第 1 课 ROS2 简介和 ROS1 的对比

1.ROS2 概述

ROS2 是第二代的 Robot Operating System，是 ROS1 的升级版本，解决了 ROS1 存在的一些问题。ROS2 最早出现的版本 Arden 是在 2017 年，随着版本的迭代，不断地更新优化，现如今已经有了稳定的版本。与 ROS1 相同，Linux 版本与 ROS2 版本的选择也是有关系的，两者对应的版本如下：

ROS2 版本	Ubuntu 版本
Foxy	Ubuntu20.04
Galactic	Ubuntu20.04
Humble	Ubuntu22.04

2.ROS2 的特点

1） 分布式架构：ROS 2 采用分布式架构，允许在多个计算机上运行不同的节点。这使得 ROS 2 系统可以在更大规模的机器人系统中运行，并支持更高级别的并行处理和通信。

2） 支持多语言：ROS 2 支持多种编程语言，包括 C++、Python、Java 等。这使得开发人员可以使用自己熟悉的语言编写 ROS 2 应用程序，提高了开发效率和灵活性。

3） 强化的通信机制：ROS 2 引入了一种新的通信机制，称为 Data Distribution Service（DDS）。DDS 是一种高性能、实时的消息传递协议，可以在 ROS 2 系统中实现可靠的数据通信。

4） 系统级别的工具：ROS 2 提供了一套系统级别的工具，用于管理和监控 ROS 2 系统。这些工具包括包管理工具（如 Colcon）、日志记录工具（如 Rosout）和诊断工具（如 Rqt）等，可以帮助开发人员更好地调试和管理机器人应用程序。

5） 实时性能：ROS 2 在设计上考虑了实时性能，并提供了一些实时性能工具和库，如

Real-Time Executor（RTE）、Real-Time Publisher（RTPS）等。这使得 ROS 2 可以在对实时性能要求较高的应用场景中使用。

6） 易于扩展和集成：ROS 2 支持模块化的架构，允许开发人员方便地添加新的功能和扩展 ROS 2 系统。此外，ROS 2 还与其他常用的机器人软件和库集成，例如 Gazebo 仿真器和 MoveIt 机器人运动规划库。

3.ROS2 与 ROS1 的区别

3.1 平台

ROS1 目前来说仅支持在 Linux 系统中运行使用，常见的是在 Ubuntu 中搭建使用。而 ROS2 目前在 Ubuntu、Windows 甚至嵌入式开发板上都可以搭建使用，平台更加广泛。

3.2 语言

- C++

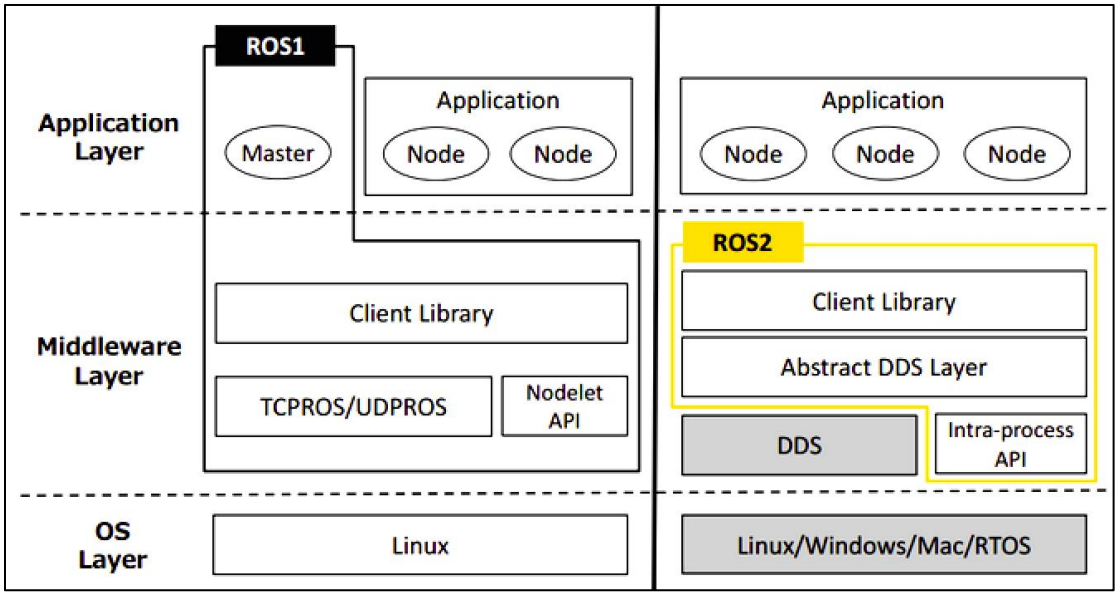
ROS1 的核心是 C++03，而 ROS2 广泛使用 C++11.

- Python

ROS1 的 Python 使用版本是 Python2，而 ROS2 使用的 Python 版本至少是 3.5 以上，Humble 使用的 Python 版本是 3.6。

3.3 中间件

ROS1 启动前需要开启 roscore，这个 master 掌握所有的节点之间的通讯，而 ROS2 则没有，只有一个抽象的中间件接口，通过该接口进行传输数据。目前，此接口的所有实现都基于 DDS 标准。这使得 ROS2 能够提供各种优质的 Qos 服务策略，从而改善不同网络的通信。



3.4 编译命令

ROS1 的编译命令是 catkin_make，而 ROS2 的编译命令使用 colcon build 命令。

如需要进一步开发学习，可以参考官方（https://docs.ros.org/en/rolling/index.html）的教程。

第 2 课 ROS2 安装方法

注意：我们的树莓派镜像出厂时已经安装好 ROS2 环境，这里是测试 ROS2 是否可以正常使用。

1.测试步骤（测试已安装的 ROS2 环境）

（1）按下“**Ctrl+Alt+T**”，打开命令行终端，在终端输入 docker version。如果报错提示 docker 未找到命令，输入 sudo snap install docker 安装 docker 后再输入 docker version，就会显示 docker 版本。

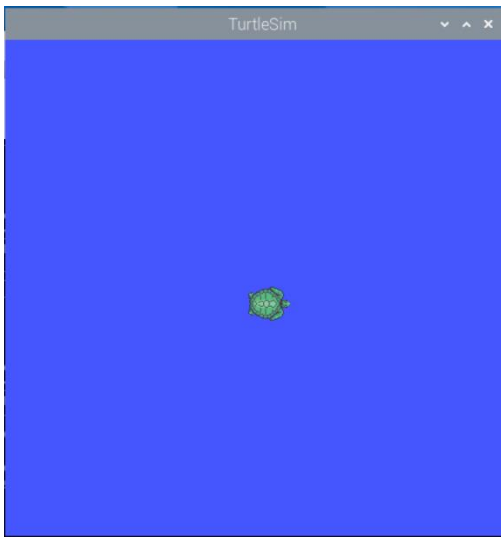
```
ubuntu@ubuntu-virtual-machine:~$ docker version
Command 'docker' not found, but can be installed with:
sudo snap install docker          # version 24.0.5, or
sudo apt  install docker.io       # version 24.0.5-0ubuntu1~22.04.1
sudo apt  install podman-docker   # version 3.4.4+ds1-1ubuntu1.22.04.2
See 'snap info docker' for additional versions.
```

```
ubuntu@ubuntu-virtual-machine:~$ sudo snap install docker
Setup snap "docker" (2932) security profiles for auto-connections
docker 24.0.5 from Canonical✓ installed
```

```
ubuntu@ubuntu-virtual-machine:~$ docker version
Client:
 Version:           24.0.5
 API version:       1.43
 Go version:        go1.20.14
 Git commit:        ced0996
 Built:             Tue Jun 25 22:37:33 2024
 OS/Arch:           linux/amd64
 Context:           default
```

（2）输入“**ros2 run turtlesim turtlesim_node**”，启动小海龟 GUI 界面，启动成功即说明 ROS2 能正常使用。

```
ubuntu@ubuntu-virtual-machine:~$ ros2 run turtlesim turtlesim_node
```



2.安装步骤

如果想自己学习安装配置 ROS2 的环境，可以参考下方步骤，这里以 humble 版本为例（需要联网。请到第 2 章 树莓派 5 基础操作和配置 第 7 课 配置网络方法）。

1） 按下“**Ctrl+Alt+T**”，打开命令行终端，在终端输入“**docker pull ros:humble**”，下载 ROS2 镜像，镜像下载需要一些时间，请耐心等待一会。

```
pi@raspberrypi:~ $ docker pull ros:humble
```

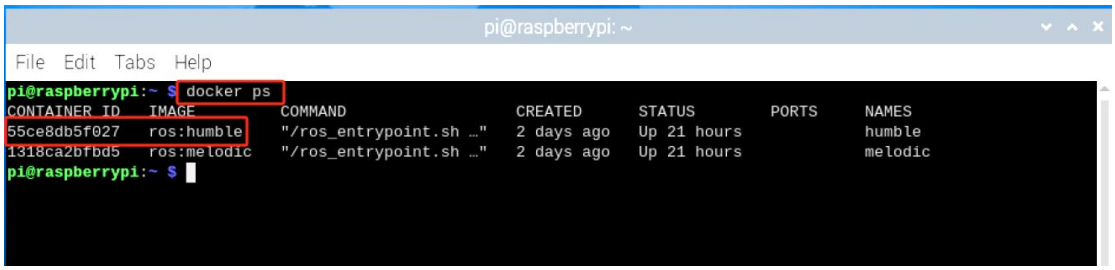
2） 镜像下载完成后，在终端输入“**docker run -it --network=host -d -v=/dev:/dev -v /tmp/.X11-unix:/tmp/.X11-unix --name humble -e DISPLAY=${DISPLAY} --restart=always ros:humble /bin/bash**”，运行容器并指定名称为 humble。

```
pi@raspberrypi:~ $ docker run -it --network=host -d -v=/dev:/dev -v /tmp/.X11-unix:/tmp/.X11-unix --name melodic -e DISPLAY=${
DISPLAY} --restart=always ros:melodic /bin/bash
75b6634e622762255058569dd769ed3925f77b9976539e85d1d70cd029929f9d
```

3） 在终端输入“**xhost +**”，开启 X Server 的访问控制。

```
pi@raspberrypi:~ $ xhost +
access control disabled, clients can connect from any host
```

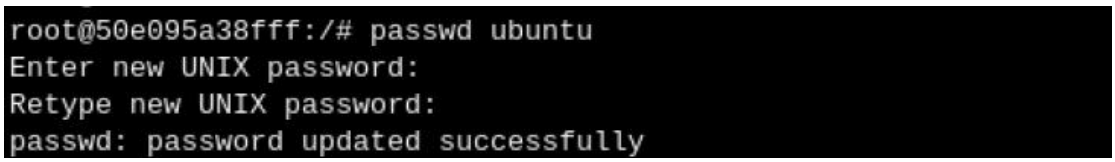
4） 在终端输入“**docker ps**”查看新创建容器的 ID。



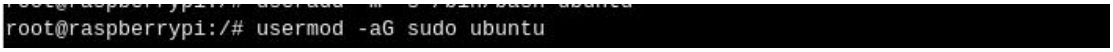
5） 在终端输入“**docker exec -it 55ce /bin/bash**”进入容器（容器的 ID 可以简写，只要是该容器唯一标识即可）。

6） 在终端输入“**useradd -m -s /bin/bash ubuntu**”，创建一个新用户。

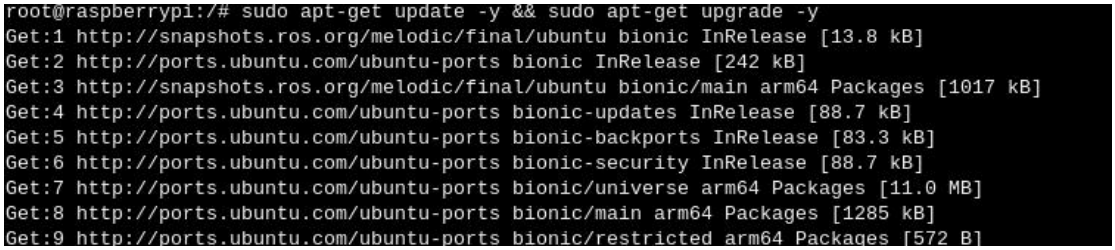
7） 在终端输入“**passwd ubuntu**”，设置“**ubuntu**”的密码，这里我们设置为“**ubuntu**”输入密码按下之后，需要重新输入一遍密码。



8） 在终端输入“**usermod -aG sudo ubuntu**”，将新用户 ubuntu 添加 sudo，使其具有超级用户权限。



9） 在终端输入“**sudo apt-get update -y && sudo apt-get upgrade -y**”，更新可用软件包列表并升级系统上已安装的软件包。



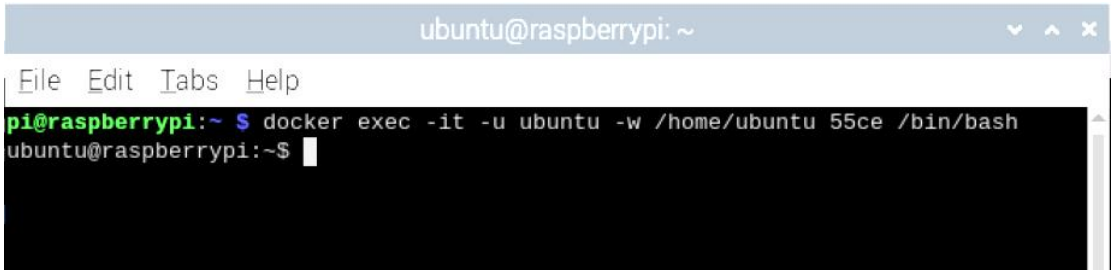
10) 在终端输入“**sudo apt-get install vim -y**”，安装 Vim 文本编辑器。

```
root@raspberrypi:/# sudo apt-get install vim -y
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  libgpm2 vim-common vim-runtime xxd
Suggested packages:
  gpm ctags vim-doc vim-scripts
The following NEW packages will be installed:
```

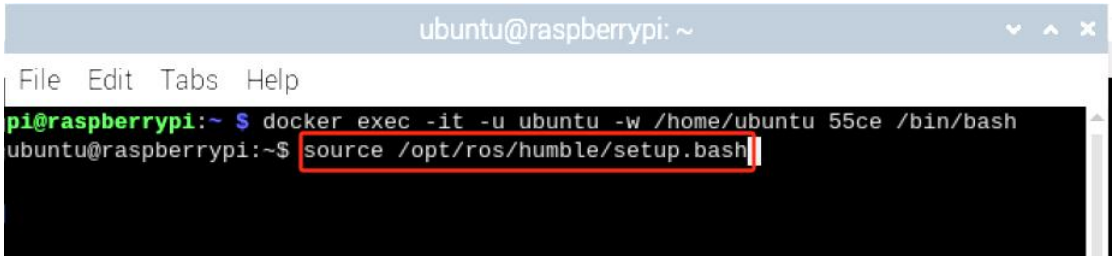
11）在终端输入“**sudo apt-get install ros-humble-desktop-full -y**”，安装 ROS Humble 版本的完整桌面环境。

```
root@raspberrypi:/# sudo apt-get install ros-melodic-desktop-full -y
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  autoconf automake autopoint autotools-dev avahi-daemon bind9-host blt bsdmainutils curl cython dbus d
  dh-autoreconf dh-strip-nondeterminism dmsetup file fltk1.3-doc fluid fonts-lato fonts-liberation font
```

12）输入“**docker exec -it -u ubuntu -w /home/ubuntu 55ce /bin/bash**”指令，进入容器。（注意：55ce 是装有 ROS2 环境的容器 ID）



13）输入“**source /opt/ros/humble/setup.bash**”指令，手动配置 ROS2 环境。



14）每一次打开终端都执行（3）步骤加载工作空间，可以在终端输入“**echo “source /opt/ros/humble/setup.bash” >> ~/.bashrc**”指令,将该指令写入.bashrc 文件中。

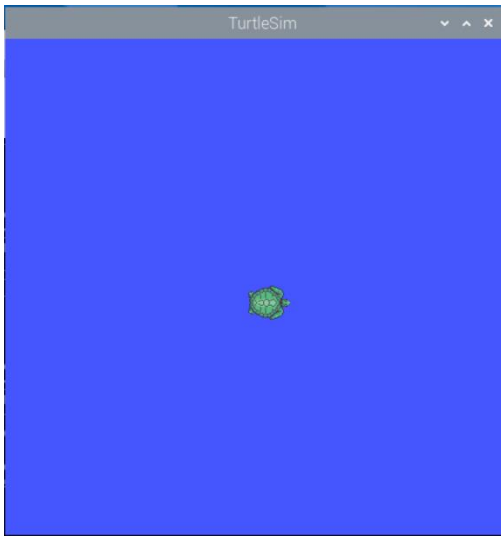
```
ubuntu@raspberrypi:~$ echo "source /opt/ros/humble/setup.bash" >> ~/.bashrc
```

15）然后输入“**source ~/.bashrc**”指令，让**.bashrc** 文件生效。这样就不需要每一次加载工作空间环境。

```
ubuntu@raspberrypi:~$ source ~/.bashrc
ubuntu@raspberrypi:~$
```

16）输入“**ros2 run turtlesim turtlesim_node**”，启动小海龟 GUI 界面，启动成功即说明 ROS2 安装成功。

```
ubuntu@raspberrypi:~$ ros2 run turtlesim turtlesim_node
```



第 3 课 ROS2 命令行操作

1.ROS2 文件系统的组成

ROS2 文件是由 Packages 和 Manifests（package.xml）组成。

Packages（功能包）是 ROS 2 软件中的基本单元，它是一组相关的文件和目录的集合，用于组织和管理 ROS 2 的节点、库和资源等。功能包中包含了节点的源代码、配置文件、消息和服务定义等。

Manifests（package.xml）是功能包的描述文件，用于定义功能包的相关元信息和依赖关系。在 ROS 2 中，功能包通常包含一个名为 package.xml 的文件，其中包含有关功能包的元数据信息，如名称、版本、维护者、许可协议和依赖关系等。

Manifest 文件（package.xml）在 ROS 2 中具有重要的作用，它提供了对功能包的描述和管理，使得 ROS 2 系统能够正确地处理功能包之间的依赖关系，并进行合适的构建和运行。

2.了解 ROS2 的基本术语

下表是对 ROS2 的部分基本术语的解释说明：

术语名称	说明
底层通讯层（DDS）	提供了高性能、可靠、实时的数据通信和集成能力，为 ROS 2 的节点之间的消息传递和服务调用提供了基础支持
节点（Node）	在 ROS 中运行的最小处理单元，通常是一个可执行文件。每个节点可使用话题或服务与其它节点进行通信。
消息（Message）	是 int、float 和 boolean 等数据类型的变量。
话题（Topic）	一种单向异步通信机制。通过发布消息到话题或订阅话题

	的形式，可以在节点之间实现数据的传输。话题的类型由对应消息的类型决定。
发布（Publishing）	以与话题内容对应的消息类型发送数据。
发布者（Publishs）	为执行发布，发布者节点在主节点上注册自己的话题等多种信息，并向希望订阅的订阅者节点发送消息。
订阅（Subscribing）	以与话题内容对应的消息类型接收数据。
订阅者（Subscribers）	为执行订阅，订阅者节点在主节点上注册自己的话题等多种信息，并从主节点接收所有发布了此节点需订阅话题的发布者节点的信息。
服务（Services）	一种双向同步通信机制。服务客户端请求对应于特定目的任务的服务，服务服务端进行服务响应。
服务服务器（Service Servers）	以请求作为输入，响应作为输出的节点。
服务客户端（Service Clients）	以响应作为输入，请求作为输出的节点。

3.了解 ROS2 的常用文件

下表是对 ROS2 中部分常用文件的介绍说明：

术语名称	说明
urdf 文件	描述机器人所有元素的模型文件，包含连杆（link）、关节（joint）、运动学参数（axis）、动力学参数（dynamics）、可视化

	模型（visual）和碰撞检测模型（collision）。
srv 文件	存放在 srv 文件夹下，用于定义 ROS 服务消息，包含请求和响应两个部分，请求与响应之间使用符号“---”进行分隔。
msg 文件	存放在 msg 文件夹下，用于定义 ROS 话题消息。
package.xml	描述功能包的属性，包含功能包的名字、版本号、作者等。
CmakeLists.txt	编译配置文件，使用 Cmake 编译。
launch 文件	启动文件，包含系统性启动机器人所需的 node 和 services。

4. 了解常用命令

4.1 包管理工具

ros2 pkg 有 create、executables、list、prefix、xml 共 5 个命令。

命令	详细说明
ros2 pkg create	创建功能包，指定包名、编译方式、依赖项、节点名等
ros2 pkg list	查看系统中功能包列表
ros2 pkg executables	查看包内可执行文件列表
ros2 pkg prefix	查看指定功能包的安装路径前缀
ros2 pkg xml	查看指定功能包的 package.xml 文件内容

4.2 节点运行工具

ros2 run 的功能是运行包内节点，格式为：“**ros2 run <package_name> <node_name>**”，如：

```
ubuntu@raspberrypi:~$ ros2 run turtlesim turtlesim_node
[StandardPaths: XDG_RUNTIME_DIR not set, defaulting to '/tmp/runtime-ubuntu'
[INFO] [1705999223.034178002] [turtlesim]: Starting turtlesim with node name /turtlesim
[INFO] [1705999223.044200929] [turtlesim]: Spawning turtle [turtle1] at x=[5.544445], y=[5.544445], theta=[0.000000]
libGL error: glx: failed to create dri3 screen
libGL error: failed to load driver: vc4
```

4.3 节点查看工具

ros2 node 命令用于查看节点消息。

命令	详细说明
ros2 node list	查看当前域内（ROS_DOMAIN_ID 相同的节点组）活动的节点列表
ros2 node info	查看节点详细信息，包括订阅、发布的消息，开启的服务和动作等

4.4 主题操作工具

ros2 topic 命令是用于对 ROS2 中的话题进行操作。

命令	详细说明
ros2 topic list	列出域内可使用的主题列表
ros2 topic info	显示主题消息类型、订阅者数量、发布者数量等
ros2 topic type	查看主题消息类型

ros2 topic find	按消息类型查找相关主题
ros2 topic hz	显示主题平均发布频率
ros2 topic bw	显示所查阅主题的带宽
ros2 topic delay	通过 header 中的时间戳计算消息延迟
ros2 topic echo	在控制台显示主题消息
ros2 topic pub	通过命令行发布指定主题消息

4.5 接口操作工具

ros2 interface 指令是用于对 ROS2 中的接口进行操作。

命令	详细说明
ros2 interface list	显示系统内所有的接口，包括消息（Messages）、服务（Services）、动作（Actions）
ros2 interface package	显示指定接口包内的子接口
ros2 interface packages	显示指定接口包
ros2 interface show	显示指定接口的详细内容
ros2 interface proto	显示消息模板

第 4 课 ROS2 开发环境配置

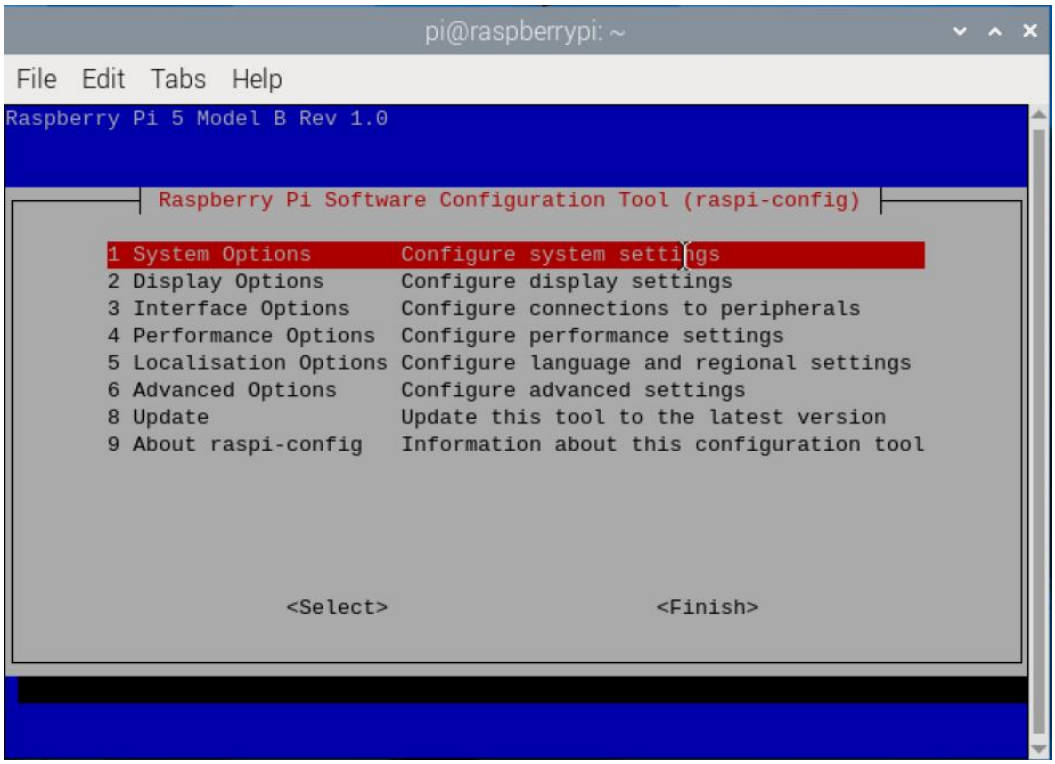
注意：我们的树莓派镜像出厂时已经安装好配置好开发环境，这里只供学习参考。

在后续学习和开发程序时，我们可以通过优化开发环境和安装插件来提高效率。

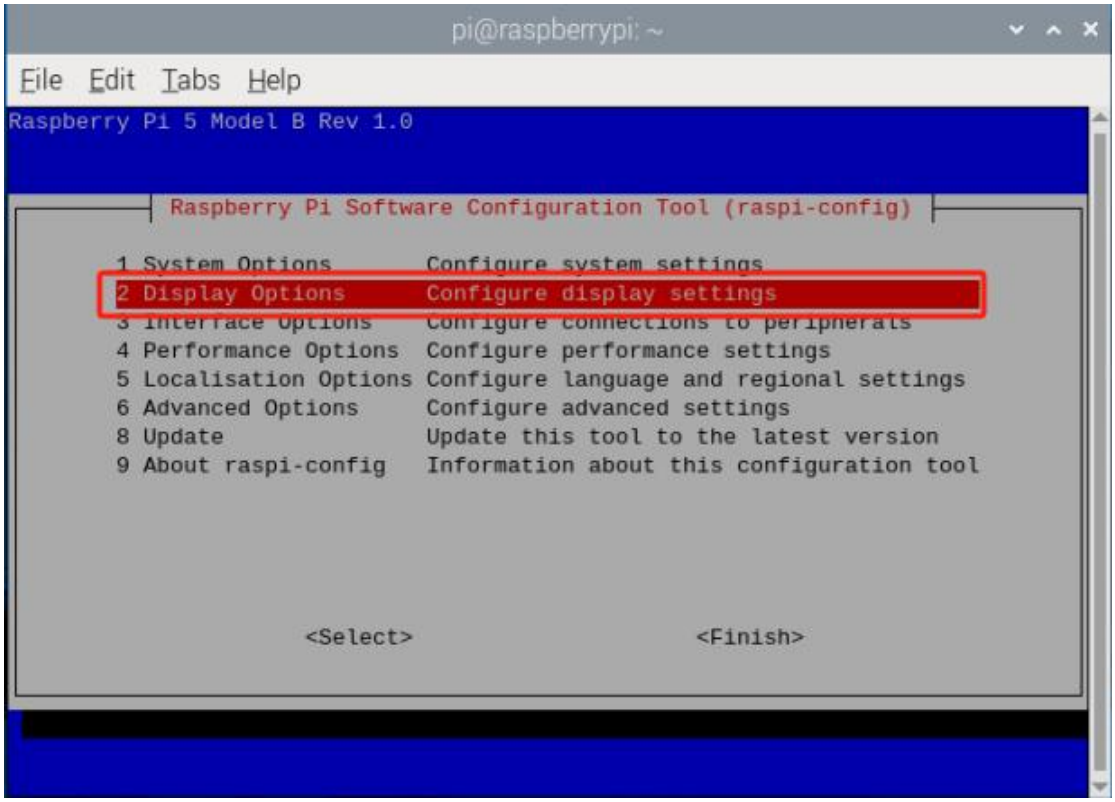
1.调整分辨率

1) 按下“**Ctrl+Alt+T**”，打开命令行终端，输入“**sudo raspi-config**”命令，然后按下回车。

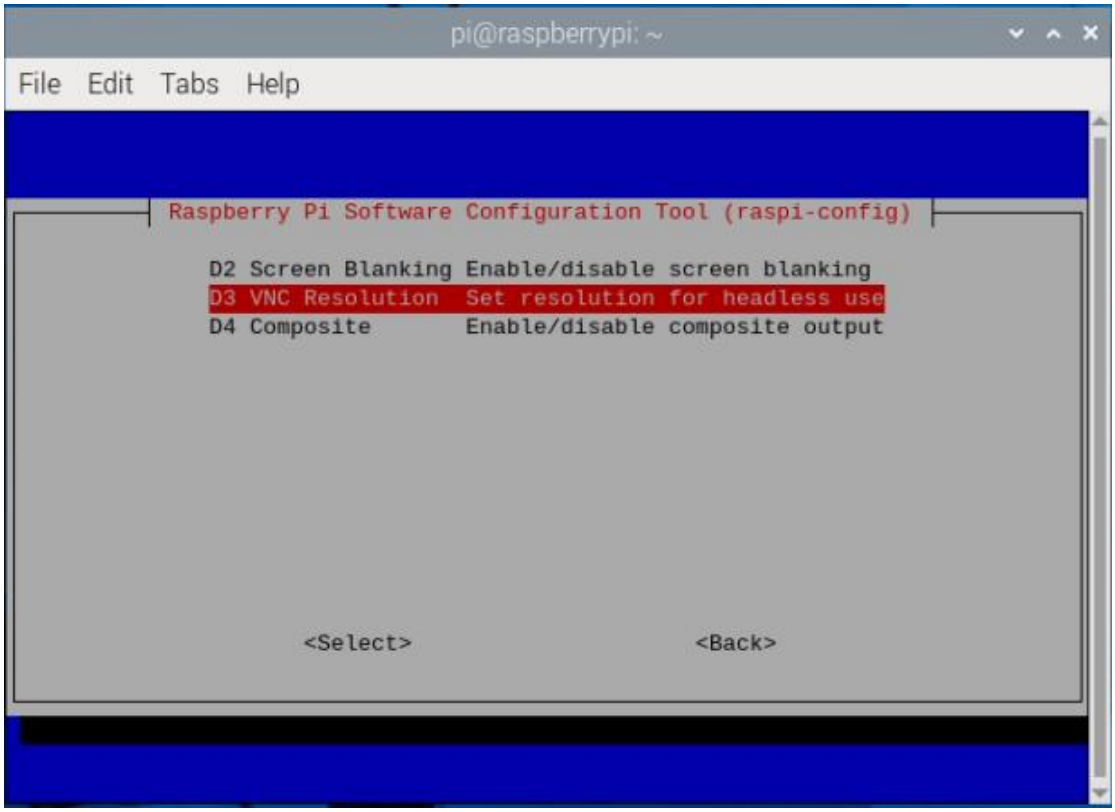
2) 在该界面使用“**↑↓**”箭头选择，“**Enter**”回车键用于确定，“**Esc**”键用于返回上一级。



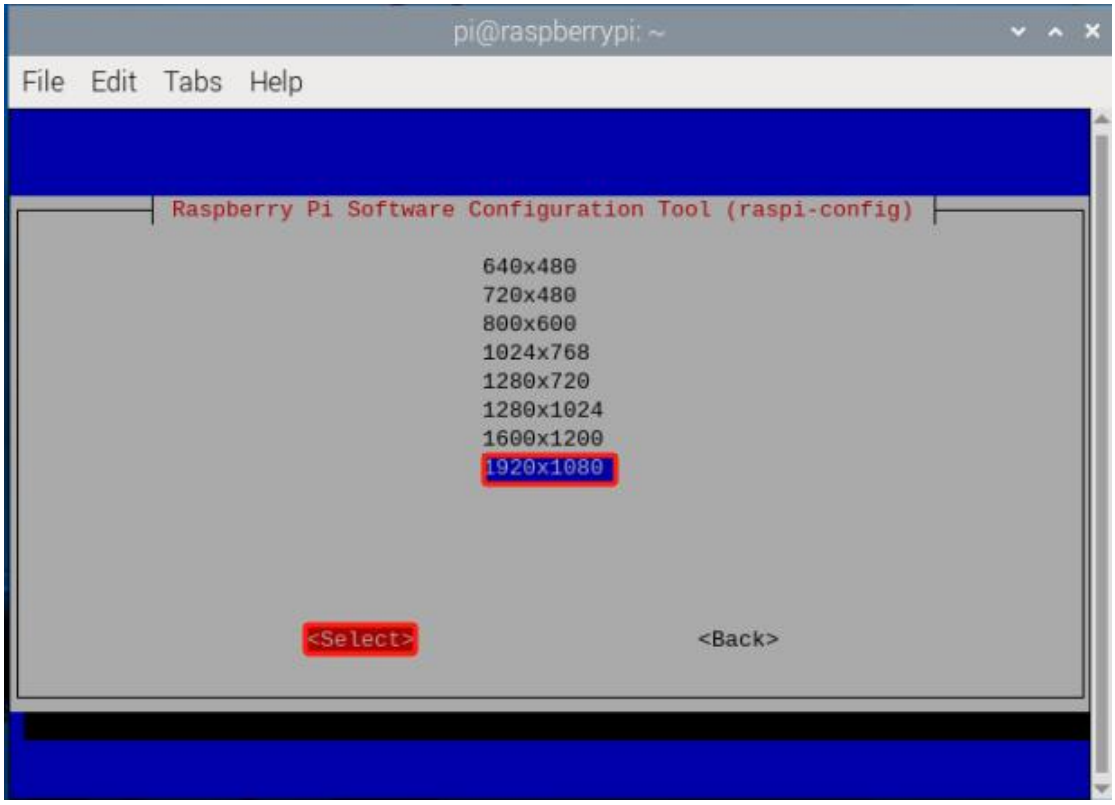
3) 选择“**2 Display Options**”，然后按下“Enter”回车键确定。



4) 选择“**D3 VNC Resolution**”，然后按下“Enter”回车键确定。



5) 选择“**1920*1080**”，然后选择“**<Select>**”，按下“Enter”回车键确定。

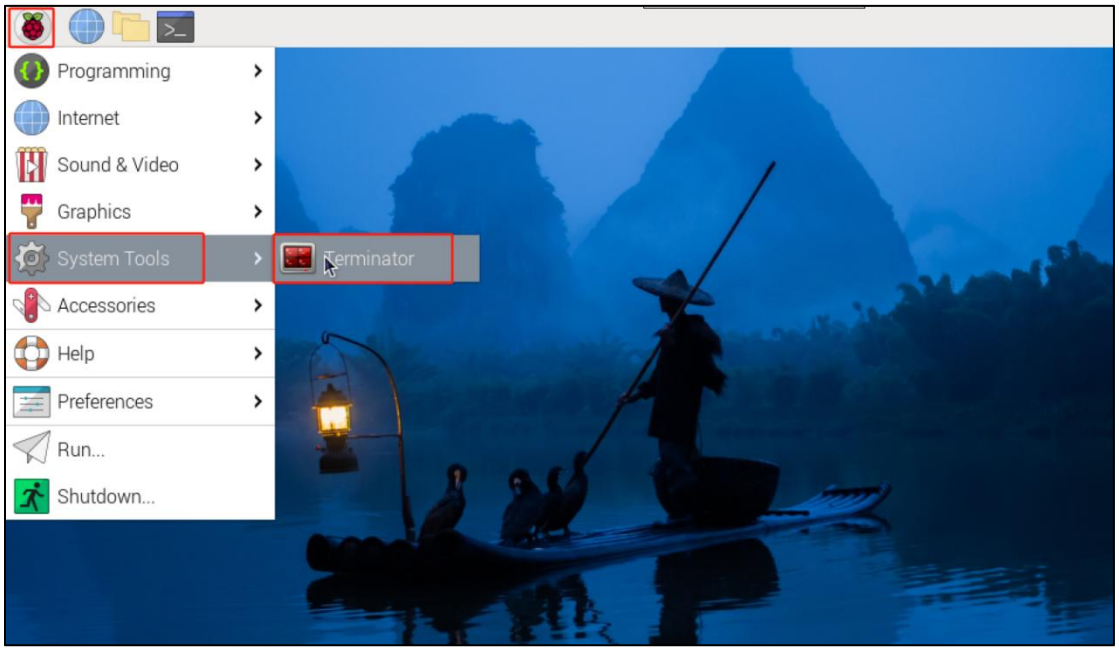


6） 修改完后会重启树莓派，重新进入树莓派后即可完成修改。

2.更换软件源

我们从 DockerHub 上拉取镜像、创建容器以实现各种各样的功能。但是由于大部分的 Docker 镜像都是默认国外的软件源，下载更新非常慢，更改 Docker 容器国内软件源，即可提高下载更新速度。

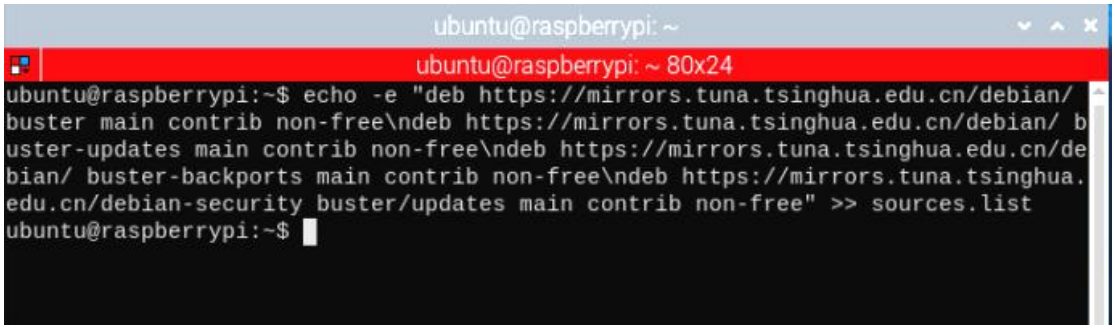
1） 点击左上角的 logo，选择 System Tools →Terminator。



2） 输入以下指令，按下回车键即可换回国内的清华源。

```
echo -e "deb https://mirrors.tuna.tsinghua.edu.cn/debian/ buster main co
ntrib non-free\ndeb

https://mirrors.tuna.tsinghua.edu.cn/debian/ buster-updates main contrib
 non-free\ndeb

https://mirrors.tuna.tsinghua.edu.cn/debian/ buster-backports main contr
ib non-free\ndeb
```

```
https://mirrors.tuna.tsinghua.edu.cn/debian-security buster/updates main
 contrib non-free" >> sources.list
```

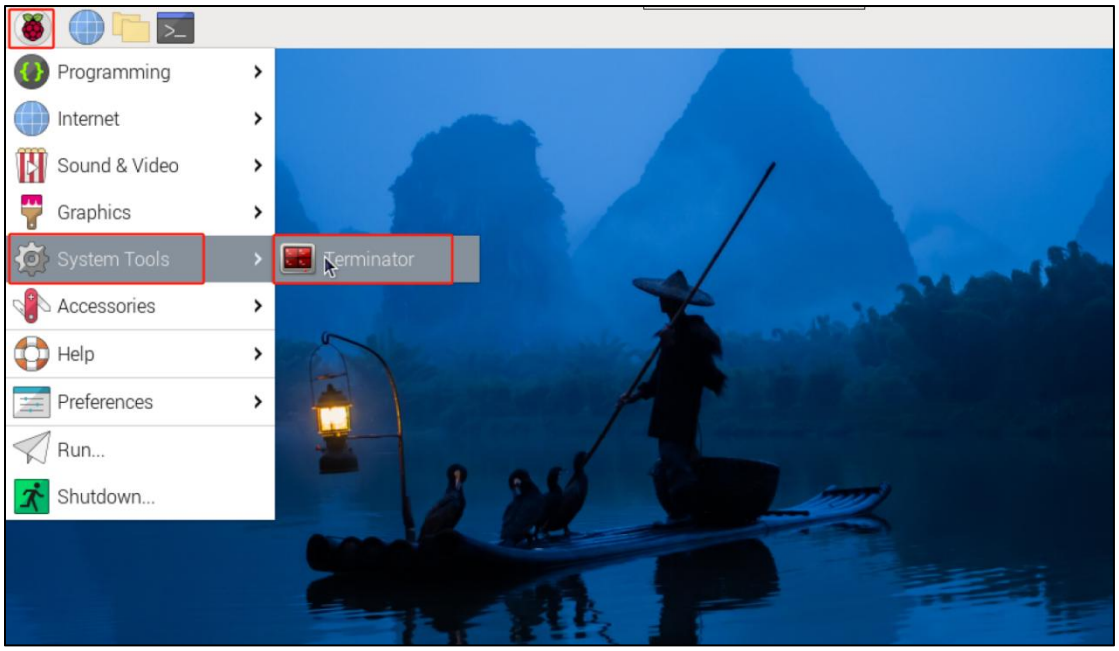


3） 输入“sudo apt update”指令，更新软件源即可。

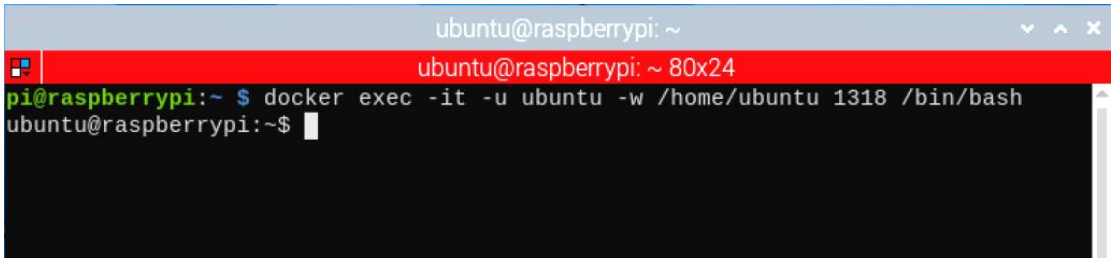
3.便携工具使用

在后续的 ROS 开发，需要同时开启多个终端，在这个时候就需要 terminator 工具。

4） 点击左上角的 logo，选择 System Tools →Terminator。

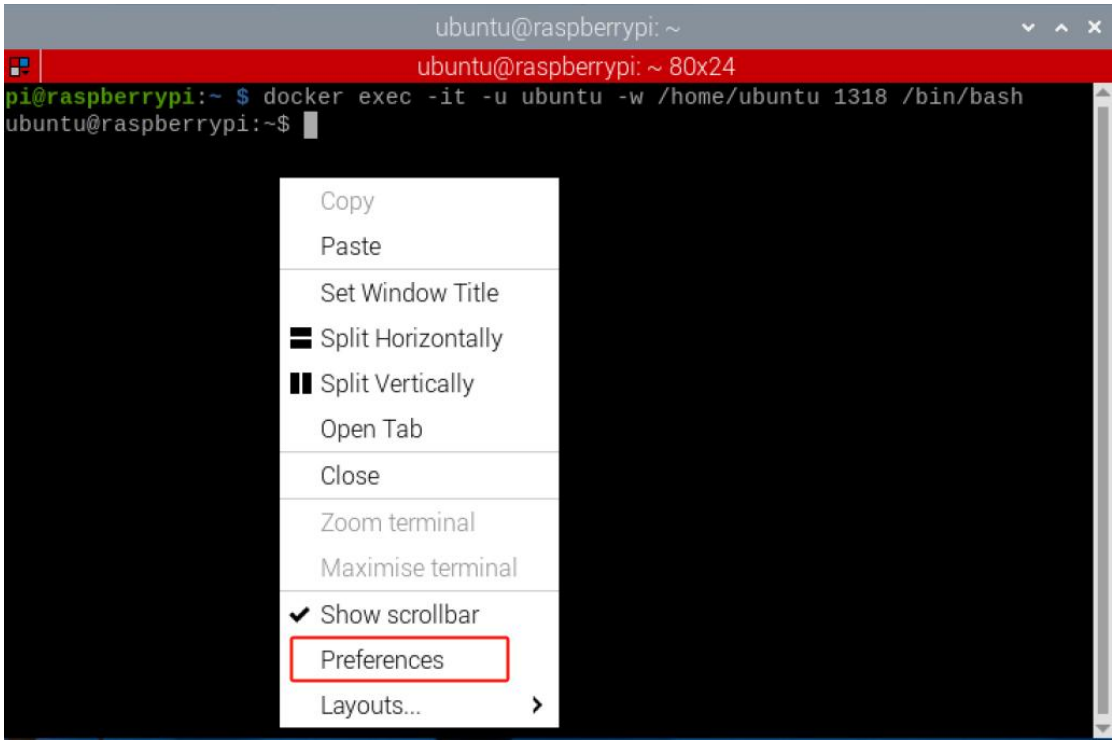


5） 输入“**docker exec -it -u ubuntu -w /home/ubuntu 1318 /bin/bash**”进入容器。

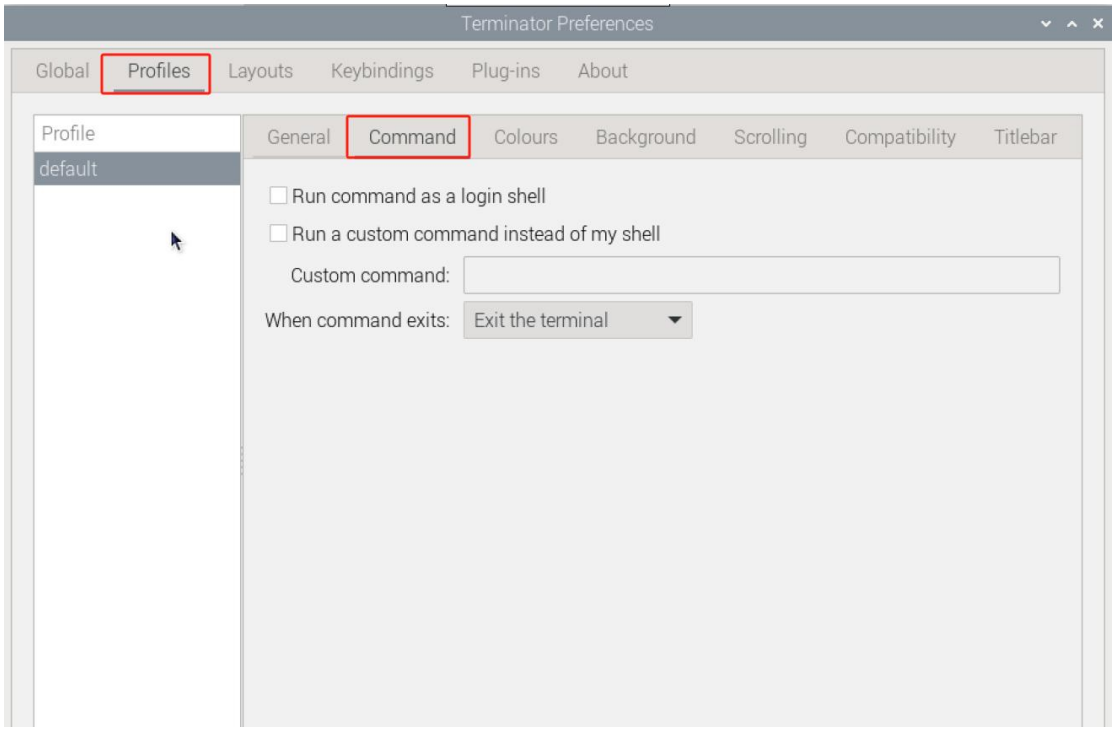


每次进入容器之前都需要输入在 terminator 终端输入指令，这就很麻烦，可以在 terminator 工具里面设置进入容器的指令。

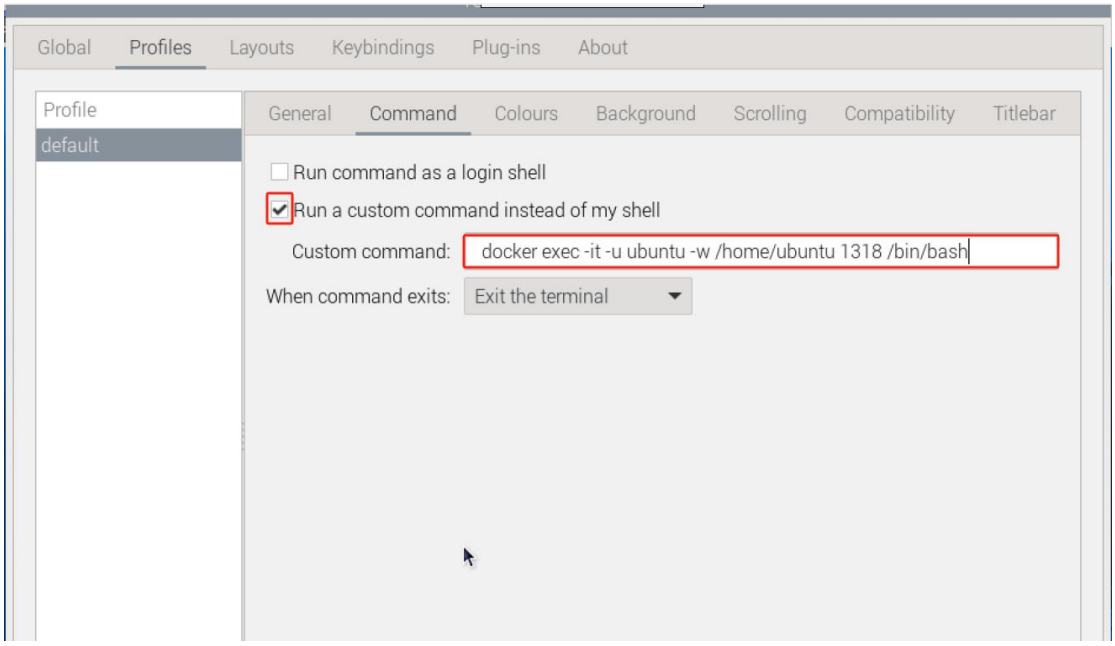
6） 在 terminator 窗口右键，选择“**Preference**”点击。



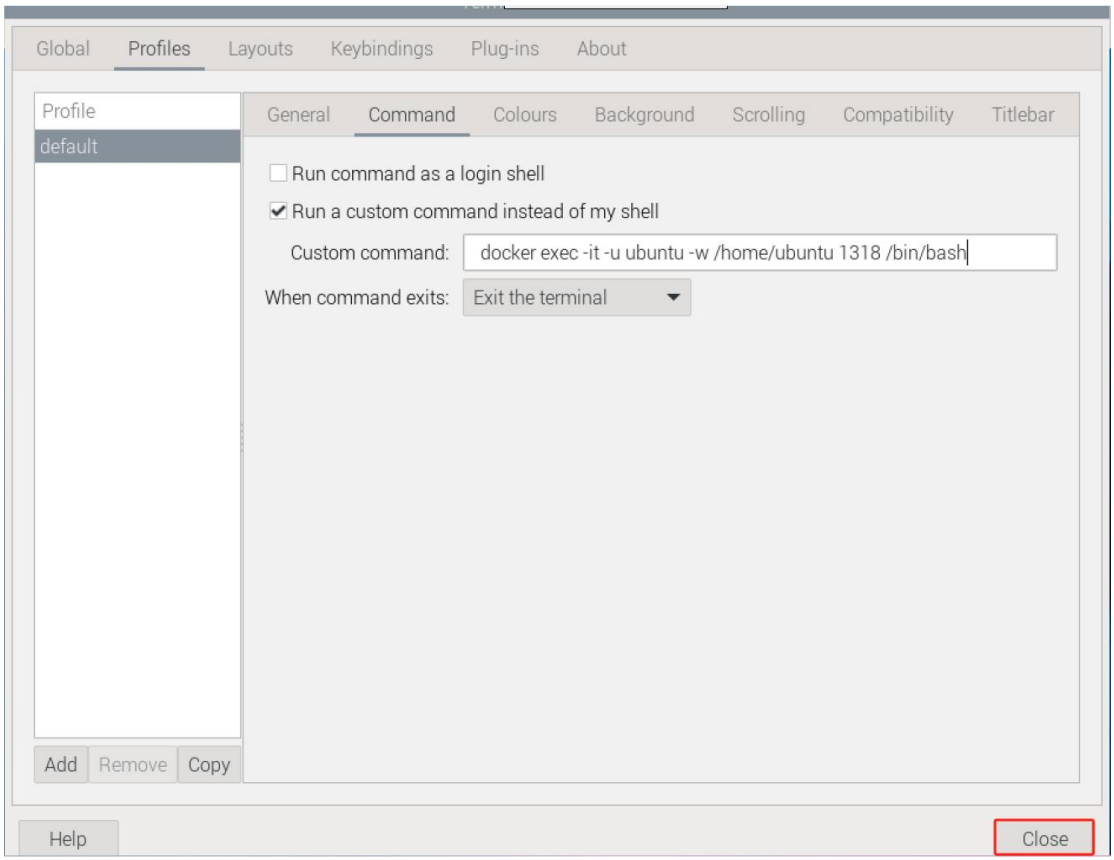
7） 选择 **Profiles→Command**。



8） 在方框内“√”中，输入“**docker exec -it -u ubuntu -w /home/ubuntu 1318 /bin/bash**”进入容器的指令。（注意：1318 是装有 ros2 环境的容器 ID）



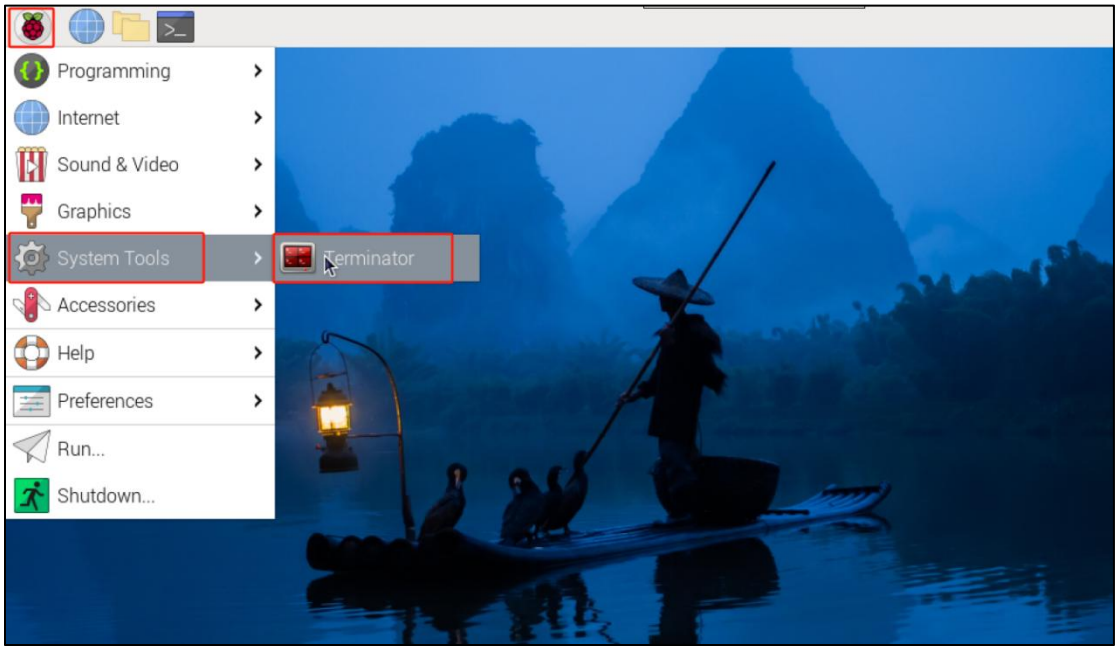
9） 然后点击关闭，这样每次开启 terminal，可以直接进入装 ros2 环境的容器内。



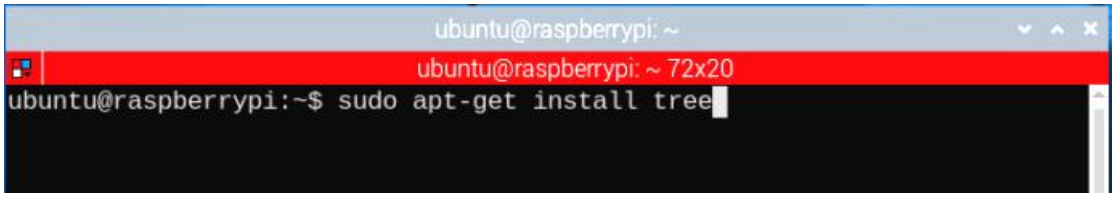
4.Tree 工具安装

Tree 工具是一个命令行工具，用于以树状结构显示目录和文件的层次结构。它可以帮助用户以清晰的方式查看文件系统中的目录结构，包括文件和子目录。

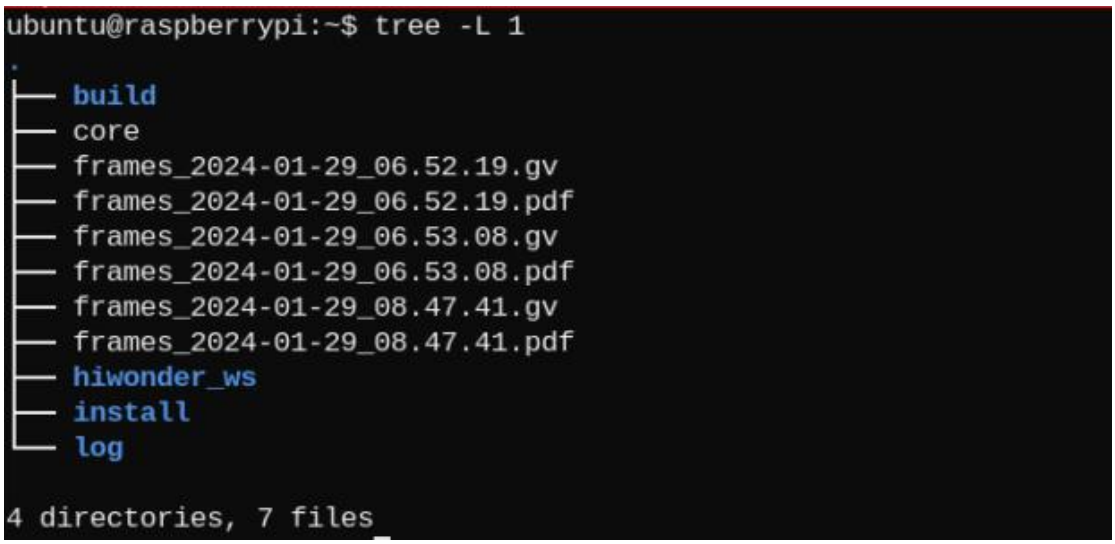
1） 点击左上角的 logo，选择 System Tools →Terminator。



2） 输入“sudo apt-get install tree”指令，即可对 tree 工具进行安装。



3） 安装成功后，输入“**tree -L 1**”，即可对当前文件夹的一级目录结构进行显示。



5.pip 软件包安装

python3-pip 是 Python 3 的包管理工具 pip 的一个软件包。pip 是 Python 的官方包管理工具，用于安装、升级和管理 Python 包。

输入“sudo apt-get install python3-pip”即可进行安装。

```
ubuntu@raspberrypi:~$ sudo apt-get install python3-pip
```

6.transforms3d 库安装

transforms3d 是一个 Python 库，用于进行三维变换和旋转矩阵的计算。它提供了一组函数和类，用于在三维空间中执行各种变换操作，如旋转、平移和缩放。transforms3d 库具有简单易用的接口，可以用于处理 3D 图形、机器人运动学、计算机视觉等领域。它支持多种常见的旋转表示方式，如欧拉角、四元数和旋转矩阵，并提供了转换函数，可以在这些表示之间进行转换。

