ROS基础 Vanguard视觉培训教程

ROS基础 Vanguard视觉培训教程

- 引言
- ROS2安装方法
  - 1.设置编码
  - 2.添加源
  - 3.安装ROS2
  - 4.设置环境变量
- ROS2开发环境配置
  - Git
  - VS Code
  - 工作空间
    - 创建工作空间：
    - 自动安装依赖：
    - 编译工作空间：
    - 设置环境变量：
- 功能包
  - 创建功能包
  - ros2命令中：
- 编译
- 节点
  - 节点代码实现（面向过程）
  - 代码运行
  - 节点代码实现（面向对象）
  - 总结创建节点流程
  - 节点命令行操作
- 话题
  - 发布/订阅模型
  - 多对多通信模型
  - 异步通信
  - 通信接口
  - 代码实现（发布者）
  - 代码实现（订阅者）
- 通信接口
  - 语言无关
  - 功能包的CMakeLists.txt中配置编译选项‘
  - 功能包的package.xml文件中需要添加代码生成的功能依赖：
- 服务
  - 客户端/服务器模型
  - 同步通信
  - 一对多模型
  - 通信接口
  - 代码实现（服务端）
  - 代码实现（客户端）
  - 服务命令行操作

动作
通信模型
客户端/服务器模型
一对多通信
同步通信
接口定义
通信模型
代码实现（服务器端）
代码实现（客户端）
动作命令行操作

引言

机器人的发展横跨七八十年，经历了三个重要时期。

2000年前，机器人主要应用于工业生产，俗称工业机器人，由示教器操控，帮助工厂释放劳动力，此时的机器人并没有太多智能而言，完全按照人类的命令执行动作，更加关注电气层面的驱动器、伺服电机、减速机、控制器等设备，这是机器人的电气时代。

2000年后，计算机和视觉技术逐渐应用，机器人的类型不断丰富，出现了AGV、视觉检测等应用，此时的机器人传感器更加丰富，但是依然缺少自主思考的过程，智能化有限，只能感知局部环境，这是机器人的数字时代，不过这也是机器人大时代的前夜。

2015年之后，随着人工智能技术的快速发展，机器人成为了AI技术的最佳载体，家庭服务机器人、送餐机器人、四足仿生机器狗、自动驾驶汽车等应用呈井喷状爆发，智能机器人时代正式拉开序幕。

智能机器人的快速发展，必将对机器人开发提出更高的要求，软件层面最为热点的技术之一就是机器人操作系统，这也是我们课程的主角——Robot Operating System（即ROS）。

ROS2安装方法

目前来讲，Linux发展迅猛，已经成为了性能稳定的多用户操作系统，也是ROS2依赖的重要底层系统。虽然ROS2目前也支持Windows、MacOS，但对Linux系统的支持最好，在本教程中，我们主要讲解Linux之上的ROS2使用方法，其他系统原理也基本相同。

本教程将使用Ubuntu22.04LTS版本来作为我们的操作系统，安装方法很多，如果你之前已经熟悉Linux，建议在电脑上硬盘安装Ubuntu，这样可以发挥出硬件最大的性能，如果你是第一次接触Linux，建议在已有的windows上通过虚拟机安装，未来熟悉之后再考虑硬盘安装。（个人建议还是利用虚拟机进行调试，双系统崩溃之后可能会影响到主系统，同时出现问题重新安装比较麻烦）

系统镜像：https://ubuntu.com/download/desktop

VMware16Pro： https://www.aliyundrive.com/s/SaE6abZYjAP

16pro密钥：https://www.haozhuangji.com/xtjc/145413111.html

VMware17：https://www.aliyundrive.com/s/quVKXGzbBzJ

具体安装可自行到CSDN查看
推荐安装配置：硬盘30G，内存4G，处理器数量2， 每个处理器的核心数4， 请视个人电脑情况安装，后续可以调整

接下来，我们就可以把ROS2安装到Ubuntu系统中了。安装步骤如下：

1.设置编码

```
sudo apt update && sudo apt install locales

sudo locale-gen en_US en_US.UTF-8

sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8

export LANG=en_US.UTF-8
```

2.添加源

```
sudo apt update && sudo apt install curl gnupg lsb-release

sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -o /usr/share/keyrings/ros-archive-keyring.gpg

echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/ros-archive-keyring.gpg] http://packages.ros.org/ros2/ubuntu $(source /etc/os-release && echo $UBUNTU_CODENAME) main" | sudo tee /etc/apt/sources.list.d/ros2.list > /dev/null
```

如遇报错“Failed to connect to raw.githubusercontent.com”，可参https://www.guyuehome.com/37844

3.安装ROS2

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

4.设置环境变量

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

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

ROS2开发环境配置

Git

git是一个版本管理软件，也是因Linux而生。

```
sudo apt install git
```

VS Code

下载链接：https://code.visualstudio.com/Download

插件配置：中文语言包、python插件、c++插件、CMake、vscode-icons、ROS、Msg Language Support、 Visual Studio IntelliCode、URDF、Markdown All in One

---

工作空间

创建工作空间：

```
mkdir -p ~/dev_ws/src
cd ~/dev_ws/src
```

自动安装依赖：

```
sudo apt install -y python3-pip
sudo pip3 install rosdepc
sudo rosdepc init
rosdepc update
cd ..
rosdep install -i --from-path src --rosdistro humble -y
```

编译工作空间：

```
sudo apt install python3-colcon-ros
cd ~/dev_ws/
colcon build
```

设置环境变量：

```
source install/local_setup.sh # 仅在当前终端生效
echo " source ~/dev_ws/install/local_setup.sh" >> ~/.bashrc # 所有终端均生效
```

功能包

举个例子，我们手上有很多红豆、绿豆、黄豆，假设都放在一个袋子里，如果只想把黄豆都拿出来，是不是得在五颜六色的豆子里一颗一颗都找出来，数量越多，你就越头疼；如果我们把不同颜色的豆子放在不同的三个袋子里，需要拿出某种豆子的时候，不就立刻可以找出来了么。

