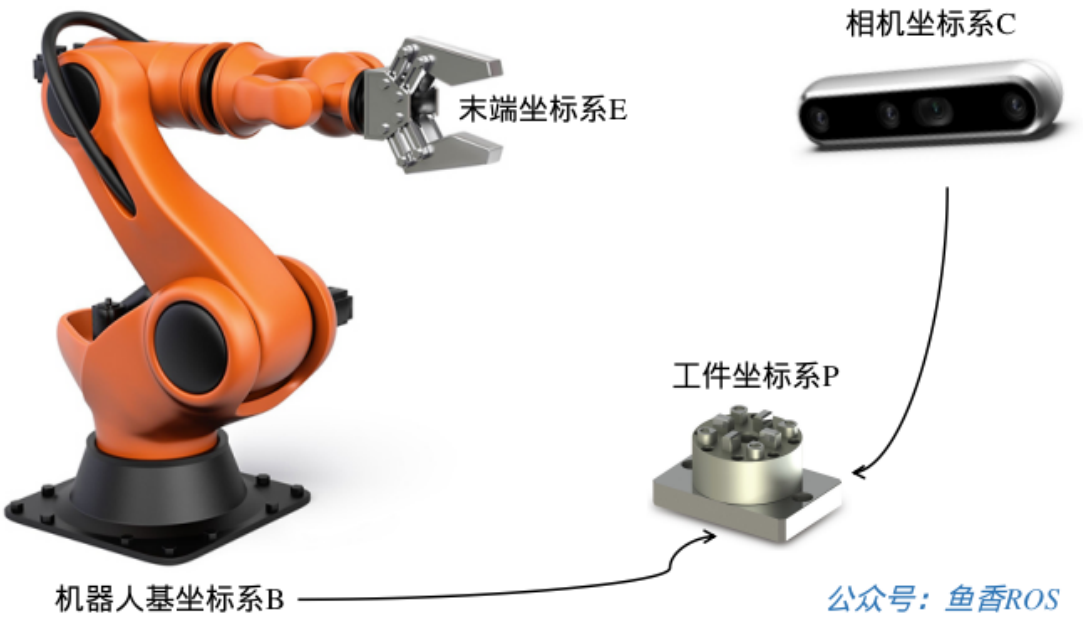
```
In [30]: tfs.quaternions.mat2quat(T[0:3,0:3]),T[:3,3:4]
Out[30]: (array([1., 0., 0., 0.]),
          array([[2.],
                 [1.],
                 [2.]]))
```

3.齐次矩阵的乘法

对应numpy中矩阵的乘法 `np.dot` 讲两个矩阵相乘即可，我们以一道例题来讲解这个问题。

3.1 练习-眼在手外

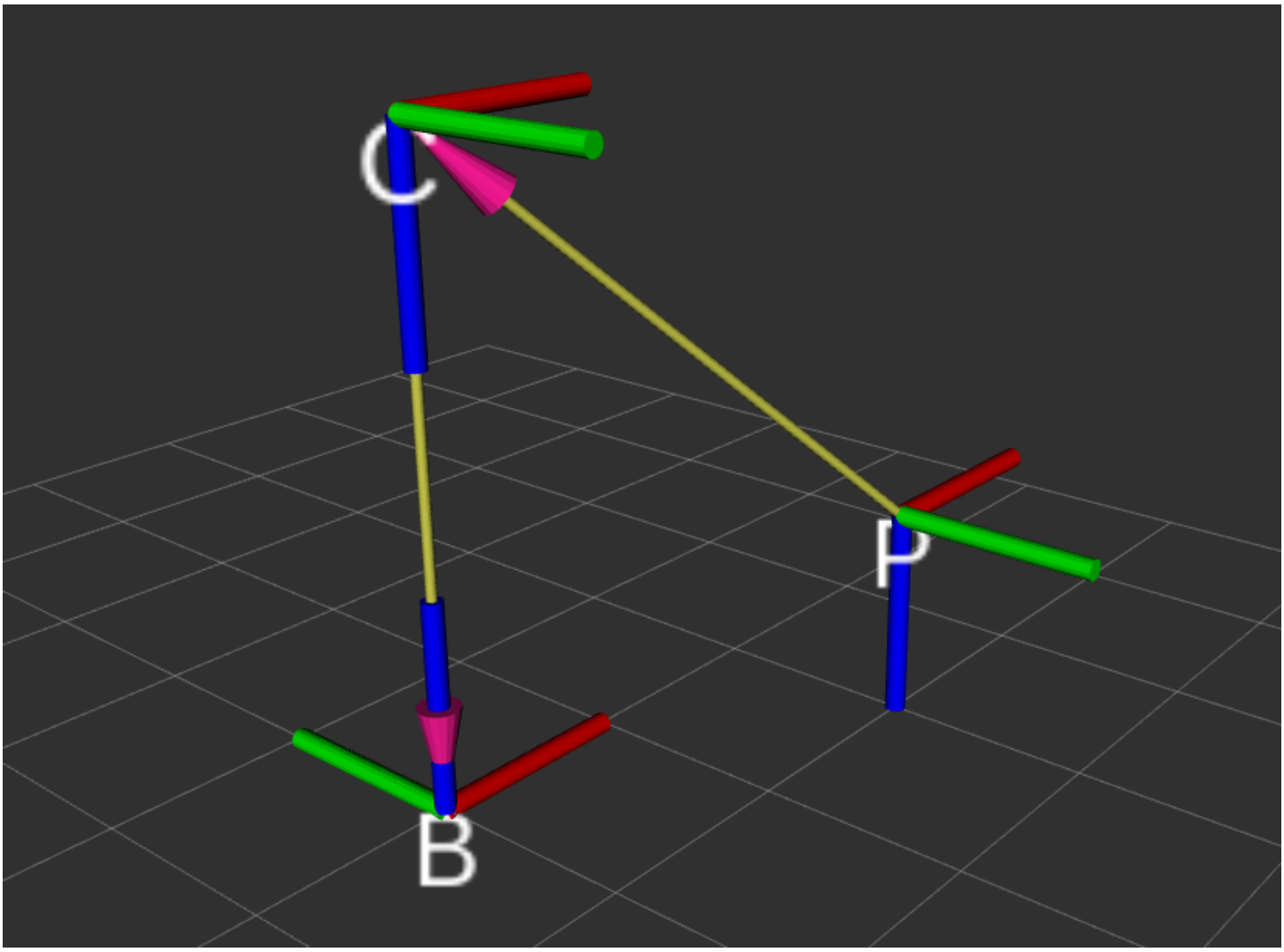
如图🔓示，已知：



1.相机坐标系{C}为参考坐标系，工具坐标系{P}的位置矢量在相机坐标系{C} `x,y,z` 各轴投影为$2,1,2$，并且工具坐标系和相机坐标系姿态相同。

2.机器人基坐标系{B}为参考坐标系，相机坐标系{C}在的位置矢量在{B}各轴的投影为$0,0,3$,坐标系{C}和绕着坐标系{B}的x轴转了180度

可以参考下图看题目



求：

{B}为参考坐标系，坐标系{P}的位置矢量和旋转矩阵

解体思路也很简单，我们只要得出

- B到C的齐次变换矩阵^B_CT
- C到P的齐次变换矩阵^C_PT

得到之后将两者相乘即可得出： $$ ^B_PT=^B_CT^C_PT $$ 求出^B_PT我们再将其分解成位置矢量和旋转矩阵即可

动手写代码：

先求T_BC

```
import math
T_BC = tfs.affines.compose([0,0,3],tfs.euler.euler2mat(math.pi,0,0),[1,1,1])
```

```
T_BC
```

In [33]:
```
import math
T_BC = tfs.affines.compose([0,0,3],tfs.euler.euler2mat(math.pi,0,0),[1,1,1])
T_BC
```

Out[33]:
```
array([[ 1.0000000e+00,  0.0000000e+00,  0.0000000e+00,  0.0000000e+00],
       [ 0.0000000e+00, -1.0000000e+00, -1.2246468e-16,  0.0000000e+00],
       [ 0.0000000e+00,  1.2246468e-16, -1.0000000e+00,  3.0000000e+00],
       [ 0.0000000e+00,  0.0000000e+00,  0.0000000e+00,  1.0000000e+00]])
```

再求T_CP

```
T_CP =  T = tfs.affines.compose([2,1,2],np.identity(3),[1,1,1])
T_CP
```

In [32]:
```
T_CP =  T = tfs.affines.compose([2,1,2],np.identity(3),[1,1,1])
T_CP
```

Out[32]:
```
array([[1., 0., 0., 2.],
       [0., 1., 0., 1.],
       [0., 0., 1., 2.],
       [0., 0., 0., 1.]])
```

求T_BP

```
T_BP = np.dot(T_BC,T_CP)
T_BP
```

In [34]:
```
T_BP = np.dot(T_BC,T_CP)
T_BP
```

Out[34]:
```
array([[ 1.0000000e+00,  0.0000000e+00,  0.0000000e+00,  2.0000000e+00],
       [ 0.0000000e+00, -1.0000000e+00, -1.2246468e-16, -1.0000000e+00],
       [ 0.0000000e+00,  1.2246468e-16, -1.0000000e+00,  1.0000000e+00],
       [ 0.0000000e+00,  0.0000000e+00,  0.0000000e+00,  1.0000000e+00]])
```

分解成欧拉角对比结果

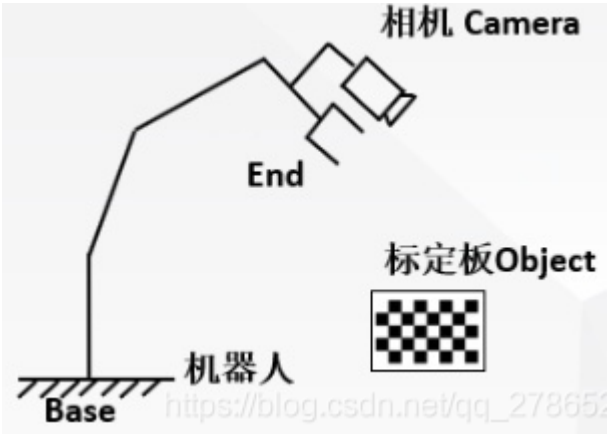
```
tfs.euler.mat2euler(T_BP[0:3,0:3]),T_BP[:3,3:4]
```

```
In [36]: tfs.euler.mat2euler(T_BP[0:3,0:3]),T_BP[:3,3:4]
Out[36]: ((3.141592653589793, -0.0, 0.0),
          array([[ 2.],
                 [-1.],
                 [ 1.]]))
```

到这里我们就利用做了齐次矩阵的乘法完成了坐标的变换

3.齐次矩阵求逆

3.1练习-眼在手上



如图机器人基座坐标系为B、末端坐标系为E、相机坐标系为C、物品坐标系为O、其中相机固定在机械臂的末端。

已知 $$ ^B_ET={ xyz:[0.5,0.6,0.8] ,qwqxqyqz:[1,0,0,0]} \ ^C_ET={ xyz:[0.00,0.05,0.05] ,qwqxqyqz:[0.707, 0.706, 0,0]} \ ^C_OT={ xyz:[0.00,0.02,0.85] ,qwqxqyqz:[0.877,0.479,0,0]} \ $$ 求：^B_OT $$ ^B_OT=^B_ET^E_CT^C_OT \ ^E_CT=^C_ET^{-1} $$ 写代码：

```
T_BE = tfs.affines.compose([0.5,0.6,0.8],tfs.quaternions.quat2mat([1,0,0,0]),
[1,1,1])
T_CE =
tfs.affines.compose([0.00,0.05,0.05],tfs.quaternions.quat2mat([0.707,0.706,0,0]),
[1,1,1])
T_CO =
```

```
tfs.affines.compose([0.00,0.02,0.85],tfs.quaternions.quat2mat([0.877,0.479,0,0]),
[1,1,1])
```

```
T_EC = np.linalg.inv(T_CE)
```

```
np.dot(np.dot(T_BE,T_EC),T_CO)
```

```
In [51]: T_BE = tfs.affines.compose([0.5,0.6,0.8],tfs.quaternions.quat2mat([1,0,0,0]),[1,1,1])
         T_CE = tfs.affines.compose([0.00,0.05,0.05],tfs.quaternions.quat2mat([0.707,0.706,0,0]),[1,1,1])
         T_CO = tfs.affines.compose([0.00,0.02,0.85],tfs.quaternions.quat2mat([0.877,0.479,0,0]),[1,1,1])

In [52]: T_EC = np.linalg.inv(T_CE)

In [53]: np.dot(np.dot(T_BE,T_EC),T_CO)

Out[53]: array([[ 1.        ,  0.        ,  0.        ,  0.5       ],
                [ 0.        ,  0.8421333 ,  0.53926942,  1.39995674],
                [ 0.        , -0.53926942,  0.8421333 ,  0.83113231],
                [ 0.        ,  0.        ,  0.        ,  1.        ]])
```

4.练习

4.1 map坐标系转换

在移动机器人导航中，存在这样一个坐标系关系.

地图坐标系(Map)->里程计坐标系(Odom)->机器人坐标系(BaseLink)

其中里程计到机器人坐标系关系一般是由底盘轮子编码器给出，而地图坐标系和里程计坐标系之间的关系是通过定位模块估算出来的。

所以请听题目：

现在通过地图匹配获取到机器人在地图中的位置为[1.5,2.3,0],姿态(固定轴欧拉角)为[0,0,3.14]

查看里程计上报的机器人坐标为：位置[1.0,3.2,0] 姿态(固定轴欧拉角)[0,0,1.0]

求地图坐标系和里程计坐标系之间的关系

4.2 机械臂运动学正解

已知一个3自由的机械臂，已知:

- 关节1和关节2坐标关系为:[0,0,0.2] 固定轴欧拉角:[0,0,1.57]
- 关节2和关节3坐标关系为:[0.5,0,0.0] 固定轴欧拉角:[0,0,1.0]

求关节1和关节3之间的关系？

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

7. 机器人运动学介绍

机器人学是一个非常庞大的学科，凡是和机器人设计应用相关的都可以划分到机器人学中，主要有运动学和动力学、系统结构、感知传感技术、运动规划技术、决策技术等。

为了方便后面的机器人建模和仿真，本节我们对机器人的运动学进行简单的介绍和学习。

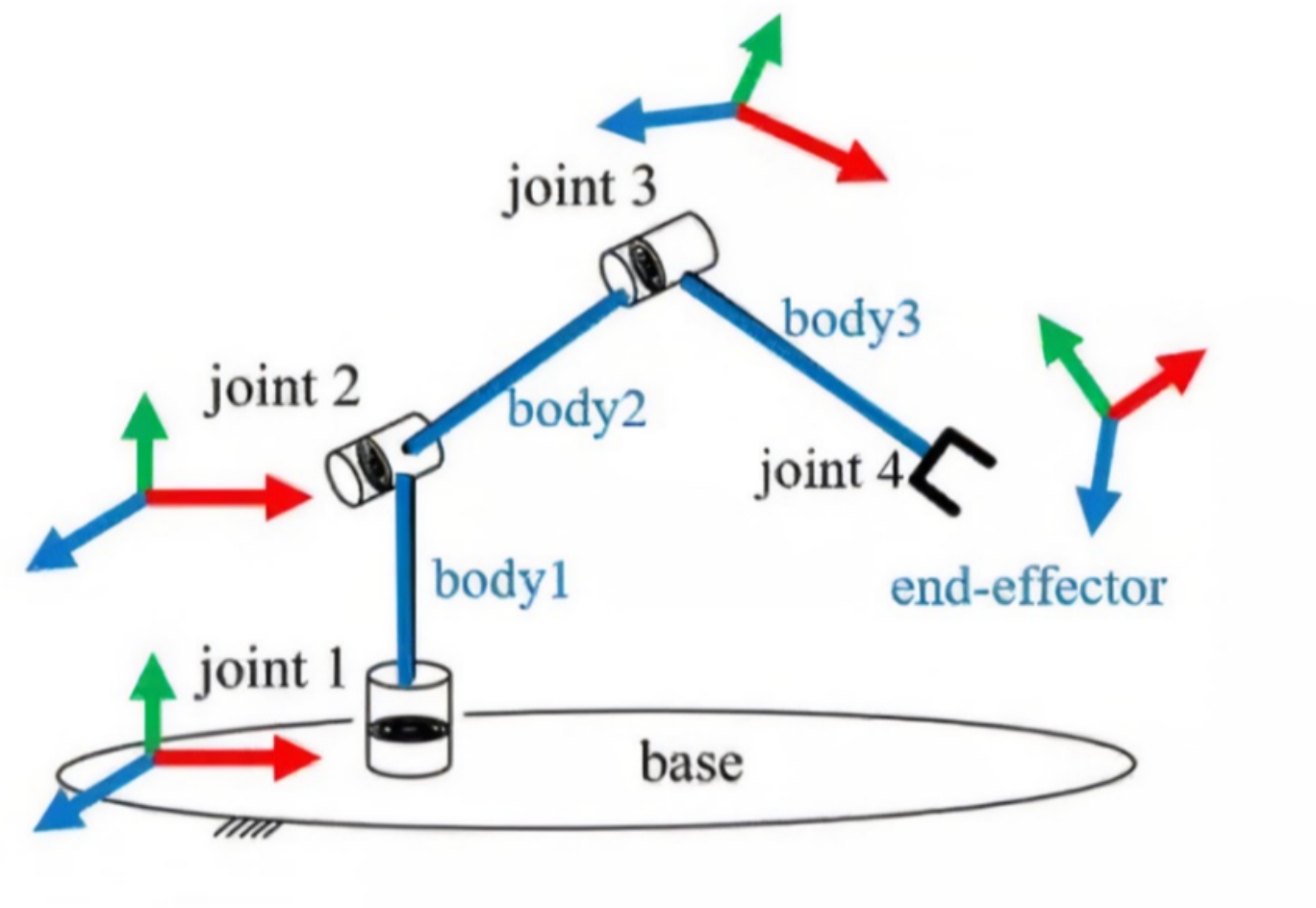
1.机器人运动学

机器人运动学研究机器人的位姿关系，主要包含正向运动学和逆向运动学两类。

正向运动学即给定机器人各关节变量，计算机器人末端的位置姿态；比如上节课我们已知机器人关节和关节之间的关系，求关节1和关节3之间的关系

逆向运动学即已知机器人末端的位置姿态，反求机器人的关节变量；比如当我们已知机器人关节1和关节3之间的关系，求关节关节1和关节2，关节2和关节3之间关系。

2.机械臂运动学介绍



正运动学：已知每个关节的角度，求末端的位姿

逆运动学：已知末端姿态，求每一个关节的角度

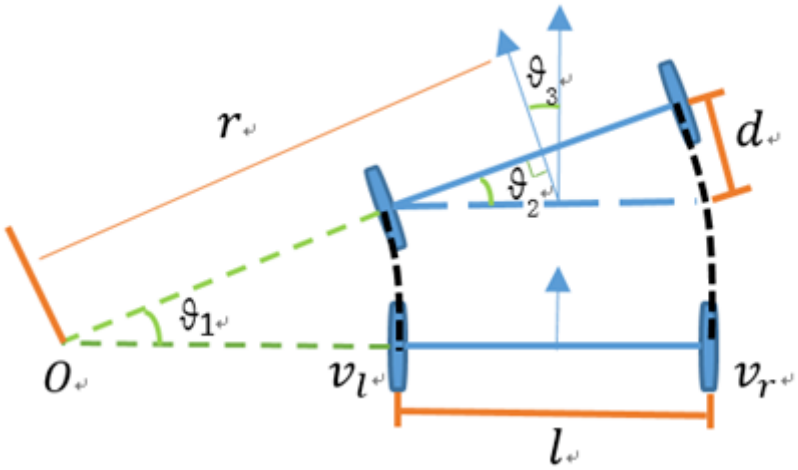
3.两轮差速底盘运动学介绍

两轮差速模型指机器人底盘由两个驱动轮和若干支撑轮构成的底盘模型，像turtlebot和开源机器人fishbot都是两轮差速模型。





两轮差速模型通过两个驱动轮可以通过不同转速和转向，使得机器人的达到某个特定的角速度和线速度。



两轮的平衡车大家都见过吧，靠着两个轮子即可实现前后移动（线速度），左转右转（角速度）。

3.1 正逆解

了解了两轮差速模型，那正逆解又是怎么回事？

正运动学：已知两个轮子的速度，求整车的角速度（弧度/秒）和线速度（米/秒）

```mermaid
graph LR;
A[左轮当前速度]-->B[正运动学]-->C[机器人当前角速度]
D[右轮当前速度]-->B
B-->E[机器人当前线速度]
```

逆运动学：已知目标角速度和线速度，求两个轮子的转速

```mermaid
    graph LR;
    A[机器人目标线速度]-->C[运动学逆解]
    B[机器人目标角速度]-->C
    C-->D[左轮目标速度]
    C-->E[右轮目标速度]
```

3.2轮式里程计

```mermaid
graph LR;
A[左右轮当前速度/位置]-->B[里程计推算]
B-->C[里程计-odom]
```

当我们知道了两个轮子之间的相对位置，同时知道了每一时刻机器人的角速度和线速度，那我们如何获取机器人的当前角度和位置呢？

3.2.1 角度

影响机器人当前角度的因素只有一个，就是角速度。

某一时刻机器人转动的角度 = 这一时刻机器人的角速度*这一时刻时长

假如我们认定初始时刻机器人的角度为0,通过对机器人转动角度角度进行累加，即可获得机器人的当前角度。

上述过程其实就是对角速度进行积分得到角度。

3.2.2 位置

通过对角速度积分，我们得到了角度。

机器人某一时刻自身方向上的前进速度可以分解为里程计坐标系中x轴和y轴方向上的速度。

 速度分解

从图中可以看出： $$ v_y = vcos(\theta) \ v_y = vsin(\theta) $$ 得到了x和y方向上的速度，乘上该速度对应的某一时刻经过的时间，即可得到这一时刻在x轴和y轴方向上的位移，对位移进行累加即可得到里程计中的x和y。

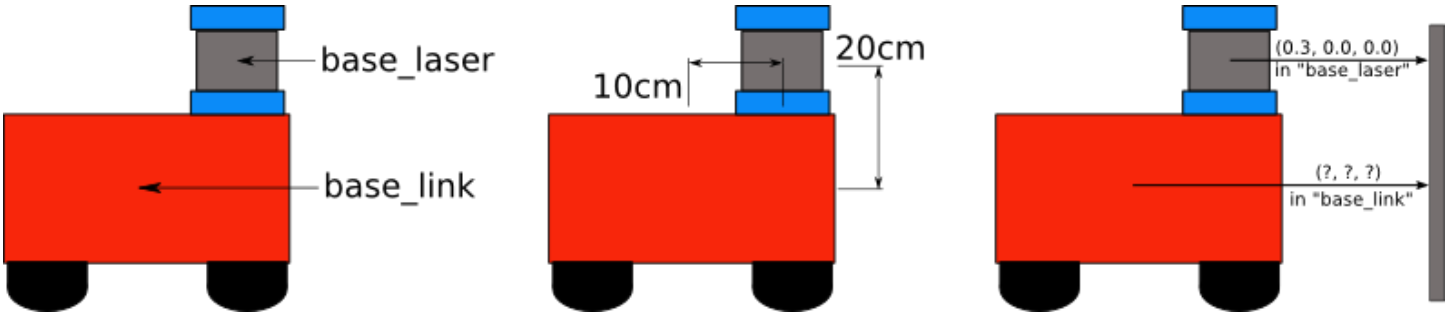
技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

1.tf2介绍

TF即 `变换` 的英文单词 `TransForm` 的缩写。所以 `ROS` 和 `ROS2` 中的 `TF` 就是指和坐标变换相关的工具。

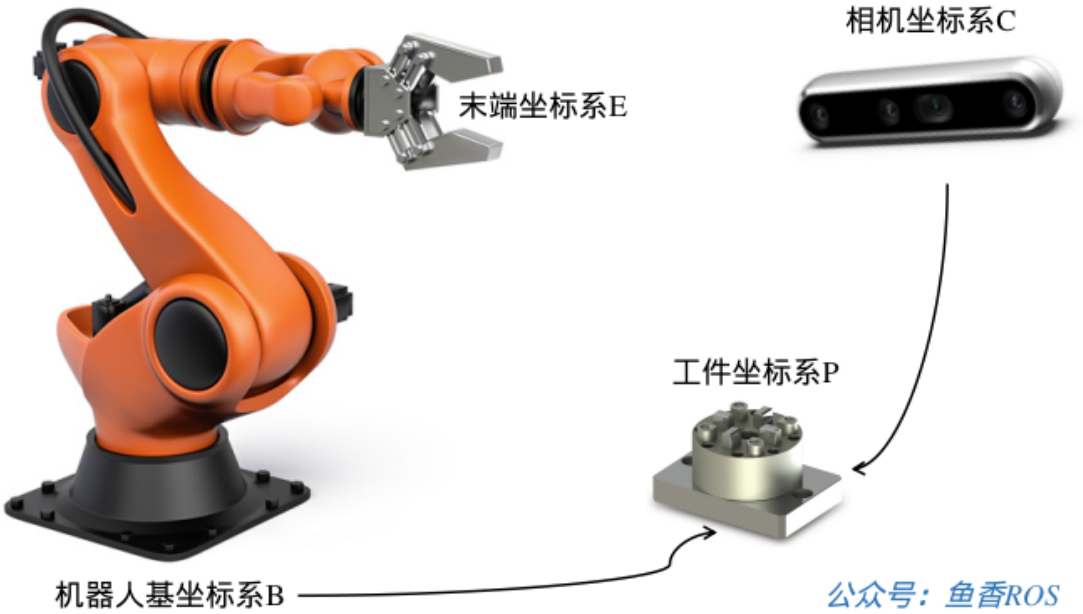
> 在搞机器人当中，坐标变换经常用到，所以 `ROS2` 帮我们做了一个强大易用的TF工具



1.发布坐标关系

我们先使用TF2的相关工具，解决上一节的手眼坐标转换问题，直观的感受一下TF2的强大。

要想让TF帮我们完成坐标变换，我们就需要告诉它坐标和坐标之间的关系。



拿上面的手眼系统来说，我们要想获取到相机的基坐标系{B}和工具{P}之间的关系，只需要将机械臂和相机、相机和工具之间的关系告诉TF即可。

2.我们如何告诉TF？

可以使用tf的坐标广播工具进行广播坐标关系，广播时需要三个数据：

- 父坐标系名称（字符串）
- 子坐标系名称（字符串）
- 父子之间的变换关系（平移关系和旋转关系）

在终端中输入：

```
ros2 run tf2_ros static_transform_publisher
```

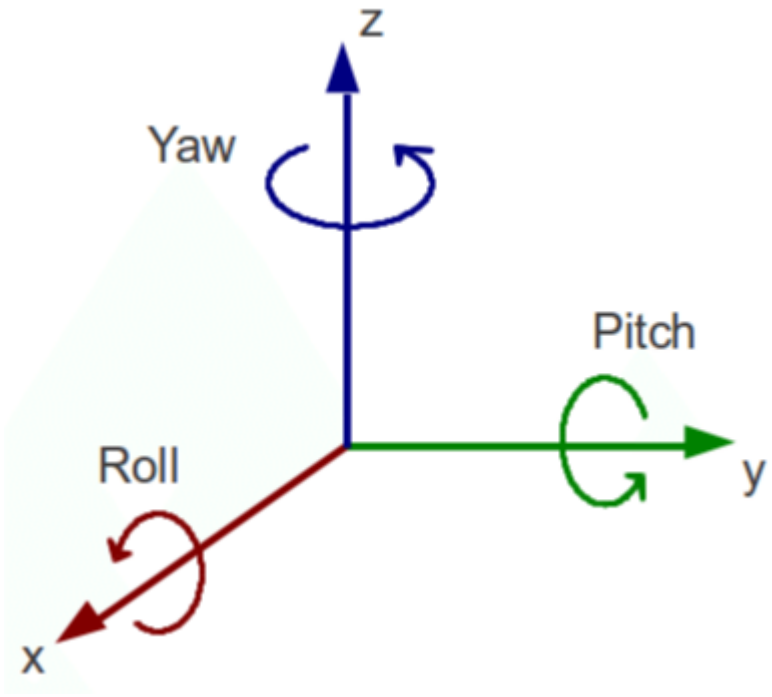
按`enter`键，可以看到

```
A command line utility for manually sending a transform.
Usage: static_transform_publisher x y z qx qy qz qw frame_id child_frame_id
OR
Usage: static_transform_publisher x y z yaw pitch roll frame_id child_frame_id
```

这是该CLI所提供的使用提示，可以看出

使用TF发布位置和姿态时，位置的描述使用的是xyz三个参数，而姿态的描述则分两种

- 第一种是四元数形式（qx qy qz qw）
- 第二种是欧拉角形式（yaw偏航角-rz pitch俯仰角-ry roll滚转角-rx），我们这里采用的是欧拉形式，绕x轴旋转采用欧拉角中的滚转角roll来描述，注意角度单位采用弧度制。



> 关于欧拉角和四元数的区别我们放到了姿态的多种表示章节来讲。

2.1 发布B到C的位姿

比如针对上面的手眼转换，广播机械臂坐标系{B}和相机坐标系{C}之间的关系。

父坐标系的名字就是B，子坐标系的名字是C，父子之间的平移关系是`0 0 3`,旋转关系是绕x轴旋转180度。

在ROS2中可以使用下面的指令发布变换，打开终端，输入下面的指令：

```
ros2 run tf2_ros static_transform_publisher 0 0 3 0 0 3.14 B C
```

如果在终端中看到下面的提示则代表发布成功

```
mzebra@mzebra:~$ ros2 run tf2_ros static_transform_publisher 0 0 3 0 0 3.14 B C
[INFO] [1636555214.264714318] [static_transform_publisher_vv7FLVsXeM4YcXvF]: Sp
inning until killed publishing transform from 'B' to 'C'
```

2.2 发布C到P的位姿

接着我们发布坐标系{C}到坐标系{T}的位姿

再打开一个新的终端，输入下面的命令：

```
ros2 run tf2_ros static_transform_publisher 2 1 2 0 0 0 C P
```

3. 监听/获取TF关系

发布也发布了，接着我们就把坐标系之间的关系打印出来，只要坐标系之间是有连接的，我们就可以使用TF求出来，使用下面的指令就可以得到机械臂基坐标系{B}和工具坐标系{P}之间的关系。

打开终端,输入命令：

bash

```
ros2 run tf2_ros tf2_echo B P
```

```
mzebra@mzebra:~$ ros2 run tf2_ros tf2_echo B P
[INFO] [1636557800.782174277] [tf2_echo]: Waiting for transform B ->  P: Invali
d frame ID "B" passed to canTransform argument target_frame - frame does not ex
ist
At time 0.0
- Translation: [2.000, -1.003, 1.002]
- Rotation: in Quaternion [1.000, 0.000, 0.000, 0.001]
At time 0.0
- Translation: [2.000, -1.003, 1.002]
- Rotation: in Quaternion [1.000, 0.000, 0.000, 0.001]
At time 0.0
- Translation: [2.000, -1.003, 1.002]
- Rotation: in Quaternion [1.000, 0.000, 0.000, 0.001]
```

可以看到终端中不断输出B和C之间的平移和旋转，平移采用的是xyz，基本正确，y和z的微小差异是因为我们发布变换时旋转输入的是3.14并不精确。

至于旋转部分采用的是四元数表示，关于这部分姿态的表示，小鱼在下一节会讲，大家不必纠结。

除了使用TF获取关系外，ros2还提供很多工具来查看坐标之间的关系，大家可以在终端中输入下面的命令自行尝试。

4.TF常用工具

4.1 rqt_tf_tree

> 2022-04-26更新的，这个工具的二进制安装版本作者3月底才发布，之前没提的原因是需要源码装太麻烦了

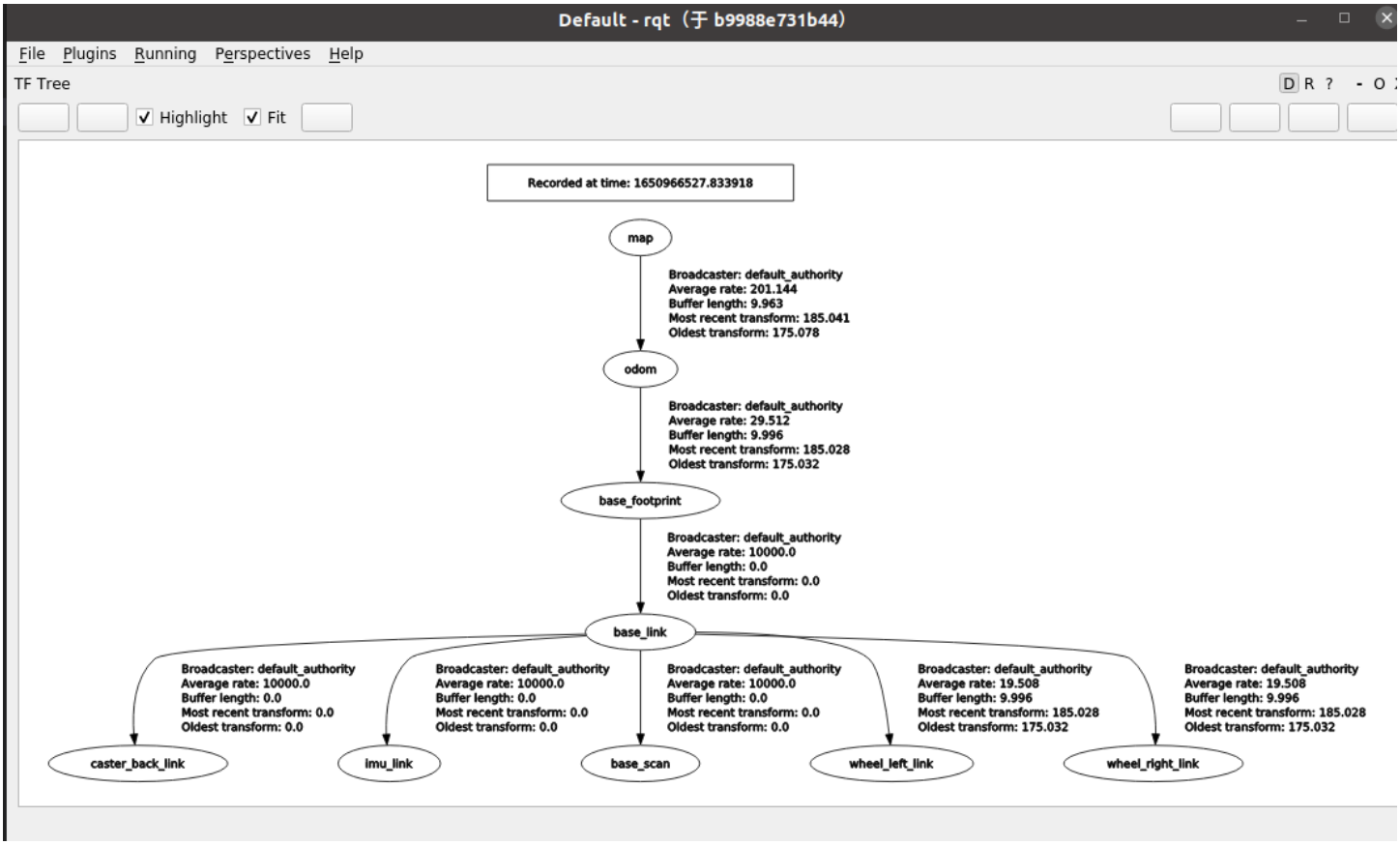
这个工具需要我们手动安装下

```
sudo apt install ros-humble-rqt-tf-tree
```

安装完成后，再次打开rqt工具， `Plugins` -> `Visualization` -> `TF Tree`

接着你就可以看到这个强大的，几乎可以实时看到系统tf更新信息的工具，这个工具对于后面我们进行导航和机械臂的调试非常有帮助。

> 长的不一样没关系，这是后面补充的图。



4.2 tf2_monitor

查看所有的发布者和频率。

```
ros2 run tf2_ros tf2_monitor
Gathering data on all frames for 10 seconds...

RESULTS: for all Frames

Frames:
Frame: C, published by <no authority available>, Average Delay: 3001.98, Max
Delay: 3001.98
Frame: P, published by <no authority available>, Average Delay: 741.497, Max
Delay: 741.497
All Broadcasters:
Node: <no authority available> 5029.14 Hz, Average Delay: 1871.74 Max Delay:
3001.98

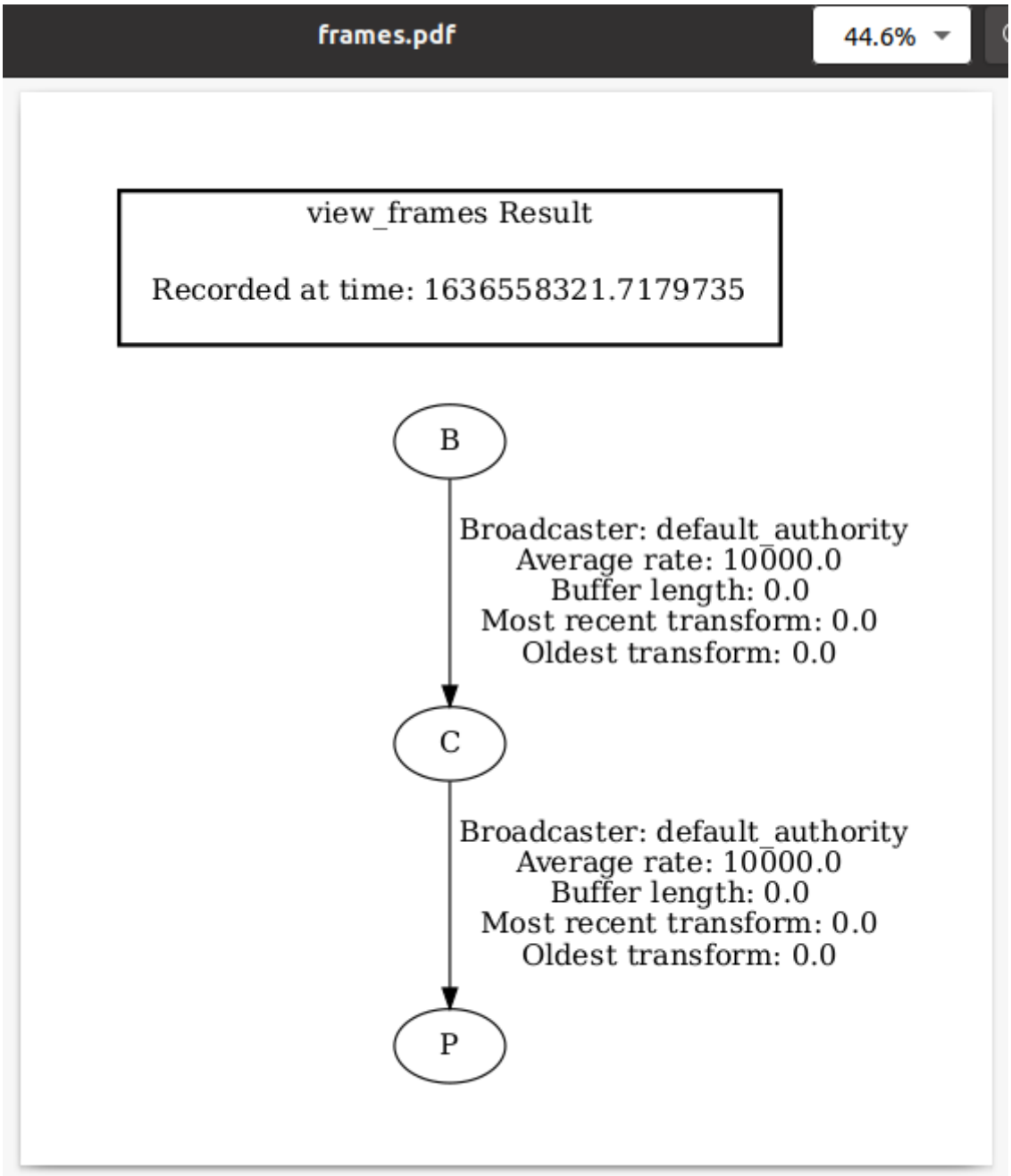

RESULTS: for all Frames
Frames:
Frame: C, published by <no authority available>, Average Delay: 3001.98, Max
Delay: 3001.98
Frame: P, published by <no authority available>, Average Delay: 741.497, Max
Delay: 741.497
All Broadcasters:
Node: <no authority available> 5029.14 Hz, Average Delay: 1871.74 Max Delay:
3001.98
```

view_frames.py

可以生成TF的pdf，目前也有在线的实时查看工具。

```
ros2 run tf2_tools view_frames.py
[INFO] [1636558316.667894410] [view_frames]: Listening to tf data during 5
seconds...
[INFO] [1636558321.702280144] [view_frames]: Generating graph in frames.pdf
file...
[INFO] [1636558321.709904442] [view_frames]:
Result:tf2_msgs.srv.FrameGraph_Response(frame_yaml="C: \n  parent: 'B'\n
broadcaster: 'default_authority'\n  rate: 10000.000\n  most_recent_transform:
```

```
0.000000\n  oldest_transform: 0.000000\n  buffer_length: 0.000\nP: \n  parent:
'C'\n  broadcaster: 'default_authority'\n  rate: 10000.000\n
most_recent_transform: 0.000000\n  oldest_transform: 0.000000\n  buffer_length:
0.000\n")
/opt/ros/humble/lib/tf2_tools/view_frames.py:75: YAMLLoadWarning: calling
yaml.load() without Loader=... is deprecated, as the default Loader is unsafe.
Please read https://msg.pyyaml.org/load for full details.
  data = yaml.load(result.frame_yaml)
```



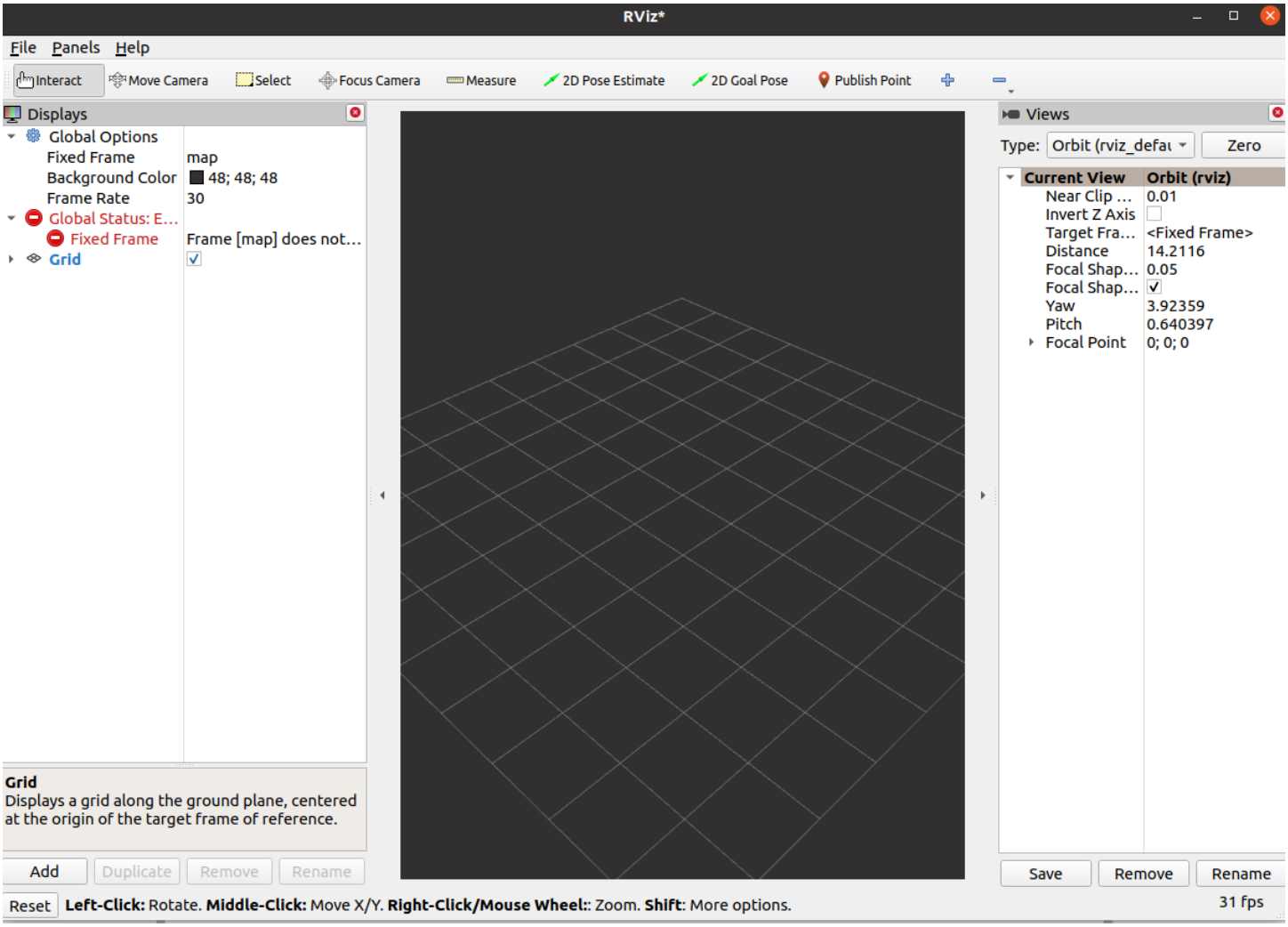
除了使用命令行进行坐标关系的广播和监听，我们还可以使用代码来广播和监听，接下来小鱼就带你一起用程序来发布TF广播和获取坐标关系。

2.可视化坐标变换

运行上节课的示例，打开终端输入rviz2，打开rviz2，我们尝试在rviz2中直观的看到坐标之间的关系

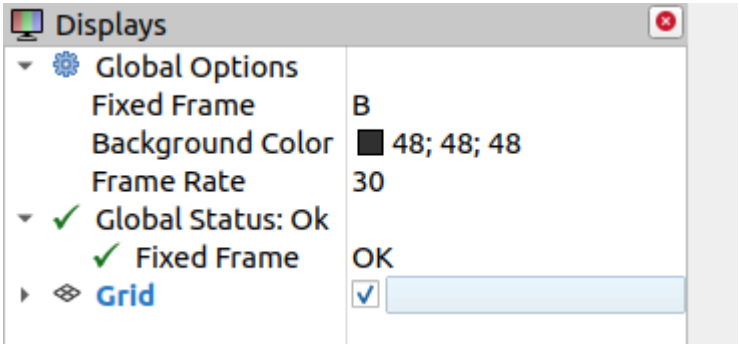
1.设置默认坐标系

刚打开RVIZ2，你看到的应该是这样一个界面



我们可以看到窗口左边的配置选项，全局选项默认选择的FixedFrame为map，这个map就是rviz2默认的坐标系的名字，动动脚趾头想一下我们并没有发布map这个坐标系，所以下面Global Status也是红色的错误。

此时我们可以手动的修改以下固定的Frame为B，让默认的坐标系设置成机械臂的基坐标系{B}。

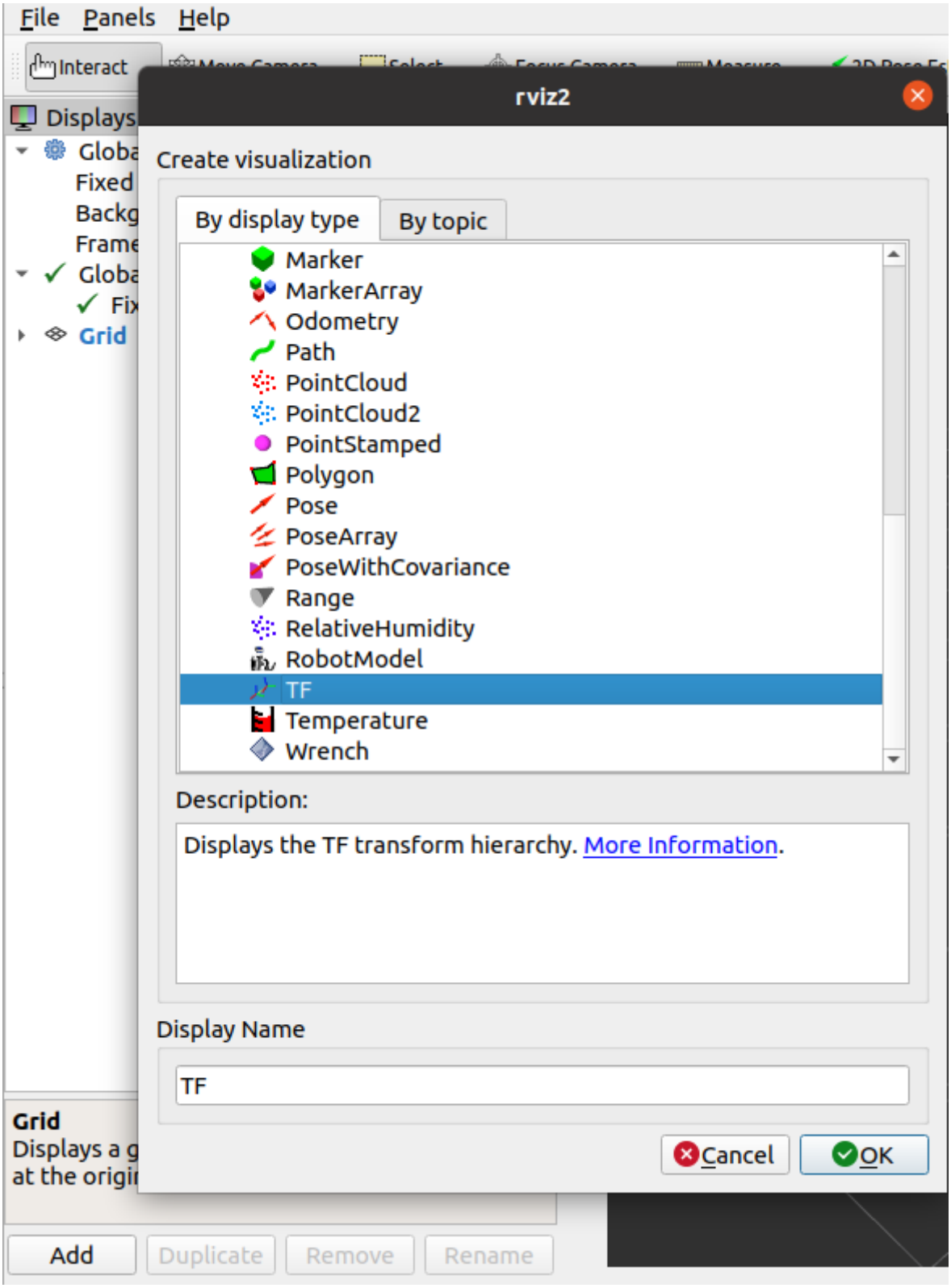


设置完成后，错误也没有了，因为此时的ROS2的TF中确确实实找到了一个叫做B的坐标系。

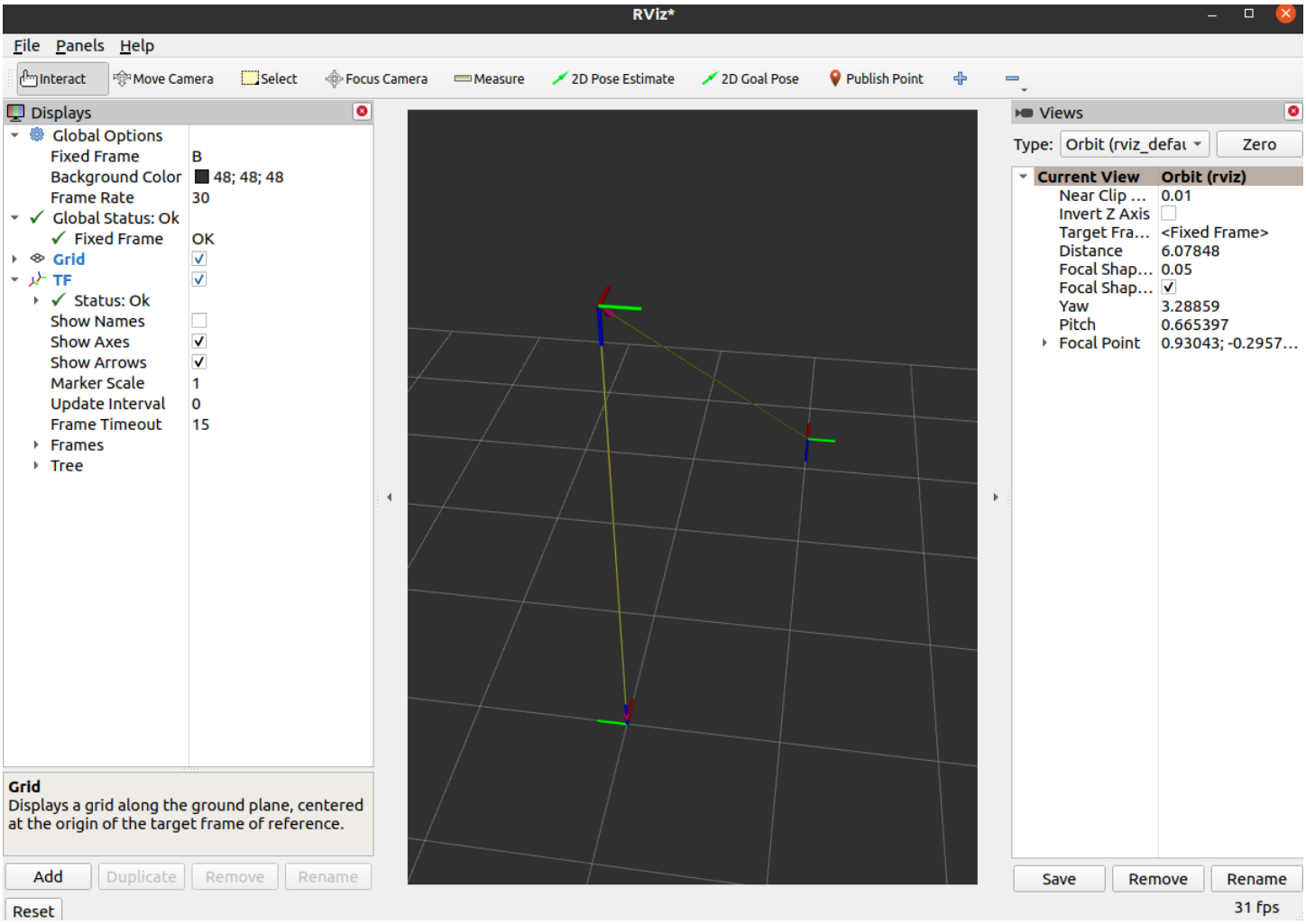
2.添加TF插件

即使没有错误，现在我们还是看不到坐标系，这是为什么呢？在小鱼的动手学ROS2课程当中，小鱼讲过，RVIZ2是一个插件化的软件，所以我们要添加TF相关的插件才能看到TF数据。

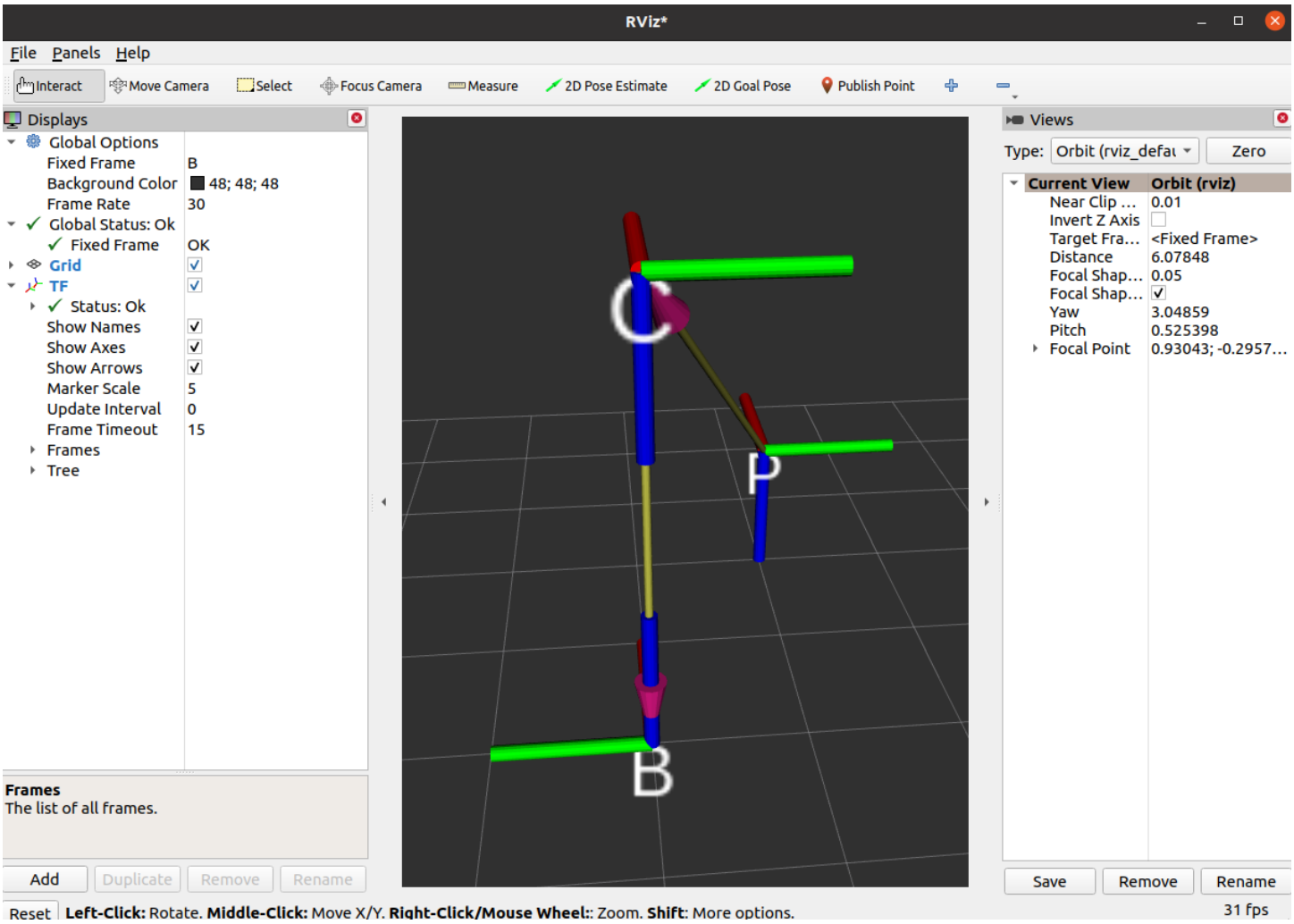
点击左下角的Add，在弹出的窗口中选择TF点击OK



之后你在RVIZ2中就可以看到下图的坐标关系



终于显示出来了，但是没有名字又太小了，修改下左边的选项，勾选Show Names，修改Marker Scale 为5



如果觉得视角不好，可以使用鼠标左键右键以及按下滚轮拖动修改。

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

3.坐标变换发布监听Python实现

运行前面安装的jupyter，我们尝试使用代码来操作tf

在开始之前，我们总结下TF相关的操作有哪些？

- 广播（TransformBroadcaster）：将坐标系鱼坐标系之间的位姿态关系发布出去
- 监听（TransformListener）：获取坐标系和坐标系之间的位姿关系
- TF(坐标变换TransformStamped)帧：广播出去的一组数据可以称作一个TF帧，包含父坐标系名称、子坐标系名称，父坐标系和子坐标系之间的关系

接着我们就可以编写代码来实现上面对TF帧的发布和监听。

1.坐标变换广播

在进行坐标变换广播前，我们根据两个坐标系之间的关系是否会随着时间变化分成以下两种情况：

1. 坐标系之间的关系不随时间推移而改变，称为静态坐标变换，需要使用静态广播发布器（StaticTransformBroadcaster）发布。比如：机器人的两个轮子之间关系，可以认为随时间的变换其相对位置不变。
2. 坐标系之间的关系随时间的推移而改变，称为（动态）坐标变换，使用广播发布器（TransformBroadcaster）发布坐标关系。比如机器人在世界坐标系中的位置，因为机器人会动。

我们依然使用手眼系统为例，尝试使用广播发布器来发布坐标系之间的关系。 在手眼系统中，机械臂基座和相机坐标系之间的位置是固定不变的，我们可以通过静态广播发布器来发布，而相机坐标系下工件的位置是实时识别得到的，其值会随时间变化，故我们用广播发布器来发布。

2.静态广播发布器

在jupyter中输入并运行以下程序，使用方式就像ros2的话题发布，只不过不需要指名发布的话题（发布到系统的TF树上）。

1.导入rclpy和Node

In [1]:
```
import rclpy
from rclpy.node import Node
```

2.导入TF帧：TF帧对应的消息接口为geometry_msgs.msg下的TransformStamped 其结构可以使用下面指令查看：

```
!ros2 interface show geometry_msgs/msg/TransformStamped
```

In [2]:
```
!ros2 interface show geometry_msgs/msg/TransformStamped
```

...

In [3]:
```
from geometry_msgs.msg import TransformStamped
```

3.从tf2_ros包中导入静态坐标变换广播器

In [4]:
```
from tf2_ros import StaticTransformBroadcaster
```

4.初始化节点

In [5]:
```
rclpy.init()
node = Node("transform_node")
```

5.构造静态广播发布器

In [6]:
```
static_tf_pub = StaticTransformBroadcaster(node)
```

6.构造TF帧，，其要发布出去的消息

In [7]:
```
t = TransformStamped()
t.header.stamp = node.get_clock().now().to_msg()
# parent-name:基坐标系B
t.header.frame_id = 'B'
# chrild-name: 相机坐标系C
t.child_frame_id = 'C'
# 平移关系，单位m
t.transform.translation.x = 0.0
t.transform.translation.y = 0.0
t.transform.translation.z = 3.0
# 旋转关系，四元数形式，我们需要将欧拉角的形式转换成四元数
# 可以使用在线坐标转换工具：https://quaternions.online/
t.transform.rotation.x = 1.0
t.transform.rotation.y = 0.0
t.transform.rotation.z = 0.0
t.transform.rotation.w = 0.0
```

7.发布坐标关系

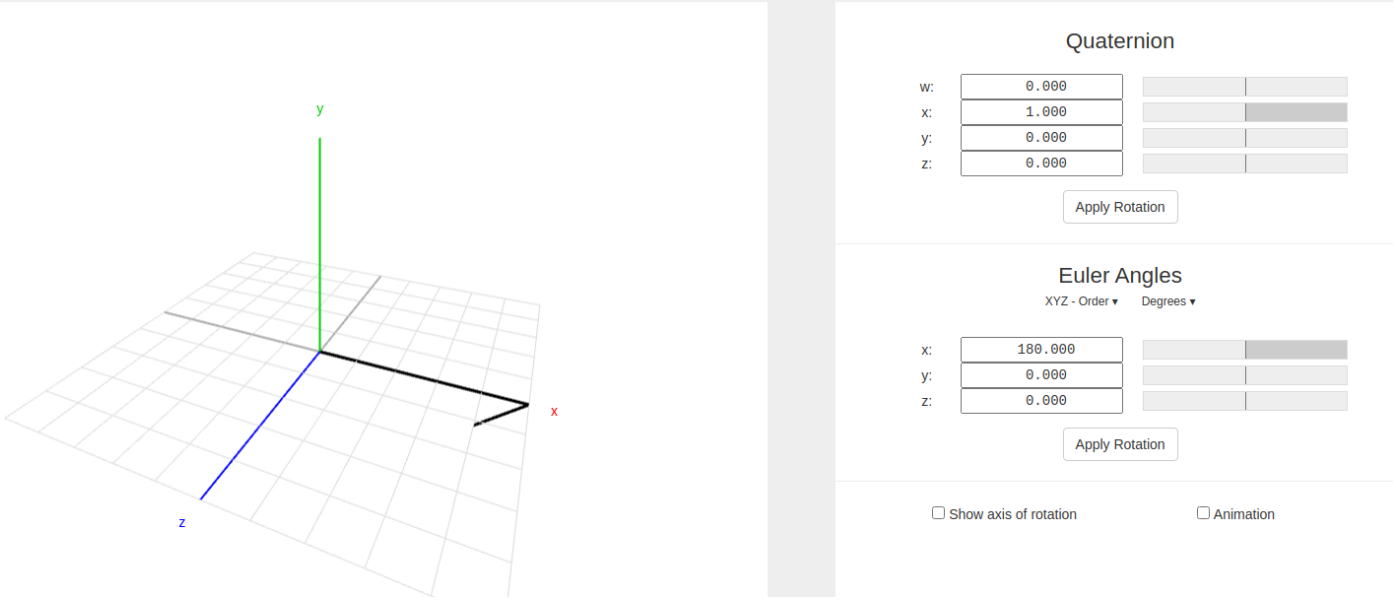
In [13]:
```
static_tf_pub.sendTransform(t)
```

使用命令行监听坐标关系

```
ros2 run tf2_ros tf2_echo B C
```

In [ ]:
```
!ros2 run tf2_ros tf2_echo B C
```

这里四元数的值需要通过在线的坐标转换获取，选择角度，绕x轴旋转 `180` ，上面就是对应的四元数，`x,y,z,w` 为 `1,0,0,0`



最后通过坐标监听工具可以查找出B和C之间的关系：

```
In [9]: !ros2 run tf2_ros tf2_echo B C
[INFO] [1637481580.682273537] [tf2_echo]: Waiting for transform B ->  C: Invalid frame ID "B" passed to canTransfo
rm argument target_frame - frame does not exist
At time 0.0
- Translation: [0.000, 0.000, 3.000]
- Rotation: in Quaternion [1.000, 0.000, 0.000, 0.000]
At time 0.0
- Translation: [0.000, 0.000, 3.000]
- Rotation: in Quaternion [1.000, 0.000, 0.000, 0.000]
At time 0.0
- Translation: [0.000, 0.000, 3.000]
- Rotation: in Quaternion [1.000, 0.000, 0.000, 0.000]
```

> 注意这里输出的At time 0.0 代表任意时刻

3.广播发布器

接着我们来使用广播发布器发布C和工具P之间的关系平移：`x:2 y:1 z:2` 旋转:`qx:0 qy:0 qz:0 qw:1`

只需要将上面的静态广播发布器改为广播发布器，但需要注意的是，发布坐标变换时要以一定的频率实时发布，这样当我们获取坐标时才能获取到当前时刻的坐标（有点不好理解，就是坐标关系和时间有关系）。

1.导入rclpy和Node

In [1]:
```
import rclpy
from rclpy.node import Node
```

2.导入TF帧：TF帧对应的消息接口为geometry_msgs.msg下的TransformStamped

In [3]:
```
from geometry_msgs.msg import TransformStamped
```

3.从tf2_ros包中导入坐标变换广播器

In [4]:
```
from tf2_ros import TransformBroadcaster
```

4.初始化节点

In [5]:
```
rclpy.init()
node = Node("transform_node2")
```

5.构造静态广播发布器

In [6]:
```
tf_pub = TransformBroadcaster(node)
```

6.构造TF帧，，其要发布出去的消息

In [7]:
```
t = TransformStamped()
t.header.stamp = node.get_clock().now().to_msg()
# parent-name:相机坐标系C
t.header.frame_id = 'C'
# chrild-name:工件坐标系P
t.child_frame_id = 'P'
# 平移关系，单位m
t.transform.translation.x = 2.0
t.transform.translation.y = 1.0
t.transform.translation.z = 2.0
# 旋转关系，四元数形式，我们需要将欧拉角的形式转换成四元数
# 可以使用在线坐标转换工具：https://quaternions.online/
t.transform.rotation.x = 1.0
t.transform.rotation.y = 0.0
t.transform.rotation.z = 0.0
t.transform.rotation.w = 0.0
```

7.以10Hz发布坐标关系

In [*]:
```
def send_transform():
    t.header.stamp = node.get_clock().now().to_msg()
    tf_pub.sendTransform(t)
node.create_timer(0.1,send_transform)
rclpy.spin(node)
```

同样也可以使用命令行获取到C和P之间关系。

```
(base) ros2@ros2-TM1613R:~$ ros2 run tf2_ros tf2_echo C P
[INFO] [1637494022.504313173] [tf2_echo]: Waiting for transform C ->  P: Invalid
 frame ID "C" passed to canTransform argument target_frame - frame does not exis
t
At time 1637494023.481877906
- Translation: [2.000, 1.000, 2.000]
- Rotation: in Quaternion [1.000, 0.000, 0.000, 0.000]
At time 1637494024.481557242
- Translation: [2.000, 1.000, 2.000]
- Rotation: in Quaternion [1.000, 0.000, 0.000, 0.000]
At time 1637494025.481980971
- Translation: [2.000, 1.000, 2.000]
- Rotation: in Quaternion [1.000, 0.000, 0.000, 0.000]
```

也可以通过命令行获取到B和P之间关系，完成手眼转换

```
(base) ros2@ros2-TM1613R:~$ ros2 run tf2_ros tf2_echo B P
[INFO] [1637494819.363532486] [tf2_echo]: Waiting for transform B ->  P: Invalid
 frame ID "B" passed to canTransform argument target_frame - frame does not exis
t
At time 1637494820.281587519
- Translation: [2.000, 3.000, 2.000]
- Rotation: in Quaternion [1.000, 0.000, 0.000, 0.000]
At time 1637494821.281523357
- Translation: [2.000, 3.000, 2.000]
- Rotation: in Quaternion [1.000, 0.000, 0.000, 0.000]
At time 1637494822.281105208
- Translation: [2.000, 3.000, 2.000]
- Rotation: in Quaternion [1.000, 0.000, 0.000, 0.000]
```

> 注意这里的结果中的时间：At time 1637494822.281105208.代表具体的某一个时刻，不同时刻坐标之间的平移和旋转可以不同

4.坐标变换监听

所谓坐标变换监听就是监听整个系统的坐标变换关系。

通过 `TransformListener` 即可获取到整个 `tf` 系统中窗口大小为10s的坐标关系，并且我们创建了一个 `buffer` ， `TransformListener` 会把收到的坐标关系放入 `buffer` 中，我们后面就可以通过 `buffer` 的 `lookup_transform()` 函数获取到坐标之间的关系。

1.导入rclpy和Node

```
In [1]: import rclpy
        from rclpy.node import Node
```

2.从tf2_ros包中导入坐标变换监听器

```
In [2]: from tf2_ros import TransformException
        from tf2_ros.buffer import Buffer
        from tf2_ros.transform_listener import TransformListener
```

3.初始化节点

```
In [3]: rclpy.init()
        node = Node("transform_node3")
```

4.初始化节点

```
In [4]: tf_buffer = Buffer()
        tf_listener = TransformListener(tf_buffer,node)
```

```
In [5]: def transform_callback():
            try:
                now = rclpy.time.Time()
                trans = tf_buffer.lookup_transform('B','P',now)
                print("trans",trans)
            except TransformException as ex:
                print(f'Could not transform B to P: {ex}')
```

```
In [*]: node.create_timer(1,transform_callback)
        rclpy.spin(node)
```

```
Could not transform B to P: "B" passed to lookupTransform argument target_frame does not exist.
Could not transform B to P: "B" passed to lookupTransform argument target_frame does not exist.
trans geometry_msgs.msg.TransformStamped(header=std_msgs.msg.Header(stamp=builtin_interfaces.msg.Time(sec=163749
7306, nanosec=381159299), frame_id='B'), child_frame_id='P', transform=geometry_msgs.msg.Transform(translation=g
eometry_msgs.msg.Vector3(x=2.0, y=3.0, z=2.0), rotation=geometry_msgs.msg.Quaternion(x=1.0, y=0.0, z=0.0, w=0.
0)))
trans geometry_msgs.msg.TransformStamped(header=std_msgs.msg.Header(stamp=builtin_interfaces.msg.Time(sec=163749
7307, nanosec=381037198), frame_id='B'), child_frame_id='P', transform=geometry_msgs.msg.Transform(translation=g
eometry_msgs.msg.Vector3(x=2.0, y=3.0, z=2.0), rotation=geometry_msgs.msg.Quaternion(x=1.0, y=0.0, z=0.0, w=0.
0)))
trans geometry_msgs.msg.TransformStamped(header=std_msgs.msg.Header(stamp=builtin_interfaces.msg.Time(sec=163749
7308, nanosec=381039805), frame_id='B'), child_frame_id='P', transform=geometry_msgs.msg.Transform(translation=g
eometry_msgs.msg.Vector3(x=2.0, y=3.0, z=2.0), rotation=geometry_msgs.msg.Quaternion(x=1.0, y=0.0, z=0.0, w=0.
0)))
trans geometry_msgs.msg.TransformStamped(header=std_msgs.msg.Header(stamp=builtin_interfaces.msg.Time(sec=163749
7309, nanosec=381050863), frame_id='B'), child_frame_id='P', transform=geometry_msgs.msg.Transform(translation=g
eometry_msgs.msg.Vector3(x=2.0, y=3.0, z=2.0), rotation=geometry_msgs.msg.Quaternion(x=1.0, y=0.0, z=0.0, w=0.
```

可以看到最终打印的结果和我们上面用命令行和numpy计算结果一致。

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

4.坐标变换发布监听C++实现

因为C++无法在jupyter中直接运行，所以小鱼将C++TF相关的代码放到这里，有需要的同学自取。

```cpp
#include <tf2/utils.h>
#include <tf2_ros/transform_broadcaster.h>


//声明&初始化发布者
std::unique_ptr<tf2_ros::TransformBroadcaster> tf_broadcaster_;
tf_broadcaster_ = std::make_unique<tf2_ros::TransformBroadcaster>(this);

//发布
geometry_msgs::msg::TransformStamped transform;
transform.header.stamp = this->now();
transform.header.frame_id = "odom";
transform.child_frame_id = "base_link";
// Fill in transform.transform.translation
transform.transform.translation.x = 0.0;
transform.transform.translation.y = 0.0;
transform.transform.translation.z =0.0;
// Fill in transform.transform.rotation
auto quat = tf2::Quaternion();
quat.setRPY(0.0;, 0.0, 0.0);
transform.transform.rotation.x = 0.0;
transform.transform.rotation.y = 0.0;
transform.transform.rotation.z = 0.0;
transform.transform.rotation.w = 0.0;
tf_broadcaster_->sendTransform(transform);
```

第 8 章 机器人建模

记得小时候，我眼中的机器人一直是钢铁侠那个样子，但后来开始学习才知道，机器人并不是都是人型的，常见的有轮式机器人和串联的工业机器人。

虽然学完了基础篇和机器人学内容，但至今你还没有在我们的课程里看到机器人的影子，请不要郁闷，因为从本章开始，我们将会对最简单的机器人开始建模，并尝试让它动起来。

本章我们将通过机器人建模语言——URDF创建简单的机器人模型，并实现在RVIZ2中的可视化。

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

1. URDF统一机器人建模语言

大家好，我是小鱼，本节课我们来介绍机器人的URDF建模。

URDF（Unified Robot Description Format）统一机器人描述格式，URDF使用XML格式描述机器人文件。

> XML是**被设计用来传输和存储数据的可扩展标记语言**，注意语言本身是没有含义的，只是规定了其数据格式
>
> 比如说下面这段信息：
>
> ```
> <robot name="阿童木">
> <link name="大手臂">具体的描述</link>
> <joint name="胳膊肘">具体的描述</joint>
> <link name="小手臂">具体的描述</link>
> </robot>
> ```
>
> XML格式在线校验工具:在线代码格式化 (oschina.net)
>
> XML格式注释： 标签: `robot` `link`
>
> robot标签的属性`name`：`<robot name="fishbot"></robot>`
>
> robot标签的子标签`link`：`<robot name="fishbot"><link name="base_link"></link>link></robot>`

使用XML定义的一个最简单的URDF模型可以像下面这样

```xml
<?xml version="1.0"?>
<robot name="fishbot">
  <link name="base_link">
    <visual>
      <geometry>
        <cylinder length="0.18" radius="0.06"/>
      </geometry>
    </visual>
```

```
    </link>
  </robot>
```

接着我们从下面四个方面介绍URDF：

1. URDF的组成介绍
2. URDF-Link介绍
3. URDF-Joint介绍
4. 创建一个简单的URDF并在RVIZ2中可视化

1.URDF的组成介绍

一般情况下，URDF由 `声明信息` 和 `两种关键组件` 共同组成

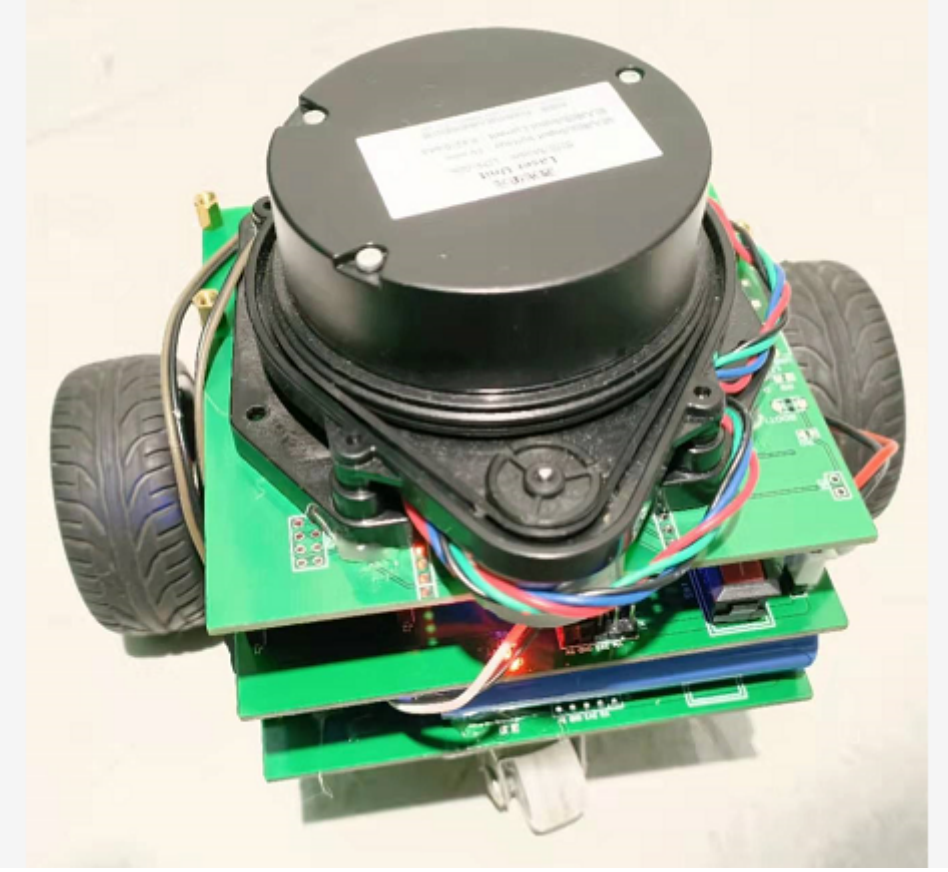
1.1 声明信息

声明信息包含两部分，第一部分是xml的声明信息，放在第一行 第二部分是机器人的声明，通过robot标签就可以声明一个机器人模型

```
<?xml version="1.0"?>
<robot name="fishbot">
     <link></link>
     <joint></joint>
  ......
</robot>
```

1.2 两种关键组件(Joint&Link)

以FishBot机器人为例分析。观察下图机器人的结构。



可以简化为如下五个部件组成：

- 躯体
- 左右轮子
- 支撑轮
- 雷达激光
- IMU模块

这五个部件之间的固定方式为：

```mermaid
graph
A[左轮] -->C[躯体]
B[右轮] -->C[躯体]
D[IMU] -->C[躯体]
E[雷达] -->C[躯体]
F[支撑轮子] -->C[躯体]
```

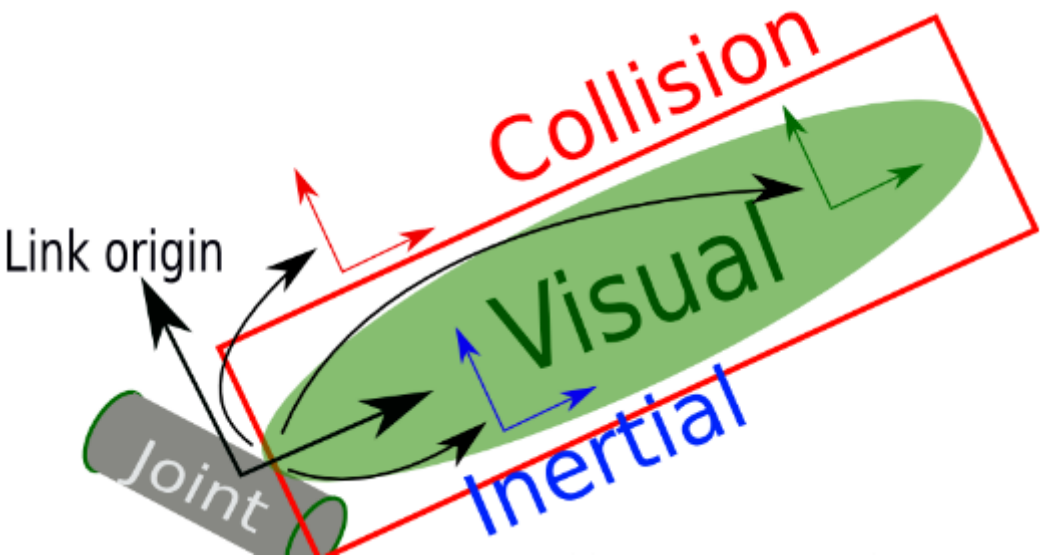
我们把左轮，右轮、支撑轮子，IMU和雷达部件称为机器人的Link

而Link和Link之间的连接部分称之为Joint关节

接着我们给每个link和joint取个名字。

```mermaid
graph
A[左轮:left_wheel_link] --left_wheel_joint-->C[躯体]
B[右轮:right_wheel_link] --right_wheel_joint-->C[躯体]
D[IMU:imu_link] --imu_joint-->C[躯体]
E[雷达:laser_link] --laser_joint-->C[躯体]
F[支撑轮子:caster_link] --caster_joint-->C[躯体:base_link]
```

所以我们就可以使用6个link和5个joint来描述这个机器人，接着我们分别对link和joint进行详细的介绍。



2.Link介绍

上面我们介绍完了link，那一个link该怎么写呢？

我们来看一个base_link的写法，通过link标签即可声明一个link,属性name指定部件名字

```xml
<link name="base_link">

</link>
```

通过两行代码就可以定义好base_link，但现在的base_link是空的，我们还要声明我们的base_link长什么样，通过 `visual` 子标签就可以声明出来机器人的visual形状。

```xml
<!-- base link -->
<link name="base_link">
    <visual>
    <origin xyz="0 0 0.0" rpy="0 0 0"/>
    <geometry>
      <cylinder length="0.12" radius="0.10"/>
    </geometry>
  </visual>
</link>
```

2.1 link标签定义

link的子标签列表

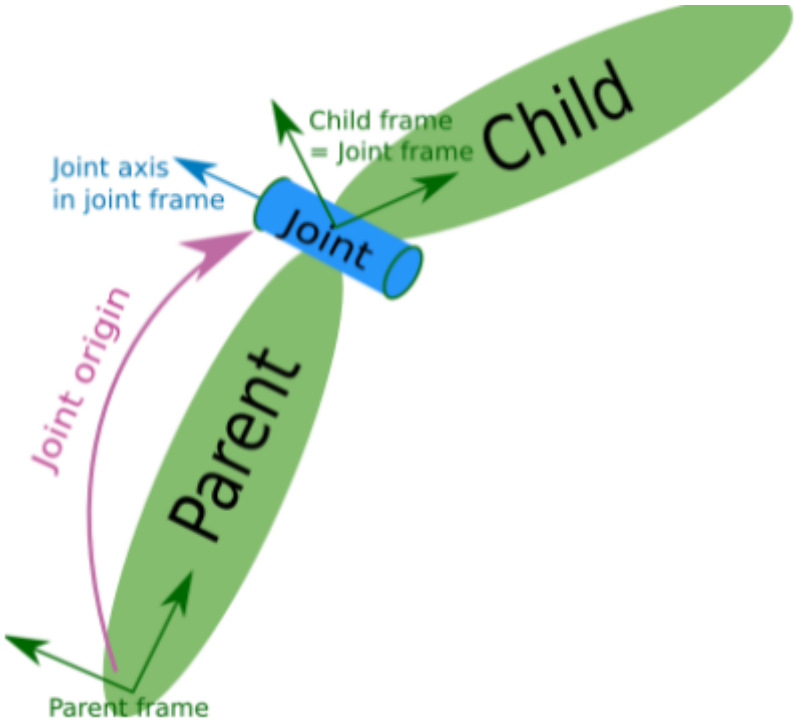
- visual 显示形状
  - `<geometry>` (几何形状)
    - `<box>` 长方体
      - 标签属性: `size` -长宽高
      - 举例： `<box size="1 1 1" />`
    - `<cylinder>` 圆柱体
      - 标签属性: `radius` -半径 `length` -高度
      - 举例： `<cylinder radius="1" length="0.5"/>`
    - `sphere` 球体
      - 属性： `radius` -半径
      - 举例： `<sphere radius="0.015"/>`
    - `mesh` 第三方导出的模型文件
      - 属性：filename
      - 举例: `<mesh filename="package://robot_description/meshes/base_link.DAE"/>`
  - origin (可选：默认在物体几何中心)
    - 属性 `xyz` 默认为零矢量 `rpy` 弧度表示的翻滚、俯仰、偏航
    - 举例： `<origin xyz="0 0 0" rpy="0 0 0" />`
  - material 材质
    - 属性 `name` 名字
      - color
        - 属性 `rgba` a代表透明度
        - 举例： `<material name="white"><color rgba="1.0 1.0 1.0 0.5" /></material>`

- collision 碰撞属性，仿真章节中讲解
- inertial 惯性参数 质量等，仿真章节中讲解

3.Joint介绍

joint为机器人关节，机器人关节用于连接两个机器人部件，主要写明父子关系

- 父子之间的连接类型，包括是否固定的，可以旋转的等
- 父部件名字
- 子部件名字
- 父子之间相对位置
- 父子之间的旋转轴，绕哪个轴转



比如我们再建立一个雷达部件 `laser_link` ，然后将 `laser_link` 固定到 `base_link`

```xml
<?xml version="1.0"?>
<robot name="fishbot">

  <!-- base link -->
  <link name="base_link">
      <visual>
```

```
      <origin xyz="0 0 0.0" rpy="0 0 0"/>
      <geometry>
        <cylinder length="0.12" radius="0.10"/>
      </geometry>
    </visual>
  </link>

  <!-- laser link -->
  <link name="laser_link">
      <visual>
      <origin xyz="0 0 0" rpy="0 0 0"/>
      <geometry>
        <cylinder length="0.02" radius="0.02"/>
      </geometry>
      <material name="black">
          <color rgba="0.0 0.0 0.0 0.5" />
      </material>
    </visual>
  </link>

  <!-- laser joint -->
    <joint name="laser_joint" type="fixed">
        <parent link="base_link" />
        <child link="laser_link" />
        <origin xyz="0 0 0.075" />
    </joint>

</robot>
```

3.1 joint标签详解

joint属性

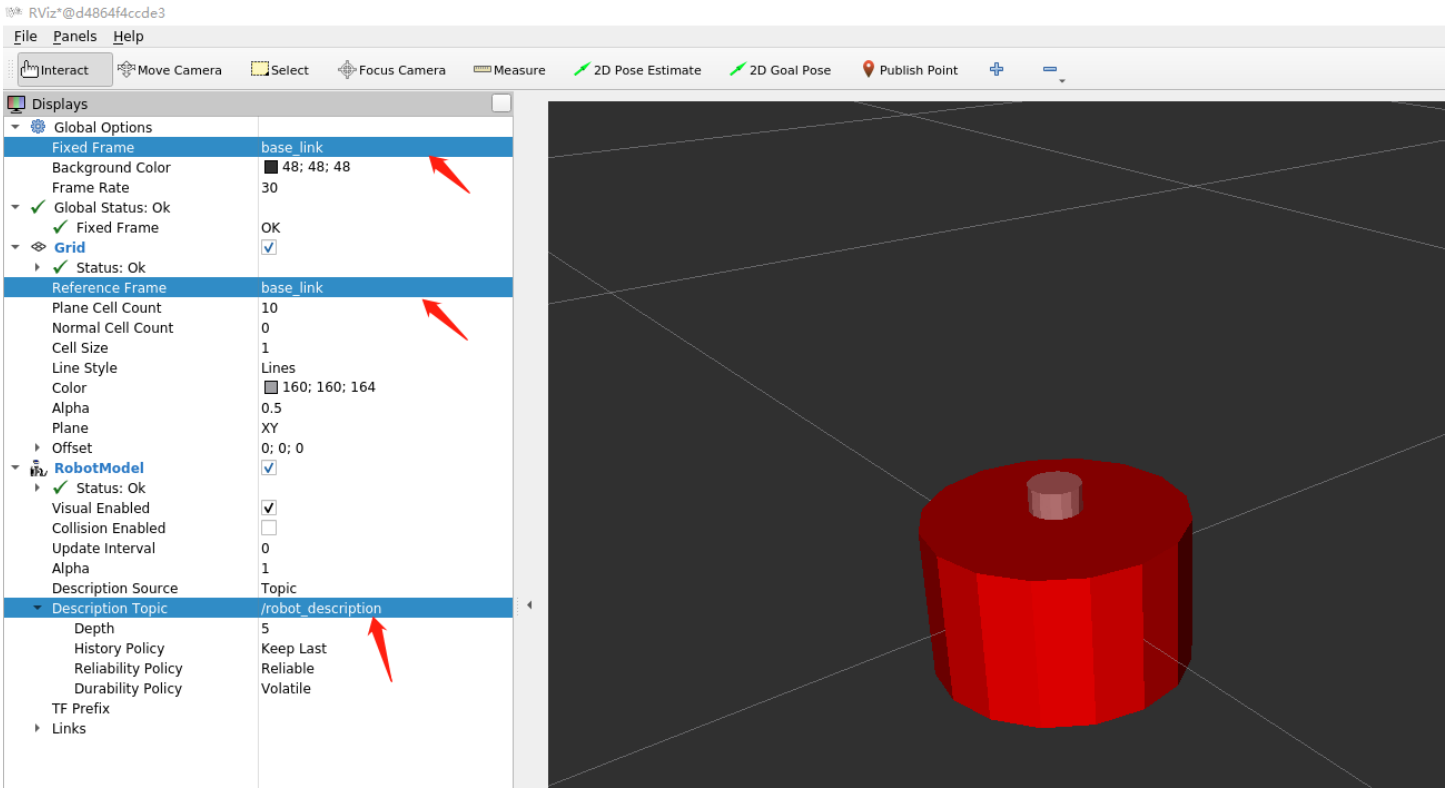
- name 关节的名称
- type 关节的类型
  - **revolute: 旋转关节，绕单轴旋转,角度有上下限,比如舵机0-180**
  - **continuous: 旋转关节，可以绕单轴无限旋转,比如自行车的前后轮**
  - **fixed: 固定关节，不允许运动的特殊关节**
  - prismatic: 滑动关节，沿某一轴线移动的关节，有位置极限

- planer: 平面关节，允许在xyz，rxryrz六个方向运动
- floating: 浮动关节，允许进行平移、旋转运动

joint的子标签

- `parent` 父link名称
  - `<parent link="base_link" />`
- `child` 子link名称
  - `<child link="laser_link" />`
- `origin` 父子之间的关系xyz rpy
  - `<origin xyz="0 0 0.014" />`
- `axis` 围绕旋转的关节轴
  - `<axis xyz="0 0 1" />`

下一节我们尝试将我们的机器人模型在RVIZ2中显示出来



参考文档

- urdf/XML/link - ROS Wiki
- http://docs.ros.org/en/humble/Tutorials/URDF/URDF-Main.html

---

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

8.2 RVIZ2可视化移动机器人模型

大家好，我是小鱼，上一节讲完joint和link，我们来把我们上面定义的简单的URDF(包含身体和雷达)用RVIZ2显示出来，直观的感受下，我们的机器人模型。

URDF可视化的步骤如下：

1. 建立机器人描述功能包
2. 建立 `urdf` 文件夹编写urdf文件
3. 建立 `launch` 文件夹，编写launch文件
4. 修改 `setup.py` 配置，编译测试

1.建立功能包

轻车熟路，先创建一个 `chapt8_ws` 工作空间，然后建立功能包，包的类型选 `ament_python`

```shell
ros2 pkg create fishbot_description --build-type ament_python
```

2.建立URDF文件

建立URDF文件夹，创建urdf文件

```
cd fishbot_description && mkdir urdf
touch fishbot_base.urdf
```

完成后src下的目录结构：

```
├── fishbot_description
│   ├── __init__.py
├── package.xml
├── setup.cfg
├── setup.py
```

```
└── urdf
    └── fishbot_base.urdf
```

编辑 `fishbot_base.urdf`

```xml
<?xml version="1.0"?>
<robot name="fishbot">

  <!-- base link -->
  <link name="base_link">
      <visual>
      <origin xyz="0 0 0.0" rpy="0 0 0"/>
      <geometry>
        <cylinder length="0.12" radius="0.10"/>
      </geometry>
    </visual>
  </link>

  <!-- laser link -->
  <link name="laser_link">
      <visual>
      <origin xyz="0 0 0" rpy="0 0 0"/>
      <geometry>
        <cylinder length="0.02" radius="0.02"/>
      </geometry>
      <material name="black">
          <color rgba="0.0 0.0 0.0 0.8" />
      </material>
    </visual>
  </link>

  <!-- laser joint -->
    <joint name="laser_joint" type="fixed">
        <parent link="base_link" />
        <child link="laser_link" />
        <origin xyz="0 0 0.075" />
    </joint>

</robot>
```

3.建立launch文件

在目录 `src/fishbot_description` 下创建 `launch` 文件夹并在其下新建 `display_rviz2.launch.py` 文件。

```
mkdir launch
touch display_rviz2.launch.py
```

完成后的目录结构：

```
├── fishbot_description
│   ├── __init__.py
├── launch
│   └── display_rviz2.launch.py
├── package.xml
├── setup.cfg
├── setup.py
└── urdf
    └── fishbot_base.urdf
```

python

```
import os
from launch import LaunchDescription
from launch.substitutions import LaunchConfiguration
from launch_ros.actions import Node
from launch_ros.substitutions import FindPackageShare


def generate_launch_description():
    package_name = 'fishbot_description'
    urdf_name = "fishbot_base.urdf"

    ld = LaunchDescription()
    pkg_share = FindPackageShare(package=package_name).find(package_name)
    urdf_model_path = os.path.join(pkg_share, f'urdf/{urdf_name}')

    robot_state_publisher_node = Node(
        package='robot_state_publisher',
```

```
            executable='robot_state_publisher',
            arguments=[urdf_model_path]
            )

        joint_state_publisher_node = Node(
            package='joint_state_publisher_gui',
            executable='joint_state_publisher_gui',
            name='joint_state_publisher_gui',
            arguments=[urdf_model_path]
            )

        rviz2_node = Node(
            package='rviz2',
            executable='rviz2',
            name='rviz2',
            output='screen',
            )

        ld.add_action(robot_state_publisher_node)
        ld.add_action(joint_state_publisher_node)
        ld.add_action(rviz2_node)

        return ld
```

想要可视化模型需要三个节点参与

- `joint_state_publisher_gui` 负责发布机器人关节数据信息，通过 `joint_states` 话题发布
- `robot_state_publisher_node` 负责发布机器人模型信息 `robot_description` ，并将 `joint_states` 数据转换tf信息发布
- `rviz2_node` 负责显示机器人的信息

```mermaid
graph
A[joint_state_publisher]--joint_states-->B
B[robot_state_publisher]--robot_description-->C
C[rviz2]
```

这里我们用到了 `joint_state_publisher_gui` 和 `robot_state_publisher` 两个包，如果你的系统没有安装这两个包，可以手动安装:

```
sudo apt install ros-$ROS_DISTRO-joint-state-publisher-gui ros-$ROS_DISTRO-robot-state-publisher
```

> `joint_state_publisher_gui`，还有一个兄弟叫做`joint_state_publisher`两者区别在于`joint_state_publisher_gui`运行起来会跳出一个界面，通过界面可以操作URDF中能动的关节

4.修改setup.py

导入头文件

```python
from glob import glob
import os
```

加入目录安装

```python
('share/ament_index/resource_index/packages',
['resource/' + package_name]),
('share/' + package_name, ['package.xml']),
```

完整

```python
from setuptools import setup
from glob import glob
import os

package_name = 'fishbot_description'

setup(
    name=package_name,
    version='0.0.0',
    packages=[package_name],
    data_files=[
        ('share/ament_index/resource_index/packages',
```

```
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
        (os.path.join('share', package_name, 'launch'),
glob('launch/*.launch.py')),
        (os.path.join('share', package_name, 'urdf'), glob('urdf/**')),
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='root',
    maintainer_email='root@todo.todo',
    description='TODO: Package description',
    license='TODO: License declaration',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
        ],
    },
)
```

5.编译测试

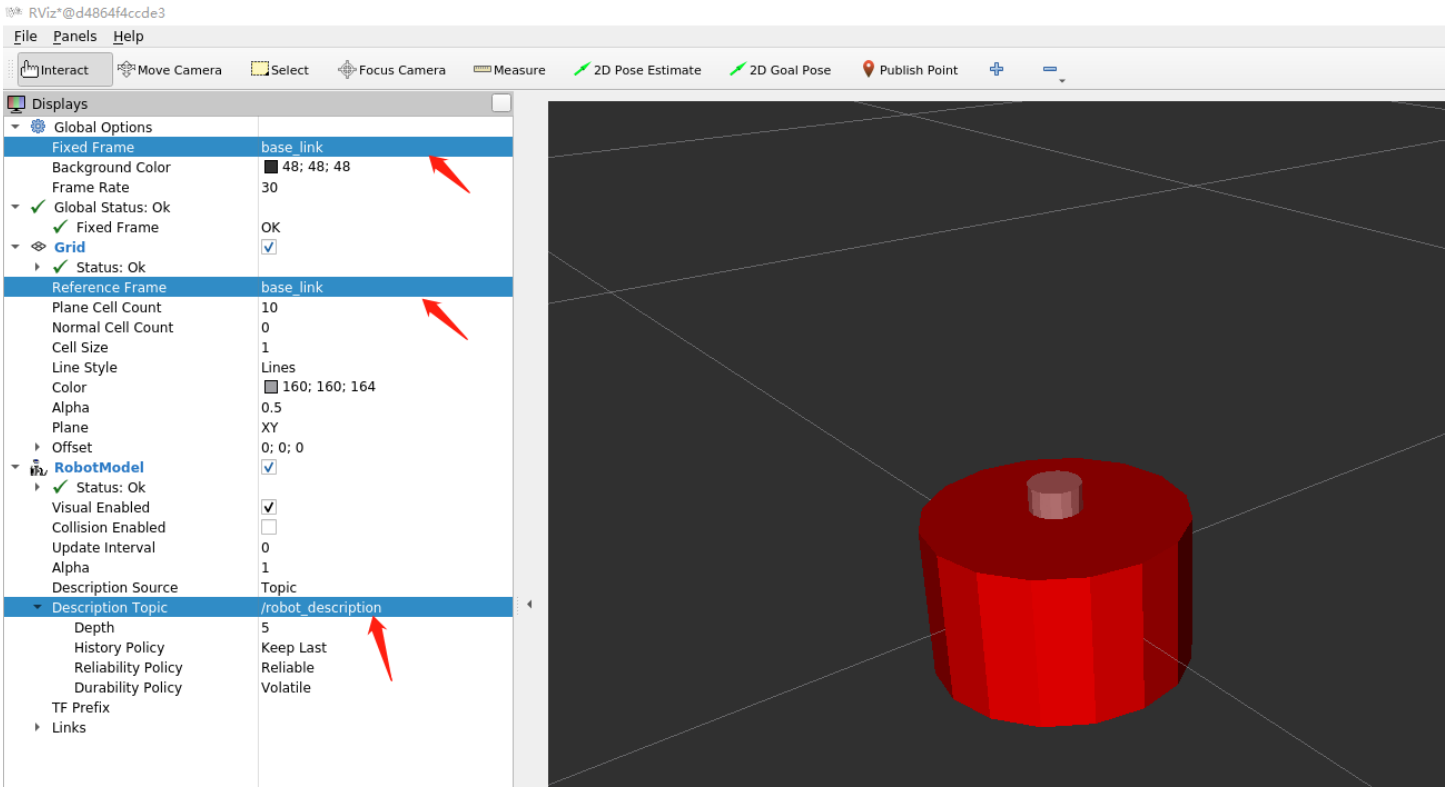
编译

```
colcon build
```

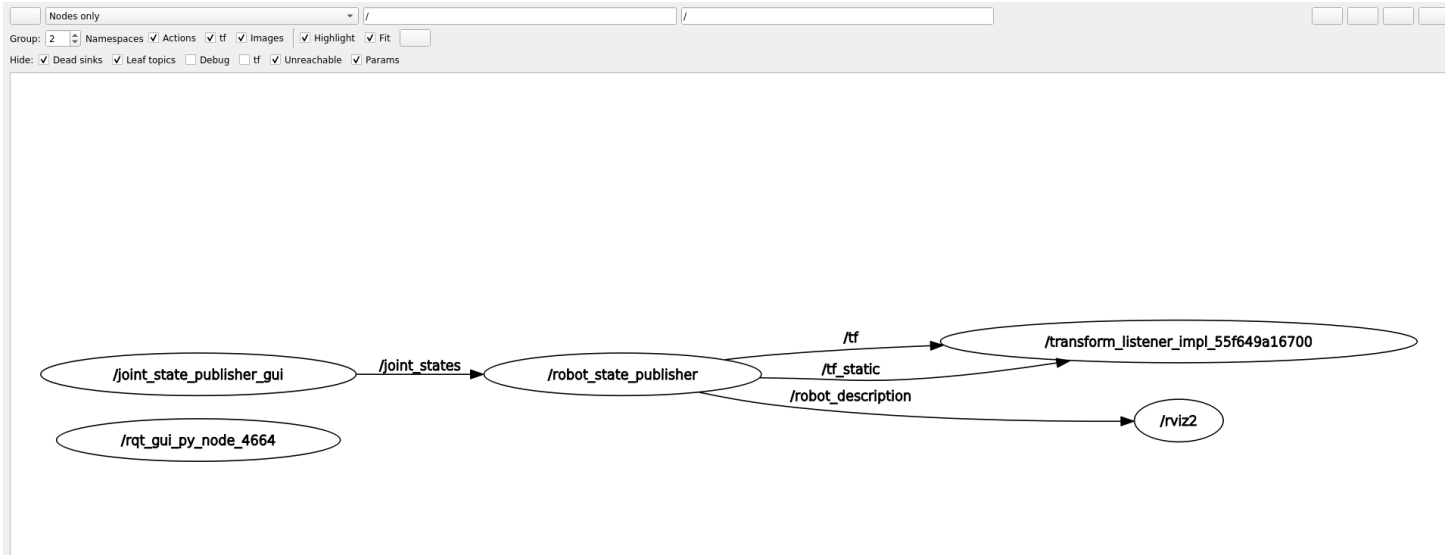
运行测试

```
source install/setup.bash
ros2 launch fishbot_description display_rviz2.launch.py
```

添加robotmodel模块，分别选择link名称如下，即可看到机器人的模型显示

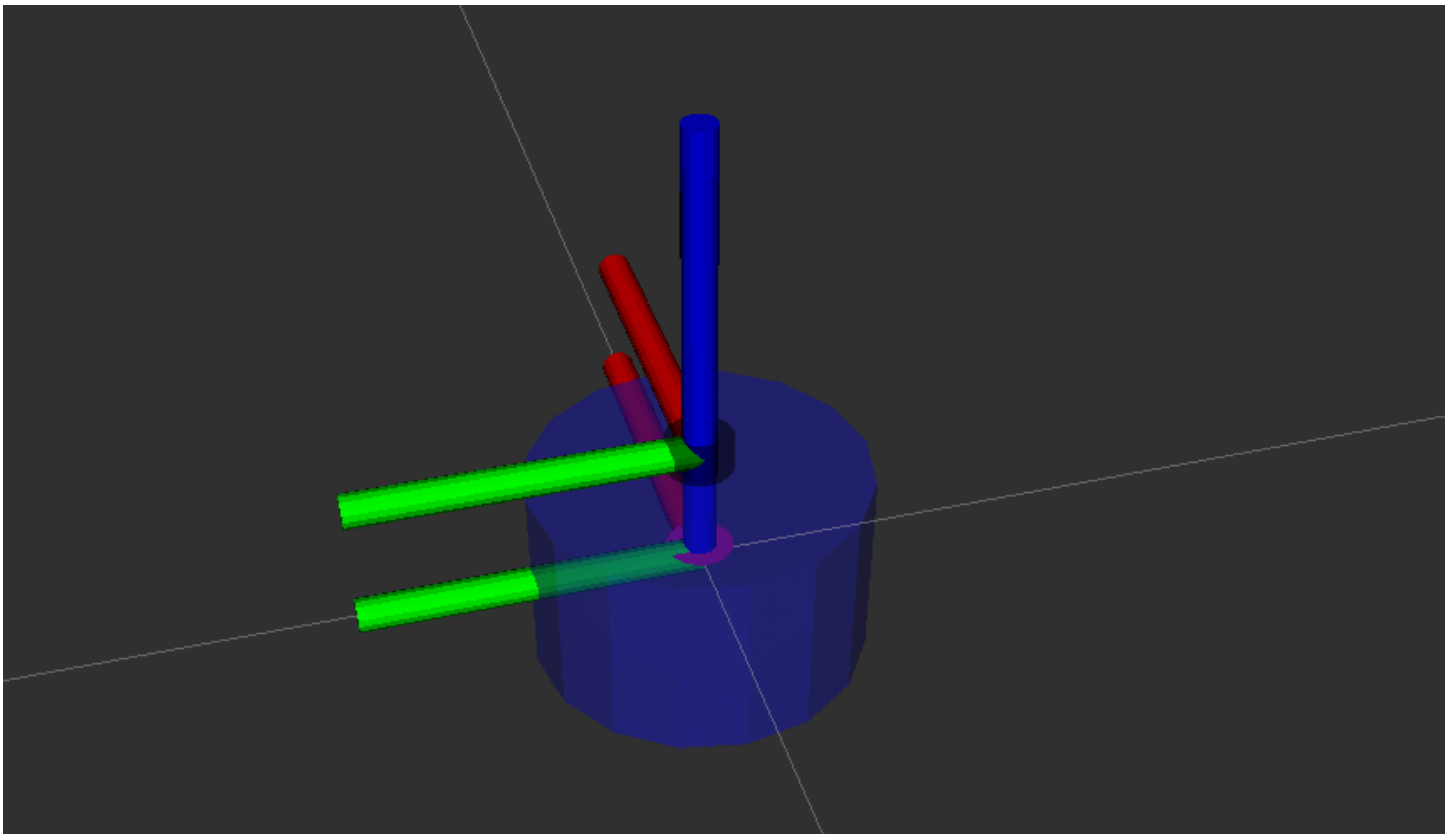


此时看看节点关系图



这里大家可以参考图理一理launch文件中启动的三个节点的关系。

然后打开TF模块，看一下机器人的坐标系关系

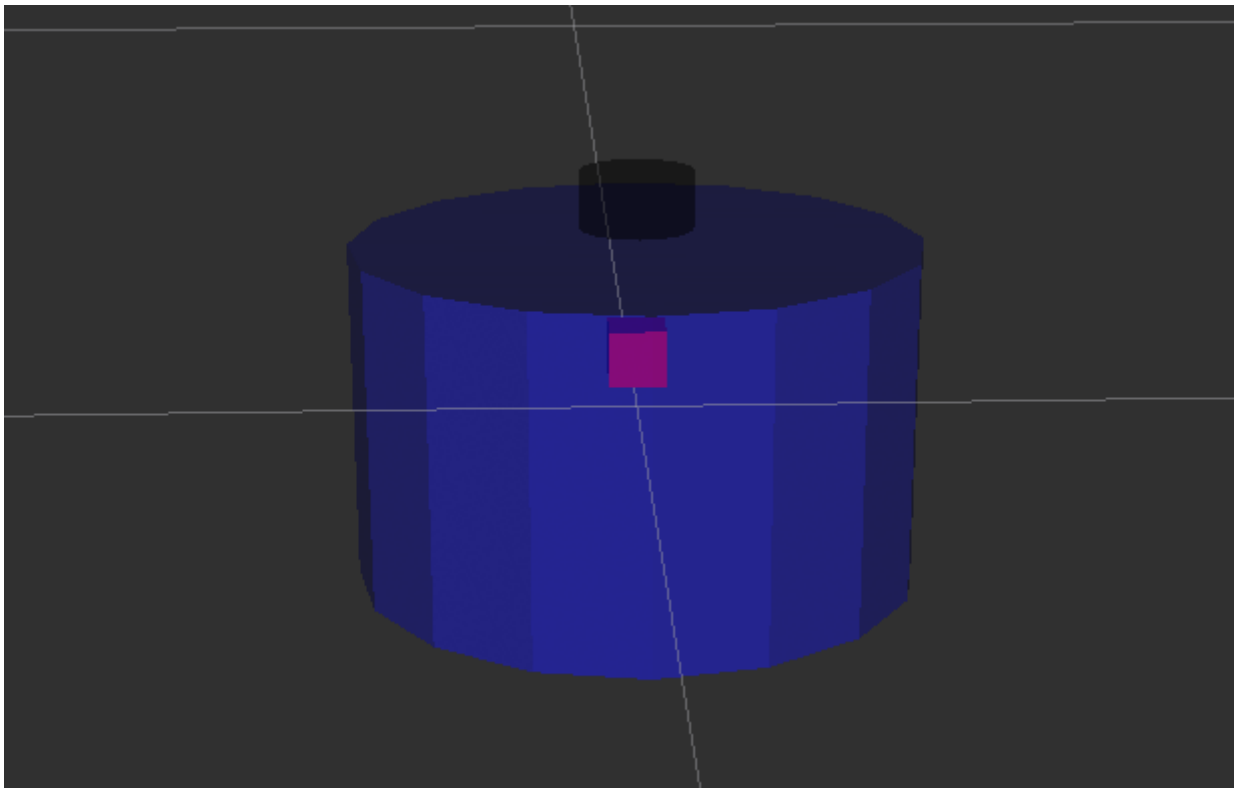


6.本节练习

练习1：尝试将修改机器人身体颜色为蓝色，透明度为50% `(0.1 0.1 1.0 0.5)`

练习2：尝试在URDF中添加imu_link并使用imu_joint将其固定在车体的中心上方2cm，imu采用的几何形状为box。长宽高均为2cm

结果展示：



技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

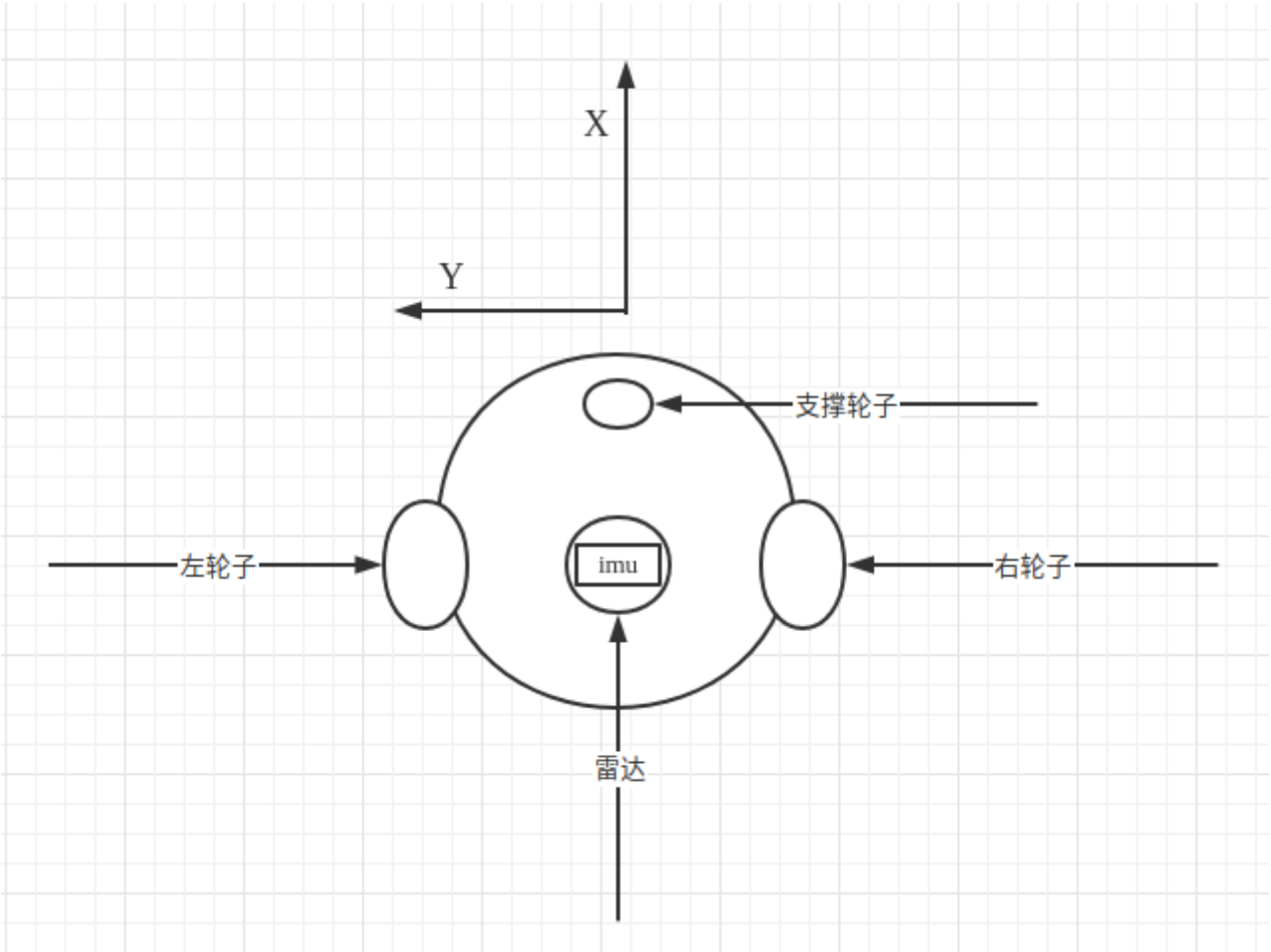
8.3 动手创建一个移动机器人

大家好，我是小鱼，本节课我们来一起动手创建一个两轮差速的移动机器人 `fishbot` ,在上一节的时候我们已经给我们的机器人安装上了雷达，本节课我们接着上节课的来继续完善我们的机器人模型。

机器人除了雷达之外，还需要 `IMU` 加速度传感器以及可以驱动的轮子，在第七章中我们曾介绍过机器人学部分，曾对两差速模型进行过介绍，所以我们还需要再创建两个差速驱动轮和一个支撑轮。

所以本节小鱼带你一起给机器人添加如下部件和关节：

1. IMU传感器部件与关节
2. 左轮子部件与关节
3. 右轮子部件与关节
4. 支撑轮子部件与关节



1.添加IMU传感器(上节作业)

IMU传感器和透明度与颜色修改是上节课，作业，小鱼先带你一起完成一下

> 练习1：尝试将修改机器人身体颜色为蓝色，透明度为50% `(0.1 0.1 1.0 0.5)`
>
> 练习2：尝试在URDF中添加imu_link并使用imu_joint将其固定在车体的中心上方2cm，imu采用的几何形状为box,长宽高各是2cm

1.1 修改颜色

透明度修改只需要在 `base_link` 中添加 `material`

```xml
<link name="base_link">
    <visual>
        <origin xyz="0 0 0.0" rpy="0 0 0"/>
    <geometry>
        <cylinder length="0.12" radius="0.10"/>
    </geometry>
    <material name="blue">
        <color rgba="0.1 0.1 1.0 0.5" />
    </material>
    </visual>
</link>
```

1.2 添加imu

```xml
  <link name="imu_link">
      <visual>
      <origin xyz="0 0 0.0" rpy="0 0 0"/>
      <geometry>
            <box size="0.02 0.02 0.02"/>
      </geometry>
    </visual>
  </link>

  <!-- imu joint -->
  <joint name="imu_joint" type="fixed">
      <parent link="base_link" />
      <child link="imu_link" />
```

```
        <origin xyz="0 0 0.02" />
    </joint>
```

2.添加右轮

2.1 添加关节

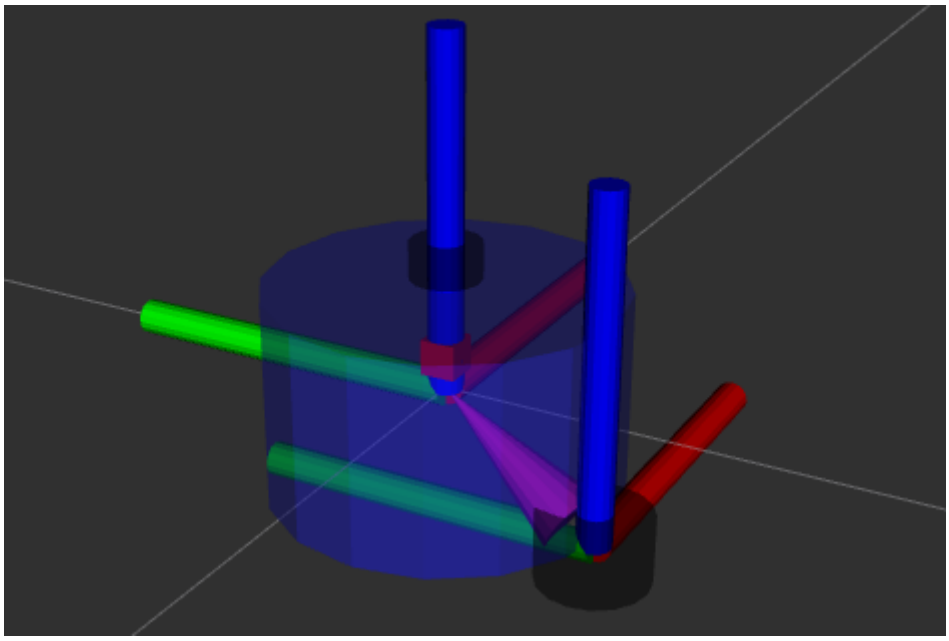
关节名称为 `right_wheel_link` ,小鱼在做ros2小车的时候采用的轮子如下图：



轮子的宽为4cm,直径为6.4cm,几何形状是个圆柱体，所以 `geometry` 配置如下：

```xml
<geometry>
    <cylinder length="0.04" radius="0.032"/>
</geometry>
```

需要注意的是,圆柱默认的朝向是向上的



我们可通过 `origin` 的 `rpy` 改变轮子的旋转角度，让其绕 `x轴` 旋转 `pi/2` ,所以 `origin` 的配置为

```xml
<origin xyz="0 0 0" rpy="1.57079 0 0"/>
```

颜色换黑色，可以得到下面的配置：

```xml
<link name="right_wheel_link">
    <visual>
      <origin xyz="0 0 0" rpy="1.57079 0 0"/>
      <geometry>
        <cylinder length="0.04" radius="0.032"/>
      </geometry>
        <material name="black">
          <color rgba="0.0 0.0 0.0 0.5" />
        </material>
    </visual>
</link>
```

2.2 添加joint

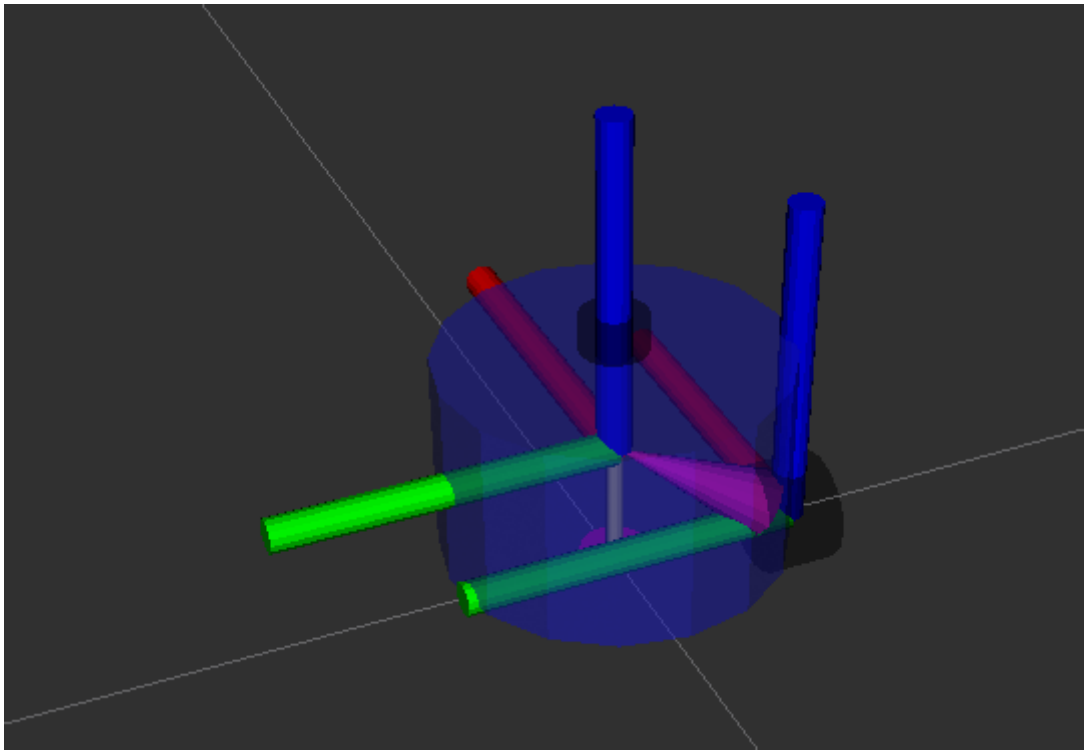
我们把左轮子的中心固定在机器人左后方

需要注意的是 `origin` 和 `axis` 值的设置

先看origin

因为base_link的高度是0.12,我们

- z表示child相对parent的z轴上的关系，想将轮子固定在机器人的下表面,所以 `origin` 的z向下偏移0.12/2=0.06m(向下符号为负)
- y表示child相对parent的y轴上的关系，base_link的半径是0.10,所以我们让轮子的y轴向负方向偏移0.10m(向左符号为负)
- x表示child相对parent的x轴上的关系，向后偏移则是x轴向后进行偏移，我们用个差不多的值0.02m(向后符号为负)



再看axis

轮子是会转动的，那应该按照哪个轴转动呢？从上图可以看出是绕着y轴的逆时针方向，所以axis的设置为：

```xml
<axis xyz="0 1 0" />
```

```xml
  <joint name="right_wheel_joint" type="continuous">
      <parent link="base_link" />
      <child link="right_wheel_link" />
      <origin xyz="-0.02 -0.10 -0.06" />
```

```
        <axis xyz="0 1 0" />
    </joint>
```

3.添加左轮

左轮就是右轮的映射，不再赘述

```xml
    <link name="left_wheel_link">
        <visual>
          <origin xyz="0 0 0" rpy="1.57079 0 0"/>
          <geometry>
            <cylinder length="0.04" radius="0.032"/>
          </geometry>
            <material name="black">
              <color rgba="0.0 0.0 0.0 0.5" />
            </material>
        </visual>
    </link>

    <joint name="left_wheel_joint" type="continuous">
        <parent link="base_link" />
        <child link="left_wheel_link" />
        <origin xyz="-0.02 0.10 -0.06" />
        <axis xyz="0 1 0" />
    </joint>
```

4.添加支撑轮

支撑轮子固定在机器人的前方，用个球体，半径用 `0.016m` ，小球的直径为0.032m与左右轮子半径相同，然后向下偏移 `0.016+0.06=0.076m` ,向下值为负，同时把支撑论向前移动一些，选个 `0.06m`