输入“sudo pip3 install transforms3d”即可进行安装。

```
ubuntu@raspberrypi:~$ sudo pip3 install transforms3d
```

注意：安装 transforms3d 库请先安装 5.pip 软件包安装。

7.turtle-tf2-py 和 tf2-tools 库安装

turtle-tf2-py 和 tf2-tools 是 ROS（机器人操作系统）的两个软件包，用于在 Python 中使用 TF2（Transform Library）进行坐标变换的相关功能。

turtle-tf2-py 是一个 ROS 软件包，它为 Python 提供了一个轻量级的 TF2 客户端库，可以方便地在 ROS 中进行坐标变换操作。通过使用 ros-humble-turtle-tf2-py，你可以监听坐标变换、查询坐标变换、执行坐标变换等操作。

tf2-tools 是另一个 ROS 软件包，它提供了一些与 TF2 相关的实用工具。其中包含了一些常用的功能，如坐标变换的插值、坐标变换的发布和监听、坐标系的可视化等。

输入“**sudo apt install ros-humble-turtle-tf2-py ros-humble-tf2-tools**”即可进行安装。（humble 为 ROS2 的版本号）

```
ubuntu@raspberrypi:~$ sudo apt install ros-humble-turtle-tf2-py ros-humb
le-tf2-tools
```

8.Gazebo 安装

Gazebo 是一个功能强大的开源三维机器人模拟软件，它可以帮助我们在计算机上快速地创建和测试各种真实世界场景。Gazebo 内置了精确的物理引擎来模拟各种动态对象在场景中的交互，如地心引力、摩擦力等影响。它同时提供出色的三维可视化效果，我们可以通过视觉客户端清晰地看到模拟过程。最重要的是，Gazebo 与 ROS 完美集成，我们可以直接在其中开发和测试 ROS 节点，将仿真代码轻松移植到实体机器人上。

输入“**sudo apt-get install ros-humble-ros-gz**”即可进行安装。（humble 为 ROS2 的版本号）

```
ubuntu@raspberrypi:~$ sudo apt-get install ros-humble-ros-gz
```

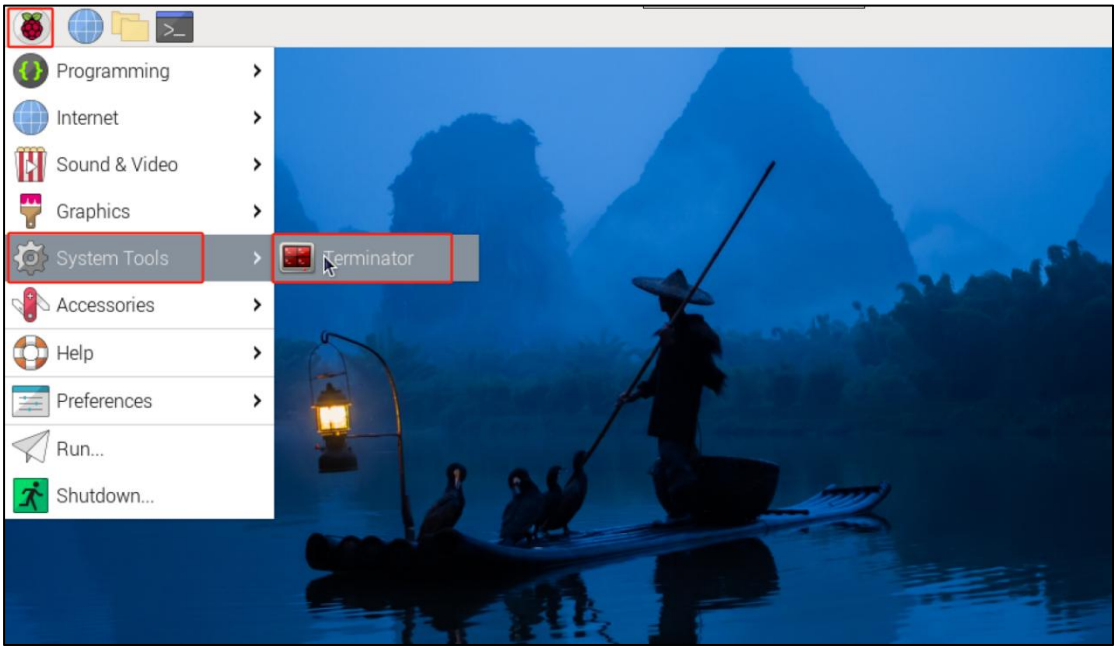
第 5 课 ROS2 工作空间

1.工作空间简介

在 ROS 机器人开发中，我们针对机器人某些功能进行开发时，各种编写的代码、参数、脚本等文件，需要放置在某一个文件夹里进行管理，这个文件夹在 ROS 系统中就叫做工作空间。所以工作空间是一个存放项目开发相关文件的文件夹，也是开发过程中存放所有资料的大本营。

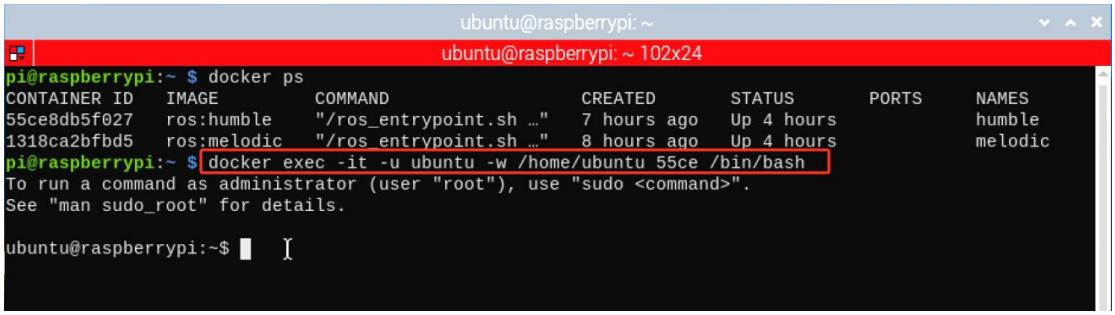
2.创建和编译工作空间

1） 点击左上角的 logo，选择 System Tools →Terminator。

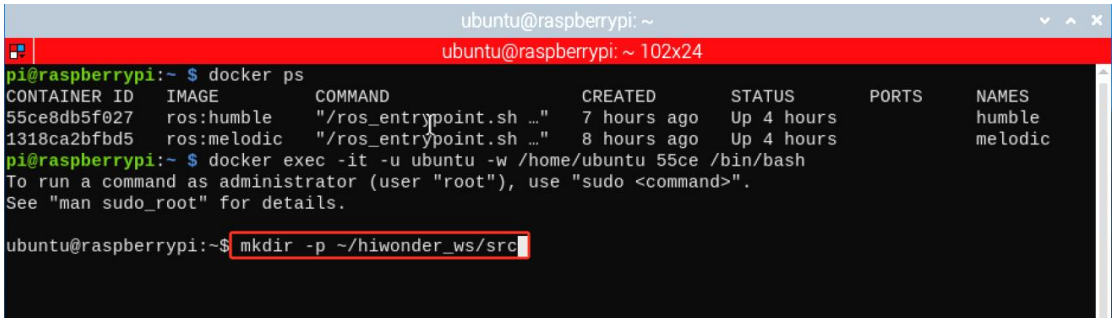


2） 在命令行终端输入“**docker ps**”，显示当前正在运行的容器。

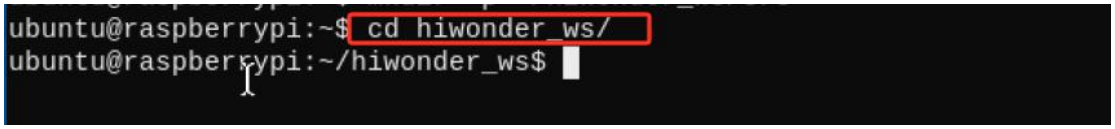
3） 输入“**docker exec -it -u ubuntu -w /home/ubuntu 55ce /bin/bash**”（容器的 ID 可以简写，只要是该容器唯一标识即可）指令，进行正在装有 ros2 的容器中。



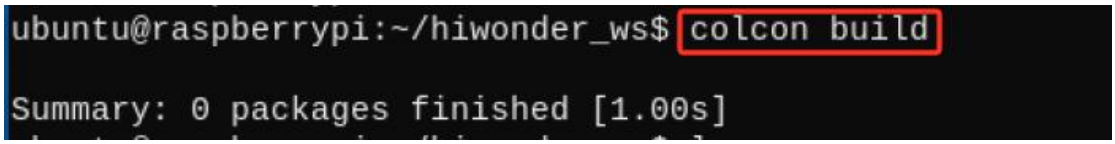
4） 输入“**mkdir -p ~/hiwonder_ws/src**”指令，创建一个叫“**hiwonder_ws**”的工作空间。



5） 输入“**cd hiwonder_ws**”指令，切换到“**hiwonder_ws**”的工作空间。



6） 输入“**colcon build**”指令，对工作空间进行编译。



7） 输入“**source ~/hiwonder_ws/install/setup.bash**”指令，在 ROS2 中加载工作空间环境。

8） 每一次打开都执行 7）步骤加载工作空间，可以在终端输入

“**echo “source ~/hiwonder_ws/install/setup.bash” >> ~/.bashrc**”指令，将该指令写入.bashrc 文件中。

```
ubuntu@raspberrypi:~$ echo "source ~/hiwonder_ws/install/setup.bash" >> ~/.bashrc
ubuntu@raspberrypi:~$
```

9） 然后输入“source ~/.bashrc”指令，让**.bashrc** 文件生效。这样就不需要每一次加载工作空间环境。

```
ubuntu@raspberrypi:~$ source ~/.bashrc
ubuntu@raspberrypi:~$
```

3.工作空间介绍

编译完成后，可以输入“**tree -L 1**”指令，查看工作空间的根目录。

```
ubuntu@raspberrypi:~/hiwonder_ws$ tree -L 1
.
├── build
├── install
├── log
└── src
```

ROS 系统中一个典型的工作空间结构如上所示，这个 hiwonder_ws 就是工作空间的根目录，里边会有四个子目录，或者叫做四个子空间。

名字	含义	详细说明
build	编译空间	保存编译过程中产生的中间文件
install	安装空间	放置编译得到的可执行文件和脚本
log	日志空间	编译和运行过程中，保存各种警告、错误、信息等日志。
src	代码空间	未来编写的代码、脚本，都需要人为地放置到这里。

总体来讲，这四个空间的文件夹，我们绝大部分操作都是在 src 中进行的，编译成功

后，就会执行 install 里边的结果，build 和 log 两个文件夹用的很少。

此外，工作空间的名称是可以自定义，且数量也不是唯一的，比如：

工作空间 1：hiwonder_ws_a，用于 A 机器人的开发
工作空间 2：hiwonder_ws_b，用于 B 机器人的开发
工作空间 3：hiwonder_ws_c，用于 C 机器人的开发

以上情况是完成允许的，就像是在集成开发环境中创建了多个新工程一样，都是并列存在的关系。

第 6 课 ROS2 功能包

1.功能包简介

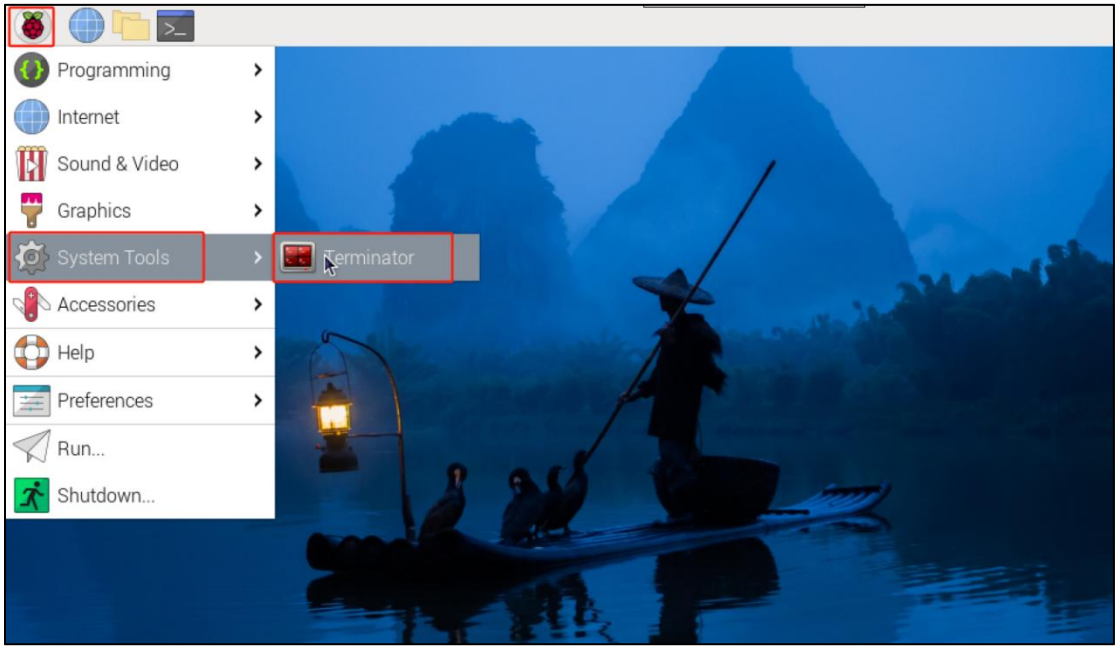
每个机器人可能有很多功能，比如移动控制、视觉感知、自主导航等，如果我们把这些功能的源码都放在一起当然也是可以的，但是当我们想把其中某些功能分享给别人时，就会发现代码都混合到了一起，很难拆分出来。

功能就是这个原理，我们把不同功能的代码划分到不同的功能包中，尽量降低他们之间的耦合关系，当需要在 ROS 社区中分享给别人的时候，只需要说明这个功能包该如何使用，别人很快就可以用起来了。

所以功能包的机制，是提高 ROS 中软件复用率的重要方法之一。

2.创建和编译功能包

1） 点击左上角的 logo，选择 System Tools →Terminator。



2） 输入“**cd hiwonder_ws/src/**”指令，切换到 hiwonder_ws 工作空间的 src 文件夹中。

```
ubuntu@raspberrypi:~$ cd hiwonder_ws/src/
ubuntu@raspberrypi:~/hiwonder_ws/src$
```

3） 输入“**ros2 pkg create hello_world_demo --build-type ament_python --dependencies rclpy --node-name hello_world**”并按下回车，创建一个名为“**hello_world_demo**”的功能包，添加依赖关系 rclpy，并生成一个名为“**hello_world**”的执行程序。

```
ubuntu@raspberrypi:~/hiwonder_ws/src$ ros2 pkg create hello_world_demo --build-type ament_python --dependencies rclpy --node-name hello_world
```

4） 输入“**cd ~/hiwonder_ws**”指令，切换到工作空间的根目录。

```
ubuntu@raspberrypi:~/hiwonder_ws/src$ cd ~/hiwonder_ws
ubuntu@raspberrypi:~/hiwonder_ws$
```

5） 输入“**colcon build**”指令，对工作空间的功能包进行编译。

```
ubuntu@raspberrypi:~/hiwonder_ws$ colcon build
```

6） 编译完成后，输入进入功能包所在目录的指令“**cd src/hello_world_demo/**”，并按下回车，验证功能包是否创建成功。

```
ubuntu@raspberrypi:~/hiwonder_ws$ cd src/hello_world_demo/
ubuntu@raspberrypi:~/hiwonder_ws/src/hello_world_demo$
```

3.功能包介绍

编译完成后，可以输入“**tree -L 1**”指令，查看工作空间的根目录。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/hello_world_demo$ tree -L 1
.
├── hello_world_demo
├── package.xml
├── resource
├── setup.cfg
├── setup.py
└── test

3 directories, 3 files
ubuntu@raspberrypi:~/hiwonder_ws/src/hello_world_demo$
```

ROS 系统中一个典型的功能包结构如上所示，这个 hello_world_demo 就是功能包的根目录，里边会有 6 个文件或子目录。

名字	详细说明
hello_world_demo	功能包同名目录：Python 源文件目录
package.xml	包的信息，比如：包名、版本、作者、依赖性
resource	资源目录
setup.cfg	功能包基本配置文件
setup.py	与 C++功能包的 CMakeLists.txt 类似
test	存储测试相关文件

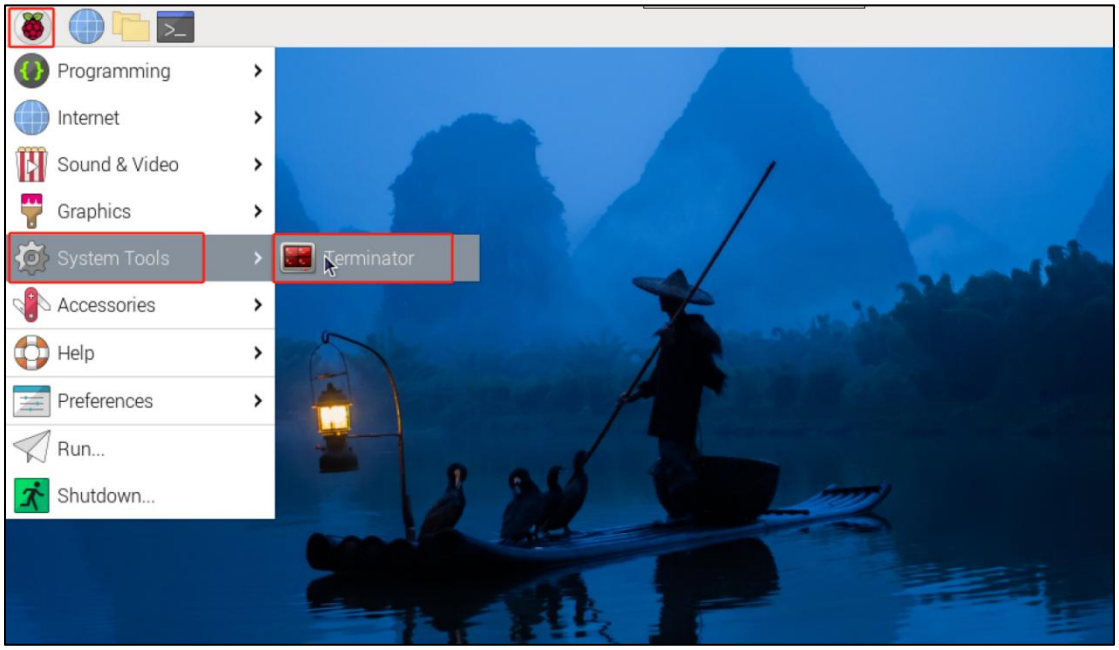
第 7 课 ROS2 节点

1.节点简介

在通信时，不论采用何种方式，通信对象的构建都依赖于节点（Node），在 ROS2 中，一般情况下每个节点都对应某一单一的功能模块（例如：雷达驱动节点可能负责发布雷达消息，摄像头驱动节点可能负责发布图像消息）。一个完整的机器人系统可能由许多协同工作的节点组成，ROS2 中的单个可执行文件（C++程序或 python 程序）可以包含一个或多个节点。

2.创建节点

1） 点击左上角的 logo，选择 System Tools →Terminator。



2） 输入指令“**cd hiwonder_ws/src/hello_world_demo/hello_world_demo**”，回车，切换到“**hello_world_demo**”功能包的路径下。

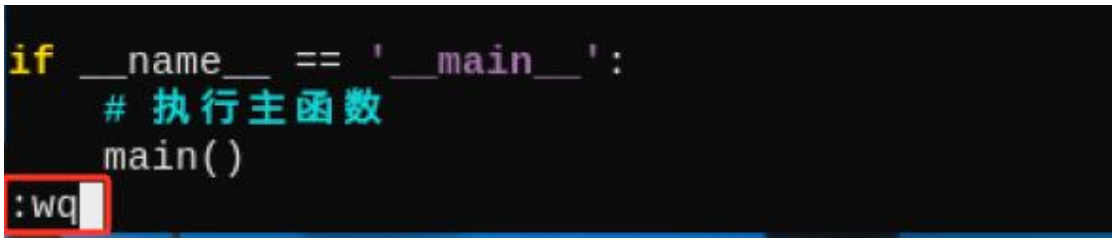
```
ubuntu@raspberrypi:~$ cd hiwonder_ws/src/hello_world_demo/hello_world_demo
ubuntu@raspberrypi:~/hiwonder_ws/src/hello_world_demo/hello_world_demo$
```

3） 输入指令“**vim hello_world.py**”编辑程序，复制下面程序。如需修改，再按下“**i**”即可修改。修改完成，按下“**Esc**”，输入“**：wq**”保存并退出。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/hello_world_demo/hello_world_demo$ vim hell
o_world.py
```

```
import rclpy

from rclpy.node import Node

import time



class HelloWorldNode(Node):

    def __init__(self):

        # 调用基类 Node 的构造函数，设置节点名称

        super().__init__('hello_world_demo')



    def run(self):

        # 在 ROS2 系统正常运行的情况下执行循环

        while rclpy.ok():

            # 打印 "Hello World" 到节点的日志

            self.get_logger().info('Hello World')

            # 休眠 0.5 秒，控制循环时间

            time.sleep(0.5)



def main(args=None):

    # 初始化 ROS2 Python 接口

    rclpy.init(args=args)

    # 创建 HelloWorldNode 实例
```

```
    node = HelloWorldNode()



    try:

        # 运行节点的主循环

        node.run()

    except KeyboardInterrupt:

        pass

    finally:

        # 销毁节点对象

        node.destroy_node()

        # 关闭 ROS2 Python 接口

        rclpy.shutdown()



if __name__ == '__main__':

    # 执行主函数

    main()
```



4） 输入指令“**chmod +x hello_world.py**”，并按下回车，为保存的 hello_world.py 赋予可执行权限。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/hello_world_demo/hello_world_demo$ chmod +x
 hello_world.py
ubuntu@raspberrypi:~/hiwonder_ws/src/hello_world_demo/hello_world_demo$
```

3.编译运行

1） 赋予可执行权限后，输入“**cd ~/hiwonder_ws/**”指令，切换到工作空间的目录。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/hello_world_demo$ cd ~/hiwonder_ws/
ubuntu@raspberrypi:~/hiwonder_ws$
```

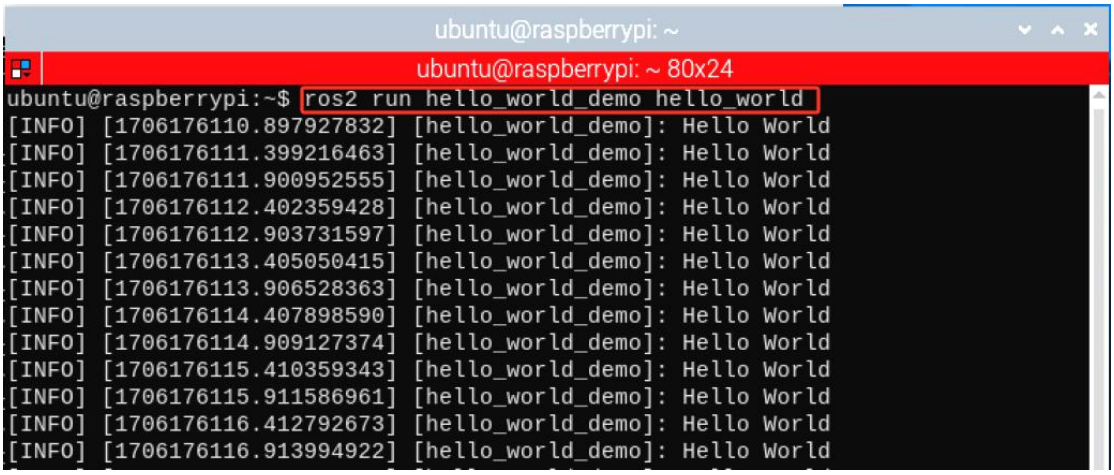
2） 输入“**colcon build**”指令，并按下回车，编译工作空间中的功能包。

```
ubuntu@raspberrypi:~/hiwonder_ws$ colcon build
```

3） 再输入指令“**source ./install/setup.bash**”，并按下回车，让环境变量生效。

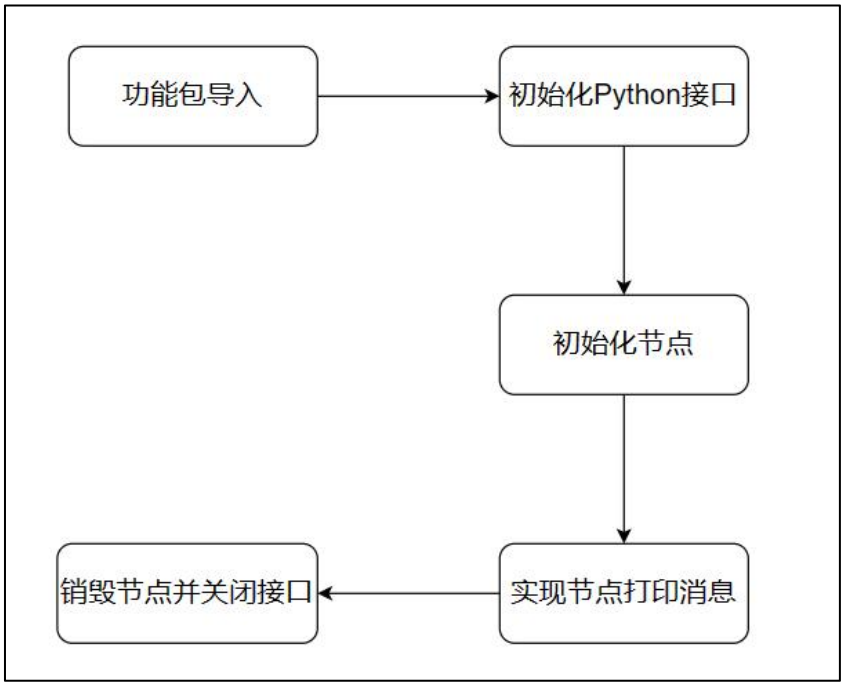
```
ubuntu@raspberrypi:~/hiwonder_ws$ source ./install/setup.bash
```

4） 输入指令“**ros2 run hello_world_demo hello_world**”，并按下回车，启动 hello_world 节点。



4.程序分析

根据实现的效果，梳理程序的过程逻辑，如下图所示：



创建了一个名为 HelloWorldNode 的节点，并在节点的主循环中以 0.5 秒的间隔打印"Hello World"到节点的日志中。在程序运行时，首先会初始化 ROS2 Python 接口，然后创建节点实例并运行其主循环。当程序被中断时，将销毁节点对象并关闭 ROS2 Python 接口。

◆ 主函数

```
def main(args=None):
    # 初始化 ROS2 Python 接口
    rclpy.init(args=args)
    # 创建 HelloWorldNode 实例
    node = HelloWorldNode()

    try:
        # 运行节点的主循环
        node.run()
    except KeyboardInterrupt:
        pass
    finally:
        # 销毁节点对象
        node.destroy_node()
        # 关闭 ROS2 Python 接口
        rclpy.shutdown()
```

首先调用 rclpy.init()对 ROS2 Python 接口进行初始化，然后实例化 HelloWorldNode，然后执行 node.run()函数。

◆ HelloWorldNode 类

```
class HelloWorldNode(Node):
    def __init__(self):
        # 调用基类 Node 的构造函数，设置节点名称
        super().__init__('hello_world_demo')

    def run(self):
        # 在 ROS2 系统正常运行的情况下执行循环
        while rclpy.ok():
            # 打印 "Hello World" 到节点的日志
            self.get_logger().info('Hello World')
            # 休眠 0.5 秒，控制循环时间
            time.sleep(0.5)
```

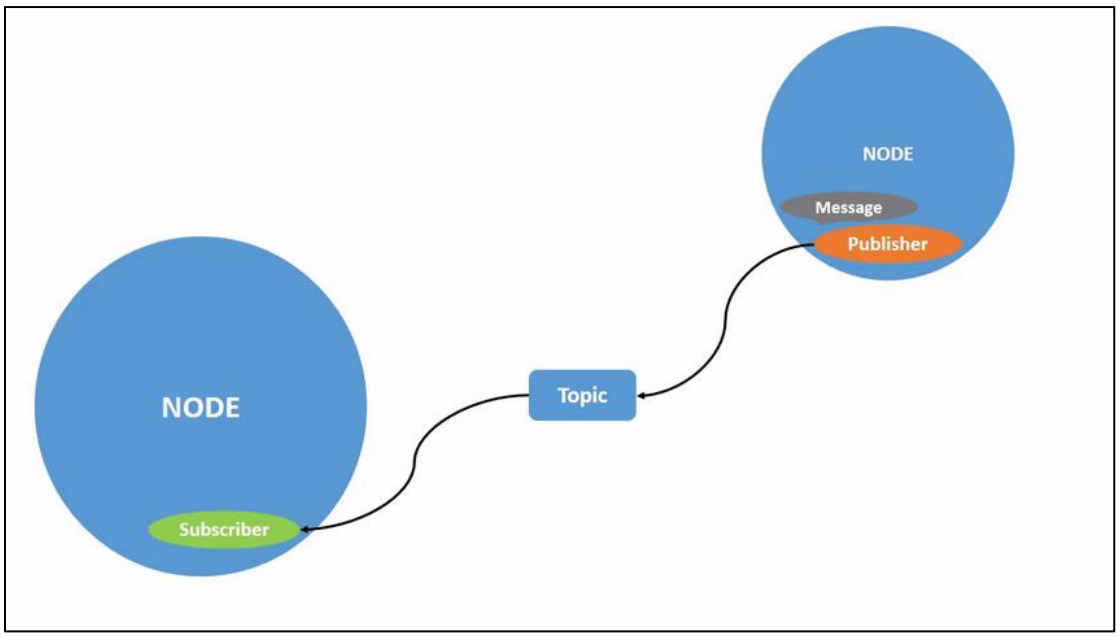
首先创建了一个名为 HelloWorldNode 的节点，run()函数里面在在节点的主循环中以 0.5 秒的间隔打印"Hello World"到节点的日志中。

第 8 课 ROS2 话题

1.话题通信简介

话题通信是 ROS2 视频频率最高的一种通信方式，有发布者发布指定话题的数据，订阅者只要订阅了该话题的数据，就可以接收到数据。

话题通信是基于发布/订阅模型，如图：

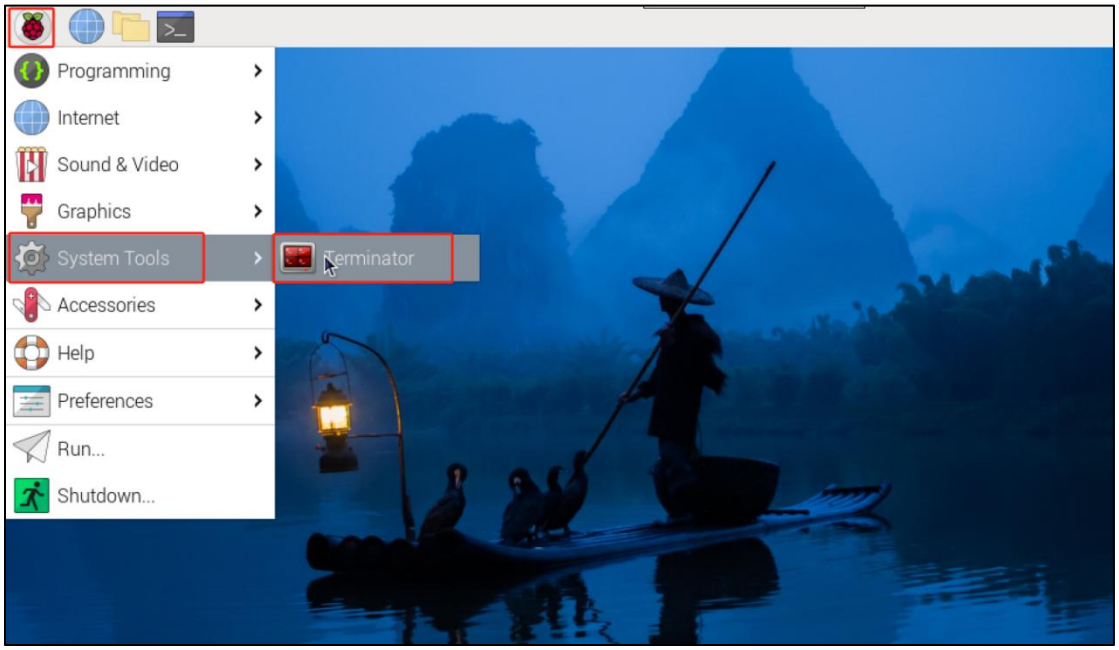


话题数据传输的特性是从一个节点到另外一个节点，发送数据的对象称之为发布者，接收数据的对象称之为订阅者，每一个话题都需要一个名字，传输的数据也需要有固定的数据类型。

2.创建话题

2.1 创建发布者

1） 点击左上角的 logo，选择 System Tools →Terminator。



2） 输入“**cd hiwonder_ws/src/**”指令，切换到 hiwonder_ws 工作空间的 src 文件夹中。

```
ubuntu@raspberrypi:~$ cd hiwonder_ws/src/
ubuntu@raspberrypi:~/hiwonder_ws/src$
```

3） 输入“**ros2 pkg create topic_demo --build-type ament_python --dependencies rclpy**”并按下回车，创建一个名为“**topic_demo**”的功能包，添加依赖关系 rclpy。

```
ubuntu@raspberrypi:~/hiwonder_ws/src$ ros2 pkg create topic_demo --build-type am
ent_python --dependencies rclpy
```

4） 输入“**cd topic_demo/topic_demo/**”指令，切换到 topic_demo 功能包下。

```
ubuntu@raspberrypi:~/hiwonder_ws/src$ cd topic_demo/topic_demo/
ubuntu@raspberrypi:~/hiwonder_ws/src/topic_demo/topic_demo$
```

5） 输入指令“**vim topic_pub.py**”编辑程序，复制下面程序。如需修改，再按下“**i**”即可修改。修改完成，按下“**Esc**”，输入“**：wq**”保存并退出。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/topic_demo/topic_demo$ vim topic_pub.py
```

```
import rclpy

from rclpy.node import Node
```

```
from std_msgs.msg import String


# 定义一个继承自 Node 的 MinimalPublisher 类
class MinimalPublisher(Node):


    # 类的初始化方法
    def __init__(self):
        # 调用 Node 类的初始化方法，设置节点名称为'minimal_publisher'
        super().__init__('minimal_publisher')


        # 创建一个发布者，发布 String 类型消息到'topic'，队列大小为 10
        self.publisher_ = self.create_publisher(String, 'topic', 10)


        # 创建一个定时器，每 0.5 秒触发一次 timer_callback 方法
        timer_period = 0.5  # 秒
        self.timer = self.create_timer(timer_period, self.timer_callbac
k)


        # 计数器，用于生成消息中的数字
        self.i = 0
```

```
    # 定义定时器回调函数
    def timer_callback(self):
        # 创建一个 String 类型的消息
        msg = String()

        # 设置消息数据为'Hello World: 数字'
        msg.data = 'Hello World: %d' % self.i

        # 发布消息
        self.publisher_.publish(msg)

        # 计数器自增
        self.i += 1


# 主函数
def main(args=None):
    # 初始化 ROS 2 节点
    rclpy.init(args=args)

    # 创建 MinimalPublisher 对象
    minimal_publisher = MinimalPublisher()
```

```
    # 进入 ROS 2 节点的事件循环
    rclpy.spin(minimal_publisher)


    # 销毁节点对象
    minimal_publisher.destroy_node()


    # 关闭 ROS 2 节点
    rclpy.shutdown()



# 如果该脚本是主程序，则执行 main 函数
if __name__ == '__main__':
    main()
```

6） 输入指令“**chmod +x topic_pub.py**”，并按下回车，为保存的 topic_pub.py 赋予可执行权限。

2.2 创建订阅者

1） 输入指令“**vim topic_sub.py**”编辑程序，复制下面程序。如需修改，再按下“**i**”

即可修改。修改完成，按下“**Esc**”，输入“**：wq**”保存并退出。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/topic_demo/topic_demo$ vim topic_sub.py
```

```
import rclpy

from rclpy.node import Node



from std_msgs.msg import String



# 定义一个继承自 Node 的 MinimalSubscriber 类

class MinimalSubscriber(Node):



    # 类的初始化方法

    def __init__(self):

        # 调用 Node 类的初始化方法

        super().__init__('minimal_subscriber')



        # 创建一个订阅者，订阅名为'topic'的 String 类型消息，回调函数为 listene
r_callback，队列大小为 10

        self.subscription = self.create_subscription(String, 'topic', sel
f.listener_callback, 10)



    # 定义消息回调函数

    def listener_callback(self, msg):
```

```
        # 获取日志记录器并打印收到的消息

        self.get_logger().info('I heard: "%s"' % msg.data)



# 主函数

def main(args=None):

    # 初始化 ROS 2 节点

    rclpy.init(args=args)



    # 创建 MinimalSubscriber 对象

    minimal_subscriber = MinimalSubscriber()



    # 进入 ROS 2 节点的事件循环

    rclpy.spin(minimal_subscriber)



    # 销毁节点对象

    minimal_subscriber.destroy_node()



    # 关闭 ROS 2 节点

    rclpy.shutdown()



# 如果该脚本是主程序，则执行 main 函数

if __name__ == '__main__':
```

```
    main()
```

```
# 如果该脚本是主程序，则执行main函数
if __name__ == '__main__':
    main()
:wq
```

2） 输入指令“**chmod +x topic_sub.py**”，并按下回车，为保存的 topic_sub.py 赋予可执行权限。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/topic_demo/topic_demo$ chmod +x topic_sub.p
y
```

3.setup.py 文件设置

setup.py 文件是定义一个 ROS2 包的远数据和构建配置，提供了包的元数据、依赖关系、构建配置和安装逻辑等信息，可以帮助开发者正确构建、安装和使用 ROS2 包。需要将 **topic_pub.py** 和 **topic_sub.py** 的程序入口写到 setup.py 文件中。

1） 输入“**cd ..**”指令，切换到上一级目录。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/topic_demo/topic_demo$ cd ..
ubuntu@raspberrypi:~/hiwonder_ws/src/topic_demo$
```

2） 输入“**vim setup.py**”指令，并按下回车，打开 setup.py 文件。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/topic_demo$ vim setup.py
```

3） 按下“i”键进入可编辑模式，输入以下代码到相应的位置上。

```
'topic_pub = topic_demo.topic_pub:main',

'topic_sub = topic_demo.topic_sub:main'
```

```
    maintainer='ubuntu',
    maintainer_email='ubuntu@todo.todo',
    description='TODO: Package description',
    license='TODO: License declaration',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'topic_pub = topic_demo.topic_pub:main',
            'topic_sub = topic_demo.topic_sub:main'
        ],
    },
)
~
~
-- INSERT --                                          24,52         Bot
```

4） 然后输入“:wq”，保存并退出文件。

```
            'topic_pub = topic_demo.topic_pub:main',
            'topic_sub = topic_demo.topic_sub:main'
        ],
    },
)
:wq
```

4.编译运行

1） 赋予可执行权限后，输入“**cd ~/hiwonder_ws/**”指令，切换到工作空间的目录。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/hello_world_demo$ cd ~/hiwonder_ws/
ubuntu@raspberrypi:~/hiwonder_ws$
```

2） 输入“**colcon build**”指令，并按下回车，编译工作空间中的功能包。

```
ubuntu@raspberrypi:~/hiwonder_ws$ colcon build
```

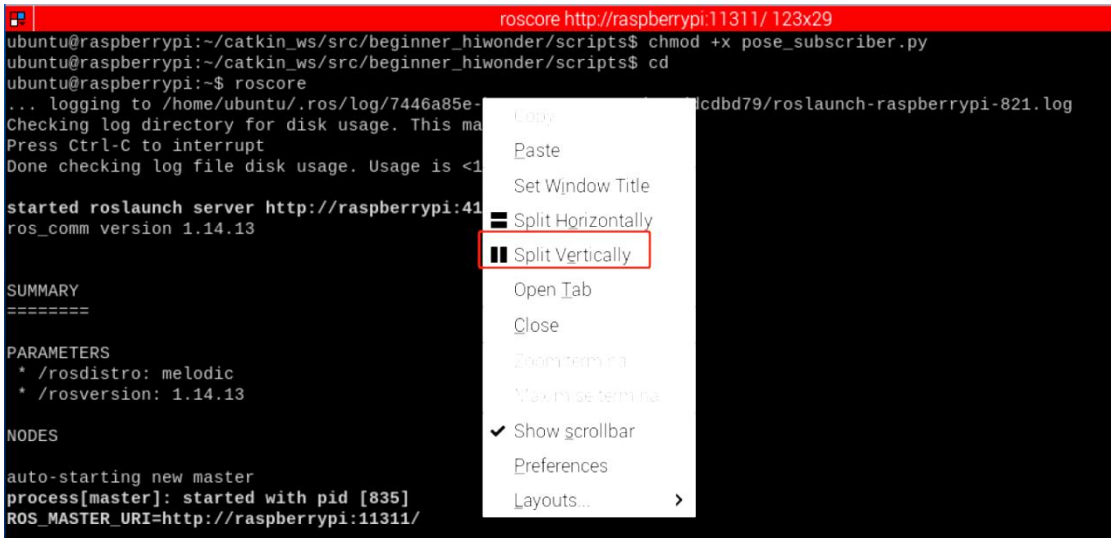
3） 再输入指令“**source ./install/setup.bash**”，并按下回车，让环境变量生效。

```
ubuntu@raspberrypi:~/hiwonder_ws$ source ./install/setup.bash
```

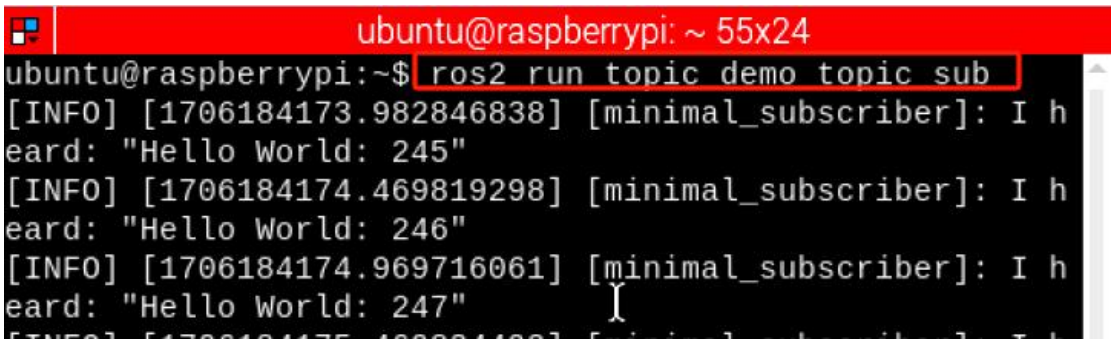
4） 输入指令“**ros2 run topic_demo topic_pub**”，并按下回车，启动 topic_pub 话题发布节点。

```
ubuntu@raspberrypi:~$ ros2 run topic_demo topic_pub
```

5） 鼠标右键，选择“**Split Vertically**”点击，新建一个新的终端窗口。



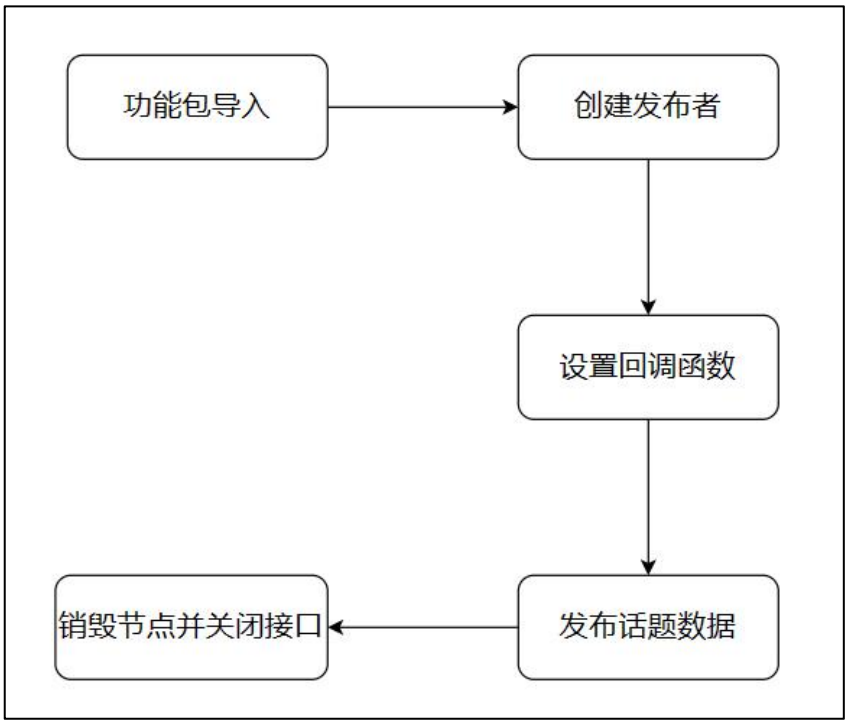
6） 输入指令“**ros2 run topic_demo topic_sub**”，并按下回车，启动 topic_sub 话题发布节点。



5.程序分析

5.1 话题发布

根据实现的效果，梳理程序的过程逻辑，如下图所示：



创建了一个名为 minimal_publisher 的发布者，该发布者每 0.5 秒发布一个带有递增数字的消息到 topic 话题。在程序运行时，首先会初始化 ROS2 节点，然后创建 MinimalPublisher 对象并进入 ROS 2 节点的事件循环。当程序被中断时，会销毁节点对象并关闭 ROS 2 节点。

◆ 主函数

```
# 主函数
def main(args=None):
    # 初始化ROS 2节点
    rclpy.init(args=args)

    # 创建MinimalPublisher对象
    minimal_publisher = MinimalPublisher()

    # 进入ROS 2节点的事件循环
    rclpy.spin(minimal_publisher)

    # 销毁节点对象
    minimal_publisher.destroy_node()

    # 关闭ROS 2节点
    rclpy.shutdown()
```

首先调用 rclpy.init()对 ROS2 Python 接口进行初始化，然后实例化 MinimalPublisher()，然后在 ROS2 节点的事件循环执行 minimal_publisher。

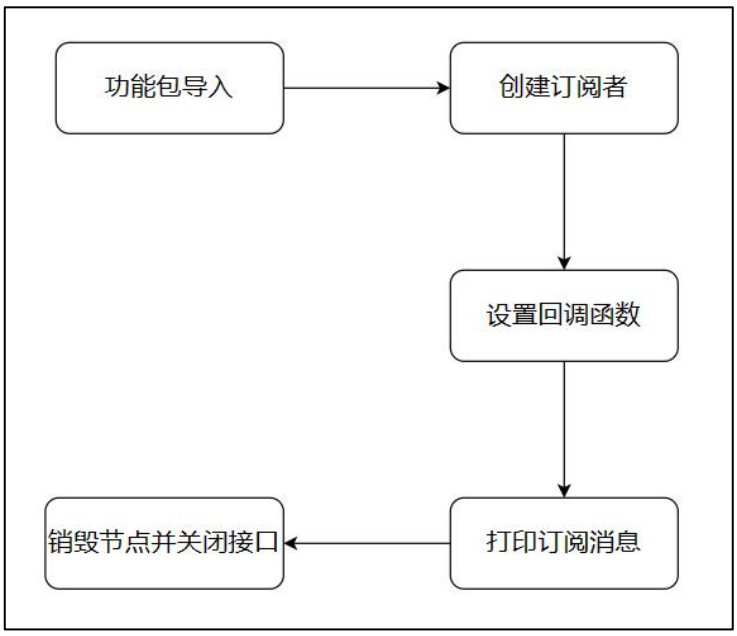
◆ MinimalPublisher 类

```
# 定义一个继承自Node的MinimalPublisher类
class MinimalPublisher(Node):

    # 类的初始化方法
    def __init__(self):
        # 调用Node类的初始化方法，设置节点名称为'minimal_publisher'
        super().__init__('minimal_publisher')

        # 创建一个发布者，发布String类型消息到'topic'，队列大小为10
        self.publisher_ = self.create_publisher(String, 'topic', 10)

        # 创建一个定时器，每0.5秒触发一次timer_callback方法
        timer_period = 0.5  # 秒
        self.timer = self.create_timer(timer_period, self.timer_callback)

        # 计数器，用于生成消息中的数字
        self.i = 0

    # 定义定时器回调函数
    def timer_callback(self):
        # 创建一个String类型的消息
        msg = String()

        # 设置消息数据为'Hello World: 数字'
        msg.data = 'Hello World: %d' % self.i

        # 发布消息
        self.publisher_.publish(msg)

        # 计数器自增
        self.i += 1
                                                        10,1          14%
```

首先创建了一个名为 minimal_publisher 的节点，然后创建一个名为 publisher_的发布者，timier_callback()回调函数里面以 0.5 秒的间隔打印带有递增数字的消息到节点的日志中。

5.2 话题订阅

根据实现的效果，梳理程序的过程逻辑，如下图所示：



创建了一个名为 minimal_sublisher 的订阅者，该订阅者打印订阅的消息到日志记录器。在程序运行时，首先会初始化 ROS2 节点，然后创建 MinimalSublisher 对象并进入 ROS 2 节点的事件循环。当程序被中断时，会销毁节点对象并关闭 ROS 2 节点。

◆ 主函数

```python
# 主函数
def main(args=None):
    # 初始化ROS 2节点
    rclpy.init(args=args)

    # 创建MinimalSubscriber对象
    minimal_subscriber = MinimalSubscriber()

    # 进入ROS 2节点的事件循环
    rclpy.spin(minimal_subscriber)

    # 销毁节点对象
    minimal_subscriber.destroy_node()

    # 关闭ROS 2节点
    rclpy.shutdown()
```

首先调用 rclpy.init()对 ROS2 Python 接口进行初始化，然后实例化 MinimalSublisher()，然后在 ROS2 节点的事件循环执行 minimal_sublisher。

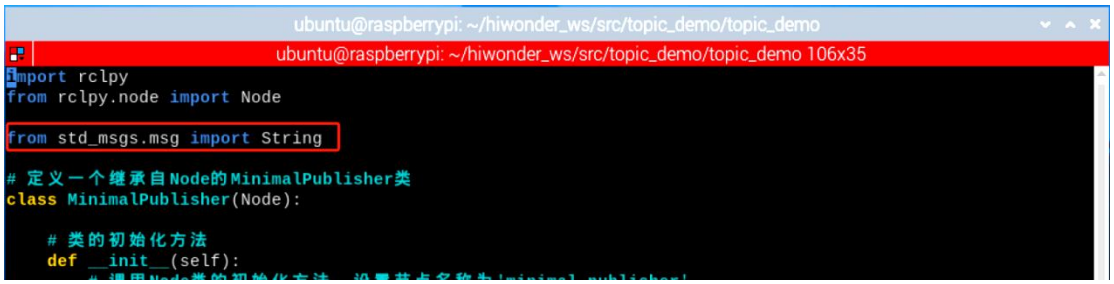
◆ MinimalSublisher 类

```python
# 定义一个继承自Node的MinimalSubscriber类
class MinimalSubscriber(Node):

    # 类的初始化方法
    def __init__(self):
        # 调用Node类的初始化方法
        super().__init__('minimal_subscriber')

        # 创建一个订阅者，订阅名为'topic'的String类型消息，回调函数为listener_callback，队列大小为10
        self.subscription = self.create_subscription(String, 'topic', self.listener_callback, 10)

    # 定义消息回调函数
    def listener_callback(self, msg):
        # 获取日志记录器并打印收到的消息
        self.get_logger().info('I heard: "%s"' % msg.data)
```

首先创建了一个名为 minimal_sublisher 的节点，然后创建一个名为 subscription 的订阅者，listener_callback()回调函数里面打印订阅到的消息内容到日志记录器上。

6.自定义接口

topic_pub.py 和 topic_pub.py 都使用了 ROS2 官方的接口。



虽然使用预定义的接口定义是一种很好的做法，但有时可能还需要定义自己的消息和服务。接下来演示如何创建自定义接口定义的方法。

1） 输入“**cd ~/hiwonder_ws/src/**”指令，切换到 hiwonder_ws 工作空间的 src 文件夹中。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/topic_demo$ cd ~/hiwonder_ws/src/
ubuntu@raspberrypi:~/hiwonder_ws/src$
```

2） 输入“**ros2 pkg create demo_interfaces --build-type ament_cmake --dependencies rclcpp** ”并按下回车，创建一个名为“**demo_interfaces**”的功能包。

```
ubuntu@raspberrypi:~/hiwonder_ws/src$ ros2 pkg create demo_interfaces --build-type ament_cmake
--dependencies rclcpp
```

3） 输入“**cd demo_interfaces**”指令，进入自定义接口的功能包。

```
ubuntu@raspberrypi:~/hiwonder_ws/src$ cd demo_interfaces
ubuntu@raspberrypi:~/hiwonder_ws/src/demo_interfaces$
```

4） 输入“**mkdir msg**”指令，创建文件夹。

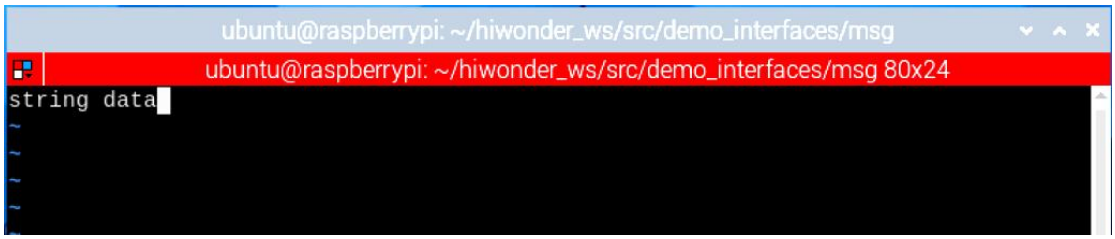
```
ubuntu@raspberrypi:~/hiwonder_ws/src/demo_interfaces$ mkdir msg
```

5） 输入“**cd msg**”指令，进入 msg 文件夹。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/demo_interfaces$ cd msg
ubuntu@raspberrypi:~/hiwonder_ws/src/demo_interfaces/msg$
```

6） 输入指令“**vim String.msg**”编辑程序，输入“**string data**”。如需修改，再按下“**i**”即可修改。修改完成，按下“**Esc**”，输入“**：wq**”保存并退出。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/demo_interfaces/msg$ vim String.msg
```



7） 先输入“cd ..”返回上一级文件夹，再输入“vim CMakeLists.txt”，复制下面程序，在下图所示位置添加代码。如需修改，再按下“i”即可修改。修改完成，按下“Esc”，输入“：wq”保存并退出。

```
find_package(rosidl_default_generators REQUIRED)

rosidl_generate_interfaces( ${PROJECT_NAME}

  "msg/String.msg"

)
```

```
# find dependencies
find_package(ament_cmake REQUIRED)
# uncomment the following section in order to fill in
# further dependencies manually.
# find_package(<dependency> REQUIRED)

find_package(rosidl_default_generators REQUIRED)
rosidl_generate_interfaces( ${PROJECT_NAME}
  "msg/String.msg"
)

if(BUILD_TESTING)
  find_package(ament_lint_auto REQUIRED)
  # the following line skips the linter which checks for copyrights
  # comment the line when a copyright and license is added to all source files
  set(ament_cmake_copyright_FOUND TRUE)
```

8） 再输入“**vim package.xml**”，复制下面程序，在下图所示位置添加代码。如需修改，再按下“i”即可修改。修改完成，按下“Esc”，输入“：wq”保存并退出。

```
<build_depend>rosidl_default_generators</build_depend>

<exec_depend>rosidl_default_runtime</exec_depend>

<member_of_group>rosidl_interface_packages</member_of_group>
```

```
ubuntu@raspberrypi: ~/hiwonder_ws/src/demo_interfaces
ubuntu@raspberrypi: ~/hiwonder_ws/src/demo_interfaces 80x24
<package format="3">
  <name>demo_interfaces</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="ubuntu@todo.todo">ubuntu</maintainer>
  <license>TODO: License declaration</license>

  <buildtool_depend>ament_cmake</buildtool_depend>

  <buildtool_depend>rosidl_default_generators</buildtool_depend>
  <depend>action_msgs</depend>

  <build_depend>rosidl_default_generators</build_depend>
  <exec_depend>rosidl_default_runtime</exec_depend>
  <member_of_group>rosidl_interface_packages</member_of_group>



  <export>
    <build_type>ament_cmake</build_type>
  </export>
</package>
-- INSERT --                                    25,1          Bot
```

9） 参考 4.编译运行的 1）、2）、3）步骤，对工作空间进行编译。

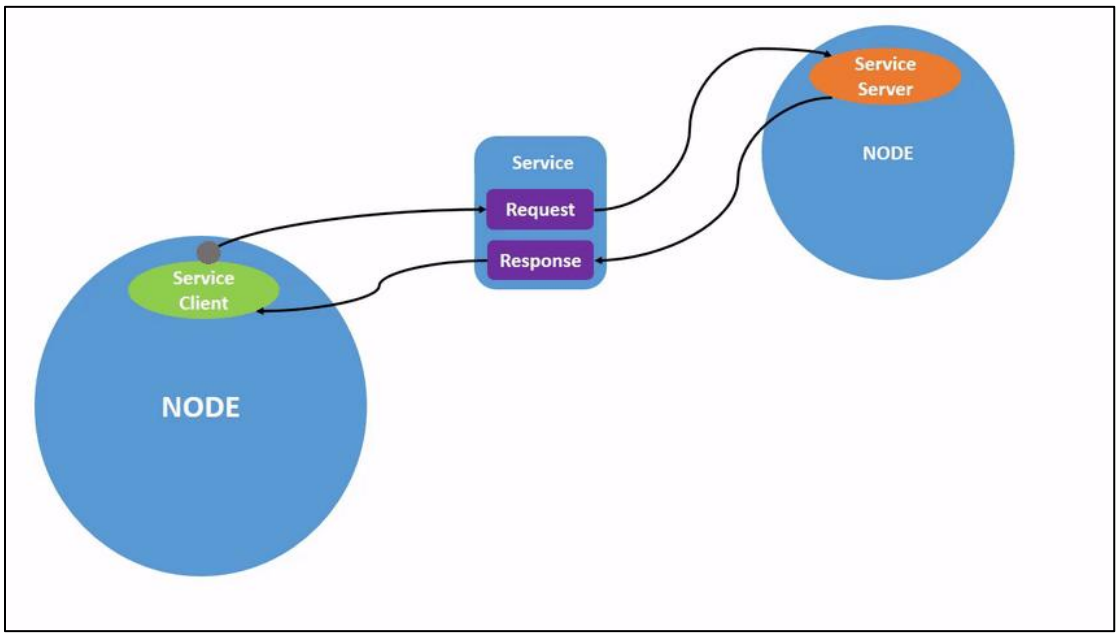
10）将红色方框内的代码修改成“**from demo_interfaces.msg import String**”语

句，即可完成自定义消息的使用，实现效果与原来的一致。

第 9 课 ROS2 服务说明

1.服务通讯简介

服务通讯是一种基于请求响应的通信模型，在通信双方中，客户端发送请求数据到服务端，服务端相应结果给客户端。

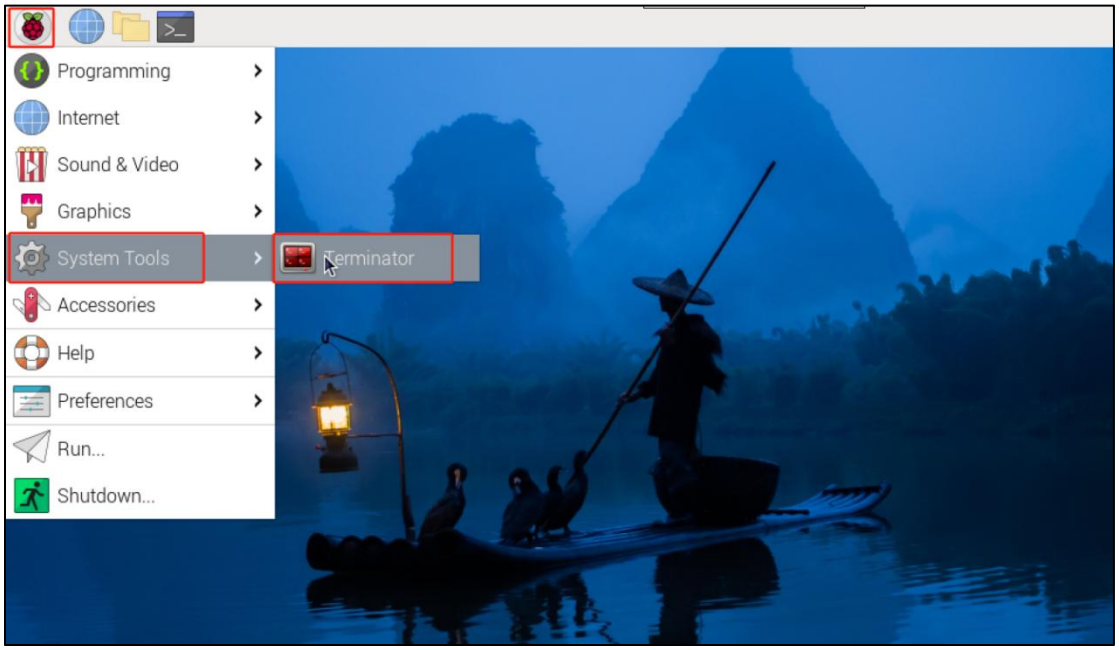


从服务的实现机制上来看，这种你问我答的形式叫做客户端/服务器模型，简称为 CS 模型，客户端在需要某些数据的时候，针对某个具体的服务，发送请求信息。服务器端收到请求之后，就会进行处理并反馈应答信息。

这种通信机制在生活中也很常见，比如我们经常浏览的各种网页，此时你的电脑浏览器就是客户端，通过域名或者各种操作，向网站服务器发送请求，服务器收到之后返回需要展现的页面数据。

2.创建接口

1) 点击左上角的 logo，选择 System Tools →Terminator。



2） 输入“**cd hiwonder_ws/src/demo_interfaces/**”指令，切换到 demo_interfaces 功能包中。