功能包就是这个原理，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。

创建功能包

```
ros2 pkg create --build-type <build-type> <package_name>
```

ros2命令中：

pkg：表示功能包相关的功能；

create：表示创建功能包；

build-type：表示新创建的功能包是C++还是Python的，如果使用C++或者C，那这里就跟ament*cmake，如果使用Python，就跟ament*python；

package_name：新建功能包的名字。

编译

在创建好的功能包中，我们可以继续完成代码的编写，之后需要编译和配置环境变量，才能正常运行：

```
cd ~/dev_ws
colcon build   # 编译工作空间所有功能包，在后面加后缀可指定编译某个功能包
source install/local_setup.bash
```

节点

在机器人身体里搭载了一台计算机A，它可以通过机器人的眼睛——摄像头，获取外界环境的信息，也可以控制机器人的腿——轮子，让机器人移动到想要去的地方。除此之外，可能还会有另外一台计算机B，放在你的桌子上，它可以远程监控机器人看到的信息，也可以远程配置机器人的速度和某些参数，还可以连接一个摇杆，人为控制机器人前后左右运动。

这些功能虽然位于不同的计算机中，但都是这款机器人的工作细胞，也就是节点，他们共同组成了一个完整的机器人系统。

节点在机器人系统中的职责就是执行某些具体的任务，从计算机操作系统的角度来看，也叫做进程；

每个节点都是一个可以独立运行的可执行文件，比如执行某一个python程序，或者执行C++编译生成的结果，都算是运行了一个节点；

既然每个节点都是独立的执行文件，那自然就可以想到，得到这个执行文件的编程语言可以是不同的，比如C++、Python，乃至Java、Ruby等更多语言。

这些节点是功能各不相同的细胞，根据系统设计的不同，可能位于计算机A，也可能位于计算机B，还有可能运行在云端，这叫做分布式，也就是可以分布在不同的硬件载体上；

每一个节点都需要有唯一的命名，当我们想要去找到某一个节点的时候，或者想要查询某一个节点的状态时，可以通过节点的名称来做查询。

节点也可以比喻是一个一个的工人，分别完成不同的任务，他们有的在一线厂房工作，有的在后勤部门提供保障，他们互相可能并不认识，但却一起推动机器人这座“工厂”，完成更为复杂的任务。

节点代码实现（面向过程）

```python
import rclpy                                    # ROS2 Python接口库
from rclpy.node import Node                     # 引入ROS2 节点类
import time                                     # time库进行定时

def main(args=None):                            # ROS2节点主入口main函数
    rclpy.init(args=args)                       # ROS2 Python接口初始化
    node = Node("node_helloworld")              # 创建ROS2节点对象并进行初始化

    while rclpy.ok():                           # ROS2系统是否正常运行
        node.get_logger().info("Hello World")   # ROS2日志输出
        time.sleep(0.5)                         # 休眠控制循环时间

    node.destroy_node()                         # 销毁节点对象
    rclpy.shutdown()                            # 关闭ROS2 Python接口
```

之后在setup.py文件配置程序入口

```
entry_points={
        'console_scripts': [
         '<入口名称，在终端处所输入的名称>       = <功能包名称（一般来讲会将代码文件放在功能包下同名文件夹）>.<代码文件名称>:main',
        ],
```

示例：

```
entry_points={
    'console_scripts': [
     'node_helloworld       = learning_node.node_helloworld:main',
    ],
```

代码运行

```
ros2 run <功能包名称> <程序入口>
```

示例：

```
ros2 run learning_node node_helloworld
```

节点代码实现（面向对象）

```python
import rclpy                                    # ROS2 Python接口库
```

```python
from rclpy.node import Node                                  # ROS2 节点类
import time                                                  # time库进行定时

"""
创建一个HelloWorld节点，初始化时输出“hello world”日志
"""
class HelloWorldNode(Node):
    def __init__(self, name):                                # 类初始化
        super().__init__(name)                               # ROS2节点父类初始化
        while rclpy.ok():                                    # ROS2系统是否正常运行
            self.get_logger().info("Hello World")            # ROS2日志输出
            time.sleep(0.5)                                  # 休眠控制循环时间

def main(args=None):                                         # ROS2节点主入口main函数
    rclpy.init(args=args)                                    # ROS2 Python接口初始化
    node = HelloWorldNode("node_helloworld_class")           # 创建ROS2节点对象并进行初始化
    rclpy.spin(node)                                         # 循环等待ROS2退出，和rclpy.ok()效果
一致，只有crtl+c在终端停止运行才会退出
    node.destroy_node()                                      # 销毁节点对象
    rclpy.shutdown()                                         # 关闭ROS2 Python接口
```