最终结果如下：

```xml
   <link name="caster_link">
       <visual>
         <origin xyz="0 0 0" rpy="0 0 0"/>
```

```
        <geometry>
            <sphere radius="0.016"/>
        </geometry>
          <material name="black">
            <color rgba="0.0 0.0 0.0 0.5" />
          </material>
      </visual>
    </link>

    <joint name="caster_joint" type="fixed">
        <parent link="base_link" />
        <child link="caster_link" />
        <origin xyz="0.06 0.0 -0.076" />
    </joint>
```

最终URDF文件

```
<?xml version="1.0"?>
<robot name="fishbot">

  <!-- base link -->
  <link name="base_link">
      <visual>
      <origin xyz="0 0 0.0" rpy="0 0 0"/>
      <geometry>
        <cylinder length="0.12" radius="0.10"/>
      </geometry>
      <material name="blue">
          <color rgba="0.1 0.1 1.0 0.5" />
      </material>
    </visual>
  </link>

  <!-- laser link -->
  <link name="laser_link">
      <visual>
      <origin xyz="0 0 0" rpy="0 0 0"/>
      <geometry>
        <cylinder length="0.02" radius="0.02"/>
      </geometry>
```

```
    <material name="black">
        <color rgba="0.0 0.0 0.0 0.5" />
    </material>
  </visual>
</link>

<!-- laser joint -->
<joint name="laser_joint" type="fixed">
    <parent link="base_link" />
    <child link="laser_link" />
    <origin xyz="0 0 0.075" />
</joint>

<link name="imu_link">
    <visual>
    <origin xyz="0 0 0.0" rpy="0 0 0"/>
    <geometry>
          <box size="0.02 0.02 0.02"/>
    </geometry>
  </visual>
</link>

<!-- imu joint -->
<joint name="imu_joint" type="fixed">
    <parent link="base_link" />
    <child link="imu_link" />
    <origin xyz="0 0 0.02" />
</joint>


<link name="left_wheel_link">
    <visual>
      <origin xyz="0 0 0" rpy="1.57079 0 0"/>
      <geometry>
        <cylinder length="0.04" radius="0.032"/>
      </geometry>
        <material name="black">
          <color rgba="0.0 0.0 0.0 0.5" />
        </material>
    </visual>
</link>
```

```
    <joint name="left_wheel_joint" type="continuous">
        <parent link="base_link" />
        <child link="left_wheel_link" />
        <origin xyz="-0.02 0.10 -0.06" />
        <axis xyz="0 1 0" />
    </joint>

    <link name="right_wheel_link">
        <visual>
          <origin xyz="0 0 0" rpy="1.57079 0 0"/>
          <geometry>
            <cylinder length="0.04" radius="0.032"/>
          </geometry>
            <material name="black">
              <color rgba="0.0 0.0 0.0 0.5" />
            </material>
        </visual>
    </link>

    <joint name="right_wheel_joint" type="continuous">
        <parent link="base_link" />
        <child link="right_wheel_link" />
        <origin xyz="-0.02 -0.10 -0.06" />
        <axis xyz="0 1 0" />
    </joint>

    <link name="caster_link">
      <visual>
        <origin xyz="0 0 0" rpy="1.57079 0 0"/>
        <geometry>
            <sphere radius="0.016"/>
        </geometry>
          <material name="black">
            <color rgba="0.0 0.0 0.0 0.5" />
          </material>
      </visual>
    </link>

    <joint name="caster_joint" type="fixed">
        <parent link="base_link" />
        <child link="caster_link" />
        <origin xyz="0.06 0.0 -0.076" />
```

```
        <axis xyz="0 1 0" />
    </joint>



</robot>
```

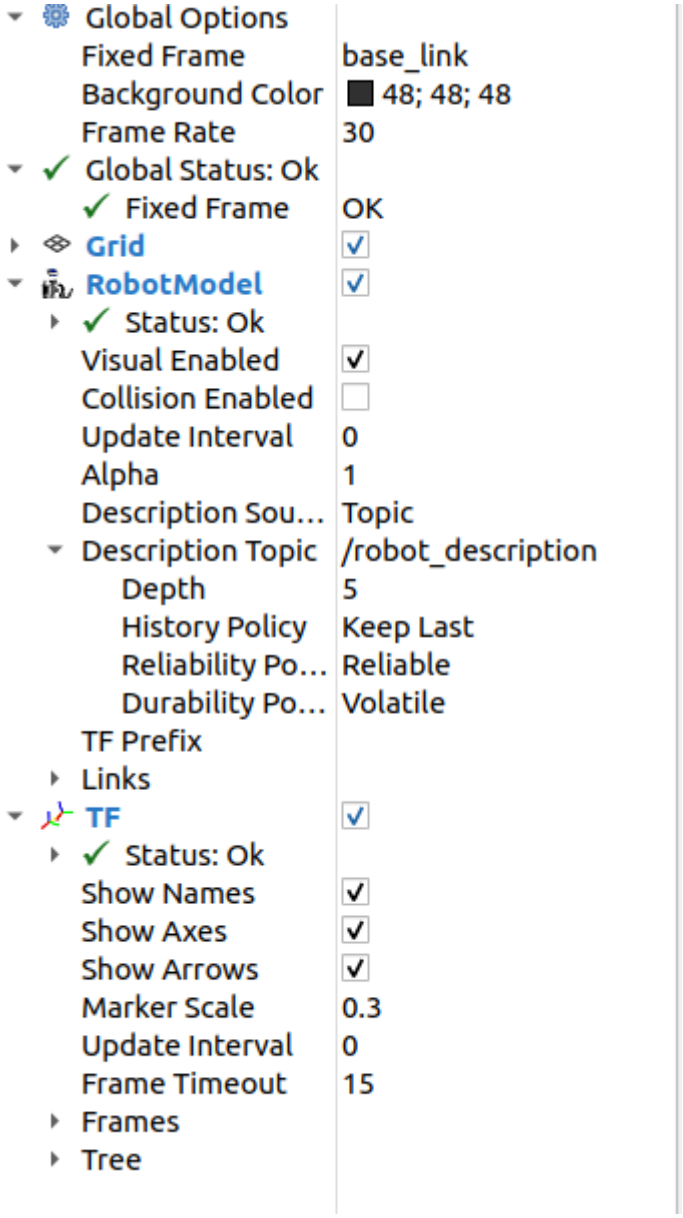
5.测试运行

5.1 编译测试

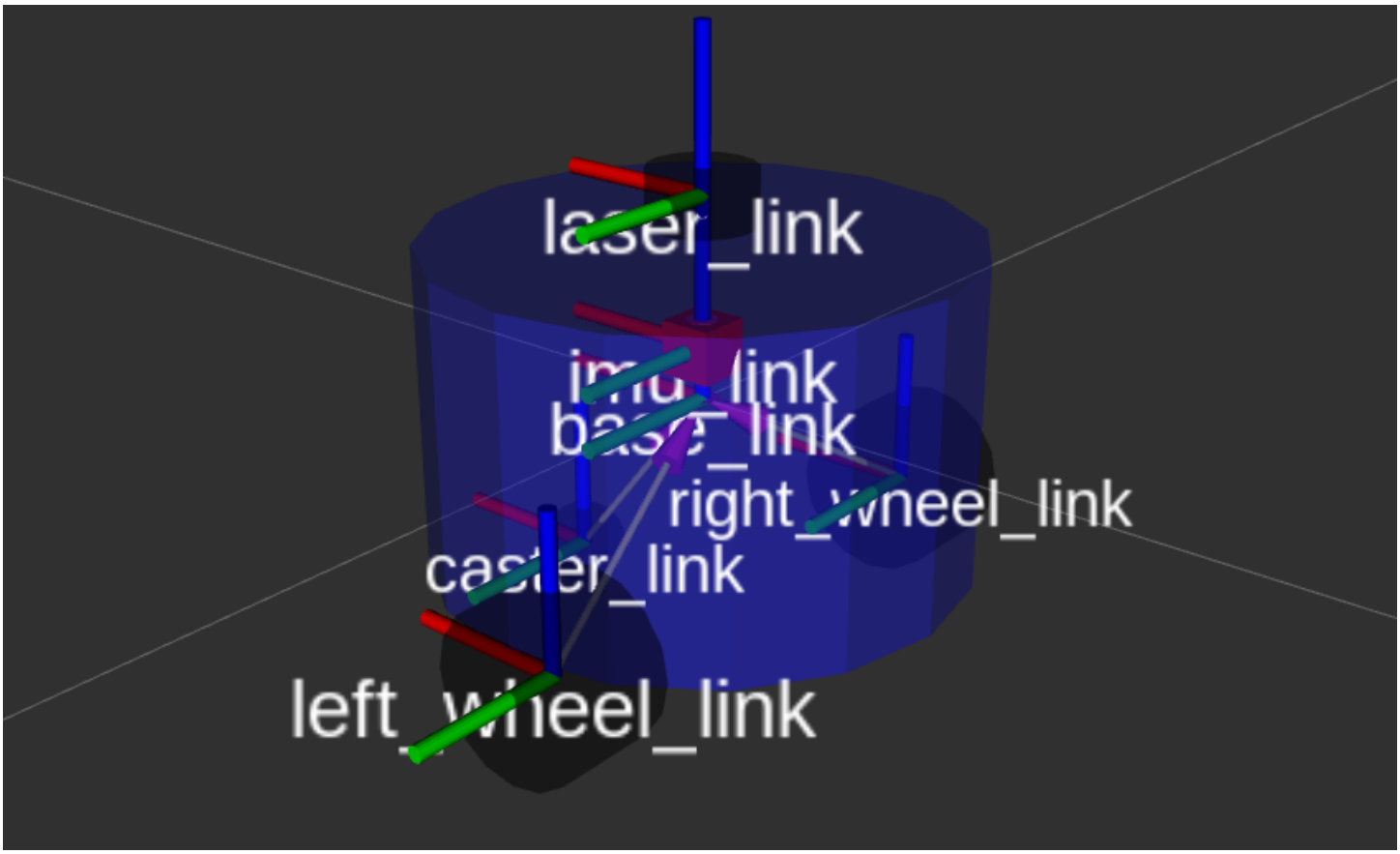
```shell
colcon build
source install/setup.bash
ros2 launch fishbot_description display_rviz2.launch.py
```

5.2 最终结果

rviz的配置


Global Options
Fixed Frame
base_link
Background Color
48; 48; 48
Frame Rate
30
Global Status: Ok
Fixed Frame
OK
Grid
RobotModel
Status: Ok
Visual Enabled
Collision Enabled
Update Interval
0
Alpha
1
Description Sou…
Topic
Description Topic
/robot_description
Depth
5
History Policy
Keep Last
Reliability Po…
Reliable
Durability Po…
Volatile
TF Prefix
Links
TF
Status: Ok
Show Names
Show Axes
Show Arrows
Marker Scale
0.3
Update Interval
0
Frame Timeout
15
Frames
Tree

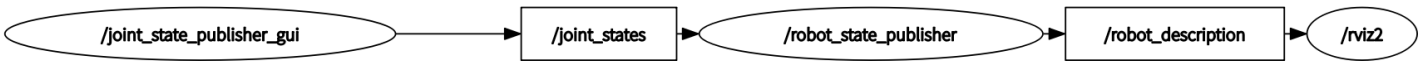
最终结果



jointstate多出两个滑动条

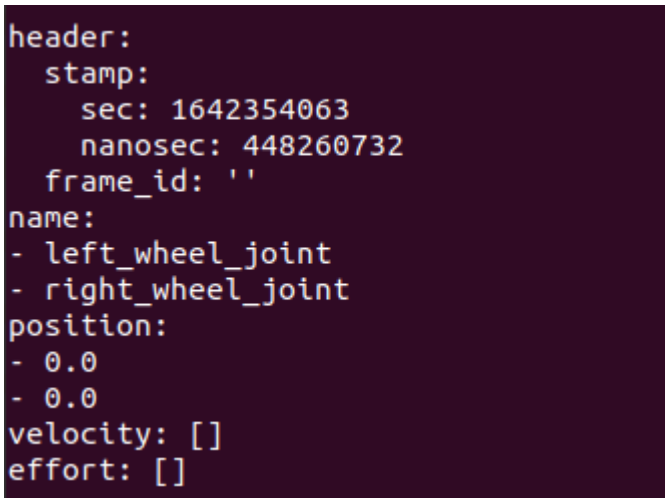


节点关系



打印`joint_states`话题

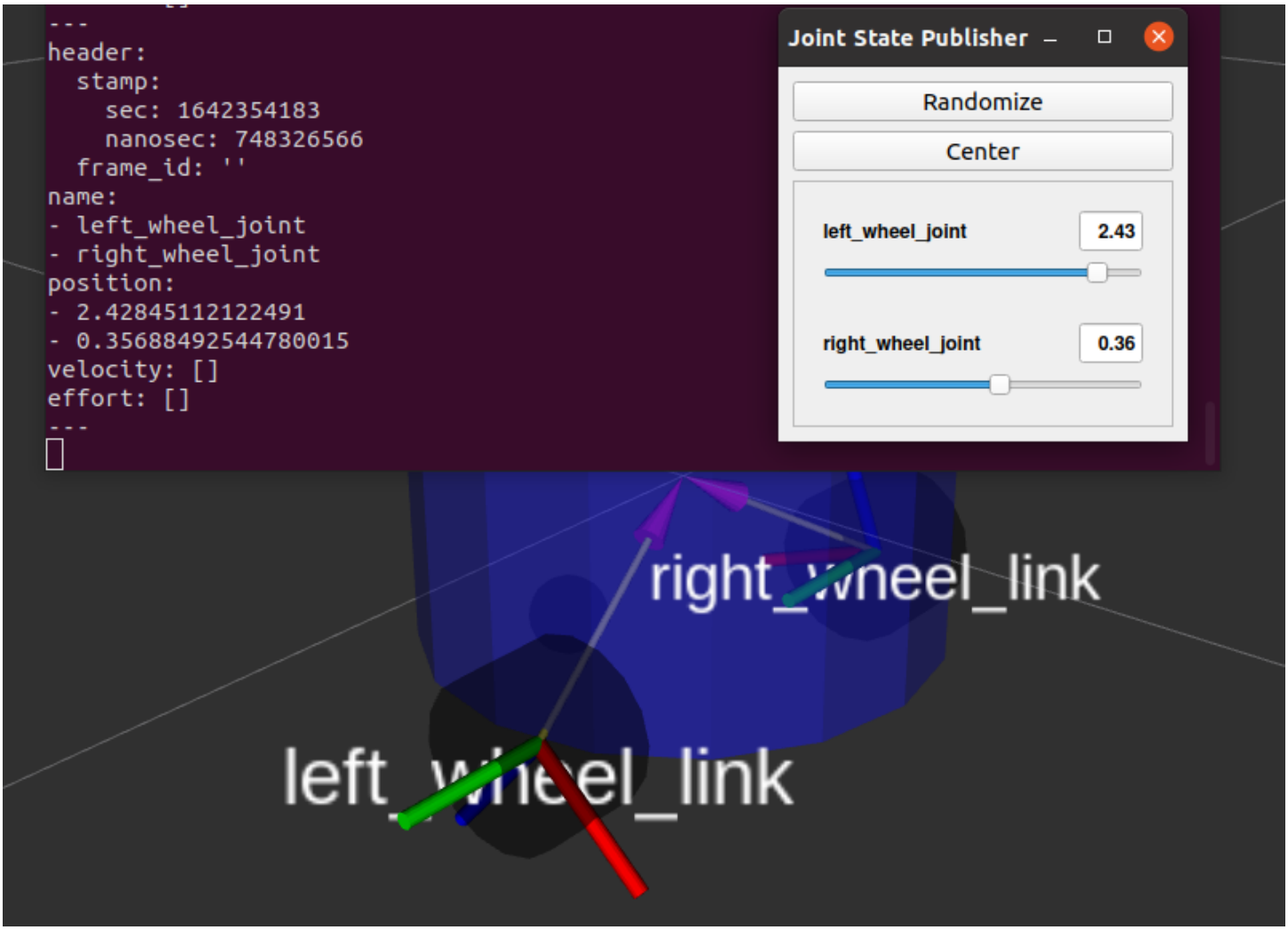
```
ros2 topic echo /joint_states
```



5.3 通过joint_state_gui改变关节tf中关节角度

在JointStatePublisher中,拖动滑动条,观察

1. `rviz2` 中 `tf` 的变换
2. `joint_states` 中的值的变换



可以看到随着进度条拖动,话题中的值和rviz2中机器人关节在同步的旋转,joint_states话题也可以手动发送,下一节课小鱼带你一起通过手动发送 `joint_states` 来控制机器人轮子转动

5.4 论如何让车轮着地

虽然显示出了机器人模型，但有一个问题不知道你发现没有，那就是在RVIZ中的机器人轮子是在地面之下的。

原因在于我们fixed-frame选择的是base_link,base_link的位置本来就在left_wheel_link和right_wheel_link只上，那该怎么办呢？

其实很简单，我们增加一个虚拟link和关节，这个关节与base_link相连，位置位于base_link向下刚好到车轮下表面的位置。

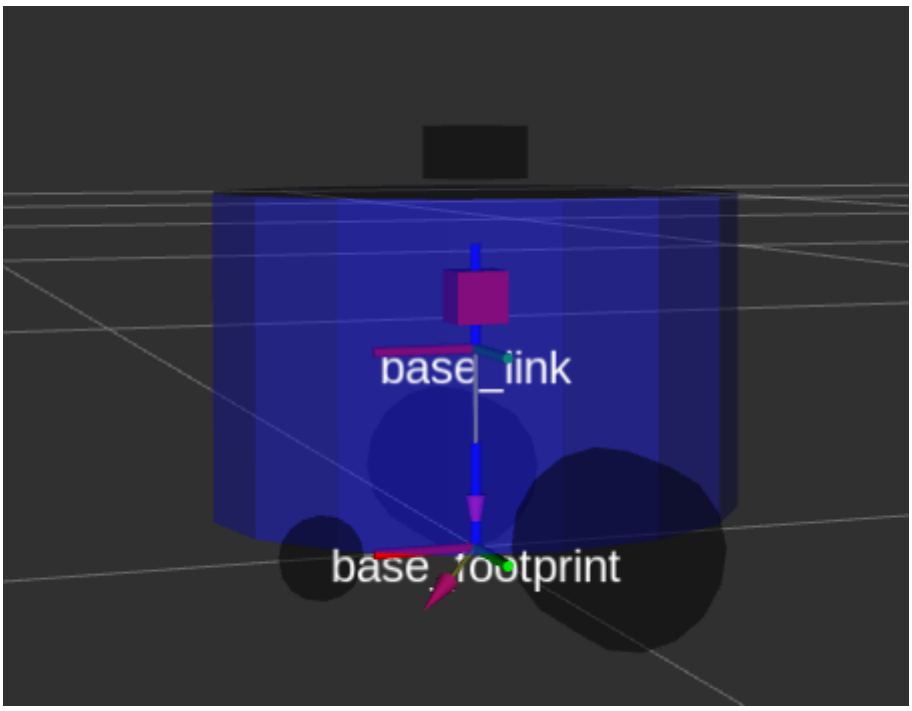
来，让我们**给base_link添加一个父link- `base_footprint` **，新增的URDF代码如下：

```
    <!-- Robot Footprint -->
    <link name="base_footprint"/>
```

```
    <joint name="base_joint" type="fixed">
      <parent link="base_footprint"/>
      <child link="base_link"/>
      <origin xyz="0.0 0.0 0.076" rpy="0 0 0"/>
    </joint>
```

因为是虚拟关节，我们不用对这个link的形状进行描述，joint的origin设置为 `xyz="0.0 0.0 0.076"` 表示关节base_footprint向上0.076就是base_link（觉得不好理解可以看下图）。

保存编译再次运行测试，此时车轮就在地面只上啦~

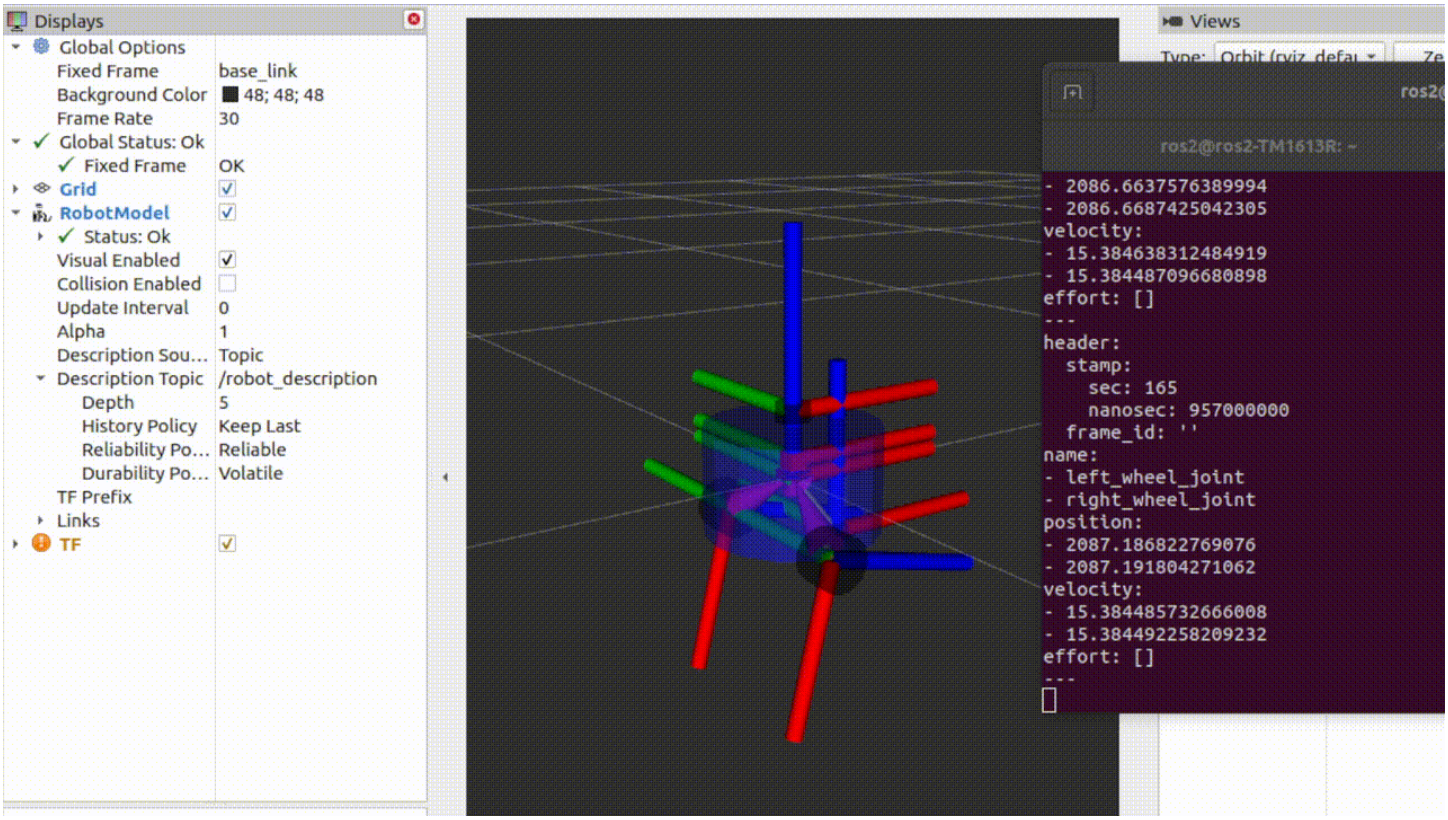


技术交流&&问题求助：

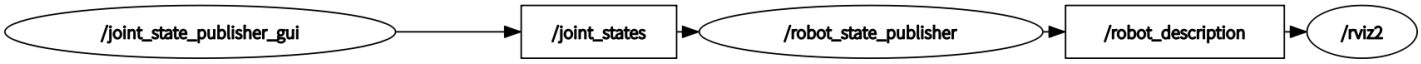
- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

8.4 控制移动机器人轮子运动

我是小鱼,本节我们来看看如何手动的发送 `joint_states` 来控制机器人轮子连续转动



要实现上图效果，我们需要自己编写节点,取代 `joint_state_publisher` 发送关节位姿给 `robot_state_pubsher` ，robot_state_publisher发送tf控制机器人的关节转动。



1.新建节点

2.创建发布者

3.编写发布逻辑

4.编译测试

1.新建节点

方便起见，我们就在 `fishbot_describle` 包中新建节点（参考李四节点代码）

```shell
cd fishbot_ws
touch fishbot_description/fishbot_description/rotate_wheel.py
```

```python
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node


class RotateWheelNode(Node):
    def __init__(self,name):
        super().__init__(name)
        self.get_logger().info(f"node {name} init..")

def main(args=None):
    """
    ros2运行该节点的入口函数
    1. 导入库文件
    2. 初始化客户端库
    3. 新建节点
    4. spin循环节点
    5. 关闭客户端库
    """
    rclpy.init(args=args) # 初始化rclpy
    node = RotateWheelNode("rotate_fishbot_wheel")  # 新建一个节点
    rclpy.spin(node) # 保持节点运行，检测是否收到退出指令（Ctrl+C）
    rclpy.shutdown() # 关闭rclpy
```

配置下setup.py

```
    entry_points={
        'console_scripts': [
            "rotate_wheel= fishbot_description.rotate_wheel:main"
        ],
    },
```

编译运行

```
colcon build
source install/setup.bash
ros2 run fishbot_description rotate_wheel
```

```
(base) root@d4864f4ccde3:~/code/fishbot_ws# source install/setup.bash
(base) root@d4864f4ccde3:~/code/fishbot_ws# ros2 run fishbot_description rotate_wheel
[INFO] [1643083477.660927790] [rotate_fishbot_wheel]: node rotate_fishbot_wheel init..
```

2.创建发布者

创建发布者之前，要知道`robot_state_pubsher`所订阅的话题类型是什么？

回忆前面章节中学习的内容，我们可以采用如下指令查看

```
ros2 topic info /joint_states
```

```
Type: sensor_msgs/msg/JointState
Publisher count: 1
Subscription count: 1
```

接着

```
ros2 interfaces show sensor_msgs/msg/JointState
```

```
# This is a message that holds data to describe the state of a set of torque
controlled joints.
#
# The state of each joint (revolute or prismatic) is defined by:
#  * the position of the joint (rad or m),
#  * the velocity of the joint (rad/s or m/s) and
#  * the effort that is applied in the joint (Nm or N).
#
```

```
# Each joint is uniquely identified by its name
# The header specifies the time at which the joint states were recorded. All the
joint states
# in one message have to be recorded at the same time.
#
# This message consists of a multiple arrays, one for each part of the joint
state.
# The goal is to make each of the fields optional. When e.g. your joints have no
# effort associated with them, you can leave the effort array empty.
#
# All arrays in this message should have the same size, or be empty.
# This is the only way to uniquely associate the joint name with the correct
# states.

std_msgs/Header header

string[] name
float64[] position
float64[] velocity
float64[] effort
```

知道了话题类型，我们就可以来创建发布者了.

python

```python
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
# 1.导入消息类型JointState
from sensor_msgs.msg import JointState

class RotateWheelNode(Node):
    def __init__(self,name):
        super().__init__(name)
        self.get_logger().info(f"node {name} init..")
        # 2.创建并初始化发布者成员属性pub_joint_states_
        self.pub_joint_states_ = self.create_publisher(JointState,"joint_states",
10)
```

3.编写发布逻辑

3.1 多线程定频发布Rate

创建好发布者，我们想让话题按照某个固定的速度进行发布，可以采用ROS2中的定时神器Rate,不清楚Rate的小伙伴可以看看小鱼的这篇文章：ROS中的定频神器你会用吗

为了能够一直循环使用rate，我们单独开一个线程用于发布joint_states话题数据，在ROS2程序中单独开线程进行话题发布的方法为：

```python
import threading
from rclpy.node import Node

class RotateWheelNode(Node):
    def __init__(self):
        # 创建一个Rate和线程
        self.pub_rate = self.create_rate(5) #5Hz
        # 创建线程
        self.thread_ = threading.Thread(target=self._thread_pub)
        self.thread_.start()

    def _thread_pub(self):
        while rclpy.ok():
            #做一些操作，使用rate保证循环频率
            self.pub_rate.sleep()
```

3.2 构造发布数据

接着我们来构造发布的数据：

joint_states有一个头和四个数组需要赋值（可通过ros2 interface指令查询）

```
std_msgs/Header header #时间戳信息 和 frame_id
string[] name
float64[] position
```

```
float64[] velocity
float64[] effort
```

对应的含义为：

```
# 这是一个持有数据的信息，用于描述一组扭矩控制的关节的状态。
#
# 每个关节（渐进式或棱柱式）的状态由以下因素定义。
# #关节的位置（rad或m）。
# #关节的速度（弧度/秒或米/秒）和
# #在关节上施加的力（Nm或N）。
#
# 每个关节都由其名称来唯一标识
# 头部规定了记录关节状态的时间。所有的联合状态
# 必须是在同一时间记录的。
#
# 这个消息由多个数组组成，每个部分的联合状态都有一个数组。
# 目标是让每个字段都是可选的。例如，当你的关节没有
# 扭矩与它们相关，你可以让扭矩数组为空。
#
# 这个信息中的所有数组都应该有相同的大小，或者为空。
# 这是唯一能将关节名称与正确的
# 状态。
string[] name #关节名称数组
float64[] position #关节位置数组
float64[] velocity #关节速度数组
float64[] effort #扭矩数据
```

3.2.1 name

name是关节的名称，要与urdf中的定义的关节名称相同，根据我们的URDF定义有

```python
self.joint_states.name = ['left_wheel_joint','right_wheel_joint']
```

3.2.2 position

表示关节转动的角度值，因为关节类型为 `continuous` ,所以其值无上下限制，初始值赋值为0.0

```python
# 关节的位置
self.joint_states.position = [0.0,0.0]
```

我们采用速度控制机器人轮子转动，所以机器人的位置更新则可以通过下面式子计算得出

> 某一段时间内轮子转动的角度 = （当前时刻-上一时刻）*两个时刻之间的轮子转速

```python
delta_time =  time.time()-last_update_time
# 更新位置
self.joint_states.position[0]  += delta_time*self.joint_states.velocity[0]
self.joint_states.position[1]  += delta_time*self.joint_states.velocity[1]
```

3.2.3 velocity

因为我们采用速度进行控制，所以对外提供一个速度更改接口。

```python
def update_speed(self,speeds):
    self.joint_speeds = speeds
```

3.3 完成后代码

```python
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
# 1.导入消息类型JointState
from sensor_msgs.msg import JointState

import threading
import time

class RotateWheelNode(Node):
```

```
    def __init__(self,name):
        super().__init__(name)
        self.get_logger().info(f"node {name} init..")
        # 创建并初始化发布者成员属性pub_joint_states_
        self.joint_states_publisher_ =
self.create_publisher(JointState,"joint_states", 10)
        # 初始化数据
        self._init_joint_states()
        self.pub_rate = self.create_rate(30)
        self.thread_ = threading.Thread(target=self._thread_pub)
        self.thread_.start()


    def _init_joint_states(self):
        # 初始左右轮子的速度
        self.joint_speeds = [0.0,0.0]
        self.joint_states = JointState()
        self.joint_states.header.stamp = self.get_clock().now().to_msg()
        self.joint_states.header.frame_id = ""
        # 关节名称
        self.joint_states.name = ['left_wheel_joint','right_wheel_joint']
        # 关节的位置
        self.joint_states.position = [0.0,0.0]
        # 关节速度
        self.joint_states.velocity = self.joint_speeds
        # 力
        self.joint_states.effort = []

    def update_speed(self,speeds):
        self.joint_speeds = speeds

    def _thread_pub(self):
        last_update_time = time.time()
        while rclpy.ok():
            delta_time =  time.time()-last_update_time
            last_update_time = time.time()
            # 更新位置
            self.joint_states.position[0]  +=
delta_time*self.joint_states.velocity[0]
            self.joint_states.position[1]  +=
delta_time*self.joint_states.velocity[1]
            # 更新速度
```

```
            self.joint_states.velocity = self.joint_speeds
            # 更新 header
            self.joint_states.header.stamp = self.get_clock().now().to_msg()
            # 发布关节数据
            self.joint_states_publisher_.publish(self.joint_states)
            self.pub_rate.sleep()

def main(args=None):
    rclpy.init(args=args) # 初始化rclpy
    node = RotateWheelNode("rotate_fishbot_wheel")  # 新建一个节点
    node.update_speed([15.0,-15.0])
    rclpy.spin(node) # 保持节点运行，检测是否收到退出指令（Ctrl+C）
    rclpy.shutdown() # 关闭rclpy
```

4.编译测试

编译程序

```
colcon build --packages-select fishbot_description
```

此时运行关节数据发布节点

```
ros2 run fishbot_description rotate_wheel
```

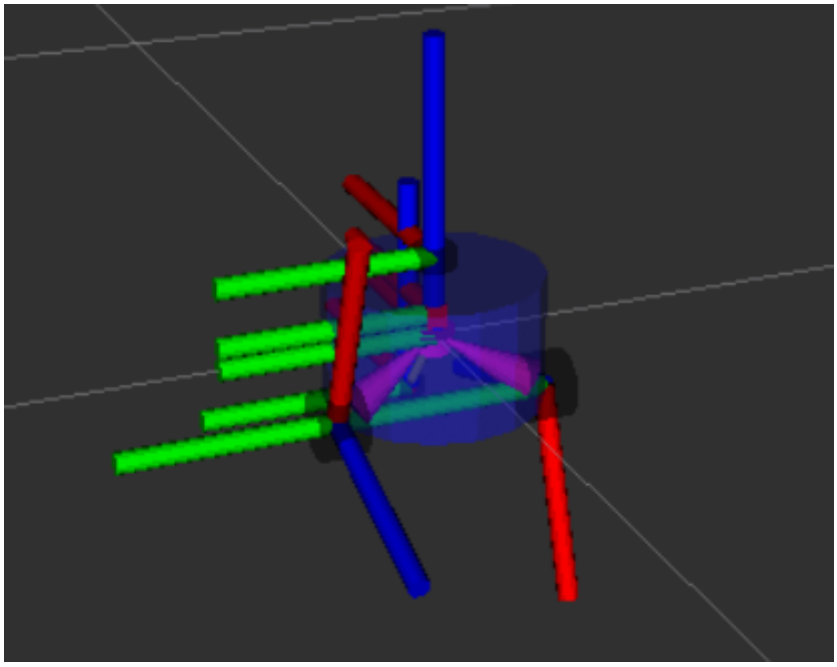
测试之前还需要修改下 `display_rviz2.launch.py` 文件，注释其 `joint_state_publisher` 节点

```
# ld.add_action(joint_state_publisher_node)
ld.add_action(robot_state_publisher_node)
ld.add_action(rviz2_node)
```

先运行rviz和robot_state_publisher

```
source install/setup.bash
ros2 launch fishbot_description display_rviz2.launch.py
```

观察此时rviz界面，可以看到轮子疯狂转动



5.课后练习

尝试将左右轮速度参数化，然后尝试采用rqt动态参数配置工具，实时控制轮子的转速。

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- c 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

9.1 机器人仿真介绍

大家好，我是小鱼，本节课我们主要对机器人仿真进行介绍，同时对ROS2支持的常用仿真平台进行介绍。

1.为什么需要机器人仿真

所谓机器人仿真其实就是通过软件来模仿硬件的特性，用于验证机器人算法、架构等。

肯定有同学会问为什么要做机器人仿真，有这个用软件来模拟硬件的空挡，直接搞真实的机器人不行吗?

答案肯定是可以的，对于个人极客来说，直接动手比仿真要来的快和直接的多。单对于公司和非常复杂的项目来说，仿真就变得很重要了。

原因在于：

1. 仿真可以解决真机资源不足，真实的机器人一般价格都很贵，搭建起来也很耗费资源，小鱼是从一家创业公司一路走来的，不同部门抢机器人的现象时常发生。
2. 仿真可以保证环境的一致和稳定，举个例子，之前小鱼在部署导航系统时发现在A机器人上没问题，但在B机器人上老是丢位置，明明算法是一致的，后来发现是B机器人的IMU模块出现了松动。
3. 仿真场景可以更加灵活，在测试机器人算法时可以通过仿真软件快速更改仿真环境，验证算法（甚至还可以让机器人原地起飞）

2.仿有哪些缺点

上面说完仿真的各种好，那仿真有哪些缺陷呢？

> **之前听一位做机械臂动力学的朋友说，他们做研发时候从来没做过仿真，原因在于仿真环境中的机器人和真实环境机器人差别过大。**

所以机器人仿真的主要缺陷就是仿不全，现实世界中的环境非常复杂，光线、材质、电磁干扰等等，仿真平台无法做到100%的仿真。

3.常用仿真平台

相较于ROS，ROS2支持更多更专业的仿真平台，常用的有：

3.1 Gazebo

> 官网链接：https://www.cyberbotics.com/

Gazebo是ROS中常用的机器人仿真平台，也是OSRF（开源机器人基金会）的作品之一，关于Gazebo的介绍，小鱼已经在第六章进行介绍了，忘记的同学可以翻看下。

3.2 WeBots

> 官网链接：https://www.cyberbotics.com/

Webots由Cyberbotics公司开发，是一个用于模拟机器人的开源和多平台桌面应用程序。它提供了一个完整的开发环境来对机器人进行建模、编程和仿真。Webots内核基于开源动力学引擎ODE和OpenGL，可以在Windows，Linux和macOS上运行，并且支持多种编程语言(C/C++，Python，Java，MATLAB)。

3.3 Ignition

> 官网链接：https://gazebosim.org/ 官方文档:https://ignitionrobotics.org/docs

Ignition是继承于Gazebo的下一代仿真平台，Ignition Robotics基于开发库和云服务等丰富全面的工具箱，提供了一种全新的仿真方式，进一步简化仿真。高度逼真的传感器可在接近真实的环境中快速迭代更新机器人物理设计。在安全上可测试控制策略，并在持续的集成化侧重中利用仿真的诸多优势。

3.4 Unity

> 官网链接：hhttps://unity.com/ UnityForROS2:https://github.com/RobotecAI/ros2-for-unity

Unity Robotics软件包带有许多现成的接口，能让你轻松与ROS或ROS 2的交换信息。你也能用URDF Importer直接从URDF文件中导入机器人配置，在Unity高质量的渲染管线与高精度的物理模拟加持下开始训练机器人。Unity的Asset Store还售有大量现成的环境和道具，可用于补充机器人的训练环境、完善训练任务。只需几键，你搭建的模拟就可以构建并部署到Windows 10、Mac OS或Linux等任意主流操作系统。你甚至可以使用C#、Bolt可视化编程及Asset Store上的众多脚本和实用程序来根据自己的需求进一步定制模拟环境。

4.我们用哪个仿真平台？

鉴于从ROS和Gazebo中得到的大量经验，我们依然选择Gazebo作为本次进行仿真的主要平台。但在本章节的后面，小鱼会带你一起探索下其他的仿真平台。

让我们保持好奇心，下一节在上一章节机器人建模的基础上，为我们的机器人模型注入仿真需要的物理属性参数。

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**

2.为机器人URDF模型注入物理属性

上节我们知道，机器人仿真就是用软件来模拟硬件的特性，那么我们必须要告诉仿真平台机器人各个关节的物理属性，比如：

- 有多重，
- 有多大的惯性
- 重心在哪
- 碰撞边界在哪
- 关节的上下界限
- 其他的一些必要信息等等

所以这节课小鱼就带你将物理信息写入到urdf中，让机器人在gazebo中显示出来。

1.需要哪些物理信息？

一般来说有 `碰撞` 和 `内参` 两个就够了，但是因为小鱼之前的偷懒，还要加一个摩擦力配置。

碰撞描述是物体的用于碰撞检测的包围形状。内参用于描述物体的质量，惯性矩阵。link的摩擦力。

1.1 碰撞检测

在机器人仿真中，我们要对物体之前是否接触，是否发生碰撞做检测，常用的检测方法比如包围盒，判断两个物体的包围盒是否相交来快速判断物体是否发生碰撞。

在URDF中，我们可以可以在link标签下添加collison子标签来对物体的形状进行描述。

collision可以包含的子标签如下：

- origin，表示碰撞体的中心位姿
- geometry，用于表示用于碰撞检测的几何形状
- `material` ，可选的，描述碰撞几何体的材料(这个设置可以在gazebo仿真时通过view选项看到碰撞包围体的形状)

一个完整的collision标签实例如下：

```xml
    <collision>
      <origin xyz="0 0 0.0" rpy="0 0 0"/>
      <geometry>
```

```
          <cylinder length="0.12" radius="0.10"/>
        </geometry>
        <material name="blue">
            <color rgba="0.1 0.1 1.0 0.5" />
        </material>
      </collision>
```

1.2 旋转惯量

旋转惯量矩阵是用于描述物体的惯性的，在做动力学仿真的时候，这些参数尤为重要。

在URDF中我们可以通过在link下添加inertial子标签，为link添加惯性参数的描述。

intertial标签包含的子标签如下：

- mass，描述link的质量
- inertia，描述link的旋转惯量（该标签有六个属性值ixx\ixy\ixz\iyy\iyz\izz）

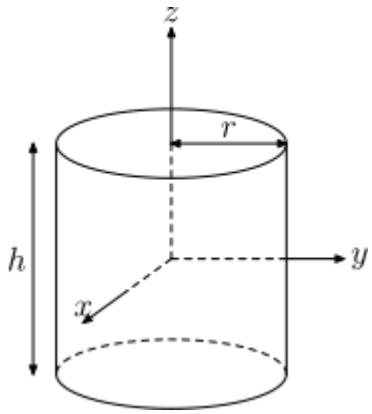
一个完整的inertial标签示例如下：

```
    <inertial>
       <mass value="0.2"/>
       <inertia ixx="0.0122666" ixy="0" ixz="0" iyy="0.0122666" iyz="0"
  izz="0.02"/>
     </inertial>
```

关于intertial的属性设置，不是随意设置的，常见的几何体我们可以通过公式进行计算。计算方法可以看小鱼的这篇文章-URDF仿真惯性参数不知道怎么配？快收藏，常见几何物体URDF分享。

比如我们上一章节的fishbot的轮子和车体，都是实心圆柱，可以采用下面的公式进行计算：

> 注意：这个矩阵是一个对称矩阵，所以只需要通过其上三角即可描述完整描述这个矩阵，所以在URDF中只需要填写六个数字即可。 实心圆柱体的惯性矩阵:半径为r，高度为h，质量为m 的实心圆柱体 形状：



矩阵：

$$I = \begin{bmatrix} \frac{1}{12}m(3r^2+h^2) & 0 & 0 \\ 0 & \frac{1}{12}m(3r^2+h^2) & 0 \\ 0 & 0 & \frac{1}{2}mr^2 \end{bmatrix}$$

1.3 摩擦力和刚性系数

在Fishbot的URDF中，前面的支撑轮主要起支撑作用，因为我们将其使用fixed标签固定到了base_link上，所以它无法转动。

哪该怎么办呢？小鱼教你一个取巧的办法，我们可以要把这个轮子的摩擦力设置为0，让它直接在地上滑动即可，

如何设置呢？6行代码放到URDF中：

```
<gazebo reference="caster_link">
  <mu1 value="0.0"/>
  <mu2 value="0.0"/>
  <kp value="1000000.0" />
  <kd value="10.0" />
</gazebo>
```

其中mu1,mu2代表摩擦力，kp,kd代表刚性系数。

2.为FishBot添加物理惯性

利用上面的方法公式，为我们的fishbot哥哥link添加好物理属性，完成后的base_link如下：

```xml
  <!-- base link -->
  <link name="base_link">
      <visual>
      <origin xyz="0 0 0.0" rpy="0 0 0"/>
      <geometry>
        <cylinder length="0.12" radius="0.10"/>
      </geometry>
      <material name="blue">
          <color rgba="0.1 0.1 1.0 0.5" />
      </material>
    </visual>
    <collision>
      <origin xyz="0 0 0.0" rpy="0 0 0"/>
      <geometry>
        <cylinder length="0.12" radius="0.10"/>
      </geometry>
      <material name="blue">
          <color rgba="0.1 0.1 1.0 0.5" />
      </material>
    </collision>
    <inertial>
      <mass value="0.2"/>
      <inertia ixx="0.0122666" ixy="0" ixz="0" iyy="0.0122666" iyz="0"
izz="0.02"/>
    </inertial>
  </link>
```

完全添加好的机器人URDF模型小鱼已经放到了这里：[fishbot_gazebo.urdf]

可以将小鱼配置好的模型下载到 `src/c/urdf` 文件夹下，等下我们要用gazebo将该模型显示出来。

3.使用gazebo加载URDF

1.Gazebo-ROS2插件介绍

在第六章中小鱼曾介绍过，gazebo是独立于ROS/ROS2之外的仿真软件，我们可以独立使用Gazebo。如果我们想要通过ROS2和Gazebo进行交互，需要通过gazebo_ros插件来进行。

接下来小鱼先带你通过命令行的形式来启动gazebo-ros2插件以及使用插件提供的服务来将fishbot的urdf模型在gazebo中显示出来。

1.1 安装Gazebo插件

```shell
sudo apt install ros-humble-gazebo-ros
```

1.2 启动Gazebo并启动插件

安装完成后，我们就可以通过下面的命令行来启动gazebo并加载ros2插件。

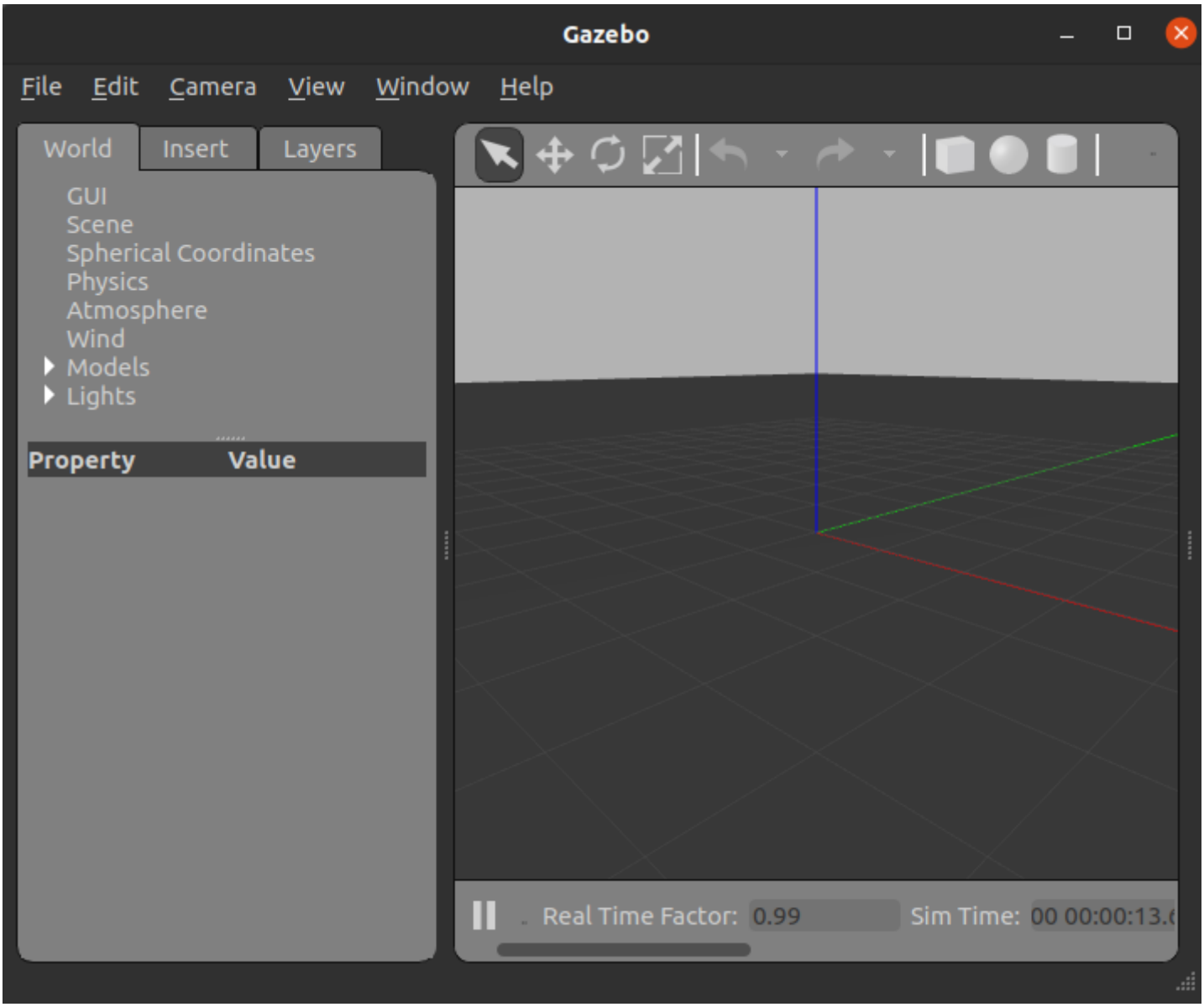
```shell
gazebo --verbose -s libgazebo_ros_init.so -s libgazebo_ros_factory.so
```

看到下面的日志和Gazebo界面代表启动成功

```
Gazebo multi-robot simulator, version 11.9.0
Copyright (C) 2012 Open Source Robotics Foundation.
Released under the Apache 2 License.
http://gazebosim.org

[Msg] Waiting for master.
Gazebo multi-robot simulator, version 11.9.0
Copyright (C) 2012 Open Source Robotics Foundation.
Released under the Apache 2 License.
http://gazebosim.org
```

```
[Msg] Waiting for master.
[Msg] Connected to gazebo master @ http://127.0.0.1:11345
[Msg] Publicized address: 192.168.2.103
[Msg] Loading world file [/usr/share/gazebo-11/worlds/empty.world]
[INFO] [1649151283.208884022] [gazebo_ros_node]: ROS was initialized without
arguments.
[Msg] Connected to gazebo master @ http://127.0.0.1:11345
[Msg] Publicized address: 192.168.2.103
```



2.插件节点及其服务介绍

使用3.1中的指令启动Gazebo并加载gazebo_ros插件，我们使用下面的指令来看插件的节点，以及改节点为我们提供的服务有哪些？

节点列表

```shell
ros2 node list
```

正确返回

```
/gazebo
```

然后我们看看这个节点对外提供的服务有哪些？

```
ros2 service list
```

```
/delete_entity
/get_model_list
/spawn_entity
/gazebo/describe_parameters
/gazebo/get_parameter_types
/gazebo/get_parameters
/gazebo/list_parameters
/gazebo/set_parameters
/gazebo/set_parameters_atomically
```

除去和参数相关的几个服务，我们可以看到另外三个特殊服务：

- /spawn_entity，用于加载模型到gazebo中
- /get_model_list，用于获取模型列表
- /delete_entity，用于删除gazbeo中已经加载的模型

我们想要让gazebo显示出我们配置好的fishbot使用/spawn_entity来加载即可。

接着我们可以来请求服务来加载模型，小鱼先带你看一下服务的接口类型。

```shell
ros2 service type /spawn_entity
```

返回

```
gazebo_msgs/srv/SpawnEntity
```

指令

```
ros2 interface show gazebo_msgs/srv/SpawnEntity
```

返回

```
string name                     # Name of the entity to be spawned (optional).
string xml                      # Entity XML description as a string, either
URDF or SDF.
string robot_namespace          # Spawn robot and all ROS interfaces under this
namespace
geometry_msgs/Pose initial_pose # Initial entity pose.
string reference_frame          # initial_pose is defined relative to the frame
of this entity.
                                # If left empty or "world" or "map", then gazebo
world frame is
                                # used.
                                # If non-existent entity is specified, an error
is returned
                                # and the entity is not spawned.
---
bool success                    # Return true if spawned successfully.
string status_message           # Comments if available.
```

可以看到服务的请求内容包括：

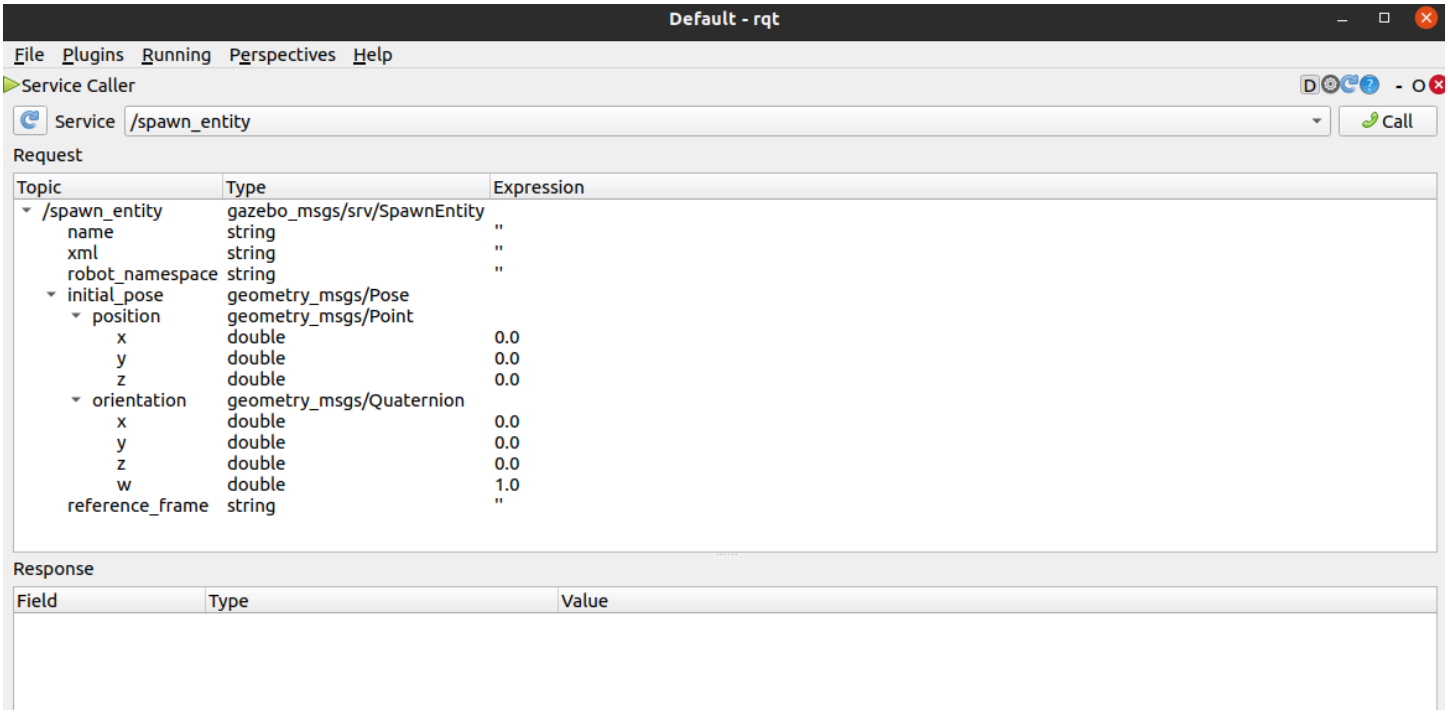
- string name ，需要加载的实体的名称 (可选的)。
- string xml ，实体的XML描述字符串, URDF或者SDF。

- string robot_namespace ，产生的机器人和所有的ROS接口的命名空间，多机器人仿真的时候很有用。
- geometry_msgs/Pose initial_pose ，机器人的初始化位置
- string reference_frame ，初始姿态是相对于该实体的frame定义的。如果保持"empty"或"world"或“map”，则使用 gazebo的world作为frame。如果指定了不存在的实体，则会返回错误

3.调用服务加载fishbot

看到这里你是不是迫不及待敲起来命令行来加载我们的机器人到gazebo了，别着急，小鱼再推荐一个可视化服务请求工具，其实在第六章中小鱼介绍过，在rqt工具集里有一个叫服务请求工具。

命令行输入rqt，在插件选项中选择Services->Service Caller,然后再下拉框选择/spawn_entity服务，即可看到下面的界面。



接着我们把我们的FishBot的URDF模型复制粘贴，放到xml中（注意要把原来的"删掉哦！），然后拿起我们的小电话，和小鱼一起Call。

Service Caller

Service /spawn_entity Call

Request

Topic	Type	Expression
/spawn_entity	gazebo_msgs/srv/SpawnEntity	
name	string	''

```
<?xml version="1.0"?>
<robot name="fishbot">


 <!-- Robot Footprint -->
 <link name="base_footprint"/>

 <joint name="base_joint" type="fixed">
  <parent link="base_footprint"/>
  <child link="base_link"/>
  <origin xyz="0.0 0.0 0.076" rpy="0 0 0"/>
 </joint>

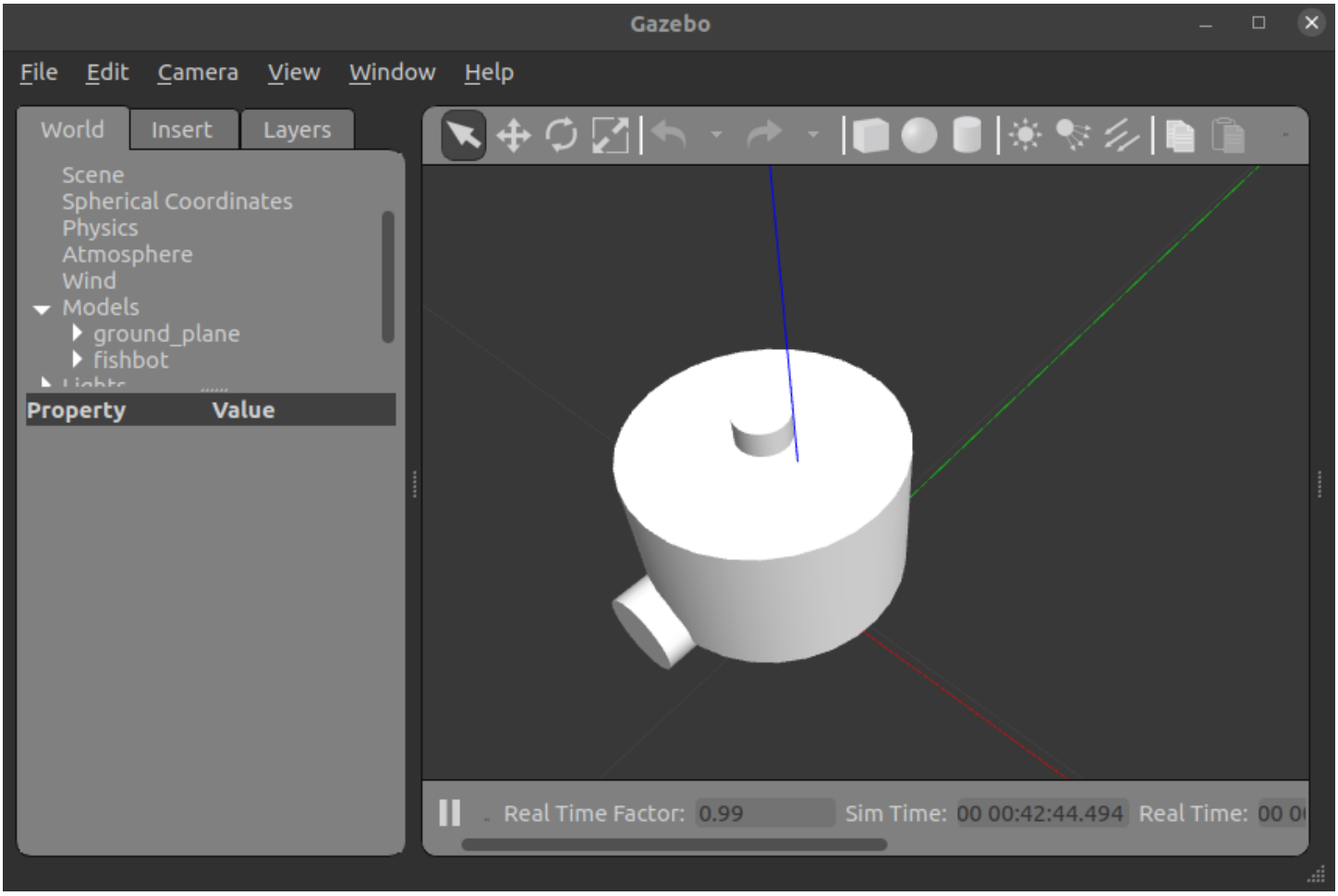

 <!-- base_link -->
```

Response

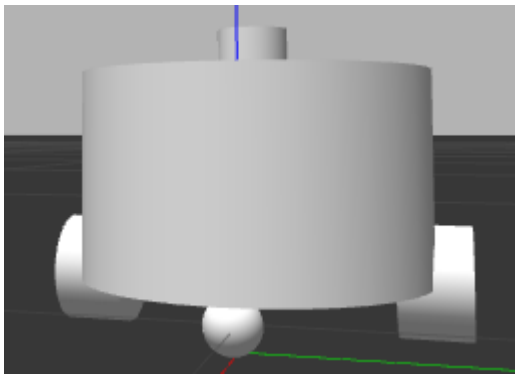
Field	Type	Value
/	gazebo_msgs/srv/SpawnEntity.Response	
success	boolean	True
status_message	string	'Successfully sent factory message to Gazebo.'

接着就可以看到工厂返回说成功把机器人制作出来送入gazebo了。

此时再看我们的Gazebo,一个小小的，白白的机器人出现了。



按住Shift加鼠标左键，拖动一下，来好好的欣赏欣赏我们的机器人。



3.4 在不同位置加载多个机器人

欣赏完毕后，小鱼再带你生产一个fishbot（为了后面需要多机器人仿真的小伙伴）。

修改rqt中的参数，增加一个命名空间，然后修改一个位置，让第二个机器人和第一个相距1m的地方生产，然后点击Call。

File Plugins Running Perspectives Help

Service Caller

Service /spawn_entity Call

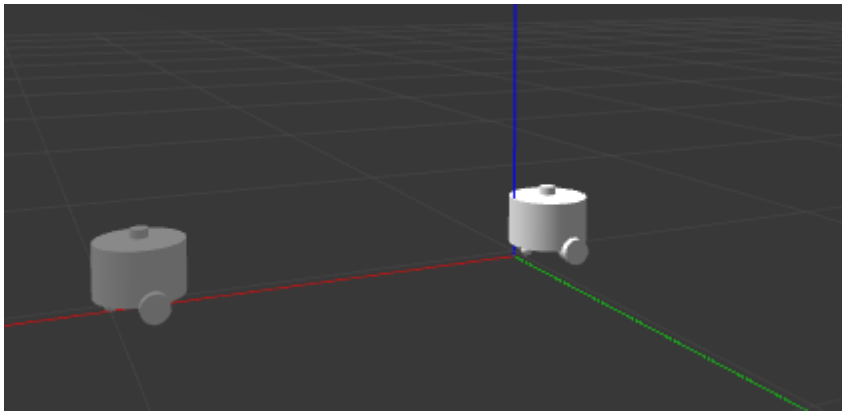
Request

Topic	Type	Expression
robot_namespace	string	mutirobot2
initial_pose	geometry_msgs/Pose	
position	geometry_msgs/Point	
x	double	1.0
y	double	0.0
z	double	0.0
orientation	geometry_msgs/Quaternion	
x	double	0.0
y	double	0.0
z	double	0.0
w	double	1.0
reference_frame	string	''

Response

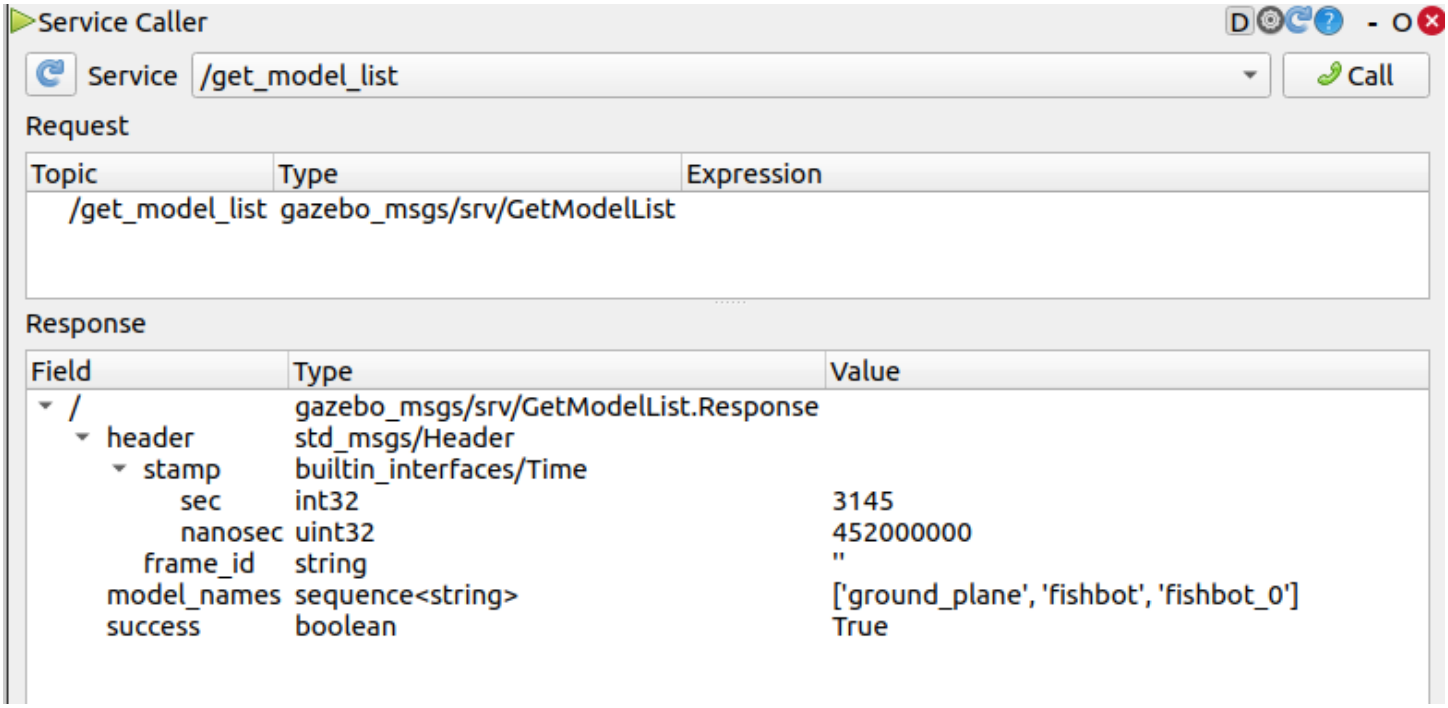
Field	Type	Value
/	gazebo_msgs/srv/SpawnEntity.Response	
success	boolean	True
status_message	string	'Successfully sent factory message to Gazebo.'

返回成功，此时拖送Gazebo观察一下，发现多出了一个机器人，距离刚好是在X轴（红色）1米（一个小格子一米）处。



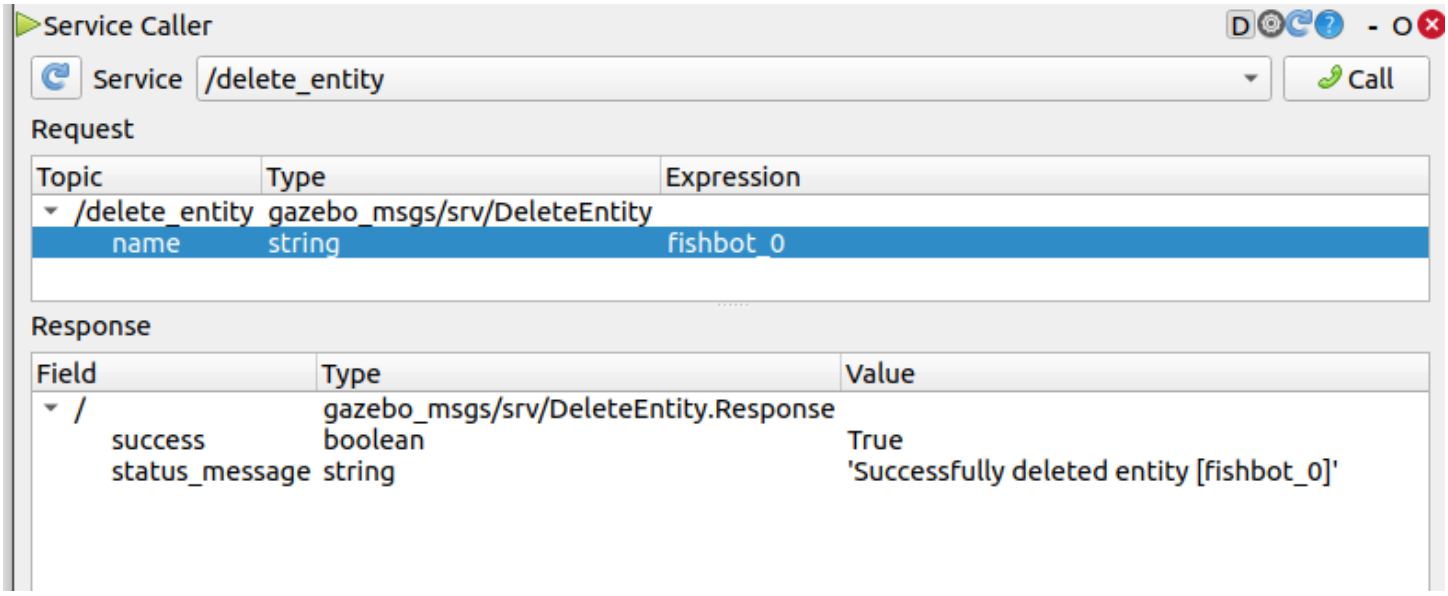
3.5 查询和删除机器人

利用rqt工具，我们再对另外两个服务接口进行请求。

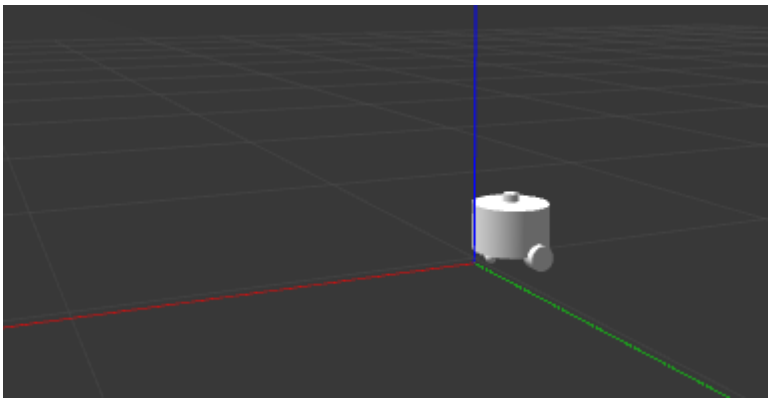


查到了三个模型，一个大地，一个fishbot，一个fishbot_0。

我们接着尝试把fishbot_0删掉，选择删除实体，输入fishbot_0的名字，拿起小电话通知工厂回收我们的0号fishbot。



调用成功，观察gazebo发现机器人没了



4. 将启动gazebo和生产fishbot写成launch文件

打开fishbot工作空间，在 `src/fishbot_description/launch` 中添加一个 `gazebo.launch.py` 文件，我们开始编写launch文件来在gazebo中加载机器人模型。

启动gazebo，我们可以将命令行写成一个launch节点

```
ExecuteProcess(
        cmd=['gazebo', '--verbose','-s', 'libgazebo_ros_init.so', '-s',
'libgazebo_ros_factory.so', gazebo_world_path],
        output='screen')
```

上面我们加载机器人是直接将XML格式的URDF复制过去进行加载的，这样很不方便，我们可以使用gazebo_ros为我们提供好的一个叫做 `spawn_entity.py` 节点，该节点支持从文件地址直接生产机器人到Gazebo。

> 其实该节点的原理也很简单，从URDF中读取机器人模型，然后再调用服务，和我们手动操作一个样子，小鱼只道没差别。

该节点需要两个参数，一个机器人的模型名字和urdf的文件地址，这个简单，前面我们曾经使用package_share来拼接过urdf路径。

```python
spawn_entity_cmd = Node(
    package='gazebo_ros',
    executable='spawn_entity.py',
    arguments=['-entity', robot_name_in_model,  '-file', urdf_model_path ],
output='screen')
```

最终写好的launch文件如下：

```python
import os
from launch import LaunchDescription
from launch.actions import ExecuteProcess
from launch_ros.actions import Node
from launch_ros.substitutions import FindPackageShare


def generate_launch_description():
    robot_name_in_model = 'fishbot'
    package_name = 'fishbot_description'
    urdf_name = "fishbot_gazebo.urdf"

    ld = LaunchDescription()
    pkg_share = FindPackageShare(package=package_name).find(package_name)
    urdf_model_path = os.path.join(pkg_share, f'urdf/{urdf_name}')

    # Start Gazebo server
    start_gazebo_cmd =  ExecuteProcess(
        cmd=['gazebo', '--verbose','-s', 'libgazebo_ros_init.so', '-s',
'libgazebo_ros_factory.so'],
        output='screen')

    # Launch the robot
    spawn_entity_cmd = Node(
        package='gazebo_ros',
        executable='spawn_entity.py',
        arguments=['-entity', robot_name_in_model,  '-file', urdf_model_path ],
output='screen')

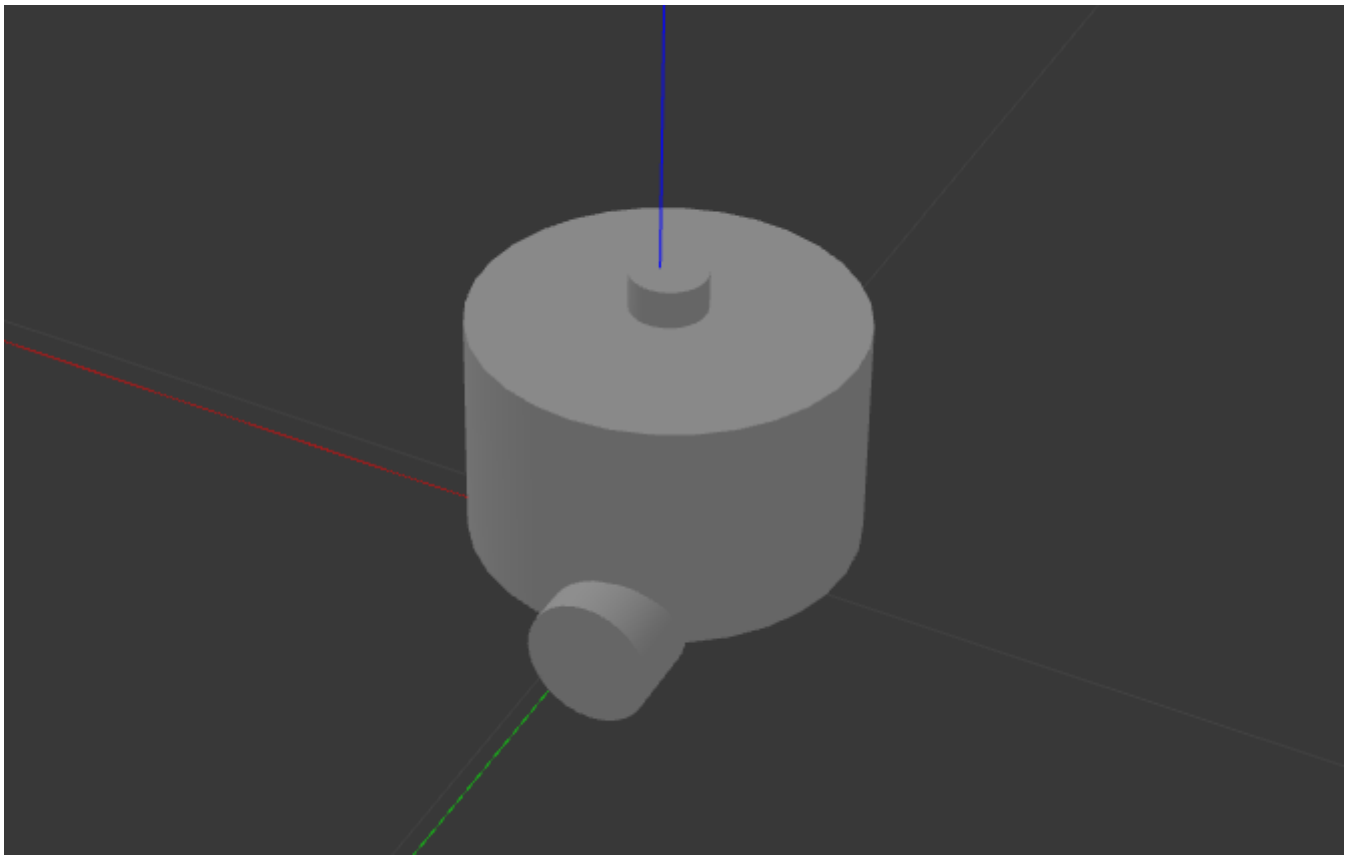
    ld.add_action(start_gazebo_cmd)
    ld.add_action(spawn_entity_cmd)


    return ld
```

编译运行

```
colcon build --packages-select fishbot_description
source install/setup.bash
ros2 launch fishbot_description gazebo.launch.py
```

完美显示



5.总结

这节课我们为Fishbot注入了仿真必须的物理属性，但是机器人还是不会动，下一节课我们就利用Gazebo的其他插件，让我们的机器人动起来。

最后再留一个课后作业：

1. 尝试将fishbot的物理属性去掉，再加载机器人看看会发生什么？
2. 尝试将fishbot的碰撞改成很小，再看看会发生什么？
3. gazebo还支持link的材料修改，在URDF中添加下面的代码，给支撑轮一个不一样的材质吧，你也可以将reference改成其他link，装点一下你的机器人。

```xml
<gazebo reference="caster_link">
  <material>Gazebo/Black</material>
</gazebo>
```

欢迎将你的实验结果在我们的fishros社区分享~

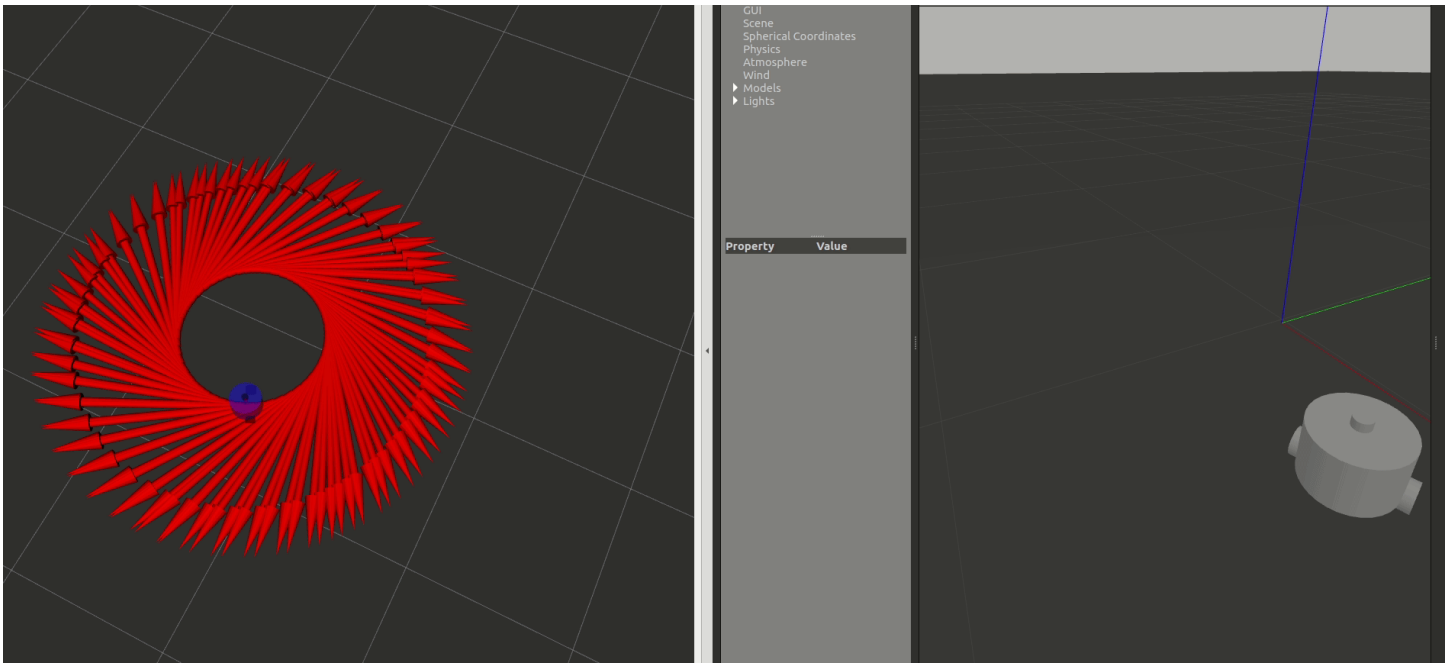
技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

4.Gazebo仿真插件之两轮差速

小鱼又来了，完成了上节课的Gazebo加载FishBot，但是机器人还是不会动，你一定很不开心吧，本节课小鱼就带你一起通过配置两轮差速控制插件，让我们的机器人动起来~

最终效果：



1.Gazebo插件介绍

之前小鱼说过Gazebo是一个独立于ROS的软件，对外提供了丰富的API可以使用，gazebo的插件按照用途大致可以分为两种：

1. **用于控制的插件**，通过插件可以控制机器人关节运动，可以进行位置、速度、力的控制，比如我们这节课的两轮差速控制器。
2. **用于数据采集的插件**，比如IMU传感器用于采集机器人的惯性，激光雷达用于采集机器人周围的点云信息。

当然上面两类插件功能也可以写到一个插件里，两轮差速插件就是一个二合一加强版。

2.两轮差速插件介绍

两轮差速插件用于控制机器人轮子关节的位置变化，同时该插件还会获取轮子的位置以及速度的信息的反馈，根据反馈的位置信息结合运动学模型即可计算出当前机器人的位姿（里程计）。

> 该插件的名称为：gazebo_ros_diff_drive

两轮差速控制器和Gazebo的关系

两轮差速控制器可以将轮子的目标转速发送给Gazebo，并从Gazebo获取到实际的速度和位置。

> **注意：发送给Gazebo是目标速度，反馈回来的是实际速度。目标!= 实际，比如轮子卡住了，无论你发什么目标速度，实际速度都是0。**

要想快速了解一个系统的功能，最直接的就是看系统的对外的输入和输出是什么？什么都不要说，看下图：

```mermaid
graph LR;
D[配置参数-轮距直径等]-->B
A[控制指令-cmdvel]-->B[gazebo_ros_diff_drive]
B-->C[里程计-odom]
B-->E[里程计tf-可选]
B-->F[轮子tf-可选]
```

上图就是对gazebo_ros_diff_drive的输入和输出信息的总结，可以很直观的看到该插件主要**输入控制指令**，主要**输出里程计信息**。接着小鱼带你分别认识一下输入和输出两个部分。

2.2 输入参数

2.2.1 配置参数

不知道你是否还记得在第七章中，小鱼对两轮差速底盘的运动学正的介绍。如果要完成底盘的正逆解和里程计的推算就必须要知道轮子的直径和间距。

同时该插件还提供了一些可以控制输出的选项，因为是仿真，所以还要告诉插件轮子对应的joint名称等信息，这样就有了下面这个参数表格：

配置项	含义
ros	ros相关配置，包含命名空间和话题重映射等
update_rate	数据更新速率
left_joint	左轮关节名称
right_joint	右轮关节名称

wheel_separation	左右轮子的间距
wheel_diameter	轮子的直径
max_wheel_torque	轮子最大的力矩
max_wheel_acceleration	轮子最大的加速度
publish_odom	是否发布里程计
publish_odom_tf	是否发布里程计的tf开关
publish_wheel_tf	是否发布轮子的tf数据开关
odometry_frame	里程计的framed ID，最终体现在话题和TF上
robot_base_frame	机器人的基础frame的ID

2.2.2 控制指令

两轮差速控制器默认通过订阅话题 `cmd_vel` 来获取目标线速度和角速度。该话题的类型为：`geometry_msgs/msg/Twist`

我们通过ros2的CLI来看一下这个消息包含的内容有哪些？

```
ros2 interface show geometry_msgs/msg/Twist
```

```
# This expresses velocity in free space broken into its linear and angular parts.

Vector3  linear
Vector3  angular
```

可以看到包含线速度和角速度，我们用proto在看一下包含的基本数据类型有哪些？

```
ros2 interface proto geometry_msgs/msg/Twist
```

```
"linear:
  x: 0.0
  y: 0.0
  z: 0.0
```

```
  angular:
    x: 0.0
    y: 0.0
    z: 0.0
  "
```

线速度和角速度都包含在x、y、z，代表坐标系的三个方向上的对应速度。

两轮差速控制器收到这个话题数据后将其中的角速度和线速度转换上两个轮子的转动速度发送给Gazebo。

2.3 输出参数

2.3.1 里程计

里程计信息默认的输出话题为 `odom` ，其消息类型为： `nav_msgs/msg/Odometry`

同样的使用CLI看一下其消息的组成结构：

```
ros2 interface show nav_msgs/msg/Odometry
```

```
# This represents an estimate of a position and velocity in free space.
# The pose in this message should be specified in the coordinate frame given by
header.frame_id
# The twist in this message should be specified in the coordinate frame given by
the child_frame_id

# Includes the frame id of the pose parent.
std_msgs/Header header

# Frame id the pose points to. The twist is in this coordinate frame.
string child_frame_id

# Estimated pose that is typically relative to a fixed world frame.
geometry_msgs/PoseWithCovariance pose
```

```
# Estimated linear and angular velocity relative to child_frame_id.
geometry_msgs/TwistWithCovariance twist
```

```
ros2 interface proto nav_msgs/msg/Odometry
```

```
"header:
  stamp:
    sec: 0
    nanosec: 0
  frame_id: ''
child_frame_id: ''
pose:
  pose:
    position:
      x: 0.0
      y: 0.0
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: 0.0
      w: 1.0
  covariance:
  - 0.0
  - 0.0
    ...
twist:
  twist:
    linear:
      x: 0.0
      y: 0.0
      z: 0.0
    angular:
      x: 0.0
      y: 0.0
      z: 0.0
  covariance:
  - 0.0
```

```
    - 0.0
      ...
"
```

可以看到其数据主要包含三个部分：

- header，表示该消息发布的时间
- pose，表示当前机器人位置和朝向
- twist，表示当前机器人的线速度和角速度

> 数据中还包含一个covariance，其代表协方差矩阵，后面小鱼写篇文章来介绍下，这里只需了解其含义即可。

2.3.2 里程计TF信息

设为true，订阅tf话题里你就可以看到像下面的msg，建议后面配置好后，手动修改下，对比区别

```
- header:
    stamp:
      sec: 6157
      nanosec: 907000000
    frame_id: odom
  child_frame_id: base_footprint
  transform:
    translation:
      x: 0.0005557960241049835
      y: -0.0007350446303238693
      z: 0.01599968753145574
    rotation:
      x: 4.691143395208505e-07
      y: 7.115496626557812e-06
      z: -0.018531475772549166
      w: 0.9998282774331005
```

2.3.3 左右轮子TF信息

设为true，订阅tf话题里你就可以看到像下面的msg，建议后面配置好后，手动修改下，对比区别

```
- header:
    stamp:
      sec: 6157
      nanosec: 941000000
    frame_id: base_link
  child_frame_id: left_wheel_link
  transform:
    translation:
      x: -0.02
      y: 0.1
      z: -0.06
    rotation:
      x: 0.0
      y: 0.049519025127821005
      z: 0.0
      w: 0.9987731805321918
- header:
    stamp:
      sec: 6157
      nanosec: 941000000
    frame_id: base_link
  child_frame_id: right_wheel_link
  transform:
    translation:
      x: -0.02
      y: -0.1
      z: -0.06
    rotation:
      x: 0.0
      y: -0.0663387077034509
      z: 0.0
      w: 0.9977971616817898
```

3.在URDF中配置两轮差速模型

上面该介绍的我们都给介绍了，接着我们直接来配置。

因为是给Gazebo的插件，所以在 `URDF` 中，我们需要使用 `<gazebo>` 进行配置，因为是要给 `gazebo` 配置插件，所有要在 `gazebo` 标签下添加 `plugin` 子插件。

话不多说，上代码

```
    <gazebo>
      <plugin name='diff_drive' filename='libgazebo_ros_diff_drive.so'>
            <ros>
              <namespace>/</namespace>
              <remapping>cmd_vel:=cmd_vel</remapping>
              <remapping>odom:=odom</remapping>
            </ros>
            <update_rate>30</update_rate>
            <!-- wheels -->
            <left_joint>left_wheel_joint</left_joint>
            <right_joint>right_wheel_joint</right_joint>
            <!-- kinematics -->
            <wheel_separation>0.2</wheel_separation>
            <wheel_diameter>0.065</wheel_diameter>
            <!-- limits -->
            <max_wheel_torque>20</max_wheel_torque>
            <max_wheel_acceleration>1.0</max_wheel_acceleration>
            <!-- output -->
            <publish_odom>true</publish_odom>
            <publish_odom_tf>true</publish_odom_tf>
            <publish_wheel_tf>true</publish_wheel_tf>
            <odometry_frame>odom</odometry_frame>
            <robot_base_frame>base_footprint</robot_base_frame>
        </plugin>
```

将这段代码加到我们的URDF中，然后对着上面小鱼介绍的配置项，一一看下，接着我们就可以来测试运行了。

完整的URDF可以参考：

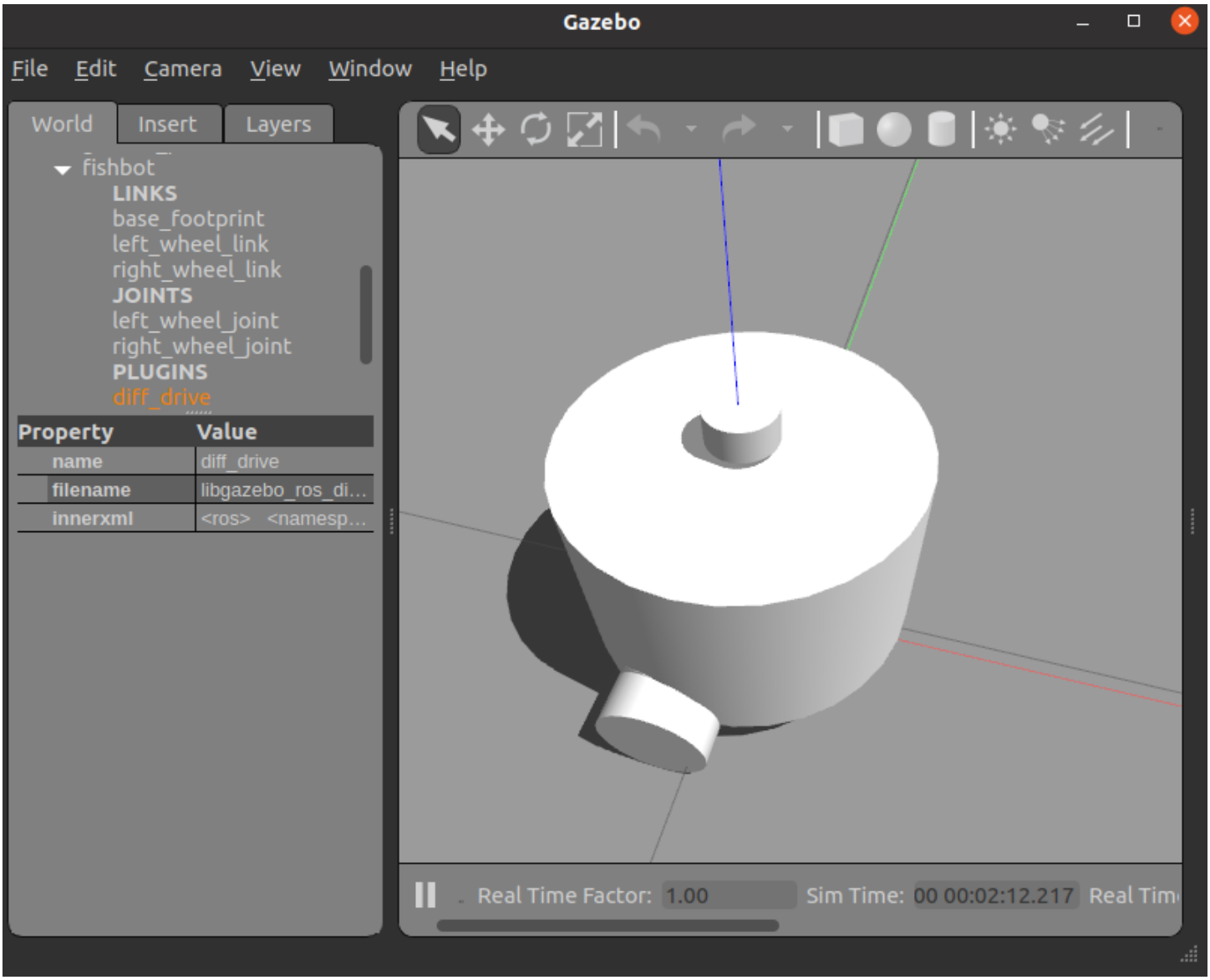
4.两轮差速插件测试

修改完了URDF模型我们将代码编译一下，让更新后的URDF文件安装到install目录，接着就可以运行9.2中的launch文件，将模型加载到Gazebo中。

4.1 编译-启动

```
colcon build
```

```
ros2 launch fishbot_description gazebo.launch.py
```



接着你可以使用CLI工具看一下系统有哪些节点在运行

```
ros2 node list
ros2 topic list
```

```
ros2 node list
---------------
/diff_drive
/gazebo

ros2 topic list
---------------
/clock
/cmd_vel
/odom
/parameter_events
/performance_metrics
/rosout
/tf
```

相信此时你已经看到了我们插件订阅的的/cmd_vel和发布的/odom了。

4.3 使用键盘控制fishbot

你还记得第二章中小鱼带你玩的小乌龟吗？当时我们用键盘来控制小乌龟运动，现在我们可以用键盘来控制fishbot动起来了。

你需要一个键盘控制工具，可以用下面的指令安装

```
sudo apt install ros-humble-teleop-twist-keyboard
```

这个功能包下有一个节点，这个节点会监听键盘的按键事件，然后发布cmd_vel话题，该话题被gazebo的两轮差速插件所订阅。所以我们就可以通过这个节点来控制fishbot。

```
ros2 run teleop_twist_keyboard teleop_twist_keyboard
```

> 如果你想让这个节点不是发布cmd_vel话题，而是别的，可以采用ROS2的话题重映射功能。 eg: `ros2 run teleop_twist_keyboard teleop_twist_keyboard --ros-args --remap cmd_vel:=cmd_vel1`

接着尝试使用来控制机器人运动

```
    U    I    O
    J    K    L
    M    <    >
```

点一下I，你就能看到fishbot在Gazebo中飞速的移动。接着打开终端，打印一下odom话题和tf话题，移动机器人观察数据变化。

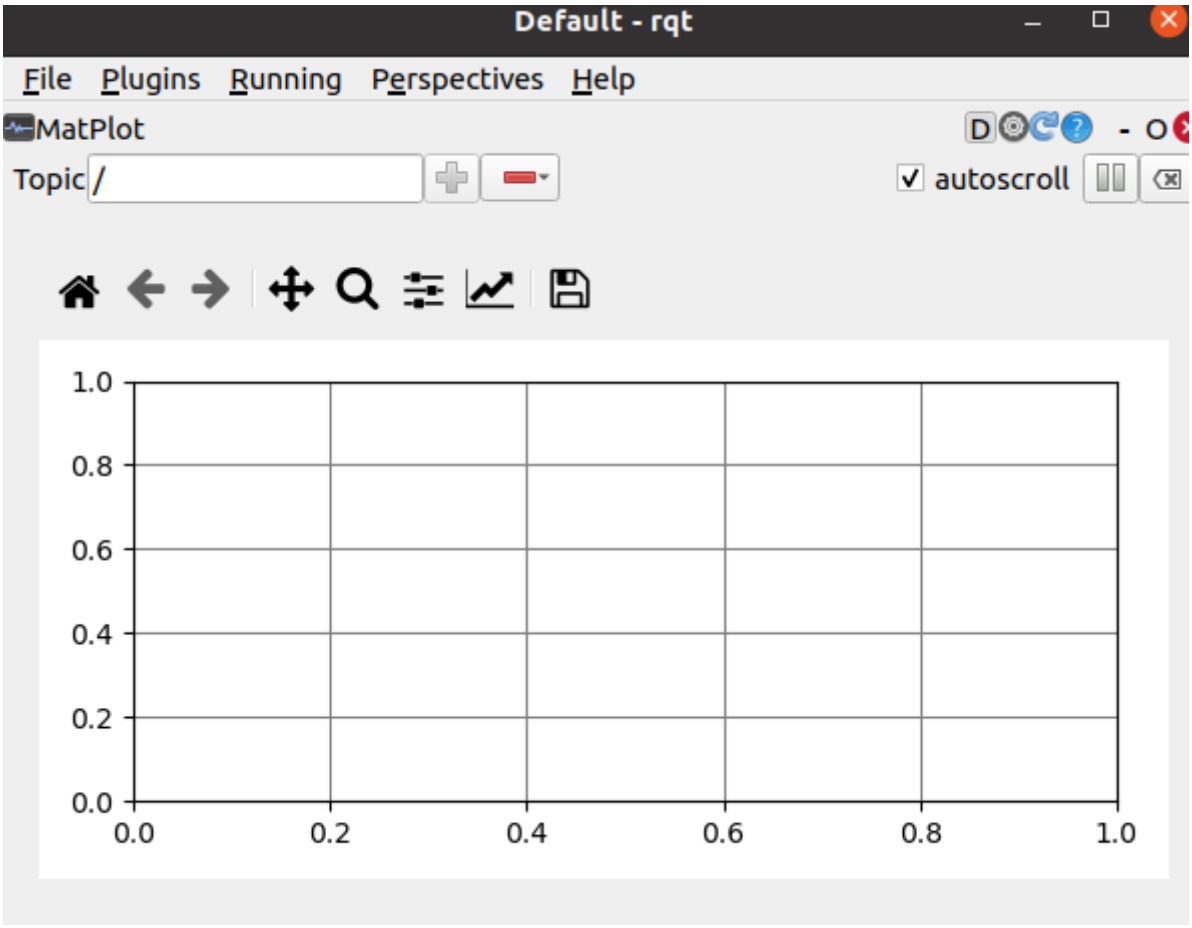
此时你应该玩一会，体验一下各种花式走法。

4.4 使用rqt显示速度数据

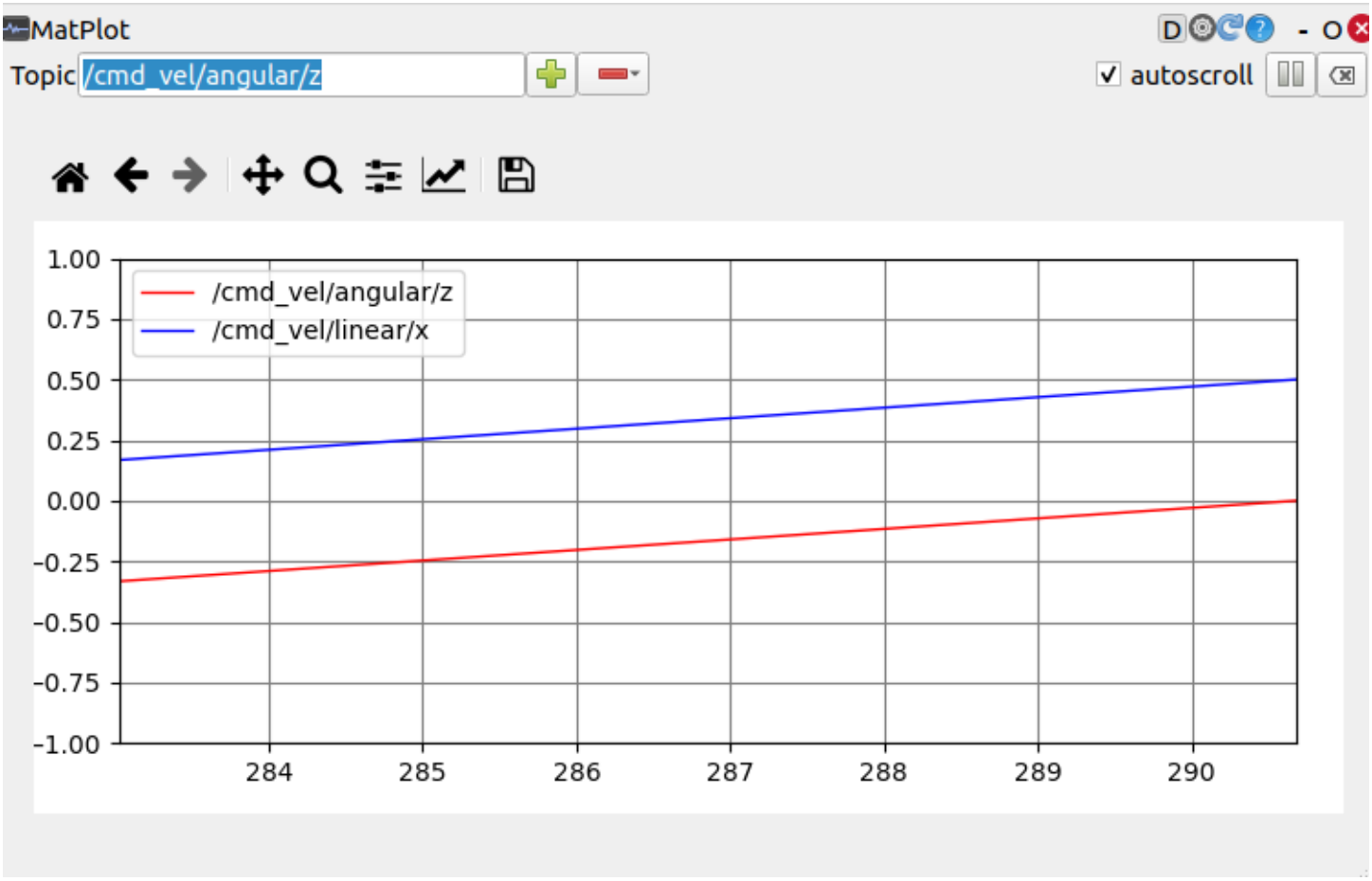
接着我们尝试使用rqt将数据在rqt中可视化出来，打开终端输入rqt。

```
rqt
```

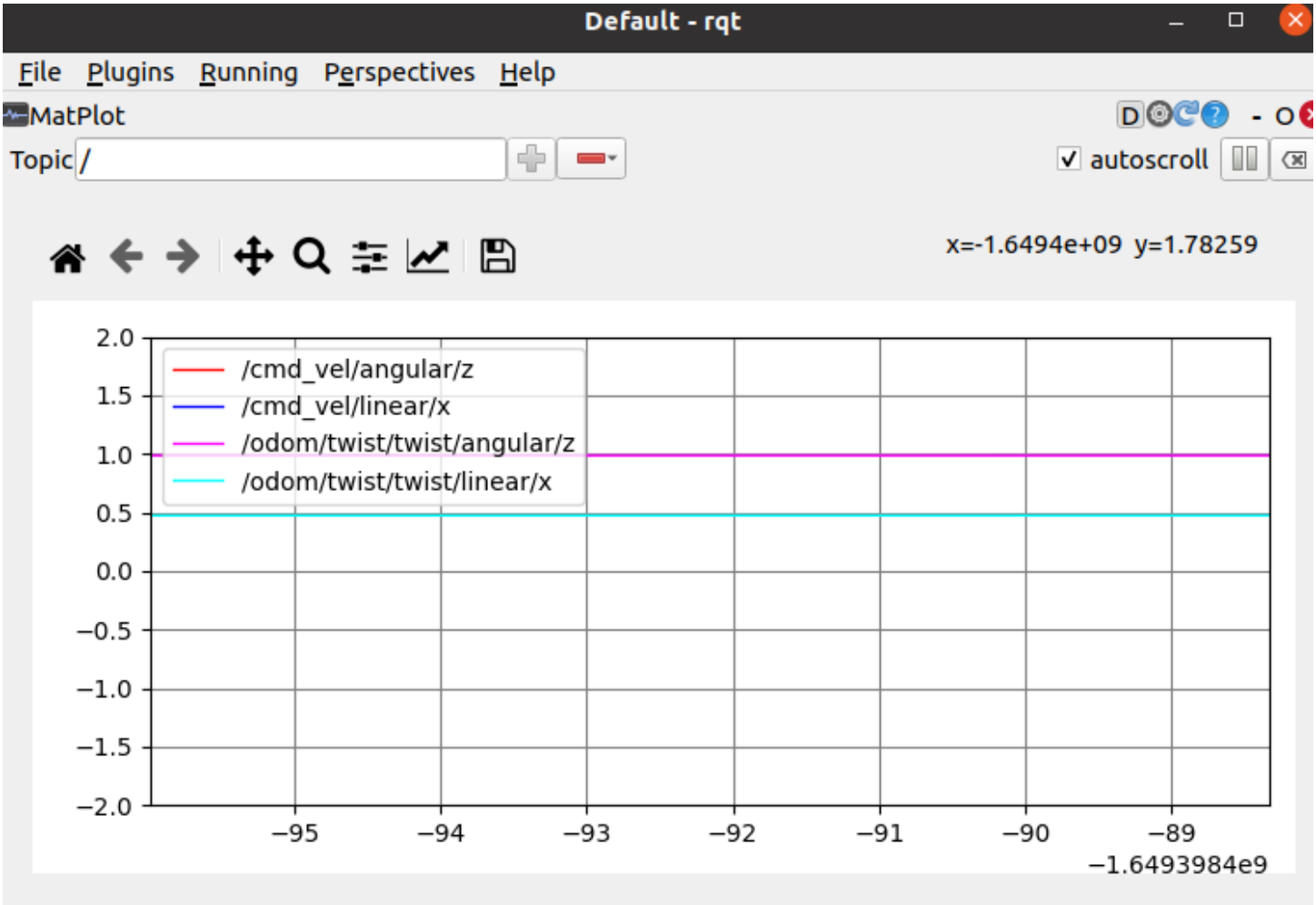
选择Plugin->Visualization->Plot



在上方Topic输入 `/cmd_vel/linear/x` ，再输入 `/cmd_vel/angular/z` ，然后用键盘控制机器人移动。



cmd_vel中的速度代表目标速度，接着我们显示一下当前速度（在odom.twist中）



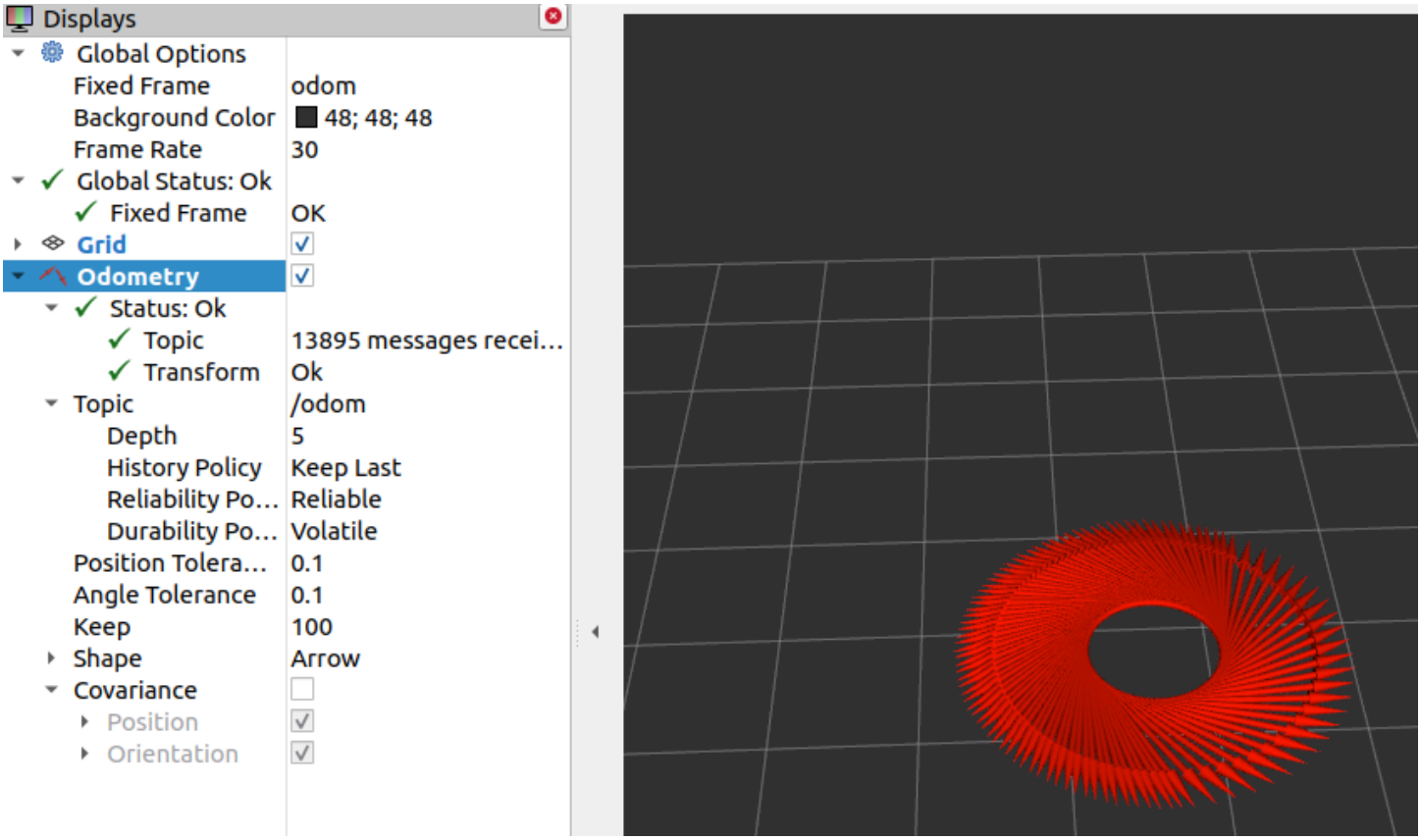
4.5 在RVIZ2中显示Fishbot及其轨迹

打开rviz2

```
rviz2
```

1. 修改FixedFrame为odom
2. 添加插件，Add->Odometry->OK
3. 选择话题，Odometry->Topic->选/odom
4. 去除协方差显示，Odometry->Covariance>取消勾选
5. 键盘控制节点，点个U，原地转圈圈

最终结果：



4.6 在RVIZ2中显示机器人模型

虽然机器人的轨迹已经在RVIZ中显示出来了，但是并没有机器人的模型，也看不到轮子的转动，小鱼来带你一起解决这个问题。

前面小鱼介绍过，要发布机器人模型我们所使用的节点是 `robot_state_publisher` ,所以我们在 `gazebo.launch.py` 中加入这个节点，同时再加上rviz2的启动节点，最终的 `gazebo.launch.py` 内容如下：

```python
import os
from launch import LaunchDescription
from launch.actions import ExecuteProcess
from launch_ros.actions import Node
from launch_ros.substitutions import FindPackageShare


def generate_launch_description():
    robot_name_in_model = 'fishbot'
    package_name = 'fishbot_description'
    urdf_name = "fishbot_gazebo.urdf"

    ld = LaunchDescription()
    pkg_share = FindPackageShare(package=package_name).find(package_name)
```

```
    urdf_model_path = os.path.join(pkg_share, f'urdf/{urdf_name}')

    # Start Gazebo server
    start_gazebo_cmd =  ExecuteProcess(
        cmd=['gazebo', '--verbose','-s', 'libgazebo_ros_init.so', '-s',
'libgazebo_ros_factory.so', gazebo_world_path],
        output='screen')

    # Launch the robot
    spawn_entity_cmd = Node(
        package='gazebo_ros',
        executable='spawn_entity.py',
        arguments=['-entity', robot_name_in_model,  '-file', urdf_model_path ],
output='screen')

    # Start Robot State publisher
    start_robot_state_publisher_cmd = Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        arguments=[urdf_model_path]
    )

    # Launch RViz
    start_rviz_cmd = Node(
        package='rviz2',
        executable='rviz2',
        name='rviz2',
        output='screen',
        # arguments=['-d', default_rviz_config_path]
        )

    ld.add_action(start_gazebo_cmd)
    ld.add_action(spawn_entity_cmd)
    ld.add_action(start_robot_state_publisher_cmd)
    ld.add_action(start_rviz_cmd)

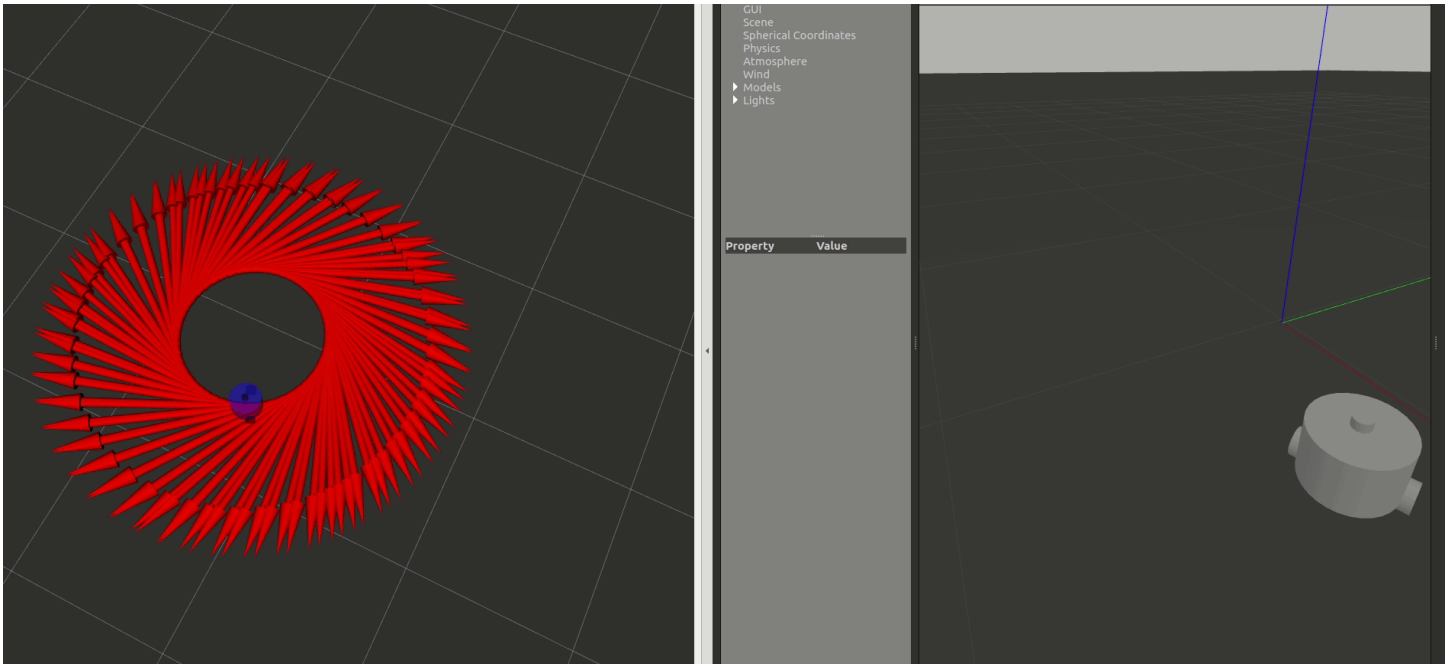

    return ld
```

保存编译启动

```shell
colcon build
```

```shell
ros2 launch fishbot_description gazebo.launch.py
```

然后继续启动键盘控制，Enjoy It!