```
ubuntu@raspberrypi:~$ cd hiwonder_ws/src/demo_interfaces/
ubuntu@raspberrypi:~/hiwonder_ws/src/demo_interfaces$
```

3） 输入“**mkdir srv**”指令，创建 srv 文件夹。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/demo_interfaces$ mkdir srv
```

4） 输入“**cd srv**”指令，进入 srv 文件夹。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/demo_interfaces$ cd srv
ubuntu@raspberrypi:~/hiwonder_ws/src/demo_interfaces/srv$
```

5） 输入指令“**vim AddInts.srv**”编辑程序，输入以下代码。如需修改，再按下“**i**”即可修改。修改完成，按下“**Esc**”，输入“**：wq**”保存并退出。

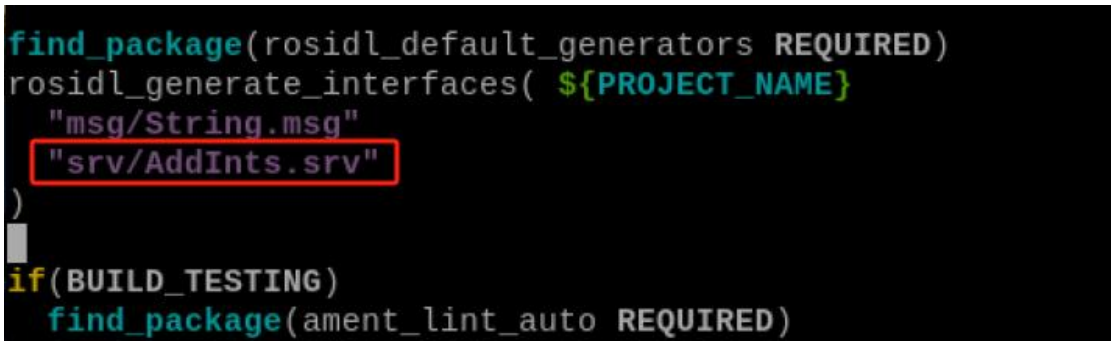
```
ubuntu@raspberrypi:~/hiwonder_ws/src/demo_interfaces/msg$ vim String.msg
```

```
int32 num1

int32 num2

---
```

```
int32 sum
```

6） 先输入“**cd ..**”返回上一级文件夹，再输入“**vim CMakeLists.txt**”，复制下面程序，在下图所示位置添加代码。如需修改，再按下“i”即可修改。修改完成，按下“**Esc**”，输入“**：wq**”保存并退出。

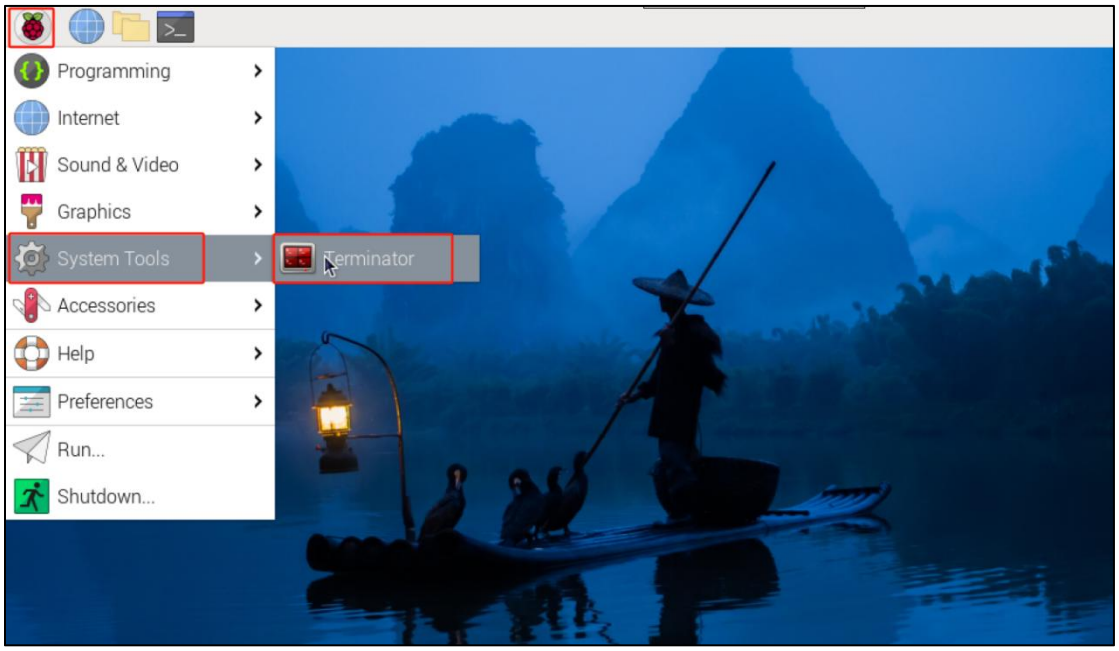
```
find_package(rosidl_default_generators REQUIRED)

rosidl_generate_interfaces( ${PROJECT_NAME}

  "msg/String.msg"

  "srv/AddInts.srv"

)
```



3.创建服务

3.1 创建服务端

1） 点击左上角的 logo，选择 System Tools →Terminator。



2） 输入“**cd hiwonder_ws/src/**”指令，切换到 hiwonder_ws 工作空间的 src 文件夹中。

```
ubuntu@raspberrypi:~$ cd hiwonder_ws/src/
ubuntu@raspberrypi:~/hiwonder_ws/src$
```

3） 输入“**ros2 pkg create service_demo --build-type ament_python --dependencies rclpy**”并按下回车，创建一个名为“**service_demo**”的功能包，添加依赖关系 rclpy。

```
ubuntu@raspberrypi:~/hiwonder_ws/src$ ros2 pkg create service_demo --build-type am
ent_python --dependencies rclpy
```

4） 输入“**cd service_demo/service_demo/**”指令，切换到 service_demo 功能包下。

```
ubuntu@raspberrypi:~/hiwonder_ws/src$ cd service_demo/service_demo/
ubuntu@raspberrypi:~/hiwonder_ws/src/service_demo/service_demo$
```

5） 输入指令“**vim service_server.py**”编辑程序，复制下面程序。如需修改，再按下“**i**”即可修改。修改完成，按下“**Esc**”，输入“**: wq**”保存并退出。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/service_demo/service_demo$ vim service_server
.py
```

```
import rclpy

from rclpy.node import Node

from demo_interfaces.srv import AddInts



# 定义一个继承自 Node 的 MinimalService 类

class MinimalService(Node):



    # 类的初始化方法

    def __init__(self):

        # 调用 Node 类的初始化方法，设置节点名称为'minimal_service'

        super().__init__('minimal_service')



        # 创建一个服务，提供 AddInts 类型的服务，服务名称为'add_two_ints'，回调函数为 add_two_ints_callback

        self.srv = self.create_service(AddInts, 'add_two_ints', self.add_two_ints_callback)



    # 定义服务回调函数

    def add_two_ints_callback(self, request, response):
```

```
        # 在日志中记录接收到的请求的 num1 和 num2

        self.get_logger().info('Incoming request\nnum1: %d num2: %d' % (r
equest.num1, request.num2))



        # 计算并设置响应的 sum 字段

        response.sum = request.num1 + request.num2



        # 返回响应

        return response



# 主函数

def main():

    # 初始化 ROS 2 节点

    rclpy.init()



    # 创建 MinimalService 对象

    minimal_service = MinimalService()



    # 进入 ROS 2 节点的事件循环

    rclpy.spin(minimal_service)



    # 关闭 ROS 2 节点
```

```
    rclpy.shutdown()



# 如果该脚本是主程序，则执行 main 函数

if __name__ == '__main__':

    main()
```

6） 输入指令“**chmod +x service_server.py**”，并按下回车，为保存的 service_server.py 赋予可执行权限。

3.2 创建客户端

1） 输入指令“**vim service_client.py**”编辑程序，复制下面程序。如需修改，再按下“**i**”即可修改。修改完成，按下“**Esc**”，输入“**：wq**”保存并退出。


```
import sys

import rclpy

from rclpy.node import Node

from rclpy.logging import get_logger

from demo_interfaces.srv import AddInts



```

```
# 定义一个继承自 Node 的 MinimalClient 类
class MinimalClient(Node):


    # 类的初始化方法
    def __init__(self):
        # 调用 Node 类的初始化方法，设置节点名称为'minimal_client'
        super().__init__('minimal_client')


        # 创建一个服务客户端，连接到名为'add_two_ints'的 AddInts 服务
        self.cli = self.create_client(AddInts, 'add_two_ints')


        # 等待服务连接，最多等待 1 秒
        while not self.cli.wait_for_service(timeout_sec=1.0):
            self.get_logger().info('service is connecting...')


    # 发送服务请求的方法
    def send_request(self):
        # 创建一个 AddInts 请求对象
        request = AddInts.Request()


        # 从命令行参数中获取两个整数，并设置到请求对象中
        request.num1 = int(sys.argv[1])
```

```
        request.num2 = int(sys.argv[2])


        # 异步调用服务，并获取一个 Future 对象
        self.future = self.cli.call_async(request)


# 主函数
def main():
    # 检查命令行参数是否包含两个整数
    if len(sys.argv) != 3:
        get_logger("rclpy").error("请提供两个整数值")
        return


    # 初始化 ROS 2 节点
    rclpy.init()


    # 创建 MinimalClient 对象
    minimal_client = MinimalClient()


    # 发送服务请求
    minimal_client.send_request()


    # 阻塞等待服务调用完成
```

```
    rclpy.spin_until_future_complete(minimal_client, minimal_client.futu
re)


    try:
        # 获取服务调用的响应
        response = minimal_client.future.result()

        # 打印响应结果
        minimal_client.get_logger().info("请求结果：sum = %d" % response.s
um)
    except Exception:
        # 打印请求失败的错误信息
        minimal_client.get_logger().error("请求失败")

    # 销毁节点对象
    minimal_client.destroy_node()

    # 关闭 ROS 2 节点
    rclpy.shutdown()

# 如果该脚本是主程序，则执行 main 函数
if __name__ == '__main__':
```

```
    main()
```

```
# 如果该脚本是主程序，则执行main函数
if __name__ == '__main__':
    main()
:wq
```

2） 输入指令“**chmod +x service_client.py**”，并按下回车，为保存的 service_client.py 赋予可执行权限。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/service_demo/service_demo$ chmod +x service_c
lient.py
ubuntu@raspberrypi:~/hiwonder_ws/src/service_demo/service_demo$
```

4.setup.py 文件设置

setup.py 文件是定义一个 ROS2 包的远数据和构建配置，提供了包的元数据、依赖关系、构建配置和安装逻辑等信息，可以帮助开发者正确构建、安装和使用 ROS2 包。需要将 service_client.py 和 service_server.py 的程序入口写到 setup.py 文件中。

1） 输入“**cd ..**”指令，切换到上一级目录。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/service_demo/service_demo$ cd ..
ubuntu@raspberrypi:~/hiwonder_ws/src/service_demo$
```

2） 输入“**vim setup.py**”指令，并按下回车，打开 setup.py 文件。

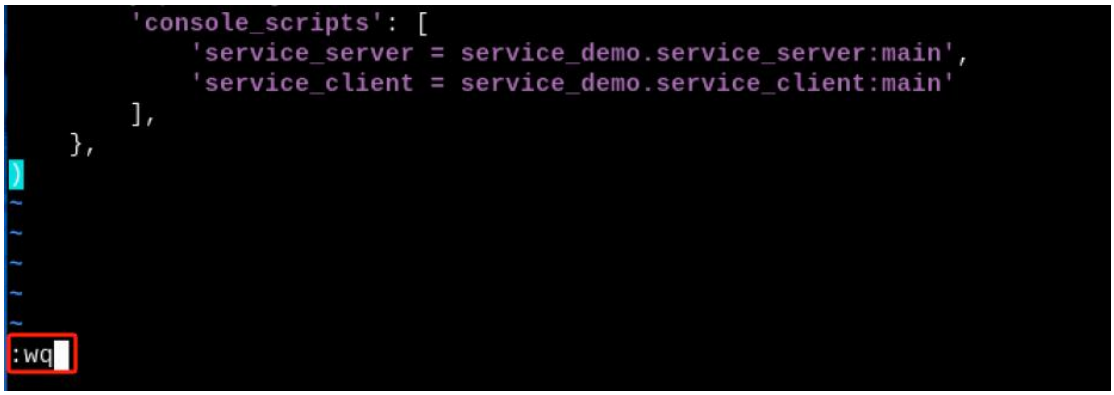
```
ubuntu@raspberrypi:~/hiwonder_ws/src/service_demo$ vim setup.py
```

3） 按下“i”键进入可编辑模式，输入以下代码到相应的位置上。

```
'service_server = service_demo.service_server:main',

'service_client = service_demo.service_client:main'
```

```
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'service_server = service_demo.service_server:main',
            'service_client = service_demo.service_client:main'
        ],
    },
)
```

4） 然后输入“:wq”，保存并退出文件。



5.编译运行

1） 赋予可执行权限后，输入“**cd ~/hiwonder_ws/**”指令，切换到工作空间的目录。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/hello_world_demo$ cd ~/hiwonder_ws/
ubuntu@raspberrypi:~/hiwonder_ws$
```

2） 输入“**colcon build**”指令，并按下回车，编译工作空间中的功能包。

```
ubuntu@raspberrypi:~/hiwonder_ws$ colcon build
```

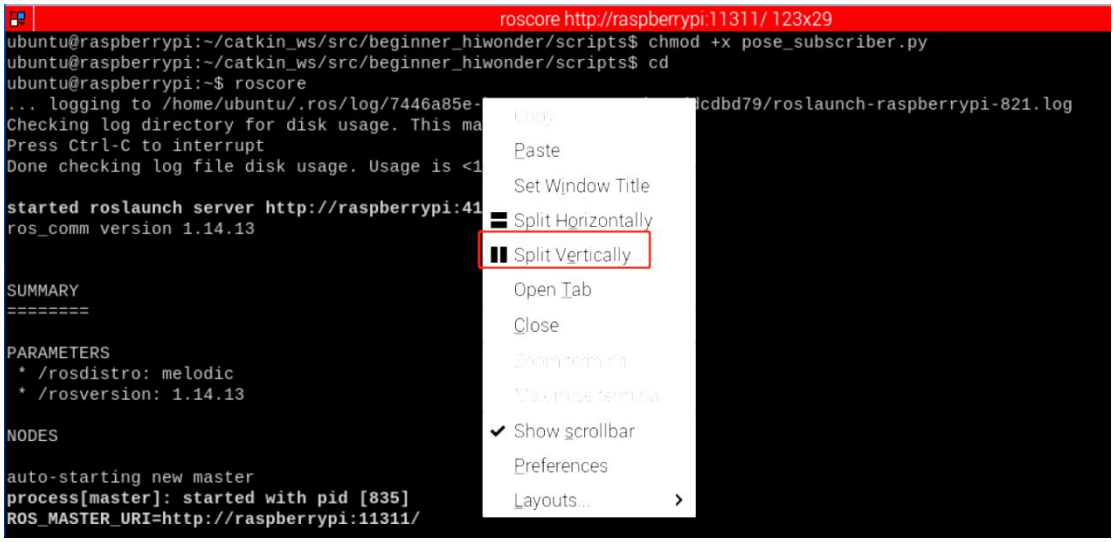
3） 再输入指令“**source ./install/setup.bash**”，并按下回车，让环境变量生效。

```
ubuntu@raspberrypi:~/hiwonder_ws$ source /opt/ros/humble/setup.bash
ubuntu@raspberrypi:~/hiwonder_ws$
```

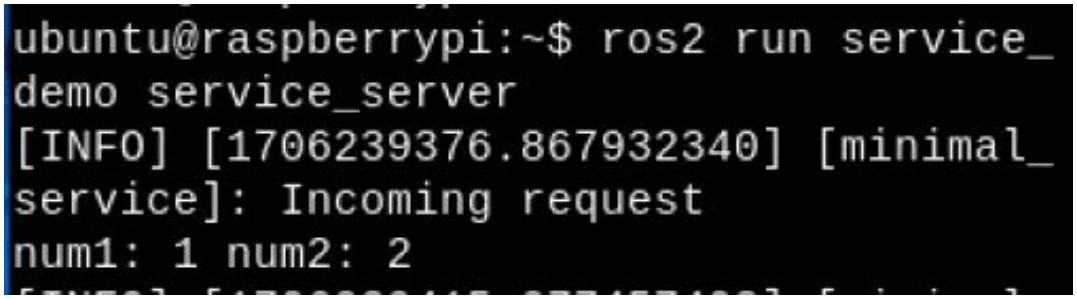
4） 输入指令“**ros2 run service_demo service_server**”，并按下回车，启动 service_server 服务端。

```
ubuntu@raspberrypi:~/hiwonder_ws$ source ./install/setup.bash
```

5） 鼠标右键，选择“**Split Vertically**”点击，新建一个新的终端窗口。



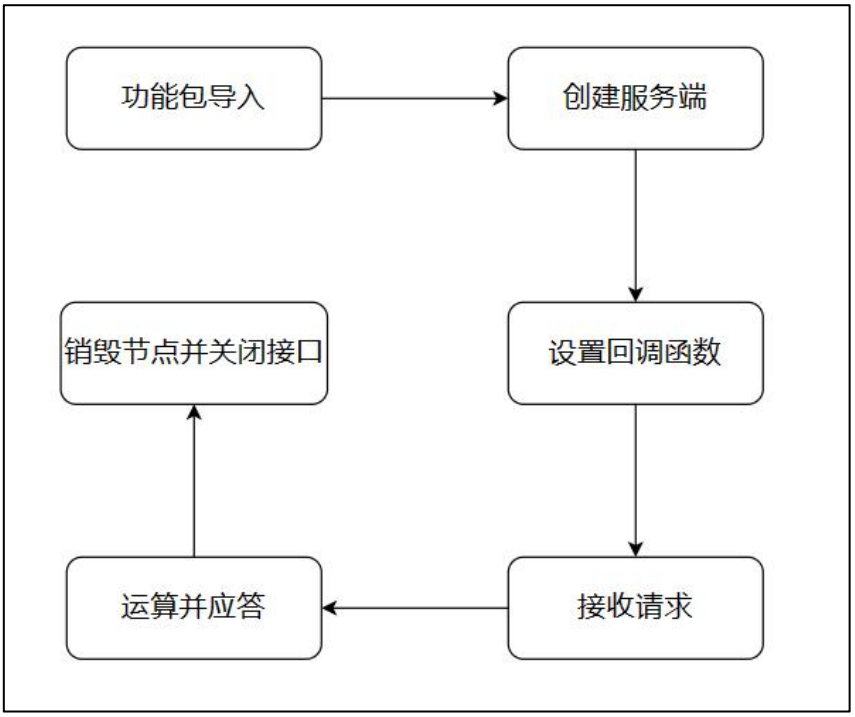
6） 输入指令“**ros2 run service_demo service_client 1 2**”，并按下回车，启动 **service_client** 客户端发送数字 1 和数字 2 的运算请求，服务端收到数字 1 和数字 2 的运算请求后，将运算后的结果发送给客户端。



6.程序分析

6.1 服务端

根据实现的效果，梳理程序的过程逻辑，如下图所示：



创建了一个名为 minimal_service 的服务端，该服务端将收到的请求进行运算，把运算后的结果进行应答响应。在程序运行时，首先会初始化 ROS2 节点，然后创建 MinimalService 对象并进入 ROS 2 节点的事件循环。当程序被中断时，会销毁节点对象并关闭 ROS 2 节点。

◆ 主函数

```
# 主函数
def main():
    # 初始化ROS 2节点
    rclpy.init()

    # 创建MinimalService对象
    minimal_service = MinimalService()

    # 进入ROS 2节点的事件循环
    rclpy.spin(minimal_service)

    # 关闭ROS 2节点
    rclpy.shutdown()
```

首先调用 rclpy.init()对 ROS2 Python 接口进行初始化，然后实例化 MinimalService()，然后在 ROS2 节点的事件循环执行 minimal_service。

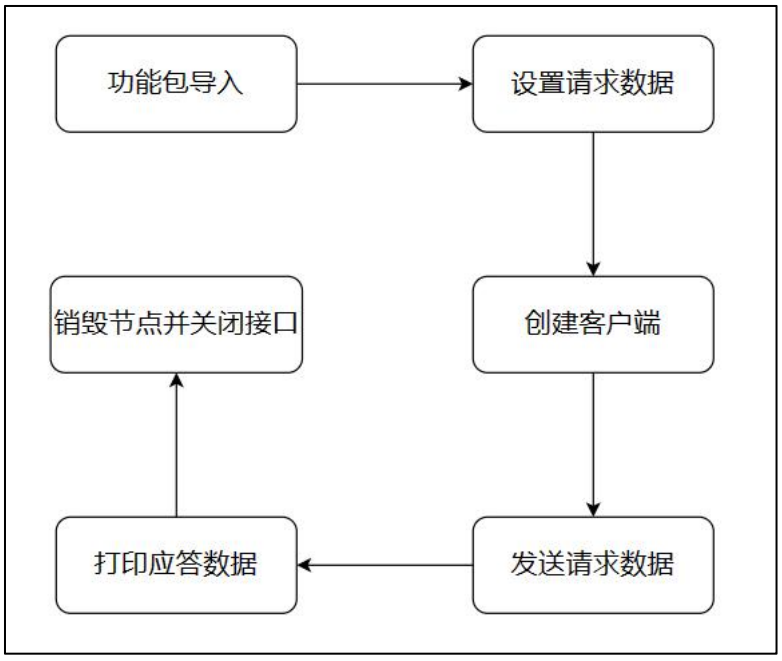
◆ MinimalService 类

```python
# 定义一个继承自Node的MinimalService类
class MinimalService(Node):

    # 类的初始化方法
    def __init__(self):
        # 调用Node类的初始化方法，设置节点名称为'minimal_service'
        super().__init__('minimal_service')

        # 创建一个服务，提供AddInts类型的服务，服务名称为'add_two_ints'，回调函数为add_two_ints_callback
        self.srv = self.create_service(AddInts, 'add_two_ints', self.add_two_ints_callback)

    # 定义服务回调函数
    def add_two_ints_callback(self, request, response):
        # 在日志中记录接收到的请求的num1和num2
        self.get_logger().info('Incoming request\nnum1: %d num2: %d' % (request.num1, request.num2))

        # 计算并设置响应的sum字段
        response.sum = request.num1 + request.num2

        # 返回响应
        return response
```

首先创建了一个名为 minimal_service 的节点，然后创建一个服务，提供 AddInts 类型的服务，服务名称为'add_two_ints'，add_two_ints_callback()回调函数里面接收请求的两个数据，并对其进行运算处理，把运算后的结果响应返回。

6.2 客户端

根据实现的效果，梳理程序的过程逻辑，如下图所示：



创建了一个名为 minimal_client 的客户端，该客户端连接到连接到名为'add_two_ints'的 AddInts 服务，将请求的两个数据发送出去。在程序运行时，首先会初始化 ROS2 节点，然后创建 MinimalClient 对象，发送服务请求，并将响应的结果打印

到出来。当程序被中断时，会销毁节点对象并关闭 ROS 2 节点。

◆ 主函数

```
def main():
    # 检查命令行参数是否包含两个整数
    if len(sys.argv) != 3:
        get_logger("rclpy").error("请提供两个整数值")
        return

    # 初始化ROS 2节点
    rclpy.init()

    # 创建MinimalClient对象
    minimal_client = MinimalClient()

    # 发送服务请求
    minimal_client.send_request()

    # 阻塞等待服务调用完成
    rclpy.spin_until_future_complete(minimal_client, minimal_client.future)

    try:
        # 获取服务调用的响应
        response = minimal_client.future.result()

        # 打印响应结果
        minimal_client.get_logger().info("请求结果：sum = %d" % response.sum)
    except Exception:
        # 打印请求失败的错误信息
        minimal_client.get_logger().error("请求失败")

    # 销毁节点对象
    minimal_client.destroy_node()

    # 关闭ROS 2节点
    rclpy.shutdown()
```

首先调用 rclpy.init()对 ROS2 Python 接口进行初始化，然后实例化 MinimalClient()，接着发送服务请求，最后处理响应的数据。

◆ MinimalClient 类

```python
# 定义一个继承自Node的MinimalClient类
class MinimalClient(Node):

    # 类的初始化方法
    def __init__(self):
        # 调用Node类的初始化方法，设置节点名称为'minimal_client'
        super().__init__('minimal_client')

        # 创建一个服务客户端，连接到名为'add_two_ints'的AddInts服务
        self.cli = self.create_client(AddInts, 'add_two_ints')

        # 等待服务连接，最多等待1秒
        while not self.cli.wait_for_service(timeout_sec=1.0):
            self.get_logger().info('service is connecting...')

    # 发送服务请求的方法
    def send_request(self):
        # 创建一个AddInts请求对象
        request = AddInts.Request()

        # 从命令行参数中获取两个整数，并设置到请求对象中
        request.num1 = int(sys.argv[1])
        request.num2 = int(sys.argv[2])

        # 异步调用服务，并获取一个Future对象
        self.future = self.cli.call_async(request)
```

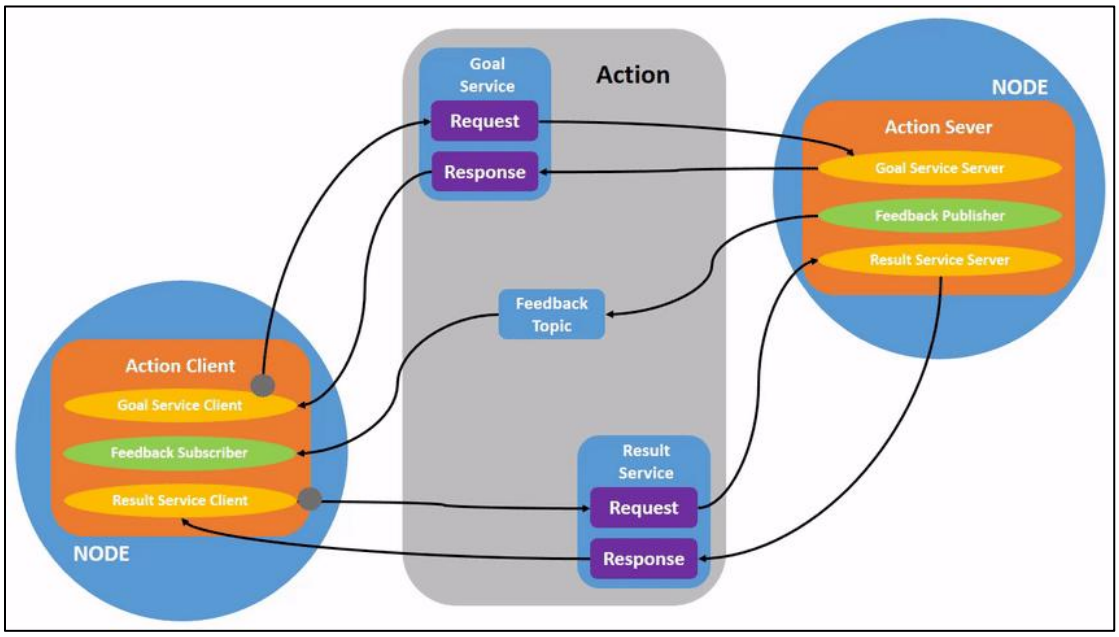
首先创建了一个名为 minimal_client 的节点，然后 创建一个服务客户端，连接到名为'add_two_ints'的 AddInts 服务，然后在 send_request()函数，将命令行终端的两个参数作为请求对象的数据，发送请求。

第 10 课 ROS2 动作

1.动作通信简介

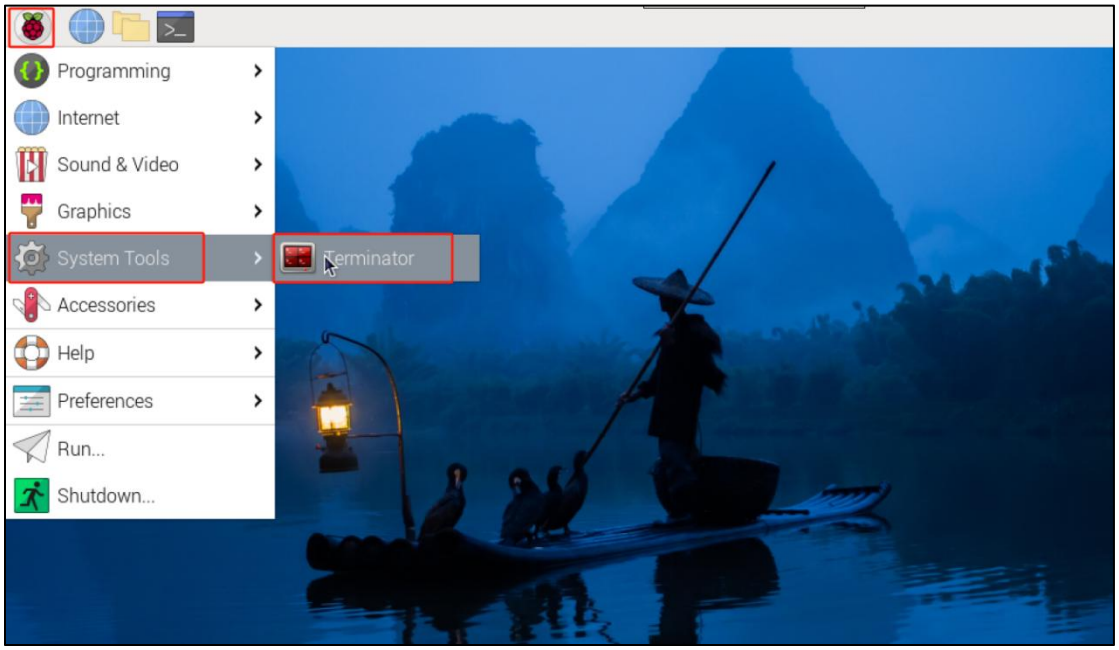
动作通信是一种带有连续反馈的通信模型，在通信双方中，客户端发送请求数据到服务端，服务端响应结果给客户端，但是在服务端接收到请求到产生最终响应的过程中，会发送连续的反馈信息到客户端。

动作通信客户端/服务端模型如下：



2.创建接口

1） 点击左上角的 logo，选择 System Tools →Terminator。



2） 输入“**cd hiwonder_ws/src/demo_interfaces/**”指令，切换到 demo_interfaces 功能包中。

```
ubuntu@raspberrypi:~$ cd hiwonder_ws/src/demo_interfaces/
ubuntu@raspberrypi:~/hiwonder_ws/src/demo_interfaces$
```

3） 输入“**mkdir action**”指令，创建 action 文件夹。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/demo_interfaces$ mkdir action
```

4） 输入“**cd action**”指令，进入 action 文件夹。

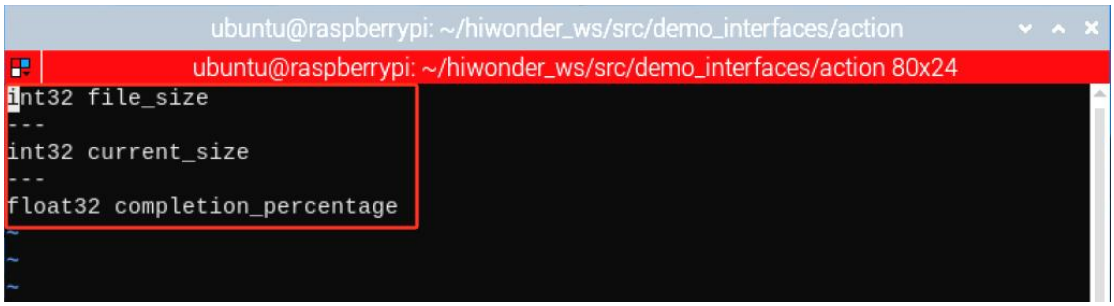
```
ubuntu@raspberrypi:~/hiwonder_ws/src/demo_interfaces$ cd action/
ubuntu@raspberrypi:~/hiwonder_ws/src/demo_interfaces/action$
```

5） 输入指令“**vim FileDownload.action**”编辑程序，输入以下代码。如需修改，再按下“**i**”即可修改。修改完成，按下“**Esc**”，输入“**：wq**”保存并退出。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/demo_interfaces/action$ vim FileDownload.ac
tion
```

```
int32 file_size

---

int32 current_size
```

```
---

float32 completion_percentage
```



6） 先输入“**cd ..**”返回上一级文件夹，再输入“**vim CMakeLists.txt**”，复制下面程序，在下图所示位置添加代码。如需修改，再按下“i”即可修改。修改完成，按下“**Esc**”，输入“**：wq**”保存并退出。

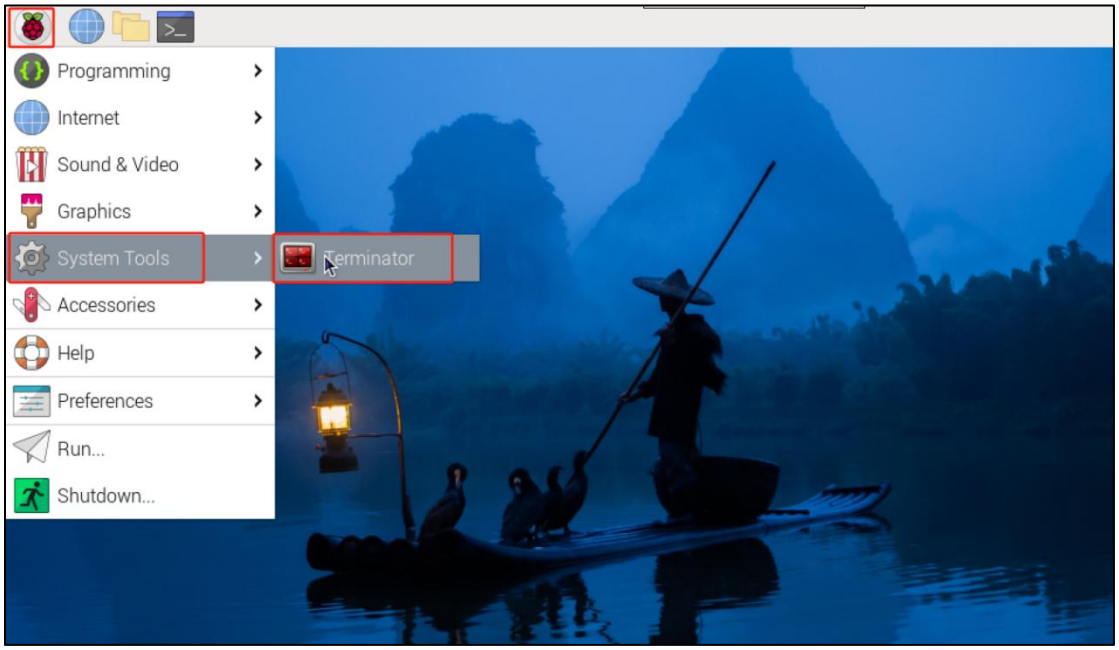
```
find_package(rosidl_default_generators REQUIRED)

rosidl_generate_interfaces( ${PROJECT_NAME}

  "msg/Student.msg"

  "srv/AddInts.srv"

  "action/FileDownload.action"

)
```

```
find_package(rosidl_default_generators REQUIRED)
rosidl_generate_interfaces( ${PROJECT_NAME}
  "msg/String.msg"
  "srv/AddInts.srv"
  "action/FileDownload.action"
)

if(BUILD_TESTING)
  find_package(ament_lint_auto REQUIRED)
```

3.创建动作通信

3.1 创建服务端

1） 点击左上角的 logo，选择 System Tools →Terminator。



2） 输入“**cd hiwonder_ws/src/**”指令，切换到 hiwonder_ws 工作空间的 src 文件夹中。

```
ubuntu@raspberrypi:~$ cd hiwonder_ws/src/
ubuntu@raspberrypi:~/hiwonder_ws/src$
```

3） 输入“**ros2 pkg create action_demo --build-type ament_python --dependencies rclpy**”并按下回车，创建一个名为“**action_demo**”的功能包，添加依赖关系 rclpy。

```
ubuntu@raspberrypi:~/hiwonder_ws/src$ ros2 pkg create action_demo --build-type a
ment_python --dependencies rclpy
```

4） 输入“**cd action_demo/action_demo/**”指令，切换到 action_demo 功能包下。

```
ubuntu@raspberrypi:~/hiwonder_ws/src$ cd action_demo/action_demo/
ubuntu@raspberrypi:~/hiwonder_ws/src/action_demo/action_demo$
```

5） 输入指令“**vim action_server.py**”编辑程序，复制下面程序。如需修改，再按下“**i**”即可修改。修改完成，按下“**Esc**”，输入“**：wq**”保存并退出。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/action_demo/action_demo$ vim action_server.
py
```

```
import rclpy  # 导入 rclpy 模块

from rclpy.node import Node  # 导入 Node 类

from rclpy.action import ActionServer  # 导入 ActionServer 类

from demo_interfaces.action import FileDownload  # 导入 FileDownload acti
on 接口

import random  # 导入 random 模块



class FileDownloadActionServer(Node):  # 定义一个继承自 Node 类的 FileDownlo
adActionServer 类

    def __init__(self):

        super().__init__('file_download_action_server')  # 调用父类构造函
数初始化节点

        self._action_server = ActionServer(

            self,

            FileDownload,  # 使用 FileDownload action 接口

            'file_download',  # 定义 action 的名称为 file_download
```

```
            self.execute_callback)  # 设置回调函数为 execute_callback


    def execute_callback(self, goal_handle):
        # 服务器端执行的回调函数


        self.get_logger().info(f'Start file download for {goal_handle.req
uest.file_size} bytes...')  # 打印开始下载的日志
        feedback_msg = FileDownload.Feedback()  # 创建 Feedback 消息类型的对
象


        current_size = 0  # 初始化当前下载的文件大小为 0
        while current_size < goal_handle.request.file_size:
            increment_size = random.randint(1, 10)  # 模拟随机增加下载大小
            current_size += increment_size  # 更新当前下载的文件大小
            if current_size > goal_handle.request.file_size:
                current_size = goal_handle.request.file_size
            completion_percentage = (current_size / goal_handle.request.f
ile_size) * 100  # 计算下载进度百分比


            feedback_msg.completion_percentage = completion_percentage  #
 更新 Feedback 消息的下载进度
            self.get_logger().info(f'Publishing feedback: {completion_per
centage:.2f}% downloaded')  # 打印发布的反馈消息
```

```
            goal_handle.publish_feedback(feedback_msg)  # 发布反馈消息
            rclpy.spin_once(self, timeout_sec=1.0)  # 模拟时间的流逝

        goal_handle.succeed()  # 指示目标已完成
        result = FileDownload.Result()  # 创建 Result 消息类型的对象
        result.current_size = current_size  # 设置 Result 消息的当前文件大小
        self.get_logger().info('File download completed!')  # 打印文件下载完成的日志
        return result  # 返回结果

def main(args=None):
    rclpy.init(args=args)  # 初始化 ROS 节点
    server = FileDownloadActionServer()  # 创建 FileDownloadActionServer 对象
    rclpy.spin(server)  # 进入主循环
    server.destroy_node()  # 销毁节点
    rclpy.shutdown()  # 关闭 ROS

if __name__ == '__main__':
    main()
```

```
# 如果该脚本是主程序，则执行main函数
if __name__ == '__main__':
    main()
:wq
```

6） 输入指令“**chmod +x action_server.py**”，并按下回车，为保存的 action_server.py 赋予可执行权限。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/action_demo/action_demo$ chmod +x action_server.py
ubuntu@raspberrypi:~/hiwonder_ws/src/action_demo/action_demo$
```

3.2 创建客户端

1） 输入指令“**vim action_client.py**”编辑程序，复制下面程序。如需修改，再按下“**i**”即可修改。修改完成，按下“**Esc**”，输入“**：wq**”保存并退出。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/action_demo/action_demo$ vim action_client.py
```

```
import rclpy  # 导入 rclpy 模块

from rclpy.node import Node  # 导入 Node 类

from rclpy.action import ActionClient  # 导入 ActionClient 类

from demo_interfaces.action import FileDownload  # 导入 FileDownload action 接口



class FileDownloadActionClient(Node):  # 定义一个继承自 Node 类的 FileDownloadActionClient 类

    def __init__(self):

        super().__init__('file_download_action_client')  # 调用父类构造函数初始化节点

        self._action_client = ActionClient(self, FileDownload, 'file_down
```

```
load')  # 创建 ActionClient 对象


    def send_goal(self, file_size):
        goal_msg = FileDownload.Goal()  # 创建 Goal 消息类型的对象
        goal_msg.file_size = file_size  # 设置文件大小

        self.get_logger().info(f'Sending file download goal for {file_siz
e} bytes')  # 打印发送目标的日志

        self._action_client.wait_for_server()  # 等待服务器可用
        self.future = self._action_client.send_goal_async(goal_msg, feed
back_callback=self.feedback_callback)  # 发送目标异步请求

        self.future.add_done_callback(self.goal_response_callback)  # 添
加目标响应回调函数


    def goal_response_callback(self,future):
        goal_handle = future.result()  # 获取目标句柄
        if not goal_handle.accepted:
            self.get_logger().info("Goal rejected")  # 打印目标被拒绝的日志
            return
        self.get_logger().info("Goal accepted")  # 打印目标被接受的日志
        self._get_result_future = goal_handle.get_result_async()  # 获取
```

```
目标结果异步请求

        self._get_result_future.add_done_callback(self.get_result_callba
ck)  # 添加获取结果的回调函数



    def get_result_callback(self,future):

        self.get_logger().info("File download completed successfully.")
# 打印文件下载成功完成的日志



    def feedback_callback(self, feedback_msg):

        self.get_logger().info(f'Received feedback: {feedback_msg.feedba
ck.completion_percentage:.2f}% downloaded')  # 打印接收到的反馈消息



def main(args=None):

    rclpy.init(args=args)  # 初始化 ROS 节点

    client = FileDownloadActionClient()  # 创建 FileDownloadActionClient
对象



    result = client.send_goal(100)  # 发送目标



    rclpy.spin(client)  # 进入主循环

    client.destroy_node()  # 销毁节点

    rclpy.shutdown()  # 关闭 ROS
```

```
if __name__ == '__main__':

    main()
```

2） 输入指令“**chmod +x action_client.py**”，并按下回车，为保存的 action_client.py 赋予可执行权限。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/action_demo/action_demo$ chmod +x action_cl
ient.py
ubuntu@raspberrypi:~/hiwonder_ws/src/action_demo/action_demo$
```

4.setup.py 文件设置

setup.py 文件是定义一个 ROS2 包的远数据和构建配置，提供了包的元数据、依赖关系、构建配置和安装逻辑等信息，可以帮助开发者正确构建、安装和使用 ROS2 包。需要将 action_server.py 和 action_client.py 的程序入口写到 setup.py 文件中。

1） 输入“**cd ..**”指令，切换到上一级目录。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/action_demo/action_demo$ cd ..
ubuntu@raspberrypi:~/hiwonder_ws/src/action_demo$
```

2） 输入“**vim setup.py**”指令，并按下回车，打开 setup.py 文件。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/action_demo$ vim setup.py
```

3） 按下“i”键进入可编辑模式，输入以下代码到相应的位置上。

```
'action_server = action_demo.action_server:main',

'action_client = action_demo.action_client:main'
```

```
    license='TODO: License declaration',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'action_server = action_demo.action_server:main',
            'action_client = action_demo.action_client:main'

        ],
    },
)
-- INSERT --                                                    28,1
```

4） 然后输入“:wq”，保存并退出文件。

```
        'console_scripts': [
            'action_server = action_demo.action_server:main',
            'action_client = action_demo.action_client:main'

        ],
    },
)
:wq
```

4.编译运行

1） 赋予可执行权限后，输入“**cd ~/hiwonder_ws/**”指令，切换到工作空间的目录。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/hello_world_demo$ cd ~/hiwonder_ws/
ubuntu@raspberrypi:~/hiwonder_ws$
```

2） 输入“**colcon build**”指令，并按下回车，编译工作空间中的功能包。

```
ubuntu@raspberrypi:~/hiwonder_ws$ colcon build
```

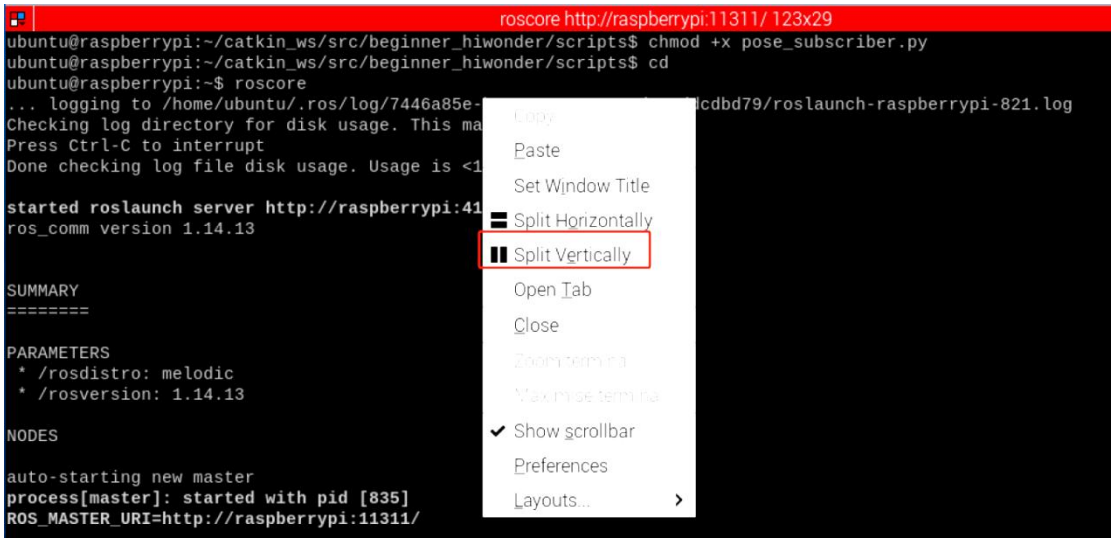
3） 再输入指令“**source ./install/setup.bash**”，并按下回车，让环境变量生效。

```
ubuntu@raspberrypi:~/hiwonder_ws$ source ./install/setup.bash
```

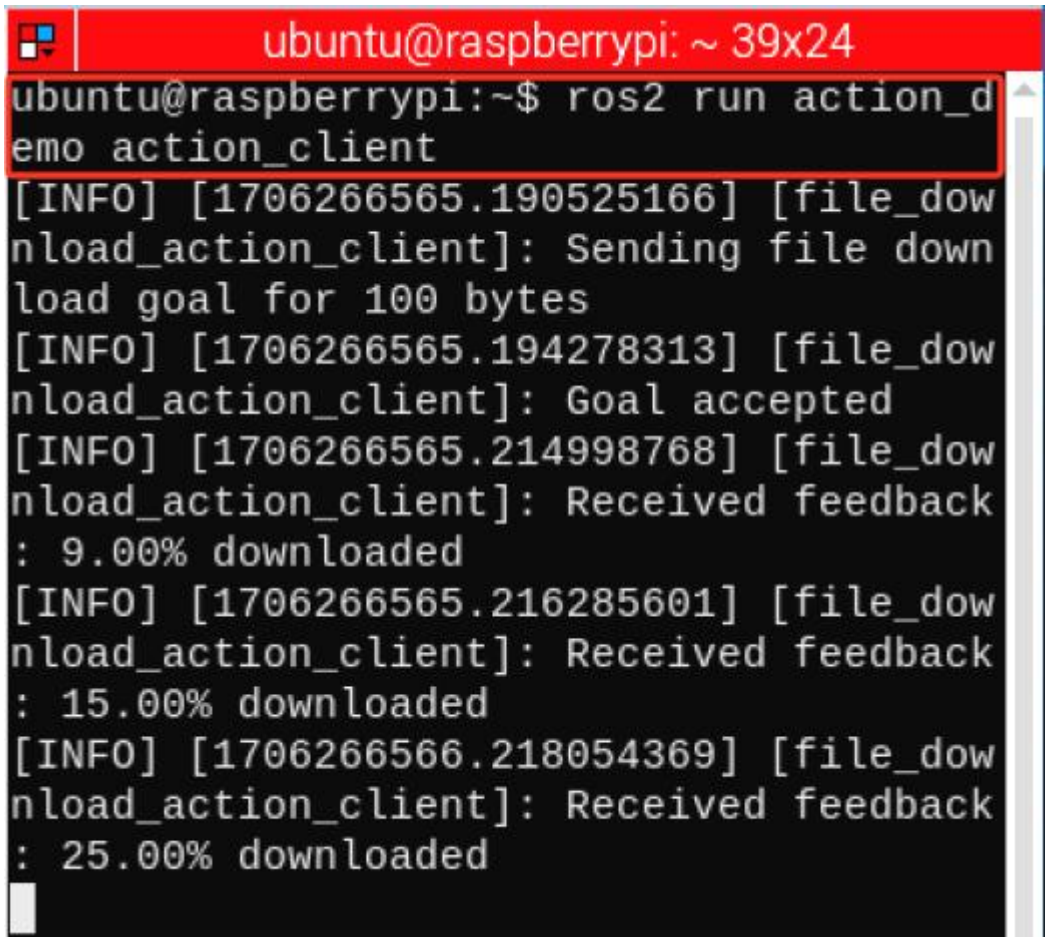
4） 输入指令“**ros2 run action_demo action_server**”，并按下回车，启动 action_server 动作服务端。

```
ubuntu@raspberrypi:~$ ros2 run action_demo action_server
```

5） 鼠标右键，选择“**Split Vertically**”点击，新建一个新的终端窗口。



6） 输入指令“**ros2 run action_demo action_client**”，并按下回车，启动 action_client 动作客户端，这时候服务端也接收到请求。

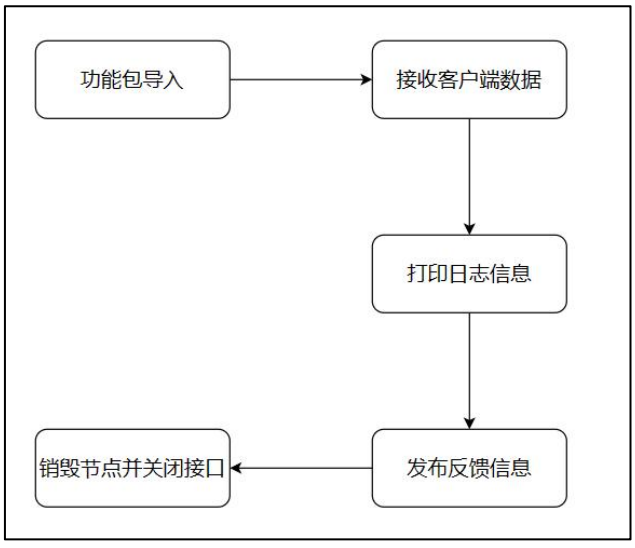


```
ubuntu@raspberrypi: ~ 38x24
ubuntu@raspberrypi:~$ ros2 run action_
demo action_server
[INFO] [1706266565.212394047] [file_do
wnload_action_server]: Start file down
load for 100 bytes...
[INFO] [1706266565.213241269] [file_do
wnload_action_server]: Publishing feed
back: 9.00% downloaded
[INFO] [1706266565.215019324] [file_do
wnload_action_server]: Publishing feed
back: 15.00% downloaded
[INFO] [1706266566.216574240] [file_do
wnload_action_server]: Publishing feed
back: 25.00% downloaded
[INFO] [1706266567.218210101] [file_do
wnload_action_server]: Publishing feed
back: 28.00% downloaded
```

5.程序分析

5.1 服务端

根据实现的效果，梳理程序的过程逻辑，如下图所示：



首先定义了一个 FileDownloadActionServer 节点，构造一个 FileDownload 动作类型的 ActionServer 来提供任务服务。ActionServer 的执行回调函数负责模拟执行真实下载任务的流程，它会随机增加下载进度来模拟文件下载，同时通过 goal_handle 对象不断发布下载进度反馈给客户端。任务执行完毕后，会将任务状态设为成功，并返回下载结果。

◆ 主函数

```
def main(args=None):
    rclpy.init(args=args)  # 初始化ROS节点
    server = FileDownloadActionServer()  # 创建FileDownloadActionServer对象
    rclpy.spin(server)  # 进入主循环
    server.destroy_node()  # 销毁节点
    rclpy.shutdown()  # 关闭ROS
```

首先调用 rclpy.init()对 ROS2 Python 接口进行初始化，然后实例化 FileDownloadActionServer()，然后在 ROS2 节点的事件循环执行 server。

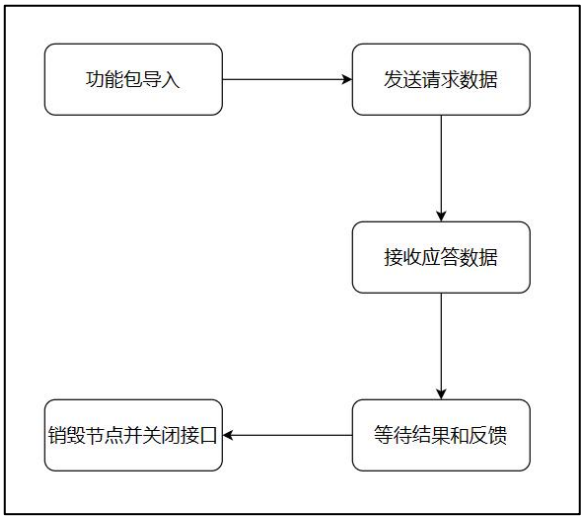
◆ FileDownloadActionServer 类

```
class FileDownloadActionServer(Node):  # 定义一个继承自Node类的FileDownloadActionServer类
    def __init__(self):
        super().__init__('file_download_action_server')  # 调用父类构造函数初始化节点
        self._action_server = ActionServer(
            self,
            FileDownload,  # 使用FileDownload action接口
            'file_download',  # 定义action的名称为file_download
            self.execute_callback)  # 设置回调函数为execute_callback

    def execute_callback(self, goal_handle):
        # 服务器端执行的回调函数

        self.get_logger().info(f'Start file download for {goal_handle.request.file_size} bytes...')  # 打印开始下载的日志
        feedback_msg = FileDownload.Feedback()  # 创建Feedback消息类型的对象

        current_size = 0  # 初始化当前下载的文件大小为0
        while current_size < goal_handle.request.file_size:
            increment_size = random.randint(1, 10)  # 模拟随机增加下载大小
            current_size += increment_size  # 更新当前下载的文件大小
            if current_size > goal_handle.request.file_size:
                current_size = goal_handle.request.file_size
            completion_percentage = (current_size / goal_handle.request.file_size) * 100  # 计算下载进度百分比

            feedback_msg.completion_percentage = completion_percentage  # 更新Feedback消息的下载进度
            self.get_logger().info(f'Publishing feedback: {completion_percentage:.2f}% downloaded')  # 打印发布的反馈消息
            goal_handle.publish_feedback(feedback_msg)  # 发布反馈消息
            rclpy.spin_once(self, timeout_sec=1.0)  # 模拟时间的流逝

        goal_handle.succeed()  # 指示目标已完成
        result = FileDownload.Result()  # 创建Result消息类型的对象
        result.current_size = current_size  # 设置Result消息的当前文件大小
        self.get_logger().info('File download completed!')  # 打印文件下载完成的日志
        return result  # 返回结果
```

首先创建了一个 ActionServer 对象，将文件下载任务定义为 FileDownload 动作类型，并注册回调函数 execute_callback。这个回调函数在被调用时就会执行文件下载任务，它通过一个 while 循环内部随机增加当前下载量，来模拟文件的实时下载过程。在下载过程中，它不断通过 goal_handle 对象发布当前下载进度作为反馈信息。任务下载完成后，会使用 goal_handle 将任务状态设置为成功，并返回最终下载结果。

5.2 订阅端

根据实现的效果，梳理程序的过程逻辑，如下图所示：



定义了一个 FileDownloadActionClient 类,在构造函数中创建一个 ActionClient 对象,配置文件下载任务类型。客户端提供了一个 send_goal 方法来发送下载目标文件大小作为动作 Goal。同时它还注册了 3 个回调函数,用于处理 Goal 状态响应、任务进度反馈和最后结果。主函数中构造客户端实例,调用 send_goal 发送下载任务,并通过 rclpy.spin()来运行回调函数从而获取任务执行全过程的状态变化。

◆ 主函数

```
def main(args=None):
    rclpy.init(args=args)  # 初始化ROS节点
    client = FileDownloadActionClient()  # 创建FileDownloadActionClient对象

    result = client.send_goal(100)  # 发送目标

    rclpy.spin(client)  # 进入主循环
    client.destroy_node()  # 销毁节点
    rclpy.shutdown()  # 关闭ROS
```

首先调用 rclpy.init()对 ROS2 Python 接口进行初始化，然后实例化 FileDownloadActionClient()，发送一个下载任务，然后在 ROS2 节点的事件循环执行 client。

◆ FileDownloadActionClient 类

```python
class FileDownloadActionClient(Node):  # 定义一个继承自Node类的FileDownloadActionClient类
    def __init__(self):
        super().__init__('file_download_action_client')  # 调用父类构造函数初始化节点
        self._action_client = ActionClient(self, FileDownload, 'file_download')  # 创建ActionClient对象

    def send_goal(self, file_size):
        goal_msg = FileDownload.Goal()  # 创建Goal消息类型的对象
        goal_msg.file_size = file_size  # 设置文件大小

        self.get_logger().info(f'Sending file download goal for {file_size} bytes')  # 打印发送目标的日志

        self._action_client.wait_for_server()  # 等待服务器可用
        self.future = self._action_client.send_goal_async(goal_msg, feedback_callback=self.feedback_callback)  # 发送目标>
异步请求

        self.future.add_done_callback(self.goal_response_callback)  # 添加目标响应回调函数

    def goal_response_callback(self,future):
        goal_handle = future.result()  # 获取目标句柄
        if not goal_handle.accepted:
            self.get_logger().info("Goal rejected")  # 打印目标被拒绝的日志
            return
        self.get_logger().info("Goal accepted")  # 打印目标被接受的日志
        self._get_result_future = goal_handle.get_result_async()  # 获取目标结果异步请求
        self._get_result_future.add_done_callback(self.get_result_callback)  # 添加获取结果的回调函数

    def get_result_callback(self,future):
        self.get_logger().info("File download completed successfully.")  # 打印文件下载成功完成的日志

    def feedback_callback(self, feedback_msg):
        self.get_logger().info(f'Received feedback: {feedback_msg.feedback.completion_percentage:.2f}% downloaded')  # 打>
印接收到的反馈消息
```