总结创建节点流程

总结一下，想要实现一个节点，代码的实现流程是这样做：

1.编程接口初始化

2.创建节点并初始化

3.实现节点功能

4.销毁节点并关闭接口

节点命令行操作

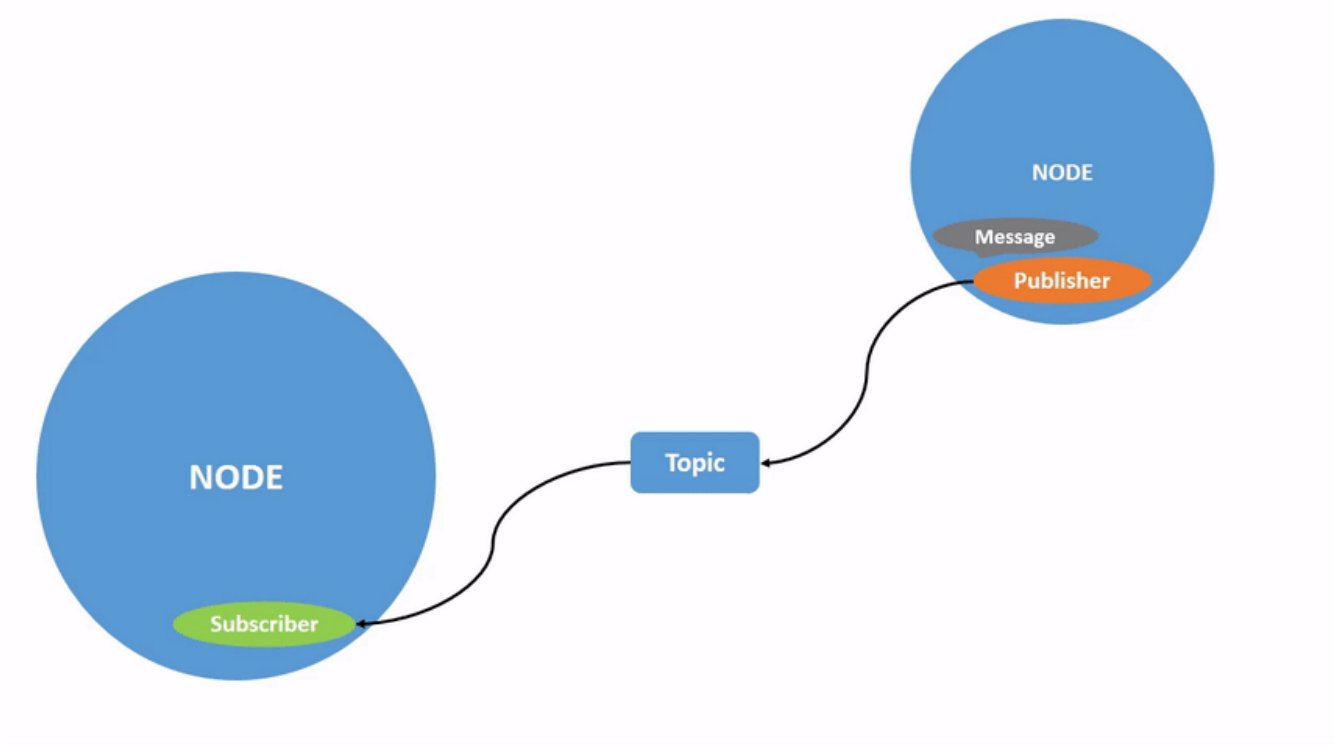
节点常用命令：

```
ros2 node list                # 查看节点列表
ros2 node info <node_name>    # 查看节点信息
```