技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

5.Gazebo仿真插件之IMU

大家好，我是小鱼，上节课通过配置两轮差速控制器我们已经成功的让fishbot在gazebo中动了起来，本节课我们通过给fishbot的URDF配置IMU传感器插件，让IMU模块工作起来。

1.惯性测量单元IMU介绍

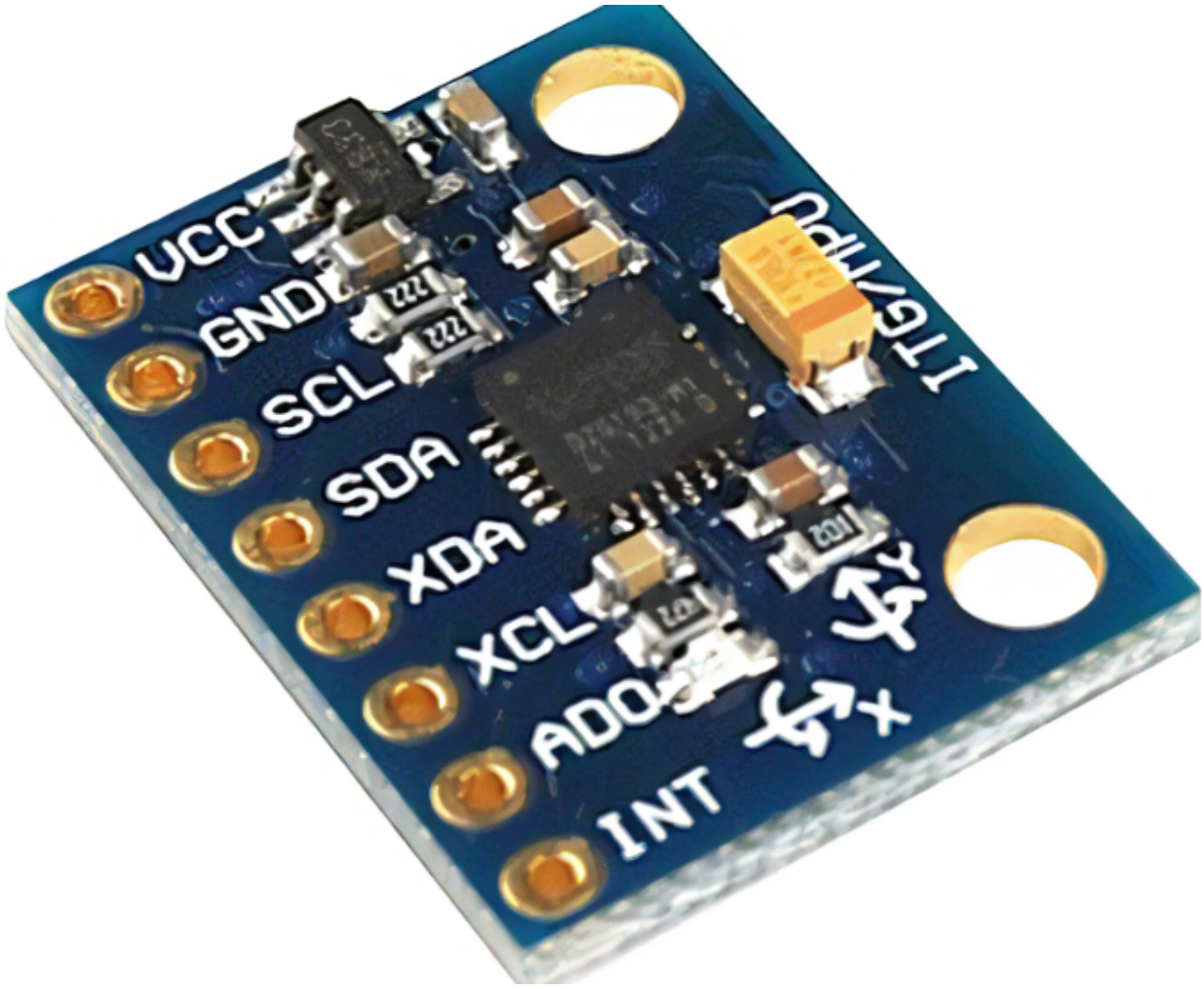
惯性测量单元是测量物体三轴姿态角(或角速率)以及加速度的装置。一般的，一个IMU包含了三个单轴的加速度计和三个单轴的陀螺，加速度计检测物体在载体坐标系统独立三轴的加速度信号，而陀螺检测载体相对于导航坐标系的角速度信号，**测量物体在三维空间中的角速度和加速度**，并以此解算出**物体的姿态**。在导航中有着很重要的应用价值。

上面这段话是小鱼从百科中摘抄出来的，你需要知道的一个关键点是IMU可以测量以下三组数据：

- 三维角度
- 三维加速度
- 三维角速度

1.1 IMU长啥样？

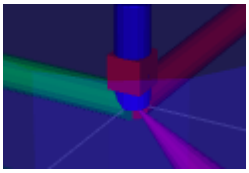
便宜的长这样：



贵的长这样：



不要钱的长什么样？



仿真的不要钱哈哈，接着我们来配置一下仿真的IMU。

2.Gazebo-IMU插件介绍

仿真的IMU也是对应一个后缀为 `.so` 的动态链接库，使用下面的指令可以查看所有的动态链接库：

```
ls /opt/ros/humble/lib/libgazebo_ros*
```

```
/opt/ros/humble/lib/libgazebo_ros2_control.so
/opt/ros/humble/lib/libgazebo_ros_ackermann_drive.so
/opt/ros/humble/lib/libgazebo_ros_bumper.so
/opt/ros/humble/lib/libgazebo_ros_camera.so
/opt/ros/humble/lib/libgazebo_ros_diff_drive.so
/opt/ros/humble/lib/libgazebo_ros_elevator.so
/opt/ros/humble/lib/libgazebo_ros_factory.so
/opt/ros/humble/lib/libgazebo_ros_force.so
/opt/ros/humble/lib/libgazebo_ros_force_system.so
/opt/ros/humble/lib/libgazebo_ros_ft_sensor.so
/opt/ros/humble/lib/libgazebo_ros_gps_sensor.so
/opt/ros/humble/lib/libgazebo_ros_hand_of_god.so
/opt/ros/humble/lib/libgazebo_ros_harness.so
/opt/ros/humble/lib/libgazebo_ros_imu_sensor.so
/opt/ros/humble/lib/libgazebo_ros_init.so
/opt/ros/humble/lib/libgazebo_ros_joint_pose_trajectory.so
/opt/ros/humble/lib/libgazebo_ros_joint_state_publisher.so
/opt/ros/humble/lib/libgazebo_ros_node.so
/opt/ros/humble/lib/libgazebo_ros_p3d.so
/opt/ros/humble/lib/libgazebo_ros_planar_move.so
/opt/ros/humble/lib/libgazebo_ros_projector.so
/opt/ros/humble/lib/libgazebo_ros_properties.so
/opt/ros/humble/lib/libgazebo_ros_ray_sensor.so
/opt/ros/humble/lib/libgazebo_ros_state.so
/opt/ros/humble/lib/libgazebo_ros_template.so
/opt/ros/humble/lib/libgazebo_ros_tricycle_drive.so
/opt/ros/humble/lib/libgazebo_ros_utils.so
/opt/ros/humble/lib/libgazebo_ros_vacuum_gripper.so
/opt/ros/humble/lib/libgazebo_ros_video.so
/opt/ros/humble/lib/libgazebo_ros_wheel_slip.so
```

IMU对应的消息类型为 `sensor_msgs/msg/Imu`

```
ros2 interface show sensor_msgs/msg/Imu
```

```
# This is a message to hold data from an IMU (Inertial Measurement Unit)
#
# Accelerations should be in m/s^2 (not in g's), and rotational velocity should be
in rad/sec
#
# If the covariance of the measurement is known, it should be filled in (if all
you know is the
# variance of each measurement, e.g. from the datasheet, just put those along the
diagonal)
# A covariance matrix of all zeros will be interpreted as "covariance unknown",
and to use the
# data a covariance will have to be assumed or gotten from some other source
#
# If you have no estimate for one of the data elements (e.g. your IMU doesn't
produce an
# orientation estimate), please set element 0 of the associated covariance matrix
to -1
# If you are interpreting this message, please check for a value of -1 in the
first element of each
# covariance matrix, and disregard the associated estimate.

std_msgs/Header header

geometry_msgs/Quaternion orientation
float64[9] orientation_covariance # Row major about x, y, z axes

geometry_msgs/Vector3 angular_velocity
float64[9] angular_velocity_covariance # Row major about x, y, z axes

geometry_msgs/Vector3 linear_acceleration
float64[9] linear_acceleration_covariance # Row major x, y z
```

可以看到除了每个数据对应的三个协方差之外，每一个还都对应一个 `3*3` 的协方差矩阵。

3.给FIshbot配置IMU传感器

有了上节课的经验，我们可以很轻松的添加IMU传感器，但是还有一个需要注意的地方，为了更真实的模拟IMU传感器，我们需要给我们的仿真IMU传感器加点料。

加什么？加点高斯噪声，高斯噪声只需要指定平均值和标准差两个参数即可，不过因为IMU传感器的特殊性，我们还需要给模型添加两个偏差参数，分别是 `平均值偏差` 和 `标准差偏差` 。

有关Gazebo仿真和噪声模型更深入的介绍可以参考小鱼发的两篇推文：

- Gazebo仿真进阶教程之传感器高斯噪声（一）
- Gazebo仿真进阶教程之传感器高斯噪声（二）

下面是IMU传感器的URDF配置代码，大家结合文章对应可以理解一下，IMU对应的插件库 `libgazebo_ros_imu_sensor.so`

```
      <gazebo reference="imu_link">
        <sensor name="imu_sensor" type="imu">
        <plugin filename="libgazebo_ros_imu_sensor.so" name="imu_plugin">
            <ros>
              <namespace>/</namespace>
              <remapping>~/out:=imu</remapping>
            </ros>

  <initial_orientation_as_reference>false</initial_orientation_as_reference>
          </plugin>
          <always_on>true</always_on>
          <update_rate>100</update_rate>
          <visualize>true</visualize>
          <imu>
            <angular_velocity>
              <x>
                <noise type="gaussian">
                  <mean>0.0</mean>
                  <stddev>2e-4</stddev>
                  <bias_mean>0.0000075</bias_mean>
                  <bias_stddev>0.0000008</bias_stddev>
                </noise>
              </x>
              <y>
                <noise type="gaussian">
```

```
            <mean>0.0</mean>
            <stddev>2e-4</stddev>
            <bias_mean>0.0000075</bias_mean>
            <bias_stddev>0.0000008</bias_stddev>
          </noise>
        </y>
        <z>
          <noise type="gaussian">
            <mean>0.0</mean>
            <stddev>2e-4</stddev>
            <bias_mean>0.0000075</bias_mean>
            <bias_stddev>0.0000008</bias_stddev>
          </noise>
        </z>
      </angular_velocity>
      <linear_acceleration>
        <x>
          <noise type="gaussian">
            <mean>0.0</mean>
            <stddev>1.7e-2</stddev>
            <bias_mean>0.1</bias_mean>
            <bias_stddev>0.001</bias_stddev>
          </noise>
        </x>
        <y>
          <noise type="gaussian">
            <mean>0.0</mean>
            <stddev>1.7e-2</stddev>
            <bias_mean>0.1</bias_mean>
            <bias_stddev>0.001</bias_stddev>
          </noise>
        </y>
        <z>
          <noise type="gaussian">
            <mean>0.0</mean>
            <stddev>1.7e-2</stddev>
            <bias_mean>0.1</bias_mean>
            <bias_stddev>0.001</bias_stddev>
          </noise>
        </z>
      </linear_acceleration>
    </imu>
```

```
        </sensor>
      </gazebo>
```

将上面的代码加到 `fishbot_gazebo.urdf` 中，接着我们就可以进行测试了。

4.编译测试

编译

```shell
colcon build
```

运行

```
ros2 launch fishbot_description gazebo.launch.py
```

CLI看话题

```
ros2 topic list
ros2 topic info /imu
```

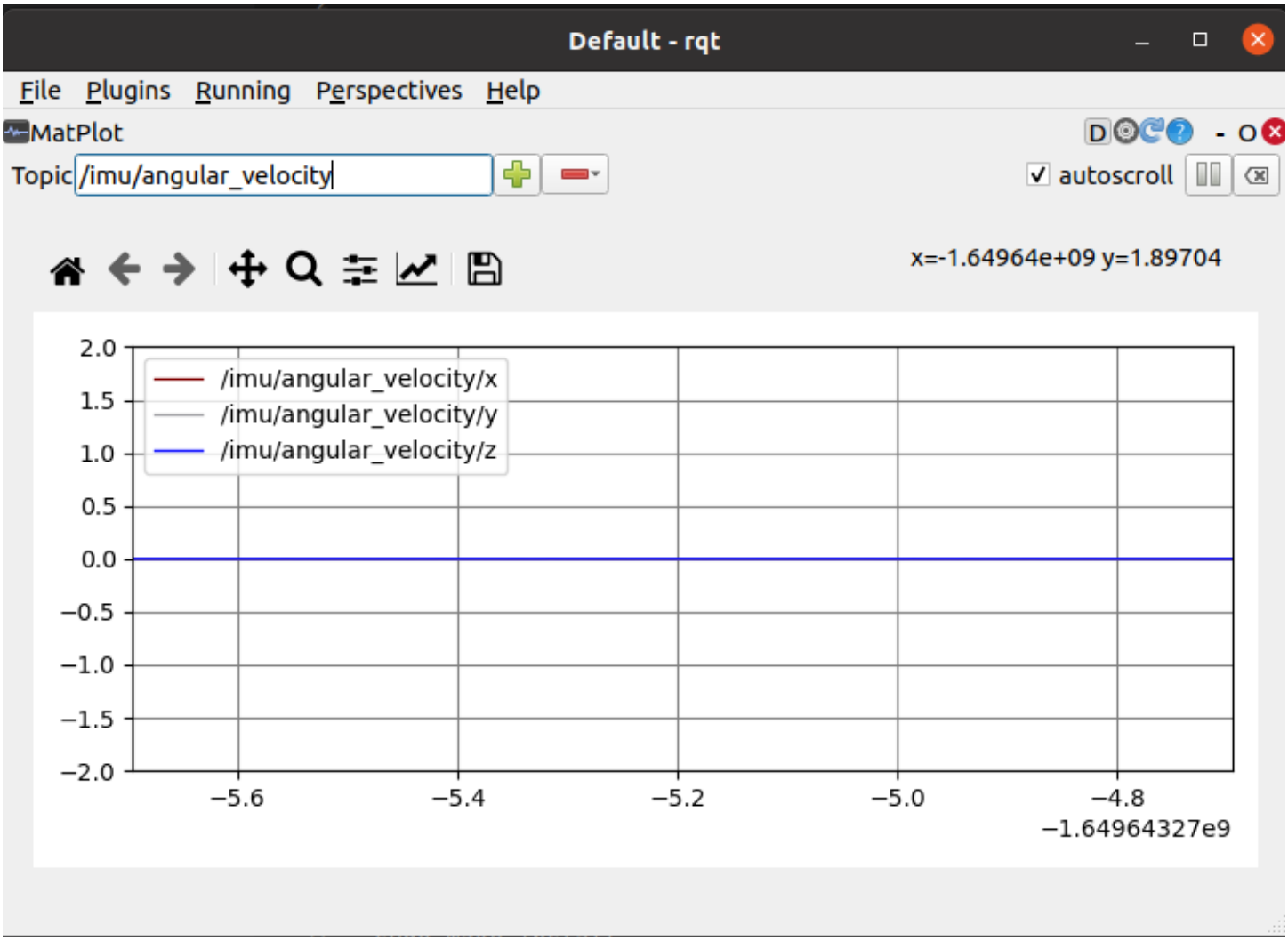
```
ros2 topic echo /imu
```

输出：

```
header:
  stamp:
    sec: 150
    nanosec: 599000000
  frame_id: base_footprint
orientation:
  x: 3.434713830866392e-07
```

```
  y: 7.119913105768616e-06
  z: -0.00028312437320413914
  w: 0.9999999598948884
orientation_covariance:
- 0.0
- 0.0
- 0.0
- 0.0
- 0.0
- 0.0
- 0.0
- 0.0
- 0.0
angular_velocity:
  x: -0.00013597855247901325
  y: 0.0006306135617081868
  z: -0.00015794894627685146
angular_velocity_covariance:
- 4.0e-08
- 0.0
- 0.0
- 0.0
- 4.0e-08
- 0.0
- 0.0
- 0.0
- 4.0e-08
linear_acceleration:
  x: 0.08679200038530369
  y: 0.07753419258567491
  z: 9.687910969061628
linear_acceleration_covariance:
- 0.00028900000000000003
- 0.0
- 0.0
- 0.0
- 0.00028900000000000003
- 0.0
- 0.0
- 0.0
- 0.00028900000000000003
```

用rqt可视化：



5.总结

本节我们对IMU传感器进行介绍，并通过gazbeo的imu插件完成了fishbot的IMU数据的输出。

最后还有小练习等着你：

1. 再次启动遥控节点，控制fishbot，观察IMU传感器的数据变化

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**

- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划为FishBot添加IMU传感器

6.为FishBot添加添加激光雷达传感器

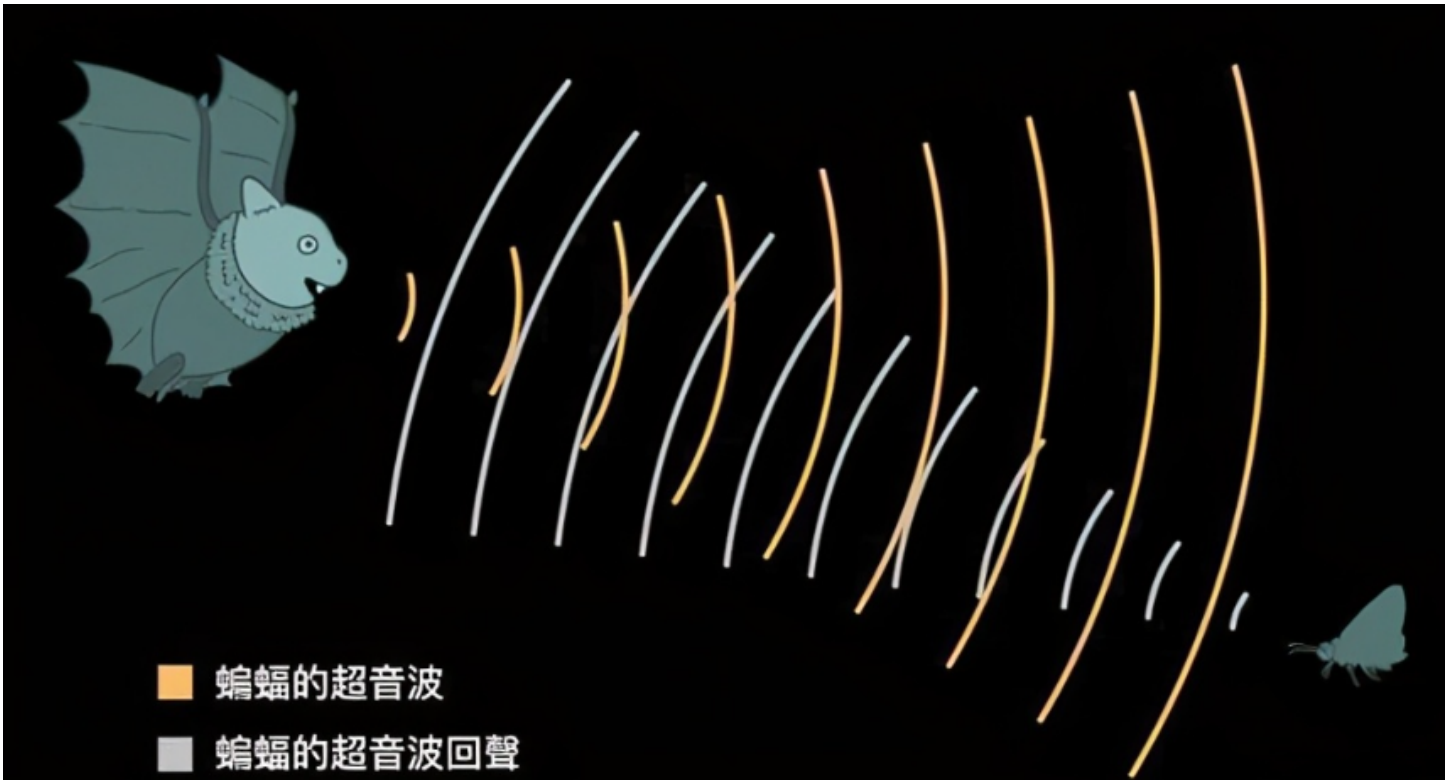
本节我们来认识一个新的传感器，该传感器在自动驾驶、室内导航等应用非常多，比如扫地机器人上就是用的它作为感知环境的重要工具，该传感器是激光雷达。

1.激光雷达介绍

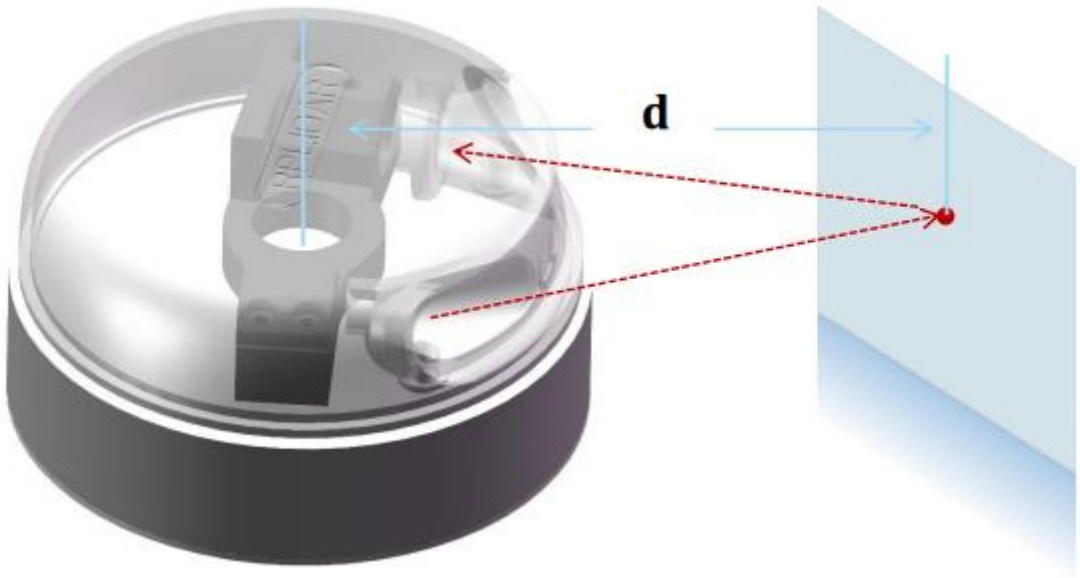
激光雷达（Light Detection And Ranging）,缩写 `LiDAR` ，翻译一下叫——激光探测与测距。

1.1 激光雷达原理介绍

激光雷达的原理也很简单，就像蝙蝠的定位方法一样，蝙蝠定位大家都知道吧，像下面这样子的回声定位。



普通的单线激光雷达一般有一个发射器，一个接收器，发射器发出激光射线到前方的目标上，物品会将激光反射回来，然后激光雷达的接受器可以检测到反射的激光。

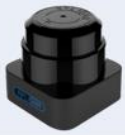


通过计算发送和反馈之间的时间间隔，乘上激光的速度，就可以计算出激光飞行的距离，该计算方法成为TOF（飞行时间法Time of flight，也称时差法）。

除了TOF之外还有其他方法进行测距，比如三角法，这里就不拓展了放一篇文章，大家自行阅读。

- 激光三角测距原理详述

目前市面上的激光雷达，几乎都是采用三角测距，比如思岚的：

系列	RPLIDAR				Mapper
型号	A1	A2	A3	S1	M2M1
产品图片					
推荐应用	智能扫地机、家用机器人	商用或消费类机器人 3D建模，多点触摸	商用机器人、室内外均可使用	商用机器人、室内外均可使用	商用机器人、环境测绘，手持测量、室内外均可使用
测距技术	三角测距	三角测距	三角测距	TOF	TOF
测量半径	0.15m - 12m	0.2m - 16m	0.2m - 25m	0.1m - 40m	0.1m - 40m
采样频率	8K	16K	16K	9.2K	9.2K
扫描频率	5.5Hz	5Hz-15Hz	5Hz - 15Hz	8Hz - 15Hz	8Hz-15Hz
角度分辨率	≤1°	0.255°	0.225°	0.391°	0.391°

需要注意的是虽然只有一个发射器和一个接受器，激光雷达通过电机可以进行旋转，这样就可以达到对周围环境360度测距的目的。

1.2 激光雷达大赏

五位数的长这样



四位数的长这样



三位数的长这样



两位数的长这样



不要钱的长这样

仿真的，不要钱



2.Gazebo激光雷达插件

因为激光雷达是属于射线类传感器，该类传感在在Gazebo插件中都被封装成了一个动态库 `libgazebo_ros_ray_sensor.so` 。

接着我们来看看LiDAR的话题消息接口 `sensor_msgs/msg/LaserScan` 。

```
ros2 interface show  sensor_msgs/msg/LaserScan
```

```
# Single scan from a planar laser range-finder
#
# If you have another ranging device with different behavior (e.g. a sonar
# array), please find or create a different message, since applications
# will make fairly laser-specific assumptions about this data

std_msgs/Header header # timestamp in the header is the acquisition time of
                            # the first ray in the scan.
                            #
                            # in frame frame_id, angles are measured around
                            # the positive Z axis (counterclockwise, if Z is up)
                            # with zero angle being forward along the x axis
```

```
float32 angle_min            # start angle of the scan [rad]
float32 angle_max            # end angle of the scan [rad]
float32 angle_increment      # angular distance between measurements [rad]

float32 time_increment       # time between measurements [seconds] - if your
scanner
                             # is moving, this will be used in interpolating
position
                             # of 3d points
float32 scan_time            # time between scans [seconds]

float32 range_min            # minimum range value [m]
float32 range_max            # maximum range value [m]

float32[] ranges             # range data [m]
                             # (Note: values < range_min or > range_max should be
discarded)
float32[] intensities        # intensity data [device-specific units].  If your
                             # device does not provide intensities, please leave
                             # the array empty.
```

雷达的数据结构有些复杂，但通过注释和名字相信你可以看的七七八八，看不懂也没关系，一般情况下我们不会直接的对雷达的数据做操作。小鱼这里也就先skip。

3.为FishBot添加雷达插件

有了前面的经验，我们需要在URDF添加以下内容即可

xml

```xml
  <gazebo reference="laser_link">
      <sensor name="laser_sensor" type="ray">
      <always_on>true</always_on>
      <visualize>true</visualize>
      <update_rate>5</update_rate>
      <pose>0 0 0.075 0 0 0</pose>
      <ray>
          <scan>
            <horizontal>
              <samples>360</samples>
```

```
            <resolution>1.000000</resolution>
            <min_angle>0.000000</min_angle>
            <max_angle>6.280000</max_angle>
          </horizontal>
        </scan>
        <range>
          <min>0.120000</min>
          <max>3.5</max>
          <resolution>0.015000</resolution>
        </range>
        <noise>
          <type>gaussian</type>
          <mean>0.0</mean>
          <stddev>0.01</stddev>
        </noise>
      </ray>

      <plugin name="laserscan" filename="libgazebo_ros_ray_sensor.so">
        <ros>
          <remapping>~/out:=scan</remapping>
        </ros>
        <output_type>sensor_msgs/LaserScan</output_type>
        <frame_name>laser_link</frame_name>
      </plugin>
      </sensor>
    </gazebo>
```

可以看到:

1. 雷达也可以设置更新频率 `update_rate` ，这里设置为5
2. 雷达可以设置分辨率，设置为1，采样数量360个，最终生成的点云数量就是360
3. 雷达也有噪声，模型为 `gaussian`
4. 雷达有扫描范围 `range` ，这里配置成0.12-3.5，0.015分辨率
5. 雷达的 `pose` 就是雷达的joint中位置的设置值

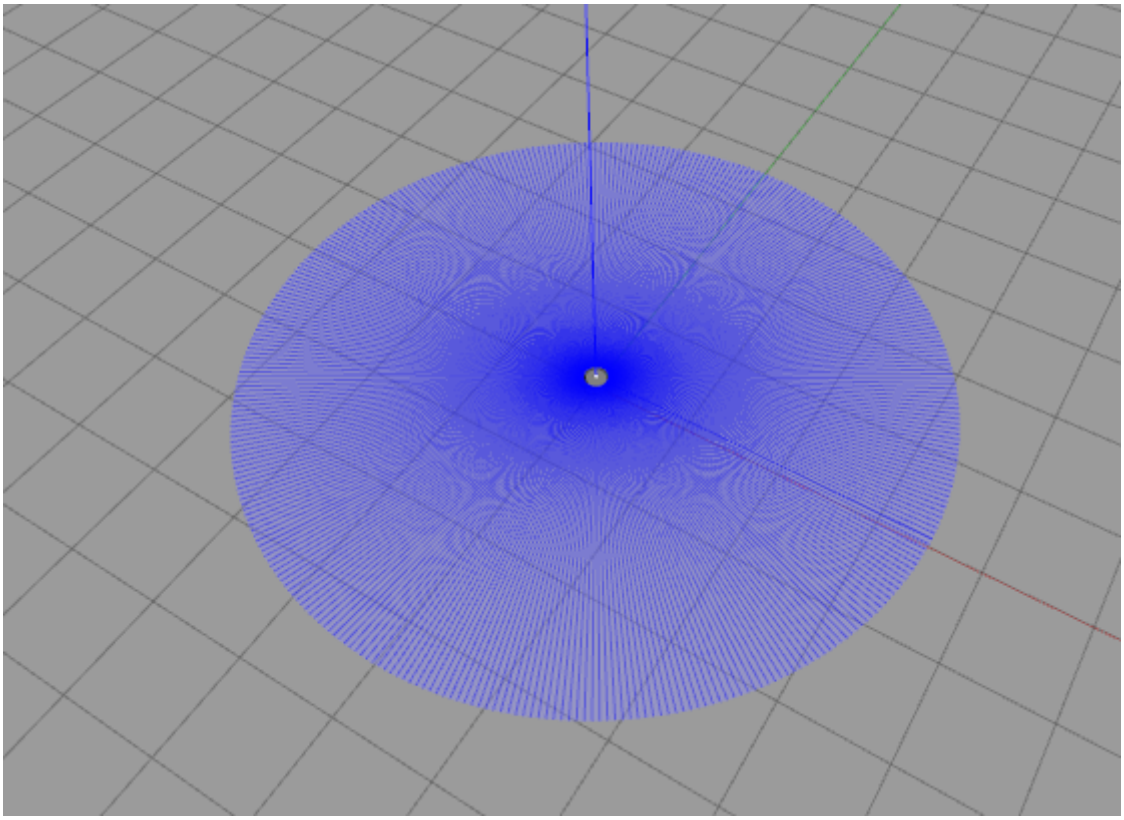
4.编译测试

编译

```shell
colcon build
```

运行

```
ros2 launch fishbot_description gazebo.launch.py
```



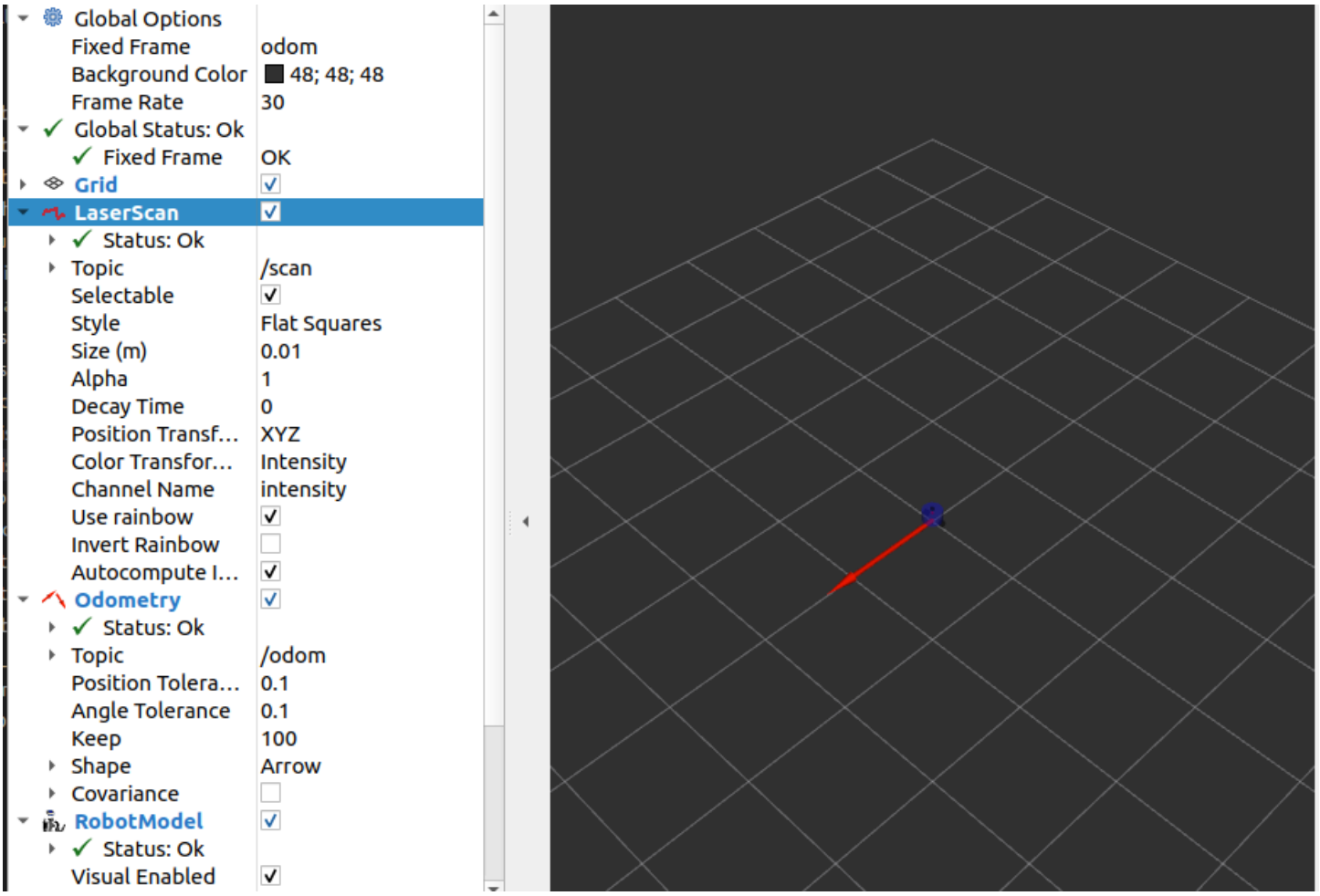
CLI看话题

```
ros2 topic list
ros2 topic info /scan
```

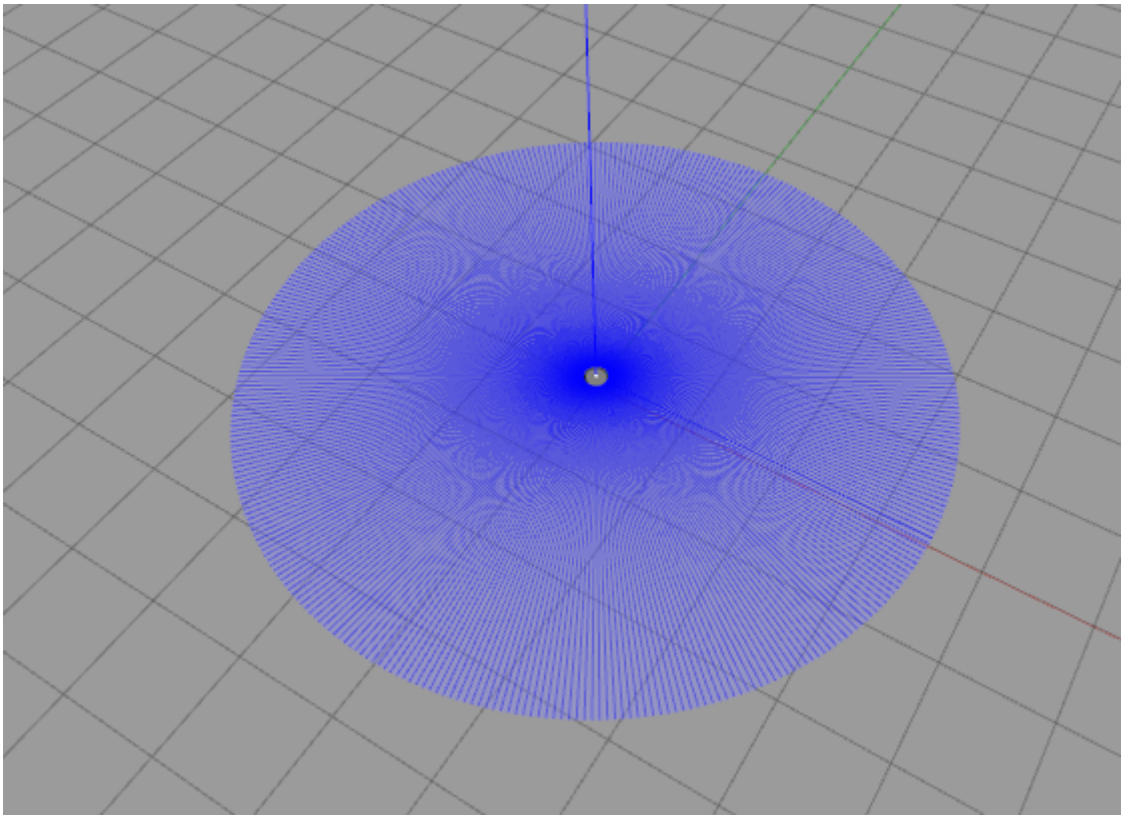
```
ros2 topic echo /scan
```

接着我们尝试使用rviz2进行可视化激光雷达数据

添加和修改RVIZ2的如下：（通过LaserScan插件可以看到激光数据）

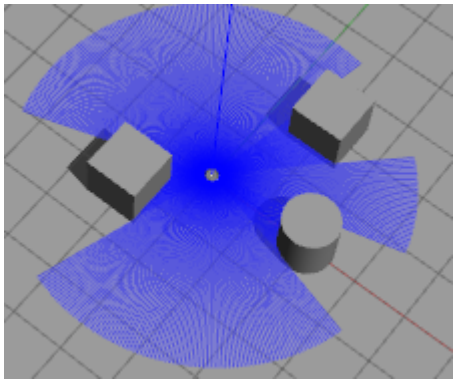


相信你改完之后依然是看不到任何激光雷达的数据的，反看topic的echo出来的数据，不是0就是inf(无限大)，再看看gazebo你会发现，激光雷达并没有达到任何一个物体上。

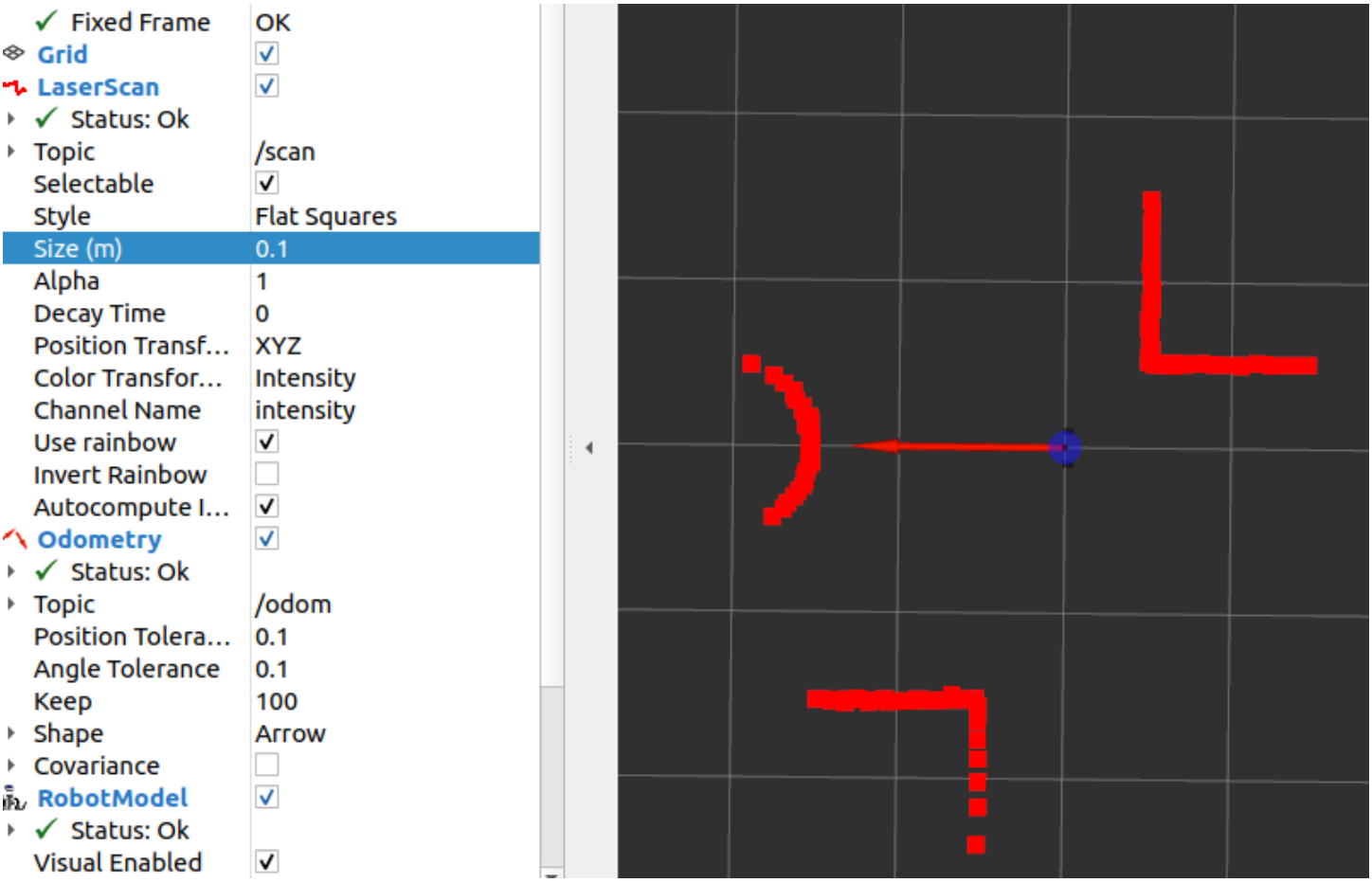


所以我们可以手动的给激光雷达周围添加一下东西，点击Gazebo工具栏的正方体，圆球或者圆柱，随意放置几个到我们激光雷达的最大扫描半径内。





接着我们再看一下RVIZ2，这里小鱼把size改大了10倍0.01->0.1。



5.总结

到这里我们就把fishbot的各个传感器都仿真出来了，第九章的内容也暂且告一段落，迎接我们的是第十章，也就是fishbot的导航仿真，nav2了。

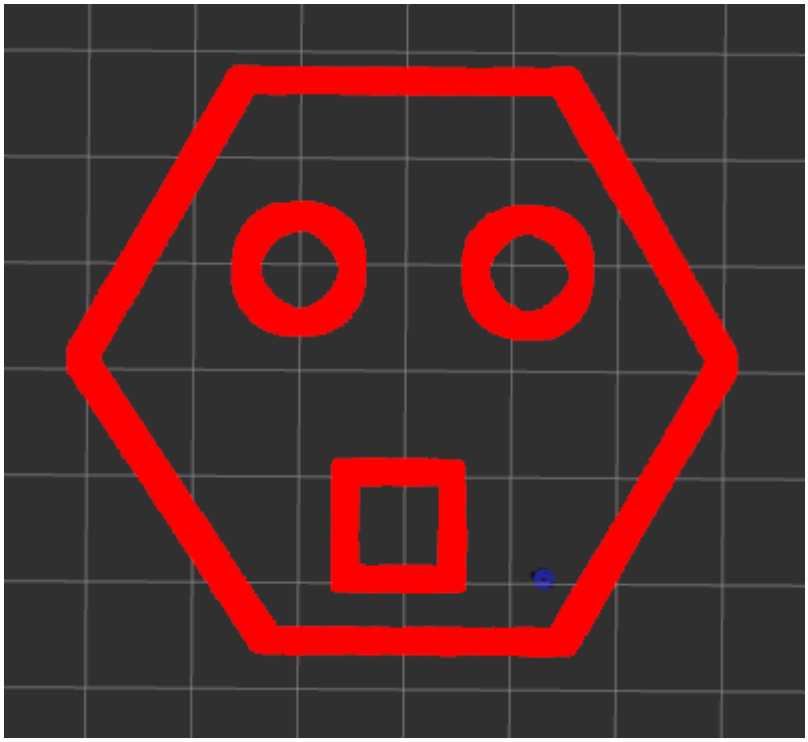
如果你想先本教程一步学习Nav2，可以到[nav2中文网](#)哦～

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

7.Gazebo仿真环境搭建

本节我们要在Gazebo中建立一个测试的环境，其实也很简单，利用Gazebo的画墙工具即可完成。



1. Gazebo的world介绍

world即世界，gazebo的world文件就是用于描述世界模型的，也就是环境模型。

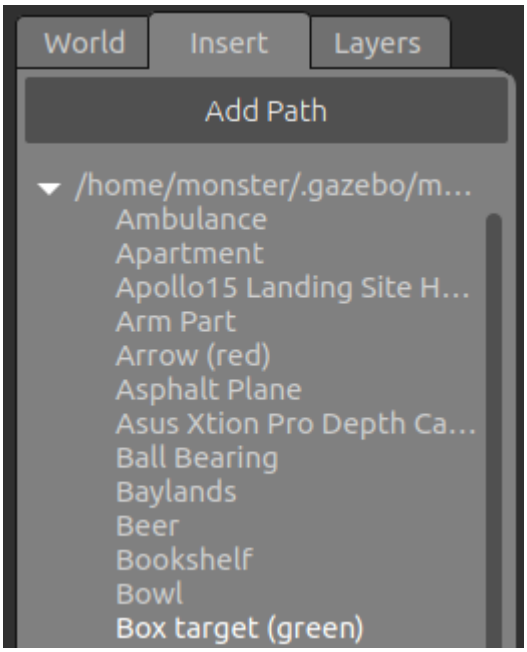
Gazebo已经为我们准备了很多常用的物体模型，除了基础的圆球，圆柱，立方体外的，其实还有飞机、汽车、房子等你现实中无法拥有的。

但是一开始安装Gazebo的时候并不会帮你下载好这些模型，需要我们手动下载，万幸的是小鱼已经帮你封装成了一行代码下载指令，打开终端，复制粘贴下面这句

```shell
cd ~/.gazebo && wget https://gitee.com/ohhuo/scripts/raw/master/gazebo_model.py && python3 gazebo_model.py
```

然后等待脚本运行完成，当然也不用等它完成，因为一共有281个模型，是逐一下载并解压到 `~/.gazebo/models/` 目录的。

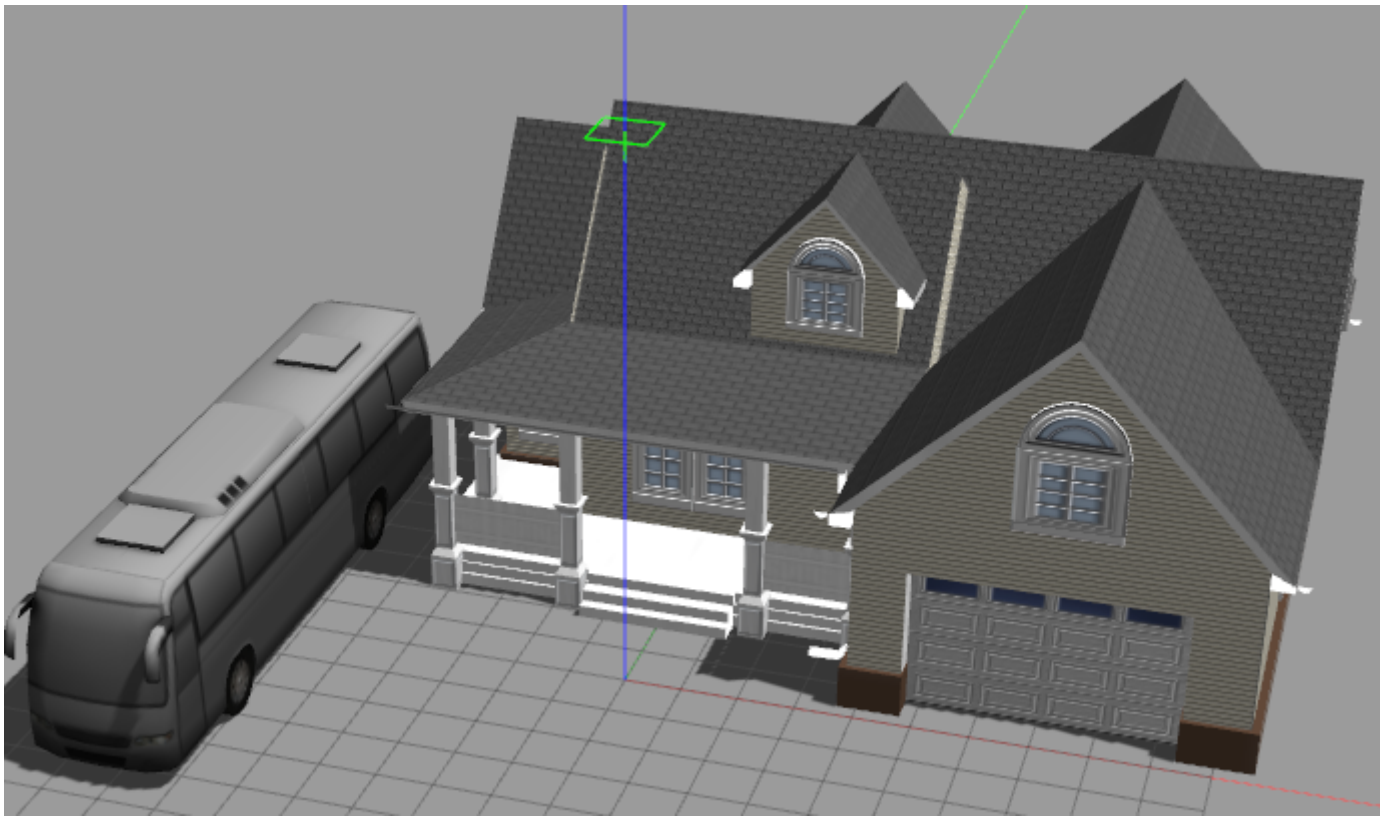
此时再次打开终端，输入 `gazebo` ，把选项卡切换到Insert



在Insert选项卡下可以看到一个目录，以及目录下的模型名称，随着下载脚本的不断下载，这里的模型会越来越多。

随手拖几个，搭建一个漂亮的环境出来~

每个成功的男人都有一辆车，小鱼也不例外

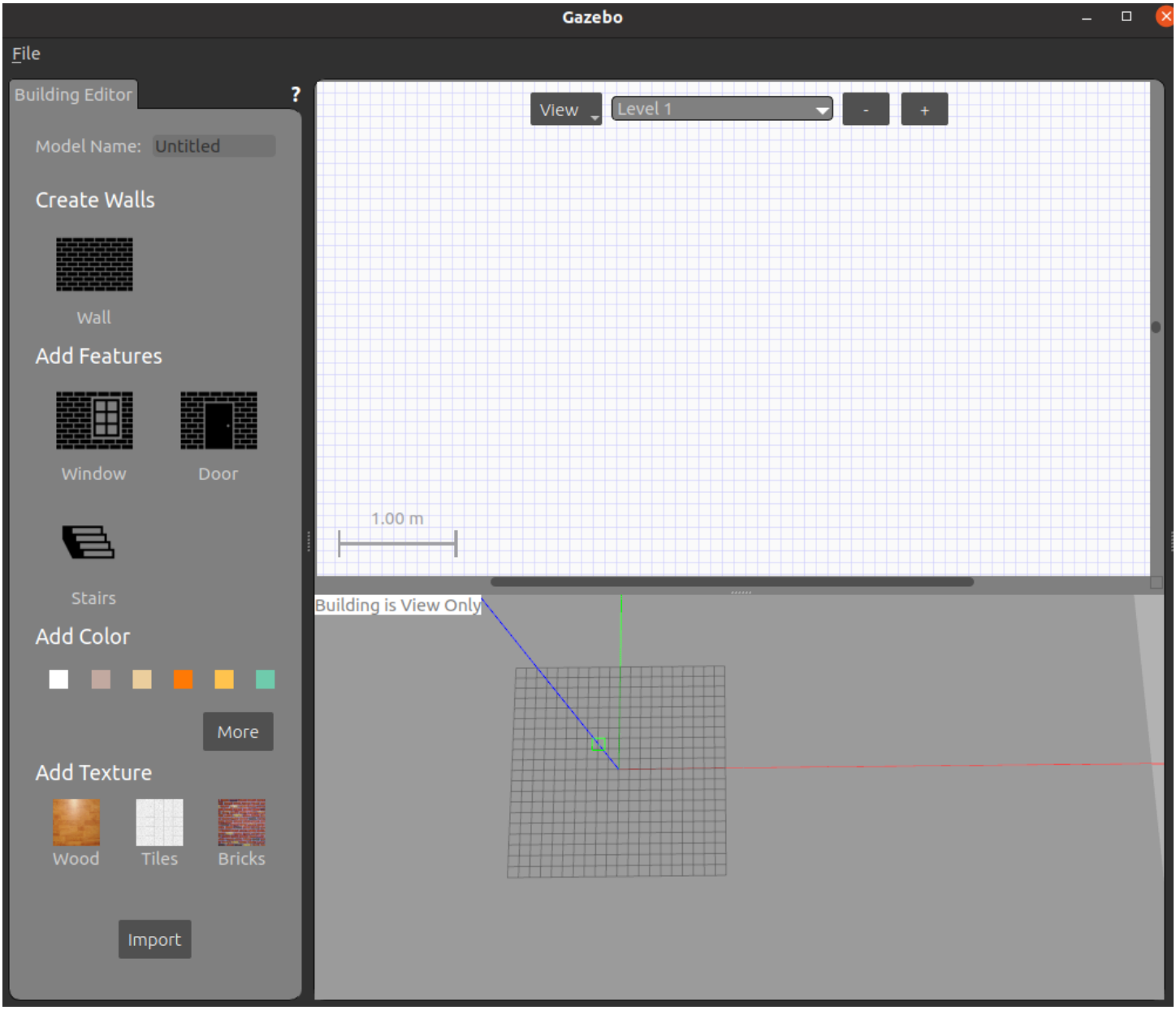


上面是Gazebo为我们准备好的开源模型，我们也可以通过Gazebo的工具来自己画一个环境。

2. 通过建墙工具建立world

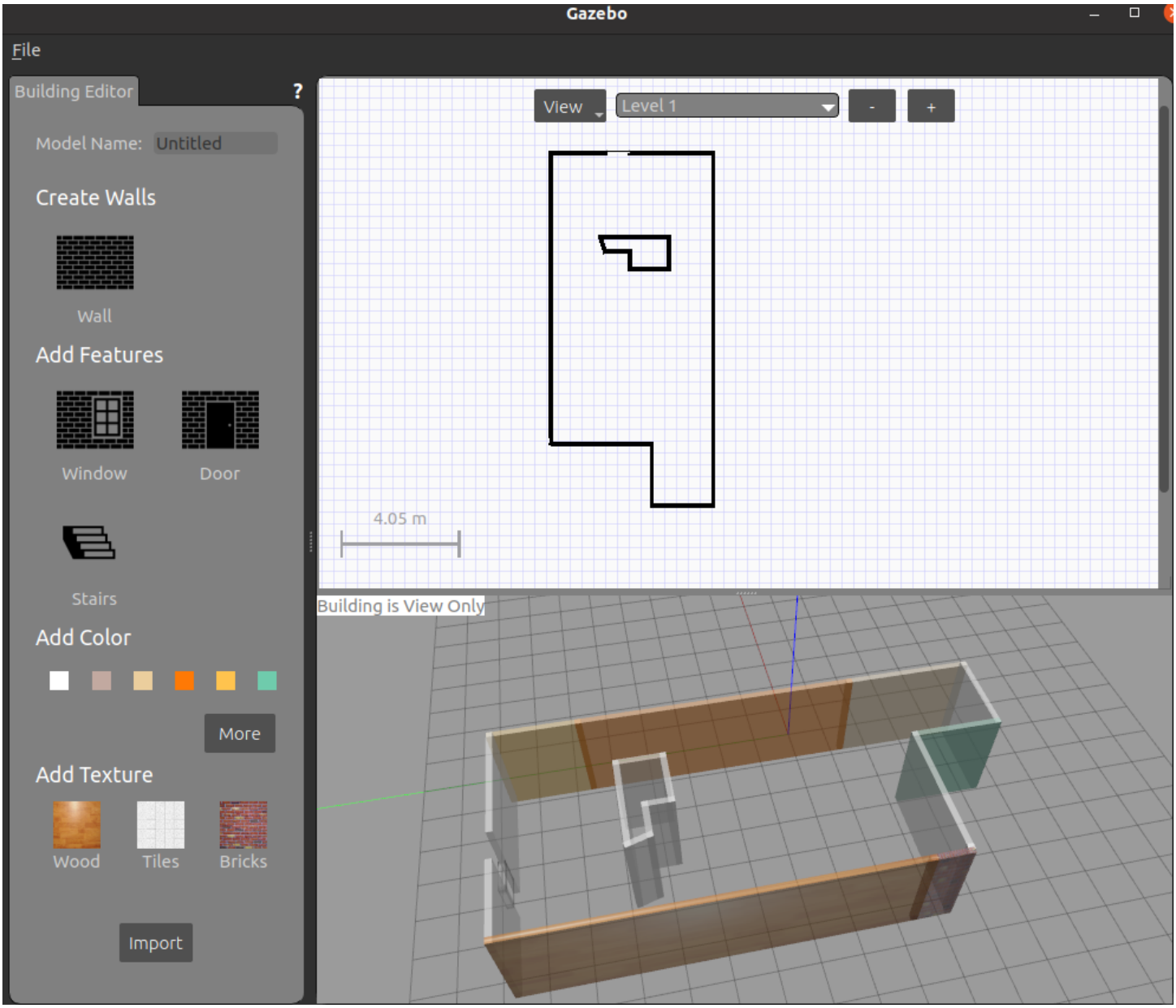
Gazebo左上角->Edit->Building Editor

接着可以看到这样一个编辑界面



2.1 随手画墙

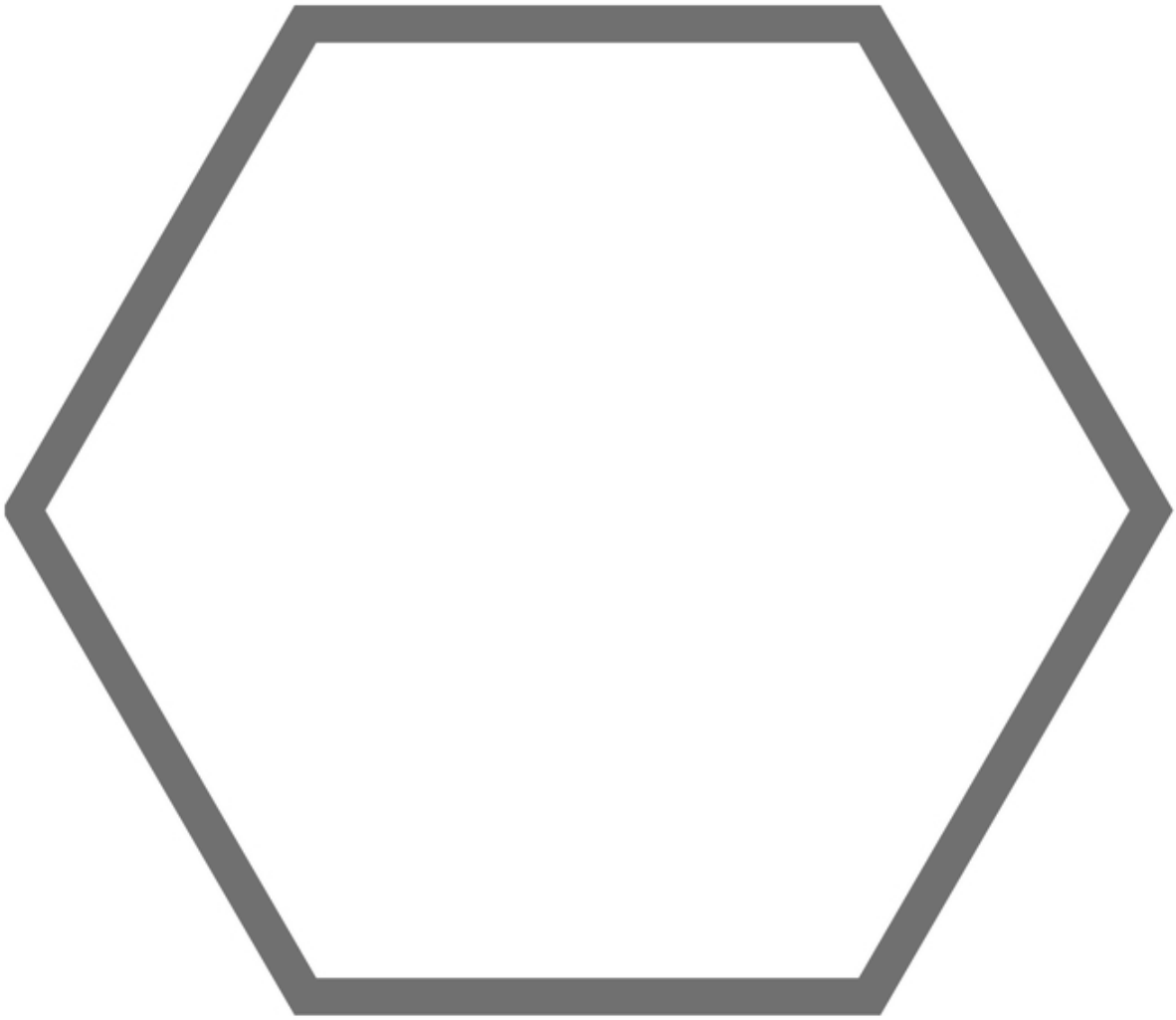
点击左边的Wall,你就可以在上方的白色区域进行建墙了。

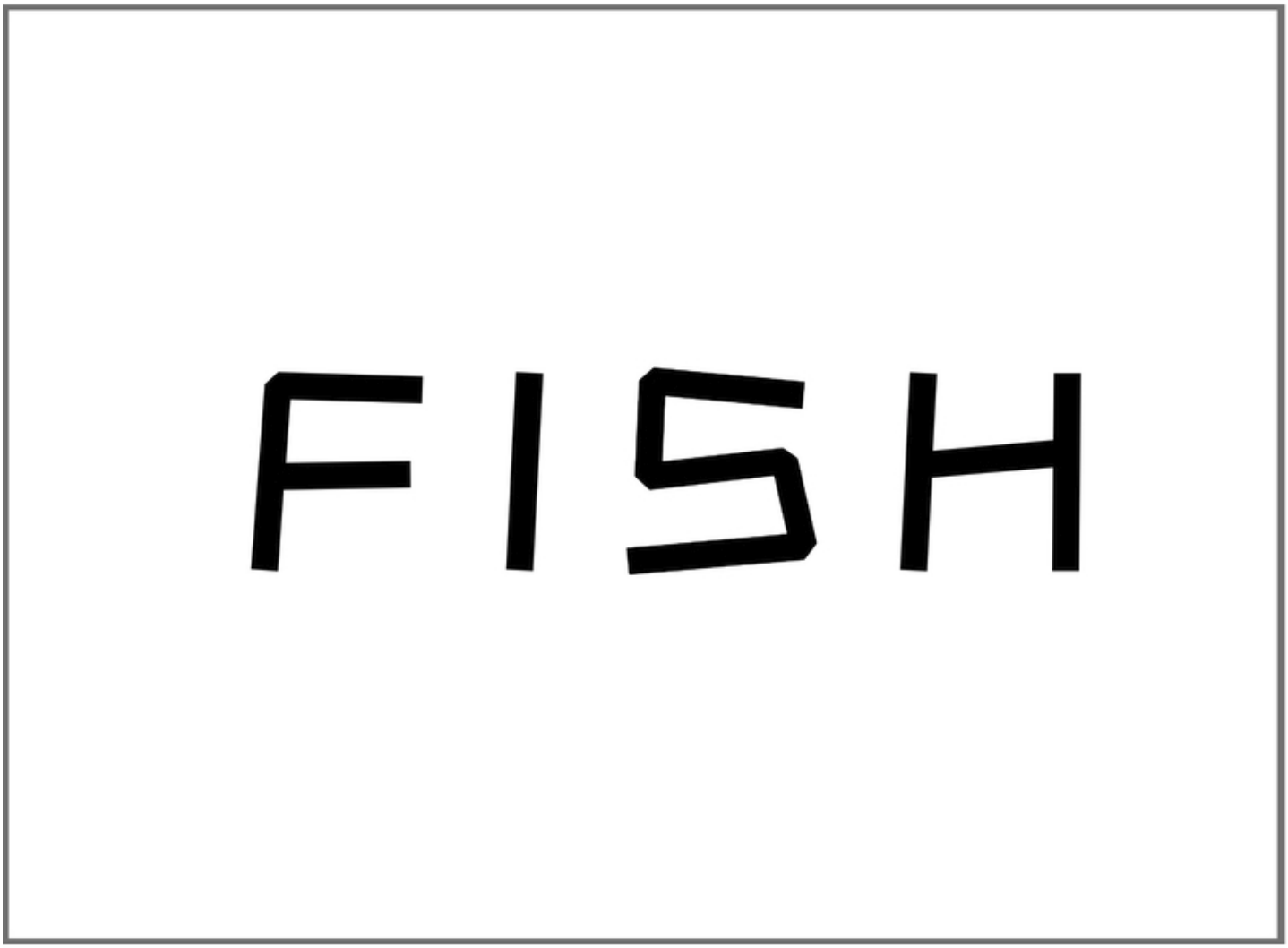


建完后还可以用选Add Color或者Add Texture，然后点击下方墙，给墙添加颜色或者纹理。

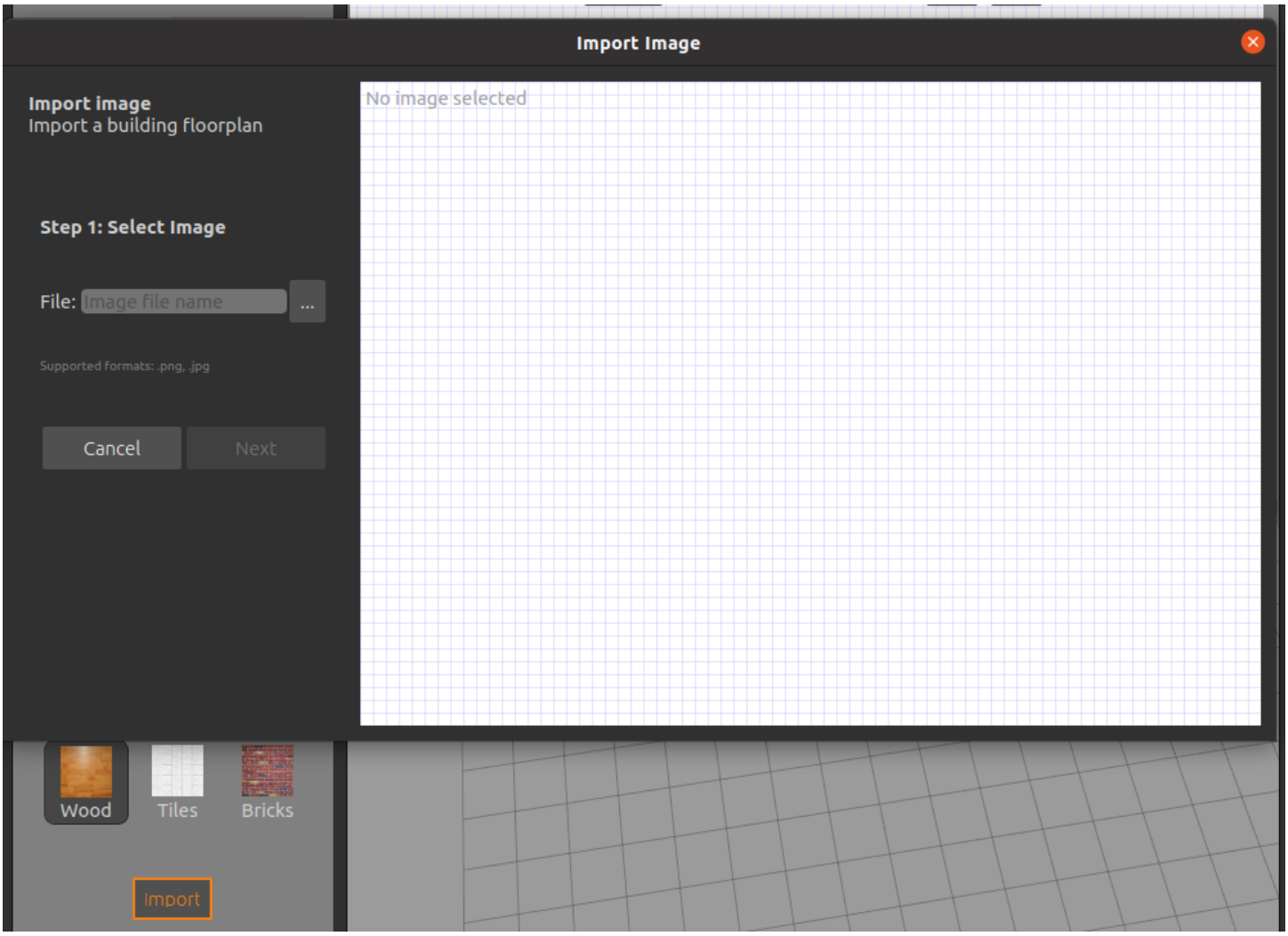
2.2 从已有地图画墙

首先你要有一个地图，小鱼为你准备了两个，两个图片都是800*600像素的。

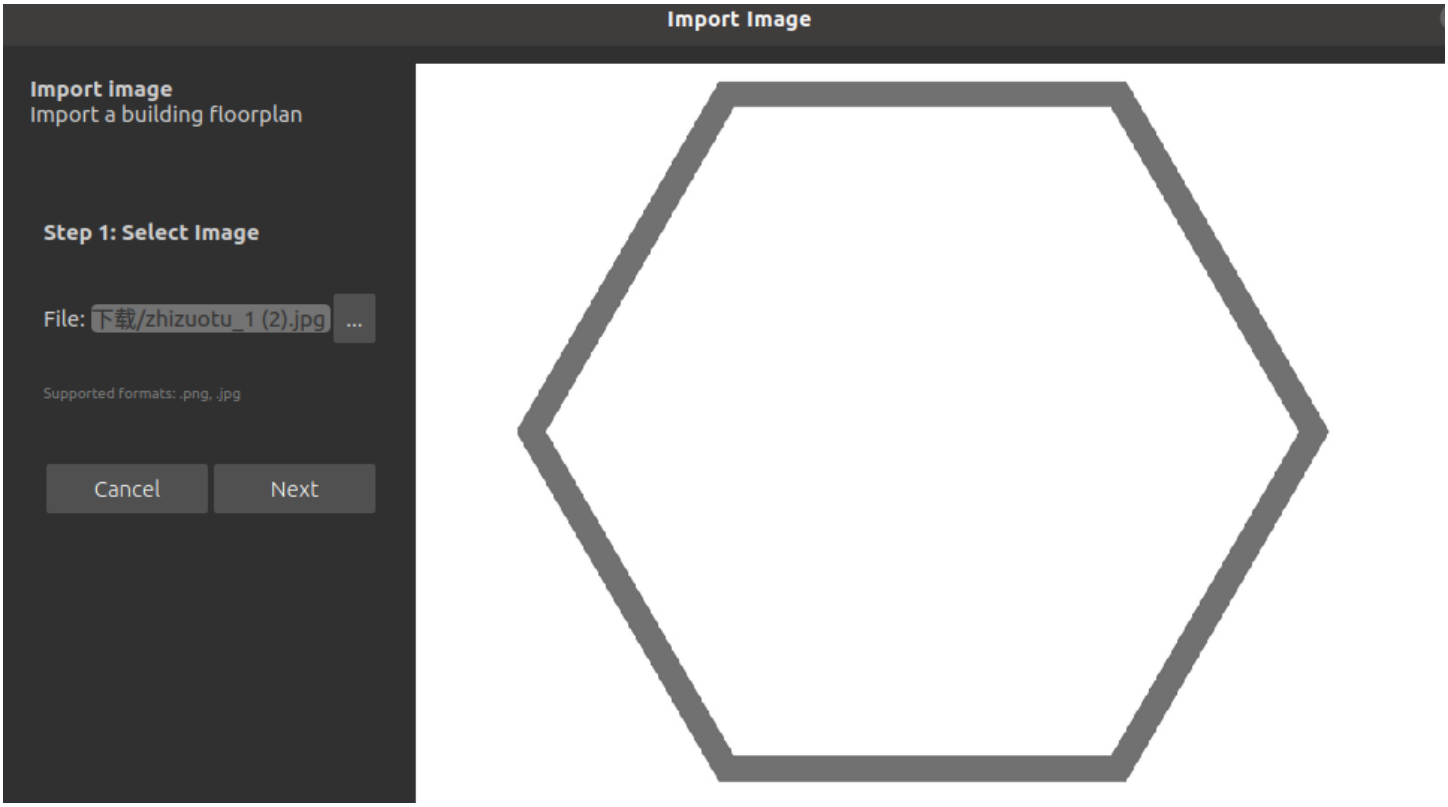




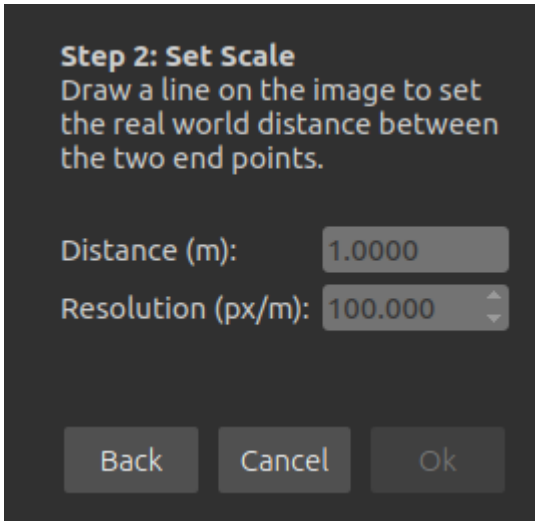
打开Gazebo->Gazebo左上角->Edit->Building Editor->左下方选Import



将上面两个图片存到本地，在这个界面选图片，记着选Next

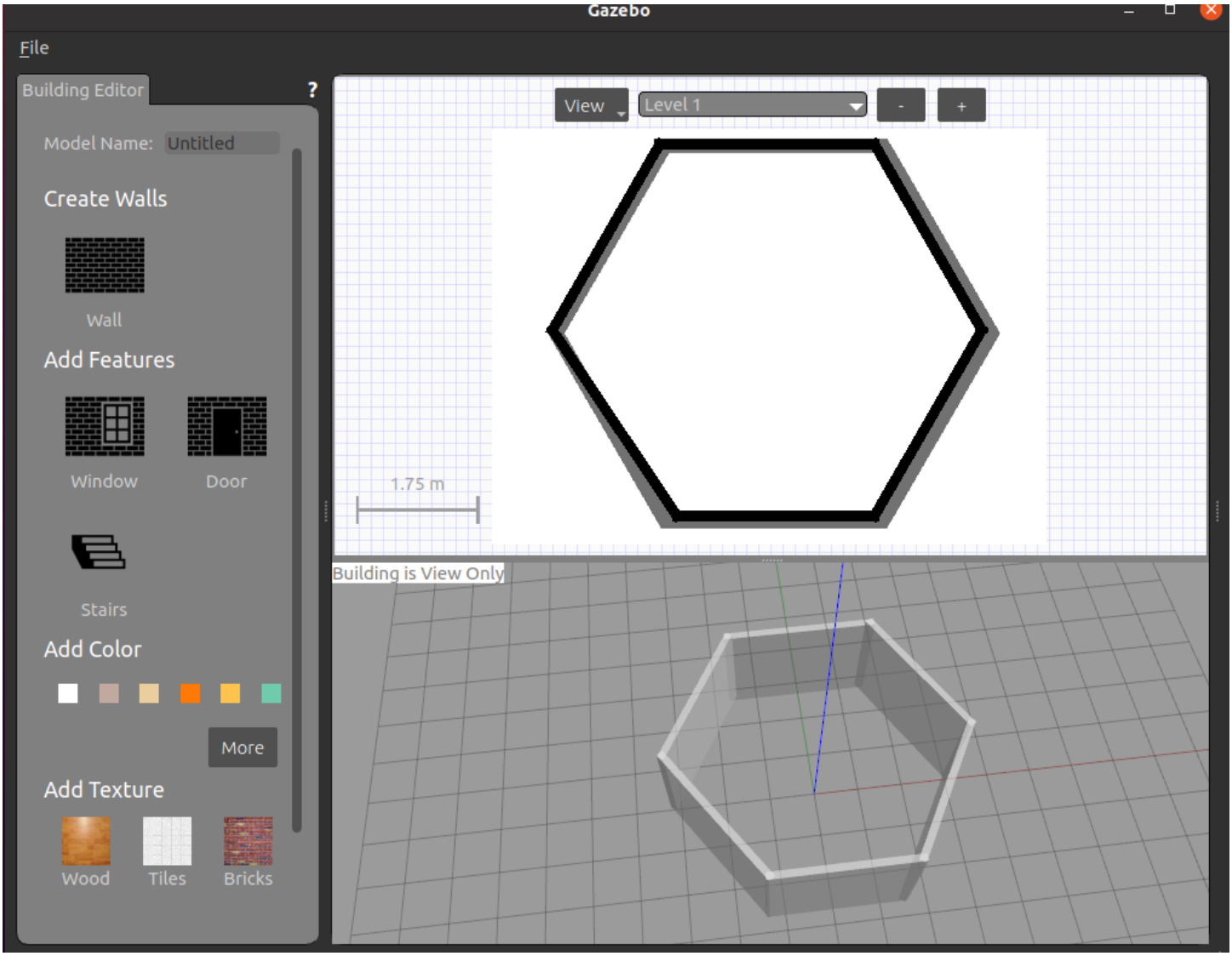


左边选尺寸对应关系



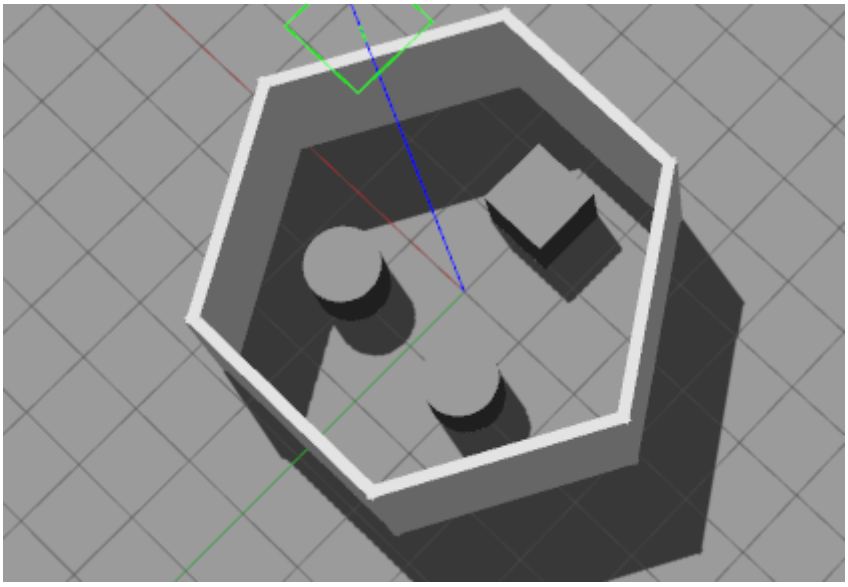
我们选择默认的，100像素/米。点击OK（需要手动将100改变一下才能点击OK哦），之后就可以用图片画墙了。

注意：导入完图片不会直接出来墙，图片只是提供了墙的大概位置，需要你手动用墙再将边描一遍。



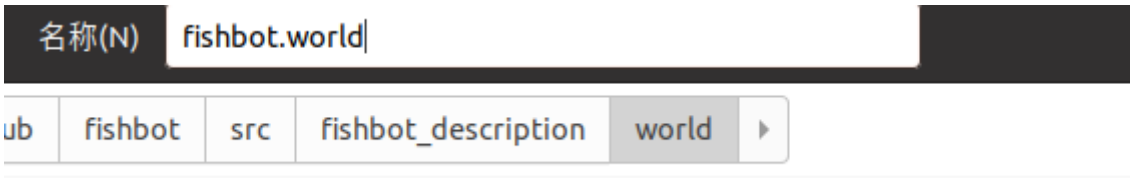
建完后点击File->Exit,在退出的弹框中选Exit。

接着在Gazebo界面中就可以看到墙了，我们再手动添加几个物体，就可以用于下面的导航使用了。

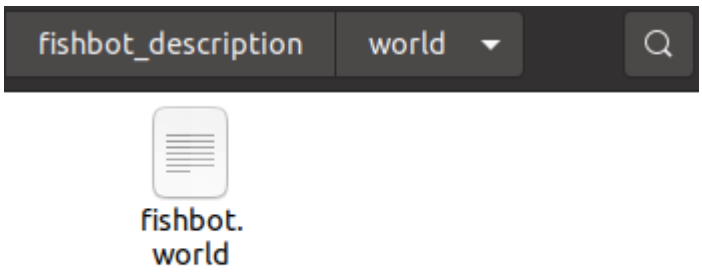


添加完，接着点击File->SaveWorld，将文件保存到我们的fishbot_descrption的world下。

> 没有world目录的小伙伴可以先手动创建下



点击右上角Sace，在文件夹就可以看到这个world文件



3.启动时加载world

3.1 命令行加载World

加载world其实也很简单，可以先启动Gazebo，再手动的加载文件，也可以在Gazebo启动时加载：

比如在前面加载ROS2插件基础上再加载fishbot.world。

```
gazebo --verbose  -s libgazebo_ros_init.so -s  libgazebo_ros_factory.so 你的world文件目录/fishbot.world
```

3.2 在launch中加载World

修改launch文件，将上面的命令行写到 `gazebo.launch.py` 中即可。

```python
    gazebo_world_path = os.path.join(pkg_share, 'world/fishbot.world')

    # Start Gazebo server
    start_gazebo_cmd =  ExecuteProcess(
        cmd=['gazebo', '--verbose','-s', 'libgazebo_ros_init.so', '-s', 'libgazebo_ros_factory.so', gazebo_world_path],
        output='screen')
```

最后记得修改setup.py文件，让编译后将world文件拷贝到install目录下

添加一行

```
        (os.path.join('share', package_name, 'world'), glob('world/**')),
```

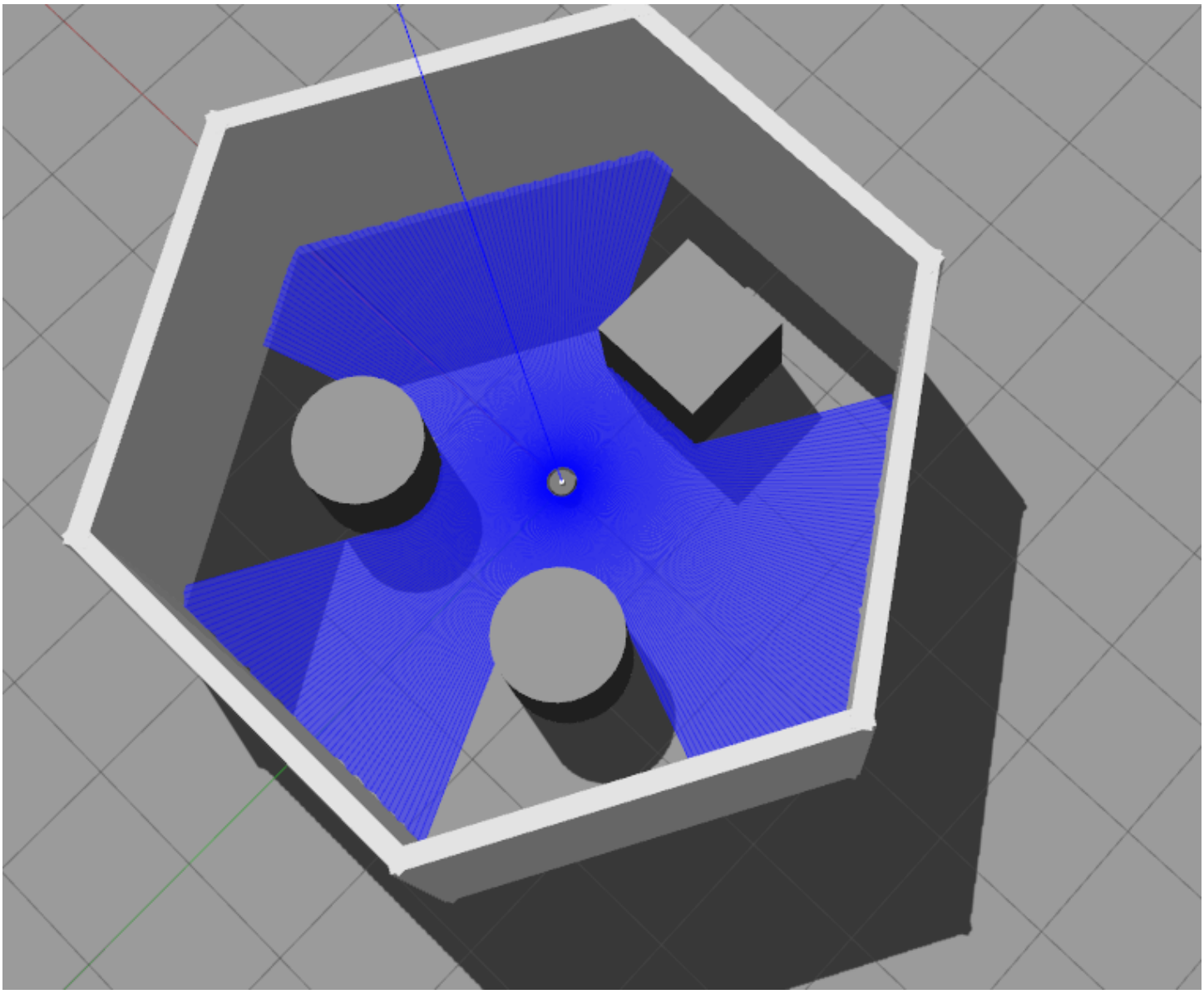
添加完后

```
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
        (os.path.join('share', package_name, 'launch'), glob('launch/*.launch.py')),
        (os.path.join('share', package_name, 'urdf'), glob('urdf/**')),
```

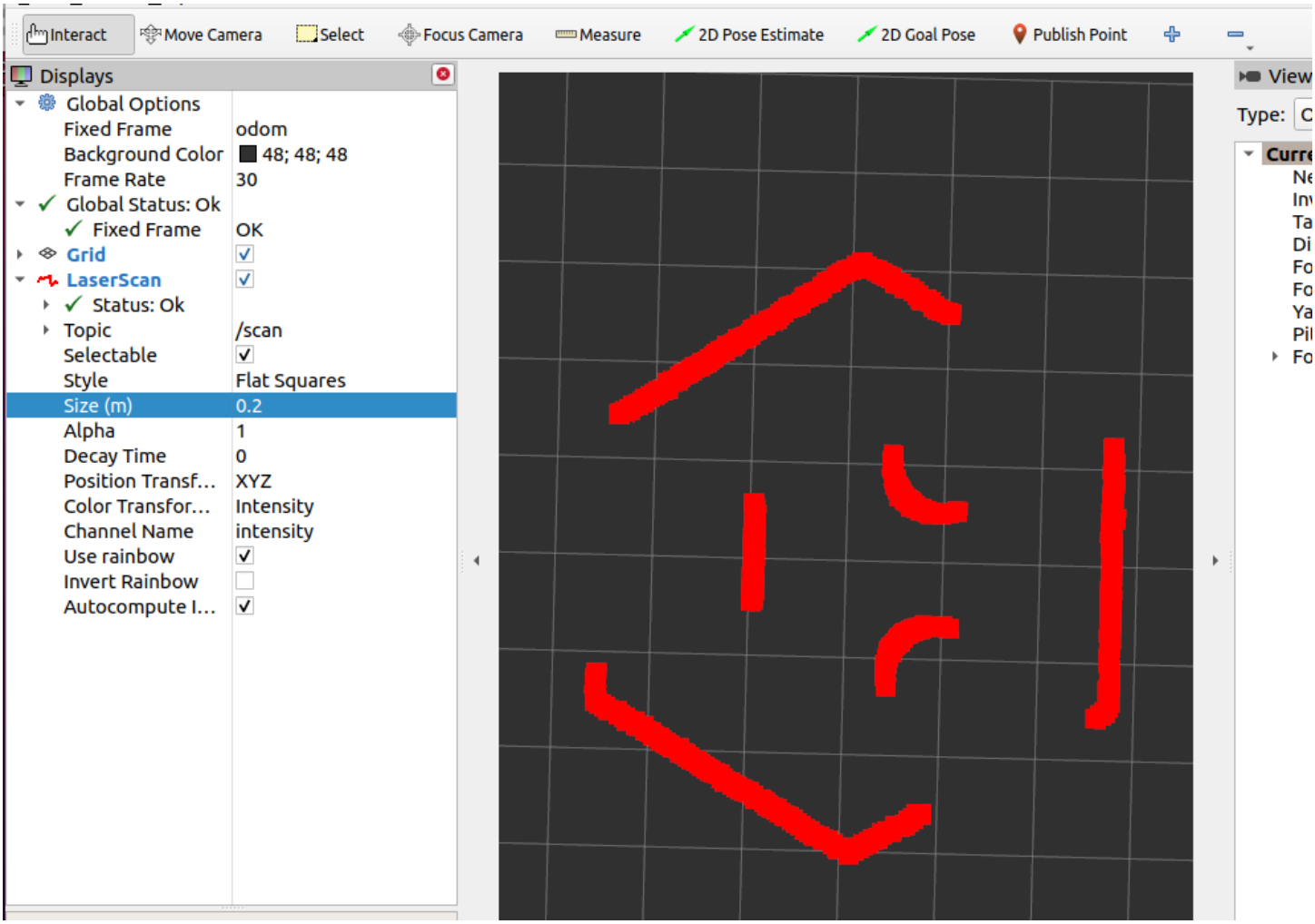
```
        (os.path.join('share', package_name, 'world'), glob('world/**')),
    ],
```

3.3 编译测试

```
colcon build
source install/setup.bash
ros2 launch fishbot_description gazebo.launch.py
```



打开RVIZ2看看雷达



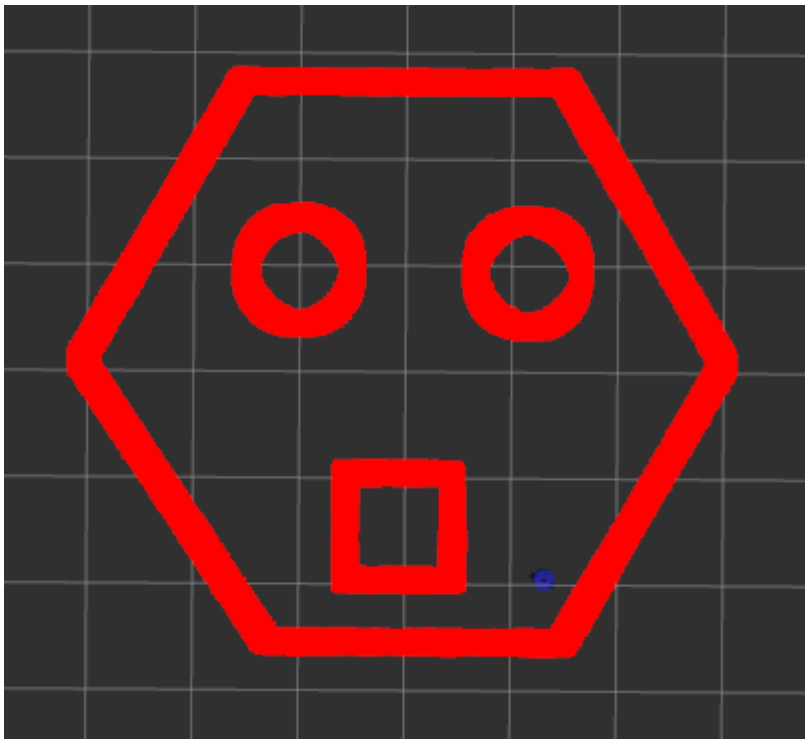
4.总结

本节我们实现了在Gazebo中简单的搭建了一个环境，下节我们就开始对SLAM建图进行介绍。

课后作业：

- 将雷达的Decay Time修改成1000,然后遥控Fishbot在环境中走一圈，然后观察雷达留下的点云形状。

正确答案如下（小鱼的环境）：



技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

1.Gazebo仿真插件之超声波

本节是拓展章节，主要讲解一下如何给FishBot添加一个超声波传感器。

在实际的机器人开发过程中，我们可能会利用超声波传感器实现实时避障的功能，毕竟超声波的价格相较于激光雷达要便宜很多（便宜的几块钱）。

所以本节我们来说一下如何使用ROS2+Gazebo来仿真超声波传感器。

1.超声波传感器介绍

百科来一段：

超声波传感器是将超声波信号转换成其它能量信号（通常是电信号）的传感器。超声波是**振动频率**高于20kHz的机械波。它具有频率高、波长短、绕射现象小，特别是方向性好、能够成为**射线**而定向传播等特点。超声波对液体、固体的穿透本领很大，尤其是在阳光不透明的固体中。超声波碰到杂质或分界面会产生显著反射形成反射回波，碰到活动物体能产生**多普勒效应**。超声波传感器广泛应用在工业、国防、生物医学等方面。

接着看看长什么样子：



便宜的就长这样子，一共两个头，一个头用于发送波，一个头接收波。这个还稍微高级一点，带一个光敏电阻，可以为超声波数据做一些补偿。

超声波传感器原理是什么呢？

```
距离=(发送时间-接收时间)*速度/2
```

看了超声波的原理，你有没有发现和前面的激光雷达传感器是一样的，是的，所以超声波传感器插件和激光雷达传感器插件在Gazebo插件中是同一个：

```
libgazebo_ros_ray_sensor.so
```

2.超声波插件配置

直接上配置，接着再解释

2.1 添加超声波关节

超声波总要装在机器人身上某个位置，所以我们先添加一个关节和Joint，为了省事，link我们就只写个名字，你如果有需要可以按照前面的章节那样添加一下。

```xml
<link name="ultrasonic_sensor_link" />

  <joint name="ultrasonic_sensor_joint" type="fixed">
    <parent link="base_link"/>
    <child link="ultrasonic_sensor_link"/>
    <origin xyz="0.07 0.0 0.076" rpy="0 0 0"/>
  </joint>
```

2.2 添加Gazebo插件

添加完了关节，我们就可以配置gazebo的插件了，gazebo插件配置如下

```xml
  <gazebo reference="ultrasonic_sensor_link">
    <sensor type="ray" name="ultrasonic_sensor">
      <pose>0 0 0 0 0 0</pose>
      <!-- 是否可视化，gazebo里能不能看到 -->
      <visualize>true</visualize>
      <!-- 扫描速率，也就是数据更新速率 -->
      <update_rate>5</update_rate>
      <ray>
        <scan>
          <!-- 水平扫描的点数 -->
          <horizontal>
```

```
            <samples>5</samples>
            <resolution>1</resolution>
            <min_angle>-0.12</min_angle>
            <max_angle>0.12</max_angle>
          </horizontal>
          <!-- 垂直方向扫描的点数 -->
          <vertical>
            <samples>5</samples>
            <resolution>1</resolution>
            <min_angle>-0.01</min_angle>
            <max_angle>0.01</max_angle>
          </vertical>
        </scan>
        <!-- 超声波检测的范围和数据分辨率单位m -->
        <range>
          <min>0.02</min>
          <max>4</max>
          <resolution>0.01</resolution>
        </range>
        <!-- 数据噪声采用高斯噪声 -->
        <noise>
          <type>gaussian</type>
          <mean>0.0</mean>
          <stddev>0.01</stddev>
        </noise>
      </ray>
      <plugin name="ultrasonic_sensor_controller"
filename="libgazebo_ros_ray_sensor.so">
        <ros>
          <!-- 重映射输出的话题名称 -->
          <remapping>~/out:=ultrasonic_sensor_1</remapping>
        </ros>
        <!-- 输出消息的类型，注意与雷达区分，这里是sensor_msgs/Range -->
        <output_type>sensor_msgs/Range</output_type>
        <!-- 射线类型，这里要写ultrasound，注意和雷达区分 -->
        <radiation_type>ultrasound</radiation_type>
        <!-- frame名称，填写link名称即可 -->
        <frame_name>ultrasonic_sensor_link</frame_name>
      </plugin>
    </sensor>
  </gazebo>
```

3.编译运行测试

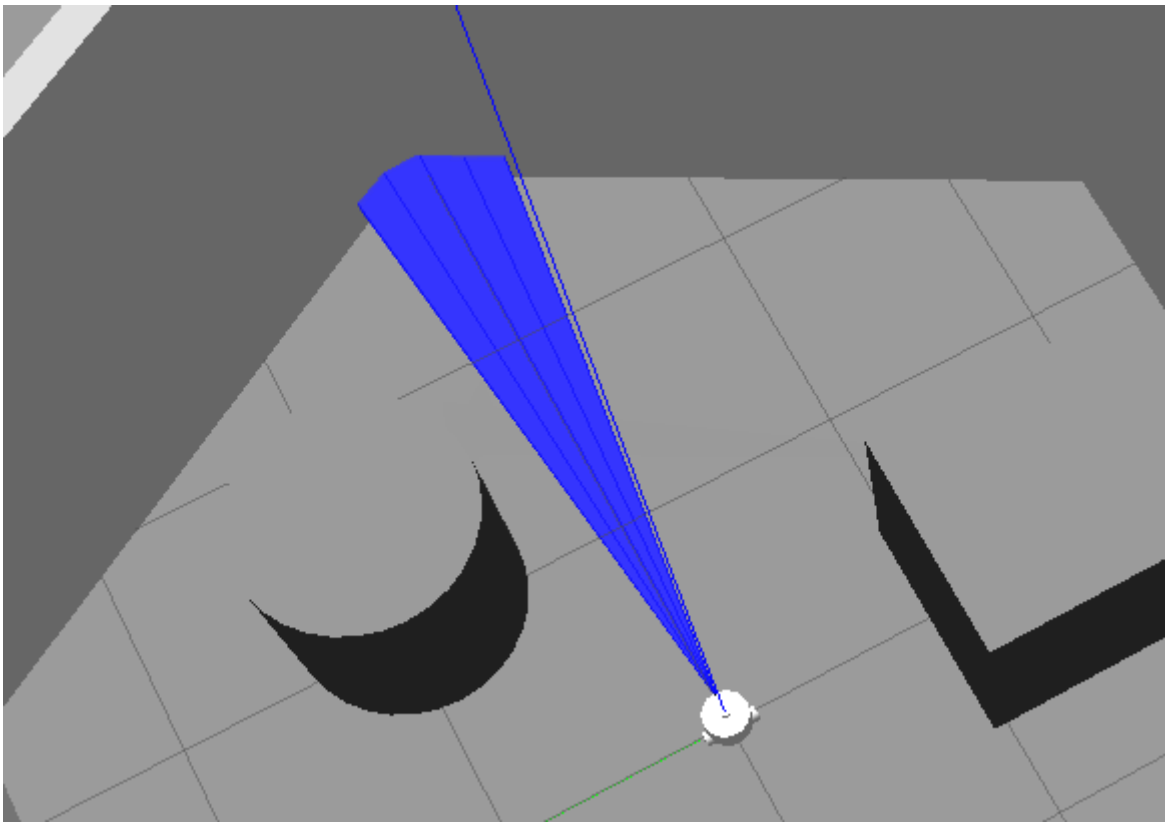
3.1Gazebo可视化

添加完成后就可以编译测试下代码

```shell
colcon build --packages-select fishbot_description
source install/setup.bash
```

```shell
ros2 launch fishbot_description gazebo.launch.py
```

没有物体的前面可以放个东西，因为本节是后面补充的，所有小鱼这里有个墙。



3.2话题数据

打开终端，输入下面指令

```
ros2 topic list
ros2 topic info /ultrasonic_sensor_1
ros2 topic echo /ultrasonic_sensor_1
```

不出意外可以看到下面的数据

```
header:
  stamp:
    sec: 4458
    nanosec: 1000000
  frame_id: ultrasonic_sensor_link
radiation_type: 0
field_of_view: 0.23999999463558197
min_range: 0.019999999552965164
max_range: 4.0
range: 2.6798219680786133
```

这里的range就是fishbot到墙之间的距离：2.67982

我们来讲一讲超声波传感器的数据类型 `sensor_msgs/msg/Range`

```
# ros2 topic info /ultrasonic_sensor_1
Type: sensor_msgs/msg/Range
Publisher count: 1
Subscription count: 0
```

你可以使用 `ros2 interface show  sensor_msgs/msg/Range` 看到详细的解释，我们翻译一下

```
# Single range reading from an active ranger that emits energy and reports
# one range reading that is valid along an arc at the distance measured.
# This message is  not appropriate for laser scanners. See the LaserScan
# message if you are working with a laser scanner.
#
# This message also can represent a fixed-distance (binary) ranger.  This
# sensor will have min_range===max_range===distance of detection.
```

```
# These sensors follow REP 117 and will output -Inf if the object is detected
# and +Inf if the object is outside of the detection range.

std_msgs/Header header # timestamp in the header is the time the ranger
                            # returned the distance reading

# Radiation type enums
# If you want a value added to this list, send an email to the ros-users list
uint8 ULTRASOUND=0
uint8 INFRARED=1

uint8 radiation_type    # 传感器射线类型
                        # (sound, IR, etc) [enum]

float32 field_of_view   # 距离数据对应的弧[rad]的大小，测量物体的范围介于
                        # -field_of_view/2 到 field_of_view/2 之间。
                        # 0 角度对应于传感器的 x 轴。

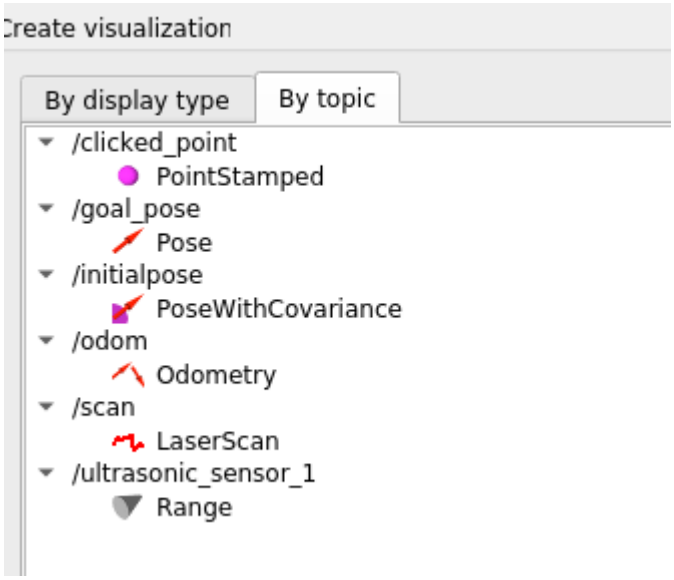
float32 min_range       # 最小范围值 [m]
float32 max_range       # 最大范围值 [m]
                        #  固定距离需要 min_range==max_range

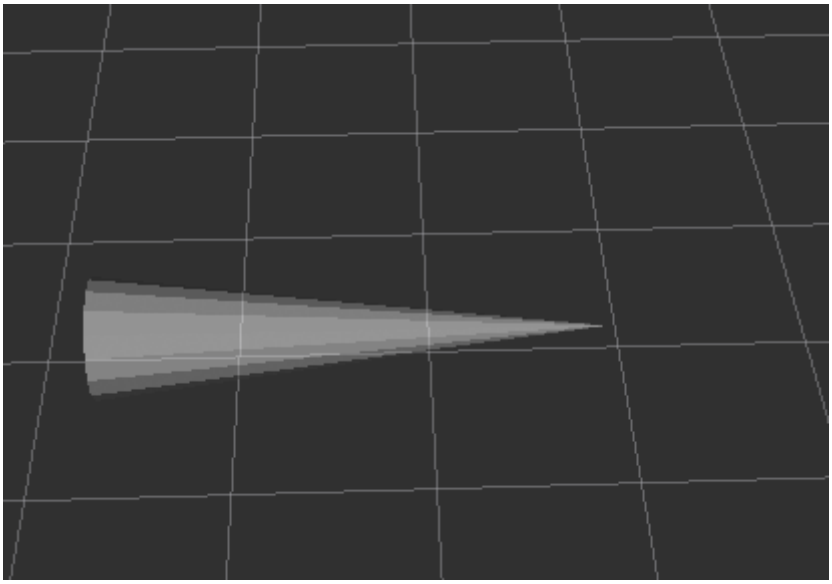
float32 range           # 范围数据 [m]
                        # (Note: values < range_min or > range_max should be
discarded)
                        # Fixed distance rangers only output -Inf or +Inf.
                        # -Inf represents a detection within fixed distance.
                        # (Detection too close to the sensor to quantify)
                        # +Inf represents no detection within the fixed distance.
                        # (Object out of range)
```

结论，主要关注range就可以了。

3.3 在RVIZ2中可视化超声波数据

Add ->By topic->Range





4.总结

本节主要介绍了如何对超声波传感器进行仿真，在导航的过程中我们通常把超声波放到一个代价地图中进行停障等行为。所以学习其仿真还是很有用的。

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

1.图像常见格式及存储

在日常生活中，我们会用到各种各样的地图，比如交通轨道图、城市地图、世界地图。

1.地图分类

我们根据地图所表达信息的不同可以将地图分为三类：

1.1 尺度地图（Metric Map）

尺度地图用于表示尺寸距离，可以理解为把真实世界按比例缩小，尺度地图中每个点都可以使用一个经纬值进行表示。

> 因为尺度地图是按照真实世界按比例缩小，所以要有比例尺来表示缩小比例。

1.2 拓扑地图（Topological Map）

拓扑地图用于表示点与点之间的连接信息，比如地铁轨道交通图。

1.3 语义地图（Semantic Map）

语义地图可以理解为在上面两个地图上增加了语义，比如尺度地图中某处是红绿灯、斑马线。拓扑地图中某个点是深圳北站。

在机器人领域，尺度地图常用于定位于地图构建（Mapping）、定位（Localization）和同时定位与地图构建（Simultaneous Localization And Mapping，SLAM），拓扑地图常用于路径规划（Path Planning），而语义地图常用于人机交互（Human Robot Interaction）。

2.地图的存储

地图也是图，所以地图一般以图片的形式存储下来，常见的格式比如 `png` 就是可以存储ROS2中的二二维地图的。

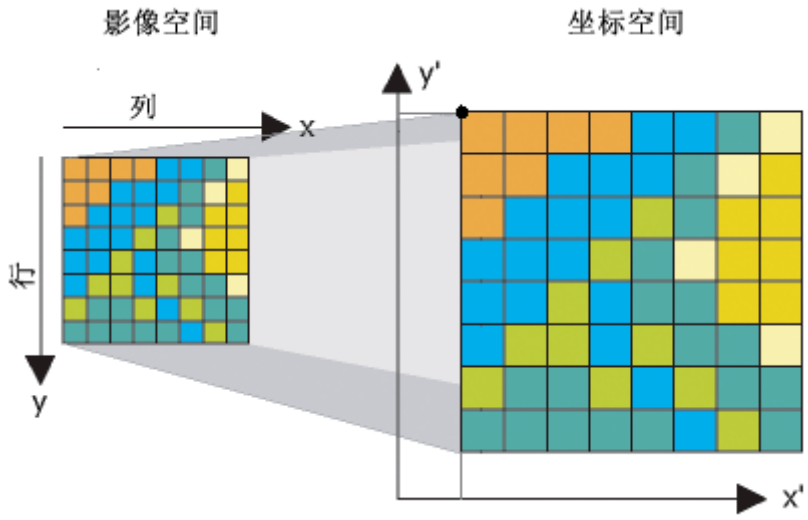
但是对于一些三维的点云图，二维的方式就不怎么好存储，就需要一些特殊格式了，因为我们后面不涉及三维空间建图，这里就不在展开了。

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

2.占据栅格地图

我们先了解下什么是栅格地图（Grid Map）。



如上图将地图数据分割为一块块的栅格来表达地图信息，就是栅格地图。

那什么是占据（Occupancy）呢？

机器人通过激光雷达等传感器来感知深度信息，但我们知道，传感器都是有噪声的（在前面的机器人仿真建模过程中，我们为了更加真实给激光雷达还添加了高斯噪声），所以机器人前方的某个位置到底有没有物体（障碍物）是不确定的。

我们可以采用概率来解决这一问题，认为确实有物体的栅格的占据率为100%，确定没有物体的栅格占据率为0%，不确定的栅格就用（确认占据概率/确认非占据概率）值表示占据率。

由此我们知道**占据栅格地图就是一张写满占据率的格子组成的地图**。

1	1	1	0.5
0	0	0	0.5
0	0	0	0.5
0	0	0	0.6

> **我们在做机器人的路径规划的时候，需要确定一个格子是有障碍物？没障碍物？还是未知呢？所以我们一般会设定两个阈值：**
>
> - 占据阈值（occupied_thresh），比如0.65，则表示栅格占据率大于0.65的认为是有障碍物。
> - 空闲阈值（free_thresh），比如0.25，则表示栅格占据率小于0.25的认为没有障碍物。
>
> **那在free_thresh和occupied_thresh之间的则认为是未知区域（未探索）。**

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

10.3 SLAM技术概述

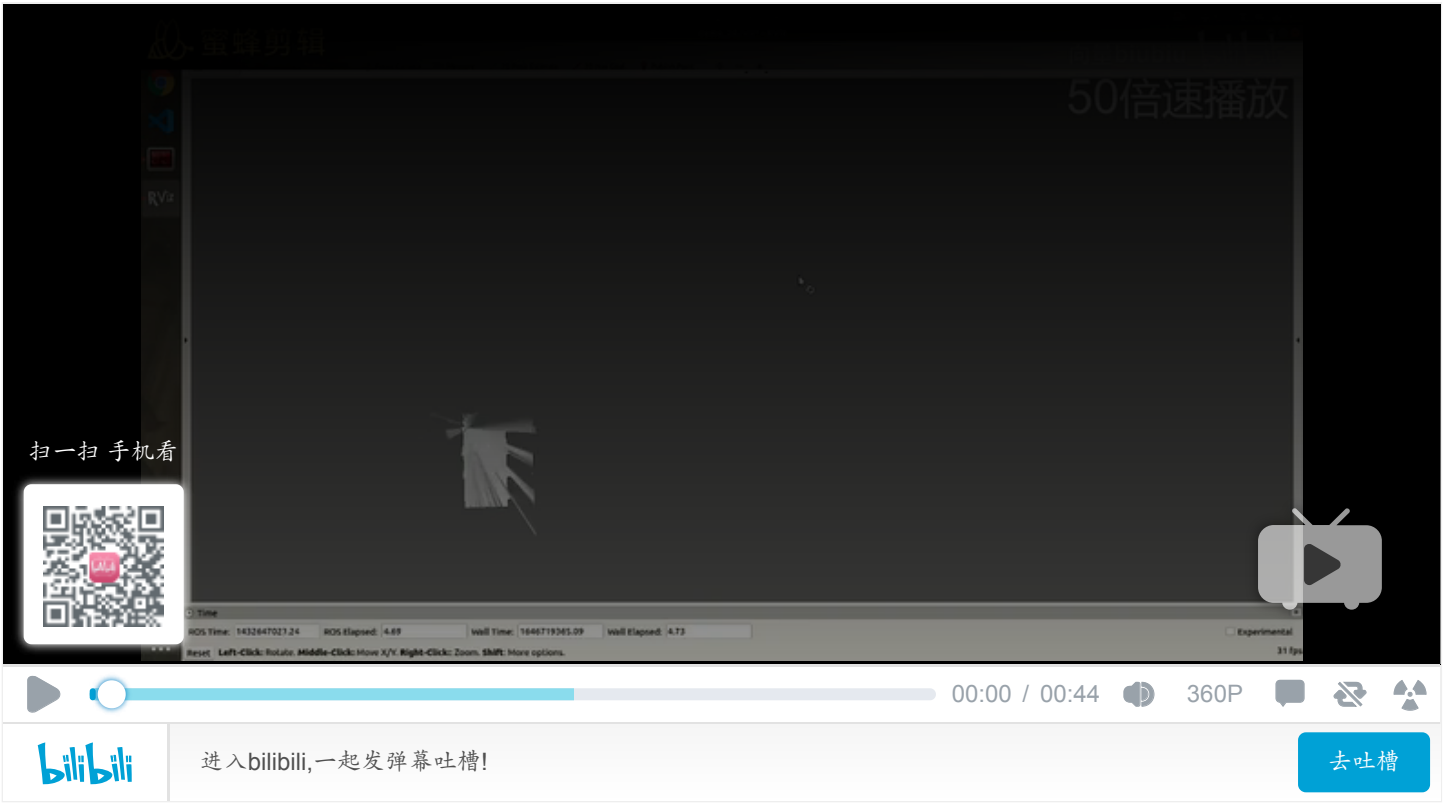
小伙伴们好，上一节我们搭建好了Gazebo中的仿真环境，本节课我们一起来了解一下SLAM技术。

第一节中我们知道，要解决机器人自主导航问题就需要有感知（建图和定位）参与，通过感知输出机器人当前环境的地图信息和位置。而本节要讲的SLAM就是解决地图和定位问题的。

1. SLAM是什么

SLAM是同步定位与地图构建(Simultaneous Localization And Mapping)的缩写。

先通过一个视频直观了解一下



视频中随着机器人的移动，机器人的传感器获取到了环境信息，然后完成对博物馆地图的构建。

> 有小伙伴可能会问，只看到建图没看到定位呀。细心观察可以发现建图的过程中，其实一直都在计算机器人的位置。

2. SLAM如何解决建图定位问题

机器人通过自身传感器数据处理进行位置估计，同时通过不断移动完成对整个未知环境的地图构建。这就是SLAM解决的问题。

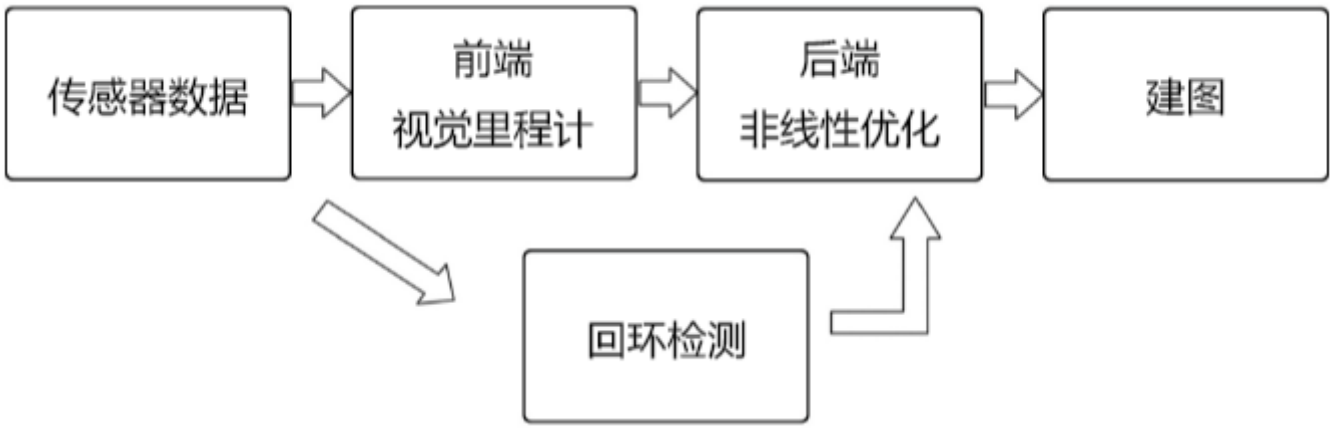
那又是如何解决的呢？SLAM实现的方案很多，但是几个比较关键的技术如下：

1. 传感器感知 通过各类传感器实现对环境的感知，比如通过激光雷达获取环境的深度信息。同时可以通过传感器融合来提高位置估计的精度，比如融合轮式里程计、IMU、雷达、深度相机数据等。

2. 视觉/激光里程计 基本原理是通过前后数据之间对比得出机器人位置的变化。

3. 回环检测 判断机器人是否到达之前到过的位置，可以解决位置估计误差问题，建图时可以纠正地图误差。

经典视觉SLAM结构



3.SLAM算法分类

从算法的对数据的处理方式上看，目前常用的SLAM开源算法可以分为两类

1.基于滤波，比如扩展卡尔曼滤波（EKF: Extended Kalman Filter）、粒子滤波(PF: Particle Filter)等。

ROS中的gmapping、hector_slam算法都是基于滤波实现的。

2.基于图优化，先通过传感器进行构图，然后对图进行优化。

目前比较主流的是图优化的方法，Cartographer就是基于图优化实现的。图优化相对于滤波，不用实时的进行计算，效率更高，消耗的资源更少，所以在实际场景中使用的更多。

4.SLAM开源库

4.1. Cartographer

github地址：https://github.com/cartographer-project/cartographer

Cartographer是一个可跨多个平台和传感器配置以2D和3D形式提供实时同时定位和建图（SLAM）的系统。

4.2. ORB_SLAM2(纯视觉)

github地址：https://github.com/raulmur/ORB_SLAM2

ORB-SLAM2是用于单目，双目和RGB-D相机的实时SLAM库，用于计算相机轨迹和稀疏3D重建

4.3 VINS

github地址：https://github.com/HKUST-Aerial-Robotics/VINS-Mono

VINS-Mono是单目视觉惯性系统的实时SLAM框架。它使用基于优化的滑动窗口配方来提供高精度的视觉惯性测距。

5.总结

本节课我们简单介绍了下感知部分的技术担当SLAM，并对常用的开源库进行介绍，下一节我们就对其中的Cartograpger开源库进行介绍和安装。

参考文章：

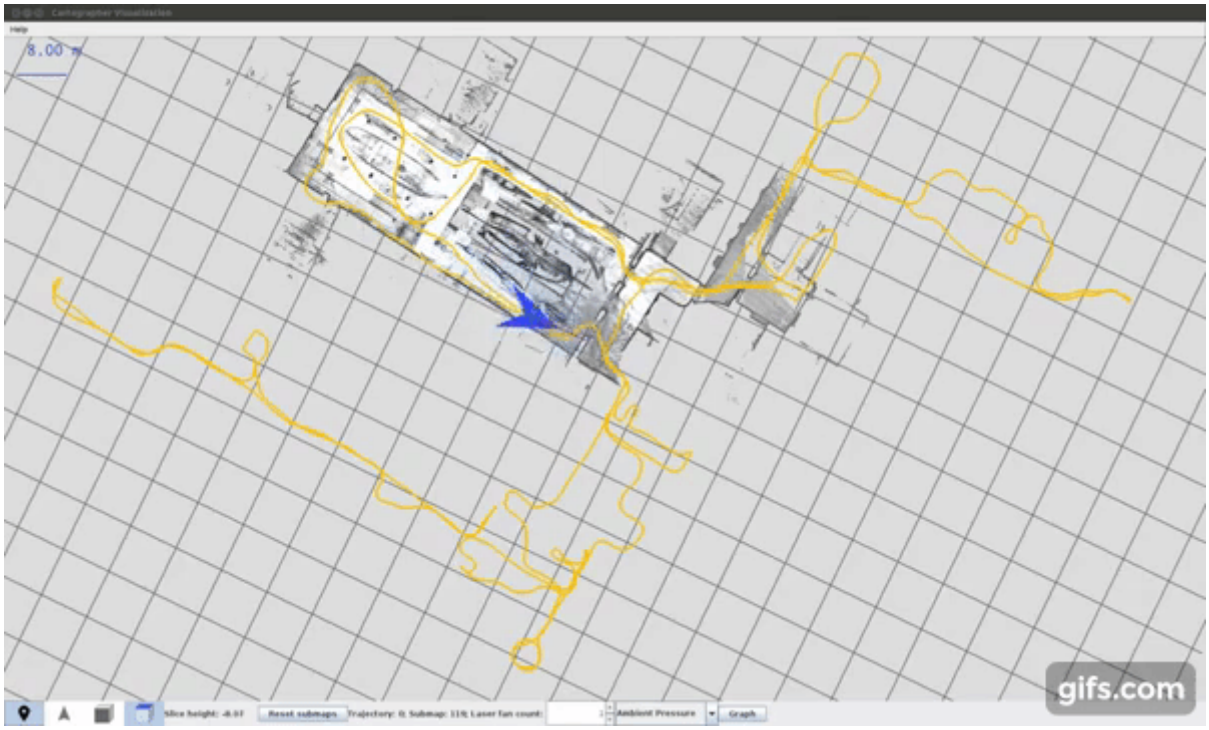
- https://blog.csdn.net/heyijia0327/article/details/47686523

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AiIotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

2. Cartographer介绍与安装

1.Cartographer介绍

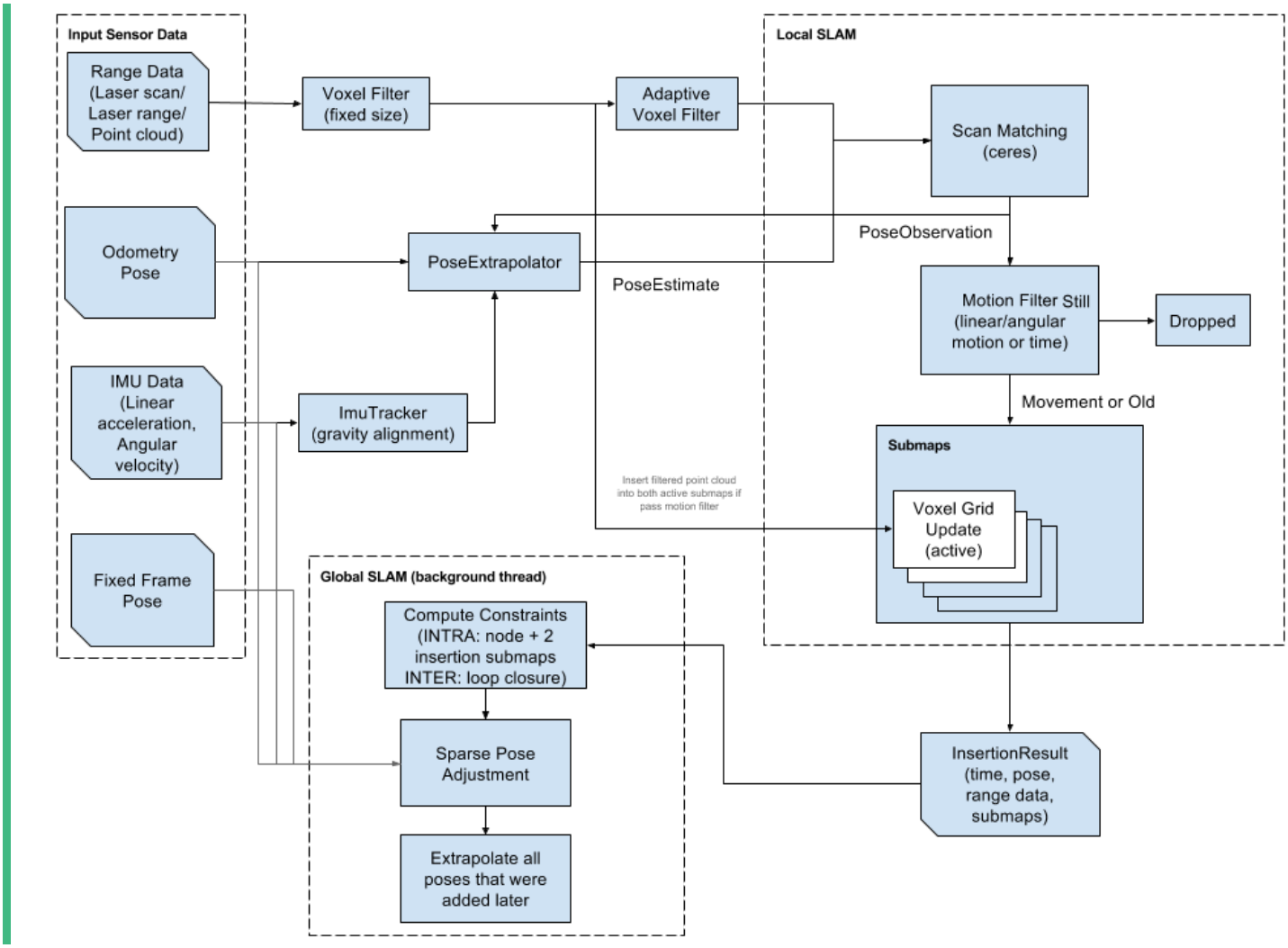


Cartographer是Google开源的一个可跨多个平台和传感器配置以2D和3D形式提供实时同时定位和建图（SLAM）的系统。

> github地址：https://github.com/cartographer-project/cartographer 文档地址：https://google-cartographer.readthedocs.io/en/latest

在小鱼呆过的机器人公司里，建图最终方案都是采用了Cartographer，甚至花费大量人力物力对Cartographer算法进行裁剪，这足以表明Cartographer算法的优越性。

> Cartographer系统架构概述（简单看看即可，如果大家后面确定研究方向是SLAM可以深入学习）：
>
> 简单的可以看到左边的可选的输入有深度信息、里程计信息、IMU数据、固定Frame姿态。



2.Carttographer安装

2.1 apt安装

安装 `carotgrapher`

```shell
sudo apt install ros-humble-cartographer
```

需要注意我们不是直接使用 `cartographer` ，而是通过 `cartographer-ros` 功能包进行相关操作，所以我们还需要安装下 `cartographer-ros`

```shell
sudo apt install ros-humble-cartographer-ros
```

2.2 源码安装

小鱼比较推荐源码安装的方式，毕竟是以学习为目的，我们后面要稍微看一下源码。

将下面的源码克隆到fishbot_ws的src目录下：

```shell
git clone https://ghproxy.com/https://github.com/ros2/cartographer.git -b ros2
git clone https://ghproxy.com/https://github.com/ros2/cartographer_ros.git -b ros2
```

安装依赖

这里我们使用rosdepc进行依赖的安装，rosdepc指令找不到可以先运行下面的一键安装命令，选择一键配置rosdep即可。

```shell
wget http://fishros.com/install -O fishros && . fishros
```

接着在fishbot_ws下运行下面这个命令进行依赖的安装。

> rosdepc 是小鱼制作的国内版rosdep，是一个用于安装依赖的工具。该工具的安装可以采用一键安装进行，选项编号为3。安装完成后运行一次rodepc update即可使用。

```shell
rosdepc install -r --from-paths src --ignore-src --rosdistro $ROS_DISTRO -y
```

编译

> 这里有一个新的命令--packages-up-to，意思是其所有依赖后再编译该包

```shell
colcon build --packages-up-to cartographer_ros
```

2.3 测试是否安装成功

如果是源码编译请先source下工作空间后再使用下面指令查看是否安装成功；

```shell
ros2 pkg list | grep cartographer
```

能看到下面的结果即可

```shell
cartographer_ros
cartographer_ros_msgs
```

> 可能你会好奇为什么没有cartographer，因为cartographer包的编译类型原因造成的，不过没关系，cartographer_ros依赖于cartographer，所以有cartographer_ros一定有cartographer。

3.Cartographer参数配置

作为一个优秀的开源库，Cartographer提供了很多可以配置的参数，虽然灵活性提高了，但同时也提高了使用难度（需要对参数进行调节配置），所以有必要在正式使用前对参数进行基本的介绍。

因为我们主要使用其进行2D的建图定位，所以我们只需要关注2D相关的参数。

Cartographer参数是使用lua文件来描述的，不会lua也没关系，我们只是改改参数而已。

> 小鱼提示：lua中的注释采用 -- 开头

3.1 前端参数

文件：trajectory_builder_2d

`src/cartographer/configuration_files/trajectory_builder_2d.lua`

请你打开这个文件自行浏览，小鱼对其中我们可能会在初次建图配置的参数进行介绍。

```lua
    -- 是否使用IMU数据
    use_imu_data = true,
    -- 深度数据最小范围
    min_range = 0.,
```

```
  -- 深度数据最大范围
  max_range = 30.,
  -- 传感器数据超出有效范围最大值时，按此值来处理
  missing_data_ray_length = 5.,
  -- 是否使用实时回环检测来进行前端的扫描匹配
  use_online_correlative_scan_matching = true
  -- 运动过滤，检测运动变化，避免机器人静止时插入数据
  motion_filter.max_angle_radians
```

3.2 后端参数

文件：pose_graph.lua-后端参数配置项

路径 `src/cartographer/configuration_files/pose_graph.lua`

该文件主要和地图构建

```lua
--Fast csm的最低分数，高于此分数才进行优化。
constraint_builder.min_score = 0.65
--全局定位最小分数，低于此分数则认为目前全局定位不准确
constraint_builder.global_localization_min_score = 0.7
```

3.3 Carotgrapher_ROS参数配置

该部分参数主要是用于和ROS2进行通信和数据收发的配置，比如配置从哪个话题读取里程记数据，从哪个话题来获取深度信息（雷达）。

文件：backpack_2d.lua

路径： `src/cartographer_ros/cartographer_ros/configuration_files/backpack_2d.lua`

```lua
include "map_builder.lua"
include "trajectory_builder.lua"

options = {
  map_builder = MAP_BUILDER,
  trajectory_builder = TRAJECTORY_BUILDER,
```

```
-- 用来发布子地图的ROS坐标系ID，位姿的父坐标系，通常是map。
map_frame = "map",
-- SLAM算法跟随的坐标系ID
tracking_frame = "base_link",
-- 将发布map到published_frame之间的tf
published_frame = "base_link",
-- 位于“published_frame ”和“map_frame”之间，用来发布本地SLAM结果（非闭环），通常是“odom”
odom_frame = "odom",
-- 是否提供里程计
provide_odom_frame = true,
-- 只发布二维位姿态（不包含俯仰角）
publish_frame_projected_to_2d = false,
-- 是否使用里程计数据
use_odometry = false,
-- 是否使用GPS定位
use_nav_sat = false,
-- 是否使用路标
use_landmarks = false,
-- 订阅的laser scan topics的个数
num_laser_scans = 0,
-- 订阅多回波技术laser scan topics的个数
num_multi_echo_laser_scans = 1,
-- 分割雷达数据的个数
num_subdivisions_per_laser_scan = 10,
-- 订阅的点云topics的个数
num_point_clouds = 0,
-- 使用tf2查找变换的超时秒数
lookup_transform_timeout_sec = 0.2,
-- 发布submap的周期间隔
submap_publish_period_sec = 0.3,
-- 发布姿态的周期间隔
pose_publish_period_sec = 5e-3,
-- 轨迹发布周期间隔
trajectory_publish_period_sec = 30e-3,
-- 测距仪的采样率
rangefinder_sampling_ratio = 1.,
--里程记数据采样率
odometry_sampling_ratio = 1.,
-- 固定的frame位姿采样率
fixed_frame_pose_sampling_ratio = 1.,
-- IMU数据采样率
imu_sampling_ratio = 1.,
```

```
  -- 路标采样率
  landmarks_sampling_ratio = 1.,
}
```

4.总结

本节我们简单的介绍了Cartographer以及二进制和源码安装的方法，并对参数进行介绍。

下一节我们就开始为fishbot配置cartographer，接着就可以使用fishbot进行建图了。

参考文章：

- https://google-cartographer.readthedocs.io/en/latest/configuration.html

---

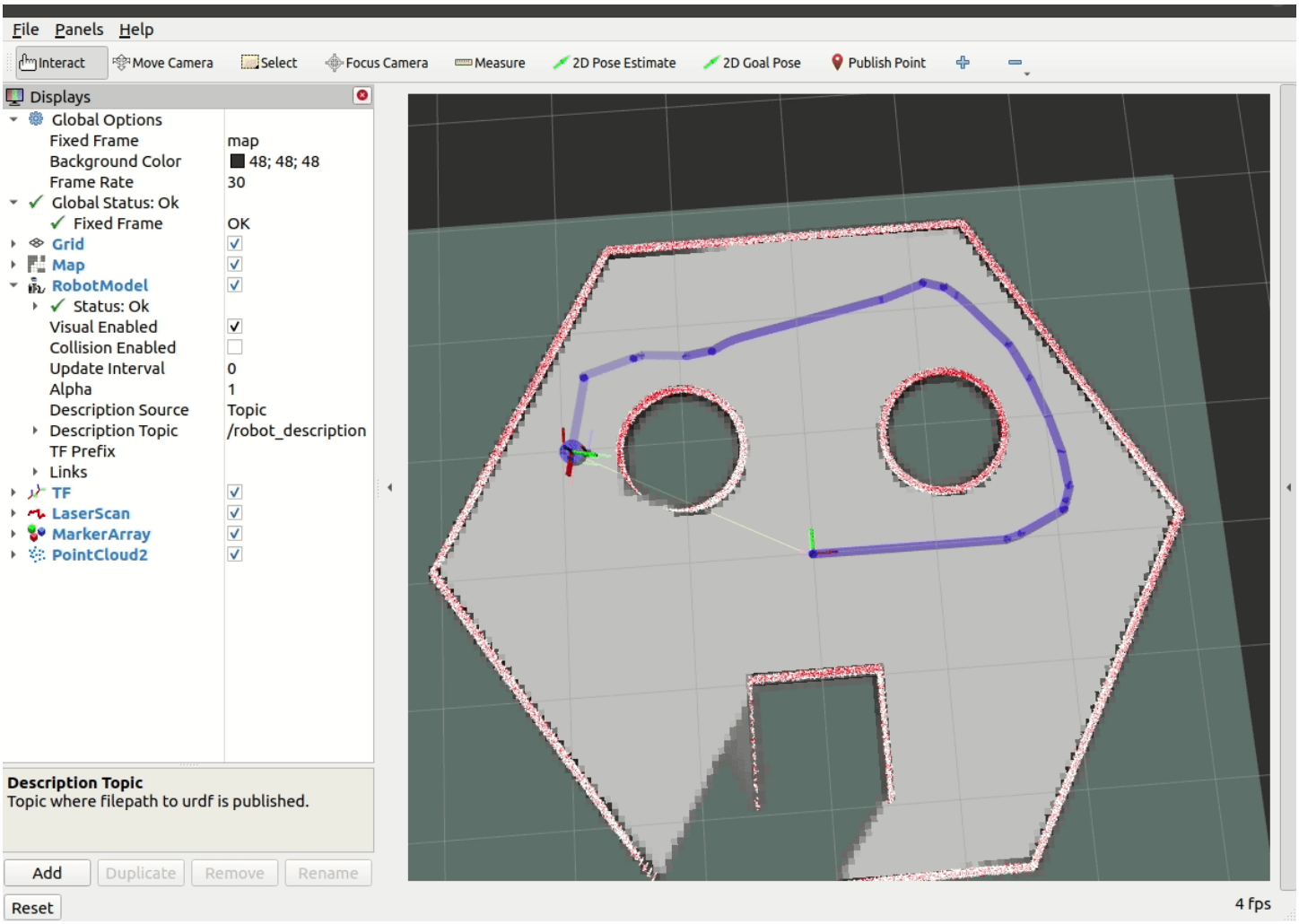
技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

3. 配置Fishbot进行建图

上一节我们安装好了cartographer，这节课我们就开始配置cartographer进行建图。

我们需要创建一个功能包，将参数文件和 `Cartographer` 启动文件放到一起然后启动。



1.创建fishbot_cartographer

在src目录下，使用创建功能包指令，创建功能包

```shell
cd src
ros2 pkg create fishbot_cartographer
```

接着创建配置文件夹、launch文件夹和rviz配置文件夹。

```shell
cd fishbot_cartographer
mkdir config
mkdir launch
mkdir rviz
```

创建完成的功能包结构

```
.
├── CMakeLists.txt
├── config
├── launch
├── src
├── package.xml
└── rviz
```

2.添加cartographer配置文件

在 `fishbot/config` 目录下创建 `fishbot_2d.lua` 文件。

接着我们来写一下配置文件，相较于默认的配置文件，主要修改以下内容（见注释）

```lua
include "map_builder.lua"
include "trajectory_builder.lua"

options = {
  map_builder = MAP_BUILDER,
  trajectory_builder = TRAJECTORY_BUILDER,
  map_frame = "map",
  tracking_frame = "base_link",
  -- base_link改为odom,发布map到odom之间的位姿态
  published_frame = "odom",
  odom_frame = "odom",
  -- true改为false，不用提供里程计数据
  provide_odom_frame = false,
  -- false改为true，仅发布2D位资
  publish_frame_projected_to_2d = true,
```

```
  -- false改为true，使用里程计数据
  use_odometry = true,
  use_nav_sat = false,
  use_landmarks = false,
  -- 0改为1,使用一个雷达
  num_laser_scans = 1,
  -- 1改为0，不使用多波雷达
  num_multi_echo_laser_scans = 0,
  -- 10改为1，1/1=1等于不分割
  num_subdivisions_per_laser_scan = 1,
  num_point_clouds = 0,
  lookup_transform_timeout_sec = 0.2,
  submap_publish_period_sec = 0.3,
  pose_publish_period_sec = 5e-3,
  trajectory_publish_period_sec = 30e-3,
  rangefinder_sampling_ratio = 1.,
  odometry_sampling_ratio = 1.,
  fixed_frame_pose_sampling_ratio = 1.,
  imu_sampling_ratio = 1.,
  landmarks_sampling_ratio = 1.,
}


-- false改为true，启动2D SLAM
MAP_BUILDER.use_trajectory_builder_2d = true

-- 0改成0.10,比机器人半径小的都忽略
TRAJECTORY_BUILDER_2D.min_range = 0.10
-- 30改成3.5,限制在雷达最大扫描范围内，越小一般越精确些
TRAJECTORY_BUILDER_2D.max_range = 3.5
-- 5改成3,传感器数据超出有效范围最大值
TRAJECTORY_BUILDER_2D.missing_data_ray_length = 3.
-- true改成false,不使用IMU数据，大家可以开启，然后对比下效果
TRAJECTORY_BUILDER_2D.use_imu_data = false
-- false改成true,使用实时回环检测来进行前端的扫描匹配
TRAJECTORY_BUILDER_2D.use_online_correlative_scan_matching = true
-- 1.0改成0.1,提高对运动的敏感度
TRAJECTORY_BUILDER_2D.motion_filter.max_angle_radians = math.rad(0.1)

-- 0.55改成0.65,Fast csm的最低分数，高于此分数才进行优化。
POSE_GRAPH.constraint_builder.min_score = 0.65
--0.6改成0.7,全局定位最小分数，低于此分数则认为目前全局定位不准确
```

```
POSE_GRAPH.constraint_builder.global_localization_min_score = 0.7

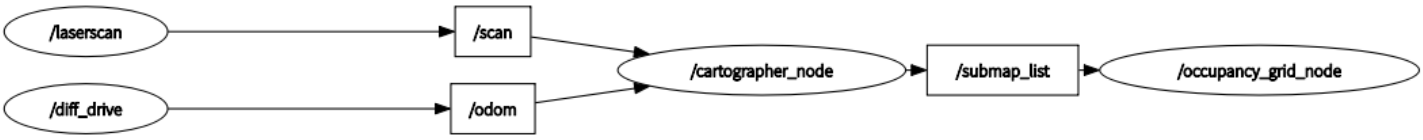
-- 设置0可关闭全局SLAM
-- POSE_GRAPH.optimize_every_n_nodes = 0

return options
```

3.添加launch文件

3.1 launch需要包含的节点

要完成使用Cartographer进行建图，需要两个节点的参与，整个过程的计算流图如下：



/cartographer_node节点:

该节点从/scan和/odom话题接收数据进行计算，输出/submap_list数据.