创建了一个 ActionClient 对象来与任务服务进行通信。客户端提供了一个 send_goal 方法发送下载目标,并注册了 3 个回调函数用于处理任务不同阶段的状态变化 feedback。send_goal 方法在发送 Goal 后,会通过 future 对象添加回调函数,等待 ActionServer 的响应。这 3 个回调函数分别用来获取 Goal 接收结果、任务进度更新以及完成结果,能覆盖任务整个生命周期。

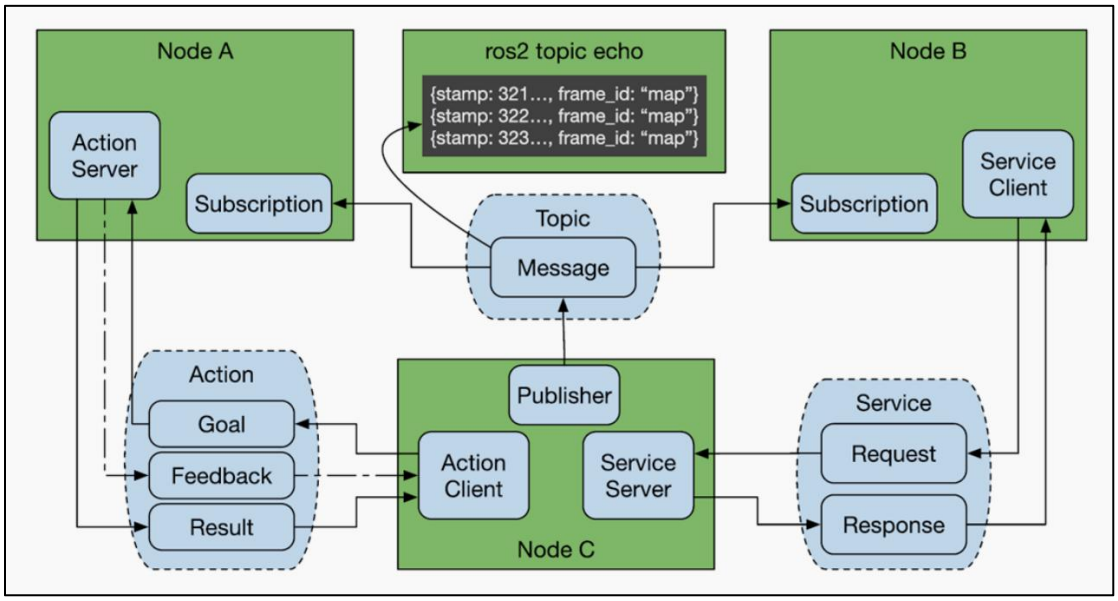
第 11 课 ROS2 通信接口

1.通信接口简介

在 ROS 系统中，无论话题、服务还是动作，都会用到一个重要的概念--通信接口。

通信并不是一个人自言自语，而是两个甚至更多人，你来我往的交流，交流的内容是什么呢？为了让大家都好理解，我们可以给传递的数据定义一个标准的结构，这就是通信接口。

接口可以让程序之间的依赖降低，便于我们使用别人的代码，也方便别人使用我们的代码，这就是 ROS 的核心目标，减少重复造轮子。

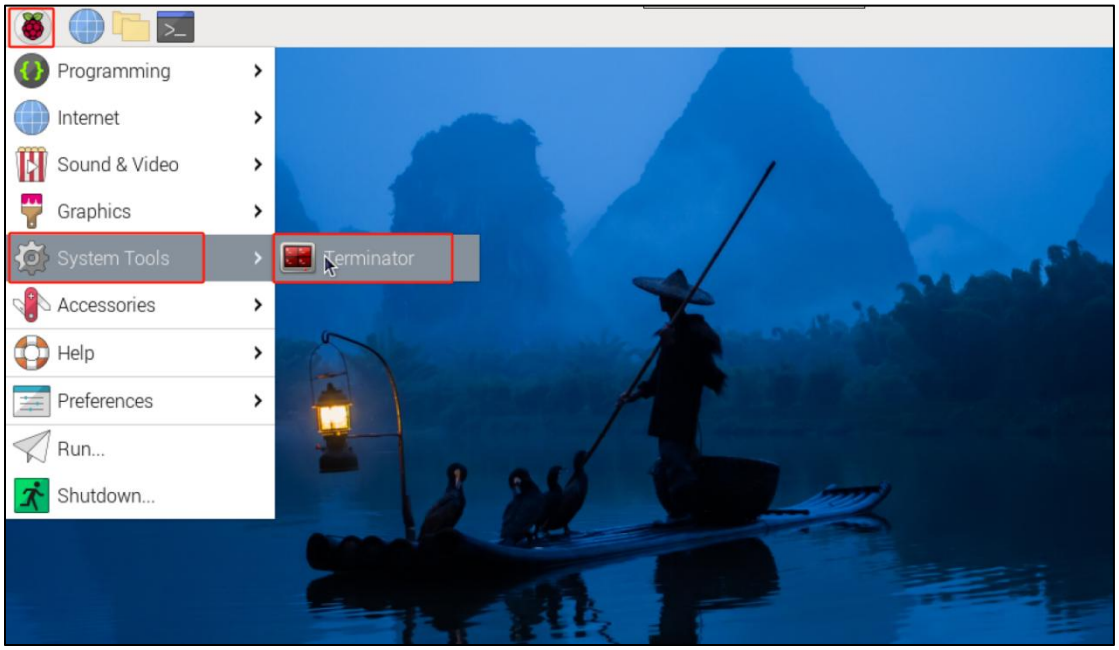


ROS 有三种常用的通信机制，分别是话题、服务、动作，通过每一种通信定义的接口，各种节点才能有机的联系到一起。

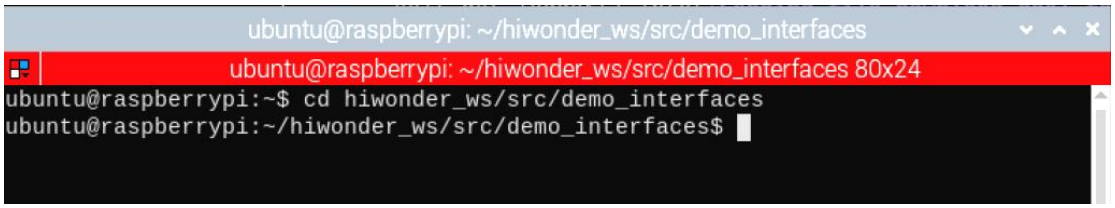
2.自定义接口

在本章节的“**第 8 课 ROS2 话题、第 9 课、服务说明、第 10 课 ROS2 动作**”的课程中，我们创建了 demo_interfaces 的自定义接口功能包，并在包里创建了 String.msg、File Download.action 和 AddInts.srv 这三个自定义的接口。

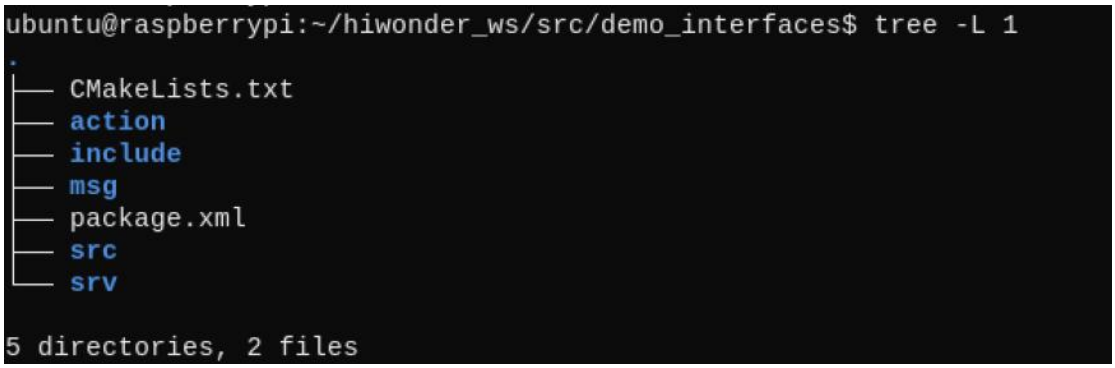
1） 点击左上角的 logo，选择 System Tools →Terminator。



2） 输入“**cd hiwonder_ws/src/demo_interfaces**”指令，切换到 demo_interfaces 的自定义接口功能包中。



3） 输入“**tree -L 1**”并按下回车，查看功能包的根目录。



demo_interfaces 功能包中有 action、msg 和 srv 存放自定义接口文件的文件夹。以下是各文件夹的说明：

名字	详细说明

action	存放动作通信的.action 文件
msg	存放话题通信的.msg 文件
srv	存放服务通信的.srv 文件

第 12 课 ROS2 参数说明

1.参数简介

类似 C++编程中的全局变量，可以便于在多个程序中共享某些数据，参数是 ROS 机器人系统中的全局字典，可以运行多个节点中共享数据。

在 ROS 系统中，参数是以全局字典的形态存在的，什么叫字典？就像真实的字典一样，由名称和数值组成，也叫做键和值，合成键值。或者我们也可以理解为，就像编程中的参数一样，有一个参数名，然后跟一个等号，后边就是参数值了，在使用的时候，访问这个参数名即可。

参数的特性非常丰富，比如某一个节点共享了一个参数，其他节点都可以访问，如果某一个节点对参数进行了修改，其他节点也有办法立刻知道，从而获取最新的数值。

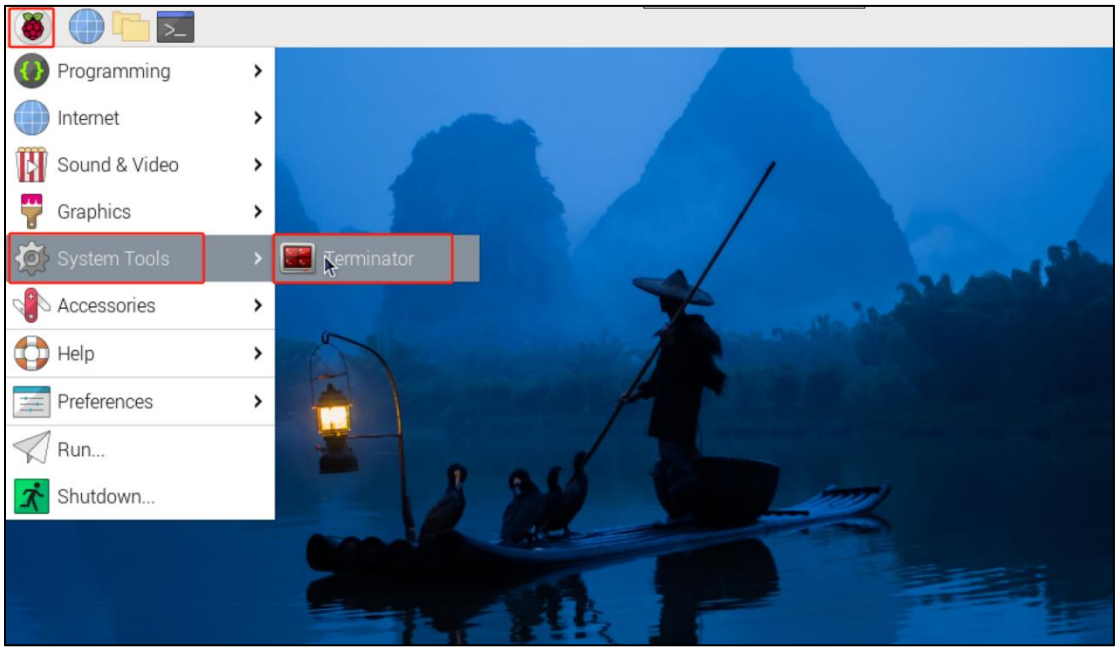
2.param 参数

我们先来了解下 param 参数指令，详细说明如下：

指令	说明
ros2 param list	列出当前多个参数
ros2 get param_key	显示某个参数值
ros2 set param_key param_value	设置某个参数值
ros2 param dump file_name	保存参数到文件
ros2 param load file_name	从文件读取参数
ros2 param delete param_key	删除参数

3.创建参数案例

1） 点击左上角的 logo，选择 System Tools →Terminator。



2） 输入“**cd hiwonder_ws/src/**”指令，切换到 hiwonder_ws 工作空间的 src 文件夹中。

```
ubuntu@raspberrypi:~$ cd hiwonder_ws/src/
ubuntu@raspberrypi:~/hiwonder_ws/src$
```

3） 输入“**ros2 pkg create param_demo --build-type ament_python --dependencies rclpy**”并按下回车，创建一个名为“**param_demo**”的功能包，添加依赖关系 rclpy。

```
ubuntu@raspberrypi:~/hiwonder_ws/src$ ros2 pkg create param_demo --build-type ament_python --dependencies rclpy
```

4） 输入“**cd param_demo/param_demo/**”指令，切换到 param_demo 功能包下。

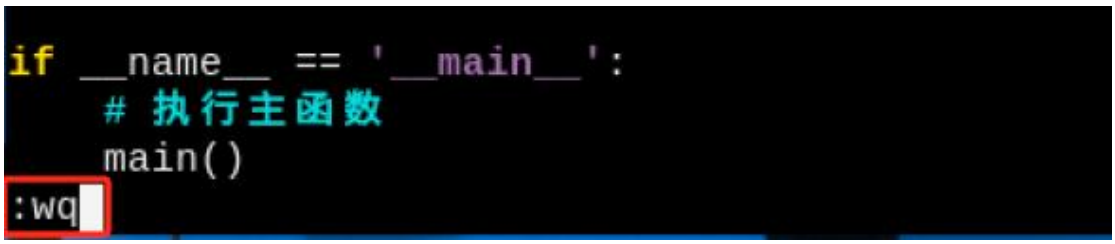
```
ubuntu@raspberrypi:~/hiwonder_ws/src$ cd param_demo/param_demo/
ubuntu@raspberrypi:~/hiwonder_ws/src/param_demo/param_demo$
```

5） 输入指令“**vim param_demo.py**”编辑程序，复制下面程序。如需修改，再按下“**i**”即可修改。修改完成，按下“**Esc**”，输入“**: wq**”保存并退出。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/param_demo/param_demo$ vim param_demo.py
```

```
import rclpy  # 导入 rclpy 模块
from rclpy.node import Node  # 导入 Node 类
from rclpy.parameter import Parameter  # 导入 Parameter 类


class MinimalParam(Node):  # 定义一个继承自 Node 类的 MinimalParam 类
    def __init__(self):
        super().__init__('minimal_param_node')  # 调用父类构造函数初始化节点


        self.declare_parameter('my_parameter', 'hiwonder')  # 声明一个名为'my_parameter'的参数，并设置默认值为'hiwonder'


        self.timer = self.create_timer(1, self.timer_callback)  # 创建定时器，设置回调函数为 timer_callback，时间间隔为 1 秒


    def timer_callback(self):
        my_param = self.get_parameter('my_parameter').get_parameter_value().string_value  # 获取参数'my_parameter'的值，并转换为字符串


        self.get_logger().info('Hello %s!' % my_param)  # 打印带有参数值的日志消息
```

```
        my_new_param = Parameter(  # 创建一个新的参数对象
            'my_parameter',  # 参数名称为'my_parameter'
            rclpy.Parameter.Type.STRING,  # 参数类型为字符串
            'hiwonder'  # 参数值为'hiwonder'
        )
        all_new_parameters = [my_new_param]  # 将新的参数对象放入列表中
        self.set_parameters(all_new_parameters)  # 设置节点的参数值为新的参数值


def main():
    rclpy.init()  # 初始化 ROS 节点
    node = MinimalParam()  # 创建 MinimalParam 对象
    rclpy.spin(node)  # 进入主循环


if __name__ == '__main__':
    main()
```



1） 输入指令“**chmod +x param_demo.py**”，并按下回车，为保存的 param_demo.py 赋予可执行权限。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/param_demo/param_demo$ chmod +x param_demo.
py
ubuntu@raspberrypi:~/hiwonder_ws/src/param_demo/param_demo$
```

4.编译运行

1） 赋予可执行权限后，输入“**cd ~/hiwonder_ws/**”指令，切换到工作空间的目录。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/param_demo/param_demo$ cd ~/hiwonder_ws/
ubuntu@raspberrypi:~/hiwonder_ws$
```

2） 输入“**colcon build**”指令，并按下回车，编译工作空间中的功能包。

```
ubuntu@raspberrypi:~/hiwonder_ws$ colcon build
```

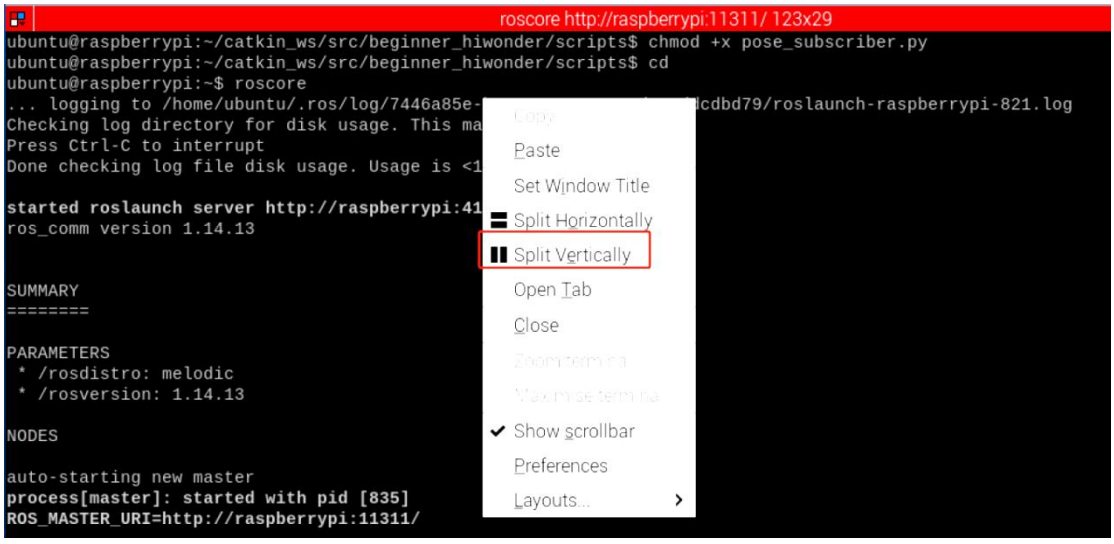
3） 再输入指令“**source ./install/setup.bash**”，并按下回车，让环境变量生效。

```
ubuntu@raspberrypi:~/hiwonder_ws$ source ./install/setup.bash
```

4） 输入指令“**ros2 run param_demo param_demo**”，并按下回车，启动 param_demo 节点。

```
ubuntu@raspberrypi:~$ ros2 run param_demo param_demo
[INFO] [1706322737.926459775] [minimal_param_node]: Hello hiwonder!
[INFO] [1706322738.914357462] [minimal_param_node]: Hello hiwonder!
[INFO] [1706322739.914370172] [minimal_param_node]: Hello hiwonder!
[INFO] [1706322740.914414199] [minimal_param_node]: Hello hiwonder!
```

5） 鼠标右键，选择“**Split Vertically**”点击，新建一个新的终端窗口。



6） 输入指令“**ros2 param set minimal_param_node my_parameter world**”，

并按下回车，修改 minimal_param_node 这个节点的 my_parameter 参数为 world。

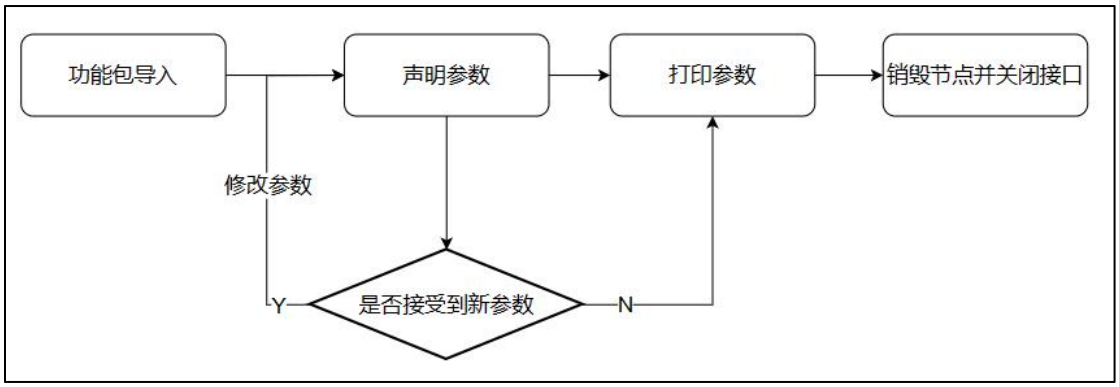
```
ubuntu@raspberrypi:~$ ros2 param set minimal_param_node my_parameter world
Set parameter successful
```

这时候可以发现 param_demo 节点输出的信息为“**Hello world!**”，说明参数修改成功。

```
[INFO] [1706322934.672806231] [minimal_param_node]: Hello hiwonder!
[INFO] [1706322935.672741228] [minimal_param_node]: Hello hiwonder!
[INFO] [1706322936.672833874] [minimal_param_node]: Hello hiwonder!
[INFO] [1706322937.673462392] [minimal_param_node]: Hello world!
[INFO] [1706322938.672808951] [minimal_param_node]: Hello hiwonder!
[INFO] [1706322939.672779104] [minimal_param_node]: Hello hiwonder!
[INFO] [1706322940.672838369] [minimal_param_node]: Hello hiwonder!
```

4.程序分析

根据实现的效果，梳理程序的过程逻辑，如下图所示：



创建了一个 MinimalParam 类,在构造函数中声明了一个字符串参数"my_parameter",并创建了一个 1 秒间隔的定时器。定时器回调函数每隔 1 秒会被调用,在回调内它首先获取参数当前的值,然后打印日志,再创建一个新的参数对象并设置新值,最后调用 set_parameters 函数来修改参数服务器中的参数值。主函数初始化 ROS 节点和环境,创建节点实例,并进入主循环来驱动定时器回调的执行。

- 主函数

```
def main():
    rclpy.init()  # 初始化ROS节点
    node = MinimalParam()  # 创建MinimalParam对象
    rclpy.spin(node)  # 进入主循环
```

首先调用 rclpy.init()对 ROS2 Python 接口进行初始化，然后实例化 MinimalParam，然后执行 node.run()函数。

◆ MinimalParam 类

```
class MinimalParam(Node):  # 定义一个继承自Node类的MinimalParam类
    def __init__(self):
        super().__init__('minimal_param_node')  # 调用父类构造函数初始化节点

        self.declare_parameter('my_parameter', 'hiwonder')  # 声明一个名为'my_parameter'的参数，并设置默认值为'hiwonder'

        self.timer = self.create_timer(1, self.timer_callback)  # 创建定时器，设置回调函数为timer_callback，时间间隔为1秒

    def timer_callback(self):
        my_param = self.get_parameter('my_parameter').get_parameter_value().string_value  # 获取参数'my_parameter'的值，并
转换为字符串

        self.get_logger().info('Hello %s!' % my_param)  # 打印带有参数值的日志消息

        my_new_param = Parameter(  # 创建一个新的参数对象
            'my_parameter',  # 参数名称为'my_parameter'
            rclpy.Parameter.Type.STRING,  # 参数类型为字符串
            'hiwonder'  # 参数值为'hiwonder'
        )
        all_new_parameters = [my_new_param]  # 将新的参数对象放入列表中
        self.set_parameters(all_new_parameters)  # 设置节点的参数值为新的参数值
```

首先声明了一个名为"my_parameter"的字符串参数,并创建了一个 1 秒周期的定时器.定时器回调函数 timer_callback 会每秒被调用。它首先获取"my_parameter"参数目前的值,然后打印日志输出。紧接着,回调函数创建一个新参数对象,设置相同名称但值为"hiwonder"的新参数值。之后调用 set_parameters 函数来修改参数值。。

第 13 课 分布式通信说明

1.分布式通信简介

多机通讯即分布式通信是指可以通过网络在不同主机之间实现数据交互的一种通信策略。

ROS2 本身是一个分布式通信框架，可以很方便的实现不同设备之间的通信，ROS2 所基于的中间件是 DDS，当处于同一个网络中，通过 DDS 的域 ID 机制（ROS_DOMAIN_ID）可以实现分布式通信，大致流程是：在启动节点之前，可以设置域 ID 的值，不同节点如果域 ID 相同，那么可以自由发现并通信，反之，如果域 ID 值不同，则不能实现。默认情况下，所有节点启动时所使用的域 ID 为 0，换而言之，只要保证在同一网络，你不需要做任何配置，不同 ROS2 设备上的不同节点即可实现分布式通信。

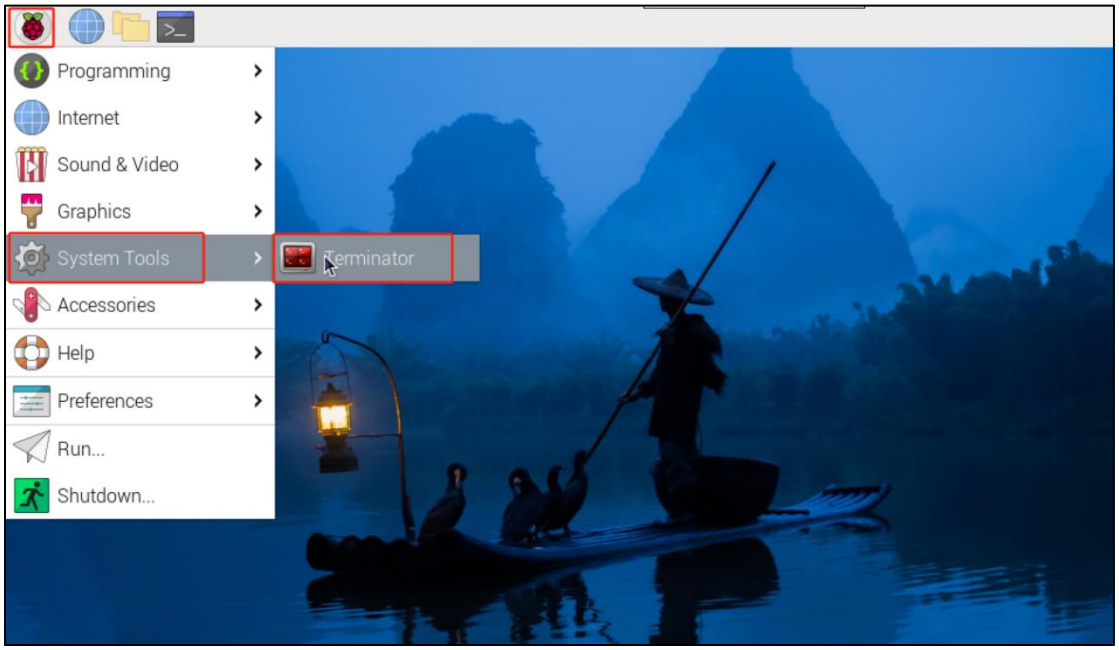
分布式通信的应用场景是较为广泛的，无人车编队、无人机编队、远程控制等等，这些数据的交互都依赖于分布式通信。

2.ROS2 分布式网络分组

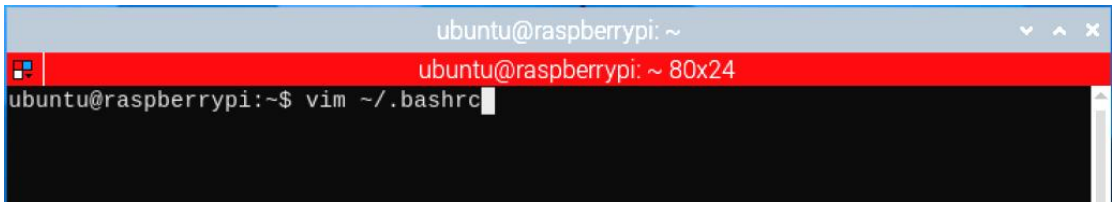
ROS2 提供了一个 DOMAIN 的机制，就类似分组一样，处于同一个 DOMAIN 中的终端才能通信。

所有的 ROS2 节点默认使用 DOMAIN 的域 ID 为 0，为避免消息混淆，同网络内运行 ROS 2 的不同组的设备应该使用不同的 DOMAIN 的域 ID，正常使用推荐[0,101]之间选择即可。

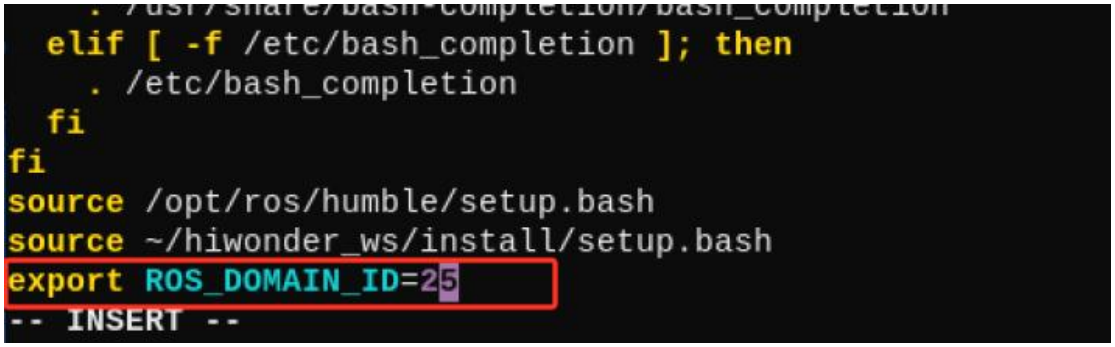
1） 点击左上角的 logo，选择 System Tools →Terminator。



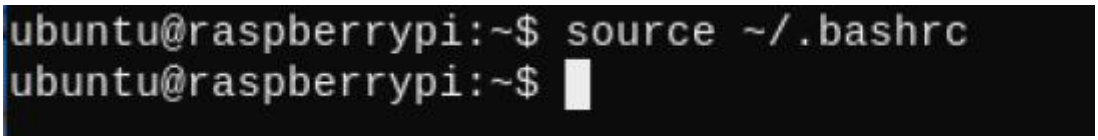
2） 输入指令“**vim ~/.bashrc**”编辑.bashrc 文件。



3） 按下“i”键，在对应的位置输入“**export ROS_DOMAIN_ID=25**”。如需修改，再按下“**i**”即可修改。修改完成，按下“**Esc**”，输入“**：wq**”保存并退出。



4） 输入“**source ~/.bashrc**”指令，让环境变量生效。



在其他终端的.bashrc 文件中加入“**export ROS_DOMAIN_ID=25**”语句，即可将两者分配到同一个小组中（即域 ID25）中，可以进行分布式通信。如果分配的域 ID 不同，则两者无法实现通信。

3.ROS1 与 ROS2 在分布式通信机制上的对比

ROS1 的分布式通信机制：

- 使用匿名的 TCP/IP 通信进行分布式通信。
- 通信采用发布/订阅模式,通过 central master 实现。
- 节点需要主动向 master 注册以获取对等节点信息。
- 通信可靠性依赖于中心节点 master。

ROS2 的分布式通信机制：

- 使用常见传输协议如 TCP、UDP、DDS 等多种方式。
- 引入了带有 discover、authentication 和 authorization 机制的分布式服务发现。
- 节点直接与对等节点通信,无需通过中心 master。
- 默认采用 ZeroMQ 搭建通信中间件,实现去中心化的分布式通信。
- 同时支持点对点及组播通信,扩展性更好。
- 容错能力更强,即使部分节点掉线也可以保持通信。

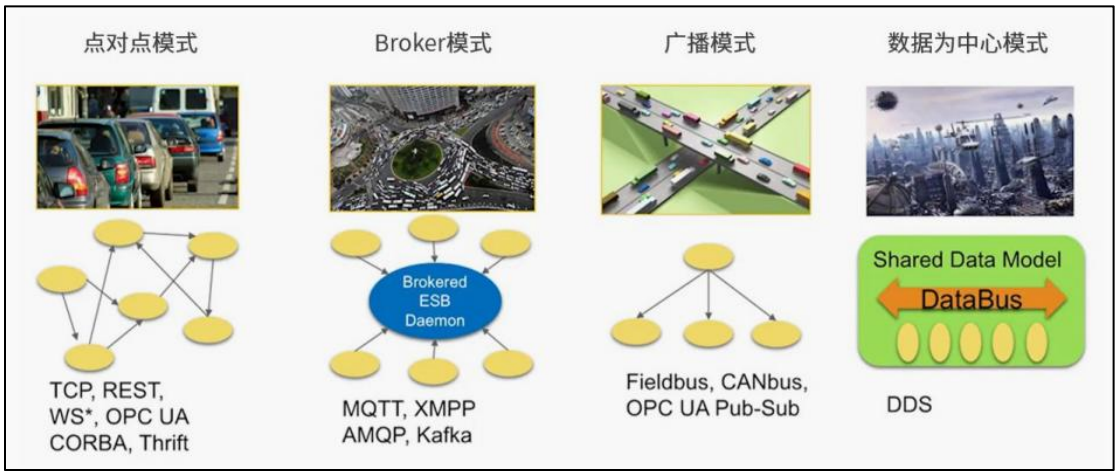
第 14 课 ROS2 DDS 说明

1.DDS 简介

DDS 的全称是 Data Distribution Service，也就是数据分发服务，2004 年由对象管理组织 OMG 发布和维护，是一套专门为实时系统设计的数据分发/订阅标准，最早应用于美国海军，解决舰船复杂网络环境中大量软件升级的兼容性问题，现在已经成为强制标准。

DDS 强调以数据为中心，可以提供丰富的服务质量策略，以保障数据进行实时、高效、灵活地分发，可满足各种分布式实时通信应用需求。

在前面课程中学习的话题、服务、动作，他们底层通信的具体实现过程，都是靠 DDS 来完成的，它相当于 ROS 机器人系统中的神经网络。常见的通信模型有以下四种：



1） 点对点模型，许多客户端连接到一个服务端，每次通信时，通信双方必须建立一条连接。当通信节点增多时，连接数也会增多。而且每个客户端都需要知道服务器的具体地址和所提供的服务，一旦服务器地址发生变化，所有客户端都会受到影响。

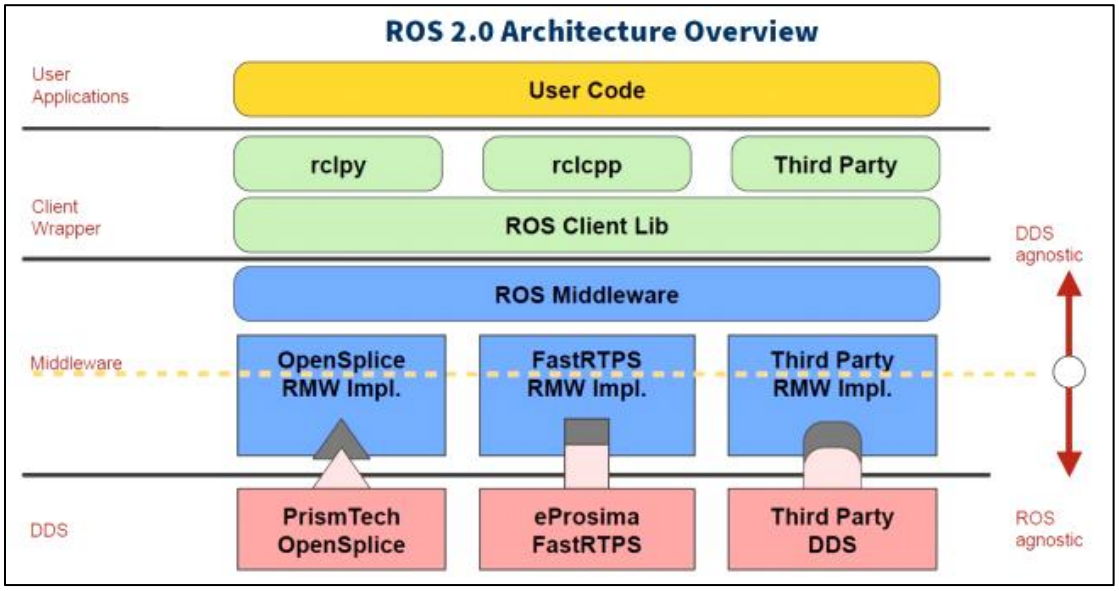
2） Broker 模型，针对点对点模型进行了优化，由 Brocker 集中处理所有人的请求，并进一步找到真正能响应该服务的角色。这样客户端就不用关心服务器的具体地址了。不过问题也很明显，Broker 作为核心，它的处理速度会影响所有节点的效率，当系统规模增长到一定程序，Broker 就会成为整个系统的性能瓶颈。更麻烦的是，如果 Broker 发生异常，可能导致整个系统都无法正常运转。之前的 ROS1 系统，使用的就是类似这样的架构。

3） 广播模型，所有节点都可以在通道上广播消息，并且节点都可以收到消息。这个模型解决了服务器地址的问题，而且通信双方也不用单独建立连接，但是广播通道上的消息太多了，所有节点都必须关心每条消息，其实很多是和自己没有关系的。

4） 以数据为中心的 DDS 模型，这种模型与广播模型有些类似，所有节点都可以在 DataBus 上发布和订阅消息。但它的先进之处在于，通信中包含了很多并行的通路，每个节点可以只关心自己感兴趣的消息，忽略不感兴趣的消息，有点像是一个旋转火锅，各种好吃的都在这个 DataBus 传送，我们只需要拿自己想吃的就行，其他的和我们没有关系。

2.DDS 在 ROS2 中的应用

DDS 在 ROS2 系统中的位置至关重要，所有上层建设都建立在 DDS 之上。在这个 ROS2 的架构图中，蓝色和红色部分就是 DDS。



在 ROS 的四大组成部分中，由于 DDS 的加入，大大提高了分布式通信系统的综合能力，这样我们在开发机器人的过程中，就不需要纠结通信的问题，可以把更多时间放在其他部分的应用开发上。

3.质量服务策略 Qos

DDS 中的基本结构是 Domain，Domain 将各个应用程序绑定在一起进行通信。DDS 中另外一个重要特性就是质量服务策略：Qos。

Qos 是一种网络传输策略，应用程序指定所需要的网络传输质量行为，Qos 服务实现这种行为要求，尽可能地满足客户对通信质量的需求，可以理解为数据提供者和接收者之间的合约。策略如下：

◆ DEADLINE 策略，表示通信数据必须要在每次截止时间内完成一次通信；

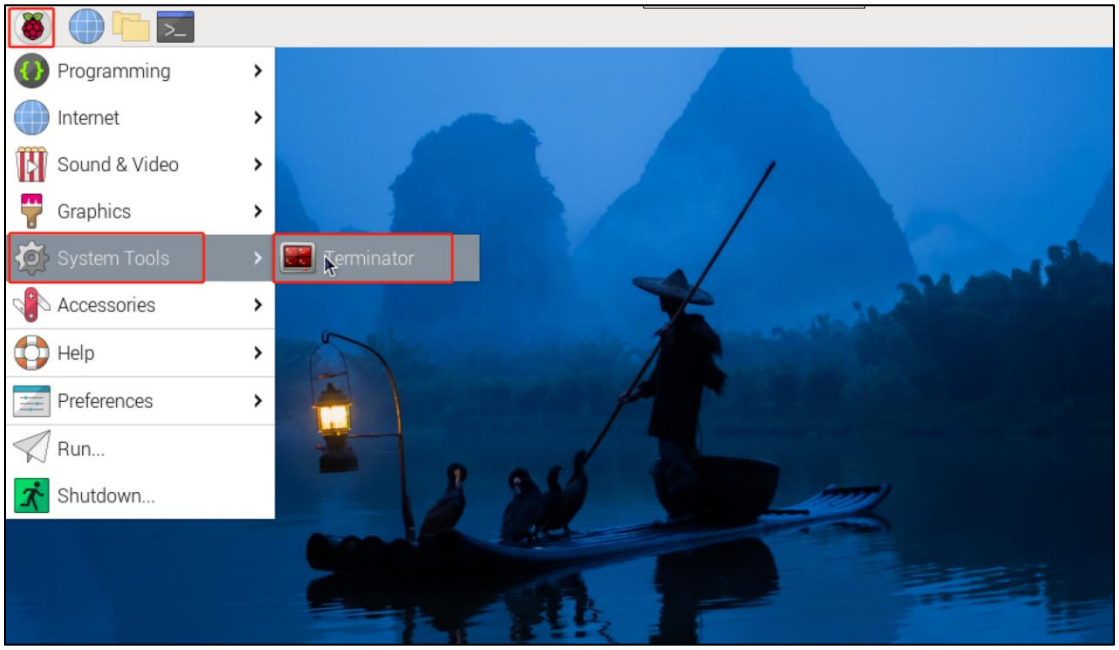
◆ HISTORY 策略，表示针对历史数据的一个缓存大小；

◆ RELIABILITY 策略，表示数据通信的模式，配置成 BEST_EFFORT，就是尽力传输模式，网络情况不好的时候，也要保证数据流畅，此时可能会导致数据丢失，配置成 RELIABLE，就是可信赖模式，可以在通信中尽量保证图像的完整性，我们可以根据应用功能场景选择合适的通信模式；

◆ DURABILITY 策略，可以配置针对晚加入的节点，也保证有一定的历史数据发送过去，可以让新节点快速适应系统。

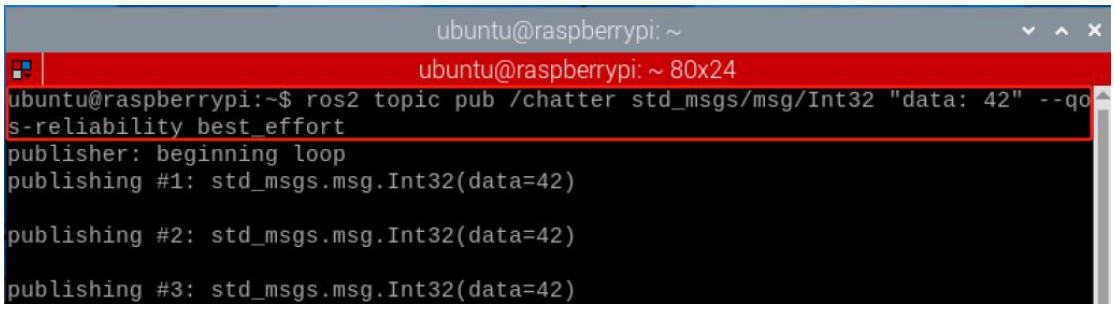
4.命令行中配置 DDS

1） 点击左上角的 logo，选择 System Tools →Terminator。

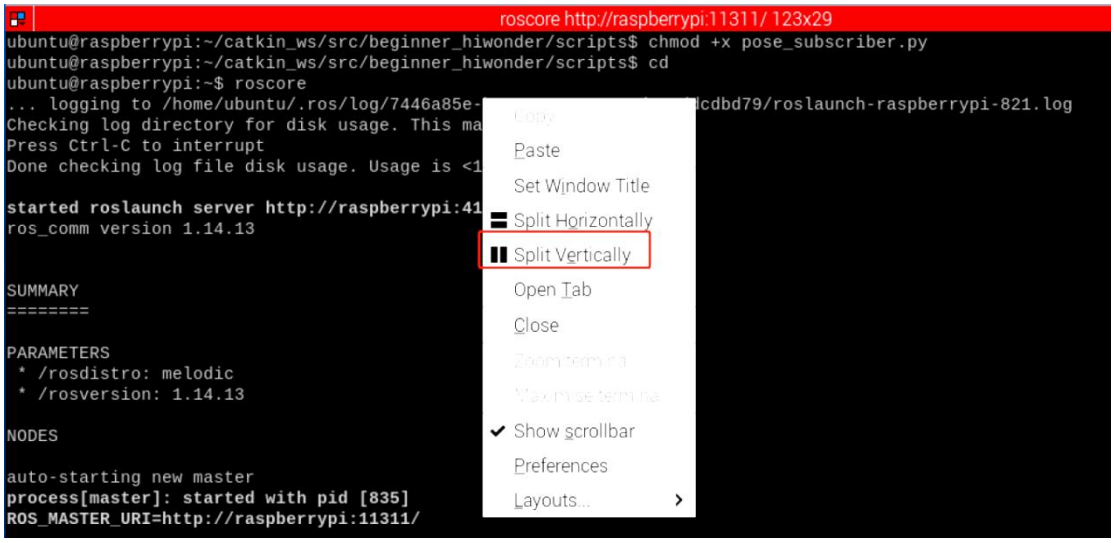


2） 输入“**ros2 topic pub /chatter std_msgs/msg/Int32 "data: 42" --qos-reliability best_effort**”指令，发布一个名为"/chatter"的话题，使用 std_msgs/

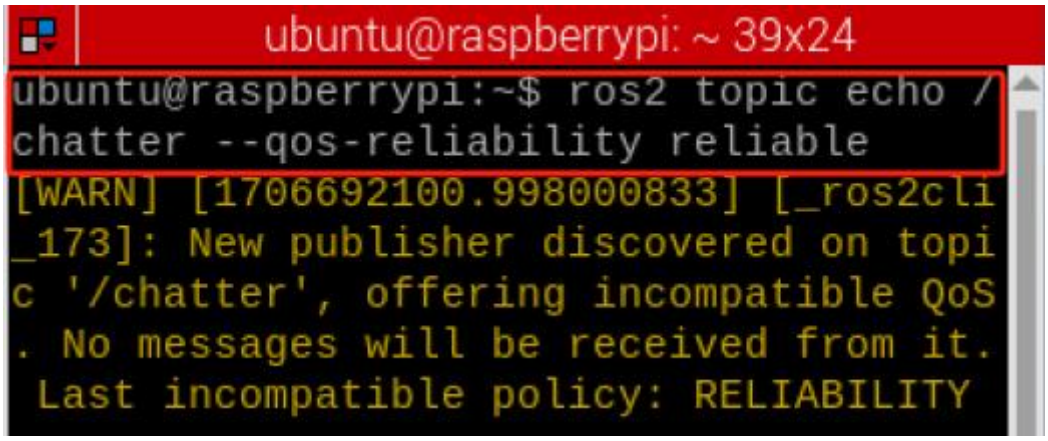
msg/Int32 消息类型，并发送一个数据为 42 的整数消息。通过添加"--qos-reliability best_effort"选项，指定了发布者使用最佳尽力传输（best effort）的可靠性。



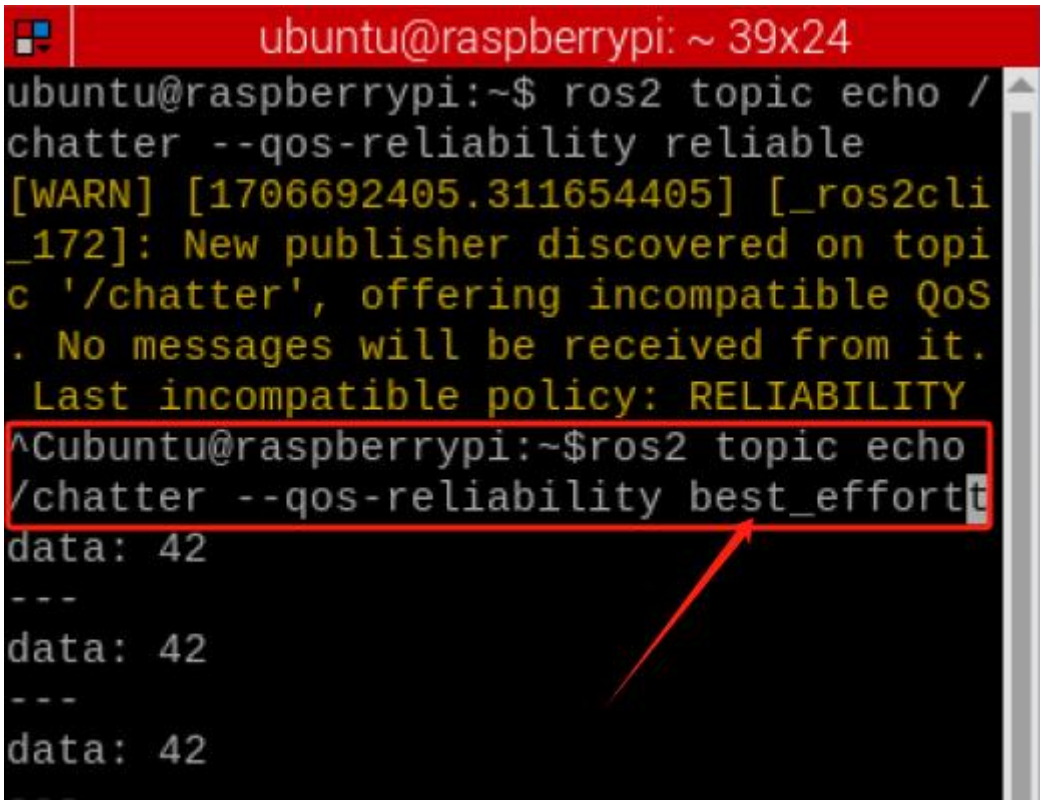
3） 鼠标右键，选择“**Split Vertically**”点击，新建一个新的终端窗口。



4） 输入“**ros2 topic echo /chatter --qos-reliability reliable**”指令，订阅一个名为"/chatter"的话题，并打印出接收到的消息。如果发布端使用 reliableQoS 策略发布,而接收端使用 best_effort 策略订阅,则无法实现数据通信。只有当发布端和接收端使用相同的 QoS 策略时,才能保证数据的正确传输。



5） 输入“**ros2 topic echo /chatter --qos-reliability best_effort**”指令，订阅一个名为"/chatter"的话题，并打印出接收到的消息。通过添加"--qos-reliability reliable"选项

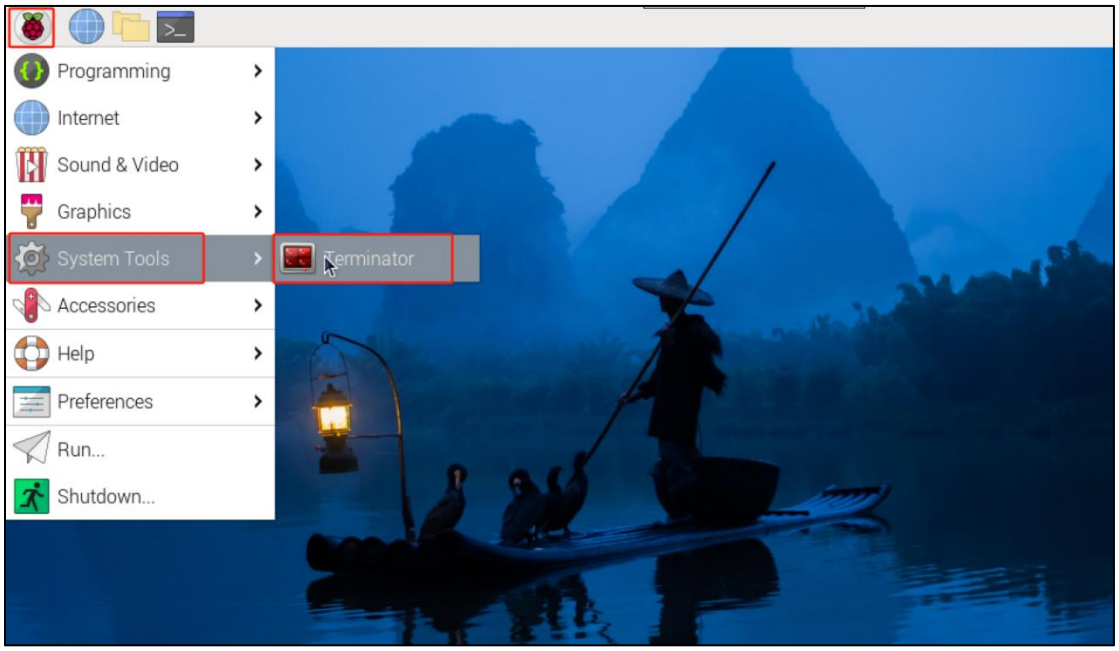


修改为同样的 best_effort，才能实现数据传输。

5.DDS 编程示例

5.1 创建发布者

1） 点击左上角的 logo，选择 System Tools →Terminator。



2） 输入“**cd hiwonder_ws/src/**”指令，切换到 hiwonder_ws 工作空间的 src 文件夹中。

```
ubuntu@raspberrypi:~$ cd hiwonder_ws/src/
ubuntu@raspberrypi:~/hiwonder_ws/src$
```

3） 输入“**ros2 pkg create DDS_qos_demo --build-type ament_python --dependencies rclpy**”并按下回车，创建一个名为“**DDS_qos_demo**”的功能包，添加依赖关系 rclpy。

```
ubuntu@raspberrypi:~/hiwonder_ws/src$ ros2 pkg create DDS_qos_demo --build-type
ament_python --dependencies rclpy
```

4） 输入“**cd DDS_qos_demo/DDS_qos_demo/**”指令，切换到 DDS_qos_demo 功能包下。

```
ubuntu@raspberrypi:~/hiwonder_ws/src$ cd DDS_qos_demo/DDS_qos_demo/
ubuntu@raspberrypi:~/hiwonder_ws/src/DDS_qos_demo/DDS_qos_demo$
```

5） 输入指令“**vim DDS_qos_pub.py**”编辑程序，复制下面程序。如需修改，再按下“**i**”即可修改。修改完成，按下“**Esc**”，输入“**：wq**”保存并退出。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/DDS_qos_demo/DDS_qos_demo$ vim DDS_qos_pub.
py
```

```
import rclpy  # 导入 rclpy 模块
from rclpy.node import Node  # 导入 Node 类


from std_msgs.msg import String  # 导入 String 消息类型
from rclpy.qos import QoSProfile, QoSReliabilityPolicy, QoSHistoryPolicy
  # 导入 QoSProfile 和 QoSReliabilityPolicy、QoSHistoryPolicy 类


class MinimalPublisher(Node):  # 定义一个继承自 Node 类的 MinimalPublisher 类


    def __init__(self):
        super().__init__('minimal_publisher')  # 调用父类构造函数初始化节点
        qos_profile = QoSProfile(  # 创建 QoSProfile 对象
            reliability=QoSReliabilityPolicy.RELIABLE,  # 设置可靠性策略为
RELIABLE
            history=QoSHistoryPolicy.KEEP_LAST,  # 设置历史策略为 KEEP_LAST
            depth=1  # 设置深度为 1
        )
        self.publisher_ = self.create_publisher(String, 'topic', qos_prof
ile)  # 创建发布者对象
        timer_period = 0.5  # 设置定时器回调的时间间隔为 0.5 秒
```

```
        self.timer = self.create_timer(timer_period, self.timer_callback)
  # 创建定时器，设置回调函数为 timer_callback
        self.i = 0


    def timer_callback(self):
        msg = String()  # 创建 String 消息类型的对象
        msg.data = 'Hello World: %d' % self.i  # 设置消息的数据
        self.publisher_.publish(msg)  # 发布消息
        self.i += 1  # 自增计数器



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

    minimal_publisher = MinimalPublisher()  # 创建 MinimalPublisher 对象

    rclpy.spin(minimal_publisher)  # 进入主循环

    minimal_publisher.destroy_node()  # 销毁节点
    rclpy.shutdown()  # 关闭 ROS

if __name__ == '__main__':
```

```
    main()
```

```
# 如果该脚本是主程序，则执行main函数
if __name__ == '__main__':
    main()
:wq
```

6） 输入指令“**chmod +x DDS_qos_pub.py**”，并按下回车，为保存的 topic_pub.py 赋予可执行权限。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/DDS_qos_demo/DDS_qos_demo$ chmod +x DDS_qos
_pub.py
ubuntu@raspberrypi:~/hiwonder_ws/src/DDS_qos_demo/DDS_qos_demo$
```

5.2 创建订阅者

1） 输入指令“**vim DDS_qos_sub.py**”编辑程序，复制下面程序。如需修改，再按下“**i**”即可修改。修改完成，按下“**Esc**”，输入“**：wq**”保存并退出。

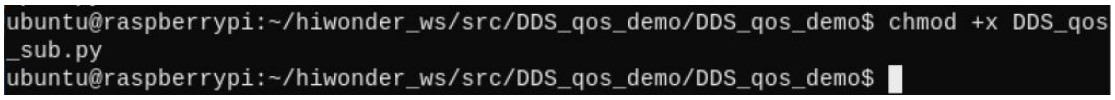
```
ubuntu@raspberrypi:~/hiwonder_ws/src/DDS_qos_demo/DDS_qos_demo$ vim DDS_qos_sub.
py
```

```
import rclpy  # 导入 rclpy 模块

from rclpy.node import Node  # 导入 Node 类



from std_msgs.msg import String  # 导入 String 消息类型

from rclpy.qos import QoSProfile, QoSReliabilityPolicy, QoSHistoryPolicy
  # 导入 QoSProfile 和 QoSReliabilityPolicy、QoSHistoryPolicy 类



class MinimalSubscriber(Node):  # 定义一个继承自 Node 类的 MinimalSubscriber
类



    def __init__(self):
```

```
        super().__init__('minimal_subscriber')  # 调用父类构造函数初始化节点

        qos_profile = QoSProfile(  # 创建 QoSProfile 对象

            reliability=QoSReliabilityPolicy.RELIABLE,  # 设置可靠性策略为 RELIABLE

            history=QoSHistoryPolicy.KEEP_LAST,  # 设置历史策略为 KEEP_LAST

            depth=1  # 设置深度为 1

        )

        self.subscription = self.create_subscription(String, 'topic', self.listener_callback, qos_profile)  # 创建订阅者对象，并设置回调函数为 listener_callback



    def listener_callback(self, msg):

        self.get_logger().info('I heard: "%s"' % msg.data)  # 打印接收到的消息内容




def main(args=None):

    rclpy.init(args=args)  # 初始化 ROS 节点



    minimal_subscriber = MinimalSubscriber()  # 创建MinimalSubscriber对象



    rclpy.spin(minimal_subscriber)  # 进入主循环
```

```
    minimal_subscriber.destroy_node()  # 销毁节点
    rclpy.shutdown()  # 关闭 ROS


if __name__ == '__main__':
    main()
```

2） 输入指令“**chmod +x DDS_qos_sub.py**”，并按下回车，为保存的 topic_sub.py 赋予可执行权限。



3.setup.py 文件设置

setup.py 文件是定义一个 ROS2 包的远数据和构建配置，提供了包的元数据、依赖关系、构建配置和安装逻辑等信息，可以帮助开发者正确构建、安装和使用 ROS2 包。需要将 **topic_pub.py** 和 **topic_sub.py** 的程序入口写到 setup.py 文件中。

1） 输入“**cd ..**”指令，切换到上一级目录。

2） 输入“**vim setup.py**”指令，并按下回车，打开 setup.py 文件。

3） 按下“i”键进入可编辑模式，输入以下代码到相应的位置上。

```
'DDS_qos_pub = DDS_qos_demo.DDS_qos_pub:main',

'DDS_qos_sub = DDS_qos_demo.DDS_qos_sub:main'
```

```
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'DDS_qos_pub = DDS_qos_demo.DDS_qos_pub:main',
            'DDS_qos_sub = DDS_qos_demo.DDS_qos_sub:main'
        ],
    },
)
-- INSERT --                                          27,1          Bot
```

4） 然后输入“:wq”，保存并退出文件。

```
        'console_scripts': [
            'DDS_qos_pub = DDS_qos_demo.DDS_qos_pub:main',
            'DDS_qos_sub = DDS_qos_demo.DDS_qos_sub:main'
        ],
    },
)
:wq
```

6.编译运行

1） 赋予可执行权限后，输入“**cd ~/hiwonder_ws/**”指令，切换到工作空间的目录。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/hello_world_demo$ cd ~/hiwonder_ws/
ubuntu@raspberrypi:~/hiwonder_ws$
```