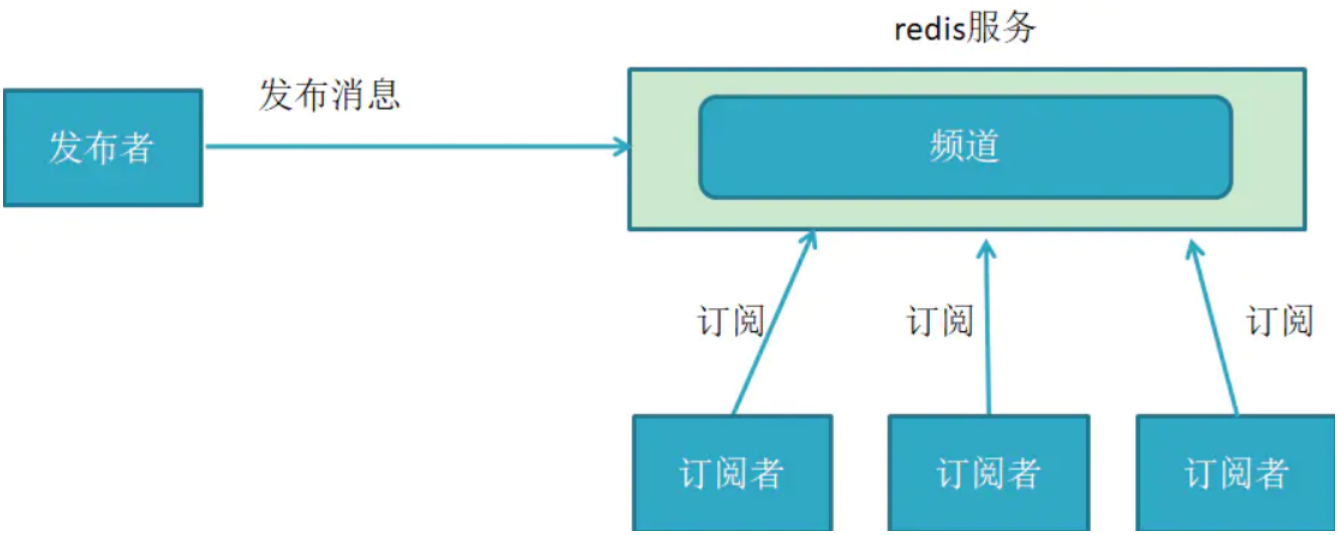
话题

话题是节点间传递数据的桥梁

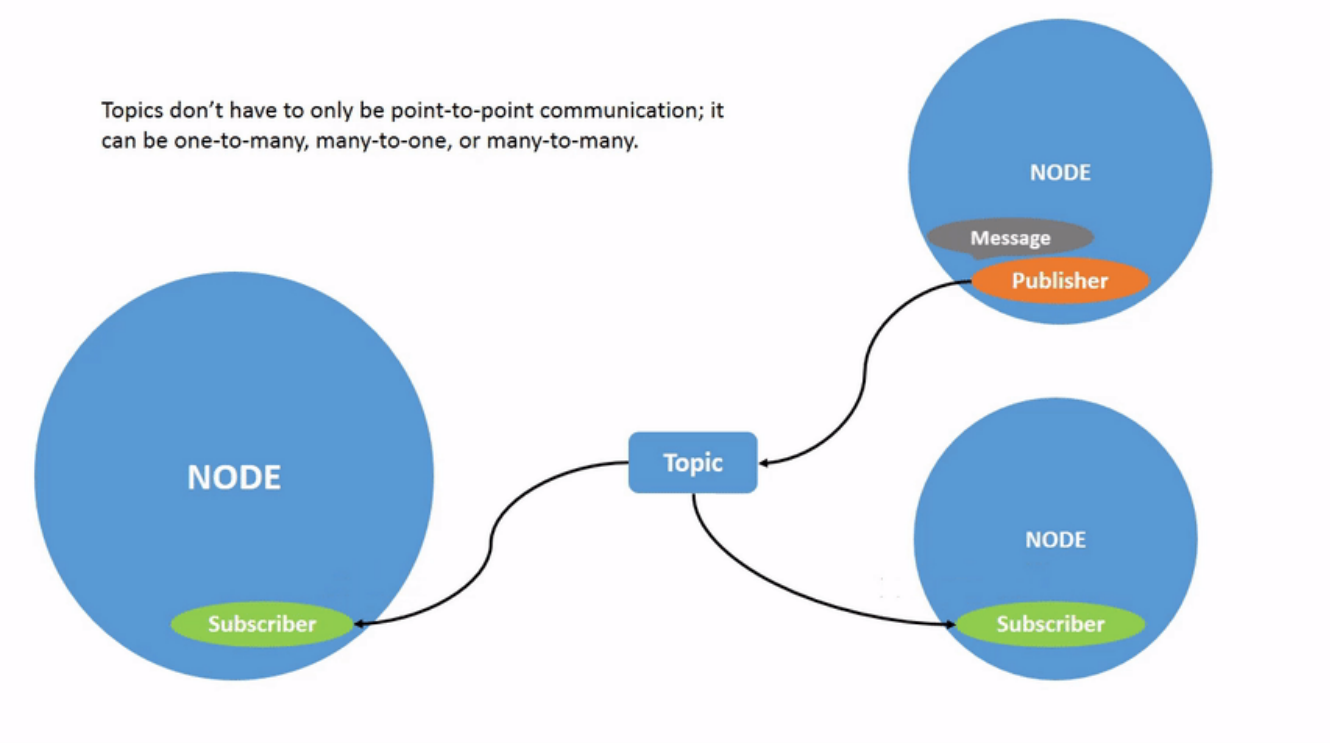
发布/订阅模型



从话题本身的实现角度来看，使用了基于DDS的发布/订阅模型，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，每一个话题都需要有一个名字，传输的数据也需要有固定的数据类型。



多对多通信模型



大家再仔细想下这些可以订阅的东西，是不是并不是唯一的，我们每个人可以订阅很多公众号、报纸、杂志，这些公众号、报纸、杂志也可以被很多人订阅，没错，ROS里的话题也是一样，发布者和订阅者的数量并不是唯一的，可以称之为是多对多的通信模型。

异步通信

话题通信还有一个特性，那就是异步，这个词可能有同学是第一次听说？所谓异步，只要是指发布者发出数据后，并不知道订阅者什么时候可以收到，类似公众号发布一篇文章，你什么时候阅读的，管理者根本不知道，报社发出一份报纸，你什么时候收到，报社也是不知道的。这就叫做异步。

异步的特性也让话题更适合用于一些周期发布的数据，比如传感器的数据，运动控制的指令等等，如果某些逻辑性较强的指令，比如修改某一个参数，用话题传输就不太合适了。

通信接口

既然是数据传输，发布者和订阅者就得统一数据的描述格式，不能一个说英文，一个理解成了中文。在ROS中，话题通信数据的描述格式称之为消息，对应编程语言中数据结构的概念。比如这里的一个图像数据，就会包含图像的长宽像素值、每个像素的RGB等等，在ROS中都有标准定义。

消息是ROS中的一种接口定义方式，与编程语言无关，我们也可以通过.msg后缀的文件自行定义，有了这样的接口，各种节点就像积木块一样，通过各种各样的接口进行拼接，组成复杂的机器人系统。

代码实现（发布者）

```
import rclpy                                    # ROS2 Python接口库
from rclpy.node import Node                     # ROS2 节点类
from std_msgs.msg import String                 # 字符串消息类型

```

```python
class PublisherNode(Node):

    def __init__(self, name):
        super().__init__(name)                                  # ROS2节点父类初始化
        self.pub = self.create_publisher(String, "chatter", 10)   # 创建发布者对象（消息类型、话题名、队列长度）
        self.timer = self.create_timer(0.5, self.timer_callback)  # 创建一个定时器（单位为秒的周期，定时执行的回调函数）

    def timer_callback(self):                                   # 创建定时器周期执行的回调函数
        msg = String()                                          # 创建一个String类型的消息对象
        msg.data = 'Hello World'                                # 填充消息对象中的消息数据
        self.pub.publish(msg)                                   # 发布话题消息
        self.get_logger().info('Publishing: "%s"' % msg.data)     # 输出日志信息，提示已经完成话题发布

def main(args=None):                                # ROS2节点主入口main函数
    rclpy.init(args=args)                           # ROS2 Python接口初始化
    node = PublisherNode("topic_helloworld_pub")    # 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
```

流程如下：

- 编程接口初始化
- 创建节点并初始化
- 创建发布者对象
- 创建并填充话题消息
- 发布话题消息
- 销毁节点并关闭接口

代码实现（订阅者）

```python
import rclpy                    # ROS2 Python接口库
from rclpy.node   import Node     # ROS2 节点类
from std_msgs.msg import String   # ROS2标准定义的String消息

class SubscriberNode(Node):

    def __init__(self, name):
        super().__init__(name)                              # ROS2节点父类初始化
        self.sub = self.create_subscription(\
            String, "chatter", self.listener_callback, 10) # 创建订阅者对象（消息类型、话题名、订阅者回调函数、队列长度）