该节点需要接收一个参数配置文件（第二部分写的那个）参数。

/occupancy_grid_node节点：

该节点接收/submap_list子图列表，然后将其拼接成map并发布

该节点需要配置地图分辨率和更新周期两个参数。

3.2 编写launch文件

在路径 `src/fishbot_cartographer/launch/` 下新建 `cartographer.launch.py` 文件，接着我们将上面两个节点加入到这个launch文件中。

我们在第二部分写的配置文件就是给cartographer_node节点的，可以通过这个节点启动参数 `configuration_directory` 和 `configuration_basename` 进行传递。

```
import os
from launch import LaunchDescription
from launch.substitutions import LaunchConfiguration
from launch_ros.actions import Node
from launch_ros.substitutions import FindPackageShare


def generate_launch_description():
    # 定位到功能包的地址
    pkg_share =
FindPackageShare(package='fishbot_cartographer').find('fishbot_cartographer')

    #======================运行节点需要的配置
==========================================================================
    # 是否使用仿真时间，我们用gazebo，这里设置成true
    use_sim_time = LaunchConfiguration('use_sim_time', default='true')
    # 地图的分辨率
    resolution = LaunchConfiguration('resolution', default='0.05')
    # 地图的发布周期
    publish_period_sec = LaunchConfiguration('publish_period_sec', default='1.0')
    # 配置文件夹路径
    configuration_directory =
LaunchConfiguration('configuration_directory',default= os.path.join(pkg_share,
'config') )
    # 配置文件
    configuration_basename = LaunchConfiguration('configuration_basename',
default='fishbot_2d.lua')
    rviz_config_dir = os.path.join(pkg_share, 'config')+"/cartographer.rviz"
    print(f"rviz config in {rviz_config_dir}")


    #======================声明三个节点，
cartographer/occupancy_grid_node/rviz_node=================================
    cartographer_node = Node(
        package='cartographer_ros',
        executable='cartographer_node',
        name='cartographer_node',
        output='screen',
        parameters=[{'use_sim_time': use_sim_time}],
        arguments=['-configuration_directory', configuration_directory,
```

```
                    '-configuration_basename', configuration_basename])

    cartographer_occupancy_grid_node = Node(
        package='cartographer_ros',
        executable='cartographer_occupancy_grid_node',
        name='cartographer_occupancy_grid_node',
        output='screen',
        parameters=[{'use_sim_time': use_sim_time}],
        arguments=['-resolution', resolution, '-publish_period_sec',
publish_period_sec])

    rviz_node = Node(
        package='rviz2',
        executable='rviz2',
        name='rviz2',
        arguments=['-d', rviz_config_dir],
        parameters=[{'use_sim_time': use_sim_time}],
        output='screen')

    #================================================定义启动文件
============================================================
    ld = LaunchDescription()
    ld.add_action(cartographer_node)
    ld.add_action(cartographer_occupancy_grid_node)
    ld.add_action(rviz_node)

    return ld
```

4.添加安装指令

做完上面的操作，我们还需要添加安装指令。

打开CmakeLists.txt，添加下面一条指令，将三个目录安装到install目录。

```
install(
  DIRECTORY config launch rviz
  DESTINATION share/${PROJECT_NAME}
)
```

5.开始建图

5.1编译启动

```shell
colcon build --packages-select fishbot_cartographer
```

启动建图前，需要先启动gazebo仿真环境，因为我们的建图程序依赖于Gazebo提供雷达和里程计等数据。

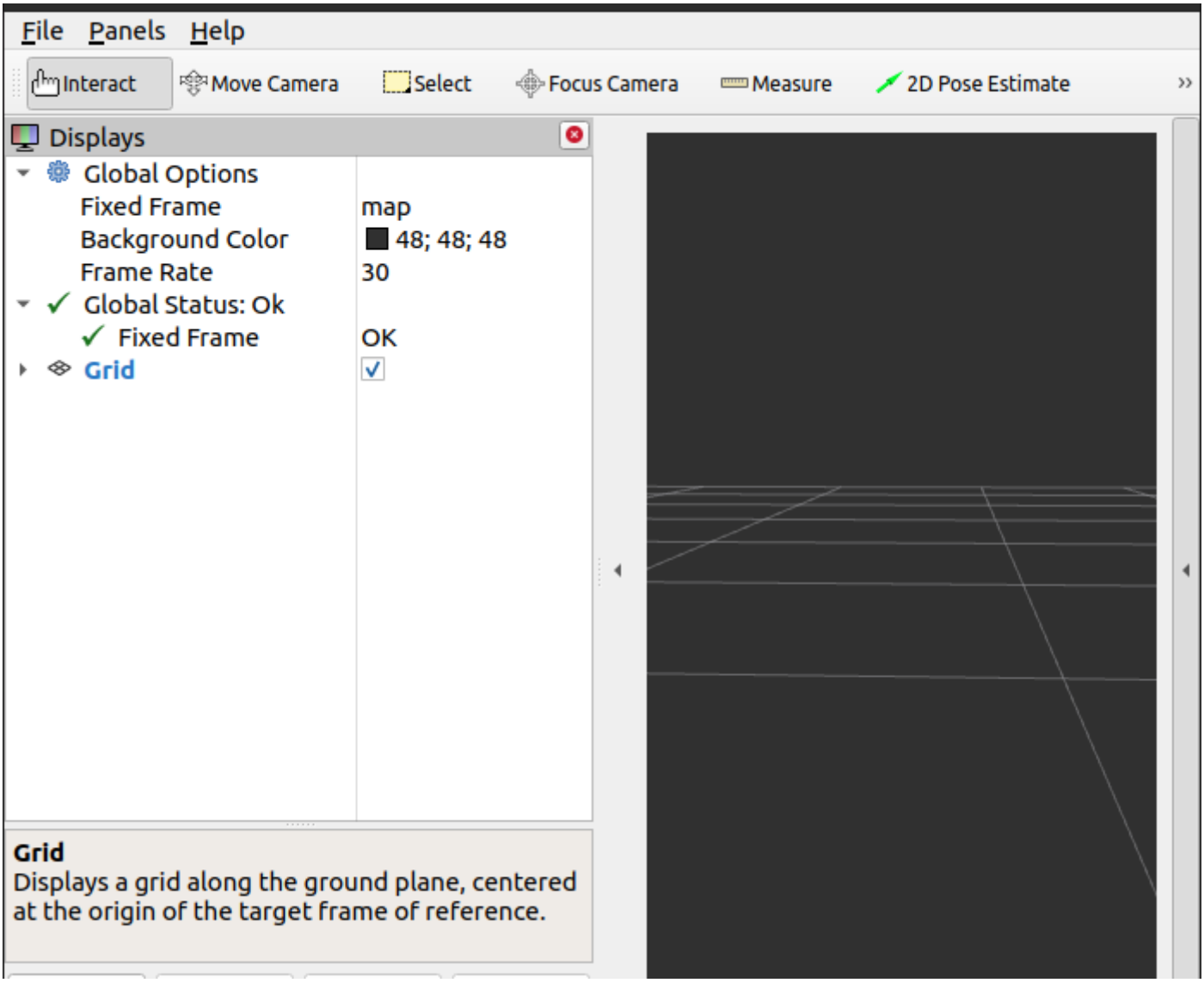
```shell
source install/setup.bash
ros2 launch fishbot_description gazebo.launch.py
```

source,启动建图

```shell
source install/setup.bash
ros2 launch fishbot_cartographer cartographer.launch.py
```

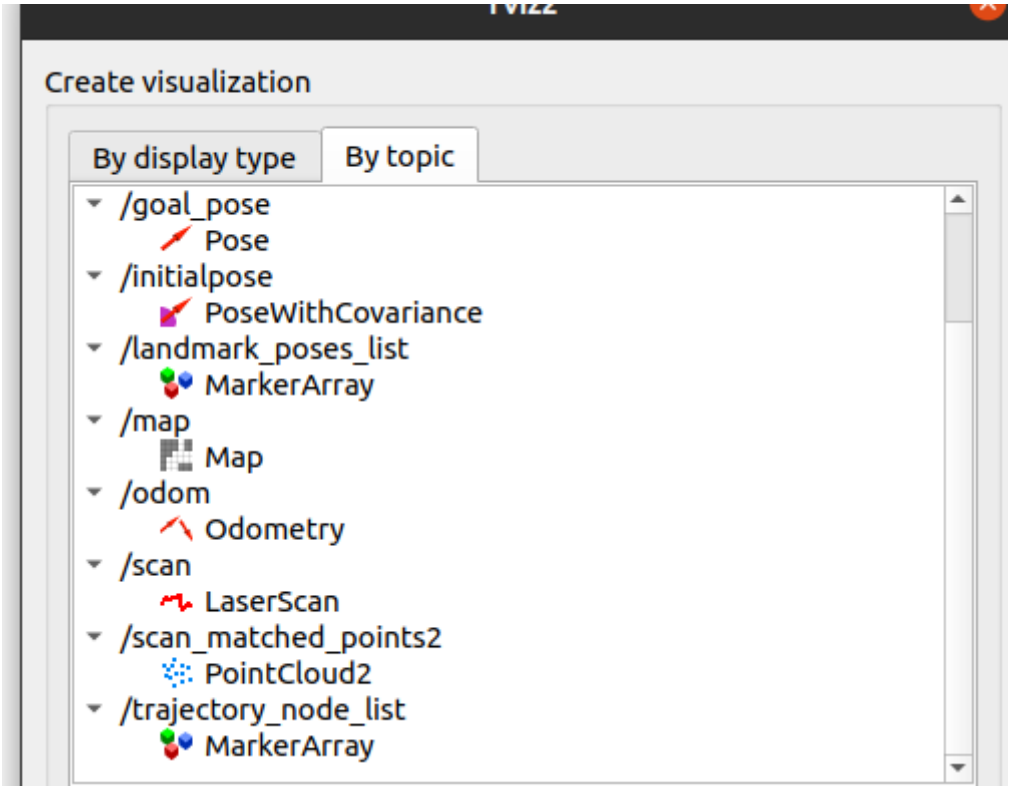
5.2修改配置

如果一切正常，你应该看到的是一个空空如也的RVIZ界面

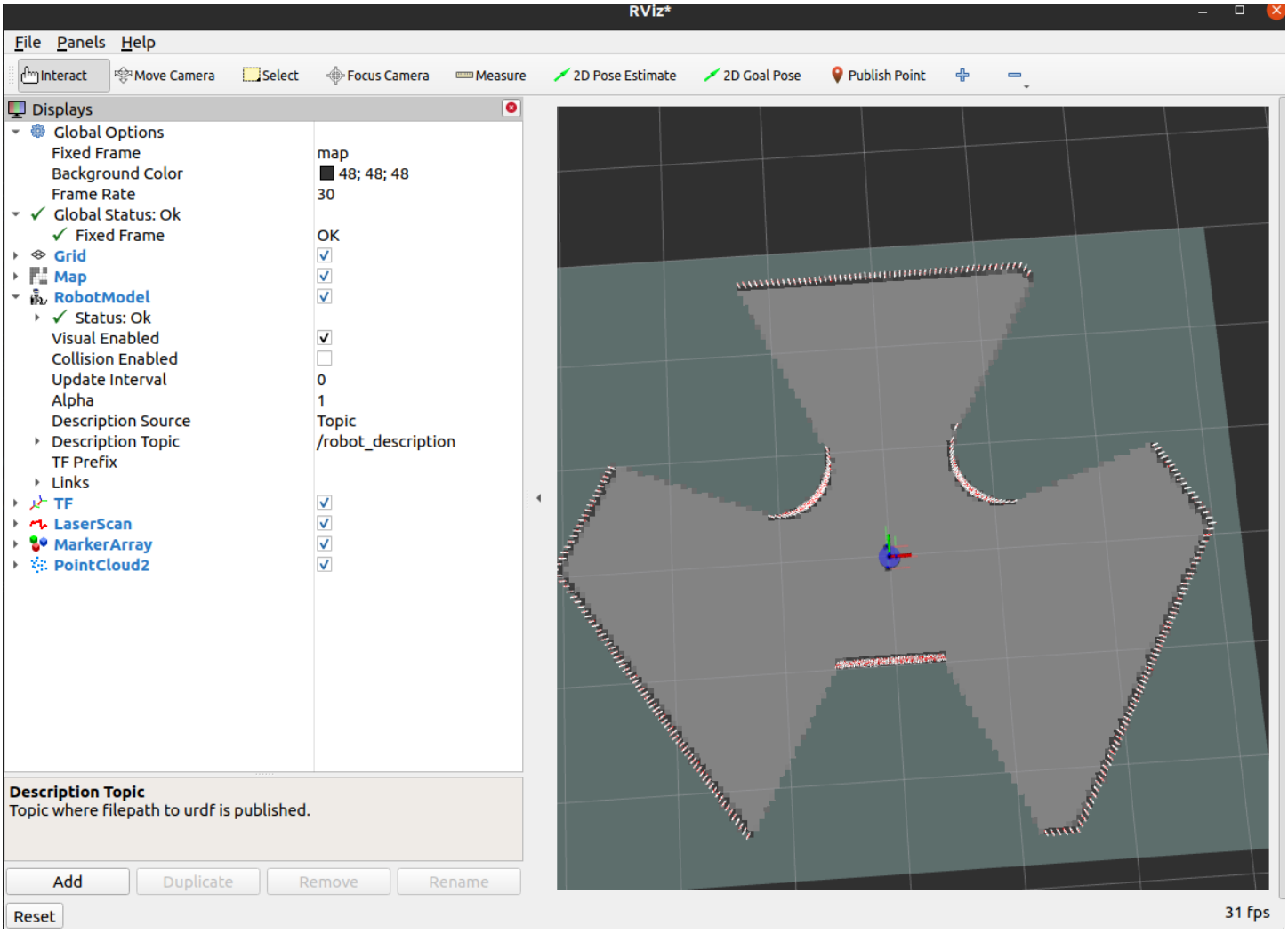


不用担心，此时地图其实已经有了，我们需要添加一下地图相关的插件即可。

通过 `Add` -> `By Topic` 添加组件。



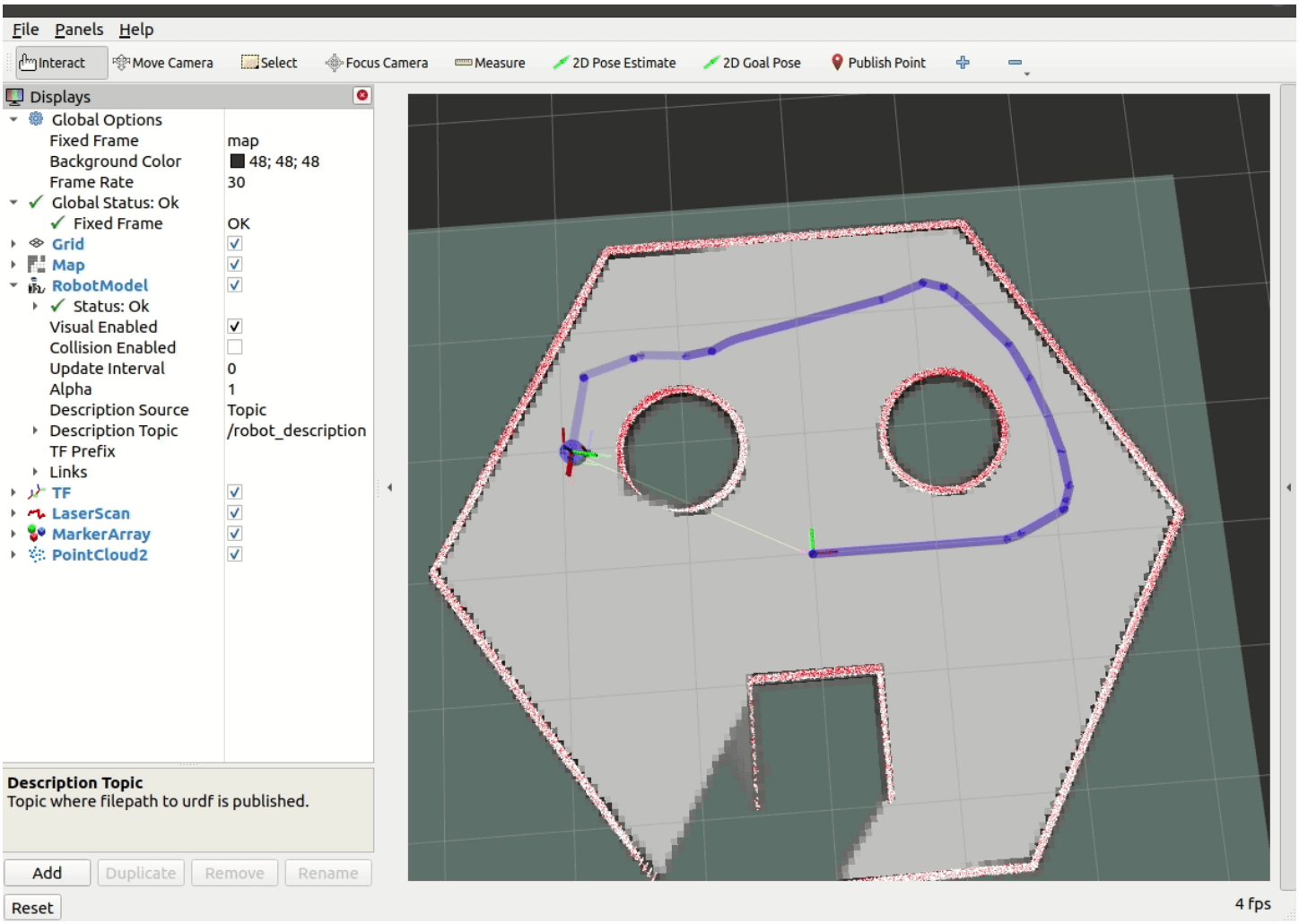
最后通过左边的插件你应该可以看到图和机器人了。



5.3开始建图

打开我们机器人遥控节点，降低速度，控制机器人走一圈，看看地图的变化。

```
ros2 run teleop_twist_keyboard teleop_twist_keyboard
```



6.保存地图

走完一圈，没有黑影部分，我们就可以保存地图为一个本地文件了。我们需要使用一个叫做 `nav2_map_server` 的工具。

6.1 安装nav2_map_server

```
sudo apt install ros-humble-nav2-map-server
```

6.2 保存地图

```
ros2 run nav2_map_server map_saver_cli --help
```

可以看到有下面的用法

```shell
Usage:
  map_saver_cli [arguments] [--ros-args ROS remapping args]

Arguments:
  -h/--help
  -t <map_topic>
  -f <mapname>
  --occ <threshold_occupied>
  --free <threshold_free>
  --fmt <image_format>
  --mode trinary(default)/scale/raw

NOTE: --ros-args should be passed at the end of command line
```

我们的地图话题为map，文件名字我们用fishbot_map,所以有下面这个这样写的命令行。

```shell
# 先将地图保存到src/fishbot_cartographer/map目录下
cd src/fishbot_cartographer/ && mkdir map && cd map
ros2 run nav2_map_server map_saver_cli -t map -f fishbot_map
```

接着我们就可以得到下面的两个文件

```shell
.
├── fishbot_map.pgm
└── fishbot_map.yaml

0 directories, 2 files
```

这两个文件就是对当前地图保存下来的文件，其中.pgm是地图的数据文件，.yaml后缀的是地图的描述文件。

下面的导航过程中我们将要使用到地图文件进行路径的搜索和规划。

7.总结

本节小鱼带你一起完成了Fishbot的Cartographer配置和建图，下一节小鱼会对地图是什么以及如何使用进行介绍，接着我们就可以配置Nav2进行Fishbot的导航了。

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

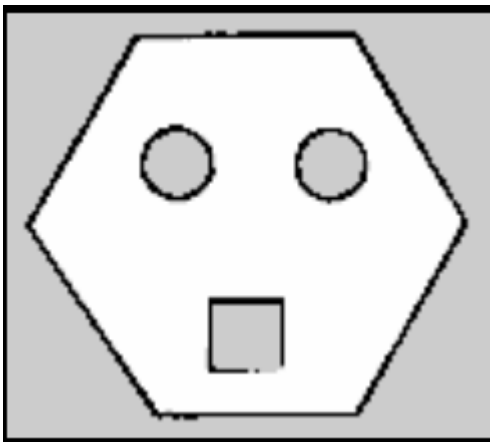
1.地图文件介绍

1.地图数据文件.pgm介绍

OccupancyGrid由一个 `.yaml格式` 的元数据文件，和.pgm图片格式的地图数据文件组成。从上节课建图后保存的文件也可以看出。

```
.
├── fishbot_map.pgm
└── fishbot_map.yaml

0 directories, 2 files
```

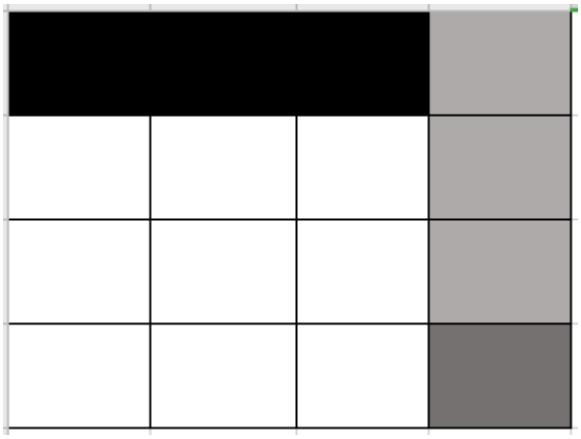


打开上节课建好的地图，观察下，你应该会有几点疑问：

1. 不是说占据栅格地图每个栅格都有一个概率吗？为什么看不出来？
2. 立方体内和圆柱体内和墙之外的区域为什么是灰色的？

原因如下：

1. 一个栅格对应到图片上其实是一个像素，每一个像素的值在0-255之间，所以将像素值和占据率之间的映射即可，而像素值反应到图像上就是颜色的深浅，1.2图对应的像素颜色如下：



2. 建图的时候物体内和墙之外的区域机器人并没有探索到，没有数据参考就认为其值是未知的。

2.地图描述文件.yaml介绍

除了 `fishbot_map.pgm` 文件外，还有另外一个 `fishbot_map.yaml` 的文件， `fishbot_map.yaml` 文件是地图的配置文件，该文件内容如下：

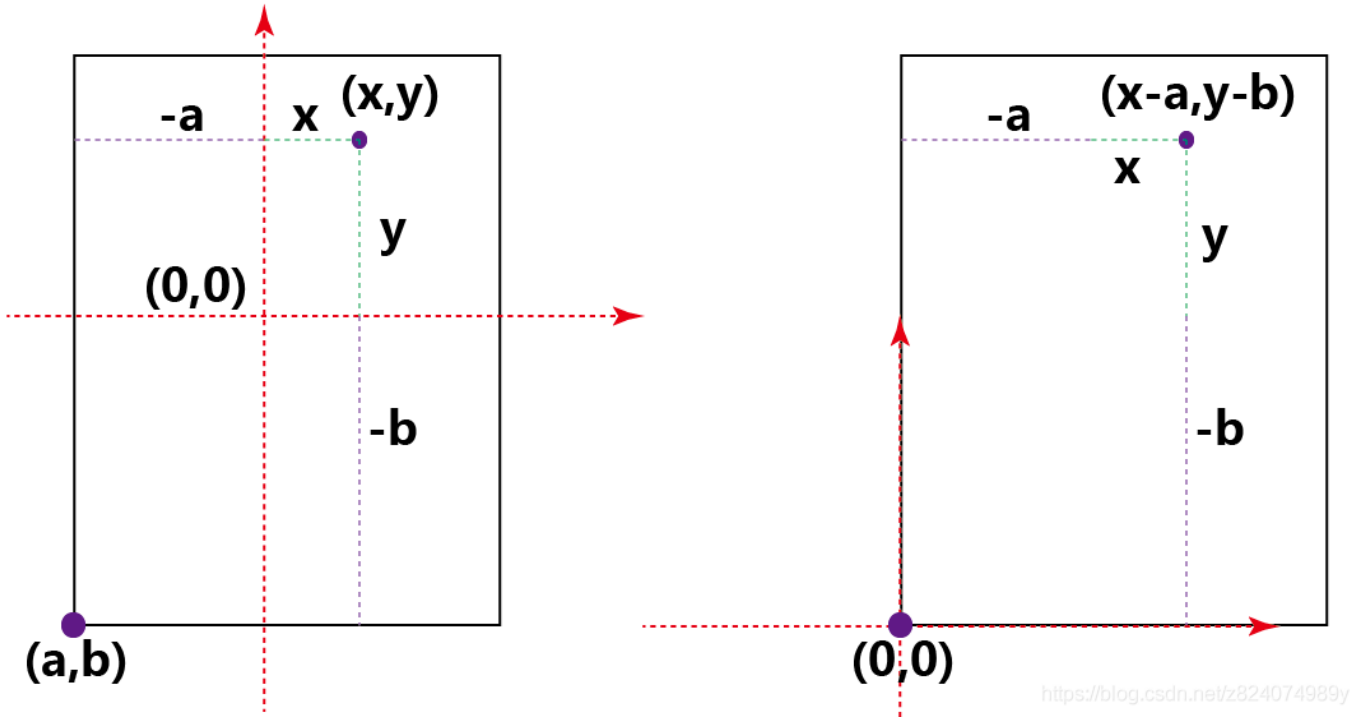
```yaml
image: fishbot_map.pgm
mode: trinary
resolution: 0.05
origin: [-3.37, -2.88, 0]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.25
```

- image：图像名称
- mode：图像模式，默认为trinary(三进制)，还有另外两个可选项scale(缩放的)和raw(原本的值)。
- resolution：分辨率，一个栅格对应的物理尺寸，单位为m。0.05则表示一个栅格为0.05m
- origin：地图原点位置，单位是m。
- negate：是否反转图像
- cooupied_thresh：占据阈值
- free_thresh：空闲阈值

如何在地图上找出机器人原点的像素位置？

1. 图像的像素原点在左下角

2. 借张图左边的0,0是地图原点，右边的图是图像的像素原点



3. 假如机器人地图原点是[-3.37m, -2.88m]，除上分辨率则可得到像素原点[-67.4px,-57.6px]。`地图原点`在x和y轴分别偏移[-67.4px,-57.6px]就到了像素原点，反过来说，地图原点就在像素原点的[67.4px,57.6px]处。

3.地图加载

3.1 启动地图

了解完地图的格式和描述文件，接着我们看看如何将地图加载到rviz2中进行显示。我们使用nav2_map_server进行地图的保存，也可以使用它来加载地图。

打开终端，进入`src/map/`运行下面指令：

```shell
ros2 run nav2_map_server map_server --ros-args --param
yaml_filename:=map/fishbot_map.yaml
```

其中`--ros-args --param yaml_filename:=map/fishbot_map.yaml`指定地图描述文件名称`.fishbot_map.yaml`。

3.2 打开rviz2

打开终端

```
rviz2
```

通过add->bytopic->选择map组件。此时你应该什么都看不到，没关系，这是正常的。

3.3 配置并激活map_server

`map_server` 节点采用的是ROS2的生命周期进行编写的，除了启动节点，我们还需要手动的配置和激活节点才能使用。

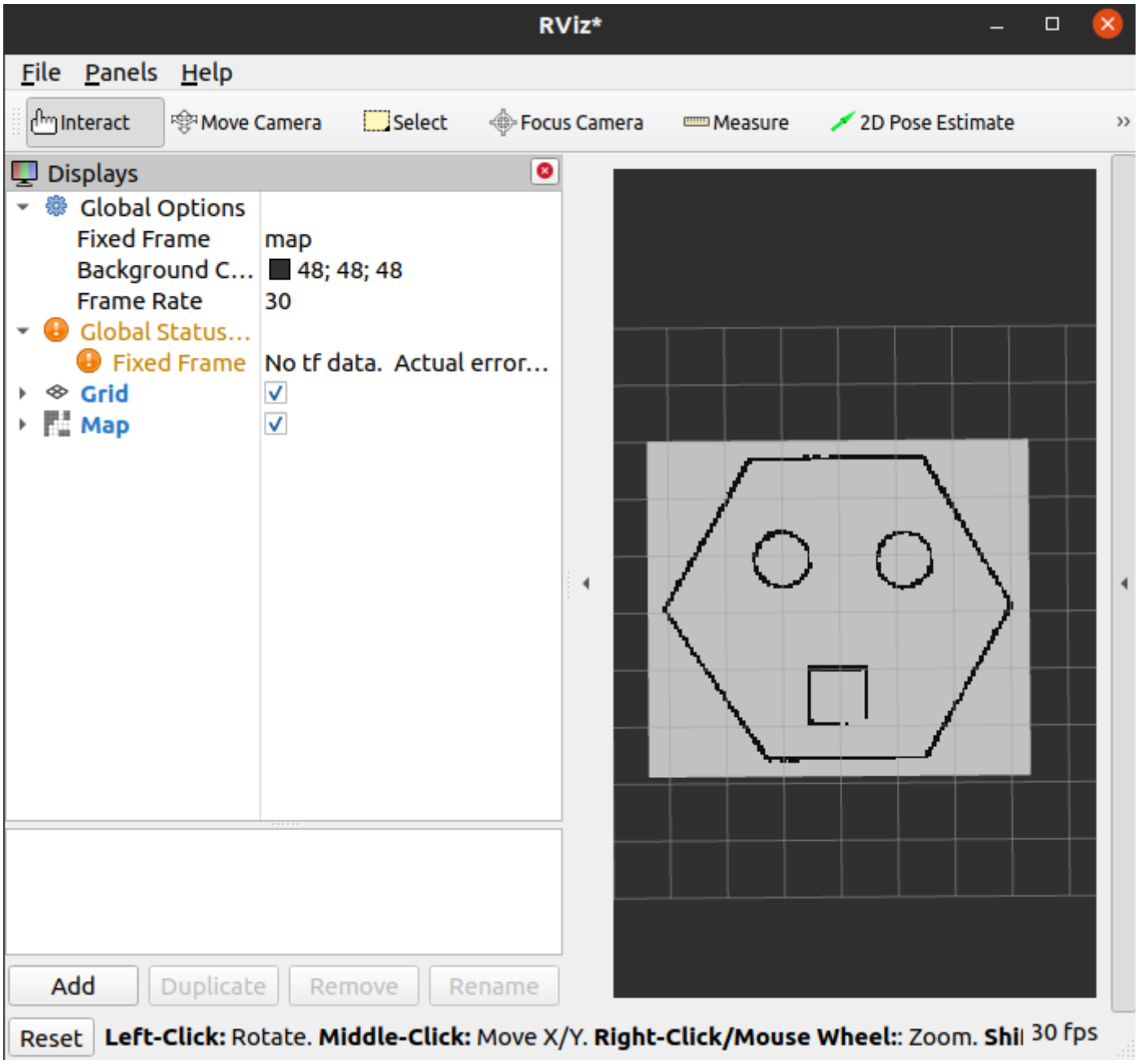
打开新的终端，使用下面指令进行配置：

```
ros2 lifecycle set /map_server configure
```

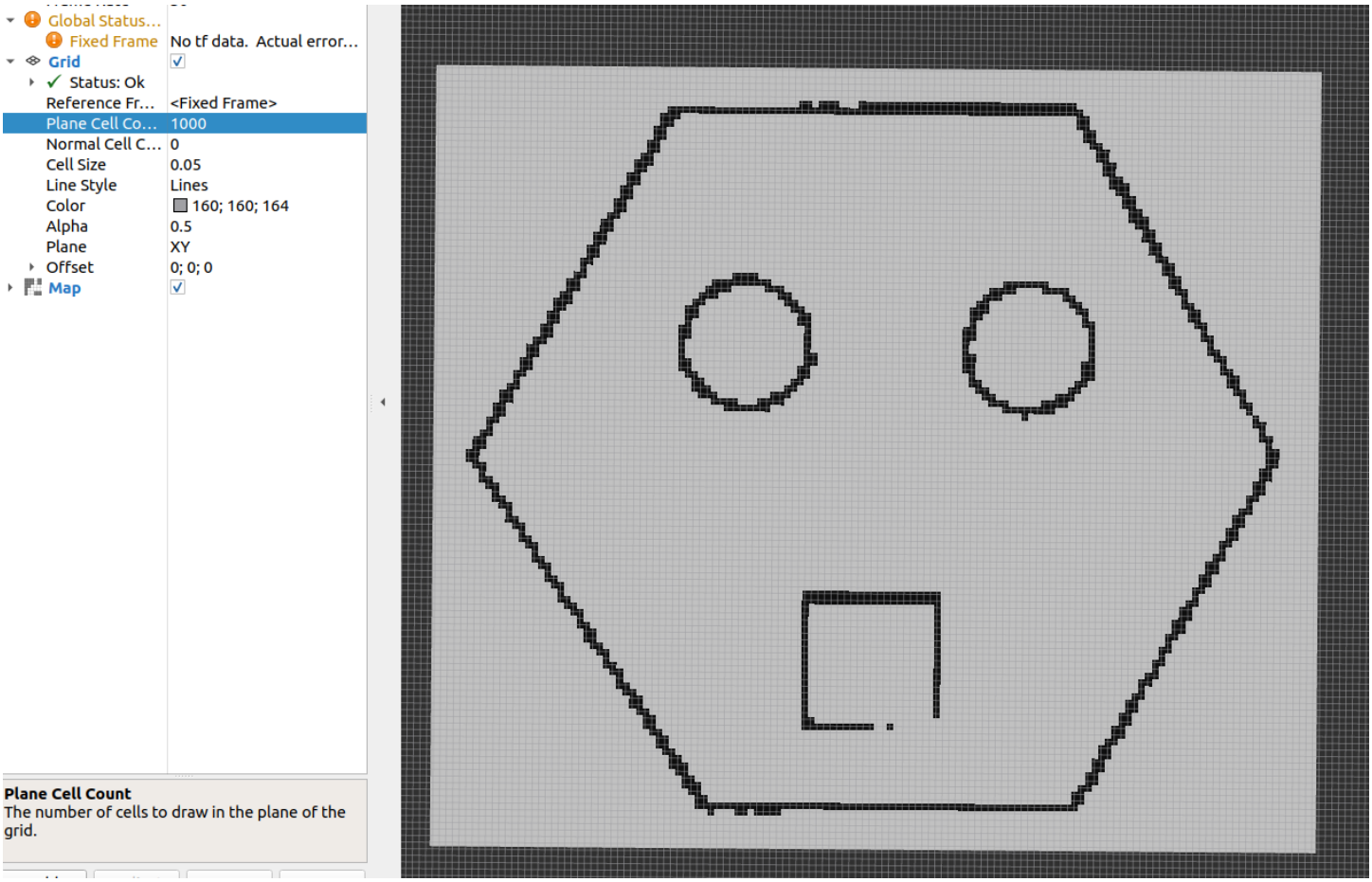
使用下面的指令进行激活：

```
ros2 lifecycle set /map_server activate
```

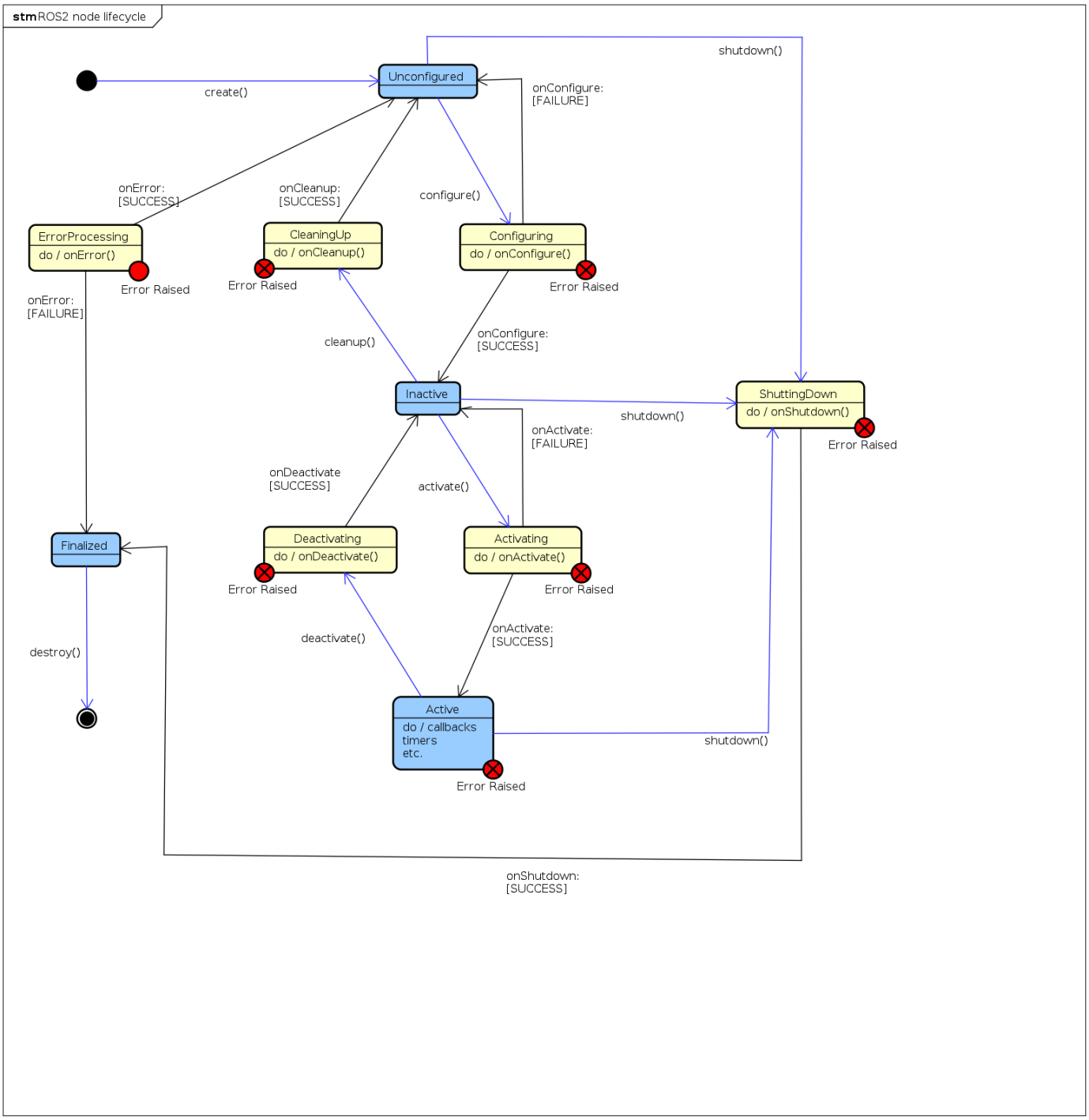
完成这一步，你应该就可以在 `rviz2` 中看到地图了。



将栅格数量改大，然后将Grid的cellsize改成0.05,你就可以看到我们之间建立的地图每一个栅格的占据情况了。



生命周期节点转换图如下：



powered by Astah

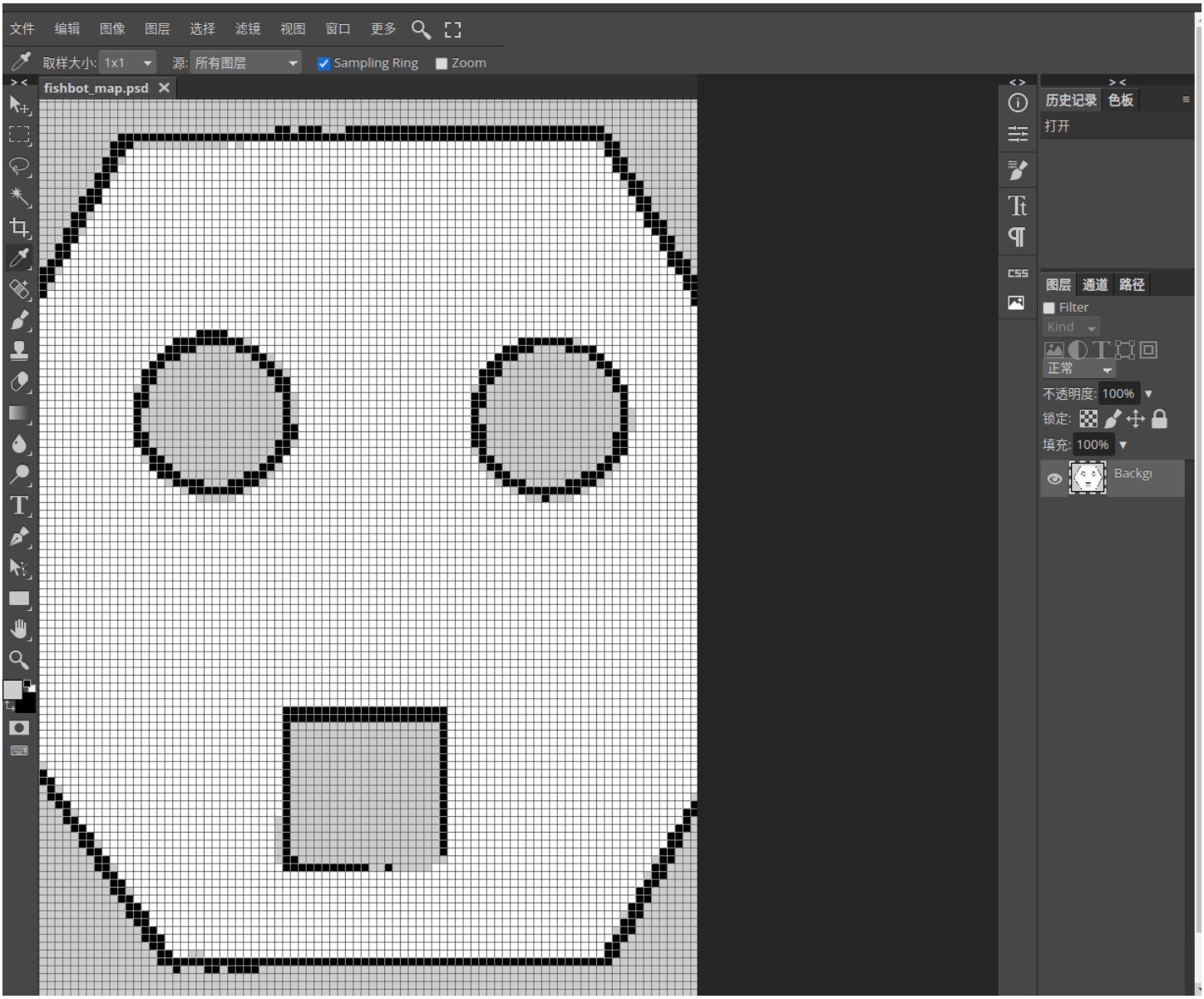
4.地图编辑

最后我们来说一下如何对地图进行编辑，地图编辑的方法有很多，你可以手动改图片，也可以通过opencv等图像处理库进行图像的去除噪点等操作。

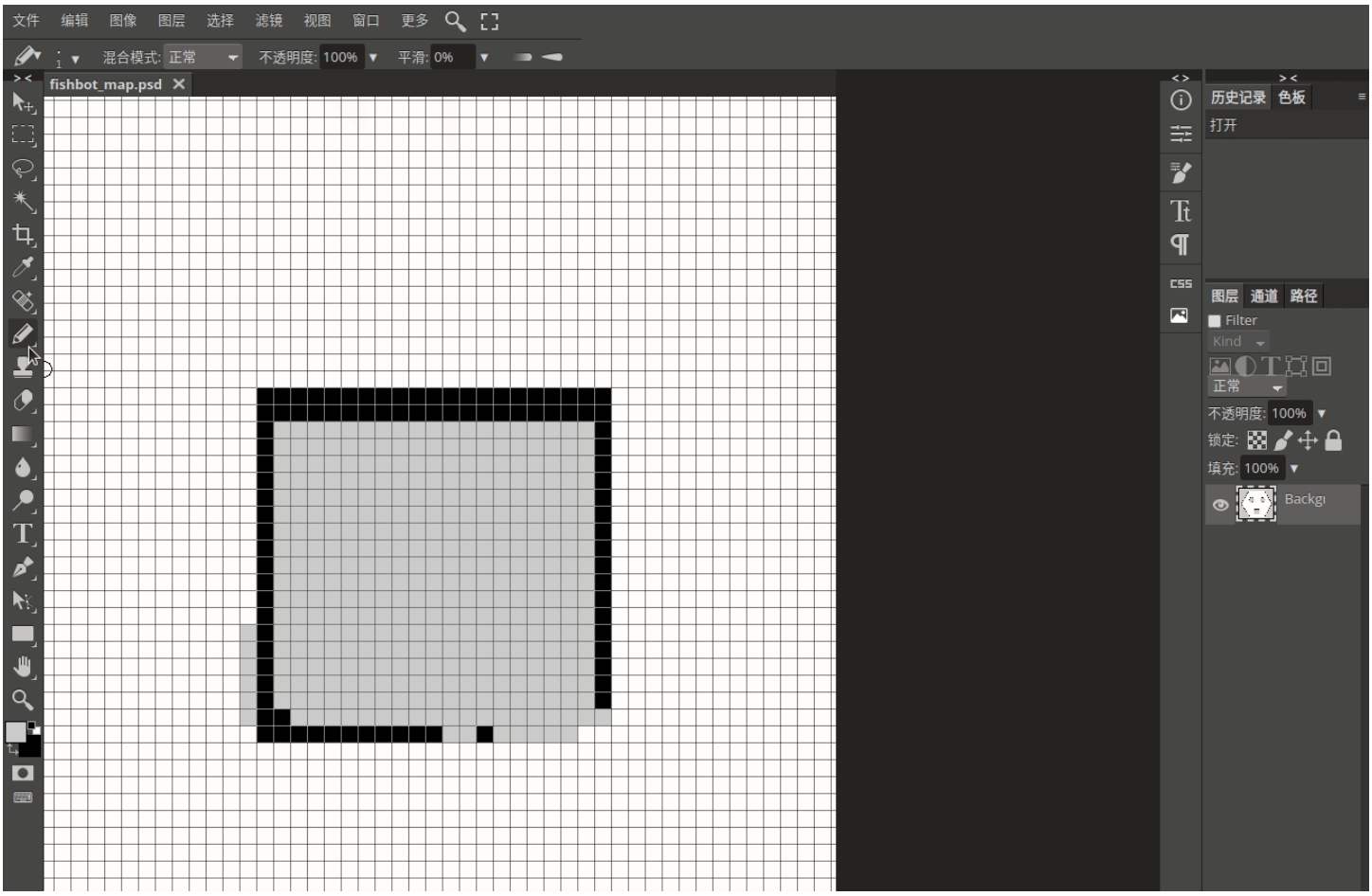
这里小鱼介绍的是手动添加一个地图用PS进行编辑 。

打开网址：https://www.gaituya.com/ps/

接着点击文件打开选择我们的.pgm文件，即可看到下图。



接着小鱼将下面的正方形给补上。



保存后在文件选择导出为 `png` 格式即可，因为PS并不支持 `pgm` 格式的导出，所以我们选择 `png` 格式，幸运的是 `png` 格式的地图也是被map_server所支持的。

把导出的图片放到map文件夹下，接着我们需要修改下yaml配置文件中图片的后缀。

```
image: fishbot_map.png
mode: trinary
resolution: 0.05
origin: [-3.37, -2.88, 0]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.25
```

接着你可以重新运行下map_server（记得先关闭rviz2），然后看一下map是否发生了改变。

5. 总结

本节我们主要学习了地图的相关概念以及加载和编辑的方式，下一节我们就开始正式学习Nav2导航框架。

参考文章：

- https://zhuanlan.zhihu.com/p/21738718

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

1.Nav2导航框架介绍与安装

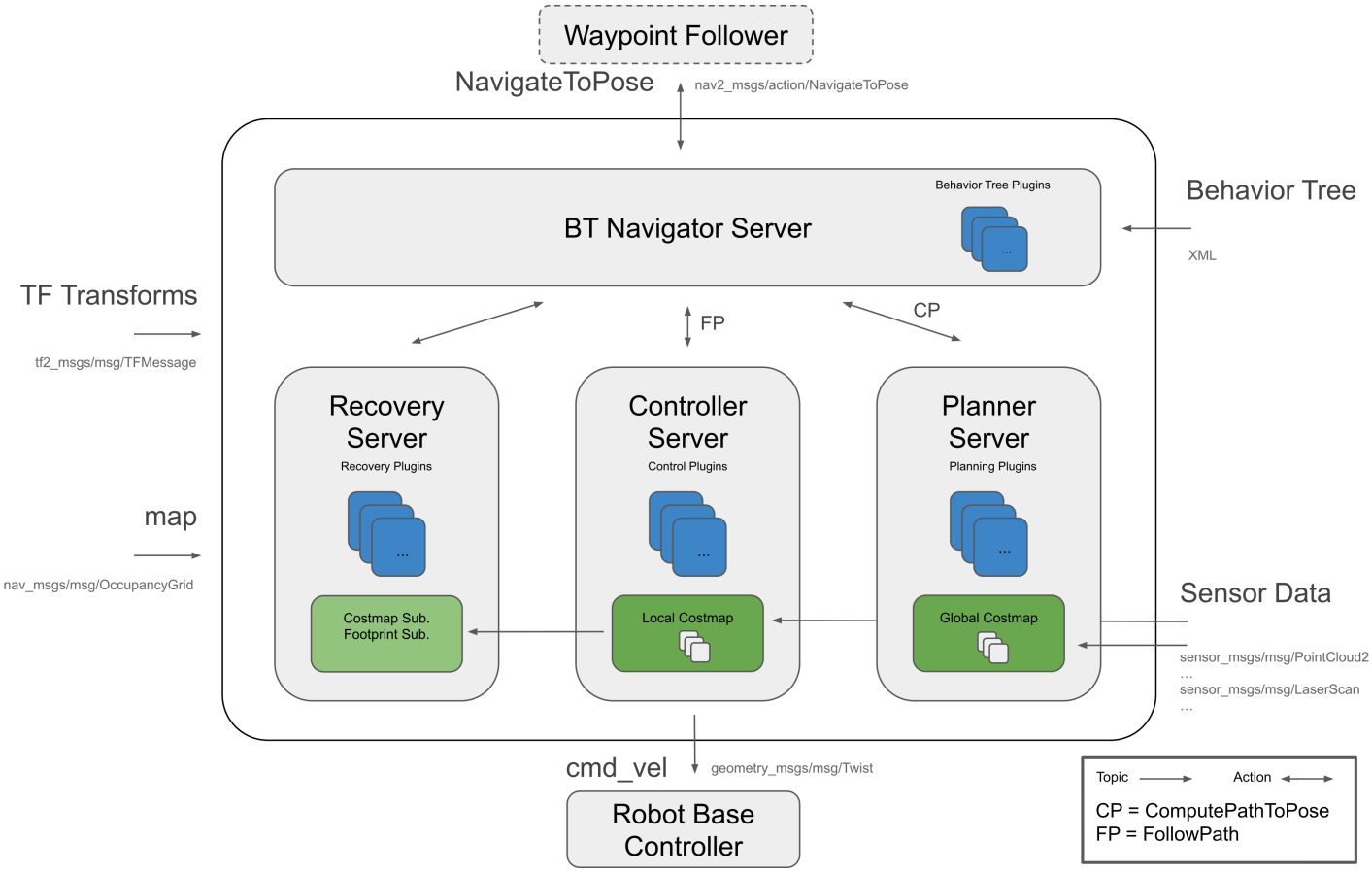
上一节我们对机器人导航过程中所用到的2D地图进行了介绍，本节我们就来正式的学习下Nav2导航框架，关于Nav2相关的更多资料，请访问[Nav2中文网](#)。

1.Nav2是什么

Nav2项目继承并发扬ROS导航栈的精神。该项目力求以安全的方式让移动机器人从A点移动到B点。Nav2也可以应用于其他应用，包括机器人导航，如下动态点跟踪，在这个过程中需要完成动态路径规划、计算电机的速度、避免障碍、恢复行为。

2.Nav2如何做到的

Nav2使用行为树调用模块化服务器来完成一个动作。动作可以是计算路径、控制力、恢复或任何其他与导航相关的操作。这些都是通过ROS Action服务器与行为树 (BT) 通信的独立节点。下图可以让你对Nav2的架构有一个很好的初步了解。



2.2 四个服务

小鱼再对上面的架构图进一步的进行解释，可以简单分为一大三小四个服务。

一大：

1. BT Navigator Server 导航行为树服务，通过这个大的服务来进行下面三个小服务组织和调用。

三小：

1. Planner Server，规划服务器，其任务是计算完成一些目标函数的路径。根据所选的命名法和算法，该路径也可以称为路线。说白了就是在地图上找路。
2. Controller Server，控制服务器，在ROS 1中也被称为局部规划器，是我们跟随全局计算路径或完成局部任务的方法。说白了就是根据找的路控制机器人走路。
3. Recovery Server，恢复服务器，恢复器是容错系统的支柱。恢复器的目标是处理系统的未知状况或故障状况并自主处理这些状况。说白了就是机器人可能出现意外的时候想办法让其正常，比如走着走着掉坑如何爬出来。

通过 `规划路径` 、 `控制机器人沿着路径运动` 、 `遇到问题自主恢复` 三者进行不断切换完成机器人的自主导航（在这个过程还需要很多节点和数据的辅助，详建下面的组建部分）。

2.3 两大代价（成本）地图

在机器人导航的时候，仅仅靠一张SLAM建立的原始地图是不够的，机器人在运动过程中可能会出现新的障碍物，也有可能发现原始地图中某一块的障碍物消失了，所以在机器人导航过程中维护的地图是一个动态的地图，根据更新频率方式和用途不同，可以分为下面两种。

> **注意：不论是全局代价地图还是局部代价地图都是有很多个图层共同叠加而成的，你也可以自己定义图层。**

2.3.1 全局代价地图 （Global Costmap）

全局代价地图主要用于全局的路径规划器。从上面结构图中其在可以看到在Planner Server中。

通常包含的图层有：

- Static Map Layer：静态地图层，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），以避免机器人的外壳会撞上障碍物。

2.3.2 局部代价地图（Local Costmap）

局部代价地图主要用于局部的路径规划器。从上面结构图中其在可以看到在Controller Server中。

通常包含的图层有：

- 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，在障碍地图层上进行膨胀（向外扩张），以避免机器人的外壳会撞上障碍物。

3.Nav2导航相关概念

该部分摘抄自Nav2中文网，中间删除了一些信息，全部文章请访问**原文页面**。

3.1 动作服务器（Action Server）

动作服务器在**第五章中**我们有简单介绍过，Nav2框架中大量的才用了这种通信方式，并将起分装成了行为树（关于行为树下面会介绍）的基础节点进行调用。

通过动作服务器通信，来计算路径规划、控制机器人运动和恢复。每个动作服务器都有自己独特的 `nav2_msgs` 格式的 `.action` 类型，用于与服务器进行交互。

3.2 生命周期节点和绑定

生命周期 (或被管理的，更正确的) 节点是ROS 2独有的。更多信息可以是 **在这里** 。它们是包含状态机转换的用于加载和卸载ROS 2服务器的节点。这有助于确定ROS系统启动和关闭的状态是否正常。

生命周期节点框架在整个项目中被广泛使用，所有服务器都使用它。如果可能的话，所有ROS系统最好使用生命周期节点。

上一节我们通过命令行来配置和激活map_server节点就是因为map_server节点采用了生命周期节点进行管理。

3.3 行为树

行为树 (BT) 在复杂的机器人任务中变得越来越普遍。它们是待完成任务的树形结构。行为树为定义多步或多状态应用程序创建了一个更具可扩展性和人类可理解性的框架。这与有限状态机 (FSM) 相反，后者可能有几十个状态和数百个状态过渡。

> **一个例子就是踢足球机器人。将足球比赛的逻辑嵌入FSM将具有挑战性，且容易出错因为有许多可能的状态和规则。此外，像从左侧、右侧或中间射门这样的建模选择尤其不清楚。使用行为树则可以为许多行为**

> 创建和重用基本原语，像 "kick" "walk" "go to ball" 。更多信息可以在 这本书 找到。强烈建议阅读第1-3章，以更好地理解术语和工作流程。大约需要30分钟。

Nav2项目使用 BehaviorTree CPP V3 作为行为树库。在 `BT Navigator` 中，创建了可以构建为行为树的节点插件。将节点插件加载到BT中，并且在解析该行为树的XML文件时，将关联注册的名称。此时，我们可以通过该行为树进行导航。

> 使用此库的一个原因是它能够加载子树。这意味着可以将Nav2行为树可以进行套娃操作。举个例子是在足球比赛中，使用Nav2行为树作为 "go to ball" 节点，将足球检测作为更大任务的一部分。此外，为BT提供了一个 `NavigateToPoseAction` 插件，因此可以从客户端应用程序通过通常的动作接口调用Nav2软件导航栈。

3.4 导航服务器

规划器和控制器是导航任务的核心。恢复器用于使机器人摆脱不良状态或尝试处理各种形式的问题，以使系统具有容错能力。在本节中，将分析有关它们的一般概念及其在Nav2项目中的用途。

3.4.1 规划器，控制器和恢复服务器

该项目中的三个Action Server是规划器、恢复器和控制器服务器。这些Action Server用于托管一个地图算法插件，以完成各种任务。它们还托管由该算法插件使用的环境表达，以计算其输出。

规划器（Planners）

规划器的任务是计算完成一些目标函数的路径。根据所选的命名法和算法，该路径也可以称为路线。两个典型示例是计算一个到达目标位姿的规划（例如从当前位置到达一个目标位姿）或者完全覆盖（例如覆盖所有空闲空间的规划）。规划器可以访问全局环境表达和缓存在其中的传感器数据。规划器可以被编写为具有以下功能的工具：

- 计算最短路径
- 计算完整覆盖路径
- 沿稀疏或预定义路线计算路径

Nav2中规划器的一般任务是计算从当前位置到达目标位姿的一个有效且可能是最佳的路径。

控制器（Controllers）

控制器，在ROS 1中也被称为局部规划器，是我们跟随全局计算路径或完成局部任务的方法。控制器有权访问局部环境表达，以尝试计算要跟随的基准路径的可行控制工作。许多控制器会将机器人向前投射到空间中，并在每次更新迭代时计算局部可行路径。控制器可以被编写为具有以下功能的工具：

- 跟随路径
- 使用里程计坐标系中的检测器与充电站（桩）对接
- 登上电梯
- 与某个工具的接口

在Nav2中，控制器的一般任务是计算一个有效的控制工作以跟随全局规划路径。然而，有多个控制器类和局部规划器类。Nav2项目的目标就是所有控制器算法都可以作为此服务器中的插件，以用于一般研究和产业任务中。

恢复器（Recovery）

恢复器是容错系统的支柱。恢复器的目标是处理系统的未知状况或故障状况并自主处理这些状况。例子包括感知系统中会导致环境表达充满假障碍物的故障。这样就会触发清除成本地图恢复以允许机器人移动。

另一个例子就是机器人由于动态障碍物或控制不佳而卡住。在允许的情况下，倒退或原地旋转会允许机器人从卡住的位置移动到可以成功进行导航的自由空间中。

最后，在完全故障的情况下，可以实施恢复以引起操作员的注意以寻求帮助。这可以通过电子邮件、短信、Slack、Matrix等来完成。

3.4.2 航点跟随

航点跟随是导航系统的基本功能之一。它会告知系统如何使用导航程序到达多个目的地。

`nav2_waypoint_follower` 软件包含一个航路点跟踪程序，该程序具有特定任务执行程序的插件接口。如果需要让机器人前往给定位姿并完成像拍照、捡起盒子或等待用户输入之类的特定任务，这会非常有用。

> 关于机器人队管理器/调度器有两种思想流派：哑机器人+智能集中式调度器；智能机器人+哑集中式调度器。
>
> 在第一种思想中， `nav2_waypoint_follower` 软件包足以创造一个产品级的机器人解决方案。由于自主系统/调度器在分配任务时会考虑机器人的姿势、电池电量、当前任务等因素，机器人上的应用程序只需要关心手头的任务，而不用关心完成系统要求任务的其他复杂因素。在这种情况下，应该将发送至航点跟随者的请求视为1个工作单元（例如，仓库中的1次拣货、1个安全巡逻循环、1个过道等）来执行任务，然后返回给调度器以进行下一个任务或者要求充电。在这种思想流派中，航点跟随应用程序只是导航软件堆栈之上和系统自主应用程序之下的一个步骤。
>
> 在第二种思想中， `nav2_waypoint_follower` 软件包是一个不错的示例应用程序/概念证明，但确实需要机器人上的航点跟踪/自主系统来承担更多任务以制定健壮的解决方案。在这种情况下，应该使用 `nav2_behavior_tree` 软件包创建自定义应用程序级别的行为树，以使用导航来完成任务。这可以包含子树，例如在任务中检查充电状态以返回停泊坞，或者在更复杂的任务中处理1个以上的工作单元。很

> 快，将会有一个 `nav2_bt_waypoint_follower` (名称有待调整)，它将允许用户更容易地创建此应用程序。在这个思想流派中，航点跟随应用程序与自主系统的联系更加紧密，或者在很多情况下，航点跟随应用程序就是自主系统本身。
>
> 这两种思想流派并不能简单地说谁比谁更好，谁更好很大程度上取决于机器人正在完成何种任务、处于何种类型的环境中以及有何种可用的云资源。通常，对于既定的业务案例，这种区别非常明显。

3.5 状态估计（重要组件）

> Nav2中，默认进行状态估计的组件是AMCL (Adaptive Monte Carlo Localization)自适应蒙特卡洛定位。nav2中对应的功能包是nav2_amcl。

根据ROS社区标准，在导航项目中，需要提供两个主要的坐标转换。 `map` 到 `odom` 的坐标变换由定位系统(定位，建图，SLAM)提供， `odom` 到 `base_link` 的坐标转换由里程计系统提供。

注解：无需在机器人上使用LIDAR即可使用导航系统。不需要使用基于激光雷达的防撞、定位或SLAM系统。但是，Nav2确实可以提供说明和支持使用激光雷达对这些系统进行尝试和真实实现。使用基于视觉或深度传感器的定位系统和使用其他传感器来避免碰撞可以同样成功。唯一的要求就是在选择具体实现方式时遵循REP-105标准。

> REP-105标准
>
> REP 105 定义了导航和更大的ROS生态系统所需的框架和约定。应始终遵循这些约定，以利用社区中丰富的定位、里程计和SLAM项目。
>
> 简而言之，REP-105至少必须为机器人构造一个包含 `map` -> `odom` -> `base_link` -> `[sensorframes]` 的完整 的TF树。TF2是ROS 2中的时变坐标变换库，Nav2使用TF2来表达和获取时间同步的坐标变换。全球定位系统 (GPS、SLAM、动作捕捉Motion Capture) 的工作是至少要提供 `map-> odom` 的坐标转换。然后，里程计系统的作用是提供 `odom` -> `base_link` 的坐标转化。关于 `base_link` 的其余坐标转换应该是静态的，并应在 URDF 中定义。

3.6 全局定位: 定位与SLAM

全局定位系统 (GPS、SLAM、运动捕捉) 的工作是至少提供 `map` -> `odom` 的坐标转换。Nav2项目提供的 `amcl` 是一种基于粒子过滤器的自适应蒙特卡罗定位技术，用于静态地图的定位。Nav2还提供用于定位和生成静态映射的SLAM工具箱作为默认的SLAM算法。

这些方法还可能产生其他输出，包括位置话题、地图或其他元数据，但它们必须提供该转换才能有效。使用机器人定位可以将多种定位方法融合在一起，下面将详细讨论。

3.7 里程计（Odometry）

里程计系统的作用是提供 `odom` -> `base_link` 的坐标转换。里程计可以来自许多数据源，包括激光雷达、车轮编码器、VIO和IMU。里程计的目标是提供基于机器人运动的平滑和连续的局部坐标系。全局定位系统会相对全局坐标的坐标变换进行更新，以解决里程计的漂移问题。

这个 Robot Localization 通常用于这种融合。它将采用各种类型的 `N` 个传感器，并为TF和话题提供连续平滑的里程计。一个典型的移动机器人装置可能有来自车轮编码器或IMU的里程计以及融合在这个工作区内的视觉。

这样平滑输出就可用于精确运动的航行位置推算和在全局位置更新之间准确地更新机器人的位置。

3.8 环境表达（建模）

环境表征是机器人感知环境的方式。它还充当各种算法和数据源的中心定位工具，以将它们的信息组合到一个空间中。这样，控制器、规划器和恢复器就可以使用该空间来安全有效地计算它们的任务。

成本地图和图层

当前的环境表达是一个成本地图。成本地图是包含来自未知、空闲、占用或膨胀成本的单元格的规则2D单元格网格。然后搜索该成本地图以计算全局计划或采样以计算局部控制工作。

各种成本地图图层被实现为pluginlib插件，以将信息缓冲到成本地图中。这包括来自LIDAR、RADAR、声纳、深度传感器、图像传感器等的信息。最好在传感器数据输入到层本地图之前进行处理，但这取决于开发人员。

可以使用相机或深度传感器创建代价地图层来检测和跟踪场景中的障碍物，以避免碰撞。此外，可以创建层来基于一些规则或启发式算法来改变基础成本图。最后，它们可用于将实时数据缓冲到2D或3D世界中，以进行障碍物的二值化标记。

成本地图过滤器

想象一下，您正在注释地图文件 (或任何图像文件)，以便根据注释地图中的位置执行特定操作。

通过使用成本地图过滤器可以实现以下功能：

- 机器人永远不会进入的禁区/安全区。
- 限速区，机器人进入这些区域的最大速度将受到限制。
- 机器人在工业环境和仓库中移动的首选通道。

其他形式的环境表示

存在各种其他形式的环境表征。包括:

- 梯度图，类似于成本地图，但梯度图会表达表面梯度以检查可穿越性
- 3D成本图，以3D形式表示空间，但这样就也需要3D规划和碰撞检测
- 网格图，类似于梯度图，但具有多个角度的表面网格
- Vector space ，接收传感器信息并使用机器学习算法来检测要跟踪的单个物品和位置，而不是对离散点进行缓冲区计算

4.Nav2下载安装

看完了第三节的是不是似懂非懂，接下来我们将通过实验进一步来对照着学习。

Nav2的安装有两种方式，一种是通过二进制直接安装，另外一种是通过源码的方式进行安装，这里小鱼依然推荐源码的方式，毕竟我们还是要看一看Nav2的源码的。

4.1 apt安装

安装 `nav2`

```shell
sudo apt install ros-humble-nav2-*
```

4.2 源码安装

将下面的源码克隆到fishbot_ws的src目录下：

```shell
git clone https://ghproxy.com/https://github.com/ros-planning/navigation2.git -b humble
```

安装依赖

这里我们使用rosdepc进行依赖的安装，rosdepc指令找不到可以先运行下面的一键安装命令，选择一键配置rosdep即可。

```shell
wget http://fishros.com/install -O fishros && . fishros
```

接着在fishbot_ws下运行下面这个命令进行依赖的安装。

```shell
rosdepc install -r --from-paths src --ignore-src --rosdistro $ROS_DISTRO -y
```

编译

> 这里有一个新的命令--packages-up-to，意思是其所有依赖后再编译该包

```shell
colcon build --packages-up-to navigation2
```

4.3 测试是否安装成功

如果是源码编译**请先source下工作空间后再使用下面指令查看**是否安装成功；

```shell
ros2 pkg list | grep navigation2
```

能看到下面的结果即可

```shell
navigation2
```

5.源码功能包拆解

包含代码的功能包及其功能见下面列表

```
#==============控制器及其实现相关功能包======================#
nav2_controller   |   控制器
```

nav2_dwb_controller | DWB控制器，Nav2控制器的一个实现
nav2_regulated_pure_pursuit_controller | 纯追踪控制器，Nav2控制器的一个实现
nav2_constrained_smoother

#==============规划器及其实现相关功能包======================#
nav2_planner | Nav2规划器
nav2_navfn_planner | navfn规划器，Nav2规划器的一个实现
nav2_smac_planner | smac规划器，Nav2规划器的一个实现

#====================恢复器============================#
nav2_recoveries | Nav2恢复器

#====================行为树节点及其定义==================#
nav2_bt_navigator | 导航行为树
nav2_behavior_tree | 行为树节点插件定义

#====================地图和定位==========================#
nav2_map_server | 地图服务器
nav2_costmap_2d | 2D代价地图
nav2_voxel_grid | 体素栅格
nav2_amcl | 自适应蒙特卡洛定位。 状态估计，输入地图、激光、里程计数据，输出机器人map和odom之间的位资关系。

#====================通用插件系统管理等==================#
nav2_bringup | 启动入口
nav2_common | 公共功能包
nav2_msgs | 通信相关消息定义
nav2_util | 常用工具
nav2_lifecycle_manager |节点生命周期管理器
nav2_rviz_plugins | RVIZ插件

#====================核心定义==========================#
nav2_core | Nav2核心包
navigation2 | nav2导航汇总配置

#====================应用==============================#
nav2_waypoint_follower | 路点跟踪

#====================测试==============================#
nav2_system_tests | 系统测试

上面的每个节点都有自己的参数可以配置，下一节我们就对个个节点的配置进行介绍并进行适配

6.总结

本节我们对Nav2进行了详细的介绍和安装，下一节我们就开始为FishBot配置Nav2，然后开始尝试让Nav2控制Fishbot自主的移动起来。

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

2.为FishBot配置Nav2

安装好了Nav2，我们开始针对我们的fishbot改变一些参数进行导航相关的参数配置。Nav2可配置的参数比起Cartographer更多，但是不要害怕，因为大多数参数我们可能不会改变。

> 有关Nav2的更多参数介绍和详细的配置意义，可以参考**Nav2中文网配置指南**一节。

本节主要准备两个文件给launch节点使用，第一个是地图文件，第二个是nav2参数文件。

1.创建fishbot_navigation2

1.1创建功能包

和前面在Cartographer中一样，我们需要创建一个文件夹放置配置文件、launch文件、rviz配置和地图等。

进入到src目录下，使用下面指令创建功能包：

```shell
ros2 pkg create fishbot_navigation2 --dependencies nav2_bringup
```

这里我们添加了一个依赖 `nav2_bringup` ，后面写launch文件要用到，这里提前添加一下依赖。

创建完成后的目录结构：

```
.
├── CMakeLists.txt
├── include
│   └── fishbot_navigation2
├── package.xml
└── src

3 directories, 2 files
```

1.2 添加maps文件夹

```
cd src/fishbot_navigation2
mkdir launch config maps param rviz
```

1.3 复制地图文件

将上一节的地图文件复制到map文件夹下。

复制完成后 `fishbot_navigation2` 的文件结构如下

```
.
├── CMakeLists.txt
├── config
├── launch
├── maps
│   ├── fishbot_map.png
│   ├── fishbot_map.pgm
│   ├── fishbot_map.yaml
├── package.xml
├── param
└── rviz

5 directories, 5 files
```

2.添加Nav2配置文件

2.1 创建参数文件

我们需要配置的文件是Nav2的参数文件，同样的，贴心的Nav2已经为我们准备好了参数模板

```
src/navigation2/nav2_bringup/bringup/params/nav2_params.yaml
```

在 `src/fishbot_navigation2/param/` 目录下创建 `fishbot_nav2.yaml`

```shell
cd src/fishbot_navigation2/param/
touch fishbot_nav2.yaml
```

2.2 复制参数

然后将 `src/navigation2/nav2_bringup/bringup/params/nav2_params.yaml` 的内容复制粘贴到 `fishbot_nav2.yaml` 文件中。

> **参数文件中的参数是谁的？**
>
> 在5.1.4章节保存参数中，我们曾用 `ros2 param dump <node_name>` 指令将某个节点的参数保存为一个 `.yaml` 格式的文件。 `fishbot_nav2.yaml` 文件就是保存Nav2相关节点参数的文件。

3. 配置参数

其实参数不配置也是可以将Nav2跑起来的，但是后期想要更改运行的效果就需要对参数进行修改，所以有必要大概了解下参数的配置项和含义查询方法和修改方法。

3.1 参数列表

编号	配置项	用途	对应模块与参数详解
1	amcl	机器人定位	nav2_amcl
2	bt_navigator	导航行为树（用于加载行为树节点并根据xml配置进行	nav2_bt_navigator,nav2_behavior_tree

		调度）	
3	controller_server	控制器服务器	nav2_controller,nav2_dwb_controller,nav2_regulated_pure_pursuit_
4	planner_server	规划服务器	nav2_planner,nav2_navfn_planner,smac_planner
5	recoveries_server	恢复服务器	nav2_recoveries
6	local_costmap	局部代价地图	nav2_costmap_2d,static_layer,inflation_layer
7	global_costmap	全局代价地图	nav2_costmap_2d,nav2_map_server

有关更多的Nav2所有参数的详细介绍，可以访问Nav2中文网中的配置指南章节，非常的详细，无比的具体。

配置指南

本指南提供了一个流程，通过该流程，用户可以调整可调参数以获得最佳导航性能。 [校准@小鱼]

- 路点追踪
- 行为树导航器
- 行为树XML节点
- 2D代价地图
- 生命周期管理
- 规划器服务器
- NavFn规划器
- Smac规划器
- Theta星规划器
- 控制器服务器 [待校准@2573]
- DWB控制器
- Map Server / Saver
- AMCL（自适应蒙特卡洛定位）
- 恢复服务器
- 纯追踪控制
- 旋转控制器

3.2 配置机器人半径和碰撞半径

在全局代价地图和局部代价地图配置用，默认的机器人半径是0.22，而我们fishbot的半径是0.12，所以需要修改机器人的半径为0.12。

```yaml
local_costmap:
  local_costmap:
    ros__parameters:
      robot_radius: 0.12

global_costmap:
  global_costmap:
    ros__parameters:
      robot_radius: 0.12
```

为了防止机器人发生碰撞，一般我们会给代价地图添加一个碰撞层（inflation_layer），在 `local_costmap` 和 `global_costmap` 配置中，你可以看到下面关于代价地图相关的配置：

```yaml
global_costmap:
  global_costmap:
    ros__parameters:
      plugins: ["static_layer", "obstacle_layer", "inflation_layer"]
      inflation_layer:
        plugin: "nav2_costmap_2d::InflationLayer"
        cost_scaling_factor: 3.0
        inflation_radius: 0.55
```

打开参数配置中的inflation_layer，我们来看看其配置项和含义。

可以看到 `inflation_radius` 默认0.55对fishbot来说可能有些大了，我们改小些。

```
global_costmap:
  global_costmap:
    ros__parameters:
      plugins: ["static_layer", "obstacle_layer", "inflation_layer"]
      inflation_layer:
        plugin: "nav2_costmap_2d::InflationLayer"
```

```
        cost_scaling_factor: 3.0
        inflation_radius: 0.35
```

以上就是以代价地图碰撞半径为例的配置方法，nav2可以配置的参数非常多，假如你在导航过程中遇到问题，根据问题的表现推断下是哪个模块中造成的，接着修改其对应参数，大概率就可以解决问题，解决不了的可以看源码详细分析。

3.3 配置frame_id和话题

这里也不用配置，因为我们的fishbot话题名称和tf名称都是遵循着默认的话题的。

如果你的机器人不是，或者你改变了话题，这里就需要重新配置。

- 默认全局的坐标系：map
- 默认里程计坐标系：odom
- 默认雷达话题：scan
- 默认机器人基坐标系：base_link
- 默认地图话题：map

4. 总结

本节我们简单的了解了Nav2各个模块参数的配置方法和参数的介绍，下一节就开始编写文件正式建图。

完整的配置文件如下

```
amcl:
  ros__parameters:
    use_sim_time: True
    alpha1: 0.2
    alpha2: 0.2
    alpha3: 0.2
    alpha4: 0.2
    alpha5: 0.2
    base_frame_id: "base_link"
    beam_skip_distance: 0.5
    beam_skip_error_threshold: 0.9
    beam_skip_threshold: 0.3
```

```
    do_beamskip: false
    global_frame_id: "map"
    lambda_short: 0.1
    laser_likelihood_max_dist: 2.0
    laser_max_range: 100.0
    laser_min_range: -1.0
    laser_model_type: "likelihood_field"
    max_beams: 60
    max_particles: 2000
    min_particles: 500
    odom_frame_id: "odom"
    pf_err: 0.05
    pf_z: 0.99
    recovery_alpha_fast: 0.0
    recovery_alpha_slow: 0.0
    resample_interval: 1
    robot_model_type: "nav2_amcl::DifferentialMotionModel"
    save_pose_rate: 0.5
    sigma_hit: 0.2
    tf_broadcast: true
    transform_tolerance: 10.0
    update_min_a: 0.2
    update_min_d: 0.25
    z_hit: 0.5
    z_max: 0.05
    z_rand: 0.5
    z_short: 0.05
    scan_topic: scan

amcl_map_client:
  ros__parameters:
    use_sim_time: True

amcl_rclcpp_node:
  ros__parameters:
    use_sim_time: True

bt_navigator:
  ros__parameters:
    use_sim_time: True
    global_frame: map
    robot_base_frame: base_link
```

```
    odom_topic: /odom
    bt_loop_duration: 10
    default_server_timeout: 20
    # 'default_nav_through_poses_bt_xml' and 'default_nav_to_pose_bt_xml' are use
defaults:
    # nav2_bt_navigator/navigate_to_pose_w_replanning_and_recovery.xml
    # nav2_bt_navigator/navigate_through_poses_w_replanning_and_recovery.xml
    # They can be set here or via a RewrittenYaml remap from a parent launch file
to Nav2.
    plugin_lib_names:
    - nav2_compute_path_to_pose_action_bt_node
    - nav2_compute_path_through_poses_action_bt_node
    - nav2_smooth_path_action_bt_node
    - nav2_follow_path_action_bt_node
    - nav2_spin_action_bt_node
    - nav2_wait_action_bt_node
    - nav2_back_up_action_bt_node
    - nav2_drive_on_heading_bt_node
    - nav2_clear_costmap_service_bt_node
    - nav2_is_stuck_condition_bt_node
    - nav2_goal_reached_condition_bt_node
    - nav2_goal_updated_condition_bt_node
    - nav2_globally_updated_goal_condition_bt_node
    - nav2_is_path_valid_condition_bt_node
    - nav2_initial_pose_received_condition_bt_node
    - nav2_reinitialize_global_localization_service_bt_node
    - nav2_rate_controller_bt_node
    - nav2_distance_controller_bt_node
    - nav2_speed_controller_bt_node
    - nav2_truncate_path_action_bt_node
    - nav2_truncate_path_local_action_bt_node
    - nav2_goal_updater_node_bt_node
    - nav2_recovery_node_bt_node
    - nav2_pipeline_sequence_bt_node
    - nav2_round_robin_node_bt_node
    - nav2_transform_available_condition_bt_node
    - nav2_time_expired_condition_bt_node
    - nav2_path_expiring_timer_condition
    - nav2_distance_traveled_condition_bt_node
    - nav2_single_trigger_bt_node
    - nav2_is_battery_low_condition_bt_node
    - nav2_navigate_through_poses_action_bt_node
```

```
    - nav2_navigate_to_pose_action_bt_node
    - nav2_remove_passed_goals_action_bt_node
    - nav2_planner_selector_bt_node
    - nav2_controller_selector_bt_node
    - nav2_goal_checker_selector_bt_node
    - nav2_controller_cancel_bt_node
    - nav2_path_longer_on_approach_bt_node
    - nav2_wait_cancel_bt_node
    - nav2_spin_cancel_bt_node
    - nav2_back_up_cancel_bt_node
    - nav2_drive_on_heading_cancel_bt_node

bt_navigator_rclcpp_node:
  ros__parameters:
    use_sim_time: True

controller_server:
  ros__parameters:
    use_sim_time: True
    controller_frequency: 20.0
    min_x_velocity_threshold: 0.001
    min_y_velocity_threshold: 0.5
    min_theta_velocity_threshold: 0.001
    failure_tolerance: 0.3
    progress_checker_plugin: "progress_checker"
    goal_checker_plugins: ["general_goal_checker"] # "precise_goal_checker"
    controller_plugins: ["FollowPath"]

    # Progress checker parameters
    progress_checker:
      plugin: "nav2_controller::SimpleProgressChecker"
      required_movement_radius: 0.5
      movement_time_allowance: 10.0
    # Goal checker parameters
    #precise_goal_checker:
    #  plugin: "nav2_controller::SimpleGoalChecker"
    #  xy_goal_tolerance: 0.25
    #  yaw_goal_tolerance: 0.25
    #  stateful: True
    general_goal_checker:
      stateful: True
      plugin: "nav2_controller::SimpleGoalChecker"
```

```
      xy_goal_tolerance: 0.25
      yaw_goal_tolerance: 0.25
    # DWB parameters
    FollowPath:
      plugin: "dwb_core::DWBLocalPlanner"
      debug_trajectory_details: True
      min_vel_x: 0.0
      min_vel_y: 0.0
      max_vel_x: 0.26
      max_vel_y: 0.0
      max_vel_theta: 1.0
      min_speed_xy: 0.0
      max_speed_xy: 0.26
      min_speed_theta: 0.0
      # Add high threshold velocity for turtlebot 3 issue.
      # https://github.com/ROBOTIS-GIT/turtlebot3_simulations/issues/75
      acc_lim_x: 2.5
      acc_lim_y: 0.0
      acc_lim_theta: 3.2
      decel_lim_x: -2.5
      decel_lim_y: 0.0
      decel_lim_theta: -3.2
      vx_samples: 20
      vy_samples: 5
      vtheta_samples: 20
      sim_time: 1.7
      linear_granularity: 0.05
      angular_granularity: 0.025
      transform_tolerance: 0.2
      xy_goal_tolerance: 0.25
      trans_stopped_velocity: 0.25
      short_circuit_trajectory_evaluation: True
      stateful: True
      critics: ["RotateToGoal", "Oscillation", "BaseObstacle", "GoalAlign",
"PathAlign", "PathDist", "GoalDist"]
      BaseObstacle.scale: 0.02
      PathAlign.scale: 32.0
      PathAlign.forward_point_distance: 0.1
      GoalAlign.scale: 24.0
      GoalAlign.forward_point_distance: 0.1
      PathDist.scale: 32.0
      GoalDist.scale: 24.0
```

```
      RotateToGoal.scale: 32.0
      RotateToGoal.slowing_factor: 5.0
      RotateToGoal.lookahead_time: -1.0

controller_server_rclcpp_node:
  ros__parameters:
    use_sim_time: True

local_costmap:
  local_costmap:
    ros__parameters:
      update_frequency: 5.0
      publish_frequency: 2.0
      global_frame: odom
      robot_base_frame: base_link
      use_sim_time: True
      rolling_window: true
      width: 3
      height: 3
      resolution: 0.05
      robot_radius: 0.22
      plugins: ["voxel_layer", "inflation_layer"]
      inflation_layer:
        plugin: "nav2_costmap_2d::InflationLayer"
        cost_scaling_factor: 3.0
        inflation_radius: 0.55
      voxel_layer:
        plugin: "nav2_costmap_2d::VoxelLayer"
        enabled: True
        publish_voxel_map: True
        origin_z: 0.0
        z_resolution: 0.05
        z_voxels: 16
        max_obstacle_height: 2.0
        mark_threshold: 0
        observation_sources: scan
        scan:
          topic: /scan
          max_obstacle_height: 2.0
          clearing: True
          marking: True
          data_type: "LaserScan"
```

```
          raytrace_max_range: 3.0
          raytrace_min_range: 0.0
          obstacle_max_range: 2.5
          obstacle_min_range: 0.0
      static_layer:
        map_subscribe_transient_local: True
      always_send_full_costmap: True
  local_costmap_client:
    ros__parameters:
      use_sim_time: True
  local_costmap_rclcpp_node:
    ros__parameters:
      use_sim_time: True

global_costmap:
  global_costmap:
    ros__parameters:
      update_frequency: 1.0
      publish_frequency: 1.0
      global_frame: map
      robot_base_frame: base_link
      use_sim_time: True
      robot_radius: 0.22
      resolution: 0.05
      track_unknown_space: true
      plugins: ["static_layer", "obstacle_layer", "inflation_layer"]
      obstacle_layer:
        plugin: "nav2_costmap_2d::ObstacleLayer"
        enabled: True
        observation_sources: scan
        scan:
          topic: /scan
          max_obstacle_height: 2.0
          clearing: True
          marking: True
          data_type: "LaserScan"
          raytrace_max_range: 3.0
          raytrace_min_range: 0.0
          obstacle_max_range: 2.5
          obstacle_min_range: 0.0
      static_layer:
        plugin: "nav2_costmap_2d::StaticLayer"
```

```
        map_subscribe_transient_local: True
      inflation_layer:
        plugin: "nav2_costmap_2d::InflationLayer"
        cost_scaling_factor: 3.0
        inflation_radius: 0.55
      always_send_full_costmap: True
  global_costmap_client:
    ros__parameters:
      use_sim_time: True
  global_costmap_rclcpp_node:
    ros__parameters:
      use_sim_time: True

map_server:
  ros__parameters:
    use_sim_time: True
    yaml_filename: "turtlebot3_world.yaml"

map_saver:
  ros__parameters:
    use_sim_time: True
    save_map_timeout: 5.0
    free_thresh_default: 0.25
    occupied_thresh_default: 0.65
    map_subscribe_transient_local: True

planner_server:
  ros__parameters:
    expected_planner_frequency: 20.0
    use_sim_time: True
    planner_plugins: ["GridBased"]
    GridBased:
      plugin: "nav2_navfn_planner/NavfnPlanner"
      tolerance: 0.5
      use_astar: false
      allow_unknown: true

planner_server_rclcpp_node:
  ros__parameters:
    use_sim_time: True

smoother_server:
```

```
    ros__parameters:
      use_sim_time: True
      smoother_plugins: ["simple_smoother"]
      simple_smoother:
        plugin: "nav2_smoother::SimpleSmoother"
        tolerance: 1.0e-10
        max_its: 1000
        do_refinement: True

  behavior_server:
    ros__parameters:
      costmap_topic: local_costmap/costmap_raw
      footprint_topic: local_costmap/published_footprint
      cycle_frequency: 10.0
      behavior_plugins: ["spin", "backup", "drive_on_heading", "wait"]
      spin:
        plugin: "nav2_behaviors/Spin"
      backup:
        plugin: "nav2_behaviors/BackUp"
      drive_on_heading:
        plugin: "nav2_behaviors/DriveOnHeading"
      wait:
        plugin: "nav2_behaviors/Wait"
      global_frame: odom
      robot_base_frame: base_link
      transform_tolerance: 0.1
      use_sim_time: true
      simulate_ahead_time: 2.0
      max_rotational_vel: 1.0
      min_rotational_vel: 0.4
      rotational_acc_lim: 3.2

  robot_state_publisher:
    ros__parameters:
      use_sim_time: True

  waypoint_follower:
    ros__parameters:
      loop_rate: 20
      stop_on_failure: false
      waypoint_task_executor_plugin: "wait_at_waypoint"
      wait_at_waypoint:
```

```
        plugin: "nav2_waypoint_follower::WaitAtWaypoint"
        enabled: True
        waypoint_pause_duration: 200
```

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

3.使用FishBot进行自主导航

经过前面三节的铺垫，我们只需要再编写一个launch文件启动nav2就可以让fishbot自己动起来了。

1.编写launch文件

我们将地图、配置文件传递给nav2为我们提供好的launch文件中即可。

> 再一个launch文件中包裹另一个功能包中的luanch文件采用的是`IncludeLaunchDescription`和`PythonLaunchDescriptionSource`

```python
'''
作者: 小鱼
公众号: 鱼香ROS
QQ交流群: 2642868461
描述: Nav2 launch启动文件
'''
import os
from ament_index_python.packages import get_package_share_directory
from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import PythonLaunchDescriptionSource
from launch.substitutions import LaunchConfiguration
from launch_ros.actions import Node


def generate_launch_description():
    #============================1.定位到包的地址
============================================================
    fishbot_navigation2_dir = get_package_share_directory('fishbot_navigation2')
    nav2_bringup_dir = get_package_share_directory('nav2_bringup')


    #============================2.声明参数，获取配置文件路径
=====================================================
    # use_sim_time 这里要设置成true,因为gazebo是仿真环境，其时间是通过/clock话题获取，而不是
系统时间
    use_sim_time = LaunchConfiguration('use_sim_time', default='true')
```

```
    map_yaml_path =
LaunchConfiguration('map',default=os.path.join(fishbot_navigation2_dir,'maps','fis
hbot_map.yaml'))
    nav2_param_path =
LaunchConfiguration('params_file',default=os.path.join(fishbot_navigation2_dir,'pa
ram','fishbot_nav2.yaml'))
    rviz_config_dir =
os.path.join(nav2_bringup_dir,'rviz','nav2_default_view.rviz')

    #============================3.声明启动launch文件，传入：地图路径、是否使用仿真时间以及
nav2参数文件==============
    nav2_bringup_launch = IncludeLaunchDescription(

PythonLaunchDescriptionSource([nav2_bringup_dir,'/launch','/bringup_launch.py']),
            launch_arguments={
                'map': map_yaml_path,
                'use_sim_time': use_sim_time,
                'params_file': nav2_param_path}.items(),
        )
    rviz_node =  Node(
            package='rviz2',
            executable='rviz2',
            name='rviz2',
            arguments=['-d', rviz_config_dir],
            parameters=[{'use_sim_time': use_sim_time}],
            output='screen')

    return LaunchDescription([nav2_bringup_launch,rviz_node])
```

2.安装并添加依赖

2.1 修改CMakeLists.txt

添加install指令，将文件拷贝到install目录

```cmake
cmake_minimum_required(VERSION 3.5)
project(fishbot_navigation2)
```

```
# find dependencies
find_package(ament_cmake REQUIRED)
install(
  DIRECTORY launch param maps
  DESTINATION share/${PROJECT_NAME}
)

ament_package()
```

2.2 添加依赖

主要是添加这行 `<exec_depend>nav2_bringup</exec_depend>`

```xml
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd"
schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>fishbot_navigation2</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="sangxin2014@sina.com">root</maintainer>
  <license>TODO: License declaration</license>

  <buildtool_depend>ament_cmake</buildtool_depend>

  <test_depend>ament_lint_auto</test_depend>
  <test_depend>ament_lint_common</test_depend>
  <exec_depend>nav2_bringup</exec_depend>
  <export>
    <build_type>ament_cmake</build_type>
  </export>
</package>
```

3.构建运行

3.1 构建

```shell
colcon build --packages-up-to  fishbot_navigation2
```

3.2 运行

3.2.1 运行仿真

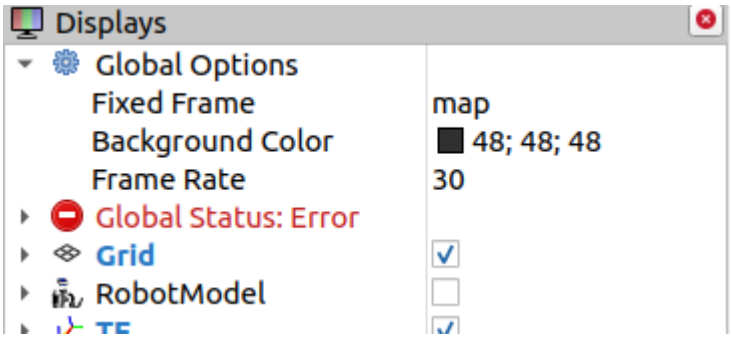
```
source install/setup.bash
ros2 launch fishbot_description gazebo.launch.py
```

3.2.2 运行Nav2

```
source install/setup.bash
ros2 launch fishbot_navigation2 navigation2.launch.py
```

4.初始化位置

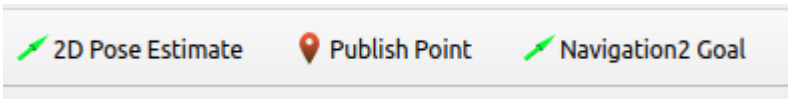
启动后正常你应该在RVIZ2和终端看到一个错误，这是因为没有给定初始化位置（告诉机器人它在地图的大概位置）导致的。



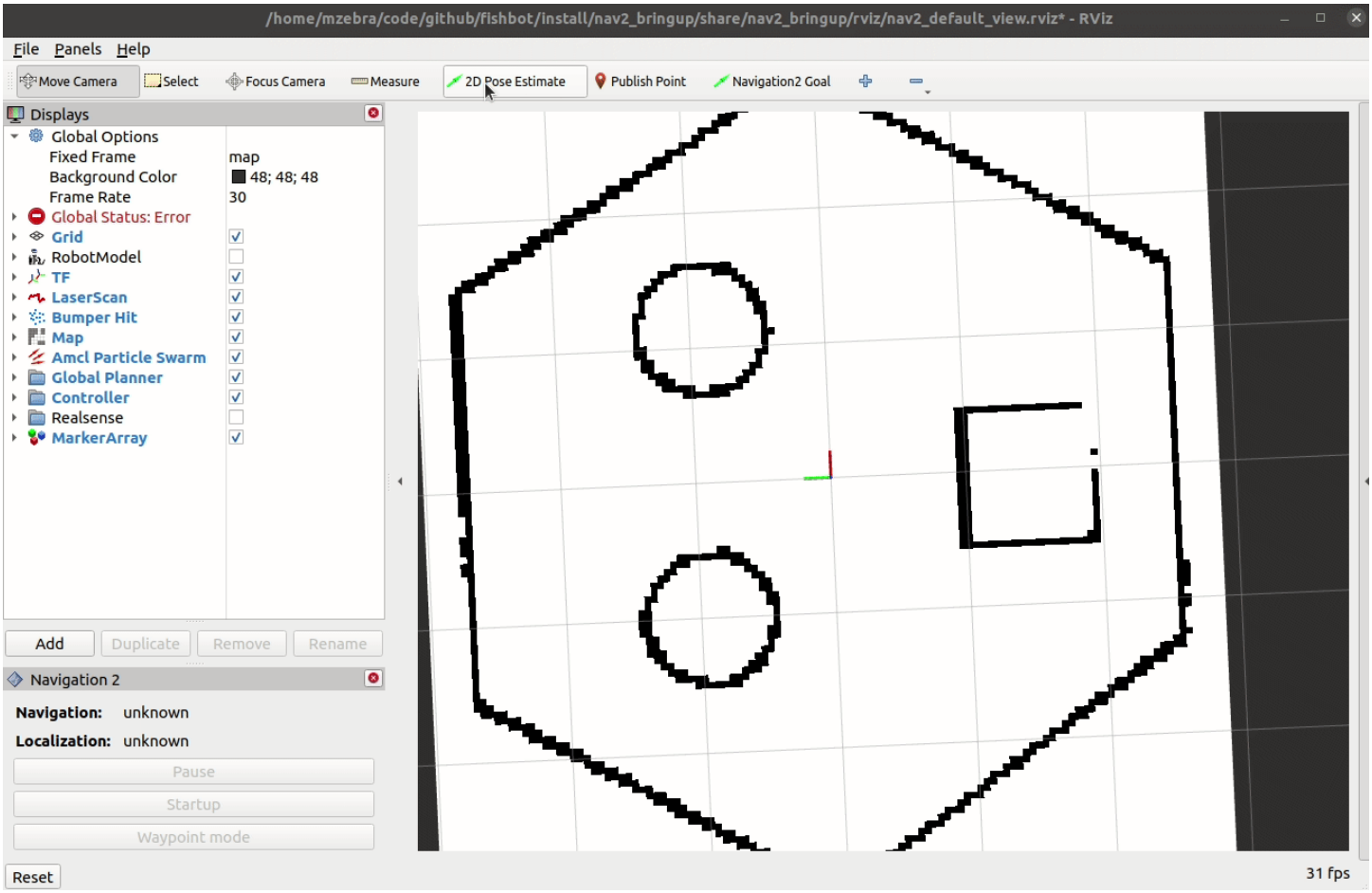

```shell
[planner_server-5] [INFO] [1652973621.731976741] [global_costmap.global_costmap]:
Timed out waiting for transform from base_link to map to become available, tf
error: Invalid frame ID "map" passed to canTransform argument target_frame - frame
```

```
does not exist
[rviz2-10] [INFO] [1652973621.760971376] [rviz2]: Message Filter dropping message:
frame 'odom' at time 0.000 for reason 'Unknown'
[rviz2-10] [INFO] [1652973621.856298950] [rviz2]: Message Filter dropping message:
frame 'laser_link' at time 4392.881 for reason 'Unknown'
[rviz2-10] [INFO] [1652973621.951345246] [rviz2]: Message Filter dropping message:
frame 'laser_link' at time 4392.981 for reason 'Unknown'
[rviz2-10] [INFO] [1652973621.951468235] [rviz2]: Message Filter dropping message:
frame 'odom' at time 0.000 for reason 'Unknown'
[rviz2-10] [INFO] [1652973622.047860791] [rviz2]: Message Filter dropping message:
frame 'laser_link' at time 4393.081 for reason 'Unknown'
```

通过RVIZ2的工具栏上的 `2D Pose Estimate` 可以给迷茫的fishbot指明“机生方向”。

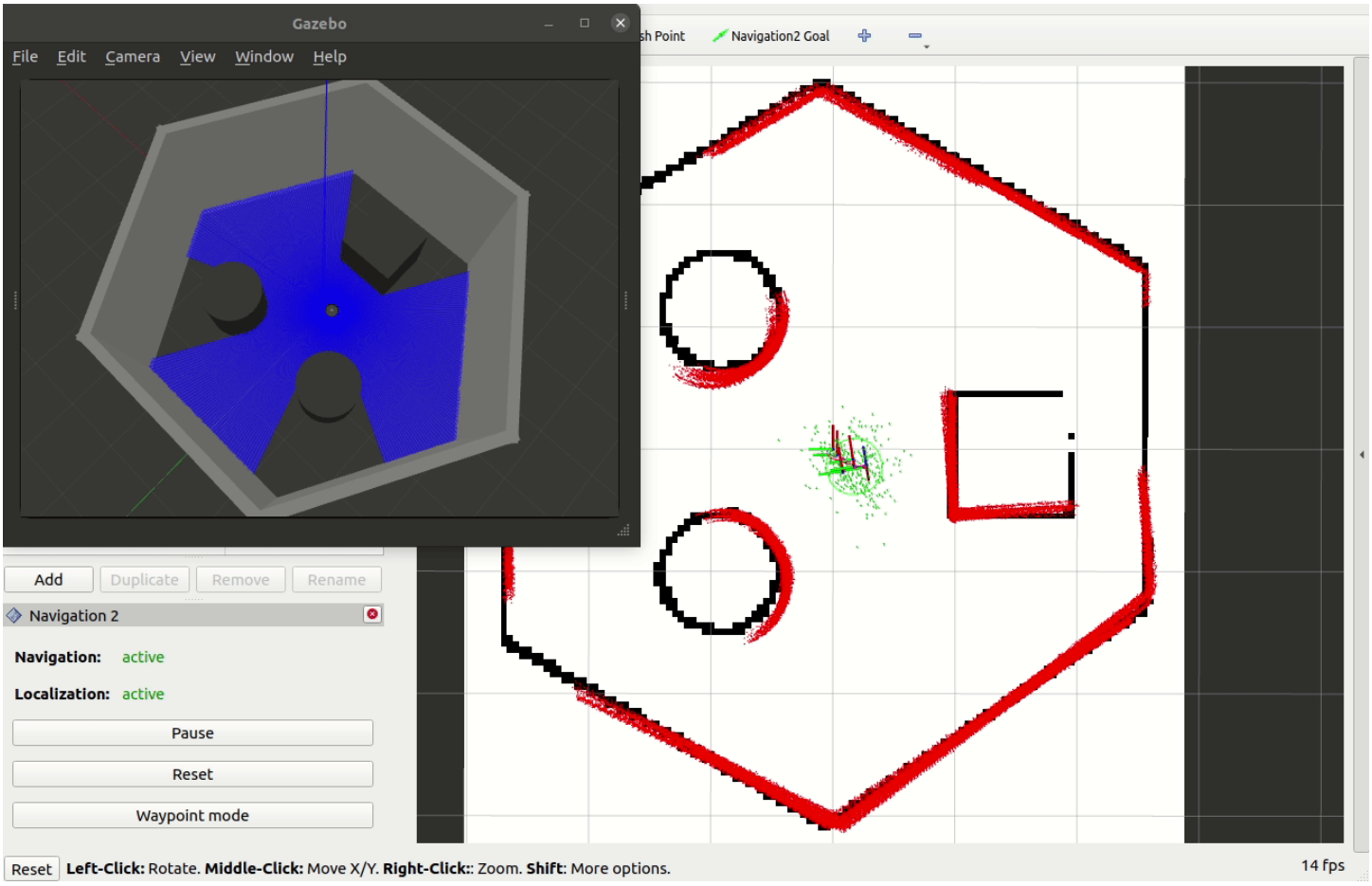


点击 `2D Pose Estimate` ，进行姿态初始化（选中机器人在Gazebo位置差不多的点，左键点击不要松开，移动鼠标给定方向），初始化完后，左边的Global Status 就正常了。



5.单点导航

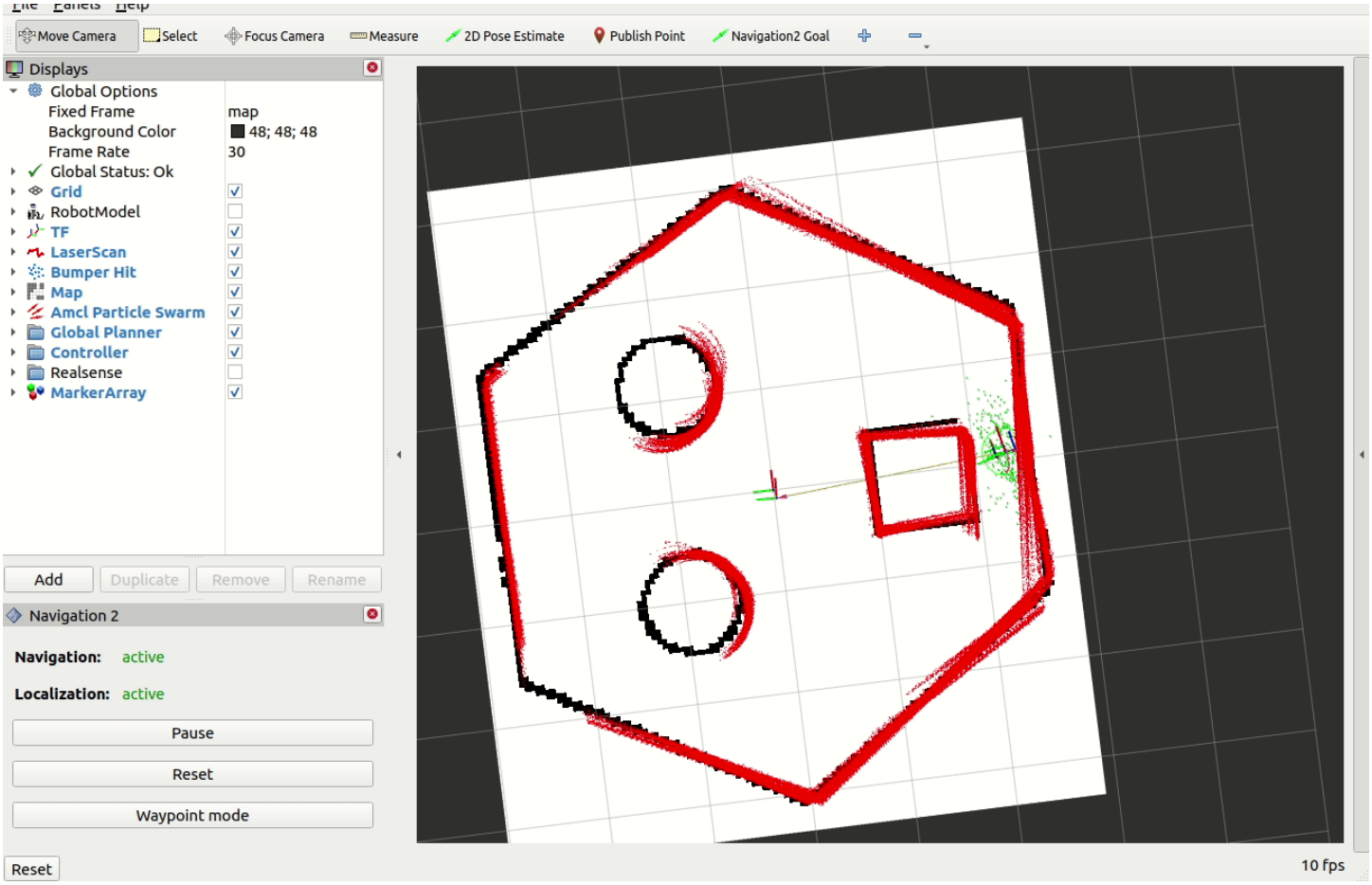
点击RVIZ2工具栏上的 就可以给fishbot安排一个目标点了，点击按钮，到地图上任意一点击鼠标左键，注意不要松开，移动鼠标给定一个方向。



6.多点（路点）导航

观察左下角，有一个Nav2的Rviz2小插件，可以进行启动停止和导航模式的切换，点击 切换到路点模式。

接着你可以使用工具栏的 按钮，给FishBot指定多个要移动的点，接着点击左下角的启动，就可以看到FishBot依次到达这些目标点。



7.查看机器人当前在地图中的位置

在机器人导航过程中我们如何实时查看机器人在地图中的位置呢？我们可以通过tf进行查看。

打开终端，输入指令：

```shell
ros2 run tf2_ros  tf2_echo map base_link
```

接着就可以看到下面的信息，旋转和变换位资数据了

```shell
[INFO] [1653215686.225862749] [tf2_echo]: Waiting for transform map ->  base_link:
Invalid frame ID "map" passed to canTransform argument target_frame - frame does
not exist
At time 4873.528000000
- Translation: [-0.009, -0.016, 0.076]
- Rotation: in Quaternion [0.000, -0.000, -0.000, 1.000]
At time 4874.514000000
```

```
  - Translation: [-0.009, -0.016, 0.076]
  - Rotation: in Quaternion [0.000, -0.000, -0.000, 1.000]
```

8.总结

本节我们终于将FishBot的自主导航给跑了起来，不过我们都是通过RVIZ2的工具给机器人指定的目标点，除了工具栏的工具，我们还可以使用代码的方式发送目标点给FishBot，下一节小鱼就会带你一起学习Nav2的API（应用程序接口）。

课后作业：

1. 使用rqt_tf_tree观察tf树的结构
2. 阅读下Nav2中文网配置指南章节内容（顺便挑几个代校准的段落进行校准）

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

4.使用Nav2导航API进行导航

Nav2的API其实是Nav2提供的一个Python库，通过该库你可以事先调用你的机器人进行简单的控制（比如导航到点）。

很遗憾的是，该功能包官方并没有发布foxy版本的，再下一个galactic版本才开始正式发布。

从2022年5月23号开始，教程将开始向humble版本迁移，该部分内容将在humble版本发布。

1.导入nav2_simple_commander

```
from nav2_simple_commander.robot_navigator import BasicNavigator
import rclpy
from copy import deepcopy
```

初始化BasicNavigator

```
rclpy.init()
nav = BasicNavigator()
navigator.waitUntilNav2Active()
```

2.初始化位置

```
# ======================初始化位置，代替rviz2的2D Pose
Estimate================================
initial_pose = PoseStamped()
initial_pose.header.frame_id = 'map'
initial_pose.header.stamp = navigator.get_clock().now().to_msg()
initial_pose.pose.position.x = 0.0
initial_pose.pose.position.y = 0.0
initial_pose.pose.orientation.w = 1.0
navigator.setInitialPose(initial_pose)
```

3.导航到点

```
#=======================导航到目标点1=========================================
goal_pose1 = deepcopy(initial_pose)
goal_pose1.pose.position.x = 1.5
nav.goToPose(goal_pose1)
while not nav.isNavComplete():
  feedback = nav.getFeedback()
  #检查是否超时，超时则停止导航到点
  if feedback.navigation_duration > 600:
    nav.cancelNav()


#================================导航到目标点2==================================
goal_pose2 = deepcopy(initial_pose)
goal_pose2.pose.position.x = -1.5

nav.goToPose(goal_pose2)
while not nav.isNavComplete():
  feedback = nav.getFeedback()
  #检查是否超时，超时则停止导航到点
  if feedback.navigation_duration > 600:
    nav.cancelNav()

#=============================查看返回结果=====================================
result = nav.getResult()
if result == NavigationResult.SUCCEEDED:
    print('Goal succeeded!')
elif result == NavigationResult.CANCELED:
    print('Goal was canceled!')
elif result == NavigationResult.FAILED:
    print('Goal failed!')
```

4.总结

上面是对nav2_simple_commander的简单介绍，介于本教程目前主要维护foxy版本问题，暂时该部分无法使用，不过你可以使用galactic版本进行实践。

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

第十三章嵌入式开发之从点灯开始

你好，我是小鱼。本章开始我们将一起走进机器人硬件的开发，学习如何通过程序控制机器人运动，通过代码驱动传感器获取环境的信息，本教程的食用方法如下。

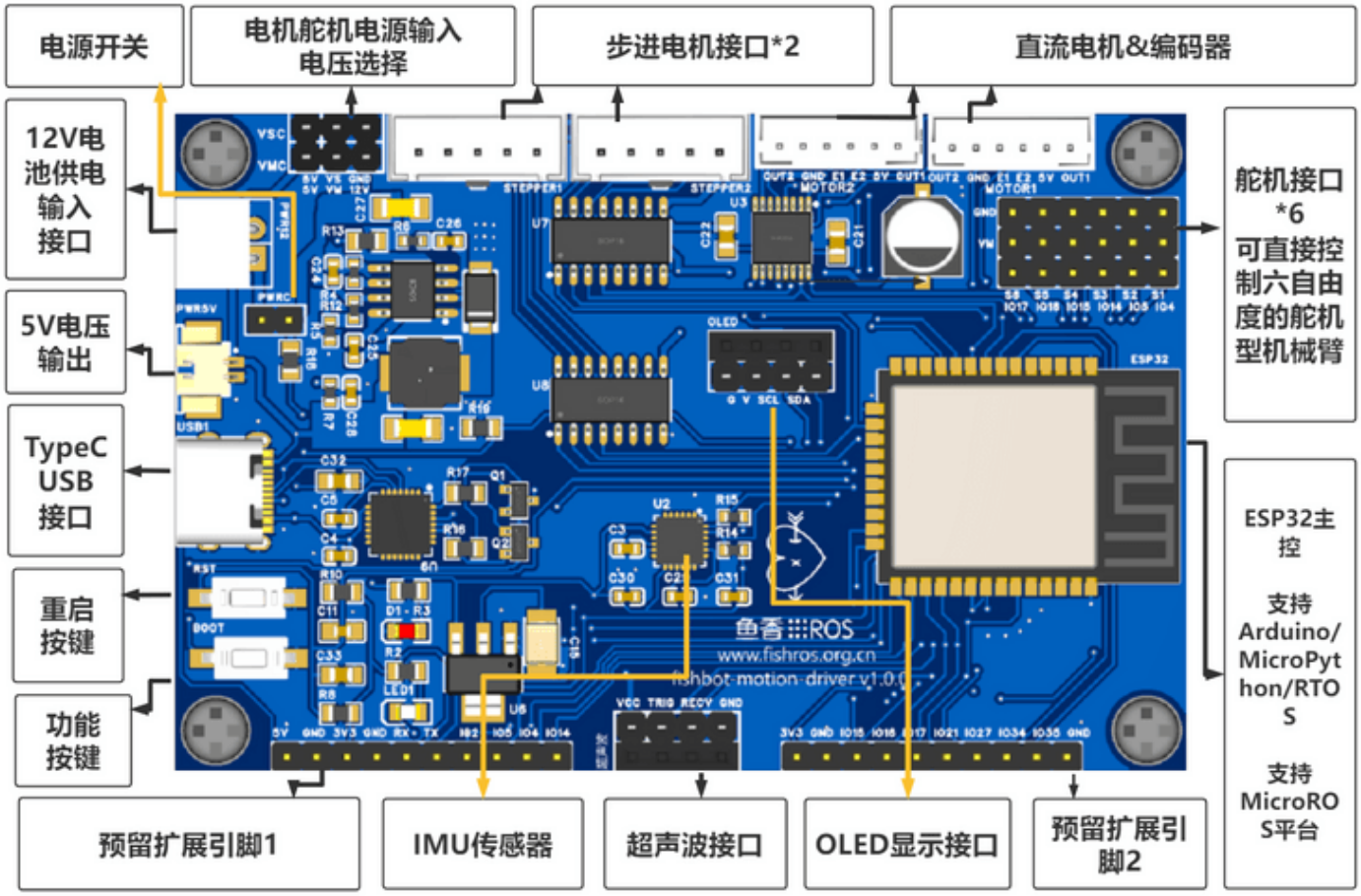
- 基础篇，如果你是嵌入式方面的小白，即看不懂原理图，也不知道如何在微处理器上运行代码，那么请从基础篇开始，小鱼将从微处理器介绍以及点灯原理开始讲起。
- 入门篇，如果你对嵌入式有一定了解，但是对ESP32以及在PlatFormIO上开发Arduino不太了解，那么请从入门篇开始学习，小鱼将从安装PlatformIO开始带你一步步实现点灯。
- 进阶篇，最后的进阶篇，小鱼将带你学习驱动OLED显示器，学习如何在Arduino上安装依赖库以及在微处理器上用C++实现面向对象开发的方法。

友情提示：和我们课程的名称一样，本教程中所有代码都非常建议跟着敲一遍，不要直接复制粘贴，写了才是自己的。

硬件开发平台

嵌入式开发无法脱离硬件，为方便学习，本教程配套的硬件是小鱼自制的MicroROS学习板，同时该板可以作为下一章节搭建实体移动机器人的主控板以及后续制作机械臂的驱动板使用。

板载资源图如下：



该主控板可以在小鱼的店铺直接购买，性价比接地气，直达链接：https://item.taobao.com/item.htm?id=695473143304。

同时该板的原理图和PCB小鱼都已经开源了，时间充足且动手能力强工具齐全的的小伙伴可以自己购买元器件制作，开源地址见鱼香ROS社区：https://www.fishros.org.cn/forum/topic/894

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

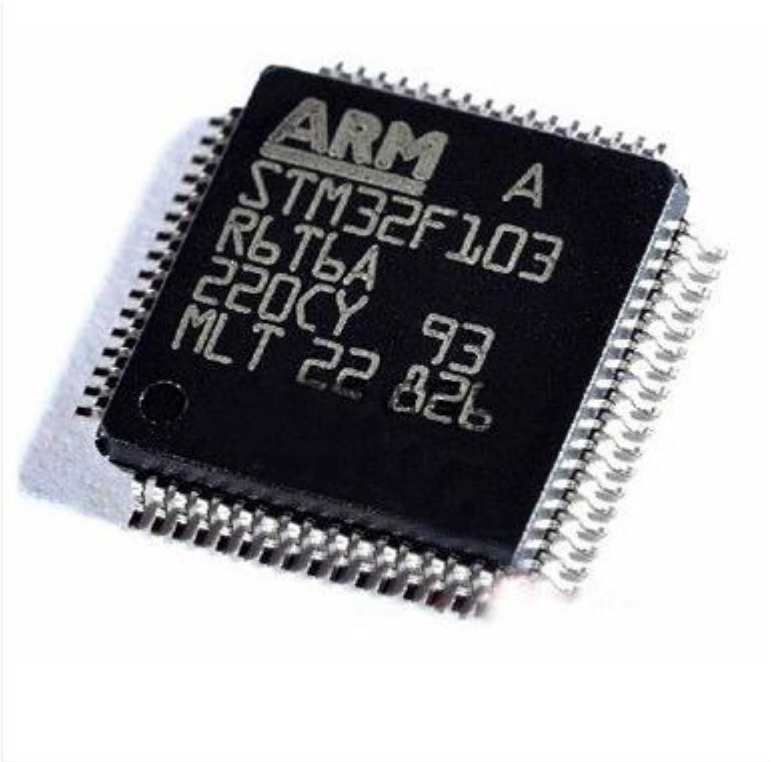
1.什么是单片机MCU

你好，我是爱吃鱼香ROS的小鱼。第一节基础课，我们先认识一下接下来要和我们打交道的主角——MCU。

一、什么是MCU？和CPU什么区别？



你知道我们电脑用的处理器叫做CPU（Central Processing Unit 中央处理器），那你知道现在的智能家电里的处理器是什么吗？比如可以连WIFI和蓝牙的空调，比如在寒冷的冬天里小鱼常用的小米电暖器是什么处理器吗？