2） 输入“**colcon build**”指令，并按下回车，编译工作空间中的功能包。

```
ubuntu@raspberrypi:~/hiwonder_ws$ colcon build
```

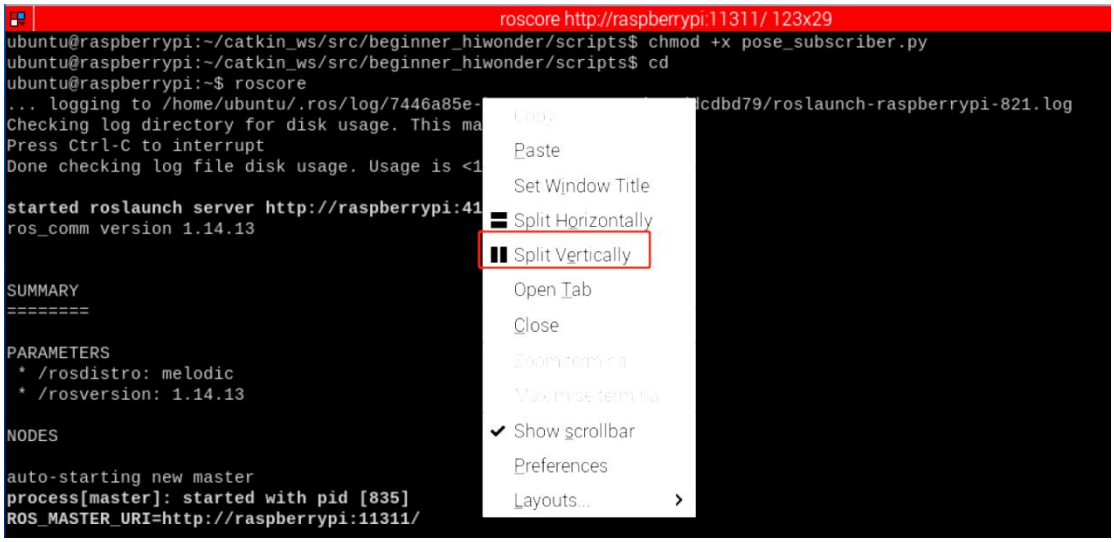
3） 再输入指令“**source ./install/setup.bash**”，并按下回车，让环境变量生效。

```
ubuntu@raspberrypi:~/hiwonder_ws$ source ./install/setup.bash
```

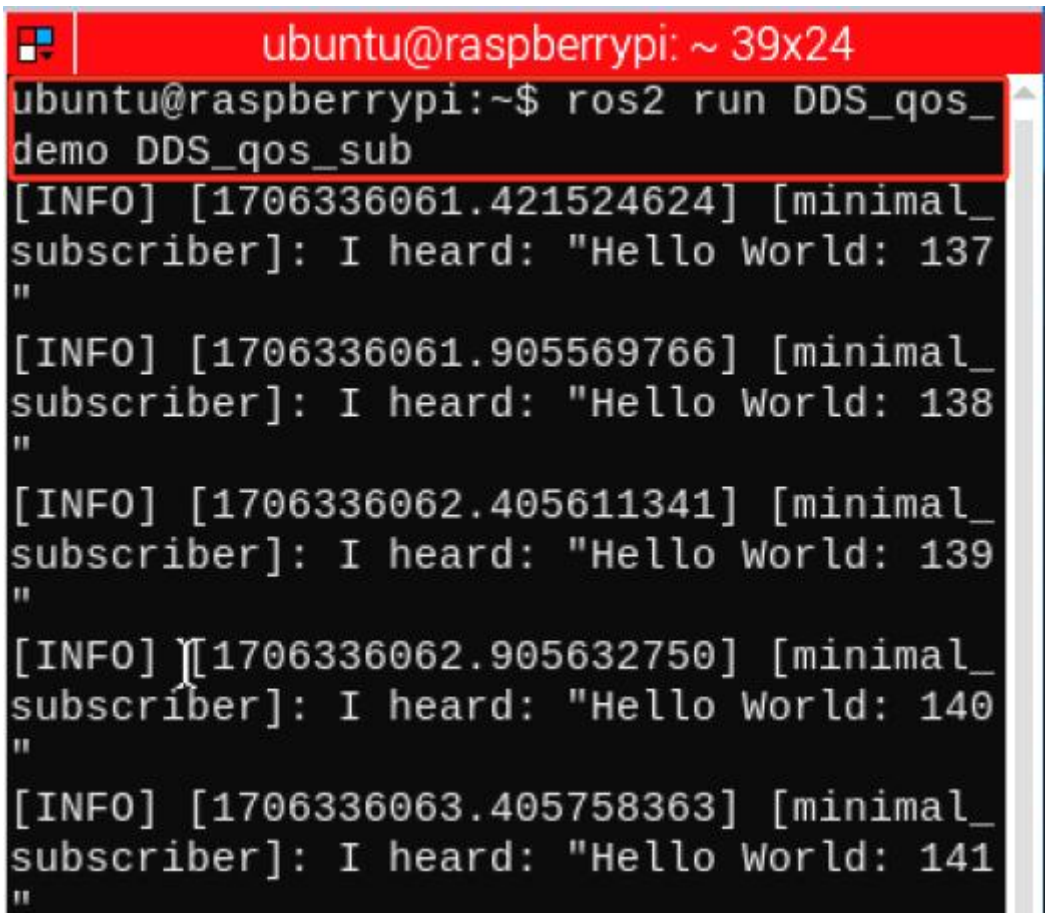
4） 输入指令“**ros2 run DDS_qos_demo DDS_qos_pub**”，并按下回车，启动 DDS_qos_pub 话题发布节点。

```
ubuntu@raspberrypi:~$ ros2 run DDS_qos_demo DDS_qos_pub
```

5） 鼠标右键，选择“**Split Vertically**”点击，新建一个新的终端窗口。



6） 输入指令“**ros2 run DDS_qos_demo DDS_qos_sub**”，并按下回车，启动 DDS_qos_sub 话题订阅节点。

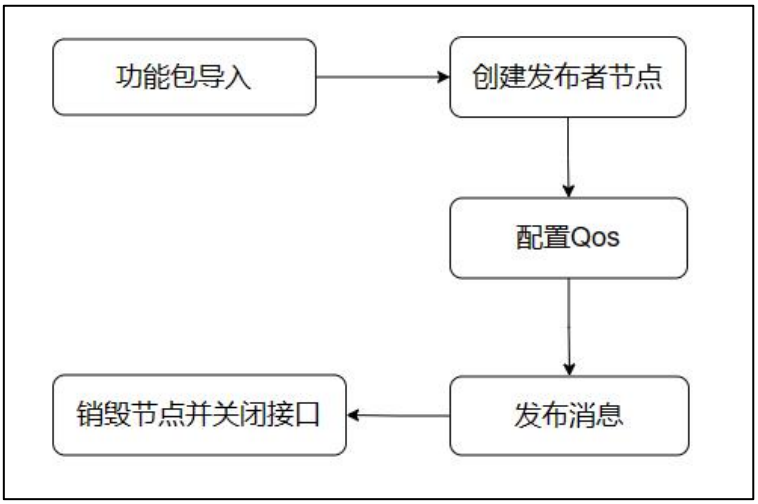


7）

7.程序分析

7.1 话题发布

根据实现的效果，梳理程序的过程逻辑，如下图所示：



创建了一个 MinimalPublisher 节点类,在构造函数中创建了一个 QosProfile 对象,配置发布为可靠模式,保证最后一次或第一条消息,并设置缓冲深度为 1。然后创建了一个 0.5 秒周期的定时器对象。定时器回调函数 timer_callback 中,会构造 String 类型消息,随着循环计数器 i 增加,改变消息内容。并通过发布者对象 publisher_发布这条消息字符串。主函数首先初始化 ROS 节点环境,然后创建节点实例,进入主循环来驱动定时任务的执行。定时任务使发布内容能循环发送。最后清理节点资源。

- 主函数

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

    minimal_publisher = MinimalPublisher()  # 创建MinimalPublisher对象

    rclpy.spin(minimal_publisher)  # 进入主循环

    minimal_publisher.destroy_node()  # 销毁节点
    rclpy.shutdown()  # 关闭ROS
```

首先调用 rclpy.init()对 ROS2 Python 接口进行初始化，然后实例化 MinimalPublisher()，然后在 ROS2 节点的事件循环执行 minimal_publisher。

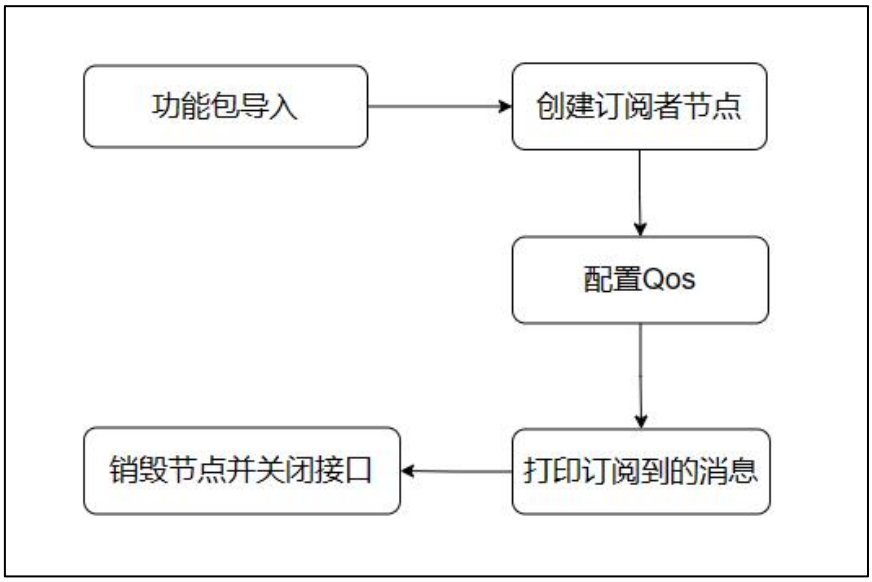
◆ MinimalPublisher 类

```
class MinimalPublisher(Node):  # 定义一个继承自Node类的MinimalPublisher类

    def __init__(self):
        super().__init__('minimal_publisher')  # 调用父类构造函数初始化节点
        qos_profile = QoSProfile(  # 创建QoSProfile对象
            reliability=QoSReliabilityPolicy.RELIABLE,  # 设置可靠性策略为RELIABLE
            history=QoSHistoryPolicy.KEEP_LAST,  # 设置历史策略为KEEP_LAST
            depth=1  # 设置深度为1
        )
        self.publisher_ = self.create_publisher(String, 'topic', qos_profile)  # 创建发布者对象
        timer_period = 0.5  # 设置定时器回调的时间间隔为0.5秒
        self.timer = self.create_timer(timer_period, self.timer_callback)  # 创建定时器，设置回调函数为timer_callback
        self.i = 0

    def timer_callback(self):
        msg = String()  # 创建String消息类型的对象
        msg.data = 'Hello World: %d' % self.i  # 设置消息的数据
        self.publisher_.publish(msg)  # 发布消息
        self.i += 1  # 自增计数器
```

首先创建了一个 MinimalPublisher 节点类,用于实现 ROS2 可靠模式下的循环发布。在构造函数__init__()中,首先创建了一个 QosProfile 对象,将发布质量配置为可靠模式;然后通过 QosProfile 对创建的发布者对象 publisher_进行了初始化。构造函数还设置了定时器周期 0.5s,并创建定时器对象。并将计数器 i 初始化为 0。定义了定时器回调函数 timer_callback,在函数体中,首先构造了 String 类型的消息对象 msg,并利用字符串格式化将当前计数 i 值写入消息内容。然后通过之前定义的 publisher_对象发布这条消息。最后计数 i 增加 1。

7.2 话题订阅

根据实现的效果，梳理程序的过程逻辑，如下图所示：



创建了一个 MinimalSubscriber 节点类,用于实现 ROS2 下的可靠模式订阅。在构造函数中,它首先使用 QosProfile 对象配置订阅质量,将可靠性设置为 RELIABLE,保留历史记录的最近一条消息,并将缓冲区 depth 设置为 1。然后通过 QosProfile 对象来创建订阅者对象,订阅 topic 并指定回调函数。回调函数 listener_callback 只是简单打印接收到的消息内容。主函数用于初始化节点,创建订阅者实例,进入主循环驱动回调函数,和释放节点资源。通过 QosProfile 的配置,可以保证订阅者在回调中能获得最近一条正确的消息。

◆ 主函数

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

    minimal_subscriber = MinimalSubscriber()  # 创建MinimalSubscriber对象

    rclpy.spin(minimal_subscriber)  # 进入主循环

    minimal_subscriber.destroy_node()  # 销毁节点
    rclpy.shutdown()  # 关闭ROS
```

首先调用 rclpy.init()对 ROS2 Python 接口进行初始化，然后实例化 MinimalSublisher()，然后在 ROS2 节点的事件循环执行 minimal_sublisher。

◆ MinimalSubscriber 类

```python
class MinimalSubscriber(Node):  # 定义一个继承自Node类的MinimalSubscriber类

    def __init__(self):
        super().__init__('minimal_subscriber')  # 调用父类构造函数初始化节点
        qos_profile = QoSProfile(  # 创建QoSProfile对象
            reliability=QoSReliabilityPolicy.RELIABLE,  # 设置可靠性策略为RELIABLE
            history=QoSHistoryPolicy.KEEP_LAST,  # 设置历史策略为KEEP_LAST
            depth=1  # 设置深度为1
        )
        self.subscription = self.create_subscription(String, 'topic', self.listener_callback, qos_profile)  # 创建订阅者对象，并设置回调函数为listener_callback

    def listener_callback(self, msg):
        self.get_logger().info('I heard: "%s"' % msg.data)  # 打印接收到的消息内容
```

首先创建了一个 MinimalSubscriber 节点类,用于实现 ROS2 下的可靠订阅功能。它在构造函数__init__()中,首先创建了 QosProfile 对象,将订阅质量配置为可靠模式(RELIABLE、KEEP_LAST 等)。然后通过 QosProfile 对象和指定的回调函数,创建了订阅对象 subscription,订阅 topic 和指定回调函数。定义了订阅回调函数 listener_callback,这个回调函数仅仅是打印接收到的消息内容 msg。

第 15 课 ROS2 Launch 多节点启动与配置脚本

1.Launch 简介

到目前为止，每当我们运行一个 ROS 节点，都需要打开一个新的终端运行一个命令。机器人系统中节点很多，每次都这样启动好麻烦。有没有一种方式可以一次性启动所有节点呢？答案当然是肯定的，那就是 Launch 启动文件，它是 ROS 系统中多节点启动与配置的一种脚本。

ROS2 中，Launch 用于多节点启动和配置程序运行参数等功能，ROS2 的 Launch 文件格式有 xml、yaml 和 Python 格式。本节课程以 Python 格式的 Launch 文件为例，相对于另外两种格式，Python 格式的更加灵活：

- Python 拥有众多的函数库，可以在启动文件中使用；
- ROS2 通用启动特性和特定启动特性是用 Python 编写的，因此可以访问 XML 和 YAML 可能没有公开的启动特性。

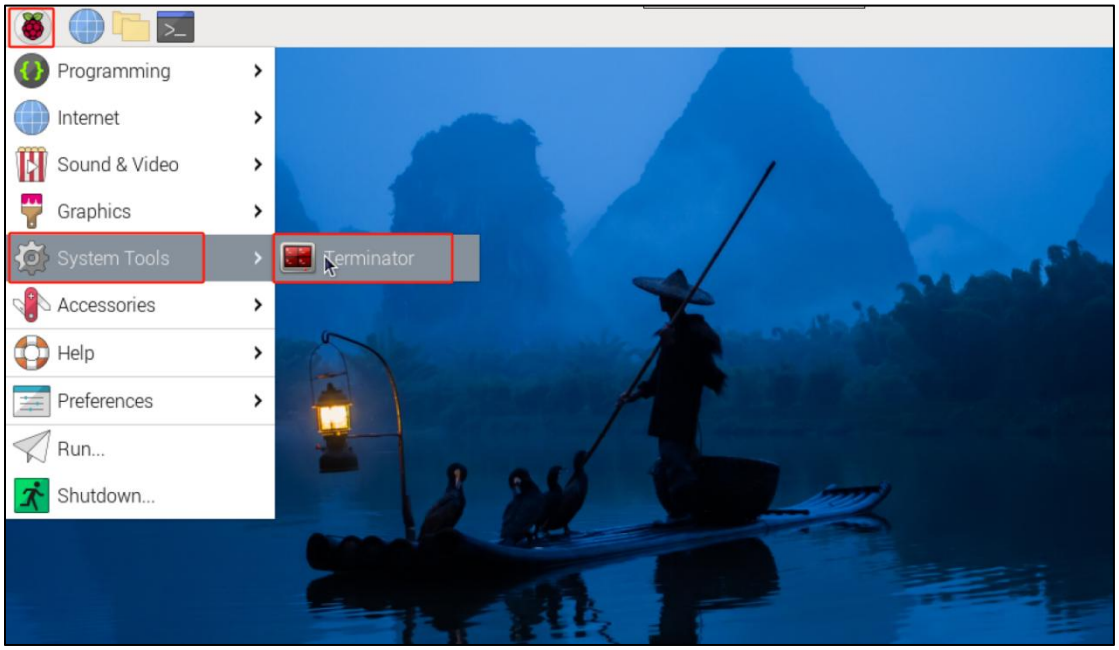
使用 Python 语言编写 ROS2 launch 文件，最主要的是每个节点、文件、脚本等抽象成一个 action，用统一的接口来启动，主要结构是：

```
def generate_launch_description():
    return LaunchDescription([
        action_1,
        action_2,
        ...
        action_n
    ])
```

2.单个节点 Launch 文件

2.1 single_node.launch.py 创建

1） 点击左上角的 logo，选择 System Tools →Terminator。



2） 输入“**cd hiwonder_ws/src/**”指令，切换到 hiwonder_ws 工作空间的 src 文件夹中。

```
ubuntu@raspberrypi:~$ cd hiwonder_ws/src/
ubuntu@raspberrypi:~/hiwonder_ws/src$
```

3） 输入“**ros2 pkg create launch_demo --build-type ament_python --dependencies rclpy**”并按下回车，创建一个名为“**launch_demo**”的功能包，添加依赖关系 rclpy。

```
ubuntu@raspberrypi:~/hiwonder_ws/src$ ros2 pkg create launch_demo --build-type a
ment_python --dependencies rclpy
```

4） 输入“**cd launch_demo/**”指令，切换到 launch_demo 功能包下。

```
ubuntu@raspberrypi:~/hiwonder_ws/src$ cd launch_demo/
ubuntu@raspberrypi:~/hiwonder_ws/src/launch_demo$
```

5） 输入“**mkdir launch**”指令，创建 launch 文件夹。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/launch_demo$ mkdir launch
```

6） 输入“**cd launch**”指令，切换到 launch 文件夹。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/launch_demo$ cd launch
ubuntu@raspberrypi:~/hiwonder_ws/src/launch_demo/launch$
```

7） 输入指令“**vim single_node.launch.py**”编辑程序，复制下面程序。如需修改，再按下“**i**”即可修改。修改完成，按下“**Esc**”，输入“**：wq**”保存并退出。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/topic_demo/topic_demo$ vim topic_pub.py
```

```
from launch import LaunchDescription            # launch 文件的描述类
from launch_ros.actions import Node             # 节点启动的描述类


def generate_launch_description():              # 自动生成 launch 文件的函数
    return LaunchDescription([                  # 返回 launch 文件的描述信息
        Node(                                   # 配置一个节点的启动
            package='hello_world_demo',         # 节点所在的功能包
            executable='hello_world',           # 节点的可执行文件
        ),
    ])
```

```
~
~
~
~
~
~
:wq
```

8） 输入指令“**chmod +x single_node.launch.py**”，并按下回车，为保存的 single_node.launch.py 赋予可执行权限。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/launch_demo/launch$ chmod +x single_node.la
unch.py
ubuntu@raspberrypi:~/hiwonder_ws/src/launch_demo/launch$
```

2.2 setup.py 文件设置

setup.py 文件是定义一个 ROS2 包的远数据和构建配置，提供了包的元数据、依赖关系、构建配置和安装逻辑等信息，可以帮助开发者正确构建、安装和使用 ROS2 包。需要将 **single_node.launch.py** 的程序入口写到 setup.py 文件中。

1） 输入“**cd ..**”指令，切换到上一级目录。

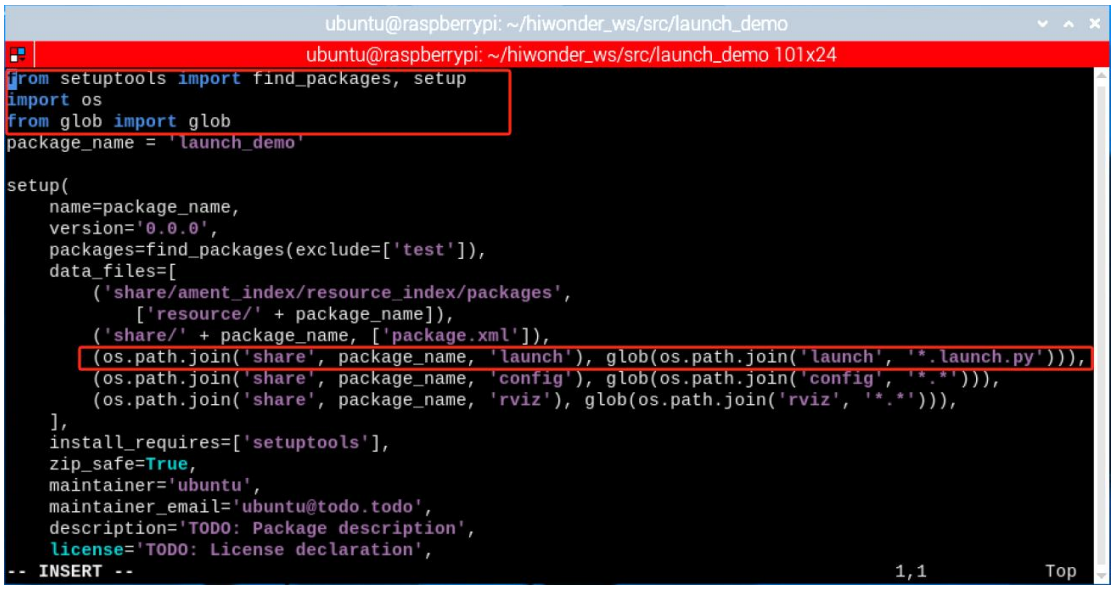
```
ubuntu@raspberrypi:~/hiwonder_ws/src/launch_demo/launch$ cd ..
ubuntu@raspberrypi:~/hiwonder_ws/src/launch_demo$
```

2） 输入“**vim setup.py**”指令，并按下回车，打开 setup.py 文件。

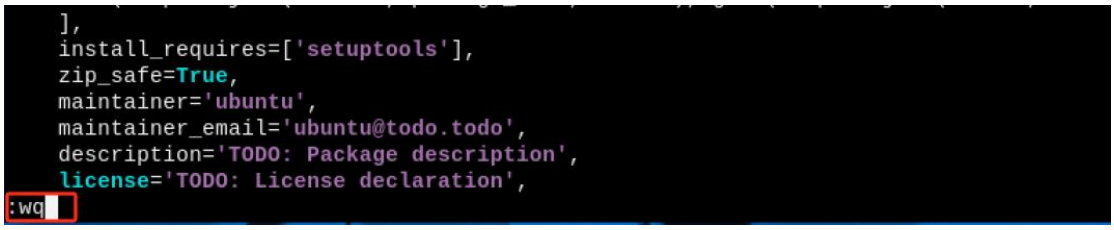
```
ubuntu@raspberrypi:~/hiwonder_ws/src/launch_demo$ vim setup.py
```

3） 按下“i”键进入可编辑模式，输入以下代码到相应的位置上。

```
from setuptools import find_packages, setup

import os

from glob import glob

(os.path.join('share', package_name, 'launch'), glob(os.path.join('launc
h', '*.launch.py'))),
```



4） 然后输入“:wq”，保存并退出文件。



2.3 编译运行

1） 赋予可执行权限后，输入“**cd ~/hiwonder_ws/**”指令，切换到工作空间的目录。

2） 输入“**colcon build**”指令，并按下回车，编译工作空间中的功能包。

3） 再输入指令“**source ./install/setup.bash**”，并按下回车，让环境变量生效。

4） 输入指令“**ros2 launch launch_demo single_node.launch.py**”，并按下回车，启动 single_node.launch.py 文件。

```
ubuntu@raspberrypi:~$ ros2 launch launch_demo single_node.launch.py
[INFO] [launch]: All log files can be found below /home/ubuntu/.ros/log/2024-01-29-02-44-56-330823-raspberrypi-125
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [hello_world-1]: process started with pid [126]
[hello_world-1] [INFO] [1706496296.926882405] [hello_world_demo]: Hello World
[hello_world-1] [INFO] [1706496297.428765081] [hello_world_demo]: Hello World
[hello_world-1] [INFO] [1706496297.930032612] [hello_world_demo]: Hello World
[hello_world-1] [INFO] [1706496298.431273776] [hello_world_demo]: Hello World
```

2.4 程序分析

通过 Node 描述一个 hello_world 节点的启动任务,包括节点所在的包名和执行文件,并返回 LaunchDescription 对象包含该节点任务,实现生成一个 launch 文件以描述和启动 hello_world 节点。

```
from launch import LaunchDescription                    # launch文件的描述类
from launch_ros.actions import Node                     # 节点启动的描述类

def generate_launch_description():                      # 自动生成launch文件的函数
    return LaunchDescription([                          # 返回launch文件的描述信息
        Node(                                           # 配置一个节点的启动
            package='hello_world_demo',                 # 节点所在的功能包
            executable='hello_world',                   # 节点的可执行文件
        ),
    ])
```

3.多个节点 Launch 文件

3.1 multi_node.launch.py 创建

1） 输入“**cd hiwonder_ws/src/launch_demo/launch**”指令，切换到 launch_demo 工作空间的 launch 文件夹中。

```
ubuntu@raspberrypi:~$ cd hiwonder_ws/src/launch_demo/launch
ubuntu@raspberrypi:~/hiwonder_ws/src/launch_demo/launch$
```

2） 输入指令“**vim multi_node.launch.py**”编辑程序，复制下面程序。如需修改，再按下“**i**”即可修改。修改完成，按下“**Esc**”，输入“**: wq**”保存并退出。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/topic_demo/topic_demo$ vim topic_pub.py
```

```
from launch import LaunchDescription                    # launch 文件的描述类

from launch_ros.actions import Node                     # 节点启动的描述类
```

```
def generate_launch_description():                # 自动生成 launch 文件的函数
    return LaunchDescription([                    # 返回 launch 文件的描述信息
        Node(                                     # 配置一个节点的启动
            package='topic_demo',                 # 节点所在的功能包
            executable='topic_pub',               # 节点的可执行文件
        ),
        Node(                                     # 配置一个节点的启动
            package='topic_demo',                 # 节点所在的功能包
            executable='topic_sub',               # 节点的可执行文件名
        ),
    ])
```

3) 输入指令“**chmod +x multi_node.launch.py**”，并按下回车，为保存的 multi_node.launch.py 赋予可执行权限。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/launch_demo/launch$ chmod +x multi_node.launch.py
ubuntu@raspberrypi:~/hiwonder_ws/src/launch_demo/launch$
```

3.2 编译运行

1) 赋予可执行权限后，输入“**cd ~/hiwonder_ws/**”指令，切换到工作空间的目录。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/hello_world_demo$ cd ~/hiwonder_ws/
ubuntu@raspberrypi:~/hiwonder_ws$
```

2) 输入“**colcon build**”指令，并按下回车，编译工作空间中的功能包。

```
ubuntu@raspberrypi:~/hiwonder_ws$ colcon build
```

3） 再输入指令“**source ./install/setup.bash**”，并按下回车，让环境变量生效。

```
ubuntu@raspberrypi:~/hiwonder_ws$ source ./install/setup.bash
```

4） 输入指令“**ros2 launch launch_demo multi_node.launch.py**”，并按下回车，启动 multi_node.launch.py 文件。

```
ubuntu@raspberrypi:~$ ros2 launch launch_demo multi_node.launch.py
[INFO] [launch]: All log files can be found below /home/ubuntu/.ros/log/2024-01-29-02-59-33-472130-raspberrypi-160
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [topic_pub-1]: process started with pid [161]
[INFO] [topic_sub-2]: process started with pid [163]
[topic_sub-2] [INFO] [1706497174.504283132] [minimal_subscriber]: I heard: "Hello World: 0"
[topic_sub-2] [INFO] [1706497174.988132575] [minimal_subscriber]: I heard: "Hello World: 1"
[topic_sub-2] [INFO] [1706497175.488120599] [minimal_subscriber]: I heard: "Hello World: 2"
[topic_sub-2] [INFO] [1706497175.988234457] [minimal_subscriber]: I heard: "Hello World: 3"
[topic_sub-2] [INFO] [1706497176.488416704] [minimal_subscriber]: I heard: "Hello World: 4"
[topic_sub-2] [INFO] [1706497176.988116785] [minimal_subscriber]: I heard: "Hello World: 5"
[topic_sub-2] [INFO] [1706497177.488334477] [minimal_subscriber]: I heard: "Hello World: 6"
[topic_sub-2] [INFO] [1706497177.988232411] [minimal_subscriber]: I heard: "Hello World: 7"
```

3.3 程序分析

通过 Node 描述了两个 topic_demo 包下的节点启动任务(一个发布节点和一个订阅节点),并将两个节点任务加入到 LaunchDescription 对象中返回,生成一个描述式的 launch 文件自动启动这两个 topic 交互的节点任务。

```python
from launch import LaunchDescription                    # launch文件的描述类
from launch_ros.actions import Node                     # 节点启动的描述类

def generate_launch_description():                      # 自动生成launch文件的函数
    return LaunchDescription([                          # 返回launch文件的描述信息
        Node(                                           # 配置一个节点的启动
            package='topic_demo',                       # 节点所在的功能包
            executable='topic_pub',                     # 节点的可执行文件
        ),
        Node(                                           # 配置一个节点的启动
            package='topic_demo',                       # 节点所在的功能包
            executable='topic_sub',                     # 节点的可执行文件名
        ),
    ])
```

4.重映射 Launch 文件

ROS 社区中的资源非常多，当使用别人代码的时候，经常会发现通信的话题名称不太符合我们的要求，能否对类似的资源重新命名呢？为了提高软件的复用性，ROS 提供了资源重

映射的机制。

4.1 remapping.launch.py 创建

1） 输入“**cd hiwonder_ws/src/launch_demo/launch**”指令，切换到 launch_demo 工作空间的 launch 文件夹中。

```
ubuntu@raspberrypi:~$ cd hiwonder_ws/src/launch_demo/launch
ubuntu@raspberrypi:~/hiwonder_ws/src/launch_demo/launch$
```

2） 输入指令“**vim remapping.launch.py**”编辑程序，复制下面程序。如需修改，再按下“**i**”即可修改。修改完成，按下“**Esc**”，输入“**：wq**”保存并退出。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/launch_demo/launch$ vim remapping.launch.py
```

```
from launch import LaunchDescription            # launch 文件的描述类
from launch_ros.actions import Node             # 节点启动的描述类


def generate_launch_description():              # 自动生成 launch 文件的函数
    return LaunchDescription([                  # 返回 launch 文件的描述信息
        Node(                                   # 配置一个节点的启动
            package='topic_demo',           # 节点所在的功能包
            executable='topic_pub', # 节点的可执行文件
            remappings=[("/topic","topic_pub")]
        ),
    ])
```

3） 输入指令“**chmod +x remapping.launch.py**”，并按下回车，为保存的 remapping.launch.py 赋予可执行权限。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/launch_demo/launch$ chmod +x remapping.launch.py
ubuntu@raspberrypi:~/hiwonder_ws/src/launch_demo/launch$
```

4.2 编译运行

1） 赋予可执行权限后，输入“**cd ~/hiwonder_ws/**”指令，切换到工作空间的目录。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/hello_world_demo$ cd ~/hiwonder_ws/
ubuntu@raspberrypi:~/hiwonder_ws$
```

2） 输入“**colcon build**”指令，并按下回车，编译工作空间中的功能包。

```
ubuntu@raspberrypi:~/hiwonder_ws$ colcon build
```

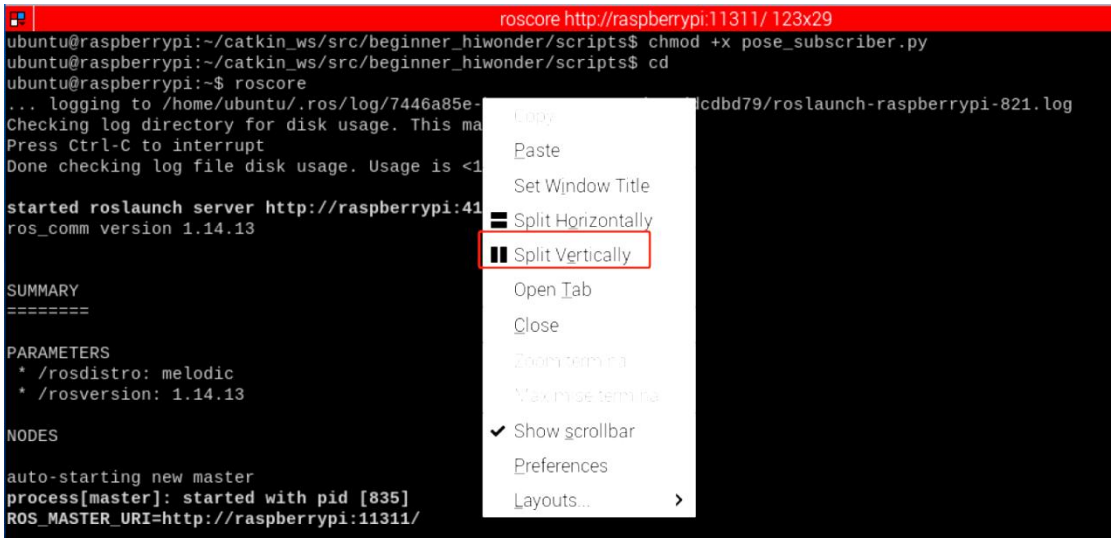
3） 再输入指令“**source ./install/setup.bash**”，并按下回车，让环境变量生效。

```
ubuntu@raspberrypi:~/hiwonder_ws$ source ./install/setup.bash
```

4） 输入指令“**ros2 launch launch_demo remapping.launch.py**”，并按下回车，启动 remapping.launch.py 文件。原先话题名称为“**/topic**”，重映射后话题名称为“**/topic_pub**”。

```
ubuntu@raspberrypi:~$ ros2 launch launch_demo remapping.launch.py
[INFO] [launch]: All log files can be found below /home/ubuntu/.ros/log/2024-01-29-03-23-58-673471-raspberrypi-197
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [topic_pub-1]: process started with pid [198]
```

5） 鼠标右键，选择“**Split Vertically**”点击，新建一个新的终端窗口。



6） 输入“**ros2 topic list**”指令，查看当前话题列表。原先话题名称为“**/topic**”，重映射后话题名称为“**/topic_pub**”。

4.3 程序分析

通过 Node 动作描述了一个从 topic_demo 包下执行 topic_pub 节点的启动任务，并且通过 remappings 参数将节点发布的 topic 名称从"/topic"重映射为"topic_pub"，最后将此节点任务添加到 LaunchDescription 对象并返回，目的就是自动生成一个描述文件的 launch 文件来启动描述中的节点任务，同时实现了话题名称的重映射配置。

```
from launch import LaunchDescription                    # launch文件的描述类
from launch_ros.actions import Node                     # 节点启动的描述类

def generate_launch_description():                      # 自动生成launch文件的函数
    return LaunchDescription([                          # 返回launch文件的描述信息
        Node(                                           # 配置一个节点的启动
            package='topic_demo',                  # 节点所在的功能包
            executable='topic_pub', # 节点的可执行文件
            remappings=[("/topic","topic_pub")]
        ),
    ])
```

5.设置参数 Launch 文件

5.1 param.launch.py 创建

1） 输入“**cd hiwonder_ws/src/launch_demo/launch**”指令，切换到 launch_demo 工作空间的 launch 文件夹中。

```
ubuntu@raspberrypi:~$ cd hiwonder_ws/src/launch_demo/launch
ubuntu@raspberrypi:~/hiwonder_ws/src/launch_demo/launch$
```

2） 输入指令“**vim param.launch.py**”编辑程序，复制下面程序。如需修改，再按下“**i**”即可修改。修改完成，按下“**Esc**”，输入“**：wq**”保存并退出。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/launch_demo/launch$ vim param.launch.py
```

```
from launch import LaunchDescription                    # launch 文件的描述类
from launch.actions import DeclareLaunchArgument       # 声明 launch 文件内使用的 Argument 类
from launch.substitutions import LaunchConfiguration, TextSubstitution


from launch_ros.actions import Node                     # 节点启动的描述类



def generate_launch_description():                      # 自动生成 launch 文件的函数
   background_r_launch_arg = DeclareLaunchArgument(
      'background_r', default_value=TextSubstitution(text='0')     # 创建一个 Launch 文件内参数（arg）background_r
```

```
    )
    background_g_launch_arg = DeclareLaunchArgument(
        'background_g', default_value=TextSubstitution(text='120')    # 创
建一个 Launch 文件内参数（arg）background_g
    )
    background_b_launch_arg = DeclareLaunchArgument(
        'background_b', default_value=TextSubstitution(text='90')   # 创建
一个 Launch 文件内参数（arg）background_b
    )


    return LaunchDescription([                                      # 返回 la
unch 文件的描述信息
        background_r_launch_arg,                                    # 调用以
上创建的参数（arg）
        background_g_launch_arg,
        background_b_launch_arg,
        Node(                                                       # 配置一个
节点的启动
            package='turtlesim',
            executable='turtlesim_node',                            # 节点所
在的功能包
            name='sim',                                             # 对节点重
新命名
            parameters=[{                                           # ROS 参数
```

```
列表

            'background_r': LaunchConfiguration('background_r'),   # 创建
参数 background_r

            'background_g': LaunchConfiguration('background_g'),   # 创建
参数 background_g

            'background_b': LaunchConfiguration('background_b'),   # 创建
参数 background_b

        }]

    ),

  ])
```

3） 输入指令“**chmod +x param.launch.py**”，并按下回车，为保存的 param.launch.py 赋予可执行权限。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/launch_demo/launch$ chmod +x param.launch.py
ubuntu@raspberrypi:~/hiwonder_ws/src/launch_demo/launch$
```

5.2 编译运行

1） 赋予可执行权限后，输入“**cd ~/hiwonder_ws/**”指令，切换到工作空间的目录。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/hello_world_demo$ cd ~/hiwonder_ws/
ubuntu@raspberrypi:~/hiwonder_ws$
```

2） 输入“**colcon build**”指令，并按下回车，编译工作空间中的功能包。

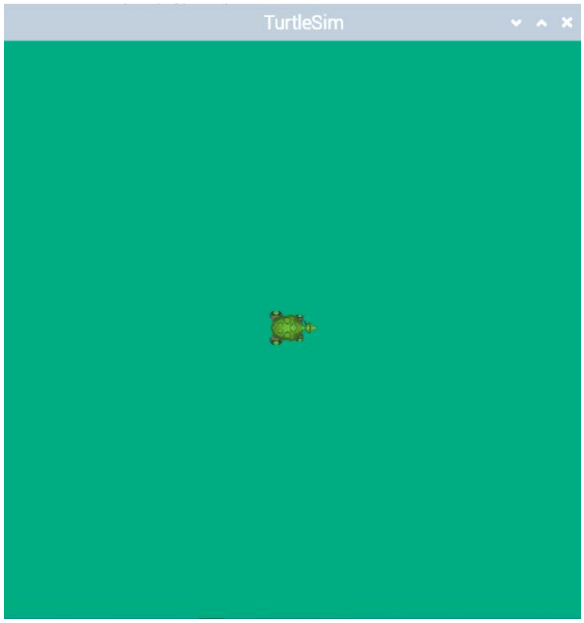
```
ubuntu@raspberrypi:~/hiwonder_ws$ colcon build
```

3） 再输入指令“**source ./install/setup.bash**”，并按下回车，让环境变量生效。

```
ubuntu@raspberrypi:~/hiwonder_ws$ source ./install/setup.bash
```

4） 输入指令“**ros2 launch launch_demo param.launch.py**”，并按下回车，启动 param.launch.py 文件。

```
ubuntu@raspberrypi:~$ ros2 launch launch_demo param.launch.py
[INFO] [launch]: All log files can be found below /home/ubuntu/.ros/log/2024-01-29-03-41-56-064345-raspberrypi-320
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [turtlesim_node-1]: process started with pid [321]
[turtlesim_node-1] QStandardPaths: XDG_RUNTIME_DIR not set, defaulting to '/tmp/runtime-ubuntu'
[turtlesim_node-1] [INFO] [1706499716.356949010] [sim]: Starting turtlesim with node name /sim
[turtlesim_node-1] [INFO] [1706499716.374705014] [sim]: Spawning turtle [turtle1] at x=[5.544445], y=[5.544445], theta=[0.000000]
[turtlesim_node-1] libGL error: glx: failed to create dri3 screen
[turtlesim_node-1] libGL error: failed to load driver: vc4
```



5.3 程序分析

通过 DeclareLaunchArgument 声明了 3 个参数 background_r、g、b,并在参数中设置默认值,然后通过 Node 描述 turtlesim 节点的启动任务,将这 3 个参数通过 LaunchConfiguration 注入到节点参数中,生成一个携带参数定义的 launch 文件,通过启动文件可以动态设置背景颜色参数以控制 turtle 模拟器的背景色。

```
from launch import LaunchDescription                    # launch文件的描述类
from launch.actions import DeclareLaunchArgument       # 声明launch文件内使用的Argument类
from launch.substitutions import LaunchConfiguration, TextSubstitution

from launch_ros.actions import Node                     # 节点启动的描述类


def generate_launch_description():                      # 自动生成launch文件的函数
    background_r_launch_arg = DeclareLaunchArgument(
        'background_r', default_value=TextSubstitution(text='0')     # 创建一个Launch文件内参数（arg）background_r
    )
    background_g_launch_arg = DeclareLaunchArgument(
        'background_g', default_value=TextSubstitution(text='120')    # 创建一个Launch文件内参数（arg）background_g
    )
    background_b_launch_arg = DeclareLaunchArgument(
        'background_b', default_value=TextSubstitution(text='90')   # 创建一个Launch文件内参数（arg）background_b
    )

    return LaunchDescription([                                      # 返回launch文件的描述信息
        background_r_launch_arg,                                    # 调用以上创建的参数（arg）
        background_g_launch_arg,
        background_b_launch_arg,
        Node(                                                       # 配置一个节点的启动
            package='turtlesim',
            executable='turtlesim_node',                            # 节点所在的功能包
            name='sim',                                             # 对节点重新命名
            parameters=[{                                           # ROS参数列表
                'background_r': LaunchConfiguration('background_r'),   # 创建参数background_r
                'background_g': LaunchConfiguration('background_g'),   # 创建参数background_g
                'background_b': LaunchConfiguration('background_b'),   # 创建参数background_b
            }]
        ),
    ])
```

6.YAML 参数 Launch 文件

6.1 param_yaml.launch.py 创建

1） 输入“**cd hiwonder_ws/src/launch_demo/launch**”指令，切换到 launch_demo 工作空间的 launch 文件夹中。

```
ubuntu@raspberrypi:~$ cd hiwonder_ws/src/launch_demo/launch
ubuntu@raspberrypi:~/hiwonder_ws/src/launch_demo/launch$
```

2） 输入指令“**vim param_yaml.launch.py**”编辑程序，复制下面程序。如需修改，再按下“**i**”即可修改。修改完成，按下“**Esc**”，输入“**：wq**”保存并退出。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/launch_demo/launch$ vim param_yaml.launch.p
y
```

```
import os



from ament_index_python.packages import get_package_share_directory  #
查询功能包路径的方法
```

```
from launch import LaunchDescription   # launch 文件的描述类
from launch_ros.actions import Node    # 节点启动的描述类



def generate_launch_description():     # 自动生成 launch 文件的函数
   config = os.path.join(              # 找到参数文件的完整路径
      get_package_share_directory('launch_demo'),
      'config',
      'turtlesim.yaml'
      )

   return LaunchDescription([          # 返回 launch 文件的描述信息
      Node(                            # 配置一个节点的启动
         package='turtlesim',          # 节点所在的功能包
         executable='turtlesim_node',  # 节点的可执行文件名
         namespace='turtlesim2',       # 节点所在的命名空间
         name='sim',                   # 对节点重新命名
         parameters=[config]           # 加载参数文件
      )
   ])
```

3） 输入指令“**chmod +x param_yaml.launch.py**”，并按下回车，为保存的 param_yaml.launch.py 赋予可执行权限。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/launch_demo/launch$ chmod +x param_yaml.lau
nch.py
ubuntu@raspberrypi:~/hiwonder_ws/src/launch_demo/launch$
```

6.2 setup.py 文件设置

setup.py 文件是定义一个 ROS2 包的远数据和构建配置，提供了包的元数据、依赖关系、构建配置和安装逻辑等信息，可以帮助开发者正确构建、安装和使用 ROS2 包。需要将 **param_yaml.launch.py** 的程序入口写到 setup.py 文件中。

1） 输入“**cd ..**”指令，切换到上一级目录。

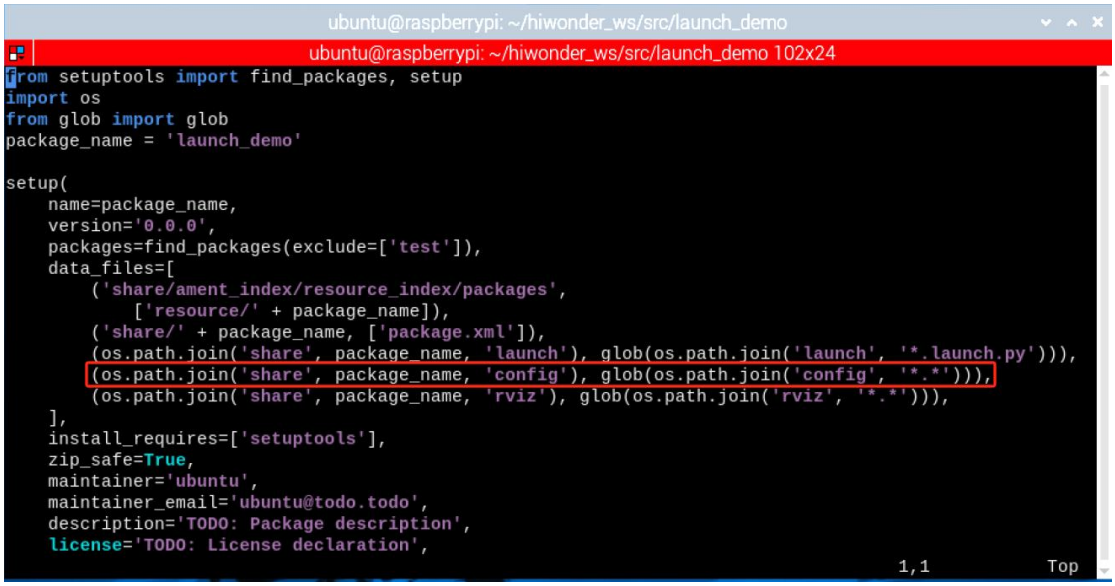
```
ubuntu@raspberrypi:~/hiwonder_ws/src/launch_demo/launch$ cd ..
ubuntu@raspberrypi:~/hiwonder_ws/src/launch_demo$
```

2） 输入“**vim setup.py**”指令，并按下回车，打开 setup.py 文件。

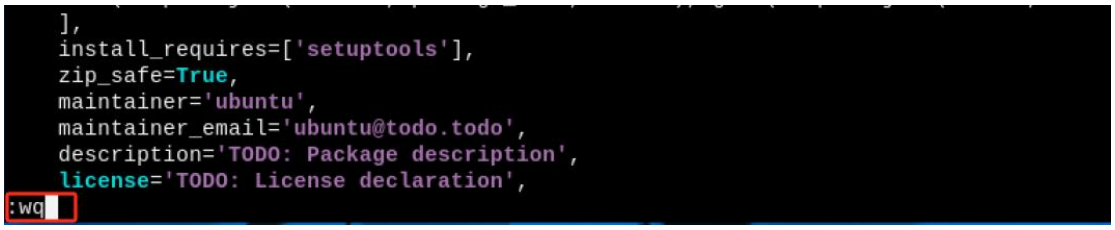
```
ubuntu@raspberrypi:~/hiwonder_ws/src/launch_demo$ vim setup.py
```

3） 按下“i”键进入可编辑模式，输入以下代码到相应的位置上。

```
(os.path.join('share', package_name, 'config'), glob(os.path.join('confi
g', '*.*'))),
```



4） 然后输入“:wq”，保存并退出文件。



6.3 编译运行

1） 赋予可执行权限后，输入“**cd ~/hiwonder_ws/**”指令，切换到工作空间的目录。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/hello_world_demo$ cd ~/hiwonder_ws/
ubuntu@raspberrypi:~/hiwonder_ws$
```

2） 输入“**colcon build**”指令，并按下回车，编译工作空间中的功能包。

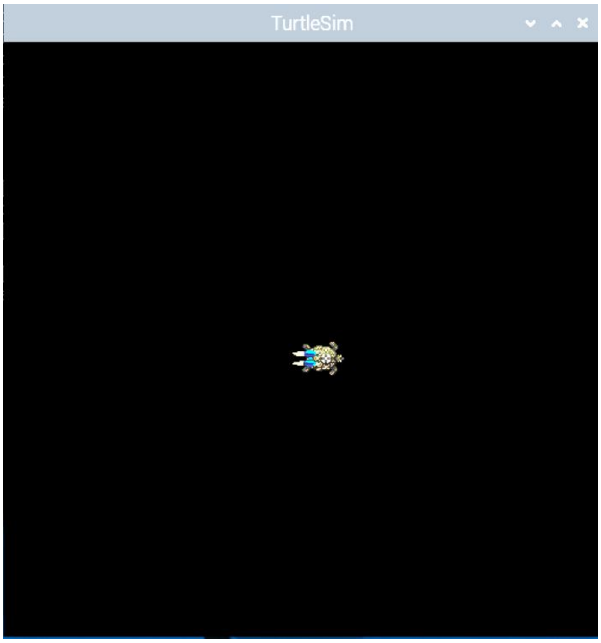
```
ubuntu@raspberrypi:~/hiwonder_ws$ colcon build
```

3） 再输入指令“**source ./install/setup.bash**”，并按下回车，让环境变量生效。

```
ubuntu@raspberrypi:~/hiwonder_ws$ source ./install/setup.bash
```

4） 输入指令“**ros2 launch launch_demo param_yaml.launch.py**”，并按下回车，启动 param_yaml.launch.py 文件。

```
ubuntu@raspberrypi:~$ ros2 launch launch_demo param_yaml.launch.py
[INFO] [launch]: All log files can be found below /home/ubuntu/.ros/log/2024-01-29-03-57-37-719249-ras
pberrypi-419
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [turtlesim_node-1]: process started with pid [420]
[turtlesim_node-1] QStandardPaths: XDG_RUNTIME_DIR not set, defaulting to '/tmp/runtime-ubuntu'
[turtlesim_node-1] [INFO] [1706500657.905322051] [turtlesim2.sim]: Starting turtlesim with node name /
turtlesim2/sim
[turtlesim_node-1] [INFO] [1706500657.911099878] [turtlesim2.sim]: Spawning turtle [turtle1] at x=[5.5
44445], y=[5.544445], theta=[0.000000]
```



6.4 程序分析

通过 Node 描述一个 turtlesim 节点的启动任务,设置了包名、执行文件名、命名空间和名称标签,并通过 parameters 从指定路径加载配置文件 turtlesim.yaml,最后将此任务添加到 LaunchDescription 中返回,生成一个用来描述和启动带参数的 turtlesim 节点的 launch 文件。

```
import os

from ament_index_python.packages import get_package_share_directory  # 查询功能包路径的方法

from launch import LaunchDescription    # launch文件的描述类
from launch_ros.actions import Node     # 节点启动的描述类


def generate_launch_description():      # 自动生成launch文件的函数
    config = os.path.join(              # 找到参数文件的完整路径
        get_package_share_directory('launch_demo'),
        'config',
        'turtlesim.yaml'
        )

    return LaunchDescription([          # 返回launch文件的描述信息
        Node(                           # 配置一个节点的启动
            package='turtlesim',        # 节点所在的功能包
            executable='turtlesim_node',# 节点的可执行文件名
            namespace='turtlesim2',     # 节点所在的命名空间
            name='sim',                 # 对节点重新命名
            parameters=[config]         # 加载参数文件
        )
    ])
```

7.Launch 文件启动 Launch

7.1 launch_include.launch.py 创建

1） 输入“**cd hiwonder_ws/src/launch_demo/launch**”指令，切换到 launch_demo 工作空间的 launch 文件夹中。

```
ubuntu@raspberrypi:~$ cd hiwonder_ws/src/launch_demo/launch
ubuntu@raspberrypi:~/hiwonder_ws/src/launch_demo/launch$
```

2） 输入指令“**vim launch_include.launch.py**”编辑程序，复制下面程序。如需修改，再按下“**i**”即可修改。修改完成，按下“**Esc**”，输入“**: wq**”保存并退出。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/launch_demo/launch$ vim launch_include.launch.py
```

```
import os



from ament_index_python.packages import get_package_share_directory  #
查询功能包路径的方法



from launch import LaunchDescription                    # launch 文件的描述类
```

```
from launch.actions import IncludeLaunchDescription  # 节点启动的描述类
from launch.launch_description_sources import PythonLaunchDescriptionSou
rce
from launch.actions import GroupAction               # launch 文件中的执行
动作
from launch_ros.actions import PushRosNamespace      # ROS 命名空间配置


def generate_launch_description():                   # 自动生成 launch 文件
的函数
   hello_world = IncludeLaunchDescription(        # 包含指定路径下的另外一
个 launch 文件
      PythonLaunchDescriptionSource([os.path.join(
         get_package_share_directory('launch_demo'), 'launch'),
         '/single_node.launch.py'])
      )


   return LaunchDescription([                        # 返回 launch 文件的描述
信息
      hello_world
   ])
```

3） 输入指令“**chmod +x launch_include.launch.py**”，并按下回车，为保存的 launch_include.launch.py 赋予可执行权限。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/launch_demo/launch$ chmod +x launch_include.launch.py
ubuntu@raspberrypi:~/hiwonder_ws/src/launch_demo/launch$
```

7.2 编译运行

1） 赋予可执行权限后，输入“**cd ~/hiwonder_ws/**”指令，切换到工作空间的目录。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/hello_world_demo$ cd ~/hiwonder_ws/
ubuntu@raspberrypi:~/hiwonder_ws$
```

2） 输入“**colcon build**”指令，并按下回车，编译工作空间中的功能包。

```
ubuntu@raspberrypi:~/hiwonder_ws$ colcon build
```

3） 再输入指令“**source ./install/setup.bash**”，并按下回车，让环境变量生效。

```
ubuntu@raspberrypi:~/hiwonder_ws$ source ./install/setup.bash
```

4） 输入指令“**ros2 launch launch_demo launch_include.launch.py**”，并按下回车，启动 **launch_include.launch.py** 文件。

```
ubuntu@raspberrypi:~$ ros2 launch launch_demo launch_include.launch.py
[INFO] [launch]: All log files can be found below /home/ubuntu/.ros/log/2024-01-29-04-08-56-256102-ras
pberrypi-437
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [hello_world-1]: process started with pid [438]
[hello_world-1] [INFO] [1706501336.651495522] [hello_world_demo]: Hello World
[hello_world-1] [INFO] [1706501337.152876092] [hello_world_demo]: Hello World
[hello_world-1] [INFO] [1706501337.654117719] [hello_world_demo]: Hello World
[hello_world-1] [INFO] [1706501338.155382328] [hello_world_demo]: Hello World
[hello_world-1] [INFO] [1706501338.656693476] [hello_world_demo]: Hello World
```

7.3 程序分析

通过 IncludeLaunchDescription 动作从指定路径引入了另一个 launch 文件 single_node.launch.py 描述的任务,并将这个任务加入到 LaunchDescription 对象返回,生成一个主 launch 文件以包含和运行在子 launch 文件中定义的节点任务。