```

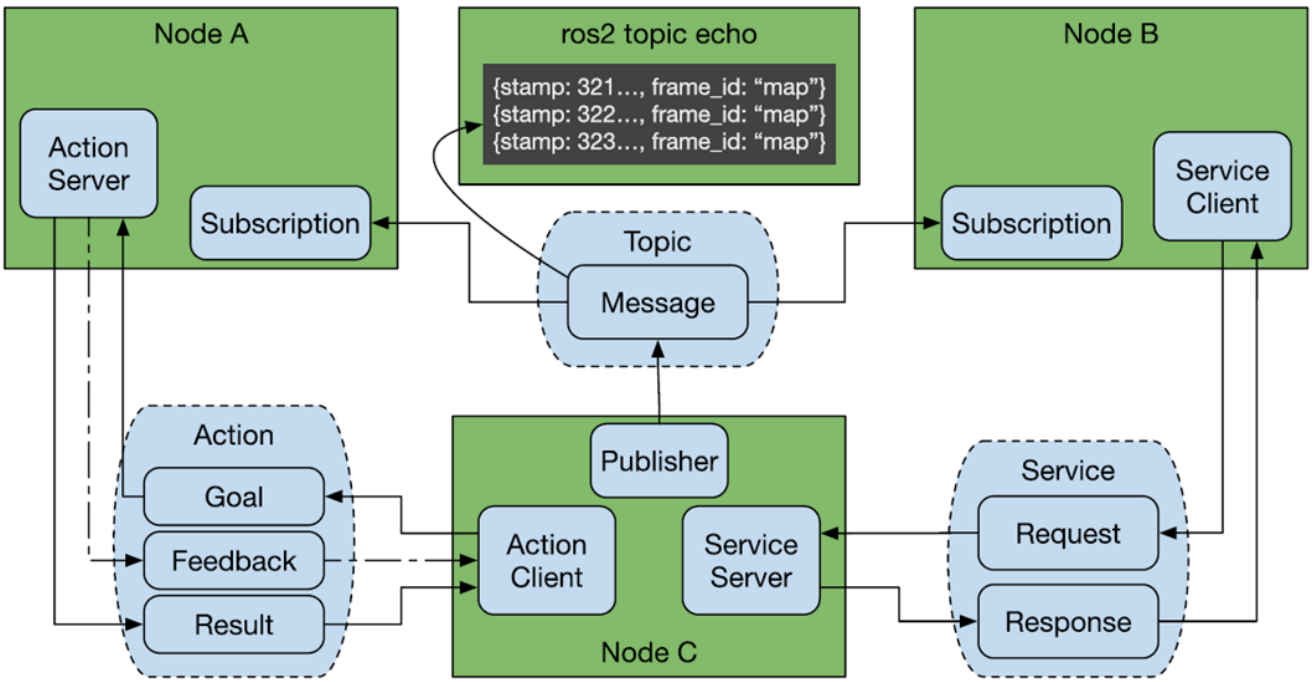
```
    def listener_callback(self, msg):                    # 创建回调函数，执行收到话题消息后对数据的处理
        self.get_logger().info('I heard: "%s"' % msg.data) # 输出日志信息，提示订阅收到的话题消息

def main(args=None):                                     # ROS2节点主入口main函数
    rclpy.init(args=args)                                # ROS2 Python接口初始化
    node = SubscriberNode("topic_helloworld_sub")        # 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
```

流程如下：

- 编程接口初始化
- 创建节点并初始化
- 创建订阅者对象
- 回调函数处理话题数据
- 销毁节点并关闭接口

通信接口

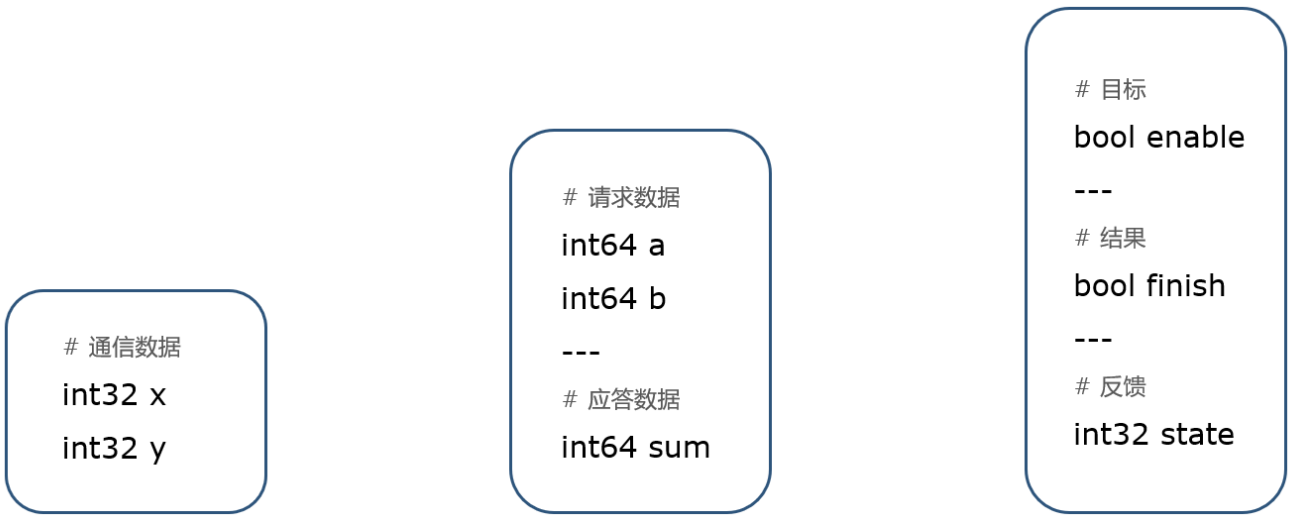
接口可以让程序之间的依赖降低，便于我们使用别人的代码，也方便别人使用我们的代码，这就是ROS的核心目标，减少重复造轮子。



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

语言无关

为了保证每一个节点可以使用不同语言编程，ROS将这些接口的设计做成了和语言无关的，比如这里看到的int32表示32位的整型数，int64表示64位的整型数，bool表示布尔值，还可以定义数组、结构体，这些定义在编译过程中，会自动生成对应到C++、Python等语言里的数据结构。



- 话题通信接口的定义使用的是.msg文件，由于是单向传输，只需要描述传输的每一帧数据是什么就行，比如在这个定义里，会传输两个32位的整型数，x、y，我们可以用来传输二维坐标的数值。
- 服务通信接口的定义使用的是.srv文件，包含请求和应答两部分定义，通过中间的“---”区分，比如之前我们学习的加法求和功能，请求数据是两个64位整型数a和b，应答是求和的结果sum。
- 动作是另外一种通信机制，用来描述机器人的一个运动过程，使用.action文件定义，比如我们让小海龟转90度，一边转一边周期反馈当前的状态，此时接口的定义分成了三个部分，分别是动作的目标，比如是开始运动，运动的结果，最终旋转的90度是否完成，还有一个周期反馈，比如每隔1s反馈一下当前转到第10度、20度还是30度了，让我们知道运动的进度。