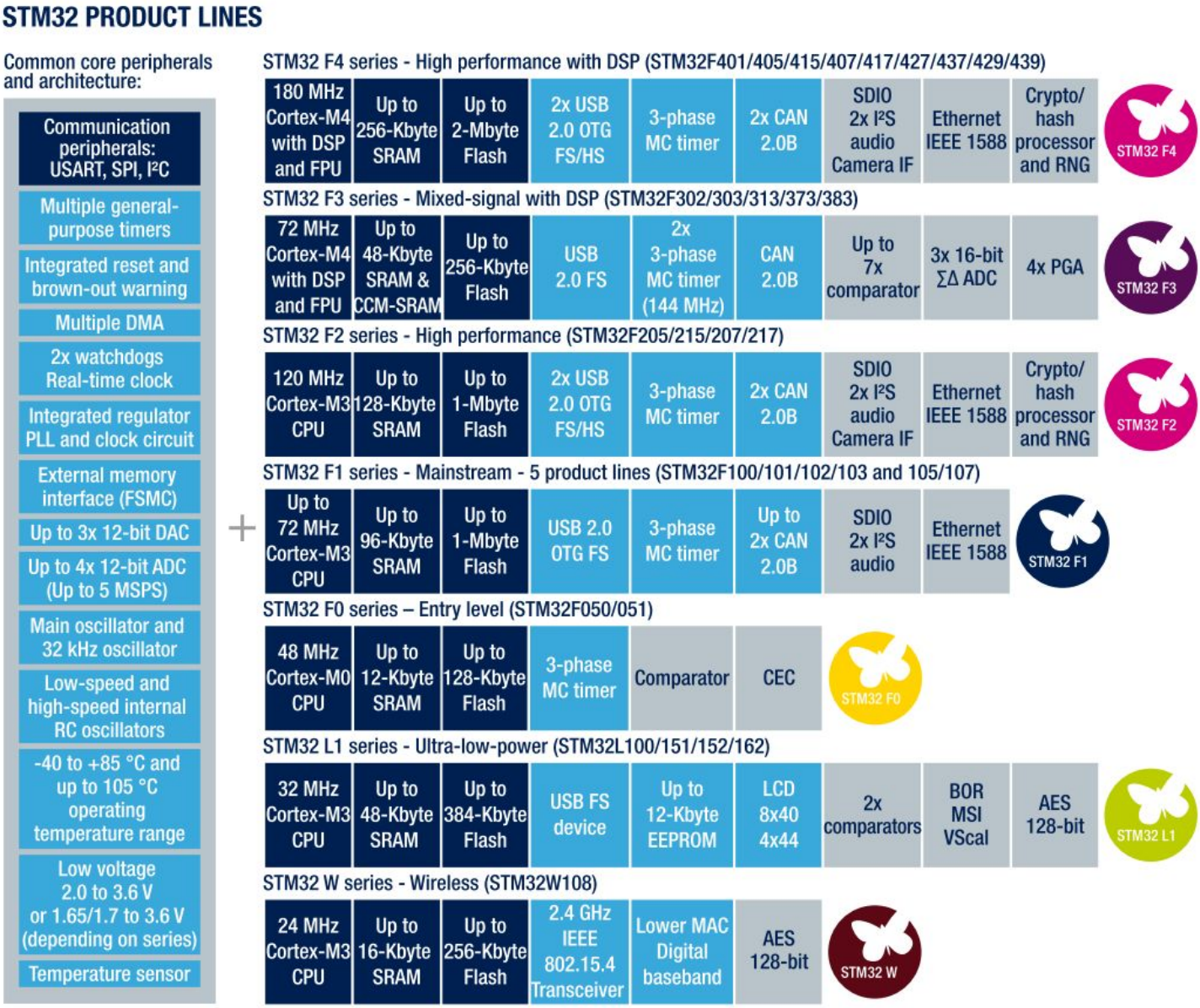


答案就是微型控制单元-MCU（Micro Control Unit），我们常说单片微型计算机，简称就是单片机，上面两张图就是两种不同类型的单片机。

如果你有组装过电脑，你应该知道，电脑想要运行起来除了CPU还要内存条、硬盘等设备才能正常运行，而单片机则将CPU、内存、蓝牙等外设集成到了一起，这一点也是CPU和MCU之间的主要区别。

二、单片机为什么有那么多种类？

在上面的单片机介绍中，小鱼放了两种单片机图片，事实上单片机的种类是非常之多的，用在洗衣机上的单片机、用在3D打印机的单片机和用在电暖气上的单片机都是不同的类型，那为什么有那么多分类呢？



答案是不同单片机具有不同的配置，比如不同的主频，慢的有几兆赫兹，快的有几百兆赫兹，不同的外设，比如有的支持USB、有的支持WIFI、有的支持蓝牙、还有的甚至多加了个处理器（比如小鱼卖的MicroROS学习板主控就是双核的）。

三、总结

本节简单的了解下单片机的硬件，但想让单片机实现我们想要的功能，我们还需要在单片机上运行我们的软件，此时就需要单片机开发平台了，下一节小鱼将要介绍在单片机上常用的几个开发平台。

2.微处理器开发平台

你好，我是爱吃鱼香ROS的小鱼。上一节我们对单片机硬件进行了简单介绍，但单片机运行不仅仅需要硬件，类似于电脑需要配套的操作系统一样，单片机还需要与之配套的软件，本节我们学习下常见的开发平台。

我们的MicroROS板采用的单片机是ESP32芯片，该芯片支持蓝牙和WIFI并且是双核的国产芯片，用途很广，所以小鱼就介绍下该芯片的常用的几个开发平台。

一、官方平台-ESPIDF(ESP IoT Development FrameWork)

官网地址：https://www.espressif.com/zh-hans/products/sdks/esp-idf



所谓官方平台就是单片机的厂商，针对单片机提供的开发框架，该框架为我们提供了一个C/C++ SDK，我们通过include相应的头文件就可以实现对硬件的控制。

下面这一段是官方介绍

> **ESP-IDF 是乐鑫官方的物联网开发框架，适用于 ESP32、ESP32-S、ESP32-C 和 ESP32-H 系列 SoC。它基于 C/C++ 语言提供了一个自给自足的 SDK，方便用户在这些平台上开发通用应用程序。ESP-IDF 目前已服务支持数以亿计的物联网设备，并已开发构建了多种物联网产品，例如照明、消费电子大小家电、支付终端、工控等各类物联网设备。**

ESP-IDF的核心其实是基于开源的FreeRTOS优化而来的，而FreeRTOS是一个迷你（几k大小）的实时操作系统内核，所以别看它小，照样跑了个操作系统。

展示一段ESP_IDF版本的HelloWorld，感受一下

```
/*
 * SPDX-FileCopyrightText: 2010-2022 Espressif Systems (Shanghai) CO LTD
 *
 * SPDX-License-Identifier: CC0-1.0
 */

#include <stdio.h>
#include <inttypes.h>
#include "sdkconfig.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"

void app_main(void)
{
    printf("Hello world!\n");
    vTaskDelay(1000 / portTICK_PERIOD_MS);
    fflush(stdout);
    esp_restart();
}
```

二、Arduino平台

官网地址：https://www.arduino.cc/en/about



Arduino是一款便捷灵活、方便上手的开源电子原型平台，本次MicroROS学习就是基于该平台进行开发。

展示一段Arduino版本的HelloWorld代码，感受一下

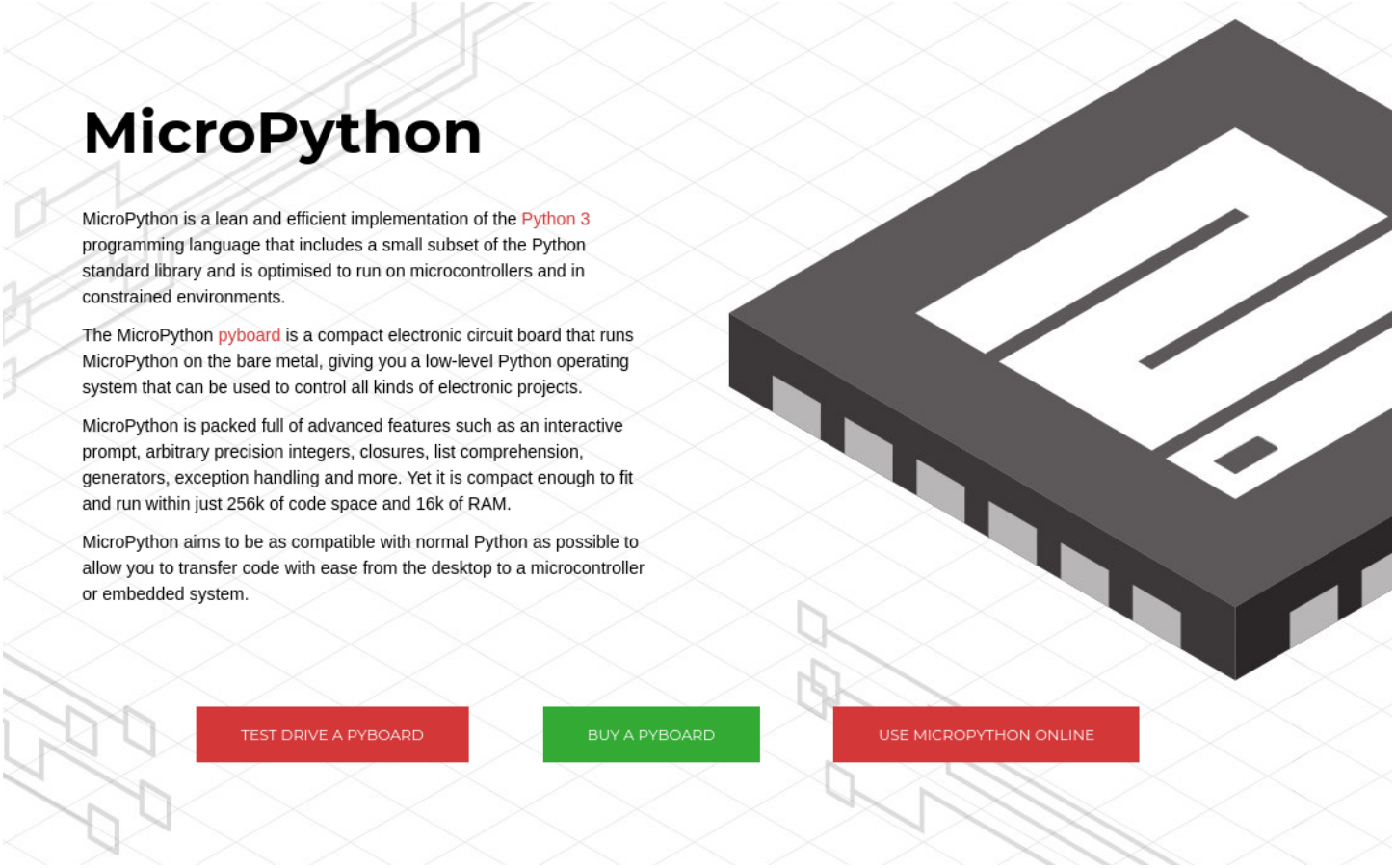
```
#include <Arduino.h>
void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
  Serial.println("Hello World!");
}
```

```
void loop() {
  // put your main code here, to run repeatedly:

}
```

三、MicroPython平台

官网地址：https://micropython.org/



大家都知道，人生苦短，我用Python，针对单片机平台，有没有可能使用Python开发呢？——MicroPython来了。

MicroPython是 Python 3 语言的精简实现 ，包括Python标准库的一小部分，经过优化可在微控制器和受限环境中运行。

同样的我们的MicroROS板同时也是支持使用MicroPython进行开发，只需要刷入相应的固件即可。

展示一段MicroPython的HelloWorld代码，感受下它的简单

```python
print('Hello, World!')
```

四、对比与总结

上面介绍了三种常见的平台，小鱼做个表格对比下三种平台的优缺点。

平台名称	优点	缺点
ESP_IDF	官方出品、测试完成度高、安全稳定、有官方支持、适合产品化、支持microROS	三方教程少、工程复杂、新手不友好
Arduino	社区庞大，教程丰富、新手友好，简单易用、支持microROS	封装较多
MicroPython	Python语言、简单易用	解释执行，效率低下，封装较多，不支持microROS

看完上面的对比，对于新手来说选择Ardunio平台容易入门且教程丰富，并且ESP32单片机是官方出品了Arduino支持（开源地址：https://github.com/espressif/arduino-esp32）。

了解完单片机开发平台，下一节我们正式搭建开发环境，然后开始编写我们的第一个HelloWorld工程！

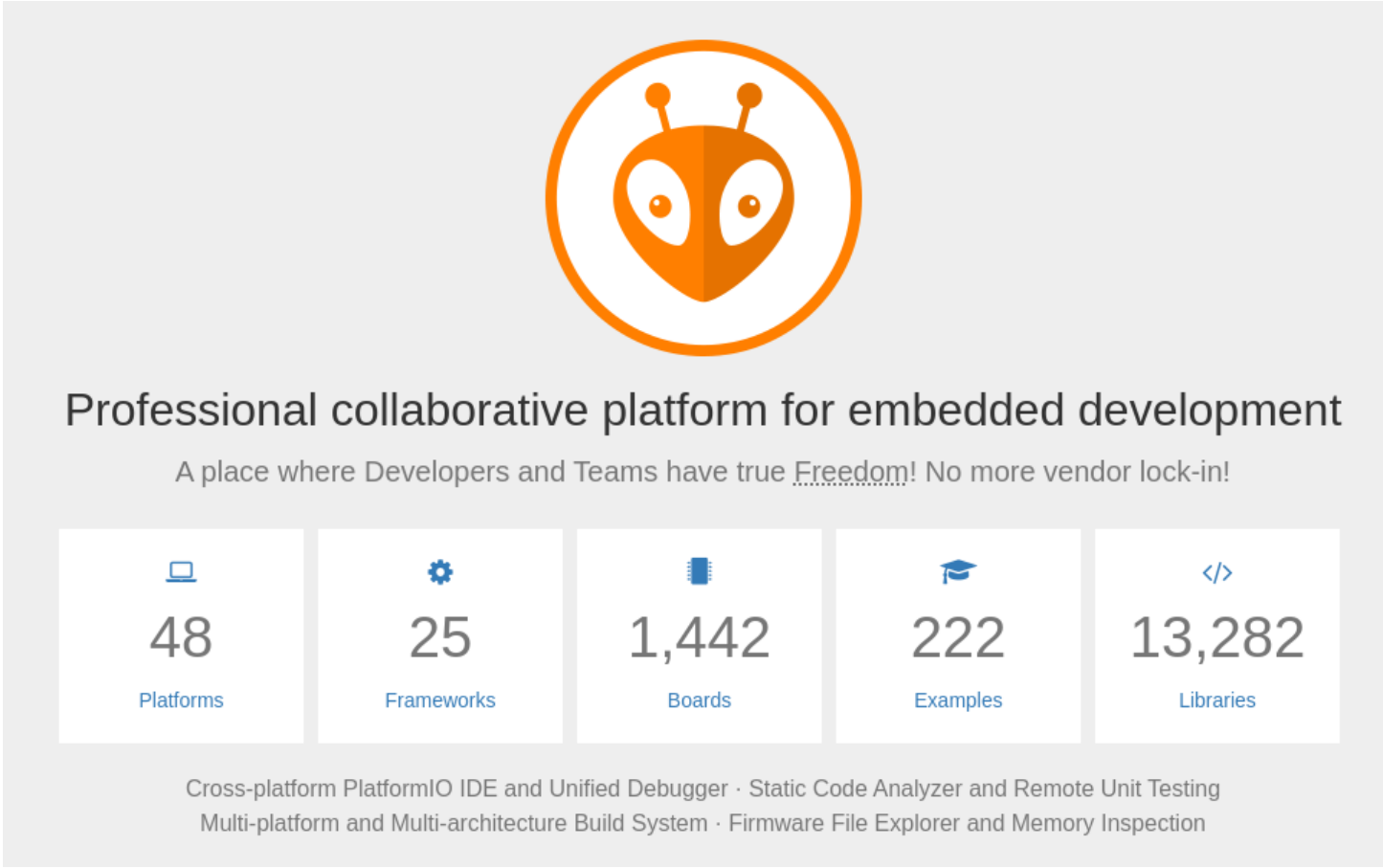
3.搭建PlateFormIO开发环境

你好，我是爱吃鱼香ROS的小鱼。本节我们开始介绍并搭建PlatformIO（以下简称PIO）的开发环境。因为网络原因，PIO搭建起来是一个挺困难的事情，但是小鱼在一键安装里添加了一键安装PIO，为你解决这一难题。

开始之前，小鱼想和你约定好本次学习之旅的开发环境和平台，小鱼这里采用的开发环境信息如下：

- 系统版本：Ubuntu 22.04（虚拟机实体机都可）
- ROS版本：ROS2-Humble
- 开发板：MicroROS学习板V1.0.0

一、PIO介绍



官网地址：https://platformio.org/

PIO是一个面向嵌入式开发的专业协作平台，它提供了一个适配VsCode的插件，它具有一个用户友好且可扩展的集成开发环境，具有一组专业开发工具，提供现代而强大的功能，以加快并简化嵌入式产品的创建和交付。

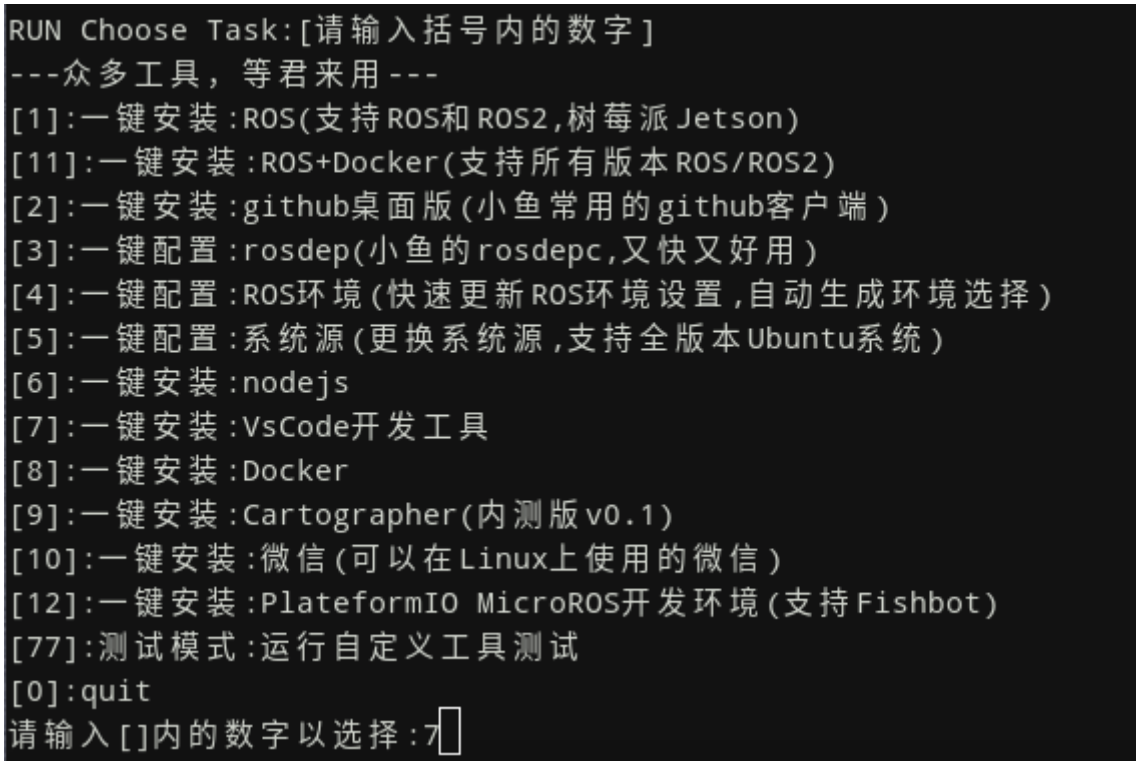
二、安装PIO

PIO的开发我们采用VsCode，所以在这之前你需要安装VsCode，VsCode安装可以使用一键安装，快速且稳定。

2.1 安装VsCode

运行一键安装指令，之后选择7即可

```
wget http://fishros.com/install -O fishros && . fishros
```



安装完成VsCode后，推荐你继续使用一键安装来安装PIO。

2.2 安装PIO（可跳过）

注意这一步仅适用Ubuntu22.04系统，非该系统请直接跳过

继续使用一键安装，选项12

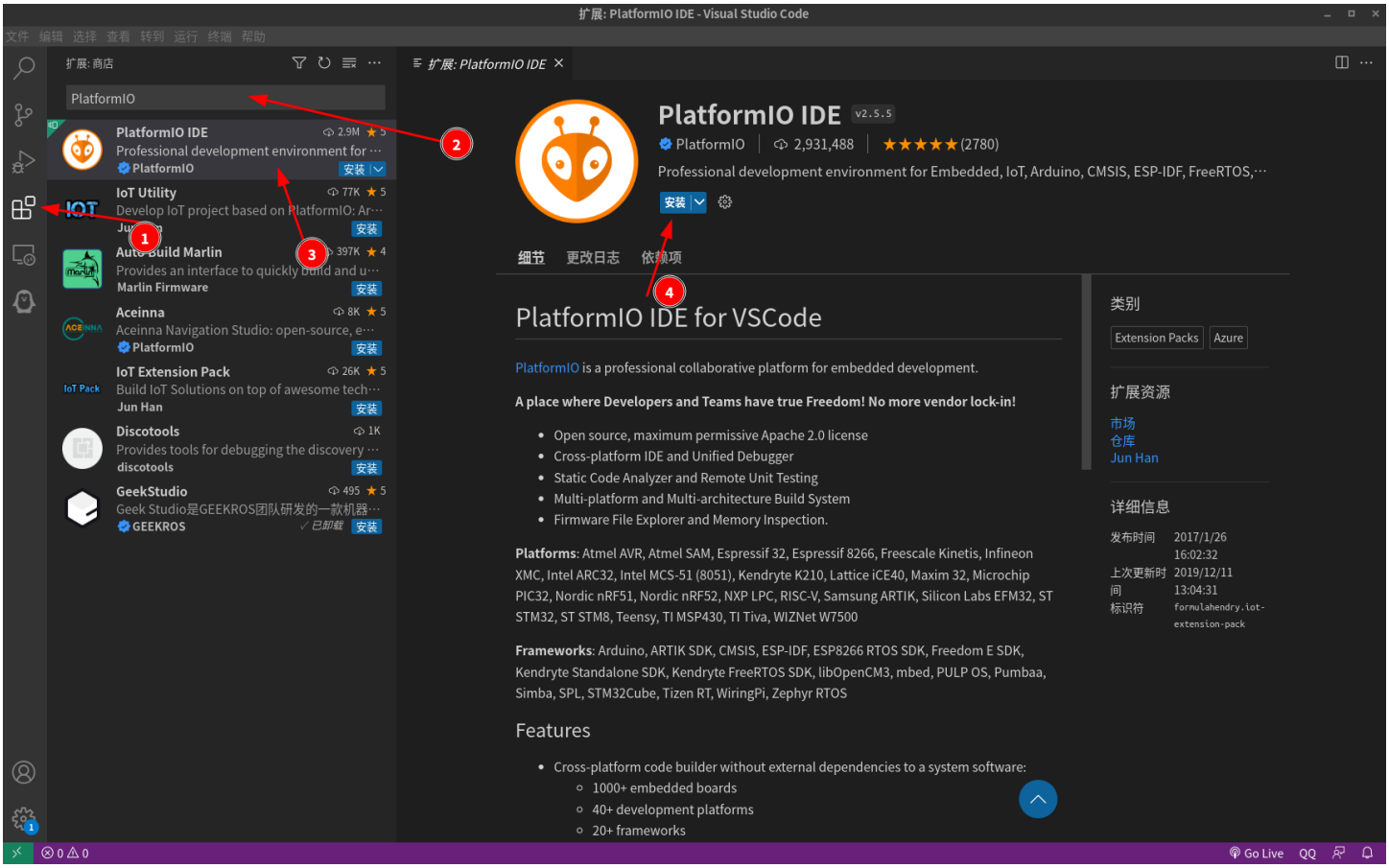
```
wget http://fishros.com/install -O fishros && . fishros
```

```
[8]:一键安装:Docker
[9]:一键安装:Cartographer(内测版v0.1)
[10]:一键安装:微信(可以在Linux上使用的微信)
[12]:一键安装:PlateformIO MicroROS开发环境(支持Fishbot)
[77]:测试模式:运行自定义工具测试
[0]:quit
请输入[]内的数字以选择:12
```

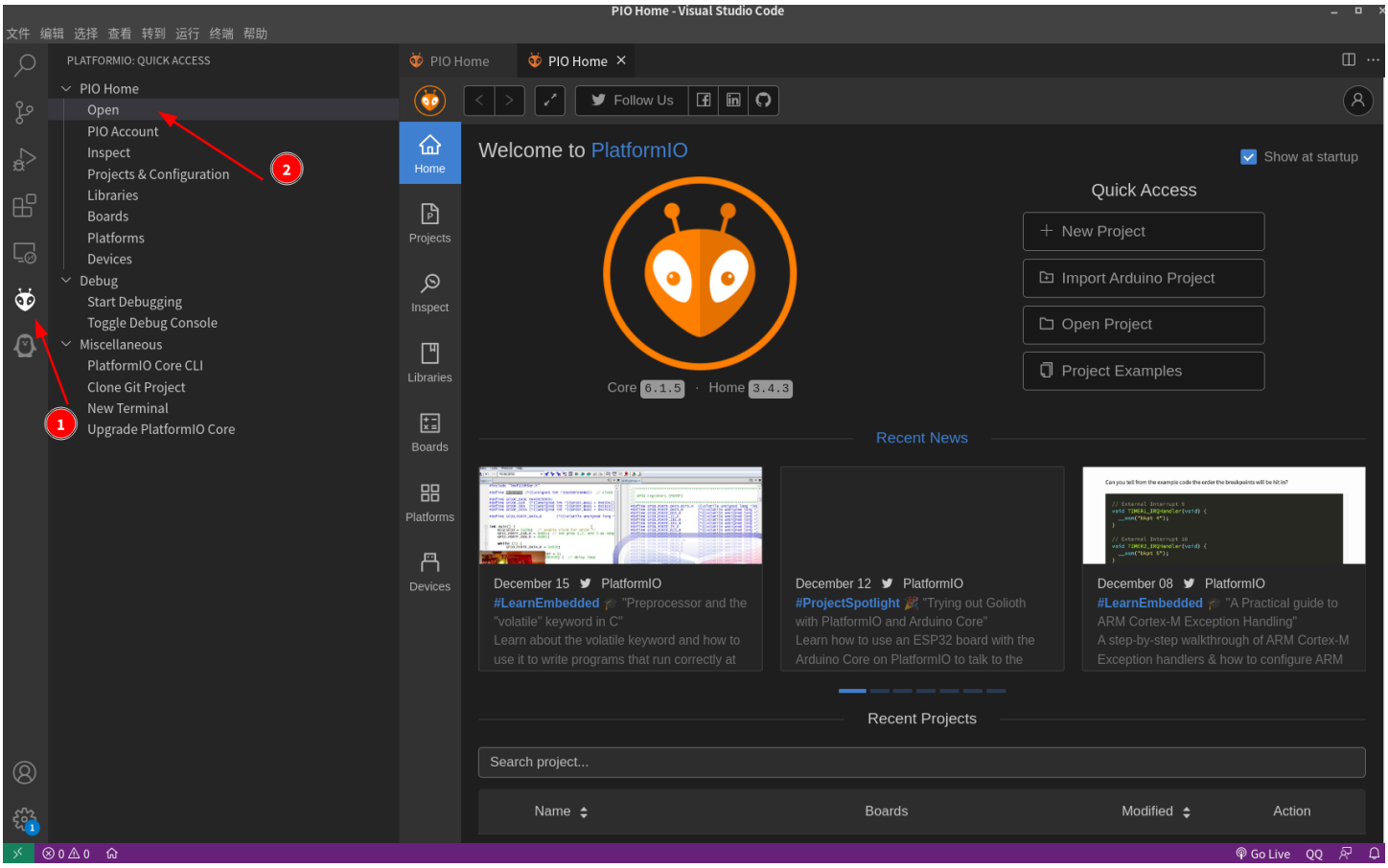
2.3 安装VsCode插件

打开VsCode，我们还需安装一个PlatformIO插件就能正常Work了。

任意终端输入code，或者在菜单中找到vscode都可以打开vscode，接着在扩展中搜索 `PlatformIO` ，选择后点击安装即可。



安装完成后在侧方栏应该可以看到一个蚂蚁头的图标，这个就是PIO，点击图标，点击上访的Open就可以打开PIO HOME。



三、新建工程测试

接着我们就可以利用PIO建立第一个Arduino工程进行测试，首先点击New Project。



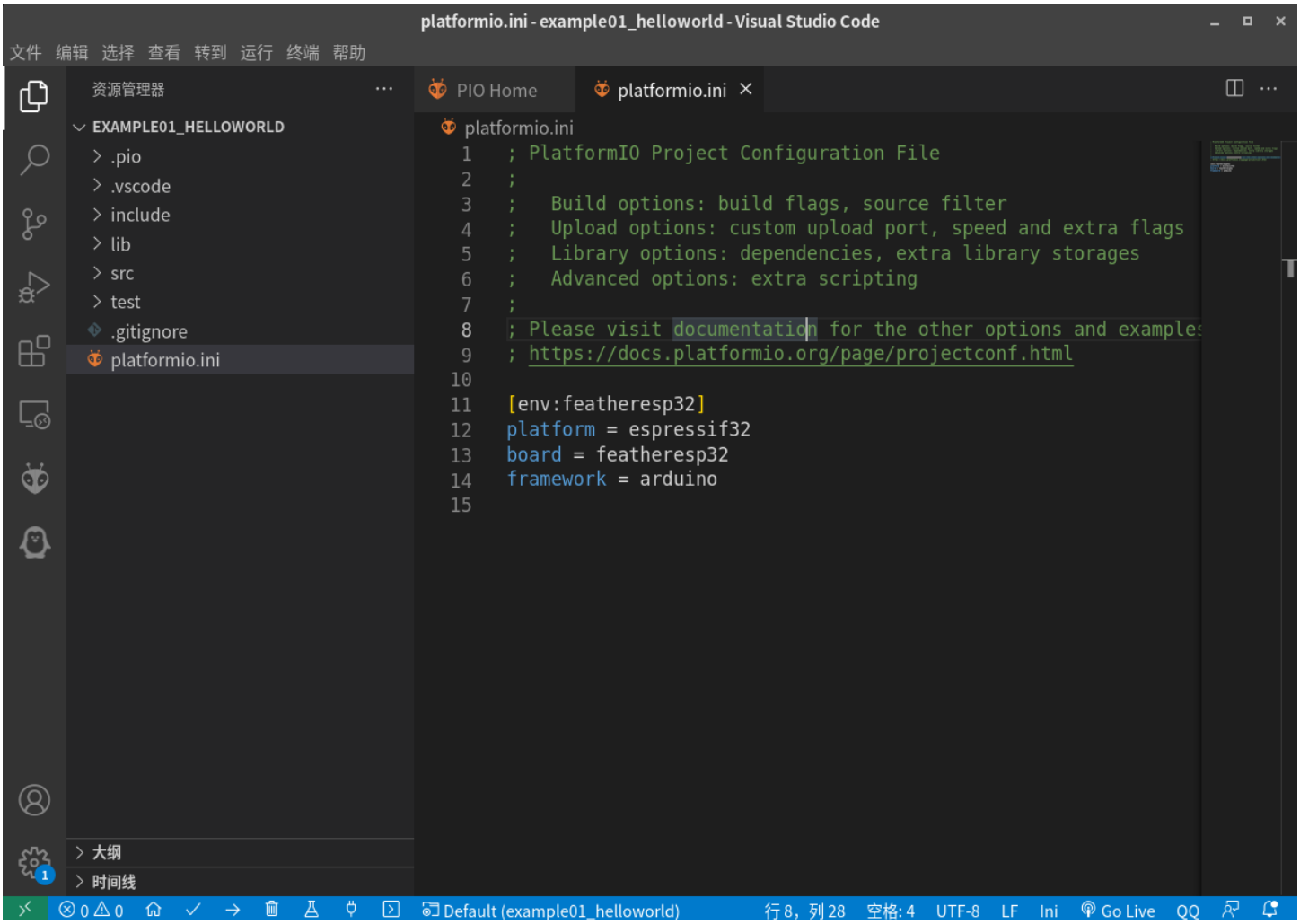
四步新建工程

1. 输入工程名 `example01_helloworld`
2. 选择开发板，这里选择 `Adafruit ESP32 Feather`

3. 选择开发框架，这里我们用Arduino，PIO还支持IDF（IoT Development FrameWork）
4. 开发位置可以选择默认的位置，也可以自定义位置



最后点击 `Finish` 即可，这样我们就得到了一个支持ESP32的Arduino工程。



四、总结

本节我们学习通过VsCode安装PIO并新建一个Arduino工程，下一节小鱼带你一起了解下这个工程文件并学习如何编译工程。

4.PIO工程结构&构建方式

你好，我是爱吃鱼香ROS的小鱼。上一节我们搭建好了PIO的开发环境，并新建了第一个工程。本节我们详细了解下该工程，了解我们需要在哪里写代码，在哪里改配置。

一、工程目录概述



首先展开工程，可以看到工程一共有8个部分如上图所示。

1. PIO配置文件
2. VsCode配置文件
3. **头文件放置目录**
4. **库文件放置目录**
5. **代码资源放置目录，主函数就在这里**
6. 测试文件放置目录
7. git忽略文件
8. **platformio配置文件**

二、在哪里写代码？

打开 `src/main.cpp` 就是我们工程的程序入口文件，打开该文件，已经默认给我们生成了9行代码，后续的主要开发就在这里进行。

```c++
#include <Arduino.h>

void setup() {
  // put your setup code here, to run once:
}

void loop() {
  // put your main code here, to run repeatedly:
}
```

三、工程配置文件-platformio.ini

打开工程主目录下的 `platformio.ini` 文件，预生成的配置文件如下

```ini
; PlatformIO Project Configuration File
;
;   Build options: build flags, source filter
;   Upload options: custom upload port, speed and extra flags
;   Library options: dependencies, extra library storages
;   Advanced options: extra scripting
;
; Please visit documentation for the other options and examples
; https://docs.platformio.org/page/projectconf.html

[env:featheresp32]
platform = espressif32
board = featheresp32
framework = arduino
```

这里用到的主要配置有四个

1. `[env:featheresp32]` 编译环境
2. `platform = espressif32` ，单片机平台
3. `board = featheresp32` ，开发板
4. `framework = arduino` ，开发框架-arduino

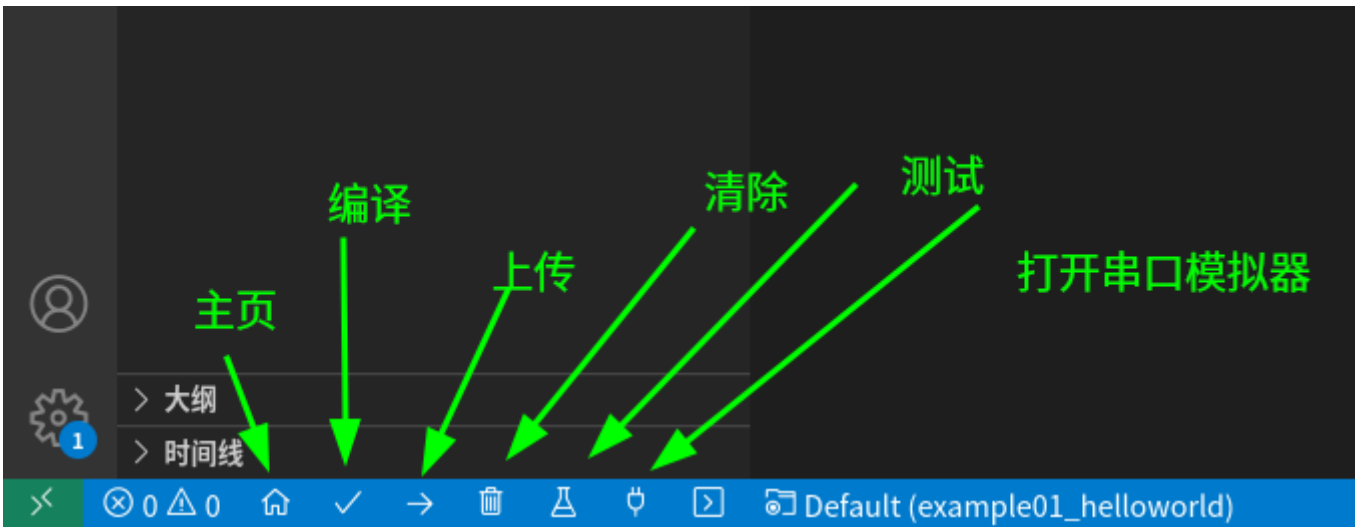
后续还有很多关于工程的配置都放在这里，同时我们可以添加一条配置 `board_build.f_cpu = 240000000L` ，将单片机的主频提高到 `240MHZ` 的主频。

```ini
[env:featheresp32]
platform = espressif32
board = featheresp32
framework = arduino
board_build.f_cpu = 240000000L
```

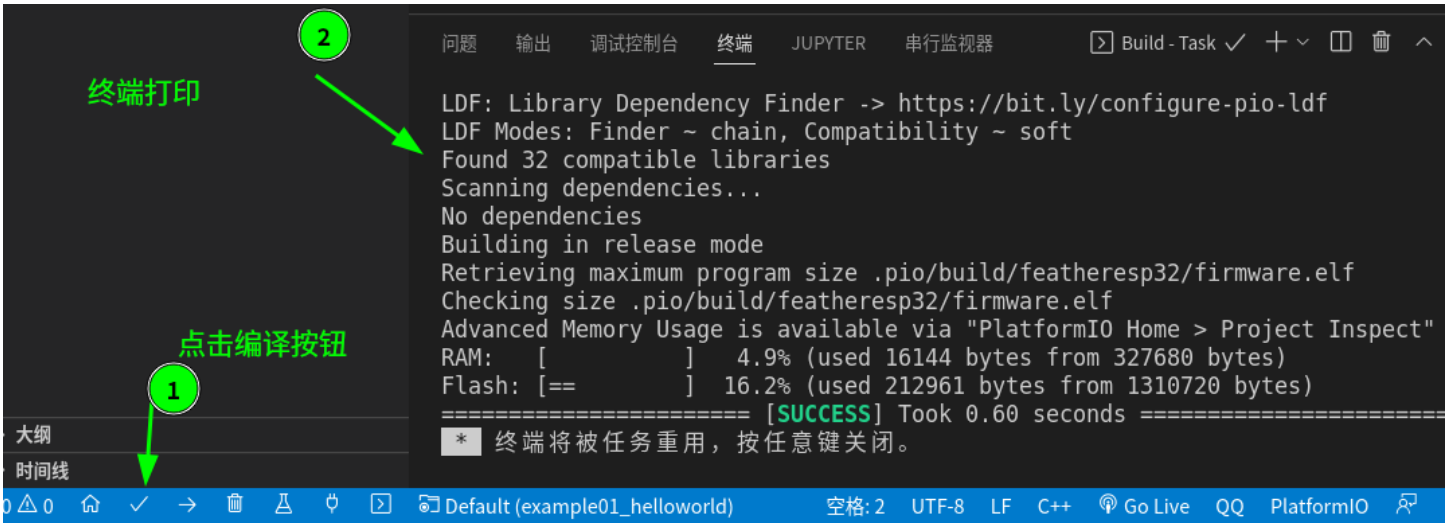
四、编译工程

在VsCode中编译PIO，编译工程和将编译结果下载到开发板上都非常的方便。

编译工程可以手动点击左下角的对号进行，其他操作也可以通过按钮进行。



点击编译按钮，看到如下界面则代表编译成功



其中打印信息有很多有用的提示，比如工程占用的RAM和Flash大小（可以理解为系统程序大小）

```
RAM:   [          ]   4.9% (used 16144 bytes from 327680 bytes)
Flash: [==        ]  16.2% (used 212961 bytes from 1310720 bytes)
```

编译完成工程，在 `.pio/build/featheresp32` 目录下可以看到 `firmware.bin` ，这个就是我们工程编译之后生成的二进制文件，将该文件下载到开发板上就可以运行了。

五、PIO快捷键

这里再介绍几个PIO的快捷键，在接下来的学习中你肯定能用到

快捷键	内容
Ctrl+Alt+B	编译工程
Ctrl+Alt+U	将程序上传烧录到开发板
Ctrl+Alt+S	打开串口Monitor

六、总结

本节我们简单的了解下PIO工程的结构以及配置文件，下一节我们开始学习如何输出 `Hello World!`

5.第一个HelloWord工程

你好，我是爱吃鱼香ROS的小鱼。这一节我们正式编写代码，输出HelloWorld到电脑上。在正是开始编写代码前，我们先了解下开发流程。

一、Arduino开发流程

Arduino和其他单片机开发，一共分为四步。

1. 编写代码，根据相关的API和SDK进行代码的编写。
2. 编译工程，将工程的代码文件编译成二进制文件。
3. 烧录二进制文件，将上一步生成的二进制文件通过工具烧录到开发板中。
4. 运行测试，重启开发板，观察硬件执行情况（数据打印一般通过串口查看）

接下来小鱼就按照上面总结的几个步骤来尝试编写HelloWorld!

二、编写代码

2.1 Arduino函数介绍

前面有介绍，Ardunio平台的一大特点就是简单易用，而Ardunio使用的开发语言是C/C++，从工程生成的默认代码就可以看出来。

```C++
#include <Arduino.h>

void setup() {
  // put your setup code here, to run once:
}

void loop() {
  // put your main code here, to run repeatedly:
}
```

整个代码可以分为三个部分

1. 头文件 `#include <Arduino.h>`

2. `setup()` 函数，该函数只会在启动时被系统调用一次
3. `loop()` 函数，该函数会被系统循环调用，直到重启或者断电

2.2 为什么没有入口函数main函数？

在学习C语言和C++时你应该学过，程序的入口文件是main函数，但在这个Arduino中却没有main函数的存在，这是为什么？

Arduino其实是有main函数的，Arduino的main函数长这样（ESP32-Arduino库有所不同，但原理一样：https://github.com/espressif/arduino-esp32/blob/master/cores/esp32/main.cpp）

```C++
#include <Arduino.h>

// Declared weak in Arduino.h to allow user redefinitions.
int atexit(void (* /*func*/ )()) { return 0; }

// Weak empty variant initialization function.
// May be redefined by variant files.
void initVariant() __attribute__((weak));
void initVariant() { }

void setupUSB() __attribute__((weak));
void setupUSB() { }

int main(void)
{
    init();

    initVariant();

#if defined(USBCON)
    USBDevice.attach();
#endif

    setup();

    for (;;) {
        loop();
        if (serialEventRun) serialEventRun();
    }
```

```
    return 0;
}
```

核心的代码就这一段

```c++
    setup();

    for (;;) {
        loop();
        if (serialEventRun) serialEventRun();
    }
```

从这里就可以看出来，setup和loop函数之间的关系，在main函数中先调用一次setup函数，再使用for死循环调用loop函数。

2.3 串口输出HelloWorld

要实现将 `HelloWorld!` 从开发板输出到电脑上，我们需要了解一个常用的通信协议 `Serial` ，常称串口通信。

关于串口通信的原理可以到B站搜索相关视频，但在这里使用时你只需要了解如何使用即可。

这里我们了解三个函数，串口初始化、串口打印、串口读取。

函数原型	参数	返回值	描述
void begin(unsigned long baud)	baud：串口波特率	void	该函数用于初始化串口，主要配置串口波特率，波特率类似于频道号，串口收发双方保持相同的波特率才能进行正常通信。常见的波特率有9600，115200等，波特率其实代表每秒数据手法的频率，波特率越高，速度越快。
size_t printf(const char *format, ...)	format：格式化字符串	size_t 打印的字符数量	该函数和我们常见的printf函数一致，eg：`Serial.printf("Hello World!");`
int read(void)	void	int 读取的字符值，ASSIC表示	该函数用于读取一个字节的数据，返回值就是这个字节的值，如果没有数据则返回 `-1`

基于上面的函数，我们可以这样输出HelloWorld!

```cpp
#include <Arduino.h>

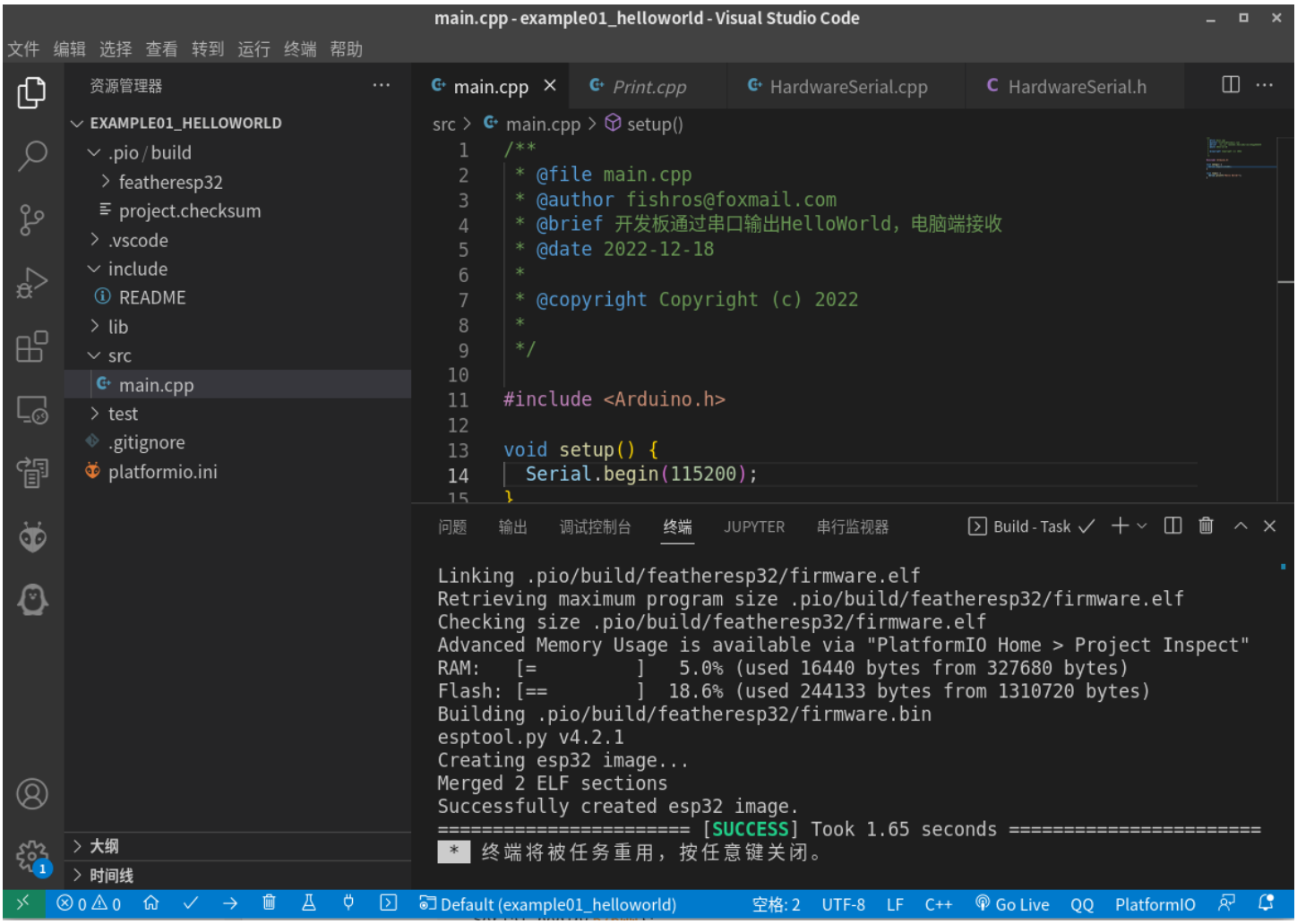
void setup() {
  Serial.begin(115200);
}

void loop() {
  Serial.printf("Hello World!\n");
}
```

在setup()函数里进行串口的初始化，**波特率设置成了115200**，在loop函数中不断的输出`Hello World!`。

三、编译代码

点击对号，或者使用快捷键`Ctrl+Alt+B`，即可编译。



看到 `Building .pio/build/featheresp32/firmware.bin` 和 `Successfully created esp32 image.` 就代表已经成功生成了二进制文件，下一步我们就开始烧录二进制文件。

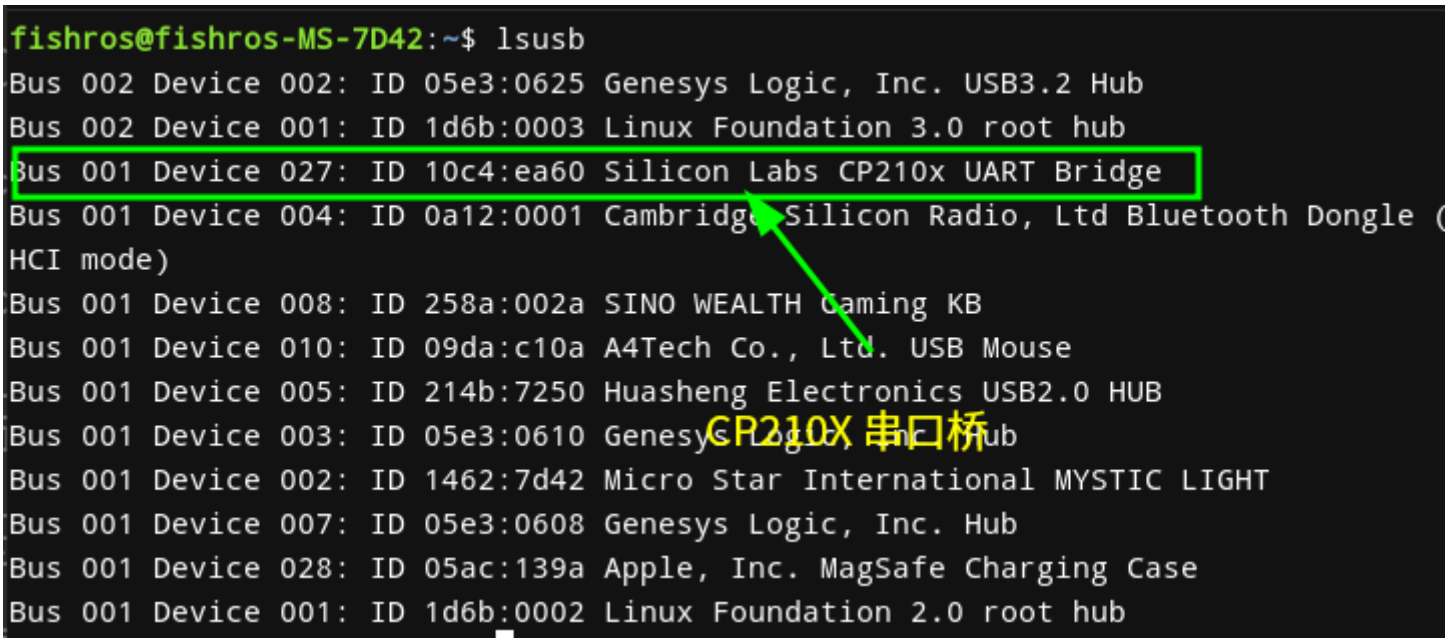
四、烧录二进制文件

4.1 连接开发板到电脑

MicroROS学习板采用TypeC接口，你需要一个USB数据线将开发板连接到你的电脑。连接电脑后，Linux系统驱动会被自动搜索和加载，查看是否有正确驱动，可以使用lsusb进行测试。

```
lsusb
```

输入后，如果可以看到CP210x这个设备，就代表驱动加载成功了



驱动加载成功后在/dev目录下会多出一个ttyUSBx的设备，别如小鱼这里就是/dev/ttyUSB0

使用 `ls /dev/ttyUSB*` 指令可以将其列出

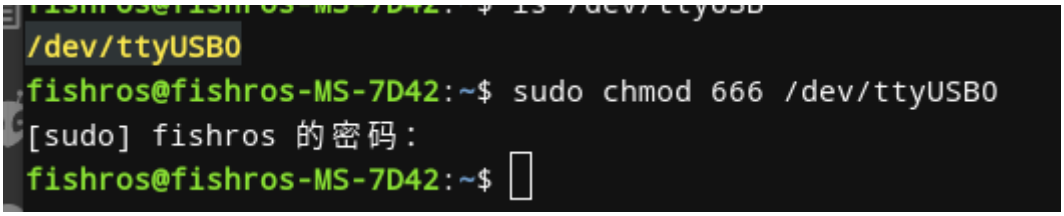


4.2 设置设备权限

我们想让开发板和电脑通过串口进行通信，电脑端只需对这个串口进行读写就行了。因为设备默认的生成目录是在/dev目录下，普通用户是没有读写权限的，所以在使用之前我们可以修改下该设备的权限。

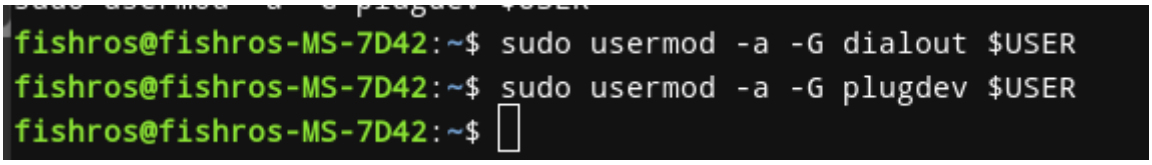
临时修改

```
sudo chmod 666 /dev/ttyUSBx
```



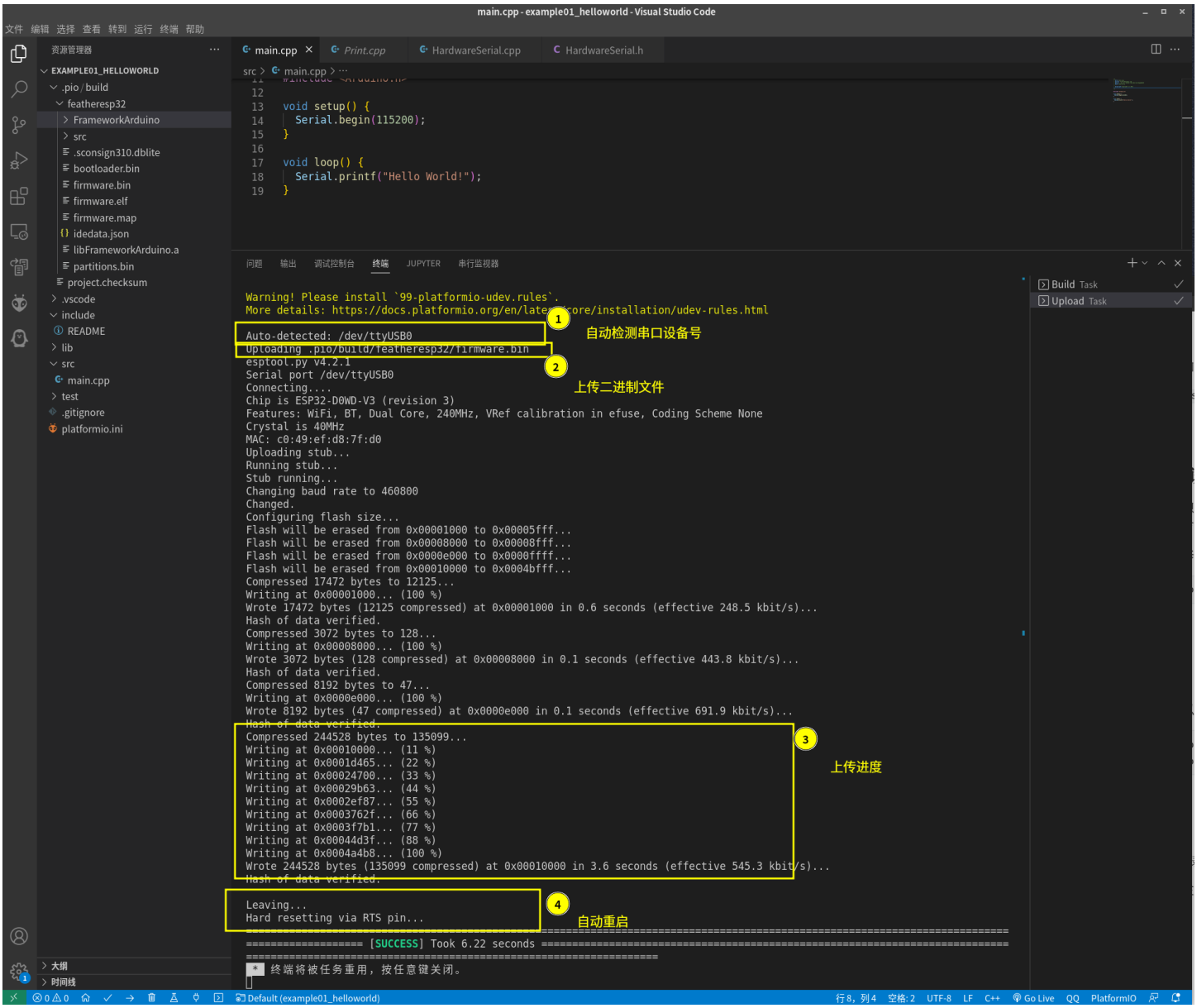
也可以永久修改，将用户添加到dialout和plugdev组（重启后方生效）

```
sudo usermod -a -G dialout $USER
sudo usermod -a -G plugdev $USER
```



4.3 烧录二进制文件

点击左下角的上传烧录按钮，或者使用快捷键 `Ctrl+Alt+U` 进行烧录。



看到上面四部分打印代表烧录成功，可以看到PIO可以自动检测串口并进行连接，接着上传文件到开发板，最后自动重启。

五、运行测试

因为在下载完成后，下载程序帮我们自动重启了，所以这里我们不需要进行重启。接着我们使用串口Monitor打开串口看看有没有数据。

点击Serial Monitor按钮，或者使用快捷键 `Ctrl+Alt+S` ，如果没有出错，你将看到下面的乱码。

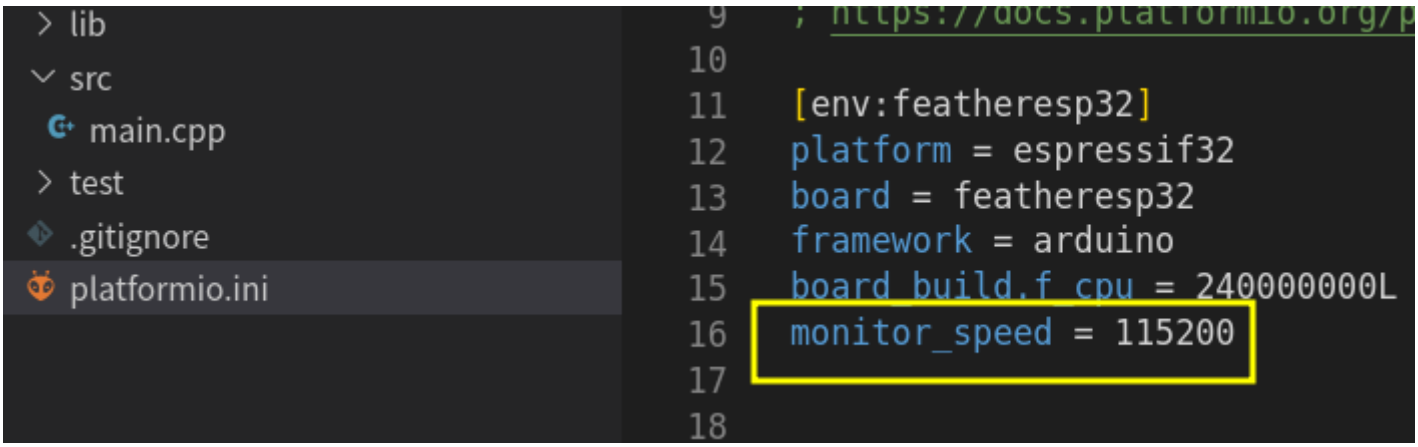
原因是终端的波特率不对，开发板发送给电脑数据的波特率是115200，而电脑接手的波特率是9600，不匹配就会造成乱码。



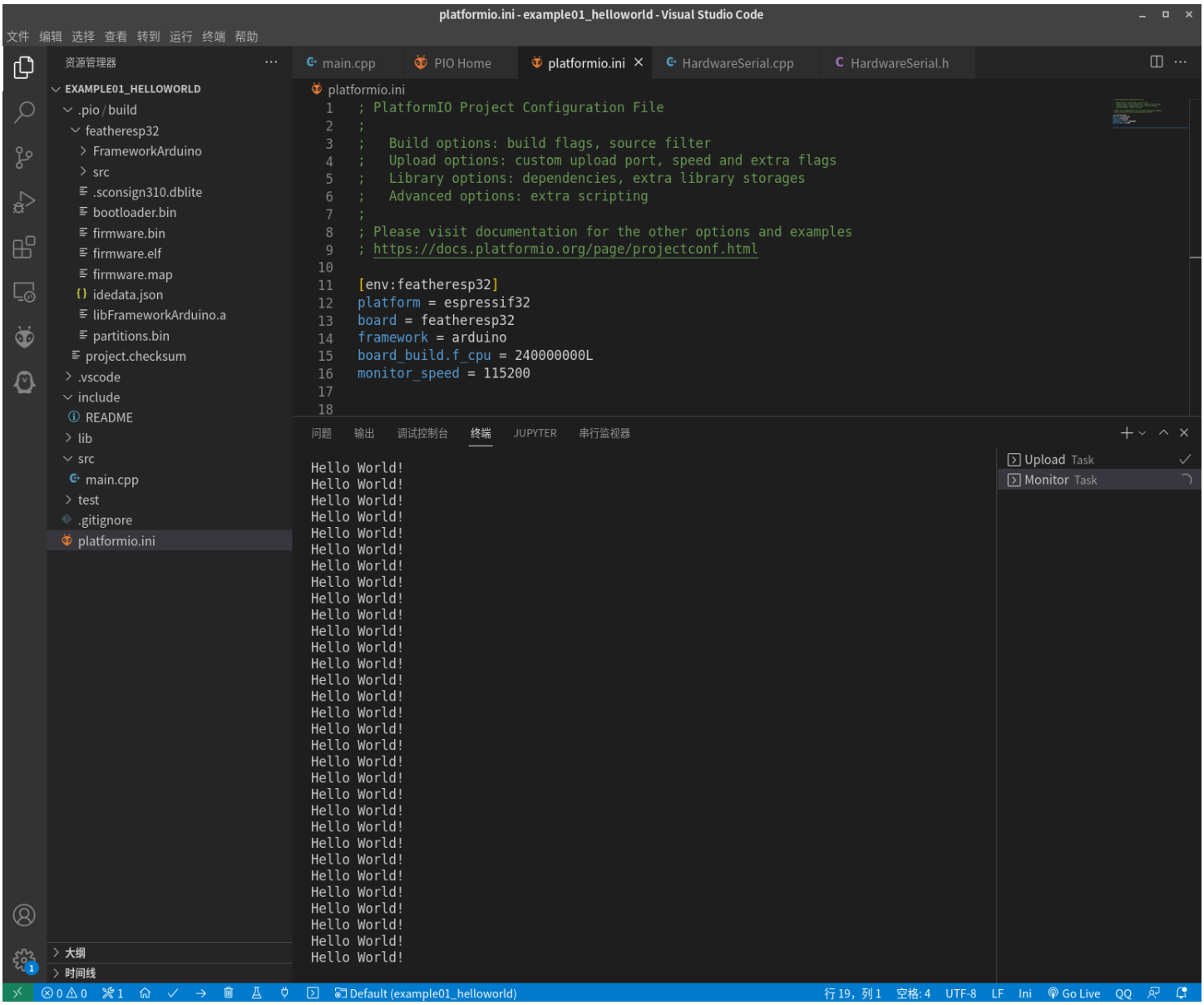
通过修改配置文件，可以修改Serial Monitor的默认波特率。

在 `platformio.ini` 中添加一行代码

```
monitor_speed = 115200
```



接着关闭刚刚的终端，再重新打开，接着我们就可以看到嗖嗖嗖的 `Hello World!`



六、总结

本节我们成功将自己的代码上传到开发板上了，然后通过串口成功的和单片机建立了单向连接（开发板向电脑发送数据），下一节我们学习下电脑向单片机发送消息。

最后还有一个几个小作业

1.上面我们输出 `Hello World!` 在不断的输出，如果想要改成只输出一次，代码该怎么写？

答案：

```c++
#include <Arduino.h>

void setup() {
```

```
  Serial.begin(115200);
  Serial.printf("Hello World!\n");
}

void loop() {
}
```

2.上面我们输出`Hello World!`在快速的输出，如果想要改成每秒输出一次，代码该怎么写？

提示函数：`void delay(uint32_t ms)`延时指定ms。

答案：

```c++
#include <Arduino.h>

void setup() {
  Serial.begin(115200);
}

void loop() {
  delay(1000);
  Serial.printf("Hello World!\n");
}
```

6.串口通信-接收实验

你好，我是爱吃鱼香ROS的小鱼。上一节我们完成了第一个Hello World工程，学习使用了串口模块的初始化和发送，本节我们再来一个串口接收小实验，把串口收发数据补齐。

一、检测并接收单个字符

1.1 代码编写

```c++
/**
 * @file demo01_read_byte.cpp
 * @author fishros@foxmail.com
 * @brief 初始化串口，当有数据过来的时候读取并将数据打印出来
 * @version 0.1
 * @date 2022-12-18
 *
 * @copyright Copyright (c) 2022
 *
 */
#include <Arduino.h>

void setup()
{
    // 初始化串口
    Serial.begin(115200);
}

void loop()
{
    // 判断是否有有效数据，返回值是有效数据的长度
    if (Serial.available())
    {
        // 读取一个数据
        int c = Serial.read();
        // -1 代表接收失败
        if (c != -1)
        {
            // 以%c字符的格式输出接收的数据
```

```
                Serial.printf("I receve %c\n", c);
            }
        }
    }
```

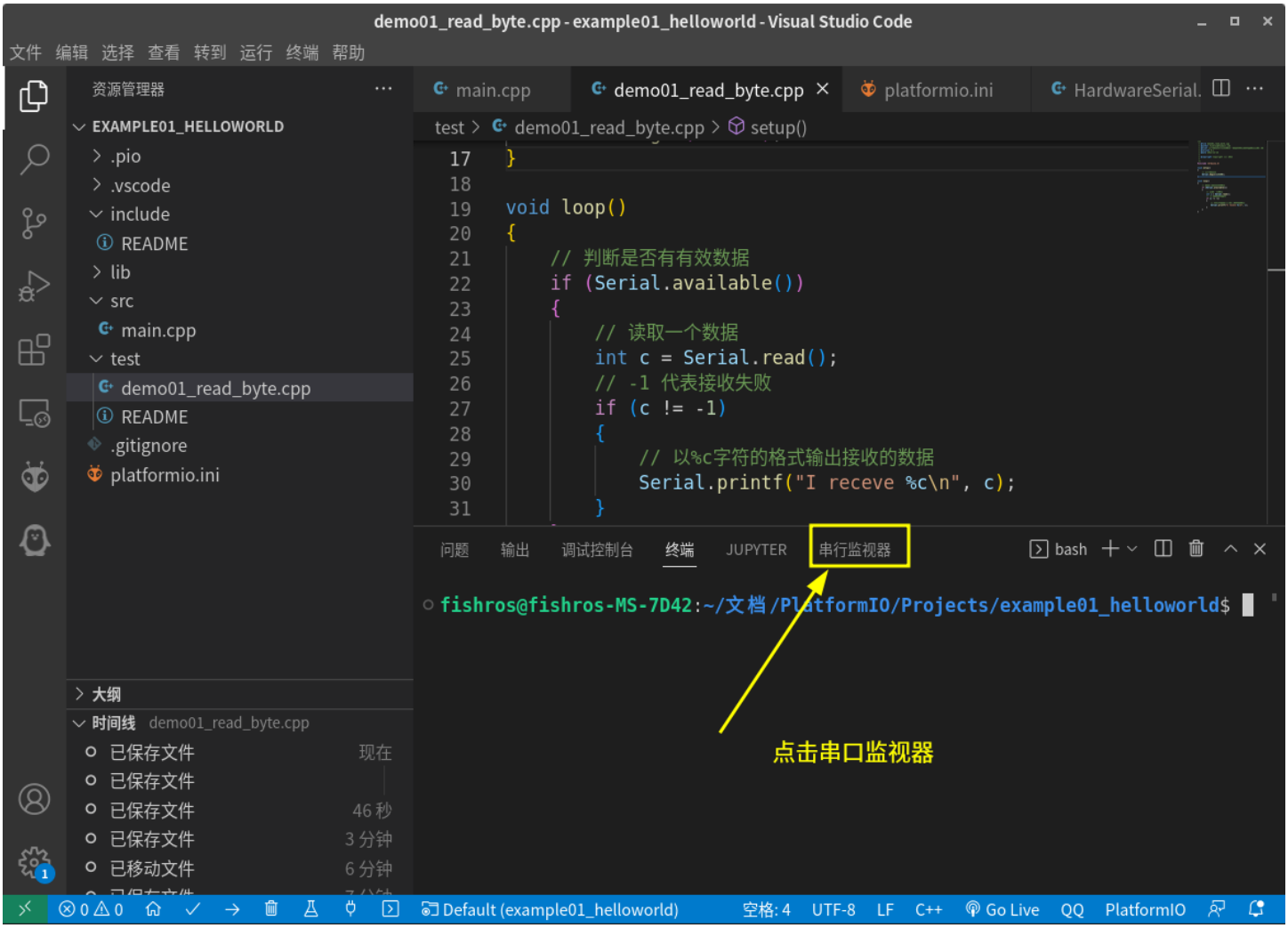
这里多用了一个函数 `Serial.available()` ，该函数代表当前串口中缓存有效数据的长度。

1.2 使用串口监视器发送消息

使用快捷键，编译 `Ctrl+Alt+B` 、上传 `Ctrl+Alt+U` ，接着准备发送数据

因为要发送消息，这里我们换一个收发分离的串口监视器来进行数据发送

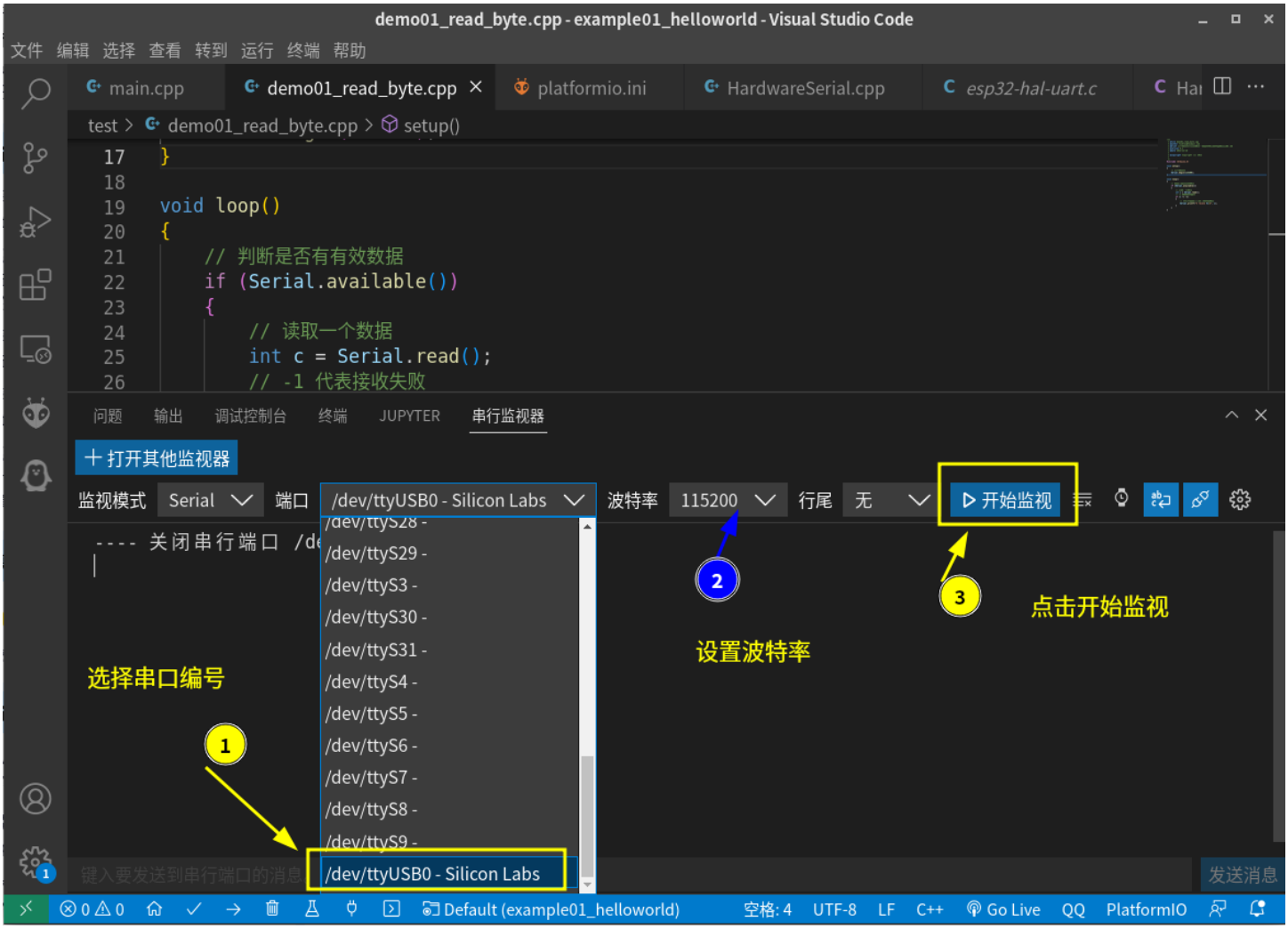
使用 `Ctrl+Alt+~` 打开终端，接着在终端中你可以看到串口监视器一栏



接着打开我们板子对应的串口设备

- 选择串口编号

- 设置波特率
- 点击开始监视



发送测试

- 输入数据
- 点击发送
- 查看返回



尝试发送12



可以看到受到了两条返回，这是因为我们每次只接收一个数据，所以即使发送 `12` ，接收数据也是一个一个接收和打印的。

那有没有办法一次性接收多个数据呢？我们换个函数即可。

二、一次性接收一串数据

2.1 代码编写

```c++
/**
 * @file demo01_read_byte.cpp
 * @author fishros@foxmail.com
 * @brief 初始化串口，当有数据过来的时候读取并将数据打印出来
 * @version 0.1
 * @date 2022-12-18
 *
 * @copyright Copyright (c) 2022
```

```
 *
 */
#include <Arduino.h>

void setup()
{
    // 初始化串口
    Serial.begin(115200);
}

void loop()
{
    // 判断是否有有效数据
    if (Serial.available())
    {
        // 读取一个String字符串数据
        String str = Serial.readString();
        // 以%s的格式输出接收的数据
        Serial.printf("I receve %s\n", str.c_str());
    }
}
```

2.2 编译下载

点击按钮或者使用快捷键编译下载代码。

如果你在下载代码时遇到下面的错误，是因为刚刚的串口监视器没有关闭，

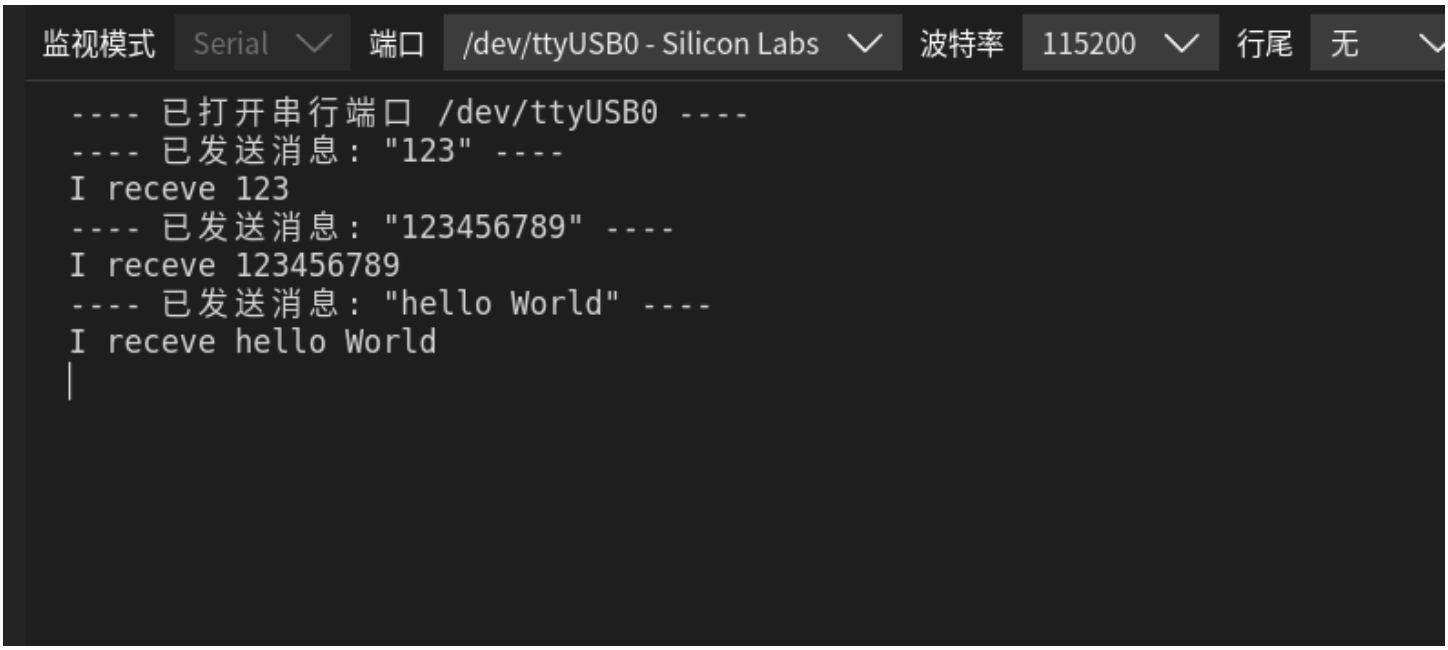
```
Auto-detected: /dev/ttyUSB0
Uploading .pio/build/featheresp32/firmware.bin
esptool.py v4.2.1
Serial port /dev/ttyUSB0
Connecting.....
serial.serialutil.SerialException: device reports readiness to read but returned
no data (device disconnected or multiple access on port?)
*** [upload] Error 1
```

点击停止监视后，继续下载即可



2.3 测试

下载完成后，重新打开串口，接着发送一串消息



三、总结

本节我们通过两个串口接收数据小实验，学习了串口数据的接收和发送。

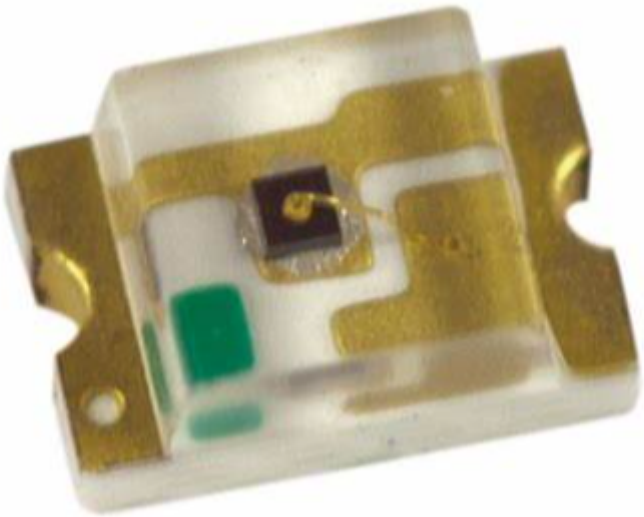
1.点灯基础-看懂LED驱动电路

你好，我是爱吃鱼香ROS的小鱼。上一节我们学习了单片机的开发流程，接下来我们就尝试通过代码来控制LED灯的开关，但是在正式写代码前，你需要先学会如何看电路图。

小鱼提示：

在学习下面的内容前，需要你了解基础的电路基础知识（电流、电压和电阻，比如串联分压，并联分流），简单复习下初中物理知识即可。

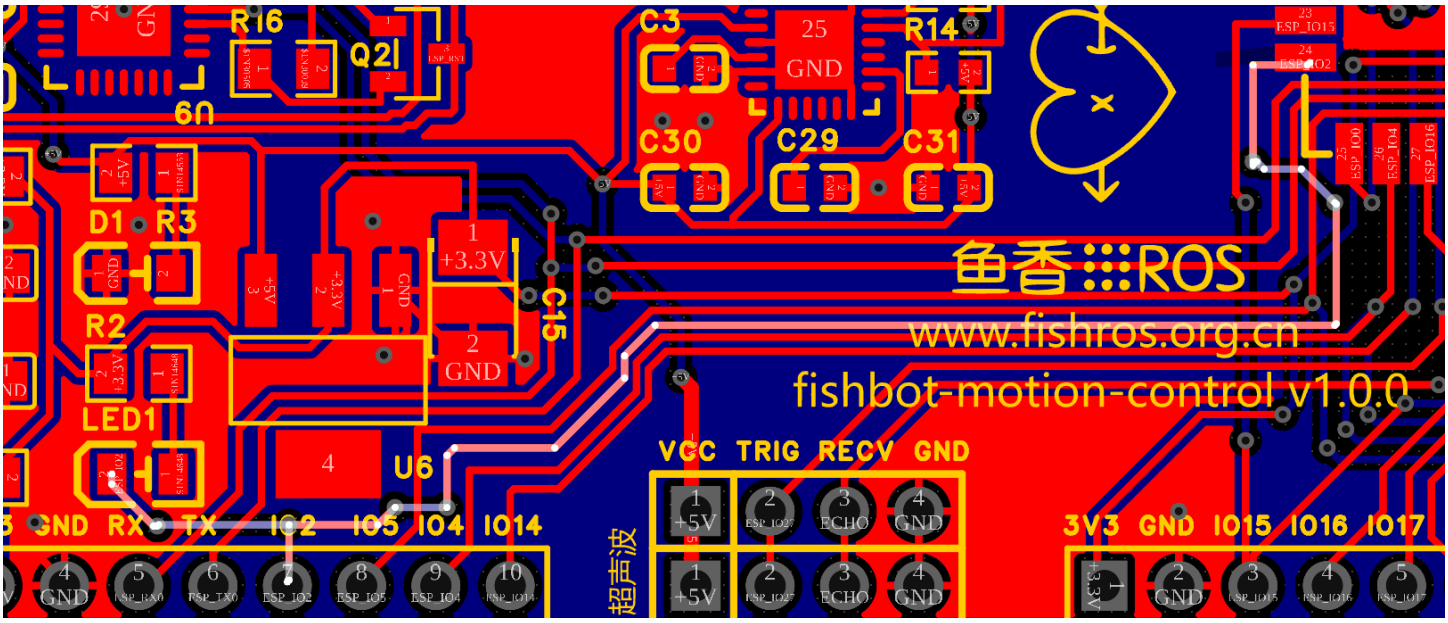
一、LED介绍

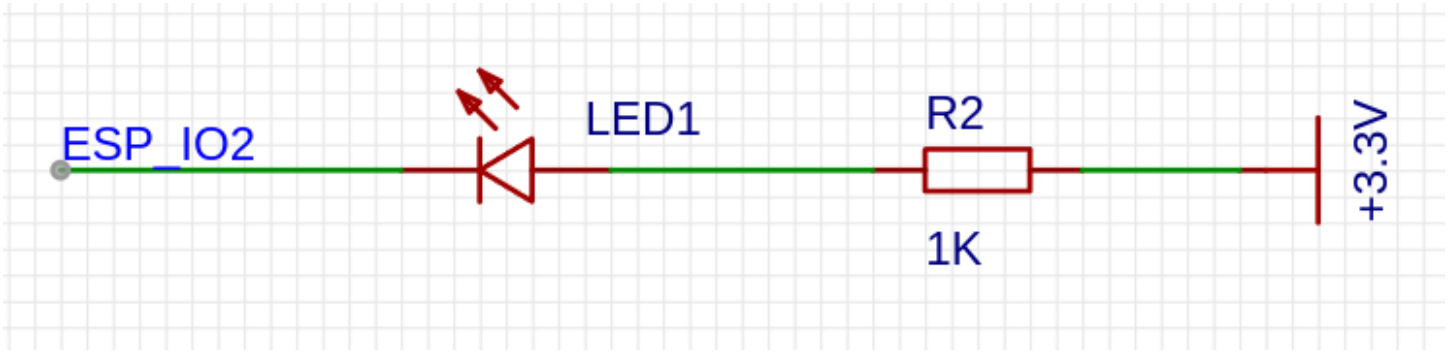


LED（Light Emitting Diode），发光二极管，是一种能够将电能转化为可见光的固态的半导体器件，它可以直接把电转化为光。LED的心脏是一个半导体的晶片，晶片的一端附在一个支架上，一端是负极，另一端连接电源的正极，使整个晶片被环氧树脂封装起来。

既然是二极管，就一个非常重要的特性，就是**单向导电性**。所以它的一端要接正极，一端是负极。并且不能反接。

二、点灯电路原理





上面这张图是 MicroROS 学习板的蓝色LED电路图和实际电路， R2 是一个 1K 欧姆的电阻， LED1 是一个蓝色的 LED 灯，右侧是 3.3V 的电压源，左侧 ESP_IO2 是单片机的引脚。

电流是从电压高的地方流向电压低的地方，如果我们将 ESP_IO2 的电压设成 3.3V 时，此时电路两端电压相同，没有电流经过，此时LED1不工作。

如果我们将 ESP_IO2 设置成 0V 时，此时右侧电压高，左侧电压低，电流从右侧流过 LED1 到 ESP_IO2 ，此时LED1开始工作。

所以我们可以通过 ESP_IO2 的电压高低来控制 LED1 灯的亮灭，这个就是点灯电路的原理。

三、总结

本节简单的介绍了LED点灯电路的原理，下一节我们通过控制 ESP_IO2 的电平来测试是否可以控制 LED1 灯的亮灭。

2.完成点灯-GPIO控制

你好，我是爱吃鱼香ROS的小鱼。上一节了解了MicroROS学习板的LED电路原理，最后得到结论是通过控制ESP_IO2的电平就可以控制电压，接下来我们就尝试利用Arduino的ESP_IO2对IO进行控制。

一、GPIO控制介绍

1.1 什么是GPIO

首先我们了解下GPIO（General-purpose input/output），中文名通用型之输入输出的简称。

- 输出模式，指GPIO是可以通过程序**控制其电压高低**，普通的GPIO只能输出（低电平0V）和高电平（3.3V，有的单片机是5V）。
- 输入模式，指GPIO可以**读取其上的电压**，普通的GPIO只能读取低电平和高电平两种。比如当我们想测试一个按键是否被按下，就可以GPIO的输入功能。

1.2 GPIO控制API

Arduino提供了简单易用的API来控制IO的输入和输出。

1.2.1 引脚模式设置- `pinMode`

该函数用于定义特定引脚的 GPIO 操作模式。

```c++
void pinMode(uint8_t pin, uint8_t mode);
```

- `pin` 定义 GPIO 引脚编号。
- `mode` 设置操作模式。

基本输入和输出支持以下模式：

- **INPUT**将 GPIO 设置为不带上拉或下拉（高阻抗）的输入。
- **OUTPUT**将 GPIO 设置为输出/读取模式。
- **INPUT_PULLDOWN** 将 GPIO 设置为具有内部下拉列表的输入。
- **INPUT_PULLUP** 将 GPIO 设置为带有内部上拉的输入。

1.2.2 数字输出- digitalWrite

`digitalWrite` 用于设置被配置为 `OUTPUT` 模式的引脚电平为 `HIGH` 或 `LOW` 。

```c++
void digitalWrite(uint8_t pin, uint8_t val);
```

- `pin` 所设置的GPIO编号。
- `val` 将输出数字状态设置为 `HIGH` 或 `LOW` 。

1.2.3 数字输入- digitalRead

`digitalRead` 用于读取配置为 `INPUT` 模式的给定引脚的状态。

```c++
int digitalRead(uint8_t pin);
```

- `pin` 所设置的GPIO编号。

此函数将返回所选引脚的逻辑状态为 `HIGH` 或 `LOW`。

有了上面三个函数，我们就可以编写代码了。

二、编写LED闪烁代码

新建 `example02_led` 工程，接着输入下面的代码。

```c++
/**
 * @file main.cpp
 * @author fishros@foxmail.com
 * @brief 使LED灯亮1s关闭1s,持续闪烁
 * @version 0.1
 * @date 2022-12-19
 *
 * @copyright Copyright (c) 2022
 *
 */
```

```
#include <Arduino.h>

void setup()
{
  // put your setup code here, to run once:
  pinMode(2, OUTPUT); // 设置2号引脚模式为OUTPUT模式
}

void loop()
{
  // put your main code here, to run repeatedly:
  digitalWrite(2, LOW);  // 低电平，打开LED灯
  delay(1000);           // 休眠1000ms
  digitalWrite(2, HIGH); // 高电平，关闭LED灯
  delay(1000);           // 休眠1000ms
}
```

这里我们让LED亮1s关闭1s，持续闪烁，代码很简单，不再讲解。

编译下载代码到开发板，接着观察现象，你会发现蓝色LED灯在不断闪烁。

三、通过串口控制LED

我们继续更新下工程代码，实现通过串口指令来控制LED的亮灭。

c++

```
#include <Arduino.h>

void setup()
{
  Serial.begin(115200);
  pinMode(2, OUTPUT); // 设置2号引脚模式为OUTPUT模式
}

void loop()
{
  // LED串口控制代码
  if (Serial.available())
  {
    String command = Serial.readString();
```

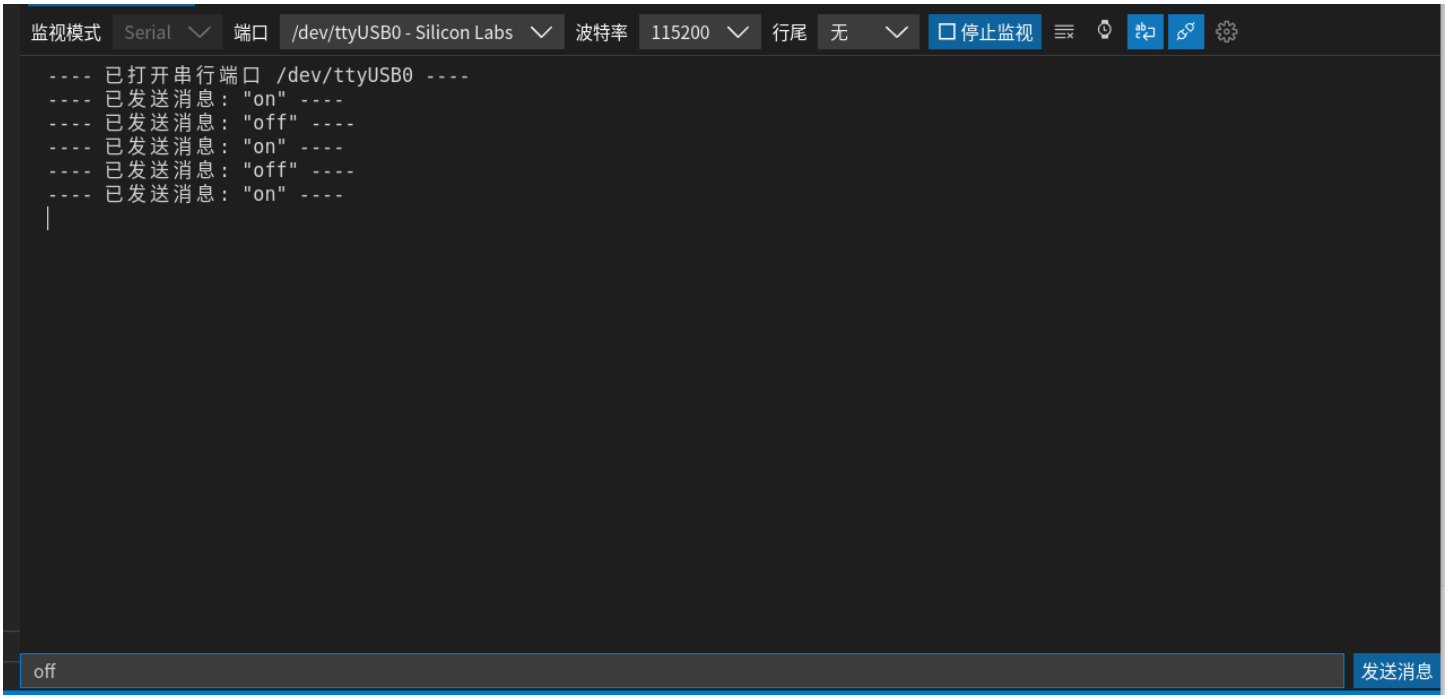
```
      if (command == "on")
      {
        digitalWrite(2, LOW); // 低电平，打开LED灯
      }
      else if (command == "off")
      {

        digitalWrite(2, HIGH); // 高电平，关闭LED灯
      }
    }

    /* // LED闪烁代码
      digitalWrite(2, LOW);  // 低电平，打开LED灯
      delay(1000);           // 休眠1000ms
      digitalWrite(2, HIGH); // 高电平，关闭LED灯
      delay(1000);           // 休眠1000ms
    */
  }
```

通过串口读取指令，收到on则打开LED，收到off关闭LED。

将代码下载到开发板，接着打开串口监视器，输入指令进行测试。



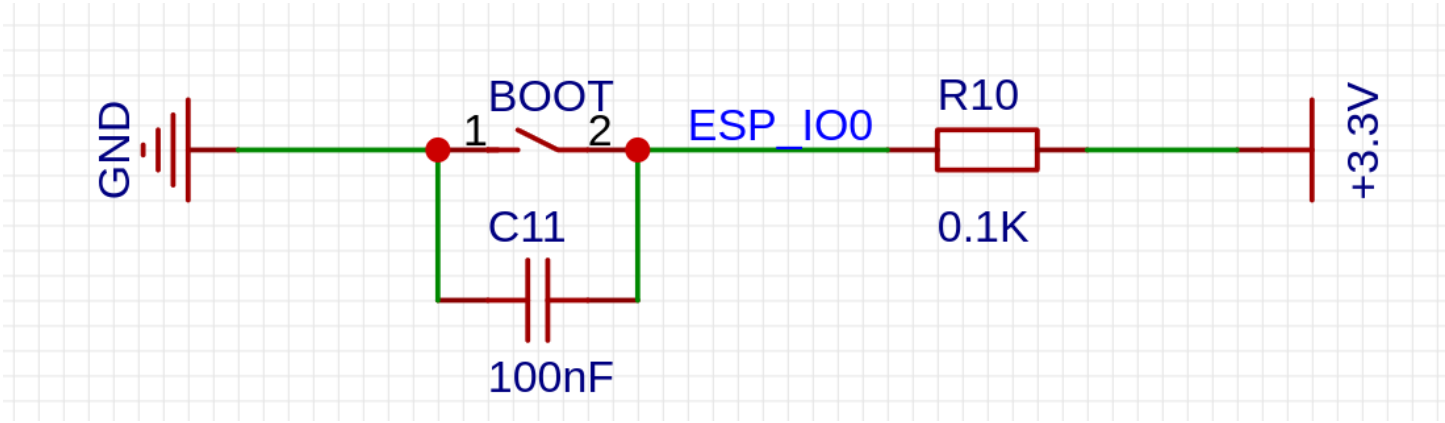
四、总结

至此，我们成功完成了点灯，掌握了基本的GPIO输出控制，下一节我们尝试使用按键输入来控制LED灯。

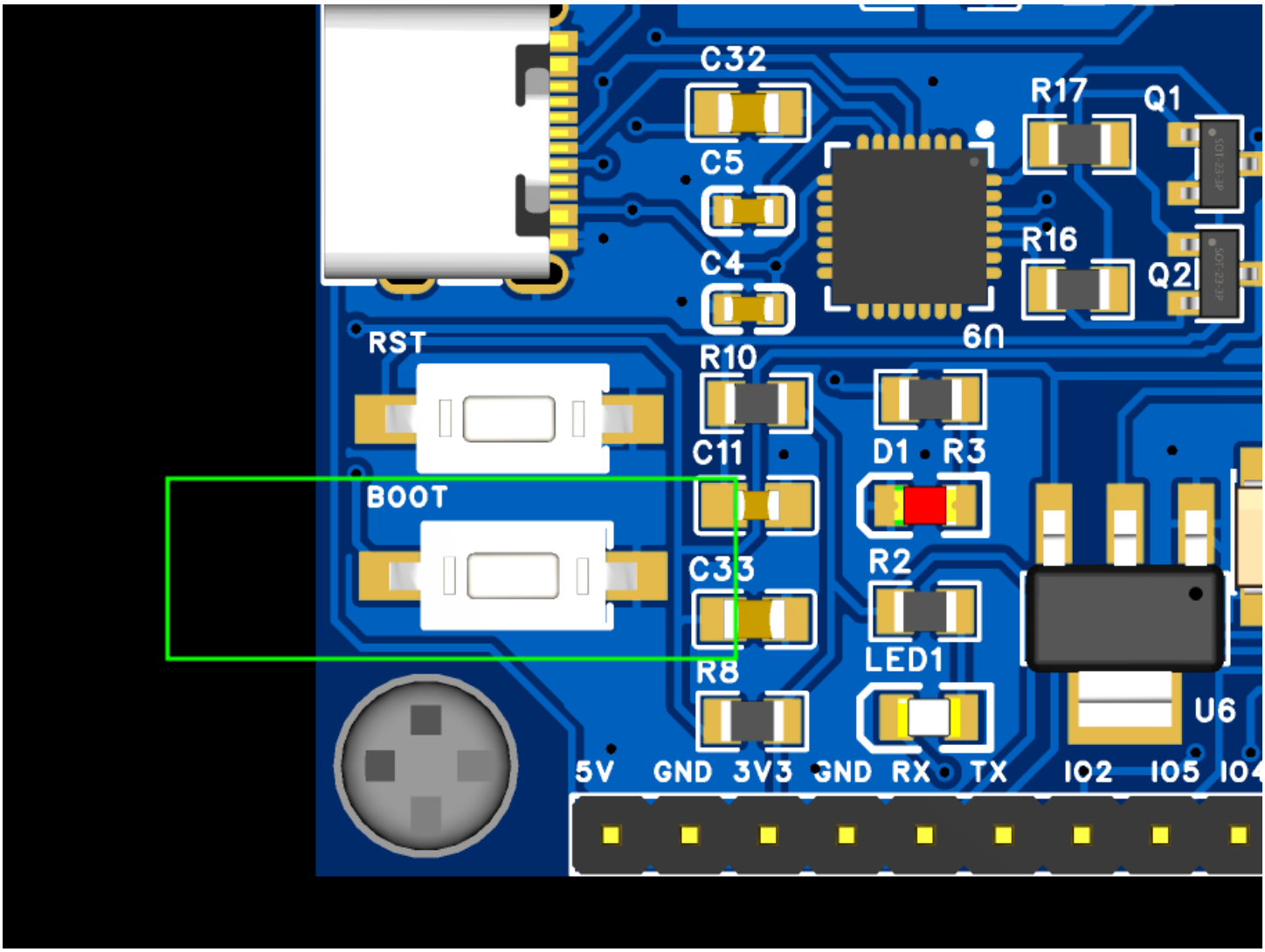
3.学会使用按键-GPIO输入

你好，我是爱吃鱼香ROS的小鱼。上一节完成了LED灯的控制，本节我们学习下如何使用按键。

一、 按键检测原理



上面的原理图就是我们MicroROS开发板的BOOT按键的原理图，该按键的位置在板子的左下角。



简单的看下原理图可知

- 当 `BOOT` 按下时，1,2就被联通了，此时 `ESP_IO2` 就连接到了 `GND` 上，也就是连接到 `GND` ， `ESP_IO2` 的电平被拉低了。
- 当 `BOOT` 没有按下时， `ESP_IO0` 通过 `R10` 连接到了 `3.3V` ， `ESP_IO2` 引脚上的电压就被拉高了。

二、使用按键控制LED灯（按着开松开关）

新建 `example03_key` 工程，在 `main.cpp` 输入代码。

```c++
/**
 * @file main.cpp
 * @author fishros@foxmail.com
 * @brief 使用按键控制LED灯,按着开，松开关
 * @version 0.1
 * @date 2022-12-19
 *
 * @copyright Copyright (c) 2022
 *
 */
#include <Arduino.h>

void setup()
{
  Serial.begin(115200);
  pinMode(0, INPUT);  // 设置2号引脚模式为OUTPUT模式
  pinMode(2, OUTPUT); // 设置2号引脚模式为OUTPUT模式
}

void loop()
{

  if (digitalRead(0) == LOW)
  {
    Serial.println("LED ON");
    digitalWrite(2, LOW); // 低电平，打开LED灯
  }
  else
  {
```

```
    Serial.println("LED OFF");
    digitalWrite(2, HIGH); // 高电平，关闭LED灯
  }
}
```

接着将代码编译烧录到开发板上，按下按键，查看LED灯。

三、使用按键控制LED（自锁开关）

所谓自锁，就是按下开，再按一下关，看一下代码实现

```c++
/**
 * @file main.cpp
 * @author fishros@foxmail.com
 * @brief 使用按键控制LED灯,按着开，松开关，自锁开关
 * @version 0.1
 * @date 2022-12-19
 *
 * @copyright Copyright (c) 2022
 *
 */
#include <Arduino.h>

bool status = false;

void setup()
{
  Serial.begin(115200);
  pinMode(0, INPUT);  // 设置2号引脚模式为OUTPUT模式
  pinMode(2, OUTPUT); // 设置2号引脚模式为OUTPUT模式
}

void loop()
{
  // 自锁开关
  if (digitalRead(0) == LOW)
  {
    delay(50); // 休眠50ms再次判断，防止误触
    if (digitalRead(0) == LOW)
```

```
      {
        status = !status;
        while (digitalRead(0) == LOW) // 死循环等待放开按键
          ;
      }
    }

    if (status == true)
    {
      digitalWrite(2, LOW); // 低电平，打开LED灯
    }
    else
    {

      digitalWrite(2, HIGH); // 低电平，打开LED灯
    }

    /*
    // 按着开，松开关
    if (digitalRead(0) == LOW)
    {
      Serial.println("LED ON");
      digitalWrite(2, LOW); // 低电平，打开LED灯
    }
    else
    {

      Serial.println("LED OFF");
      digitalWrite(2, HIGH); // 高电平，关闭LED灯
    }
    */
  }
```

这里稍微复杂一些，当按键按下时我们需要休眠50ms，防止误触，接着反置状态，等待按键松开。

将代码下载到开发板，按下按键，测试一下。

四、总结

本节我们通过学习按键的使用，掌握了GPIO的输入，下一节我们将尝试进阶使用DAC，测量电池的电。

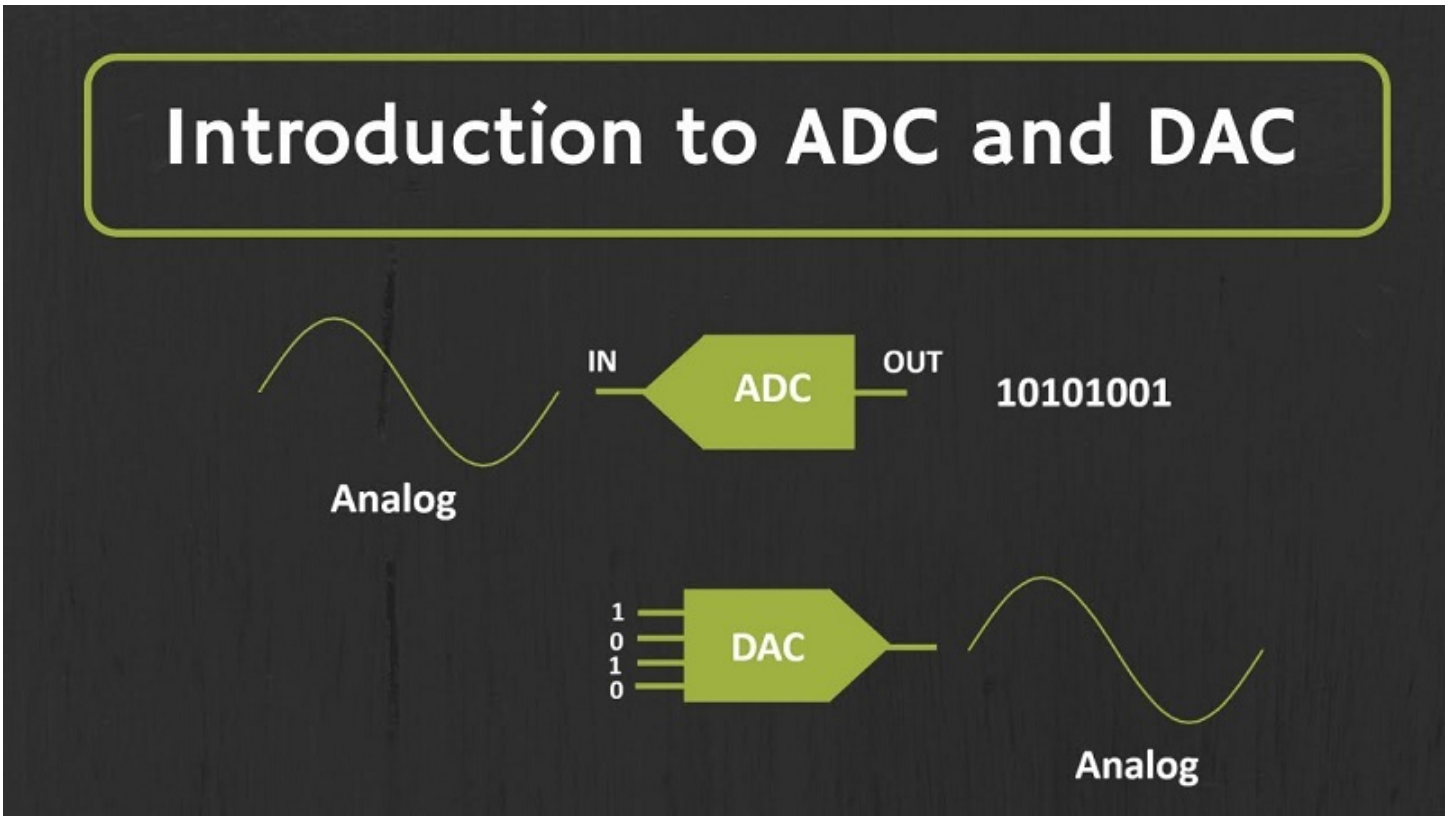
4.电池电压测量-学会使用ADC

你好，我是爱吃鱼香ROS的小鱼。上面两节通过LED和按键学习了GPIO的输出和输入。

但这种输入和输出只有两种状态 `HIGH` 或者 `LOW` ，我们称这种为数字逻辑，这也是输入输出函数称为数字输入和数字输出的原因。

但是电池的电压是一个在一个范围内不断变化的值，明显无法通过 `HIGH` 和 `LOW` 来表示，所以本节我们学习使用ADC，将电压值这个模拟信号转换成数字信号。

一、ADC介绍



ADC（analog to digital converter）模数转换器是一种非常常见的外设，用于**将电压等模拟信号转换为数字形式**，以便微控制器可以读取和处理。

ADC在控制和监控应用中非常有用，因为大多数传感器（例如温度、压力、力）都是输出的模拟电压，所以我们需要掌握ADC。

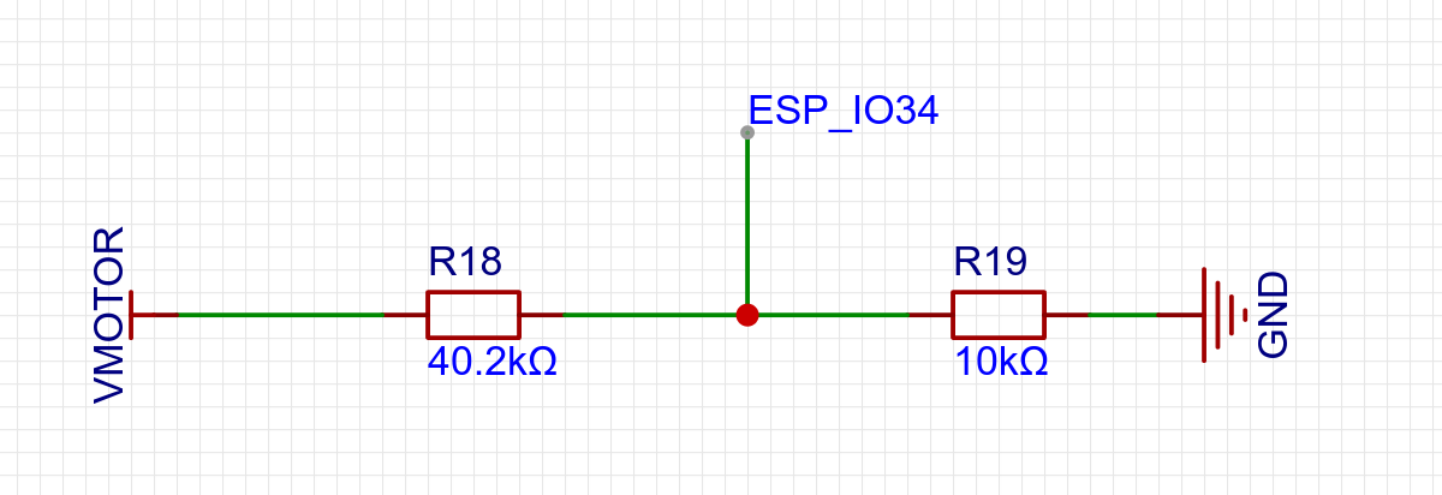
> **与ADC相对应的DAC——用于将数字信号转换成模拟信号，比如将一段二进制的音乐文件转换成一段连续的电压信号播放出来就需要DAC。**

二、电池电压测量原理

在我们的开发板所使用的ESP32单片机上，自带了ADC模块，我们只需要将需要测量的模拟电压接入相应引脚，接着调用ADC相关API即可读取。

因为ADC原理是采用电压比较方式进行测量，而我们的单片机的供电电压为3.3V，所以测量的电压范围最大不能超过3.3V，但我们采用的电池电压和板子的供电电压分别是12V和5V的。

所以想要使用单片机测量电池电压，那么就要想办法将板子测量的电压按照比例缩小即可，根据初中物理知识，串联分压原理，小鱼就设计了这样的电路。



串联分压，左边接电机电压输入引脚，右侧接地，R18是40.2千欧阻值的电阻，R19是10千欧的，假如此时VMOTOR的输入电压为5V，那么ESP_IO34的电压就是 $$ V_{ESPIO34}=V_{VMOTOR}*(10/(40.2+10))=V_{VMOTOR}/5.02 $$ 那么如果此时通过ADC测量出$V_{ESPIO34}$上的电压，通过下面的等式就可以算出V_{VMOTOR}的电压值。 $$ V_{VMOTOR} =V_{ESPIO34}*5.02 $$

三、Arduino ADC API

了解了原理，我们来看看Arduino为我们提供了哪些API可以直接获取到引脚上的电压。

3.1 设置ADC衰减系数（analogReadResolution）

这个函数用来设置ADC读取时的衰减系数。

输入到引脚的电压在输入到ADC之前可能会衰减。有 4 种可用的衰减选项，衰减越高，可测量的输入电压就越高，为了能够测量到12V以上的电压，我们采用最高的衰减比-ADC_11db。

```
typedef enum {
    ADC_0db,
    ADC_2_5db,
    ADC_6db,
    ADC_11db,
} adc_attenuation_t;
```

3.2 读取ADC值（analogRead）

此函数用于获取给定引脚或ADC通道的ADC原始值，默认是12位分辨率，所以这个读出来的值的范围就是0-2^12，也就是最大4096。

```
uint16_t analogRead(uint8_t pin);
```

- `pin` 要读取ADC值的GPIO 引脚

返回值：ADC原始值

3.3 读取电压值（analogReadMillivolts）

此函数用于获取给定引脚或ADC 通道的 ADC 值（以毫伏为单位）。

```
uint32_t analogReadMilliVolts(uint8_t pin);
```

- `pin` 要读取ADC值的GPIO 引脚

返回值：此函数将以毫伏为单位返回模拟值。

四、编写程序

利用上面的三个API和计算公式就可以实现电压的测量，新建 `example04_adc` 工程，在 `main.cpp` 输入代码。

cpp

```cpp
/**
 * @file main.cpp
 * @author fishros@foxmail.com
 * @brief 4.电池电压测量-学会使用ADC
 * @version 0.1
 * @date 2023-01-04
 *
 * @copyright Copyright(c) fishros.com  2023
 *
 */
#include <Arduino.h>

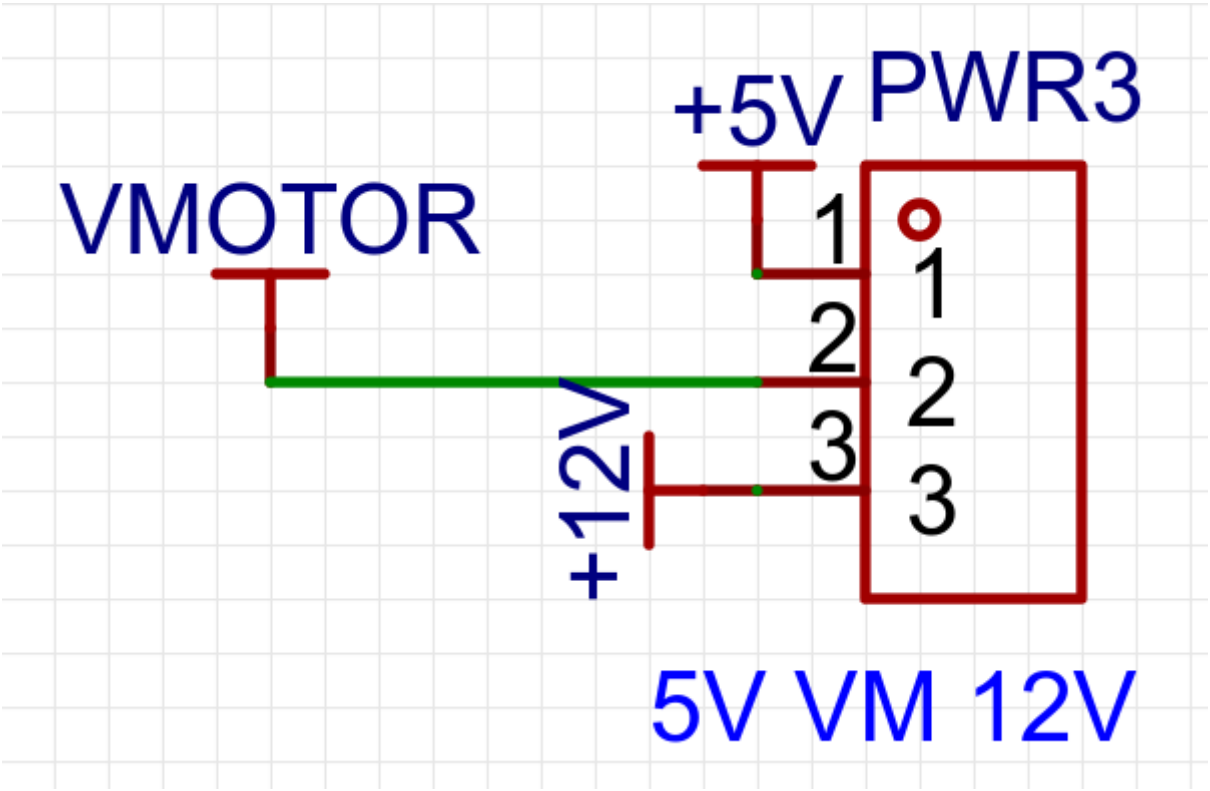
void setup()
{
  Serial.begin(115200);
  pinMode(34, INPUT);
  analogSetAttenuation(ADC_11db);
}

void loop()
{
  int analogValue = analogRead(34);                      // 读取原始值0-4096
  int analogVolts = analogReadMilliVolts(34);            // 读取模拟电压，单位毫伏
  float realVolts = 5.02 * ((float)analogVolts * 1e-3); // 计算实际电压值

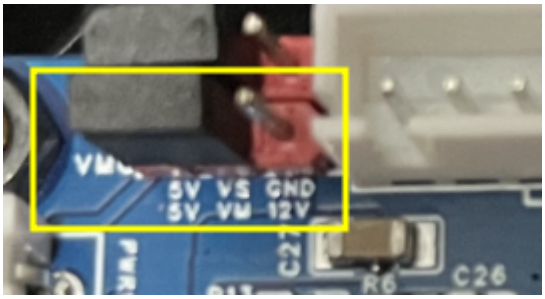
  Serial.printf("ADC analog value = %d\n", analogValue);
  Serial.printf("ADC millivolts value = %d\n", analogVolts);
  Serial.printf("realVolts value = %f\n", realVolts);
  delay(100);
}
```

五、测试

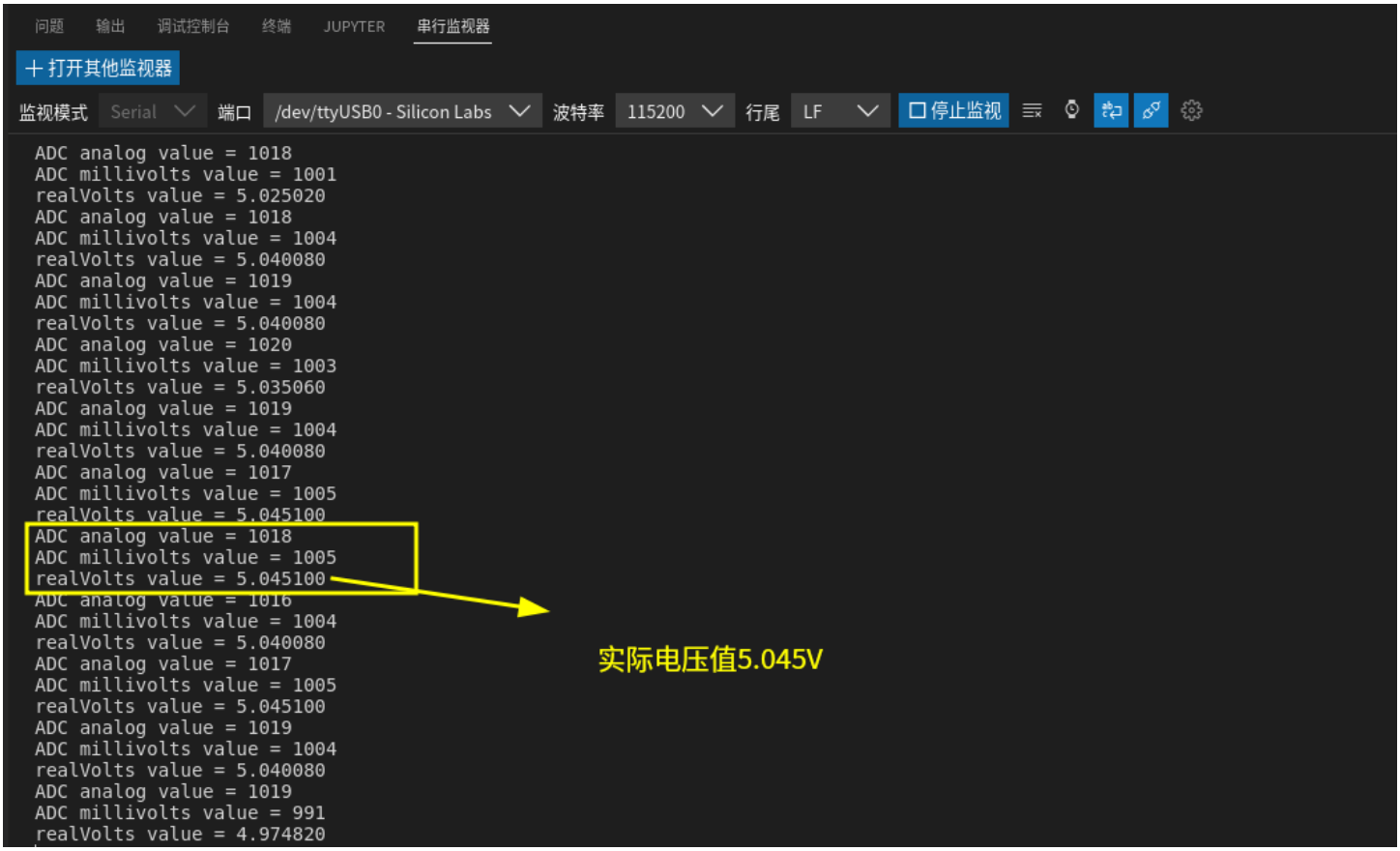
根据第三节中的原理图，我们测量的是VMOTOR的电压，VMOTOR是通过一个跳线帽选择连接到5V还是12V上的，详情可以看下面的原理图。



对应板子上的位置



这里我们把跳线帽调整到左侧，让VMOTOR和5V连接，接着打开串口观察测量到的电压值。



可以看到我们成功的测量到了实际的电压值为5.045V，符合正常电压值。

如果你有整台FishBot小车，可以将跳线帽调整到12V的位置，打开电池电源开关，看看是否可以正常测量到12V的电压。

六、总结

本节我们通过电池电压测量的例程，学习了ADC的使用，电池电压测量对我们机器人来说是非常重要的，当电压低的时候可以进行自动返回充电，然后等充电完成后再继续工作。

1.学会安装第三方开源库

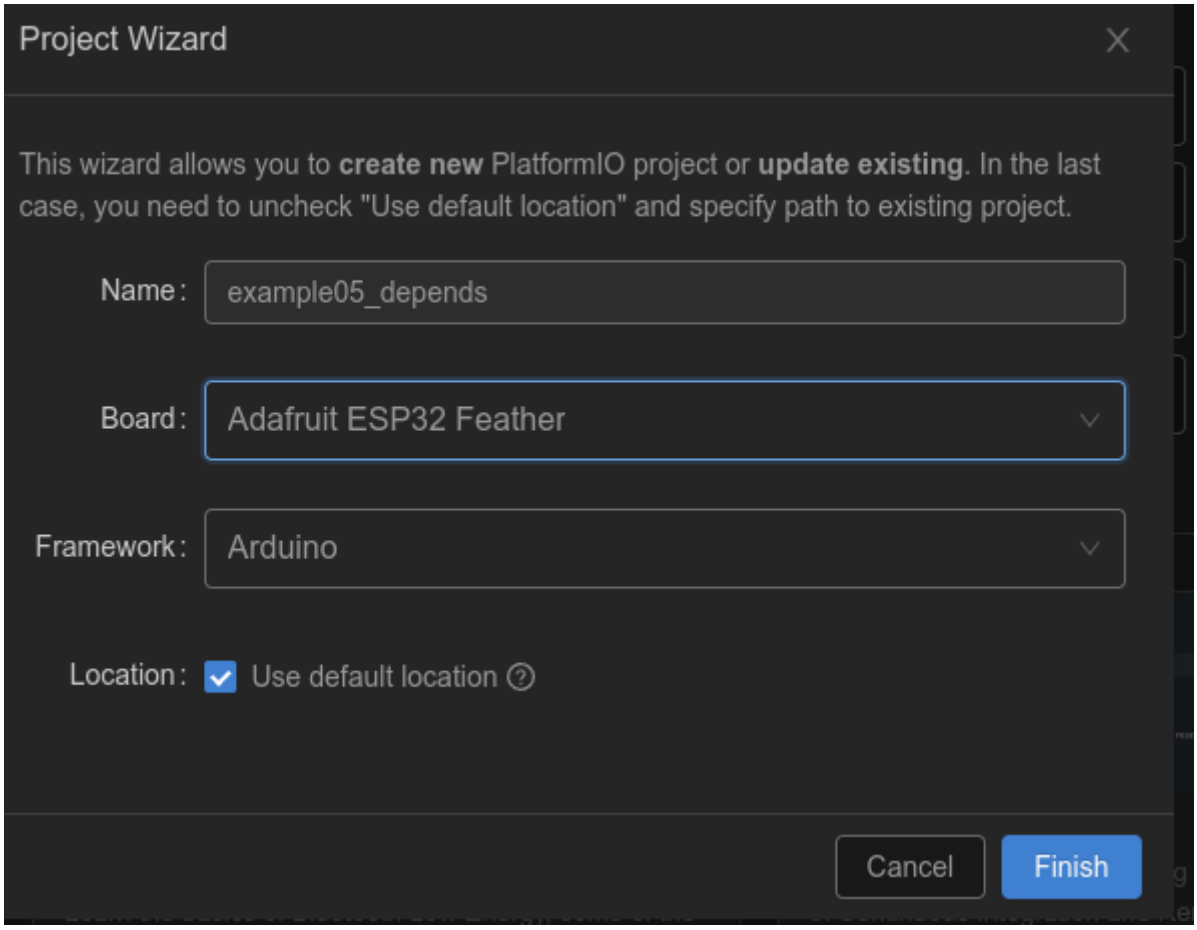
你好，我是爱吃鱼香ROS的小鱼。秉承着拒绝重复造轮子的ROS精神，本节我们学习如何在我们的工程里安装第三方开源库。