```python
import os

from ament_index_python.packages import get_package_share_directory  # 查询功能包路径的方法

from launch import LaunchDescription                # launch文件的描述类
from launch.actions import IncludeLaunchDescription  # 节点启动的描述类
from launch.launch_description_sources import PythonLaunchDescriptionSource
from launch.actions import GroupAction              # launch文件中的执行动作
from launch_ros.actions import PushRosNamespace     # ROS命名空间配置

def generate_launch_description():                  # 自动生成launch文件的函数
    hello_world = IncludeLaunchDescription(        # 包含指定路径下的另外一个launch文件
        PythonLaunchDescriptionSource([os.path.join(
            get_package_share_directory('launch_demo'), 'launch'),
            '/single_node.launch.py'])
        )

    return LaunchDescription([                      # 返回launch文件的描述信息
        hello_world
    ])
```

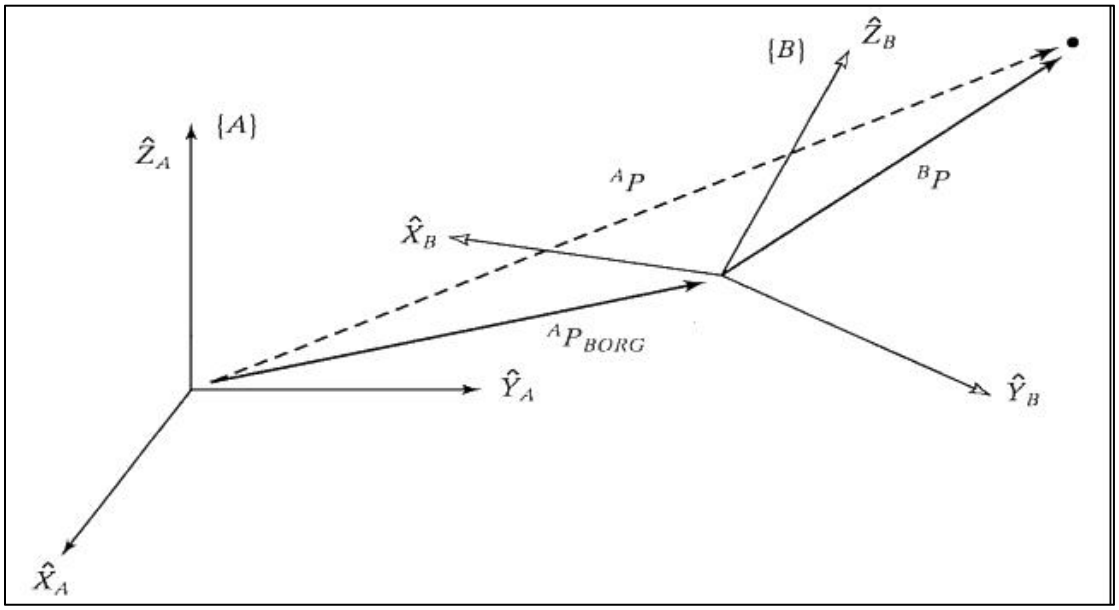
第 16 课 ROS2 TF2 机器人坐标系管理器

1.TF2 简介

坐标系是我们非常熟悉的一个概念，也是机器人学中的重要基础，在一个完整的机器人系统中，会存在很多坐标系，这些坐标系之间的位置关系该如何管理？ROS 给我们提供了一个坐标系的管理神奇：TF2。

在移动机器人系统中，坐标系一样至关重要，比如一个移动机器人的中心点是基坐标系 Base Link，雷达所在的位置叫做雷达坐标系 laser link，机器人要移动，里程计会累计位置，这个位置的参考系叫做里程计坐标系 odom，里程计又会有累计误差和飘逸，绝对位置的参考系叫做地图坐标系 map。

一层一层坐标系之间关系复杂，有一些是相对固定的，也有一些是不断变化的，看似简单的坐标系也在空间范围内变得复杂，良好的坐标系管理系统就显得格外重要。

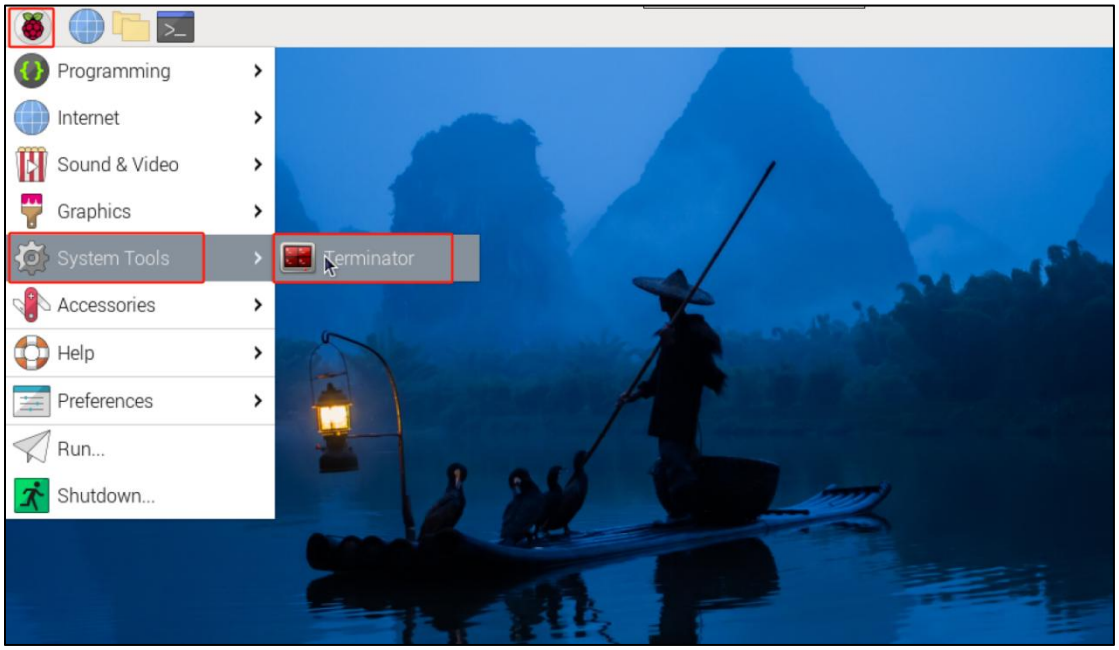


关于坐标系变换关系的基本理论，在每一个机器人学的教材中都会讲解，可以分解为平移和旋转两个部分，通过一个 4*4 的矩阵进行描述，在空间中画出坐标系，那两者之间的变换关系，其实就是向量的教学描述。

ROS 中 TF 功能的底层原理，就是对这些数学变换进行了封装，详细的理论知识大家可以参考机器人学的教材，我们主要讲解 TF 坐标管理系统的使用方法。

2.TF 命令行操作

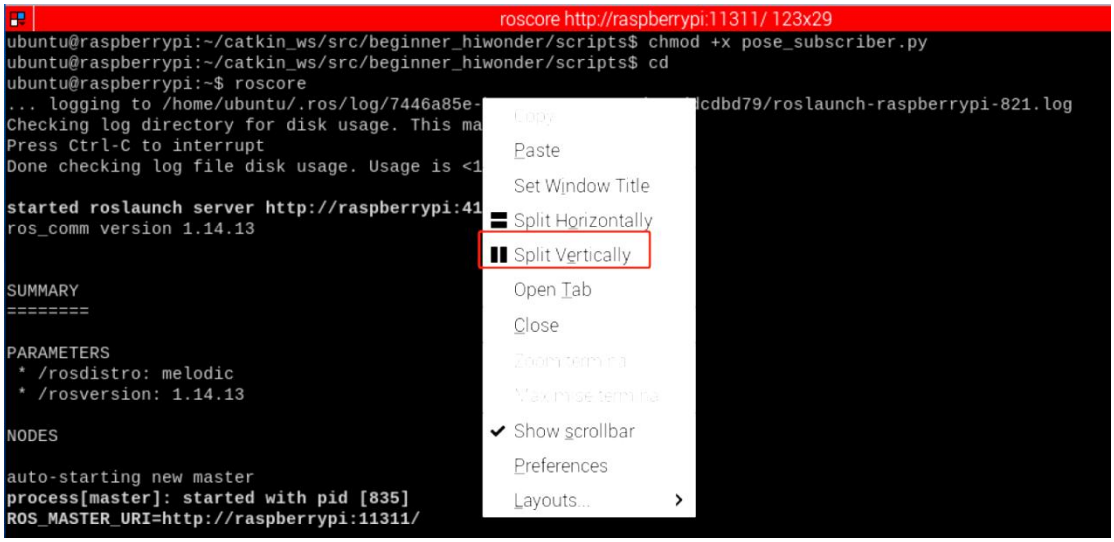
1） 点击左上角的 logo，选择 System Tools →Terminator。



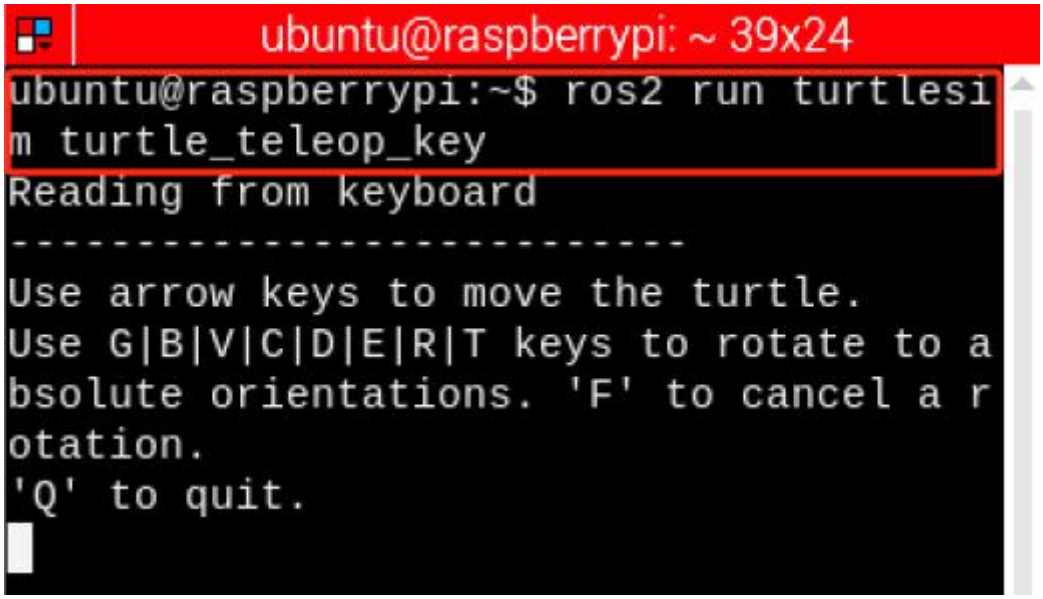
2） 输入“**ros2 launch turtle_tf2_py turtle_tf2_demo.launch.py**”指令，运行 turtle_tf2_demo.launch.py 文件。

```
ubuntu@raspberrypi:~$ ros2 launch turtle_tf2_py turtle_tf2_demo.launch.py
```

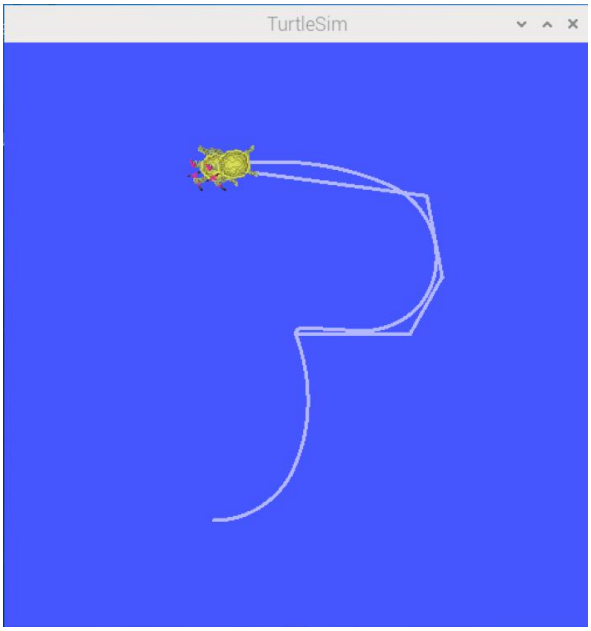
3） 鼠标右键，选择“**Split Vertically**”点击，新建一个新的终端窗口。



4） 输入“**ros2 run turtlesim turtle_teleop_key**”指令，运行小海龟键盘控制节点。



5） 按“↑↓←→”键控制小海龟移动，另外一只小海龟也会跟着运动。



6） 参考 3）步骤，新建一个终端。输入“**ros2 run tf2_ros tf2_echo turtle2 turtle1**”指令，查看 turtle1 和 turtle2 两个坐标系之间的具体关系。

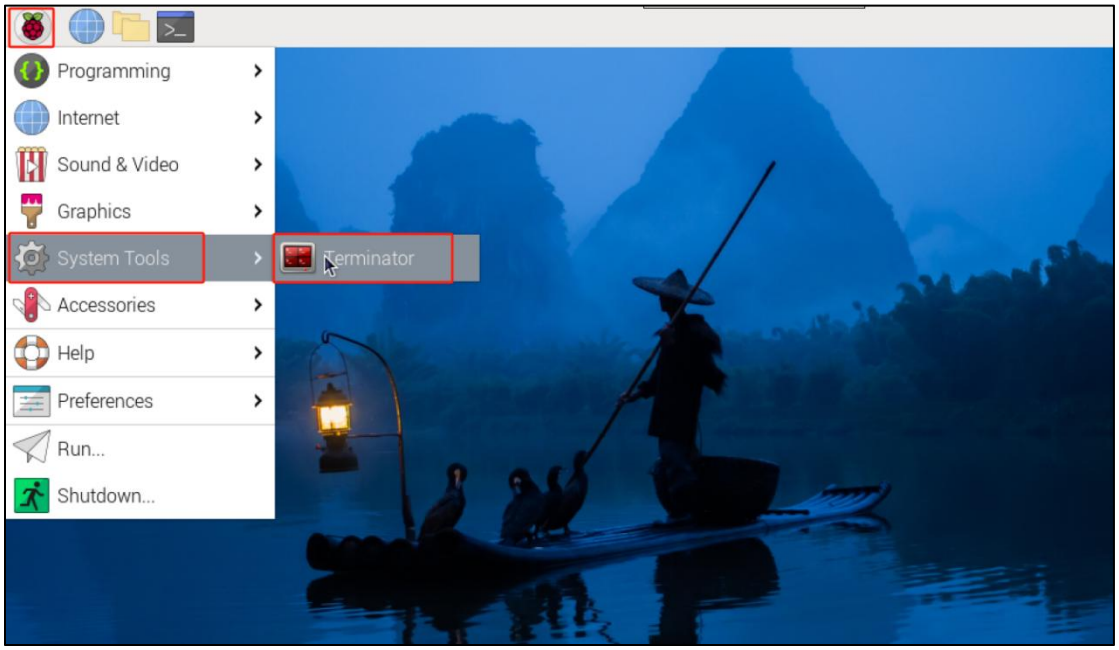
```
ubuntu@raspberrypi:~$ ros2 run tf2_ros tf2_echo turtle2 turtle1
[INFO] [1706512580.259346595] [tf2_echo]: Waiting for transform turtle2 ->  turt
le1: Invalid frame ID "turtle2" passed to canTransform argument target_frame - f
rame does not exist
At time 1706512582.236660386
- Translation: [0.000, 0.000, 0.000]
- Rotation: in Quaternion [0.000, 0.000, 0.017, 1.000]
- Rotation: in RPY (radian) [0.000, -0.000, 0.035]
- Rotation: in RPY (degree) [0.000, -0.000, 1.994]
- Matrix:
  0.999 -0.035  0.000  0.000
  0.035  0.999  0.000  0.000
  0.000  0.000  1.000  0.000
  0.000  0.000  0.000  1.000
At time 1706512583.228831090
- Translation: [0.000, 0.000, 0.000]
- Rotation: in Quaternion [0.000, 0.000, 0.017, 1.000]
- Rotation: in RPY (radian) [0.000, -0.000, 0.035]
- Rotation: in RPY (degree) [0.000, -0.000, 1.994]
```

终端中循环打印坐标系的变换数值，由平移和旋转两个部分组成，还有旋转矩阵。

3.静态 TF 广播

3.1 创建 static_tf_broadcaster.py

1） 点击左上角的 logo，选择 System Tools →Terminator。



2） 输入“**cd hiwonder_ws/src/**”指令，切换到 hiwonder_ws 工作空间的 src 文件夹中。

```
ubuntu@raspberrypi:~$ cd hiwonder_ws/src/
ubuntu@raspberrypi:~/hiwonder_ws/src$
```

3） 输入“**ros2 pkg create tf_demo --build-type ament_python --dependencies rclpy**”并按下回车，创建一个名为“**tf_demo**”的功能包，添加依赖关系 rclpy。

```
ubuntu@raspberrypi:~/hiwonder_ws/src$ ros2 pkg create tf_demo --build-type ament_python --dependencies rclpy
```

4） 输入“**cd tf_demo/tf_demo/**”指令，切换到 tf_demo 功能包下。

```
ubuntu@raspberrypi:~/hiwonder_ws/src$ cd tf_demo/tf_demo/
ubuntu@raspberrypi:~/hiwonder_ws/src/tf_demo/tf_demo$
```

5） 输入指令“**vim static_tf_broadcaster.py**”编辑程序，复制下面程序。如需修改，再按下“**i**”即可修改。修改完成，按下“**Esc**”，输入“**：wq**”保存并退出。

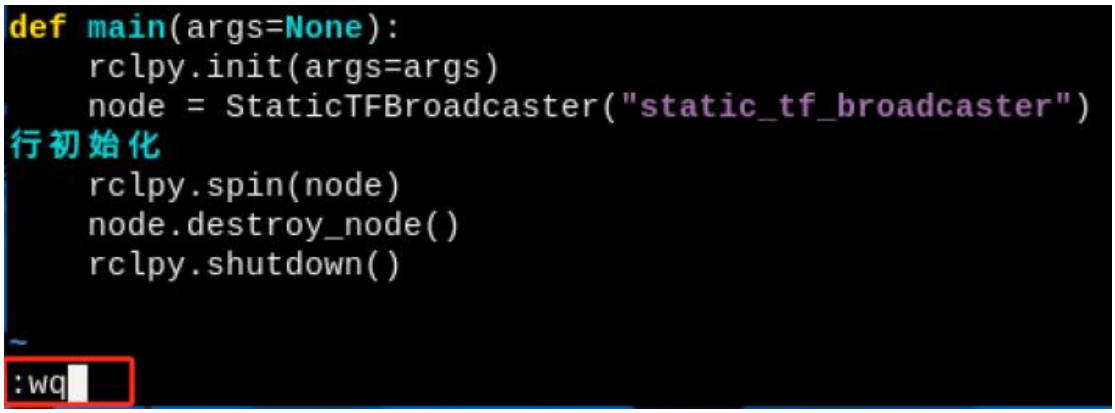
```
ubuntu@raspberrypi:~/hiwonder_ws/src/tf_demo/tf_demo$ vim static_tf_broadcaster.py
```

```
#!/usr/bin/env python3

# -*- coding: utf-8 -*-
```

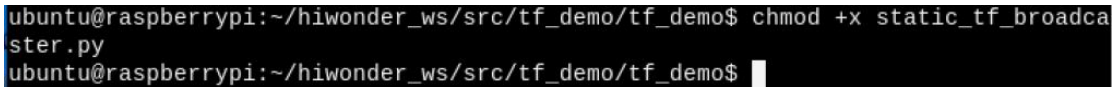
```
import rclpy
# ROS2 Python 接口库
from rclpy.node import Node
 # ROS2 节点类
from geometry_msgs.msg import TransformStamped
   # 坐标变换消息
import tf_transformations
 # TF 坐标变换库
from tf2_ros.static_transform_broadcaster import StaticTransformBroadcas
ter  # TF 静态坐标系广播器类


class StaticTFBroadcaster(Node):
    def __init__(self, name):
        super().__init__(name)
    # ROS2 节点父类初始化
        self.tf_broadcaster = StaticTransformBroadcaster(self)
       # 创建一个 TF 广播器对象


        static_transformStamped = TransformStamped()
      # 创建一个坐标变换的消息对象
        static_transformStamped.header.stamp = self.get_clock().now().to
```

```
_msg()  # 设置坐标变换消息的时间戳

        static_transformStamped.header.frame_id = 'world'

      # 设置一个坐标变换的源坐标系

        static_transformStamped.child_frame_id  = 'house'

      # 设置一个坐标变换的目标坐标系

        static_transformStamped.transform.translation.x = 10.0

       # 设置坐标变换中的 X、Y、Z 向的平移

        static_transformStamped.transform.translation.y = 5.0


        static_transformStamped.transform.translation.z = 0.0

        quat = tf_transformations.quaternion_from_euler(0.0, 0.0, 0.0)

       # 将欧拉角转换为四元数（roll, pitch, yaw）

        static_transformStamped.transform.rotation.x = quat[0]

       # 设置坐标变换中的 X、Y、Z 向的旋转（四元数）

        static_transformStamped.transform.rotation.y = quat[1]

        static_transformStamped.transform.rotation.z = quat[2]

        static_transformStamped.transform.rotation.w = quat[3]


        self.tf_broadcaster.sendTransform(static_transformStamped)

       # 广播静态坐标变换，广播后两个坐标系的位置关系保持不变


def main(args=None):

    rclpy.init(args=args)                                   # ROS2 Python 接口
初始化
```

```
    node = StaticTFBroadcaster("static_tf_broadcaster")  # 创建 ROS2 节点对象并进行初始化
    rclpy.spin(node)                                     # 循环等待 ROS2 退出
    node.destroy_node()                                  # 销毁节点对象
    rclpy.shutdown()
```



6） 输入指令“**chmod +x static_tf_broadcaster.py**”，并按下回车，为保存的 static_tf_broadcaster.py 赋予可执行权限。



3.2 创建 tf_listener.py

1） 输入指令“**vim tf_listener.py**”编辑程序，复制下面程序。如需修改，再按下“**i**”即可修改。修改完成，按下“**Esc**”，输入“**: wq**”保存并退出。


```
#!/usr/bin/env python3

# -*- coding: utf-8 -*-
```

```
import rclpy                                                    # ROS2 Python 接口
库
from rclpy.node import Node                                     # ROS2 节点类
import tf_transformations                                       # TF 坐标变换库
from tf2_ros import TransformException                          # TF 左边变换的异
常类
from tf2_ros.buffer import Buffer                               # 存储坐标变换信息
的缓冲类
from tf2_ros.transform_listener import TransformListener  # 监听坐标变换
的监听器类


class TFListener(Node):


    def __init__(self, name):
        super().__init__(name)                                  # ROS2 节
点父类初始化


        self.declare_parameter('source_frame', 'world')             # 创建
一个源坐标系名的参数
        self.source_frame = self.get_parameter(                     # 优先
使用外部设置的参数值，否则用默认值
            'source_frame').get_parameter_value().string_value


        self.declare_parameter('target_frame', 'home')              # 创建
```

```
一个目标坐标系名的参数
        self.target_frame = self.get_parameter(                    # 优先
使用外部设置的参数值，否则用默认值
            'target_frame').get_parameter_value().string_value


        self.tf_buffer = Buffer()                                  # 创建保
存坐标变换信息的缓冲区
        self.tf_listener = TransformListener(self.tf_buffer, self)  # 创
建坐标变换的监听器


        self.timer = self.create_timer(1.0, self.on_timer)         # 创建
一个固定周期的定时器，处理坐标信息


    def on_timer(self):
        try:
            now = rclpy.time.Time()                                # 获取 RO
S 系统的当前时间
            trans = self.tf_buffer.lookup_transform(               # 监听
当前时刻源坐标系到目标坐标系的坐标变换
                self.target_frame,
                self.source_frame,
                now)
        except TransformException as ex:                           # 如果坐
标变换获取失败，进入异常报告
```

```
            self.get_logger().info(
                f'Could not transform {self.target_frame} to {self.source
_frame}: {ex}')
            return


        pos  = trans.transform.translation                          # 获取
位置信息
        quat = trans.transform.rotation                             # 获取姿
态信息（四元数）
        euler = tf_transformations.euler_from_quaternion([quat.x, quat.y,
 quat.z, quat.w])
        self.get_logger().info('Get %s --> %s transform: [%f, %f, %f] [%f,
 %f, %f]'
          % (self.source_frame, self.target_frame, pos.x, pos.y, pos.z, e
uler[0], euler[1], euler[2]))


def main(args=None):
    rclpy.init(args=args)                           # ROS2 Python 接口初始化
    node = TFListener("tf_listener")                # 创建 ROS2 节点对象并进行初
始化
    rclpy.spin(node)                                # 循环等待 ROS2 退出
    node.destroy_node()                             # 销毁节点对象
    rclpy.shutdown()                                # 关闭 ROS2 Python 接口
```

```
def main(args=None):
    rclpy.init(args=args)
    node = StaticTFBroadcaster("static_tf_broadcaster")
行初始化
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

~
:wq
```

2） 输入指令“**chmod +x tf_listener.py**”，并按下回车，为保存的 tf_listener.py 赋予可执行权限。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/tf_demo/tf_demo$ chmod +x tf_listener.py
ubuntu@raspberrypi:~/hiwonder_ws/src/tf_demo/tf_demo$
```

3.3 setup.py 文件设置

setup.py 文件是定义一个 ROS2 包的远数据和构建配置，提供了包的元数据、依赖关系、构建配置和安装逻辑等信息，可以帮助开发者正确构建、安装和使用 ROS2 包。需要将 **static_tf_broadcaster.py** 和 **tf_listener.py** 的程序入口写到 setup.py 文件中。

1） 输入“**cd ..**”指令，切换到上一级目录。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/topic_demo/topic_demo$ cd ..
ubuntu@raspberrypi:~/hiwonder_ws/src/topic_demo$
```

2） 输入“**vim setup.py**”指令，并按下回车，打开 setup.py 文件。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/topic_demo$ vim setup.py
```

3） 按下“i”键进入可编辑模式，输入以下代码到相应的位置上。

```
'static_tf_broadcaster = tf_demo.static_tf_broadcaster:main',

'tf_listener = tf_demo.tf_listener:main',
```

```
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='ubuntu',
    maintainer_email='ubuntu@todo.todo',
    description='TODO: Package description',
    license='TODO: License declaration',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'static_tf_broadcaster = tf_demo.static_tf_broadcaster:main',
            'tf_listener = tf_demo.tf_listener:main',
            'turtle_tf_broadcaster = tf_demo.turtle_tf_broadcaster:main',
            'turtle_following = tf_demo.turtle_following:main',

        ],
    },
)
-- INSERT --                                          34,1          Bot
```

4） 然后输入“:wq”，保存并退出文件。

```
        'console_scripts': [
            'static_tf_broadcaster = tf_demo.static_tf_broadcaster:main',
            'tf_listener = tf_demo.tf_listener:main',
            'turtle_tf_broadcaster = tf_demo.turtle_tf_broadcaster:main',
            'turtle_following = tf_demo.turtle_following:main',

        ],
    },
)
:wq
```

3.4 编译运行

1） 赋予可执行权限后，输入“**cd ~/hiwonder_ws/**”指令，切换到工作空间的目录。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/hello_world_demo$ cd ~/hiwonder_ws/
ubuntu@raspberrypi:~/hiwonder_ws$
```

2） 输入“**colcon build**”指令，并按下回车，编译工作空间中的功能包。

```
ubuntu@raspberrypi:~/hiwonder_ws$ colcon build
```

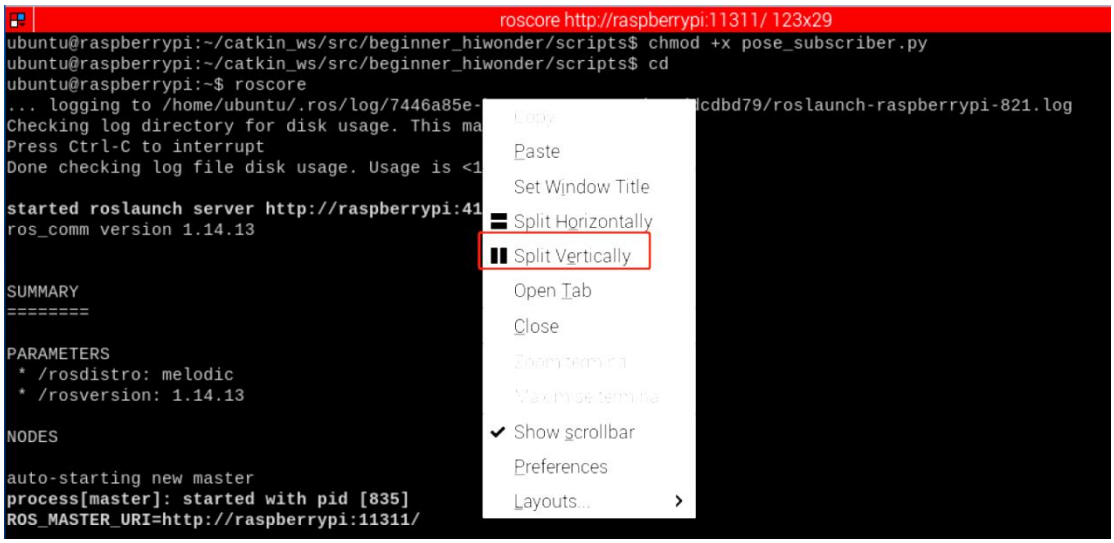
3） 再输入指令“**source ./install/setup.bash**”，并按下回车，让环境变量生效。

```
ubuntu@raspberrypi:~/hiwonder_ws$ source ./install/setup.bash
```

4） 输入指令“**ros2 run tf_demo static_tf_broadcaster**”，并按下回车，启动 static_tf_broadcaster 节点。

```
ubuntu@raspberrypi:~$ ros2 run tf_demo static_tf_broadcaster
```

5） 鼠标右键，选择“**Split Vertically**”点击，新建一个新的终端窗口。



6） 输入指令“**ros2 run tf_demo tf_listener**”，并按下回车，启动 tf_listener 监听。

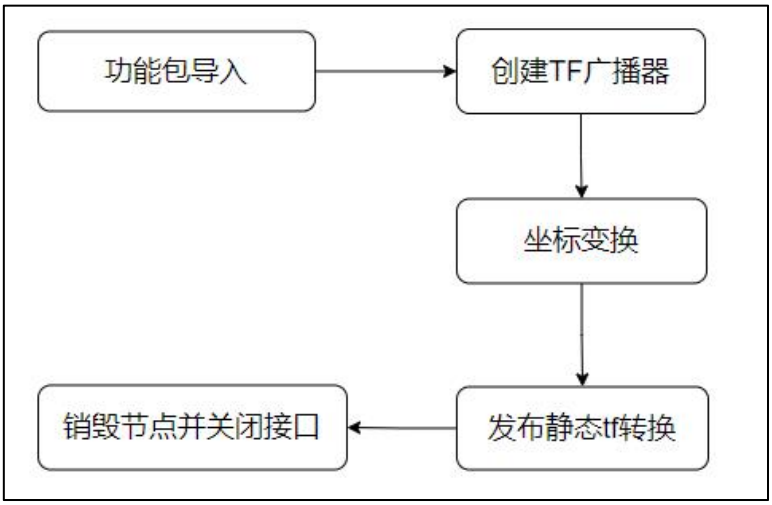
```
ubuntu@raspberrypi:~$ ros2 run tf_demo tf_listener
[INFO] [1706519151.659170311] [tf_listener]: Get world --> home transform: [-6.000000, -11.000000, 0.000000] [0.000000, -0.000000, 0.000000]
[INFO] [1706519152.635825534] [tf_listener]: Get world --> home transform: [-6.000000, -11.000000, 0.000000] [0.000000, -0.000000, 0.000000]
[INFO] [1706519153.637413658] [tf_listener]: Get world --> home transform: [-6.000000, -11.000000, 0.000000] [0.000000, -0.000000, 0.000000]
```

可以看到当前系统中存在两个坐标系，一个是 world，一个是 house，两者之间的相对位置不会发生改变，通过一个静态的 TF 对象进行维护。

3.5 程序分析

3.5.1 static_tf_broadcaster.py

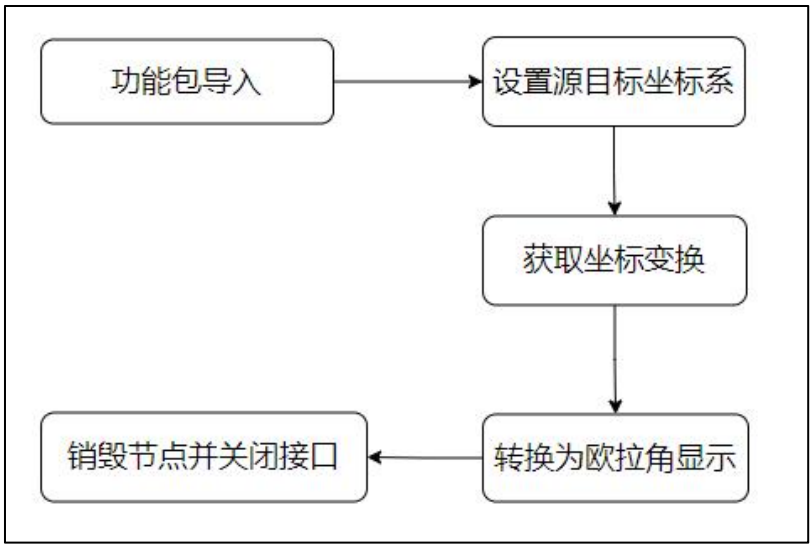
根据实现的效果，梳理程序的过程逻辑，如下图所示：



创建了 StaticTFBroadcaster 节点类，初始化节点对象时创建了 TF 广播器，然后生成包含源坐标系世界坐标和目标 home 坐标以及转换信息如平移旋转等的坐标转换消息，利用 tf_transformations 将欧拉角转换成四元数格式，在节点运行期间通过广播器对象持续广播这个静态坐标转换信息，目的是为其他节点或工具提供从世界坐标到 home 坐标的坐标变换支持。

3.5.2 tf_listener.py

根据实现的效果，梳理程序的过程逻辑，如下图所示：

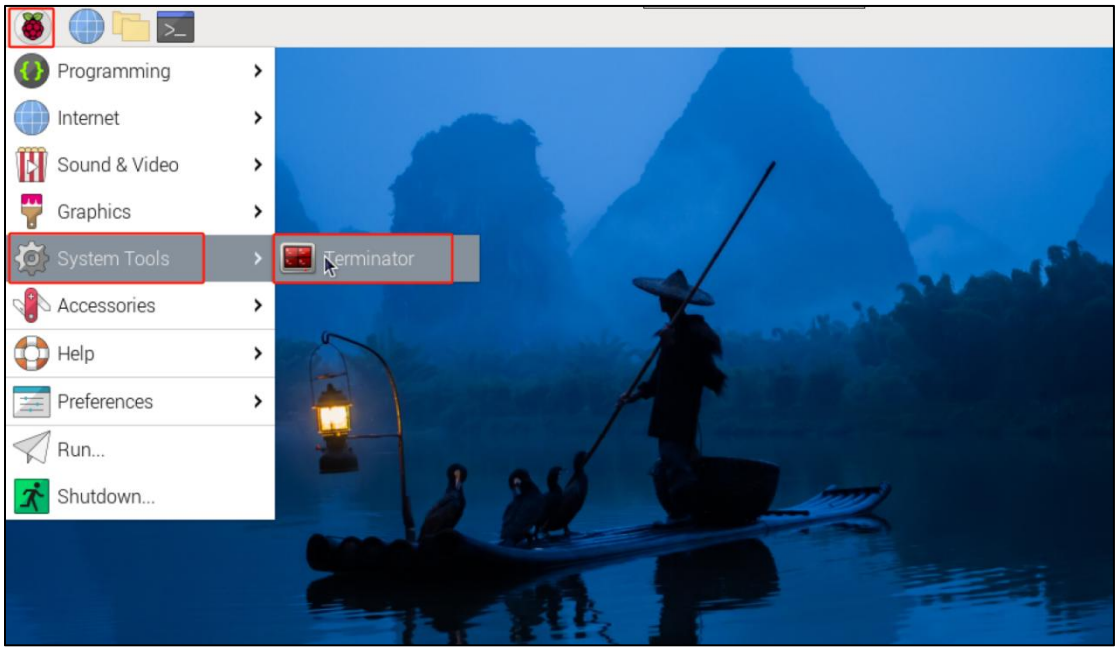


创建了 TFListener 节点类，初始化节点对象时创建了 TF 缓冲区和监听器，设置了动态获取源目标坐标系的参数，通过定时器接口定期从监听器查询指定坐标系间的坐标转换关系，获取其位置姿态信息打印到日志，同时处理了坐标转换失败的异常情况，其作用是实时监测指定坐标系间的动态转换关系，并为其他节点或用户提供实时的坐标转换支持。

4.TF 监听

4.1 创建 turtle_tf_broadcaster.py

1） 点击左上角的 logo，选择 System Tools →Terminator。



2） 输入“**cd hiwonder_ws/src/tf_demo/tf_demo/**”指令，切换到 tf_demo 的文件夹中。

3） 输入指令“**vim turtle_tf_broadcaster.py**”编辑程序，复制下面程序。如需修改，再按下“**i**”即可修改。修改完成，按下“**Esc**”，输入“**：wq**”保存并退出。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/tf_demo/tf_demo$ vim turtle_tf_broadcaster.
py
```

```
#!/usr/bin/env python3

# -*- coding: utf-8 -*-
```

```
import rclpy                                        # ROS2 Python 接口库
from rclpy.node import Node                         # ROS2 节点类
from geometry_msgs.msg import TransformStamped      # 坐标变换消息
import tf_transformations                           # TF 坐标变换库
from tf2_ros import TransformBroadcaster            # TF 坐标变换广播器
from turtlesim.msg import Pose                      # turtlesim 小海龟位置消息

class TurtleTFBroadcaster(Node):

    def __init__(self, name):
        super().__init__(name)                                  # ROS2 节点父类初始化

        self.declare_parameter('turtlename', 'turtle')          # 创建一个海龟名称的参数
        self.turtlename = self.get_parameter(                   # 优先使用外部设置的参数值，否则用默认值
            'turtlename').get_parameter_value().string_value

        self.tf_broadcaster = TransformBroadcaster(self)        # 创建一个 T
```

```
F 坐标变换的广播对象并初始化



        self.subscription = self.create_subscription(         # 创建一个订
阅者，订阅海龟的位置消息
            Pose,
            f'/{self.turtlename}/pose',                       # 使用参数中获
取到的海龟名称
            self.turtle_pose_callback, 1)



    def turtle_pose_callback(self, msg):                              # 创
建一个处理海龟位置消息的回调函数，将位置消息转变成坐标变换
        transform = TransformStamped()                               # 创建
一个坐标变换的消息对象



        transform.header.stamp = self.get_clock().now().to_msg()      #
设置坐标变换消息的时间戳
        transform.header.frame_id = 'world'                           # 设
置一个坐标变换的源坐标系
        transform.child_frame_id = self.turtlename                    # 设
置一个坐标变换的目标坐标系
        transform.transform.translation.x = msg.x                     # 设
置坐标变换中的 X、Y、Z 向的平移
        transform.transform.translation.y = msg.y
        transform.transform.translation.z = 0.0
```

```
        q = tf_transformations.quaternion_from_euler(0, 0, msg.theta) #
将欧拉角转换为四元数（roll, pitch, yaw）

        transform.transform.rotation.x = q[0]                          # 设
置坐标变换中的 X、Y、Z 向的旋转（四元数）

        transform.transform.rotation.y = q[1]

        transform.transform.rotation.z = q[2]

        transform.transform.rotation.w = q[3]



        # Send the transformation

        self.tf_broadcaster.sendTransform(transform)     # 广播坐标变换，海
龟位置变化后，将及时更新坐标变换信息



def main(args=None):

    rclpy.init(args=args)                                  # ROS2 Python 接口
初始化

    node = TurtleTFBroadcaster("turtle_tf_broadcaster")  # 创建 ROS2 节点对
象并进行初始化

    rclpy.spin(node)                                      # 循环等待 ROS2 退出

    node.destroy_node()                                    # 销毁节点对象

    rclpy.shutdown()                                      # 关闭 ROS2 Python
接口
```

```
def main(args=None):
    rclpy.init(args=args)
    node = StaticTFBroadcaster("static_tf_broadcaster")
行初始化
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

~
:wq
```

4） 输入指令“**chmod +x turtle_tf_broadcaster.py**”，并按下回车，为保存的 turtle_tf_broadcaster.py 赋予可执行权限。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/tf_demo/tf_demo$ chmod +x turtle_tf_broadca
ster.py
ubuntu@raspberrypi:~/hiwonder_ws/src/tf_demo/tf_demo$
```

4.2 创建 turtle_following.py

3） 输入指令“**vim turtle_following.py**”编辑程序，复制下面程序。如需修改，再按下“**i**”即可修改。修改完成，按下“**Esc**”，输入“**：wq**”保存并退出。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/tf_demo/tf_demo$ vim turtle_following.py
```

```
#!/usr/bin/env python3

# -*- coding: utf-8 -*-









import math

import rclpy                                                    # ROS2 Python 接口
库

from rclpy.node import Node                                     # ROS2 节点类
```

```
import tf_transformations                                    # TF 坐标变换库
from tf2_ros import TransformException                       # TF 左边变换的异常类
from tf2_ros.buffer import Buffer                            # 存储坐标变换信息的缓冲类
from tf2_ros.transform_listener import TransformListener     # 监听坐标变换的监听器类
from geometry_msgs.msg import Twist                          # ROS2 速度控制消息
from turtlesim.srv import Spawn                              # 海龟生成的服务接口
class TurtleFollowing(Node):


    def __init__(self, name):
        super().__init__(name)                               # ROS2 节点父类初始化


        self.declare_parameter('source_frame', 'turtle1')    # 创建一个源坐标系名的参数
        self.source_frame = self.get_parameter(              # 优先使用外部设置的参数值，否则用默认值
            'source_frame').get_parameter_value().string_value


        self.tf_buffer = Buffer()                            # 创建保
```

```
存坐标变换信息的缓冲区
        self.tf_listener = TransformListener(self.tf_buffer, self)  # 创
建坐标变换的监听器


        self.spawner = self.create_client(Spawn, 'spawn')          # 创建
一个请求产生海龟的客户端
        self.turtle_spawning_service_ready = False                 # 是否
已经请求海龟生成服务的标志位
        self.turtle_spawned = False                                # 海龟是
否产生成功的标志位


        self.publisher = self.create_publisher(Twist, 'turtle2/cmd_vel',
1) # 创建跟随运动海龟的速度话题


        self.timer = self.create_timer(1.0, self.on_timer)         # 创建
一个固定周期的定时器，控制跟随海龟的运动


    def on_timer(self):
        from_frame_rel = self.source_frame                         # 源坐标
系
        to_frame_rel   = 'turtle2'                                 # 目标坐
标系


        if self.turtle_spawning_service_ready:                     # 如果已
```

```
经请求海龟生成服务
            if self.turtle_spawned:                                    # 如果跟
随海龟已经生成
                try:
                    now = rclpy.time.Time()                            # 获取 ROS
系统的当前时间
                    trans = self.tf_buffer.lookup_transform(           # 监听当
前时刻源坐标系到目标坐标系的坐标变换
                        to_frame_rel,
                        from_frame_rel,
                        now)
                except TransformException as ex:                       # 如果坐
标变换获取失败，进入异常报告
                    self.get_logger().info(
                        f'Could not transform {to_frame_rel} to {from_fram
e_rel}: {ex}')
                    return

                msg = Twist()                                          # 创建速度
控制消息
                scale_rotation_rate = 1.0                              # 根据海
龟角度，计算角速度
                msg.angular.z = scale_rotation_rate * math.atan2(
                    trans.transform.translation.y,
```

```
                trans.transform.translation.x)



            scale_forward_speed = 0.5                          # 根据海
龟距离，计算线速度
            msg.linear.x = scale_forward_speed * math.sqrt(
                trans.transform.translation.x ** 2 +
                trans.transform.translation.y ** 2)



            self.publisher.publish(msg)                        # 发布速
度指令，海龟跟随运动
        else:                                                  # 如果跟随
海龟没有生成
            if self.result.done():                             # 查看海
龟是否生成
                self.get_logger().info(
                    f'Successfully spawned {self.result.result().name}
')
                self.turtle_spawned = True
            else:                                              # 依然没有
生成跟随海龟
                self.get_logger().info('Spawn is not finished')
    else:                                                      # 如果没有
请求海龟生成服务
        if self.spawner.service_is_ready():                    # 如果海
```

```
龟生成服务器已经准备就绪
                request = Spawn.Request()                                # 创建一
个请求的数据
                request.name = 'turtle2'                                 # 设置请
求数据的内容，包括海龟名、xy 位置、姿态
                request.x = float(4)
                request.y = float(2)
                request.theta = float(0)


                self.result = self.spawner.call_async(request)      # 发送
服务请求
                self.turtle_spawning_service_ready = True           # 设置标
志位，表示已经发送请求
            else:
                self.get_logger().info('Service is not ready')      # 海龟
生成服务器还没准备就绪的提示



def main(args=None):
    rclpy.init(args=args)                           # ROS2 Python 接口初始化
    node = TurtleFollowing("turtle_following")  # 创建 ROS2 节点对象并进行初
始化
    rclpy.spin(node)                               # 循环等待 ROS2 退出
```

```
    node.destroy_node()                    # 销毁节点对象

    rclpy.shutdown()                       # 关闭 ROS2 Python 接口
```

```
def main(args=None):
    rclpy.init(args=args)
    node = StaticTFBroadcaster("static_tf_broadcaster")
行初始化
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

~
:wq
```

4） 输入指令“**chmod +x turtle_following.py**”，并按下回车，为保存的 turtle_following.py 赋予可执行权限。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/tf_demo/tf_demo$ chmod +x turtle_following.
py
ubuntu@raspberrypi:~/hiwonder_ws/src/tf_demo/tf_demo$
```

4.3 创建 turtle_following.launch.py

在这个例程我们需要开启小海龟仿真器、海龟 1 的坐标系广播、海龟 2 的坐标系广播和海龟跟随控制的四个节点，可以将开启这四个节点的指令写在同一个 launch 文件中，运行 launch 文件即可完成四个节点开启。

1） 输入“**cd ..**”指令，返回上一级目录。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/tf_demo/tf_demo$ cd ..
ubuntu@raspberrypi:~/hiwonder_ws/src/tf_demo$
```

2） 然后输入“**mkdir launch**”指令，创建 launch 文件夹。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/tf_demo$ mkdir launch
```

3） 输入“cd launch/”指令，进入 launch 文件夹。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/tf_demo$ cd launch/
ubuntu@raspberrypi:~/hiwonder_ws/src/tf_demo/launch$
```

4） 输入指令“**vim turtle_following.launch.py**”编辑程序，复制下面程序。如

需修改，再按下“**i**”即可修改。修改完成，按下“**Esc**”，输入“**：wq**”保存并退出。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/tf_demo/launch$ vim turtle_following.launch
.py
```

```
from launch import LaunchDescription

from launch.actions import DeclareLaunchArgument

from launch.substitutions import LaunchConfiguration

from launch_ros.actions import Node




def generate_launch_description():

    return LaunchDescription([

        Node(

            package='turtlesim',

            executable='turtlesim_node',

            name='sim'

        ),

        Node(

            package='tf_demo',

            executable='turtle_tf_broadcaster',

            name='broadcaster1',

            parameters=[

                {'turtlename': 'turtle1'}

            ]
```

```
    ),
    DeclareLaunchArgument(
        'target_frame', default_value='turtle1',
        description='Target frame name.'
    ),
    Node(
        package='tf_demo',
        executable='turtle_tf_broadcaster',
        name='broadcaster2',
        parameters=[
            {'turtlename': 'turtle2'}
        ]
    ),
    Node(
        package='tf_demo',
        executable='turtle_following',
        name='listener',
        parameters=[
            {'target_frame': LaunchConfiguration('target_frame')}
        ]
    ),
])
```

```
            package='tf_demo',
            executable='turtle_tf_broadcaster',
            name='broadcaster1',
            parameters=[
                {'turtlename': 'turtle1'}
            ]
        ),
        DeclareLaunchArgument(
            'target_frame', default_value='turtle1',
:wq
```

5） 输入指令“**chmod +x turtle_following.launch.py**”，并按下回车，为保存的 turtle_following.launch.py 赋予可执行权限。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/tf_demo/launch$ chmod +x turtle_following.l
aunch.py
ubuntu@raspberrypi:~/hiwonder_ws/src/tf_demo/launch$
```

4.3 setup.py 文件设置

setup.py 文件是定义一个 ROS2 包的远数据和构建配置，提供了包的元数据、依赖关系、构建配置和安装逻辑等信息，可以帮助开发者正确构建、安装和使用 ROS2 包。需要将 **turtle_tf_broadcaster.py** 和 **turtle_following.py** 的程序入口写到 setup.py 文件中。

5） 输入“**cd ..**”指令，切换到上一级目录。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/topic_demo/topic_demo$ cd ..
ubuntu@raspberrypi:~/hiwonder_ws/src/topic_demo$
```

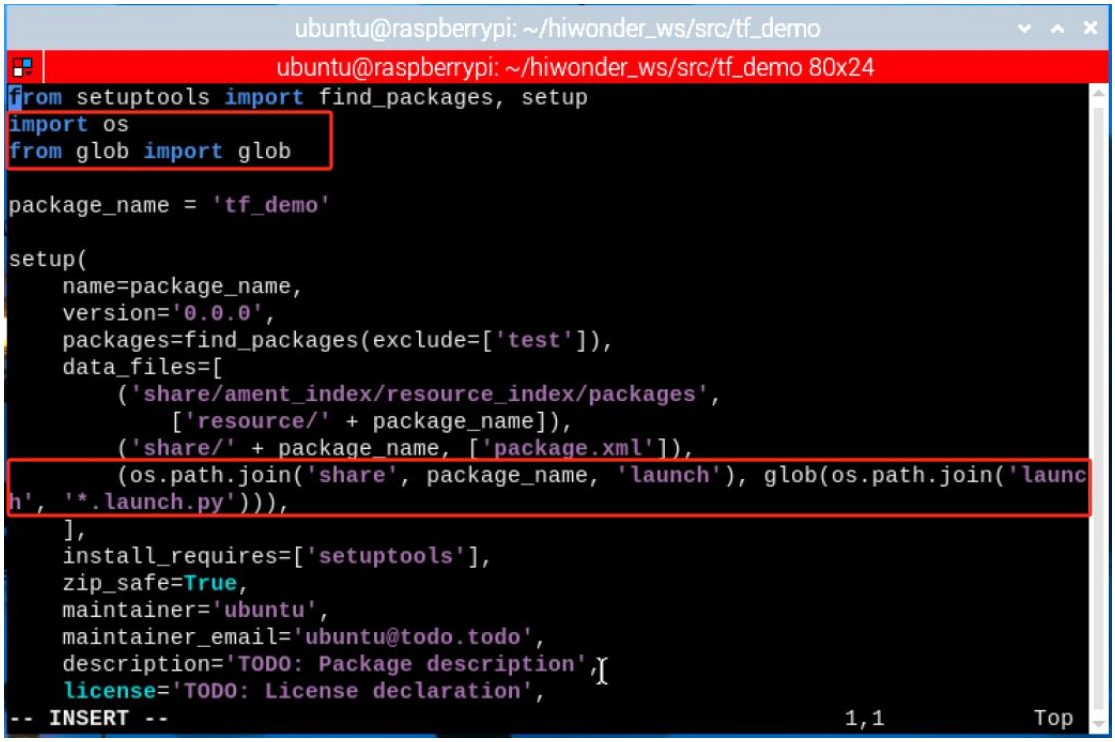
6） 输入“**vim setup.py**”指令，并按下回车，打开 setup.py 文件。

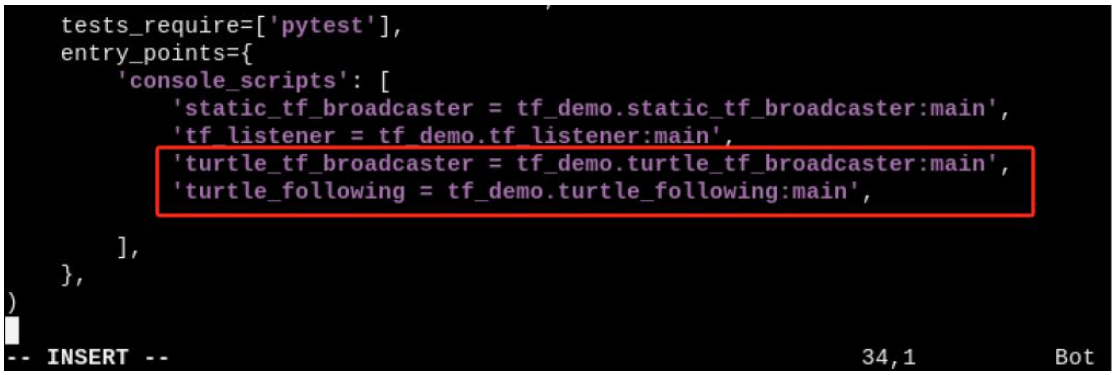
```
ubuntu@raspberrypi:~/hiwonder_ws/src/topic_demo$ vim setup.py
```

7） 按下“i”键进入可编辑模式，输入以下代码到相应的位置上。

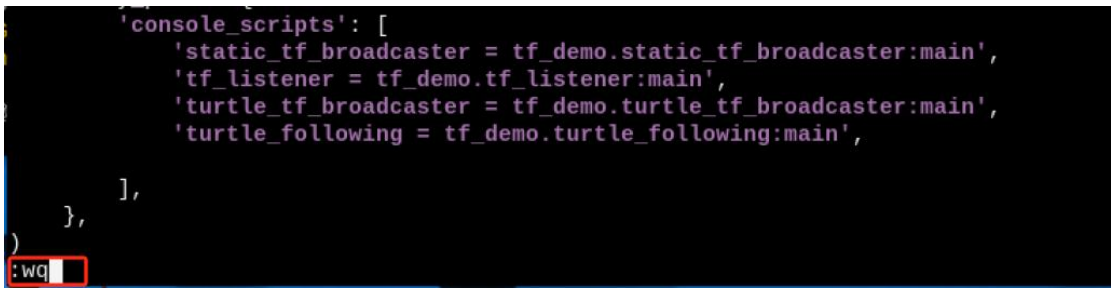
```
import os

from glob import glob

(os.path.join('share', package_name, 'launch'), glob(os.path.join('launc
h', '*.launch.py'))),
```

```
'turtle_tf_broadcaster = tf_demo.turtle_tf_broadcaster:main',

'turtle_following = tf_demo.turtle_following:main',
```





8） 然后输入“:wq”，保存并退出文件。



4.4 编译运行

1） 赋予可执行权限后，输入“**cd ~/hiwonder_ws/**”指令，切换到工作空间的目录。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/hello_world_demo$ cd ~/hiwonder_ws/
ubuntu@raspberrypi:~/hiwonder_ws$
```

2） 输入“**colcon build**”指令，并按下回车，编译工作空间中的功能包。

```
ubuntu@raspberrypi:~/hiwonder_ws$ colcon build
```

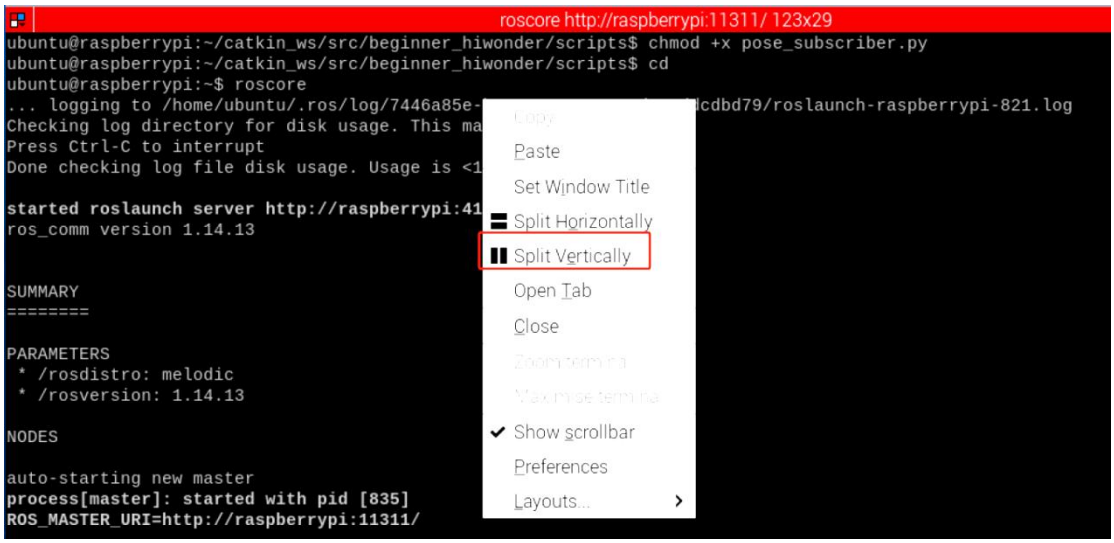
3） 再输入指令“**source ./install/setup.bash**”，并按下回车，让环境变量生效。

```
ubuntu@raspberrypi:~/hiwonder_ws$ source ./install/setup.bash
```

4） 输入指令“**ros2 launch tf_demo turtle_fllowing.launch.py**”，并按下回车，启动 turtle_fllowing.launch.py 文件。

```
ubuntu@raspberrypi:~$ ros2 launch tf_demo turtle_following.launch.py
```

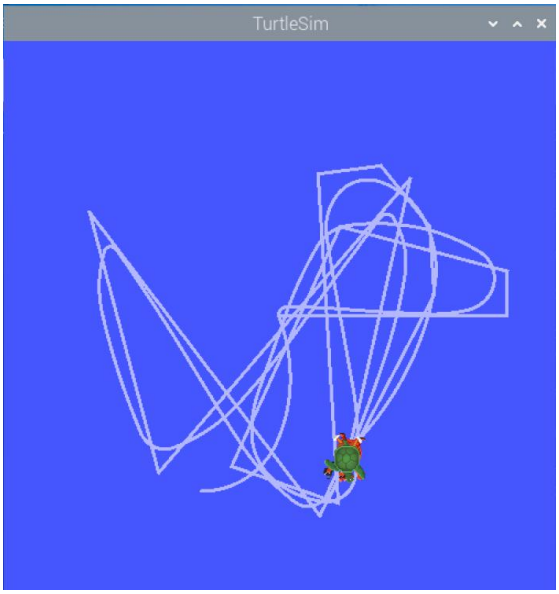
5） 鼠标右键，选择“**Split Vertically**”点击，新建一个新的终端窗口。



6） 输入指令“**ros2 run turtlesim turtle_teleop_key**”，并按下回车，启动 turtle_teleop_key 控制节点。

```
ubuntu@raspberrypi:~$ ros2 run turtlesim turtle_teleop_k
ey
Reading from keyboard
---------------------------
Use arrow keys to move the turtle.
Use G|B|V|C|D|E|R|T keys to rotate to absolute orientati
ons. 'F' to cancel a rotation.
'Q' to quit.
```

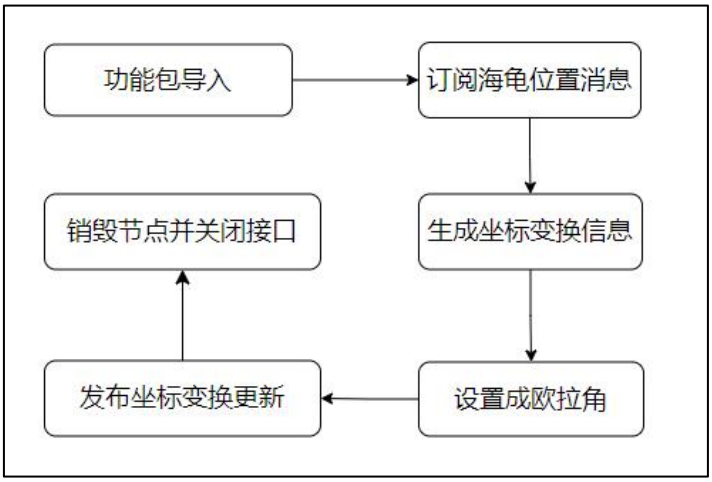
按“↑↓←→”键控制小海龟移动，另外一只小海龟也会跟着运动。



4.5 程序分析

4.5.1 turtle_tf_broadcaster.py

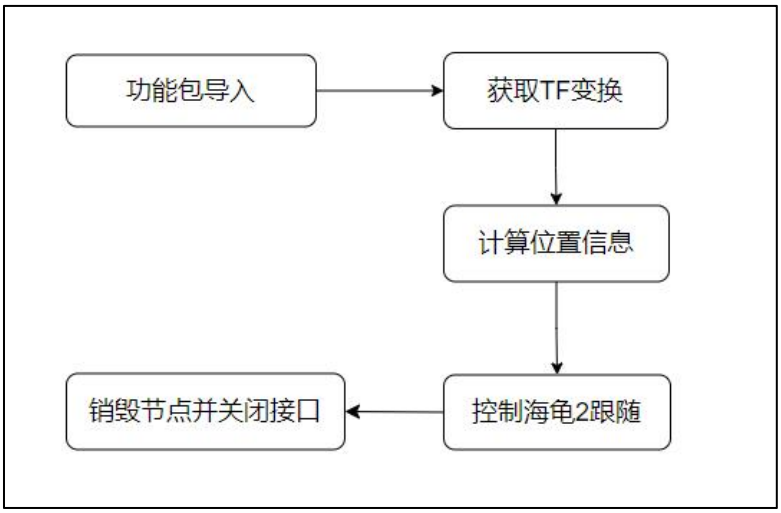
根据实现的效果，梳理程序的过程逻辑，如下图所示：



创建了 TurtleTFBroadcaster 节点类，初始化节点对象时创建 TF 广播器，进而订阅小海龟位置消息，在位置回调函数中根据消息生成世界坐标到海龟坐标的坐标变换消息，设置平移为位置，旋转利用欧拉角转换成四元数，并通过广播器实时将坐标变换信息传播出去，从而能根据小海龟实时位置动态更新它在世界坐标下的坐标变换关系，为后续规划控制等节点和用户提供小海龟实时坐标转换支持。

4.5.2 turtle_following.py

根据实现的效果，梳理程序的过程逻辑，如下图所示：



创建了 TurtleFollowing 节点类，初始化节点后创建了 TF 监听器和速度 publisher，通过定期查询 TF 获取源海龟与目标海龟的坐标关系，计算出目标海龟的速度指令值，第一步请求生成跟随海龟，然后按源海龟的位置和姿态实时控制目标海龟运动实现跟随效果，从而基于 TF 坐标变换关系实现了 ROS2 环境下两个海龟间的跟随控制应用。