功能包的CMakeLists.txt中配置编译选项‘

```
...

find_package(rosidl_default_generators REQUIRED)

rosidl_generate_interfaces(${PROJECT_NAME}
  "srv/GetObjectPosition.srv"
)

...
```

功能包的package.xml文件中需要添加代码生成的功能依赖：

```
...

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

...
```

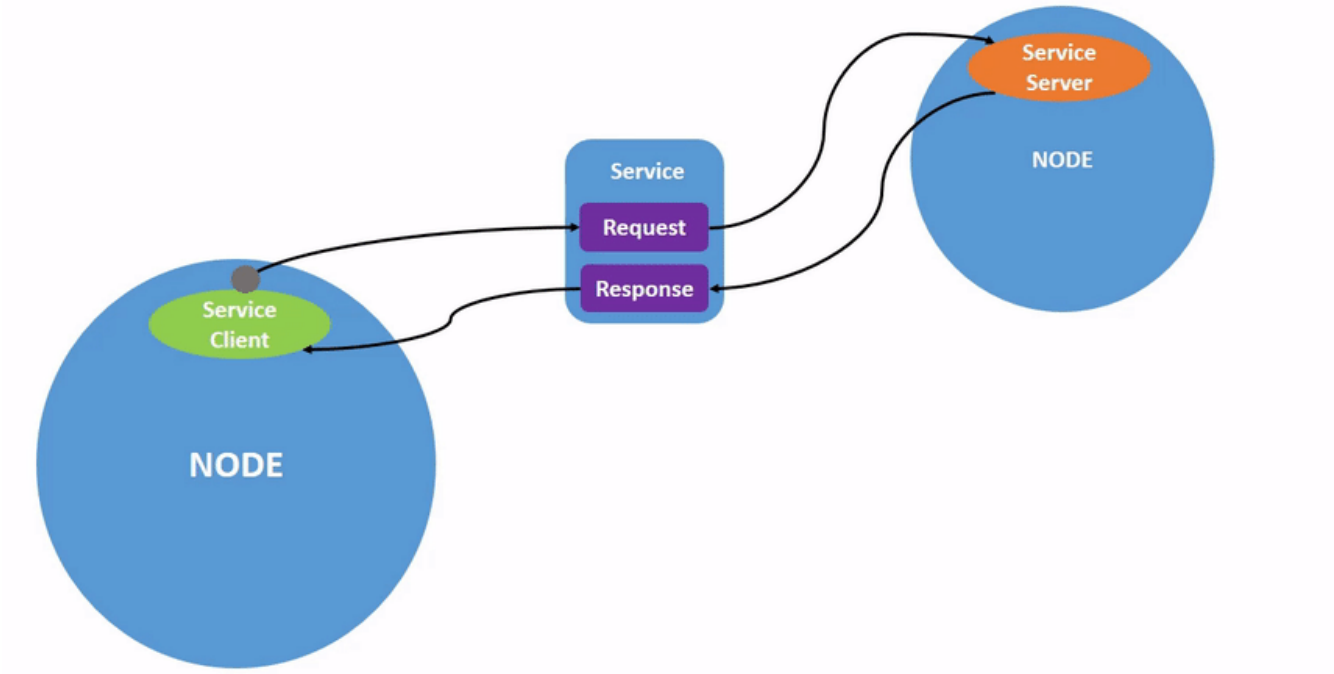
服务

话题通信可以实现多个ROS节点之间数据的单向传输，使用这种异步通信机制，发布者无法准确知道订阅者是否收到消息，这一部分我们将一起学习ROS另外一种常用的通信方法——服务，可以实现类似你问我答的同步通信效果。

客户端/服务器模型

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。

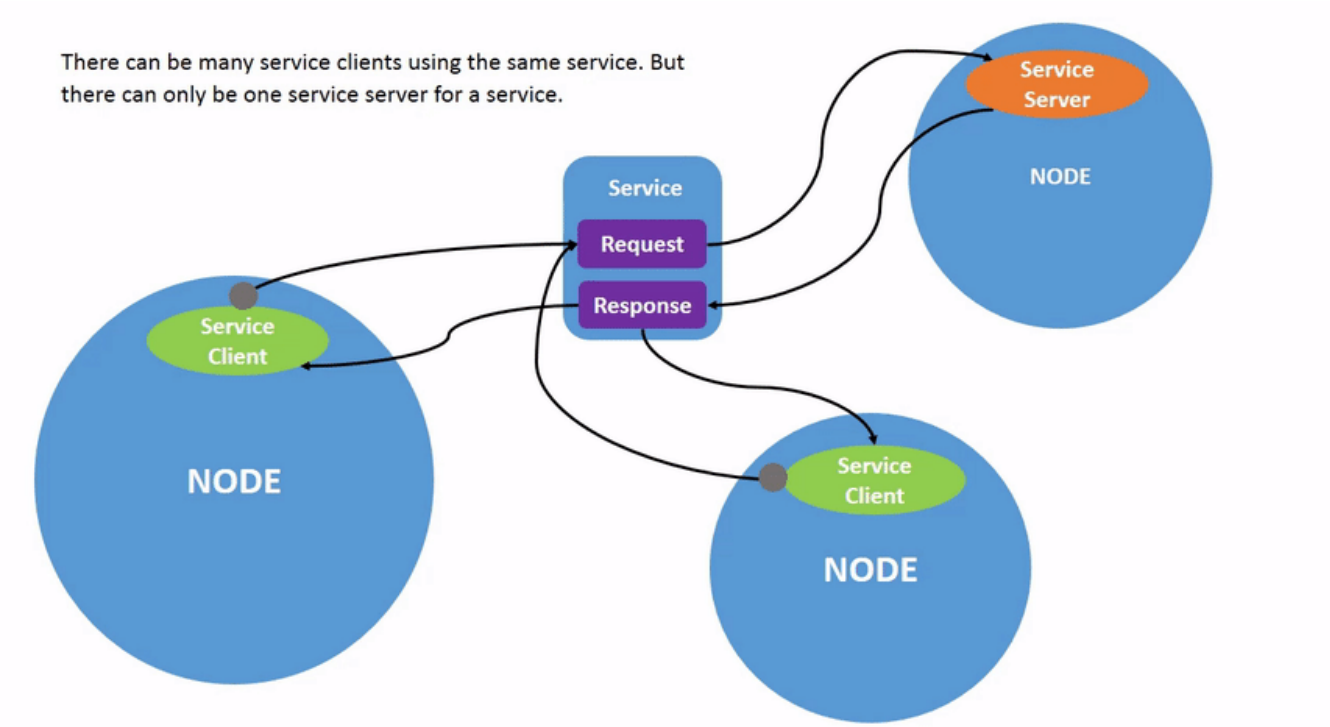


同步通信

这个过程一般要求越快越好，假设服务器半天没有反应，你的浏览器一直转圈圈，那有可能是服务器宕机了，或者是网络不好，所以相比话题通信，在服务通信中，客户端可以通过接收到的应答信息，判断服务器端的状态，我们也称之为同步通信。