在我们的PIO工程中有多种方式可以添加第三方库，常用的有以下三种：

- 通过PIO搜索安装
- 通过GIT地址安装
- 手动下载安装

接下来我们以安装OLED库和IMU的驱动库为例，学习安装第三方库的方法。

开始之前先新建一个工程 `example05_depends`



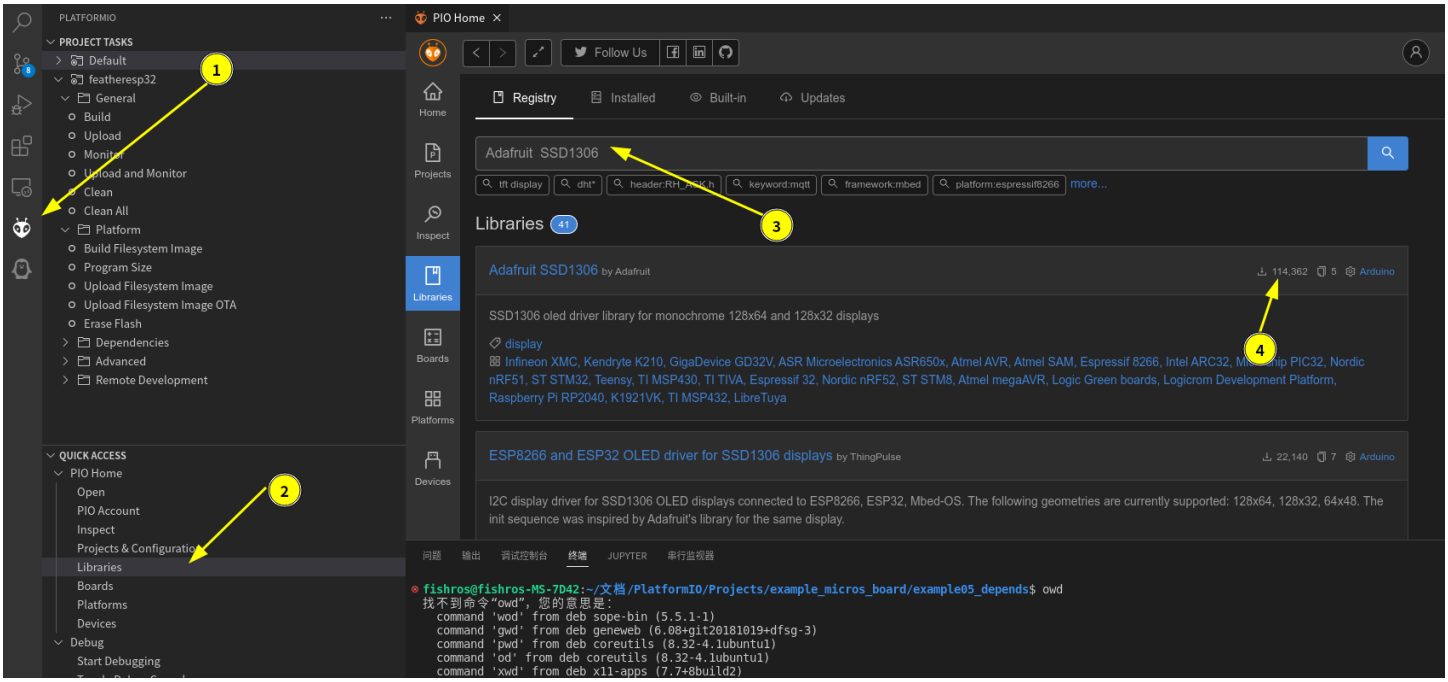
一、通过PIO搜索安装

我们以安装OLED常用的三方库 `Adafruit SSD1306` 安装为例。

1. 点击PIO图标
2. 点击Libraries

3. 输入 `Adafruit SSD1306`

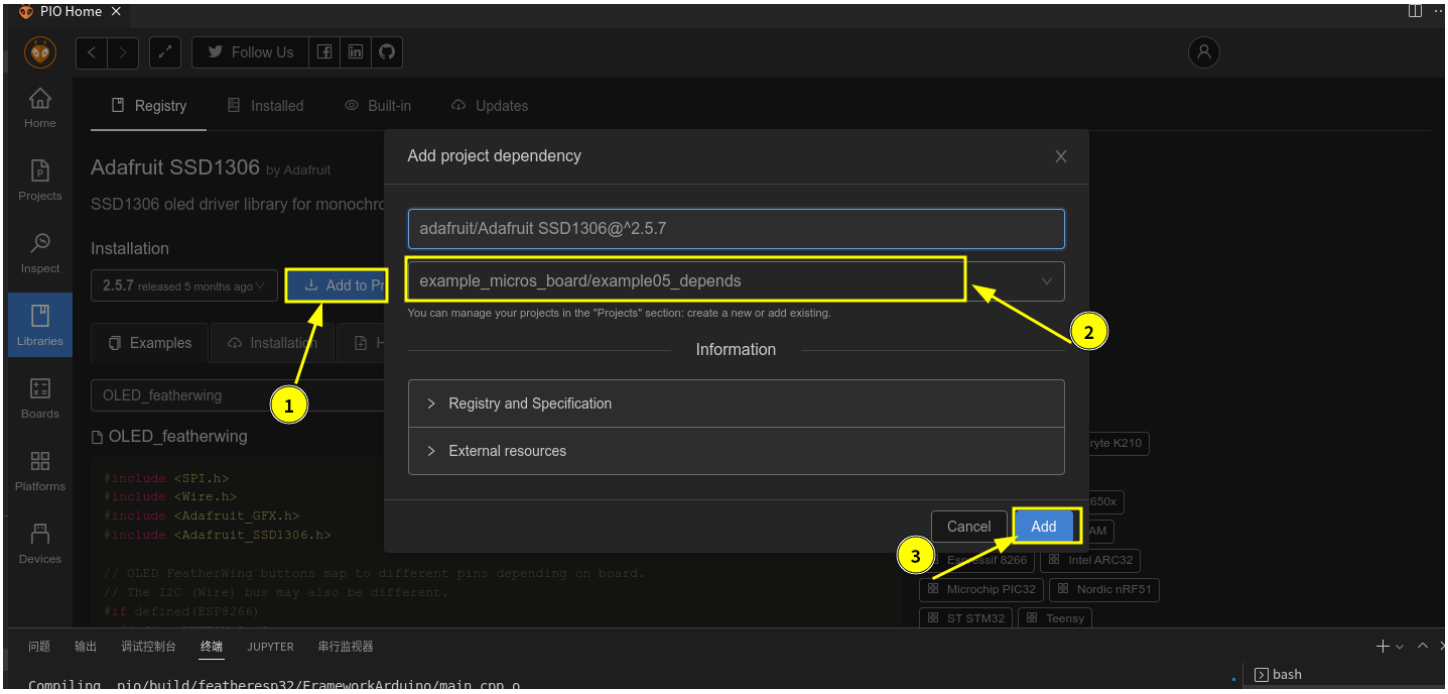
4. 点击下载按钮



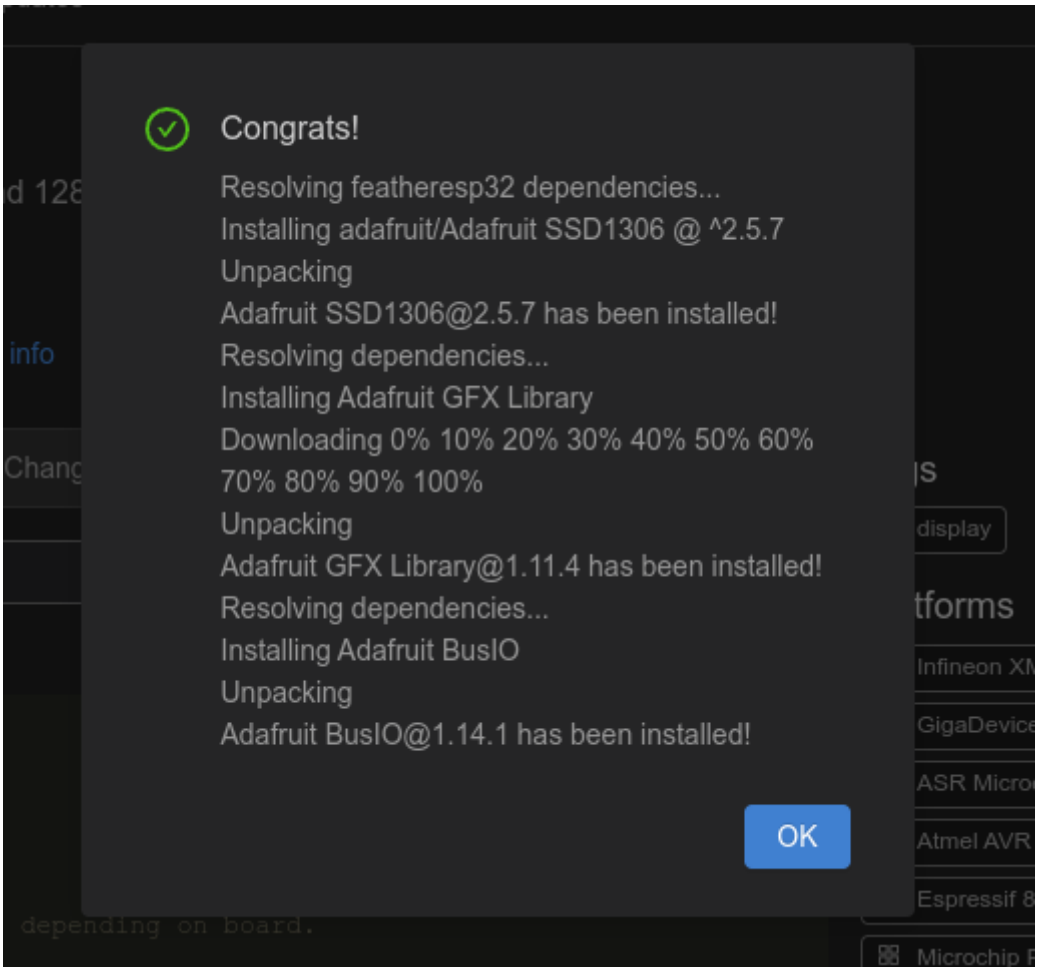
5.点击Add to Project

6.选择要添加到的工程

7.点击Add



看到界面表示成功



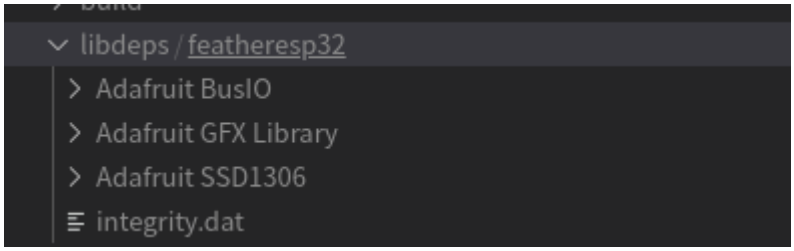
此时打开 `platformio.ini` 你将看到

```
[env:featheresp32]
platform = espressif32
board = featheresp32
framework = arduino

lib_deps = adafruit/Adafruit SSD1306@^2.5.7
```

`lib_deps = adafruit/Adafruit SSD1306@^2.5.7` 就是我们安装的库的名字。

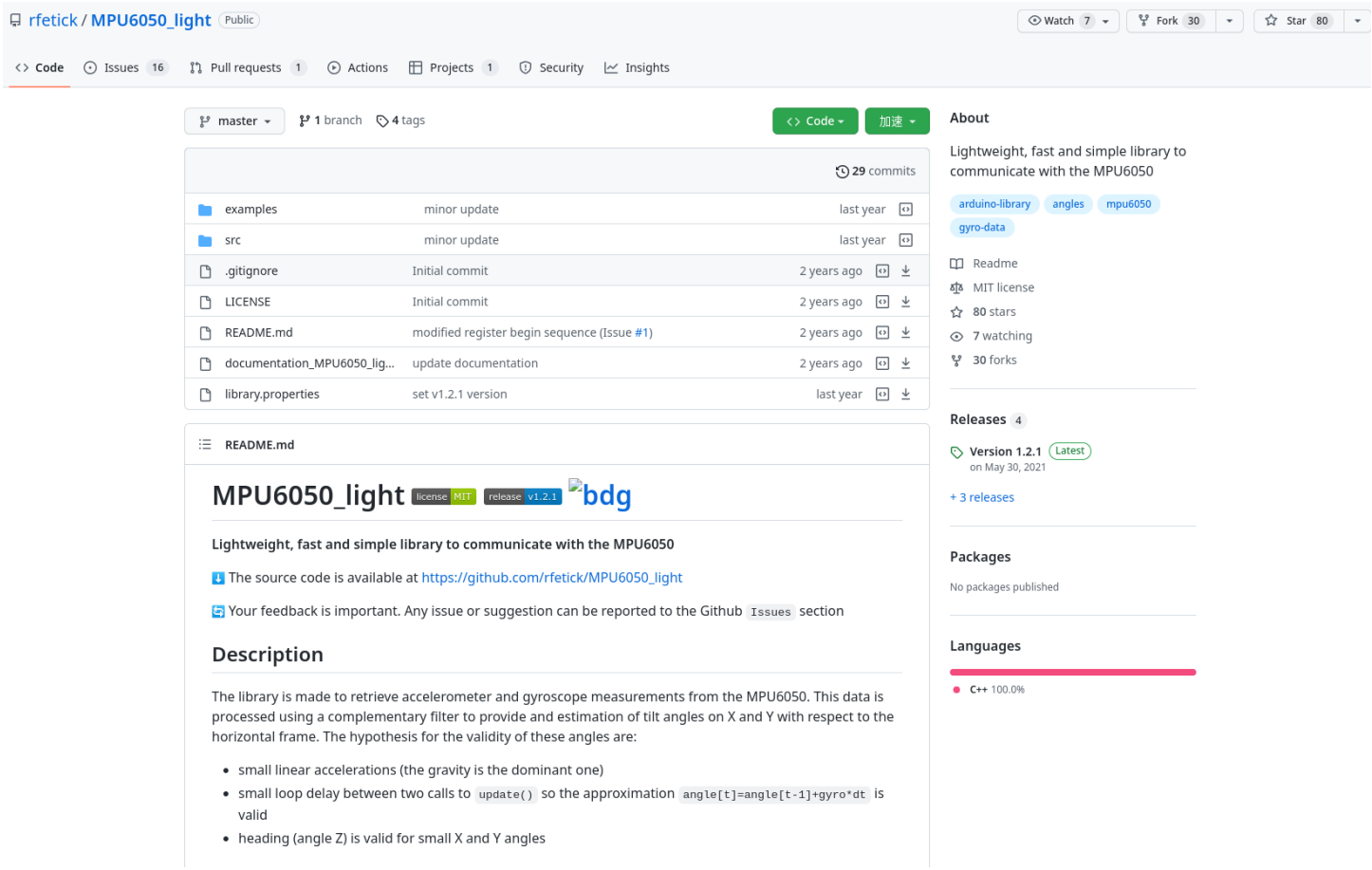
既然安装好了那安装的文件位置在哪里呢？打开 `.pio/libdeps/featheresp32` ，这里就是我们安装的第三方库的代码位置。



二、通过GIT地址安装

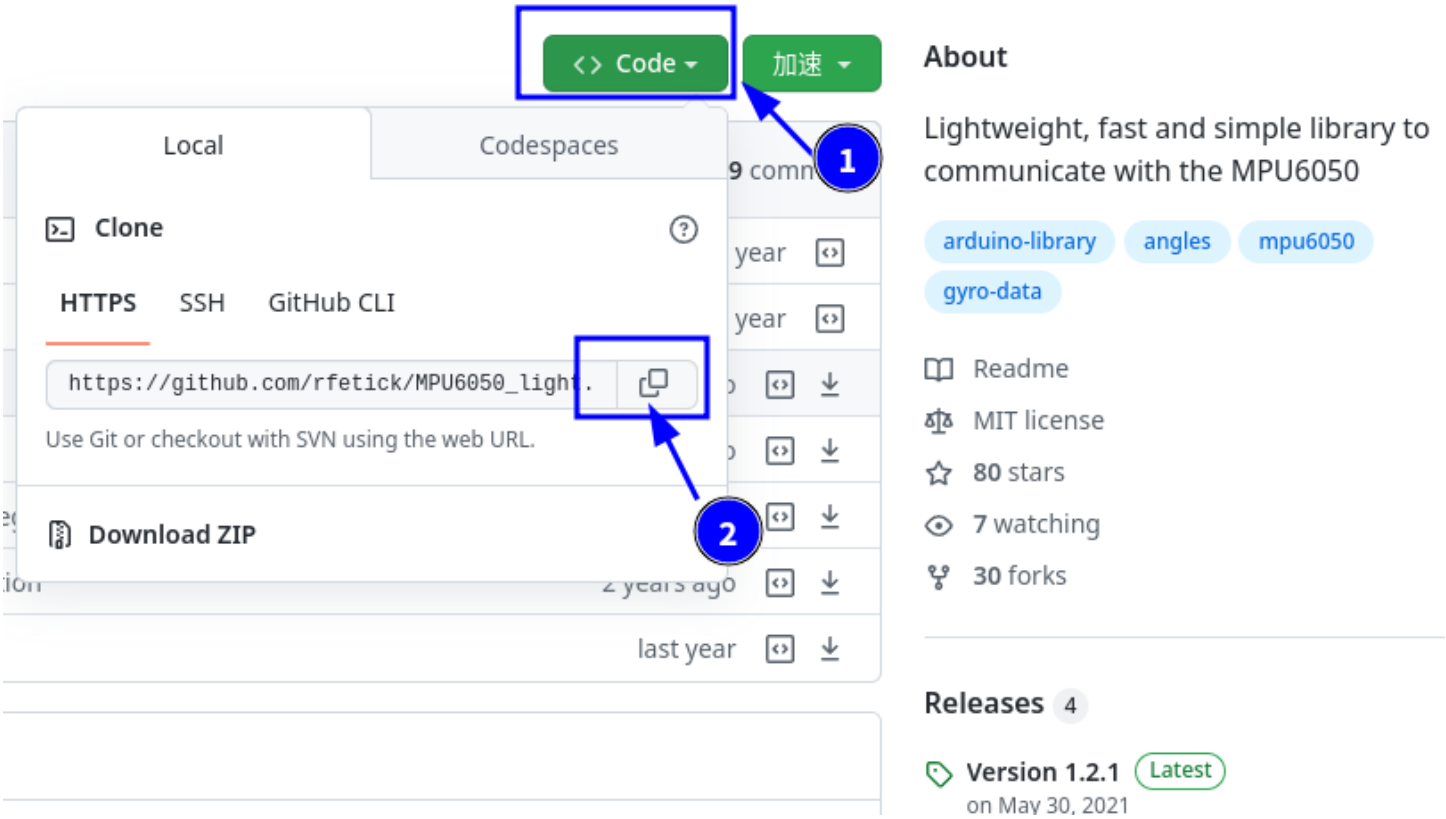
我们的开发板，板载了一块MPU6050模块，通过该模块可以实现对温度、加速度、加速度、重力测量。

这里小鱼推荐一个简单易用MPU6050的三方驱动库MPU6050_light: https://github.com/rfetick/MPU6050_light



接着我们来看如何将该库添加到我们的工程中

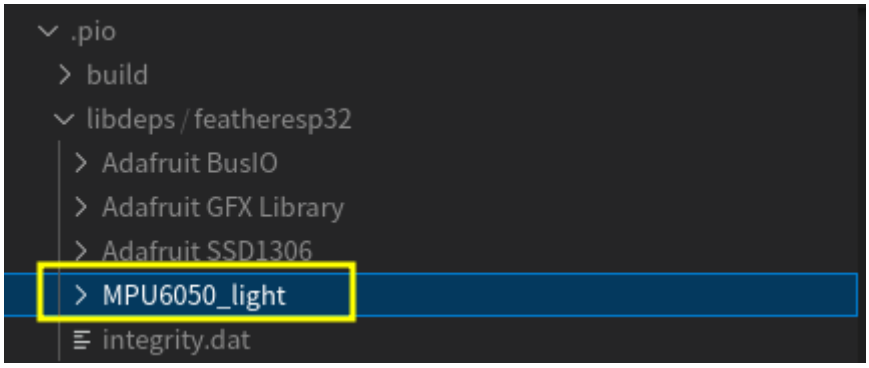
1.复制仓库地址



2.打开 `platformio.ini` ，将地址复制进去即可

```ini
[env:featheresp32]
platform = espressif32
board = featheresp32
framework = arduino
lib_deps =
    adafruit/Adafruit SSD1306@^2.5.7
    https://github.com/rfetick/MPU6050_light.git
```

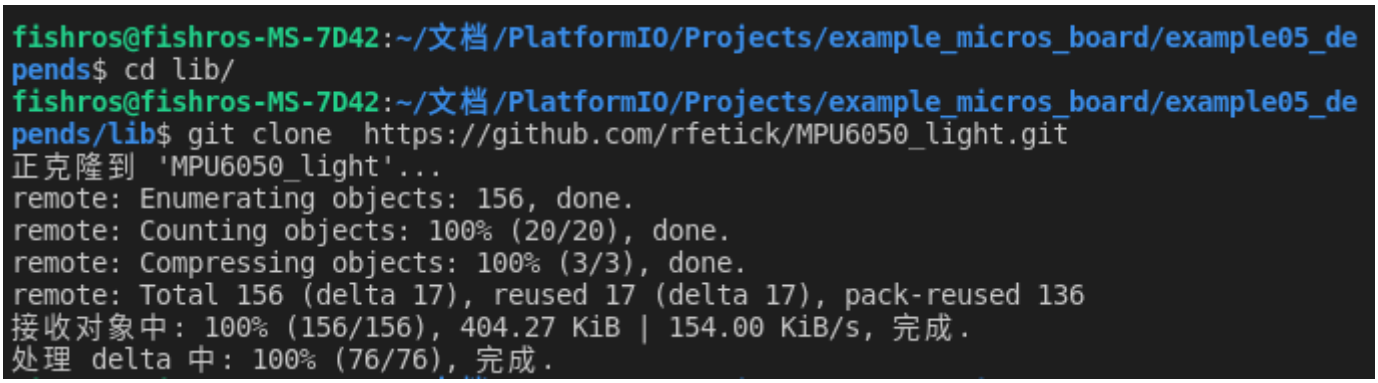
稍等片刻，等待PIO下载完成，接着打开 `.pio/libdeps/featheresp32` 可以看到 `MPU6050_light` 库被下载到该目录。



三、手动下载安装

该方式更简单，我们直接将工程克隆到工程的lib目录下即可。

```shell
cd lib
git clone https://github.com/rfetick/MPU6050_light.git
```



四、总结

本节我们学习了三种安装三方库的方式，下一节我们开始尝试使用三方库来驱动MPU6050和OLED。

2.使用开源库驱动IMU

你好，我是爱吃鱼香ROS的小鱼。上一节我们安装好了MPU6050的三方库，这一节我们尝试使用该库将我们板子上的IMU模块驱动起来。

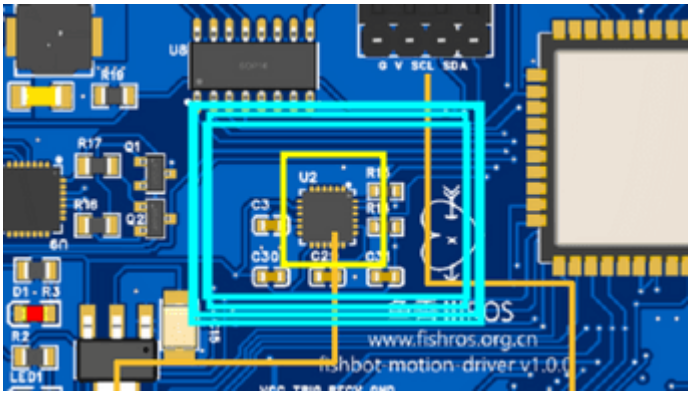
一、MPU6050介绍

首先我们了解下MPU6050模块，从外观看，长这个样子



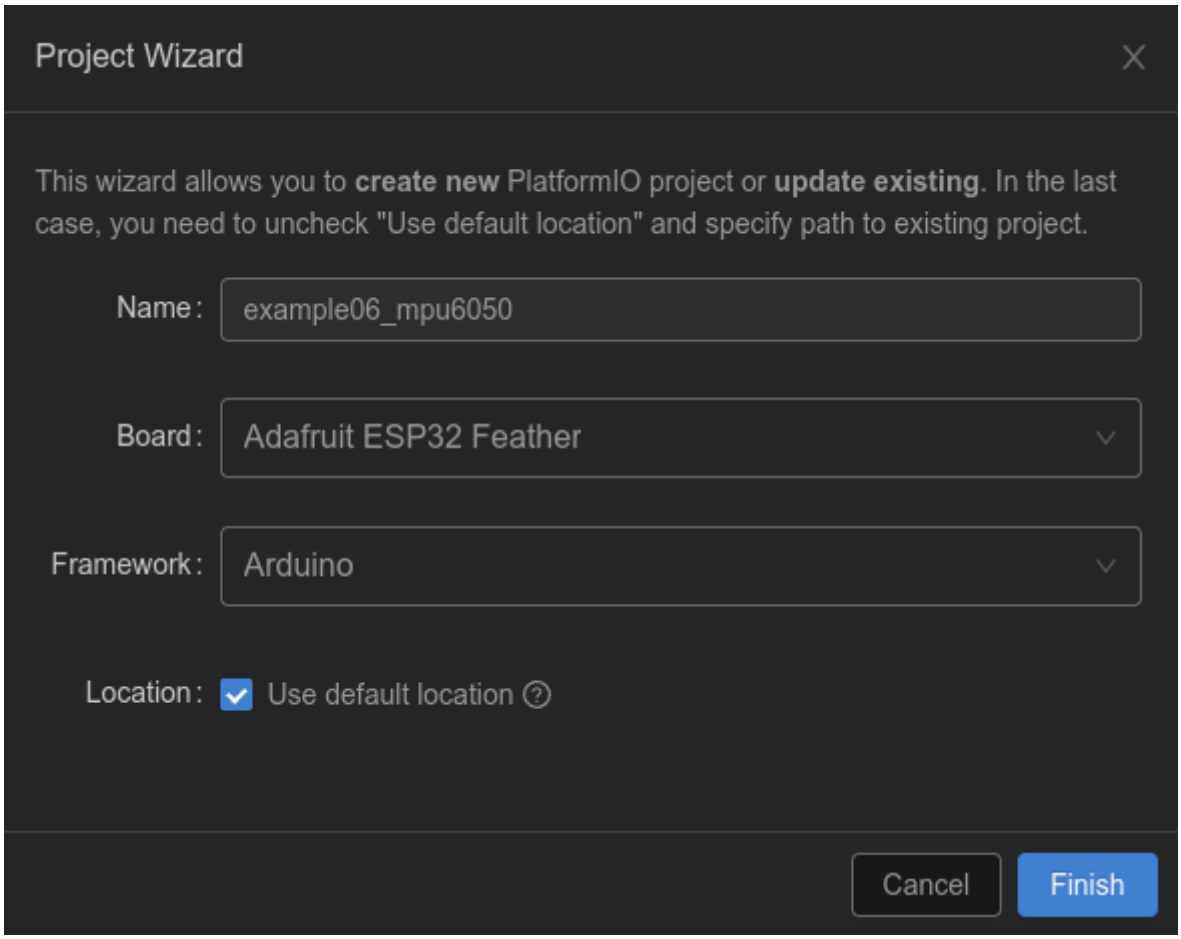
MPU6050 为全球首例集成六轴传感器的运动处理组件，内置了运动融合引擎，用于手持和桌面的应用程序、游戏控制器、体感遥控以及其他消费电子设备。它内置一个三轴 MEMS 陀螺仪、一个三轴 MEMS 加速度计、一个数字运动处理引擎（DMP）以及用于第三方的数字传感器接口的辅助 I2C 端口（常用于扩展磁力计）。当辅助 I2C 端口连接到一个三轴磁力计，MPU6050 能提供一个完整的九轴融合输出到其主 I2C 端口。

在我们板子上的位置是这里



二、调用开源库驱动

新建工程 `example06_mpu6050`



2.1 添加依赖

修改 `platformio.ini`

```ini
[env:featheresp32]
platform = espressif32
board = featheresp32
framework = arduino
lib_deps =
    https://ghproxy.com/https://github.com/rfetick/MPU6050_light.git
```

2.2 复制样例程序

该开源库作者提供了开源库的使用方式，

将 `.pio/libdeps/featheresp32/MPU6050_light/examples/GetAllData/GetAllData.ino` 复制到main.cpp中。

```
/* Get all possible data from MPU6050
 * Accelerometer values are given as multiple of the gravity [1g = 9.81 m/s²]
 * Gyro values are given in deg/s
 * Angles are given in degrees
 * Note that X and Y are tilt angles and not pitch/roll.
 *
 * License: MIT
 */

#include "Wire.h"
#include <MPU6050_light.h>

MPU6050 mpu(Wire);

unsigned long timer = 0;

void setup() {
  Serial.begin(9600);
  Wire.begin();

  byte status = mpu.begin();
  Serial.print(F("MPU6050 status: "));
  Serial.println(status);
  while(status!=0){ } // stop everything if could not connect to MPU6050

  Serial.println(F("Calculating offsets, do not move MPU6050"));
  delay(1000);
  mpu.calcOffsets(true,true); // gyro and accelero
  Serial.println("Done!\n");

}

void loop() {
  mpu.update();

  if(millis() - timer > 1000){ // print data every second
    Serial.print(F("TEMPERATURE: "));Serial.println(mpu.getTemp());
    Serial.print(F("ACCELERO  X: "));Serial.print(mpu.getAccX());
    Serial.print("\tY: ");Serial.print(mpu.getAccY());
    Serial.print("\tZ: ");Serial.println(mpu.getAccZ());
```

```
    Serial.print(F("GYRO      X: "));Serial.print(mpu.getGyroX());
    Serial.print("\tY: ");Serial.print(mpu.getGyroY());
    Serial.print("\tZ: ");Serial.println(mpu.getGyroZ());

    Serial.print(F("ACC ANGLE X: "));Serial.print(mpu.getAccAngleX());
    Serial.print("\tY: ");Serial.println(mpu.getAccAngleY());

    Serial.print(F("ANGLE     X: "));Serial.print(mpu.getAngleX());
    Serial.print("\tY: ");Serial.print(mpu.getAngleY());
    Serial.print("\tZ: ");Serial.println(mpu.getAngleZ());
    Serial.println(F("=====================================================\n"));
    timer = millis();
  }

}
```

2.3 修改代码

1.修改波特率 `9600` -> `115200`

2.修改IO地址 `Wire.begin();` -> `Wire.begin(18, 19);`

修改完后代码，并附上小鱼对代码的注释讲解

```
#include "Wire.h"          // 导入I2C相关头文件
#include <MPU6050_light.h> // 导入MPU6050库

MPU6050 mpu(Wire); // 新建MPU6050对象mpu

unsigned long timer = 0;

void setup()
{
  Serial.begin(115200);
  Wire.begin(18, 19); // 初始化I2C，设置sda引脚为GPIO18,SCL引脚为GPIO19

  byte status = mpu.begin(); // 检测IMU模块状态
  Serial.print(F("MPU6050 status: "));
  Serial.println(status);
```

```
  while (status != 0)
  {
  } // stop everything if could not connect to MPU6050

  Serial.println(F("Calculating offsets, do not move MPU6050"));
  delay(1000);
  mpu.calcOffsets(true, true); // gyro and accelero 校准
  Serial.println("Done!\n");
}

void loop()
{
  mpu.update();

  if (millis() - timer > 1000)
  { // print data every second
    Serial.print(F("TEMPERATURE: "));
    Serial.println(mpu.getTemp()); // 温度
    Serial.print(F("ACCELERO  X: "));
    Serial.print(mpu.getAccX()); // X轴加速度
    Serial.print("\tY: ");
    Serial.print(mpu.getAccY()); // Y轴加速度
    Serial.print("\tZ: ");
    Serial.println(mpu.getAccZ()); // Z轴加速度

    Serial.print(F("GYRO      X: "));
    Serial.print(mpu.getGyroX()); // X轴 角速度
    Serial.print("\tY: ");
    Serial.print(mpu.getGyroY()); // Y轴 角速度
    Serial.print("\tZ: ");
    Serial.println(mpu.getGyroZ()); // Z轴 角速度

    Serial.print(F("ACC ANGLE X: "));
    Serial.print(mpu.getAccAngleX()); // X轴角加速度
    Serial.print("\tY: ");
    Serial.println(mpu.getAccAngleY()); // Y轴角加速度

    Serial.print(F("ANGLE     X: "));
    Serial.print(mpu.getAngleX()); // X角度
    Serial.print("\tY: ");
    Serial.print(mpu.getAngleY()); // Y角度
    Serial.print("\tZ: ");
```

```
      Serial.println(mpu.getAngleZ()); // Z角度
      Serial.println(F("=====================================================\n"));
      timer = millis();
    }
  }
```

三、编译测试

保存代码，编译下载到开发板。打开串口监视器，查看结果。

```
Hash of data verified.
Compressed 3072 bytes to 128...
Writing at 0x00008000... (100 %)
Wrote 3072 bytes (128 compressed) at 0x00008000 in 0.1 seconds (effective 483.3 kbit/s)...
Hash of data verified.
Compressed 8192 bytes to 47...
Writing at 0x0000e000... (100 %)
Wrote 8192 bytes (47 compressed) at 0x0000e000 in 0.1 seconds (effective 661.3 kbit/s)...
Hash of data verified.
Compressed 267760 bytes to 149527...
Writing at 0x00010000... (10 %)
Writing at 0x0001c701... (20 %)
Writing at 0x00024de8... (30 %)
Writing at 0x0002a29e... (40 %)
Writing at 0x0002f7e5... (50 %)
Writing at 0x00034d0b... (60 %)
Writing at 0x0003d65a... (70 %)
Writing at 0x00045b1f... (80 %)
Writing at 0x0004b0dd... (90 %)
Writing at 0x000509a9... (100 %)
Wrote 267760 bytes (149527 compressed) at 0x00010000 in 3.9 seconds (effective 544.8 kbit/s)...
Hash of data verified.

Leaving...
Hard resetting via RTS pin...
========================================================== [SUCCESS] Took 9.02 seconds ==========================
*  终端将被任务重用，按任意键关闭。
```

结果

```
load:0x40078000,len:13160
load:0x40080400,len:3036
entry 0x400805e4
MPU6050 status: 0
Calculating offsets, do not move MPU6050
Done!

TEMPERATURE: 35.52
ACCELERO  X: -0.00      Y: 0.00 Z: 0.99
GYRO      X: 0.13       Y: 0.07 Z: 0.05
ACC ANGLE X: 0.27       Y: 0.05
ANGLE     X: 1.65       Y: -0.59        Z: 0.10
=====================================================

TEMPERATURE: 35.56
ACCELERO  X: 0.00       Y: -0.00        Z: 1.00
GYRO      X: -0.13      Y: -0.01        Z: -0.04
ACC ANGLE X: -0.04      Y: -0.16
ANGLE     X: -0.03      Y: 0.01 Z: 0.10
=====================================================
```

四、总结

本节我们通过调用开源库实现了对IMU传感器的的调用，如果你对该库感兴趣，可以随时到 `.pio/libdeps/featheresp32/MPU6050_light/src/MPU6050_light.h` 查看源码

```c++
class MPU6050{
  public:
    // INIT and BASIC FUNCTIONS
    MPU6050(TwoWire &w);
    byte begin(int gyro_config_num=1, int acc_config_num=0);
    ...

  private:
    ...
};
```

可以看到，这里是通过面向对象的方式将MPU6050封装成了一个类，我们使用的时候也是通过实例化后使用的，所以下一节我们将学习如何在我们的工程里使用面向对象的方式进行封装。

3.学会面向对象编程-封装IMU驱动

上一节我们成功读取到了IMU的数据，其中角度用欧拉角的方式表示的，在我们机器人世界里姿态的表示往往使用四元数表示（如果不清楚他们之间的关系可以回看第六章机器人学篇），所以我们需要将欧拉角转换成四元数。除此之外我们还需要将其坐标系矫正到右手坐标系。

所以本节我们将通过面向对象的方式将IMU驱动进行封装，并为其添加坐标系转换以及四元数转换函数。

一、理论介绍

1.1.欧拉角转四元数

欧拉角转四元数的公式我们在第六章入门篇第三节有介绍，这里回顾一下

$$
\begin{aligned}
\mathbf{q}_{\mathrm{IB}} &= \begin{bmatrix} \cos(\psi/2) \\ 0 \\ 0 \\ \sin(\psi/2) \end{bmatrix} \begin{bmatrix} \cos(\theta/2) \\ 0 \\ \sin(\theta/2) \\ 0 \end{bmatrix} \begin{bmatrix} \cos(\phi/2) \\ \sin(\phi/2) \\ 0 \\ 0 \end{bmatrix} \\
&= \begin{bmatrix} \cos(\phi/2)\cos(\theta/2)\cos(\psi/2) + \sin(\phi/2)\sin(\theta/2)\sin(\psi/2) \\ \sin(\phi/2)\cos(\theta/2)\cos(\psi/2) - \cos(\phi/2)\sin(\theta/2)\sin(\psi/2) \\ \cos(\phi/2)\sin(\theta/2)\cos(\psi/2) + \sin(\phi/2)\cos(\theta/2)\sin(\psi/2) \\ \cos(\phi/2)\cos(\theta/2)\sin(\psi/2) - \sin(\phi/2)\sin(\theta/2)\cos(\psi/2) \end{bmatrix}
\end{aligned}
$$

根据公式我们可以写出代码

```c++
typedef struct
{
    float w;
    float x;
    float y;
    float z;
} quaternion_t;

Euler2Quaternion(float roll, float pitch, float yaw, quaternion_t &q)
{
    double cr = cos(roll * 0.5);
    double sr = sin(roll * 0.5);
```

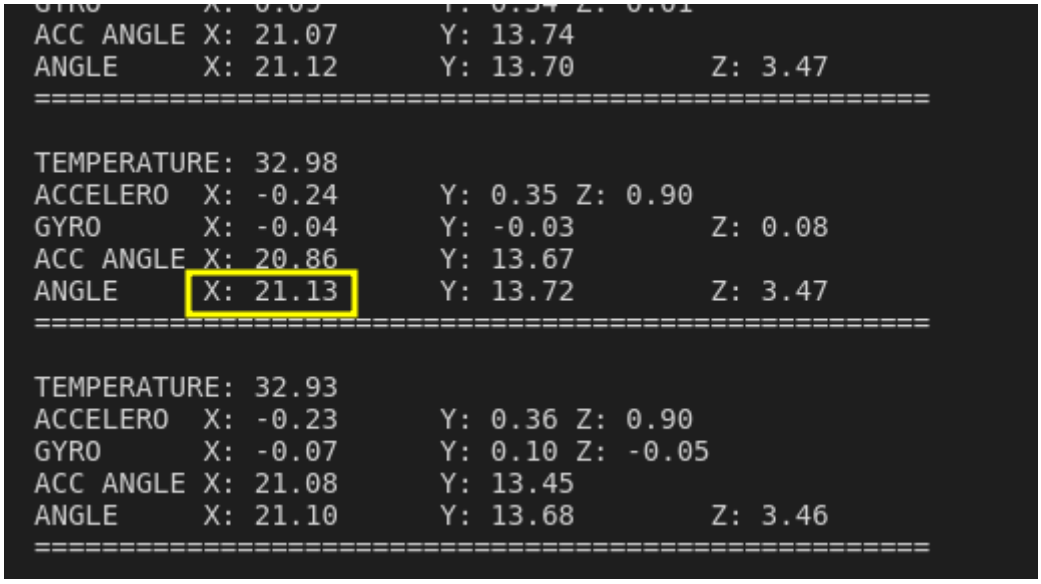
```
    double cy = cos(yaw * 0.5);
    double sy = sin(yaw * 0.5);
    double cp = cos(pitch * 0.5);
    double sp = sin(pitch * 0.5);
    q.w = cy * cp * cr + sy * sp * sr;
    q.x = cy * cp * sr - sy * sp * cr;
    q.y = sy * cp * sr + cy * sp * cr;
    q.z = sy * cp * cr - cy * sp * sr;
}
```

1.2 坐标系校准

我们采用右手坐标系，接着我们依次来校准角度数据的方向。

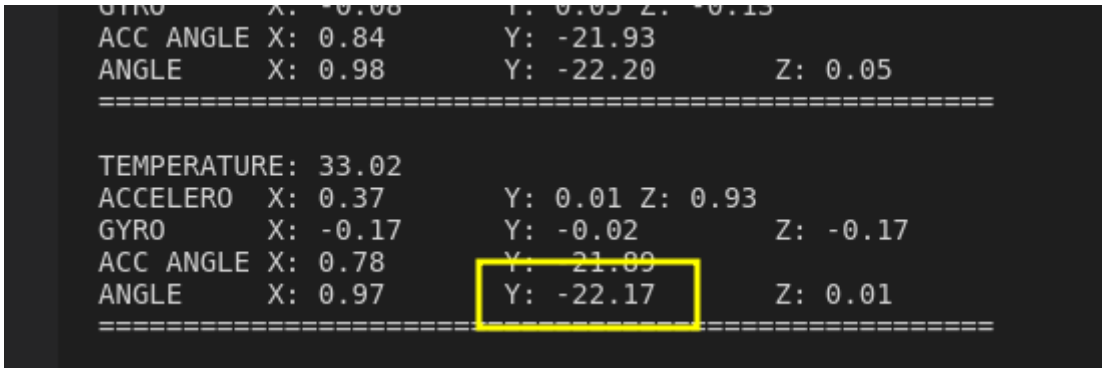
打开终端，点击RST,查看IMU数据。

首先是X轴，我们让开发板上爱神丘比特的剑头指向自己，然后从右侧往左侧倾斜。



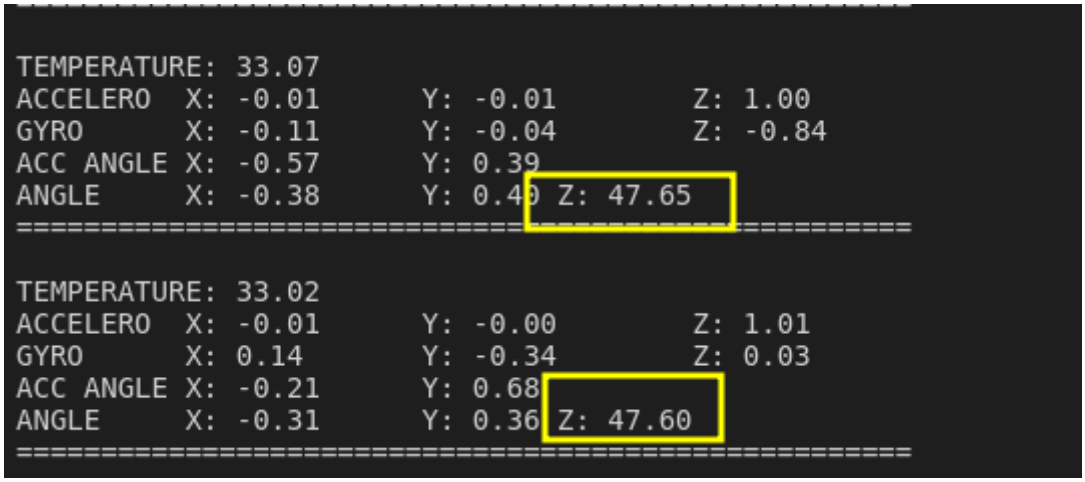
可以看到此时X轴为正值，符合右手坐标系法则。

接着是Y轴，平放，将箭头朝向自己的胸口，接着抬高板子，让箭头指向自己的头部，观察Y轴的变化。



Y轴为负值，不符合右手坐标系法则，所以Y的值应该取一次负，使其为正。

接着是Z轴，平放，将箭头朝向自己的胸口，然后逆时针旋转板子，观察数值变化。

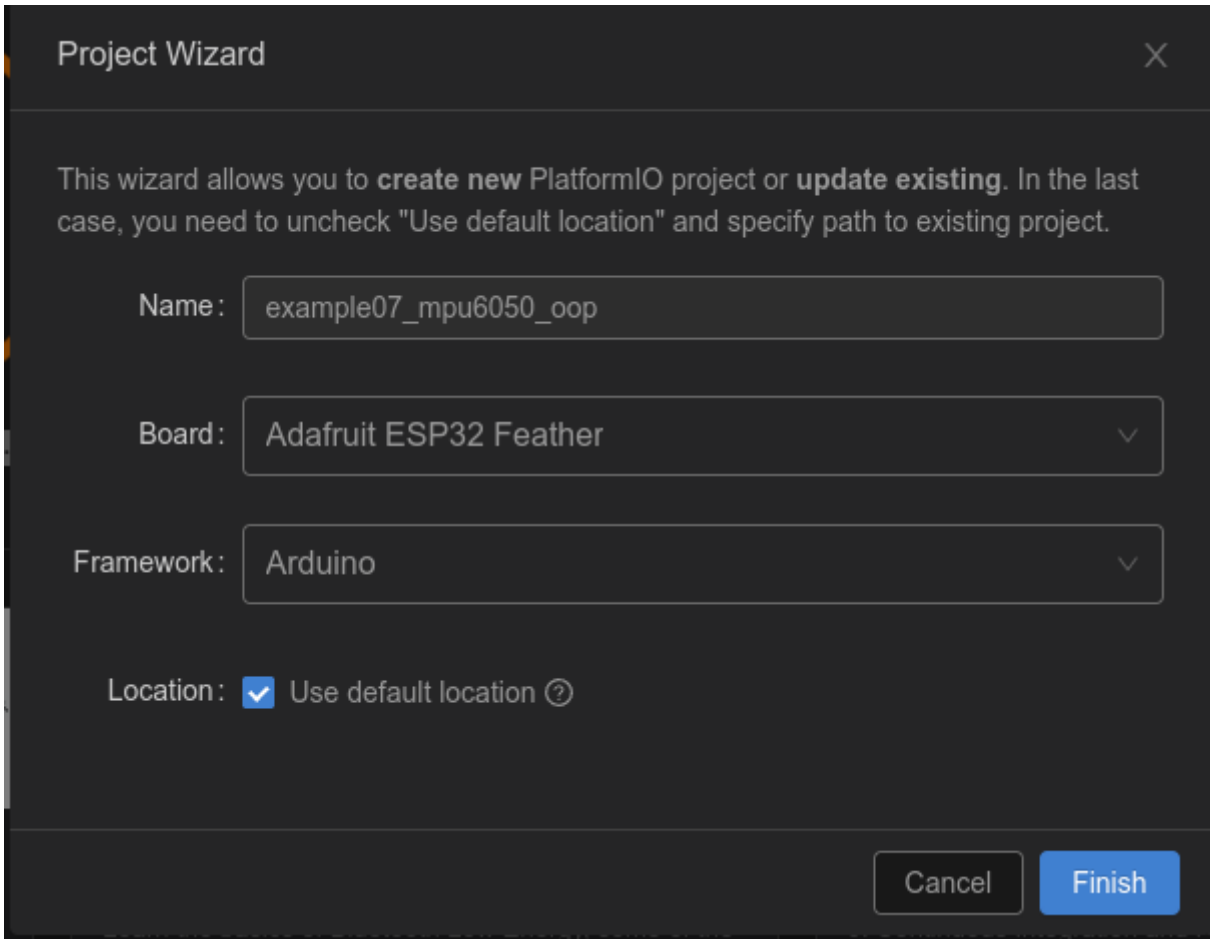


值为正，表示符合右手坐标系法则。

> 你可能会问小鱼怎么确认怎样旋转是正，怎样旋转是负，首先要确认轴向，我们开发板的Z轴朝上，X轴朝前，此时Y轴应该朝左。接着摊开右手手掌，用大拇指朝向轴的方向，比如朝向X轴，然后握起手掌，那么你握的方向就是正方向。

二、开始写代码

新建工程 `example07_mpu6050_oop`

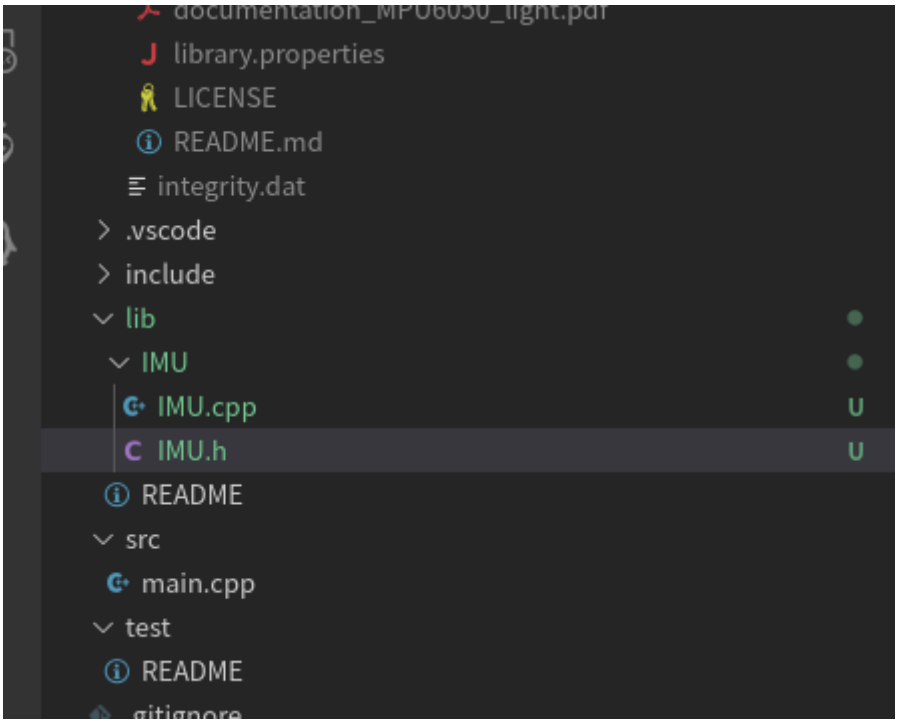


接着为其添加依赖

修改 `platformio.ini`

```ini
[env:featheresp32]
platform = espressif32
board = featheresp32
framework = arduino
lib_deps =
    https://ghproxy.com/https://github.com/rfetick/MPU6050_light.git
```

接着在 `lib` 下新建 `IMU` 文件夹，并在文件夹下新建 `IMU.h` 和 `IMU.cpp`



IMU.h

```c++
#ifndef __IMU_H__
#define __IMU_H__
#include "Wire.h"
#include "MPU6050_light.h"

typedef struct
{
    float w;
    float x;
    float y;
    float z;
} quaternion_t; // 四元数结构体

typedef struct
{
    float x;
    float y;
    float z;
} vector_3d_t; // 通用3D点结构体

typedef struct
{
    quaternion_t orientation;
```

```
    vector_3d_t angle_euler;
    vector_3d_t angular_velocity;
    vector_3d_t linear_acceleration;
} imu_t; // IMU数据结构体

class IMU
{
private:
    MPU6050 *mpu_; // mpu6050指针

public:
    /**
     * @brief 同u哦NPU6050构造一个新的IMU对象
     *
     * @param mpu
     */
    IMU(MPU6050 &mpu);

    ~IMU() = default;
    /**
     * @brief 初始化函数
     *
     * @param sda 引脚编号
     * @param scl 引脚编号
     * @return true
     * @return false
     */
    bool begin(int sda, int scl);
    /**
     * @brief 欧拉角转四元数
     *
     * @param roll 输入X
     * @param pitch 输入y
     * @param yaw 输入Z
     * @param q  返回的四元数引用
     */
    static void Euler2Quaternion(float roll, float pitch, float yaw, quaternion_t
&q);
    /**
     * @brief 获取IMU数据函数
     *
     * @param imu
```

```
     */
    void getImuData(imu_t &imu);
    /**
     * @brief 更新IMU数据，同上一节中的mou.update
     *
     */
    void update();
};

#endif // __IMU_H__
```

IMU.cpp

```c++
#include "IMU.h"

IMU::IMU(MPU6050 &mpu)
{
    mpu_ = &mpu;
};

bool IMU::begin(int sda, int scl)
{
    Wire.begin(sda, scl);
    byte status = mpu_->begin();
    Serial.print(F("MPU6050 status: "));
    Serial.println(status);
    if (status != 0)
    {
        return false;
    } // stop everything if could not connect to MPU6050

    Serial.println(F("Calculating offsets, do not move MPU6050"));
    delay(1000);
    // mpu.upsideDownMounting = true; // uncomment this line if the MPU6050 is
mounted upside-down
    mpu_->calcOffsets(); // gyro and accelero
    Serial.println("Done!\n");
    return true;
}
```

```
void IMU::Euler2Quaternion(float roll, float pitch, float yaw, quaternion_t &q)
{
    double cr = cos(roll * 0.5);
    double sr = sin(roll * 0.5);
    double cy = cos(yaw * 0.5);
    double sy = sin(yaw * 0.5);
    double cp = cos(pitch * 0.5);
    double sp = sin(pitch * 0.5);
    q.w = cy * cp * cr + sy * sp * sr;
    q.x = cy * cp * sr - sy * sp * cr;
    q.y = sy * cp * sr + cy * sp * cr;
    q.z = sy * cp * cr - cy * sp * sr;
}

void IMU::getImuData(imu_t &imu)
{
    imu.angle_euler.x = mpu_->getAngleX();
    imu.angle_euler.y = -mpu_->getAngleY();
    imu.angle_euler.z = mpu_->getAngleZ();

    imu.angular_velocity.x = mpu_->getAccAngleX();
    imu.angular_velocity.y = -mpu_->getAccAngleY();
    imu.angular_velocity.z = mpu_->getGyroZ();

    imu.linear_acceleration.x = mpu_->getAccX();
    imu.linear_acceleration.y = mpu_->getAccY();
    imu.linear_acceleration.z = mpu_->getAccZ();

    IMU::Euler2Quaternion(imu.angle_euler.x, imu.angle_euler.y, imu.angle_euler.z,
                          imu.orientation);
}

void IMU::update()
{
    mpu_->update();
}
```

main.cpp

```c++
#include <Arduino.h>
#include "IMU.h"

MPU6050 mpu(Wire); // 初始化MPU6050对象
IMU imu(mpu);      // 初始化IMU对象

imu_t imu_data;
unsigned long timer = 0;

void setup()
{
  Serial.begin(115200);
  imu.begin(18, 19); // 初始化IMU,使用18，19引脚
}

void loop()
{
  imu.update();
  if ((millis() - timer) > 100)
  {
    imu.getImuData(imu_data); // 获取IMU数据结构体
    Serial.printf("imu:\teuler(%f,%f,%f)\n",
                  imu_data.angle_euler.x, imu_data.angle_euler.y,
imu_data.angle_euler.z);
    Serial.printf("imu:\torientation(%f,%f,%f,%f)\n",
                  imu_data.orientation.w, imu_data.orientation.x,
imu_data.orientation.y, imu_data.orientation.z);
    timer = millis();
  }
}
```

对于代码的解释已经放到了注释之中。

编译下载后，你将看到

```
imu:    euler(-0.027950,-0.045859,5.550223)
imu:    orientation(-0.933143,0.021258,0.016397,0.358502)
imu:    euler(-0.023656,-0.035493,5.552567)
imu:    orientation(-0.933726,0.017385,0.012349,0.357353)
imu:    euler(0.020969,-0.032352,5.553412)
imu:    orientation(-0.934051,-0.004021,0.018850,0.356618)
imu:    euler(-0.018387,-0.004851,5.554946)
imu:    orientation(-0.934387,0.009455,-0.001008,0.356132)
imu:    euler(-0.074680,-0.025603,5.557319)
imu:    orientation(-0.933962,0.039438,-0.001293,0.355188)
imu:    euler(-0.040194,-0.017326,5.556992)
imu:    orientation(-0.934516,0.021861,0.000959,0.355249)
imu:    euler(0.000016,-0.047852,5.559347)
imu:    orientation(-0.934951,0.008463,0.022377,0.353968)
imu:    euler(0.017953,-0.032103,5.559355)
imu:    orientation(-0.935113,-0.002711,0.018188,0.353871)
imu:    euler(0.000915,-0.027144,5.562002)
imu:    orientation(-0.935604,0.004360,0.012860,0.352790)
imu:    euler(-0.003213,-0.071244,5.562299)
imu:    orientation(-0.935126,0.014063,0.032760,0.352518)
```

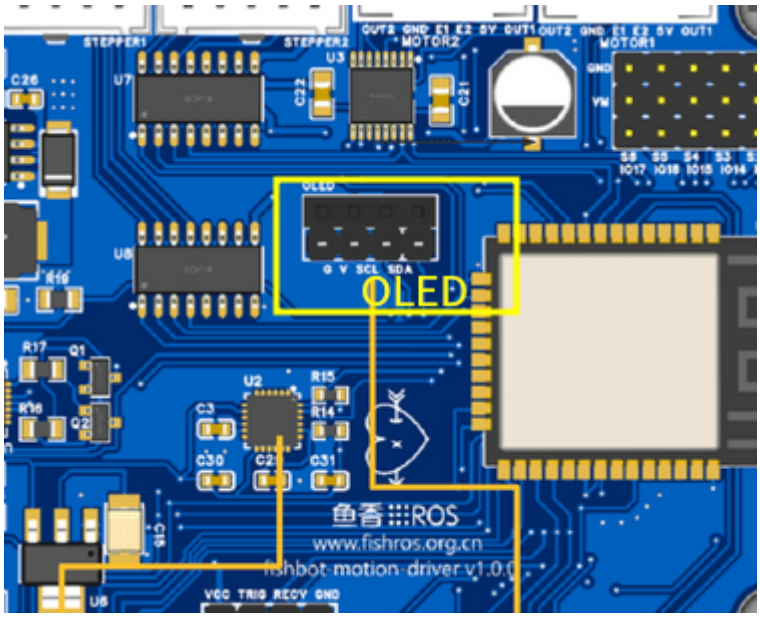
三、总结

本节我们通过对MPU6050驱动的封装，学习了如何在嵌入式上使用面向对象编程的方法，下一节我们继续尝试使用开源库来驱动OLED模块，让我们的显示器亮起来。.

4.使用开源库驱动OLED

你好，我是爱吃鱼香ROS的小鱼。本节我们继续尝试使用开源库，驱动OLED模块，最后的效果实现在OLED上显示当前的角度信息。

我们MicroROS开发板上的OLED位置如图所示。



一、OLED模块介绍



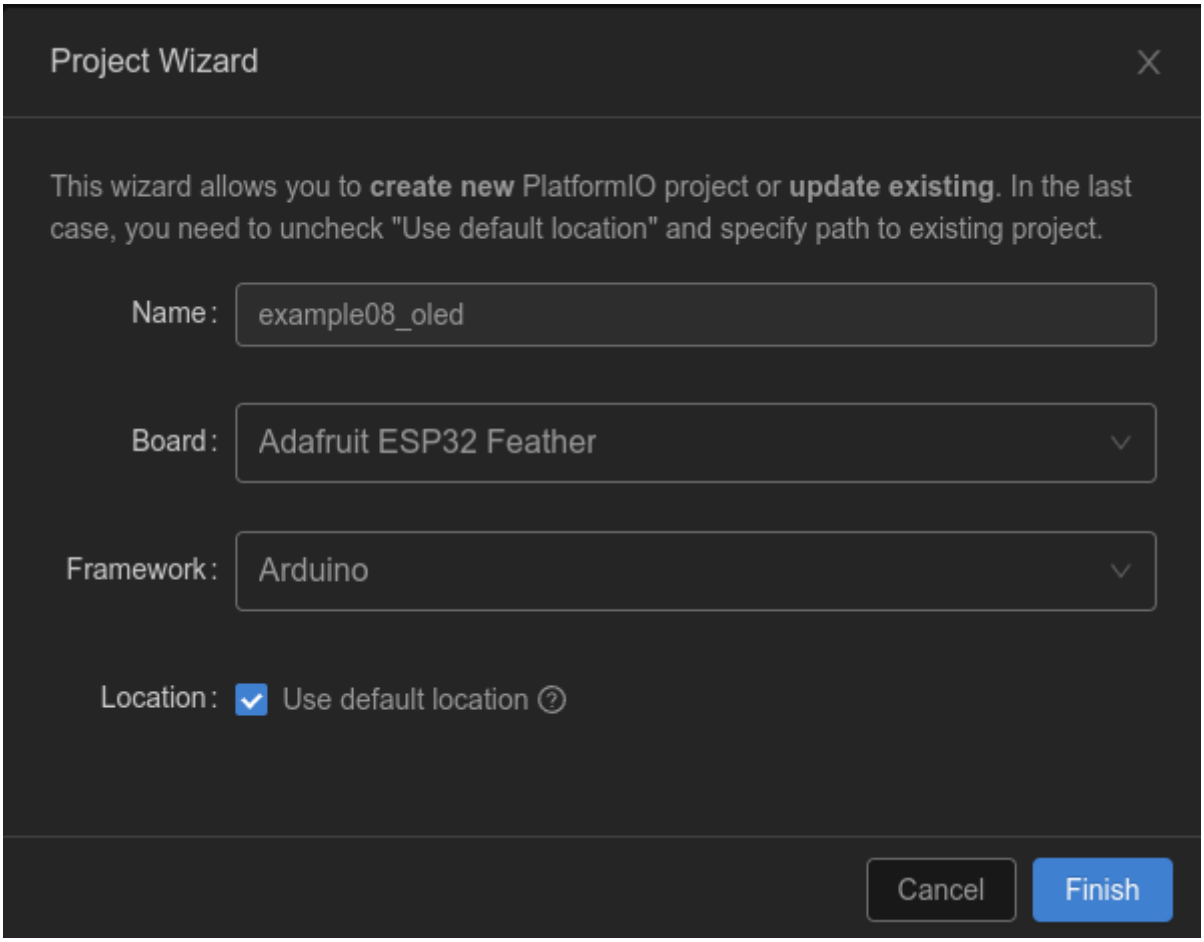
0.96 OLED

我们的OLDE模块样子如上图所示，整个屏幕有128*64个像素点，我们可以实现对每一个像素点的亮灭控制，以此实现对屏幕显示内容的控制。注意我们并不能控制屏幕上像素的颜色，所以我们OLED一般是单色的。

那我们如何控制它的亮灭呢，可以看到在OLED的上方一共有四个引脚，从左到右依次是GND、VCC、SCL、SDA，其中GND、VCC是用于OLED的供电使用，SCL和SDA是I2C通信使用。

听到I2C通信是不是觉得很熟悉，毕竟上一节驱动MPU6050时我们就是使用的I2C协议（Wrie），别着急，我们先用着，下一节我们再详细介绍I2C通信。

二、新建工程并安装依赖



安装依赖，可以直接修改 `platformio.ini`

```ini
[env:featheresp32]
platform = espressif32
board = featheresp32
framework = arduino
lib_deps =
    https://ghproxy.com/https://github.com/rfetick/MPU6050_light.git
    adafruit/Adafruit SSD1306@^2.5.7
```

接着打开IMU的源码目录，

将 `.pio/libdeps/featheresp32/MPU6050_light/examples/GetAngle/GetAngle.ino` 文件内容复制到main.cpp中，接着修改波特率和I2C地址。

```c++
#include "Wire.h"
#include <MPU6050_light.h>

MPU6050 mpu(Wire);
unsigned long timer = 0;

void setup()
{
  Serial.begin(115200);
  Wire.begin(18, 19);

  byte status = mpu.begin();
  Serial.print(F("MPU6050 status: "));
  Serial.println(status);
  while (status != 0)
  {
  } // stop everything if could not connect to MPU6050

  Serial.println(F("Calculating offsets, do not move MPU6050"));
  delay(1000);
  // mpu.upsideDownMounting = true; // uncomment this line if the MPU6050 is
mounted upside-down
  mpu.calcOffsets(); // gyro and accelero
  Serial.println("Done!\n");
}

void loop()
{
  mpu.update();

  if ((millis() - timer) > 10)
  { // print data every 10ms
    Serial.print("X : ");
    Serial.print(mpu.getAngleX());
    Serial.print("\tY : ");
    Serial.print(mpu.getAngleY());
    Serial.print("\tZ : ");
    Serial.println(mpu.getAngleZ());
    timer = millis();
```

```
  }
}
```

三、使用Adafruit库驱动OLED

该库提供的驱动例程较为复杂，小鱼这里提供一个简易版本。

```c++
#include "Wire.h"
#include <Adafruit_GFX.h>     // 加载Adafruit_GFX库
#include <Adafruit_SSD1306.h> // 加载Adafruit_SSD1306库
Adafruit_SSD1306 display;     // 声明对象

void setup()
{
  Wire.begin(18, 19);
  display = Adafruit_SSD1306(128, 64, &Wire);
  display.begin(SSD1306_SWITCHCAPVCC, 0x3C); // 设置OLED的I2C地址，默认0x3C
  display.clearDisplay();                    // 清空屏幕
  display.setTextSize(2);                    // 设置字体大小，最小为1
  display.setCursor(0, 0);                   // 设置开始显示文字的坐标
  display.setTextColor(SSD1306_WHITE);       // 设置字体颜色
  display.println("hello oled!");             // 输出的字符
}

void loop()
{

}
```

根据上面的简易版本，修改原有的IMU代码，最后得到如下代码

```c++
/* Get tilt angles on X and Y, and rotation angle on Z
 * Angles are given in degrees
 *
 * License: MIT
 */
```

```
#include "Wire.h"
#include <MPU6050_light.h>
#include <Adafruit_GFX.h>     // 加载Adafruit_GFX库
#include <Adafruit_SSD1306.h> // 加载Adafruit_SSD1306库

Adafruit_SSD1306 display;
MPU6050 mpu(Wire);
unsigned long timer = 0;

void setup()
{
  Serial.begin(115200);
  Wire.begin(18, 19);
  /*=======================OLED初始化====================================*/
  display = Adafruit_SSD1306(128, 64, &Wire);
  display.begin(SSD1306_SWITCHCAPVCC, 0x3C); // 设置OLED的I2C地址
  display.clearDisplay();                    // 清空屏幕
  display.setTextSize(2);                    // 设置字体大小
  display.setCursor(0, 0);                   // 设置开始显示文字的坐标
  display.setTextColor(SSD1306_WHITE);       // 设置字体颜色
  display.println("hello oled!");            // 输出的字符
  display.display();

  /*=======================IMU初始化=====================================*/
  byte status = mpu.begin();
  Serial.print(F("MPU6050 status: "));
  Serial.println(status);
  while (status != 0)
  {
  } // stop everything if could not connect to MPU6050

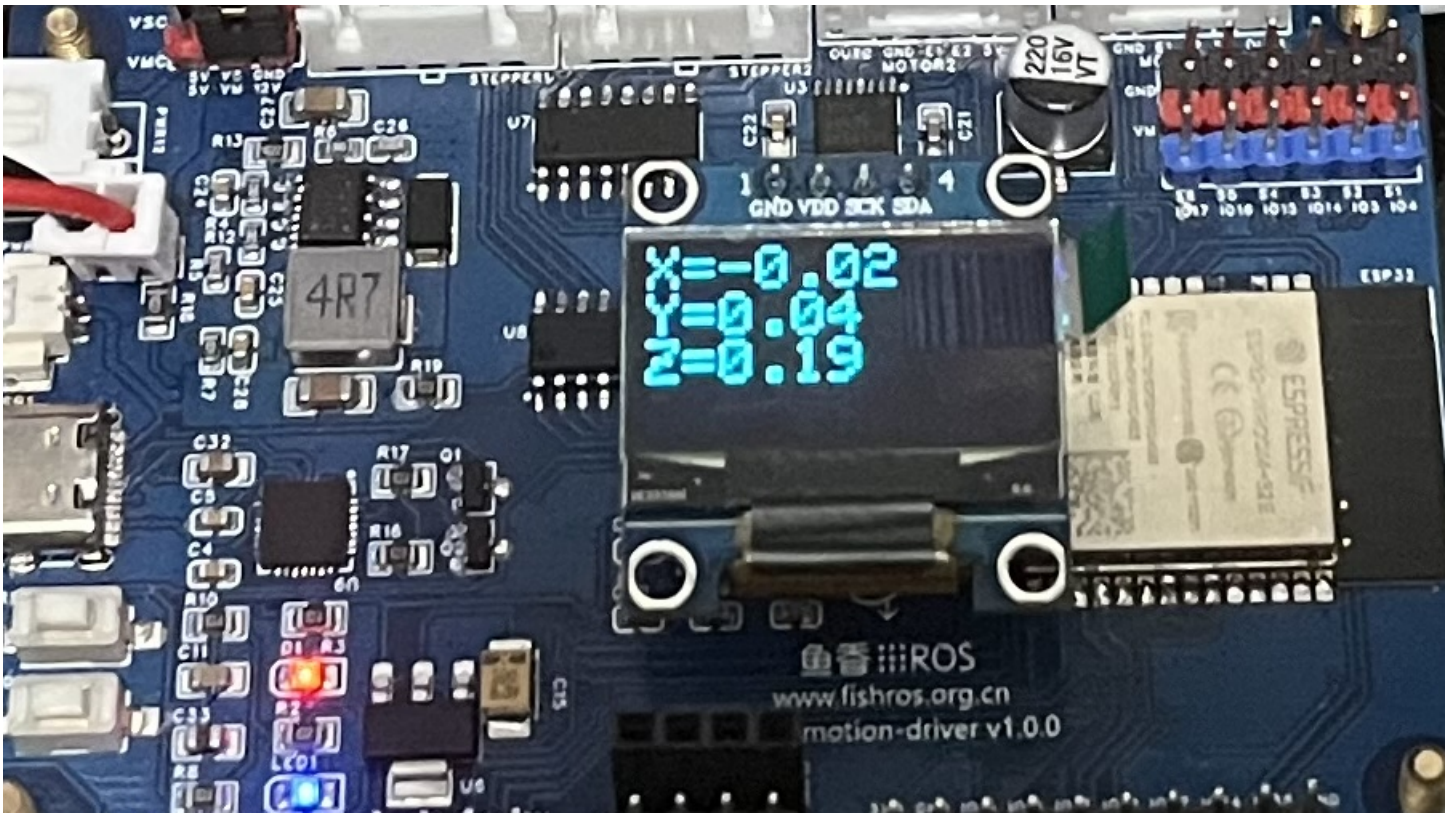
  Serial.println(F("Calculating offsets, do not move MPU6050"));
  delay(1000);
  // mpu.upsideDownMounting = true; // uncomment this line if the MPU6050 is
mounted upside-down
  mpu.calcOffsets(); // gyro and accelero
  Serial.println("Done!\n");
}

void loop()
{
  mpu.update();
```

```
    if ((millis() - timer) > 100)
    { // print data every 100ms
      Serial.print("X : ");
      Serial.print(mpu.getAngleX());
      Serial.print("\tY : ");
      Serial.print(mpu.getAngleY());
      Serial.print("\tZ : ");
      Serial.println(mpu.getAngleZ());
      timer = millis();
      /*=========================OLED显示===========================*/
      display.clearDisplay();  // 清空屏幕
      display.setCursor(0, 0); // 设置开始显示文字的坐标
      display.print("X=");   // 输出X
      display.println(mpu.getAngleX());
      display.print("Y="); // 输出Y
      display.println(mpu.getAngleY());
      display.print("Z="); // 输出Z
      display.println(mpu.getAngleZ());
      display.display();
    }
  }
```

四、下载测试

接上OLED，将代码编译下载到开发板上，观察OLED的显示。



五、总结

本节依然是很轻松的完成了OLED驱动，但你应该有个疑问，为什么OLED和MPU6050代码里都有这么一句 `Wire.begin(18, 19);` ，为什么都是18和19，不能是其他的数值吗？带着疑惑，下一节小鱼带你一起探秘I2C通信以及原理图。

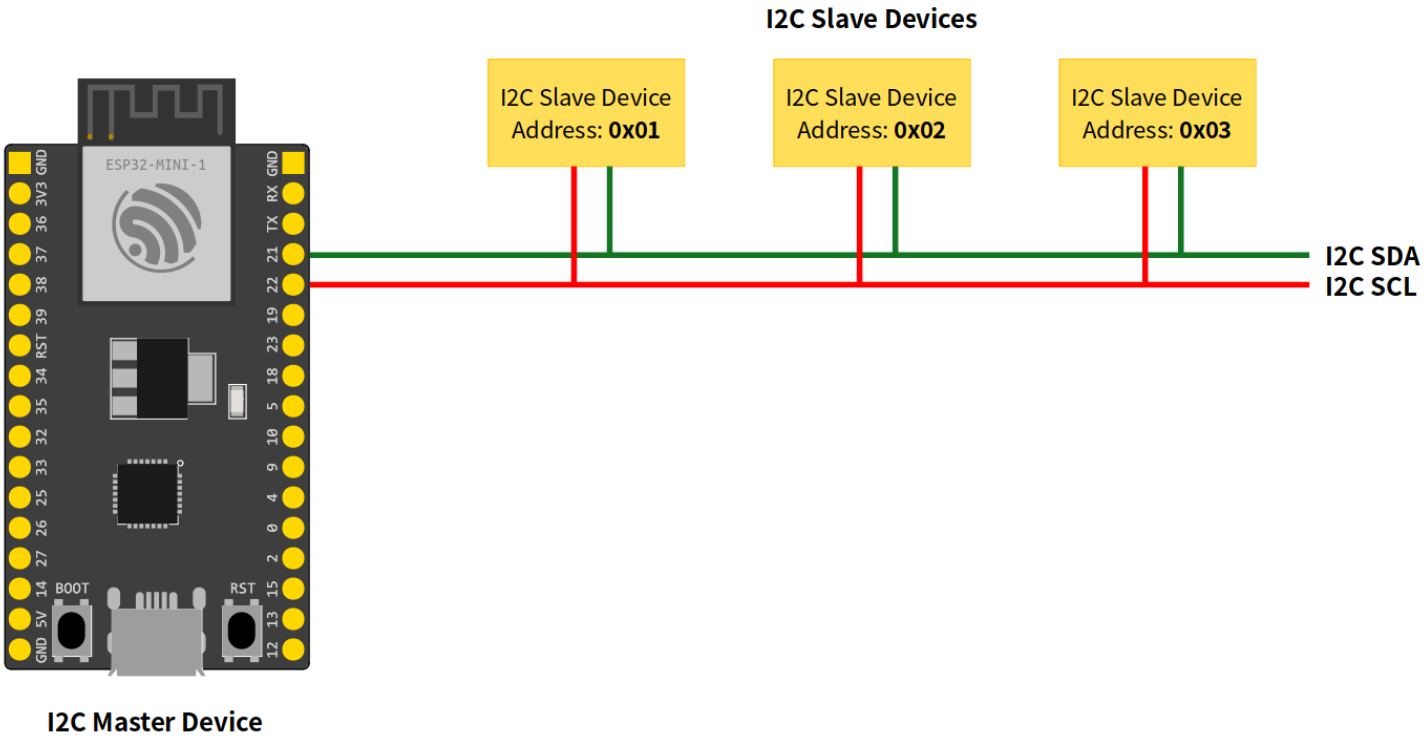
5.通讯协议小课堂-I2C通信

你好，我是爱吃鱼香ROS的小鱼。本节主要介绍I2C协议，并将其主要特点拎出来和大家说说。

网上关于I2C协议介绍的有很多，但大都非常细致，将高低电平数据协议等等都介绍了，但新手学习时不用关注那么多底层的东西，比起了解通信原理，了解它是什么，怎么用对我们更加重要。

一、I2C是一种总线通讯协议

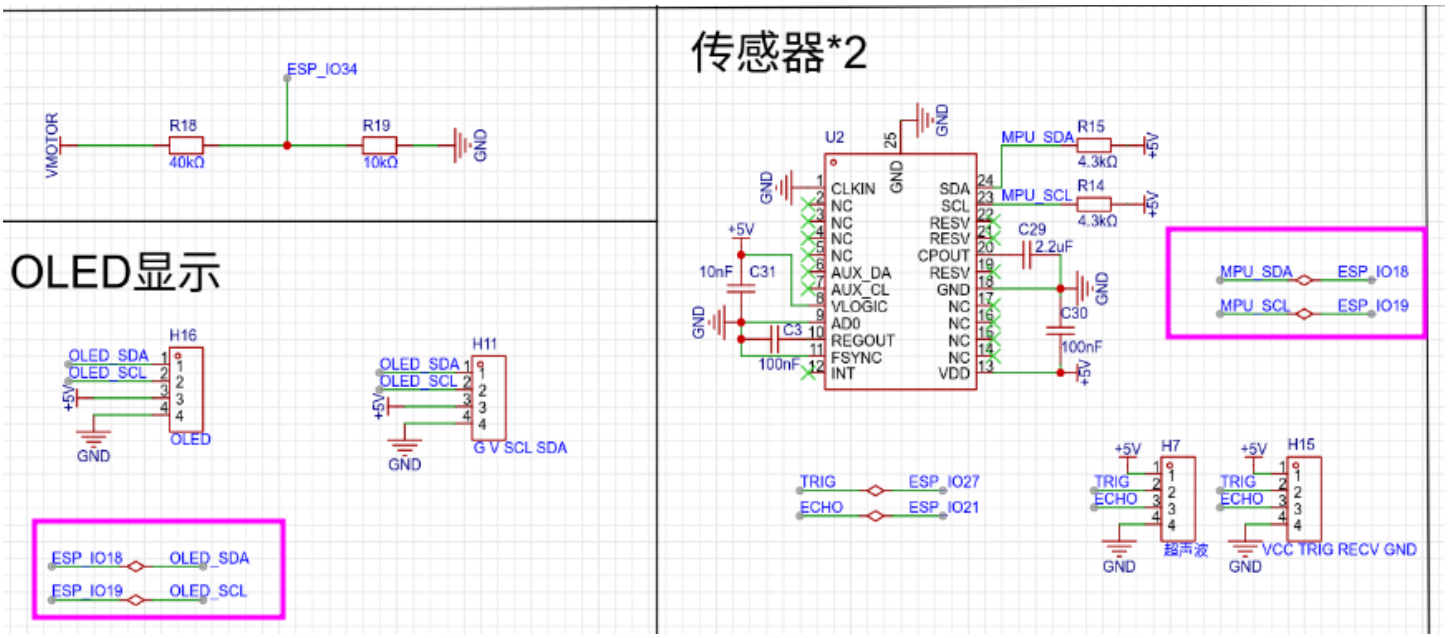
和之前我们介绍的串口通信类似，I2C也是一种通信协议。但它是一种总线通讯协议，也就是说，一个I2C可以连接多个设备，物理连接上像这样。



那它串的设备数量有没有上限呢？有的，上限是127个。

那为什么我们都是用18和19来驱动OLED和MPU6050呢？原因很简单，他们都是连在同一跟线上。

上原理图



> 像ESP32这种引脚资源匮乏的单片机，引脚的复用就显得很重要了，这里我们就将OLED和MPU6050都接在同一个I2C上，减少引脚占用。

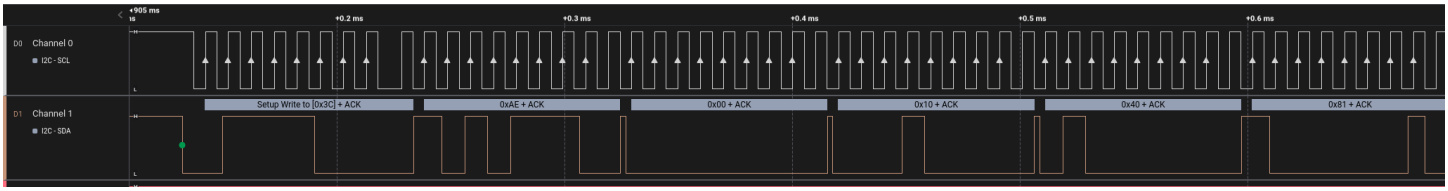
二、I2C是一种两线协议

在上几节我们驱动OLED和IMU的时候，都导入了"Wire.h"，这个就是Arudino提供的好的头文件。我们使用的Wire的定义如下

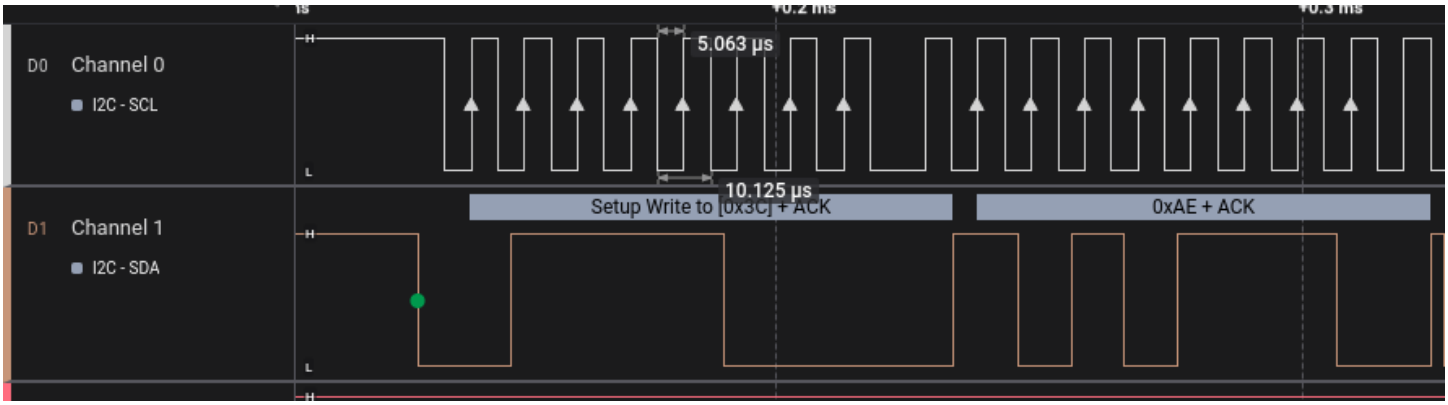
```
TwoWire Wire = TwoWire(0);
```

TwoWire也就是两线的意思，两线也就是一个是SCL，一个是SDA

SCL即时钟线，以一个固定的周期进行电平变换，SDA即数据线，用于数据的传输，这样说有些抽象，小鱼用逻辑分析仪**截取了一段OLED初始化时的SCl（GPIO18）和SDA（GPIO19）上的电平变化**，并分析出其对应的数据。



再放大一些看



可以直观的看到SCL在数据传输开始前一直保持高电平，SDA变成了低电平，这意为着数据传输的开始，也就是那个绿点的地方。

根据I2C协议规定：SCL处于高电平时，SDA由高到低变化，这种信号是起始信号。

继续观察，你会发现，在**传输过程中SCL引脚电平在以一个固定的周期来回跳变**，从机这边**当检测到SCL电平跳变后就开始读取SDA上的的电平，高记为1,低记为0**。所以你可以看到逻辑分析仪检测到**八个上升箭头，一共传输了八位数据，这八位数据用16进制表示就是0X3C**。

0X3C是什么，就是我们OLED的设备地址，所以这段信号的意思就是，我接下来要给0X3C的设备发数据了，请编号为0X3C的设备准备接收。

这个信号会被0X3C的设备接收和处理，而同一总线上ID非0X3C的接收到数据后就会将其扔掉，这就是I2C支持多个设备的核心原因。

在我们的开发板上，OLED的默认地址为：0X3C，IMU的默认地址为：0x68

> 附件:OLED初始化时I2C引脚数据图

	Type	Start	Duration	ack	address	read	data
■	start	905.132063 ms	62 ns				
■	address	905.142125 ms	91.188 µs	true	0x3C	false	
■	data	905.238375 ms	85.812 µs	true			0xAE
■	data	905.32925 ms	85.875 µs	true			0x00
■	data	905.420188 ms	85.875 µs	true			0x10
■	data	905.511125 ms	85.813 µs	true			0x40
■	data	905.602 ms	85.875 µs	true			0x81
■	data	905.692938 ms	85.875 µs	true			0xCF
■	data	905.783813 ms	85.875 µs	true			0xA1
■	data	905.87475 ms	85.875 µs	true			0xC8
■	data	905.965688 ms	85.813 µs	true			0xA6
■	data	906.056562 ms	85.875 µs	true			0xA8
■	data	906.1475 ms	85.875 µs	true			0x3F
■	data	906.238375 ms	85.875 µs	true			0xD3
■	data	906.329313 ms	85.875 µs	true			0x00
■	data	906.42025 ms	85.812 µs	true			0xD5
■	data	906.511125 ms	85.875 µs	true			0x80
■	data	906.602063 ms	85.875 µs	true			0xD9
■	data	906.692938 ms	85.875 µs	true			0xF1
■	data	906.783875 ms	85.875 µs	true			0xDA
■	data	906.874813 ms	85.812 µs	true			0x12
■	data	906.965688 ms	85.875 µs	true			0xDB
■	data	907.056625 ms	85.875 µs	true			0x40
■	data	907.147562 ms	85.875 µs	true			0x20
■	data	907.238438 ms	85.875 µs	true			0x00
■	data	907.329375 ms	85.875 µs	true			0x8D
■	data	907.420313 ms	85.813 µs	true			0x14
■	data	907.511188 ms	85.875 µs	true			0xA4
■	data	907.602125 ms	85.812 µs	true			0xA6
■	data	907.693 ms	86.375 µs	true			0xAF
■	stop	907.78975 ms	63 ns				

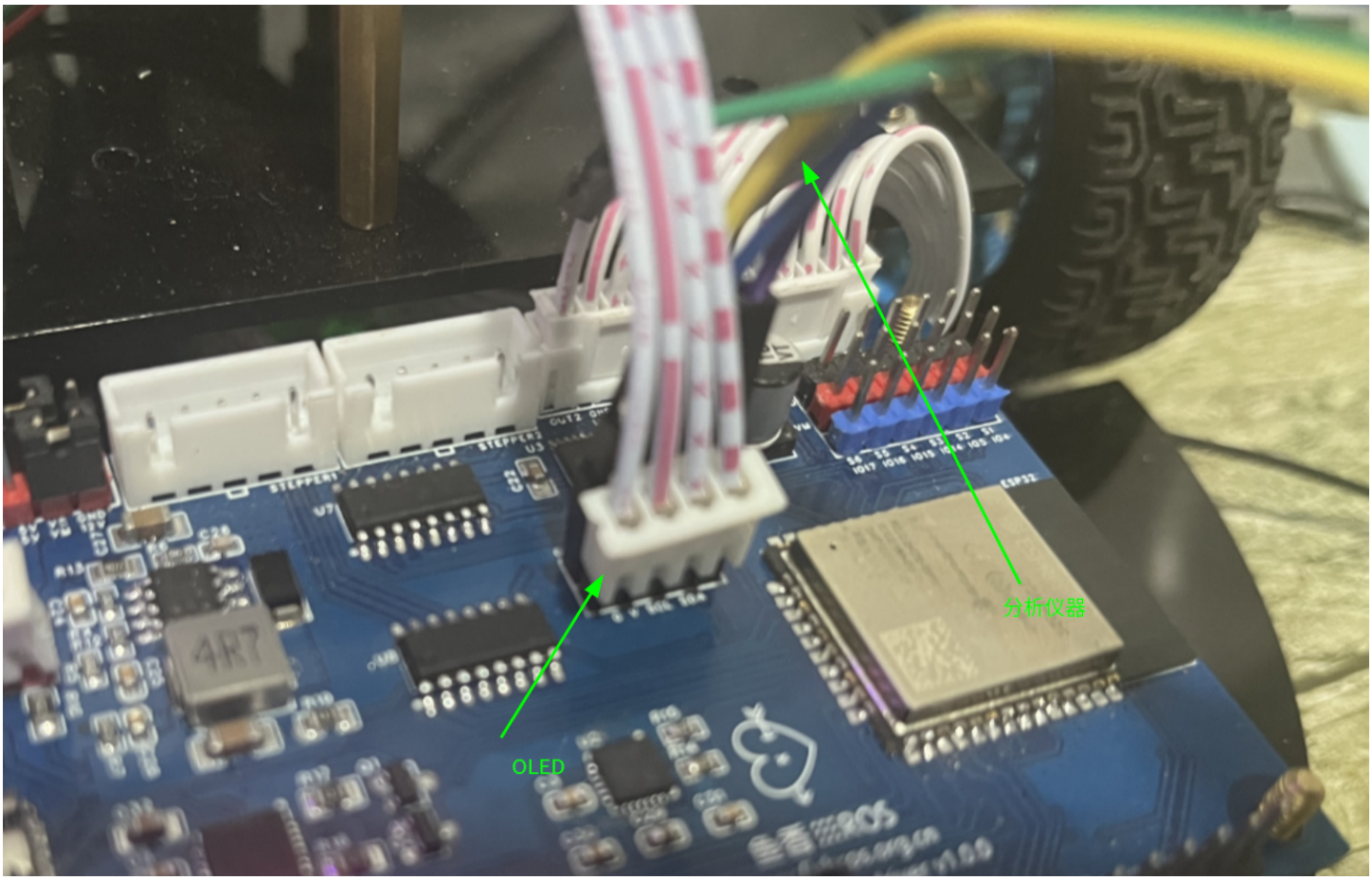
从该表可以看出，数据从905ms开始，到907ms完成了初始化数据的传输，下一节我们将按照这个数据传输内容编写代码，初始化OLED。

三、I2C是一种半双工协议

上面我们仔细分析了I2C协议，SCL用于周期的变换，SDA用于传输数据，所以同一时间，SDA要么用于接收，要么用于发送。所以我们可以得到I2C是一种半双工协议，同一时间只能进行接收或发送。

四、总结

本节小鱼带你从I2C协议信号入手，详细介绍了I2C数据的特点，如果你手头也有逻辑分析工具，也可以尝试进行分析，开发板设计的时也非常人性化，你可以这样随手拿几个杜邦线就可以接入进行测量。

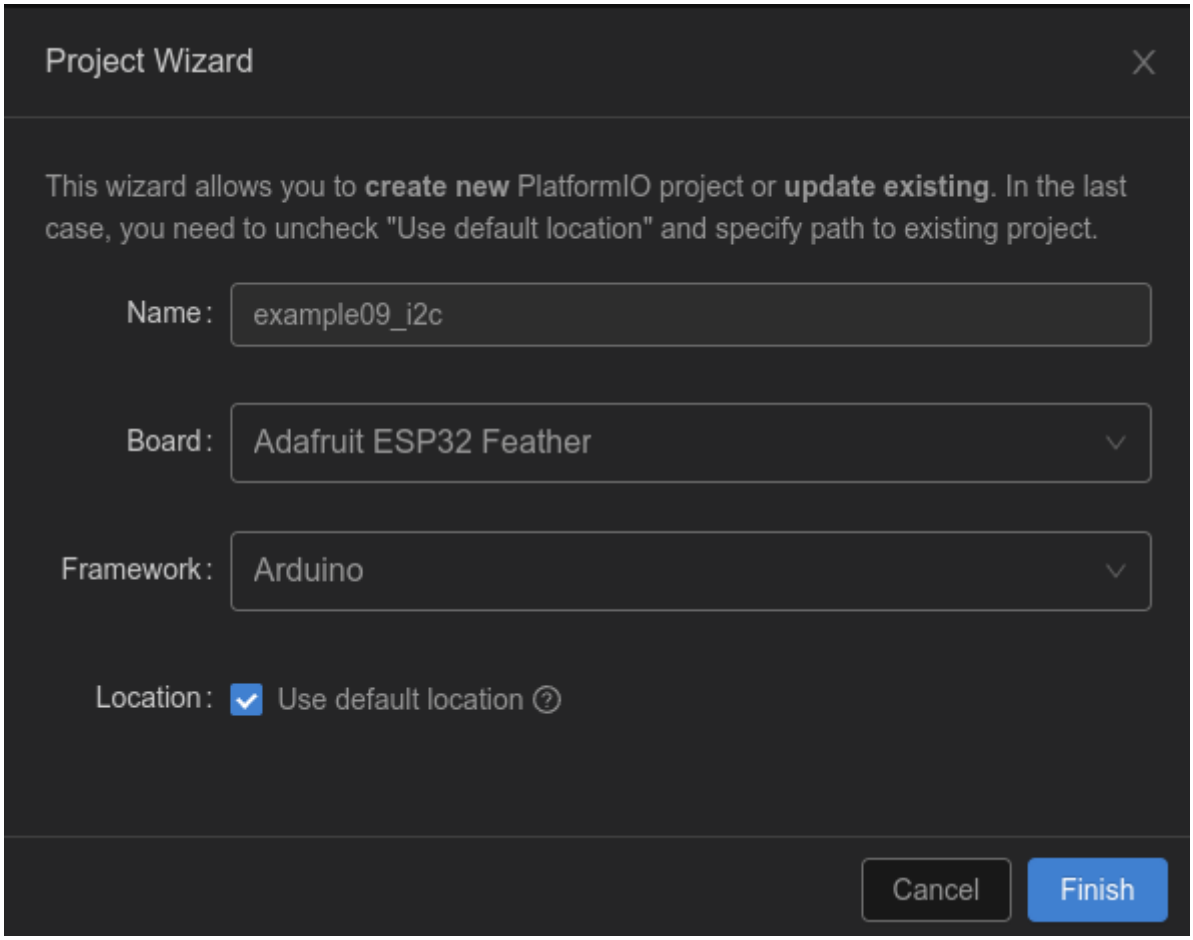

分析仪器
OLED

6.I2C通信实验-点亮OLED

你好，我是爱吃鱼香ROS的小鱼。本节我们就尝试直接使用I2C协议来点亮OLED，因为主要测试I2C协议，所以对于复杂的显示处理部分小鱼就掠过了，毕竟有方便的开源库使用，我们也不用那么纠结，如果实在想使用I2C直接驱动OLED，可以去看小鱼在雷达驱动板上手撸的代码（https://github.com/fishros/fishbot-laser-control/blob/main/components/oled/oled.c）。

一、新建工程

还是老样子，不过这次不需要添加任何依赖。



二、I2CAPI介绍

这里我们需要了解几个Wire常用的API。

1.Wire.begin 初始化可以设置引脚，如 `Wire.begin(18, 19);`

2.Wire.beginTransmission，开始传输，传入目标地址，如OLED-0x3C `Wire.beginTransmission(0x3c);`

3.Wire.write 开始写数据， 直接传如要写的数据即可

4.Wire.endTransmission 结束传输，写入完成后调用

三、编写代码

```c++
#include <Arduino.h>
#include "Wire.h"

uint8_t cmd_ssd1315[] = {0xae, 0x00, 0x10, 0x40, 0x81, 0xcf, 0xa1, 0xc8, 0xa6,
0xa8,
                         0x3f, 0xd3, 0x00, 0xd5, 0x80, 0xd9, 0xf1, 0xda, 0x12,
0xdb,
                         0x40, 0x20, 0x00, 0x8d, 0x14, 0xa4, 0xa6, 0xaf};

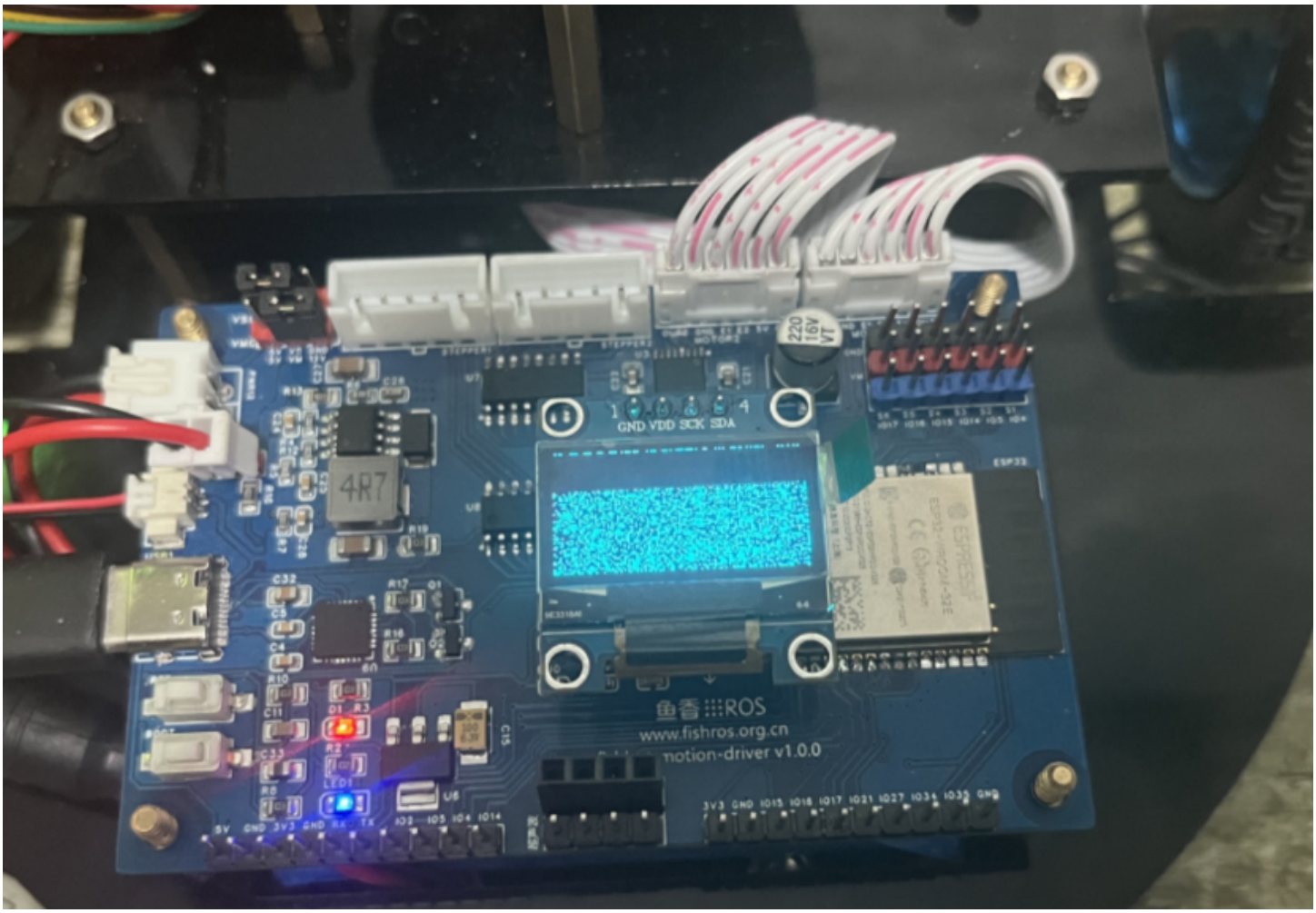
void setup()
{
  Wire.begin(18, 19);
  Wire.beginTransmission(0x3c);
  for (int i = 0; i < 28; i++)
  {
    Wire.write(cmd_ssd1315[i]);
  }
  Wire.endTransmission();
}

void loop()
{
  // put your main code here, to run repeatedly:
}
```

这里的代码，我们将 `cmd_ssd1315` 中的数据依次通过I2C写入地址为0x3C的设备中，这些数据分别代表这初始化OLED所需要的参数，如亮度、方向等信息，具体请参考SSD1315的手册，或自行百度。

四、测试

将代码下载到开发板，查看OLED,可以看到OLED此时成功亮起，有很多半点，属于正常现象。



五、总结

本节我们主要尝试通过I2C尝试发送一段指令给OLED，并将其成功点亮。至此我们嵌入式开发基础篇的内容就全部完成了，下一章开始我们将正是进入MicroROS的开发中来，到时更有我们熟悉的WIFI、蓝牙等模块的使用。

第十四章 接入ROS2-MicroROS

你好，我是小鱼。上一章我们通过对嵌入式开发的学习，了解了如果通过代码来和硬件打交道，但如果你想拥抱ROS的强大生态，还需要把我们的硬件系接入到ROS中来，MicroROS就是帮你完成这一事情的一大利器。所以本章的重点就是带你一起学习MicroROS的使用。

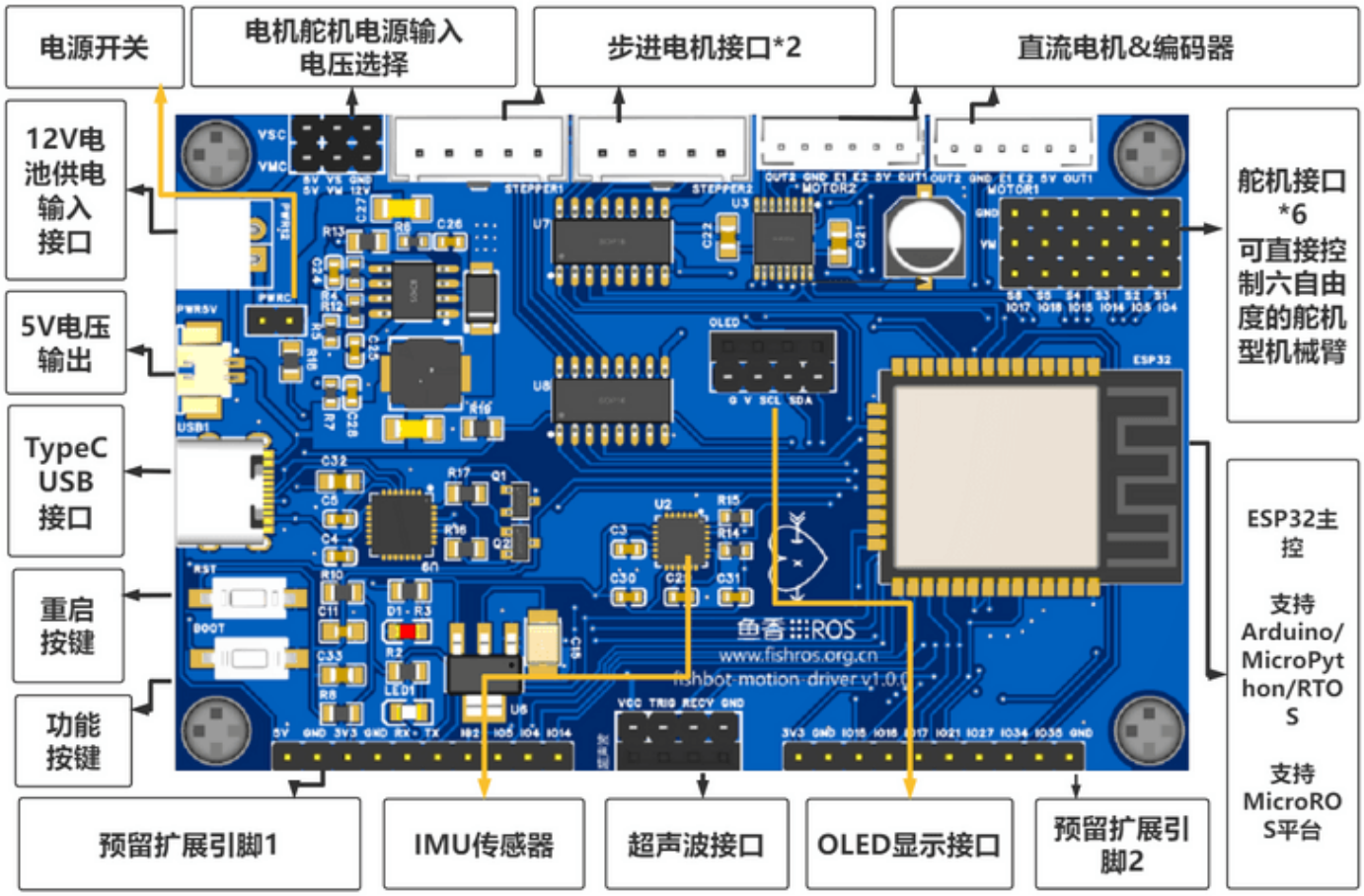
- 基础篇，如果你不了解MicroROS，不知道如何安装以及启动，那么建议你从基础篇开始。
- 入门篇，主要介绍使用MicroROS实现话题的订阅与发布，以及服务通信的实现。
- 进阶篇，通过对MicroROS配置以及开发中常用点的介绍，带你深入了解MicroROS。

友情提示：和我们课程的名称一样，本教程中所有代码都非常建议跟着敲一遍，不要直接复制粘贴，写了才是自己的。

硬件开发平台

为方便学习，本教程配套的硬件是小鱼自制的MicroROS学习板，同时该板可以作为下一章节搭建实体移动机器人的主控板以及后续制作机械臂的驱动板使用。

板载资源图如下：



该主控板可以在小鱼的店铺直接购买，性价比接地气，直达链接：https://item.taobao.com/item.htm?id=695473143304。

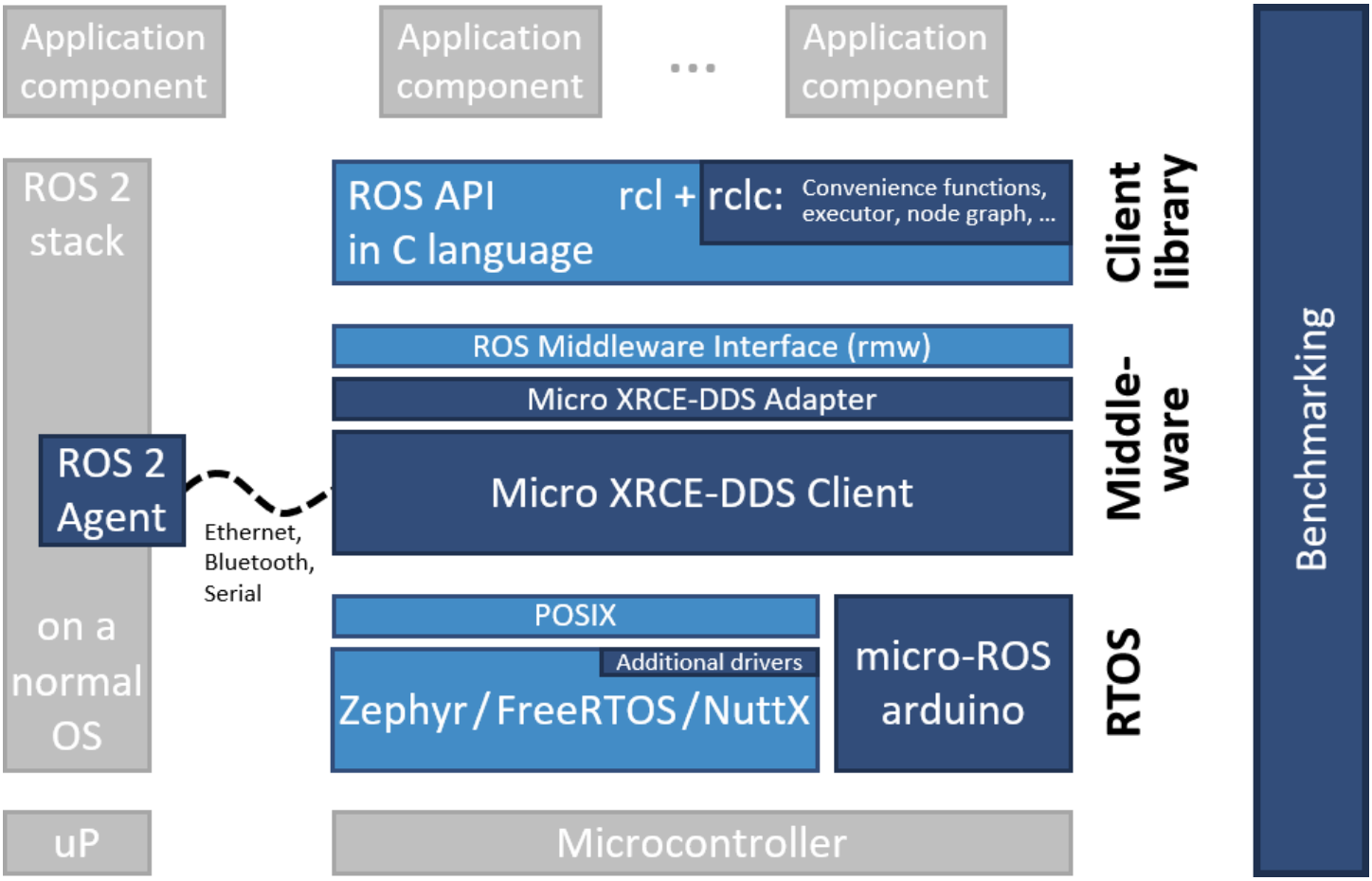
同时该板的原理图和PCB小鱼都已经开源了，时间充足且动手能力强工具齐全的的小伙伴可以自己购买元器件制作，开源地址见鱼香ROS社区：https://www.fishros.org.cn/forum/topic/894

技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**
- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

1.Micro-ROS介绍与服务安装

你好，我是爱吃鱼香ROS的小鱼。本节我们主要介绍下Micro-ROS几大主要特点。



先上系统框架图，下面再一一介绍。

一、特点1：运行在微控制器上的ROS2

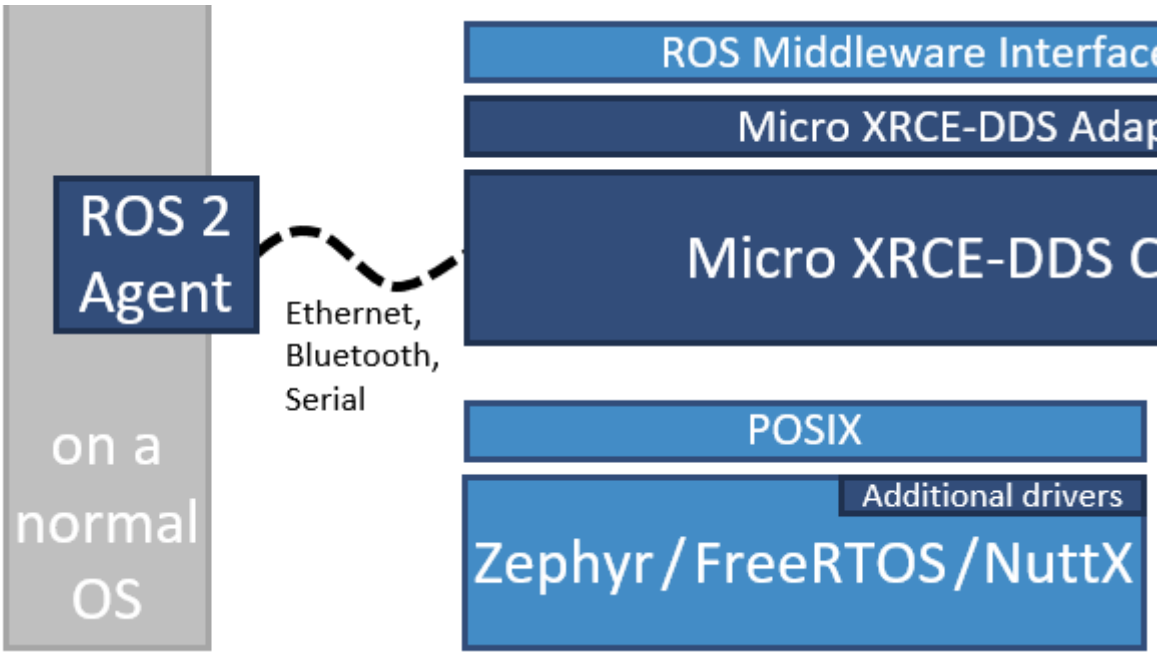
首先从名称看，Micro-ROS，Micro指的就是 `microcontrollers` 即微控制器。

micro-ROS puts ROS 2 onto microcontrollers

核心作用就是上面这句话`micro-ROS puts ROS 2 onto microcontrollers`。既然是在微控制器上，因硬件资源受限，其功能肯定会有所裁剪，但核心的ROS2通信功能依然保有。

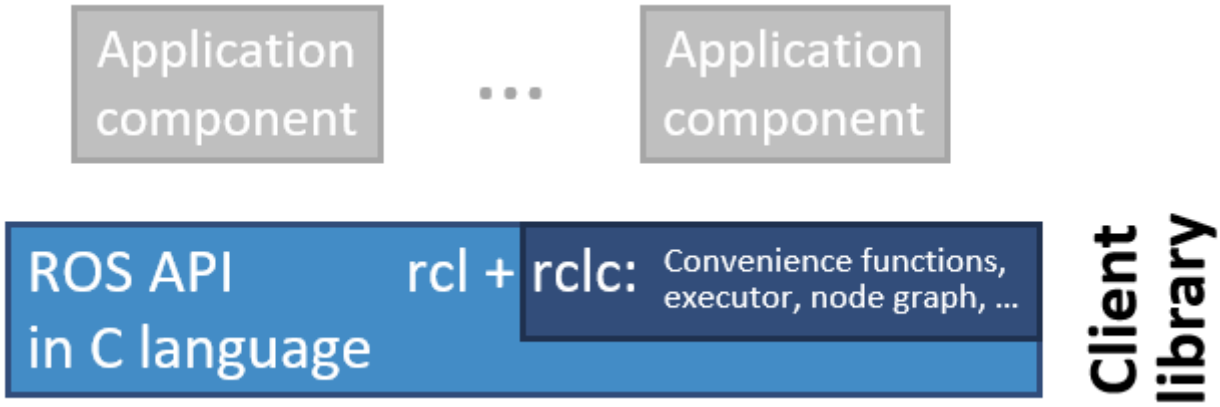


二、特点2：MicroROS支持多种通信协议并依赖Agent



所谓Agen其实就是一个代理，**微控制器可以通过串口，蓝牙、以太网、Wifi等多种协议将数据传递给Agent**，Agent再将其转换成ROS2的话题等数据，以此完成通信。

三、特点3：通过RCLC-API调用MicroROS



因为MicroROS遵循RCLCAPI，所以和在上位机中使用Python或者C++调用MicroROS有所不同，最终代码风格如下面这段所示

```c++
#include <rcl/rcl.h>
#include <rclc/rclc.h>
#include <rclc/executor.h>

rclc_executor_t executor;
rclc_support_t support;
rcl_allocator_t allocator;
```

```
rcl_node_t node;
rcl_timer_t timer;


void setup() {
  // Configure serial transport
  Serial.begin(115200);
  set_microros_serial_transports(Serial);
  delay(2000);

  allocator = rcl_get_default_allocator();

  //create init_options
  RCCHECK(rclc_support_init(&support, 0, NULL, &allocator));

  // create node
  RCCHECK(rclc_node_init_default(&node, "micro_ros_platformio_node", "",
&support));

  // create publisher
  RCCHECK(rclc_publisher_init_default(
    &publisher,
    &node,
    ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, Int32),
    "micro_ros_platformio_node_publisher"));


  // create executor
  RCCHECK(rclc_executor_init(&executor, &support.context, 1, &allocator));
  RCCHECK(rclc_executor_add_timer(&executor, &timer));

  msg.data = 0;
}
```

四、在上位机上安装Agent

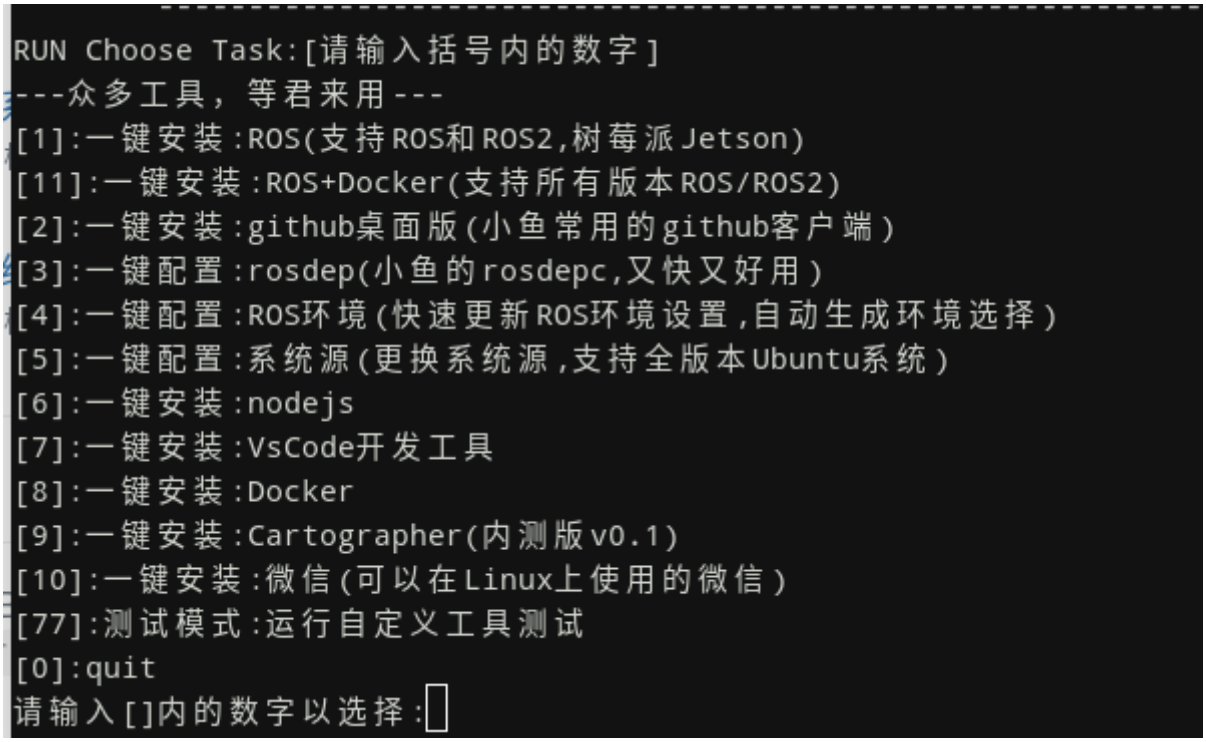
我们使用Docker来进行Agent的安装。

4.1 安装Docker

打开终端，复制粘贴输入下面代码

```
wget http://fishros.com/install -O fishros && . fishros
```

接着输入密码，在下面的界面输入8,一键安装Docker,完成后等待即可。



4.2 运行Agent

安装完成Docker后打开终端，输入下面的指令

```
sudo docker run -it --rm -v /dev:/dev -v /dev/shm:/dev/shm --privileged --net=host
microros/micro-ros-agent:$ROS_DISTRO serial --dev /dev/ttyUSB0 -v6
```

稍微等待下载完成，看到如下界面表示成功启动。

```
fishros@fishros-MS-7D42:~$ docker run -it --rm -v /dev:/dev -v /dev/shm:/dev/shm
 --privileged --net=host microros/micro-ros-agent:$ROS_DISTRO serial --dev /dev/
ttyUSB0 -v6
[1674232832.262144] info     | TermiosAgentLinux.cpp | init
| Serial port not found. | device: /dev/ttyUSB0, error 2, waiting for connection
...
[1674232833.264643] info     | TermiosAgentLinux.cpp | init
| Serial port not found. | device: /dev/ttyUSB0, error 2, waiting for connection
...
[1674232834.266831] info     | TermiosAgentLinux.cpp | init
| Serial port not found. | device: /dev/ttyUSB0, error 2, waiting for connection
...
[1674232835.270785] info     | TermiosAgentLinux.cpp | init
| Serial port not found. | device: /dev/ttyUSB0, error 2, waiting for connection
...
```

上面的指令是使用串口通讯协议运行microros-agent，还可以通过UDP、TCP、CAN等协议运行，具体指令如下

```shell
# UDPv4 micro-ROS Agent
docker run -it --rm -v /dev:/dev -v /dev/shm:/dev/shm --privileged --net=host
microros/micro-ros-agent:$ROS_DISTRO udp4 --port 8888 -v6

# Serial micro-ROS Agent
docker run -it --rm -v /dev:/dev -v /dev/shm:/dev/shm --privileged --net=host
microros/micro-ros-agent:$ROS_DISTRO serial --dev [YOUR BOARD PORT] -v6

# TCPv4 micro-ROS Agent
docker run -it --rm -v /dev:/dev -v /dev/shm:/dev/shm --privileged --net=host
microros/micro-ros-agent:$ROS_DISTRO tcp4 --port 8888 -v6

# CAN-FD micro-ROS Agent
docker run -it --rm -v /dev:/dev -v /dev/shm:/dev/shm --privileged --net=host
microros/micro-ros-agent:$ROS_DISTRO canfd --dev [YOUR CAN INTERFACE] -v6
```

五、总结

本节我们主要介绍了MicroROS的主要特点，接着介绍使用Docker下载和运行Agent，既然搞定了上位机，下一节我们正是开始在开发板上编写MicroROS节点，然后测试与上位机的连接是否正常。

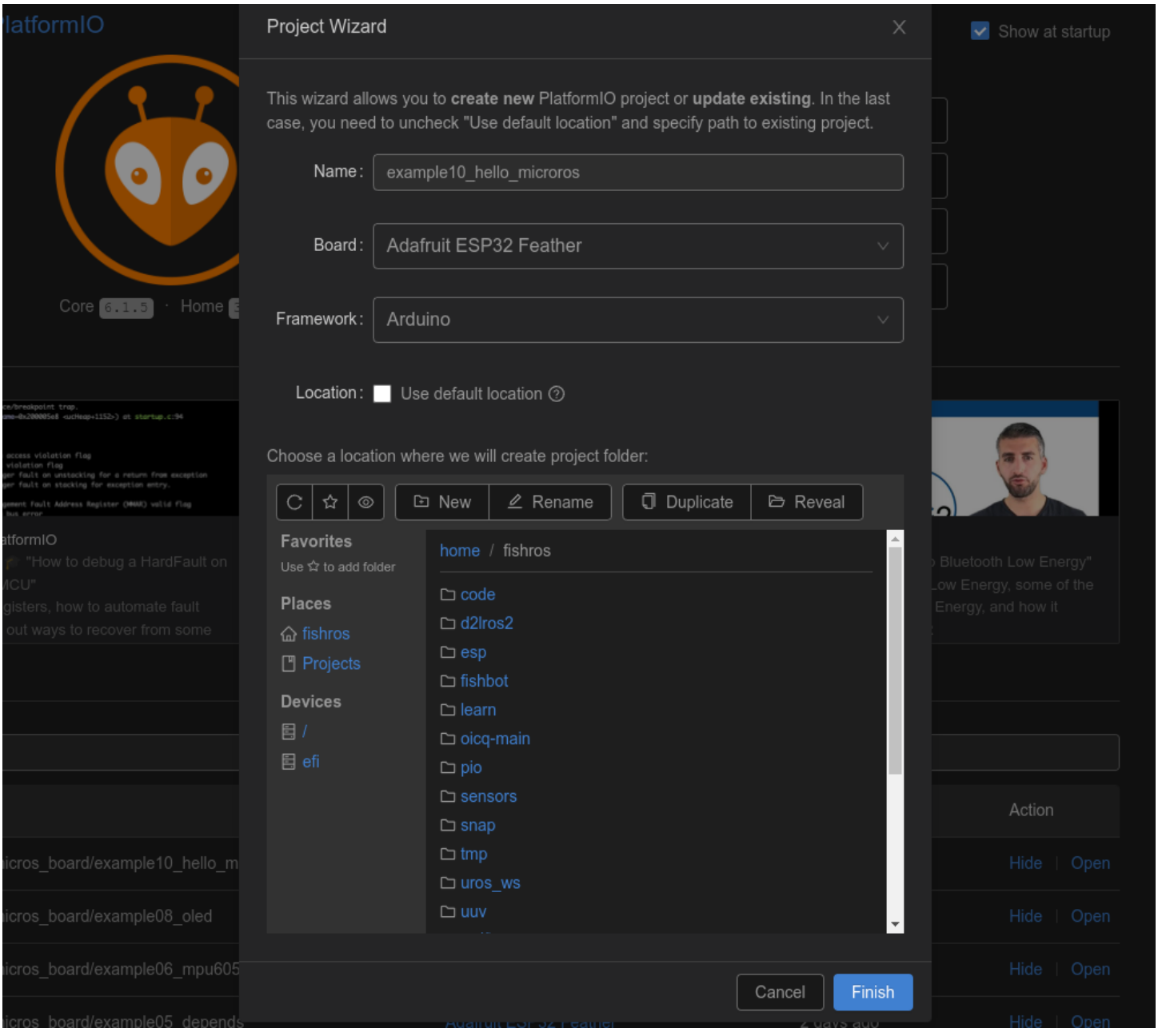
2.你的第一个MicroROS节点

你好，我是爱吃鱼香ROS的小鱼。上一节我们介绍了MicroROS和Agent的安装，本节我们开始正是编写代码，接入ROS2。

一、新建工程添加依赖

1.1 新建工程

新建`example10_hello_microros`工程，这里需要更改下工程的位置，默认目录是在文档目录下，小鱼在测试时发现目录定位上有bug，所以建议建议直接放到主目录或其下目录，小鱼这里直接放到主目录。