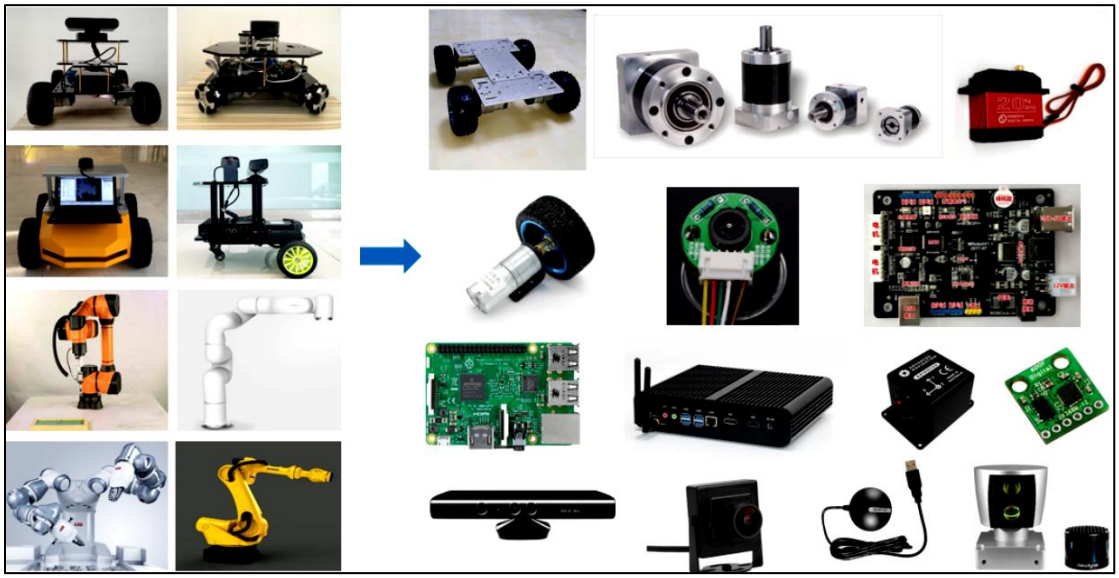
第 17 课 ROS2 URDF 机器人建模方法

1.URDF 简介

ROS 中的建模方法叫做 URDF，全称是统一机器人描述格式，不仅可以清晰描述机器人自身的模型，还可以描述机器人的外部环境。URDF 模型文件使用的是 XML 格式。

2.机器人的组成

建模描述机器人的过程中，我们自己需要先熟悉机器人的组成和参数，比如机器人一般是由硬件结构、驱动系统、传感器系统、控制系统四大部分组成，市面上一些常见的机器人，无论是移动机器人还是机械臂，我们都可以按照这四大组成部分进行分解。

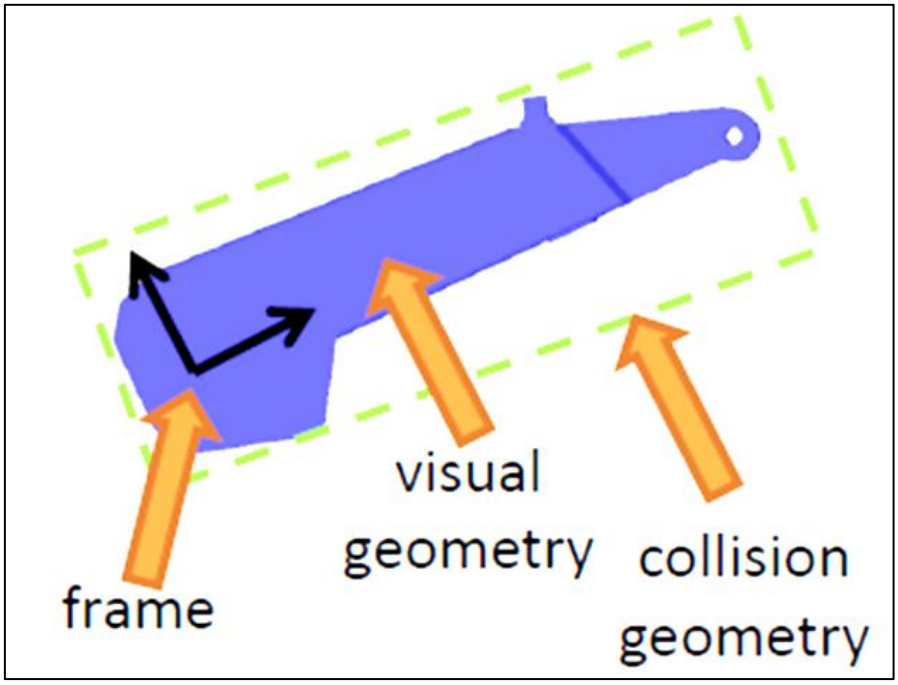


- 硬件结构就是底盘、外壳、电机等实打实可以看到的设备；
- 驱动系统就是可以驱使这些设备正常使用的装置，比如电机的驱动器、电源管理系统等；
- 传感系统包括电机上的编码器、板载的 IMU、安装的摄像头、雷达等等，便于机器人感知自己的状态和外部的环境；
- 控制系统就是我们开发过程的主要载体了，一般是树莓派、电脑等计算平台，以及里边的操作系统和应用软件。

3.URDF 语法

3.1 连杆 Link 的描述

标签用来描述机器人某个刚体部分的外观和物理属性，外观包括尺寸、颜色、形状，物理属性包括质量、惯性矩阵、碰撞参数等。



以这个机械臂连杆为例，它的 link 描述如下：

```
<link name="link 4">
    <visual>
        <geometry>
            <mesh filename="link 4.stl"/>
        </geometry>
        <origin xyz="0 0 0" rpy="0 0 0" />
    </visual>
    <collision>
```

```
    <geometry>
        <cylinder length="0.5" radius="0.1"/>
    </geometry>
    <origin xyz="0 0 -0.05" rpy="0 0 0" />
    </collision>
</link>
```

link 标签中的 name 表示该连杆的名称，我们可以自定义，未来 joint 连接 link 的时候，会使用到这个名称。

link 里边的<visual>部分用来描述机器人的外观，比如：

- <geometry>表示几何形状，里边使用<mesh>调用了一个在三维软件中提前设计好的蓝色外观，就是这个 stl 文件，看上去和真实机器人是一致的。
- <origin>表示坐标系相对初始位置的偏移，分别是 x、y、z 方向上的平移，和 roll、pitch、raw 旋转，不需要偏移的话，就全为 0。

第二部分<collision>，描述碰撞参数，里边的内容似乎和<visual>一样，也有<geometry>和<origin>，看似相同，其实区别还是比较大的。

- <visual>部分重在描述机器人看上去的状态，也就是视觉效果。
- <collision>部分则是描述机器人运动过程中的状态,比如机器人与外界如何接触算作碰撞。

3.2 关节 joint 描述

机器人模型中的刚体最终要通过关节 joint 连接之后，才能产生相对运动。

URDF 中的关节有六种运动类型。

1） continuous，描述旋转运动，可以围绕某一个轴无限旋转，比如小车的轮子，就属于这种类型。

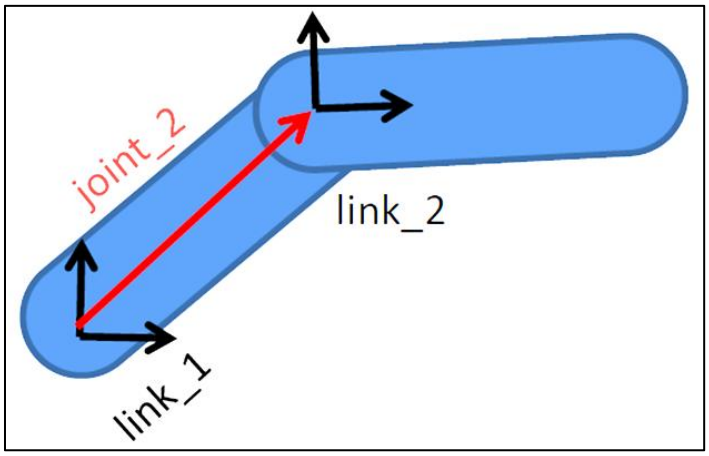
关节类型	描述
continuous	旋转关节，可以围绕单轴无限旋转
revolute	旋转关节，类似于 continuous，但是有旋转的角度极限
prismatic	滑动关节，沿某一轴线移动的关节，带有位置极限
fixed	固定关节，不允许运动的特殊关节
floating	浮动关节，允许进行平移、旋转运动
planar	平面关节，允许在平面正交方向上平移或者旋转

2） revolute，也是旋转关节，和 continuous 类型的区别在于不能无限旋转，而是带有角度限制，比如机械臂的两个连杆，就属于这种运动。

3） prismatic，是滑动关节，可以沿某一个轴平移，也带有位置的极限，一般直线电机就是这种运动方式。

4） fixed，固定关节，是唯一一种不允许运动的关节，不过使用还是比较频繁的，比如相机这个连杆，安装在机器人上，相对位置是不会变化的，此时使用的连接方式就是 Fixed。

5） Floating 是浮动关节，第六种 planar 是平面关节，这两种使用相对较少。



在 URDF 模型中，每一个 link 都使用这样一段 xml 内容描述，比如关节的名字叫什么，运动类型是哪一种。

```
<joint name="joint 2" type="revoluteu">
    <parent link=link 1"/>
    <child link=ulink 2"/>
    <origin xyz="0.2 0.2 0" rpy="0 0 0"/>
    <axis xyz="0 0 1"/>
    <limit lower="-3.14" upper="3.14" velocity="1.0"/>
</joint>
```

- parent 标签：描述父连杆；
- child 标签：描述子连杆，子连杆会相对父连杆发生运动；
- origin：表示两个连杆坐标系之间的关系，也就是图中红色的向量，可以理解为这两个连杆该如何安装到一起；
- axis 表示关节运动轴的单位向量，比如 z 等于 1，就表示这个旋转运动是围绕 z 轴的正方向进行的；
- limit 就表示运动的一些限制了，比如最小位置，最大位置，和最大速度等。

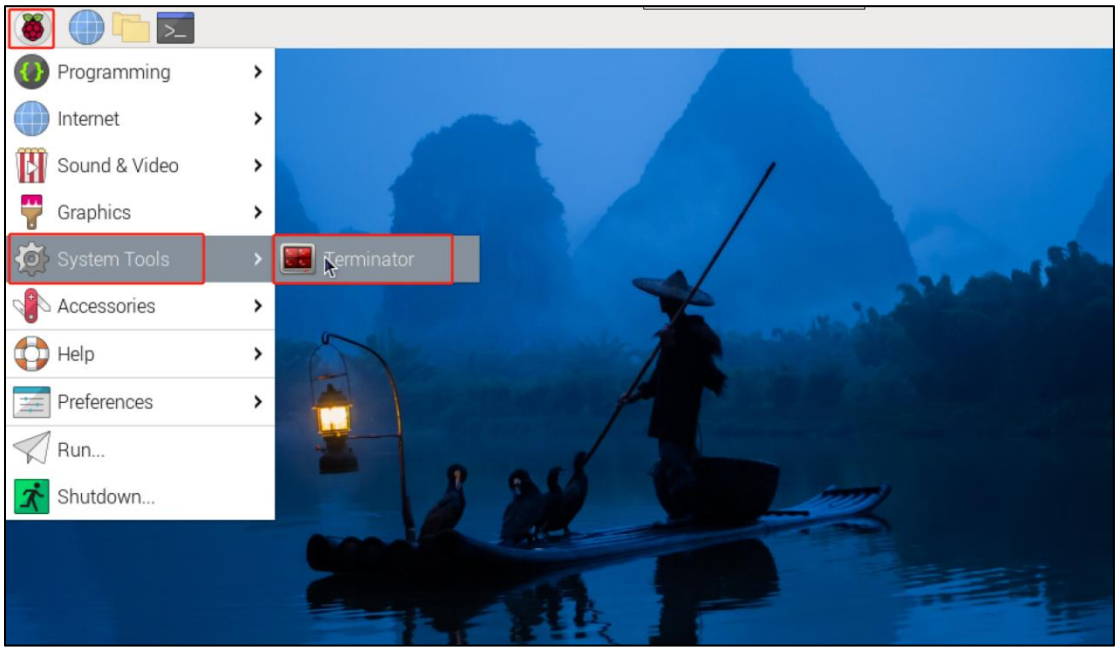
注意：ROS 中关于平移的默认单位是 m，旋转是弧度（不是度），所以这里的 3.14 就

表示可以在-180 度到 180 度之间运动，线速度是 m/s，角速度是 rad/s。

4.完整机器人模型

4.1 创建机器人模型

1） 点击左上角的 logo，选择 System Tools →Terminator。



2） 输入“**cd hiwonder_ws/src/**”指令，切换到 hiwonder_ws 工作空间的 src 文件夹中。

```
ubuntu@raspberrypi:~$ cd hiwonder_ws/src/
ubuntu@raspberrypi:~/hiwonder_ws/src$
```

3） 输入“**ros2 pkg create urdf_demo --build-type ament_python --dependencies rclpy**”并按下回车，创建一个名为“**urdf_demo**”的功能包，添加依赖关系 rclpy。

```
ubuntu@raspberrypi:~/hiwonder_ws/src$ ros2 pkg create urdf_demo --build-type ament_python --dependencies rclpy
```

4） 输入“**cd urdf_demo/**”指令，切换到 urdf_demo 功能包下。

```
ubuntu@raspberrypi:~/hiwonder_ws/src$ cd urdf_demo/
ubuntu@raspberrypi:~/hiwonder_ws/src/urdf_demo$
```

5） 输入“mkdir urdf”指令，创建 urdf 文件夹。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/urdf_demo$ mkdir urdf
```

6） 输入“**cd urdf/**”指令，切换到 urdf 文件夹下。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/urdf_demo$ cd urdf
ubuntu@raspberrypi:~/hiwonder_ws/src/urdf_demo/urdf$
```

7） 输入指令“**vim simple_demo.urdf**”编辑程序，复制下面程序。如需修改，再按下“**i**”即可修改。修改完成，按下“**Esc**”，输入“**：wq**”保存并退出。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/urdf_demo/urdf$ vim simple_demo.urdf
```

```
<?xml version="1.0"?>

<robot name="materials">



  <material name="blue">

    <color rgba="0 0 0.8 1"/>

  </material>



  <material name="white">

    <color rgba="1 1 1 1"/>

  </material>







  <link name="base_link">

    <visual>
```

```
    <geometry>
      <cylinder length="0.6" radius="0.2"/>
    </geometry>
    <material name="blue"/>
  </visual>
</link>

<link name="right_leg">
  <visual>
    <geometry>
      <box size="0.6 0.1 0.2"/>
    </geometry>
    <origin rpy="0 1.57075 0" xyz="0 0 -0.3"/>
    <material name="white"/>
  </visual>
</link>

<joint name="base_to_right_leg" type="fixed">
  <parent link="base_link"/>
  <child link="right_leg"/>
  <origin xyz="0 -0.22 0.25"/>
</joint>
```

```
  <link name="left_leg">
    <visual>
      <geometry>
        <box size="0.6 0.1 0.2"/>
      </geometry>
      <origin rpy="0 1.57075 0" xyz="0 0 -0.3"/>
      <material name="white"/>
    </visual>
  </link>

  <joint name="base_to_left_leg" type="fixed">
    <parent link="base_link"/>
    <child link="left_leg"/>
    <origin xyz="0 0.22 0.25"/>
  </joint>

</robot>
```

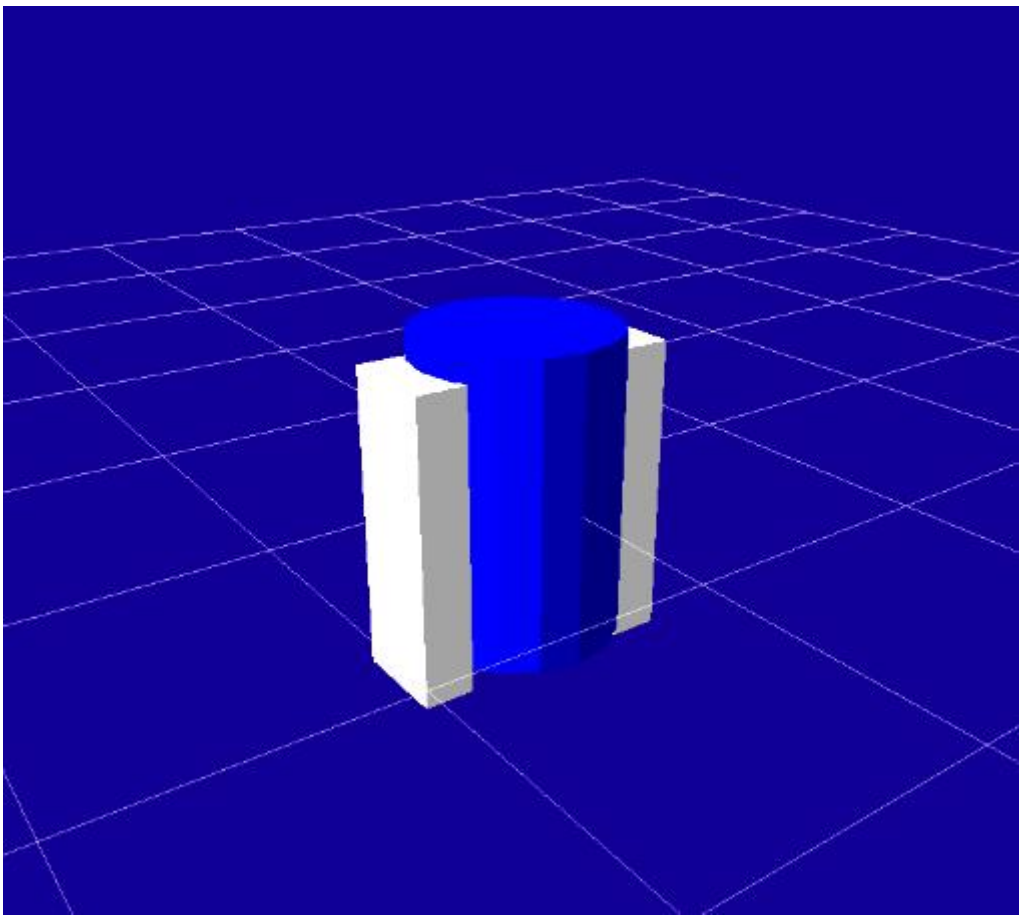
```
  <joint name="base_to_left_leg" type="fixed">
    <parent link="base_link"/>
    <child link="left_leg"/>
    <origin xyz="0 0.22 0.25"/>
  </joint>

</robot>

:wq
```

8） 输入指令“**chmod +x simple_demo.urdf**”，并按下回车，为保存的 simple_demo.urdf 赋予可执行权限。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/urdf_demo/urdf$ chmod +x simple_demo.urdf
ubuntu@raspberrypi:~/hiwonder_ws/src/urdf_demo/urdf$
```

4.2 机器人模型加载

上面代码加载出来的模型如下所示：



定义了一个简单的机器人模型,主要组成和功能如下：

1) 定义了两个材质 - 蓝色和白色。

```
<?xml version="1.0"?>
<robot name="materials">

  <material name="blue">
    <color rgba="0 0 0.8 1"/>
  </material>

  <material name="white">
    <color rgba="1 1 1 1"/>
  </material>
```

2) 定义了基座连杆(base_link),使用蓝色材质和圆柱体几何.

```
<link name="base_link">
  <visual>
    <geometry>
      <cylinder length="0.6" radius="0.2"/>
    </geometry>
    <material name="blue"/>
  </visual>
</link>
```

3) 定义了右腿和左腿连杆,使用白色材质和箱体几何。

```
<link name="right_leg">
  <visual>
    <geometry>
      <box size="0.6 0.1 0.2"/>
    </geometry>
    <origin rpy="0 1.57075 0" xyz="0 0 -0.3"/>
    <material name="white"/>
  </visual>
</link>
```

```
<link name="left_leg">
  <visual>
    <geometry>
      <box size="0.6 0.1 0.2"/>
    </geometry>
    <origin rpy="0 1.57075 0" xyz="0 0 -0.3"/>
    <material name="white"/>
  </visual>
</link>
```

4) 为右腿和左腿定义了固定连接的转动关节,分别连接到基座连杆右腿连接的原点位于(0, -0.22, 0.25),左腿位于(0, 0.22, 0.25),腿部箱体设置了旋转角度,使其竖立直立。

```
<joint name="base_to_right_leg" type="fixed">
  <parent link="base_link"/>
  <child link="right_leg"/>
  <origin xyz="0 -0.22 0.25"/>
</joint>
```

```
<joint name="base_to_left_leg" type="fixed">
  <parent link="base_link"/>
  <child link="left_leg"/>
  <origin xyz="0 0.22 0.25"/>
</joint>
```

主要功能是定义了这个简单机器人的材质、几何形状、连杆及其连接关系。

通过定义材质、连杆、视觉组件、关节等基本语法,完成了这个机器人模型的定义。

第 18 课 ROS2 Gazebo 三维物理仿真平台

1.Gazebo 简介

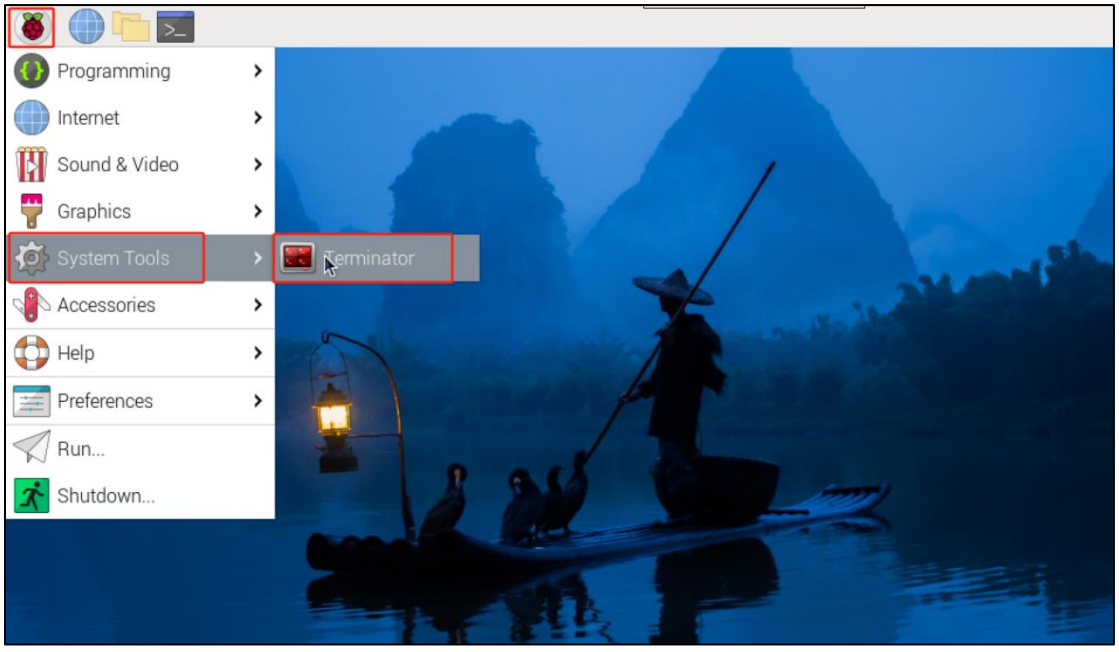
Gazebo 是 ROS 系统中最为常用的三维物理仿真平台，支持动力学引擎，可以实现高质量的图形渲染，不仅可以模拟机器人及周边环境，还可以加入摩擦力、弹性系数等物理属性。

比如我们要开发一个火星车，那就可以在 Gazebo 中模拟火星表面的环境，再比如我们做无人机续航和限飞都导致我们没有办法频繁用实物做实验，此时不妨使用 Gazebo 先做仿真，等算法开发得差不多了，再部署到实物上来运行。

所以类似 Gazebo 这样的仿真平台，可以帮助我们验证机器人算法、优化机器人设计、测试机器人场景应用，为机器人开发提供更多可能。

2.运行 Gazebo

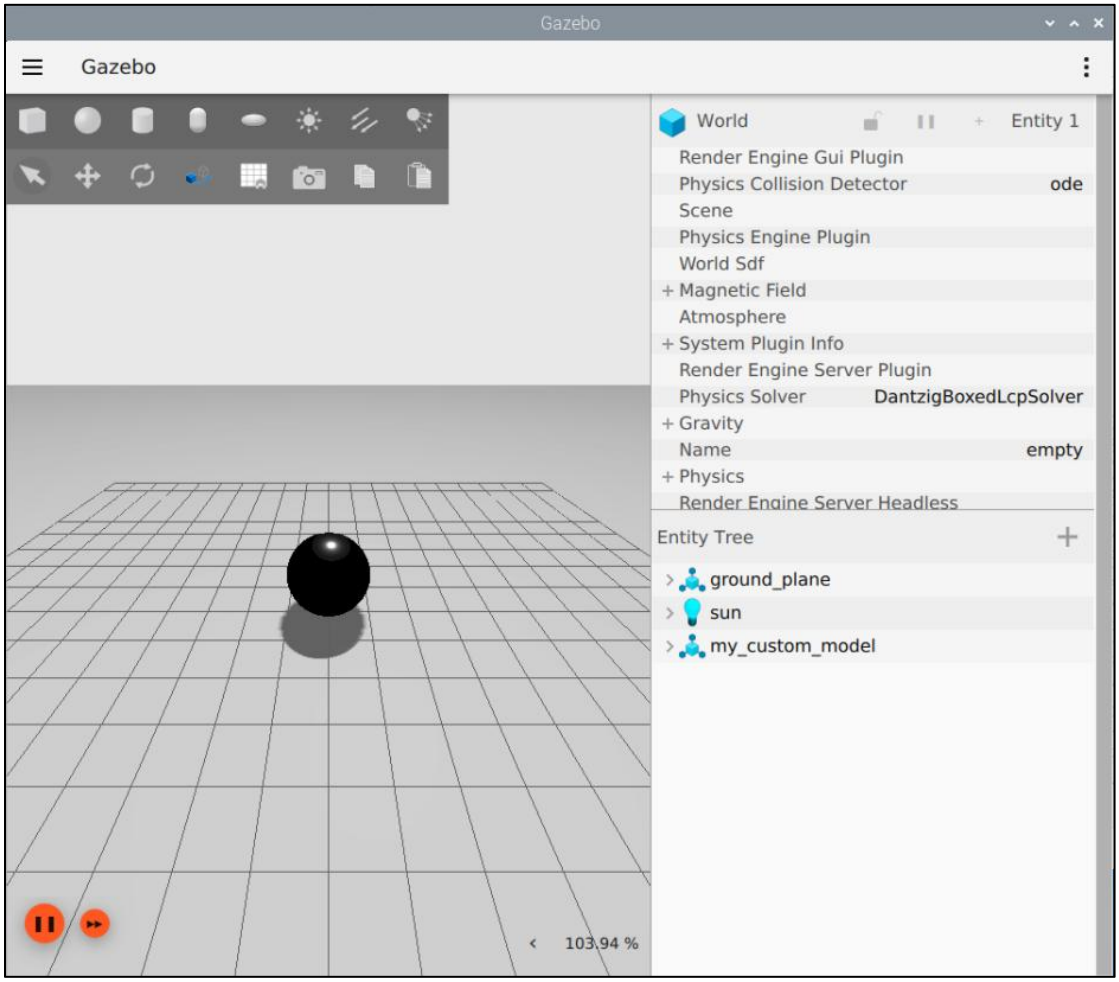
1） 点击左上角的 logo，选择 System Tools →Terminator。

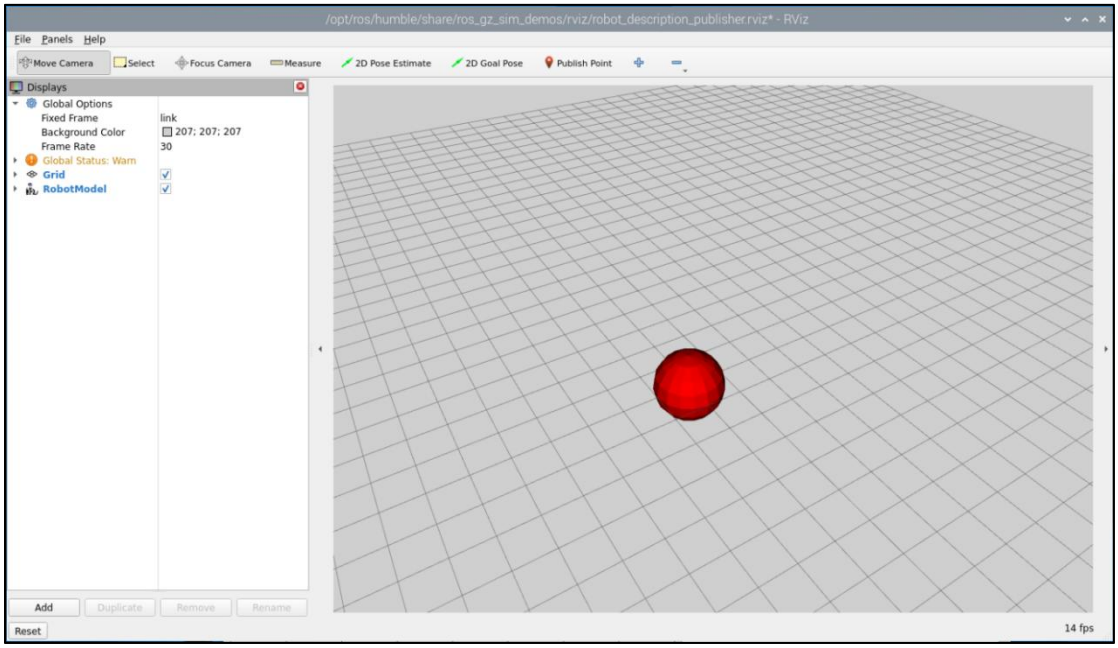


2） 输入“**ros2 launch ros_gz_sim_demos robot_description_publisher.launch.py**”并按下回车，运行 robot_description_publisher.launch.py 文件。

```
ubuntu@raspberrypi:~$ ros2 launch ros_gz_sim_demos robot_description_publisher.launch.py
```

运行成功后，会打开 Ignition 的仿真界面和 Rviz 上位机，我们可以看到小球的图像数据。




/opt/ros/humble/share/ros_gz_sim_demos/rviz/robot_description_publisher.rviz* - RViz
File Panels Help
Move Camera
Select
Focus Camera
Measure
2D Pose Estimate
2D Goal Pose
Publish Point
Displays
Global Options
Fixed Frame
link
Background Color
207; 207; 207
Frame Rate
30
Global Status: Warn
Grid
RobotModel
Add
Duplicate
Remove
Rename
Reset
14 fps

第 19 课 ROS2 Rviz 工具使用

1.Rviz2 简介

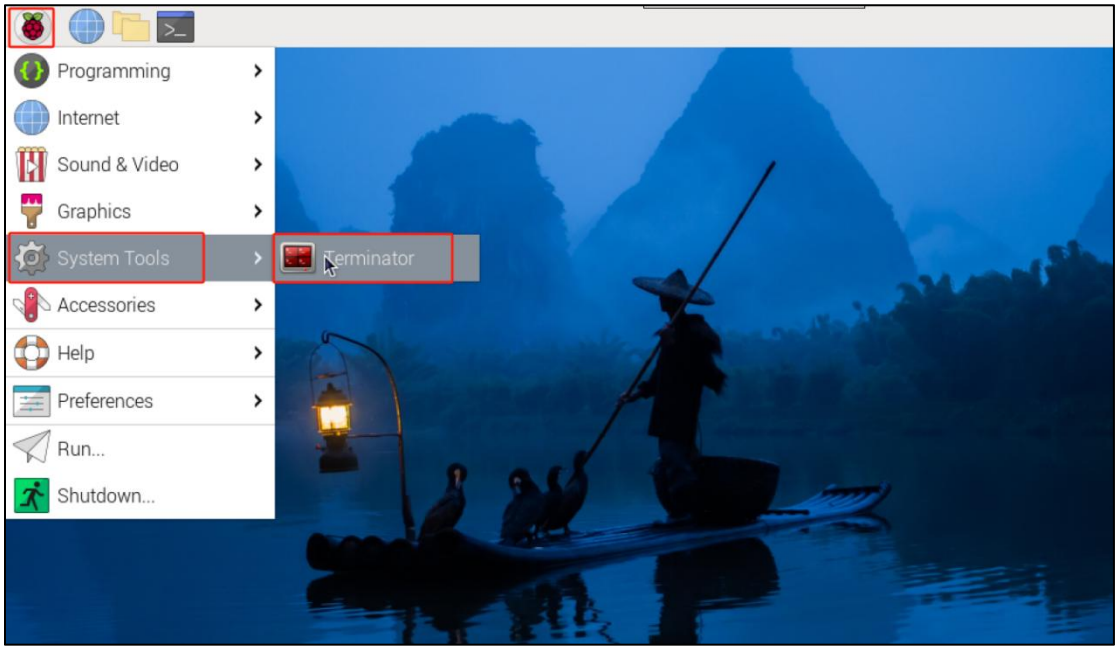
机器人开发过程中，各种各样的功能，如果我们只是从数据层面去做分析，很难快速理解数据的效果，比如机器人模型，我们需要知道自己设计的模型长啥样，还有模型内部众多坐标系在运动过程中都在哪些位置。

再比如机械臂运动规划和移动机器人自主导航，我们希望可以看到机器人周边的环境、规划的路径，当然会还有传感器的信息，摄像头、三维相机、激光雷达等等，数据是用来做计算的，可视化的效果才是给人看的。

所以，数据可视化可以大大提高开发效率，Rviz2 就是这样一款机器人开发过程中的数据可视化软件，机器人模型、传感器信息、环境信息等等，全都可以在这里搞定。

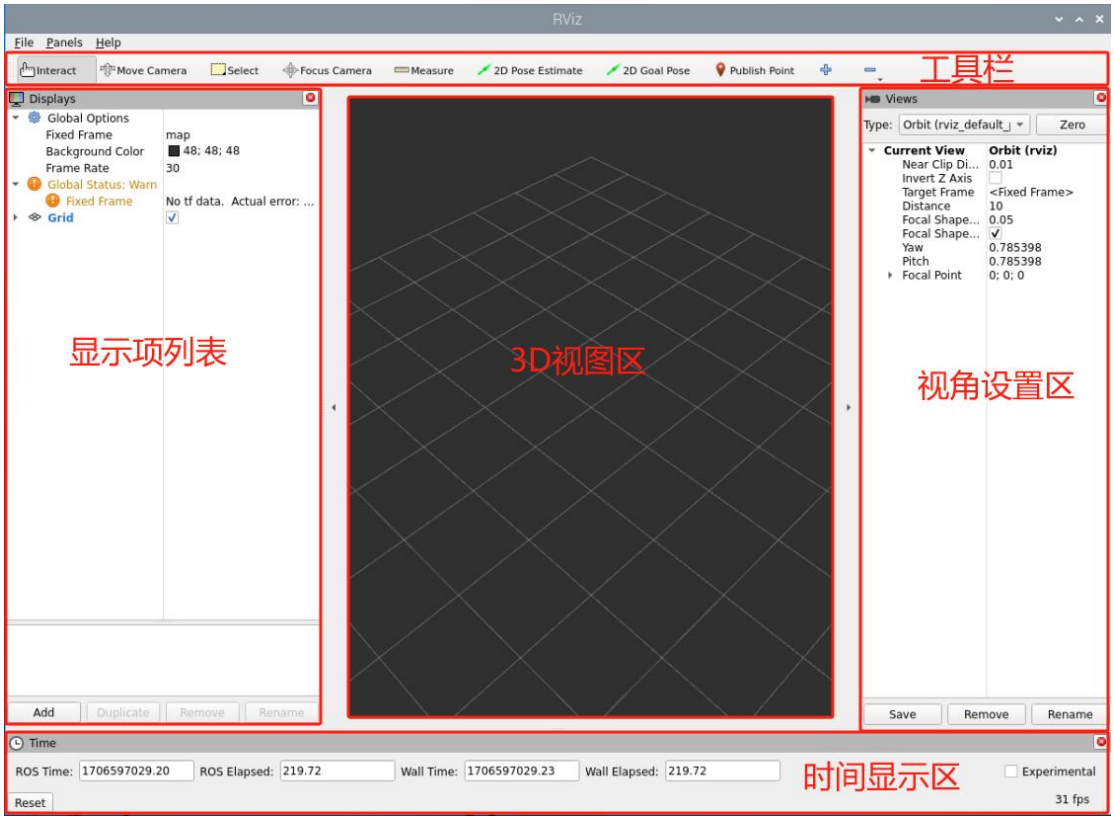
2.Rviz2 启动

1） 点击左上角的 logo，选择 System Tools →Terminator。



2） 输入“**rviz2**”指令，打开 Rviz2 工具。

Rviz 的界面主要可分为五个区域，分别是工具栏区域、显示项列表、3D 视图区、视角设置区和时间显示区。



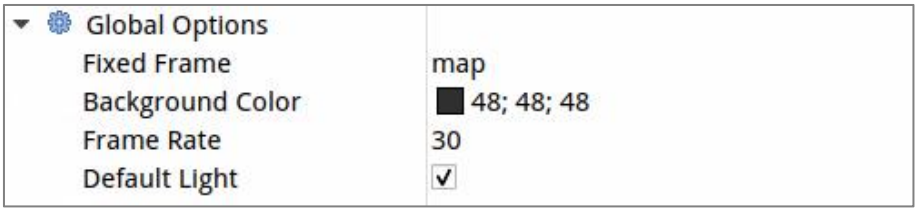
2.1 工具栏区域

图标	功能说明
Interact	选择机器人或者指定位置调整角度。
Move Camera	选中该工具后，在三维视图内长按鼠标左键并拖动鼠标，即可调整相机角度。
Select	选中该工具后，在三维视图内长按鼠标左键，即可框选模型。

	选中该工具后，在三维视图内的任意一处单击鼠标左键，即可将该点设置为地图中心点。
	选中该工具后，在三维视图内的起点与终点处各单击鼠标左键一次，即可测出两点间距。
	设置机器人在三维视图内的位置，此功能仅在导航时可用。 当机器人发生位移，即实际位置与三维视图内的位置不匹配，需重新设置机器人在三维视图内的位置，以免影响导航效果。 选中该工具后，在三维视图内的任意一处单击鼠标左键，即可将该点设置为机器人位置。
	设置机器人的单个目标点，此功能仅在导航时可用。 选中该工具后，在地图显示区域内的任意一处单击鼠标左键，即可将该点设置为目标点。 设置完成后，机器人会自动生成行进路线，并跟随路线移动至目标点。
	设置机器人的多个目标点，此功能仅在导航时可用。 此工具的用法与“2D Nav Goal”大致相同，但此工具可设置多个目标点。每点击一次即可设置一个目标点，最多可设置三个目标点。 设置完成后，机器人会根据设置顺序自动生成行进路线，并跟随路线依次移动至目标点。

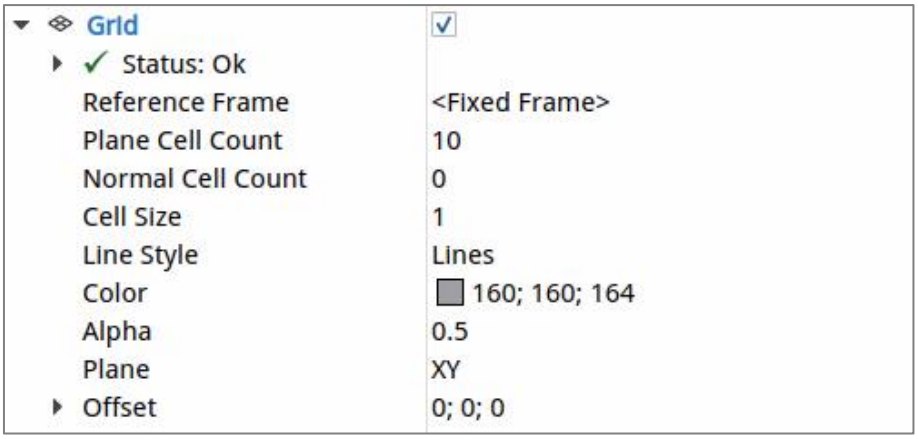
2.2 显示项列表

- **Global Options：全局选项**



图标	功能说明
Fixed Frame	用作所有其他坐标系（frame）的参考坐标系。
Background Color	三维视图的背景颜色。
Frame Rate	三维视图的刷新帧率。
Default Light	当前三维视图的光源。

◆ **Grid：网格**

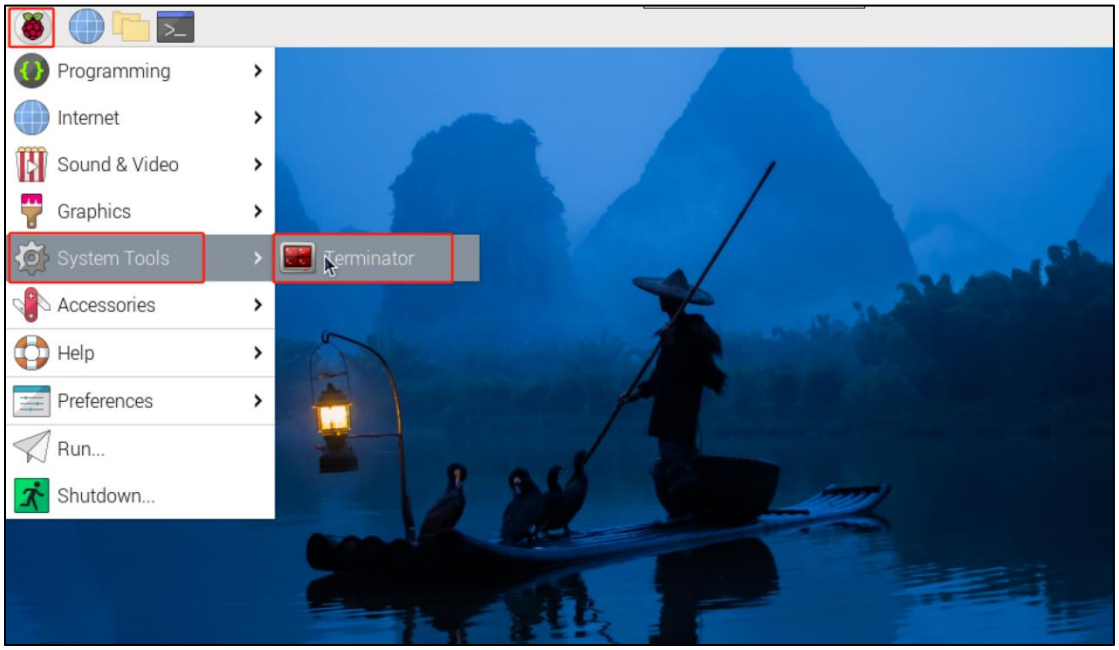


图标	功能说明
Reference Frame	用作网格坐标的参考坐标系。
Plane Cell Count	网格平面中要绘制的单元格数。
Normal Cell Count	沿网格法向量绘制的单元格数。
Cell Size	单元格大小，即每个网格单元格的尺寸，单位为米。
Line Style	网格线条维度。
Color	网格底色。
Alpha	网格线条粗细。
Plane	用于绘制网格的平面。
Offset	从坐标系原点偏移栅格，三个参数分别是 X、Y、Z 轴的偏移量，单位为米。

3.Rviz 加载 URDF 模型

3.1 编写 launch 文件

1） 点击左上角的 logo，选择 System Tools →Terminator。



2） 输入“**cd hiwonder_ws/src/urdf_demo/**”指令，切换到 urdf_demo 工作空间中。

```
ubuntu@raspberrypi:~$ cd hiwonder_ws/src/urdf_demo/
ubuntu@raspberrypi:~/hiwonder_ws/src/urdf_demo$
```

3） 输入“**mkdir rviz launch**”并按下回车，创建 rviz 和 launch 两个文件夹。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/urdf_demo$ mkdir rviz launch
```

4） 输入“**cd launch/**”指令，切换到 launch 文件夹下。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/urdf_demo$ cd launch/
ubuntu@raspberrypi:~/hiwonder_ws/src/urdf_demo/launch$
```

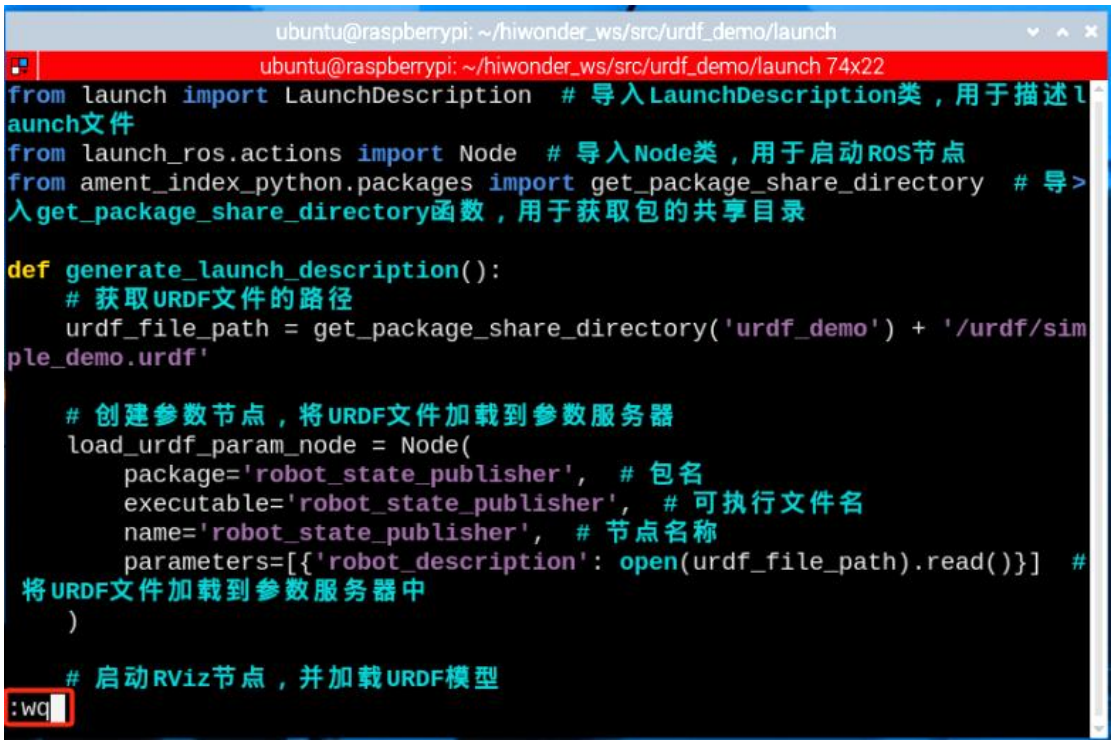
5） 输入指令“**vim rviz_view.launch.py**”编辑程序，复制下面程序。如需修改，再按下“**i**”即可修改。修改完成，按下“**Esc**”，输入“**：wq**”保存并退出。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/urdf_demo/launch$ vim rviz_view.launch.py
```


```
ubuntu@raspberrypi:~/hiwonder_ws/src/urdf_demo/launch$ vim rviz_view.launch.py
```


```
from launch import LaunchDescription  # 导入 LaunchDescription 类，用于描述 launch 文件
from launch_ros.actions import Node  # 导入 Node 类，用于启动 ROS 节点
from ament_index_python.packages import get_package_share_directory  # 导入 get_package_share_directory 函数，用于获取包的共享目录


def generate_launch_description():
    # 获取 URDF 文件的路径
    urdf_file_path = get_package_share_directory('urdf_demo') + '/urdf/simple_demo.urdf'

    # 创建参数节点，将 URDF 文件加载到参数服务器
    load_urdf_param_node = Node(
        package='robot_state_publisher',  # 包名
        executable='robot_state_publisher',  # 可执行文件名
        name='robot_state_publisher',  # 节点名称
        parameters=[{'robot_description': open(urdf_file_path).read()}]  # 将 URDF 文件加载到参数服务器中
    )

    # 启动 RViz 节点，并加载 URDF 模型
```

```
    rviz_node = Node(

        package='rviz2',  # 包名

        executable='rviz2',  # 可执行文件名

        name='rviz2',  # 节点名称

        output='screen',  # 输出屏幕信息

    )



    return LaunchDescription([

        load_urdf_param_node,

        rviz_node

    ])
```



3.2 setup.py 文件设置

setup.py 文件是定义一个 ROS2 包的远数据和构建配置，提供了包的元数据、依赖关系、构建配置和安装逻辑等信息，可以帮助开发者正确构建、安装和使用 ROS2 包。需要将 rviz_view.launch.py 的程序入口写到 setup.py 文件中。

1） 输入“**cd ..**”指令，切换到上一级目录。

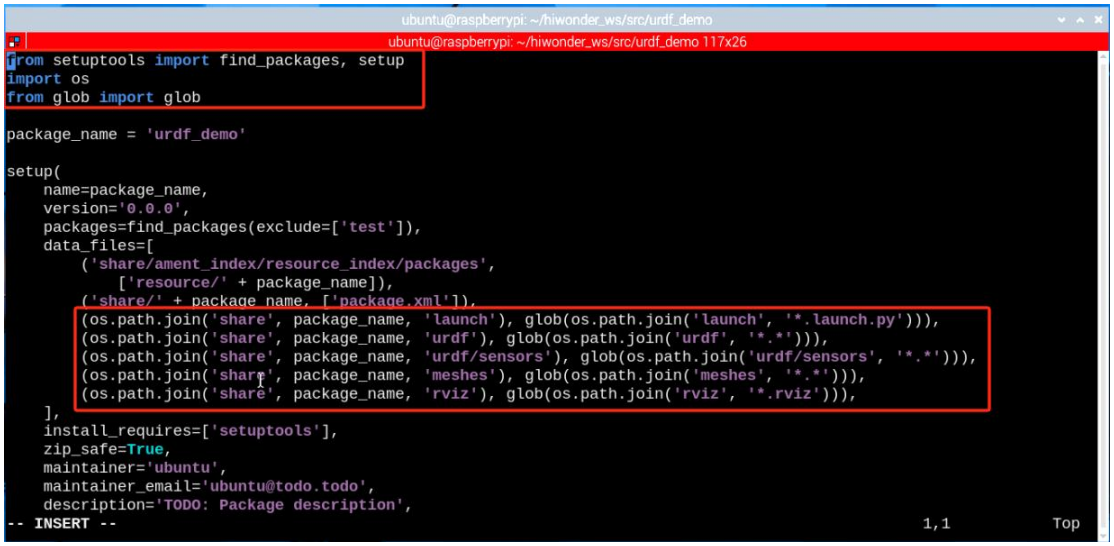
```
ubuntu@raspberrypi:~/hiwonder_ws/src/urdf_demo/launch$ cd ..
ubuntu@raspberrypi:~/hiwonder_ws/src/urdf_demo$
```

2） 输入“**vim setup.py**”指令，并按下回车，打开 setup.py 文件。

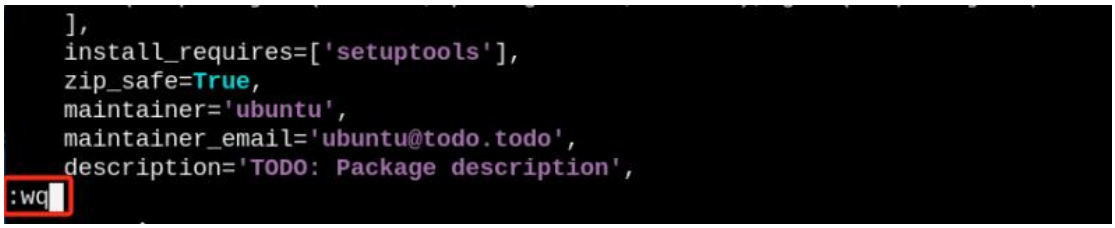
```
ubuntu@raspberrypi:~/hiwonder_ws/src/urdf_demo$ vim setup.py
```

3） 按下“i”键进入可编辑模式，输入以下代码到相应的位置上。

```
from setuptools import find_packages, setup

import os

from glob import glob

 (os.path.join('share', package_name, 'launch'), glob(os.path.join('laun
ch', '*.launch.py'))),

       (os.path.join('share', package_name, 'urdf'), glob(os.path.join
('urdf', '*.*'))),

       (os.path.join('share', package_name, 'urdf/sensors'), glob(os.pa
th.join('urdf/sensors', '*.*'))),

       (os.path.join('share', package_name, 'meshes'), glob(os.path.joi
n('meshes', '*.*'))),

       (os.path.join('share', package_name, 'rviz'), glob(os.path.join
('rviz', '*.rviz'))),
```



4） 然后输入“:wq”，保存并退出文件。



3.3 编译运行

1） 赋予可执行权限后，输入“**cd ~/hiwonder_ws/**”指令，切换到工作空间的目录。

```
ubuntu@raspberrypi:~/hiwonder_ws/src/hello_world_demo$ cd ~/hiwonder_ws/
ubuntu@raspberrypi:~/hiwonder_ws$
```

2） 输入“**colcon build**”指令，并按下回车，编译工作空间中的功能包。

```
ubuntu@raspberrypi:~/hiwonder_ws$ colcon build
```

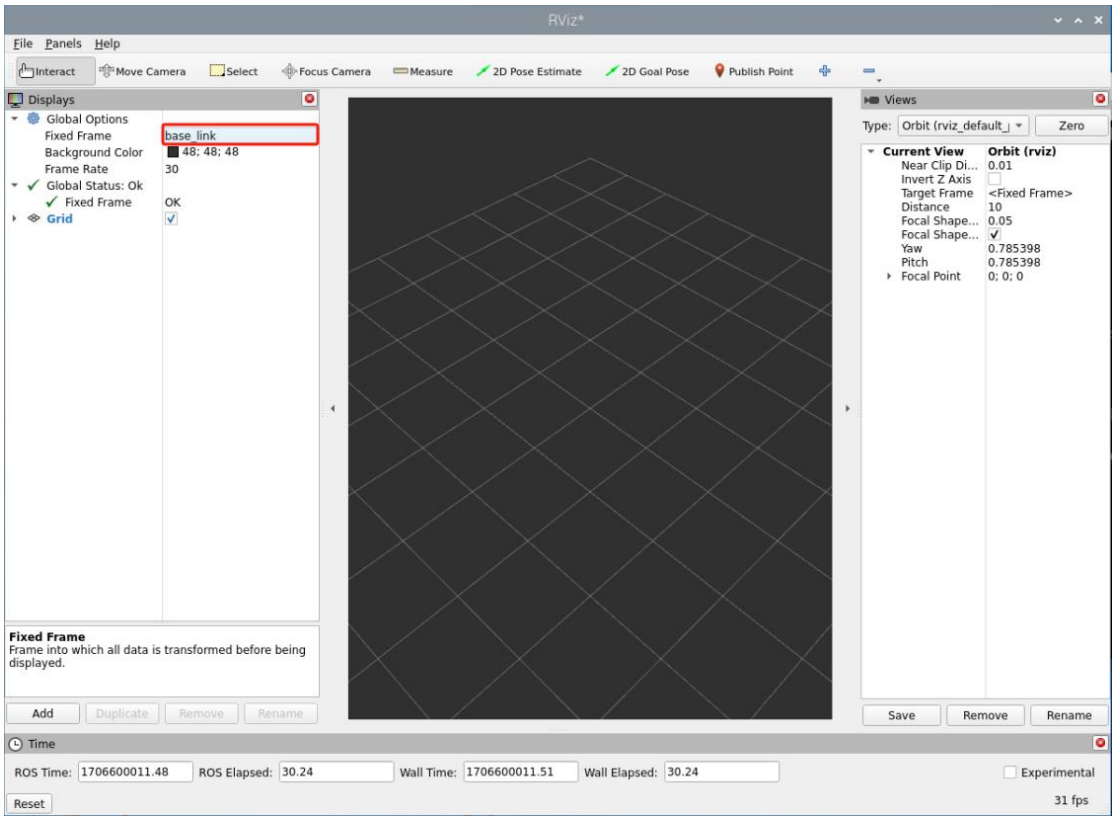
3） 再输入指令“**source ./install/setup.bash**”，并按下回车，让环境变量生效。

```
ubuntu@raspberrypi:~/hiwonder_ws$ source ./install/setup.bash
```

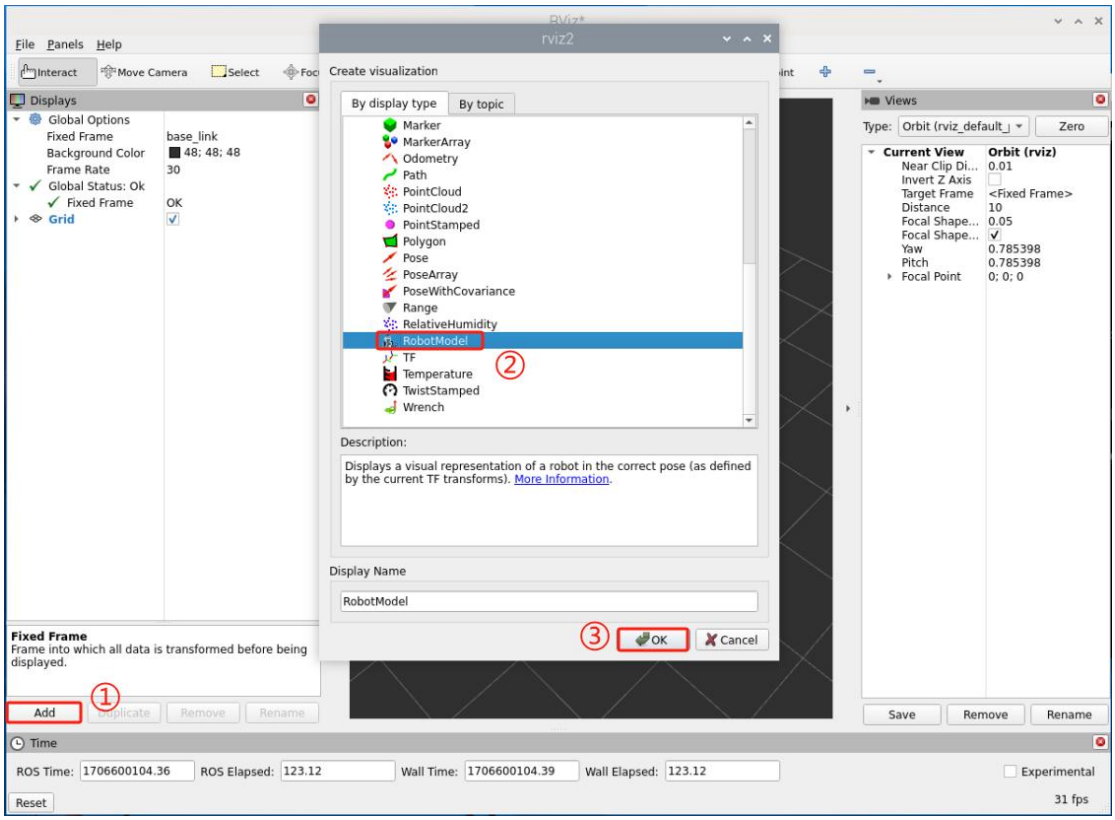
4） 输入指令“**ros2 launch urdf_demo rviz_view.launch.py**”，并按下回车，启动 rviz_view.launch.py 文件。