一对多模型

比如www.baidu.com这个网站， 服务器是唯一存在的，并不存在多个完全一样的网站， 但是可以同时访问这个网站的客户端不是唯一的，每个人看到的都是相同的界面



通信接口

和话题通信类似，服务通信的核心还是要传递数据，数据变成了两个部分，一个请求的数据，比如请求苹果位置的命令，还有一个反馈的数据，比如反馈苹果坐标位置的数据，这些数据和话题消息一样，在ROS中也是要标准定义的，话题使用.msg文件定义，服务使用的是.srv文件定义

代码实现（服务端）

```
import rclpy                                    # ROS2 Python接口库
from rclpy.node   import Node                   # ROS2 节点类
from learning_interface.srv import AddTwoInts   # 自定义的服务接口

class adderServer(Node):
    def __init__(self, name):
        super().__init__(name)
    # ROS2节点父类初始化
        self.srv = self.create_service(AddTwoInts, 'add_two_ints',
self.adder_callback)  # 创建服务器对象（接口类型、服务名、服务器回调函数）
```

```
9
10      def adder_callback(self, request, response):    # 创建回调函数，执行收到请求后对数据的处理
11          response.sum = request.a + request.b        # 完成加法求和计算，将结果放到反馈的数据中
12          self.get_logger().info('Incoming request\na: %d b: %d' % (request.a, request.b))    # 输出日志信息，提示已经完成加法求和计算
13          return response                             # 反馈应答信息
14
15  def main(args=None):                                # ROS2节点主入口main函数
16      rclpy.init(args=args)                           # ROS2 Python接口初始化
17      node = adderServer("service_adder_server")      # 创建ROS2节点对象并进行初始化
18      rclpy.spin(node)                                # 循环等待ROS2退出
19      node.destroy_node()                             # 销毁节点对象
20      rclpy.shutdown()                                # 关闭ROS2 Python接口
```

流程如下：

- 编程接口初始化
- 创建节点并初始化
- 创建服务器对象
- 回调函数处理话题数据
- 向客户端反馈应答结果
- 销毁节点并关闭接口

代码实现（客户端）

```
1   import sys                                          # 用于获取数据输入
2
3   import rclpy                                        # ROS2 Python接口库
4   from rclpy.node   import Node                       # ROS2 节点类
5   from learning_interface.srv import AddTwoInts # 自定义的服务接口
6
7   class adderClient(Node):
8       def __init__(self, name):
9           super().__init__(name)                                          # ROS2节点父类初始化
10          self.client = self.create_client(AddTwoInts, 'add_two_ints') # 创建服务客户端对象（服务接口类型，服务名）
11          while not self.client.wait_for_service(timeout_sec=1.0):      # 循环等待服务器端成功启动
12              self.get_logger().info('service not available, waiting again...')
13          self.request = AddTwoInts.Request()                             # 创建服务请求的数据对象
14
15      def send_request(self):                                             # 创建一个发送服务请求的函数
16          self.request.a = int(sys.argv[1])
17          self.request.b = int(sys.argv[2])
```

```
        self.future = self.client.call_async(self.request)          # 异步方式发送服务请求

def main(args=None):
    rclpy.init(args=args)                                # ROS2 Python接口初始化
    node = adderClient("service_adder_client")           # 创建ROS2节点对象并进行初始化
    node.send_request()                                  # 发送服务请求

    while rclpy.ok():                                    # ROS2系统正常运行
        rclpy.spin_once(node)                            # 循环执行一次节点

        if node.future.done():                           # 数据是否处理完成
            try:
                response = node.future.result()  # 接收服务器端的反馈数据
            except Exception as e:
                node.get_logger().info(
                    'Service call failed %r' % (e,))
            else:
                node.get_logger().info(          # 将收到的反馈信息打印输出
                    'Result of add_two_ints: for %d + %d = %d' %
                    (node.request.a, node.request.b, response.sum))
            break

    node.destroy_node()                                  # 销毁节点对象
    rclpy.shutdown()                                     # 关闭ROS2 Python接口
```

流程如下：

- 编程接口初始化
- 创建节点并初始化
- 创建客户端对象
- 创建并发送请求数据
- 等待服务器端应答数据
- 销毁节点并关闭接口

服务命令行操作

常用服务命令如下：

```
ros2 service list                 # 查看服务列表
ros2 service type <service_name>    # 查看服务数据类型
ros2 service call <service_name> <service_type> <service_data>    # 发送服务请求
```

动作

通信模型

举个例子，比如我们想让机器人转个圈，这肯定不是一下就可以完成的，机器人得一点一点旋转，直到360度才能结束，假设机器人并不在我们眼前，发出指令后，我们根本不知道机器人到底有没有开始转圈，转到哪里了？

现在我们需要的是一个反馈，比如每隔1s，告诉我们当前转到多少度了，10度、20度、30度，一段时间之后，到了360度，再发送一个信息，表示动作执行完成。

这样一个需要执行一段时间的行为，使用动作的通信机制就更为合适，就像装了一个进度条，我们可以随时把控进度，如果运动过程当中，我们还可以随时发送一个取消运动的命令。

客户端/服务器模型

动作和服务类似，使用的也是客户端和服务器模型，客户端发送动作的目标，想让机器人干什么，服务器端执行动作过程， 控制机器人达到运动的目标，同时周期反馈动作执行过程中的状态。