1.2 添加依赖

打开 `platform.ini` ,接着我们添加MicroROS的依赖。

```ini
[env:featheresp32]
platform = espressif32
board = featheresp32
framework = arduino
lib_deps =
    https://gitee.com/ohhuo/micro_ros_platformio.git
```

这里使用的地址并不是MicroROS官方仓库，而是小鱼经过修改后的国内仓库地址，里面放置了小鱼编译好后可以直接使用的microros静态库，并对仓库中需要梯子的地址进行了替换。

二、编写代码-第一个节点

开始编写代码，因为Micro-ROS遵循RCLC-API，所以这里小鱼通过一个最简单的例程介绍如何新建一个节点。

```c++
#include <Arduino.h>
#include <micro_ros_platformio.h>

#include <rcl/rcl.h>
#include <rclc/rclc.h>
#include <rclc/executor.h>

rclc_executor_t executor;
rclc_support_t support;
rcl_allocator_t allocator;
rcl_node_t node;

void setup()
{
  Serial.begin(115200);
  // 设置通过串口进行MicroROS通信
  set_microros_serial_transports(Serial);
  // 延时时一段时间，等待设置完成
  delay(2000);
  // 初始化内存分配器
```

```
  allocator = rcl_get_default_allocator();
  // 创建初始化选项
  rclc_support_init(&support, 0, NULL, &allocator);
  // 创建节点 hello_microros
  rclc_node_init_default(&node, "hello_microros", "", &support);
  // 创建执行器
  rclc_executor_init(&executor, &support.context, 1, &allocator);
}

void loop()
{
  delay(100);
  // 循环处理数据
  rclc_executor_spin_some(&executor, RCL_MS_TO_NS(100));
}
```

上面代码并不复杂，小鱼已经将注释写上，强烈建议你跟着代码敲一遍，不要直接复制粘贴。

相比在上位机中开发ROS，这里多了几步

- 设置通信协议，因为可以通过多种方式连接，所以需要进行提前设置
- 初始化内存分配器，在微控制器上资源受限，内存的管理要很细致
- 创建初始化选项，用于初始化rcl并创建一些需要用到的数据结构体

关于rclc的api小鱼并没有找到文档，不过源码的头文件依然非常清晰，直接安装Ctrl点击某个函数即可跳转（不行的，重启下Vscode）。

比如关于rclc_support_init 的源码及参数介绍。

c++

```
/**
 *  Initializes rcl and creates some support data structures.
 *  Initializes clock as RCL_STEADY_TIME.
 *  * <hr>
 * Attribute          | Adherence
 * ------------------ | -------------
 * Allocates Memory   | Yes (in RCL)
 * Thread-Safe        | No
 * Uses Atomics       | No
 * Lock-Free          | Yes
 *
```

```
 * \param[inout] support a zero-initialized rclc_support_t
 * \param[in] argc number of args of main
 * \param[in] argv array of arguments of main
 * \param[in] allocator allocator for allocating memory
 * \return `RCL_RET_OK` if RCL was initialized successfully
 * \return `RCL_RET_INVALID_ARGUMENT` if any null pointer as argument
 * \return `RCL_RET_ERROR` in case of failure
 */
RCLC_PUBLIC
rcl_ret_t
rclc_support_init(
  rclc_support_t * support,
  int argc,
  char const * const * argv,
  rcl_allocator_t * allocator);
```

三、运行测试

连接开发板，编译下载，如果遇到端口被占用，多半是你的microros_agent没有关闭，Ctrl+C打断运行再次尝试。

```
Writing at 0x0000e000... (100 %)
Wrote 8192 bytes (47 compressed) at 0x0000e000 in 0.1 seconds (effective 706.3 kbit/s)...
Hash of data verified.
Compressed 296608 bytes to 170470...
Writing at 0x00010000... (9 %)
Writing at 0x0001c13c... (18 %)
Writing at 0x000246f4... (27 %)
Writing at 0x00029b76... (36 %)
Writing at 0x0002ef30... (45 %)
Writing at 0x000343cf... (54 %)
Writing at 0x0003a2e2... (63 %)
Writing at 0x0004301b... (72 %)
Writing at 0x0004b58a... (81 %)
Writing at 0x00050a72... (90 %)
Writing at 0x00056337... (100 %)
Wrote 296608 bytes (170470 compressed) at 0x00010000 in 4.4 seconds (effective 536.7 kbit/s)...
Hash of data verified.

Leaving...
Hard resetting via RTS pin...
============================================================ [SUCCESS] Took 8.58 seconds ============================
* 终端将被任务重用，按任意键关闭。
```

接着打开Agent

```
fishros@fishros-MS-7D42:~$ docker run -it --rm -v /dev:/dev -v /dev/shm:/dev/shm --privileged --net=host microros/mi
cro-ros-agent:$ROS_DISTRO serial --dev /dev/ttyUSB0 -v6
[1674234780.105699] info     | TermiosAgentLinux.cpp | init                     | running...             | fd: 3
[1674234780.105779] info     | Root.cpp              | set_verbose_level        | logger setup           | verbose_leve
l: 6
```

然而并没有什么反应，重新点击一次RST即可看到有数据发送和接收过来了。

```
fishros@fishros-MS-7D42:~$ docker run -it --rm -v /dev:/dev -v /dev/shm:/dev/shm --privileged --net=host microros/mi
cro-ros-agent:$ROS_DISTRO serial --dev /dev/ttyUSB0 -v6
[1674234780.105699] info     | TermiosAgentLinux.cpp | init                     | running...             | fd: 3
[1674234780.105779] info     | Root.cpp              | set_verbose_level        | logger setup           | verbose_leve
l: 6
[1674234844.680542] info     | Root.cpp              | create_client            | create                 | client_key:
0x61A77199, session_id: 0x81
[1674234844.680647] info     | SessionManager.hpp    | establish_session        | session established    | client_key:
0x61A77199, address: 0
[1674234844.680856] debug    | SerialAgentLinux.cpp  | send_message             | [** <<SER>> **]        | client_key
: 0x61A77199, len: 19, data:
0000: 81 00 00 00 04 01 0B 00 00 00 58 52 43 45 01 00 01 0F 00
[1674234844.690920] debug    | SerialAgentLinux.cpp  | recv_message             | [==>> SER <<==]        | client_key
: 0x61A77199, len: 56, data:
0000: 81 80 00 00 01 07 30 00 00 0A 00 01 01 03 00 00 22 00 00 00 00 01 00 00 1A 00 00 00 6D 69 63 72
0020: 6F 5F 72 6F 73 5F 70 6C 61 74 66 6F 72 6D 69 6F 5F 6E 6F 64 65 00 00 00
[1674234844.700017] info     | ProxyClient.cpp       | create_participant       | participant created    | client_key:
0x61A77199, participant_id: 0x000(1)
[1674234844.700222] debug    | SerialAgentLinux.cpp  | send_message             | [** <<SER>> **]        | client_key
: 0x61A77199, len: 14, data:
0000: 81 80 00 00 05 01 06 00 00 0A 00 01 00 00
[1674234844.700356] debug    | SerialAgentLinux.cpp  | send_message             | [** <<SER>> **]        | client_key
: 0x61A77199, len: 13, data:
0000: 81 00 00 00 0A 01 05 00 01 00 00 00 80
[1674234844.706953] debug    | SerialAgentLinux.cpp  | recv_message             | [==>> SER <<==]        | client_key
: 0x61A77199, len: 13, data:
0000: 81 00 00 00 0A 01 05 00 01 00 00 00 80
```

接着打开新的终端，输入指令

```shell
ros2 node list
ros2 node info /hello_microros
```

```
fishros@fishros-MS-7D42:~$ ros2 node list
/hello_microros
fishros@fishros-MS-7D42:~$ ros2 node info /hello_microros
/hello_microros
  Subscribers:

  Publishers:

  Service Servers:

  Service Clients:

  Action Servers:

  Action Clients:
```

可以看到，我们的第一个节点成功运行起来了。

四、总结

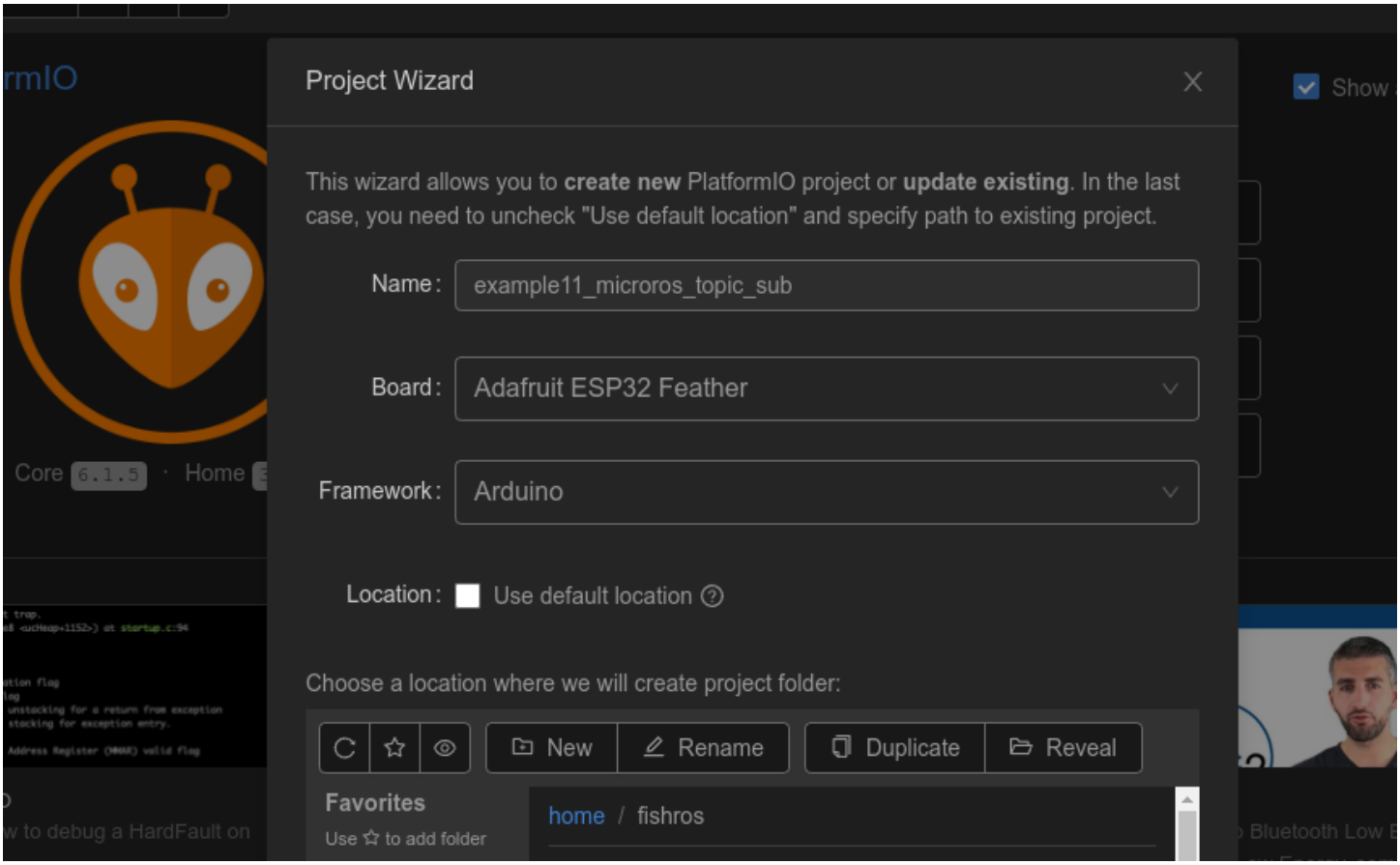
本节我们成功的在微控制器平台上将MicroROS节点运行起来了，下一节我们开始正式进行ROS2通信的学习。

1.话题订阅-控制LED

你好，我是爱吃鱼香ROS的小鱼。本节我们正式进入到MicroROS的核心通信部分的学习中来，本节我们将通过话题订阅实现，通过话题控制LED的亮灭。

一、新建工程添加依赖

新建`example11_microros_topic_sub`工程



修改`platformio.ini`添加依赖

```ini
; PlatformIO Project Configuration File
;
;   Build options: build flags, source filter
;   Upload options: custom upload port, speed and extra flags
;   Library options: dependencies, extra library storages
;   Advanced options: extra scripting
;
; Please visit documentation for the other options and examples
; https://docs.platformio.org/page/projectconf.html
```

```
[env:featheresp32]
platform = espressif32
board = featheresp32
framework = arduino
lib_deps =
    https://gitee.com/ohhuo/micro_ros_platformio.git
```

二、编写代码-实现订阅

编辑main.cpp，代码如下，注释小鱼已经添加到代码中来了

```c++
#include <Arduino.h>
#include <micro_ros_platformio.h>

#include <rcl/rcl.h>
#include <rclc/rclc.h>
#include <rclc/executor.h>

#include <std_msgs/msg/int32.h>

rclc_executor_t executor;
rclc_support_t support;
rcl_allocator_t allocator;
rcl_node_t node;
// 声明话题订阅者
rcl_subscription_t subscriber;
// 声明消息文件
std_msgs__msg__Int32 sub_msg;
// 定义话题接收回调函数
void callback_subscription_(const void *msgin)
{
  const std_msgs__msg__Int32 *msg = (const std_msgs__msg__Int32 *)msgin;
  if (msg->data == 0)
  {
    digitalWrite(2, HIGH);
  }
  else
  {
```

```
    digitalWrite(2, LOW);
  }
}

void setup()
{
  Serial.begin(115200);
  // 设置通过串口进行MicroROS通信
  set_microros_serial_transports(Serial);
  // 延时时一段时间，等待设置完成
  delay(2000);
  // 初始化内存分配器
  allocator = rcl_get_default_allocator();
  // 创建初始化选项
  rclc_support_init(&support, 0, NULL, &allocator);
  // 创建节点 topic_sub_test
  rclc_node_init_default(&node, "topic_sub_test", "", &support);
  // 订阅者初始化
  rclc_subscription_init_default(
      &subscriber,
      &node,
      ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, Int32),
      "led_control");
  // 创建执行器
  rclc_executor_init(&executor, &support.context, 1, &allocator);
  // 为执行器添加一个订阅者
  rclc_executor_add_subscription(&executor, &subscriber, &sub_msg,
&callback_subscription_, ON_NEW_DATA);
  // 初始化LED
  pinMode(2, OUTPUT);
}

void loop()
{
  delay(100);
  // 循环处理数据
  rclc_executor_spin_some(&executor, RCL_MS_TO_NS(100));
}
```

三、代码注解

相比之前的节点代码这里主要多了这几行

- `#include <std_msgs/msg/int32.h>` 添加消息类型头文件
- `rcl_subscription_t subscriber;` 声明话题订阅者
- `std_msgs__msg__Int32 sub_msg;` **声明消息文件，这一点和上位机不同，因为内存紧缺，所以提前定义**
- `void callback_subscription_(const void *msgin)` 接收到数据的回调函数
- `rclc_subscription_init_default` **初始化话题订阅者**
- `rclc_executor_add_subscription(&executor, &subscriber, &sub_msg, &callback_subscription_, ON_NEW_DATA);` **,为执行器添加一个订阅者**

四、下载测试

4.1 编译下载

连接开发板，编译下载。

```
Flash will be erased from 0x0000e000 to 0x0000ffff...
Flash will be erased from 0x00010000 to 0x0005afff...
Compressed 17472 bytes to 12125...
Writing at 0x00001000... (100 %)
Wrote 17472 bytes (12125 compressed) at 0x00001000 in 0.6 seconds (effective 244.7 kbit/s)...
Hash of data verified.
Compressed 3072 bytes to 128...
Writing at 0x00008000... (100 %)
Wrote 3072 bytes (128 compressed) at 0x00008000 in 0.0 seconds (effective 516.0 kbit/s)...
Hash of data verified.
Compressed 8192 bytes to 47...
Writing at 0x0000e000... (100 %)
Wrote 8192 bytes (47 compressed) at 0x0000e000 in 0.1 seconds (effective 669.4 kbit/s)...
Hash of data verified.
Compressed 305600 bytes to 176950...
Writing at 0x00010000... (9 %)
Writing at 0x0001c0b2... (18 %)
Writing at 0x00024b1e... (27 %)
Writing at 0x00029bf8... (36 %)
Writing at 0x0002f0c7... (45 %)
Writing at 0x00034482... (54 %)
Writing at 0x0003a5ae... (63 %)
Writing at 0x00040b94... (72 %)
Writing at 0x0004b010... (81 %)
Writing at 0x00050b76... (90 %)
Writing at 0x0005608f... (100 %)
Wrote 305600 bytes (176950 compressed) at 0x00010000 in 4.6 seconds (effective 528.8 kbit/s)...
Hash of data verified.

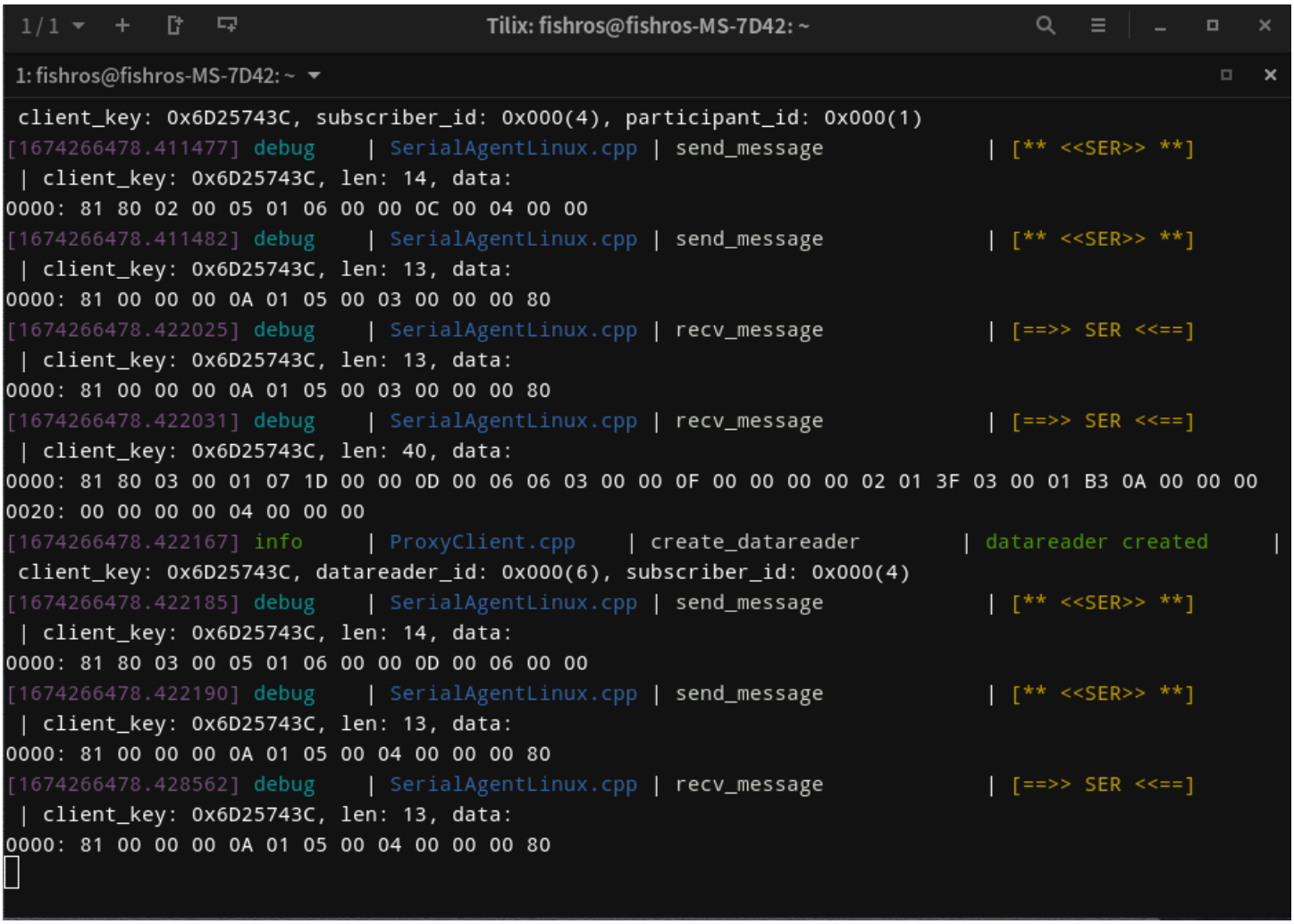
Leaving...
Hard resetting via RTS pin...
========================================================================== [SUCCESS] Took 11.05 seconds ===
*  终端将被任务重用，按任意键关闭。
```

4.2 启动Agent

接着打开终端启动agent

```shell
sudo docker run -it --rm -v /dev:/dev -v /dev/shm:/dev/shm --privileged --net=host microros/micro-ros-agent:$ROS_DISTRO serial --dev /dev/ttyUSB0 -v
```

点击下RST按钮，重启开发板，正常可以看到下图内容



4.3 查看是否连通

接着打开终端查看节点和话题

```shell
ros2 node list
ros2 topic list
```

```
fishros@fishros-MS-7D42:~$ ros2 node list
/topic_sub_test
fishros@fishros-MS-7D42:~$ ros2 topic list
/led_control
/parameter_events
/rosout
fishros@fishros-MS-7D42:~$ ros2 topic info /led_control
Type: std_msgs/msg/Int32
Publisher count: 0
Subscription count: 1
```

4.4 测试控制

关闭LED

```shell
ros2 topic pub /led_control  std_msgs/msg/Int32 "{data: 0}" --once
```

打开LED

```shell
ros2 topic pub /led_control  std_msgs/msg/Int32 "{data: 1}" --once
```

五、总结

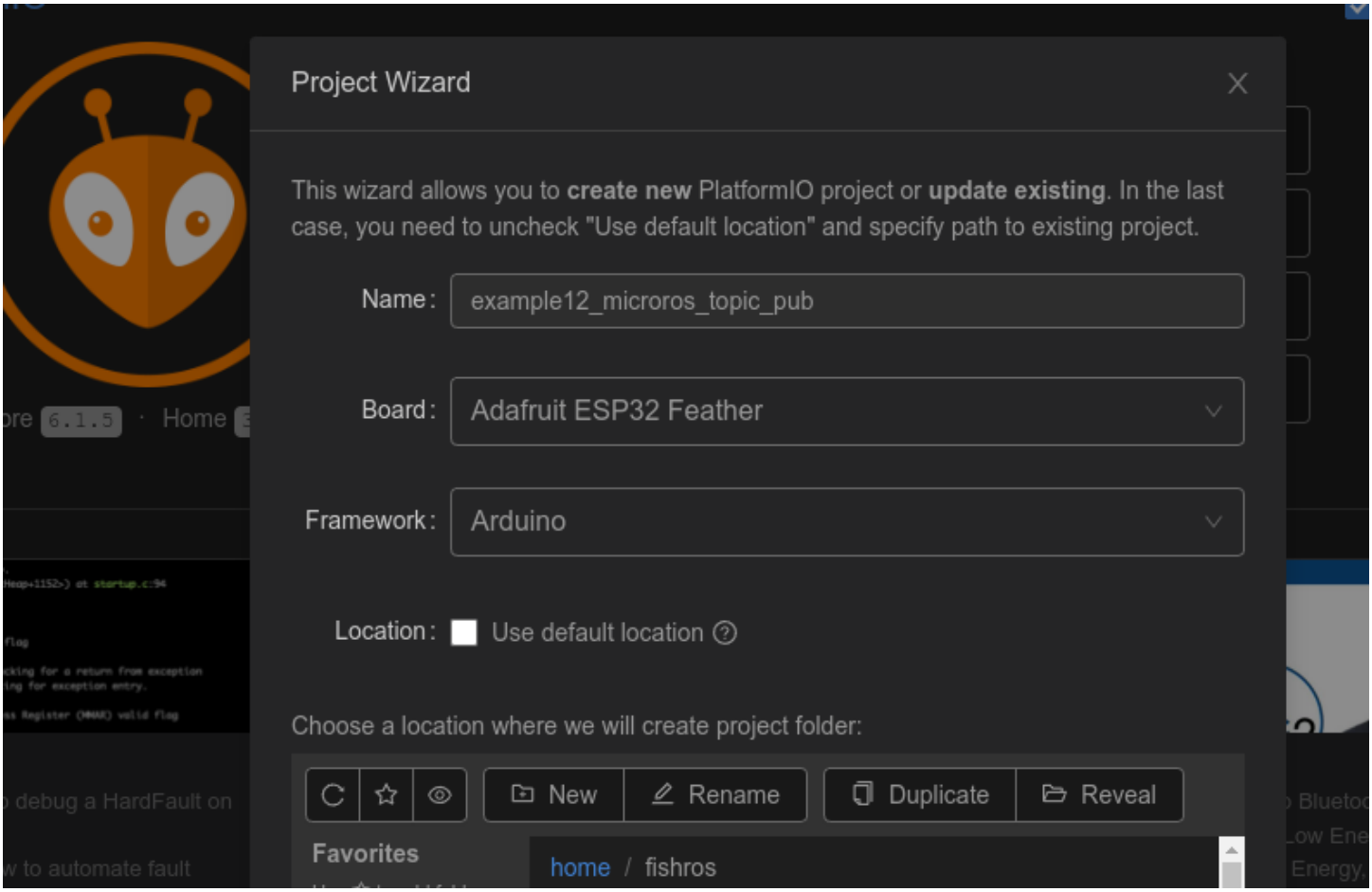
本节我们通过话题订阅，实现对开发板上LED的控制，下一节我们将尝试读取开发板上的VM引脚电压，并将其通过话题发布到上位机中。

2.MicroROS-话题发布实现

你好，我是爱吃鱼香ROS的小鱼。本节将学习在开发板上实现话题的发布，最终实现通过话题发布当前开发板的电池电量信息，关于电量信息的测量，请参考：**4.电池电压测量-学会使用ADC**

一、新建工程添加依赖

新建`example12_microros_topic_pub`工程



修改`platformio.ini`添加依赖

```ini
; PlatformIO Project Configuration File
;
;   Build options: build flags, source filter
;   Upload options: custom upload port, speed and extra flags
;   Library options: dependencies, extra library storages
;   Advanced options: extra scripting
;
; Please visit documentation for the other options and examples
```

```
; https://docs.platformio.org/page/projectconf.html

[env:featheresp32]
platform = espressif32
board = featheresp32
framework = arduino
lib_deps =
    https://gitee.com/ohhuo/micro_ros_platformio.git
```

二、编写代码-实现订阅

编辑main.cpp，代码如下，注释小鱼已经添加到代码中来了

```c++
#include <Arduino.h>
#include <micro_ros_platformio.h>

#include <rcl/rcl.h>
#include <rclc/rclc.h>
#include <rclc/executor.h>
// 添加头文件
#include <std_msgs/msg/float32.h>

rclc_executor_t executor;
rclc_support_t support;
rcl_allocator_t allocator;
rcl_node_t node;
rcl_timer_t timer;

// 声明话题发布者
rcl_publisher_t publisher;
// 声明消息文件
std_msgs__msg__Float32 pub_msg;

// 定义定时器接收回调函数
void timer_callback(rcl_timer_t *timer, int64_t last_call_time)
{
  RCLC_UNUSED(last_call_time);
  if (timer != NULL)
  {
```

```
    rcl_publish(&publisher, &pub_msg, NULL);
  }
}

void setup()
{
  Serial.begin(115200);
  // 设置通过串口进行MicroROS通信
  set_microros_serial_transports(Serial);
  // 延时时一段时间，等待设置完成
  delay(2000);
  // 初始化内存分配器
  allocator = rcl_get_default_allocator();
  // 创建初始化选项
  rclc_support_init(&support, 0, NULL, &allocator);
  // 创建节点 topic_sub_test
  rclc_node_init_default(&node, "topic_pub_test", "", &support);
  // 订阅者初始化
  rclc_publisher_init_default(
      &publisher,
      &node,
      ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, Float32),
      "battery_voltage");

  // 创建定时器，200ms发一次
  const unsigned int timer_timeout = 200;
  rclc_timer_init_default(
      &timer,
      &support,
      RCL_MS_TO_NS(timer_timeout),
      timer_callback);

  // 创建执行器
  rclc_executor_init(&executor, &support.context, 1, &allocator);
  // 给执行器添加定时器
  rclc_executor_add_timer(&executor, &timer);
  // 初始化ADC
  pinMode(34, INPUT);
  analogSetAttenuation(ADC_11db);
}

void loop()
```

```
{
  delay(100);
  // 循环处理数据
  rclc_executor_spin_some(&executor, RCL_MS_TO_NS(100));
  // 通过ADC获取电压值
  int analogValue = analogRead(34);                    // 读取原始值0-4096
  int analogVolts = analogReadMilliVolts(34);          // 读取模拟电压，单位毫伏
  float realVolts = 5.02 * ((float)analogVolts * 1e-3); // 计算实际电压值
  pub_msg.data = realVolts;
}
```

三、代码注解

相比之前的节点代码这里主要多了这几行

- `#include <std_msgs/msg/float32.h>` 包含flaot32类型头文件
- `rcl_publisher_t publisher;` 定义发布者
- `std_msgs__msg__Float32 pub_msg;` 定义发布消息，也需要提前定义
- `void timer_callback(rcl_timer_t *timer, int64_t last_call_time)` **定义定时器回调函数，当我们需要以某个频率做什么的时候定时器可以派上用场**
- `rclc_publisher_init_default` **初始化发布者**
- `rclc_timer_init_default` **初始化定时器**
- `rclc_executor_add_timer` **给执行器添加一个定时器回调**

四、下载测试

4.1 编译下载

连接开发板，编译下载。

```
Flash will be erased from 0x0000e000 to 0x0000ffff...
Flash will be erased from 0x00010000 to 0x0005afff...
Compressed 17472 bytes to 12125...
Writing at 0x00001000... (100 %)
Wrote 17472 bytes (12125 compressed) at 0x00001000 in 0.6 seconds (effective 244.7 kbit/s)...
Hash of data verified.
Compressed 3072 bytes to 128...
Writing at 0x00008000... (100 %)
Wrote 3072 bytes (128 compressed) at 0x00008000 in 0.0 seconds (effective 516.0 kbit/s)...
Hash of data verified.
Compressed 8192 bytes to 47...
Writing at 0x0000e000... (100 %)
Wrote 8192 bytes (47 compressed) at 0x0000e000 in 0.1 seconds (effective 669.4 kbit/s)...
Hash of data verified.
Compressed 305600 bytes to 176950...
Writing at 0x00010000... (9 %)
Writing at 0x0001c0b2... (18 %)
Writing at 0x00024b1e... (27 %)
Writing at 0x00029bf8... (36 %)
Writing at 0x0002f0c7... (45 %)
Writing at 0x00034482... (54 %)
Writing at 0x0003a5ae... (63 %)
Writing at 0x00040b94... (72 %)
Writing at 0x0004b010... (81 %)
Writing at 0x00050b76... (90 %)
Writing at 0x0005608f... (100 %)
Wrote 305600 bytes (176950 compressed) at 0x00010000 in 4.6 seconds (effective 528.8 kbit/s)...
Hash of data verified.

Leaving...
Hard resetting via RTS pin...
================================================================== [SUCCESS] Took 11.05 seconds ===
* 终端将被任务重用，按任意键关闭。
```

4.2 启动Agent服务

接着打开终端启动agent

```shell
sudo docker run -it --rm -v /dev:/dev -v /dev/shm:/dev/shm --privileged --net=host
microros/micro-ros-agent:$ROS_DISTRO serial --dev /dev/ttyUSB0 -v
```

点击下RST按钮，重启开发板，正常可以看到下图内容

```
fishros@fishros-MS-7D42:~/example12_microros_topic_pub$ sudo docker run -it --rm -v /dev:/dev -v /dev/shm:/dev/shm
--privileged --net=host microros/micro-ros-agent:$ROS_DISTRO serial --dev /dev/ttyUSB0 -v
[sudo] fishros 的密码：
[1674273464.285999] info     | TermiosAgentLinux.cpp | init                     | running...             | fd: 3
[1674273464.286097] info     | Root.cpp              | set_verbose_level        | logger setup           | verbose_lev
el: 4
[1674273464.351244] info     | Root.cpp              | create_client            | create                 | client_key:
 0x1CED357E, session_id: 0x81
[1674273464.351348] info     | SessionManager.hpp    | establish_session        | session established    | client_key:
 0x1CED357E, address: 0
[1674273464.377537] info     | ProxyClient.cpp       | create_participant       | participant created    | client_key:
 0x1CED357E, participant_id: 0x000(1)
[1674273464.392583] info     | ProxyClient.cpp       | create_topic             | topic created          | client_key:
 0x1CED357E, topic_id: 0x000(2), participant_id: 0x000(1)
[1674273464.402204] info     | ProxyClient.cpp       | create_publisher         | publisher created      | client_key:
 0x1CED357E, publisher_id: 0x000(3), participant_id: 0x000(1)
[1674273464.413377] info     | ProxyClient.cpp       | create_datawriter        | datawriter created     | client_key:
 0x1CED357E, datawriter_id: 0x000(5), publisher_id: 0x000(3)
```

4.3 测试是否连通

```shell
ros2 node list
ros2 topic list
```

```
fishros@fishros-MS-7D42:~/example12_microros_topic_pub$ ros2 node list
/topic_pub_test
fishros@fishros-MS-7D42:~/example12_microros_topic_pub$ ros2 topic list
/battery_voltage
/parameter_events
/rosout
```

4.4 查看话题数据

```shell
ros2 topic echo /battery_voltage
```

```
fishros@fishros-MS-7D42:~/example12_microros_topic_pub$ ros2 topic echo /battery_voltage
data: 12.956620216369629
---
data: 13.031920433044434
---
data: 13.006819725036621
---
data: 13.011839866638184
---
data: 13.031920433044434
---
data: 13.011839866638184
---
data: 13.021880149841309
---
data: 12.966659545898438
---
data: 13.011839866638184
---
data: 13.052000045776367
---
data: 13.011839866638184
---
data: 13.011839866638184
---
data: 13.026900291442871
---
data: 13.026900291442871
```

这里小鱼连接了小车的电池，VM电压代表电池电压，符合正常电压值范围。

同时可以使用下面指令测量话题频率

```
fishros@fishros-MS-7D42:~/example12_microros_topic_pub$ ros2 topic hz
/battery_voltage
average rate: 4.828
        min: 0.207s max: 0.208s std dev: 0.00021s window: 6
average rate: 5.034
        min: 0.106s max: 0.208s std dev: 0.02793s window: 12
average rate: 4.973
        min: 0.106s max: 0.208s std dev: 0.02378s window: 17
average rate: 4.941
        min: 0.106s max: 0.208s std dev: 0.02104s window: 22
average rate: 5.005
        min: 0.106s max: 0.208s std dev: 0.02594s window: 28
average rate: 4.977
        min: 0.106s max: 0.208s std dev: 0.02404s window: 33
average rate: 4.958
        min: 0.106s max: 0.208s std dev: 0.02249s window: 38
```

```
average rate: 4.997
        min: 0.106s max: 0.208s std dev: 0.02541s window: 44
```

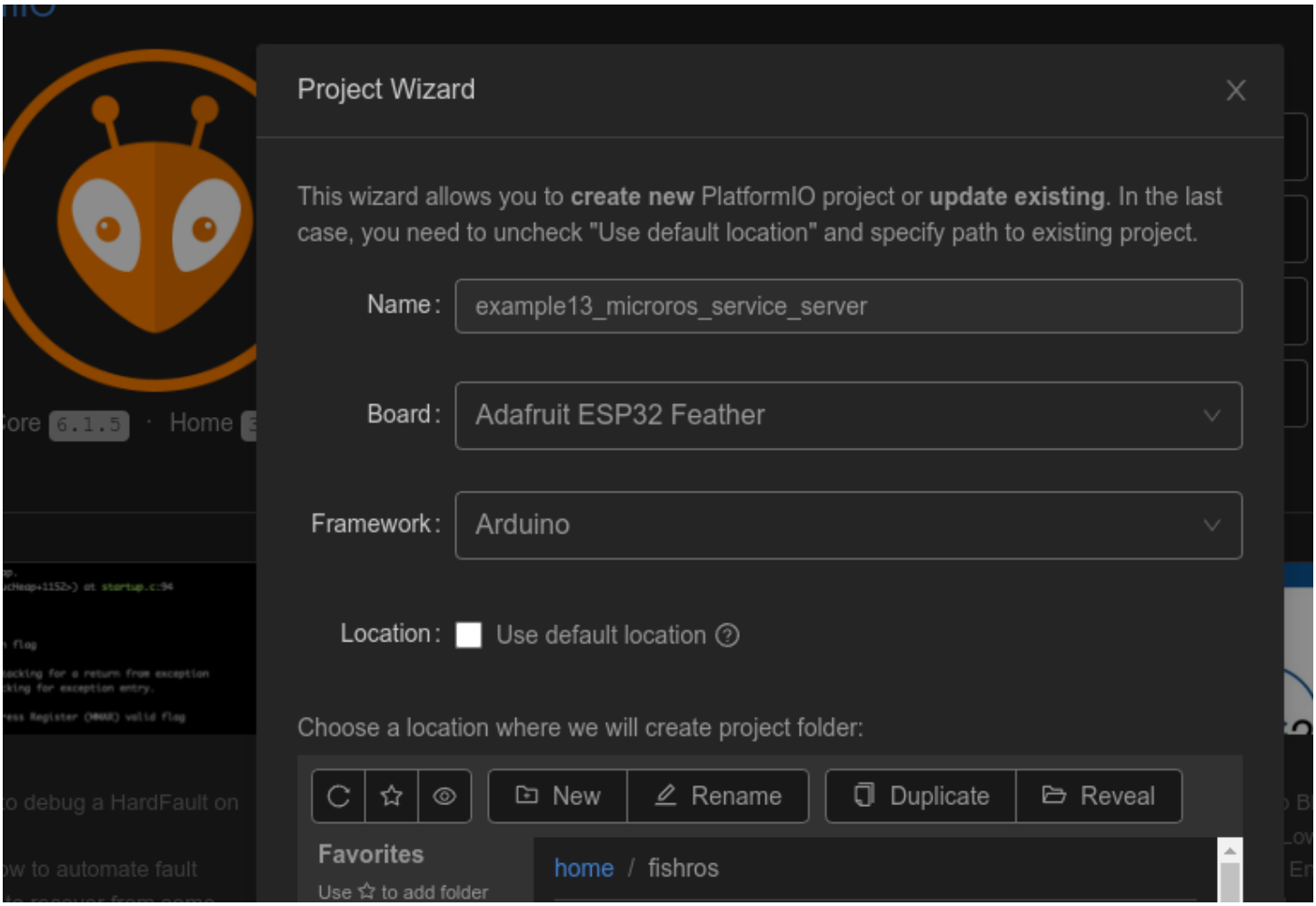
五、总结

本节我们通过电量信息发布例程，学习了如何在开发板上实现话题发布流程。下一节我们开始尝试在开发板上建立服务端，尝试服务通信。

3.MicroROS-服务通信服务端实现

一、新建工程添加依赖

新建`example13_microros_service_server`工程



修改`platformio.ini`添加依赖

```ini
; PlatformIO Project Configuration File
;
;   Build options: build flags, source filter
;   Upload options: custom upload port, speed and extra flags
;   Library options: dependencies, extra library storages
;   Advanced options: extra scripting
;
; Please visit documentation for the other options and examples
; https://docs.platformio.org/page/projectconf.html
```

```
[env:featheresp32]
platform = espressif32
board = featheresp32
framework = arduino
lib_deps =
    https://gitee.com/ohhuo/micro_ros_platformio.git
```

二、编写代码-实现服务代码

编辑main.cpp，代码如下，注释小鱼已经添加到代码中来了

```c++
#include <Arduino.h>
#include <micro_ros_platformio.h>

#include <rcl/rcl.h>
#include <rclc/rclc.h>
#include <rclc/executor.h>

// 添加接口
#include <example_interfaces/srv/add_two_ints.h>

rclc_executor_t executor;
rclc_support_t support;
rcl_allocator_t allocator;
rcl_node_t node;

// 定义服务
rcl_service_t service;

// 服务请求和返回消息定义
example_interfaces__srv__AddTwoInts_Request req;
example_interfaces__srv__AddTwoInts_Response res;

// 服务回调函数
void service_callback(const void *req, void *res)
{
  example_interfaces__srv__AddTwoInts_Request *req_in =
(example_interfaces__srv__AddTwoInts_Request *)req;
  example_interfaces__srv__AddTwoInts_Response *res_in =
```

```
  (example_interfaces__srv__AddTwoInts_Response *)res;
    // 计算sum
    res_in->sum = req_in->a + req_in->b;
  }

  void setup()
  {
    Serial.begin(115200);
    // 设置通过串口进行MicroROS通信
    set_microros_serial_transports(Serial);
    // 延时时一段时间，等待设置完成
    delay(2000);
    // 初始化内存分配器
    allocator = rcl_get_default_allocator();
    // 创建初始化选项
    rclc_support_init(&support, 0, NULL, &allocator);
    // 创建节点 hello_microros
    rclc_node_init_default(&node, "service_test", "", &support);
    // 使用默认配置创建服务
    rclc_service_init_default(&service, &node,
  ROSIDL_GET_SRV_TYPE_SUPPORT(example_interfaces, srv, AddTwoInts), "/addtwoints");
    // 创建执行器
    rclc_executor_init(&executor, &support.context, 1, &allocator);
    // 执行器添加服务
    rclc_executor_add_service(&executor, &service, &req, &res, service_callback);
  }

  void loop()
  {
    delay(100);
    // 循环处理数据
    rclc_executor_spin_some(&executor, RCL_MS_TO_NS(100));
  }
```

三、代码注解

相比之前的节点代码这里主要多了这几行

- `#include <example_interfaces/srv/add_two_ints.h>` 添加接口头文件
- `rcl_service_t service;` 定义服务

- `example_interfaces__srv__AddTwoInts_Request res;` 定义请求数据存储位置
- `example_interfaces__srv__AddTwoInts_Response req;`定义响应数据存储位置
- `void service_callback(const void *req, void *res)` 服务回调函数
- `rclc_service_init_default(&service, &node, ROSIDL_GET_SRV_TYPE_SUPPORT(example_interfaces, srv, AddTwoInts), "/addtwoints");` 使用默认配置初始化服务
- `rclc_executor_add_service(&executor, &service, &req, &res, service_callback);` 为执行器添加服务定义

四、下载测试

4.1 编译下载

连接开发板，编译下载。

```
Flash will be erased from 0x0000e000 to 0x0000ffff...
Flash will be erased from 0x00010000 to 0x0005afff...
Compressed 17472 bytes to 12125...
Writing at 0x00001000... (100 %)
Wrote 17472 bytes (12125 compressed) at 0x00001000 in 0.6 seconds (effective 244.7 kbit/s)...
Hash of data verified.
Compressed 3072 bytes to 128...
Writing at 0x00008000... (100 %)
Wrote 3072 bytes (128 compressed) at 0x00008000 in 0.0 seconds (effective 516.0 kbit/s)...
Hash of data verified.
Compressed 8192 bytes to 47...
Writing at 0x0000e000... (100 %)
Wrote 8192 bytes (47 compressed) at 0x0000e000 in 0.1 seconds (effective 669.4 kbit/s)...
Hash of data verified.
Compressed 305600 bytes to 176950...
Writing at 0x00010000... (9 %)
Writing at 0x0001c0b2... (18 %)
Writing at 0x00024b1e... (27 %)
Writing at 0x00029bf8... (36 %)
Writing at 0x0002f0c7... (45 %)
Writing at 0x00034482... (54 %)
Writing at 0x0003a5ae... (63 %)
Writing at 0x00040b94... (72 %)
Writing at 0x0004b010... (81 %)
Writing at 0x00050b76... (90 %)
Writing at 0x0005608f... (100 %)
Wrote 305600 bytes (176950 compressed) at 0x00010000 in 4.6 seconds (effective 528.8 kbit/s)...
Hash of data verified.

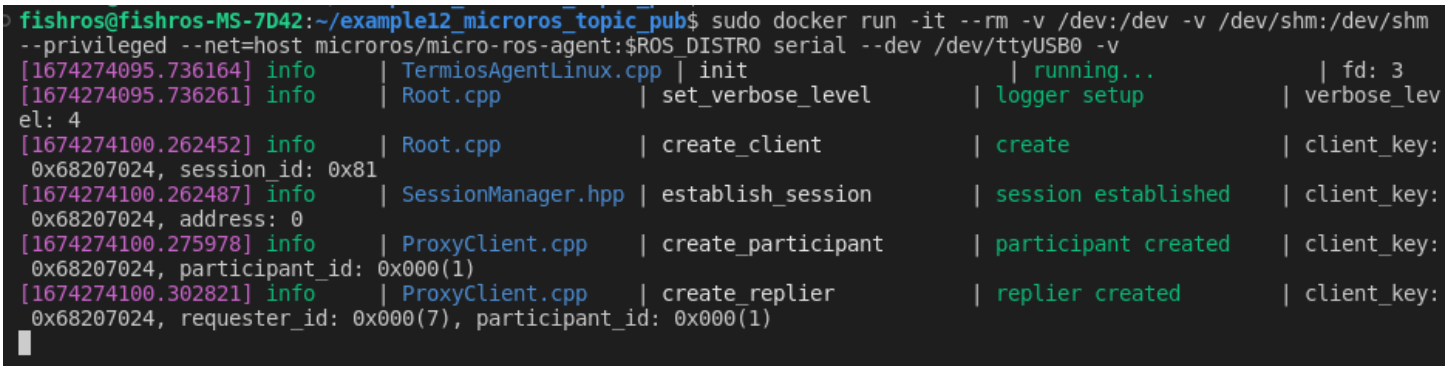
Leaving...
Hard resetting via RTS pin...
======================================================================== [SUCCESS] Took 11.05 seconds ===
* 终端将被任务重用，按任意键关闭。
```

4.2 启动Agent服务

接着打开终端启动agent

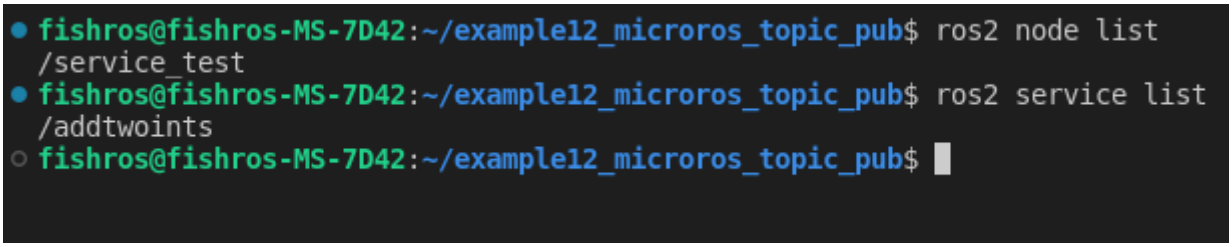
```shell
sudo docker run -it --rm -v /dev:/dev -v /dev/shm:/dev/shm --privileged --net=host microros/micro-ros-agent:$ROS_DISTRO serial --dev /dev/ttyUSB0 -v
```

点击下RST按钮，重启开发板，正常可以看到下图内容



4.3 测试是否连通

```shell
ros2 node list
ros2 service list
```



4.4 测试服务

测试个1+2吧，使用ROS2CLI

```shell
ros2 service call /addtwoints example_interfaces/srv/AddTwoInts "{a: 1, b: 2}"
```

成功的返回了3

```
fishros@fishros-MS-7D42:~/example12_microros_topic_pub$ ros2 service call /addtwoints example_interfaces/srv/AddTwo
Ints "{a: 1, b: 2}"
requester: making request: example_interfaces.srv.AddTwoInts_Request(a=1, b=2)

response:
example_interfaces.srv.AddTwoInts_Response(sum=3)
```

五、总结

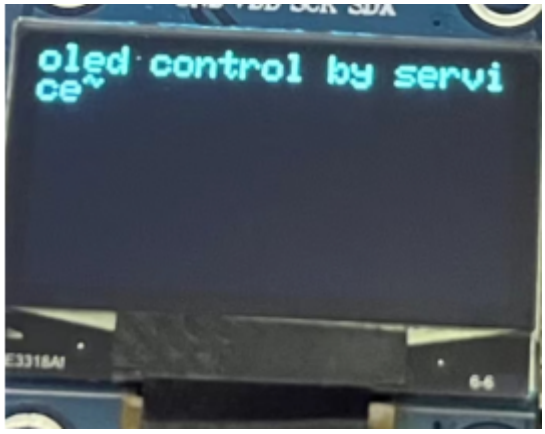
本节我们通过两数相加的Demo实现了服务通信，但是我们使用的是样例服务接口，在嵌入式平台上如何才能自定义接口呢？我们进阶篇见！

1.控制OLED-自定义消息接口

你好，我是爱吃鱼香ROS的小鱼。前面章节中我们使用ROS2提供的两数相加服务接口实现了服务通信，但是很多场景需要我们自定定义接口，所以本节我们就学习如何自定义MicroROS的接口，以及如何将自己的功能包添加到MicroROS中来。

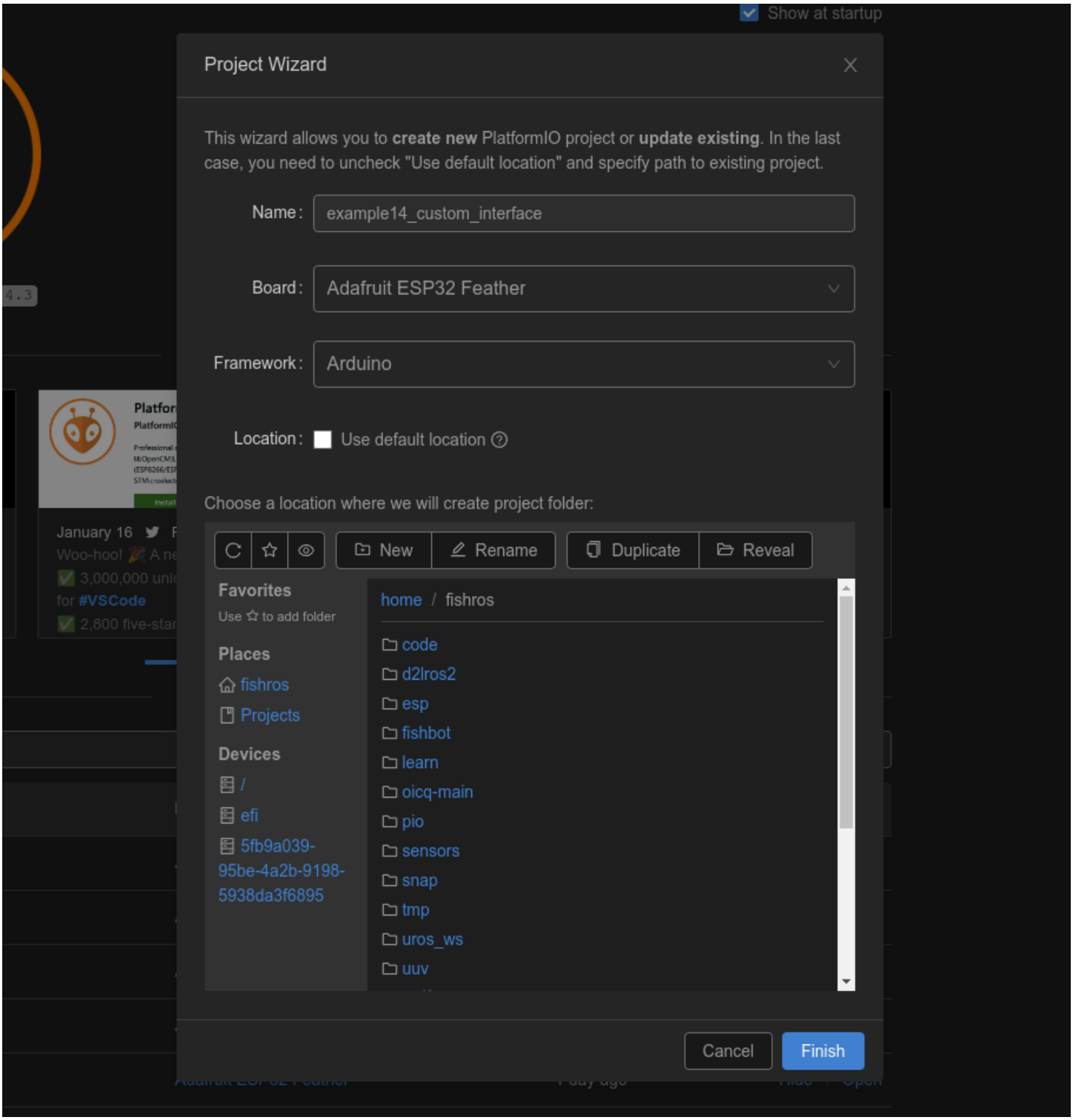
本节课最终效果是：通过自定义的服务接口控制开发板上的OLED显示器的内容。

```
ros2 service call /oled_control fishbot_interfaces/srv/OledControl "{px: 0, py: 0
,data: 'oled control by service~'}"
```



一、新建工程添加依赖

新建`example14_custom_interface`



修改 `platformio.ini`

```ini
[env:featheresp32]
platform = espressif32
board = featheresp32
framework = arduino
lib_deps =
```

```
    https://gitee.com/ohhuo/micro_ros_platformio.git
    adafruit/Adafruit SSD1306@^2.5.7
```

这里除了添加micro_ros库之外再添加oled驱动库。

二、添加自定义接口

添加自定义接口一共需要三步。

1.创建 `extra_packages` 文件夹并创建接口功能包

2.编译功能包（主要为了测试功能包是否正常）

3.删除 `.pio/libdeps/featheresp32/micro_ros_platformio/libmicroros` 文件夹，重新编译

2.1 创建功能包

在工程的主目录下创建 `extra_packages` 文件夹，接着在文件夹下创建 `fishbot_interfaces` 功能包

```shell
cd example14_custom_interface
mkdir extra_packages
cd extra_packages
ros2 pkg create fishbot_interfaces
```

接着添加服务接口文件并修改CMakeLists.txt

文件 `extra_packages/fishbot_interfaces/srv/OledControl.srv`

```
int32 px
int32 py
string data
---
int32 result
```

文件 `extra_packages/fishbot_interfaces/CMakeLists.txt`

```cmake
cmake_minimum_required(VERSION 3.5)
project(fishbot_interfaces)



if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()

# find dependencies
find_package(ament_cmake REQUIRED)
# uncomment the following section in order to fill in
# further dependencies manually.
# find_package(<dependency> REQUIRED)

if(BUILD_TESTING)
  find_package(ament_lint_auto REQUIRED)
  # the following line skips the linter which checks for copyrights
  # uncomment the line when a copyright and license is not present in all source
files
  #set(ament_cmake_copyright_FOUND TRUE)
  # the following line skips cpplint (only works in a git repo)
  # uncomment the line when this package is not in a git repo
  #set(ament_cmake_cpplint_FOUND TRUE)
  ament_lint_auto_find_test_dependencies()
endif()

find_package(rosidl_default_generators REQUIRED)

rosidl_generate_interfaces(${PROJECT_NAME}
  "srv/OledControl.srv"
 )

ament_package()
```

2.2 编译功能包

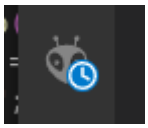
```shell
cd extra_packages/
colcon build
```

```
fishros@fishros-MS-7D42:~/example14_custom_interface$ cd extra_packages/
fishros@fishros-MS-7D42:~/example14_custom_interface/extra_packages$ colcon build
Starting >>> fishbot_interfaces
Finished <<< fishbot_interfaces [0.32s]

Summary: 1 package finished [0.41s]
fishros@fishros-MS-7D42:~/example14_custom_interface/extra_packages$
```

2.3 重新编译工程

编译前需要删除`.pio/libdeps/featheresp32/micro_ros_platformio/libmicroros`文件夹，使用Ctrl+Alt+B重新重新编译工程。



三、编写代码

和两数相加服务相似的代码，只不过更换了接口并添加了OLED的驱动。

```c++
#include <Arduino.h>
#include <micro_ros_platformio.h>

#include <rcl/rcl.h>
#include <rclc/rclc.h>
#include <rclc/executor.h>
#include <micro_ros_utilities/string_utilities.h>

#include "Wire.h"
#include <Adafruit_GFX.h>     // 加载Adafruit_GFX库
#include <Adafruit_SSD1306.h> // 加载Adafruit_SSD1306库

#include <fishbot_interfaces/srv/oled_control.h> // 添加接口

rclc_executor_t executor;
```

```
rclc_support_t support;
rcl_allocator_t allocator;
rcl_node_t node;
// 定义服务
rcl_service_t service;

// 服务请求和返回消息定义
fishbot_interfaces__srv__OledControl_Request req;
fishbot_interfaces__srv__OledControl_Response res;

Adafruit_SSD1306 display;

// 服务回调函数
void service_callback(const void *req, void *res)
{
  fishbot_interfaces__srv__OledControl_Request *req_in =
(fishbot_interfaces__srv__OledControl_Request *)req;
  fishbot_interfaces__srv__OledControl_Response *res_in =
(fishbot_interfaces__srv__OledControl_Response *)res;
  // 计算sum
  display.clearDisplay();                    // 清空屏幕
  display.setCursor(req_in->px, req_in->py); // 设置开始显示文字的坐标
  display.println(req_in->data.data);        // 输出的字符
  display.display();
  res_in->result = 0;
}

void setup()
{
  Serial.begin(115200);
  // 设置通过串口进行MicroROS通信
  set_microros_serial_transports(Serial);
  // 延时时一段时间，等待设置完成
  delay(2000);
  // 初始化内存分配器
  allocator = rcl_get_default_allocator();
  // 创建初始化选项
  rclc_support_init(&support, 0, NULL, &allocator);
  // 创建节点 hello_microros
  rclc_node_init_default(&node, "example14_interface", "", &support);
  // 使用默认配置创建服务
  rclc_service_init_default(&service, &node,
```

```
  ROSIDL_GET_SRV_TYPE_SUPPORT(fishbot_interfaces, srv, OledControl),
  "/oled_control");
    // 创建执行器
    rclc_executor_init(&executor, &support.context, 1, &allocator);
    // 执行器添加服务
    rclc_executor_add_service(&executor, &service, &req, &res, service_callback);
    // 重要，为string类型消息分配空间
    req.data = micro_ros_string_utilities_init_with_size(100);

    /*=======================OLED初始化====================================*/
    Wire.begin(18, 19);
    display = Adafruit_SSD1306(128, 64, &Wire);
    display.begin(SSD1306_SWITCHCAPVCC, 0x3C); // 设置OLED的I2C地址
    display.clearDisplay();                    // 清空屏幕
    display.setTextSize(1);                    // 设置字体大小
    display.setCursor(0, 0);                   // 设置开始显示文字的坐标
    display.setTextColor(SSD1306_WHITE);       // 设置字体颜色
    display.println("hello fishros!");         // 输出的字符
    display.display();
  }

  void loop()
  {
    delay(100);
    // 循环处理数据
    rclc_executor_spin_some(&executor, RCL_MS_TO_NS(100));
  }
```

四、代码讲解

这里对新增的几行主要代码进行讲解

- `#include <micro_ros_utilities/string_utilities.h>` 添加string工具类
- `req.data = micro_ros_string_utilities_init_with_size(100);` 使用string类型内存分配工具为data分配100字节e的空间

主要就是这两部分，值得注意的是，如果不提前为string类型的数据分配内容空间，最终会导致无法正常接收数据。

五、下载测试

5.1 编译下载

连接开发板，编译下载。

```
Flash will be erased from 0x0000e000 to 0x0000ffff...
Flash will be erased from 0x00010000 to 0x0005afff...
Compressed 17472 bytes to 12125...
Writing at 0x00001000... (100 %)
Wrote 17472 bytes (12125 compressed) at 0x00001000 in 0.6 seconds (effective 244.7 kbit/s)...
Hash of data verified.
Compressed 3072 bytes to 128...
Writing at 0x00008000... (100 %)
Wrote 3072 bytes (128 compressed) at 0x00008000 in 0.0 seconds (effective 516.0 kbit/s)...
Hash of data verified.
Compressed 8192 bytes to 47...
Writing at 0x0000e000... (100 %)
Wrote 8192 bytes (47 compressed) at 0x0000e000 in 0.1 seconds (effective 669.4 kbit/s)...
Hash of data verified.
Compressed 305600 bytes to 176950...
Writing at 0x00010000... (9 %)
Writing at 0x0001c0b2... (18 %)
Writing at 0x00024b1e... (27 %)
Writing at 0x00029bf8... (36 %)
Writing at 0x0002f0c7... (45 %)
Writing at 0x00034482... (54 %)
Writing at 0x0003a5ae... (63 %)
Writing at 0x00040b94... (72 %)
Writing at 0x0004b010... (81 %)
Writing at 0x00050b76... (90 %)
Writing at 0x0005608f... (100 %)
Wrote 305600 bytes (176950 compressed) at 0x00010000 in 4.6 seconds (effective 528.8 kbit/s)...
Hash of data verified.

Leaving...
Hard resetting via RTS pin...
======================================================================== [SUCCESS] Took 11.05 seconds ===
* 终端将被任务重用，按任意键关闭。
```

5.2 启动Agent

接着打开终端启动agent

```shell
sudo docker run -it --rm -v /dev:/dev -v /dev/shm:/dev/shm --privileged --net=host microros/micro-ros-agent:$ROS_DISTRO serial --dev /dev/ttyUSB0 -v
```

点击下RST按钮，重启开发板，正常可以看到下图内容

```
fishros@fishros-MS-7D42:~/example14_custom_interface$ sudo docker run -it --rm -v /dev:/dev -v /dev/shm:/dev/shm --privileged --net=host microros/micro-ros-agent:$ROS_DISTRO serial --dev /dev/ttyUSB0 -v
[sudo] fishros 的密码：
[1674321648.226965] info     | TermiosAgentLinux.cpp | init                     | running...             | fd: 3
[1674321648.227050] info     | Root.cpp              | set_verbose_level        | logger setup           | verbose_level: 4
[1674321651.929351] info     | Root.cpp              | create_client            | create                 | client_key: 0x69946A97, session_id: 0x81
[1674321651.929383] info     | SessionManager.hpp    | establish_session        | session established    | client_key: 0x69946A97, address: 0
[1674321651.943186] info     | ProxyClient.cpp       | create_participant       | participant created    | client_key: 0x69946A97, participant_id: 0x000(1)
[1674321651.969257] info     | ProxyClient.cpp       | create_replier           | replier created        | client_key: 0x69946A97, requester_id: 0x000(7), participant_id: 0x000(1)
```

5.3 查看是否连通

接着打开终端查看节点和话题

```shell
ros2 node list
ros2 service list
```

```
fishros@fishros-MS-7D42:~/example14_custom_interface/extra_packages$ ros2 node list
/example14_interface
fishros@fishros-MS-7D42:~/example14_custom_interface/extra_packages$ ros2 service list
/oled_control
fishros@fishros-MS-7D42:~/example14_custom_interface/extra_packages$
```

4.4 测试控制

进入`extra_packages`，source环境

```
source install/setup.bash
```

显示`nihao`

```shell
ros2 service call /oled_control fishbot_interfaces/srv/OledControl "{px: 0, py: 0, data: 'nihao'}"
```



显示`oled control by service~`

```
ros2 service call /oled_control fishbot_interfaces/srv/OledControl "{px: 0, py: 0
,data: 'oled control by service~'}"
```



六、总结

本节通过使用自定义服务接口控制oled显示的例程，学习了如何在工程中添加自己的功能包和接口文件。下一节我们将学习如何让开发板的时间和上位机的时间进行同步。

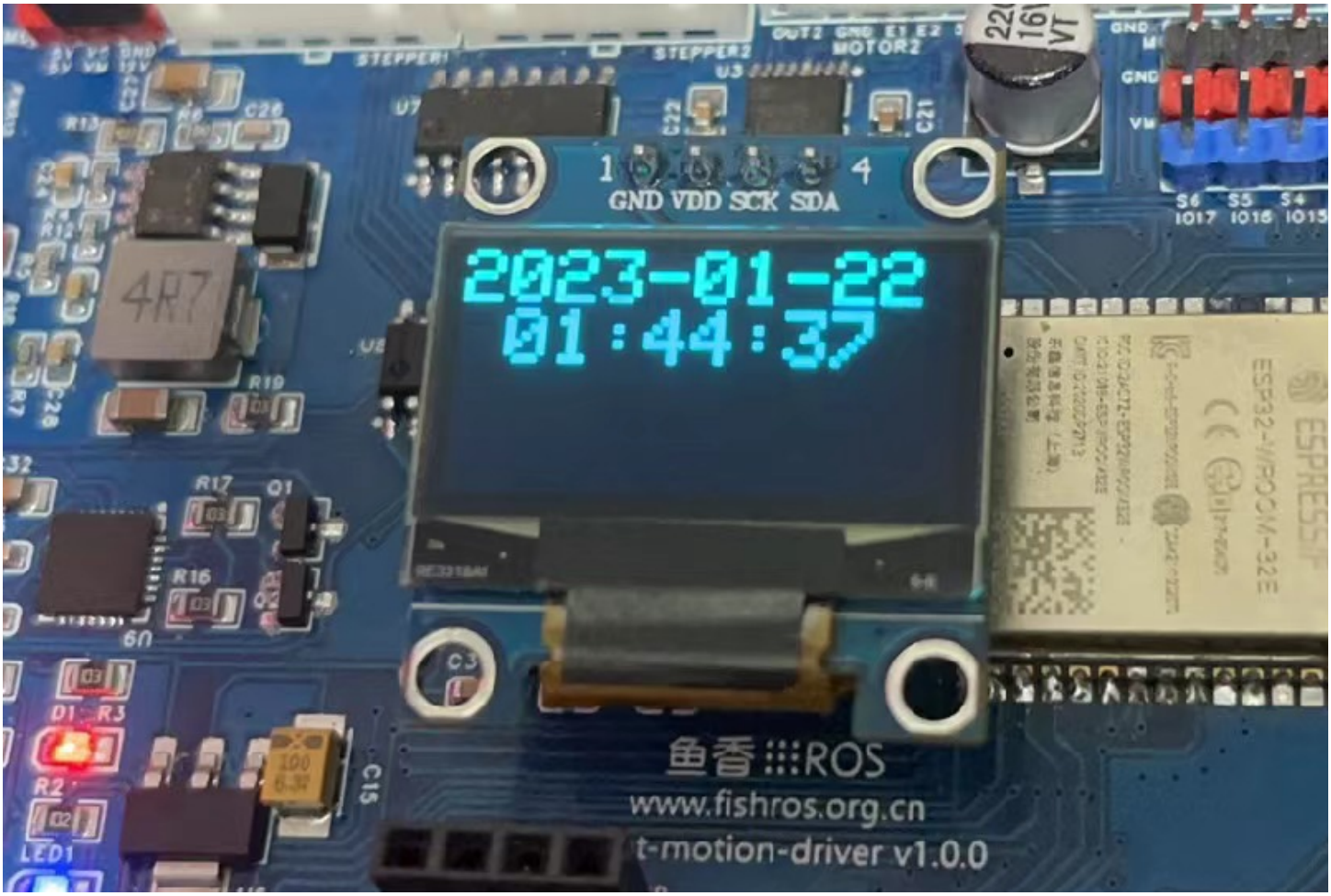
2.做个时钟-系统时间同步

你好，我是爱吃鱼香ROS的小鱼。在机器人系统中，时间同步非常重要。

原因在于，硬件系统采集的数据都是随时间变换而变化的，如果当前的控制使用上一时刻的传感器数据判断，就会造成各种问题，比如机器人上一时刻检测到前方有障碍，下一时刻障碍消失了，但此时如果采用过期的数据，就会造成误判。

MicroROS在设计时为我们提供了一系列的API用于时间同步和时间获取，本节我们就通过MicroROS进行时间同步，并最终在OLED上实现一个时钟功能。

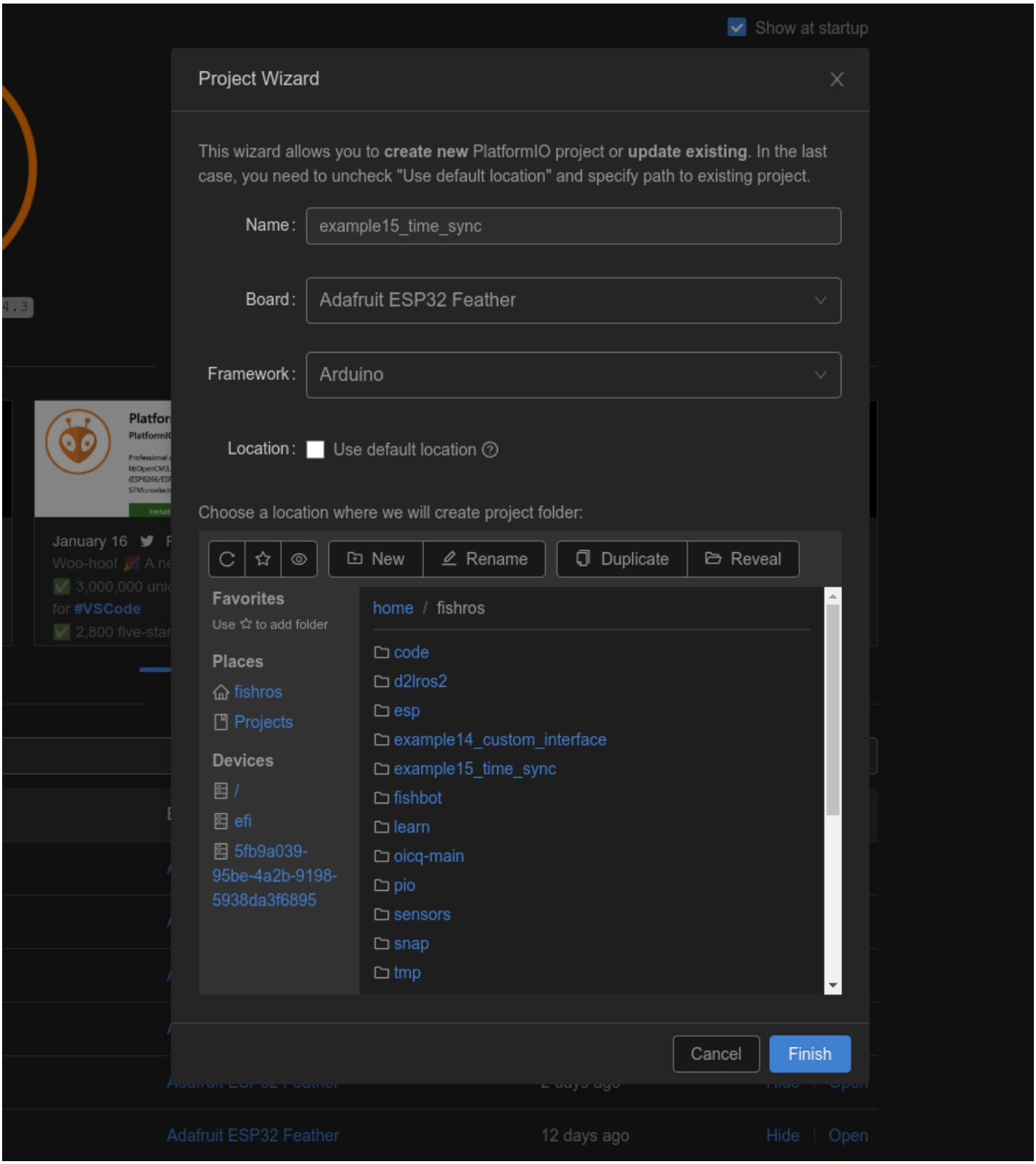
最终效果如下:



一、新建工程并添加依赖

1.1 新建工程

新建 `example15_time_sync` 工程



1.2 添加依赖

这里需要使用三个库，microros、oled驱动以及时间库Time

```ini
[env:featheresp32]
platform = espressif32
```

```
board = featheresp32
framework = arduino
lib_deps =
    https://gitee.com/ohhuo/micro_ros_platformio.git
    adafruit/Adafruit SSD1306@^2.5.7
    paulstoffregen/Time@^1.6.1
```

二、编写代码

代码并不多，单个文件，小鱼将相应注释已经加上。

```c++
#include <Arduino.h>
#include <micro_ros_platformio.h>

#include <rcl/rcl.h>
#include <rclc/rclc.h>
#include <rclc/executor.h>

#include <TimeLib.h>           // 加载时间库，提供setTime\year\month...函数
#include <Adafruit_GFX.h>      // 加载Adafruit_GFX库
#include <Adafruit_SSD1306.h> // 加载Adafruit_SSD1306库
Adafruit_SSD1306 display;      // 声明对象

rclc_executor_t executor;
rclc_support_t support;
rcl_allocator_t allocator;
rcl_node_t node;

const int timeout_ms = 1000;
static int64_t time_ms;
static time_t time_seconds;
char time_str[25];

void setup()
{
  Serial.begin(115200);
  // 设置通过串口进行MicroROS通信
  set_microros_serial_transports(Serial);
  // 延时时一段时间，等待设置完成
```

```
  delay(2000);
  // 初始化内存分配器
  allocator = rcl_get_default_allocator();
  // 创建初始化选项
  rclc_support_init(&support, 0, NULL, &allocator);
  // 创建节点 example15_time_sync
  rclc_node_init_default(&node, "example15_time_sync", "", &support);
  // 创建执行器
  rclc_executor_init(&executor, &support.context, 1, &allocator);

  Wire.begin(18, 19);
  display = Adafruit_SSD1306(128, 64, &Wire);
  display.begin(SSD1306_SWITCHCAPVCC, 0x3C); // 设置OLED的I2C地址，默认0x3C
  display.setTextSize(2);                    // 设置字体大小，最小为1
  display.clearDisplay();                    // 清空屏幕
  display.setCursor(0, 0);                   // 设置开始显示文字的坐标
  display.setTextColor(SSD1306_WHITE);       // 设置字体颜色
  display.println("hello oled!");            // 输出的字符
}

void loop()
{
  /*=========================同步时间=====================================*/
  while (!rmw_uros_epoch_synchronized()) // 判断时间是否同步
  {
    rmw_uros_sync_session(timeout_ms); //  同步时间
    if (rmw_uros_epoch_synchronized())
    {
      time_ms = rmw_uros_epoch_millis(); // 获取当前时间
      time_seconds = time_ms / 1000;
      setTime(time_seconds + 8 * 3600); // 将当前时间+8H到北京时间然后设置到系统
    }
    delay(10);
    return;
  }

  /*=======================获取时间与显示==================================*/
  sprintf(time_str, "%04d-%02d-%02d %02d:%02d:%02d ", year(), month(), day(),
hour(), minute(), second());

  display.clearDisplay();    // 清空屏幕
  display.setCursor(00, 0);  // 设置开始显示文字的坐标
```

```
    display.println(time_str); // 输出的字符
    display.display();
    delay(100);
  }
```

三、代码注解&API介绍

核心的时间同步代码就三行

- `rmw_uros_epoch_synchronized` 判断microros是否已经同步时间
- `rmw_uros_sync_session 同步时间`
- `rmw_uros_epoch_millis 获取当前时间`

四、下载测试

4.1 编译下载

```
Wrote 8192 bytes (47 compressed) at 0x0000e000 in 0.1 seconds (effective 689.0 kbit/s)...
Hash of data verified.
Compressed 315600 bytes to 180652...
Writing at 0x00010000... (8 %)
Writing at 0x0001acfc... (16 %)
Writing at 0x00023efa... (25 %)
Writing at 0x00029604... (33 %)
Writing at 0x0002ec38... (41 %)
Writing at 0x00033f78... (50 %)
Writing at 0x00039488... (58 %)
Writing at 0x000405aa... (66 %)
Writing at 0x0004aa0e... (75 %)
Writing at 0x00051ec9... (83 %)
Writing at 0x0005740b... (91 %)
Writing at 0x0005cea4... (100 %)
Wrote 315600 bytes (180652 compressed) at 0x00010000 in 4.9 seconds (effective 519.3 kbit/s)
Hash of data verified.

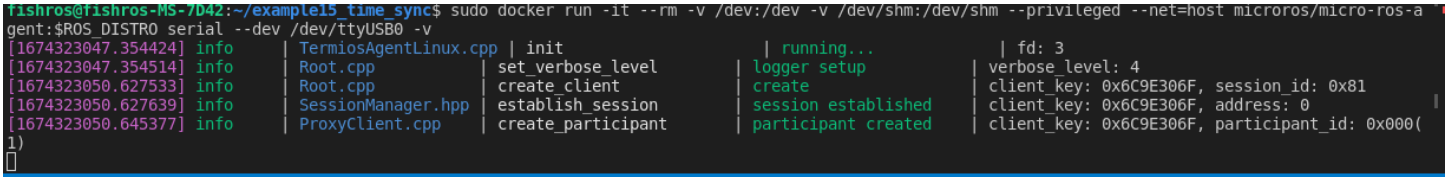
Leaving...
Hard resetting via RTS pin...
======================================================================== [SUCCESS] Took 9.55 seconds ===
*  终端将被任务重用，按任意键关闭。
```

4.2 启动Agent

接着打开终端启动agent

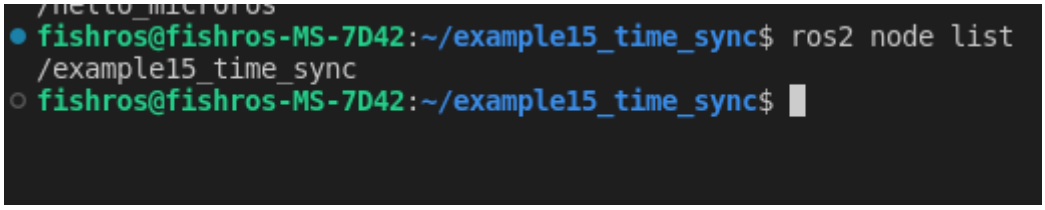
```shell
sudo docker run -it --rm -v /dev:/dev -v /dev/shm:/dev/shm --privileged --net=host microros/micro-ros-agent:$ROS_DISTRO serial --dev /dev/ttyUSB0 -v
```

点击下RST按钮，重启开发板，正常可以看到下图内容



4.3 查看是否连通

```
ros2 node list
```



4.4 查看时间



五、总结

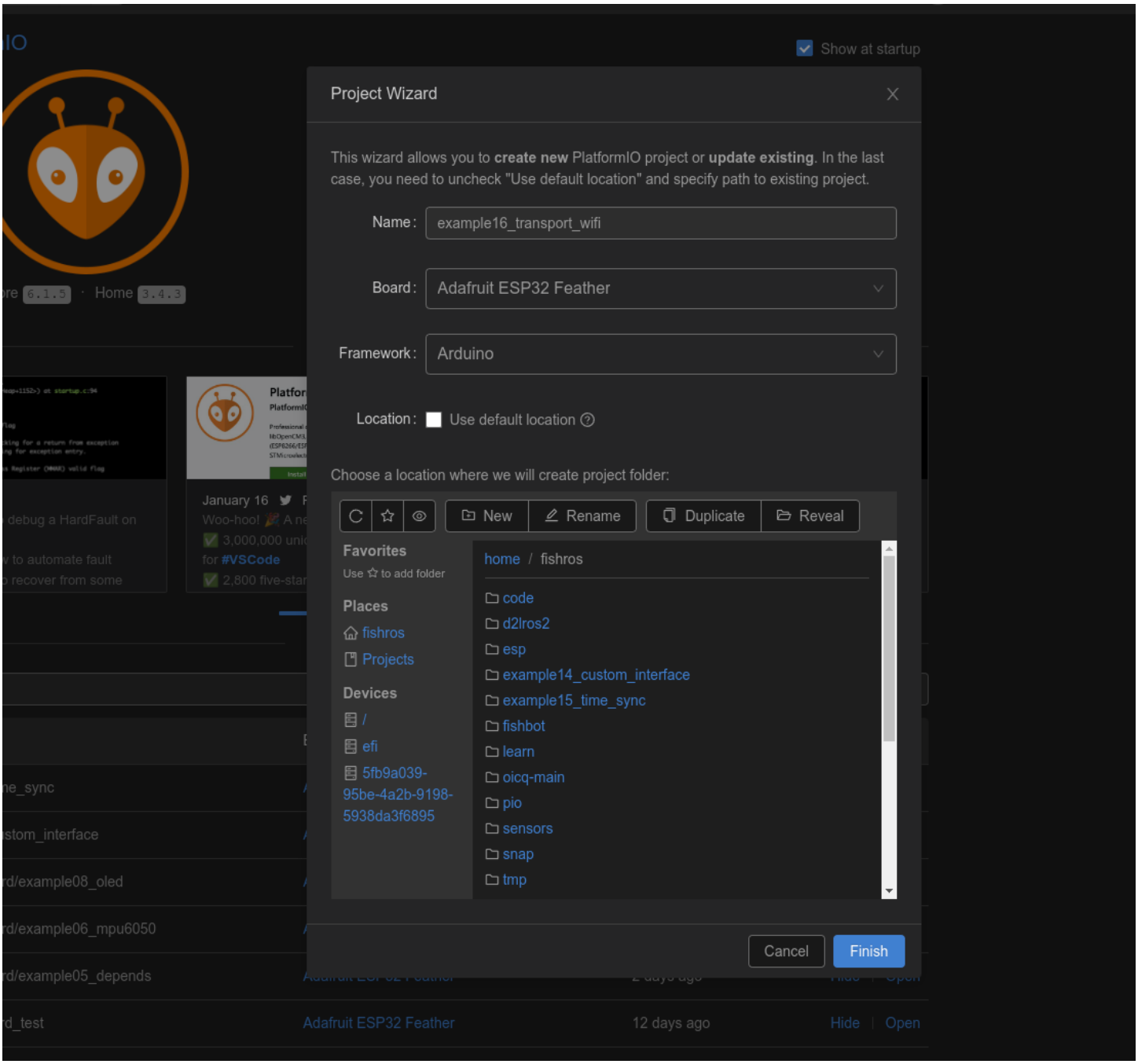
本节我们通过三个API完成了MicroROS时间同步功能的开发，最终并将当前时间在OLED上显示出来，但使用有线的方式过于麻烦，下一节我们尝试通过无线WIFI完成时间的同步与MicroROS的通信开发。

3.无线通讯-了解传输原理

你好，我是爱吃鱼香ROS的小鱼。在前面的学习中，我们一直通过串口通信来连接MicroROS，但一直扯着跟线是不是觉得很麻烦，本节我们利用开发板上的WIFI功能尝试使用无线的方式连接Agent。

一、新建工程并添加依赖

1.1 新建工程



1.2 添加依赖&修改配置

修改`platformio.ini`

```ini
[env:featheresp32]
platform = espressif32
board = featheresp32
framework = arduino
board_microros_transport = wifi
lib_deps =
    https://gitee.com/ohhuo/micro_ros_platformio.git
```

注意这里的配置，我们多添加了一个`board_microros_transport`，这个配置值我们给的是wifi，表示无线传输。

除了WIFI还支持其他方式，比如蓝牙，但是需要们自定义协议，后续FishBot开发教程中，小鱼将带你一起实现自定义协议。

二、编写代码

代码相对于串口通信只改变了三行，主要是设置wifi传输函数——`set_microros_wifi_transports`，需要传入wifi名称，密码，电脑IP,端口号四个参数。电脑IP获取方式请参考第三部分。

```c++
#include <Arduino.h>
#include <micro_ros_platformio.h>
#include <WiFi.h>
#include <rcl/rcl.h>
#include <rclc/rclc.h>
#include <rclc/executor.h>

rclc_executor_t executor;
rclc_support_t support;
rcl_allocator_t allocator;
rcl_node_t node;

void setup()
{
  Serial.begin(115200);
```

```
    // 设置通过WIFI进行MicroROS通信
    IPAddress agent_ip;
    agent_ip.fromString("192.168.2.105");
    // 设置wifi名称，密码，电脑IP,端口号
    set_microros_wifi_transports("fishbot", "12345678", agent_ip, 8888);
    // 延时时一段时间，等待设置完成
    delay(2000);
    // 初始化内存分配器
    allocator = rcl_get_default_allocator();
    // 创建初始化选项
    rclc_support_init(&support, 0, NULL, &allocator);
    // 创建节点 microros_wifi
    rclc_node_init_default(&node, "microros_wifi", "", &support);
    // 创建执行器
    rclc_executor_init(&executor, &support.context, 1, &allocator);
}

void loop()
{
    delay(100);
    // 循环处理数据
    rclc_executor_spin_some(&executor, RCL_MS_TO_NS(100));
}
```

三、电脑IP获取

打开一个新的终端，输入ip -4 a | grep inet看看电脑的ip地址，一般可以看到多个网卡的，此时可以忽略172(docker)和127(本地)开头的ip地址，剩下的一般就是我们要的ip地址，比如小鱼这里的就是192.168.0.105

```
fishros@fishros-MS-7D42:~$ ip -4 a | grep inet
    inet 127.0.0.1/8 scope host lo
    inet 192.168.2.105/24 brd 192.168.2.255 scope global dynamic noprefixroute e
np5s0
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
    inet 172.18.0.1/16 brd 172.18.255.255 scope global br-dbfbf7213281
    inet 172.20.10.4/28 brd 172.20.10.15 scope global dynamic noprefixroute enxa
a8fd9f18178
```

四、下载测试

4.1 编译下载

```
Wrote 8192 bytes (47 compressed) at 0x0000e000 in 0.1 seconds (effective 689.0 kbit/s)...
Hash of data verified.
Compressed 315600 bytes to 180652...
Writing at 0x00010000... (8 %)
Writing at 0x0001acfc... (16 %)
Writing at 0x00023efa... (25 %)
Writing at 0x00029604... (33 %)
Writing at 0x0002ec38... (41 %)
Writing at 0x00033f78... (50 %)
Writing at 0x00039488... (58 %)
Writing at 0x000405aa... (66 %)
Writing at 0x0004aa0e... (75 %)
Writing at 0x00051ec9... (83 %)
Writing at 0x0005740b... (91 %)
Writing at 0x0005cea4... (100 %)
Wrote 315600 bytes (180652 compressed) at 0x00010000 in 4.9 seconds (effective 519.3 kbit/s)
Hash of data verified.

Leaving...
Hard resetting via RTS pin...
================================================================ [SUCCESS] Took 9.55 seconds ===
* 终端将被任务重用，按任意键关闭。
```

4.2 启动Agent

打开终端输入指令，注意WIFI方式和之前的指令不太一样

```shell
docker run -it --rm -v /dev:/dev -v /dev/shm:/dev/shm --privileged --net=host microros/micro-ros-agent:$ROS_DISTRO udp4 --port 8888 -v6
```

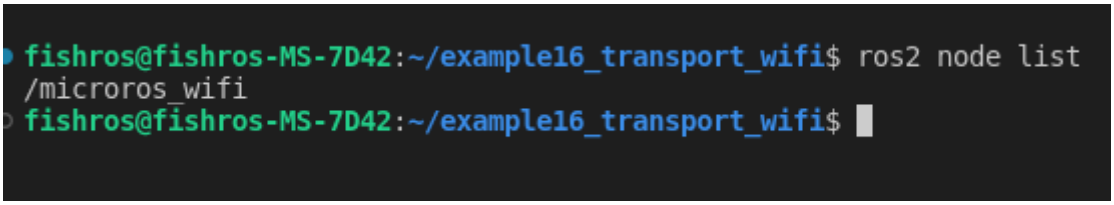
点击RST，正确连接上可以看到

```
fishros@fishros-MS-7D42:~/example16_transport_wifi$ docker run -it --rm -v /dev:/dev -v /dev/shm:/dev/shm --privileged --net=host microros/micro-ros-a
gent:$ROS_DISTRO udp4 --port 8888 -v6
[1674324822.643084] info     | UDPv4AgentLinux.cpp | init                     | running...             | port: 8888
[1674324822.643169] info     | Root.cpp           | set_verbose_level        | logger setup           | verbose_level: 6
[1674324829.076911] debug    | UDPv4AgentLinux.cpp | recv_message             | [==>> UDP <<==]        | client_key: 0x00000000, len: 24, data:
0000: 80 00 00 00 00 01 10 00 58 52 43 45 01 00 01 0F 5F 69 CE 89 81 00 FC 03
[1674324829.077246] info     | Root.cpp           | create_client            | create                 | client_key: 0x5F69CE89, session_id: 0x81
[1674324829.077357] info     | SessionManager.hpp | establish_session        | session established    | client_key: 0x5F69CE89, address: 192.168.2.100
:47138
[1674324829.077528] debug    | UDPv4AgentLinux.cpp | send_message             | [** <<UDP>> **]        | client_key: 0x5F69CE89, len: 19, data:
0000: 81 00 00 00 04 01 0B 00 00 00 58 52 43 45 01 00 01 0F 00
[1674324829.082557] debug    | UDPv4AgentLinux.cpp | recv_message             | [==>> UDP <<==]        | client_key: 0x5F69CE89, len: 44, data:
0000: 81 80 00 00 01 07 24 00 00 0A 00 01 01 03 00 00 16 00 00 00 00 01 00 00 0E 00 00 00 6D 69 63 72
0020: 6F 72 6F 73 5F 77 69 66 69 00 00 00
[1674324829.091023] info     | ProxyClient.cpp    | create_participant       | participant created    | client_key: 0x5F69CE89, participant_id: 0x000(
1)
[1674324829.091063] debug    | UDPv4AgentLinux.cpp | send_message             | [** <<UDP>> **]        | client_key: 0x5F69CE89, len: 14, data:
0000: 81 80 00 00 05 01 06 00 00 0A 00 01 00 00
[1674324829.091071] debug    | UDPv4AgentLinux.cpp | send_message             | [** <<UDP>> **]        | client_key: 0x5F69CE89, len: 13, data:
0000: 81 00 00 00 0A 01 05 00 01 00 00 00 80
[1674324829.100275] debug    | UDPv4AgentLinux.cpp | recv_message             | [==>> UDP <<==]        | client_key: 0x5F69CE89, len: 13, data:
0000: 81 00 00 00 0A 01 05 00 01 00 00 00 80
```

4.3 测试是否连通

```
ros2 node list
```

看到节点表示连接成功～



五、总结

本节开始，我们成功通过无线的方式将MicroROS连接到WIFI上来了，那问题来了，什么时候用串口模式，什么时候用WIFI模式呢？

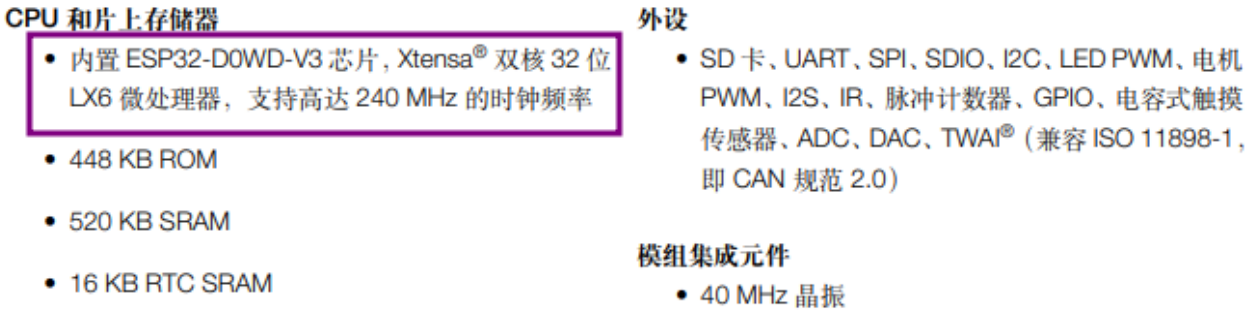
- 串口模式，适合当我们用树莓派等主控板在机器人上时，直接串口连接树莓派即可
- WIFI模式，像FishBot一样，直接无线驱动机器人，WIFI此时就很合适了

4.榨干性能-使用双核运行MicroROS

你好，我是爱吃鱼香ROS的小鱼。在硬件篇开始的第一节时，小鱼曾提到，我们所使用的开发板单片机是双核240M主频的，但是在后面的开发中我们并没有真正的使用了双核，主频也是使用的默认160MHZ。

所以本节小鱼带你一起带你一起提升主频并启动双核进行MicoROS的双核。

一、双核与RTOS介绍



所谓双核指的是ESP32单片机有两个内核，所有的外设都通过一个总线连接到两个内核上，也就是说，程序无论在哪个核上运行都可以操作硬件。



在前面的单片机开发平台介绍中，小鱼曾介绍ESP32的官方开发平台ESP-IDF的核心其实是基于开源的FreeRTOS优化而来的，而ESP32-Arduino则是对ESP-IDF的进一步封装，所以毋庸置疑，ESP32-Ardunio也是支持FreeRTOS的。

二、双核打印实验

接下来我们通过一个双核打印小实验来测试是否可以使用双核。

开始之前你需要了解两个函数

- `xPortGetCoreI()` 获取当前程序所运行的内核ID，ID有0和1
- `xTaskCreatePinnedToCore` 启动一个TASK并将其绑定到指定ID的内核，ID有0和1

新建`example17_micoros2core`，修改``，提高主频

```ini
; PlatformIO Project Configuration File
;
;   Build options: build flags, source filter
;   Upload options: custom upload port, speed and extra flags
;   Library options: dependencies, extra library storages
;   Advanced options: extra scripting
;
; Please visit documentation for the other options and examples
; https://docs.platformio.org/page/projectconf.html

[env:featheresp32]
platform = espressif32
board = featheresp32
framework = arduino
board_build.f_cpu = 240000000L
board_microros_transport = wifi
lib_deps =
    https://gitee.com/ohhuo/micro_ros_platformio.git
```

测试代码如下

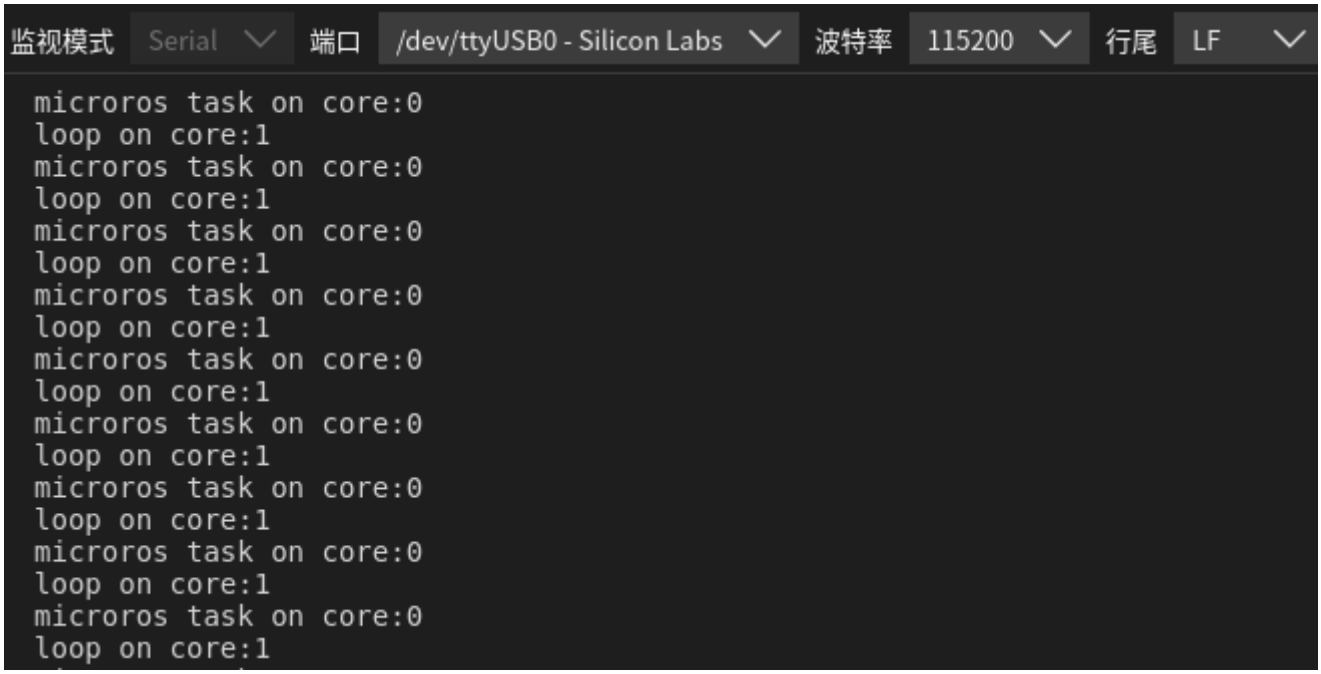
```c++
#include <Arduino.h>
/**
 * @brief MicroROSTASK,打印ID
 *
 * @param param
 */
void microros_task(void *param)
{
  while (true)
  {
    delay(1000);
```

```
    Serial.printf("microros_task on core:%d\n", xPortGetCoreID());
  }
}

void setup()
{
  Serial.begin(115200);
  /**
   * @brief 创建一个人物在Core 0 上
   * microros_task    任务函数
   * "microros_task"  任务名称
   * 10240      任务占用内存大小
   * NULL          任务参数，为空
   * 1               任务优先级
   * NULL     任务Handle可以为空
   * 0                 内核编号
   */
  xTaskCreatePinnedToCore(microros_task, "microros_task", 10240, NULL, 1, NULL,
0);
}

void loop()
{
  delay(1000);
  Serial.printf("loop on core:%d\n", xPortGetCoreID());
}
```

测试结果



三、MicroROS双核实验

编写代码，在上节的代码稍微做些修改即可。

```c++
#include <Arduino.h>
#include <micro_ros_platformio.h>
#include <WiFi.h>
#include <rcl/rcl.h>
#include <rclc/rclc.h>
#include <rclc/executor.h>

rclc_executor_t executor;
rclc_support_t support;
rcl_allocator_t allocator;
rcl_node_t node;

/**
 * @brief MicroROSTASK,打印ID
 *
 * @param param
 */
void microros_task(void *param)
{
  // 设置通过WIFI进行MicroROS通信
  IPAddress agent_ip;
```

```
  agent_ip.fromString("192.168.2.105");
  // 设置wifi名称，密码，电脑IP,端口号
  set_microros_wifi_transports("fishbot", "12345678", agent_ip, 8888);
  // 延时时一段时间，等待设置完成
  delay(2000);
  // 初始化内存分配器
  allocator = rcl_get_default_allocator();
  // 创建初始化选项
  rclc_support_init(&support, 0, NULL, &allocator);
  // 创建节点 microros_wifi
  rclc_node_init_default(&node, "microros_wifi", "", &support);
  // 创建执行器
  rclc_executor_init(&executor, &support.context, 1, &allocator);
  while (true)
  {
    delay(100);
    Serial.printf("microros_task on core:%d\n", xPortGetCoreID());
    // 循环处理数据
    rclc_executor_spin_some(&executor, RCL_MS_TO_NS(100));
  }
}

void setup()
{
  Serial.begin(115200);
  /**
   * @brief 创建一个人物在Core 0 上
   * microros_task    任务函数
   * "microros_task"  任务名称
   * 10240      任务占用内存大小
   * NULL          任务参数，为空
   * 1               任务优先级
   * NULL     任务Handle可以为空
   * 0                 内核编号
   */
  xTaskCreatePinnedToCore(microros_task, "microros_task", 10240, NULL, 1, NULL,
0);
}

void loop()
{
  delay(1000);
```

```
    Serial.printf("do some control on core:%d\n", xPortGetCoreID());
}
```

下载后，运行Agent即可测试

```
docker run -it --rm -v /dev:/dev -v /dev/shm:/dev/shm --privileged --net=host
microros/micro-ros-agent:$ROS_DISTRO udp4 --port 8888 -v6
```

```
fishros@fishros-MS-7D42:~/example16_transport_wifi$ docker run -it --rm -v /dev:/dev -v /dev/shm:/dev/shm --privileged --net=host microros/micro-ros-a
gent:$ROS_DISTRO udp4 --port 8888 -v6
[1674324822.643084] info     | UDPv4AgentLinux.cpp | init                     | running...             | port: 8888
[1674324822.643169] info     | Root.cpp           | set_verbose_level        | logger setup           | verbose_level: 6
[1674324829.076911] debug    | UDPv4AgentLinux.cpp | recv_message             | [==>> UDP <<==]        | client_key: 0x00000000, len: 24, data:
0000: 80 00 00 00 00 01 10 00 58 52 43 45 01 00 01 0F 5F 69 CE 89 81 00 FC 03
[1674324829.077246] info     | Root.cpp           | create_client            | create                 | client_key: 0x5F69CE89, session_id: 0x81
[1674324829.077357] info     | SessionManager.hpp | establish_session        | session established    | client_key: 0x5F69CE89, address: 192.168.2.100
:47138
[1674324829.077528] debug    | UDPv4AgentLinux.cpp | send_message             | [** <<UDP>> **]        | client_key: 0x5F69CE89, len: 19, data:
0000: 81 00 00 00 04 01 0B 00 00 00 58 52 43 45 01 00 01 0F 00
[1674324829.082557] debug    | UDPv4AgentLinux.cpp | recv_message             | [==>> UDP <<==]        | client_key: 0x5F69CE89, len: 44, data:
0000: 81 80 00 00 01 07 24 00 00 0A 00 01 01 03 00 00 16 00 00 00 00 01 00 00 0E 00 00 00 6D 69 63 72
0020: 6F 72 6F 73 5F 77 69 66 69 00 00 00
[1674324829.091023] info     | ProxyClient.cpp    | create_participant       | participant created    | client_key: 0x5F69CE89, participant_id: 0x000(
1)
[1674324829.091063] debug    | UDPv4AgentLinux.cpp | send_message             | [** <<UDP>> **]        | client_key: 0x5F69CE89, len: 14, data:
0000: 81 80 00 00 05 01 06 00 00 0A 00 01 00 00
[1674324829.091071] debug    | UDPv4AgentLinux.cpp | send_message             | [** <<UDP>> **]        | client_key: 0x5F69CE89, len: 13, data:
0000: 81 00 00 00 0A 01 05 00 01 00 00 00 80
[1674324829.100275] debug    | UDPv4AgentLinux.cpp | recv_message             | [==>> UDP <<==]        | client_key: 0x5F69CE89, len: 13, data:
0000: 81 00 00 00 0A 01 05 00 01 00 00 00 80
```

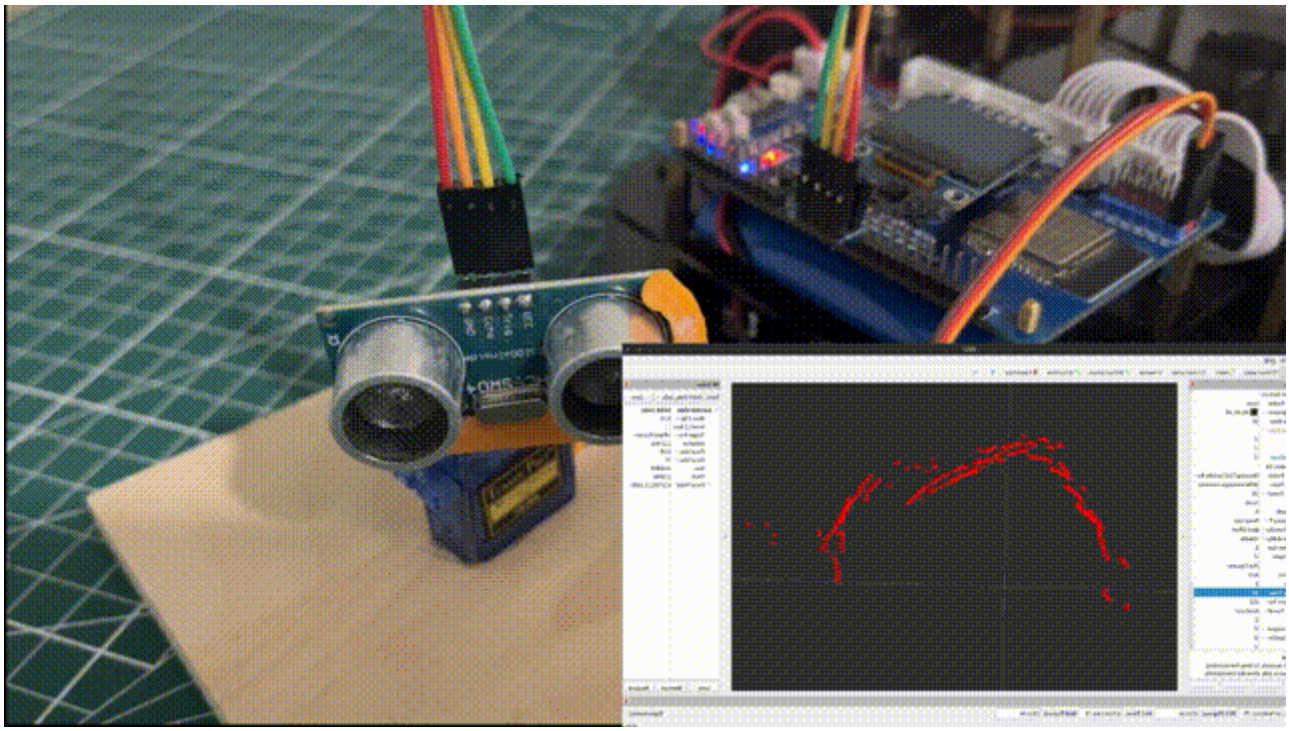
四、总结

本节通过配置和启动新任务成功开启了另一内核并完成MicroROS相关的传输。你可能会问使用双核240M有什么坏处，坏处就是耗电，不过相比我们的电池来说是不值一提的。

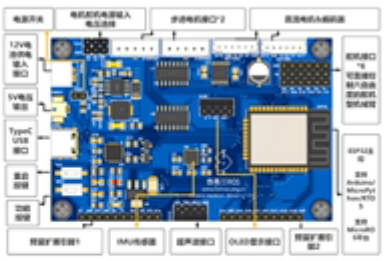
第 15 章 ROS2硬件实战（自制简易雷达）

你好，我是小鱼。本章是ROS2硬件实战教程的实践章节，我们将通过自制一个简易雷达同时学习使用传感器、执行器和MicroROS。

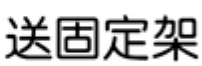
最终效果如下：



> 上图中，主控板、超声波、舵机及支架 板可以在小鱼的店铺直接购买，性价比接地气，直达链接：https://item.taobao.com/item.htm?id=695473143304。











技术交流&&问题求助：

- **微信公众号及交流群：鱼香ROS**

- **小鱼微信：AilotRobot**
- **QQ交流群：139707339**
- 版权保护：已加入“维权骑士”（rightknights.com）的版权保护计划

1.简易雷达原理介绍

你好，我是爱吃鱼香ROS的小鱼。在正式开始制作我们的简易雷达前，我们先了解下原理。

我们所说的雷达是一种测距设备，比如FishBot上搭载的雷达就可以实现360度的旋转测距——测量指定角度前方障碍物的距离。

所以要实现一个简易雷达，我们必须要有一个可以测量距离的传感器，一个可以指定角度的电机。

一、测距传感器超声波

百度百科介绍

超声波传感器是将超声波信号转换成其它能量信号（通常是电信号）的传感器。超声波是**振动频率**高于20kHz的机械波。它具有频率高、波长短、绕射现象小，特别是方向性好、能够成为**射线**而定向传播等特点。超声波对液体、固体的穿透本领很大，尤其是在阳光不透明的固体中。超声波碰到杂质或分界面会产生显著反射形成反射回波，碰到活动物体能产生**多普勒效应**。超声波传感器广泛应用在工业、国防、生物医学等方面。

接着看看长什么样子：



便宜的就长这样子，一共两个头，一个头用于发送波，一个头接收波。

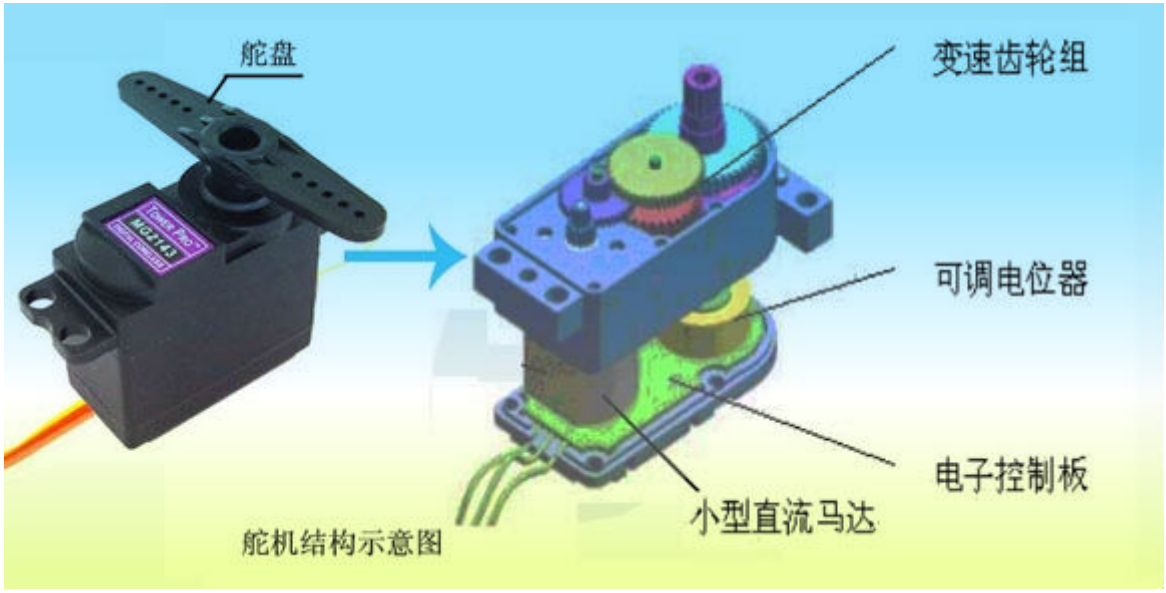
那么超声波传感器原理是什么呢？

```
距离=(发送时间-接收时间)*速度/2
```

下一节我们将通过代码根据这一原理进行距离测量测试。

二、常用执行器舵机

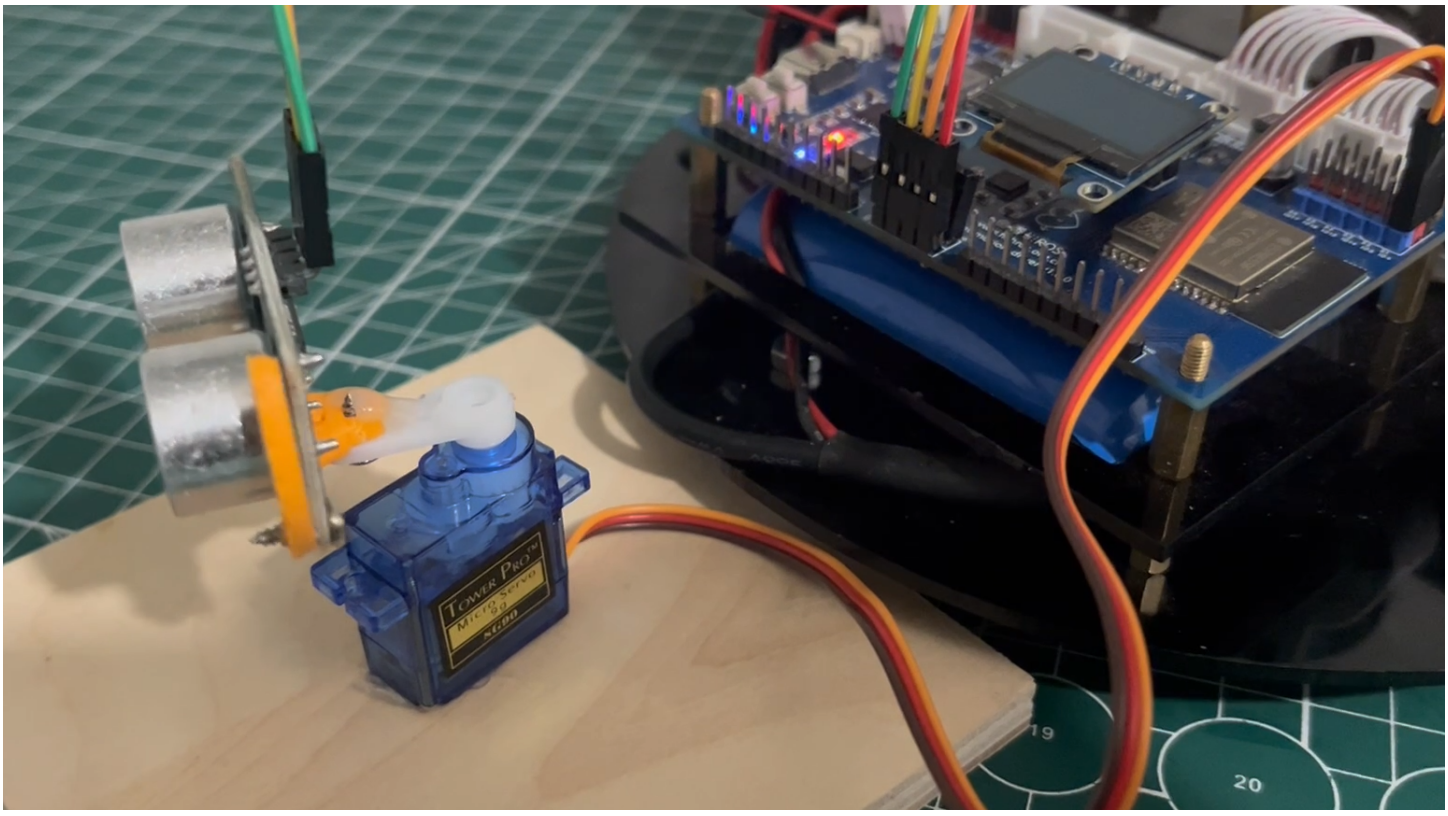
舵机是可以根据指定角度进行旋转的特殊电机。



其硬件结构如上图所示，当我们把角度信息发送到控制板时，控制板通过电位器测量出当前的角度，然后根据当前角度和目标角度的角度差控制电机旋转，直到角度差变的几乎为零。

三、结构设计

主控板依然使用MicroROS学习板，购买雷达套餐（购买链接：https://item.taobao.com/item.htm?id=695473143304）的小伙伴可以通过赠送的支架将超声波固定到舵机上，也可以自行用胶枪之类的固定。最终的结构示意图如下



2.测量距离学会超声波传感器

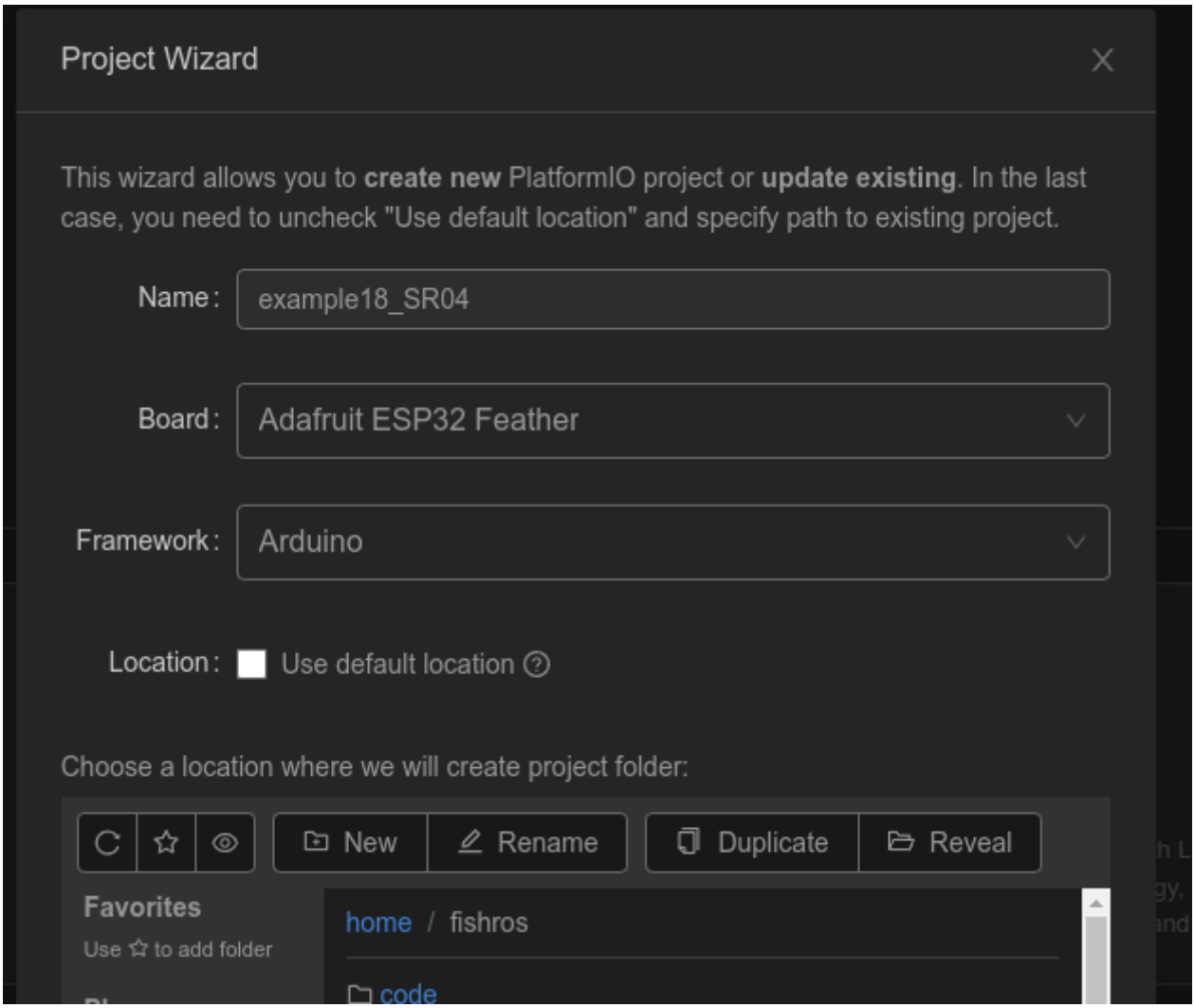
你好，我是爱吃鱼香ROS的小鱼。上一节简单的介绍了超声波传感器，但是没有介绍如何通过代码使用，本节我们尝试使用并封装超声波模块。

我们使用的超声波模块一共有四个引脚，分别是

- `TRIG` 即发送引脚，用于发送超声波
- `ECHO` 即接收引脚，用于接收反射回来的超声波
- `VCC` 电源接5V
- `GND` 电源地

一、新建工程

新建`example18_sr04`



二、编写代码

带注释的代码如下

```c++
#include <Arduino.h>
#define Trig 27 // 设定SR04连接的Arduino引脚
#define Echo 21

void setup()
{
  Serial.begin(115200);
  pinMode(Trig, OUTPUT); // 初始化舵机和超声波
  pinMode(Echo, INPUT);  // 要检测引脚上输入的脉冲宽度，需要先设置为输入状态
}

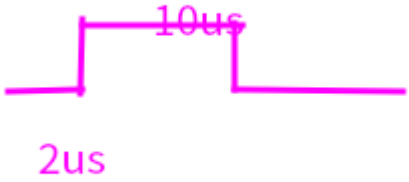
void loop()
{
  static double mtime;
  digitalWrite(Trig, LOW); // 测量距离
  delayMicroseconds(2);    // 延时2us
  digitalWrite(Trig, HIGH);
  delayMicroseconds(10); // 产生一个10us的高脉冲去触发SR04
  digitalWrite(Trig, LOW);
  mtime = pulseIn(Echo, HIGH);                  // 检测脉冲宽度，注意返回值是微秒us
  float detect_distance = mtime / 58.0 / 100.0; // 计算出距离,输出的距离的单位是厘米cm
  Serial.printf("distance=%f\n", detect_distance);
  delay(500);
}
```

三、代码注解

核心代码分为两部分

3.1发送超声

方波产生，低-高-低




```c++
digitalWrite(Trig, LOW); // 测量距离
delayMicroseconds(2);    // 延时2us
digitalWrite(Trig, HIGH);
delayMicroseconds(10); // 产生一个10us的高脉冲去触发SR04
digitalWrite(Trig, LOW);
```

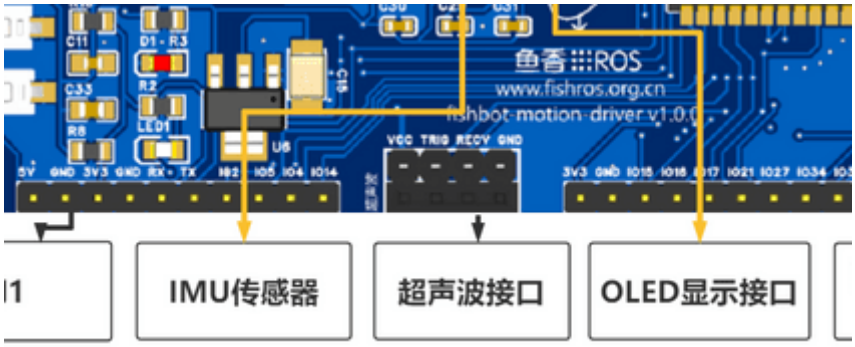
3.2 检测回响计算距离

```c++
mtime = pulseIn(Echo, HIGH);                    // 检测脉冲宽度，注意返回值是微秒us
float detect_distance = mtime / 58.0 / 100.0; // 计算出距离,输出的距离的单位是米m
```

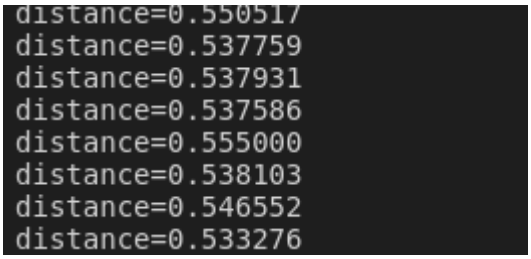
58是一个时间系数，根据声音在空气中传播速度计算而来。`pulseIn` 函数用于检测某个引脚从当前时间跳变到高电平之间持续的时间。