```
ubuntu@raspberrypi:~$ ros2 launch urdf_demo rviz_view.launch.py
```

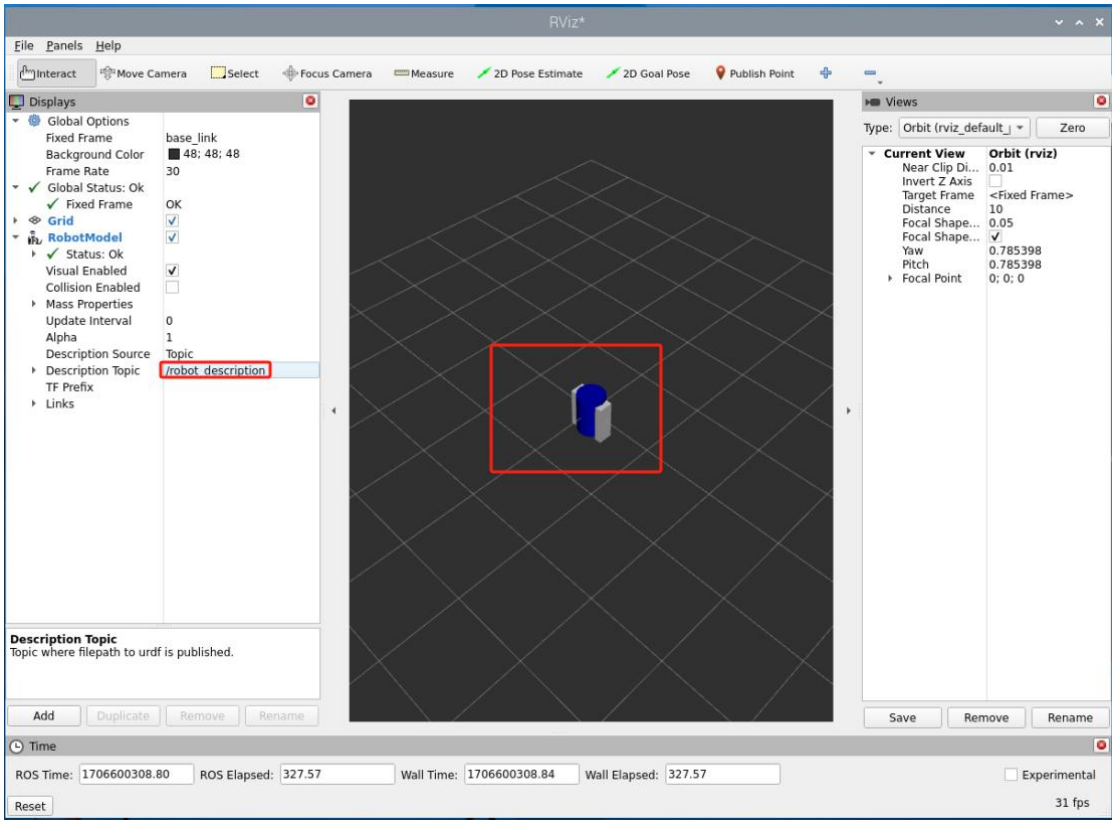
5） 在 Rviz 工具中，选择“**base_link**”。



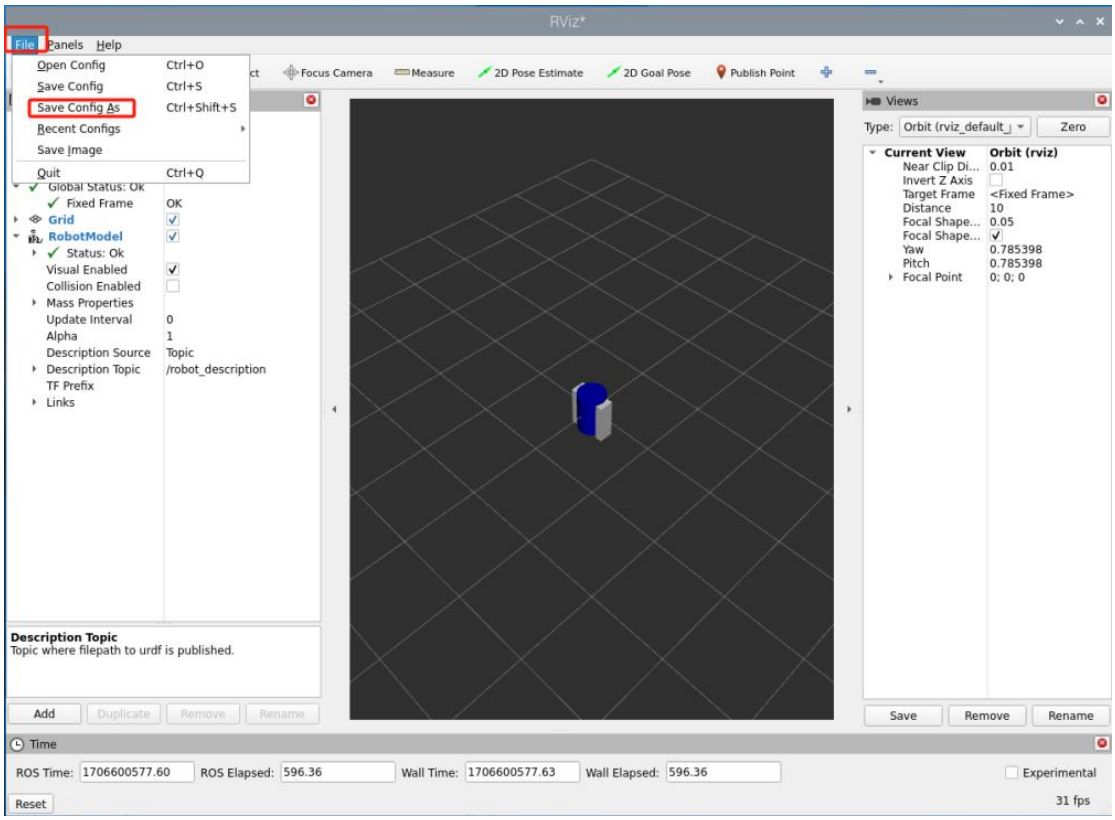
6） 点击“**Add**”按钮，选中“**RobotModel**”模型，然后点击“**OK**”，对模型进行添加。

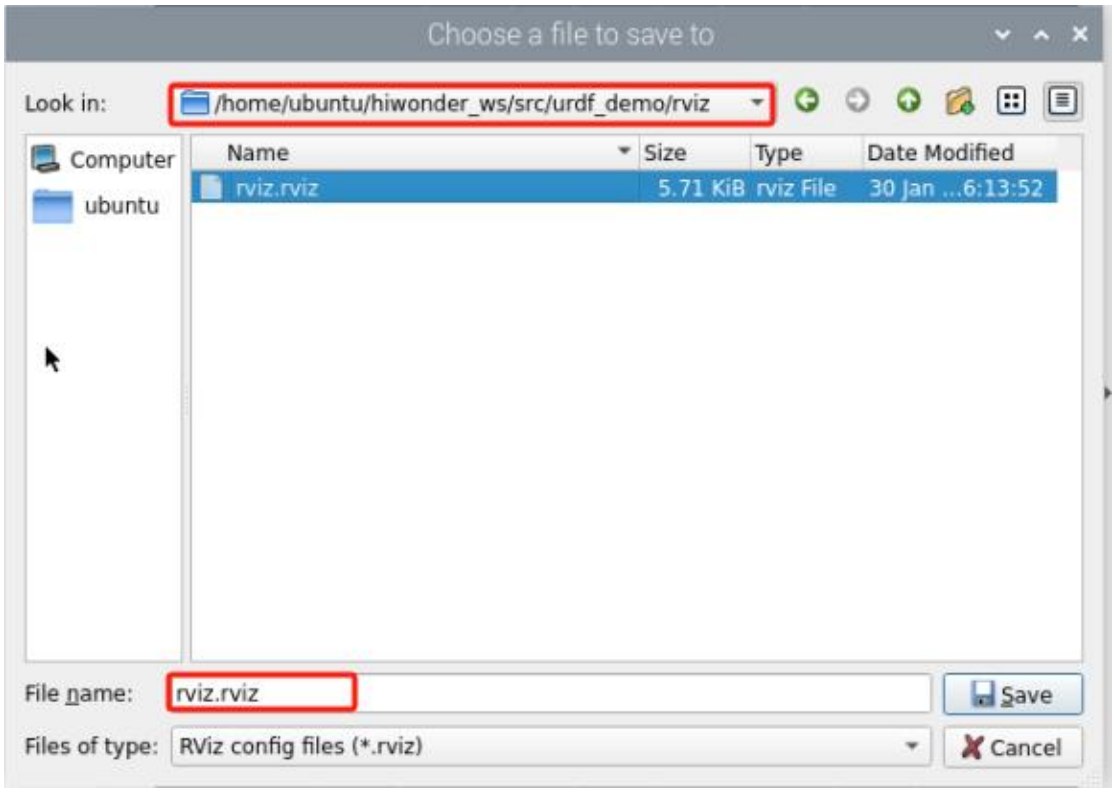


7） 在“**Description Source**”一栏中选择“**/robot_description**”，然后点击空白处，即可对模型进行加载。

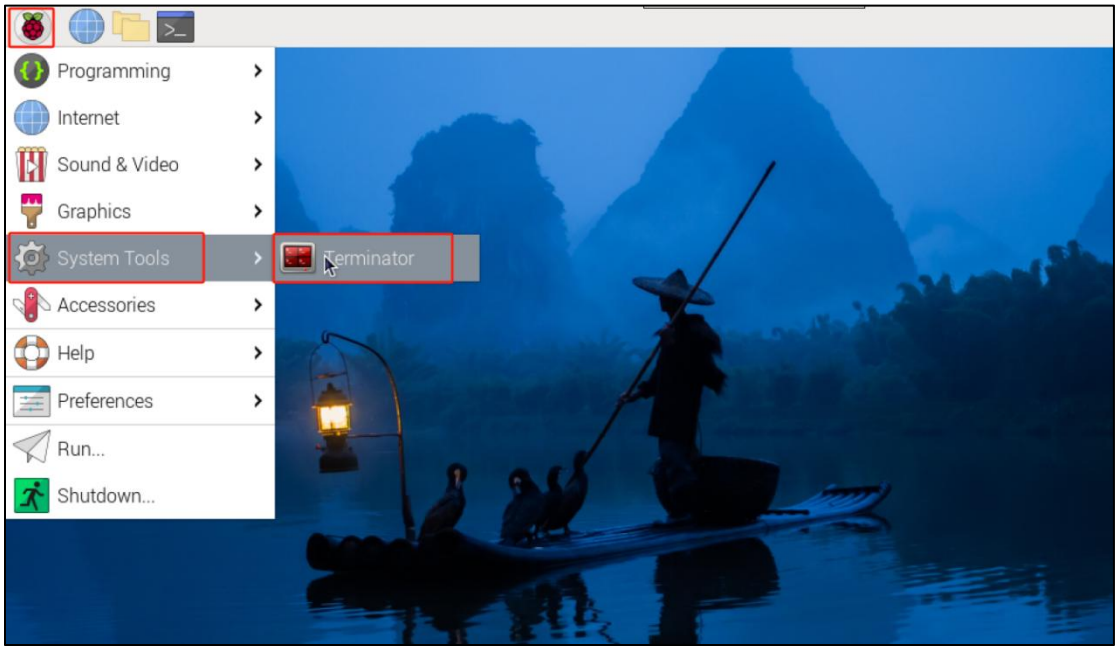


8） 选择左上角 File→Save Config As,将模型设置参数进行保存。

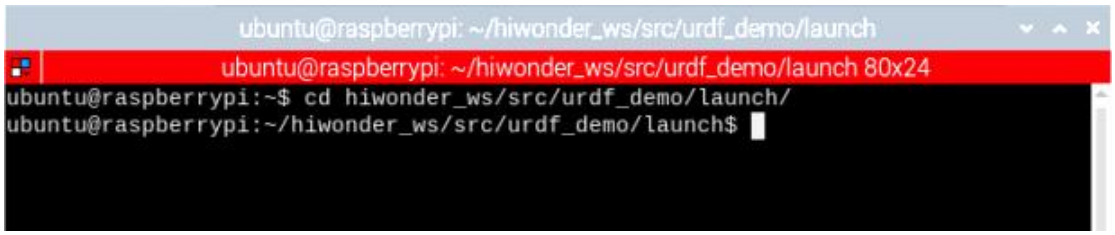




9） 点击左上角的 logo，选择 System Tools →Terminator。

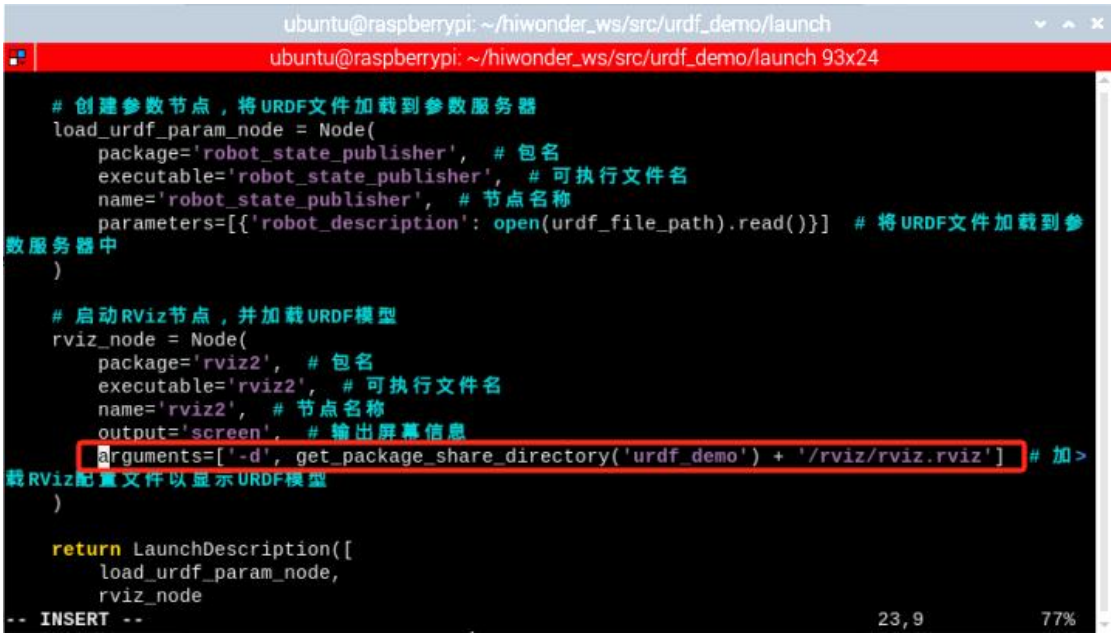


10）输入“**cd hiwonder_ws/src/urdf_demo/launch/**”指令，切换到 launch 文件夹下。

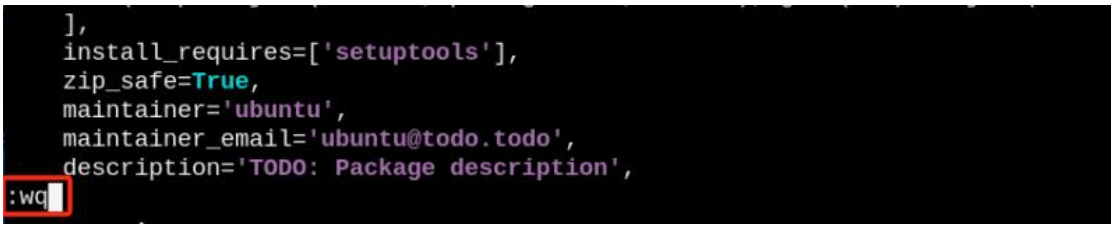


11）输入指令“**vim rviz_view.launch.py**”编辑程序，复制下面程序。如需修改，再按下“**i**”即可修改。修改完成，按下“**Esc**”，输入“**：wq**”保存并退出。


```
arguments=['-d', get_package_share_directory('urdf_demo') + '/rviz/rviz.rviz']  # 加载 RViz 配置文件以显示 URDF 模型
```



12）然后输入“:wq”，保存并退出文件。



13）重新执行 3.3 编译运行步骤，即可完成修改。

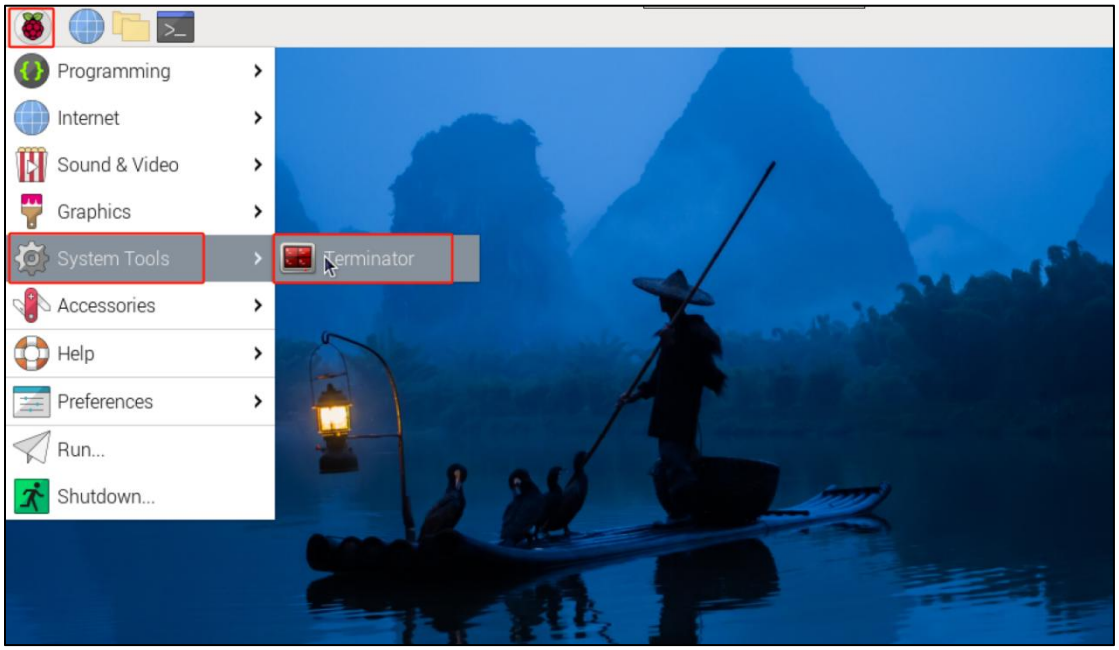
第 20 课 ROS2 RQT 工具使用

1.RQT 简介

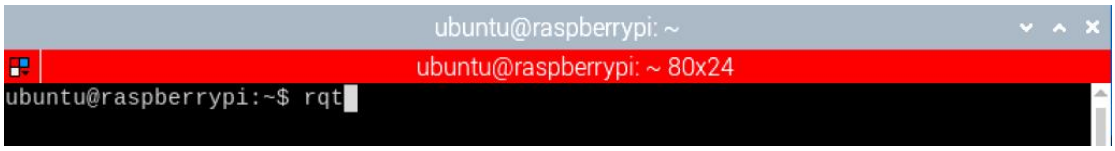
ROS 中的 Rviz 功能已经很强大了，不过有些场景下，我们可能更需要一些简单的模块化的可视化工具，比如只显示一个摄像头的图像，使用 Rviz 的话，难免会觉得操作有点麻烦。此时，我们就会用到 ROS 提供的另外一种模块化可视化工具--RQT。

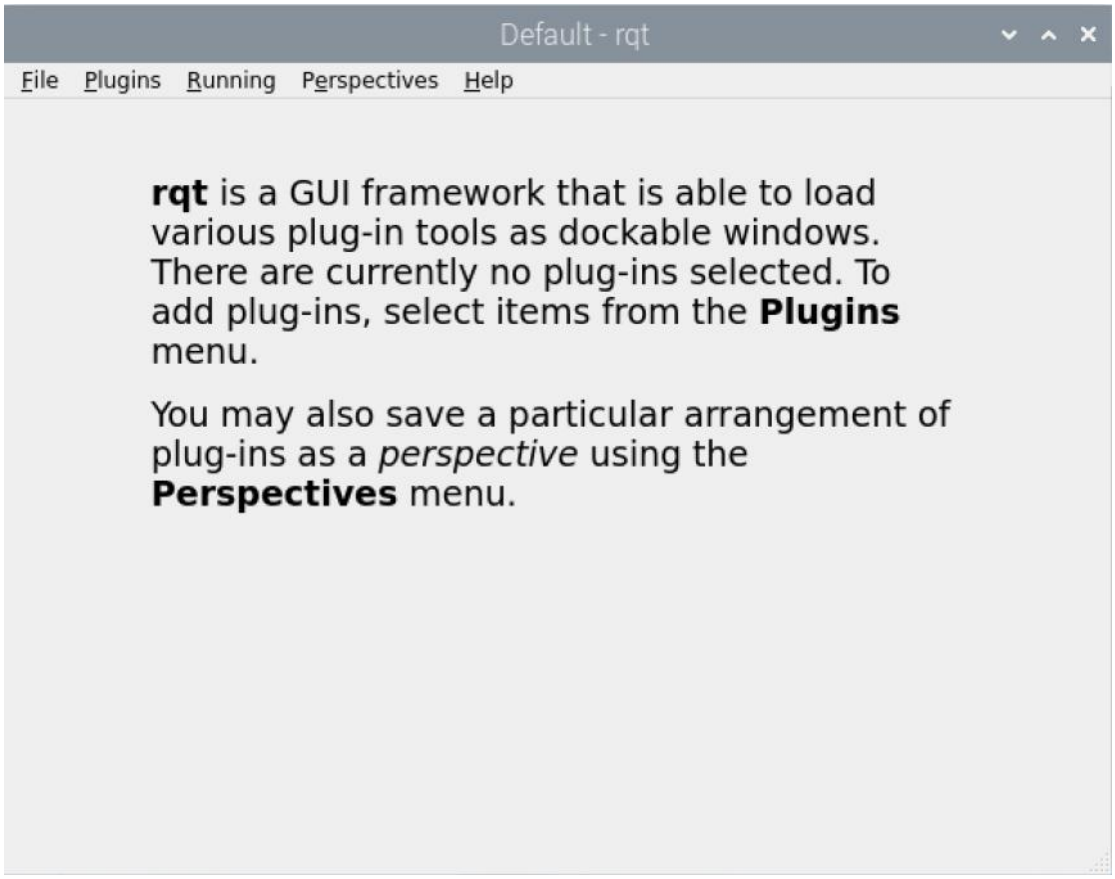
正如 RQT 的命名，它和 Rviz 一样，也是基于 QT 可视化工具而来。

1） 点击左上角的 logo，选择 System Tools →Terminator。



2） 输入“**rqt**”指令，打开 RQT 工具。

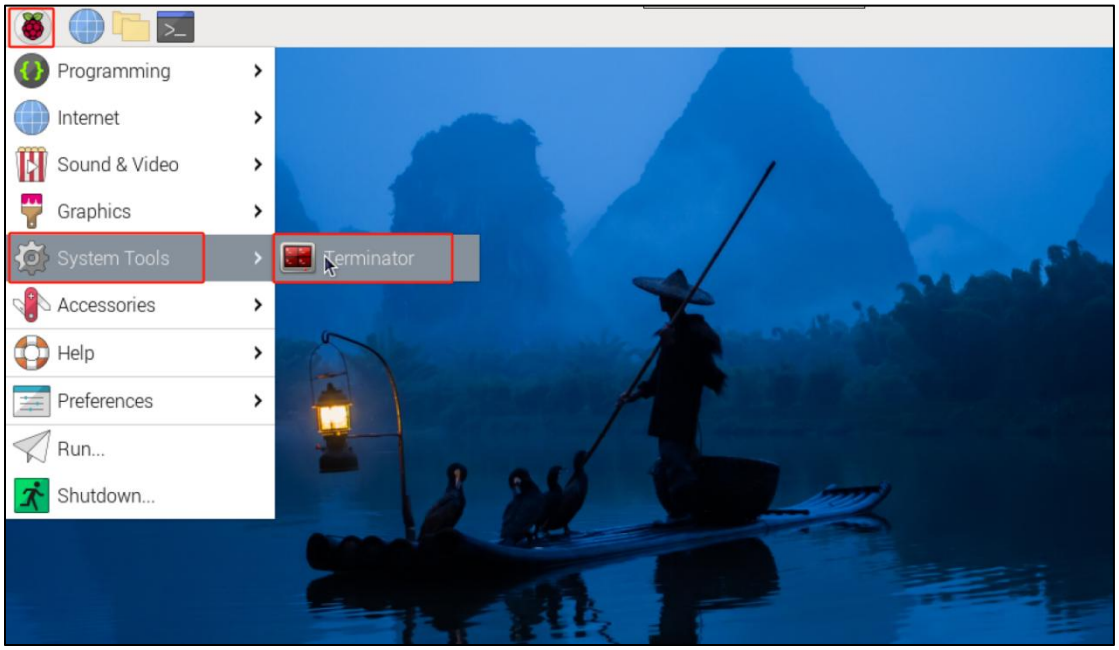




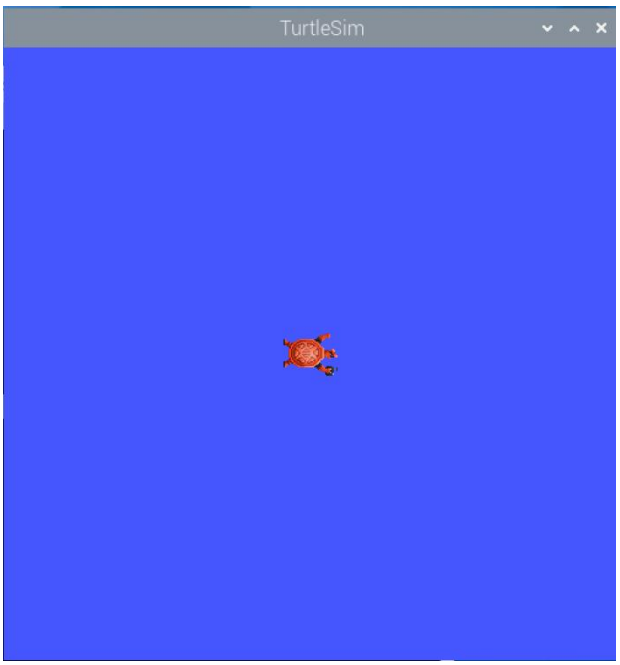
里面可以加载很多小模块，每个模块都可以实现一个具体的的小功能，常用的模块有话题、服务、动作、参数、日志等，可以按照我们的需求选用，方便地调试 ROS2 程序。

2.日志显示

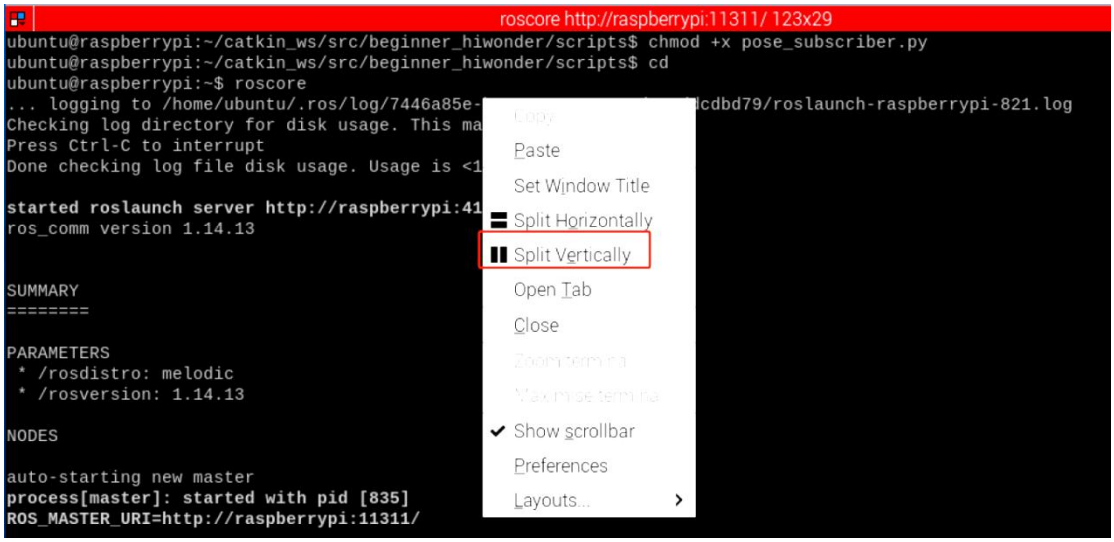
1） 点击左上角的 logo，选择 System Tools →Terminator。



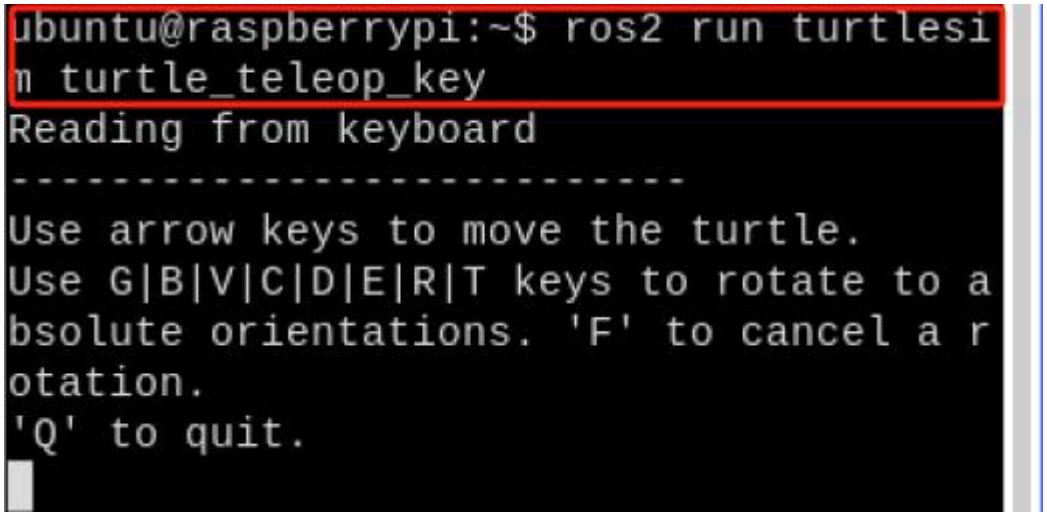
2） 输入“**ros2 run turtlesim turtlesim_node**”指令，启动小海龟节点。



3） 鼠标右键，选择“**Split Vertically**”点击，新建一个新的终端窗口。

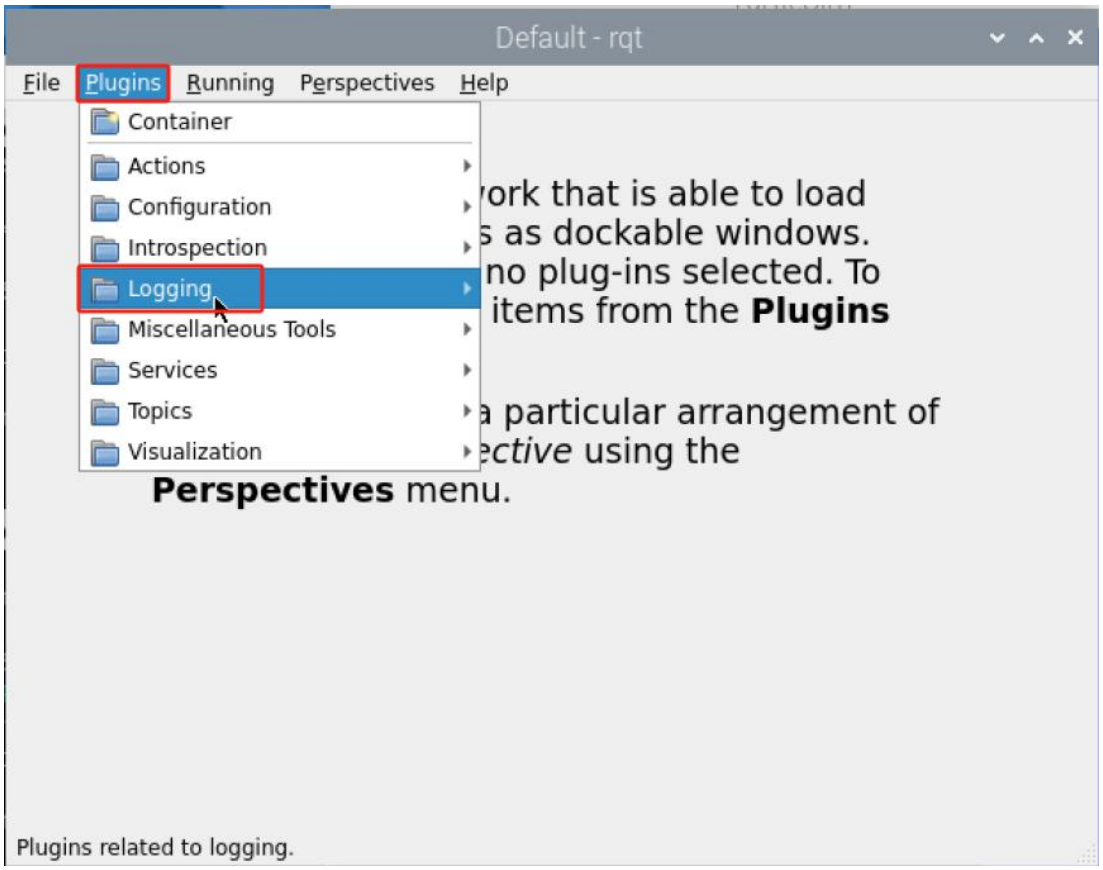


4） 输入“**ros2 run turtlesim turtle_teleop_key**”并按下回车，启动小海龟键盘控制节点。

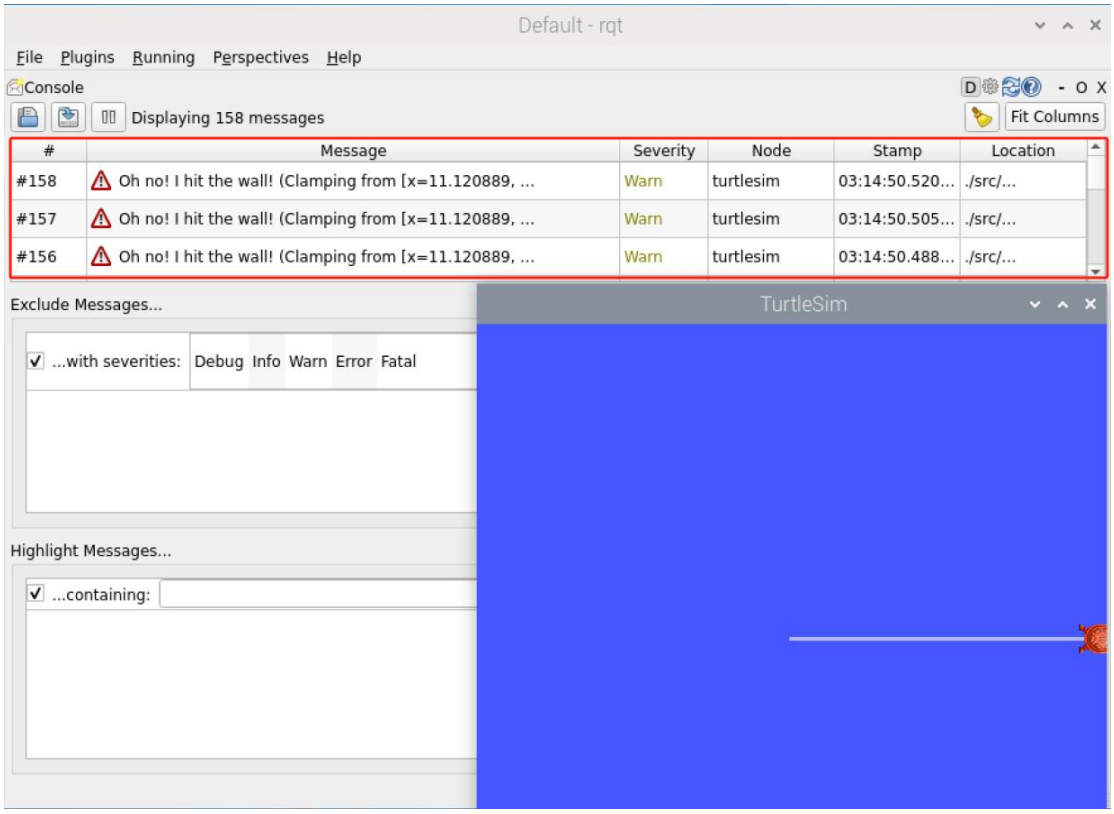


5） 参考 3）步骤，新开一个窗口，输入“rqt”指令，开启 RQT 工具。

6） 选择 Plugins→Logging→Console，打开日志显示。

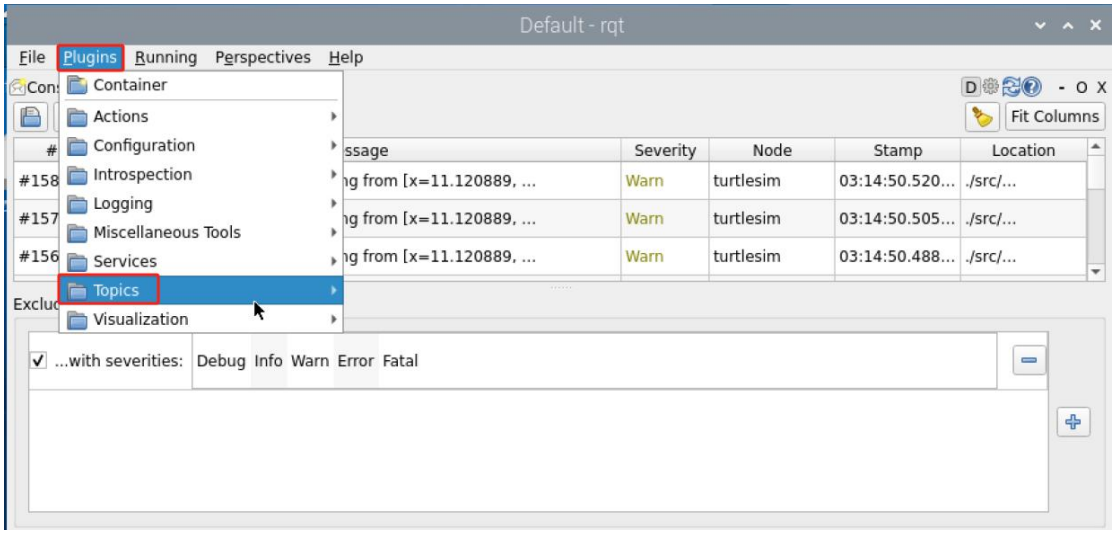


7） 使用“↑↓←→”键控制小海龟移动，如果小海龟触碰到墙壁，会显示日志信息。

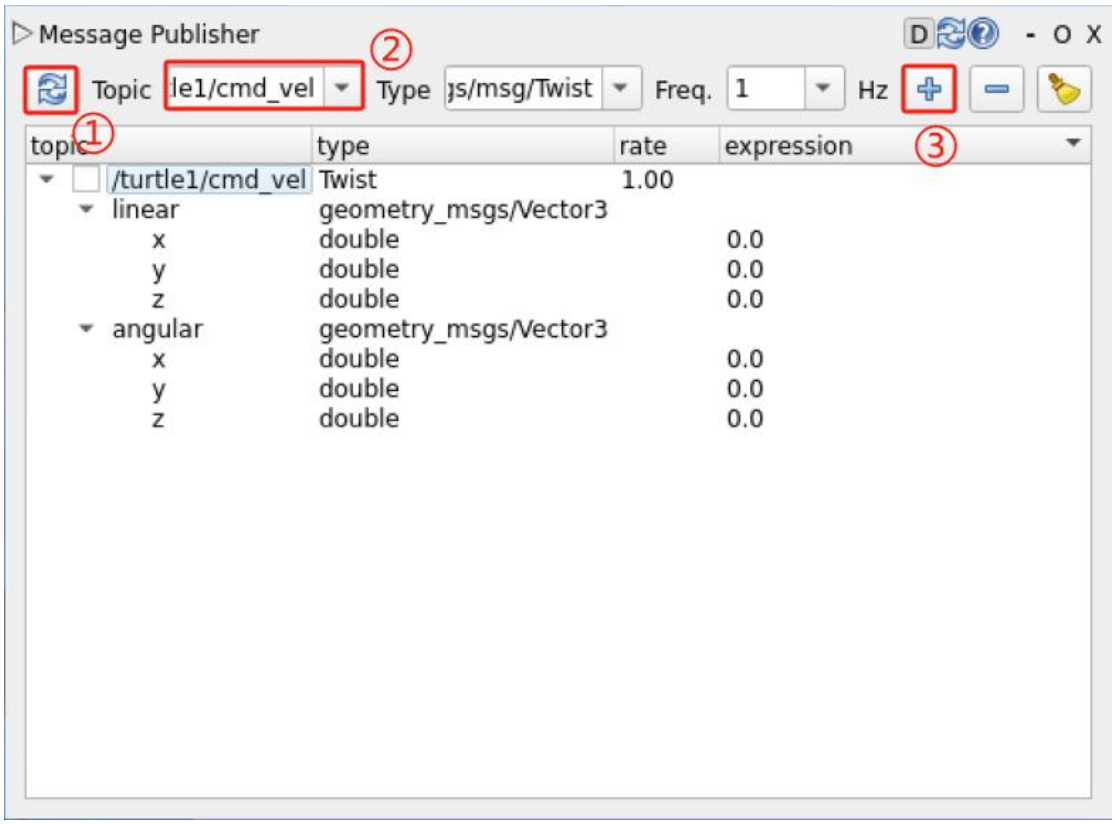


3.topic 插件

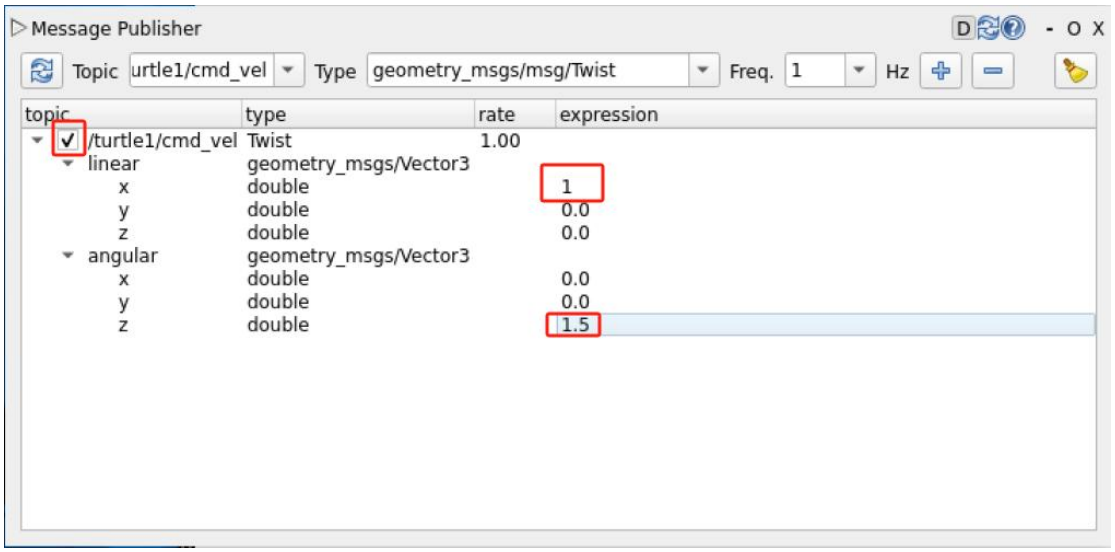
1） 选择 Plugins→Topic→Message Publisher，打开话题插件。



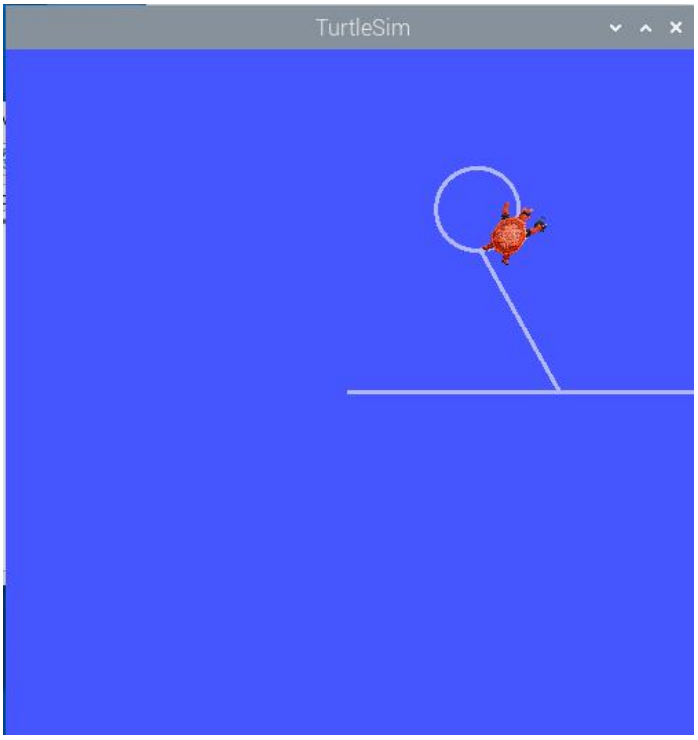
2） 点击刷新按钮，然后选择“/turtel/cmd_vel/”话题，然后点击添加按钮。



3） 修改线速度 X 的值和角速度 Z 的值，然后选中勾选。

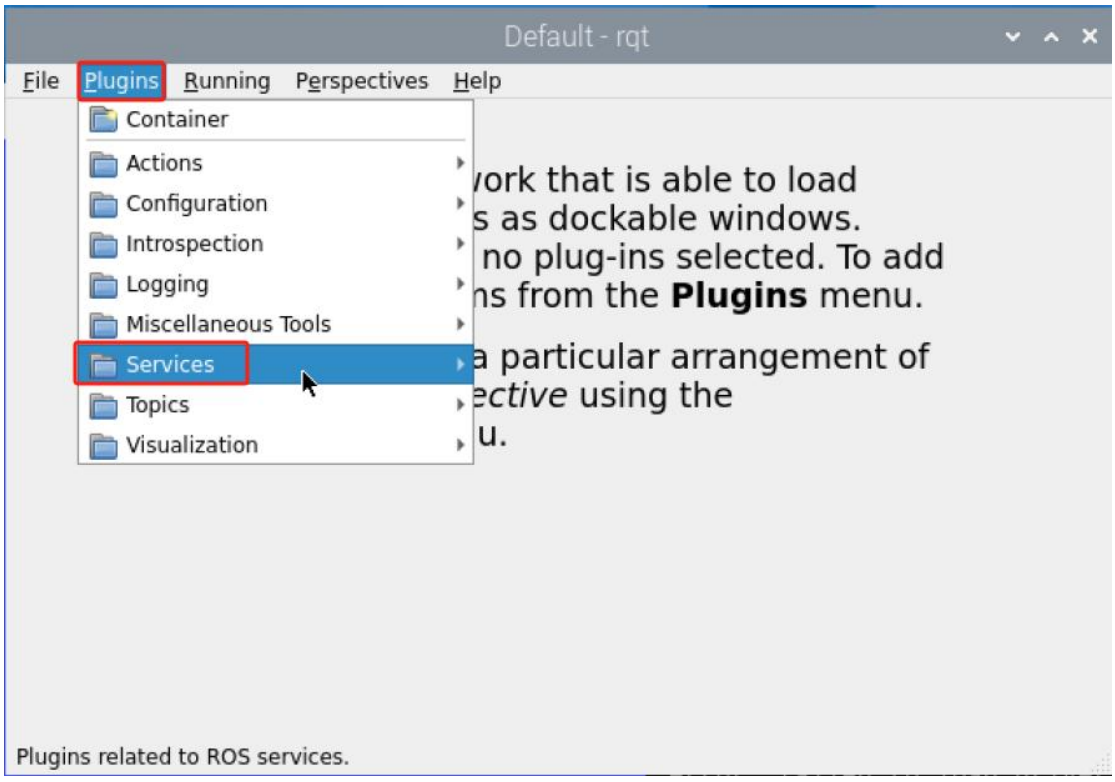


小海龟则会根据给定的线速度和角速度数据进行移动了。

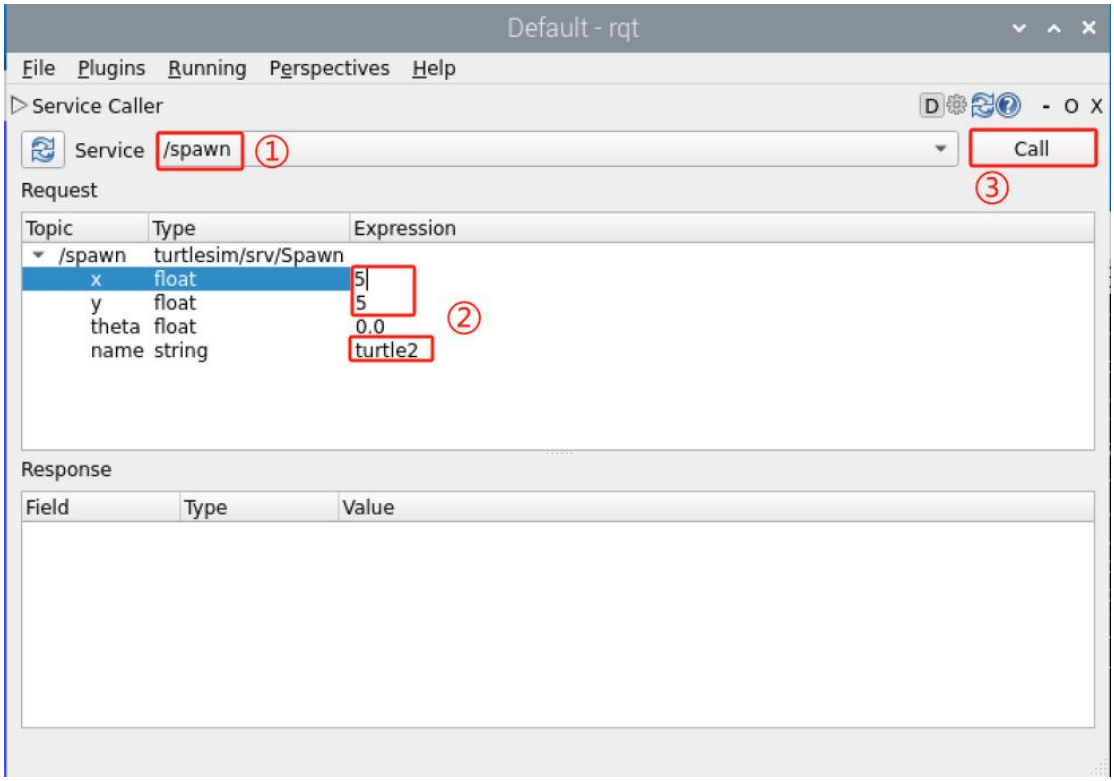


4.Service 插件

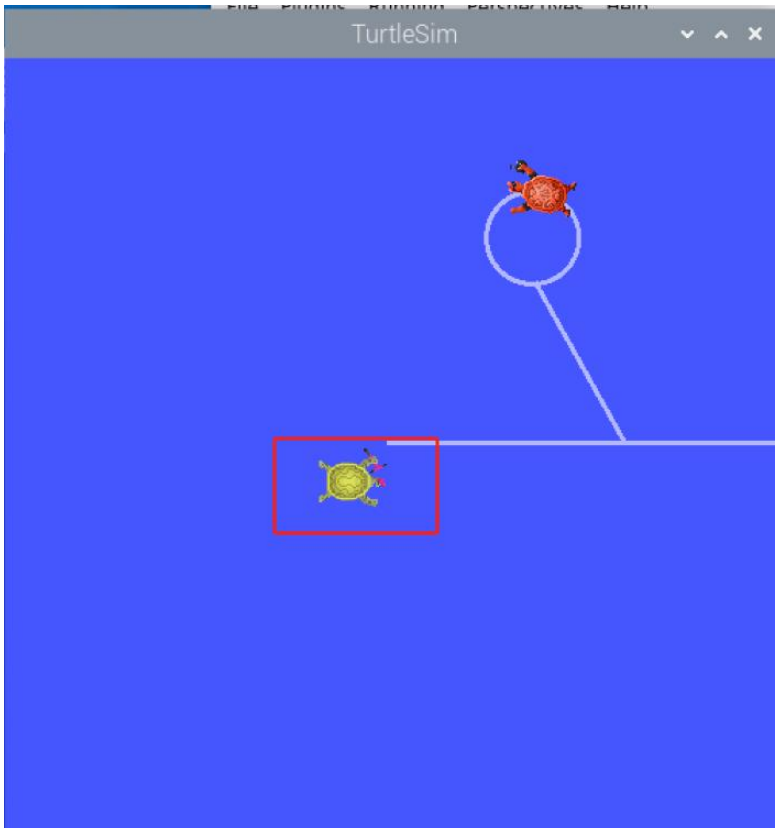
1） 选择 Plugins→Services→Services Caller，打开服务插件。



2） 选择“**/spawn**”服务，然后设置新生成小海龟的位置，名字为 turtle2，最后点击 Call。

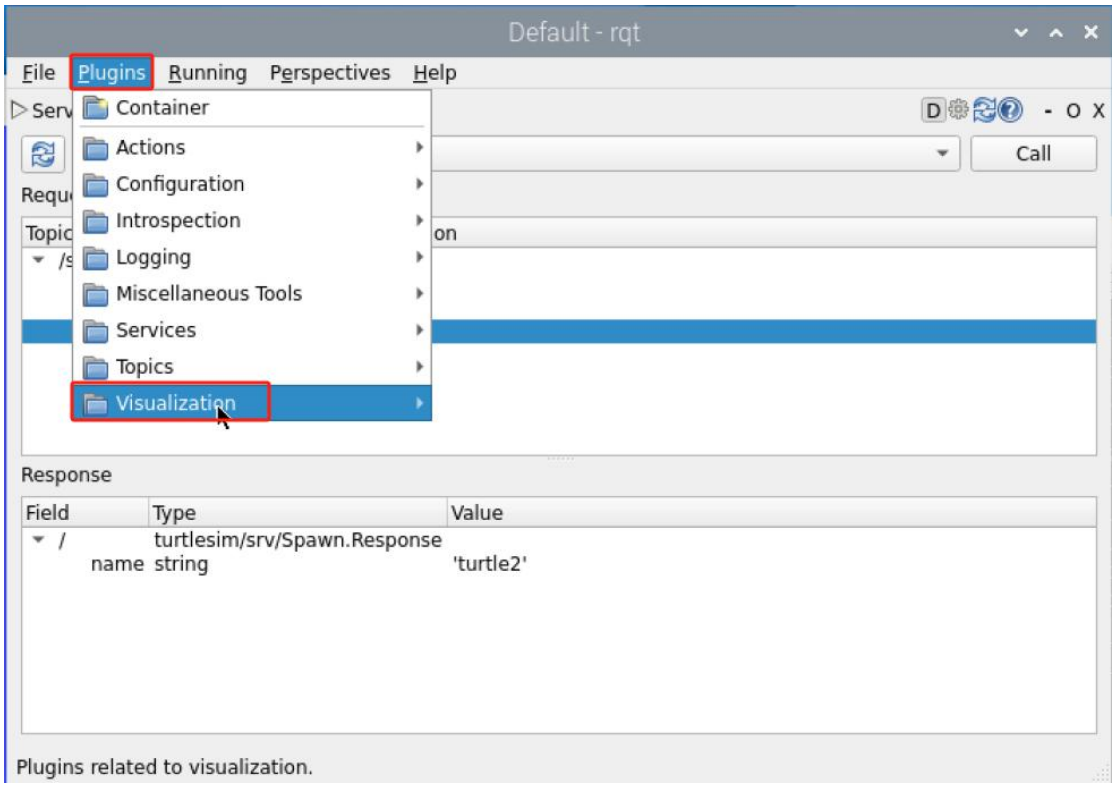


3） 这个时候会生成一只叫“turtle2”的小海龟。

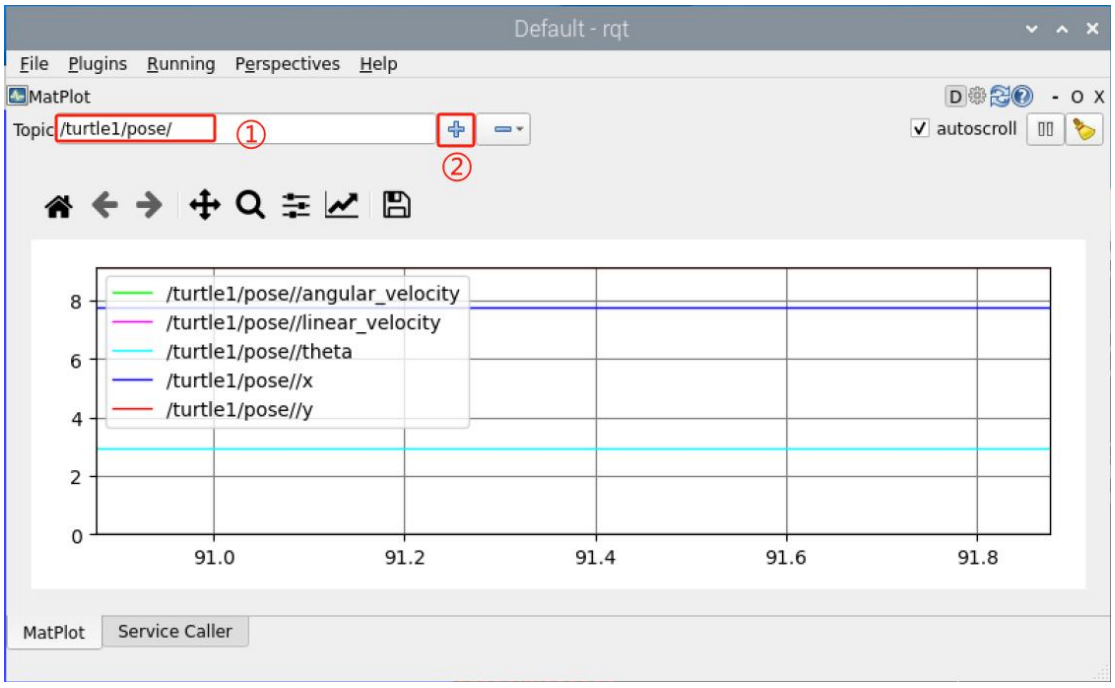


5.画图插件

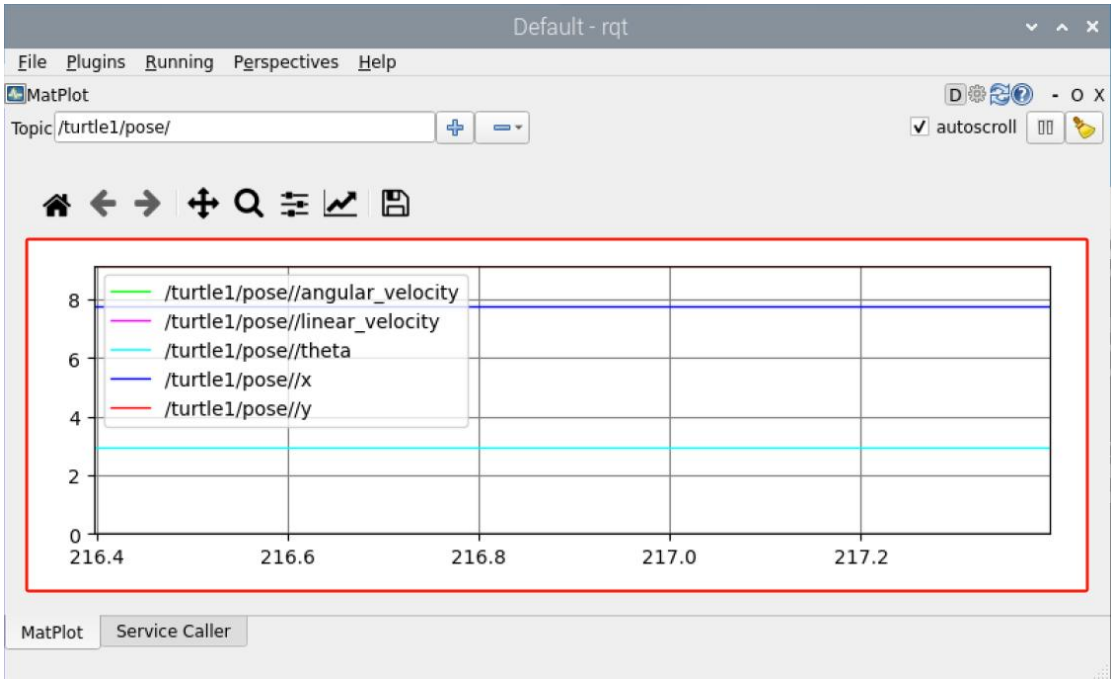
1） 选择 Plugins→Visualization→Plot，打开画图插件。



2） 输入“**/turtle/pose/**”，选择小海龟位置数据，然后点击添加。



3） 插件就会将小海龟的角速度、线速度、角度、X 坐标和 Y 坐标进行实时可视化。



想进一步学习 RQT 工具其他插件，请去官方（https://www.ros.org/）查看教程。