客户端发送一个运动的目标，想让机器人动起来，服务器端收到之后，就开始控制机器人运动，一边运动，一边反馈当前的状态，如果是一个导航动作，这个反馈可能是当前所处的坐标，如果是机械臂抓取，这个反馈可能又是机械臂的实时姿态。当运动执行结束后，服务器再反馈一个动作结束的信息。整个通信过程就此结束。

大家再仔细看下上边的动图，是不是还会发现一个隐藏的秘密。

动作的三个通信模块，竟然有两个是服务，一个是话题，当客户端发送运动目标时，使用的是服务的请求调用，服务器端也会反馈一个应带，表示收到命令。动作的反馈过程，其实就是一个话题的周期发布，服务器端是发布者，客户端是订阅者。

没错，动作是一种应用层的通信机制，其底层就是基于话题和服务来实现的。

一对多通信

和服务一样，动作通信中的客户端可以有多个，大家都可以发送运动命令，但是服务器端只能有一个，毕竟只有一个机器人，先执行完成一个动作，才能执行下一个动作。

同步通信

既然有反馈，那动作也是一种同步通信机制，之前我们也介绍过，动作过程中的数据通信接口，使用.action文件进行定义。

接口定义

包含三个部分：

- 第一块是动作的目标
- 第二块是动作的执行结果
- 第三块是动作的周期反馈

通信模型

通信模型就是这样，客户端发送给一个动作目标，服务器控制机器人开始运动，并周期反馈，结束后反馈结束信息。

代码实现（服务器端）

```python
import time

import rclpy                                        # ROS2 Python接口库
from rclpy.node   import Node                       # ROS2 节点类
from rclpy.action import ActionServer               # ROS2 动作服务器类
from learning_interface.action import MoveCircle  # 自定义的圆周运动接口

class MoveCircleActionServer(Node):
    def __init__(self, name):
        super().__init__(name)                      # ROS2节点父类初始化
        self._action_server = ActionServer(      # 创建动作服务器（接口类型、动作名、回调函数）
            self,
            MoveCircle,
            'move_circle',
            self.execute_callback)

    def execute_callback(self, goal_handle):            # 执行收到动作目标之后的处理函数
        self.get_logger().info('Moving circle...')
        feedback_msg = MoveCircle.Feedback()            # 创建一个动作反馈信息的消息

        for i in range(0, 360, 30):                     # 从0到360度，执行圆周运动，并周期反馈信息
            feedback_msg.state = i                      # 创建反馈信息，表示当前执行到的角度
            self.get_logger().info('Publishing feedback: %d' % feedback_msg.state)
            goal_handle.publish_feedback(feedback_msg)  # 发布反馈信息
            time.sleep(0.5)

        goal_handle.succeed()                           # 动作执行成功
        result = MoveCircle.Result()                    # 创建结果消息
        result.finish = True
        return result                                   # 反馈最终动作执行的结果

def main(args=None):                                    # ROS2节点主入口main函数
    rclpy.init(args=args)                               # ROS2 Python接口初始化
    node = MoveCircleActionServer("action_move_server") # 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
```

代码实现（客户端）

```python
import rclpy                                        # ROS2 Python接口库
from rclpy.node   import Node                       # ROS2 节点类
from rclpy.action import ActionClient               # ROS2 动作客户端类

from learning_interface.action import MoveCircle  # 自定义的圆周运动接口

class MoveCircleActionClient(Node):
    def __init__(self, name):
        super().__init__(name)                      # ROS2节点父类初始化
        self._action_client = ActionClient(         # 创建动作客户端（接口类型、动作名）
            self, MoveCircle, 'move_circle')

    def send_goal(self, enable):                    # 创建一个发送动作目标的函数
        goal_msg = MoveCircle.Goal()                # 创建一个动作目标的消息
        goal_msg.enable = enable                    # 设置动作目标为使能，希望机器人开始运动

        self._action_client.wait_for_server()     # 等待动作的服务器端启动
        self._send_goal_future = self._action_client.send_goal_async(    # 异步方式发送动作的目标
            goal_msg,                                                    # 动作目标
            feedback_callback=self.feedback_callback)                    # 处理周期反馈消息的回调函数

        self._send_goal_future.add_done_callback(self.goal_response_callback) # 设置一个服务器收到目标之后反馈时的回调函数

    def goal_response_callback(self, future):           # 创建一个服务器收到目标之后反馈时的回调函数
        goal_handle = future.result()                   # 接收动作的结果
        if not goal_handle.accepted:                    # 如果动作被拒绝执行
            self.get_logger().info('Goal rejected :(')
            return

        self.get_logger().info('Goal accepted :)')                                # 动作被顺利执行

        self._get_result_future = goal_handle.get_result_async()                 # 异步获取动作最终执行的结果反馈
        self._get_result_future.add_done_callback(self.get_result_callback)    # 设置一个收到最终结果的回调函数

    def get_result_callback(self, future):                                        # 创建一个收到最终结果的回调函数
        result = future.result().result                                           # 读取动作执行的结果
        self.get_logger().info('Result: {%d}' % result.finish)                    # 日志输出执行结果
```

```
38
39      def feedback_callback(self, feedback_msg):                                  # 创建处理周期反馈消息的回调函数
40          feedback = feedback_msg.feedback                                        # 读取反馈的数据
41          self.get_logger().info('Received feedback: {%d}' % feedback.state)
42
43  def main(args=None):                                                            # ROS2节点主入口main函数
44      rclpy.init(args=args)                                                       # ROS2 Python接口初始化
45      node = MoveCircleActionClient("action_move_client")                         # 创建ROS2节点对象并进行初始化
46      node.send_goal(True)                                                        # 发送动作目标
47      rclpy.spin(node)                                                            # 循环等待ROS2退出
48      node.destroy_node()                                                         # 销毁节点对象
49      rclpy.shutdown()                                                            # 关闭ROS2 Python接口
```

动作命令行操作

动作命令常用操作：

```
1  ros2 action list                    # 查看服务列表
2  ros2 action info <action_name>      # 查看服务数据类型
3  ros2 action send_goal <action_name> <action_type> <action_data>    # 发送服务请求
```