四、下载测试

将超声波模块连接到开发板上的超声波接口上



下载代码，打开串口，查看距离不断变化



五、总结

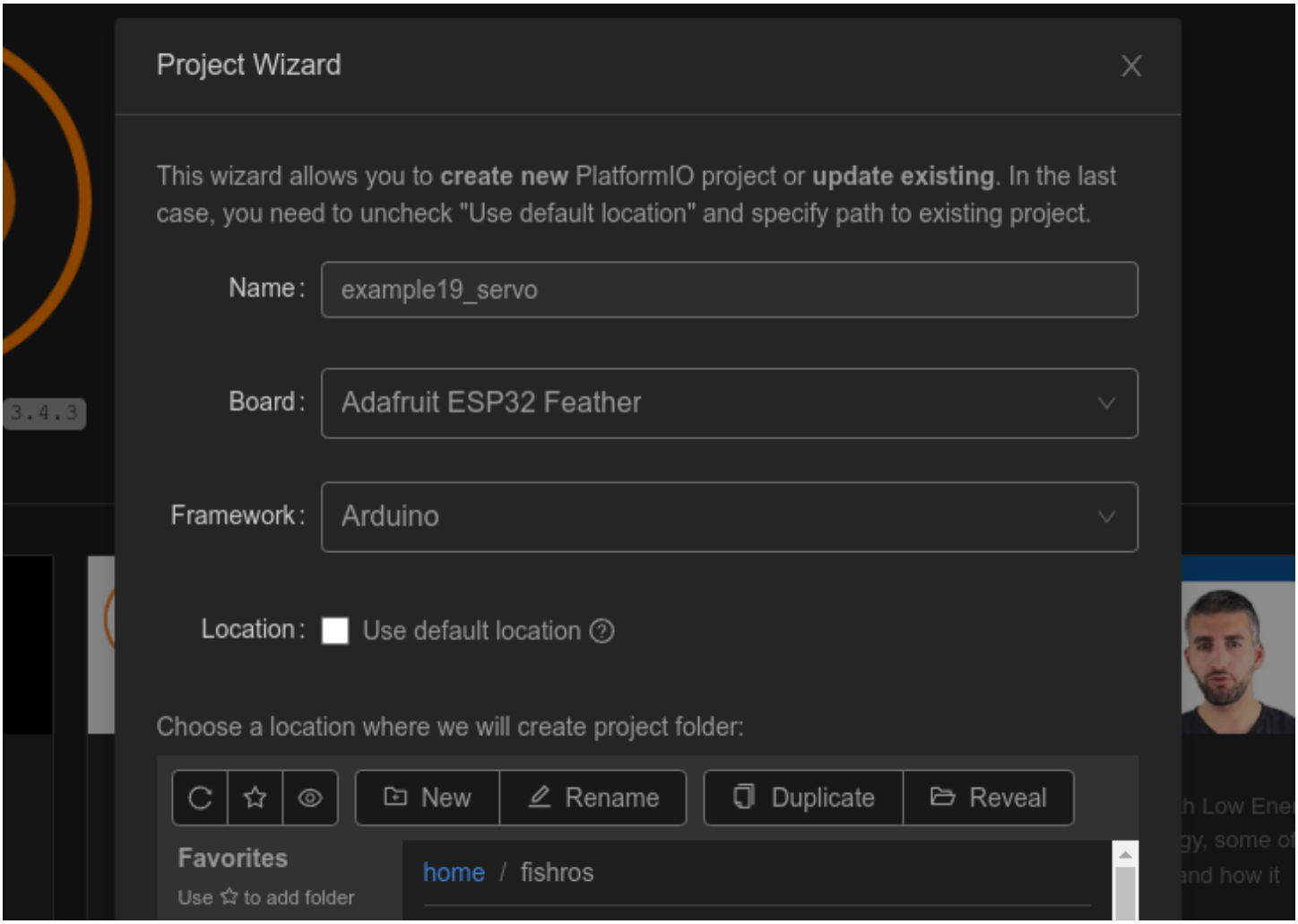
本节我们成功实现使用超声波实现距离测量功能，下一节我们尝试使用第三方库驱动舵机。

3.控制舵机学会使用执行器

你好，我是爱吃鱼香ROS的小鱼。本节我们尝试使用第三方库来驱动舵机，实现让舵机指针指向任意角度。

一、新建工程

新建 `example19_servo`



在 `platformio.ini` 中添加舵机库

```
; PlatformIO Project Configuration File
;
;   Build options: build flags, source filter
;   Upload options: custom upload port, speed and extra flags
;   Library options: dependencies, extra library storages
;   Advanced options: extra scripting
;
```

```
; Please visit documentation for the other options and examples
; https://docs.platformio.org/page/projectconf.html

[env:featheresp32]
platform = espressif32
board = featheresp32
framework = arduino
lib_deps =
    madhephaestus/ESP32Servo@^0.12.0
```

二、编写代码

带注释代码如下。

```c++
#include <Arduino.h>
#include <ESP32Servo.h>

Servo servo1; // 创建对象

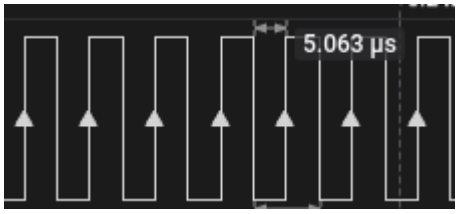
void setup()
{
  servo1.setPeriodHertz(50);    // 舵机控制周期为50hz，即一个周期1000/50=20ms
  servo1.attach(4, 500, 2500); // 使用GPIO4作为舵机1信号引脚，占空比为500-2500us即 0.5-2.5ms
  servo1.write(90.0);           // 设置90度
}

void loop()
{
  for (int i = 0; i < int(180); i++)
  {
    servo1.write(i); // 设置角度
    delay(5);
  }
}
```

三、代码注解

这里主要需要介绍的是关于舵机的控制周期及占空比是如何设置的，这里设计到了PWM相关的知识。

PWM即脉宽调制（Pulse-width Modulation, PWM），之前在I2C介绍章节中，小鱼曾介绍过通信时SCL上就是一个固定周期的脉冲



PWM有两个重要的参数，第一个是周期，就像是正弦波，其周期就是2pi，指的是多久循环一次，我们这里设置的是50HZ，也就是说20ms。

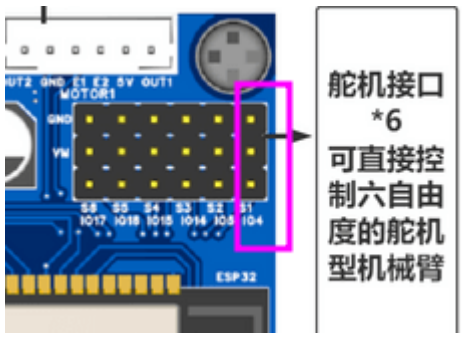
在一个周期里，引脚高电平的时间就称为占空比，这里我们设置的是0.5ms-2.5ms之间作为舵机控制的占空比范围。

换句话说，假设我们设置当前舵机角度为90度，此时占空比

```
占空比 = 500+90*(2500-500)/180
```

四、下载测试

将舵机插在S1接口，注意黄色线接蓝色信号。



接着下载代码，观察舵机。

五、总结

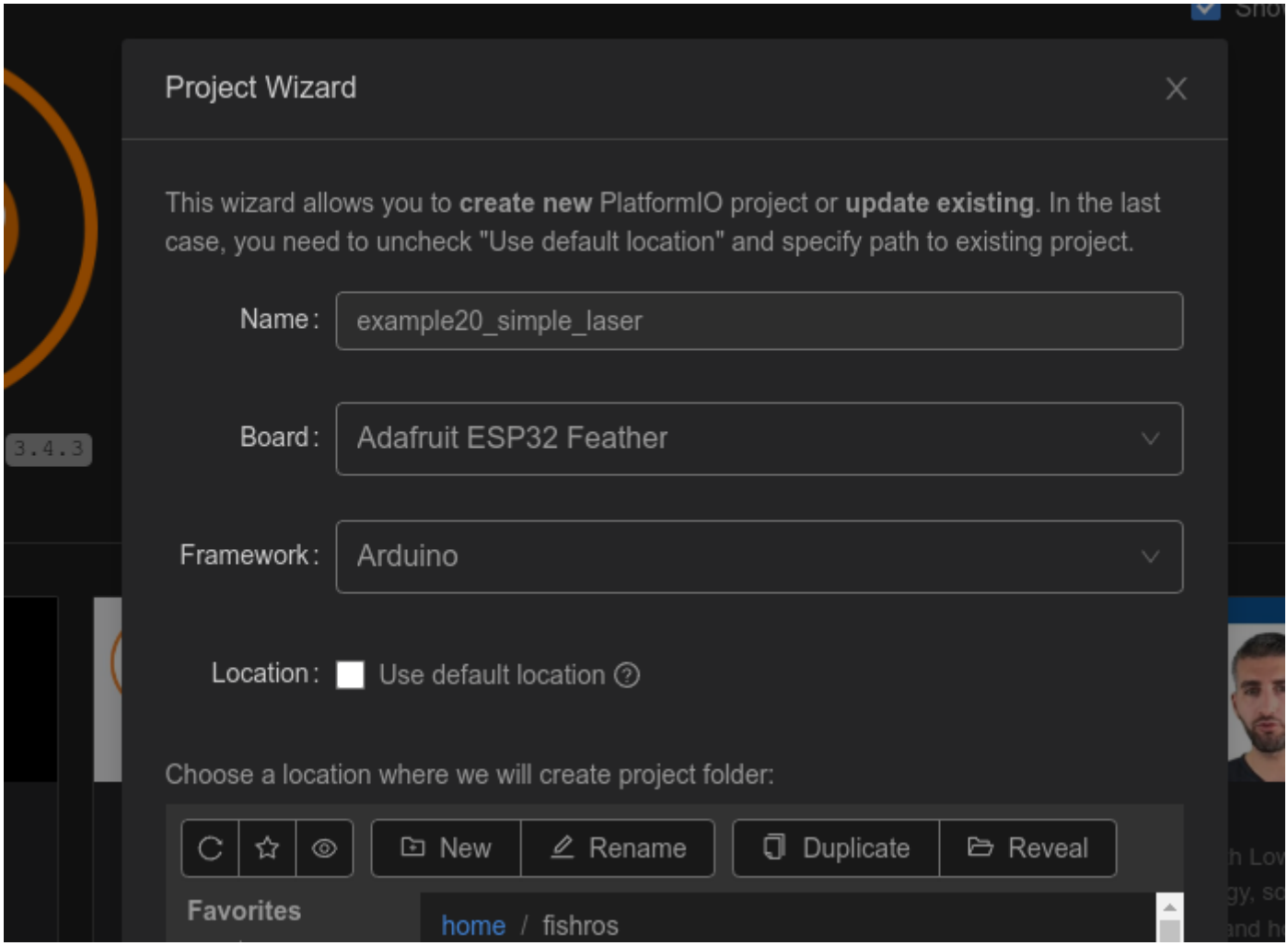
本节我们通过三方库完成了对舵机的控制，下一节我们正式将舵机和超声波结合起来，测量指定角度下的距离。

4.舵机+超声波循环扫描

你好，我是爱吃鱼香ROS的小鱼。本节我们尝试将超声波的舵机结合起来实现循环扫描功能。

一、新建工程

新建`example20_simple_laser`



添加依赖，这里顺便吧microros的添加上，下一节直接使用

```c++
; PlatformIO Project Configuration File
;
;   Build options: build flags, source filter
;   Upload options: custom upload port, speed and extra flags
;   Library options: dependencies, extra library storages
;   Advanced options: extra scripting
;
```

```
; Please visit documentation for the other options and examples
; https://docs.platformio.org/page/projectconf.html

[env:featheresp32]
platform = espressif32
board = featheresp32
framework = arduino
board_microros_transport = wifi
lib_deps =
    https://gitee.com/ohhuo/micro_ros_platformio.git
    madhephaestus/ESP32Servo@^0.12.0
```

二、编写代码

原理是先控制舵机走到某个角度，接着调用超声波测量距离，这里小鱼将超声波测距离封装了一个函数，并用一个数组存储10个历史数据。

```c++
#include <Arduino.h>
#include <micro_ros_platformio.h>
#include <WiFi.h>
#include <ESP32Servo.h>


#define Trig 27 // 设定SR04连接的Arduino引脚
#define Echo 21

Servo servo1;

float get_distance(int angle)
{
  static double mtime;
  servo1.write(angle);      // 移动到指定角度
  delay(25);                // 稳定身形
  digitalWrite(Trig, LOW); // 测量距离
  delayMicroseconds(2);
  digitalWrite(Trig, HIGH);
  delayMicroseconds(10); // 产生一个10us的高脉冲去触发SR04
  digitalWrite(Trig, LOW);
  mtime = pulseIn(Echo, HIGH);                    // 检测脉冲宽度，注意返回值是微秒us
```

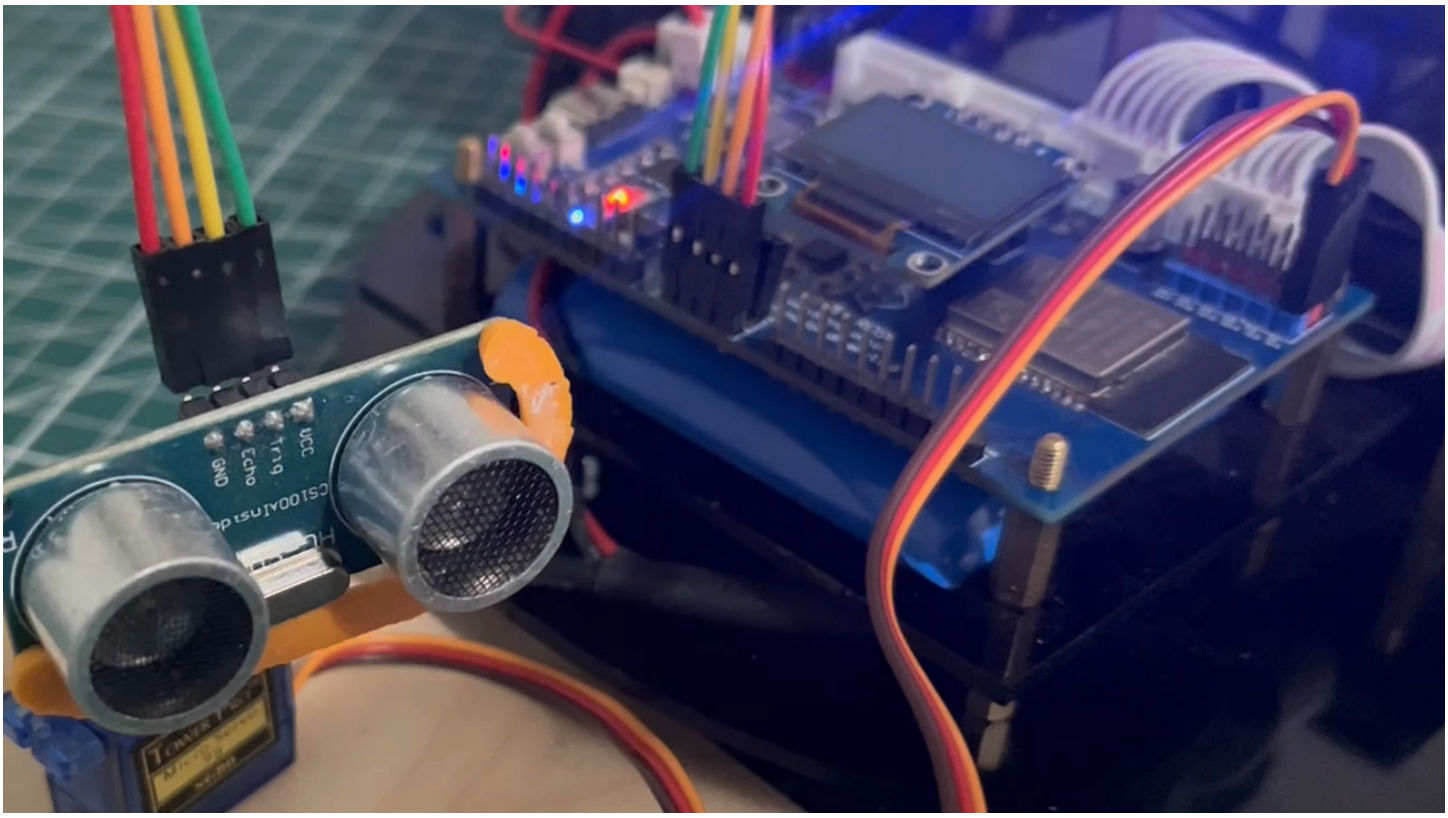
```
  float detect_distance = mtime / 58.0 / 100.0; // 计算出距离,输出的距离的单位是厘米cm
  Serial.printf("point(%d,%f)\n", angle, detect_distance);
  return detect_distance;
}

void setup()
{
  Serial.begin(115200);
  pinMode(Trig, OUTPUT);      // 初始化舵机和超声波
  pinMode(Echo, INPUT);       // 要检测引脚上输入的脉冲宽度，需要先设置为输入状态
  servo1.setPeriodHertz(50); // Standard 50hz servo
  servo1.attach(4, 500, 2500);
  servo1.write(90.0);
}

void loop()
{
  for (int i = 0; i < 180; i++)
  {
    float distance = get_distance(i);
    delay(10);
  }
}
```

三、下载测试

连接好超声波和舵机。



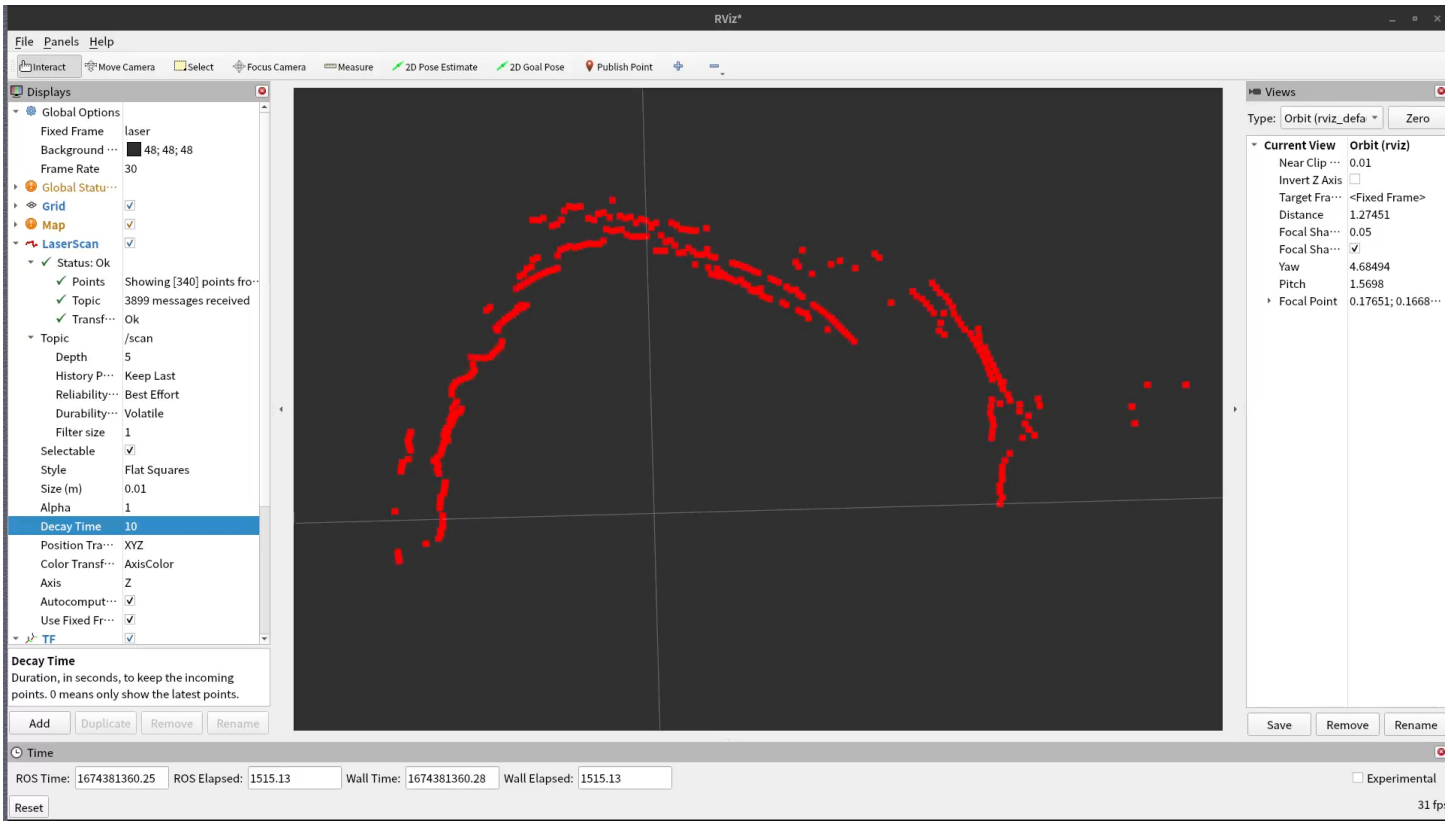
下载代码，观察串口输出

```
point(7,1.393448)
point(8,1.393621)
point(9,1.393621)
point(10,1.389310)
point(11,0.524828)
point(12,1.389138)
point(13,1.389310)
point(14,1.389310)
point(15,0.523793)
point(16,0.523793)
point(17,0.523966)
point(18,1.389138)
point(19,5.688965)
point(20,0.520345)
point(21,1.393965)
point(22,1.393965)
point(23,1.394310)
```

这里显示的就是角度以及距离信息。

5.可视化点云-雷达消息合成

你好，我是爱吃鱼香ROS的小鱼。上一节完成了指定角度距离的测量这一节我们将其合成ROS的laserscan消息，并将其通过microros发布到上位机，最终实现rviz2的可视化。



一、雷达消息介绍

使用指令 `ros2 interface show sensor_msgs/msg/LaserScan` ，可以看到ROS2对雷达数据接口的定义。

```
# Single scan from a planar laser range-finder
#
# If you have another ranging device with different behavior (e.g. a sonar
# array), please find or create a different message, since applications
# will make fairly laser-specific assumptions about this data

std_msgs/Header header # timestamp in the header is the acquisition time of
    builtin_interfaces/Time stamp
        int32 sec
        uint32 nanosec
    string frame_id
                            # the first ray in the scan.
```

```
                            #
                            # in frame frame_id, angles are measured around
                            # the positive Z axis (counterclockwise, if Z is up)
                            # with zero angle being forward along the x axis

float32 angle_min           # start angle of the scan [rad]
float32 angle_max           # end angle of the scan [rad]
float32 angle_increment     # angular distance between measurements [rad]

float32 time_increment      # time between measurements [seconds] - if your
scanner
                            # is moving, this will be used in interpolating
position
                            # of 3d points
float32 scan_time           # time between scans [seconds]

float32 range_min           # minimum range value [m]
float32 range_max           # maximum range value [m]

float32[] ranges            # range data [m]
                            # (Note: values < range_min or > range_max should be
discarded)
float32[] intensities       # intensity data [device-specific units].  If your
                            # device does not provide intensities, please leave
                            # the array empty.
```

1.1 header部分

头部分，主要是设置雷达的frame_id和时间戳，在microros中可以这样赋值

```c++
pub_msg.header.frame_id = micro_ros_string_utilities_set(pub_msg.header.frame_id,
"laser"); // 初始化消息内容
int64_t current_time = rmw_uros_epoch_millis();
pub_msg.header.stamp.sec = current_time * 1e-3;
pub_msg.header.stamp.nanosec = current_time - pub_msg.header.stamp.sec * 1000;
```

1.2 数据部分

- angle_min 当前数据中最小的测量角度
- angle_max 当前数据中最大的测量角度
- angle_increment 我们默认就是一次1度，所以可以直接写

```
pub_msg.angle_increment = 1.0 / 180 * PI;
```

- time_increment 每个数据之间递增的时间，可以直接使用扫描的总之间除点数
- scan_time 扫描时间，开始扫描到结束扫描的时间
- range_min 最小范围可以直接赋值 我们设置成 `0.05` 即5cm
- range_max 最大范围，我们直接设置成5.0m
- ranges 测量的距离值数组
- intensities 测量的强度，这里我们直接忽略即可

二、代码编写

直接在上一节工程上修改，全部代码如下，一次我们发布10个点，并且启动了ESP32的双核，同时采取了时间同步，保证雷达数据的时间戳正常。

```c++
#include <Arduino.h>
#include <micro_ros_platformio.h>
#include <WiFi.h>
#include <rcl/rcl.h>
#include <rclc/rclc.h>
#include <rclc/executor.h>
#include <ESP32Servo.h>

#include <sensor_msgs/msg/laser_scan.h>
#include <micro_ros_utilities/string_utilities.h>

#define PCOUNT 10
#define Trig 27 // 设定SR04连接的Arduino引脚
#define Echo 21
```

```
rclc_executor_t executor;
rclc_support_t support;
rcl_allocator_t allocator;
rcl_node_t node;

rcl_publisher_t publisher;           // 声明话题发布者
sensor_msgs__msg__LaserScan pub_msg; // 声明消息文件

Servo servo1;
bool connected_agent = false;

void microros_task(void *param)
{

  IPAddress agent_ip;                                                  // 设置通过
WIFI进行MicroROS通信
  agent_ip.fromString("192.168.2.105");                                // 从字符串
获取IP地址
  set_microros_wifi_transports("fishbot", "12345678", agent_ip, 8888);   // 设置
wifi名称，密码，电脑IP,端口号
  delay(2000);                                                         // 延时时一
段时间，等待设置完成
  allocator = rcl_get_default_allocator();                             // 初始化内
存分配器
  rclc_support_init(&support, 0, NULL, &allocator);                    // 创建初始
化选项
  rclc_node_init_default(&node, "example20_simple_laser", "", &support); // 创建节点
  rclc_publisher_init_default(                                         // 发布初始
化
      &publisher,
      &node,
      ROSIDL_GET_MSG_TYPE_SUPPORT(sensor_msgs, msg, LaserScan),
      "/scan");

  rclc_executor_init(&executor, &support.context, 1, &allocator);
// 创建执行器
  pub_msg.header.frame_id =
micro_ros_string_utilities_set(pub_msg.header.frame_id, "laser"); // 初始化消息内容
  pub_msg.angle_increment = 1.0 / 180 * PI;
  pub_msg.range_min = 0.05;
  pub_msg.range_max = 5.0;
```

```
  while (true)
  {
    delay(10);
    if (!rmw_uros_epoch_synchronized()) // 判断时间是否同步
    {
      rmw_uros_sync_session(1000); //  同步时间
      continue;
    }
    connected_agent = true;
    rclc_executor_spin_some(&executor, RCL_MS_TO_NS(100)); // 循环处理数据
  }
}

float get_distance(int angle)
{
  static double mtime;
  servo1.write(angle);     // 移动到指定角度
  delay(25);               // 稳定身形
  digitalWrite(Trig, LOW); // 测量距离
  delayMicroseconds(2);
  digitalWrite(Trig, HIGH);
  delayMicroseconds(10); // 产生一个10us的高脉冲去触发SR04
  digitalWrite(Trig, LOW);
  mtime = pulseIn(Echo, HIGH);                  // 检测脉冲宽度，注意返回值是微秒us
  float detect_distance = mtime / 58.0 / 100.0; // 计算出距离,输出的距离的单位是厘米cm
  Serial.printf("point(%d,%f)\n", angle, detect_distance);
  return detect_distance;
}

void setup()
{
  Serial.begin(115200);
  pinMode(Trig, OUTPUT);     // 初始化舵机和超声波
  pinMode(Echo, INPUT);      // 要检测引脚上输入的脉冲宽度，需要先设置为输入状态
  servo1.setPeriodHertz(50); // Standard 50hz servo
  servo1.attach(4, 500, 2500);
  servo1.write(90.0);
  xTaskCreatePinnedToCore(microros_task, "microros_task", 10240, NULL, 1, NULL,
0);
}
```

```
void loop()
{
  if (!connected_agent)
    return;

  static float ranges[PCOUNT + 1];
  for (int i = 0; i < int(180 / PCOUNT); i++)
  {
    int64_t start_scan_time = rmw_uros_epoch_millis();
    for (int j = 0; j < PCOUNT; j++)
    {
      int angle = i * 10 + j;
      ranges[j] = get_distance(angle);
    }
    pub_msg.angle_min = float(i * 10) / 180 * PI;       // 结束角度
    pub_msg.angle_max = float(i * (10 + 1)) / 180 * PI; // 结束角度

    int64_t current_time = rmw_uros_epoch_millis();
    pub_msg.scan_time = float(current_time - start_scan_time) * 1e-3;
    pub_msg.time_increment = pub_msg.scan_time / PCOUNT;
    pub_msg.header.stamp.sec = current_time * 1e-3;
    pub_msg.header.stamp.nanosec = current_time - pub_msg.header.stamp.sec * 1000;
    pub_msg.ranges.data = ranges;
    pub_msg.ranges.capacity = PCOUNT;
    pub_msg.ranges.size = PCOUNT;
    rcl_publish(&publisher, &pub_msg, NULL);
    delay(10);
  }
}
```

三、下载测试

下载代码

```
Compressed 17472 bytes to 12125...
Writing at 0x00001000... (100 %)
Wrote 17472 bytes (12125 compressed) at 0x00001000 in 0.6 seconds (effective 242.4 kbit/s)...
Hash of data verified.
Compressed 3072 bytes to 128...
Writing at 0x00008000... (100 %)
Wrote 3072 bytes (128 compressed) at 0x00008000 in 0.0 seconds (effective 521.9 kbit/s)...
Hash of data verified.
Compressed 8192 bytes to 47...
Writing at 0x0000e000... (100 %)
Wrote 8192 bytes (47 compressed) at 0x0000e000 in 0.1 seconds (effective 660.1 kbit/s)...
Hash of data verified.
Compressed 783440 bytes to 510177...
Writing at 0x00010000... (3 %)
Writing at 0x0001c2e8... (6 %)
Writing at 0x000267f4... (9 %)
Writing at 0x00030b50... (12 %)
Writing at 0x00036457... (15 %)
Writing at 0x0003bc2f... (18 %)
```

启动agent。

```shell
docker run -it --rm -v /dev:/dev -v /dev/shm:/dev/shm --privileged --net=host microros/micro-ros-agent:$ROS_DISTRO udp4 --port 8888 -v6
```

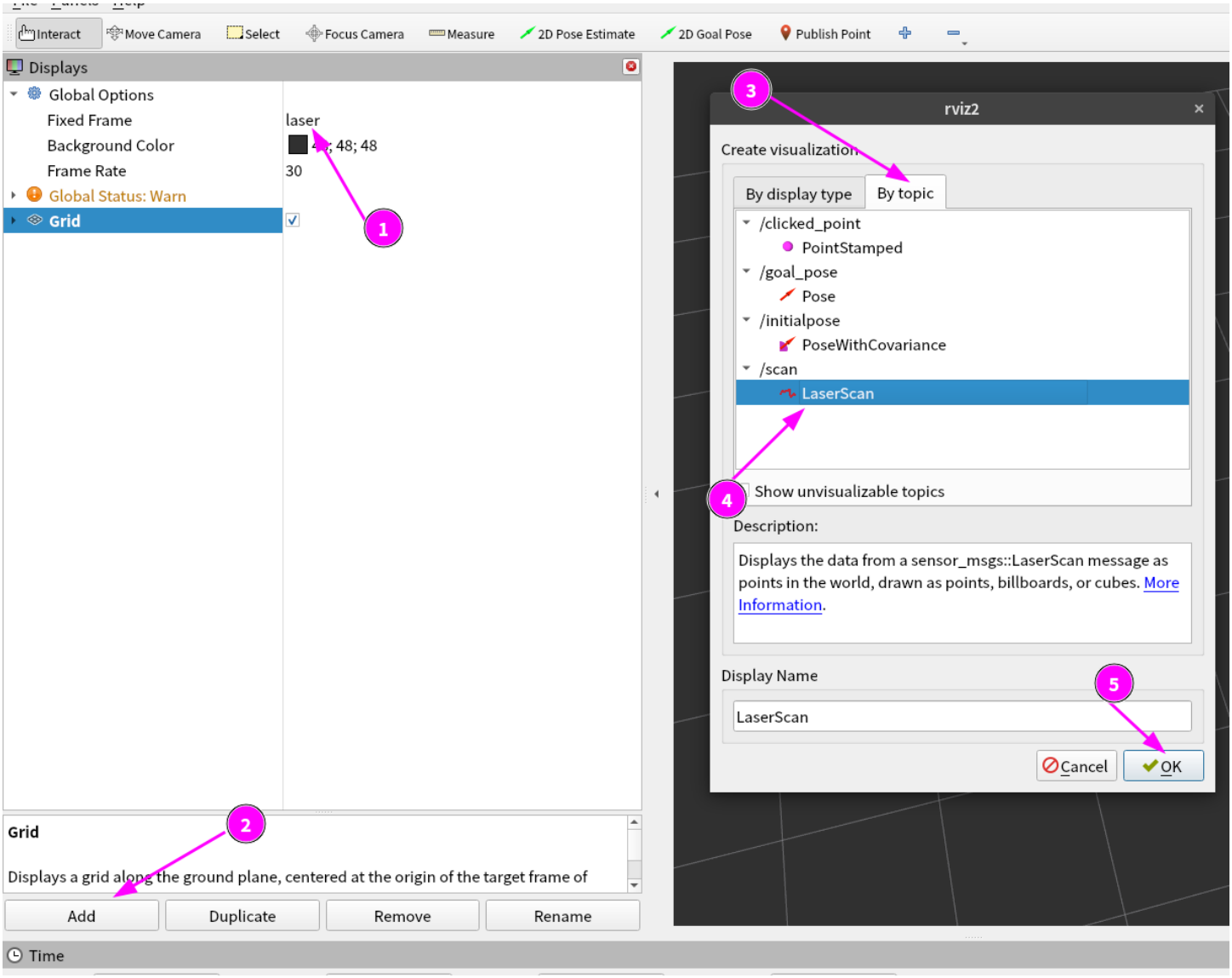
```
[1674475970.880966] debug     | UDPv4AgentLinux.cpp | send_message
[** <<UDP>> **]         | client_key: 0x19DAB9BE, len: 13, data:
0000: 81 00 00 00 0A 01 05 00 31 00 00 00 80
[1674475971.208438] debug     | UDPv4AgentLinux.cpp | recv_message
[==>> UDP <<==]         | client_key: 0x19DAB9BE, len: 108, data:
0000: 81 80 31 00 07 01 64 00 00 3B 00 05 C3 79 CE 63 CB 00 00 00 06 00 00 00 6
 61 73 65 72 00 6E 00
0020: 00 00 00 00 00 00 00 00 35 FA 8E 3C 6D C5 FE 3C 64 3B 9F 3E CD CC 4C 3D 0
 00 A0 40 0A 00 00 00
0040: C8 25 C1 3E A6 FF BC 3E CF 61 BF 3E 98 0A B6 40 59 71 C3 3E FC E1 C0 3E 9
 C3 BE 3E 9F C3 BE 3E
0060: D1 F0 BE 3E 9D 34 BF 3E 00 00 00 00
[1674475971.208672] debug     | DataWriter.cpp      | write                    |
** <<DDS>> **]         | client_key: 0x00000000, len: 96, data:
0000: C3 79 CE 63 CB 00 00 00 06 00 00 00 6C 61 73 65 72 00 6E 00 00 00 00 00 0
 00 00 00 35 FA 8E 3C
0020: 6D C5 FE 3C 64 3B 9F 3E CD CC 4C 3D 00 00 A0 40 0A 00 00 00 C8 25 C1 3E A
 FF BC 3E CF 61 BF 3E
0040: 98 0A B6 40 59 71 C3 3E FC E1 C0 3E 9F C3 BE 3E 9F C3 BE 3E D1 F0 BE 3E 9
 34 BF 3E 00 00 00 00
[1674475971.208880] debug     | UDPv4AgentLinux.cpp | send_message
[** <<UDP>> **]         | client_key: 0x19DAB9BE, len: 13, data:
0000: 81 00 00 00 0A 01 05 00 32 00 00 00 80
```

测试

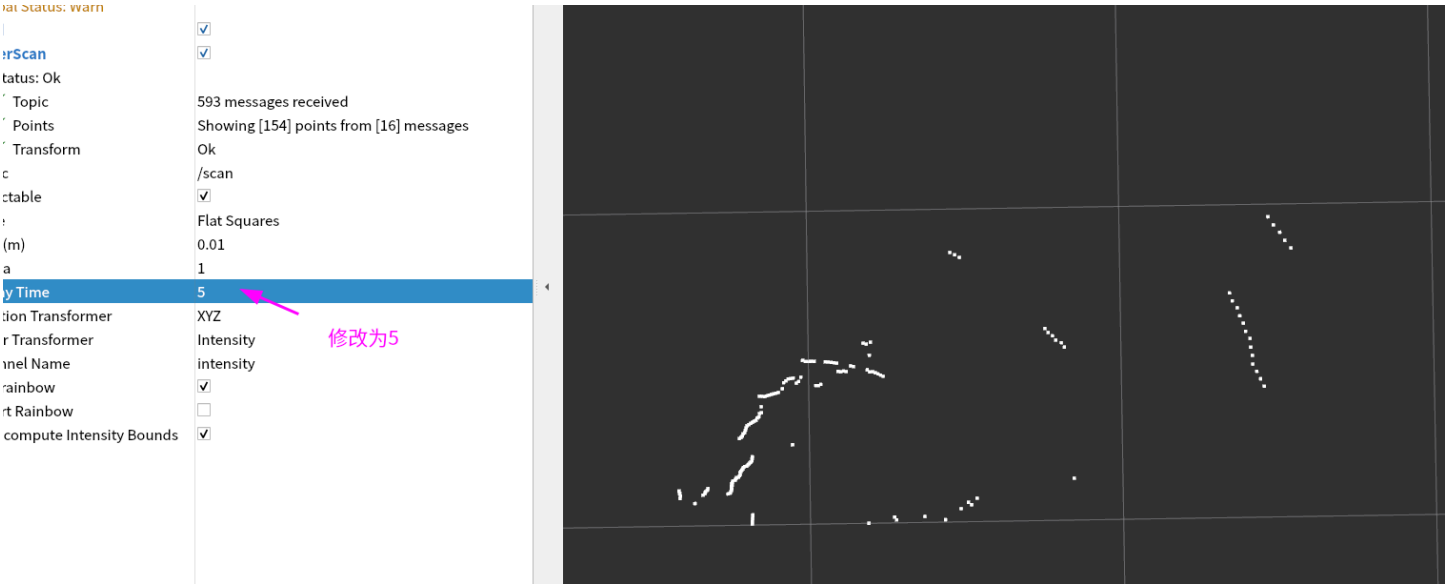
```
ros2 node list
ros2 topic list
ros2 topic echo /scan
```

```
fishros@fishros-MS-7D42:~$ ros2 node list
/example20_simple_laser
fishros@fishros-MS-7D42:~$ ros2 topic list
/parameter_events
/rosout
/scan
fishros@fishros-MS-7D42:~$ ros2 topic echo /scan
header:
  stamp:
    sec: 1674476012
    nanosec: 405
  frame_id: laser
angle_min: 0.0
angle_max: 3.1415927410125732
angle_increment: 0.01745329238474369
time_increment: 0.02800000086426735
scan_time: 0.2800000011920929
range_min: 0.05000000074505806
range_max: 5.0
ranges:
- 0.3641379177570343
- 0.3639655113220215
- 0.3641379177570343
- 0.36465516686439514
```

接着打开终端，输入rviz2打开rviz



修改配置，显示过去5s数据



四、总结

本节我们成功实现了使用超声波和舵机模拟雷达数据，并将其合成scan发布到电脑上使用rviz2进行可视化。至此我们完成了ROS2硬件控制的所有课程。下面迎接你的将是移动机器人和机械臂开发课程，请做好准备，继续出发。

第 16 章 FishBot控制系统搭建

你好，我是爱吃鱼香肉丝的小鱼。本章我们将学习如何从零开始搭建移动机器人控制系统，同时结合下面的章节一起完成移动机器人建图导航平台开发。



1.章节介绍

本章主要从移动机器人底盘角度入手，带你了解一个真实机器人的结构，并在主控板上完成电机驱动，两轮差速运动学正逆解等代码，最终通过MicroROS将里程计发送给上位机，并订阅速度控制指令来控制电机运动。

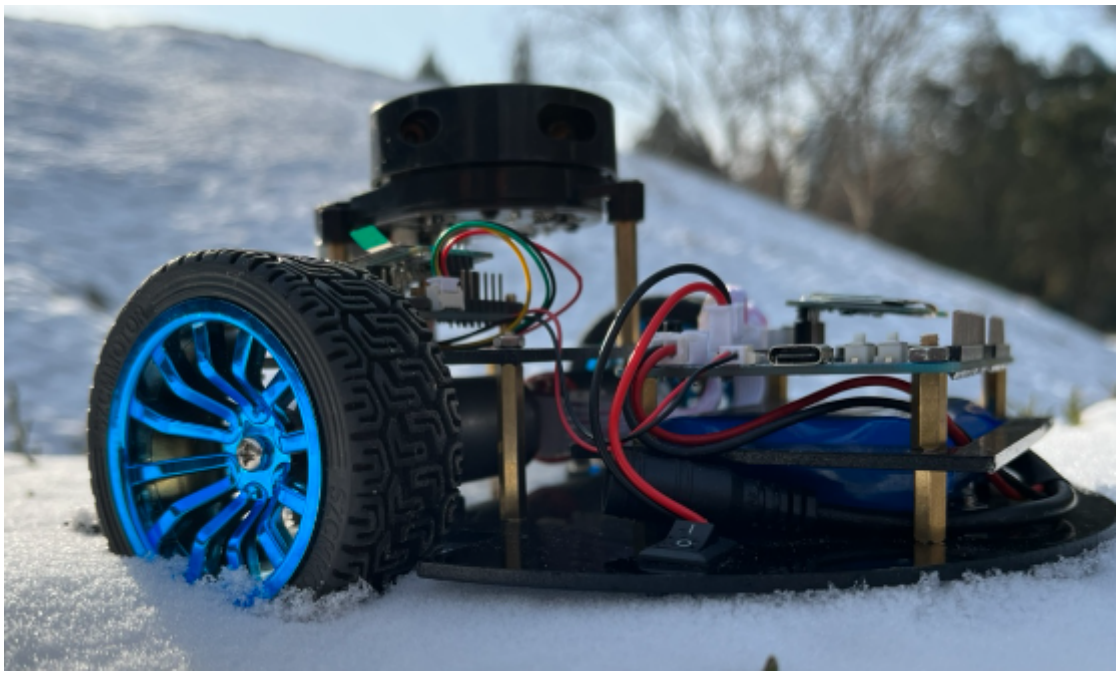
本章章节目录如下：

- 1.移动机器人底盘结构介绍
- 2.让电机转起来-使用开源库驱动电机
- 3.做个遥控车-订阅ROS2 Twist
- 4.速度测量-学会使用编码器
- 5.控制电机速度-PID控制器实现
- 6.控制机器人速度-运动学逆解
- 7.里程计计算-运动学正解与积分
- 8.可视化里程计-发布Odometry

2.FishBot开发平台

本课程的开发平台采用的由小鱼出品的是FishBot基础版，该平台性价比高，资料丰富，非常适合作为机器人学习使用。

需要的小伙伴可以在鱼香小铺购买，对于学生党可享受499优惠：https://item.taobao.com/item.htm?id=696573635888

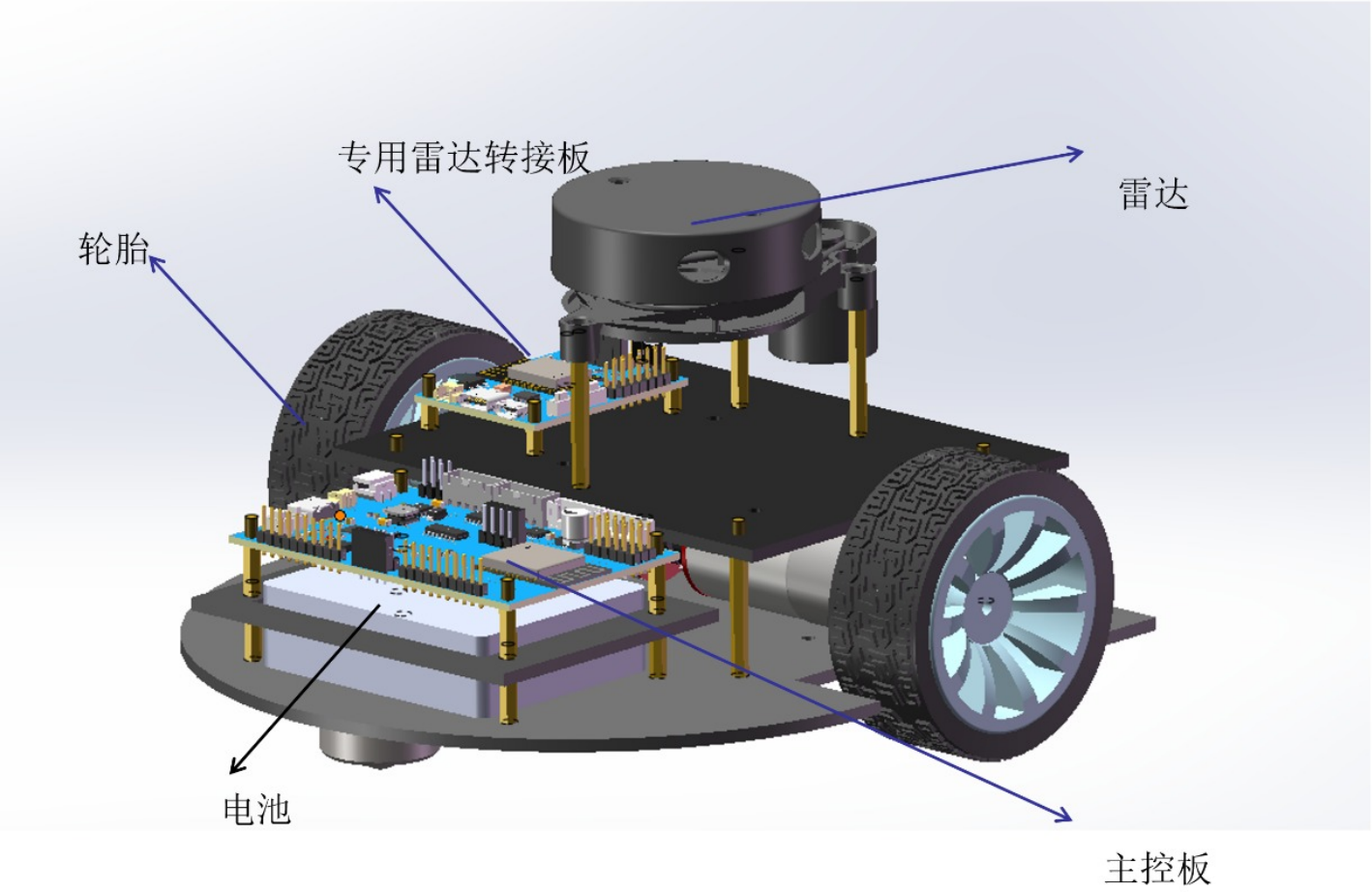


1.移动机器人底盘结构介绍

你好，我是小鱼。本节我们对移动机器人底盘结构进行简单的介绍，并着重介绍FishBot基础版的组成结构。

对于一个移动底盘来说，所要提供的核心功能有两个-感知和执行能力，我们分别进行介绍。

FishBot基础版-结构图



一、感知-传感器

所谓感知即通过给类传感器获取环境信息的能力。在移动机器人中，我们常用的传感器有

- 距离传感器——雷达、超声波
- 轮子速度传感器——编码器
- 惯性测量单元——IMU传感器
- 图像传感器——单目、双目、深度摄像头

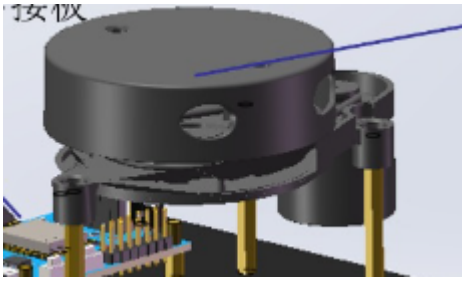
除了上面几类之外，你也可以根据需求挂载其他你需要的传感器，比如温湿度测量传感器等。

对于FishBot基础版来说，搭载的传感器有雷达、超声波、编码器和IMU四种。

在上一篇中我们已经使用了超声波和IMU，所以本节我们对雷达和编码器这两种传感器进行简单介绍。

1.1 雷达

我们采用的激光雷达是单线旋转式三角测距激光雷达。



1.1.1 雷达测距原理介绍

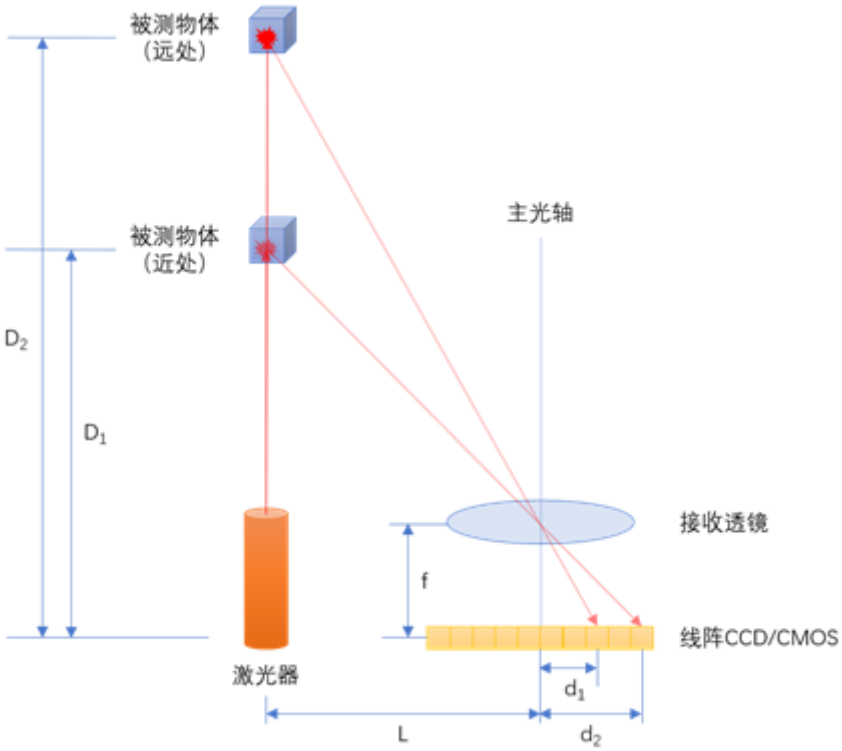
在雷达的头部分别有一个激光发射头和线性的CCD接收头。

小鱼特意拆了个雷达，放张图给大家看看



右边黑色的是发射头，左边是CCD接收头，发射头发射出的光属于波长大约在1000nm左右的红外光，肉眼是不可见的。

FishBot所采用的这种激光雷达的测距原理是三角测距法

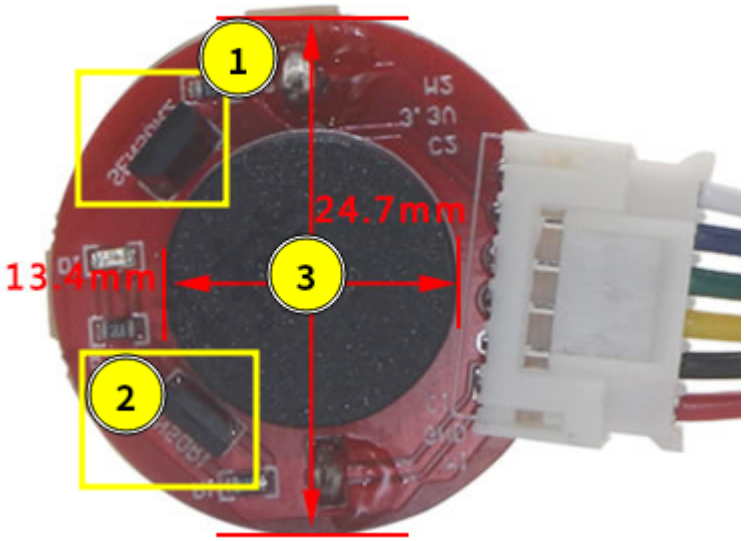


从上图可以看出，当我们已知L和d1和f的情况下，我们可以得到D1的值，即激光雷达到被测物体的距离 $$ D_1/(L+d_1)=f/d_1 \ D_1 = (f*(L+d_1))/d_1 =fL/d_1+f $$ 激光头通过不断旋转，这样就可以测量出360度的深度信息。

1.2 编码器

在移动机器人中我们需要实时的获取到机器人各个轮子的转速，通过转速根据机器人的运动学模型将轮子的速度转换成机器人的速度，通过对速度进行积分（速度*距离）得到机器人行走的距离。

我们如何对轮子速度的测量所使用的传感器就是编码器，在FishBot上，我们采用的是AB电磁编码器。



电磁编码器是由1和2两个霍尔传感器和圆形磁铁3共同组成的，该磁铁的磁性是间隔分布的，磁铁固定在电机的转子上，当电机转动时，带动磁铁转动，此时用于检测磁性的霍尔传感器就会检测到磁性的变化，从而就可以测量出电机在某段时间内转了多少圈即电机的转速。

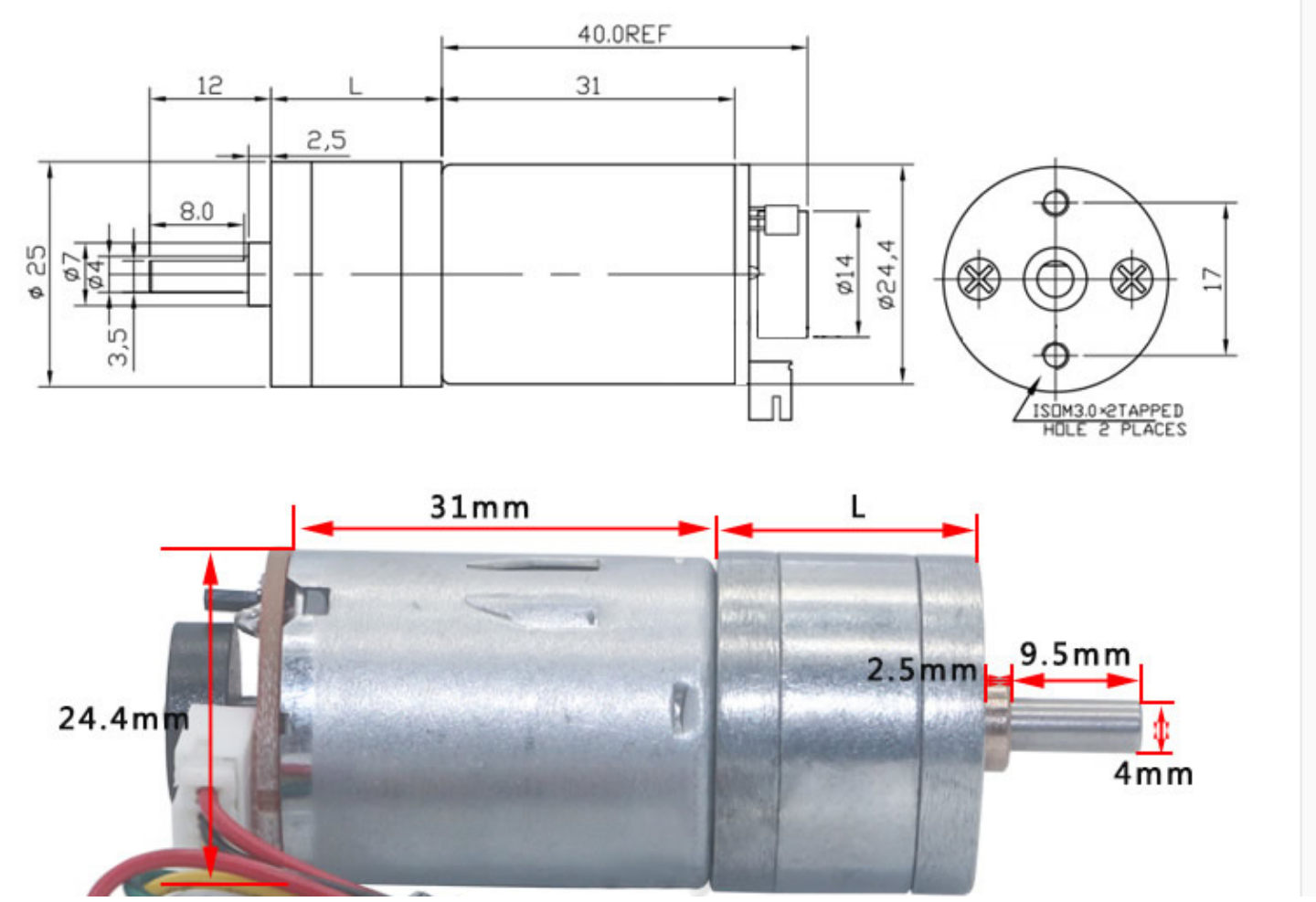
二、执行器

所谓执行器就是负责动的部件，在移动机器人上，最重要的一个执行器就是电机了，除此之外我们也可以把OLED这种显示设备看作一个可以操作的执行器。

电机有很多分类，大类可以按照有刷无刷、直流交流来分类，具体就不展开了，我们下面主要介绍FishBot所采用的电机。

2.1 电机

我们FishBot上采用的是一个额定电压12V的370减速电机，额定转速为130转/分、额定电流0.5A，转矩600克力厘米。



电机相信你很熟悉，那什么是减速电机呢？减速电机指的是带减速器的电机。

电机一般由定子和转子组成的，一般转速都比较快，但输出的力矩比较小，所以我们会给电机配备减速器，让转速降低，提高力矩。

三、其他配件

除了执行器和传感器外，还有一些必要配件

- 电池，提供电力支持
- 稳压模块，提供电压转换，FishBot以将其集成到主控板上
- 万向轮等必要支撑结构

四、总结

本节我们主要介绍FishBot的硬件组成，并对其中新出现的传感器和执行器原理进行介绍，下一节我们开始正式编写代码，先尝试让电机转起来。

2.从H桥说起-电机驱动原理介绍

你好，我是小鱼，本节我们尝试通过代码让机器人上的电机动起来。

一、电机驱动原理介绍

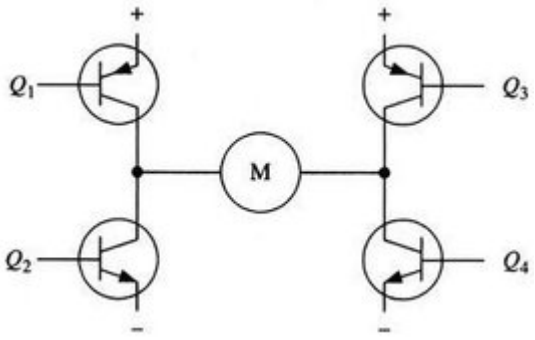
正式编写代码前，我们先了解下电机驱动的原理，以便于我们了解我们如何才能通过代码控制电机的转速和正反转。

1.1 H桥电路

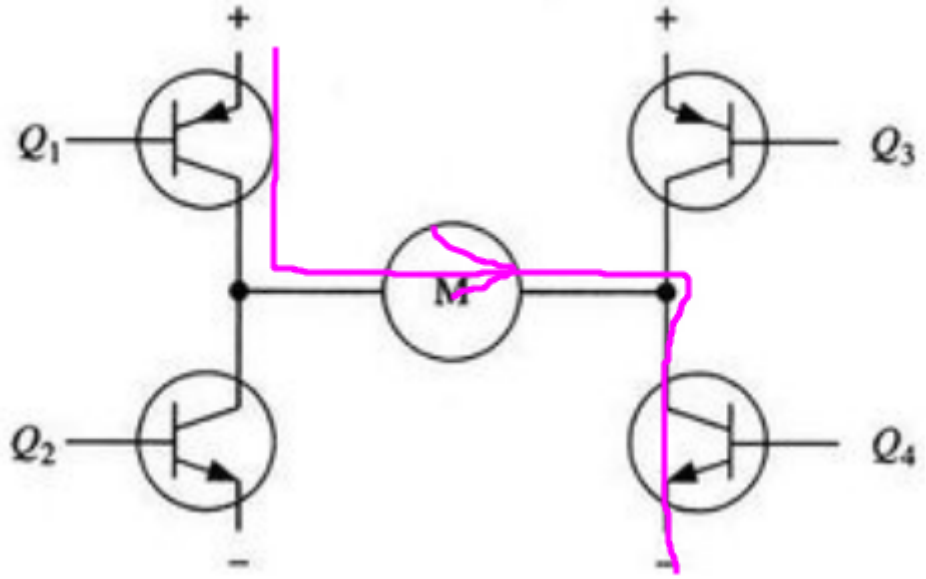
让电机动起来只需要通电就行，比如我们用的额定电压为12V 130RPM的电机，当给到12V的电压时可以达到额定转速130转/分，但如果我们给8V的电压，此时电机依然可以转动，但转速就相对较低。

只是转起来还不行，要想控制机器人前进后退，左转右转，我们还需要控制轮子的转的方向，想要控制转速我们则需要控制给到电机的电压。

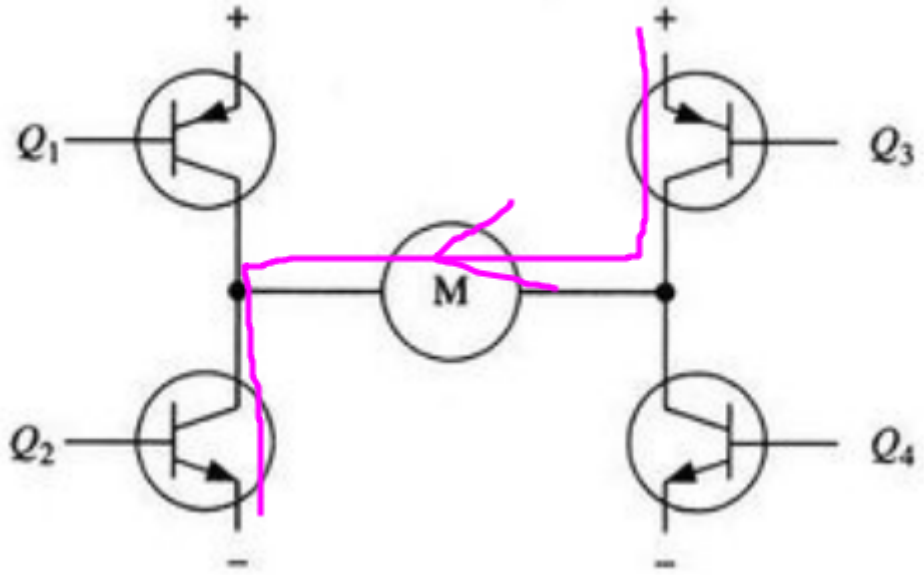
H桥（H-Bridge）电路可以帮我们实现上面两个功能，之所以叫H桥，因为这个电路长的像H。通过该电路可以将电源电压正向或反向地施加到电动机上，从而实现正向、反向转动和制动等功能。



该电路由四个独立开关管（MOSFET）组成，在H桥电路中，两个开关管组成一对，如上图所示。当我们接通Q1和Q4，关闭Q3和Q2时，电源电压正向施加到电动机上，使其正向转动。



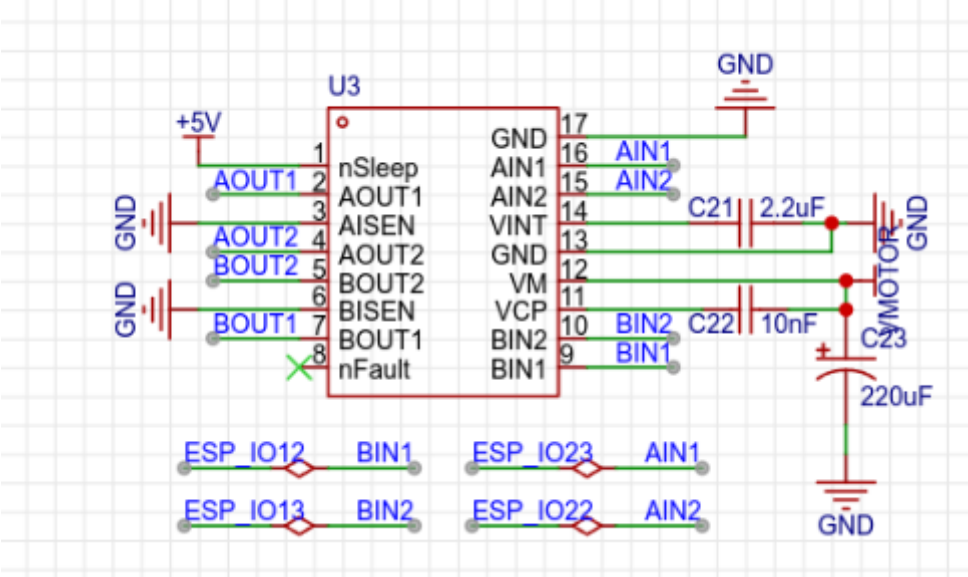
我们关闭Q1和Q4，接通Q2和Q3时，电源电压反向施加到电动机上，使其反向转动。



在制动时，同时关闭四个开关管，这样电动机会在短时间内停止转动。另外，通过之前介绍的在PWM（脉宽调制）控制下，可以通过改变开关管的开关时间比例，就可以实现电动机的速度控制。

1.2 原理图

看完H桥电路原理，我们来讲讲在FishBot主控板上如何搭载的该电路的。在开发板上我们采用了一款H桥电路芯片DRV8833来实现电机的驱动。



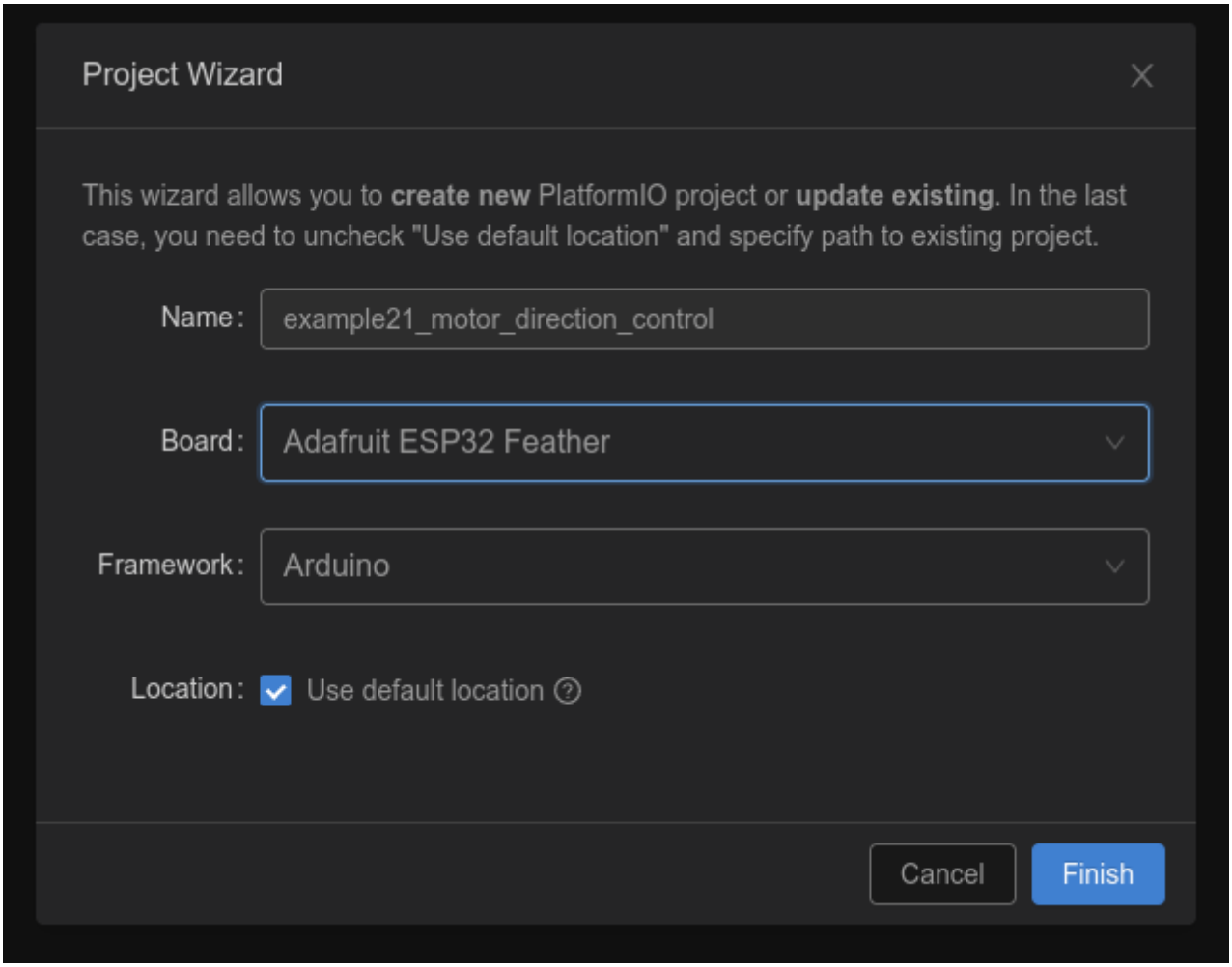
该电路原理图如上图所示，我们通过AIN1（IO23）的高低电平控制H桥中Q1和Q4的开关，通过AIN2（IO22）控制Q2和Q3的开关，所以我们在程序中通过控制AIN1和AIN2的高低电平变化就可以完成对电机正反转的控制。

3.电机控制之正反转实验

你好，我是小鱼。前面说到通过控制对应的IO电平变换即可完成对电机正反转的控制，本节我们尝试编写代码，进行实验。

一、新建工程

新建`example21_motor_direction_control`



二、编写代码

根据第前面硬件控制章节学到的内容可知，控制IO电平只需要使用`pinMode`和`digitalWrite`相关函数即可。

```cpp
/**
 * @file main.cpp
```

```
 * @author fishros@foxmail.com
 * @brief 电机正反转控制
 * @version 0.1
 * @date 2022-12-19
 *
 * @copyright Copyright (c) 2022
 *
 */

#include <Arduino.h>

#define AIN1 23 // 电机驱动模块AIN1引脚
#define AIN2 22 // 电机驱动模块AIN2引脚
#define KEY 0   // 按键引脚

int motorStatus = 0; // 电机状态变量，0-3循环变化

void setup()
{
  Serial.begin(115200);    // 初始化串口通信
  pinMode(KEY, INPUT);     // 设置按键引脚为输入模式
  pinMode(AIN1, OUTPUT);   // 设置AIN1引脚为输出模式
  pinMode(AIN2, OUTPUT);   // 设置AIN2引脚为输出模式
}

void loop()
{
  if (digitalRead(KEY) == LOW) // 检测按键是否按下
  {
    delay(50);                 // 延迟50ms，以防止误触
    if (digitalRead(KEY) == LOW)
    {
      while (digitalRead(0) == LOW) // 等待按键松开，避免连续按下
        ;
      motorStatus++;                // 切换电机状态
      motorStatus = motorStatus % 4; // 保持该变量值在0-4之间
    }
  }

  // 根据电机状态切换IO电平
  switch (motorStatus)
  {
```

```
      case 0:
        Serial.println("AIN1: HIGH, AIN2: LOW"); // 调试信息：AIN1为高电平，AIN2为低电平
        digitalWrite(AIN1, HIGH);
        digitalWrite(AIN2, LOW);
        break;
      case 1:
        Serial.println("AIN1: LOW, AIN2: HIGH"); // 调试信息：AIN1为低电平，AIN2为高电平
        digitalWrite(AIN1, LOW);
        digitalWrite(AIN2, HIGH);
        break;
      case 2:
        Serial.println("AIN1: HIGH, AIN2: HIGH"); // 调试信息：AIN1和AIN2均为高电平
        digitalWrite(AIN1, HIGH);
        digitalWrite(AIN2, HIGH);
        break;
      case 3:
        Serial.println("AIN1: LOW, AIN2: LOW"); // 调试信息：AIN1和AIN2均为低电平
        digitalWrite(AIN1, LOW);
        digitalWrite(AIN2, LOW);
        break;
      default:
        break;
    }
  }
```

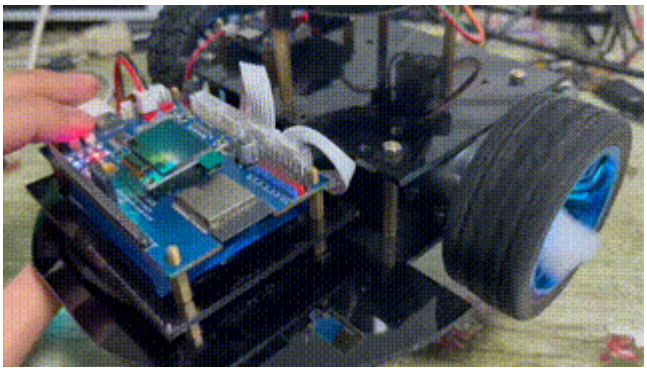
代码解释

在setup函数中，通过pinMode函数将KEY、AIN1和AIN2引脚设置为对应的输入输出模式。

在loop函数中，首先通过digitalRead函数检测按键是否按下，如果检测到按键按下，则会将电机状态变量motorStatus加1，然后通过switch语句根据电机状态改变AIN1和AIN2引脚的电平状态。同时，还会通过Serial.println函数将AIN1和AIN2引脚的电平状态输出到串口，方便调试。

三、测试

将代码下载到FishBot主控板上，点击按键，查看轮子转动效果。



你可以发现，当AIN1和AIN2同时为高电平或者同时为低时，电机并不会转动，符合上一节中对H桥的介绍。

四、总结

本节我们成功通过代码验证了上一节电机控制理论，下一节我们尝试通过PWM的占空比控制电机的转速。

4.电机控制之速度控制实验

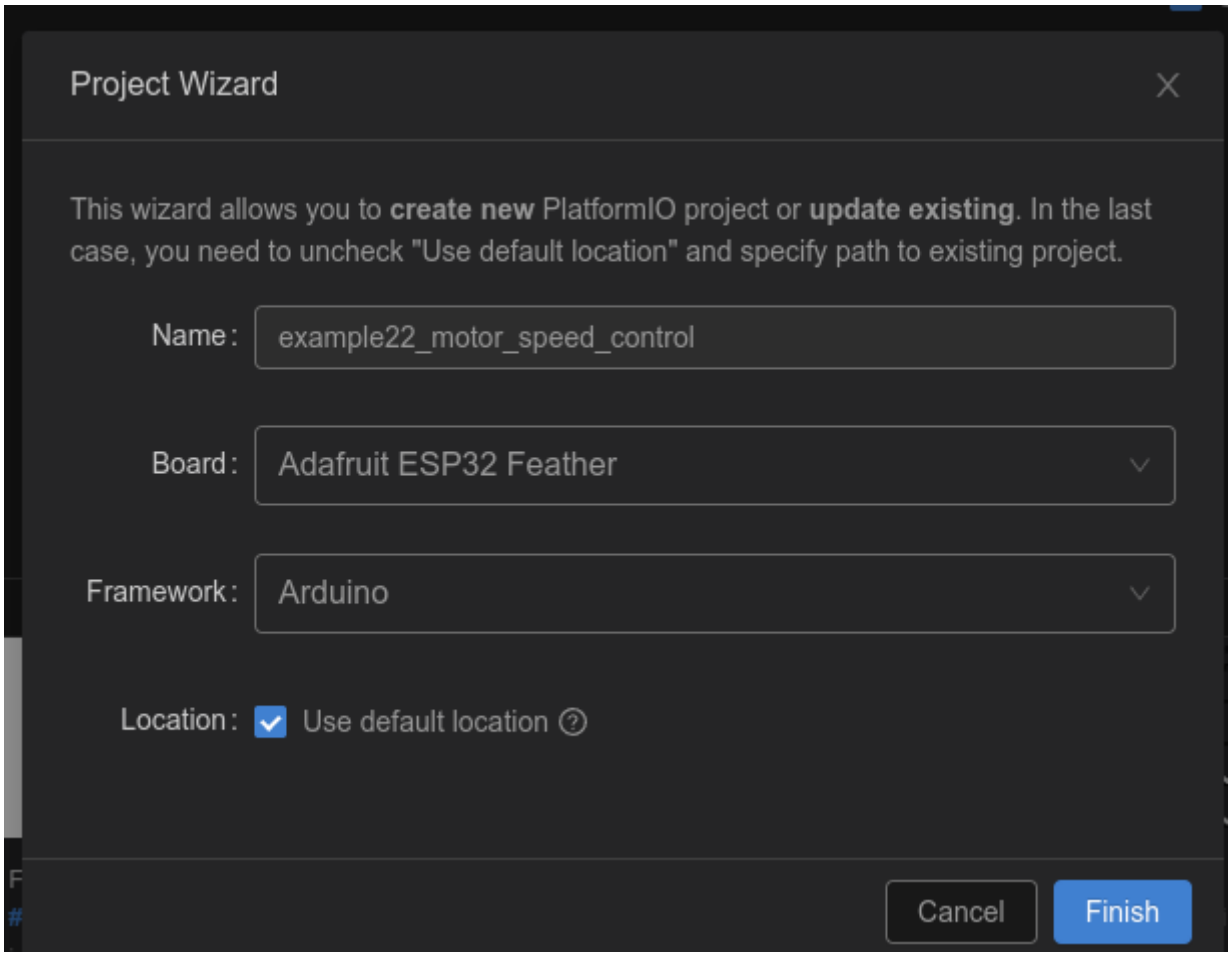
你好，我是小鱼。前面说到通过控制对应的IO上的PWM占空比即可完成对电机速度的控制。

关于PWM的介绍在ROS2硬件控制篇的舵机控制中已经介绍过了，所以我们知道通过改变PWM的占空比可以实现对输出电压的大小调节。占空比越大，输出电压越高；占空比越小，输出电压越低。

接着我们通过一个实验来验证下

一、新建工程

新建`example22_motor_speed_control`



二、编写代码

程序的基本思路是，通过检测按键输入来改变占空比的大小，从而控制电机的转速。按下按键后，每次增加0.1的占空比，当占空比达到1.0时，重新从0开始计数。在loop函数中，通过控制AIN1引脚的高低电平来实现PWM信号的输出，从而控制电机的速度。

c++

```
/**
 * @file main.cpp
 * @author fishros@foxmail.com
 * @brief 电机速度控制
 * @version 0.1
 * @date 2022-12-19
 *
 * @copyright Copyright (c) 2022
 *
 */

#include <Arduino.h>

#define AIN1 23  // 电机驱动模块AIN1引脚
#define AIN2 22  // 电机驱动模块AIN2引脚
#define KEY 0    // 按键引脚
#define CYCLE 10 // 定义PWM信号的周期长度，单位为ms

float duty = 0.0; // 定义占空比变量，并初始化为0.0

void setup()
{
  Serial.begin(115200);   // 初始化串口通信
  pinMode(KEY, INPUT);    // 设置按键引脚为输入模式
  pinMode(AIN1, OUTPUT);  // 设置AIN1引脚为输出模式
  pinMode(AIN2, OUTPUT);  // 设置AIN2引脚为输出模式
  digitalWrite(AIN2, LOW);// 设置AIN2引脚为低电平，控制电机转向
}

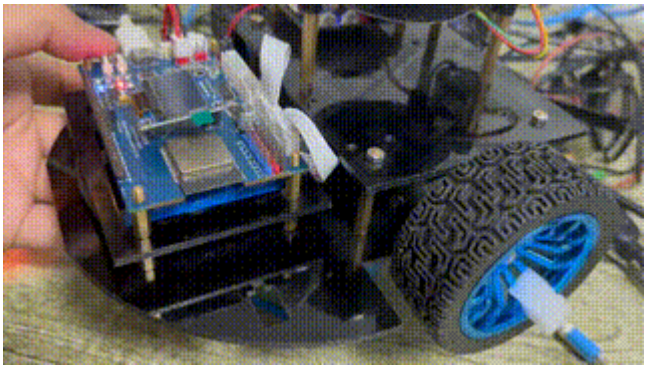
void loop()
{
  // 检测按键是否按下
  if (digitalRead(KEY) == LOW)
  {
    delay(50); // 延迟50ms，以防止误触
    // 确认按键已经按下
    if (digitalRead(KEY) == LOW)
    {
      // 等待按键松开，避免连续按下
      while (digitalRead(0) == LOW)
        ;
```

```
      // 每次增加0.1的占空比，当占空比达到1.0时，重新从0开始计数
      duty = duty + 0.1;
      if (duty > 1.0)
        duty = 0;
    }
  }

  // 输出PWM信号控制电机转速
  digitalWrite(AIN1, HIGH);      // 将AIN1引脚设置为高电平
  delay(CYCLE * duty);            // 延迟一段时间，时间长度由占空比决定
  digitalWrite(AIN1, LOW);       // 将AIN1引脚设置为低电平
  delay(CYCLE * (1 - duty));      // 延迟一段时间，时间长度由占空比决定
}
```

三、测试

将代码下载到主控板上，点击BOOT按键，观察电机转速。



四、总结

本节我们通过简单的一个实验学习了如何通过PWM调节电机的PWM，但有一点需要注意，程序中使用了delay函数来控制PWM信号的占空比，这种方法在简单的应用场景下是可行的，但是在需要更高精度的控制场景下可能会产生问题。为了实现更高精度的PWM控制，我们可以采用ESP32的电机PWM控制单元，下一节我们就尝试使用这一开源库实现更精细化的控制。

ROS2 代码模板

增加该部分内容主要是为了大家写代码的时候方便复制粘贴，本工程主要代码来自国外大佬开源仓库（mikeferguson/ros2_cookbook），小鱼根据日常使用需求和最新的ROS2特性进行了增改。

该部分工程分为以下几个部分：

- 客户端库
  - rclcpp API
    - 节点和组件
    - 参数
    - 点云处理
    - 时间
    - TF2
    - 解决方案
  - rclpy API
    - 节点
    - 参数
    - 时间
    - TF2
- Launch
- 通信相关 (DDS & CycloneDDS)
- 命令行工具
  - `ros2 run <pkg> <node>`
  - `ros2 node list`
  - `ros2 topic list`
  - `ros2 topic info <topic_name> --verbose` Qos细节查看.
  - `ros2 param list`
  - colcon 构建工具.
  - `ros2 doctor --report` gives tons of information.
- CMake
- URDF
  - Gazebo插件配置
  - Xacro
  - URDF
- 功能包
  - 设置 bloom/git 总是使用 ssh
  - `rosdep install --from-paths src --ignore-src --rosdistro=foxy -y`
- Package Documentation

- 状态页面
  - Foxy Debian 构建状态
  - Rolling Debian 构建状态
  - 对比 Foxy/Rolling
  - 对比 Foxy/Humble

Rclcpp：节点和组件

创建组件

```cpp
#include <rclcpp/rclcpp.hpp>

namespace my_pkg
{

class MyComponent : public rclcpp::Node
{
public:
  MyComponent(const rclcpp::NodeOptions& options)
  : rclcpp::Node("node_name", options)
  {
    // Note: you cannot use shared_from_this()
    //       here because the node is not fully
    //       initialized.
  }
};

}  // namespace my_pkg

#include "rclcpp_components/register_node_macro.hpp"
RCLCPP_COMPONENTS_REGISTER_NODE(my_pkg::MyComponent)
```

CMakeLists.txt 中的：

```cmake
add_library(my_component SHARED
  src/my_component.cpp
)
ament_target_dependencies(my_component
  rclcpp
  rclcpp_components
)

# Also add a node executable which simply loads the component
```

```
rclcpp_components_register_node(my_component
  PLUGIN "my_pkg::MyComponent"
  EXECUTABLE my_node
)
```

Executors

要在线程中运行执行器，请执行以下操作：

```cpp
#include <thread>
#include <rclcpp/rclcpp.hpp>
#include <rclcpp/executors/single_threaded_executor.hpp>

rclcpp::executors::SingleThreadedExecutor executor;

// Node is rclcpp::Node::SharedPtr instance
executor.add_node(node);
std::thread executor_thread(
  std::bind(&rclcpp::executors::SingleThreadedExecutor::spin,
            &executor));
```

Rclcpp：参数

需要声明参数。同时，如果你不打算稍后再次更新值，则可以获得该值：

```cpp
// node is an instance of rclcpp::Node
// 42 is a great default for a parameter
int param = node.declare_parameter<int>("my_param_name", 42);
```

要获取值，请执行以下操作：

```cpp
int param;
node.get_parameter("my_param_name", param);
```

动态参数

在 ROS2 中，所有参数都可以通过 ROS2 服务动态更新(不需要像动态重新配置那样定义重复内容)。

下面的例子适用于 eloquent 或更高版本(较早的 ROS2 版本只支持单个回调，并且有一个略有不同的 API)。有关有效类型的信息，请参阅的文档。

```cpp
#include <vector>
#include <rclcpp/rclcpp.hpp>

class MyNode : public rclcpp::Node
{
public:
  MyNode()
  {
    // Declare parameters first

    // Then create callback
    param_cb_ = this->add_on_set_parameters_callback(
      std::bind(&MyNode::updateCallback, this, std::placeholders::_1));
  }
```

```
private:
  // This will get called whenever a parameter gets updated
  rcl_interfaces::msg::SetParametersResult updateCallback(
    const std::vector<rclcpp::Parameter> & parameters)
  {
    rcl_interfaces::msg::SetParametersResult result;
    result.successful = true;

    for (const rclcpp::Parameter & param : parameters)
    {
      if (param.get_name() == "my_param_name")
      {
        if (param.get_type() != rclcpp::ParameterType::PARAMETER_STRING)
        {
          result.successful = false;
          result.reason = "my_param_name must be a string";
          break;
        }

        // Optionally do something with parameter
      }
    }

    return result;
  }

  // Need to hold a pointer to the callback
  rclcpp::node_interfaces::OnSetParametersCallbackHandle::SharedPtr param_cb_;
};
```

Rclcpp：TF2

TF2 库提供了对转换的轻松访问。以下所有示例都需要对 _tf2_ros_ 的依赖关系。

发布TF

```cpp
#include <tf2_ros/transform_broadcaster.h>
std::unique_ptr<tf2_ros::TransformBroadcaster> broadcaster;

broadcaster = std::make_unique<tf2_ros::TransformBroadcaster>(nodePtr);

geometry_msgs::msg::TransformStamped transform;
transform.header.stamp = node->now();
transform.header.frame_id = "odom";
transform.child_frame_id = "base_link";

// Fill in transform.transform.translation
// Fill in transform.transform.rotation

broadcaster->sendTransform(transform);
```

监听TF

```cpp
#include "tf2_ros/transform_listener.h"

std::shared_ptr<tf2_ros::Buffer> tf_buffer;
std::shared_ptr<tf2_ros::TransformListener> tf_listener;

rclcpp::Node node("name_of_node");

tf_buffer.reset(new tf2_ros::Buffer(node.get_clock()));
tf_listener.reset(new tf2_ros::TransformListener(*tf_buffer_));
```

TF变换

TF2 可以通过提供实现的包进行扩展。GEOMETRY_msgs 程序包为 msgs_ 提供这些。下面的示例使用 msgs：：msg：：PointStamed_，但这应该适用于 msgs_ 中的任何数据类型：

```cpp
#include <tf2_geometry_msgs/tf2_geometry_msgs.h>

geometry_msg::msgs::PointStamped in, out;
in.header.frame_id = "source_frame";

try
{
  tf_buffer->transform(in, out, "target_frame");
}
catch (const tf2::TransformException& ex)
{
  RCLCPP_ERROR(rclcpp::get_logger("logger_name"), "Could not transform point.");
}
```

transform 函数还可以接受超时。然后它将等待转换可用的时间达到这个时间量:

```cpp
tf_buffer->transform(in, out, "target_frame", tf2::durationFromSec(1.0));
```

获取最新TF

常见的工作方式是获得“最新”转换。在 ros2中，这可以使用 tf2::TimePointZero 来实现，但是需要使用 lookupTransform 然后调用 doTransform (基本上就是在内部进行转换) :

```cpp
geometry_msgs::msg::PointStamped in, out;

geometry_msgs::msg::TransformStamped transform =
   tf_buffer->lookupTransform("target_frame",
                              in.header.frame_id,
                              tf2::TimePointZero);
```

```
tf2::doTransform(in, out, transform);
```

rclcpp: Time

rclcpp::Time 和 *rclcpp::Duration* 和ROS1中的用法偏差较大，但与std::chrono 的关系更为密切。ROS Discourse 可以看到与其有关的比较深入的讨论。

在移植某些ros1库时，时间戳可能会被大量用作浮点秒。从 rclcpp 获取浮点秒 *rclcpp::Time*:

```cpp
// node is instance of rclcpp::Node
rclcpp::Time t = node.now();
double seconds = t.seconds();
```

没有用于从浮点表示的秒开始的时间的构造函数，因此你首先需要转换为纳秒：

```cpp
rclcpp::Time t(static_cast<uin64_t>(seconds * 1e9));
```

确实具有双向功能：

```cpp
rclcpp::Duration d = rclcpp::Duration::from_seconds(1.0);
double seconds = d.seconds();
```

rclcpp: Point Clouds

`sensor_msgs/PointCloud2` 是一种非常常见的 ROS 消息类型，用于处理 ROS 中的感知数据。这也是实际要解释的最复杂的信息之一。

消息的复杂性源于它在单个巨型数据存储中包含任意字段这一事实。这允许 PointCloud2 消息与任何类型的云(例如，仅 XYZ 点、XYZRGB，甚至 XYZI)一起工作，但在访问云中的数据时增加了一些复杂性。

在 ROS1 中，有一个更简单的 PointCloud 消息，但这已经被删除，并将在 ROS2-G 中删除。

使用 PointCloud2 迭代器

sensor_msgs 包提供了一个 C++ PointCloud2Iterator，可用于创建、修改和访问 PointCloud2 消息。

要创建新消息：

```cpp
#include "sensor_msgs/msg/point_cloud2.hpp"
#include "sensor_msgs/point_cloud2_iterator.hpp"

sensor_msgs::msg::PointCloud2 msg;

// Fill in the size of the cloud
msg.height = 480;
msg.width = 640;

// Create the modifier to setup the fields and memory
sensor_msgs::PointCloud2Modifier mod(msg);

// Set the fields that our cloud will have
mod.setPointCloud2FieldsByString(2, "xyz", "rgb");

// Set up memory for our points
mod.resize(msg.height * msg.width);

// Now create iterators for fields
sensor_msgs::PointCloud2Iterator<float> iter_x(cloud_msg, "x");
sensor_msgs::PointCloud2Iterator<float> iter_y(cloud_msg, "y");
sensor_msgs::PointCloud2Iterator<float> iter_z(cloud_msg, "z");
sensor_msgs::PointCloud2Iterator<uint8_t> iter_r(cloud_msg, "r");
sensor_msgs::PointCloud2Iterator<uint8_t> iter_g(cloud_msg, "g");
```

```
sensor_msgs::PointCloud2Iterator<uint8_t> iter_b(cloud_msg, "b");

for (; iter_x != iter_x.end(); ++iter_x, ++iter_y, ++iter_z, ++iter_r, ++iter_g,
++iter_b)
{
  *iter_x = 0.0;
  *iter_y = 0.0;
  *iter_z = 0.0;
  *iter_r = 0;
  *iter_g = 255;
  *iter_b = 0;
}
```

使用 PCL

对于许多操作，你可能希望转换为 pcl::PointCloud 以便使用的扩展 API Point Cloud Library。

在 ROS1，pcl_ros 包允许你编写一个订阅者，它的回调直接接受 pcl::PointCloud，但是这个包还没有被移植到 ROS2. 无论如何，使用 pcl_conversions 包自己进行转换是非常简单的：

```cpp
#include "pcl_conversions/pcl_conversions.h"

void cloud_callback(const sensor_msgs::msg::PointCloud2::SharedPtr msg)
{
  // PCL still uses boost::shared_ptr internally
  pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud =
    boost::make_shared<pcl::PointCloud<pcl::PointXYZRGB>>();

  // This will convert the message into a pcl::PointCloud
  pcl::fromROSMsg(*msg, *cloud);
}
```

反之，你也可以反方向转换：

```cpp
#include "pcl_conversions/pcl_conversions.h"

pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud;
```

```
sensor_msgs::msg::PointCloud2 msg;

pcl::toROSMsg(*cloud, msg);
cloud_publisher->publish(msg);
```

rclcpp: Workarounds

懒订阅

ROS2 还没有订户连接回叫。此代码创建一个函数，定期调用该函数来检查我们是否需要启动或停止订阅者：

```cpp
#include <rclcpp/rclcpp.hpp>
#include <std_msgs/msg/float64.hpp>

class LazyPublisherEx : rclcpp::Node
{
public:
  LazyPublisherEx(const rclcpp::NodeOptions & options) :
    Node("lazy_ex", options)
  {
    // Setup timer
    timer = this->create_wall_timer(
      std::chrono::seconds(1),
      std::bind(&LazyPublisher::periodic, this));
  }

private:
  void periodic()
  {
    if (pub_.get_subscription_count() > 0)
    {
        // We have a subscriber, do any setup required
    }
    else
    {
        // Subscriber has disconnected, do any shutdown
    }
  }

  rclcpp::Publisher<std_msgs::msg::Float64>::SharedPtr pub_;
  rclcpp::TimerBase::SharedPtr timer_;
};
```

在使用图像传输时也可以这样做，你只需要将 *get_subscription_count* 更改为 *getNumSubscribers*:

```cpp
#include <rclcpp/rclcpp.hpp>
#include <image_transport/image_transport.h>

class LazyPublisherEx : rclcpp::Node
{
public:
  LazyPublisherEx(const rclcpp::NodeOptions & options) :
    Node("lazy_ex", options)
  {
    // Setup timer
    timer = this->create_wall_timer(
      std::chrono::seconds(1),
      std::bind(&LazyPublisher::periodic, this));
  }

private:
  void periodic()
  {
    if (pub_.getNumSubscribers() > 0)
    {
        // We have a subscriber, do any setup required
    }
    else
    {
        // Subscriber has disconnected, do any shutdown
    }
  }

  image_transport::CameraPublisher pub_;
  rclcpp::TimerBase::SharedPtr timer_;
};
```

rclpy：节点基础知识

大多数节点都有发布者和订阅者，它们都是通过调用实例的函数创建的：

```python
import rclpy
from rclpy.node import Node

from std_msgs.msg import String

class MyNode(Node):

    def __init__(self):
        super().__init__('my_node_name')

        self.publisher = self.create_publisher(String, 'output_topic', 10)
        self.subscription = self.create_subscription(
            String,
            'input_topic',
            self.callback,
            10)

    def callback(self, msg):
        self.get_logger().info("Recieved: %s" % msg.data)
        self.publisher.publish(msg)

if __name__ == "__main__":
    rclpy.init()
    my_node = MyNode()
    rclpy.spin(my_node)
    my_node.destroy_node()  # cleans up pub-subs, etc
    rclpy.shutdown()
```

关闭Handle

ROS1 有rospy.on_shutdown() - 但是 不等于ROS2也有，不过，这真的不需要，因为我们手动关闭了一些东西，而不是像 rospy 那样使用了许多全局变量:

```python
try:
    rclpy.spin(my_node)
except KeyboardInterrupt:
    pass
finally:
    my_node.on_shutdown()  # do any custom cleanup
    my_node.destroy_node()
    rclpy.shutdown()
```

rclpy：参数

```python
# node is rclpy.node.Node instance
# 42 is a great default for a parameter
node.declare_parameter("my_param_name", 42)

# To get the value:
param = node.get_parameter("my_param_name").value
```

动态参数

在 ROS2 中，所有参数都可以通过 ROS2 服务动态更新(不需要像动态重新配置那样定义重复内容)。

```python
from rcl_interfaces.msg import SetParametersResult

import rclpy
from rclpy.node import Node

class MyNode(Node):

    def __init__(self):
        super().__init__('my_node_name')

        # Declare a parameter
        self.declare_parameter("my_param_name", 42)

        # Then create callback
        self.set_parameters_callback(self.callback)

    def callback(self, parameters):
        result = SetParametersResult(successful=True)

        for p in parameters:
            if p.name == "my_param_name":
                if p.type_ != p.Type.INTEGER:
                    result.successful = False
                    result.reason = 'my_param_name must be an Integer'
```

```
                return result
            if p.value < 20:
                result.successful = False
                result.reason = "my_param_name must be >= 20"
                return result

        # Return success, so updates are seen via get_parameter()
        return result
```

有关调用 SET_PARAMETERS 服务的示例，请参阅 ROS Answers

RCLPY：TF2

TF2 库提供了对转换的轻松访问。以下所有示例都需要对 ros_Package 的依赖关系。

监听转换

```python
import rclpy
from rclpy.node import Node
from tf2_ros.buffer import Buffer
from tf2_ros.transform_listener import TransformListener

class MyNode(Node):
    def __init__(self):
        super().__init__("my_node")

        self.buffer = Buffer()
        self.listener = TransformListener(self.buffer, self)
```

应用变换

TF2 可以通过提供实现的包进行扩展。GEOMETRY_msgs 程序包为 msgs_ 提供这些。下面的示例使用 msgs.msg.PointStamed_，但这应该适用于 msgs_ 中的任何数据类型：

```python
from geometry_msgs.msg import PointStamped
from tf2_ros.buffer import Buffer
from tf2_ros.transform_listener import TransformListener
import tf2_geometry_msgs

# Setup buffer/listener as above

p1 = PointStamped()
p1.header.frame_id = "source_frame"
# fill in p1

p2 = buffer.transform(p1, "target_frame")
```

变换

在 ROS1 中，Tf 包括模块。TF2 没有类似的模块。建议使用 Transforms3d Python 包，可通过 pip 获取：

```
sudo pip3 install transforms3d
```

例如，要旋转点：

```python
import numpy as np
from transforms3d.quaternion import quat2mat

# Create rotation matrix from quaternion
R = quat2mat([w, x, y, z])
# Create a vector to rotate
V = np.array([x, y, z]).reshape((3, 1))
# Rotate the vector
M = np.dot(R, V)

p = PointStamped()
p.point.x = M[0, 0]
p.point.y = M[1, 0]
p.point.z = M[2, 0]
```

rclpy: Time

要获得相当于 rospy.Time.now()的内容，你现在需要一个 ROS2 节点：

```python
import rclpy
from rclpy.node import Node

class MyNode(Node):

    def some_func(self):
        t = self.get_clock().now()
        msg.header.stamp = t.to_msg()
```

从持续时间转换为消息很常见：

```python
import rclpy
from rclpy.duration import Duration

msg.duration = Duration(seconds=1).to_msg()
```

计时器是从节点创建的：

```python
import rclpy
from rclpy.node import Node

class MyNode(Node):

    def __init__(self):
        super().__init__("my_node")

        # Create a timer that fires every quarter second
        timer_period = 0.25
        self.timer = self.create_timer(timer_period, self.callback)
```

```
        def callback(self):
            self.get_logger().info("timer has fired")
```

ROS2中常用的Xacro模板

URDF默认格式是纯文本的，我们并不能在其中加入计算公式和定义，用URDF定义一个机器人模型会导致整个文件非常冗长，使用Xacro工具可以解决这个问题。

Xacro是urdf的定义和生成工具，你按照Xacro提供的方式定义可以复用的模型描述块，之后就可以直接调用这些描述，最后使用xacro指令生成最终的urdf模型了。

1.添加模板

小鱼这里提供了常用的xacro描述定义的代码块，你可以直接引入的你的工程里进行使用。

在你的功能包里新建 `xacro_template.xacro` 文件，复制粘贴下面的内容到其中。

```xml
<?xml version="1.0"?>
<robot xmlns:xacro="http://ros.org/wiki/xacro" xmlns:fishros="http://fishros.com">
  <xacro:macro name="box_inertia" params="m w h d">
    <inertial>
      <origin xyz="0 0 0" rpy="${pi/2} 0 ${pi/2}"/>
      <mass value="${m}"/>
      <inertia ixx="${(m/12) * (h*h + d*d)}" ixy="0.0" ixz="0.0" iyy="${(m/12) *
(w*w + d*d)}" iyz="0.0" izz="${(m/12) * (w*w + h*h)}"/>
    </inertial>
  </xacro:macro>

  <xacro:macro name="cylinder_inertia" params="m r h">
    <inertial>
      <origin xyz="0 0 0" rpy="${pi/2} 0 0" />
      <mass value="${m}"/>
      <inertia ixx="${(m/12) * (3*r*r + h*h)}" ixy = "0" ixz = "0" iyy="${(m/12) *
(3*r*r + h*h)}" iyz = "0" izz="${(m/2) * (r*r)}"/>
    </inertial>
  </xacro:macro>

  <xacro:macro name="sphere_inertia" params="m r">
    <inertial>
      <mass value="${m}"/>
      <inertia ixx="${(2/5) * m * (r*r)}" ixy="0.0" ixz="0.0" iyy="${(2/5) * m *
(r*r)}" iyz="0.0" izz="${(2/5) * m * (r*r)}"/>
```

```
    </inertial>
  </xacro:macro>

  <xacro:macro name="sphere_visual" params="r origin_r origin_p origin_y">
    <visual>
      <origin xyz="0 0 0" rpy="${origin_r} ${origin_p} ${origin_y}"/>
      <geometry>
          <sphere radius="${r}"/>
      </geometry>
      <material name="blue">
        <color rgba="0.0 0.0 0.8 1.0"/>
      </material>
    </visual>
  </xacro:macro>

<xacro:macro name="sphere_collision" params="r origin_r origin_p origin_y">
    <collision>
      <origin xyz="0 0 0" rpy="${origin_r} ${origin_p} ${origin_y}"/>
      <geometry>
          <sphere radius="${r}"/>
      </geometry>
        <material name="green">
            <color rgba="0.0 0.8 0.0 1.0"/>
        </material>
    </collision>
</xacro:macro>


<xacro:macro name="box_visual" params="w d h origin_r origin_p origin_y">
    <visual>
      <origin xyz="0 0 0" rpy="${origin_r} ${origin_p} ${origin_y}"/>
      <geometry>
          <box size="${w} ${d} ${h}" />
      </geometry>
      <material name="blue">
        <color rgba="0.0 0.0 0.8 1.0"/>
      </material>
    </visual>
</xacro:macro>

<xacro:macro name="box_collision" params="w d h origin_r origin_p origin_y">
    <collision>
```

```
      <origin xyz="0 0 0" rpy="${origin_r} ${origin_p} ${origin_y}"/>
      <geometry>
          <box size="${w} ${d} ${h}" />
      </geometry>
        <material name="green">
            <color rgba="0.0 0.8 0.0 1.0"/>
        </material>
    </collision>
</xacro:macro>


<xacro:macro name="cylinder_visual" params="r h origin_r origin_p origin_y">
    <visual>
      <origin xyz="0 0 0" rpy="${origin_r} ${origin_p} ${origin_y}"/>
      <geometry>
         <cylinder length="${h}" radius="${r}"/>
      </geometry>
      <material name="blue">
        <color rgba="0.0 0.0 0.8 1.0"/>
      </material>
    </visual>
</xacro:macro>


<xacro:macro name="cylinder_collision" params="r h origin_r origin_p origin_y">
    <collision>
      <origin xyz="0 0 0" rpy="${origin_r} ${origin_p} ${origin_y}"/>
      <geometry>
         <cylinder length="${h}" radius="${r}"/>
      </geometry>
      <material name="blue">
        <color rgba="0.0 0.0 0.8 1.0"/>
      </material>
    </collision>
</xacro:macro>


</robot>
```

2.使用模板生成URDF

接着你可以新建你的机器人模型描述文件，比如 `fishbot.urdf.xacro` ,之后你就可以在你的描述文件中调用小鱼提供的模板，快速的定义机器人。

比如创建一个正方体的base_link,并导入惯性矩阵。

```xml
<?xml version="1.0"?>
<robot name="fishbot" xmlns:xacro="http://ros.org/wiki/xacro">
  <xacro:include filename="xacro_template.xacro" />

  <link name="base_link">
    <xacro:box_visual w="0.809" d="0.5" h="0.1" origin_r="0" origin_p="0"
origin_y="0"/>
    <xacro:box_collision w="0.809" d="0.5" h="0.1" origin_r="0" origin_p="0"
origin_y="0"/>
    <xacro:box_inertia m="1.0" w="0.809" d="0.5" h="0.1"/>
  </link>

</robot>
```

上面w,d,h,代表长宽高，m代表质量。 `<xacro:include filename="xacro_template.xacro" />` 用于引入小鱼提供的模板。

接着我们就可以通过xacro指令将其变成正常的urdf，打开终端，进入fishbot.urdf.xacro同级目录，输入指令 `xacro fishbot.urdf.xacro -o fishbot.urdf` ,即可生成fishbot.urdf，正常生成后的内容如下。

```xml
<?xml version="1.0" ?>
<!--
=================================================================================
= -->
<!-- |    This document was autogenerated by xacro from fishbot.urdf.xacro
| -->
<!-- |    EDITING THIS FILE BY HAND IS NOT RECOMMENDED
| -->
<!--
=================================================================================
= -->
<robot name="fishbot" xmlns:fishros="http://fishros.com">
```

```
    <link name="base_link">
      <visual>
        <origin rpy="0 0 0" xyz="0 0 0"/>
        <geometry>
          <box size="0.809 0.5 0.1"/>
        </geometry>
        <material name="blue">
          <color rgba="0.0 0.0 0.8 1.0"/>
        </material>
      </visual>
      <collision>
        <origin rpy="0 0 0" xyz="0 0 0"/>
        <geometry>
          <box size="0.809 0.5 0.1"/>
        </geometry>
        <material name="green">
          <color rgba="0.0 0.8 0.0 1.0"/>
        </material>
      </collision>
      <inertial>
        <origin rpy="1.5707963267948966 0 1.5707963267948966" xyz="0 0 0"/>
        <mass value="1.0"/>
        <inertia ixx="0.021666666666666667" ixy="0.0" ixz="0.0"
  iyy="0.07537341666666666" iyz="0.0" izz="0.055373416666666675"/>
      </inertial>
    </link>
  </robot>
```

这就是xacro的神奇之处，将短短的三行定义根据规则生成长长的URDF，关于xacro的详细使用可以参考 http://ros.org/wiki/xacro 。

Gazebo常用插件

1.雷达

详细介绍及文章：[9.5给机器人添加激光传感器]

```xml
<gazebo reference="laser_link">
    <sensor name="laser_sensor" type="ray">
    <always_on>true</always_on>
    <visualize>true</visualize>
    <update_rate>5</update_rate>
    <pose>0 0 0.075 0 0 0</pose>
    <ray>
        <scan>
          <horizontal>
            <samples>360</samples>
            <resolution>1.000000</resolution>
            <min_angle>0.000000</min_angle>
            <max_angle>6.280000</max_angle>
          </horizontal>
        </scan>
        <range>
          <min>0.120000</min>
          <max>3.5</max>
          <resolution>0.015000</resolution>
        </range>
        <noise>
          <type>gaussian</type>
          <mean>0.0</mean>
          <stddev>0.01</stddev>
        </noise>
    </ray>

    <plugin name="laserscan" filename="libgazebo_ros_ray_sensor.so">
      <ros>
        <remapping>~/out:=scan</remapping>
      </ros>
      <output_type>sensor_msgs/LaserScan</output_type>
      <frame_name>laser_link</frame_name>
```

```
        </plugin>
        </sensor>
      </gazebo>
```

2.IMU

详细介绍及文章：[9.4为FishBot添加IMU传感器.md]

```xml
      <gazebo reference="imu_link">
        <sensor name="imu_sensor" type="imu">
        <plugin filename="libgazebo_ros_imu_sensor.so" name="imu_plugin">
            <ros>
              <namespace>/</namespace>
              <remapping>~/out:=imu</remapping>
            </ros>

  <initial_orientation_as_reference>false</initial_orientation_as_reference>
          </plugin>
          <always_on>true</always_on>
          <update_rate>100</update_rate>
          <visualize>true</visualize>
          <imu>
            <angular_velocity>
              <x>
                <noise type="gaussian">
                  <mean>0.0</mean>
                  <stddev>2e-4</stddev>
                  <bias_mean>0.0000075</bias_mean>
                  <bias_stddev>0.0000008</bias_stddev>
                </noise>
              </x>
              <y>
                <noise type="gaussian">
                  <mean>0.0</mean>
                  <stddev>2e-4</stddev>
                  <bias_mean>0.0000075</bias_mean>
                  <bias_stddev>0.0000008</bias_stddev>
                </noise>
              </y>
```

```
            <z>
              <noise type="gaussian">
                <mean>0.0</mean>
                <stddev>2e-4</stddev>
                <bias_mean>0.0000075</bias_mean>
                <bias_stddev>0.0000008</bias_stddev>
              </noise>
            </z>
          </angular_velocity>
          <linear_acceleration>
            <x>
              <noise type="gaussian">
                <mean>0.0</mean>
                <stddev>1.7e-2</stddev>
                <bias_mean>0.1</bias_mean>
                <bias_stddev>0.001</bias_stddev>
              </noise>
            </x>
            <y>
              <noise type="gaussian">
                <mean>0.0</mean>
                <stddev>1.7e-2</stddev>
                <bias_mean>0.1</bias_mean>
                <bias_stddev>0.001</bias_stddev>
              </noise>
            </y>
            <z>
              <noise type="gaussian">
                <mean>0.0</mean>
                <stddev>1.7e-2</stddev>
                <bias_mean>0.1</bias_mean>
                <bias_stddev>0.001</bias_stddev>
              </noise>
            </z>
          </linear_acceleration>
        </imu>
      </sensor>
    </gazebo>
```

3.超声波

详细介绍及文章：[9.6拓展-为Fishbot添加超声波传感器.md]

```xml
<gazebo reference="ultrasonic_sensor_link">
  <sensor type="ray" name="ultrasonic_sensor">
    <pose>0 0 0 0 0 0</pose>
    <!-- 是否可视化，gazebo里能不能看到 -->
    <visualize>true</visualize>
    <!-- 扫描速率，也就是数据更新速率 -->
    <update_rate>5</update_rate>
    <ray>
      <scan>
        <!-- 水平扫描的点数 -->
        <horizontal>
          <samples>5</samples>
          <resolution>1</resolution>
          <min_angle>-0.12</min_angle>
          <max_angle>0.12</max_angle>
        </horizontal>
        <!-- 垂直方向扫描的点数 -->
        <vertical>
          <samples>5</samples>
          <resolution>1</resolution>
          <min_angle>-0.01</min_angle>
          <max_angle>0.01</max_angle>
        </vertical>
      </scan>
      <!-- 超声波检测的范围和数据分辨率单位m -->
      <range>
        <min>0.02</min>
        <max>4</max>
        <resolution>0.01</resolution>
      </range>
      <!-- 数据噪声采用高斯噪声 -->
      <noise>
        <type>gaussian</type>
        <mean>0.0</mean>
        <stddev>0.01</stddev>
      </noise>
    </ray>
```

```
      <plugin name="ultrasonic_sensor_controller"
  filename="libgazebo_ros_ray_sensor.so">
        <ros>
          <!-- 重映射输出的话题名称 -->
          <remapping>~/out:=ultrasonic_sensor_1</remapping>
        </ros>
        <!-- 输出消息的类型，注意与雷达区分，这里是sensor_msgs/Range -->
        <output_type>sensor_msgs/Range</output_type>
        <!-- 射线类型，这里要写ultrasound，注意和雷达区分 -->
        <radiation_type>ultrasound</radiation_type>
        <!-- frame名称，填写link名称即可 -->
        <frame_name>ultrasonic_sensor_link</frame_name>
      </plugin>
    </sensor>
  </gazebo>
```

4.两轮差速

详细介绍及文章： [9.3为FishBot配置两轮差速控制插件.md]

```xml
  <gazebo>
    <plugin name='diff_drive' filename='libgazebo_ros_diff_drive.so'>
          <ros>
            <namespace>/</namespace>
            <remapping>cmd_vel:=cmd_vel</remapping>
            <remapping>odom:=odom</remapping>
          </ros>
          <update_rate>30</update_rate>
          <!-- wheels -->
          <left_joint>left_wheel_joint</left_joint>
          <right_joint>right_wheel_joint</right_joint>
          <!-- kinematics -->
          <wheel_separation>0.2</wheel_separation>
          <wheel_diameter>0.065</wheel_diameter>
          <!-- limits -->
          <max_wheel_torque>20</max_wheel_torque>
          <max_wheel_acceleration>1.0</max_wheel_acceleration>
          <!-- output -->
          <publish_odom>true</publish_odom>
```

```
        <publish_odom_tf>true</publish_odom_tf>
        <publish_wheel_tf>true</publish_wheel_tf>
        <odometry_frame>odom</odometry_frame>
        <robot_base_frame>base_footprint</robot_base_frame>
    </plugin>
</gazebo>
```

5.JointStatePublisher

待补充

6.单目相机

```xml
<gazebo reference="camera_link">
<sensor type="camera" name="camera">
  <update_rate>30.0</update_rate>
  <camera name="head">
    <horizontal_fov>1.3962634</horizontal_fov>
    <image>
      <width>800</width>
      <height>800</height>
      <format>R8G8B8</format>
    </image>
    <clip>
      <near>0.02</near>
      <far>300</far>
    </clip>
    <noise>
      <type>gaussian</type>
      <mean>0.0</mean>
      <stddev>0.007</stddev>
    </noise>
  </camera>
  <plugin name="camera_controller" filename="libgazebo_ros_camera.so">
    <alwaysOn>true</alwaysOn>
    <updateRate>0.0</updateRate>
    <cameraName>/camera</cameraName>
```

```
        <imageTopicName>image_raw</imageTopicName>
        <cameraInfoTopicName>camera_info</cameraInfoTopicName>
        <frameName>camera_link</frameName>
        <hackBaseline>0.07</hackBaseline>
      </plugin>
    </sensor>
    <material>Gazebo/Blue</material>
  </gazebo>
```

7.深度相机

待补充

CMake

虽然你不需要完全了解 CMake 就能使用 ROS2，但稍微了解一点会很有帮助。你可能会对CMake tutorial感兴趣，它解释了 CMake 的基础知识。

Ament

Ament 是一组专门为 ROS2 设计的 CMake 模块，目的是使 CMake 更易于使用。另请参阅文档。

Ament 包的基本结构：

```cmake
cmake_minimum_required(VERSION 3.5)
project(my_package_name)

# Default to C++14
if(NOT CMAKE_CXX_STANDARD)
  set(CMAKE_CXX_STANDARD 14)
endif()

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()

# Find packages
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)

# Include our own headers
include_directories(include)

# Create a node
add_executable(my_node src/my_node.cpp)
ament_target_dependencies(my_node
  rclcpp
  # Other ament dependencies
  # This sets up include and linker paths
)

add_library(my_library src/my_library.cpp)
```

```
ament_target_dependencies(my_library
  rclcpp
)

# Install our headers
install(
  DIRECTORY include/
  DESTINATION include
)

# Install our node and library
install(TARGETS my_node my_library
  ARCHIVE DESTINATION lib
  LIBRARY DESTINATION lib
  RUNTIME DESTINATION lib/${PACKAGE_NAME}
)

# Install some Python scripts
install(
  PROGRAMS
    scripts/my_script.py
  DESTINATION
    lib/${PROJECT_NAME}
)

# Tell downstream packages where to find our headers
ament_export_include_directories(include)
# Tell downstream packages our libraries to link against
ament_export_libraries(my_library)
# Help downstream packages to find transitive dependencies
ament_export_dependencies(
  rclcpp
)
ament_package()
```

Linting 配置

我更喜欢更具 ROS1 风格的代码风格。要允许大括号位于各自的线路上，请执行以下操作：

```cmake
if(BUILD_TESTING)
  find_package(ament_cmake_cpplint)
  ament_cpplint(FILTERS "-whitespace/braces" "-whitespace/newline")
endif()
```

安装 Python 脚本

```cmake
install(
  PROGRAMS
    scripts/script1.py
    scripts/script2.py
  DESTINATION lib/${PROJECT_NAME}
)
```

取决于同一包中的消息

通常最好的做法是将消息放在不同的包中，但有时，特别是对于驱动程序，你希望将消息放在同一包中。

```cmake
find_package(rosidl_default_generators REQUIRED)

# Generate some messages
rosidl_generate_interfaces(${PROJECT_NAME}
  "msg/MyMessage.msg"
)

# Add a node which uses the messages
add_executable(my_node my_node.cpp)
rosidl_target_interfaces(my_node ${PROJECT_NAME} "rosidl_typesupport_cpp")
```

从普鲁金利布中去除助推器

默认情况下，Pluginlib 同时支持 Boost：：Shared_PTRS 和 STD：：Shared_PTRS，如果你想避免在闪亮的新 ROS2 库中依赖 Boost，你需要明确告诉 pluginlib 不要包含 Boost 版本：

```cmake
target_compile_definitions(your_library PUBLIC
"PLUGINLIB__DISABLE_BOOST_FUNCTIONS")
```

使用特征 3

将<>r="10"/>作为依赖项添加到你的 Package.xml 中，然后：

```cmake
find_package(Eigen3 REQUIRED)
include_directories(${EIGEN3_INCLUDE_DIRS})
```

在 C++ 中构建 Python 扩展

下面的示例基于包。

```cmake
find_package(PythonLibs REQUIRED)
find_package(Boost REQUIRED python)
find_package(ament_cmake_python REQUIRED)
find_package(python_cmake_module REQUIRED)

ament_python_install_package(${PROJECT_NAME})

add_library(
  my_python SHARED
  ${SOURCE_FILES}
)
set_target_properties(
  my_python PROPERTIES
  LIBRARY_OUTPUT_DIRECTORY ${CMAKE_CURRENT_BINARY_DIR}/${PROJECT_NAME}
  PREFIX ""
)
target_link_libraries(my_python
```

```
    ${Boost_LIBRARIES}
    ${PYTHON_LIBRARIES}
  )

  install(
    TARGETS my_python
    DESTINATION "${PYTHON_INSTALL_DIR}/${PROJECT_NAME}"
  )
```

Command Line: Colcon

Build

用于在工作区构建 ROS2 包。

生成所有包：

```
colcon build
```

要避免在调整 Python 脚本、配置文件和启动文件时重新构建：

```
colcon build --symlink-install
```

要修复大多数构建问题，特别是在向工作区添加或删除包或安装新的 RMW 实现的情况下，请清除 CMake 缓存。有关详细信息，请参阅<>r="9"/>帖子。

```
colcon build --cmake-clean-cache
```

CMake 参数

```
colcon build --cmake-args -DCMAKE_BUILD_TYPE=RelWithDebInfo
```

测试

要测试并将结果显示在屏幕上，请执行以下操作：

```
colcon test
```

```
colcon test-result --verbose
```

构建/测试单个包：

```
colcon <verb> --packages-select <package-name>
```

格式化

将输出显示到屏幕上：

```
colcon <verb> --event-handlers console_direct+
```

动作文档

- build
- test

基于 Python 的启动文件

基于 python 的启动文件几乎都遵循相同的结构。

> 注意，在 Foxy 之前，参数名称、名称空间和可执行文件的前缀:

```python
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        Node(
            name='node_runtime_name',
            package='ros2_package_name',
            executable='name_of_executable',
            parameters=[{'name_of_int_param': 1,
                         'name_of_str_param': 'value'}],
            remappings=[('from', 'to')],
            output='screen',
        ),
        # More Nodes!
    ])
```

使启动文件成为可执行文件

正常情况下，启动文件的运行方式为：

```
ros2 launch pkg launch.py
```

但是，有时你需要一个可执行的启动文件(例如放入一个系统作业)。假设你遵循上面所示的默认模式，则只需添加以下内容：

```python
#!/usr/bin/env python3

import sys
from launch import LaunchService

# define generate_launch_description() as above

if __name__ == '__main__':
    desc = generate_launch_description()
    service = LaunchService(argv=sys.argv[1:])
    service.include_launch_description(desc)
    return service.run()
```

从文件加载参数

有些节点有很多参数，将它们放在 YAML 文件中会更容易：

```yaml
node_name:
  ros__parameters:
      some_int_param: 1
      some_str_param: "the_value"
```

要加载此文件，请执行以下操作：

```python
from ament_index_python.packages import get_package_share_directory

# Assuming you have a file called package_name/config/params.yaml
node_params = os.path.join(
    get_package_share_directory('package_name'),
    'config',
    'params.yaml'
)

# Add this to your LaunchDescription
Node(
    name='node_runtime_name',
```

```
    package='ros2_package_name',
    executable='name_of_executable',
    parameters=[{'another_param': 42.0},
                node_params]
)
```

包括 Python 启动文件

```python
import os
from ament_index_python.packages import get_package_share_directory
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import PythonLaunchDescriptionSource

# Assuming you have a file called package_name/launch/my_launch.launch.py
my_launch_file = os.path.join(
    get_package_share_directory('package_name'),
    'launch',
    'my_launch.launch.py'
)

# Add this to your LaunchDescription
IncludeLaunchDescription(
    PythonLaunchDescriptionSource([my_launch_file])
),
```

加载 URDF

大多数机器人需要将其 URDF 加载到 ROBTO_STATE_PUBLISHER 中，也可能需要加载到其他节点中：

```python
import os
from ament_index_python.packages import get_package_share_directory
from launch_ros.actions import Node

# Load the URDF into a parameter
desc_dir = get_package_share_directory('robot_description_pkg')
```

```
urdf_path = os.path.join(desc_dir, 'robots', 'my_urdf.urdf')
urdf = open(urdf_path).read()

# Add this to your LaunchDescription
Node(
    name='robot_state_publisher',
    package='robot_state_publisher',
    executable='robot_state_publisher',
    parameters=[{'robot_description': urdf}],
)
```

安装启动文件

- Python
- C++/CMake

联网

ROS2 使用 DDS 进行消息传输。

设置环境变量 RMW_IMPLICATION 以选择 DDS 实现(RMW= 机器人中间件)。例如：

```
export RMW_IMPLEMENTATION=rmw_cyclonedds_cpp
```

要检查正在使用的 RMW 实施，请执行以下操作：

```
ros2 doctor --report | grep middleware
```

DDS 发现

在 ROS1 中没有类似的。节点发现是对等的，节点在启动时宣布它们的主题，并在启动后定期发布主题。默认情况下，如果同一网络上的任何计算机具有相同的 ROS_DOMAIN_ID，则它们将相互查看。

ROS_DOMAIN_ID 可以是 0 到 253 之间的任何数字，但建议使用小于 128 的数字。

除了 ROS_DOMAIN_ID 之外，CycloneDDS 还支持域标签，允许对网络进行几乎无限的分区(见下文)。

如果希望将通信限制在本地主机上，请设置 ROS_LOCALHOST_ONLY，它是。

CycloneDDS

Cyclone 可以使用 XML 进行配置。它可以存储在文件中，也可以直接传递到环境变量 CYCLONEDDS_URI 中。可以在<>r="9"/>中找到支持选项的完整列表。另请参阅<>r="10"/>。

CycloneDDS：多个接口

气旋目前只能与单一网络接口配合使用。如果你有多个接口，请指定要在 NetworkInterfaceAddress 中使用的接口：

```xml
<CycloneDDS>
  <Domain>
    <General>
      <NetworkInterfaceAddress>wlp2s0</NetworkInterfaceAddress>
    </General>
  </Domain>
</CycloneDDS>
```

CycloneDDS：禁用组播(发现除外)

某些网络硬件在使用多播时性能不佳(尤其是在使用 WiFi 时)。你可以将多播限制为仅发现：

```xml
<CycloneDDS>
  <Domain>
    <General>
      <AllowMulticast>spdp</AllowMulticast>
    </General>
  </Domain>
</CycloneDDS>
```

CycloneDDS：域标签

CycloneDDS 还定义了“域标签”，允许使用自定义字符串大幅划分网络：

```xml
<CycloneDDS>
  <Domain>
    <Discovery>
      <Tag>my_robot_name</Tag>
    </Discovery>
  </Domain>
</CycloneDDS>
```

示例

以上标签都可以组合在一起：

```xml
<CycloneDDS>
  <Domain>
    <General>
      <!-- Explicitly set network interface -->
      <NetworkInterfaceAddress>wlp2s0</NetworkInterfaceAddress>
      <!-- Use multicast for discovery only -->
      <AllowMulticast>spdp</AllowMulticast>
    </General>
    <Discovery>
      <!-- This tag has to be the same on each machine -->
      <Tag>my_robot_name</Tag>
    </Discovery>
  </Domain>
</CycloneDDS>
```

程序包文档

目前，文档有点无处不在。这个列表当然不是很广泛，但我发现快速链接到常用的包是很有帮助的。

核心文档

- ROS2 中文网
- ROS 2 概述
- Ament CMake
- ROS2 Launch

其他功能包

- Nav2